# 前 言

PHP的编译与执行两项操作是PHP内核中的上层部分，依赖很多底层功能，如内存管理，字串、哈希表等，这些底层功能在独立篇章中介绍，本篇专注介绍PHP的编译过程和虚拟机执行。PHP底层源码使用C语言编写，熟悉C语言有助于阅读本篇，先了解一些底层功能相关知识也会对阅读本篇有帮助。

PHP的编译与执行两项操作关联非常紧密，所以放在一起介绍。编译操作比较复杂，执行更加复杂，涉及到的代码量很大。为了便于理解，尽量由浅入深进行介绍。

纵观PHP近十几年的发展历史，可以看到：PHP5以前的版本并不支持面向对象，PHP5开始支持类（class），PHP5.3开始支持命名空间（namespace）和闭包（closure），后续版又本不断推出新功能。所以PHP的功能可能从基础到高级分为几个级别：一、基础功能；二、面向过程相关功能；三、面向对象相关功能；四、其他特色功能。这些功能相对独立，耦合性不是很大，所以本篇以此为依据，划分为4个部分，从基础到高级依次进行介绍。

在PHP源码的zend\_ast.h文件中，定义了所有PHP语句在类型，共118个:

|  |
| --- |
| enum \_zend\_ast\_kind {  ZEND\_AST\_ZVAL = 1 << ZEND\_AST\_SPECIAL\_SHIFT,  ZEND\_AST\_CONSTANT,  ... 共118个  } |

其中除三个特殊语句类型ZEND\_AST\_ZVAL、ZEND\_AST\_CONSTANT、ZEND\_AST\_ZNODE外，基础功能语句类型有34个：

|  |  |  |
| --- | --- | --- |
| **结构** | **类型值** | **说明** |
| **列表元素语句** | ZEND\_AST\_ARRAY | 键值对，用于定义和遍历数组 |
| ZEND\_AST\_ENCAPS\_LIST | 引号或反引号中的变量列表 |
| ZEND\_AST\_STMT\_LIST | unset、global、static、echo语句 |
| ZEND\_AST\_CONST\_DECL | 用const 、declare语句声明的常量 |
| **有1个子节点的语句** | ZEND\_AST\_VAR | 访问变量 |
| ZEND\_AST\_CONST | 访问常量 |
| ZEND\_AST\_UNARY\_PLUS | 操作符 + |
| ZEND\_AST\_UNARY\_MINUS | 操作符 - |
| ZEND\_AST\_CAST | 强转类型语句(int),(float)等 |
| ZEND\_AST\_EMPTY | 调用empty函数 |
| ZEND\_AST\_ISSET | 调用isset函数 |
| ZEND\_AST\_SHELL\_EXEC | 反引号语句 |
| ZEND\_AST\_EXIT | exit 语句 |
| ZEND\_AST\_PRINT | 调用print函数 |
| ZEND\_AST\_INCLUDE\_OR\_EVAL | include/require/eval 语句 |
| ZEND\_AST\_UNARY\_OP | ！~ 两个操作符 |
| ZEND\_AST\_PRE\_INC | 前置自增运算 ++$var; |
| ZEND\_AST\_PRE\_DEC | 前置自减运算 --$var; |
| ZEND\_AST\_POST\_INC | 后置自增运算 $var++; |
| ZEND\_AST\_POST\_DEC | 后置自减运算 $var--; |
| ZEND\_AST\_UNSET | 调用unset 函数 |
| ZEND\_AST\_ECHO | echo 语句 |
| **有2个子节点的语句** | ZEND\_AST\_DIM | 访问多维数组 |
| ZEND\_AST\_ASSIGN | 赋值语句 |
| ZEND\_AST\_ASSIGN\_REF | 传址赋值 |
| ZEND\_AST\_ASSIGN\_OP | 二元操作赋值语句 |
| **ZEND\_AST\_BINARY\_OP** | 二元操作语句 |
| ZEND\_AST\_GREATER | 比较运算：大于> |
| ZEND\_AST\_GREATER\_EQUAL | 比较运算：大于等于>= |
| ZEND\_AST\_AND | 布尔型和逻辑型操作：AND |
| ZEND\_AST\_OR | 布尔型和逻辑型操作：OR |
| ZEND\_AST\_ARRAY\_ELEM | 数组里的一组键值对 |
| **有3个子节点的语句** | ZEND\_AST\_CONDITIONAL | 三元操作符 |
| ZEND\_AST\_CONST\_ELEM | 声名常量 |

面向过程语句类型有26个：

|  |  |  |
| --- | --- | --- |
| **结构** | **类型值** | **说明** |
| **声明型语句** | ZEND\_AST\_FUNC\_DECL | 声名函数 |
| ZEND\_AST\_CLOSURE | 声名闭包 |
| ZEND\_AST\_ARROW\_FUNC | fn()=>语句 |
| **列表元素语句** | ZEND\_AST\_ARG\_LIST | 读取传入的参数列表 |
| ZEND\_AST\_EXPR\_LIST | 用在for语句和match语句里的单个公式 |
| ZEND\_AST\_IF | if语句 |
| ZEND\_AST\_SWITCH\_LIST | switch语句 |
| ZEND\_AST\_PARAM\_LIST | 声明函数匿名函数fn()=> 或类方法时的参数列表 |
| ZEND\_AST\_CLOSURE\_USES | 闭包use语句里的变量 |
| **没有子节点的语句** | ZEND\_AST\_TYPE | 变量类型或方法的返回类型array/callable/static/mixed |
| ZEND\_AST\_CALLABLE\_CONVERT | (...) 可伸缩形参 |
| **有1个子节点的语句** | ZEND\_AST\_LABEL | 声名goto跳转点 |
| ZEND\_AST\_GOTO | goto 语句 |
| ZEND\_AST\_BREAK | break 语句 |
| ZEND\_AST\_RETURN | return 语句 |
| ZEND\_AST\_CONTINUE | continue语句 |
| **有2个子节点的语句** | ZEND\_AST\_CALL | 函数或匿名函数调用 |
| ZEND\_AST\_WHILE | 整套 while 语句 |
| ZEND\_AST\_DO\_WHILE | 整套 do while 语句 |
| ZEND\_AST\_IF\_ELEM | 整套 if elseif else 语句 |
| ZEND\_AST\_SWITCH | 整套 switch 语句 |
| ZEND\_AST\_SWITCH\_CASE | switch 语句中的每一个case |
| ZEND\_AST\_NAMED\_ARG | 参数列表中每一个规定了类型的参数（形参） |
| **有4个子节点的语句** | ZEND\_AST\_FOR | 整套for循环 |
| ZEND\_AST\_FOREACH | 整套foreach循环 |
| **有5个子节点的语句** | ZEND\_AST\_PARAM | 参数列表中的单个参数 |

在上述分类中，选择类型语句if/switch/goto和循环类型语句while/for/foreach都可算作基本功能。但为了平衡每个部分中的内容量，把它们放到面向过程部分中。

面向对象语句类型有33个：

|  |  |  |
| --- | --- | --- |
| **结构** | **类型值** | **说明** |
| **声明型语句** | ZEND\_AST\_METHOD | 声名类方法 |
| ZEND\_AST\_CLASS | 定义class/trait/interface/enum/匿名类 |
| **列表元素语句** | ZEND\_AST\_CATCH\_LIST | catch语句 |
| ZEND\_AST\_PROP\_DECL | 声明类属性 |
| ZEND\_AST\_CLASS\_CONST\_DECL | 用const 语句语句声明的类常量 |
| ZEND\_AST\_NAME\_LIST | 类名列表中的每一个类名use traitextends implements 时用到的 |
| ZEND\_AST\_TRAIT\_ADAPTATIONS | 一条 use trait 语句后面{} 里的每一段内容 |
| ZEND\_AST\_USE | use引用命名空间名类 |
| **没有子节点的语句** | ZEND\_AST\_CONSTANT\_CLASS | \_\_CLASS\_\_ 魔术常量 |
| **有1个子节点的语句** | ZEND\_AST\_CLONE | clone 语句 |
| ZEND\_AST\_CLASS\_NAME | 通过 ::class 常量获取类名 |
| ZEND\_AST\_THROW | throw 语句 |
| **有2个子节点的语句** | ZEND\_AST\_PROP | 引用非null对象的属性 |
| ZEND\_AST\_NULLSAFE\_PROP | 引用允null对象的属性 |
| ZEND\_AST\_STATIC\_PROP | 类的静态成员变量获取或赋值 |
| ZEND\_AST\_CLASS\_CONST | 访问类常量 |
| ZEND\_AST\_NEW | new 普通类或匿名类 |
| ZEND\_AST\_INSTANCEOF | instanceof 语句 |
| ZEND\_AST\_USE\_TRAIT | 引用trait的一套完整代码 |
| ZEND\_AST\_TRAIT\_PRECEDENCE | 引用一个trait ? |
| ZEND\_AST\_METHOD\_REFERENCE | 引用trait里的指定方法 |
| ZEND\_AST\_NAMESPACE | namespace 语句 |
| ZEND\_AST\_USE\_ELEM | 被use 的每个命名空间 |
| ZEND\_AST\_TRAIT\_ALIAS | 给trait的方法设置别名 |
| ZEND\_AST\_GROUP\_USE | 自定义引用命名空间use namespace\{} |
| ZEND\_AST\_CLASS\_CONST\_GROUP | 一组完整的类常量定义 |
| **有3个子节点的语句** | ZEND\_AST\_METHOD\_CALL | 调用非Null对象方法 |
| ZEND\_AST\_NULLSAFE\_METHOD\_CALL | 允许短路的成员方法调用 $obj?->method() |
| ZEND\_AST\_STATIC\_CALL | 调用类的静态方法 |
| ZEND\_AST\_TRY | try 语句 |
| ZEND\_AST\_CATCH | 单个catch 语句 |
| ZEND\_AST\_PROP\_GROUP | 类里一条完整的属性声明语句 |
| ZEND\_AST\_PROP\_ELEM | 类里每个属性 |
| ZEND\_AST\_METHOD | 声名类方法 |
| ZEND\_AST\_CLASS | 定义class/trait/interface/enum/匿名类 |

特色功能语句类型有22个：

|  |  |  |
| --- | --- | --- |
| **结构** | **类型值** | **说明** |
| **列表元素语句** | ZEND\_AST\_TYPE\_UNION | 联合类型 type1 | type2 |
| ZEND\_AST\_TYPE\_INTERSECTION | 交叉类型 type1 & type2 |
| ZEND\_AST\_ATTRIBUTE\_LIST | 修饰属性列表 |
| ZEND\_AST\_ATTRIBUTE\_GROUP | 放在 #[] 里的修饰属性列表 |
| ZEND\_AST\_MATCH\_ARM\_LIST | match 语句中的条件列表 |
| **没有子节点的语句** | ZEND\_AST\_MAGIC\_CONST | 魔术常量 |
| **有1个子节点的语句** | ZEND\_AST\_UNPACK | ...$array 数组解包装 |
| ZEND\_AST\_SILENCE | 静默操作符 @ |
| ZEND\_AST\_SHELL\_EXEC | 反引号语句 |
| ZEND\_AST\_YIELD\_FROM | yield from语句 |
| ZEND\_AST\_GLOBAL | 声明全局变量 |
| ZEND\_AST\_HALT\_COMPILER | \_\_halt\_compiler 语句 |
| **有2个子节点的语句** | ZEND\_AST\_YIELD | yield 语句 |
| ZEND\_AST\_COALESCE | ?? 连接的两个表达式 |
| ZEND\_AST\_ASSIGN\_COALESCE | ??= 连接的两个表达式 |
| ZEND\_AST\_STATIC | static 后面的每个变量 |
| ZEND\_AST\_DECLARE | declare() {} 后面这部分代码块 |
| ZEND\_AST\_ATTRIBUTE | 修饰属性语句 |
| ZEND\_AST\_MATCH | 整套match语句 |
| ZEND\_AST\_MATCH\_ARM | match语句里的arm语句 |
| **有3个子节点的语句** | ZEND\_AST\_CONST\_ENUM\_INIT | 初始化枚举时的伪节点 |
| **有4个子节点的语句** | ZEND\_AST\_ENUM\_CASE | enum里的 case语句 |

在本篇各个部分内容中，将以上述语句类型为线索来展开介绍各个语句的编译和执行过程。

本篇使用的PHP源码版本为8.2.5。调试环境为64位Win10操作系统、Microsoft Visual Studio 2013。本篇重点介绍64位操作系统中的实现逻辑，忽略32位系统中的实现逻辑，也不进行差异比较。

先从一些基本概念开始。

# 一、基本概念

## 一）PHP的执行过程

PHP的运行入口在源码的sapi目录中，常用的有cgi、fpm、cli等运行模式。以命令行模式（cli）为例，与编译和执行相关的调用过程如下：

|  |
| --- |
| main() : /sapi/cli/php\_cli.c  ->do\_cli() : /sapi/cli/php\_cli.c  ->php\_execute\_script() : /main/main.c  ->zend\_execute\_scripts() : /Zend/zend.c  ->zend\_compile() : /Zend/zend\_language\_scanner.l  ->zend\_compile\_top\_stmt() : /Zend/compile.c  ->zend\_execute() : /Zend/zend\_vm\_execute.h |

其他模式也一样，都要调用到 /main/main.c中的php\_execute\_script()函数，它调用/Zend/zend.c中的zend\_execute\_scripts()函数，这时已经进入PHP内核。再调用zend\_compile()函数编译PHP脚本，调用zend\_execute()函数来执行编译好的操作码。

zend\_execute\_scripts()函数可以每次编译执行多个脚本，注释如下：

|  |
| --- |
| ZEND\_API zend\_result zend\_execute\_scripts(int type, zval \*retval, int file\_count, ...){  ...  va\_start(files, file\_count); // 开始获取文件名列表  for (i = 0; i < file\_count; i++) { // 遍历每个脚本  file\_handle = va\_arg(files, zend\_file\_handle \*); // 取得文件句柄  ...  **// 这里实际上是调用 zend\_compile()函数**  **op\_array = zend\_compile\_file(file\_handle, type); // 编译脚本**  if (file\_handle->opened\_path) { // 如果有路径  // 添加进 EG(included\_files) 包含文件列表里。  zend\_hash\_add\_empty\_element(&EG(included\_files), file\_handle->opened\_path);  }  if (op\_array) { // 如果编译成功  **zend\_execute(op\_array, retval); // 执行操作码**  ...  zend\_destroy\_static\_vars(op\_array); // 销毁静态变量表  destroy\_op\_array(op\_array); // 销毁操作码表  efree\_size(op\_array, sizeof(zend\_op\_array)); // 释放操作码组  // 如使编译失败并且是调用了 require() 或 require\_once() 函数  } else if (type==ZEND\_REQUIRE) {  ret = FAILURE; // 结果为失败  }  }  va\_end(files); // 读取参数列表结束  return ret; // 返回结果  } |

运行一个PHP脚本需要进以下行几个步骤：

1）分词：把PHP脚本分散成一串关键词（token），关键词有几种类型如：变量（variable）、标量（scalar）、语句（statement）关键词等。

2）语法解析：使用定义好的语法规则，把关键词转化成一组语句实例。

3）编译：执行语句实例中的简单运算，如加减乘除等，并把语句实例转化成操作码（opcode）。

4）执行：执行编译好的操作码，完成运算。

其中前3个步骤由zend\_compile()函数来完成，第四个步骤由zend\_execute()函数来完成，下面展开介绍。

## 二）分词和语法解析

分词和语法解析要用到/Zend/目录里的的zend\_language\_parser.y、zend\_ast.c、zend\_ast.h这几个文件中的相关功能。PHP的语法解析规则定义在 /Zend/zend\_language\_parser.y 文件中。zend\_ast.c、zend\_ast.h中定义了\_zend\_ast（语句）结构体和相关的操作方法。

分词和语法解析并非由PHP原生实现，而是借助BISON工具来实现。，在编译PHP源码时，先使用BISON工具把/Zend/zend\_language\_parser.y转换成C语言源码 /Zend/zend\_language\_parser.h和/Zend/zend\_language\_parser.c，再使用它们来进行分词和语法解析（参见Makefile文件）。

关于BISON的介绍参见：

https://git.savannah.gnu.org/cgit/bison.git/tree/data/README.md

### 分词

分词由BISON工具实现，不做详细介绍。PHP8.2.5中有一个tokenizer扩展，通过它可以查看分词效果，例如：

|  |
| --- |
| $t = token\_get\_all("<?php echo 'hello world';");  foreach($t as &$v){  if(is\_array($v)) $v[0] = token\_name($v[0]);  }  echo json\_encode($t);  --------运行结果---------  [ ["T\_OPEN\_TAG","<?php ",1], ["T\_ECHO","echo",1], ["T\_WHITESPACE"," ",1], ["T\_CONSTANT\_ENCAPSED\_STRING","'hello world'",1], ";" ] |

### 语法解析

PHP的语法解析规则定义在 /Zend/zend\_language\_parser.y 文件中，阅读这个文件可以了解全部的语法规则。通过这些规则可以把分好的词转化成语句（zend\_ast）实例。

语法解析规则的定义很简洁，但里面有大量的递归调用，所以逻辑并不简单。

语法解析虽然也借助BISON工具来实现，但创建语句实例的业务逻辑都由PHP自己来实现，例如：

|  |
| --- |
| start:  top\_statement\_list { CG(ast) = $1; (void) zendnerrs; }  ;  top\_statement\_list:  top\_statement\_list top\_statement { $$ = zend\_ast\_list\_add($1, $2); }  | %empty { $$ = zend\_ast\_create\_list(0, ZEND\_AST\_STMT\_LIST); }  ; |

top\_statement\_list规则，定义了最外层php语句列表的解析方式：

1）在第一次进入规则时，调用zend\_ast\_create\_list()函数创建空的语句列表，然后递归调用自身。

2）进入递归后，调用top\_statement规则解析PHP语句，如果解析成功，把解析好的php语句添加到语句列表里，然后继续递归；如果解析失败，结束解析。

可以看出，PHP的脚本转成关键词后是从左到右（脚本转成一串关键词，不分上下）进行解析。一边解析一边创建语句实例，并把语句实例串成树形结构。

start规则是语法解析的入口，每次语法解析都从这里开始，调用top\_statement\_list规则进行解析，完成后把返回的语句关联到CG(ast)指针。

## 三）编译

CG()宏程序用于访问编译时全局变量，相关定义如下：

|  |
| --- |
| struct \_zend\_compiler\_globals { ... } // zend\_globals.h 中定义，编译时全局变量结构体  // zend\_globals\_macros.h 中定义，结构体别名  typedef struct \_zend\_compiler\_globals zend\_compiler\_globals;  // zend\_compile.c 中定义，编译时全局变量的实例  ZEND\_API zend\_compiler\_globals compiler\_globals;  // zend\_globals\_macros.h 中定义，用于访问编译时全局变量中的元素  # define CG(v) (compiler\_globals.v) |

compiler\_globals是\_zend\_compiler\_globals结构体的实例，也就是编译时全局变量。CG()宏用于读取编译时全局变量中的元素。CG(ast)指针指向语句（zend\_ast）树的根结点，zend\_compile()函数从CG(ast)开始编译整个语句树，并返回操作码组（zend\_op\_array）。

zend\_compile()函数的主要业务逻辑如下：

|  |
| --- |
| static zend\_op\_array \*zend\_compile(int type){  ...  if (!zendparse()) { // zendparse()函数用于分词和解析脚本，由程序生成，不在PHP源码中  ...  op\_array = emalloc(sizeof(zend\_op\_array)); // 开辟内存创建操作码组  init\_op\_array(op\_array, type, INITIAL\_OP\_ARRAY\_SIZE); // 创建操作码，初始 64个  ...  zend\_compile\_top\_stmt(CG(ast)); // 从 CG(ast) 开始编译语句树  ...  }  ...  return op\_array;  } |

## 四）执行操作码

如前文所述，编译完成后，调用zend\_execute()函数来执行操作码组，这时要用到PHP虚拟机。

**PHP虚拟机**（Virtual Machine）之所以叫做“虚拟机”是因为它和一个独立的机器（如CPU、显卡、打印机等）一样有自己的指令集，它的指令集就是操作码（opcode）。虚拟机只负责执行操作码，而不是像类库（library）或者框架（framework）一样允许外部程序调用它的成员方法和成员变量。

虚拟是一种设计方法，它可以有效地降低程序设计的复杂度，并减少耦合，例如：

1）把编译和执行相关的业务逻辑分开，降低耦合。

2）把复杂的业务逻辑抽象成少数指令，降低设计上的复杂度。

3）操作码是一个一个执行的，每个操作码的执行逻辑相对独立，可降低虚拟机内部的耦合。

4）很多操作码有多个处理分支，这样可以降低每个分支的复杂度，简化分支中的业务逻辑。

当多次执行同一个脚本程序时（尤其是在web程序中），PHP可以使用缓存的操作码，避免进行多次编译。

zend\_execute()函数用于创建执行数据，并调用zend\_execute\_ex()函数执行操作码，它的主要逻辑如下：

|  |
| --- |
| ZEND\_API void zend\_execute(zend\_op\_array \*op\_array, zval \*return\_value){  ...  // 创建执行数据 \_zend\_execute\_data  execute\_data = zend\_vm\_stack\_push\_call\_frame(call\_info,  (zend\_function\*)op\_array, 0, object\_or\_called\_scope);  ...  // 默认情况下zend\_execute\_ex = execute\_ex  zend\_execute\_ex(execute\_data);  ...  } |

execute\_ex()函数在zend\_vm\_execute.h文件中，有8000多行代码，是为了兼容多种运行模式而编写，其实逻辑并不算复杂，以64位windows系统为例，在默认情况下，实际运行的业务逻辑如下：

|  |
| --- |
| void execute\_ex(zend\_execute\_data \*ex){  zend\_execute\_data \*execute\_data = ex;  ZEND\_VM\_LOOP\_INTERRUPT\_CHECK();  while (1) {  int ret;  // 执行一个操作码：调用当前操作码实例的->handler()函数  // EX(opline) = (execute\_data)->opline  if (UNEXPECTED((ret = ((opcode\_handler\_t) OPLINE->handler) (ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU)) != 0)) {  if (EXPECTED(ret > 0)) { // 返回值大于0时，切换执行数据  execute\_data = EG(current\_execute\_data);  ZEND\_VM\_LOOP\_INTERRUPT\_CHECK();  } else { // 返回值小于0时，结束执行  return; // 最终一定在这里返回  }  }  }  }  // EX()宏程序，访问执行数据中的元素  #define EX(element) ((execute\_data)->element) |

如上所示，execute\_ex()函数的核心业务逻辑是依次调用每个操作码的handler方法，直到返回值小于0时，返回。

以上是一个PHP程序的编译和执行过程，也是本篇的内容框架。下面通过一个“Hello World”程序来展开介绍编译和执行过程中进行的操作。

## 五）Hello World

以一个最简单的Hello World程序为例，来简单说明编译和执行的基本流程：

PHP脚本如下：

|  |
| --- |
| <?php  echo 'Hello World'; |

### 解析源码，创建ZEND\_AST\_ECHO语句

解析过程如下，代码后面的注释中标明了应用的规则名和匹配结果的传递过程：

|  |
| --- |
| zend\_ast \* ast\_1 = zend\_ast\_create\_list(0, ZEND\_AST\_STMT\_LIST); /\* top\_statement\_list:[] \*/  zend\_ast \* ast\_24 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("Hello World", 0)); /\* dereferenceable\_scalar:["'hello world'"] \*/  zend\_ast \* ast\_37 = ast\_24; /\* scalar:["ast\_24"] \*/  zend\_ast \* ast\_14 = ast\_37; /\* expr:["ast\_37"] \*/  zend\_ast \* ast\_13 = zend\_ast\_create(ZEND\_AST\_ECHO, ast\_14); /\* echo\_expr:["ast\_14"] \*/  zend\_ast \* ast\_11 = zend\_ast\_create\_list(1, ZEND\_AST\_STMT\_LIST, ast\_13); /\* echo\_expr\_list:["ast\_13"] \*/  zend\_ast \* ast\_4 = ast\_11; /\* statement:["echo","ast\_11",";"] \*/  zend\_ast \* ast\_3 = ast\_4; /\* top\_statement:["ast\_4"] \*/  zend\_ast \* ast\_90 = zend\_ast\_list\_add(ast\_1, ast\_3); /\* top\_statement\_list:["ast\_1","ast\_3"] \*/  CG(ast) = ast\_90; /\* start:["ast\_90"] \*/ |

如上所示，在解析一个简单的echo语句时，解析器需要进行很多次匹配操作，从序号上可以看出，共进行了90次匹配操作，其中大部分是无效操作，上文列出的是有效的匹配操作。每个脚本的解析过程一定是从创建语句列表zend\_ast\_create\_list(0, ZEND\_AST\_STMT\_LIST)开始，到更新语句树指针CG(ast) = ast\_\*结束。

解析PHP脚本最重要的工作是创建语句（zend\_ast）树，解析完成后，整个语句树也已经创建完毕。从语句序号还可以看出匹配操作进行的顺序以及递归过程。从代码中可以看出，创建了ZEND\_AST\_ECHO类型语句。

### 编译ZEND\_AST\_ECHO语句

编译echo语句需要用到zend\_compile\_echo()函数，相关方法的调用路径如下：

|  |
| --- |
| zend\_compile\_top\_stmt() // 编译顶层语句  ->zend\_compile\_stmt() // 编译语句  ->zend\_compile\_echo() **// 编译echo语句**  ->zend\_emit\_op() // 创建操作码 |

zend\_compile\_echo()函数代码注释如下：

|  |
| --- |
| static void zend\_compile\_echo(zend\_ast \*ast) {  zend\_op \*opline;  zend\_ast \*expr\_ast = ast->child[0]; // echo后面的表达式  znode expr\_node; // 用于接收表达式返回值  **// 编译表达式语句, p1:返回结果，p2:表达式语句**  zend\_compile\_expr(&expr\_node, expr\_ast);  **// 操作码类型 ZEND\_ECHO，不接收返回结果，没有第二个运算对象**  opline = zend\_emit\_op(NULL, ZEND\_ECHO, &expr\_node, NULL);  opline->extended\_value = 0; // 扩展值为0  } |

其中关键操作是调用zend\_compile\_expr()函数编译echo语句后面的表达式。并调用zend\_emit\_op()函数创建ZEND\_ECHO操作码。

### 执行ZEND\_ECHO操作码

在上述Hello World示例用，PHP虚拟机会调用zend\_vm\_execute.h中的如下方法执行ZEND\_ECHO操作码：

|  |
| --- |
| static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_ECHO\_SPEC\_CONST\_HANDLER (ZEND\_OPCODE\_HANDLER\_ARGS){  ...  zend\_write(ZSTR\_VAL(str), ZSTR\_LEN(str));  ...  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION();  } |

这个方法是ZEND\_ECHO操作码的三个处理分支之一，用于输出常量。它的业务逻辑比较简单，主要是调用zend\_write()函数输出打印内容，并调用ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION()宏程序切换下一个操作码。

这样就完成了从语法解析，到编译和执行的所有操作。

执行完所有操作码后，PHP还需要释放内存并销毁相关组件，完成所有析构操作，具体见相关章节。

# 二、赋值操作

本章中，将会通过一个简单的赋值操作，来详细介绍编译和执行中的各个环节。赋值操作本身业务逻辑不算复杂，但运行一个php脚本需要走的整套流程比较繁琐，相关的数据结构与算法也比较多，需要一些耐心去理解。

php代码示例如下：

|  |
| --- |
| <?php  $a=1; |

## 一）解析源码，创建ZEND\_AST\_ASSIGN语句

语法解析过程如下：

|  |
| --- |
| zend\_ast \* ast\_1 = zend\_ast\_create\_list(0, ZEND\_AST\_STMT\_LIST); /\* top\_statement\_list:[] \*/  zend\_ast \* ast\_14 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("$a", 0));; /\* simple\_variable:["$a"] \*/  **zend\_ast \* ast\_13 = zend\_ast\_create(ZEND\_AST\_VAR, ast\_14); /\* callable\_variable:["ast\_14"] \*/**  zend\_ast \* ast\_12 = ast\_13; /\* variable:["ast\_13"] \*/  zend\_ast \* ast\_40 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("1", 0));; /\* dereferenceable\_scalar:["\"1\""] \*/  zend\_ast \* ast\_53 = ast\_40; /\* scalar:["ast\_40"] \*/  zend\_ast \* ast\_30 = ast\_53; /\* expr:["ast\_53"] \*/  **zend\_ast \* ast\_11 = zend\_ast\_create(ZEND\_AST\_ASSIGN, ast\_12, ast\_30); /\*** expr:["ast\_12","=","ast\_30"] \*/  zend\_ast \* ast\_4 = ast\_11; /\* statement:["ast\_11",";"] \*/  zend\_ast \* ast\_3 = ast\_4; /\* top\_statement:["ast\_4"] \*/  zend\_ast \* ast\_88 = zend\_ast\_list\_add(ast\_1, ast\_3); /\* top\_statement\_list:["ast\_1","ast\_3"] \*/  CG(ast) = ast\_88; /\* start:["ast\_88"] \*/ |

如上所示，一个简单的赋值操作，需要创建1个语句列表实例和4个语句实例。1个语句列表是根语句列表，4个语句实例分别用来存放变量名、变量语句、变量值和赋值语句。

### 创建语句列表实例

所有脚本解析的第一步都一样，先调用zend\_ast\_create\_list()函数创建空语句列表，这个方法在不同运行环境中有不同的定义。在64位windows系统中，这里调用的是zend\_ast.h中定义的zend\_ast\_create\_list()宏程序，相关定义如下：

|  |
| --- |
| # define zend\_ast\_create\_list(init\_children, ...) \  ZEND\_AST\_SPEC\_CALL(zend\_ast\_create\_list, \_\_VA\_ARGS\_\_)  // 根据传入的参数数量，自动适配调用的方法名，并调用方法  # define ZEND\_AST\_SPEC\_CALL(name, ...) \  ZEND\_EXPAND\_VA(ZEND\_AST\_SPEC\_CALL\_(name, \_\_VA\_ARGS\_\_, \_5, \_4, \_3, \_2, \_1, \_0)(\_\_VA\_ARGS\_\_))  // 在调用方法名后面拼接数字  # define ZEND\_AST\_SPEC\_CALL\_(name, \_, \_5, \_4, \_3, \_2, \_1, suffix, ...) name ## suffix  // zend\_portability.h中定义。64位操作系统中，直接应用传入的代码  # define ZEND\_EXPAND\_VA(code) code |

ZEND\_AST\_SPEC\_CALL()宏程序会根据参数数量调zend\_ast\_create\_list\_0() - zend\_ast\_create\_list\_2()这3个方法中的一个，这几个方法逻辑相似，以zend\_ast\_create\_list\_2()函数为例：

|  |
| --- |
| ZEND\_API zend\_ast \* zend\_ast\_create\_list\_2(zend\_ast\_kind kind, zend\_ast \*child1, zend\_ast \*child2) {  zend\_ast \*ast;  zend\_ast\_list \*list;  uint32\_t lineno;  ast = zend\_ast\_alloc(zend\_ast\_list\_size(4)); // 分配内存创建语句列表  list = (zend\_ast\_list \*) ast;  list->kind = kind;  list->attr = 0;  list->children = 2; // 子语句数量  list->child[0] = child1; // 关联两个子语句  list->child[1] = child2;  if (child1) {  lineno = zend\_ast\_get\_lineno(child1);  if (lineno > CG(zend\_lineno)) { lineno = CG(zend\_lineno); }  } else if (child2) {  lineno = zend\_ast\_get\_lineno(child2);  if (lineno > CG(zend\_lineno)) { lineno = CG(zend\_lineno); }  } else {  list->children = 0;  lineno = CG(zend\_lineno);  }  list->lineno = lineno; // 更新最后的行号  return ast;  } |

zend\_ast\_list\_size()函数用于计算语句列表的大小：

|  |
| --- |
| static size\_t zend\_ast\_list\_size(uint32\_t children) {  // 内存大小 = zend\_ast\_list的大小 + （子语句数-1）\* 语句指针大小  return sizeof(zend\_ast\_list) - sizeof(zend\_ast \*) + sizeof(zend\_ast \*) \* children;  } |

实例中的指针数量要比子语句数少1个，是因为实例本身自带一个语句指针，实际上最终分配的指针数和子语句数相等。语句列表结构定义如下：

|  |
| --- |
| typedef struct \_zend\_ast\_list {  zend\_ast\_kind kind;  zend\_ast\_attr attr;  uint32\_t lineno; // 行号  uint32\_t children; // 子语句数量  zend\_ast \*child[1]; // 子语句指针列表  } zend\_ast\_list; |

### 创建语句实例

在解析过程中，所有PHP语句中声名变量名会先被解析成zval语句（语句类型为ZEND\_AST\_ZVAL），再在外面加一层变量语句（类型为ZEND\_AST\_VAR）。如上文中ast\_14和ast\_13的两行。

赋值语句的类型为ZEND\_AST\_ASSIGN，它有两个子语句。

## 二）编译ZEND\_AST\_ASSIGN语句

编译赋值（ZEND\_AST\_ASSIGN）语句在本例中的调用路径如下：

|  |
| --- |
| zend\_compile\_stmt() // 编译语句  ->zend\_compile\_expr() // 编译表达式  ->zend\_compile\_expr\_inner() // 根据不同的表达式类型，调用不同的方法处理  ->zend\_compile\_assign() // 编译赋值语句  ->zend\_delayed\_compile\_var() // 编译变量名  ->zend\_compile\_expr() // 递归，编译赋值语句中的值表达式 “1”  >zend\_compile\_expr\_inner() // 把值表达式转为运算结果 znode |

如上所示，一个简单的赋值语句，它的处理过程并不简单，甚至用到了递归。

zend\_compile\_assign()函数中处理了多种情况的赋值，与ZEND\_AST\_VAR语句相关的业务逻辑可以分为三步：1）编译变量名；2）编译值表达式；3）创建赋值操作码。

代码注释如下：

|  |
| --- |
| static void zend\_compile\_assign(znode \*result, zend\_ast \*ast) {  zend\_ast \*var\_ast = ast->child[0]; // 接收变量名  zend\_ast \*expr\_ast = ast->child[1]; // 变量值 表达式  znode var\_node, expr\_node; // 操作对象临时变量  ...  switch (kind) { // 根据语句类型处理  case ZEND\_AST\_VAR: // 如是普通赋值语句  offset = zend\_delayed\_compile\_begin(); // 开始延时编译，获取起始堆栈高度  zend\_delayed\_compile\_var(&var\_node, var\_ast, BP\_VAR\_W, 0); //1）编译接收变量名  zend\_compile\_expr(&expr\_node, expr\_ast); // 2）编译表达式语句  zend\_delayed\_compile\_end(offset); // 结束延时编译  CG(zend\_lineno) = zend\_ast\_get\_lineno(var\_ast); // 取得赋值语句行号  // 3）创建临时操作码 ZEND\_ASSIGN  zend\_emit\_op\_tmp(result, ZEND\_ASSIGN, &var\_node, &expr\_node);  return; |

这里用到znode结构体，znode结构体用来存放语句（zend\_ast）的编译结果，定义如下：

|  |
| --- |
| typedef struct \_znode { // 只有编译过程中用到，占用6Bytes  zend\_uchar op\_type; // 操作码类型，1Byte  zend\_uchar flag; // 标记 1Byte  union { // 4Bytes  znode\_op op; // 运算对象，4Bytes  zval constant; // 常量，4Bytes  } u;  } znode; |

znode结构体中包含两个zend\_uchar元素，每个占1Byte；和一个联合类型（union）元素u，它的子类型可以是运算对象（znode\_op）或变量（zval）。

运算对象（znode\_op）结构体定义如下：

|  |
| --- |
| typedef union \_znode\_op { // 运算对象，占4Bytes，32位  uint32\_t constant; // 在 \_zend\_execute\_data 中，此实例的常量的序号  uint32\_t var; // 在 \_zend\_execute\_data 中，此实例的变量的序号  uint32\_t num; // 创建过程中的序号  uint32\_t opline\_num; // 操作码行号  uint32\_t jmp\_offset; // 跳转偏移量  } znode\_op;  typedef unsigned char zend\_uchar; // 无符号字符的别名 |

如上所示，znode\_op结构体是联合类型(union)，占用4Bytes空间，它的所有子类型都是32位整数，只是名称不同，用来表示不同的用途。zend\_uchar类型是无符号字符类型的别名。

znode和znode\_op都是在编译过程中非常常用的结构体。

### 1、编译变量名

在编译变量名时，调用zend\_delayed\_compile\_var()函数，调用路径如下：

|  |
| --- |
| zend\_delayed\_compile\_var() // 延时编译变量  ->zend\_compile\_simple\_var() // 编译简单变量  ->zend\_try\_compile\_cv() // 编译变量  ->lookup\_cv() // 查找变量，失败时创建变量 |

编译变量名看似简单，实际上比较复杂，因为变量名可能是数组元素、对象属性等复杂类型。在本例中，使用了最简单的变量名，跳过了很多复杂的操作，但创建变量的过程仍然比较繁琐。

zend\_try\_compile\_cv()函数用于编译变量名，其关键步骤是1）用变量名创建保留字串；2）调用lookup\_cv()函数创建编译变量，并返回变量在执行数据中的偏移量，代码注释如下：

|  |
| --- |
| static zend\_result zend\_try\_compile\_cv(znode \*result, zend\_ast \*ast){  zend\_ast \*name\_ast = ast->child[0]; // 取得第一个子语句，变量名  if (name\_ast->kind == ZEND\_AST\_ZVAL) { // 变量名语句必须是内置变量语句  zval \*zv = zend\_ast\_get\_zval(name\_ast); // 直接获取语句里的变量  zend\_string \*name;  // 从变量 zval中 获取 变量名 name  if (EXPECTED(Z\_TYPE\_P(zv) == IS\_STRING)) { // 如果变量类型是字串  name = zval\_make\_interned\_string(zv); // 创建保留字串  } else { // 否则  // 从zv中获取字串，并创建保留字串  name = zend\_new\_interned\_string(zval\_get\_string\_func(zv));  }  if (zend\_is\_auto\_global(name)) { // 如果变量名属于自动全局变量\_GET,\_POST等  return FAILURE; // 失败  }  result->op\_type = IS\_CV; // znode类型标记成编译变量  result->u.op.var = lookup\_cv(name); // znode的运算对象中保存变量的偏移量  if (UNEXPECTED(Z\_TYPE\_P(zv) != IS\_STRING)) { // 如果类不型是 string  zend\_string\_release\_ex(name, 0); // 释放变量名  }  return SUCCESS;  }  return FAILURE;  } |

如上所示，变量名被创建成保留字串（具体参见字串篇）。然后调用lookup\_cv()函数创建变量，每个变量为一个zval实例，代码注释如下：

|  |
| --- |
| static int lookup\_cv(zend\_string \*name) { // 查找编译变量，如果没有创建一个新的并返回序号  zend\_op\_array \*op\_array = CG(active\_op\_array); // 当前操作码列表  int i = 0; // 变量序号 0  zend\_ulong hash\_value = zend\_string\_hash\_val(name); // 取得name的哈希值，long型  // 遍历 op\_array 中的 CV  while (i < op\_array->last\_var) {  // 如果当前变量名的哈希值和要查找的哈希值相同， 并且当前变量名和要查找的变量名相同  if (ZSTR\_H(op\_array->vars[i]) == hash\_value  && zend\_string\_equals(op\_array->vars[i], name)) {  return EX\_NUM\_TO\_VAR(i); // 返回变量序号  }  i++; // 下一个  }  i = op\_array->last\_var; // 变量列表中的序号 = 当前变量数  op\_array->last\_var++; // 变量数增加1个  if (op\_array->last\_var > CG(context).vars\_size) { // 如果已分配的变量不够用  CG(context).vars\_size += 16; // 再多分配16个变量的空间  // 分配内存，用于存放变量名  op\_array->vars = erealloc(op\_array->vars, CG(context).vars\_size \* sizeof(zend\_string\*));  }  op\_array->vars[i] = zend\_string\_copy(name); // 把变量名复制到列表中  return EX\_NUM\_TO\_VAR(i); // 返回在变量列表中的偏移量  } |

EX\_NUM\_TO\_VAR()宏程序用于把变量序号转成变量在执行数据中的偏移量：

|  |
| --- |
| // 跳过头数据大小 ZEND\_CALL\_FRAME\_SLOT=2  #define EX\_NUM\_TO\_VAR(n) ((uint32\_t)(((n) + ZEND\_CALL\_FRAME\_SLOT) \* sizeof(zval)))  // 64位windows里值为 2  #define ZEND\_CALL\_FRAME\_SLOT \  ((int)((ZEND\_MM\_ALIGNED\_SIZE(sizeof(zend\_execute\_data)) + ZEND\_MM\_ALIGNED\_SIZE(sizeof(zval)) - 1) / ZEND\_MM\_ALIGNED\_SIZE(sizeof(zval)))) |

如上所示：编译变量名的过程，记录了变量名，并把变量的类型和偏移更新到znode实例中返回。在本例中，更新过的znode实例如下：

|  |
| --- |
| znode result = { op\_type：8(IS\_CV)，flag: 未用到，u：{ op：80(变量指针偏移量) } } |

### 2、编译值表达式

在本例中，值表达式很简单，如前文调用路径中所示，由zend\_compile\_expr()函数调用zend\_compile\_expr\_inner()函数处理，相关业务逻辑如下：

|  |
| --- |
| static void zend\_compile\_expr\_inner(znode \*result, zend\_ast \*ast) {  switch (ast->kind) { // 按类型操作  case ZEND\_AST\_ZVAL: // 内置变量  ZVAL\_COPY(&result->u.constant, zend\_ast\_get\_zval(ast)); // 复制语句中的zval  result->op\_type = IS\_CONST; // 类型为常量  return; |

如上所示，使用zend\_ast\_get\_zval()函数取出语句中的zval变量，复制给&result->u.constant，再把类型（result->op\_type）更新成常量（IS\_CONST）。更新过的znode实例如下：

|  |
| --- |
| znode result = { op\_type：1(IS\_CONST)，flag: 未用到，u：{ constant(zval实例)：{ value："1"(zend\_string实例)，u1：{ type：6，type\_flags：1，u：未用到 }，u2：未用到 } } } |

这样变量和值被编译成了两个znode，可以开始创建操作码了。

### 3、创建操作码

如前文zend\_compile\_assign()函数所示，调用zend\_emit\_op\_tmp()函数创建操作码。过程中需要调用一系列的方法和宏程序，调用路径如下：

|  |
| --- |
| zend\_compile\_assign() // 编译赋值语句  ->zend\_emit\_op\_tmp() // 创建临时操作码  ->get\_next\_op() // 取得新操作码实例  ->init\_op() // 初始化新操作码实例  ->MAKE\_NOP() // 把新操作码实例重置为 ZEND\_NOP  ->SET\_NODE() // 把外部创建的运算对象复制给操作码  ->zend\_add\_literal() // 创建文本  ->zend\_insert\_literal() // 添加文本  ->zend\_make\_tmp\_result() // 创建临时结果变量  ->get\_temporary\_variable() // 取得变量序号 |

zend\_emit\_op\_tmp()函数和zend\_emit\_op()函数用于创建操作码，在编译中被大量引用，函数中用到操作码（zend\_op）和运算对象（znode\_op）两个数据结构。

操作码（zend\_op）结构体在compile.h中定义，结构如下：

|  |
| --- |
| struct \_zend\_op { // 操作码。程序里变量名经常是 op\_line  const void \*handler; // 处理器，只有这一个元素是指针  // 这三个都是znode\_op实例，不是指针。op 是指 operand，运算对象  znode\_op op1; // 第一个运算对象  znode\_op op2; // 第二个运算对象  znode\_op result; // 运算结果  uint32\_t extended\_value; // 扩展值  uint32\_t lineno; // 行号  zend\_uchar opcode; // 操作码编号  // 上面3个运算对象的类型  zend\_uchar op1\_type;  zend\_uchar op2\_type;  zend\_uchar result\_type;  };  typedef struct \_zend\_op zend\_op; |

运算对象（znode\_op）结构体在compile.h中定义，结构如下：

|  |
| --- |
| typedef union \_znode\_op {  uint32\_t constant; // 在 \_zend\_execute\_data 中 本node 的常量的偏移量  uint32\_t var; // 在 \_zend\_execute\_data 中 本node 的变量的偏移量  uint32\_t num; // 创建过程中的序号，例如，表示这是第几个参数  uint32\_t opline\_num; // 操作码顺序号  uint32\_t jmp\_offset; // 跳转偏移量  } znode\_op; |

zend\_emit\_op\_tmp()函数的代码注释如下：

|  |
| --- |
| static zend\_op \*zend\_emit\_op\_tmp(znode \*result, zend\_uchar opcode, znode \*op1, znode \*op2){  zend\_op \*opline = get\_next\_op(); **// 创建新操作码**  opline->opcode = opcode; // 设置 opcode  // 如果有传入，更新两个运算对象 和运算结果  if (op1 != NULL) SET\_NODE(opline->op1, op1);  if (op2 != NULL) SET\_NODE(opline->op2, op2);  if (result) zend\_make\_tmp\_result(result, opline);  return opline; // 返回新操作码指针  } |

如上所示，zend\_emit\_op\_tmp()函数只更新opline->op1和opline->op2两个元素，并没有修改opline->result元素。

zend\_emit\_op\_tmp()函数接收4个参数：第一个参数是接收返回结果用的znode；第二个参数是操作码，类型为zend\_uchar，在本例中使用赋值操作码ZEND\_ASSIGN；第三个参数是存放变量名的znode，第四个参数是存放表达式值的znode。

创建一个操作码的业务逻辑可以细分为4个步骤：a）创建操作码(zend\_op)实例；b）初始化操作码(zend\_op)实例；c）复制运算对象；d）创建运算结果变量。

#### **a）创建操作码(zend\_op)实例**

get\_next\_op()函数用于获取新操作码，每个操作码是一个zend\_op实例，代码注释如下：

|  |
| --- |
| static zend\_op \*get\_next\_op(void){  zend\_op\_array \*op\_array = CG(active\_op\_array); // 正在编译的操作码组  uint32\_t next\_op\_num = op\_array->last++; // 新操作码的序号  zend\_op \*next\_op; // 新操作码指针  if (UNEXPECTED(next\_op\_num >= CG(context).opcodes\_size)) { // 如果已分配的不够用  // 初始大小为 INITIAL\_OP\_ARRAY\_SIZE：64  CG(context).opcodes\_size \*= 4; // 大小翻4倍  op\_array->opcodes = erealloc(op\_array->opcodes, CG(context).opcodes\_size \* sizeof(zend\_op)); // 分配内存创建新操作码（这时还没有初始化）  }  next\_op = &(op\_array->opcodes[next\_op\_num]); // 新操作码指针  init\_op(next\_op); // 初始化新操作码  return next\_op; // 返回新操作码指针  } |

这里用到了操作码（zend\_op）结构体，它的的定义如下：

|  |
| --- |
| struct \_zend\_op { // 操作码，占32Bytes，256位  const void \*handler; // 处理器，只有这一个元素是指针。64位系统占8Bytes，64位  znode\_op op1; // 这三个都是znode\_op实例，不是指针。op 是指 operand，运算对象  znode\_op op2;  znode\_op result; // 3个运算对象\*4=12Bytes  uint32\_t extended\_value; // 扩展值，4Bytes  uint32\_t lineno; // 行号 ，4Bytes  zend\_uchar opcode; // 操作码编号，1Byte  zend\_uchar op1\_type; // 上面3个运算对象的类型，3个共3Bytes  zend\_uchar op2\_type;  zend\_uchar result\_type;  };  typedef struct \_zend\_op zend\_op; |

如上所示，以及前文中znode结构体的定义中可以看到，用到运算对象（zend\_op）时，都需要增加一个zend\_uchar型的运算对象类型（如op1\_type、op2\_type、result\_type），但在zend\_op结构体设计中并没有把类型包含进去，这是为了控制结构体的大小。操作码（zend\_op）结构体的大小是32Bytes，运算对象（zend\_op）结构体的大小是4Bytes，zend\_value结构体的大小是8Bytes，zval结构体的大小是16Bytes，这些结构体的使用量非常大，它们的尺寸都是被精心设计过的，大小都是2的幂，这样有利于内存的分配和寻地址，可以提升内存使用性能。具体可参见内存管理篇和数据类型篇。

#### **b）初始化操作码**

get\_next\_op()函数中调用到init\_op()函数和MAKE\_NOP()宏程序初始化操作码，它们的业务逻辑比较简单，注释如下：

|  |
| --- |
| static void init\_op(zend\_op \*op){  MAKE\_NOP(op); // 重置操作码  op->extended\_value = 0; // 扩展值为0  op->lineno = CG(zend\_lineno); // 记录行号  }  #define MAKE\_NOP(opline) do { /\* 重置操作码 \*/ \  (opline)->op1.num = 0; /\* 3个运算对象序号都为0 \*/ \  (opline)->op2.num = 0; \  (opline)->result.num = 0; \  (opline)->opcode = ZEND\_NOP; /\* 类型为未使用 \*/ \  SET\_UNUSED((opline)->op1); /\* 重置3个运算对象 \*/ \  SET\_UNUSED((opline)->op2); \  SET\_UNUSED((opline)->result); \  } while (0)  #define SET\_UNUSED(op) do { /\* 重置运算对象 \*/ \  op ## \_type = IS\_UNUSED; /\* 重置类型 \*/ \  op.num = (uint32\_t) -1; /\* 重置变量编号 \*/ \  } while (0) |

#### **c）复制运算对象**

SET\_NODE()宏程序用于把zonde中的运算对象复制到操作码中，代码注释如下：

|  |
| --- |
| #define SET\_NODE(target, src) do { /\* 把运算对象复制给操作码 \*/ \  target ## \_type = (src)->op\_type; \  if ((src)->op\_type == IS\_CONST) { /\* 如果src的类型是常量 \*/ \  target.constant = zend\_add\_literal(&(src)->u.constant); \  } else { /\* src的类型不是常量，直接复制32位整数 \*/ \  target = (src)->u.op; \  } \  } while (0) |

当运算对象类型为常量时，会调用zend\_add\_literal()函数创建literal。在本例中，值为常量。zend\_add\_literal()函数和相关方法的定义如下：

|  |
| --- |
| static int zend\_add\_literal(zval \*zv) { // 开辟空间  zend\_op\_array \*op\_array = CG(active\_op\_array); // 当前操作码列表  int i = op\_array->last\_literal; // literal数量，也是本次的编号  op\_array->last\_literal++; // 数量 +1  if (i >= CG(context).literals\_size) { // 如果空间不够用  while (i >= CG(context).literals\_size) { // 每次增加16个，直到够用  CG(context).literals\_size += 16;  }  // 重新分配内存，并返回地址。每个literal是一个zval  op\_array->literals = (zval\*)erealloc(op\_array->literals, CG(context).literals\_size \* sizeof(zval));  }  zend\_insert\_literal(op\_array, zv, i); // 给op\_array 添 加literal  return i; // 返回本次添加的 literal 序号  }  // 添加literal  static void zend\_insert\_literal(zend\_op\_array \*op\_array, zval \*zv, int literal\_position) {  zval \*lit = CT\_CONSTANT\_EX(op\_array, literal\_position); // 找到zval  if (Z\_TYPE\_P(zv) == IS\_STRING) { // 如果是 zend\_string 类型  zval\_make\_interned\_string(zv); // 用zv创建保留字串  }  ZVAL\_COPY\_VALUE(lit, zv); // 赋值 literals 中的元素赋值  Z\_EXTRA\_P(lit) = 0; // (zval).u2.extra 设置成0  }  // 使用序号，找到操作码组中的 literal的指针  #define CT\_CONSTANT\_EX(op\_array, num) ((op\_array)->literals + (num)) |

#### **d）创建运算结果变量**

最后一步是调用zend\_make\_tmp\_result()函数创建结果变量，方法注释如下：

|  |
| --- |
| static inline void zend\_make\_tmp\_result(znode \*result, zend\_op \*opline) {  opline->result\_type = IS\_TMP\_VAR; // 类型为临时变量  opline->result.var = get\_temporary\_variable(); // 增加变量计数  GET\_NODE(result, opline->result);  } |

get\_temporary\_variable()函数并没有分配变量，只是增加了变量的计数，并返回需要的变量序号，真正的创建操作是在创建执行数据（execute\_data）时进行。方法代码如下：

|  |
| --- |
| static uint32\_t get\_temporary\_variable(void) {  return (uint32\_t)CG(active\_op\_array)->T++;  } |

以上每一步的操作并不算复杂，但整个创建过程还是比较繁琐的，主要原因是用到的几个结构体有些复杂，这几个结构体是编译过程中的关键，后继还要反复用到它们。

zend\_emit\_op()函数和zend\_emit\_op\_tmp()的不同之处在于，它使用zend\_make\_var\_result()函数创建结果变量（zval）：

|  |
| --- |
| static void zend\_make\_var\_result(znode \*result, zend\_op \*opline) {  opline->result\_type = IS\_VAR; // 类型为变量  opline->result.var = get\_temporary\_variable(); // 增加变量计数  GET\_NODE(result, opline->result); // 把操作码的运行结果返回给 result  } |

如代码中所示，zend\_make\_tmp\_result()函数和zend\_make\_var\_result()函数的不同之处，仅仅在于它们创建的操作码的结果变量的类型不同。所以zend\_emit\_op\_tmp()函数创建的操作码，结果变量类型为临时变量（IS\_TMP\_VAR），zend\_emit\_op()函数创建的操作码，结果变量类型为普通变量（IS\_VAR）。

在整个编译过程中，zend\_compile\_stmt()、zend\_compile\_expr()、zend\_compile\_expr\_inner()、zend\_compile\_assign()这几个方法的返回值类型都是void，它们工作的重点都不在返回结果中，而是在过程中创建的操作码列表，也就是CG(active\_op\_array) 。

## 三）编译的后续处理

在zend\_compile()函数的注释中可以看到，创建完所有操作码后，需要调用pass\_two()函数对操作码列表进行后续处理。后续处理分为几个步骤：

1）处理用户操作码（本例中没有用到）；

2）释放多余的变量名。如lookup\_cv()函数中所示，为了减少创建次数，每次创建变量名会创建16个，这一步里会把没有用到的变量名释放掉；

3）调整操作码内存，把literal合并到操作码中。这时会回收多余的literal空间，原理同上；

4）更新上下文中的变量数量；

5）处理每一个操作码，并为操作码绑定执行方法；

6）计算操作码活动区域，在本例中不需要创建活动区域，这部分放在后续章节中。

pass\_two()函数在zend\_opcode.c中定义，代码注释如下：。

|  |
| --- |
| ZEND\_API void pass\_two(zend\_op\_array \*op\_array) {  zend\_op \*opline, \*end;  ... // 步骤1：处理用户操作码    if (CG(context).vars\_size != op\_array->last\_var) { // 步骤2：释放多余的变量名  // 调整内存，释放多余的内存  op\_array->vars = (zend\_string\*\*) erealloc(op\_array->vars, sizeof(zend\_string\*)\*op\_array->last\_var);  CG(context).vars\_size = op\_array->last\_var; // 更新 编译变量的数量  }  // 步骤3：调整操作码内存，把literal合并到操作码中  op\_array->opcodes = (zend\_op \*) erealloc(op\_array->opcodes,  ZEND\_MM\_ALIGNED\_SIZE\_EX(sizeof(zend\_op) \* op\_array->last, 16) +  sizeof(zval) \* op\_array->last\_literal);  // 迁移literal  if (op\_array->literals) { // 如果有 literals  // 复制 literals  memcpy(((char\*)op\_array->opcodes) + ZEND\_MM\_ALIGNED\_SIZE\_EX(sizeof(zend\_op) \* op\_array->last, 16),  op\_array->literals, sizeof(zval) \* op\_array->last\_literal);  efree(op\_array->literals); // 释放原来的 literals  // 使用新 literals 列表  op\_array->literals = (zval\*)(((char\*)op\_array->opcodes) + ZEND\_MM\_ALIGNED\_SIZE\_EX(sizeof(zend\_op) \* op\_array->last, 16));  }  // 步骤4：更新变量数量  CG(context).opcodes\_size = op\_array->last; // 更新上下文中的操作码数量  CG(context).literals\_size = op\_array->last\_literal; // 更新上下文中的literals 数量  op\_array->T += ZEND\_OBSERVER\_ENABLED; // 如果开启了观察者，添加一个临时变量  op\_array->fn\_flags |= ZEND\_ACC\_DONE\_PASS\_TWO; // 添加完成pass\_two标记  // 步骤5：处理每一个操作码  opline = op\_array->opcodes; // 操作码列表的开头  end = opline + op\_array->last; // 操作码列表的结尾  while (opline < end) { // 遍历每个操作码  // 步骤5.1：一些操作码的特殊处理  ...  // 步骤5.2：处理znode中的变量  if (opline->op1\_type == IS\_CONST) { // 如果 操作对象1 类型是常量  // 把常量从编译时转到运行时: (p3).zv = p1->literals + p3.constant  ZEND\_PASS\_TWO\_UPDATE\_CONSTANT(op\_array, opline, opline->op1);  } else if (opline->op1\_type & (IS\_VAR|IS\_TMP\_VAR)) { // 如果op1是变量或临时变量  // 把 zval 序号 转成偏移量。临时变量排在编译变量的后面，所以+ last\_var，下同  opline->op1.var = EX\_NUM\_TO\_VAR(op\_array->last\_var + opline->op1.var);  }    if (opline->op2\_type == IS\_CONST) { // 如果 操作对象2 类型是常量  // 把常量从编译时转到运行时: (p3).zv = p1->literals + p3.constant  ZEND\_PASS\_TWO\_UPDATE\_CONSTANT(op\_array, opline, opline->op2);  } else if (opline->op2\_type & (IS\_VAR|IS\_TMP\_VAR)) { // 如果op2 是变量或临时变量  // 把 zval 序号 转成偏移量  opline->op2.var = EX\_NUM\_TO\_VAR(op\_array->last\_var + opline->op2.var);  }    if (opline->result\_type & (IS\_VAR|IS\_TMP\_VAR)) { // 如果result是变量或临时变量  // 把 zval 序号 转成偏移量  opline->result.var = EX\_NUM\_TO\_VAR(op\_array->last\_var + opline->result.var);  }  // 步骤5.3：给操作码绑定它对应的处理器，调用 zend\_vm\_set\_opcode\_handler()函数 ZEND\_VM\_SET\_OPCODE\_HANDLER(opline);  opline++; // 下一个操作码  }  // 步骤6：计算操作码活动区域。p1：操作码组，p2：zend\_needs\_live\_range\_cb  zend\_calc\_live\_ranges(op\_array, NULL);  return;  } |

代码中前4个步骤比较简单。重点介绍第5个步骤。

### 为每个操作码做后续处理

如上文代码注释中步骤5.1-5.3所示，这部分业务逻辑又可以分为3个步骤。

#### **1）为一些操作码做特殊处理**

本例中使用的ZEND\_ASSIGN操作码不需要特殊处理，这部分在后续示例中介绍。

#### **2）把运算对象（znode\_op）从编译时（compile-time）转到运行时（run-time）**

3个运算对象（znode\_op）的处理过程大致相同：如果类型是变量或临时变量，调用EX\_NUM\_TO\_VAR()宏程序把变量序号转成偏移量；如果类型是常量，调用ZEND\_PASS\_TWO\_UPDATE\_CONSTANT()宏程序，把常量序号转成相对于操作码的偏移量。

EX\_NUM\_TO\_VAR()宏程序前文中已经介绍过。在64位windows系统中，ZEND\_PASS\_TWO\_UPDATE\_CONSTANT()宏程序注释如下：

|  |
| --- |
| # define ZEND\_PASS\_TWO\_UPDATE\_CONSTANT(op\_array, opline, node) do { \  (node).constant = \  (((char\*)CT\_CONSTANT\_EX(op\_array, (node).constant)) - \  ((char\*)opline)); \  } while (0)  // 使用序号，找到操作码组中的 literal的指针  #define CT\_CONSTANT\_EX(op\_array, num) ((op\_array)->literals + (num)) |

如上所示，先用运算对象（znode\_op）中保存的序号，找到常量在literals列表中的指针，计算出它和操作码之间的偏移量（也就是在内存中相差的字节数），再把偏移量更新到运算对象中。

#### **3）给操作码链接它对应的处理器**

ZEND\_PASS\_TWO\_UPDATE\_CONSTANT()宏程序是zend\_vm\_set\_opcode\_handler()函数的别名，用于给操作码绑定处理器。相关代码注释如下：

|  |
| --- |
| #define ZEND\_VM\_SET\_OPCODE\_HANDLER(opline) zend\_vm\_set\_opcode\_handler(opline)  // 给操作码绑定它对应的处理器  ZEND\_API void zend\_vm\_set\_opcode\_handler(zend\_op\* op) {  zend\_uchar opcode = zend\_user\_opcodes[op->opcode]; // 用户操作码  // 如果两个运算对象可交换  if (zend\_spec\_handlers[op->opcode] & SPEC\_RULE\_COMMUTATIVE) {  if (op->op1\_type < op->op2\_type) { // 如果第二个类型值更大  zend\_swap\_operands(op); // 交换两个运算对象  }  }  // 绑定特定的处理器  op->handler = zend\_opcode\_handlers[zend\_vm\_get\_opcode\_handler\_idx(  zend\_spec\_handlers[opcode], op)];  }  // 256 个操作码的序号  static zend\_uchar zend\_user\_opcodes[256] = {0,1,2,...,255};  // 256个操作码的配置信息  static const uint32\_t specs[] = {  0,  1 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2,  26 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2,  51 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2 | SPEC\_RULE\_COMMUTATIVE,  ...  3450,  };  // 所有操作码所有分支的处理器，共3451 个，其中有很多重复  static const void \* const labels[] = {  (void\*)&&ZEND\_NOP\_SPEC\_LABEL,  (void\*)&&ZEND\_ADD\_SPEC\_CONST\_CONST\_LABEL,  (void\*)&&ZEND\_ADD\_SPEC\_CONST\_TMPVARCV\_LABEL,  ...  } |

zend\_swap\_operands()函数用于交换两个操作码的运算对象，代码很简单：

|  |
| --- |
| static void zend\_swap\_operands(zend\_op \*op) {  znode\_op tmp;  zend\_uchar tmp\_type;  tmp = op->op1; // 交换运算对象  tmp\_type = op->op1\_type; // 交换运算对象类型  op->op1 = op->op2;  op->op1\_type = op->op2\_type;  op->op2 = tmp;  op->op2\_type = tmp\_type;  } |

zend\_vm\_get\_opcode\_handler\_id()函数用于查找处理器编号：

|  |
| --- |
| static uint32\_t zend\_vm\_get\_opcode\_handler\_idx(uint32\_t spec, const zend\_op\* op){  // {3, 0, 1, 3, 2, 3, 3, 3, 4}  static const int zend\_vm\_decode[] = {  \_UNUSED\_CODE, // 0未使用  \_CONST\_CODE, // 1常量  \_TMP\_CODE, // 2临时变量  \_UNUSED\_CODE, // 3  \_VAR\_CODE, // 4普通变量  \_UNUSED\_CODE, // 5  \_UNUSED\_CODE, // 6  \_UNUSED\_CODE, // 7  \_CV\_CODE // 8编译变量  };  uint32\_t offset = 0; // 初始索引号0  // 如果op1有特殊规则  if (spec & SPEC\_RULE\_OP1) offset = offset \* 5 + zend\_vm\_decode[op->op1\_type];  // 如果op2有特殊规则  if (spec & SPEC\_RULE\_OP2) offset = offset \* 5 + zend\_vm\_decode[op->op2\_type];  // 如果有扩展规则  if (spec & SPEC\_EXTRA\_MASK) {  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);  } 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;  } |

在本例中，zend\_vm\_get\_opcode\_handler\_idx()函数的执行过程如下：

|  |
| --- |
| // 686 | SPEC\_RULE\_OP1 | SPEC\_RULE\_OP2 | SPEC\_RULE\_RETVAL  uint32\_t spec = 686 | 65536 | 131072 | 524288 = 721582;  uint32\_t offset = 0; // 初始索引号0  offset = offset \* 5 + zend\_vm\_decode[op->op1\_type]; // 0\*5+4=4  offset = offset \* 5 + zend\_vm\_decode[op->op2\_type]; // 4\*5+0=20  // 因为有 SPEC\_RULE\_RETVAL标记，op->result\_type为0  offset = offset \* 2 + (op->result\_type != IS\_UNUSED); // 20\*2+0=40  return (spec & SPEC\_START\_MASK) + offset; // ( 721582 & 0x0000ffff ) + 40 = 726 |

得到索引号726，处理器为ZEND\_ASSIGN\_SPEC\_CV\_CONST\_RETVAL\_UNUSED\_HANDLER()。

这部分内容更多介绍参见“虚拟机生成器篇”，了解更多虚拟机设计原理。

## 四）执行

从前文zend\_execute\_scripts()函数的介绍中可以看到，执行过程是就是执行前面创建好的操作码表，执行过程从zend\_execute(op\_array, retval)语句开始。

zend\_execute()函数的调用路径如下：

|  |
| --- |
| zend\_execute()  ->zend\_vm\_stack\_push\_call\_frame() // 创建执行数据  ->zend\_vm\_calc\_used\_stack() // 计算需要的内存大小  ->zend\_vm\_stack\_push\_call\_frame\_ex()  ->zend\_vm\_init\_call\_frame() |

zend\_execute()函数的代码注释如下：

|  |
| --- |
| ZEND\_API void zend\_execute(zend\_op\_array \*op\_array, zval \*return\_value) {  zend\_execute\_data \*execute\_data; //\_zend\_execute\_data结构体在 zend\_compile.h 中定义  void \*object\_or\_called\_scope;  uint32\_t call\_info; // 调用模式  if (EG(exception) != NULL) { return; }  // 步骤1：确定作用域  // 获取 $this 作用域  object\_or\_called\_scope = zend\_get\_this\_object(EG(current\_execute\_data));  if (EXPECTED(!object\_or\_called\_scope)) { // 如果没有 $this 作用域  // 获取作用域  object\_or\_called\_scope = zend\_get\_called\_scope(EG(current\_execute\_data));  // 顶层代码 | 有符号表  call\_info = ZEND\_CALL\_TOP\_CODE | ZEND\_CALL\_HAS\_SYMBOL\_TABLE;  } else { // 有 $this 作用域  // 顶层代码 | 有符号表 | 有 $this 作用域  call\_info = ZEND\_CALL\_TOP\_CODE | ZEND\_CALL\_HAS\_SYMBOL\_TABLE | ZEND\_CALL\_HAS\_THIS;  }    // 步骤2：获取执行数据  execute\_data = zend\_vm\_stack\_push\_call\_frame(call\_info,  (zend\_function\*)op\_array, 0, object\_or\_called\_scope);  // 步骤3：为执行数据绑定符号表  if (EG(current\_execute\_data)) { // 情况3.1：如果全局变量中有执行数据  execute\_data->symbol\_table = zend\_rebuild\_symbol\_table(); // 重新创建符号表  } else { // 情况3.2：全局变量没有执行数据  execute\_data->symbol\_table = &EG(symbol\_table); // 使用已有符号表  }  EX(prev\_execute\_data) = EG(current\_execute\_data); // 当前执行数据变成前一个  // 步骤4：创建运行时缓存，并处理符号表中的编译变量  i\_init\_code\_execute\_data(execute\_data, op\_array, return\_value);  // 步骤5：如果启用了观察者，调用zend\_observer\_fcall\_begin(execute\_data)  ZEND\_OBSERVER\_FCALL\_BEGIN(execute\_data);  // 步骤6：执行操作码。zend\_execute\_ex() = execute\_ex()  zend\_execute\_ex(execute\_data);  // 步骤7：释放执行数据  zend\_vm\_stack\_free\_call\_frame(execute\_data);  } |

### 1、确定作用域

zend\_get\_called\_scope()函数用于在**执行数据链**中获取作用域，代码注释如下：

|  |
| --- |
| ZEND\_API zend\_class\_entry \*zend\_get\_called\_scope(zend\_execute\_data \*ex) {  while (ex) { // 如果执行数据有效  if (Z\_TYPE(ex->This) == IS\_OBJECT) { // 如果 this 是对象  return Z\_OBJCE(ex->This); // 返回 zval中zend\_object 的所属类  } else if (Z\_CE(ex->This)) { // 如果this是类  return Z\_CE(ex->This); // 返回this类  } else if (ex->func) { // 如果 ex有所属函数  // 如果此函数不是内置函数（是用户定义） 或 此函数有所属域  if (ex->func->type != ZEND\_INTERNAL\_FUNCTION || ex->func->common.scope){  return NULL; // 返回null  }  }  ex = ex->prev\_execute\_data; // 转到前一个执行数据  }  return NULL; // 查找失败返回null  } |

执行数据（zend\_execute\_data）结构体定义如下：

|  |
| --- |
| struct \_zend\_execute\_data {  const zend\_op \*opline; // 操作码指针  // 此执行数据指针，当前正在调用的执行数据。它可以调用多个执行数据，这些执行数据都指向它，但它只指向正在调用的这个。  zend\_execute\_data \*call;  zval \*return\_value; // 返回值指针  zend\_function \*func; // 执行过的函数  zval This; // $this 对象 指针  zend\_execute\_data \*prev\_execute\_data; // 前一个执行数据实例指针  zend\_array \*symbol\_table; // 符号表指针  void \*\*run\_time\_cache; // 运行时缓存  zend\_array \*extra\_named\_params; // 扩展命名参数 （函数的命名参数）  };  typedef struct \_zend\_execute\_data zend\_execute\_data; |

如上所示，它有一个prev\_execute\_data元素，是指向前一个执行数据的指针。所以执行数据可以串成树形。如果从某一个节点一直向上查找，就会找到一条链，这是常用的查找方式，如果前方中查找作用域就是这样查找。

本例中，没有声名函数和类，也没有声名命名空间，所以返回的作用域是NULL。

### 2、获取执行数据

zend\_vm\_stack\_push\_call\_frame()函数代码注释如下：

|  |
| --- |
| static zend\_execute\_data \*zend\_vm\_stack\_push\_call\_frame(uint32\_t call\_info, zend\_function \*func, uint32\_t num\_args, void \*object\_or\_called\_scope) {  // 计算使用的变量大小。p1:参数数量，p2:函数  uint32\_t used\_stack = zend\_vm\_calc\_used\_stack(num\_args, func);  // 初始化并返回 堆栈最上一个执行数据，空间不够则扩展大小  // p1:使用大小，p2:调用信息，p3:函数，p4:参数数量，p5:对象或调用域  return zend\_vm\_stack\_push\_call\_frame\_ex(used\_stack, call\_info,  func, num\_args, object\_or\_called\_scope);  } |

zend\_vm\_calc\_used\_stack()函数用于计算需要的内存空间大小：

|  |
| --- |
| static uint32\_t zend\_vm\_calc\_used\_stack(uint32\_t num\_args, zend\_function \*func) {  // （ZEND\_CALL\_FRAME\_SLOT=2） + 参数数量 + 临时变量数量  uint32\_t used\_stack = ZEND\_CALL\_FRAME\_SLOT + num\_args + func->common.T;  if (EXPECTED(ZEND\_USER\_CODE(func->type))) { // 如果函数是用户定义的  // used\_stack + 编译变量数量 - 形参和实参里小的一个  used\_stack += func->op\_array.last\_var - MIN(func->op\_array.num\_args, num\_args);  }  return used\_stack \* sizeof(zval); // 使用变量数 \* 变量大小  } |

#### **a）创建执行数据**

zend\_vm\_stack\_push\_call\_frame\_ex()函数用于找到合适的堆栈页，划分执行数据，并为执行数据绑定本次调用的信息：

|  |
| --- |
| // ing3, 初始化并返回 堆栈最上一个执行数据，空间不够则扩展大小  // p1:使用大小，p2:调用信息，p3:函数，p4:参数数量，p5:对象或调用域  static zend\_execute\_data \*zend\_vm\_stack\_push\_call\_frame\_ex(uint32\_t used\_stack, uint32\_t call\_info, zend\_function \*func, uint32\_t num\_args, void \*object\_or\_called\_scope) {  // 取得虚拟机堆栈里最上一个执行数据的【空闲位置】  zend\_execute\_data \*call = (zend\_execute\_data\*)EG(vm\_stack\_top);  // 如果剩余空间不足  if (UNEXPECTED(used\_stack > (size\_t)(((char\*)EG(vm\_stack\_end)) - (char\*)call))) {  call = (zend\_execute\_data\*)zend\_vm\_stack\_extend(used\_stack); // 扩展堆栈空间  // 初始化执行数据，添加【调用分配内存】标记  zend\_vm\_init\_call\_frame(call, call\_info | ZEND\_CALL\_ALLOCATED, func, num\_args, object\_or\_called\_scope);  return call; // 返回最上一个执行数据  } else { // 剩余空间够大  EG(vm\_stack\_top) = (zval\*)((char\*)call + used\_stack); // 更新堆栈的【空闲位置】  // 初始化执行数据  zend\_vm\_init\_call\_frame(call, call\_info, func, num\_args, object\_or\_called\_scope);  return call; // 返回最上一个执行数据  }  } |

如上所示，一个堆栈页中可以存放多个执行数据（zend\_execute\_data），直到空间不够用时，创建新堆栈页。

#### **b）创建虚拟机堆栈页**

zend\_vm\_stack\_extend()函数用于创建新的虚拟机堆栈页：

|  |
| --- |
| // 创建新的虚拟机堆栈页，返回最上一个元素指针，p1:大小  ZEND\_API void\* zend\_vm\_stack\_extend(size\_t size) {  zend\_vm\_stack stack;  void \*ptr;  stack = EG(vm\_stack); // 全局变量里的堆栈指针  stack->top = EG(vm\_stack\_top); // 记录头原素  // 创建新的虚拟机堆栈页  EG(vm\_stack) = stack = zend\_vm\_stack\_new\_page(  // size < 剩余大小 ？EG(vm\_stack\_page\_size) : size 添加头数据后 对齐到 EG(vm\_stack\_page\_size)  EXPECTED(size < EG(vm\_stack\_page\_size) - (ZEND\_VM\_STACK\_HEADER\_SLOTS \* sizeof(zval))) ?  EG(vm\_stack\_page\_size) : ZEND\_VM\_STACK\_PAGE\_ALIGNED\_SIZE(size, EG(vm\_stack\_page\_size)),  stack);  ptr = stack->top; // 头元素  EG(vm\_stack\_top) = (void\*)(((char\*)ptr) + size); // 新的顶点位置  EG(vm\_stack\_end) = stack->end; // 尾元素  return ptr; // 返回头元素指针  } |

zend\_vm\_stack\_new\_page()函数用于为新的虚拟机堆栈页分配内存空间：

|  |
| --- |
| static zend\_vm\_stack zend\_vm\_stack\_new\_page(size\_t size, zend\_vm\_stack prev) {  zend\_vm\_stack page = (zend\_vm\_stack)emalloc(size); // 分配内存空间，创建新的堆栈页  page->top = ZEND\_VM\_STACK\_ELEMENTS(page); // 跳过头部8个zval，找到存数据的位置  page->end = (zval\*)((char\*)page + size); // 指向末尾，后面不再有空间  page->prev = prev; // 指向前一个堆栈页  return page; // 返回当前页指针  } |

ZEND\_VM\_STACK\_ELEMENTS()等相关宏程序用于计算需要的内存空间，定义如下：

|  |
| --- |
| // 64位系统中：ZEND\_MM\_ALIGNED\_SIZE((24+16-1)/16) = 8个zval。8\*16=128Bytes  #define ZEND\_VM\_STACK\_HEADER\_SLOTS \  ((ZEND\_MM\_ALIGNED\_SIZE(sizeof(struct \_zend\_vm\_stack)) +  ZEND\_MM\_ALIGNED\_SIZE(sizeof(zval)) - 1) / ZEND\_MM\_ALIGNED\_SIZE(sizeof(zval)))  // 跳过stack头部（64位系统中8个zval,128Bytes）,找到元素列表开头  #define ZEND\_VM\_STACK\_ELEMENTS(stack) \  (((zval\*)(stack)) + ZEND\_VM\_STACK\_HEADER\_SLOTS)  // size 添加头数据后 对齐到 page\_size  #define ZEND\_VM\_STACK\_PAGE\_ALIGNED\_SIZE(size, page\_size) \  (((size) + ZEND\_VM\_STACK\_HEADER\_SLOTS \* sizeof(zval) \  + ((page\_size) - 1)) & ~((page\_size) - 1)) |

堆栈页（\_zend\_vm\_stack）结构体定义如下：

|  |
| --- |
| struct \_zend\_vm\_stack { // 虚拟机堆栈，64位系统中占24Bytes，3个元素都是指针  zval \*top; // 头元素  zval \*end; // 尾元素  zend\_vm\_stack prev; // 指向前一个堆栈页的指针  };  // zend\_vm\_stack 是 \_zend\_vm\_stack指针  typedef struct \_zend\_vm\_stack \*zend\_vm\_stack; |

#### **c）初始化执行数据**

zend\_vm\_init\_call\_frame()函数用于初始化执行数据，为执行数据绑定本次调用的信息：

|  |
| --- |
| static void zend\_vm\_init\_call\_frame(zend\_execute\_data \*call, uint32\_t call\_info, zend\_function \*func, uint32\_t num\_args, void \*object\_or\_called\_scope) {  call->func = func; // 更新执行数据的所属函数  Z\_PTR(call->This) = object\_or\_called\_scope; // 更新执行数据的This 作用域  ZEND\_CALL\_INFO(call) = call\_info; // 更新调用信息  ZEND\_CALL\_NUM\_ARGS(call) = num\_args; // 更新参数数量  } |

### 3、为执行数据绑定符号表

如zend\_execute()函数代码中步骤3所示：如果全局变量中的当前执行数据指针EG(current\_execute\_data)有效，需要调用zend\_rebuild\_symbol\_table()方法刷新符号表，否则使用全局变量中已有的符号表&EG(symbol\_table)。

zend\_rebuild\_symbol\_table()方法代码注释如下：

|  |
| --- |
| ZEND\_API zend\_array \*zend\_rebuild\_symbol\_table(void) {  zend\_execute\_data \*ex;  zend\_array \*symbol\_table;  // 第一步：查找符号表  ex = EG(current\_execute\_data); // 当前执行数据  // 执行数据有效，并且 （没有函数 或 函数不是用户定义的）  while (ex && (!ex->func || !ZEND\_USER\_CODE(ex->func->common.type))) {  ex = ex->prev\_execute\_data; // 使用前一个执行数据  }  if (!ex) { // 情况1.1： 如果没有有效的执行数据  return NULL; // 查找失败，返回null  }  // 情况1.2：如果已经创建过符号表：Z\_TYPE\_INFO((execute\_data)->This))中有标记  if (ZEND\_CALL\_INFO(ex) & ZEND\_CALL\_HAS\_SYMBOL\_TABLE) {  return ex->symbol\_table; // 返回创建好的符号表  }    // 执行数据添加【已创建符号表】标记  ZEND\_ADD\_CALL\_FLAG(ex, ZEND\_CALL\_HAS\_SYMBOL\_TABLE);  // 情况1.3：如果符号表缓存指针不在开头位置，说明有旧的符号表可以用  if (EG(symtable\_cache\_ptr) > EG(symtable\_cache)) {  // EG(symtable\_cache\_ptr) 指针倒回一个位置，找到前一个符号表  symbol\_table = ex->symbol\_table = \*(--EG(symtable\_cache\_ptr));  if (!ex->func->op\_array.last\_var) { // 如果没有编译变量  return symbol\_table; // 直接返回符号表  }  // 调整符号表大小，为编译变量保留空间  zend\_hash\_extend(symbol\_table, ex->func->op\_array.last\_var, 0);  } else { // 情况1.4：如果指针在开头位置。按编译变量的大小创建新哈希表  symbol\_table = ex->symbol\_table = zend\_new\_array(ex->func->op\_array.last\_var);  if (!ex->func->op\_array.last\_var) { // 如果没有用到编译变量  return symbol\_table; // 直接返回空 哈希表  }  zend\_hash\_real\_init\_mixed(symbol\_table); // 重新初始化混合哈希表  }    // 第二步：把编译变量添加进符号表  if (EXPECTED(ex->func->op\_array.last\_var)) { // 如果有编译变量  zend\_string \*\*str = ex->func->op\_array.vars; // 编译变量名列表开头位置  zend\_string \*\*end = str + ex->func->op\_array.last\_var; // 编译变量名列表结束位置  zval \*var = ZEND\_CALL\_VAR\_NUM(ex, 0); // 临时变量开头  do { // 遍历所有编译变量，把它们添加进符号表, 变量名称做key  \_zend\_hash\_append\_ind(symbol\_table, \*str, var);  str++; // 下一个名称  var++; // 下一个编译变量  } while (str != end); // 直到变量名列表结束  }  return symbol\_table; // 返回符号表  } |

### 4、创建运行时缓存

i\_init\_code\_execute\_data()方法用于创建运行时缓存：

|  |
| --- |
| static void i\_init\_code\_execute\_data(zend\_execute\_data \*execute\_data, zend\_op\_array \*op\_array, zval \*return\_value) {  EX(opline) = op\_array->opcodes; // 执行数据 绑定操作码列表  EX(call) = NULL; // 清空当前调用  EX(return\_value) = return\_value; // 添加返回值  if (op\_array->last\_var) { // 如果操作码中有临时变量  // 从附加符号表里读取所有临时变量，并更新到临时变量列表中  zend\_attach\_symbol\_table(execute\_data);  }  if (!ZEND\_MAP\_PTR(op\_array->run\_time\_cache)) { // 如果没有运行时缓存  void \*ptr; // 临时指针  ptr = emalloc(op\_array->cache\_size); // 分配内存创建运行时缓存  // op\_array->run\_time\_cache\_\_ptr = ptr ,关联到运行时缓存指针  ZEND\_MAP\_PTR\_INIT(op\_array->run\_time\_cache, ptr);  memset(ptr, 0, op\_array->cache\_size); // 内存全部置为0  }  // 使用此运行时缓存 作为当前 运行时缓存, execute\_data->run\_time\_cache  EX(run\_time\_cache) = RUN\_TIME\_CACHE(op\_array);  // 此执行数据 作为 当前执行数据  EG(current\_execute\_data) = execute\_data;  } |

zend\_attach\_symbol\_table()方法，用于处理编译变量和符号表的关系：把符号表中的编译变量转成间接引用，指向执行数据中的编译变量：

|  |
| --- |
| ZEND\_API void zend\_attach\_symbol\_table(zend\_execute\_data \*execute\_data) /\* {{{ \*/  {  // 执行数据中的函数操作码  zend\_op\_array \*op\_array = &execute\_data->func->op\_array;  HashTable \*ht = execute\_data->symbol\_table; // 执行数据中的符号表  if (EXPECTED(op\_array->last\_var)) { // 如果有编译变量  zend\_string \*\*str = op\_array->vars; // 编译变量名 指针列表开头  zend\_string \*\*end = str + op\_array->last\_var; // 编译变量名 指针列表结尾  zval \*var = EX\_VAR\_NUM(0); // 执行数据中的编译变量列表开头  do { // 遍历每个变量名，在符号表中查找这个变量  zval \*zv = zend\_hash\_find\_known\_hash(ht, \*str);  if (zv) { // 如果找到  if (Z\_TYPE\_P(zv) == IS\_INDIRECT) { // 如果 是间接引用  zval \*val = Z\_INDIRECT\_P(zv); // 追踪引用目标  ZVAL\_COPY\_VALUE(var, val); // 值复制到【执行数据中的编译变量】里  } else { // 不是间接引用  ZVAL\_COPY\_VALUE(var, zv); // 值复制到【执行数据中的编译变量】里  }  } else { // 如果没找到  ZVAL\_UNDEF(var); // 变量值为 未定义  zv = zend\_hash\_add\_new(ht, \*str, var); // 添加 这个名称的临时变量  }  ZVAL\_INDIRECT(zv, var); // 符号表中的值转成间接引用,指向【执行数据中的编译变量】  str++; // 下一个变量名  var++; // 下一个变量  } while (str != end);  }  } |

关于复制和间接引用的介绍参见“类型篇”。

观察者模式不是本章重点，在后续章节中介绍。

### 5、执行ZEND\_ASSIGN操作码

execute\_ex()方法已在[前文](#_四）执行操作码)中介绍过。它会依次调用每个操作码的zend\_op->handler()方法，执行每个操作码的处理器。在本例中，如前文（“编译的后续处理”章节）所示，获取到的处理器为ZEND\_ASSIGN\_SPEC\_CV\_CONST\_RETVAL\_UNUSED\_HANDLER()函数，定义如下：

|  |
| --- |
| static int ZEND\_FASTCALL ZEND\_ASSIGN\_SPEC\_CV\_CONST\_RETVAL\_UNUSED\_HANDLER (ZEND\_OPCODE\_HANDLER\_ARGS) {  USE\_OPLINE // 默认情况下: const zend\_op \*opline = EX(opline);  zval \*value;  zval \*variable\_ptr;  SAVE\_OPLINE(); // 默认情况下为空  value = RT\_CONSTANT(opline, opline->op2);  variable\_ptr = EX\_VAR(opline->op1.var);  value = zend\_assign\_to\_variable(variable\_ptr, value, IS\_CONST, EX\_USES\_STRICT\_TYPES());  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 处理下一个操作码  } |

RT\_CONSTANT()宏程序用于定位操作对象中的常量，在前文“编译的后续处理”中已经介绍过，常量的偏移量是使用ZEND\_PASS\_TWO\_UPDATE\_CONSTANT()宏程序计算的，相对于操作码的偏移量，所以定位常量只要从操作码的位置进行偏移即可：

|  |
| --- |
| # define RT\_CONSTANT(opline, node) ((zval\*)(((char\*)(opline)) + (int32\_t)(node).constant)) |

EX\_VAR()宏程序用于通过偏移量定位 execute\_data 中的 zval实例，逻辑简单，定义如下：

|  |
| --- |
| #define EX\_VAR(n) ZEND\_CALL\_VAR(execute\_data, n)  #define ZEND\_CALL\_VAR(call, n) ((zval\*)(((char\*)(call)) + ((int)(n)))) |

在本例中，zend\_assign\_to\_variable()函数会调用zend\_copy\_to\_variable()函数进行赋值处理，这两个函数的大部分业务逻辑在本例中没有用到，所以放在后续章节中。本例中用到的业务逻辑非常简单，仅是调用：ZVAL\_COPY\_VALUE()宏程序来复制zval实例中的元素：

|  |
| --- |
| static void zend\_copy\_to\_variable(zval \*variable\_ptr, zval \*value, zend\_uchar value\_type) {  ...  ZVAL\_COPY\_VALUE(variable\_ptr, value); // 把value中的元素复制给variable\_ptr  ...  } |

ZVAL\_COPY\_VALUE()宏程序业务逻辑简单，但与底层zval类型的设计紧密相关，详情参见“类型篇”。

### 6、释放执行数据

执行完操作码已后，要释放执行数据，这步操作的调用路径很简单：

|  |
| --- |
| zend\_vm\_stack\_free\_call\_frame()  ->zend\_vm\_stack\_free\_call\_frame\_ex() |

zend\_vm\_stack\_free\_call\_frame()方法是过渡方法：

|  |
| --- |
| static void zend\_vm\_stack\_free\_call\_frame(zend\_execute\_data \*call) {  zend\_vm\_stack\_free\_call\_frame\_ex(ZEND\_CALL\_INFO(call), call);  } |

zend\_vm\_stack\_free\_call\_frame\_ex()方法用于释放执行数据：

|  |
| --- |
| static void zend\_vm\_stack\_free\_call\_frame\_ex(uint32\_t call\_info, zend\_execute\_data \*call) {  // 如果执行数据在新分配的堆栈页中，释放此堆栈页  if (UNEXPECTED(call\_info & ZEND\_CALL\_ALLOCATED)) {  zend\_vm\_stack p = EG(vm\_stack); // 虚拟机堆栈  zend\_vm\_stack prev = p->prev; // 前一个堆栈页  EG(vm\_stack\_top) = prev->top; // 前一个页的最上元素  EG(vm\_stack\_end) = prev->end; // 前一个页的最后元素  EG(vm\_stack) = prev; // 使用前一个堆栈页  efree(p); // 释放当前堆栈页  } else { // 不在新堆栈页中  EG(vm\_stack\_top) = (zval\*)call; // 空闲位置指针复位到此执行数据开头（相当于释放内存）  }  } |

至此，本例的主要操作已进行完毕。本例非常简单，在运行过程中避开了很多复杂的业务逻辑。本章重点在介绍运行环境和基本概念，希望可以通过简单的示例，将运行过程过程介绍清楚。在后续章节中，介绍的php功能会越来越复杂，这样可以尽量避免把复杂的php功能和复杂的运行环境参杂在一起介绍，减少认知负荷，降低学习难度。

PHP语言作为全世界广泛使用的优秀开源项目，在设计和编写上都把复杂的难题打散成许多不太复杂的问题，让普通程序员可以学习和参与研发。在整个项目中，有上百个这样不太复杂的问题，学习它们可以逐渐掌握PHP语言的设计思想，提升自己的编程水平。

# 三、基本数学运算

基本数学运算包括：加法、减法、乘法、除法、取余数和宇幂运算。这些表达式在解析时都会被创建成二元运算语句（ZEND\_AST\_BINARY\_OP），再进行编译和执行。

## 一）加法运算

以最简单的加法运算为例：

|  |
| --- |
| <?php  echo 1+2; |

### 1）解析源码，创建ZEND\_AST\_BINARY\_OP语句

语法解析过程如下：

|  |
| --- |
| zend\_ast \* ast\_1 = zend\_ast\_create\_list(0, ZEND\_AST\_STMT\_LIST); /\* top\_statement\_list:[] \*/  zend\_ast \* ast\_36 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("1", 0)); /\* scalar:["1"] \*/  zend\_ast \* ast\_14 = ast\_36; /\* expr:["ast\_36"] \*/  zend\_ast \* ast\_67 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("2", 0)); /\* scalar:["2"] \*/  zend\_ast \* ast\_45 = ast\_67; /\* expr:["ast\_67"] \*/  zend\_ast \* ast\_78 = zend\_ast\_create\_binary\_op(ZEND\_ADD, ast\_14, ast\_45); /\* expr:["ast\_14","+","ast\_45"] \*/  zend\_ast \* ast\_13 = zend\_ast\_create(ZEND\_AST\_ECHO, ast\_78); /\* echo\_expr:["ast\_78"] \*/  ...  CG(ast) = ast\_123; /\* start:["ast\_123"] \*/ |

如上所示，在解析过程中，会把相加的两个数字匹配成两个标量（scalar），创建两个语句实例来存放它们，再调用zend\_ast\_create\_binary\_op()函数，创建一个加法语句，把前面两个语句作为它的子语句。

默认情况下，zend\_ast\_create\_binary\_op()函数的调用路径有些绕：

|  |
| --- |
| zend\_ast\_create\_binary\_op()  ->zend\_ast\_create\_ex() 宏程序  ->ZEND\_AST\_SPEC\_CALL\_EX()宏程序  ->zend\_ast\_create\_ex\_(0-5)()  ->zend\_ast\_create\_(0-5)() |

zend\_ast\_create\_binary\_op()函数本身很简单：

|  |
| --- |
| static zend\_always\_inline zend\_ast \*zend\_ast\_create\_binary\_op(uint32\_t opcode, zend\_ast \*op0, zend\_ast \*op1) {  // 使用传入的操作码和参数，创建一个二元操作语句（类型为 ZEND\_AST\_BINARY\_OP）  return zend\_ast\_create\_ex(ZEND\_AST\_BINARY\_OP, opcode, op0, op1);  } |

关键在ZEND\_AST\_SPEC\_CALL\_EX()宏程序，它会根据参数数量调zend\_ast\_create\_ex\_0 - zend\_ast\_create\_ex\_5这6个方法中的一个，用它和相关方法的相关定义如下:

|  |
| --- |
| // 创建语句实例  # define zend\_ast\_create\_ex(...) ZEND\_AST\_SPEC\_CALL\_EX(zend\_ast\_create\_ex, \_\_VA\_ARGS\_\_)  // 根据传入的参数数量，自动适配调用的方法名，并调用方法  # define ZEND\_AST\_SPEC\_CALL\_EX(name, ...) \  ZEND\_EXPAND\_VA(ZEND\_AST\_SPEC\_CALL\_EX\_(name, \_\_VA\_ARGS\_\_, \_5, \_4, \_3, \_2, \_1, \_0)(\_\_VA\_ARGS\_\_))  // 在调用方法名后面拼接数字  # define ZEND\_AST\_SPEC\_CALL\_EX\_(name, \_, \_6, \_5, \_4, \_3, \_2, \_1, suffix, ...) name ## suffix  # define ZEND\_EXPAND\_VA(code) code // 64位操作系统中的定义，直接应用传入的代码 |

在上述zend\_ast\_create\_binary\_op(ZEND\_ADD, ast\_14, ast\_45)语句的调用中，传入了操作码和2个子语句的指针，它会调用zend\_ast\_create\_ex\_2()函数创建并更新语句实例。相关代码如下：

|  |
| --- |
| ZEND\_API zend\_ast \* ZEND\_FASTCALL zend\_ast\_create\_2(zend\_ast\_kind kind, zend\_ast \*child1, zend\_ast \*child2) {  ...  ast = zend\_ast\_alloc(zend\_ast\_size(2)); // 分配内存创建包含2个子节点的语句实例  ast->kind = kind; // 类型  ast->attr = 0;  ast->child[0] = child1;  ast->child[1] = child2;  ...  } |

zend\_ast\_size()函数用于计算语句实例占用内存大小：

|  |
| --- |
| static zend\_always\_inline size\_t zend\_ast\_size(uint32\_t children) {  // 内存大小 = zend\_ast的大小 + （子语句数-1）\* 语句指针大小  return sizeof(zend\_ast) - sizeof(zend\_ast \*) + sizeof(zend\_ast \*) \* children;  } |

实例中的指针数量要比子语句数少1个，是因为实例本身自带一个语句指针，实际上最终分配的指针数和子语句数相等。语句结构定义如下：

|  |
| --- |
| struct \_zend\_ast {  zend\_ast\_kind kind; // 语句类型,16位整数  zend\_ast\_attr attr; // 附加属性,16位整数  uint32\_t lineno; // 行号，32位整数  // 前面一共8个字节, 64位  zend\_ast \*child[1]; // 子语句指针，已经有1个  }; |

zend\_ast\_alloc()函数非常简单，它使用arena分配内存，而不是heap(zend\_alloc)：

|  |
| --- |
| static inline void \*zend\_ast\_alloc(size\_t size) {  return zend\_arena\_alloc(&CG(ast\_arena), size);  } |

解析的过程比较简单，需要重点关注的是公式（expr）规则，它在一个简单的加法语句中，一共命中过3次：第一次是数字1，第二次是数字2，第三次是1+2。每次命中都会创建一个语句实例。

在zend\_language\_parser.y文件中定义了所有的PHP语法规则。其中expr是最为常用的语法规则 ，也是比较复杂的一个，它有80多条子规则，涵盖了算术运算、二进行运算、赋值、连接、抛错、yield等各种语法，加法运算是它的子规则之一：

|  |
| --- |
| expr:  ...  | expr '.' expr { $$ = zend\_ast\_create\_concat\_op($1, $3); }  | expr '+' expr { $$ = zend\_ast\_create\_binary\_op(ZEND\_ADD, $1, $3); }  | expr '-' expr { $$ = zend\_ast\_create\_binary\_op(ZEND\_SUB, $1, $3); }  ... |

如上所示，expr规则中使用了大量的递归，无论多复杂冗长的算式都由它来递归匹配，并创建成语句树。

### 2）编译ZEND\_AST\_BINARY\_OP语句

echo语句的编译在前文（Hello World章节）中已经介绍过。编译公式（expr）需要用到zend\_compile\_expr()函数。以加法操作为例，它的调用路径如下：

|  |
| --- |
| zend\_compile\_expr()  ->zend\_compile\_expr\_inner()  ->zend\_compile\_binary\_op() |

zend\_compile\_expr()函数本身只是一个调用入口，它负责调用zend\_compile\_expr\_inner()函数。zend\_compile\_expr\_inner()函数根据语句类型来调用各个类型的处理逻辑。如前文所述，语句是通过zend\_ast\_create\_binary\_op()函数创建的，类型为ZEND\_AST\_BINARY\_OP，处理逻辑如下：

|  |
| --- |
| static void zend\_compile\_expr\_inner(znode \*result, zend\_ast \*ast) {  ...  switch (ast->kind) { // 根据类型处理  ...  case ZEND\_AST\_BINARY\_OP: zend\_compile\_binary\_op(result, ast); return; // 二元操作  ... |

zend\_compile\_binary\_op()分成几种情况进行处理，在本次加法计算用，两个数都是常量，会先尝试调用zend\_try\_ct\_eval\_binary\_op()函数直接处理，处理不成功时才创建操作码：

|  |
| --- |
| static void zend\_compile\_binary\_op(znode \*result, zend\_ast \*ast){  ...  // 如果两个运算对象都是常量  if (left\_node.op\_type == IS\_CONST && right\_node.op\_type == IS\_CONST) {  // 尝试直接计算，如果成功  if (zend\_try\_ct\_eval\_binary\_op(&result->u.constant, opcode,  &left\_node.u.constant, &right\_node.u.constant)  ) {  result->op\_type = IS\_CONST; // 结果类型也是常量  zval\_ptr\_dtor(&left\_node.u.constant); // 销毁两个运算对象中的常量  zval\_ptr\_dtor(&right\_node.u.constant);  return; // 计算完毕  }  }  do { // 直接计算失败会走到这里  ...  // 创建操作码  zend\_emit\_op\_tmp(result, opcode, &left\_node, &right\_node);  } while (0);  } |

zend\_try\_ct\_eval\_binary\_op()函数会根据操作码，调用相应的方法来进行运算：

|  |
| --- |
| static inline bool zend\_try\_ct\_eval\_binary\_op(zval \*result, uint32\_t opcode, zval \*op1, zval \*op2){  ...  binary\_op\_type fn = get\_binary\_op(opcode); // 取得运算方法  fn(result, op1, op2); // 执行运算  return 1; // 返回成功  } |

#### **在编译中执行**

zend\_try\_ct\_eval\_binary\_op()函数体现了脚本语言的一个显著的特征（或特色）：**一边编译一边执行**（又叫做**“解释执行”**）。php语言不需要像C语言一样先把所有程序编译成机器语言再来执行，它会在编译过程中处理一些简单的计算操作，并把计算结果打包到操作码里一起提交给虚拟机来执行。这样做的好处是：如前文提及，操作码是可以缓存的，这些在编译中计算的结果也会被缓存到操作中。当多次执行同一个脚本程序时（尤其是在web程序中），php可以使用缓存操作码，避免进行多次编译，这些在编译时进行过的计算也不需要再重复进行，可以减少工作量，提升性能。

### 3）编译过程中的加法运算

get\_binary\_op()函数在zend\_opcode.c中定义，它的逻辑非常简单：根据操作码返回对应的处理方法，函数定义如下：

|  |
| --- |
| // 返回 二元操作码 对应的函数  ZEND\_API binary\_op\_type get\_binary\_op(int opcode) {  switch (opcode) {  case ZEND\_ADD: return (binary\_op\_type) add\_function; // 加法  case ZEND\_SUB: return (binary\_op\_type) sub\_function; // 减法  case ZEND\_MUL: return (binary\_op\_type) mul\_function; // 乘法  case ZEND\_POW: return (binary\_op\_type) pow\_function; // 乘方  case ZEND\_DIV: return (binary\_op\_type) div\_function; // 除法  case ZEND\_MOD: return (binary\_op\_type) mod\_function; // 取余  case ZEND\_SL: return (binary\_op\_type) shift\_left\_function; // 左移  case ZEND\_SR: return (binary\_op\_type) shift\_right\_function; // 右移    case ZEND\_FAST\_CONCAT: // 连接，两边都是内置变量时 ZEND\_FAST\_CONCAT  case ZEND\_CONCAT: return (binary\_op\_type) concat\_function;    case ZEND\_IS\_IDENTICAL: // === 或 match 中的 arm 严格匹配  case ZEND\_CASE\_STRICT: return (binary\_op\_type) is\_identical\_function;    case ZEND\_IS\_NOT\_IDENTICAL: // !==  return (binary\_op\_type) is\_not\_identical\_function;    case ZEND\_IS\_EQUAL: // == ， 或 match 中的 arm  case ZEND\_CASE: return (binary\_op\_type) is\_equal\_function; // switch语句中的case    case ZEND\_IS\_NOT\_EQUAL: return (binary\_op\_type) is\_not\_equal\_function; // !=    case ZEND\_IS\_SMALLER: return (binary\_op\_type) is\_smaller\_function; // <    case ZEND\_IS\_SMALLER\_OR\_EQUAL: // <=  return (binary\_op\_type) is\_smaller\_or\_equal\_function;  case ZEND\_SPACESHIP: return (binary\_op\_type) compare\_function; // <=>    case ZEND\_BW\_OR: return (binary\_op\_type) bitwise\_or\_function; // 位运算符 |  case ZEND\_BW\_AND: return (binary\_op\_type) bitwise\_and\_function; // 位运算符 &  case ZEND\_BW\_XOR: return (binary\_op\_type) bitwise\_xor\_function; // 位运算符 ^    case ZEND\_BOOL\_XOR: // 逻辑运算符 xor  return (binary\_op\_type) boolean\_xor\_function;  default:  ZEND\_UNREACHABLE();  return (binary\_op\_type) NULL;  }  } |

当前用到加法操作码ZEND\_ADD，返回的处理方法为add\_function。

加法计算的业务逻辑并不算复杂，但需要考虑到多种情况，代码量也不少。

add\_function()函数在zend\_operators.c中定义，它调用add\_function\_fast()函数和add\_function\_slow()函数实现加法操作：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL add\_function(zval \*result, zval \*op1, zval \*op2){  if (add\_function\_fast(result, op1, op2) == SUCCESS) { // 先尝试快速加法  return SUCCESS;  } else { // 失败时尝试慢加法  return add\_function\_slow(result, op1, op2);  }  } |

#### **数字相加**

add\_function\_fast()函数定义如下：

|  |
| --- |
| static zend\_always\_inline zend\_result add\_function\_fast(zval \*result, zval \*op1, zval \*op2){  zend\_uchar type\_pair = TYPE\_PAIR(Z\_TYPE\_P(op1), Z\_TYPE\_P(op2)); // 组合类型  if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_LONG, IS\_LONG))) { // 两个长整型  fast\_long\_add\_function(result, op1, op2); return SUCCESS;  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_DOUBLE, IS\_DOUBLE))) { // 两个双精度  ZVAL\_DOUBLE(result, Z\_DVAL\_P(op1) + Z\_DVAL\_P(op2)); return SUCCESS;  // 一个长整形和一个双精度,转成双精度进行计算  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_LONG, IS\_DOUBLE))) {  ZVAL\_DOUBLE(result, ((double)Z\_LVAL\_P(op1)) + Z\_DVAL\_P(op2)); return SUCCESS;  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_DOUBLE, IS\_LONG))) { // 同上  ZVAL\_DOUBLE(result, Z\_DVAL\_P(op1) + ((double)Z\_LVAL\_P(op2))); return SUCCESS;  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_ARRAY, IS\_ARRAY))) { // 两个数组  add\_function\_array(result, op1, op2); return SUCCESS;  } else { // 其他类型全部 失败  return FAILURE;  }  } |

其中，整数和数组的加法需要特殊处理，分别调用fast\_long\_add\_function()函数和add\_function\_array()函数。

fast\_long\_add\_function()函数用于两个长整型相加，它在不同的平台上有不同的实现逻辑，在64位windows平台上业务逻辑如下：

|  |
| --- |
| static zend\_always\_inline void fast\_long\_add\_function(zval \*result, zval \*op1, zval \*op2){  // 如果整数相加会溢出，转成小数相加  if (UNEXPECTED((Z\_LVAL\_P(op1) & LONG\_SIGN\_MASK) == (Z\_LVAL\_P(op2) & LONG\_SIGN\_MASK)  && (Z\_LVAL\_P(op1) & LONG\_SIGN\_MASK) != ((Z\_LVAL\_P(op1) + Z\_LVAL\_P(op2)) & LONG\_SIGN\_MASK))) {  ZVAL\_DOUBLE(result, (double) Z\_LVAL\_P(op1) + (double) Z\_LVAL\_P(op2));  } else { // 不会溢出，直接整数相加  ZVAL\_LONG(result, Z\_LVAL\_P(op1) + Z\_LVAL\_P(op2));  }  } |

#### **数组相加**

add\_function\_array()函数调用zend\_hash\_merge()函数合并两个数组，需要注意的是，它先给原数组创建副本，然后在副本上所作，不会修改原数组：

|  |
| --- |
| static zend\_never\_inline void ZEND\_FASTCALL add\_function\_array(zval \*result, zval \*op1, zval \*op2){  // 如果结果和op1指针相同，并且op1,op2 指向同一个数组  if (result == op1 && Z\_ARR\_P(op1) == Z\_ARR\_P(op2)) {  return; // 操作无效，直接返回  }  if (result != op1) { // 结果不返回给op1， ($c = $a + $b;)  // 把op1 复制一份，关联到返回结果  ZVAL\_ARR(result, zend\_array\_dup(Z\_ARR\_P(op1)));  } else { // 结果返回给op1， ($a = $a + $b;)  SEPARATE\_ARRAY(result); // 给result 创建副本  }  // merge碰到相同索引号会跳过，例如：[1]+[2,3] => [1,3]  zend\_hash\_merge(Z\_ARRVAL\_P(result), Z\_ARRVAL\_P(op2), zval\_add\_ref, 0);  } |

数组的相关操作请参看哈希表相关介绍。

#### **其他类型相加**

当快加法add\_function\_fast()函数遇到无法处理的类型，返回失败时，需要调用慢加法add\_function\_slow()函数：

|  |
| --- |
| static zend\_never\_inline zend\_result ZEND\_FASTCALL add\_function\_slow(zval \*result, zval \*op1, zval \*op2) {  ZVAL\_DEREF(op1); // 如果是引用类型，先追踪到引用目标  ZVAL\_DEREF(op2);  if (add\_function\_fast(result, op1, op2) == SUCCESS) { // 步骤1，先尝试快加  return SUCCESS;  }  // 步骤2，尝试调用两个运算对象的 do\_operation() 方法进行计算，如果成功直接返回  ZEND\_TRY\_BINARY\_OBJECT\_OPERATION(ZEND\_ADD);  // 步骤3，把两个运算对象转成数字，放在副本变量里  zval op1\_copy, op2\_copy;  if (UNEXPECTED(zendi\_try\_convert\_scalar\_to\_number(op1, &op1\_copy) == FAILURE)  || UNEXPECTED(zendi\_try\_convert\_scalar\_to\_number(op2, &op2\_copy) == FAILURE)) {  zend\_binop\_error("+", op1, op2); // 报错,不支持此操作符  if (result != op1) { // 如果不用第一个运算对象接收结果  ZVAL\_UNDEF(result); // 清空接收变量，中断操作  }  return FAILURE; // 返回失败  }  if (result == op1) { // 如果转成数字成功，如果用第一个运算对象接收结果  zval\_ptr\_dtor(result); // 清空接收变量  }  // 步骤4，使用标量转成的数字，再次尝试快加  if (add\_function\_fast(result, &op1\_copy, &op2\_copy) == SUCCESS) {  return SUCCESS;  }  ZEND\_ASSERT(0 && "Operation must succeed"); // 断言，上一步一定成功  return FAILURE; // 返回失败  } /\* }}} \*/ |

ZEND\_TRY\_BINARY\_OBJECT\_OPERATION()宏程序是算术运算和连接操作的辅助功能，逻辑比较简单，相关代码如下：

|  |
| --- |
| // 尝试调用op1 和 op2的 do\_operation 方法，如果成功直接返回  #define ZEND\_TRY\_BINARY\_OBJECT\_OPERATION(opcode) \  ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION(opcode) \  else \  ZEND\_TRY\_BINARY\_OP2\_OBJECT\_OPERATION(opcode)  #define ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION(opcode) \  /\* 如果op1类型是对象 并且 有 do\_operation()函数 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(op1) == IS\_OBJECT) \  && UNEXPECTED(Z\_OBJ\_HANDLER\_P(op1, do\_operation))) { \  if (EXPECTED(SUCCESS == Z\_OBJ\_HANDLER\_P(op1, do\_operation)(opcode, result, op1, op2))) { /\* 调用do\_operation()函数，如果成功 \*/ \  return SUCCESS; /\* 返回成功 \*/ \  } \  }  // ZEND\_TRY\_BINARY\_OP2\_OBJECT\_OPERATION()宏程序的定义与以上类似 |

zendi\_try\_convert\_scalar\_to\_number()函数调用 \_zendi\_try\_convert\_scalar\_to\_number()函数把各种不同类型转成数字，它能体现php语言设计中，数字类型和非数字类型的关系：

|  |
| --- |
| static zend\_never\_inline zend\_result ZEND\_FASTCALL \_zendi\_try\_convert\_scalar\_to\_number (zval \*op, zval \*holder){  switch (Z\_TYPE\_P(op)) { // 按运算对象的类型进行操作  case IS\_NULL: // null 和 false 转成 整数 0  case IS\_FALSE:  ZVAL\_LONG(holder, 0);  return SUCCESS;  case IS\_TRUE: // true 转成整数 1  ZVAL\_LONG(holder, 1);  return SUCCESS;  case IS\_STRING: // 字串  {  bool trailing\_data = false; // 默认没有多余的数据  // op转成数字，放在holder里返回，并更新holder类型  if (0 == (Z\_TYPE\_INFO\_P(holder) = is\_numeric\_string\_ex(Z\_STRVAL\_P(op), Z\_STRLEN\_P(op),  &Z\_LVAL\_P(holder), &Z\_DVAL\_P(holder), /\* allow errors \*/ true, NULL, &trailing\_data))) {  // 如果失败，会导致异常：类型无效  return FAILURE;  }  if (UNEXPECTED(trailing\_data)) { // 如果有多余的数据，抛出警告信息，遇到非数字  zend\_error(E\_WARNING, "A non-numeric value encountered");  if (UNEXPECTED(EG(exception))) { // 如果有异常  return FAILURE; // 返回失败  }  }  return SUCCESS; // 返回成功  }  case IS\_OBJECT: // 对象转成数字，要调用对象自己的 cast\_object()函数  if (Z\_OBJ\_HT\_P(op)->cast\_object(Z\_OBJ\_P(op), holder, \_IS\_NUMBER) == FAILURE  || EG(exception)) {  return FAILURE; // 如果转换失败或有异常，返回失败  }  return SUCCESS; // 返回成功  case IS\_RESOURCE: // 资源或数组  case IS\_ARRAY:  return FAILURE; // 返回失败  EMPTY\_SWITCH\_DEFAULT\_CASE()  }  } |

可以看出，类型之间的转换是经过精心细致地设计的，正是因为有这了这些精心设计，php语言才能忽略类型进行编程。PHP的这一特性使它对初学者和非计算机专业的开发者来说非常友好，在PHP4和PHP5的时代，这样能忽略类型进行编程的开发语言并不多，这使PHP语言非常流行。

在各个类型向数字的转换中，字串（string）转数字最为复杂，也最为常用，它通过上文中调用到的is\_numeric\_string\_ex()函数来实现，它的代码量比较大，请在附录中查看。

### 4）执行

在以上加法运算示例中，由于相加的两个数字都是常量，运算操作在编译时就已经完成了，不需要创建加法操作码，所以也不需要执行加法操作码，只需要像“Hello World”示例中一样执行ZEND\_ECHO操作码。

## 二）加法操作码（ZEND\_ADD）

在上一节的示例中，因为表达式用没有变量，所以没有创建加法操作码。再看一个示例：

|  |
| --- |
| <?php  $a=1;  echo $a+1; // 加法表达式 |

如上所示，由于在加法表达式中有常量，在编译过程中调用zend\_try\_ct\_eval\_binary\_op()函数尝试运算时，会调用zend\_emit\_op\_tmp()函数创建加法操作码ZEND\_ADD。

所有操作码的处理器先在Zend/zend\_vm\_def.h文件中定义，它和zend\_vm\_execute.skl文件一起，通过zend\_vm\_gen.php脚本生成c语言源码，存放在zend\_vm\_execute.h文件中，所以最终使用的源码文件是zend\_vm\_execute.h。其转换过程比较复杂，可参见“虚拟机生成器篇”，本篇重点介绍操作码的处理逻辑。

### 加法操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_ADD操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HOT\_NOCONSTCONST\_HANDLER(1, ZEND\_ADD, CONST|TMPVARCV, CONST|TMPVARCV) {  // 业务逻辑略 ...  // 调用方法 zend\_add\_helper  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_add\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，此伪函数会根据两个操作对象的变量类型，被转换成4个处理器函数：

1）ZEND\_ADD\_SPEC\_CONST\_CONST\_HANDLER()函数，处理两个常量相加，例如**“1+1”**，但这种情况有可能已经在编译时处理完了，如上一节中示例；

2）ZEND\_ADD\_SPEC\_CONST\_TMPVARCV\_HANDLER()函数，处理一个常量和一个变量相加，如**“1+$a”**；

3）ZEND\_ADD\_SPEC\_TMPVARCV\_CONST\_HANDLER()函数，处理一个变量和一个常量相加，如**“$a+1”**；

4）ZEND\_ADD\_SPEC\_TMPVARCV\_TMPVARCV\_HANDLER()函数，处理两个变量相加,如**“$a+$b”**。

具体转换过程参见“虚拟机生成器篇”。

以上4个函数的业务逻辑基本相同，在本示例中“echo $a+1；”这个语句中，第一个操作对象$a是变量，第二个1是常量，所以会调用ZEND\_ADD\_SPEC\_TMPVARCV\_CONST\_HANDLER()这个处理器函数，函数代码如下：

|  |
| --- |
| static ZEND\_VM\_HOT ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_ADD\_SPEC\_TMPVARCV\_CONST\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS) {  USE\_OPLINE  zval \*op1, \*op2, \*result;  double d1, d2;  op1 = EX\_VAR(opline->op1.var); // 操作对象1，变量  op2 = RT\_CONSTANT(opline, opline->op2); // 操作对象2，常量  if (1 && (IS\_TMP\_VAR|IS\_VAR|IS\_CV) == IS\_CONST && IS\_CONST == IS\_CONST) {  /\* pass \*/  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op1) == IS\_LONG)) { // 如果op1是整数  if (EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_LONG)) { // op2也是整数  result = EX\_VAR(opline->result.var); // 运算结果变量  fast\_long\_add\_function(result, op1, op2); // 快速加法，并把结果保存到变量中  ZEND\_VM\_NEXT\_OPCODE(); // 跳转到下一个操作码  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_DOUBLE)) { // 如果op2是小数  d1 = (double)Z\_LVAL\_P(op1); // op1转成小数  d2 = Z\_DVAL\_P(op2); // 取得op2中的小数  goto add\_double; // 两个小数相加  }  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op1) == IS\_DOUBLE)) { // 如果op1是小数  if (EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_DOUBLE)) { // 如果o2是小数  d1 = Z\_DVAL\_P(op1); // 取得op1小数的值  d2 = Z\_DVAL\_P(op2); // 取得op2小数的值  add\_double:  result = EX\_VAR(opline->result.var); // 运算结果变量  ZVAL\_DOUBLE(result, d1 + d2); // 直接相加，结果保存到变量中  ZEND\_VM\_NEXT\_OPCODE(); // 跳转到下一个操作码  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_LONG)) { // 如果op2是整数  d1 = Z\_DVAL\_P(op1); // 取得op1小数的值  d2 = (double)Z\_LVAL\_P(op2); // op2转成小数  goto add\_double; // 两个小数相加  }  }  // 如果以上逻辑未处理完，调用end\_add\_helper\_SPEC()助手函数  ZEND\_VM\_TAIL\_CALL(zend\_add\_helper\_SPEC(op1, op2 ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU\_CC));  } |

如上所示，函数的业务逻辑比较简单，先针对几种特定情况做了一些优化，如果不在这几种情况中，调用zend\_add\_helper\_SPEC()函数进行计算。

在这次加法操作中，第一个数字是变量，第二个是常量，所以第一个操作对象用EX\_VAR()宏程序来获取变量，第二个操作对象用RT\_CONSTANT()宏程序来获取变量。

zend\_add\_helper\_SPEC()函数定义如下：

|  |
| --- |
| static zend\_never\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_add\_helper\_SPEC(zval \*op\_1, zval \*op\_2 ZEND\_OPCODE\_HANDLER\_ARGS\_DC){  USE\_OPLINE  SAVE\_OPLINE();  if (UNEXPECTED(Z\_TYPE\_INFO\_P(op\_1) == IS\_UNDEF)) { // op1是无效变量  op\_1 = ZVAL\_UNDEFINED\_OP1(); // 报错  }  if (UNEXPECTED(Z\_TYPE\_INFO\_P(op\_2) == IS\_UNDEF)) { // op2是无效变量  op\_2 = ZVAL\_UNDEFINED\_OP2(); // 报错  }  add\_function(EX\_VAR(opline->result.var), op\_1, op\_2); // 调用加法函数  if (opline->op1\_type & (IS\_TMP\_VAR|IS\_VAR)) { // op1是变量  zval\_ptr\_dtor\_nogc(op\_1); // 销毁op1  }  if (opline->op2\_type & (IS\_TMP\_VAR|IS\_VAR)) { // op2是变量  zval\_ptr\_dtor\_nogc(op\_2); // 销毁op2  }  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION();  } |

add\_function()函数已经在前文中介绍过。

#### **相关宏程序**

相关宏程序定义如下：

|  |
| --- |
| // 偏移到目标操作码并返回 ,p1:是否检查异常？使用 EX(opline)：使用 opline 。p2:偏移量  #define ZEND\_VM\_NEXT\_OPCODE\_EX(check\_exception, skip) \  CHECK\_SYMBOL\_TABLES() /\* 调试用 \*/ \  if (check\_exception) { /\* 如果检查异常 \*/ \  /\* #define EX(element) ((execute\_data)->element) \*/ \  OPLINE = EX(opline) + (skip); /\* 在执行数据中，找到目标操作码 \*/ \  } else { /\* 如果不检查异常 \*/ \  ZEND\_ASSERT(!EG(exception)); /\* 不可以有异常 \*/ \  OPLINE = opline + (skip); /\* 通过偏移，找到目标操作码 \*/ \  } \  ZEND\_VM\_CONTINUE() /\* windows系统: return 0 \*/  // 在 EX(opline)向右偏移1个位置，找到目标操作码并 return  #define ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION() \  ZEND\_VM\_NEXT\_OPCODE\_EX(1, 1) /\* 找到目标操作码并 return \*/  // opline向右偏移1个位置，到目标操作码并 return  #define ZEND\_VM\_NEXT\_OPCODE() \  ZEND\_VM\_NEXT\_OPCODE\_EX(0, 1) /\* 找到目标操作码并 return \*/  // 执行call语句并返回  # define ZEND\_VM\_TAIL\_CALL(call) return call  // 以下是在windows平台中的定义  # define ZEND\_VM\_CONTINUE() return 0  # define ZEND\_VM\_RETURN() return -1 |

#### **ZEND\_VM\_DISPATCH\_TO\_HELPER()伪函数**

ZEND\_VM\_DISPATCH\_TO\_HELPER()不是真实存在的函数或宏程序，它是在zend\_vm\_def.h虚拟机伪代码中使用的伪函数，在调用zend\_vm\_gen.php脚本创建虚拟机时，把伪函数替换成真实的函数调用，例如，伪函数代码如下：

|  |
| --- |
| ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_add\_helper, op\_1, op1, op\_2, op2); |

转成真实的函数调用代码如下：

|  |
| --- |
| ZEND\_VM\_TAIL\_CALL(zend\_add\_helper\_SPEC(op1, op2  ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU\_CC)); |

## 三）减法操作码（ZEND\_SUB）

### 在编译时计算

减法操作码的处理流程与加法相同，在编译时，先尝试通过get\_binary\_op()函数取得对应的处理函数sub\_function()，尝试进行处理。sub\_function()函数在zend\_operators.c中定义，它调用sub\_function\_fast()函数和sub\_function\_slow()函数实现加法操作：

|  |
| --- |
| ZEND\_API zend\_result sub\_function(zval \*result, zval \*op1, zval \*op2) {  if (sub\_function\_fast(result, op1, op2) == SUCCESS) { // 特定类型，快速减法  return SUCCESS;  } else { // 失败，使用慢减法  return sub\_function\_slow(result, op1, op2);  }  } |

sub\_function\_fast()函数定义如下：

|  |
| --- |
| static zend\_result sub\_function\_fast(zval \*result, zval \*op1, zval \*op2) {  zend\_uchar type\_pair = TYPE\_PAIR(Z\_TYPE\_P(op1), Z\_TYPE\_P(op2));  if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_LONG, IS\_LONG))) { // 两个 long 型  fast\_long\_sub\_function(result, op1, op2);  return SUCCESS;  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_DOUBLE, IS\_DOUBLE))) { // 两个小数  ZVAL\_DOUBLE(result, Z\_DVAL\_P(op1) - Z\_DVAL\_P(op2));  return SUCCESS;  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_LONG, IS\_DOUBLE))) { // 左整数右小数  ZVAL\_DOUBLE(result, ((double)Z\_LVAL\_P(op1)) - Z\_DVAL\_P(op2));  return SUCCESS;  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_DOUBLE, IS\_LONG))) { // 左小数右整数  ZVAL\_DOUBLE(result, Z\_DVAL\_P(op1) - ((double)Z\_LVAL\_P(op2)));  return SUCCESS;  } else { // 其他情况不支持  return FAILURE;  }  } |

sub\_function\_slow()函数定义如下：

|  |
| --- |
| static sub\_function\_slow(zval \*result, zval \*op1, zval \*op2) {  ZVAL\_DEREF(op1); // 先追踪到引用目标  ZVAL\_DEREF(op2);  if (sub\_function\_fast(result, op1, op2) == SUCCESS) { // 特定类型，快速减法  return SUCCESS;  }  // 尝试调用op1 和 op2的 do\_operation 方法，如果成功直接返回  ZEND\_TRY\_BINARY\_OBJECT\_OPERATION(ZEND\_SUB);  zval op1\_copy, op2\_copy;  // op1，op2 转成数字  if (UNEXPECTED(zendi\_try\_convert\_scalar\_to\_number(op1, &op1\_copy) == FAILURE)  || UNEXPECTED(zendi\_try\_convert\_scalar\_to\_number(op2, &op2\_copy) == FAILURE)) {  ... // 报错  return FAILURE; // 返回失败  }  // 到这里说明转成数字成功，没有出错  if (result == op1) { // 如果 接收变量与第一个操作对象相同  zval\_ptr\_dtor(result); // 清空接收变量  }  if (sub\_function\_fast(result, &op1\_copy, &op2\_copy) == SUCCESS) { // 再次尝试快速减法  return SUCCESS;  }  ZEND\_ASSERT(0 && "Operation must succeed");  return FAILURE;  } |

如上所示，业务逻辑比较简单，先尝试快速减法，如果失败，把两个操作对象都转成数字，再次尝试快速减法。

### “减法”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_SUB操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HOT\_NOCONSTCONST\_HANDLER(2, ZEND\_SUB, CONST|TMPVARCV, CONST|TMPVARCV) {  // 业务逻辑略 ...  // 调用方法 zend\_sub\_helper  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_sub\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，此伪函数会根据两个操作对象的变量类型，被转换成4个处理器函数：

1）ZEND\_SUB\_SPEC\_CONST\_CONST\_HANDLER()函数，处理两个常量相减，例如**“1-1”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_SUB\_SPEC\_CONST\_TMPVARCV\_HANDLER()函数，处理一个常量和一个变量相减，如**“1-$a”**；

3）ZEND\_SUB\_SPEC\_TMPVARCV\_CONST\_HANDLER()函数，处理一个变量和一个常量相减，如**“$a-1”**；

4）ZEND\_SUB\_SPEC\_TMPVARCV\_TMPVARCV\_HANDLER()函数，处理两个变量相减,如**“$a-$b”**。

以第二种情况“echo 1-$a；”为例，这个语句中，第一个操作对象1是常量，第二个$a是变量，所以会调用ZEND\_SUB\_SPEC\_CONST\_TMPVARCV\_HANDLER()这个处理器函数，函数代码如下：

|  |
| --- |
| static ZEND\_VM\_HOT ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_SUB\_SPEC\_CONST\_TMPVARCV\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS){  USE\_OPLINE  zval \*op1, \*op2, \*result;  double d1, d2;  op1 = RT\_CONSTANT(opline, opline->op1);  op2 = EX\_VAR(opline->op2.var);  // 这部分优化的业务逻辑与加法操作相似  ...  ZEND\_VM\_TAIL\_CALL(zend\_sub\_helper\_SPEC(op1, op2 ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU\_CC));  } |

如上所示，与加法操作相似，函数的业务逻辑比较简单，先针对几种特定情况做了一些优化，如果不在这几种情况中，调用zend\_sub\_helper\_SPEC()函数进行计算。zend\_sub\_helper\_SPEC()函数的业务逻辑与加法助手zend\_add\_helper\_SPEC()函数类似：先检验两个操作对象并调用sub\_function()函数进行运算。

需要注意的是，在这交减法操作中，第一个数字是常量，第二个是变量，所以第一个操作对象用RT\_CONSTANT()宏程序来获取变量，第二个操作对象用EX\_VAR()宏程序来获取变量。这体现了同一个操作码对不同类型变量处理上的差异，也正因为有这些差异，每个操作码的处理器才会有多个分支。

关于操作码分支的更多介绍参见“虚拟机生成器篇”。

## 四）乘法（ZEND\_MUL）和

### 在编译时计算

乘法操作码为ZEND\_MUL，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数mul\_function()，尝试进行处理。mul\_function()函数在zend\_operators.c中定义，它调用mul\_function\_fast()函数和mul\_function\_slow()函数实现加法操作：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL mul\_function(zval \*result, zval \*op1, zval \*op2) {  if (mul\_function\_fast(result, op1, op2) == SUCCESS) { // 快速乘法  return SUCCESS;  } else { // 如果失败  return mul\_function\_slow(result, op1, op2); // 慢乘  }  } |

如上所示，操作过程以及mul\_function\_fast()函数和mul\_function\_slow()函数业务逻辑，与减法中的操作类似。

### “乘法”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_MUL操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(3, ZEND\_MUL, CONST|TMPVARCV, CONST|TMPVARCV, SPEC(COMMUTATIVE)) {  // 业务逻辑略 ...  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_mul\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，此伪函数会根据两个操作对象的变量类型，被转换成3个处理器函数：

1）ZEND\_MUL\_SPEC\_CONST\_CONST\_HANDLER()函数，处理两个常量相减，例如**“1\*1”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_MUL\_SPEC\_TMPVARCV\_CONST\_HANDLER()函数，处理一个常量和一个变量相减，如**“1\*$a”或“$a\*1”**；

3）ZEND\_MUL\_SPEC\_TMPVARCV\_TMPVARCV\_HANDLER()函数，处理一个变量和一个常量相减，如**“$a-1”**。

由于多加了一个扩展规则COMMUTATIVE，它表示两个操作对象顺序可以交换，所以“变量\*常量”和“常量\*变量”两种情况只会创建一个函数。由于变量（TMPVARCV）的顺序号大于常量（CONST）所以方法名中变量在前，常量在后“MPVARCV\_CONST”。

函数的业务逻辑比较简单，先针对几种特定情况做了一些优化，如果不在这几种情况中，调用zend\_mul\_helper\_SPEC()函数进行计算。zend\_mul\_helper\_SPEC()函数的业务逻辑与加法zend\_add\_helper\_SPEC()函数类似：先检验两个操作对象并调用mul\_function()函数进行处理。

## 五）除法（ZEND\_DIV）

### 在编译时计算

乘法操作码为ZEND\_DIV，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数div\_function()，尝试进行处理。div\_function()函数在zend\_operators.c中定义，它的业务逻辑比前几个操作要复杂一些：

|  |
| --- |
| ZEND\_API zend\_result div\_function(zval \*result, zval \*op1, zval \*op2) {  ZVAL\_DEREF(op1); // 先追踪到引用目标  ZVAL\_DEREF(op2);  int retval = div\_function\_base(result, op1, op2); // 先尝试普通除法  if (EXPECTED(retval == SUCCESS)) { // 成功直接返回  return SUCCESS;  }  if (UNEXPECTED(retval == DIV\_BY\_ZERO)) { // 除数是0  goto div\_by\_zero; // 报错  }  // 尝试调用op1 和 op2的 do\_operation 方法，如果成功直接返回  ZEND\_TRY\_BINARY\_OBJECT\_OPERATION(ZEND\_DIV);  zval result\_copy, op1\_copy, op2\_copy;  // op1，op2 转成数字  if (UNEXPECTED(zendi\_try\_convert\_scalar\_to\_number(op1, &op1\_copy) == FAILURE)  || UNEXPECTED(zendi\_try\_convert\_scalar\_to\_number(op2, &op2\_copy) == FAILURE)) {  ... // 报错  return FAILURE; // 返回失败  }  // 到这里说明没有出错，再次尝普通除法  retval = div\_function\_base(&result\_copy, &op1\_copy, &op2\_copy);  if (retval == SUCCESS) { // 如果成功  if (result == op1) { // 如果 接收变量与第一个操作对象相同，销毁 接收变量  zval\_ptr\_dtor(result);  }  ZVAL\_COPY\_VALUE(result, &result\_copy); // 创建副本返回  return SUCCESS;  }  div\_by\_zero:  // 要有错误信息：除数是0  ZEND\_ASSERT(retval == DIV\_BY\_ZERO && "TYPES\_NOT\_HANDLED should not occur here");  if (result != op1) {  ZVAL\_UNDEF(result); // 清空接收变量，中断操作  }  zend\_throw\_error(zend\_ce\_division\_by\_zero\_error, "Division by zero"); // 抛错：除数是0  return FAILURE; // 返回失败  } |

如上所示，尝试使用div\_function\_base()函数进行除法运算，比较特别的一点是，需要校验除数是否为0，这是除法运算本身的特殊性质。

div\_function\_base()函数的业务逻辑并不复杂，定义如下：

|  |
| --- |
| static int ZEND\_FASTCALL div\_function\_base(zval \*result, zval \*op1, zval \*op2) {  zend\_uchar type\_pair = TYPE\_PAIR(Z\_TYPE\_P(op1), Z\_TYPE\_P(op2));  // 根据类型处理  if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_LONG, IS\_LONG))) { // 两个整数  if (Z\_LVAL\_P(op2) == 0) { // 如果除数是0，返回错误  return DIV\_BY\_ZERO;  // 如果第二个是 -1,并且第一个是最大数  } else if (Z\_LVAL\_P(op2) == -1 && Z\_LVAL\_P(op1) == ZEND\_LONG\_MIN) {  // 这样直接除会出错，转成小数再除  ZVAL\_DOUBLE(result, (double) ZEND\_LONG\_MIN / -1);  return SUCCESS;  }    if (Z\_LVAL\_P(op1) % Z\_LVAL\_P(op2) == 0) { // 如果结果两个整数取余结果为0  // 直接按整数算除法，返回结果  ZVAL\_LONG(result, Z\_LVAL\_P(op1) / Z\_LVAL\_P(op2));  } else { // 如果有余数  // 转成小数算除法  ZVAL\_DOUBLE(result, ((double) Z\_LVAL\_P(op1)) / Z\_LVAL\_P(op2));  }  return SUCCESS; // 返回成功  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_DOUBLE, IS\_DOUBLE))) { // 两个小数  if (Z\_DVAL\_P(op2) == 0) { // 如果除数是0，返回错误  return DIV\_BY\_ZERO;  }  ZVAL\_DOUBLE(result, Z\_DVAL\_P(op1) / Z\_DVAL\_P(op2)); // 直接除  return SUCCESS;  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_DOUBLE, IS\_LONG))) { // 小数除以整数  if (Z\_LVAL\_P(op2) == 0) { // 如果除数是0，返回错误  return DIV\_BY\_ZERO;  }  ZVAL\_DOUBLE(result, Z\_DVAL\_P(op1) / (double)Z\_LVAL\_P(op2)); // 转成小数算除法  return SUCCESS;  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_LONG, IS\_DOUBLE))) { // 整数除以小数  if (Z\_DVAL\_P(op2) == 0) { // 如果除数是0，返回错误  return DIV\_BY\_ZERO;  }  ZVAL\_DOUBLE(result, (double)Z\_LVAL\_P(op1) / Z\_DVAL\_P(op2)); // 转成小数算除法  return SUCCESS;  } else { // 其他类型，不处理  return TYPES\_NOT\_HANDLED;  }  } |

### “除法”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_DIV操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(4, ZEND\_DIV, CONST|TMPVAR|CV, CONST|TMPVAR|CV){  USE\_OPLINE // const zend\_op \*opline = EX(opline);  zval \*op1, \*op2;  SAVE\_OPLINE(); // windows: 无操作  op1 = GET\_OP1\_ZVAL\_PTR(BP\_VAR\_R); // 获取zval指针，UNUSED 类型返回null  op2 = GET\_OP2\_ZVAL\_PTR(BP\_VAR\_R); // 获取zval指针，UNUSED 类型返回null  div\_function(EX\_VAR(opline->result.var), op1, op2); // 除法函数  FREE\_OP1(); // 释放操作对象的附加变量  FREE\_OP2();  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); 下一个操作码  } |

如上所示，此伪函数规定的操作对象变量类型与前面几个操作码不同，它没有把TMPVAR和CV类型合并处理，它会根据两个操作对象的变量类型，被转换成9个处理器函数：

1）ZEND\_DIV\_SPEC\_CONST\_CONST\_HANDLER()函数，处理两个常量相减，例如**“1/1”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_DIV\_SPEC\_CONST\_TMPVAR\_HANDLER()函数，处理一个常量和一个普通（或临时）变量相除，如**“1 /（$a+1）”（表达式“（$a+1）”会转成一个临时变量）**；

3）ZEND\_DIV\_SPEC\_CONST\_CV\_HANDLER()函数，处理一个常量和一个编译变量相除，如**“1/$a”（$a会转成一个编译变量）**；

4）ZEND\_DIV\_SPEC\_TMPVAR\_CONST\_HANDLER()函数，处理一个普通（或临时）变量和一个常量相除，如**“（$a+1）/1”**；

5）ZEND\_DIV\_SPEC\_TMPVAR\_TMPVAR\_HANDLER()函数，处理两个普通（或临时）变量相除，例如**“$GLOBALS['a'] / $GLOBALS['a'];”（$GLOBALS['a']会转成一个普通变量）**；

6）ZEND\_DIV\_SPEC\_TMPVAR\_CV\_HANDLER()函数，例如**“（$a+1）/ $a**”；

7）ZEND\_DIV\_SPEC\_CV\_CONST\_HANDLER()函数，例如**“$a / 1**”；

8）ZEND\_DIV\_SPEC\_CV\_TMPVAR\_HANDLER()函数，例如**“$a /（$a+1）**”；

9）ZEND\_DIV\_SPEC\_CV\_CV\_HANDLER()函数，例如**“$a / $a”**。

## 六）取余数（ZEND\_MOD）

取余数运算也称作“模运算”或“求模”。

### 在编译时计算

取余数操作码为ZEND\_MOD，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数mod\_function()，尝试进行处理。mod\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL mod\_function(zval \*result, zval \*op1, zval \*op2) {  zend\_long op1\_lval, op2\_lval;  // 把op1,op2都转成整数 ，并按要求进行计算  convert\_op1\_op2\_long(op1, op1\_lval, op2, op2\_lval, result, ZEND\_MOD, "%");  if (op2\_lval == 0) { // 第二个整数 是0,对0取余  if (EG(current\_execute\_data) && !CG(in\_compilation)) { // 如果在执行时，不在编译时  // 报错，对0取余  zend\_throw\_exception\_ex(zend\_ce\_division\_by\_zero\_error, 0, "Modulo by zero");  } else { // 否则  // 不带返回的报错，对0取余  zend\_error\_noreturn(E\_ERROR, "Modulo by zero");  }  if (op1 != result) { // 如果不是用op1接收  ZVAL\_UNDEF(result); // 清空结果变量，中断操作  }  return FAILURE; // 返回失败  }    if (op1 == result) { // 如果用op1接收  zval\_ptr\_dtor(result); // 清空op1  }  if (op2\_lval == -1) { // 取余只算整数部分，对 -1取余 ，其实和对 1 取余一样，值一定是0。  ZVAL\_LONG(result, 0); // 结果是0  return SUCCESS; // 反回 成功  }  ZVAL\_LONG(result, op1\_lval % op2\_lval); // 取余运算，返回结果  return SUCCESS;  } |

### 把操作对象转成整数

取余数操作只对整数有效，所以要把操作对象转成整数，这是一个看似简单但实现上很复杂的操作。在上一块代码中，用convert\_op1\_op2\_long()宏程序来达成这个目的。

#### **convert\_op1\_op2\_long()宏程序**

convert\_op1\_op2\_long()宏程序用于把两个操作对象都转成整数，并按要求操作，支持的操作包括左右位移和取余，代码如下：

|  |
| --- |
| // 把op1,op2都转成整数 ，并按要求进行计算（左右位移 和 取余。全局没有其他 文件 调用）  #define convert\_op1\_op2\_long(op1, op1\_lval, op2, op2\_lval, result, opcode, sigil) \  do { \  if (UNEXPECTED(Z\_TYPE\_P(op1) != IS\_LONG)) { /\* 如果op1不是整数 \*/ \  bool failed; \  if (Z\_ISREF\_P(op1)) { /\* 如果op1是引用 \*/ \  op1 = Z\_REFVAL\_P(op1); /\* 追踪到引用目标 \*/ \  if (Z\_TYPE\_P(op1) == IS\_LONG) { /\* 如果op1是整数 \*/ \  op1\_lval = Z\_LVAL\_P(op1); /\* 取出整数值 \*/ \  break; /\* 还会进入后面的 do-while \*/ \  } \  } \  /\* 调用op1的 do\_operation \*/ \  ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION(opcode); \  /\* zval转成整数 ，不支持资源类型 和 数组 \*/ \  op1\_lval = zendi\_try\_get\_long(op1, &failed); \  if (UNEXPECTED(failed)) { /\* 如果转换失败 \*/ \  zend\_binop\_error(sigil, op1, op2); /\* 二进制操作报错,不支持此操作符 \*/ \  if (result != op1) { /\* 如果接收变量不是第一个操作对象 \*/ \  ZVAL\_UNDEF(result); /\* 清空接收变量（中断操作） \*/ \  } \  return FAILURE; /\* 返回失败 \*/ \  } \  } else { /\* 如果op1是整数 \*/ \  op1\_lval = Z\_LVAL\_P(op1); /\* 取出并返回整数值 \*/ \  } \  } while (0); \  do { \  if (UNEXPECTED(Z\_TYPE\_P(op2) != IS\_LONG)) { /\* 如果op2不是整数 \*/ \  bool failed; \  if (Z\_ISREF\_P(op2)) { /\* 如果op2是引用 \*/ \  op2 = Z\_REFVAL\_P(op2); /\* 追踪到引用目标 \*/ \  if (Z\_TYPE\_P(op2) == IS\_LONG) { /\* 如果op1是整数 \*/ \  op2\_lval = Z\_LVAL\_P(op2); /\* 取出整数值 \*/ \  break; /\* 还会进入后面的 do-while \*/ \  } \  } \  /\* 调用op2的 do\_operation \*/ \  ZEND\_TRY\_BINARY\_OP2\_OBJECT\_OPERATION(opcode); \  /\* zval转成整数 ，不支持资源类型 和 数组 \*/ \  op2\_lval = zendi\_try\_get\_long(op2, &failed); \  if (UNEXPECTED(failed)) { /\* 如果转换失败 \*/ \  zend\_binop\_error(sigil, op1, op2); /\* 二进制操作报错,不支持此操作符 \*/ \  if (result != op1) { /\* 如果接收变量不是第一个操作对象 \*/ \  ZVAL\_UNDEF(result); /\* 清空接收变量（中断操作） \*/ \  } \  return FAILURE; /\* 返回失败 \*/ \  } \  } else { /\* 如果op1是整数 \*/ \  op2\_lval = Z\_LVAL\_P(op2); /\* 取出并返回整数值 \*/ \  } \  } while (0); |

如上所示，业务逻辑可归纳如下：

第一步：1）如果op1是引用类型，追踪到引用目标，如果目标是整数，跳过这一步；2）如果op1不是整数，尝试调用它的do\_operation()函数进行处理，并把op1转成整数；3）如果上一个操作返回错误，报错；如果op1不是引用类型，直接取出里面的整数；

第二步：对op2也进行相同的操作。

经过这轮处理，要么已经正常取出op1和op2中的整数，要么已经在过程中报错。

#### **zendi\_try\_get\_long()函数**

zendi\_try\_get\_long()函数用于把任何类型转成整数，代码如下：

|  |
| --- |
| static zend\_never\_inline zend\_long ZEND\_FASTCALL zendi\_try\_get\_long(zval \*op, bool \*failed) {  \*failed = 0;  switch (Z\_TYPE\_P(op)) {  case IS\_NULL: // null,false 转成0  case IS\_FALSE:  return 0;  case IS\_TRUE: // true转成1  return 1;  case IS\_DOUBLE: { // 小数  double dval = Z\_DVAL\_P(op);  zend\_long lval = zend\_dval\_to\_lval(dval); // 转成整数  if (!zend\_is\_long\_compatible(dval, lval)) {  // 报错：隐式转换，从小数转成整数会丢失精度  zend\_incompatible\_double\_to\_long\_error(dval);  if (UNEXPECTED(EG(exception))) { // 如果有exception  \*failed = 1; // 失败标记  }  }  return lval; // 返回整数  }  case IS\_STRING: // 字串  {  zend\_uchar type;  zend\_long lval;  double dval;  bool trailing\_data = false;  // 字串转成数值  type = is\_numeric\_string\_ex(Z\_STRVAL\_P(op), Z\_STRLEN\_P(op), &lval, &dval,  /\* allow errors \*/ true, NULL, &trailing\_data);  if (type == 0) { // 如果转换失败  \*failed = 1; // 失败标记  return 0; // 返回0  }  if (UNEXPECTED(trailing\_data)) { // 如果有尾数据  // 报错，遇到非数字字串  zend\_error(E\_WARNING, "A non-numeric value encountered");  if (UNEXPECTED(EG(exception))) { // 如果有 异常  \*failed = 1; // 失败标记  }  }  if (EXPECTED(type == IS\_LONG)) { // 转成整数  return lval; // 直接返回  } else { // 其他情况：转成了小数  // 强转成整数，不兼容时，返回最小或最大整数  lval = zend\_dval\_to\_lval\_cap(dval);  if (!zend\_is\_long\_compatible(dval, lval)) { // 如果整数和小数不兼容  zend\_incompatible\_string\_to\_long\_error(Z\_STR\_P(op)); // 报错  if (UNEXPECTED(EG(exception))) { // 如果有exception  \*failed = 1; // 失败标记  }  }  return lval; // 返回整数  }  }  case IS\_OBJECT: // 对象  {  zval dst;  // 调用自身的转换方法转成整数，如果转换失败或有 exception  if (Z\_OBJ\_HT\_P(op)->cast\_object(Z\_OBJ\_P(op), &dst, IS\_LONG) == FAILURE  || EG(exception)) {  \*failed = 1; // 失败标记  return 0; // 返回0  }  ZEND\_ASSERT(Z\_TYPE(dst) == IS\_LONG); // 返回值是整数  return Z\_LVAL(dst); // 取出整数并返回  }  case IS\_RESOURCE: // 资源和数组不能转  case IS\_ARRAY:  \*failed = 1; // 转换失败  return 0; // 返回0  EMPTY\_SWITCH\_DEFAULT\_CASE() // 不会有没在case里的类型  }  }  /\* }}} \*/ |

如上所示，把IS\_NULL、IS\_FALSE、IS\_TRUE、IS\_DOUBLE、IS\_STRING、IS\_OBJECT、IS\_RESOURCE、IS\_ARRAY这些类型实例转成整数，比较复杂的是字符串和小数轻喜剧民整数的操作。

字符串转成整数需要用到is\_numeric\_string\_ex()函数，这个函数的业务逻辑很长，所以放在附录中。以下是另外几个引用到的函数。

#### **zend\_dval\_to\_lval()函数**

zend\_dval\_to\_lval()函数用于把小数转成整数，代码如下：

|  |
| --- |
| static zend\_always\_inline zend\_long zend\_dval\_to\_lval(double d) {  // # define zend\_finite(a) isfinite(a)  // 如果小数不是有限的，或者是null  if (UNEXPECTED(!zend\_finite(d)) || UNEXPECTED(zend\_isnan(d))) {  return 0; // 返回 0  } else if (!ZEND\_DOUBLE\_FITS\_LONG(d)) { // d 不在 long 型的取值范围内  // 兼容32位和64位系统的转换方法（负数取它和最大数的和）  return zend\_dval\_to\_lval\_slow(d);  }  return (zend\_long)d; // 可以直接转成long，直接转  } |

zend\_dval\_to\_lval\_slow()函数在64位系统中定义如下：

|  |
| --- |
| ZEND\_API zend\_long ZEND\_FASTCALL zend\_dval\_to\_lval\_slow(double d){  double two\_pow\_64 = pow(2., 64.), dmod;  dmod = fmod(d, two\_pow\_64);  // 不需要用ceil取整，原 double 数 d 不能包含小数部分。所以dmod 也不会包含小数。  if (dmod < 0) {  dmod += two\_pow\_64;  }  return (zend\_long)(zend\_ulong)dmod ; // 强转成整数  } |

#### **zend\_is\_long\_compatible()函数**

zend\_is\_long\_compatible()函数用最简单的方法检验一个小数转成整数之后值是否会变，代码如下：

|  |
| --- |
| static zend\_always\_inline bool zend\_is\_long\_compatible(double d, zend\_long l) {  return (double)l == d;  } |

#### **zend\_dval\_to\_lval\_cap()函数**

zend\_dval\_to\_lval\_cap()函数用于把小数强制转成整数，代码如下：

|  |
| --- |
| sstatic zend\_always\_inline zend\_long zend\_dval\_to\_lval\_cap(double d){  // 如果小数不是有限的，或者是null  if (UNEXPECTED(!zend\_finite(d)) || UNEXPECTED(zend\_isnan(d))) {  return 0; // 返回0  } else if (!ZEND\_DOUBLE\_FITS\_LONG(d)) { // 如果不兼容整数  // 小数大于0，返回最大整数，否则返回最小整数，表示转换失败  return (d > 0 ? ZEND\_LONG\_MAX : ZEND\_LONG\_MIN);  }  return (zend\_long)d; // 小数正常转成整数  } |

#### ZEND\_DOUBLE\_FITS\_LONG()宏程序

ZEND\_DOUBLE\_FITS\_LONG()宏程序在64位系统中定义如下：

|  |
| --- |
| # define ZEND\_DOUBLE\_FITS\_LONG(d) (!((d) >= (double)ZEND\_LONG\_MAX || (d) < (double)ZEND\_LONG\_MIN)) |

如上所示，小数没有大于最大整数并且没有小于最小整数时，返回真；否则返回假。

### “取余数”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_MOD操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(5, ZEND\_MOD, CONST|TMPVARCV, CONST|TMPVARCV) {  USE\_OPLINE // const zend\_op \*opline = EX(opline);  zval \*op1, \*op2, \*result;  op1 = GET\_OP1\_ZVAL\_PTR\_UNDEF(BP\_VAR\_R); // 获取zval指针, UNUSED 返回null  op2 = GET\_OP2\_ZVAL\_PTR\_UNDEF(BP\_VAR\_R); // 获取zval指针, UNUSED 返回null  // 如果两个都是常量  if (ZEND\_VM\_SPEC && OP1\_TYPE == IS\_CONST && OP2\_TYPE == IS\_CONST) {  /\* pass \*/  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op1) == IS\_LONG)) { // 如果op2不是整数  if (EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_LONG)) { // 如果 op2 是整数  result = EX\_VAR(opline->result.var); // 在执行数据中取出 指定序号的变量  if (UNEXPECTED(Z\_LVAL\_P(op2) == 0)) { // 如果操作对象2值为0  // 调用助手 zend\_mod\_by\_zero\_helper  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_mod\_by\_zero\_helper);  } else if (UNEXPECTED(Z\_LVAL\_P(op2) == -1)) { // 如果操作对象2 值为 -1  ZVAL\_LONG(result, 0); // 结果为0  } else { // 否则  ZVAL\_LONG(result, Z\_LVAL\_P(op1) % Z\_LVAL\_P(op2)); // 直接取余运算  }  ZEND\_VM\_NEXT\_OPCODE(); // 跳到下一个操作码  }  }  // 取余助手 , 检验传入的变量是否有效，进行取余运算，然后销毁运算对象op1,op2  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_mod\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，此伪函数会根据两个操作对象的变量类型，被转换成4个处理器函数：

1）ZEND\_MOD\_SPEC\_CONST\_CONST\_HANDLER()函数，处理两个常量取余数，例如**“3 % 2”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_MOD\_SPEC\_CONST\_TMPVARCV\_HANDLER()函数，处理一个常量和一个变量取余数，如**“1 % $a”**；

3）ZEND\_MOD\_SPEC\_TMPVARCV\_CONST\_HANDLER()函数，处理一个变量和一个常量取余数，如**“$a % 1”**；

4）ZEND\_MOD\_SPEC\_TMPVARCV\_TMPVARCV\_HANDLER()函数，处理两个变量取余数,如**“$a % $b”**。

函数的业务逻辑比较简单，先针对几种特定情况做了一些优化，并增加了一步和除法操作相似的检验，检验除数是否是0，如果除数是0，调用zend\_mod\_by\_zero\_helper\_SPEC()函数进行报错，函数业务逻辑如下：

|  |
| --- |
| static zend\_never\_inline ZEND\_COLD ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_mod\_by\_zero\_helper\_SPEC(ZEND\_OPCODE\_HANDLER\_ARGS) {  USE\_OPLINE // const zend\_op \*opline = EX(opline);  SAVE\_OPLINE();  zend\_throw\_exception\_ex(zend\_ce\_division\_by\_zero\_error, 0, "Modulo by zero");  ZVAL\_UNDEF(EX\_VAR(opline->result.var));  HANDLE\_EXCEPTION();  } |

如果没有在前面几种优化情况中处理完毕，最后调用zend\_mod\_helper\_SPEC()函数进行计算。zend\_mod\_helper\_SPEC()函数的业务逻辑与加法助手zend\_add\_helper\_SPEC()函数类似：检验两个操作对象并调用mod\_function()函数进行处理。

## 七）幂运算（ZEND\_POW）

### 在编译时计算

幂运算操作码为ZEND\_POW，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数pow\_function()，尝试进行处理。pow\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL pow\_function(zval \*result, zval \*op1, zval \*op2) {  ZVAL\_DEREF(op1); // 先追踪到引用目标  ZVAL\_DEREF(op2);    if (pow\_function\_base(result, op1, op2) == SUCCESS) { // 幂运算  return SUCCESS;  }    // 尝试调用op1 和 op2的 do\_operation 方法，如果成功直接返回  ZEND\_TRY\_BINARY\_OBJECT\_OPERATION(ZEND\_POW);  zval op1\_copy, op2\_copy;  // op1，op2 转成数字，如果出错  if (UNEXPECTED(zendi\_try\_convert\_scalar\_to\_number(op1, &op1\_copy) == FAILURE)  || UNEXPECTED(zendi\_try\_convert\_scalar\_to\_number(op2, &op2\_copy) == FAILURE)) {  ... // 报错  return FAILURE; // 返回失败  }  // 到这里说明没有出错  if (result == op1) { // 如果结果变量与第一个操作对象相同  zval\_ptr\_dtor(result); // 清空结果变量  }  if (pow\_function\_base(result, &op1\_copy, &op2\_copy) == SUCCESS) { // 幂运算  return SUCCESS; // 成功  }  ZEND\_ASSERT(0 && "Operation must succeed");  return FAILURE; // 失败  } |

如上所示，尝试使用pow\_function\_base()函数进行幂运算，函数代码如下：

|  |
| --- |
| // 幂运算，先用整数乘法，如果溢出转成小数幂运算  static zend\_result ZEND\_FASTCALL pow\_function\_base(zval \*result, zval \*op1, zval \*op2) {  zend\_uchar type\_pair = TYPE\_PAIR(Z\_TYPE\_P(op1), Z\_TYPE\_P(op2));  // 如果是两个整数  if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_LONG, IS\_LONG))) {  if (Z\_LVAL\_P(op2) >= 0) { // 幂次数大于等于0  zend\_long l1 = 1, l2 = Z\_LVAL\_P(op1), i = Z\_LVAL\_P(op2);  if (i == 0) { // 如果是0次幂  ZVAL\_LONG(result, 1L); // 结果为1  return SUCCESS;  } else if (l2 == 0) { // 如果是0的幂  ZVAL\_LONG(result, 0); // 结果为0  return SUCCESS;  }  while (i >= 1) { // 用循环相乘代替调用pow()函数  zend\_long overflow;  double dval = 0.0;  if (i % 2) { // 如果i（幂次数）是奇数  --i; // i自减  // 算乘法 l1 = l1 \* l2  ZEND\_SIGNED\_MULTIPLY\_LONG(l1, l2, l1, dval, overflow);  if (overflow) { // 如果有溢出  // 转成小数计数，未完成部分用 pow()函数计算  ZVAL\_DOUBLE(result, dval \* pow(l2, i));  return SUCCESS;  }  // 没有溢出，继续循环  } else {  i /= 2; // i减半  // 算乘法 l2 = l2 \* l2  ZEND\_SIGNED\_MULTIPLY\_LONG(l2, l2, l2, dval, overflow);  if (overflow) { // 如果有溢出  ZVAL\_DOUBLE(result, (double)l1 \* pow(dval, i)); // 转成小数计数  return SUCCESS;  }  }  }  // i==0 才走到这里，幂运算也用乘法运算计算完了  ZVAL\_LONG(result, l1); // 运算结果复制到结果变量里  } else { // op2 小于 0  // 调用底层函数计算（支持小数和负数的幂运算）  ZVAL\_DOUBLE(result, pow((double)Z\_LVAL\_P(op1), (double)Z\_LVAL\_P(op2)));  }  return SUCCESS;  // 如果是两个小数  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_DOUBLE, IS\_DOUBLE))) {  // 调用操作系统 pow 方法，传入两个小数  ZVAL\_DOUBLE(result, pow(Z\_DVAL\_P(op1), Z\_DVAL\_P(op2)));  return SUCCESS;  // 如果是一个整数，一个小数  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_LONG, IS\_DOUBLE))) {  // 调用操作系统 pow 方法，传入两个小数  ZVAL\_DOUBLE(result, pow((double)Z\_LVAL\_P(op1), Z\_DVAL\_P(op2)));  return SUCCESS;  // 如果是一个小数，一个整数  } else if (EXPECTED(type\_pair == TYPE\_PAIR(IS\_DOUBLE, IS\_LONG))) {  // 调用操作系统 pow 方法，传入两个小数  ZVAL\_DOUBLE(result, pow(Z\_DVAL\_P(op1), (double)Z\_LVAL\_P(op2)));  return SUCCESS;  } else { // 其他情况，返回失败  return FAILURE;  }  } |

如上所示，函数的业务逻辑不算复杂，可归纳为几种情况：

1）如果两个操作对象都是整数，并且op2大于等于0：

a）任何数字的0次幂值为1；

b）0的任何次幂值为0（0不可以有负数次幂，会报错，原理与0不能做除数相同）；

c）用循环乘法来代替调用pow()函数进行幂运算；

2）其他情况一律使用pow()函数来计算，需要转成小数时转成小数。

ZEND\_SIGNED\_MULTIPLY\_LONG()宏程序用于计算整数的乘法，由于乘法的使用频率很高，PHP内核中针对此宏程序做了大量优化，代码在zend\_multiply.h文件中，此宏程序在不同运行环境中有多个分支，64位windows中使用的分支如下：

|  |
| --- |
| # define ZEND\_SIGNED\_MULTIPLY\_LONG(a, b, lval, dval, usedval) do { \  \_\_int64 \_\_high; \  \_\_int64 \_\_low = \_mul128((a), (b), &\_\_high); \  if ((\_\_low >> 63I64) == \_\_high) { \  (usedval) = 0; \  (lval) = \_\_low; \  } else { \  (usedval) = 1; \  (dval) = (double)(a) \* (double)(b); \  } \  } while (0) |

### “幂运算”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_DIV操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(12, ZEND\_POW, CONST|TMPVAR|CV, CONST|TMPVAR|CV){  USE\_OPLINE // const zend\_op \*opline = EX(opline);  zval \*op1, \*op2;  SAVE\_OPLINE(); // windows: 无操作  op1 = GET\_OP1\_ZVAL\_PTR(BP\_VAR\_R); // 用读取模式获取变量  op2 = GET\_OP2\_ZVAL\_PTR(BP\_VAR\_R);  pow\_function(EX\_VAR(opline->result.var), op1, op2); // 幂运算  FREE\_OP1(); // 释放操作对象的附加变量  FREE\_OP2();  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  } |

如上所示，此伪函数会根据两个操作对象的变量类型，被转换成9个处理器函数：

1）ZEND\_POW\_SPEC\_CONST\_CONST\_HANDLER()函数，处理两个常量相减，例如**“1 \*\* 1”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_POW\_SPEC\_CONST\_TMPVAR\_HANDLER()函数，处理一个常量和一个普通（或临时）变量相除，如**“1 \*\* ($a+1)”（表达式“($a+1)”会转成一个临时变量）**；

3）ZEND\_POW\_SPEC\_CONST\_CV\_HANDLER()函数，处理一个常量和一个编译变量相除，如**“1 \*\* $a”（$a会转成一个编译变量）**；

4）ZEND\_POW\_SPEC\_TMPVAR\_CONST\_HANDLER()函数，处理一个普通（或临时）变量和一个常量相除，如**“($a+1) \*\* 1”**；

5）ZEND\_POW\_SPEC\_TMPVAR\_TMPVAR\_HANDLER()函数，处理两个普通（或临时）变量相除，例如**“$GLOBALS['a'] \*\* $GLOBALS['a'];”（$GLOBALS['a']会转成一个普通变量）**；

6）ZEND\_POW\_SPEC\_TMPVAR\_CV\_HANDLER()函数，例如**“($a+1) \*\* $a**”；

7）ZEND\_POW\_SPEC\_CV\_CONST\_HANDLER()函数，例如**“$a \*\* 1**”；

8）ZEND\_POW\_SPEC\_CV\_TMPVAR\_HANDLER()函数，例如**“$a \*\* ($a+1)**”；

9）ZEND\_POW\_SPEC\_CV\_CV\_HANDLER()函数，例如**“$a \*\* $a”**。

# 四、位运算

位运算包括：左位移、右位移、二进制或运算（or）、二进制与运算（and）、二进制抑或运算（xor）。这些表达式在解析时都会被创建成二元运算语句（ZEND\_AST\_BINARY\_OP），再进行编译和执行。

## 一）左位移运算（ZEND\_SL）

### 在编译时计算

左位移运算操作码为ZEND\_SL，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数shift\_left\_function()，尝试进行处理。shift\_left\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL shift\_left\_function(zval \*result, zval \*op1, zval \*op2) {  zend\_long op1\_lval, op2\_lval;  // 把op1,op2都转成整数 ，并按要求进行计算  convert\_op1\_op2\_long(op1, op1\_lval, op2, op2\_lval, result, ZEND\_SL, "<<");  // 防止某些处理器的怪癖封装，<< 64+x = <<x  // 如果位移数 大于等于 最大整数位数  if (UNEXPECTED((zend\_ulong)op2\_lval >= SIZEOF\_ZEND\_LONG \* 8)) {  if (EXPECTED(op2\_lval > 0)) { // 如果位移数 > 0  if (op1 == result) { // 如果用op1接收结果  zval\_ptr\_dtor(result); // 清空op1  }  ZVAL\_LONG(result, 0); // 结果为0  return SUCCESS;  } else { // 位移数不大于0  ... // 报错  return FAILURE; // 返回失败  }  }  ... // 用op1接收结果时，清空它  // 转成无符号，进行位操作，再转回来  ZVAL\_LONG(result, (zend\_long) ((zend\_ulong) op1\_lval << op2\_lval));  return SUCCESS;  } |

### “左位移”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_SL操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(6, ZEND\_SL, CONST|TMPVARCV, CONST|TMPVARCV) {  ... // 获取op1和op2的变量  // 如果两个都是常量，什么也不做，因为应该处理过了  if (ZEND\_VM\_SPEC && OP1\_TYPE == IS\_CONST && OP2\_TYPE == IS\_CONST) {  /\* pass \*/  // 如果两个都是整数 并且 op2 小于一个完整整数的位置数（移动后结果不为0）  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op1) == IS\_LONG)  && EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_LONG)  && EXPECTED((zend\_ulong)Z\_LVAL\_P(op2) < SIZEOF\_ZEND\_LONG \* 8)) {  // 把op1转成无符号，进行位移运算，再转回来  **ZVAL\_LONG(EX\_VAR(opline->result.var),**  **(zend\_long) ((zend\_ulong) Z\_LVAL\_P(op1) << Z\_LVAL\_P(op2)));**  ZEND\_VM\_NEXT\_OPCODE(); // 跳到下一个操作码  }  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_shift\_left\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，函数的业务逻辑比较简单，先针对特定情况做了一些优化，如果不在这种情况中，调用zend\_shift\_left\_helper\_SPEC()函数进行计算。此伪函数会被转换成4个处理器函数：

1）ZEND\_SL\_SPEC\_CONST\_CONST\_HANDLER()函数，处理左右两侧都是常量的操作，例如**“1<<1”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_SL\_SPEC\_CONST\_TMPVARCV\_HANDLER()函数，处理左侧是常量右侧是变量的操作，如**“1<<$a”**；

3）ZEND\_SL\_SPEC\_TMPVARCV\_CONST\_HANDLER()函数，处理左侧是变量右侧是常量的操作，如**“$a<<1”**；

4）ZEND\_SL\_SPEC\_TMPVARCV\_TMPVARCV\_HANDLER()函数，左右两侧都是变量的操作，如**“$a<<$b”**。

zend\_shift\_left\_helper\_SPEC()函数定义如下：

|  |
| --- |
| static zend\_never\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_shift\_left\_helper\_SPEC(zval \*op\_1, zval \*op\_2 ZEND\_OPCODE\_HANDLER\_ARGS\_DC){  ... // 取得op1，op2的变量  shift\_left\_function(EX\_VAR(opline->result.var), op\_1, op\_2);  ... // 删除op1，op2的变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  } |

## 二）右位移运算（ZEND\_SR）

### 在编译时计算

右位移运算操作码为ZEND\_SR，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数shift\_right\_function()，尝试进行处理。shift\_left\_function()函数在zend\_operators.c中定义，业务逻辑与shift\_left\_function()类似。

### “右位移”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_SR操作码的处理器定义如下，定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(7, ZEND\_SR, CONST|TMPVARCV, CONST|TMPVARCV) {  ... // 获取op1和op2的变量  // 如果两个都是常量，什么也不做，因为应该处理过了  if (ZEND\_VM\_SPEC && OP1\_TYPE == IS\_CONST && OP2\_TYPE == IS\_CONST) {  /\* pass \*/  // 如果两个都是整数 并且 op2 小于一个完整整数的位置数（移动后结果不为0）  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op1) == IS\_LONG)  && EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_LONG)  && EXPECTED((zend\_ulong)Z\_LVAL\_P(op2) < SIZEOF\_ZEND\_LONG \* 8)) {  // 直接进行位移运算  ZVAL\_LONG(EX\_VAR(opline->result.var), Z\_LVAL\_P(op1) >> Z\_LVAL\_P(op2));  ZEND\_VM\_NEXT\_OPCODE();  }  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_shift\_right\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，函数的业务逻辑比较简单，先针对特定情况做了一些优化，如果不在这种情况中，调用zend\_shift\_right\_helper\_SPEC()函数进行计算。此伪函数会被转换成4个处理器函数：

1）ZEND\_SR\_SPEC\_CONST\_CONST\_HANDLER()函数，处理左右两侧都是常量的操作，例如**“1>>1”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_SR\_SPEC\_CONST\_TMPVARCV\_HANDLER()函数，处理左侧是常量右侧是变量的操作，如**“1>>$a”**；

3）ZEND\_SR\_SPEC\_TMPVARCV\_CONST\_HANDLER()函数，处理左侧是变量右侧是常量的操作，如**“$a>>1”**；

4）ZEND\_SR\_SPEC\_TMPVARCV\_TMPVARCV\_HANDLER()函数，左右两侧都是变量的操作，如**“$a>>$b”**。

操作码的助手zend\_shift\_right\_helper\_SPEC()函数定义如下：

|  |
| --- |
| static zend\_never\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_shift\_right\_helper\_SPEC(zval \*op\_1, zval \*op\_2 ZEND\_OPCODE\_HANDLER\_ARGS\_DC) ... // 取得op1，op2的变量  shift\_right\_function(EX\_VAR(opline->result.var), op\_1, op\_2);  ... // 删除op1，op2的变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  } |

## 三）二进制或运算（ZEND\_BW\_OR）

### 在编译时计算

左位移运算操作码为ZEND\_BW\_OR，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数bitwise\_or\_function()，尝试进行处理。bitwise\_or\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL bitwise\_or\_function(zval \*result, zval \*op1, zval \*op2){  zend\_long op1\_lval, op2\_lval;  // 如果两个都是整数  if (EXPECTED(Z\_TYPE\_P(op1) == IS\_LONG) && EXPECTED(Z\_TYPE\_P(op2) == IS\_LONG)) {  ZVAL\_LONG(result, Z\_LVAL\_P(op1) | Z\_LVAL\_P(op2)); // 直接或运算  return SUCCESS;  }  ZVAL\_DEREF(op1); // 先追踪到引用目标  ZVAL\_DEREF(op2);  // 两个都是字串  if (Z\_TYPE\_P(op1) == IS\_STRING && EXPECTED(Z\_TYPE\_P(op2) == IS\_STRING)) {  zval \*longer, \*shorter;  zend\_string \*str;  size\_t i;  if (EXPECTED(Z\_STRLEN\_P(op1) >= Z\_STRLEN\_P(op2))) { // op1 更长  // 如果两个都是单个字符  if (EXPECTED(Z\_STRLEN\_P(op1) == Z\_STRLEN\_P(op2)) && Z\_STRLEN\_P(op1) == 1) {  // 转成char型，进行或运算  zend\_uchar or = (zend\_uchar) (\*Z\_STRVAL\_P(op1) | \*Z\_STRVAL\_P(op2));  if (result==op1) { // 如果用op1接收结果  zval\_ptr\_dtor\_str(result); // 清空op1  }  ZVAL\_CHAR(result, or); // 更新结果  return SUCCESS;  }  longer = op1; // 不都是单字符，按长短区分  shorter = op2;  } else { // op2更长  longer = op2; // 按长短区分  shorter = op1;  }  // 再分配一个长的字串  str = zend\_string\_alloc(Z\_STRLEN\_P(longer), 0);  // 逐个字符做或运算，结果放在新串里  for (i = 0; i < Z\_STRLEN\_P(shorter); i++) {  ZSTR\_VAL(str)[i] = Z\_STRVAL\_P(longer)[i] | Z\_STRVAL\_P(shorter)[i];  }  // 把多出的部分直接copy  memcpy(ZSTR\_VAL(str) + i, Z\_STRVAL\_P(longer) + i, Z\_STRLEN\_P(longer) - i + 1);  if (result==op1) { // 如果用op1接收结果  zval\_ptr\_dtor\_str(result); // 清空op1  }  ZVAL\_NEW\_STR(result, str); // 更新结果  return SUCCESS;  }  // 如果op1不是整数  if (UNEXPECTED(Z\_TYPE\_P(op1) != IS\_LONG)) {  bool failed;  // op1 的 do\_operation方法，操作符 ZEND\_BW\_OR  ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION(ZEND\_BW\_OR);  op1\_lval = zendi\_try\_get\_long(op1, &failed); // op1转成整数  if (UNEXPECTED(failed)) { // 如果出错了，报错，清空接收变量，返回失败  ...  return FAILURE;  }  } else { // op1是整数  op1\_lval = Z\_LVAL\_P(op1); // 取出整数  }    if (UNEXPECTED(Z\_TYPE\_P(op2) != IS\_LONG)) { // 如果op2不是整数  bool failed;  // op2 的 do\_operation方法，操作符 ZEND\_BW\_OR  ZEND\_TRY\_BINARY\_OP2\_OBJECT\_OPERATION(ZEND\_BW\_OR);  op2\_lval = zendi\_try\_get\_long(op2, &failed); // op2转成整数  // 如果出错了  if (UNEXPECTED(failed)) { // 如果出错了，报错，清空接收变量，返回失败  ...  return FAILURE;  }  } else { // op2是整数  op2\_lval = Z\_LVAL\_P(op2); // 取出整数  }  if (op1 == result) { // 如果用op1接收结果  zval\_ptr\_dtor(result); // 先清空op1  }  ZVAL\_LONG(result, op1\_lval | op2\_lval); // 两个整数作或运算，把值返回给p1  return SUCCESS;  } |

如上所示，代码虽然比较多但业务逻辑简单，按类型区分，分别独处理字符串的或运算和整数的或运算。在字符串的运算中，还考虑了两个单个字符的或运算。

### “二进制或”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_BW\_OR操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HOT\_NOCONSTCONST\_HANDLER(9, ZEND\_BW\_OR, CONST|TMPVARCV, CONST|TMPVARCV, SPEC(COMMUTATIVE)) {  ... // 获取op1和op2的变量  // op1或op2有一个不是ANY  if (ZEND\_VM\_SPEC && OP1\_TYPE == IS\_CONST && OP2\_TYPE == IS\_CONST) {  /\* pass \*/  // 否则， 如果 两个 变量类型 都为 整数  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op1) == IS\_LONG)  && EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_LONG)) {  // 直接做 或运算  ZVAL\_LONG(EX\_VAR(opline->result.var), Z\_LVAL\_P(op1) | Z\_LVAL\_P(op2));  ZEND\_VM\_NEXT\_OPCODE(); // 跳到下一个操作码  }  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_bw\_or\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，由于添加了扩展规则COMMUTATIVE（操作对象可交换），此伪函数会根据两个操作对象的变量类型，被转换成3个处理器函数：

1）ZEND\_BW\_OR\_SPEC\_CONST\_CONST\_HANDLER()函数，处理两个常量相减，例如**“1 | 1”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_BW\_OR\_SPEC\_TMPVARCV\_CONST\_HANDLER()函数，处理一个常量和一个变量相减，如**“1 | $a”或“$a | 1”**；

3）ZEND\_BW\_OR\_SPEC\_TMPVARCV\_TMPVARCV\_HANDLER()函数，处理一个变量和一个常量相减，如**“$a | $b”**。

如上所示，函数的业务逻辑比较简单，先针对特定情况做了一些优化，如果不在这种情况中，调用zend\_bw\_or\_helper\_SPEC()函数进行计算。zend\_bw\_or\_helper\_SPEC()函数定义如下：

|  |
| --- |
| static zend\_never\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_bw\_or\_helper\_SPEC(zval \*op\_1, zval \*op\_2 ZEND\_OPCODE\_HANDLER\_ARGS\_DC) {  ... // 取得op1，op2的变量  bitwise\_or\_function(EX\_VAR(opline->result.var), op\_1, op\_2);  ... // 销毁op1，op2的变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION();  } |

## ？四）二进制与运算（ZEND\_BW\_AND）

二进制与运算操作码为ZEND\_BW\_AND，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数bitwise\_and\_function()，尝试进行处理。bitwise\_and\_function()函数在zend\_operators.c中定义，业务逻辑与bitwise\_or\_function()类似。

ZEND\_BW\_AND操作码的处理器的业务逻辑也与ZEND\_BW\_OR类似。

操作码的助手zend\_bw\_and\_helper\_SPEC()函数业务逻辑与zend\_bw\_or\_helper\_SPEC()函数类似。

## ？五）二进制抑或运算（ZEND\_BW\_XOR）

二进制抑或运算操作码为ZEND\_BW\_XOR，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数bitwise\_xor\_function()，尝试进行处理。bitwise\_xor\_function()函数在zend\_operators.c中定义，业务逻辑与bitwise\_or\_function()类似。

ZEND\_XOR\_AND操作码的处理器的业务逻辑也与ZEND\_BW\_OR类似。

操作码的助手zend\_bw\_xor\_helper\_SPEC()函数业务逻辑与zend\_bw\_or\_helper\_SPEC()函数类似。

# 四、比较运算

比较运算包括：相同（ZEND\_IS\_IDENTICAL），不相同，相等（ZEND\_IS\_EQUAL），不相等，小于，小于等于，太空船（ZEND\_SPACESHIP）。这些表达式在解析时都会被创建成二元运算语句（ZEND\_AST\_BINARY\_OP），再进行编译和执行。

## 一）相同（ZEND\_IS\_IDENTICAL）

### 在编译时计算

相同运算操作码为ZEND\_IS\_IDENTICAL，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数is\_identical\_function()，尝试进行处理。is\_identical\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result is\_identical\_function(zval \*result, zval \*op1, zval \*op2){  ZVAL\_BOOL(result, zend\_is\_identical(op1, op2));  return SUCCESS;  } |

zend\_is\_identical()函数用于比较两个变量是否相同（===），定义如下：

|  |
| --- |
| ZEND\_API bool ZEND\_FASTCALL zend\_is\_identical(zval \*op1, zval \*op2) {  if (Z\_TYPE\_P(op1) != Z\_TYPE\_P(op2)) { // 如果类型不一样，直接返回 false  return 0;  }  switch (Z\_TYPE\_P(op1)) { 按类型操作  case IS\_NULL: // null ,true,false 返回 true  case IS\_FALSE:  case IS\_TRUE:  return 1;  case IS\_LONG: // 整数，转成 long 型比较  return (Z\_LVAL\_P(op1) == Z\_LVAL\_P(op2));  case IS\_RESOURCE: // 资源，比 .res 引用指针  return (Z\_RES\_P(op1) == Z\_RES\_P(op2));  case IS\_DOUBLE: // 小数  return (Z\_DVAL\_P(op1) == Z\_DVAL\_P(op2));  case IS\_STRING: // 字串  return zend\_string\_equals(Z\_STR\_P(op1), Z\_STR\_P(op2));  case IS\_ARRAY: // 数组，先比指针。不同再比内容  return (Z\_ARRVAL\_P(op1) == Z\_ARRVAL\_P(op2) ||  zend\_hash\_compare(Z\_ARRVAL\_P(op1), Z\_ARRVAL\_P(op2), (compare\_func\_t) hash\_zval\_identical\_function, 1) == 0);  case IS\_OBJECT: // 对象，必须指针相同  return (Z\_OBJ\_P(op1) == Z\_OBJ\_P(op2));  default: // 其他类型直接 false  return 0;  }  } |

zend\_string\_equals()函数的相关内容参见“类型篇”。

#### **hash\_zval\_identical\_function()函数**

zend\_hash\_compare()函数的相关内容参见“数组篇”。它用于比较两个数组的每个元素是否相同，传给它的hash\_zval\_identical\_function()函数用比较两个数组的单个元素，函数定义如下：

|  |
| --- |
| static int hash\_zval\_identical\_function(zval \*z1, zval \*z2) {  ZVAL\_DEREF(z1); // 先追踪到引用目标  ZVAL\_DEREF(z2);  return fast\_is\_not\_identical\_function(z1, z2); // 比较两个zval  } |

fast\_is\_not\_identical\_function()函数定义如下：

|  |
| --- |
| static zend\_always\_inline bool fast\_is\_not\_identical\_function(zval \*op1, zval \*op2) {  if (Z\_TYPE\_P(op1) != Z\_TYPE\_P(op2)) { // 类型不同，返回true  return 1;  } else if (Z\_TYPE\_P(op1) <= IS\_TRUE) { // op1：true,false,null,UNDEF  return 0; // 返回false  }  return !zend\_is\_identical(op1, op2); // 任意类型比较相同，取否  } |

### “相同”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_IS\_IDENTICAL操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(16, ZEND\_IS\_IDENTICAL, CONST|TMP|VAR|CV, CONST|TMP|VAR|CV, SPEC(COMMUTATIVE)) {  ... // 取出两个操作对象的变量  result = fast\_is\_identical\_function(op1, op2); // 检查是否相同  FREE\_OP1(); // 释放操作对象的附加变量  FREE\_OP2();  // 根据当前操作码结果选择下一个操作码，p1:传入值，主要用来做判断。p2:是否检查异常  ZEND\_VM\_SMART\_BRANCH(result, 1);  } |

如上所示，由于添加了COMMUTATIVE（操作对象可交换）规则，此伪函数会根据两个操作对象的变量类型，被转换成10个处理器函数：

1）ZEND\_IS\_IDENTICAL\_SPEC\_CONST\_CONST\_HANDLER()函数，比较两个常量是否相同，例如**“1===1”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_IS\_IDENTICAL\_SPEC\_TMP\_CONST\_HANDLER()函数，比较一个临时变量和一个常量是否相同，如**“($a+1) === 1”或“1 === ($a+1)”**；

3）ZEND\_IS\_IDENTICAL\_SPEC\_TMP\_TMP\_HANDLER()函数，比较两个临时变量是否相同，如**“($a+1) === ($b+1)”**；

4）ZEND\_IS\_IDENTICAL\_SPEC\_VAR\_CONST\_HANDLER()函数，比较一个普通变量和一个常量是否相同，如**“$GLOBALS['a']===1”或“1===$GLOBALS[a]”**；

5）ZEND\_IS\_IDENTICAL\_SPEC\_VAR\_TMP\_HANDLER()函数，比较一个普通变量和一个临时变量是否相同，如**“$GLOBALS['a']===($a+1)”或“($a+1)===$GLOBALS[a]”**；

6）ZEND\_IS\_IDENTICAL\_SPEC\_VAR\_VAR\_HANDLER()函数，比较两个普通变是否相同，**“$GLOBALS['a'] === $GLOBALS[b]”**；

7）ZEND\_IS\_IDENTICAL\_SPEC\_CV\_CONST\_HANDLER()函数，比较一个编译变量和一个常量是否相同，**“$a===1”或“1===$a”**；

8）ZEND\_IS\_IDENTICAL\_SPEC\_CV\_TMP\_HANDLER()函数，比较一个编译变量和一个临时变量是否相同，**“$a===($b+1)”或“($b+1) === $a”**；

9）ZEND\_IS\_IDENTICAL\_SPEC\_CV\_VAR\_HANDLER()函数，比较一个编译变量和一个普通变量是否相同，**“$a===$GLOBALS['a']”或“$GLOBALS['a']===$a”**；

10）ZEND\_IS\_IDENTICAL\_SPEC\_CV\_CV\_HANDLER()函数，比较两个编译变量是否相同，“**$a===$b”或“$b===$a”**。

#### **ZEND\_VM\_SMART\_BRANCH()宏程序**

ZEND\_VM\_SMART\_BRANCH()宏程序用于根据传入的参数，跳转到不同操作码，代码如下：

|  |
| --- |
| // 根据当前操作码结果选择下一个操作码，p1:传入值，主要用来做判断。p2:是否检查异常  #define ZEND\_VM\_SMART\_BRANCH(\_result, \_check) do { \  if ((\_check) && UNEXPECTED(EG(exception))) { /\* 如果要检查异常并且有异常 \*/ \  OPLINE = EX(opline); /\* 使用旧操作码 \*/ \  /\* 操作码结果类型为 (IS\_SMART\_BRANCH\_JMPZ|IS\_TMP\_VAR) \*/ \  }else if (EXPECTED(opline->result\_type==(IS\_SMART\_BRANCH\_JMPZ|IS\_TMP\_VAR))){\  if (\_result) { /\* 传入值为真 \*/ \  ZEND\_VM\_SET\_NEXT\_OPCODE(opline + 2); /\* 当前操作码后移2个 \*/ \  } else { /\* 传入值不为真 \*/ \  /\* 修改当前操作码并返回。EX(opline) = p1 \*/ \  /\* 后移一段并返回操作码，(zend\_op\*)((char\*)p1 + p2.jmp\_offset) \*/ \  ZEND\_VM\_SET\_OPCODE(OP\_JMP\_ADDR(opline + 1, (opline+1)->op2)); \  } \  /\* 操作码结果类型为 (IS\_SMART\_BRANCH\_JMPNZ|IS\_TMP\_VAR) \*/ \  } else if (EXPECTED(opline->result\_type ==  (IS\_SMART\_BRANCH\_JMPNZ|IS\_TMP\_VAR))) { \  if (!(\_result)) { /\* 传入值不为真 \*/ \  ZEND\_VM\_SET\_NEXT\_OPCODE(opline + 2); /\* 当前操作码后移2个 \*/ \  } else { /\* 传入值为真 \*/ \  /\* 修改当前操作码并返回。EX(opline) = p1 \*/ \  /\* 后移一段并返回操作码，(zend\_op\*)((char\*)p1 + p2.jmp\_offset) \*/ \  ZEND\_VM\_SET\_OPCODE(OP\_JMP\_ADDR(opline + 1, (opline+1)->op2)); \  } \  } else { /\* 其他情况 \*/ \  ZVAL\_BOOL(EX\_VAR(opline->result.var), \_result); /\* 结果变量赋值 布尔型 \*/ \  ZEND\_VM\_SET\_NEXT\_OPCODE(opline + 1); /\* 当前操作码后移1个 \*/ \  } \  ZEND\_VM\_CONTINUE(); /\* windows中: return 0 \*/ \  } while (0) |

如上所示，这个宏程序比较复杂，业务逻辑可归纳为如下几种情况：

1）如果需要检查异常，并且有异常存在，使用原操作码；

2）如果操作码结果类型为 (IS\_SMART\_BRANCH\_JMPZ|IS\_TMP\_VAR)：结果为真时后移2个操作码；否则通过OP\_JMP\_ADDR()宏程序使用下一个操作码，计算出跳转位置，并跳转到指定操作码；

3）如果操作码结果类型为（IS\_SMART\_BRANCH\_JMPNZ|IS\_TMP\_VAR）：结果不为真是时后移2个操作码；否则通过OP\_JMP\_ADDR()宏程序使用下一个操作码，计算出跳转位置，并跳转到指定操作码；

4）其他情况：把结果存放到结果变量中，操作码后移1个。

相关宏程序定义如下：

|  |
| --- |
| #define ZEND\_VM\_SET\_NEXT\_OPCODE(new\_op) \  CHECK\_SYMBOL\_TABLES() /\* 调试用 \*/ \  OPLINE = new\_op  // 调试用，正式环境无业务逻辑  CHECK\_SYMBOL\_TABLES()  // 修改当前操作码并返回。EX(opline) = p1  #define ZEND\_VM\_SET\_OPCODE(new\_op) \  CHECK\_SYMBOL\_TABLES() /\* 调试用 \*/ \  OPLINE = new\_op; /\* EX(opline) = new\_op \*/ \  ZEND\_VM\_INTERRUPT\_CHECK() /\* 打断并返回 \*/  #define ZEND\_VM\_INTERRUPT\_CHECK() do { \  /\* 返回 p1->value ，布尔值 \*/ \  if (UNEXPECTED(zend\_atomic\_bool\_load\_ex(&EG(vm\_interrupt)))) { \  /\* ZEND\_VM\_TAIL\_CALL(zend\_interrupt\_helper\_SPEC(execute\_data)) \*/ \  ZEND\_VM\_INTERRUPT(); /\* 牵扯太多其他逻辑 \*/ \  } \  } while (0)  // 调用zend\_interrupt\_helper\_SPEC()函数并返回  #define ZEND\_VM\_INTERRUPT() ZEND\_VM\_TAIL\_CALL(  zend\_interrupt\_helper\_SPEC(ZEND\_OPCODE\_HANDLER\_ARGS\_PASSTHRU)); |

如上所示，宏程序业务逻辑都比较简单，除了zend\_atomic\_bool\_load\_ex()函数。

zend\_atomic\_bool\_load\_ex()函数根据运行环境不同，定义了多个分支，在64位windows操作系统中，zend\_atomic\_bool\_load\_ex()函数调用操作系统中的InterlockedOr8()函数进行处理：

|  |
| --- |
| static zend\_always\_inline bool zend\_atomic\_bool\_load\_ex(zend\_atomic\_bool \*obj) {  return InterlockedOr8(&obj->value, false);  } |

如上所示，实际上是把obj->value转成布尔型并返回。关于zend\_atomic的详细介绍暂时略过。

## 二）不相同（ZEND\_IS\_NOT\_IDENTICAL）

### 在编译时计算

相同运算操作码为ZEND\_IS\_NOT\_IDENTICAL，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数is\_not\_identical\_function()，尝试进行处理。is\_not\_identical\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL is\_not\_identical\_function(zval \*result, zval \*op1, zval \*op2) {  ZVAL\_BOOL(result, !zend\_is\_identical(op1, op2));  return SUCCESS;  } |

如上所示，仍然是调用zend\_is\_identical()函数进行处理。

### “不相同”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_IS\_NOT\_IDENTICAL操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(17, ZEND\_IS\_NOT\_IDENTICAL, CONST|TMP|VAR|CV, CONST|TMP|VAR|CV, SPEC(COMMUTATIVE)) {  ... // 取出两个操作对象的变量  result = fast\_is\_not\_identical\_function(op1, op2); // 检查是否 不相同  FREE\_OP1(); // 释放操作对象的附加变量  FREE\_OP2(); // 释放操作对象的附加变量  // 根据当前操作码结果选择下一个操作码，p1:传入值，主要用来做判断。p2:是否检查异常  ZEND\_VM\_SMART\_BRANCH(result, 1);  } |

如上所示，由于添加了COMMUTATIVE（操作对象可交换）规则，此伪函数会根据两个操作对象的变量类型，被转换成10个处理器函数，这些处理器函数与“相同”操作码类似，例如：

ZEND\_IS\_NOT\_IDENTICAL\_SPEC\_CONST\_CONST\_HANDLER()函数，比较两个常量是否相同，例如**“1!==1”**。

## 三）相等（ZEND\_IS\_EQUAL）

### 在编译时计算

相同运算操作码为ZEND\_IS\_EQUAL，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数is\_equal\_function()，尝试进行处理。is\_equal\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL is\_equal\_function(zval \*result, zval \*op1, zval \*op2) {  ZVAL\_BOOL(result, zend\_compare(op1, op2) == 0);  return SUCCESS;  } |

zend\_compare()函数用于比较两个变量是否相等，函数的代码量比较大，请在附录中查看。

### “相等”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_IS\_EQUAL操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(18, ZEND\_IS\_EQUAL, CONST|TMPVAR|CV, CONST|TMPVAR|CV, SPEC(SMART\_BRANCH,COMMUTATIVE)) {  ... // 取出两个操作对象的变量  // 两个常量  if (ZEND\_VM\_SPEC && OP1\_TYPE == IS\_CONST && OP2\_TYPE == IS\_CONST) {  /\* pass \*/  } else if (EXPECTED(Z\_TYPE\_P(op1) == IS\_LONG)) { // 否则 如果op1类型是 整数  if (EXPECTED(Z\_TYPE\_P(op2) == IS\_LONG)) { // 如果op2类型是 整数  if (EXPECTED(Z\_LVAL\_P(op1) == Z\_LVAL\_P(op2))) { // 如果op1两个整数相等  ZEND\_VM\_C\_LABEL(is\_equal\_true): // 创建跳转标签is\_equal\_true，正确时跳转到这里  **ZEND\_VM\_SMART\_BRANCH\_TRUE(); // 条件为真时的智能分支**  } else { // 否则  ZEND\_VM\_C\_LABEL(is\_equal\_false): // 创建跳转标签is\_equal\_false，错误时跳转到这里  **ZEND\_VM\_SMART\_BRANCH\_FALSE(); // 条件为假时的智能分支**  }  } else if (EXPECTED(Z\_TYPE\_P(op2) == IS\_DOUBLE)) { // 如果op2类型是 小数  d1 = (double)Z\_LVAL\_P(op1); // op1转成小数  d2 = Z\_DVAL\_P(op2); // 取出op2的小数  ZEND\_VM\_C\_GOTO(is\_equal\_double); // 按小数进行比较  }  } else if (EXPECTED(Z\_TYPE\_P(op1) == IS\_DOUBLE)) { // 否则 如果op1类型是 小数  if (EXPECTED(Z\_TYPE\_P(op2) == IS\_DOUBLE)) { // 如果类型是 小数  d1 = Z\_DVAL\_P(op1); // 取出小数  d2 = Z\_DVAL\_P(op2); // 取出小数  ZEND\_VM\_C\_LABEL(is\_equal\_double): // 创建跳转标签 is\_equal\_double  if (d1 == d2) { // 如果两个小数相等  ZEND\_VM\_C\_GOTO(is\_equal\_true); // 跳转到指定标签 is\_equal\_true  } else { // 否则  ZEND\_VM\_C\_GOTO(is\_equal\_false); // 跳转到指定标签 is\_equal\_false  }  } else if (EXPECTED(Z\_TYPE\_P(op2) == IS\_LONG)) { // 否则 如果类型是 整数  d1 = Z\_DVAL\_P(op1); // 取出小数  d2 = (double)Z\_LVAL\_P(op2); // op2转成小数  ZEND\_VM\_C\_GOTO(is\_equal\_double); // 跳转到指定标签 is\_equal\_double  }  } else if (EXPECTED(Z\_TYPE\_P(op1) == IS\_STRING)) { // 否则 如果op1类型是 字串  if (EXPECTED(Z\_TYPE\_P(op2) == IS\_STRING)) { // 如果op2类型是 字串  // 比较两个字串相等  bool result = zend\_fast\_equal\_strings(Z\_STR\_P(op1), Z\_STR\_P(op2));  if (OP1\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op1 类型 是变量或临时变量  zval\_ptr\_dtor\_str(op1); // 通过指针销毁 zval（字串）  }  if (OP2\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op2 类型 是变量或临时变量  zval\_ptr\_dtor\_str(op2); // 通过指针销毁 zval（字串）  }  if (result) {  ZEND\_VM\_C\_GOTO(is\_equal\_true); // 跳转到指定标签 is\_equal\_true  } else { // 否则  ZEND\_VM\_C\_GOTO(is\_equal\_false); // 跳转到指定标签 is\_equal\_false  }  }  }  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_is\_equal\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，由于添加了智能分支规则（SMART\_BRANCH），此伪函数会根据两个操作对象的变量类型，被转换成16个处理器函数：

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **后缀1** | **后缀2** | **后缀3** |
| CONST\_CONST | \_HANDLER |  |  |
| TMPVAR\_CONST | \_HANDLER | \_JMPZ\_HANDLER | \_JMPNZ\_HANDLER |
| TMPVAR\_TMPVAR | \_HANDLER | \_JMPZ\_HANDLER | \_JMPNZ\_HANDLER |
| CV\_CONST | \_HANDLER | \_JMPZ\_HANDLER | \_JMPNZ\_HANDLER |
| CV\_TMPVAR | \_HANDLER | \_JMPZ\_HANDLER | \_JMPNZ\_HANDLER |
| CV\_CV | \_HANDLER | \_JMPZ\_HANDLER | \_JMPNZ\_HANDLER |

如上表所示，除CONST\_CONST类型外，每种类型组合都支持3种后缀，以编译变量和常量的比较操作为例：

1）ZEND\_IS\_EQUAL\_SPEC\_CV\_CONST\_HANDLER()函数，用于没有条件判断的语句，例如：**“echo $a==1;”**；

2）ZEND\_IS\_EQUAL\_SPEC\_CV\_CONST\_JMPZ\_HANDLER()，用于if等条件判断语句中，例如：**“if($a==1) ...”**；

3）ZEND\_IS\_EQUAL\_SPEC\_CV\_CONST\_JMPNZ\_HANDLER()，用于while等条件判断语句中，例如**“while($a==1) ...”**。

关于类型的组合前文中已有多个示例，暂不缀述。关于从条件判断到比较操作码的运行流程，会在条件判断相关章节详细介绍。

在以上3个函数中，差异只在在对ZEND\_VM\_SMART\_BRANCH\_TRUE()和ZEND\_VM\_SMART\_BRANCH\_FALSE()这两个伪函数的解析上，这两个伪函数已经在前文的伪码中标出。

ZEND\_IS\_EQUAL\_SPEC\_CV\_CONST\_HANDLER()函数对这两个伪函数的解析如下：

|  |
| --- |
| ...  if (EXPECTED(Z\_LVAL\_P(op1) == Z\_LVAL\_P(op2))) {  is\_equal\_true:  **ZEND\_VM\_SMART\_BRANCH\_TRUE\_NONE(); // 条件为真时的智能分支**  } else {  is\_equal\_false:  **ZEND\_VM\_SMART\_BRANCH\_FALSE\_NONE(); // 条件为假时的智能分支**  }  ... |

ZEND\_IS\_EQUAL\_SPEC\_CV\_CONST\_JMPZ\_HANDLER()函数对这两个伪函数的解析如下：

|  |
| --- |
| ...  if (EXPECTED(Z\_LVAL\_P(op1) == Z\_LVAL\_P(op2))) {  is\_equal\_true:  **ZEND\_VM\_SMART\_BRANCH\_TRUE\_JMPZ(); // 条件为真时的智能分支**  } else {  is\_equal\_false:  **ZEND\_VM\_SMART\_BRANCH\_FALSE\_JMPZ(); // 条件为假时的智能分支**  }  ... |

ZEND\_IS\_EQUAL\_SPEC\_CV\_CONST\_JMPNZ\_HANDLER()函数对这两个伪函数的解析如下：

|  |
| --- |
| ...  if (EXPECTED(Z\_LVAL\_P(op1) == Z\_LVAL\_P(op2))) {  is\_equal\_true:  **ZEND\_VM\_SMART\_BRANCH\_TRUE\_JMPNZ(); // 条件为真时的智能分支**  } else {  is\_equal\_false:  **ZEND\_VM\_SMART\_BRANCH\_FALSE\_JMPNZ(); // 条件为假时的智能分支**  }  ... |

以上智能分支中的宏程序定义如下：

|  |
| --- |
| // 当前操作码后移2个, 并return 0;  #define ZEND\_VM\_SMART\_BRANCH\_TRUE\_JMPZ() do { \  ZEND\_VM\_SET\_NEXT\_OPCODE(opline + 2); /\* 当前操作码后移2个 \*/ \  ZEND\_VM\_CONTINUE(); /\* windows系统: return 0 \*/ \  } while (0)  // 跳转到目标操作码，并return 0;  #define ZEND\_VM\_SMART\_BRANCH\_TRUE\_JMPNZ() do { \  /\* 修改当前操作码并返回。EX(opline) = p1 \*/ \  /\* 后移一段并返回操作码，(zend\_op\*)((char\*)p1 + p2.jmp\_offset) \*/ \  ZEND\_VM\_SET\_OPCODE(OP\_JMP\_ADDR(opline + 1, (opline+1)->op2)); \  ZEND\_VM\_CONTINUE(); /\* windows系统: return 0 \*/ \  } while (0)  // 操作码结果赋值为布true，并选择下一个操作码  #define ZEND\_VM\_SMART\_BRANCH\_TRUE\_NONE() do { \  ZVAL\_TRUE(EX\_VAR(opline->result.var)); /\* 给操作码结果 赋值 true \*/ \  ZEND\_VM\_NEXT\_OPCODE(); /\* opline向右偏移1个位置，到目标操作码并 return \*/ \  } while (0)  // 跳转到目标操作码，并return 0;  #define ZEND\_VM\_SMART\_BRANCH\_FALSE\_JMPZ() do { \  /\* 修改当前操作码并返回。EX(opline) = p1 \*/ \  /\* 后移一段并返回操作码，(zend\_op\*)((char\*)p1 + p2.jmp\_offset) \*/ \  ZEND\_VM\_SET\_OPCODE(OP\_JMP\_ADDR(opline + 1, (opline+1)->op2)); \  ZEND\_VM\_CONTINUE(); /\* windows系统: return 0 \*/ \  } while (0)  // 当前操作码后移2个, 并return 0;  #define ZEND\_VM\_SMART\_BRANCH\_FALSE\_JMPNZ() do { \  ZEND\_VM\_SET\_NEXT\_OPCODE(opline + 2); /\* 当前操作码后移2个 \*/ \  ZEND\_VM\_CONTINUE(); /\* windows系统: return 0 \*/ \  } while (0)  // 操作码结果赋值为布false，并选择下一个操作码  #define ZEND\_VM\_SMART\_BRANCH\_FALSE\_NONE() do { \  /\* 修改当前操作码并返回。EX(opline) = p1 \*/ \  /\* 后移一段并返回操作码，(zend\_op\*)((char\*)p1 + p2.jmp\_offset) \*/ \  ZVAL\_FALSE(EX\_VAR(opline->result.var)); \  ZEND\_VM\_NEXT\_OPCODE(); /\* opline向右偏移1个位置，到目标操作码并 return \*/ \  } while (0) |

ZEND\_IS\_EQUAL操作码处理器的业务逻辑中先针对整数，小数，字串3种类型做了一些优化，如果不匹配这些类型，调用zend\_is\_equal\_helper\_SPEC()函数进行处理，函数定义如下：

|  |
| --- |
| static zend\_never\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_is\_equal\_helper\_SPEC(zval \*op\_1, zval \*op\_2 ZEND\_OPCODE\_HANDLER\_ARGS\_DC) {  ... // 取出op1,op2  ret = zend\_compare(op\_1, op\_2);  ... // 销毁op1,op2  ZEND\_VM\_SMART\_BRANCH(ret == 0, 1);  } |

如上所示，如果不满足优化条件，最后仍然是调用zend\_compare()函数进行比较。

## 四）不相等（ZEND\_IS\_NOT\_EQUAL）

### 在编译时计算

相同运算操作码为ZEND\_IS\_NOT\_EQUAL，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数is\_not\_equal\_function()，尝试进行处理。is\_not\_equal\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL is\_not\_equal\_function(zval \*result, zval \*op1, zval \*op2) {  ZVAL\_BOOL(result, (zend\_compare(op1, op2) != 0));  return SUCCESS;  } |

如上所示，仍然是调用zend\_compare()函数进行处理。

### “不相同”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_IS\_NOT\_EQUAL操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(19, ZEND\_IS\_NOT\_EQUAL, CONST|TMPVAR|CV, CONST|TMPVAR|CV, SPEC(SMART\_BRANCH,COMMUTATIVE)) {  ... // 针对类型的优化业务逻辑  // 调用方法 zend\_is\_not\_equal\_helper  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_is\_not\_equal\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，由于添加了智能分支规则（SMART\_BRANCH），此伪函数会根据两个操作对象的变量类型，被转换成16个处理器函数，生成规则与ZEND\_IS\_EQUAL操作码相同。这些处理器函数与ZEND\_IS\_EQUAL操作码类似，以编译变量和常量的比较操作为例：

1）ZEND\_IS\_NOT\_EQUAL\_SPEC\_CV\_CONST\_HANDLER()函数，用于没有条件判断的语句，例如：**“echo $a!=1;”**；

2）ZEND\_IS\_NOT\_EQUAL\_SPEC\_CV\_CONST\_JMPZ\_HANDLER()，用于if等条件判断语句中，例如：**“if($a!=1) ...”**；

3）ZEND\_IS\_NOT\_EQUAL\_SPEC\_CV\_CONST\_JMPNZ\_HANDLER()，用于while等条件判断语句中，例如**“while($a!=1) ...”**。

操作码业务逻辑中针对不同类型的优化也与ZEND\_IS\_EQUAL操作码相似，当不满足条件时调用zend\_is\_not\_equal\_helper\_SPEC()函数进行处理，函数定义如下：

|  |
| --- |
| static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_is\_not\_equal\_helper\_SPEC(zval \*op\_1, zval \*op\_2 ZEND\_OPCODE\_HANDLER\_ARGS\_DC) {  ... // 取出op1,op2  ret = zend\_compare(op\_1, op\_2);  ... // 销毁op1,op2  ZEND\_VM\_SMART\_BRANCH(ret != 0, 1);  } |

## 五）小于（ZEND\_IS\_SMALLER）

### 在编译时计算

相同运算操作码为ZEND\_IS\_SMALLER，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数is\_smaller\_function()，尝试进行处理。is\_smaller\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL is\_smaller\_function(zval \*result, zval \*op1, zval \*op2) {  ZVAL\_BOOL(result, (zend\_compare(op1, op2) < 0));  return SUCCESS;  } |

如上所示，仍然是调用zend\_compare()函数进行处理。

### “不相同”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_IS\_SMALLER操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HOT\_NOCONSTCONST\_HANDLER(20, ZEND\_IS\_SMALLER, CONST|TMPVARCV, CONST|TMPVARCV, SPEC(SMART\_BRANCH)) {  ... // 取出op1,op2的变量  // op1或op2有一个不是ANY 并且两个都是常量  if (ZEND\_VM\_SPEC && OP1\_TYPE == IS\_CONST && OP2\_TYPE == IS\_CONST) {  /\* pass \*/  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op1) == IS\_LONG)) { // 如果op1是整数  if (EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_LONG)) { // 如果op2是整数  if (EXPECTED(Z\_LVAL\_P(op1) < Z\_LVAL\_P(op2))) { // 整数直接比较大小  ZEND\_VM\_C\_LABEL(is\_smaller\_true): // 创建跳转标签 is\_smaller\_true  // 根据当前操作码结果选择下一个操作码。如果不是跳转类型，结果赋值为true。  ZEND\_VM\_SMART\_BRANCH\_TRUE();  } else { // 否则  ZEND\_VM\_C\_LABEL(is\_smaller\_false): // 创建跳转标签 is\_smaller\_false  // 根据当前操作码结果选择下一个操作码。如果不是跳转类型，结果赋值为false。  ZEND\_VM\_SMART\_BRANCH\_FALSE();  }  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_DOUBLE)) { // 如果op2是小数  d1 = (double)Z\_LVAL\_P(op1); // op1转成小数  d2 = Z\_DVAL\_P(op2); // 取出小数  ZEND\_VM\_C\_GOTO(is\_smaller\_double); // 跳转到指定标签 is\_smaller\_double  }  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op1) == IS\_DOUBLE)) { // 如果 op1是小数  if (EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_DOUBLE)) { // 如果 op2是小数  d1 = Z\_DVAL\_P(op1); // 取出小数  d2 = Z\_DVAL\_P(op2); // 取出小数  ZEND\_VM\_C\_LABEL(is\_smaller\_double): // 创建跳转标签 is\_smaller\_double  if (d1 < d2) { // 如果op2大  ZEND\_VM\_C\_GOTO(is\_smaller\_true); // 跳转到指定标签 is\_smaller\_true  } else { // 否则  ZEND\_VM\_C\_GOTO(is\_smaller\_false); // 跳转到指定标签 is\_smaller\_false  }  } else if (EXPECTED(Z\_TYPE\_INFO\_P(op2) == IS\_LONG)) { // 如果op2是整数  d1 = Z\_DVAL\_P(op1); // op1取出小数  d2 = (double)Z\_LVAL\_P(op2); // op2转成小数  ZEND\_VM\_C\_GOTO(is\_smaller\_double); // 跳转到指定标签 is\_smaller\_double  }  }  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_is\_smaller\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，由于添加了智能分支规则（SMART\_BRANCH），此伪函数会根据两个操作对象的变量类型，被转换成10个处理器函数：

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **后缀1** | **后缀2** | **后缀3** |
| CONST\_CONST | \_HANDLER |  |  |
| CONST\_TMPVARCV | \_HANDLER | \_JMPZ\_HANDLER | \_JMPNZ\_HANDLER |
| TMPVARCV\_CONST | \_HANDLER | \_JMPZ\_HANDLER | \_JMPNZ\_HANDLER |
| TMPVARCV\_TMPVARCV | \_HANDLER | \_JMPZ\_HANDLER | \_JMPNZ\_HANDLER |

如上表所示，除CONST\_CONST类型外，每种类型组合都支持3种后缀，以编译变量和常量的比较操作为例：

1）ZEND\_IS\_SMALLER\_SPEC\_TMPVARCV\_CONST\_HANDLER()函数，用于没有条件判断的语句，例如：**“echo $a<1;”**；

2）ZEND\_IS\_SMALLER\_SPEC\_TMPVARCV\_CONST\_JMPZ\_HANDLER()，用于if等条件判断语句中，例如：**“if($a<1) ...”**；

3）ZEND\_IS\_SMALLER\_SPEC\_TMPVARCV\_CONST\_JMPNZ\_HANDLER()，用于while等条件判断语句中，例如**“while($a<1) ...”**。

关于类型的组合前文中已有多个示例，暂不缀述。关于从条件判断到比较操作码的运行流程，会在条件判断相关章节详细介绍。

ZEND\_IS\_EQUAL操作码处理器的业务逻辑中先针对整数，小数，2种类型做了一些优化，如果不匹配这些类型，调用zend\_is\_smaller\_helper\_SPEC()函数进行处理，函数定义如下：

|  |
| --- |
| static zend\_never\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_is\_smaller\_helper\_SPEC(zval \*op\_1, zval \*op\_2 ZEND\_OPCODE\_HANDLER\_ARGS\_DC){  ... // 取出op1,op2  ret = zend\_compare(op\_1, op\_2);  ... // 销毁op1,op2  ZEND\_VM\_SMART\_BRANCH(ret < 0, 1);  } |

## 六）小于等于（ZEND\_IS\_SMALLER\_OR\_EQUAL）

### 在编译时计算

相同运算操作码为ZEND\_IS\_SMALLER\_OR\_EQUAL，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数is\_smaller\_or\_equal\_function()，尝试进行处理。is\_smaller\_or\_equal\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL is\_smaller\_or\_equal\_function(zval \*result, zval \*op1, zval \*op2) {  ZVAL\_BOOL(result, (zend\_compare(op1, op2) <= 0));  return SUCCESS;  } |

如上所示，仍然是调用zend\_compare()函数进行处理。

### “小于等于”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_IS\_SMALLER\_OR\_EQUAL操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HOT\_NOCONSTCONST\_HANDLER(21, ZEND\_IS\_SMALLER\_OR\_EQUAL, CONST|TMPVARCV, CONST|TMPVARCV, SPEC(SMART\_BRANCH)) {  ... // 针对类型的优化业务逻辑  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_is\_smaller\_or\_equal\_helper, op\_1, op1, op\_2, op2);  } |

如上所示，由于添加了智能分支规则（SMART\_BRANCH），此伪函数会根据两个操作对象的变量类型，被转换成10个处理器函数，生成规则与ZEND\_IS\_SMALLER操作码相同。这些处理器函数与ZEND\_IS\_SMALLER操作码类似，以编译变量和常量的比较操作为例：

1）ZEND\_IS\_SMALLER\_OR\_EQUAL\_SPEC\_TMPVARCV\_CONST\_HANDLER()函数，用于没有条件判断的语句，例如：**“echo $a<=1;”**；

2）ZEND\_IS\_SMALLER\_OR\_EQUAL\_SPEC\_TMPVARCV\_CONST\_JMPZ\_HANDLER()，用于if等条件判断语句中，例如：**“if($a<=1) ...”**；

3）ZEND\_IS\_SMALLER\_OR\_EQUAL\_SPEC\_TMPVARCV\_CONST\_JMPNZ\_HANDLER()，用于while等条件判断语句中，例如**“while($a<=1) ...”**。

操作码业务逻辑中针对不同类型的优化也与ZEND\_IS\_SMALLER操作码相似，当不满足条件时调用zend\_is\_smaller\_or\_equal\_helper\_SPEC()函数进行处理，函数定义如下：

|  |
| --- |
| static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_is\_not\_equal\_helper\_SPEC(zval \*op\_1, zval \*op\_2 ZEND\_OPCODE\_HANDLER\_ARGS\_DC) {  ... // 取出op1,op2  ret = zend\_compare(op\_1, op\_2);  ... // 销毁op1,op2  ZEND\_VM\_SMART\_BRANCH(ret <= 0, 1);  } |

## 七）太空船（ZEND\_SPACESHIP）

太空船运算符“<=>”，在比较两个变量时如果左侧变量小于左侧变量，返回-1；如果两侧变量相等，返回0；如果左侧变量大于右侧变量，返回1。

### 在编译时计算

相同运算操作码为ZEND\_SPACESHIP，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数compare\_function()，尝试进行处理。

### “太空船”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_SPACESHIP操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(170, ZEND\_SPACESHIP, CONST|TMPVAR|CV, CONST|TMPVAR|CV){  .. // 取出两个操作对象的变量  compare\_function(EX\_VAR(opline->result.var), op1, op2); // 比较两个变量  .. // 取出两个操作对象的变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION();  } |

如上所示，，此伪函数会根据两个操作对象的变量类型，被转换成9个处理器函数：

1）ZEND\_SPACESHIP\_SPEC\_CONST\_CONST\_HANDLER()函数，处理两个常量相减，例如**“1 <=> 1”**，但这种情况有可能已经在编译时处理完了；

2）ZEND\_SPACESHIP\_SPEC\_CONST\_TMPVAR\_HANDLER()函数，处理一个常量和一个普通（或临时）变量相除，如**“1 <=> ($a+1)”**（表达式“($a+1)”会转成一个临时变量）；

3）ZEND\_SPACESHIP\_SPEC\_CONST\_CV\_HANDLER()函数，处理一个常量和一个编译变量相除，如**“1 <=> $a”**（$a会转成一个编译变量）；

4）ZEND\_SPACESHIP\_SPEC\_TMPVAR\_CONST\_HANDLER()函数，处理一个普通（或临时）变量和一个常量相除，如**“($a+1) <=> 1”**；

5）ZEND\_SPACESHIP\_SPEC\_TMPVAR\_TMPVAR\_HANDLER()函数，处理两个普通（或临时）变量相除，例如**“$GLOBALS['a'] <=> $GLOBALS['a'];”**（$GLOBALS['a']会转成一个普通变量）；

6）ZEND\_SPACESHIP\_SPEC\_TMPVAR\_CV\_HANDLER()函数，例如**“($a+1) <=> $a”**；

7）ZEND\_SPACESHIP\_SPEC\_CV\_CONST\_HANDLER()函数，例如**“$a <=> 1”**；

8）ZEND\_SPACESHIP\_SPEC\_CV\_TMPVAR\_HANDLER()函数，例如**“$a <=> ($a+1)”**；

9）ZEND\_SPACESHIP\_SPEC\_CV\_CV\_HANDLER()函数，例如“**$a <=> $a”**。

操作码业务逻辑中针对不同类型的优化也与ZEND\_IS\_SMALLER操作码相似，当不满足条件时调用zend\_is\_smaller\_or\_equal\_helper\_SPEC()函数进行处理，函数定义如下：

|  |
| --- |
| static ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_is\_not\_equal\_helper\_SPEC(zval \*op\_1, zval \*op\_2 ZEND\_OPCODE\_HANDLER\_ARGS\_DC) {  ... // 取出op1,op2  ret = zend\_compare(op\_1, op\_2);  ... // 销毁op1,op2  ZEND\_VM\_SMART\_BRANCH(ret <= 0, 1);  } |

## 八）大于和大于等于

编译“大于”和“大于等于”表达式和编译前面的表达式一样，都要用到zend\_compile\_expr\_inner()函数，调用路径如下：

|  |
| --- |
| zend\_compile\_expr()  ->zend\_compile\_expr\_inner()  ->zend\_compile\_greater() |

函数中相关业务逻辑如下：

|  |
| --- |
| static void zend\_compile\_expr\_inner(znode \*result, zend\_ast \*ast) {  ...  switch (ast->kind) { // 根据类型处理  ...  case ZEND\_AST\_GREATER: // >, >=  case ZEND\_AST\_GREATER\_EQUAL:  zend\_compile\_greater(result, ast);  return;  ... |

zend\_compile\_greater()函数用于编译“大于”和“大于等于”表达式，代码如下：

|  |
| --- |
| static void zend\_compile\_greater(znode \*result, zend\_ast \*ast) {  ... // 先编译左右两个表达式  // 如果两结果都是常量  if (left\_node.op\_type == IS\_CONST && right\_node.op\_type == IS\_CONST) {  result->op\_type = IS\_CONST;  // 计算 >,>=  zend\_ct\_eval\_greater(&result->u.constant, ast->kind,  &left\_node.u.constant, &right\_node.u.constant);  ... // 销毁两个表达式的值  return;  }  // 创建临时操作码，**交换左右两个操作对象的位置**  zend\_emit\_op\_tmp(result, ast->kind == ZEND\_AST\_GREATER ? ZEND\_IS\_SMALLER : ZEND\_IS\_SMALLER\_OR\_EQUAL, &right\_node, &left\_node);  } |

如上所示，业务逻辑可归纳如下：

1）如果两个操作对象都是常量，调用zend\_ct\_eval\_greater()函数计算结果；

2）否则调用zend\_emit\_op\_tmp()函数创建操作码：进行大于运算时，**交换左右两个操作对象的位置，**创建ZEND\_IS\_SMALLER（小于）操作码，进行大于等于运算时，创建ZEND\_IS\_SMALLER\_OR\_EQUAL（小于等于）操作码。

大于和大于等于操作没有单独的操作码，创建小于和小于等于操作码，并使用它们的处理器进行操作。

zend\_emit\_op\_tmp()函数已经在前文中介绍过，详情参见“赋值操作-编译”章节。

zend\_ct\_eval\_greater()函数用于对常量进行大于和大于等于计算，代码如下：

|  |
| --- |
| static inline void zend\_ct\_eval\_greater(zval \*result, zend\_ast\_kind kind, zval \*op1, zval \*op2) {  binary\_op\_type fn = kind == ZEND\_AST\_GREATER  ? is\_smaller\_function : is\_smaller\_or\_equal\_function;  fn(result, op2, op1); **// 交换左右两个操作对象的位置**  } |

如上所示，交换左右两个操作对象的位置，使用is\_smaller\_function()函数或is\_smaller\_or\_equal\_function()函数进行计算。

# 五、其他二元运算

其他二元运算符包含连接（ZEND\_CONCAT）、快速连接（ZEND\_FAST\_CONCAT）和布尔抑或（ZEND\_BOOL\_XOR）运算。

## 一）连接（ZEND\_CONCAT）

### 在编译时计算

连接运算操作码为ZEND\_CONCAT，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数concat\_function()，尝试进行处理。concat\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL concat\_function(zval \*result, zval \*op1, zval \*op2) {  zval \*orig\_op1 = op1;  zval op1\_copy, op2\_copy;  ZVAL\_UNDEF(&op1\_copy); // 准备两个临时变量  ZVAL\_UNDEF(&op2\_copy);  do {  if (UNEXPECTED(Z\_TYPE\_P(op1) != IS\_STRING)) { // op1 不是字串  if (Z\_ISREF\_P(op1)) { // op1 是引用类型  op1 = Z\_REFVAL\_P(op1); // 解引用  if (Z\_TYPE\_P(op1) == IS\_STRING) break; // 如果是字串，op1处理完毕，跳出do  }  // 尝试调用op1 和 op2的 do\_operation 方法，如果成功直接返回  ZEND\_TRY\_BINARY\_OBJECT\_OPERATION(ZEND\_CONCAT);  ZVAL\_STR(&op1\_copy, zval\_get\_string\_func(op1)); // zval 转成 字串, 没有try  if (UNEXPECTED(EG(exception))) { // 如果有异常  zval\_ptr\_dtor\_str(&op1\_copy); // 销毁临时变量  if (orig\_op1 != result) { // 如果不是用 op1 接收结果  ZVAL\_UNDEF(result); // 清空接收变量，中断操作  }  return FAILURE; // 返回失败  }  if (result == op1) { // 如果没有异常，并且用op1接收结果  if (UNEXPECTED(op1 == op2)) { // 如果两个操作对象相同  op2 = &op1\_copy; // op2 指向op1的副本  }  }  op1 = &op1\_copy; // op1 指向副本  }  } while (0);  do { // 这个do是为了break;  **... // 针对op2的优化与上面对op1的优化类似**  } while (0);  if (UNEXPECTED(Z\_STRLEN\_P(op1) == 0)) { // op1 长度为0  if (EXPECTED(result != op2)) { // 不是用op2接收结果  if (result == orig\_op1) { // 如果用原始的op1接收结果  i\_zval\_ptr\_dtor(result); // 清空结果  }  ZVAL\_COPY(result, op2); // 返回 处理后的op2  }  } else if (UNEXPECTED(Z\_STRLEN\_P(op2) == 0)) { // op2 长度为0  **... // 针对op2的优化与上面对op1的优化类似**  } else { // 两个都有长度  size\_t op1\_len = Z\_STRLEN\_P(op1); // 取得2个长度  size\_t op2\_len = Z\_STRLEN\_P(op2);  size\_t result\_len = op1\_len + op2\_len; // 长度和  zend\_string \*result\_str; // 结果字串  // 如果结果超长  if (UNEXPECTED(op1\_len > ZSTR\_MAX\_LEN - op2\_len)) {  // 报错：字串溢出  zend\_throw\_error(NULL, "String size overflow");  zval\_ptr\_dtor\_str(&op1\_copy); // 销毁2个副本  zval\_ptr\_dtor\_str(&op2\_copy);  if (orig\_op1 != result) { // 如果不是用原始op1接收结果  ZVAL\_UNDEF(result); // 清空接收变量，中断操作  }  return FAILURE; // 返回错误  }  if (result == op1 && Z\_REFCOUNTED\_P(result)) { // 如果用op1接收 并且 结果可计数  result\_str = zend\_string\_extend(Z\_STR\_P(result), result\_len, 0); // 调整结果字串大小  } else { // 不是用op1接收或结果 或 结果不可计数  result\_str = zend\_string\_alloc(result\_len, 0); // 重新分配结果字串  memcpy(ZSTR\_VAL(result\_str), Z\_STRVAL\_P(op1), op1\_len); // 把op1复制进来  if (result == orig\_op1) { // 如果用原op1接收结果  i\_zval\_ptr\_dtor(result); // 清空变量  }  }    ZVAL\_NEW\_STR(result, result\_str); // 把zend\_string 添加给 zval，不支持保留字  memcpy(ZSTR\_VAL(result\_str) + op1\_len, Z\_STRVAL\_P(op2), op2\_len); // 复制op2  ZSTR\_VAL(result\_str)[result\_len] = '\0'; // 最后加结束字符  }  zval\_ptr\_dtor\_str(&op1\_copy); // 销毁临时变量  zval\_ptr\_dtor\_str(&op2\_copy);  return SUCCESS; // 返回成功  } |

如上所示，根据op1和op2的类型，做了比较多的优化，但业务逻辑并不算复杂。

对字符串类型的更多介绍参见“类型篇”。

### “连接”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_CONCAT操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HANDLER(8, ZEND\_CONCAT, CONST|TMPVAR|CV, CONST|TMPVAR|CV, SPEC(NO\_CONST\_CONST)) {  ... // 取出两个操作对象的变量  // 如果两个类型都是：常量 或 字串  if ((OP1\_TYPE == IS\_CONST || EXPECTED(Z\_TYPE\_P(op1) == IS\_STRING)) &&  (OP2\_TYPE == IS\_CONST || EXPECTED(Z\_TYPE\_P(op2) == IS\_STRING))) {  zend\_string \*op1\_str = Z\_STR\_P(op1); // 取出两个字串  zend\_string \*op2\_str = Z\_STR\_P(op2);  zend\_string \*str;  // 如果 op1不是常量 并且字串长度是0。op1可忽略  if (OP1\_TYPE != IS\_CONST && UNEXPECTED(ZSTR\_LEN(op1\_str) == 0)) {  if (OP2\_TYPE == IS\_CONST || OP2\_TYPE == IS\_CV) { // op2是常量 或 编译变量  ZVAL\_STR\_COPY(EX\_VAR(opline->result.var), op2\_str); // op2的字串当成结果  } else { // 否则，op2类型是临时变量 TMPVAR  ZVAL\_STR(EX\_VAR(opline->result.var), op2\_str); // op2的字串当成结果  }  if (OP1\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op1 类型 是变量或临时变量  zend\_string\_release\_ex(op1\_str, 0); // 释放op1字串  }  // 如果 op2不是常量 并且字串长度是 0。op2可忽略  } else if (OP2\_TYPE != IS\_CONST && UNEXPECTED(ZSTR\_LEN(op2\_str) == 0)) {  if (OP1\_TYPE == IS\_CONST || OP1\_TYPE == IS\_CV) { // op1类型是常量 或 编译变量  ZVAL\_STR\_COPY(EX\_VAR(opline->result.var), op1\_str); // op1的字串当成结果  } else { // 否则,op1类型是临时变量 TMPVAR  ZVAL\_STR(EX\_VAR(opline->result.var), op1\_str);  }  if (OP2\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op2 类型 是变量或临时变量  zend\_string\_release\_ex(op2\_str, 0); // 释放op2字串  }  // op1,op2都不可忽略。如果op1不是常量和编译变量，也不是保留字，并且引用次数是1  } else if (OP1\_TYPE != IS\_CONST && OP1\_TYPE != IS\_CV &&  !ZSTR\_IS\_INTERNED(op1\_str) && GC\_REFCOUNT(op1\_str) == 1) {  size\_t len = ZSTR\_LEN(op1\_str); // 字串长度  // 如果 连接后长度会大于字串最大长度  if (UNEXPECTED(len > ZSTR\_MAX\_LEN - ZSTR\_LEN(op2\_str))) { // 报错，字串超长  zend\_error\_noreturn(E\_ERROR, "Integer overflow in memory allocation");  }  // 给op1的字串增加长度  str = zend\_string\_extend(op1\_str, len + ZSTR\_LEN(op2\_str), 0);  // 后面一段复制op2的内容  memcpy(ZSTR\_VAL(str) + len, ZSTR\_VAL(op2\_str), ZSTR\_LEN(op2\_str)+1);  ZVAL\_NEW\_STR(EX\_VAR(opline->result.var), str); //把字串添加给变量，不支持保留字  if (OP2\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op2 类型 是变量或临时变量  zend\_string\_release\_ex(op2\_str, 0); // 释放op2字串  }  } else { // 其他情况  // 创建新字串，长度是两个字串的和  str = zend\_string\_alloc(ZSTR\_LEN(op1\_str) + ZSTR\_LEN(op2\_str), 0);  // 把两个字串复制到新串里  memcpy(ZSTR\_VAL(str), ZSTR\_VAL(op1\_str), ZSTR\_LEN(op1\_str));  memcpy(ZSTR\_VAL(str) + ZSTR\_LEN(op1\_str), ZSTR\_VAL(op2\_str), ZSTR\_LEN(op2\_str)+1);  ZVAL\_NEW\_STR(EX\_VAR(opline->result.var), str); // 把字串作为结果，不支持保留字  if (OP1\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op1 类型 是变量或临时变量  zend\_string\_release\_ex(op1\_str, 0); // 释放op1字串  }  if (OP2\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op2 类型 是普通变量或临时变量  zend\_string\_release\_ex(op2\_str, 0); // 释放op2字串  }  }  ZEND\_VM\_NEXT\_OPCODE(); // 下一个操作码  } else { // 否则，op1,op2至少有一个既不是常量也不是字串  SAVE\_OPLINE(); // windows: 无操作  // 如果操作对象类型是编译变量 ，内置变量类型是 未定义  if (OP1\_TYPE == IS\_CV && UNEXPECTED(Z\_TYPE\_P(op1) == IS\_UNDEF)) {  op1 = ZVAL\_UNDEFINED\_OP1(); // 报错：p1的变量（变量名）未定义  }  // 如果 op2 是编译变量 并且 类型是 未定义  if (OP2\_TYPE == IS\_CV && UNEXPECTED(Z\_TYPE\_P(op2) == IS\_UNDEF)) {  op2 = ZVAL\_UNDEFINED\_OP2(); // 报错：p2的变量（变量名）未定义  }  concat\_function(EX\_VAR(opline->result.var), op1, op2); // 连接两个变量，并返回结果  FREE\_OP1(); // 释放操作对象的附加变量  FREE\_OP2();  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  }  } |

如上所示，因为处理字串的操作比较繁琐，所以代码量比较多，但业务逻辑并不算复杂。由于添加了禁用双常量（NO\_CONST\_CONST）扩展规则，此伪函数会根据两个操作对象的变量类型，被转换成8个处理器函数：

1）ZEND\_CONCAT\_SPEC\_CONST\_TMPVAR\_HANDLER()函数，处理一个常量和一个普通变量（或临时变量）的连接操作，例如**“'1'.$GLOBAL['a']”**；

2）ZEND\_CONCAT\_SPEC\_CONST\_CV\_HANDLER()函数，处理一个常量和一个编译变量的连接操作，例如**“'1'.$a”**；

3）ZEND\_CONCAT\_SPEC\_TMPVAR\_CONST\_HANDLER()函数，处理一个普通变量和一个常量的连接操作，例如**“$GLOBAL['a'].'1'”**；

4）ZEND\_CONCAT\_SPEC\_TMPVAR\_TMPVAR\_HANDLER()函数，处理两个普通变量（或临时变量）的连接操作，例如**“$GLOBAL['a'].$GLOBAL['a']”**；

5）ZEND\_CONCAT\_SPEC\_TMPVAR\_CV\_HANDLER()函数，处理一个普通变量（或临时变量）和一个编译变量的连接操作，例如**“$GLOBAL['a'].$a”**；

6）ZEND\_CONCAT\_SPEC\_CV\_CONST\_HANDLER()函数，处理一个编译变量和一个常量的连接操作，例如**“$a.'1'”**；

7）ZEND\_CONCAT\_SPEC\_CV\_TMPVAR\_HANDLER()函数，处理一个编译变量和一个普通变量（或临时变量）的连接操作，例如**“$a.$GLOBAL['a']”**；

8）ZEND\_CONCAT\_SPEC\_CV\_CV\_HANDLER()函数，处理两个编译变量的连接操作，例如**“$a.$a”**。

## 二）快速连接（ZEND\_FAST\_CONCAT）

### 针对ZEND\_CONCAT操作码的优化

ZEND\_FAST\_CONCAT操作码与ZEND\_CONCAT操作码在创建流程上略有不同，以一个简单的链接操作“$a.$b”为例，在通过zend\_compile\_binary\_op()函数创建操作码时，对ZEND\_CONCAT操作码进行了一些优化，代码如下：

|  |
| --- |
| static void zend\_compile\_binary\_op(znode \*result, zend\_ast \*ast) {  ...  } else if (opcode == ZEND\_CONCAT) {  if (left\_node.op\_type == IS\_CONST) { // 左侧是常量  if (Z\_TYPE(left\_node.u.constant) == IS\_ARRAY) { // 如果左侧是 array  // 创建类型转换操作码：ZEND\_CAST，返回结果类型是 string  zend\_emit\_op\_tmp(&left\_node, ZEND\_CAST, &left\_node, NULL)->extended\_value = IS\_STRING;  } else {  convert\_to\_string(&left\_node.u.constant); // 左侧不是 array，直接转换类型  }  }  if (right\_node.op\_type == IS\_CONST) { // 右侧，逻辑同上  ...  }  // 如果两侧类型都是常量  if (left\_node.op\_type == IS\_CONST && right\_node.op\_type == IS\_CONST) {  opcode = ZEND\_FAST\_CONCAT; // 操作码设置成：快速连接  }  }  zend\_emit\_op\_tmp(result, opcode, &left\_node, &right\_node); // 创建操作码  } while (0);  } |

如上所示，先检查两个操作码是否是常量，如果是常量，尝试转换成字符串。如果在创建操作码前恰巧两个操作对象都是常量，这时会把操作码从ZEND\_CONCAT替换成ZEND\_FAST\_CONCAT。

### 在字串中混合使用变量

#### **创建ZEND\_AST\_ENCAPS\_LIST语句**

不难看出，在以上优化中，要求两个操作对象都是常量，那么相当于只用到一种类型组合。并且如果两个操作对象都是常量，那么大分部情况下已经在编译时处理完毕了，不需要创建操作码。所以ZEND\_FAST\_CONCAT操作码的主要功能不是处理这种情况。

ZEND\_FAST\_CONCAT操作码比较广泛的应用场景是在计算字串中混合的变量，例如：

|  |
| --- |
| <?php  $a=$b=1;  echo "{$a}{$b}"; |

解析过程如下：

|  |
| --- |
| zend\_ast \* ast\_1 = zend\_ast\_create\_list(0, ZEND\_AST\_STMT\_LIST); /\* top\_statement\_list:[] \*/  **zend\_ast \* ast\_14 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("$a", 0)); /\* simple\_variable:["$a"] \*/**  zend\_ast \* ast\_13 = zend\_ast\_create(ZEND\_AST\_VAR, ast\_14); /\* callable\_variable:["ast\_14"] \*/  zend\_ast \* ast\_12 = ast\_13; /\* variable:["ast\_13"] \*/  **zend\_ast \* ast\_33 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("$b", 0)); /\* simple\_variable:["$b"] \*/**  zend\_ast \* ast\_32 = zend\_ast\_create(ZEND\_AST\_VAR, ast\_33); /\* callable\_variable:["ast\_33"] \*/  zend\_ast \* ast\_31 = ast\_32; /\* variable:["ast\_32"] \*/  **zend\_ast \* ast\_71 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("1", 0)); /\*** scalar:["1"] \*/  zend\_ast \* ast\_49 = ast\_71; /\* expr:["ast\_71"] \*/  **zend\_ast \* ast\_30 = zend\_ast\_create(ZEND\_AST\_ASSIGN, ast\_31, ast\_49); /\*** expr:["ast\_31","=","ast\_49"] \*/  **zend\_ast \* ast\_11 = zend\_ast\_create(ZEND\_AST\_ASSIGN, ast\_12, ast\_30); /\*** expr:["ast\_12","=","ast\_30"] \*/  zend\_ast \* ast\_4 = ast\_11; /\* statement:["ast\_11",";"] \*/  zend\_ast \* ast\_3 = ast\_4; /\* top\_statement:["ast\_4"] \*/  zend\_ast \* ast\_251 = zend\_ast\_list\_add(ast\_1, ast\_3); /\* top\_statement\_list:["ast\_1","ast\_3"] \*/  **zend\_ast \* ast\_146 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("$a", 0)); /\*** simple\_variable:["$a"] \*/  zend\_ast \* ast\_145 = zend\_ast\_create(ZEND\_AST\_VAR, ast\_146); /\* callable\_variable:["ast\_146"] \*/  zend\_ast \* ast\_144 = ast\_145; /\* variable:["ast\_145"] \*/  zend\_ast \* ast\_143 = ast\_144; /\* encaps\_var:["{","ast\_144","}"] \*/  **zend\_ast \* ast\_141 = zend\_ast\_create\_list(1, ZEND\_AST\_ENCAPS\_LIST, ast\_143); /\*** encaps\_list:["ast\_143"] \*/  **zend\_ast \* ast\_166 = zend\_ast\_create\_zval\_from\_str(ZSTR\_INIT\_LITERAL("$b", 0)); /\*** simple\_variable:["$b"] \*/  zend\_ast \* ast\_165 = zend\_ast\_create(ZEND\_AST\_VAR, ast\_166); /\* callable\_variable:["ast\_166"] \*/  zend\_ast \* ast\_164 = ast\_165; /\* variable:["ast\_165"] \*/  zend\_ast \* ast\_163 = ast\_164; /\* encaps\_var:["{","ast\_164","}"] \*/  **zend\_ast \* ast\_184 = zend\_ast\_list\_add(ast\_141, ast\_163); /\***  ...  CG(ast) = ast\_252; /\* start:["ast\_252"] \*/ |

如上所示，在解析过程中，创建了 ZEND\_AST\_ENCAPS\_LIST语句。

#### **编译ZEND\_AST\_ENCAPS\_LIST语句**

编译ZEND\_AST\_ENCAPS\_LIST语句需要用到zend\_compile\_encaps\_list()函数，调用路径如下：

|  |
| --- |
| zend\_compile\_stmt() // 编译语句  ->zend\_compile\_expr() // 编译表达式  ->zend\_compile\_expr\_inner() // 根据不同的表达式类型，调用不同的方法处理  ->zend\_compile\_encaps\_list() |

zend\_compile\_encaps\_list()函数定义如下：

|  |
| --- |
| static void zend\_compile\_encaps\_list(znode \*result, zend\_ast \*ast) {  ...  } else if (j == 2) { // 如果刚好有二个变量  opline->opcode = ZEND\_FAST\_CONCAT; // 操作码 ZEND\_FAST\_CONCAT  opline->extended\_value = 0; // 无扩展信息  opline->op1\_type = init\_opline->op2\_type; // 第一个运算对象类型，从第二个复制  opline->op1 = init\_opline->op2; // 第一个运算对象数据，从第二个复制  zend\_make\_tmp\_result(result, opline); // 操作码结果类型 为临时变量  MAKE\_NOP(init\_opline); // 清空初始化操作码  ... |

如上所示，只有在变量数量恰好为2个时，才会创建ZEND\_FAST\_CONCAT操作码。关于ZEND\_AST\_ENCAPS\_LIST语句的更多介绍参见其独立章节，这里仅介绍与ZEND\_FAST\_CONCAT操作码相关的情况。

### 在编译时计算

快速连接运算操作码ZEND\_FAST\_CONCAT在编译先尝试通过get\_binary\_op()函数取得对应的处理函数concat\_function()，尝试进行处理。concat\_function()函数在前文中已经介绍过了。

### “快速连接”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_FAST\_CONCAT操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(53, ZEND\_FAST\_CONCAT, CONST|TMPVAR|CV, CONST|TMPVAR|CV) {  .. // 取出两个操作对象的变量  if ((OP1\_TYPE == IS\_CONST || EXPECTED(Z\_TYPE\_P(op1) == IS\_STRING)) &&  (OP2\_TYPE == IS\_CONST || EXPECTED(Z\_TYPE\_P(op2) == IS\_STRING))) {  zend\_string \*op1\_str = Z\_STR\_P(op1); // 取出字串  zend\_string \*op2\_str = Z\_STR\_P(op2);  zend\_string \*str; //  // 如果 OP1 不是常量 并且 长度为0  if (OP1\_TYPE != IS\_CONST && UNEXPECTED(ZSTR\_LEN(op1\_str) == 0)) {  if (OP2\_TYPE == IS\_CONST || OP2\_TYPE == IS\_CV) { // 如果 op2 是常量或编译变量  ZVAL\_STR\_COPY(EX\_VAR(opline->result.var), op2\_str); // op2 复制到结果里  } else { // 否则  ZVAL\_STR(EX\_VAR(opline->result.var), op2\_str); // op2 的字串直接放到结果里  }  if (OP1\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op1 类型 是变量或临时变量  zend\_string\_release\_ex(op1\_str, 0); // 释放 临时字串  }  // 否则 如果 OP2 不是常量 并且 长度为0  } else if (OP2\_TYPE != IS\_CONST && UNEXPECTED(ZSTR\_LEN(op2\_str) == 0)) {  if (OP1\_TYPE == IS\_CONST || OP1\_TYPE == IS\_CV) { // 如果op1是常量或编译变量  ZVAL\_STR\_COPY(EX\_VAR(opline->result.var), op1\_str); // op1复制到结果里  } else { // 否则  ZVAL\_STR(EX\_VAR(opline->result.var), op1\_str); // op1字串指针放到结果里  }  if (OP2\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op2 类型 是变量或临时变量  zend\_string\_release\_ex(op2\_str, 0); // 释放 临时字串  }  // 否则 如果 OP1 不是常量 也不是 编译变量 也不是内置字串 并且 引用数为1  } else if (OP1\_TYPE != IS\_CONST && OP1\_TYPE != IS\_CV &&  !ZSTR\_IS\_INTERNED(op1\_str) && GC\_REFCOUNT(op1\_str) == 1) {  size\_t len = ZSTR\_LEN(op1\_str); // 长度  str = zend\_string\_extend(op1\_str, len + ZSTR\_LEN(op2\_str), 0); // 字串增加长度  // 把op2复制到后面  memcpy(ZSTR\_VAL(str) + len, ZSTR\_VAL(op2\_str), ZSTR\_LEN(op2\_str)+1);  ZVAL\_NEW\_STR(EX\_VAR(opline->result.var), str); // 把字串放到结果变量中  if (OP2\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op2 类型 是变量或临时变量  zend\_string\_release\_ex(op2\_str, 0); // 释放临时变量  }  } else { // 否则  str = zend\_string\_alloc(ZSTR\_LEN(op1\_str) + ZSTR\_LEN(op2\_str), 0); // 分配字串  // 前面一段放op1，后面一段放op2  memcpy(ZSTR\_VAL(str), ZSTR\_VAL(op1\_str), ZSTR\_LEN(op1\_str));  memcpy(ZSTR\_VAL(str) + ZSTR\_LEN(op1\_str), ZSTR\_VAL(op2\_str), ZSTR\_LEN(op2\_str)+1);  ZVAL\_NEW\_STR(EX\_VAR(opline->result.var), str); // 把字串添加到结果中  if (OP1\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op1 类型 是变量或临时变量  zend\_string\_release\_ex(op1\_str, 0); // 释放op1 临时变量  }  if (OP2\_TYPE & (IS\_TMP\_VAR|IS\_VAR)) { // 如果 op2 类型 是变量或临时变量  zend\_string\_release\_ex(op2\_str, 0); // 释放op2 临时变量  }  }  ZEND\_VM\_NEXT\_OPCODE(); // opline + 1，到目标操作码并 return  }  SAVE\_OPLINE(); // windows: 无操作  if (OP1\_TYPE == IS\_CONST) { // 如果op1是常量  op1\_str = Z\_STR\_P(op1); // 取出字串  } else if (EXPECTED(Z\_TYPE\_P(op1) == IS\_STRING)) { // 如果类型是字串  op1\_str = zend\_string\_copy(Z\_STR\_P(op1)); // op1 增加引用次数.  } else { // 否则  // 如果 op1 是 编译变量 并且 值为未定义  if (OP1\_TYPE == IS\_CV && UNEXPECTED(Z\_TYPE\_P(op1) == IS\_UNDEF)) {  ZVAL\_UNDEFINED\_OP1(); // 报错：p1的变量（变量名）未定义, 并返未初始化zval  }  op1\_str = zval\_get\_string\_func(op1); // zval 转成 字串, 没有try  }    if (OP2\_TYPE == IS\_CONST) { // 如果op2是常量  op2\_str = Z\_STR\_P(op2); // 取出字串  } else if (EXPECTED(Z\_TYPE\_P(op2) == IS\_STRING)) { // op2是字串  op2\_str = zend\_string\_copy(Z\_STR\_P(op2)); // op2 增加引用次数.  } else { // 否则  // 如果 op2 是 编译变量 并且 类型为 未定义  if (OP2\_TYPE == IS\_CV && UNEXPECTED(Z\_TYPE\_P(op2) == IS\_UNDEF)) {  ZVAL\_UNDEFINED\_OP2(); // 报错：p2的变量（变量名）未定义  }  op2\_str = zval\_get\_string\_func(op2); // zval 转成 字串, 没有try  }  do {  if (OP1\_TYPE != IS\_CONST) { // 如果 OP1 不是常量  if (UNEXPECTED(ZSTR\_LEN(op1\_str) == 0)) { // 如果op1长度为0  if (OP2\_TYPE == IS\_CONST) { // 如果类型为 常量  if (UNEXPECTED(Z\_REFCOUNTED\_P(op2))) { // 如果是 可计数类型  GC\_ADDREF(op2\_str); // 增加引用次数  }  }  ZVAL\_STR(EX\_VAR(opline->result.var), op2\_str); // 直接把op2放到结果里  zend\_string\_release\_ex(op1\_str, 0); // 释放op1临时变量  break; // 跳出  }  }  if (OP2\_TYPE != IS\_CONST) { // 如果 OP2 不是常量  if (UNEXPECTED(ZSTR\_LEN(op2\_str) == 0)) { // 如果op2长度为为  if (OP1\_TYPE == IS\_CONST) { // 如果op1是常量  if (UNEXPECTED(Z\_REFCOUNTED\_P(op1))) { // 如果是 可计数类型  GC\_ADDREF(op1\_str); // 增加引用次数  }  }  ZVAL\_STR(EX\_VAR(opline->result.var), op1\_str); // op1直接放到结果里  zend\_string\_release\_ex(op2\_str, 0); // 释放op2临时变量  break; // 跳出  }  }  str = zend\_string\_alloc(ZSTR\_LEN(op1\_str) + ZSTR\_LEN(op2\_str), 0); // 分配新字串  // 先把op1放进来，再把op2放进来  memcpy(ZSTR\_VAL(str), ZSTR\_VAL(op1\_str), ZSTR\_LEN(op1\_str));  memcpy(ZSTR\_VAL(str) + ZSTR\_LEN(op1\_str), ZSTR\_VAL(op2\_str), ZSTR\_LEN(op2\_str)+1);  ZVAL\_NEW\_STR(EX\_VAR(opline->result.var), str); // 字串保留到结果变量中  if (OP1\_TYPE != IS\_CONST) { // 如果 OP1 不是常量  zend\_string\_release\_ex(op1\_str, 0); // 释放op1临时变量  }  if (OP2\_TYPE != IS\_CONST) { // 如果 OP2 不是常量  zend\_string\_release\_ex(op2\_str, 0); // 释放op2临时变量  }  } while (0);  FREE\_OP1(); // 释放操作对象的附加变量  FREE\_OP2();  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  } |

如上所示，处理器中进行了更多繁琐的优化，但业务逻辑并不算复杂。此伪函数会根据两个操作对象的变量类型，被转换成9个处理器函数：

1）ZEND\_FAST\_CONCAT\_SPEC\_CONST\_CONST\_HANDLER()函数，两个常量的连接操作；

2）ZEND\_FAST\_CONCAT\_SPEC\_CONST\_TMPVAR\_HANDLER()函数，处理一个常量和一个普通变量（或临时变量）的连接操作，例如：**echo "11{$GLOBAL['a']}"**；

3）ZEND\_FAST\_CONCAT\_SPEC\_CONST\_CV\_HANDLER()函数，处理一个常量和一个编译变量的连接操作，例如：**echo "11{$a}"**；

4）ZEND\_FAST\_CONCAT\_SPEC\_TMPVAR\_CONST\_HANDLER()函数，处理一个普通变量（或临时变量）和一个常量的连接操作，例如：**echo "{$GLOBAL['a']}11"**；

5）ZEND\_FAST\_CONCAT\_SPEC\_TMPVAR\_TMPVAR\_HANDLER()函数，两个普通变量（或临时变量）的连接操作，例如：**echo "{$GLOBAL['a']}{$GLOBAL['a']}"**；

6）ZEND\_FAST\_CONCAT\_SPEC\_TMPVAR\_CV\_HANDLER()函数，处理一个普通变量（或临时变量）和一个编译变量的连接操作，例如：**echo "{$GLOBAL['a']}{$a}"**；

7）ZEND\_FAST\_CONCAT\_SPEC\_CV\_CONST\_HANDLER()函数，处理一个编译变量和一个常量的连接操作，例如：**echo "{$a}11"**；

8）ZEND\_FAST\_CONCAT\_SPEC\_CV\_TMPVAR\_HANDLER()函数，处理一个编译变量和一个普通变量（或临时变量）的连接操作，例如：**echo "{$a}{$GLOBAL['a']}"**；

9）ZEND\_FAST\_CONCAT\_SPEC\_CV\_CV\_HANDLER()函数，处理两个编译变量的连接操作，例如：**echo "{$a}{$a}"**。

## 三）布尔抑或运算（ZEND\_BOOL\_XOR）

布尔抑或运算会在两个运算对象的布尔值不同（一个为true一个为false）时返回true，其他情况返回false，例如：

|  |
| --- |
| <?php  var\_dump( true ^ true ); // int(0)  var\_dump( true ^ false ); // int(1)  var\_dump( false ^ true ); // int(1)  var\_dump( false ^ false ); // int(0) |

### 在编译时计算

布尔抑或运算操作码为ZEND\_BOOL\_XOR，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数boolean\_xor\_function()，尝试进行处理。boolean\_xor\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| // 布尔抑或（两个值不同时返回 true）  ZEND\_API zend\_result ZEND\_FASTCALL boolean\_xor\_function(zval \*result, zval \*op1, zval \*op2) {  int op1\_val, op2\_val;  do {  if (Z\_TYPE\_P(op1) == IS\_FALSE) { // 如果第一个是false  op1\_val = 0; // 第一个值为0  } else if (EXPECTED(Z\_TYPE\_P(op1) == IS\_TRUE)) { // 第一个是true  op1\_val = 1; // 第一个值为1  } else { // 其他情况  if (Z\_ISREF\_P(op1)) { // 第一个引用类型  op1 = Z\_REFVAL\_P(op1); // 追踪引用目标  if (Z\_TYPE\_P(op1) == IS\_FALSE) { // 如果第一个是false  op1\_val = 0; // 第一个值为0  break;  } else if (EXPECTED(Z\_TYPE\_P(op1) == IS\_TRUE)) { // 第一个是true  op1\_val = 1; // 第一个值为1  break;  }  }  // 调用第一个的 do\_operation方法，执行 ZEND\_BOOL\_XOR 操作  ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION(ZEND\_BOOL\_XOR);  op1\_val = zval\_is\_true(op1); // 第一个转成bool型  }  } while (0);  do {  **... // 第二个操作对象的处理逻辑与第一个相同**  } while (0);  ZVAL\_BOOL(result, op1\_val ^ op2\_val); // 直接进行 抑或 操作，返回布尔值  return SUCCESS;  } |

如上所示，代码虽然比较多但业务逻辑简单，对op1和op2进行一些处理后，直接使用抑或运算，并把结果转成布尔型。

### “布尔抑或运算”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_BW\_OR操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONSTCONST\_HANDLER(15, ZEND\_BOOL\_XOR, CONST|TMPVAR|CV, CONST|TMPVAR|CV, SPEC(COMMUTATIVE)) {  ... // 取出op1,op2的变量  boolean\_xor\_function(EX\_VAR(opline->result.var), op1, op2); // 布尔抑或  ... // 销毁op1,op2的变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  } |

如上所示，由于添加了扩展规则COMMUTATIVE（操作对象可交换），此伪函数会根据两个操作对象的变量类型，被转换成6个处理器函数：

1）ZEND\_BOOL\_XOR\_SPEC\_CONST\_CONST\_HANDLER()函数，处理两个常量的运算，如**“1 ^ 1”**；

2）ZEND\_BOOL\_XOR\_SPEC\_TMPVAR\_CONST\_HANDLER()函数，处理一个变量和一个常量的运算，如**“$GLBOALS['a'] ^ true”或 “true ^ $GLBOALS['a']”**；

3）ZEND\_BOOL\_XOR\_SPEC\_TMPVAR\_TMPVAR\_HANDLER()函数，处理两个变量的运算，如**“$GLBOALS['a'] ^ $GLBOALS['a']”**；

4）ZEND\_BOOL\_XOR\_SPEC\_CV\_CONST\_HANDLER()函数，处理一个编译变量和一个常量的运算，如**“$a ^ 1”或 “1 ^ $a”**；

5）ZEND\_BOOL\_XOR\_SPEC\_CV\_TMPVAR\_HANDLER()函数，处理一个编译变量和一个普通（或临时）变量的运算，如**“$a ^ $GLBOALS['a']”或 “$GLBOALS['a'] ^ $a”**；

6）ZEND\_BOOL\_XOR\_SPEC\_CV\_CV\_HANDLER()函数，处理两个编译变量的运算，如“**$a ^ $a”**。

## 四）布尔或、与运算（ZEND\_BOOL）

布尔运算语句有两个，布尔或（ZEND\_AST\_OR）和布尔与（ZEND\_AST\_AND），示例如下：

|  |
| --- |
| <?php  $a = $b = 1;  echo $a or $b; // 创建ZEND\_AST\_OR类型语句  echo $a and $b; // 创建ZEND\_AST\_AND类型语句 |

### 解析源码，创建ZEND\_AST\_AND、ZEND\_AST\_OR语句

解析过程如下：

|  |
| --- |
| ...  **zend\_ast \* ast\_194 = zend\_ast\_create(ZEND\_AST\_OR, ast\_130, ast\_161); /\* expr:["ast\_130","or","ast\_161"] \*/**  ...  **zend\_ast \* ast\_313 = zend\_ast\_create(ZEND\_AST\_AND, ast\_249, ast\_280); /\* expr:["ast\_249","and","ast\_280"] \*/**  ...  CG(ast) = ast\_360; /\* start:["ast\_360"] \*/ |

如上所示，布尔或（or）运算会创建ZEND\_AST\_OR语句，布尔与（and）运算会创建ZEND\_AST\_AND语句。

### 编译语句

编译ZEND\_AST\_AND、ZEND\_AST\_OR语句需要用到zend\_compile\_short\_circuiting()函数，调用路径如下：

|  |
| --- |
| zend\_compile\_stmt() // 编译语句  ->zend\_compile\_expr() // 编译表达式  ->zend\_compile\_expr\_inner() // 根据不同的表达式类型，调用不同的方法处理  ->zend\_compile\_short\_circuiting() |

#### **zend\_compile\_short\_circuiting()函数**

zend\_compile\_short\_circuiting()函数用于编译“布尔或”（ZEND\_AST\_OR）和“布尔与”（ZEND\_AST\_AND）语句，代码如下：

|  |
| --- |
| static void zend\_compile\_short\_circuiting(znode \*result, zend\_ast \*ast) {  zend\_ast \*left\_ast = ast->child[0]; // 左侧子句  zend\_ast \*right\_ast = ast->child[1]; // 右侧子句  znode left\_node, right\_node;  zend\_op \*opline\_jmpz, \*opline\_bool;  uint32\_t opnum\_jmpz;  // 语句类型必须是 and 或 or  ZEND\_ASSERT(ast->kind == ZEND\_AST\_AND || ast->kind == ZEND\_AST\_OR);  zend\_compile\_expr(&left\_node, left\_ast); // 编译左侧表达式  // 如果左侧结果是常量  if (left\_node.op\_type == IS\_CONST) {  // 如果(是 and 运算 并且 左侧不是 true) 或（是 or 运算 并且 左侧结果是 true) ，不需要计算右侧表达式了  if ((ast->kind == ZEND\_AST\_AND && !zend\_is\_true(&left\_node.u.constant))  || (ast->kind == ZEND\_AST\_OR && zend\_is\_true(&left\_node.u.constant))) {  // 运算结果为常量  result->op\_type = IS\_CONST;  // 值与左侧表达式相同  ZVAL\_BOOL(&result->u.constant, zend\_is\_true(&left\_node.u.constant));  } else { // 否则，需要计算右侧表达式  zend\_compile\_expr(&right\_node, right\_ast); // 编译右侧表达式  if (right\_node.op\_type == IS\_CONST) { // 如果左右侧结果都是常量  result->op\_type = IS\_CONST; // 结果类型是常量  ZVAL\_BOOL(&result->u.constant, zend\_is\_true(&right\_node.u.constant)); // 值与右侧值相同  zval\_ptr\_dtor(&right\_node.u.constant); // 销毁右侧结果常量  } else { // 如果右侧结果不是常量  // 创建操作码 ZEND\_BOOL，右侧结果的布尔运算  zend\_emit\_op\_tmp(result, ZEND\_BOOL, &right\_node, NULL);  }  }  zval\_ptr\_dtor(&left\_node.u.constant); // 销毁左侧运算对象的常量  return;  }  // 到这里说明左侧结果不是常量  // 当前上下文中的操作码数量  opnum\_jmpz = get\_next\_op\_number();    // 这行表示如果是ZEND\_AST\_AND语句，第一个表达式为true时，要计算第二个表达式；如果是ZEND\_AST\_OR语句，第一个表达式为true时，要跳过第二个表达式  opline\_jmpz = zend\_emit\_op(NULL, ast->kind == ZEND\_AST\_AND ? ZEND\_JMPZ\_EX : ZEND\_JMPNZ\_EX, &left\_node, NULL);  if (left\_node.op\_type == IS\_TMP\_VAR) { // 如果已经有了存放结果用的临时变量  SET\_NODE(opline\_jmpz->result, &left\_node); // 用已有的临时变量存放操作码执行结果  GET\_NODE(result, opline\_jmpz->result); // 当前返回结果也使用同一个临时变量  } else { // 否则，创建临时变量用来存放运算结果  zend\_make\_tmp\_result(result, opline\_jmpz);  }  zend\_compile\_expr(&right\_node, right\_ast); // 编译右侧表达式  // 创建操作码 ZEND\_BOOL，进行布尔型操作  opline\_bool = zend\_emit\_op(NULL, ZEND\_BOOL, &right\_node, NULL);  SET\_NODE(opline\_bool->result, result);  // 更新指定跳转型操作码，跳转到下一个创建的新操作码 。  zend\_update\_jump\_target\_to\_next(opnum\_jmpz);  } |

如上所示，业务逻辑可归纳成两种情况：

一、编译左侧表达式，如果编译结果类型为常量，分两种情况操作：

1）如果（是 and 运算 并且左侧不是 true） 或（是 or 运算 并且 左侧结果是 true），这时就不用计算右侧表达式了，直接使用左侧表达式的结果；

2）否则，编译右侧表达式。

a）如果编译结果为常量，直接用此常量作为结果；

b）如果编译结果不是常量，创建ZEND\_BOOL操作码，操作码的第一个运算对象为刚编译过的右侧表达式，第二个运算对象为NULL。

二、左侧表达式编译结果不为常量，分为几步操作

1）创建新操作码，语句类型为ZEND\_AST\_AND时，操作码为ZEND\_JMPZ\_EX，否则操作码类型为ZEND\_JMPNZ\_EX（这行表示如果是ZEND\_AST\_AND语句，第一个表达式为true时，要计算第二个表达式；如果是ZEND\_AST\_OR语句，第一个表达式为true时，要跳过第二个表达式）；

2）如果左侧表达式类型为临时变量（IS\_TMP\_VAR）：使用已创建的临时变量来存放返回结果；

3）编译右侧表达式；

4）创建ZEND\_BOOL操作码，第一个运算对象为右侧表达式，第二个运算对象为NULL；

5）用当前运算结果变量来接收刚创建的操作码的运算结果；

6）修正跳转操作码的跳转目标。

通过以上业务逻辑不难看出，“布尔或”（ZEND\_AST\_OR）和“布尔与”（ZEND\_AST\_AND）语句都是二元操作语句，但布尔运算操作码ZEND\_BOOL是一个一元操作码，每次只能计算一个表达式，所以如果两侧的表达式都需要使用操作码进行运算时，需要借助ZEND\_JMPZ\_EX和ZEND\_JMPNZ\_EX操作码进行跳转。

ZEND\_JMPZ\_EX和ZEND\_JMPNZ\_EX操作码是进行布尔运算时使用的辅助操作码，只有在zend\_compile\_short\_circuiting()函数中创建，所以在下文中和ZEND\_BOOL操作码一并介绍。

#### **zend\_update\_jump\_target\_to\_next()函数**

zend\_update\_jump\_target\_to\_next()函数用于修正已经操作码的跳转位置，定义如下：

|  |
| --- |
| static inline void zend\_update\_jump\_target\_to\_next(uint32\_t opnum\_jump) {  // 记录下一个即将产生的操作码的序号，那就是应该跳转的位置  zend\_update\_jump\_target(opnum\_jump, get\_next\_op\_number());  } |

#### **zend\_update\_jump\_target()函数**

zend\_update\_jump\_target()函数，定义如下：

|  |
| --- |
| static inline void zend\_update\_jump\_target(uint32\_t opnum\_jump, uint32\_t opnum\_target) {  zend\_op \*opline = &CG(active\_op\_array)->opcodes[opnum\_jump]; // 要更新的操作码  switch (opline->opcode) { // 根据类型操作, 写入目标操作码  case ZEND\_JMP: // ZEND\_JMP 操作码，跳转目标行号写在第一个操作对象里  opline->op1.opline\_num = opnum\_target;  break;  case ZEND\_JMPZ: // 这些操作码，跳转目标行号写在第二个操作对象里  case ZEND\_JMPNZ:  case ZEND\_JMPZ\_EX:  case ZEND\_JMPNZ\_EX:  case ZEND\_JMP\_SET:  case ZEND\_COALESCE:  case ZEND\_JMP\_NULL:  opline->op2.opline\_num = opnum\_target;  break;  EMPTY\_SWITCH\_DEFAULT\_CASE() // 其他情况无操作  }  } |

### “布尔运算”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_BOOL操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONST\_HANDLER(52, ZEND\_BOOL, CONST|TMPVAR|CV, ANY) {  USE\_OPLINE // const zend\_op \*opline = EX(opline);  zval \*val;  val = GET\_OP1\_ZVAL\_PTR\_UNDEF(BP\_VAR\_R); // 获取op1的zval指针, UNUSED 返回null  if (Z\_TYPE\_INFO\_P(val) == IS\_TRUE) { // 如果值为true  ZVAL\_TRUE(EX\_VAR(opline->result.var)); // 结果为 true  } else if (EXPECTED(Z\_TYPE\_INFO\_P(val) <= IS\_TRUE)) { // 否则 如果值为 undef/null/false  // result和op1两个运算对象 可能是同一个编译变量  /\* The result and op1 can be the same cv zval \*/  const uint32\_t orig\_val\_type = Z\_TYPE\_INFO\_P(val); // 原 op1 的变量类型  ZVAL\_FALSE(EX\_VAR(opline->result.var)); // 结果为 false  // 如果 op1 是 编译变量 并且 原值类型为 未定义  if (OP1\_TYPE == IS\_CV && UNEXPECTED(orig\_val\_type == IS\_UNDEF)) {  SAVE\_OPLINE(); // windows: 无操作  ZVAL\_UNDEFINED\_OP1(); // 报错：p1的变量（变量名）未定义, 并返未初始化zval  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  }  } else { // 否则，其他值  SAVE\_OPLINE(); // windows: 无操作  ZVAL\_BOOL(EX\_VAR(opline->result.var), i\_zend\_is\_true(val)); // val转成布尔型  FREE\_OP1(); // 释放操作对象的附加变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  }  ZEND\_VM\_NEXT\_OPCODE(); // 下一个操作码  } |

如上所示，伪码的业务逻辑可归纳如下，按第一个运算对象的值的类型进行操作：

1）如果值为true，结果为true；

2）如果值类型为无效或NULL或false，结果为false；

3）其他情况，使用 i\_zend\_is\_true()函数，把值转成布尔型。

此伪函数会根据第一个操作对象的变量类型，被转换成3个处理器函数：

1）ZEND\_BOOL\_SPEC\_CONST\_HANDLER()函数，用于把常量转成布尔型，例如“1 or 1”；

2）ZEND\_BOOL\_SPEC\_TMPVAR\_HANDLER()函数，用于把变量转成布尔型，例如“$GLOBALS['a'] or 1”；

3）ZEND\_BOOL\_SPEC\_CV\_HANDLER()函数，用于把编译变量转成布尔型，例如“$a or 1”。

#### **i\_zend\_is\_true()函数**

i\_zend\_is\_true()函数用于把变量转成布尔型，函数定义如下：

|  |
| --- |
| static zend\_always\_inline bool i\_zend\_is\_true(zval \*op) {  bool result = 0; // 默认值为false  again:  switch (Z\_TYPE\_P(op)) {  // 如果类型是 is\_true, 返回 1  case IS\_TRUE: result = 1; break;  // long 型, 并且 > 0 , 返回 1  case IS\_LONG: if (Z\_LVAL\_P(op)) { result = 1; } break;  // double 型, 并且 > 0 , 返回 1  case IS\_DOUBLE: if (Z\_DVAL\_P(op)) {result = 1; } break;  // string 型， 长度>1 或者， (长度是1 并且内容不是 '0'), 返回 1  case IS\_STRING:  if (Z\_STRLEN\_P(op) > 1 || (Z\_STRLEN\_P(op) && Z\_STRVAL\_P(op)[0] != '0')) {  result = 1;  }  break;  case IS\_ARRAY: // array 型，不空，返回 1  if (zend\_hash\_num\_elements(Z\_ARRVAL\_P(op))) { result = 1; }  break;  case IS\_OBJECT: // 如果类型是对象  // 可以用标记方法转换成字串的，都算有效  if (EXPECTED(Z\_OBJ\_HT\_P(op)->cast\_object == zend\_std\_cast\_object\_tostring)) {  result = 1;  } else { // 否则 ，调用对象自己的转换方法  result = zend\_object\_is\_true(op);  }  break;  case IS\_RESOURCE: // 如果是资源，取得handle，就是资源编号，编号有效即可  if (EXPECTED(Z\_RES\_HANDLE\_P(op))) { result = 1; } break;  case IS\_REFERENCE: // 如果是引用  op = Z\_REFVAL\_P(op); // 追踪引用目标  goto again; // 从头再来  break;  default: break;  }  return result; // 返回 false  } |

如上所示，当碰到不使用标准转换方法的对象时，需要使用zend\_object\_is\_true()函数对它进行转换。

#### **zend\_object\_is\_true()函数**

zend\_object\_is\_true()函数用于调用对象自身的转换方法把对象转成布尔型，代码如下：

|  |
| --- |
| ZEND\_API bool ZEND\_FASTCALL zend\_object\_is\_true(zval \*op) {  zend\_object \*zobj = Z\_OBJ\_P(op); // 取得zval里的对象  zval tmp;  // 调用对象自己的cast\_object()方法，把它转成布尔型  if (zobj->handlers->cast\_object(zobj, &tmp, \_IS\_BOOL) == SUCCESS) {  return Z\_TYPE(tmp) == IS\_TRUE;  }  // 如果转换失败，报错：此对象无法转为bool型  zend\_error(E\_RECOVERABLE\_ERROR, "Object of class %s could not be converted to bool", ZSTR\_VAL(zobj->ce->name));  return false;  } |

### ZEND\_JMPZ\_EX操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_JMPZ\_EX操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONST\_HANDLER(46, ZEND\_JMPZ\_EX, CONST|TMPVAR|CV, JMP\_ADDR){  ... // 取出op1中的变量  if (Z\_TYPE\_INFO\_P(val) == IS\_TRUE) { // 如果 变量 为 true  ZVAL\_TRUE(EX\_VAR(opline->result.var)); // 结果为 true  ZEND\_VM\_NEXT\_OPCODE(); // 跳转到下一个操作码  } else if (EXPECTED(Z\_TYPE\_INFO\_P(val) <= IS\_TRUE)) { // 如果变量为 false,null,未定义  ZVAL\_FALSE(EX\_VAR(opline->result.var)); // 结果为 false  // 如果 op1 类型为 编译变量 并且 值类型为未定义  if (OP1\_TYPE == IS\_CV && UNEXPECTED(Z\_TYPE\_INFO\_P(val) == IS\_UNDEF)) {  ... // 报错：p1的变量（变量名）未定义  }  // 跳转到op2中记录好的的位置  ZEND\_VM\_JMP\_EX(OP\_JMP\_ADDR(opline, opline->op2), 0);  }  SAVE\_OPLINE(); // windows: 无操作  ret = i\_zend\_is\_true(val); // 调用对象自己的方法，把对象转成布尔型  FREE\_OP1(); // 释放操作对象的附加变量  if (ret) { // 如果值为true  ZVAL\_TRUE(EX\_VAR(opline->result.var)); // 结果为true  opline++; // 下一个操作码  } else { // 否则，值为false  ZVAL\_FALSE(EX\_VAR(opline->result.var)); // 结果为false  opline = OP\_JMP\_ADDR(opline, opline->op2); // OP\_JMP\_ADDR: 访问 p2.jmp\_addr  }  ZEND\_VM\_JMP(opline); // 有异常时使用旧操作码，无异常使用新操作码, p1:新操作码  } |

如上所示，ZEND\_JMPZ\_EX操作码的主要功能为：根据op1的值进行跳转，如果值转成布尔型为true，则跳转到下一个操作码，否则跳转到op2指定的位置。此伪函数会根据第一个操作对象的变量类型，被转换成3个处理器函数：

1）ZEND\_JMPZ\_EX\_SPEC\_CV\_HANDLER()函数，用于处理编译变量；

2）ZEND\_JMPZ\_EX\_SPEC\_CONST\_HANDLER()函数，用于处理常量；

3）ZEND\_JMPZ\_EX\_SPEC\_TMPVAR\_HANDLER()函数，用于处理（普通或临时）变量。

相关宏程序定义如下：

|  |
| --- |
| // 要检查异常 并 有异常时使用旧操作码，无异常使用新操作码, p1:新操作码，p2:是否检查异常  #define ZEND\_VM\_JMP\_EX(new\_op, check\_exception) do { \  if (check\_exception && UNEXPECTED(EG(exception))) { /\* 要检查异常并且有异常 \*/ \  HANDLE\_EXCEPTION(); /\* windows: opline = EX(opline); return 0; \*/ \  } \  ZEND\_VM\_SET\_OPCODE(new\_op); /\* 修改当前操作码并返回。EX(opline) = p1 \*/ \  ZEND\_VM\_CONTINUE(); /\* windows: return 0 \*/ \  } while (0)  // 有异常时使用旧操作码，无异常使用新操作码, p1:新操作码  #define ZEND\_VM\_JMP(new\_op) \  ZEND\_VM\_JMP\_EX(new\_op, 1)  // 访问.jmp\_addr元素  # define OP\_JMP\_ADDR(opline, node) (node).jmp\_addr |

### ZEND\_JMPNZ\_EX操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_JMPNZ\_EX操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONST\_HANDLER(47, ZEND\_JMPNZ\_EX, CONST|TMPVAR|CV, JMP\_ADDR) {  ... // 取出op1中的变量  if (Z\_TYPE\_INFO\_P(val) == IS\_TRUE) { // 如果 变量 为 true  ZVAL\_TRUE(EX\_VAR(opline->result.var));  // 跳转到op2中记录好的的位置  ZEND\_VM\_JMP\_EX(OP\_JMP\_ADDR(opline, opline->op2), 0);  } else if (EXPECTED(Z\_TYPE\_INFO\_P(val) <= IS\_TRUE)) { // 否则如果变量为 false,null,未定义  ZVAL\_FALSE(EX\_VAR(opline->result.var)); // 结果为假  // 如果 op1 类型为 编译变量 并且 值类型为未定义  if (OP1\_TYPE == IS\_CV && UNEXPECTED(Z\_TYPE\_INFO\_P(val) == IS\_UNDEF)) {  ... // 报错：p1的变量（变量名）未定义  } else { // 否则  ZEND\_VM\_NEXT\_OPCODE(); // 跳转到下一个操作码  }  }  SAVE\_OPLINE(); // windows: 无操作  ret = i\_zend\_is\_true(val); // 值转成bool型  FREE\_OP1(); // 释放操作对象的附加变量  if (ret) { // 如果值为真  ZVAL\_TRUE(EX\_VAR(opline->result.var)); // 结果为真  opline = OP\_JMP\_ADDR(opline, opline->op2); // 跳转到op2中记录好的的位置  } else { // 否则  ZVAL\_FALSE(EX\_VAR(opline->result.var)); // 结果为假  opline++; // 到下一个操作码（值为假，的跳转目标）  }  ZEND\_VM\_JMP(opline); // 有异常时使用旧操作码，无异常使用新操作码, p1:新操作码  } |

如上所示，ZEND\_JMPNZ\_EX操作码的效果与ZEND\_JMPZ\_EX相反，主要功能为：根据op1的值进行跳转，如果值转成布尔型为true，则跳转到op2指定的位置，否则跳转到下一个操作码。此伪函数会根据第一个操作对象的变量类型，被转换成3个处理器函数：

1）ZEND\_JMPNZ\_EX\_SPEC\_CV\_HANDLER()函数，用于处理编译变量；

2）ZEND\_JMPNZ\_EX\_SPEC\_CONST\_HANDLER()函数，用于处理常量；

3）ZEND\_JMPNZ\_EX\_SPEC\_TMPVAR\_HANDLER()函数，用于处理（普通或临时）变量。

# 六、一元运算

一元运算包括：布尔非运算（ZEND\_BOOL\_NOT)、二进制取反运算（ZEND\_BW\_NOT）。

## 一）解析源码，创建ZEND\_AST\_UNARY\_OP语句

在zend\_language\_parser.y中定义的PHP语法解析规则中，有两个一元操作符，分别是“!”（布尔非）和“~”（二进制取反），语法定义如下：

|  |
| --- |
| expr:  ...  | '!' expr { $$ = zend\_ast\_create\_ex(ZEND\_AST\_UNARY\_OP, ZEND\_BOOL\_NOT, $2); }  | '~' expr { $$ = zend\_ast\_create\_ex(ZEND\_AST\_UNARY\_OP, ZEND\_BW\_NOT, $2); } |

如上所示，这两种表达式会被解析成一元运算（ZEND\_AST\_UNARY\_OP）语句

PHP代码示例如下：

|  |
| --- |
| <?php  echo !false; // 输出结果为1  echo ~0; // 取反运算会把整数的每一位都标记成1，输出结果为-1 |

下面先介绍布尔非运算（ZEND\_BOOL\_NOT)运算的编译和执行过程。

## 二）布尔非运算（ZEND\_BOOL\_NOT）

### 在编译时计算

编译ZEND\_AST\_UNARY\_OP语句需要用到get\_unary\_op()函数，调用路径如下：

|  |
| --- |
| zend\_compile\_stmt() // 编译语句  ->zend\_compile\_expr() // 编译表达式  ->zend\_compile\_expr\_inner() // 根据不同的表达式类型，调用不同的方法处理  ->get\_unary\_op() |

get\_unary\_op()函数定义如下：

|  |
| --- |
| ZEND\_API unary\_op\_type get\_unary\_op(int opcode) {  switch (opcode) {  case ZEND\_BW\_NOT: return (unary\_op\_type) bitwise\_not\_function;  case ZEND\_BOOL\_NOT: return (unary\_op\_type) boolean\_not\_function;  default: return (unary\_op\_type) NULL;  }  } |

布尔非运算操作码为ZEND\_BOOL\_NOT，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数boolean\_not\_function()，尝试进行处理。boolean\_not\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL boolean\_not\_function(zval \*result, zval \*op1) {  if (Z\_TYPE\_P(op1) < IS\_TRUE) { // 如果小于true：false,null,undef  ZVAL\_TRUE(result); // 结果为true  } else if (EXPECTED(Z\_TYPE\_P(op1) == IS\_TRUE)) { // 如果是true  ZVAL\_FALSE(result); // 结果为false  } else { // 其他情况  if (Z\_ISREF\_P(op1)) { // 如果是引用类型  op1 = Z\_REFVAL\_P(op1); // 追踪引用目标  if (Z\_TYPE\_P(op1) < IS\_TRUE) { // 如果小于true：false,null,undef  ZVAL\_TRUE(result); // 结果为true  return SUCCESS; // 返回成功  } else if (EXPECTED(Z\_TYPE\_P(op1) == IS\_TRUE)) { // 如果是true  ZVAL\_FALSE(result); // 结果为false  return SUCCESS;  }  }  // 如果是对象，调用do\_operation方法，字串等不处理  ZEND\_TRY\_UNARY\_OBJECT\_OPERATION(ZEND\_BOOL\_NOT);  ZVAL\_BOOL(result, !zval\_is\_true(op1)); // 转成布尔值再取否  }  return SUCCESS;  } |

ZEND\_TRY\_UNARY\_OBJECT\_OPERATION()宏程序定义如下：

|  |
| --- |
| #define ZEND\_TRY\_UNARY\_OBJECT\_OPERATION(opcode) \  if (UNEXPECTED(Z\_TYPE\_P(op1) == IS\_OBJECT) /\* 如果op1是对象 \*/ \  /\* 如果do\_operation方法存在 \*/ \  && UNEXPECTED(Z\_OBJ\_HANDLER\_P(op1, do\_operation)) \  /\* 调用do\_operation方法，如果成功 \*/\  && EXPECTED(SUCCESS == Z\_OBJ\_HANDLER\_P(op1, do\_operation)(opcode, result, op1, NULL))) { \  /\* 返回成功 \*/ \  return SUCCESS; \  } |

### “布尔非”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_BOOL\_NOT操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_COLD\_CONST\_HANDLER(14, ZEND\_BOOL\_NOT, CONST|TMPVAR|CV, ANY) {  USE\_OPLINE // const zend\_op \*opline = EX(opline);  zval \*val;  val = GET\_OP1\_ZVAL\_PTR\_UNDEF(BP\_VAR\_R); // 获取zval指针, UNUSED 返回null  if (Z\_TYPE\_INFO\_P(val) == IS\_TRUE) { // 如果 变量为 true  ZVAL\_FALSE(EX\_VAR(opline->result.var)); // 结果为 false  } else if (EXPECTED(Z\_TYPE\_INFO\_P(val) <= IS\_TRUE)) { // 如果变量类型为 false,null,未定义  const uint32\_t orig\_val\_type = Z\_TYPE\_INFO\_P(val); // 获取变量类型  ZVAL\_TRUE(EX\_VAR(opline->result.var)); // 结果为 true  // 如果 op1 是 编译变量 并且 原值类型是 未定义  if (OP1\_TYPE == IS\_CV && UNEXPECTED(orig\_val\_type == IS\_UNDEF)) {  SAVE\_OPLINE(); // windows: 无操作  ZVAL\_UNDEFINED\_OP1(); // 报错：p1的变量（变量名）未定义, 并返未初始化zval  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  }  } else { // 否则，其他类型  SAVE\_OPLINE(); // windows: 无操作  ZVAL\_BOOL(EX\_VAR(opline->result.var), !i\_zend\_is\_true(val)); // 转成bool型后取反  FREE\_OP1(); // 释放操作对象的附加变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  }  ZEND\_VM\_NEXT\_OPCODE(); // opline + 1，到目标操作码并 return  } |

如上所示，业务逻辑比较简单，此伪函数会根据两个操作对象的变量类型，被转换成3个处理器函数：

1）ZEND\_BOOL\_NOT\_SPEC\_CV\_HANDLER()函数，用于处理编译变量，例如“!$a”；

2）ZEND\_BOOL\_NOT\_SPEC\_CONST\_HANDLER()函数，用于处理常量，例如“!true”；

3）ZEND\_BOOL\_NOT\_SPEC\_TMPVAR\_HANDLER()函数，用于处理（普通或临时）变量，例如“!$GLOBALS['a']”。

## 三）二进制取反运算（ZEND\_BW\_NOT）

二进制取反运算的解析过程和示例已经在上一节中介绍过。

### 在编译时计算

布尔非运算操作码为ZEND\_BW\_NOT，在编译先尝试通过get\_binary\_op()函数取得对应的处理函数bitwise\_not\_function()，尝试进行处理。bitwise\_not\_function()函数在zend\_operators.c中定义，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL bitwise\_not\_function(zval \*result, zval \*op1) {  try\_again: // 跳转标签  switch (Z\_TYPE\_P(op1)) { // 按类型操作  case IS\_LONG: // 整数 ，直接取反  ZVAL\_LONG(result, ~Z\_LVAL\_P(op1));  return SUCCESS;  case IS\_DOUBLE: { // 小数  zend\_long lval = zend\_dval\_to\_lval(Z\_DVAL\_P(op1)); // 转成整数  if (!zend\_is\_long\_compatible(Z\_DVAL\_P(op1), lval)) { // 如果不兼容整数  ... // 报错：隐式转换，从小数转成整数会丢失精度。并返回失败  }  ZVAL\_LONG(result, ~lval); // 整数 取反  return SUCCESS; // 返回成功  }  case IS\_STRING: { // 字串  size\_t i;  if (Z\_STRLEN\_P(op1) == 1) { // 如果长度为1  // 对目标 ascii 码取反并转成 uchar型  zend\_uchar not = (zend\_uchar) ~\*Z\_STRVAL\_P(op1);  ZVAL\_CHAR(result, not); // 返回char 型结果  } else { // 长度不为1  ZVAL\_NEW\_STR(result, zend\_string\_alloc(Z\_STRLEN\_P(op1), 0)); // 分配新字串  for (i = 0; i < Z\_STRLEN\_P(op1); i++) { // 遍历每个字符  Z\_STRVAL\_P(result)[i] = ~Z\_STRVAL\_P(op1)[i]; // 逐个字符取反  }  Z\_STRVAL\_P(result)[i] = 0; // 最后再加个 \0  }  return SUCCESS; // 返回成功  }  case IS\_REFERENCE: // 如果是引用类型  op1 = Z\_REFVAL\_P(op1); // 追踪到引用目标  goto try\_again; // 从头再来  default: // 其他情况  // 调用do\_operation（）方法，成功时返回成功  ZEND\_TRY\_UNARY\_OBJECT\_OPERATION(ZEND\_BW\_NOT);  if (result != op1) { // 如果不用op1接收结果  ZVAL\_UNDEF(result); // 清空接收变量，中断操作  }  // 报错：不可以在此类型上进行位运算  zend\_type\_error("Cannot perform bitwise not on %s", zend\_zval\_type\_name(op1));  return FAILURE; // 返回失败  }  } |

### “二进制取反”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_BOOL\_NOT操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HOT\_NOCONST\_HANDLER(13, ZEND\_BW\_NOT, CONST|TMPVARCV, ANY) {  USE\_OPLINE // const zend\_op \*opline = EX(opline);  zval \*op1;  op1 = GET\_OP1\_ZVAL\_PTR\_UNDEF(BP\_VAR\_R); // 获取zval指针, UNUSED 返回null  if (EXPECTED(Z\_TYPE\_INFO\_P(op1) == IS\_LONG)) { // 如果 变量类型为 整数  ZVAL\_LONG(EX\_VAR(opline->result.var), ~Z\_LVAL\_P(op1)); // 整数直接 用【反】运算  ZEND\_VM\_NEXT\_OPCODE(); // 下一个操作码  }  // 其他情况由助手处理  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_bw\_not\_helper, op\_1, op1);  } |

如上所示，如果值为整数，可以直接取反，否则调用助手函数zend\_bw\_not\_helper\_SPEC()进行处理，此伪函数会根据两个操作对象的变量类型，被转换成3个处理器函数：

1）ZEND\_BW\_NOT\_SPEC\_CV\_HANDLER()函数，用于处理编译变量，例如“^$a”；

2）ZEND\_BW\_NOT\_SPEC\_CONST\_HANDLER()函数，用于处理常量，例如“^true”；

3）ZEND\_BW\_NOT\_SPEC\_TMPVAR\_HANDLER()函数，用于处理（普通或临时）变量，例如“^$GLOBALS['a']”。

zend\_bw\_not\_helper\_SPEC()函数定义如下：

|  |
| --- |
| static zend\_never\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_bw\_not\_helper\_SPEC(zval \*op\_1 ZEND\_OPCODE\_HANDLER\_ARGS\_DC) {  USE\_OPLINE  SAVE\_OPLINE();  if (UNEXPECTED(Z\_TYPE\_P(op\_1) == IS\_UNDEF)) {  op\_1 = ZVAL\_UNDEFINED\_OP1(); // 获取op1的变量，失败时报错  }  bitwise\_not\_function(EX\_VAR(opline->result.var), op\_1);  FREE\_OP(opline->op1\_type, opline->op1.var);  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION();  } |

如上所示，仍然是调用bitwise\_not\_function()函数进行计算。

# 七、自增和自减

## 一）解析源码，创建语句

自增和自减运算也只针对一个变量进行操作，与之相关的语句有4个，分别是前自增（ZEND\_AST\_PRE\_INC）、后自增（ZEND\_AST\_POST\_INC）、前自减（ZEND\_AST\_PRE\_DEC）、后自减（ZEND\_AST\_POST\_DEC）。

在zend\_language\_parser.y中定义的PHP语法如下：

|  |
| --- |
| expr:  ...  | variable T\_INC { $$ = zend\_ast\_create(ZEND\_AST\_POST\_INC, $1); }  | T\_INC variable { $$ = zend\_ast\_create(ZEND\_AST\_PRE\_INC, $2); }  | variable T\_DEC { $$ = zend\_ast\_create(ZEND\_AST\_POST\_DEC, $1); }  | T\_DEC variable { $$ = zend\_ast\_create(ZEND\_AST\_PRE\_DEC, $2); } |

其中前自增和前自减操作会返回操作后的数值，后自增和后自减会返回操作前的数值，PHP代码示例如下：

|  |
| --- |
| <?php  $a=1;  echo ++$a; // 创建前自增（ZEND\_AST\_PRE\_INC）语句，返回自增后的数值，输出结果为2  echo $a++; // 创建后自增（ZEND\_AST\_POST\_INC）语句，返回自增前的数值，输出结果为2  echo $a--; // 创建前自减（ZEND\_AST\_PRE\_DEC）语句，返回自减前的数值，输出结果为3  echo --$a; // 创建后自减（ZEND\_AST\_POST\_DEC）语句，返回自减后的数值，输出结果为1 |

先从前自增（ZEND\_AST\_PRE\_INC）语句开始介绍。

## 二）前自增（ZEND\_PRE\_INC）

### 在编译时计算

编译ZEND\_AST\_PRE\_INC语句需要用到zend\_compile\_pre\_incdec()函数，调用路径如下：

|  |
| --- |
| zend\_compile\_stmt() // 编译语句  ->zend\_compile\_expr() // 编译表达式  ->zend\_compile\_expr\_inner() // 根据不同的表达式类型，调用不同的方法处理  ->zend\_compile\_pre\_incdec() |

zend\_compile\_expr\_inner()函数中的相关业务逻辑如下：

|  |
| --- |
| static void zend\_compile\_expr\_inner(znode \*result, zend\_ast \*ast) {  ...  switch (ast->kind) { // 支持的类型  ...  case ZEND\_AST\_PRE\_INC: // 前自增  case ZEND\_AST\_PRE\_DEC: // 前自减  zend\_compile\_pre\_incdec(result, ast);  return; |

如上所示，编译前自增（ZEND\_AST\_PRE\_INC）语句和前自减（ZEND\_AST\_PRE\_DEC）语句需要用到zend\_compile\_pre\_incdec()函数。

#### **zend\_compile\_pre\_incdec()函数**

zend\_compile\_pre\_incdec()函数定义如下：

|  |
| --- |
| static void zend\_compile\_pre\_incdec(znode \*result, zend\_ast \*ast) {  zend\_ast \*var\_ast = ast->child[0];  zend\_ensure\_writable\_variable(var\_ast); // 确定变量可写  // 如果是类属性  if (var\_ast->kind == ZEND\_AST\_PROP || var\_ast->kind == ZEND\_AST\_NULLSAFE\_PROP) {  ...  } else if (var\_ast->kind == ZEND\_AST\_STATIC\_PROP) { // 如果是类的静态属性  ...  } else { **// 其他情况**  znode var\_node;  // 编译变量，p1:返回值，p2:语句，p3:操作类型，p4:是否引用传递  zend\_op \*opline = zend\_compile\_var(&var\_node, var\_ast, BP\_VAR\_RW, 0);  // 如果操作码类型是数组赋值  if (opline && opline->opcode == ZEND\_FETCH\_DIM\_RW) {  opline->extended\_value = ZEND\_FETCH\_DIM\_INCDEC; // 添加扩展信息  }  // 创建操作码 ZEND\_PRE\_INC 或 ZEND\_PRE\_DEC  zend\_emit\_op\_tmp(result, ast->kind == ZEND\_AST\_PRE\_INC ? ZEND\_PRE\_INC : ZEND\_PRE\_DEC,  &var\_node, NULL);  }  } |

如上所示，函数的业务逻辑中囊括了成员属性、静态成员属性、数组元素和其他类型的前自增和自减操作业务逻辑可归纳如下：

一、如果操作目标是成员属性：在面向对象部分介绍；

二、如果操作目标是成员静态属性：在面向对象部分介绍；

三、其他情况：

1）先编译此变量；

2）如果是数组元素，添加ZEND\_FETCH\_DIM\_INCDEC扩展标记；

3）创建操作码ZEND\_PRE\_INC或ZEND\_PRE\_DEC。

zend\_compile\_var()函数的介绍请在附录中查看。

### “前自增”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_PRE\_INC操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HOT\_HANDLER(34, ZEND\_PRE\_INC, VAR|CV, ANY, SPEC(RETVAL)) {  ... // 获取op1中的变量  if (EXPECTED(Z\_TYPE\_P(var\_ptr) == IS\_LONG)) { // 变量是整数  fast\_long\_increment\_function(var\_ptr); // 计算自增  // 如果 操作码的运算结果有效(不是IS\_UNUSED)  if (UNEXPECTED(RETURN\_VALUE\_USED(opline))) {  ZVAL\_COPY\_VALUE(EX\_VAR(opline->result.var), var\_ptr); // 值复制给 结果  }  ZEND\_VM\_NEXT\_OPCODE(); // 下一个操作码  }  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_pre\_inc\_helper); // 调用助手函数  } |

如上所示，业务逻辑比较简单：如果操作对象是整数，调用fast\_long\_increment\_function()进行处理，按要求应用自增后的结果；否则调用助手函数进行处理。由于添加了RETVAL扩展标记，此伪函数会根据操作对象的变量类型，被转换成4个处理器函数：

1）ZEND\_PRE\_INC\_SPEC\_VAR\_RETVAL\_UNUSED\_HANDLER()函数，对普通变量进行自增，不接收结果，例如“++$GLOBALS['a']”；

2）ZEND\_PRE\_INC\_SPEC\_VAR\_RETVAL\_USED\_HANDLER()函数，对普通变量进行自增，接收结果，例如“$a = ++$GLOBALS['a']”；

3）ZEND\_PRE\_INC\_SPEC\_CV\_RETVAL\_UNUSED\_HANDLER()函数，对编译变量进行自增，不接收结果，例如“++$a”；

4）ZEND\_PRE\_INC\_SPEC\_CV\_RETVAL\_USED\_HANDLER()函数，对编译变量进行自增，接收结果，例如“$b = ++$a”。

#### **RETVAL标记与返回结果**

如上所示，RETVAL扩展标记会给每个分支添加两种类型\_UNUSED或\_USED，生成的函数业务逻辑有些不同，以ZEND\_PRE\_INC\_SPEC\_CV\_RETVAL\_UNUSED\_HANDLER()函数为例，它在不接收返回结果时被调用，代码如下：

|  |
| --- |
| static ZEND\_VM\_HOT ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL ZEND\_PRE\_INC\_SPEC\_CV\_RETVAL\_UNUSED\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS){  ... // 获取op1的变量  if (EXPECTED(Z\_TYPE\_P(var\_ptr) == IS\_LONG)) {  fast\_long\_increment\_function(var\_ptr);  **if (UNEXPECTED(0)) { // RETURN\_VALUE\_USED(opline) 被替换成了0**  ZVAL\_COPY\_VALUE(EX\_VAR(opline->result.var), var\_ptr);  }  ... |

与之相对的ZEND\_PRE\_INC\_SPEC\_CV\_RETVAL\_USED\_HANDLER()函数在接收返回结果时被调用，代码如下：

|  |
| --- |
| static ZEND\_VM\_HOT ZEND\_OPCODE\_HANDLER\_RET ZEND\_PRE\_INC\_SPEC\_CV\_RETVAL\_USED\_HANDLER(ZEND\_OPCODE\_HANDLER\_ARGS){  ... // 获取op1的变量  if (EXPECTED(Z\_TYPE\_P(var\_ptr) == IS\_LONG)) {  fast\_long\_increment\_function(var\_ptr);  **if (UNEXPECTED(1)) { // RETURN\_VALUE\_USED(opline) 被替换成了1**  ZVAL\_COPY\_VALUE(EX\_VAR(opline->result.var), var\_ptr);  }  ... |

如上所示，通过把伪代码RETURN\_VALUE\_USED(opline)替换成0和1，实现了对返回结果的控制。“不接收结果”是指返回结果不会被用到，也就是说不能和赋值语句一起用，不可以和函数调用一起用，也不可以被echo，示例如下：

|  |
| --- |
| <?php  $a=1;  ++$a; // 不接收返回结果，ZEND\_PRE\_INC\_SPEC\_CV\_RETVAL\_UNUSED\_HANDLER()函数处理  // 以下都算接收返回结果，由ZEND\_PRE\_INC\_SPEC\_CV\_RETVAL\_USED\_HANDLER()函数处理  $b = ++$a;  print(++$a);  echo ++$a; |

#### **fast\_long\_increment\_function()函数**

fast\_long\_increment\_function()函数在zend\_operators.h中定义，用于计算整数的自增，此函数的业务逻辑根据不同的运行环境，有多个分支，在64位windows中业务逻辑如下：

|  |
| --- |
| static zend\_always\_inline void fast\_long\_increment\_function(zval \*op1){  if (UNEXPECTED(Z\_LVAL\_P(op1) == ZEND\_LONG\_MAX)) { // 如果是最大整数  ZVAL\_DOUBLE(op1, (double)ZEND\_LONG\_MAX + 1.0); // +1并转成小数  } else { // 不是最大整数  Z\_LVAL\_P(op1)++; // 自增  }  } |

#### **助手函数zend\_pre\_inc\_helper**

助手函数zend\_pre\_inc\_helper的伪代码如下：

|  |
| --- |
| ZEND\_VM\_HELPER(zend\_pre\_inc\_helper, VAR|CV, ANY) {  ... // 取得op1中的变量，失败时报错  do {  if (UNEXPECTED(Z\_TYPE\_P(var\_ptr) == IS\_REFERENCE)) { // 如果原值是引用类型  zend\_reference \*ref = Z\_REF\_P(var\_ptr); // 取出引用实例  var\_ptr = Z\_REFVAL\_P(var\_ptr); // 解引用  if (UNEXPECTED(ZEND\_REF\_HAS\_TYPE\_SOURCES(ref))) { // 如果引用实例 有 类型  // 给带类型的引用目标作 自增 或 自减。p1:引用对象，p2:返回值  zend\_incdec\_typed\_ref(ref, NULL OPLINE\_CC EXECUTE\_DATA\_CC);  break; // 跳过自增方法  }  }  // 不是引用类型，自增运算  increment\_function(var\_ptr);  } while (0);  if (UNEXPECTED(RETURN\_VALUE\_USED(opline))) {  ZVAL\_COPY(EX\_VAR(opline->result.var), var\_ptr); // 运算结果 复制 给结果变量  }  FREE\_OP1(); // 释放操作对象的附加变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  } |

如上所示，由于定义了op1的两种类型（VAR|CV）此伪函数会被创建成两个函数：

1）zend\_pre\_inc\_helper\_SPEC\_CV()，用于处理编译变量的自增操作，例如“++$a”；

2）zend\_pre\_inc\_helper\_SPEC\_VAR()，用于处理普通变量的自增操作，例如“++$GLOBALS['a']”。

前文中的4个处理器函数会根据各自的处理类型，来调用合适的处理函数。

在这两个助手函数中，RETURN\_VALUE\_USED(opline)语句并没有被替换成0或1，而是真实调用RETURN\_VALUE\_USED()宏程序，以zend\_pre\_inc\_helper\_SPEC\_CV()函数为例，生成代码如下：

|  |
| --- |
| static zend\_never\_inline ZEND\_OPCODE\_HANDLER\_RET ZEND\_FASTCALL zend\_pre\_inc\_helper\_SPEC\_CV(ZEND\_OPCODE\_HANDLER\_ARGS){  ...  **if (UNEXPECTED(RETURN\_VALUE\_USED(opline))) {**  ZVAL\_COPY(EX\_VAR(opline->result.var), var\_ptr);  }  ... |

RETURN\_VALUE\_USED()宏程序在zend\_execute.c中，定义如下：

|  |
| --- |
| // 检验：opline 返回类型不是IS\_UNUSED  #define RETURN\_VALUE\_USED(opline) ((opline)->result\_type != IS\_UNUSED) |

关于处理器函数和助手函数的创建过程，可参见“虚拟机生成器篇”。

zend\_incdec\_typed\_ref()函数用于对指定类型的成员变量进行自减和自增操作，详情可参见面向对向相关章节。

#### **increment\_function()函数**

increment\_function()函数用于对各种类型进行自增运算，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL increment\_function(zval \*op1) {  try\_again:  switch (Z\_TYPE\_P(op1)) { // 根据类型操作  case IS\_LONG: // 整数 ，快速自增  fast\_long\_increment\_function(op1); break;  case IS\_DOUBLE: // 小数 +1  Z\_DVAL\_P(op1) = Z\_DVAL\_P(op1) + 1; break;  case IS\_NULL: // null 变成1 （测试过）  ZVAL\_LONG(op1, 1); break;  case IS\_STRING: { // 字串  zend\_long lval;  double dval;  switch (is\_numeric\_str\_function(Z\_STR\_P(op1), &lval, &dval)) { // 转成数值  case IS\_LONG: // 如果成整数  zval\_ptr\_dtor\_str(op1); // 清空 op1  if (lval == ZEND\_LONG\_MAX) { // 如果是最大整数  double d = (double)lval; // 转成小数  ZVAL\_DOUBLE(op1, d+1); // +1后保存成小数  } else { // 普通整数  ZVAL\_LONG(op1, lval+1); // +1  }  break;  case IS\_DOUBLE: // 小数  zval\_ptr\_dtor\_str(op1); // 清空 op1  ZVAL\_DOUBLE(op1, dval+1); // +1后保存成小数  break;  default: // 其他情况，perl 风格的字串自增  /\* Perl style string increment \*/  increment\_string(op1); break;  }  }  break;  case IS\_FALSE: // false,true，不能自增（测试过）  case IS\_TRUE: break;  case IS\_REFERENCE: // 如果是引用  op1 = Z\_REFVAL\_P(op1); // 追踪到引用目标  goto try\_again; // 从头再来  case IS\_OBJECT: // 对象  if (Z\_OBJ\_HANDLER\_P(op1, do\_operation)) { // 如果有 do\_operation方法  zval op2;  ZVAL\_LONG(&op2, 1); // 临时数值1，用来做加法  if (Z\_OBJ\_HANDLER\_P(op1, do\_operation)(ZEND\_ADD, op1, op1, &op2) == SUCCESS) { // 计算加法，成功时返回成功  return SUCCESS;  }  }  ZEND\_FALLTHROUGH;  case IS\_RESOURCE: // 资源或数组，报错，不可自增  case IS\_ARRAY:  zend\_type\_error("Cannot increment %s", zend\_zval\_type\_name(op1));  return FAILURE;  EMPTY\_SWITCH\_DEFAULT\_CASE()  }  return SUCCESS;  } |

increment\_string()函数的业务逻辑请在附录中查看。

#### **is\_numeric\_str\_function()函数**

is\_numeric\_str\_function()函数用于把数字字串转成数字，代码如下：

|  |
| --- |
| // 验证是否是 数字字串。接收转换后的值，  ZEND\_API zend\_uchar ZEND\_FASTCALL is\_numeric\_str\_function(const zend\_string \*str, zend\_long \*lval, double \*dval) {  // -> is\_numeric\_string\_ex -> \_is\_numeric\_string\_ex  // allow\_errors = false, 有错时返回0  return is\_numeric\_string(ZSTR\_VAL(str), ZSTR\_LEN(str), lval, dval, false);  } |

is\_numeric\_string()函数定义如下：

|  |
| --- |
| static zend\_always\_inline zend\_uchar is\_numeric\_string(const char \*str, size\_t length, zend\_long \*lval, double \*dval, bool allow\_errors) {  // int \*oflow\_info, bool \*trailing\_data 这两个参数是null  return is\_numeric\_string\_ex(str, length, lval, dval, allow\_errors, NULL, NULL);  } |

is\_numeric\_string\_ex()函数的业务逻辑请在附录中查看。

## 三）前自减（ZEND\_PRE\_DEC）

上一节中已经介绍过，编译前自减（ZEND\_AST\_PRE\_DEC）语句需要用到zend\_compile\_pre\_incdec()函数，在进行前自减操作时，会创建ZEND\_PRE\_DEC操作码。

### “前自减”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_PRE\_DEC操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HOT\_HANDLER(34, ZEND\_PRE\_INC, VAR|CV, ANY, SPEC(RETVAL)) {  ... // 获取op1中的变量  if (EXPECTED(Z\_TYPE\_P(var\_ptr) == IS\_LONG)) { // 变量是整数  fast\_long\_decrement\_function(var\_ptr); // 计算自减  // 如果 操作码的运算结果有效(不是IS\_UNUSED)  if (UNEXPECTED(RETURN\_VALUE\_USED(opline))) {  ZVAL\_COPY\_VALUE(EX\_VAR(opline->result.var), var\_ptr); // 值复制给 结果  }  ZEND\_VM\_NEXT\_OPCODE(); // 下一个操作码  }  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_pre\_inc\_helper); // 调用助手函数  } |

如上所示，业务逻辑比较简单：如果操作对象是整数，调用fast\_long\_decrement\_function()进行处理，按要求应用自减后的结果；否则调用助手函数进行处理。由于添加了RETVAL扩展标记，此伪函数会根据操作对象的变量类型，被转换成4个处理器函数：

1）ZEND\_PRE\_DEC\_SPEC\_VAR\_RETVAL\_UNUSED\_HANDLER()函数，对普通变量进行自增，不接收结果，例如“--$GLOBALS['a']”；

2）ZEND\_PRE\_DEC\_SPEC\_VAR\_RETVAL\_USED\_HANDLER()函数，对普通变量进行自增，接收结果，例如“$a = --$GLOBALS['a']”；

3）ZEND\_PRE\_DEC\_SPEC\_CV\_RETVAL\_UNUSED\_HANDLER()函数，对编译变量进行自增，不接收结果，例如“--$a”；

4）ZEND\_PRE\_DEC\_SPEC\_CV\_RETVAL\_USED\_HANDLER()函数，对编译变量进行自增，接收结果，例如“$b = --$a”。

#### **fast\_long\_decrement\_function()函数**

fast\_long\_decrement\_function()函数在zend\_operators.h中定义，用于计算整数的自增，此函数的业务逻辑根据不同的运行环境，有多个分支，在64位windows中业务逻辑如下：

|  |
| --- |
| static zend\_always\_inline void fast\_long\_decrement\_function(zval \*op1){  if (UNEXPECTED(Z\_LVAL\_P(op1) == ZEND\_LONG\_MIN)) { // 如果是最小整数  ZVAL\_DOUBLE(op1, (double)ZEND\_LONG\_MIN - 1.0); // 转成小数再 -1  } else { // 如果不是，直接-1  Z\_LVAL\_P(op1)--;  }} |

#### **助手函数zend\_pre\_dec\_helper**

助手函数zend\_pre\_dec\_helper的伪代码如下：

|  |
| --- |
| ZEND\_VM\_HELPER(zend\_pre\_dec\_helper, VAR|CV, ANY){  ... // 取得op1中的变量，失败时报错  do {  if (UNEXPECTED(Z\_TYPE\_P(var\_ptr) == IS\_REFERENCE)) { // 如果原值是引用类型  zend\_reference \*ref = Z\_REF\_P(var\_ptr); // 取出引用 实例  var\_ptr = Z\_REFVAL\_P(var\_ptr); // 解引用  if (UNEXPECTED(ZEND\_REF\_HAS\_TYPE\_SOURCES(ref))) { // 如果引用实例 有 类型  // 给带类型的引用目标作 自增 或 自减。p1:引用对象，p2:返回值  zend\_incdec\_typed\_ref(ref, NULL OPLINE\_CC EXECUTE\_DATA\_CC);  break; // 跳过自减方法  }  }  decrement\_function(var\_ptr); // 自减运算  } while (0);  if (UNEXPECTED(RETURN\_VALUE\_USED(opline))) { // 如果 操作码的运算结果有效  ZVAL\_COPY(EX\_VAR(opline->result.var), var\_ptr); // 运算结果 复制 给结果变量  }  FREE\_OP1(); // 释放操作对象的附加变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  } |

如上所示，由于定义了op1的两种类型（VAR|CV）此伪函数会被创建成两个函数：

1）zend\_pre\_dec\_helper\_SPEC\_CV()，用于处理编译变量的自增操作，例如“--$a”；

2）zend\_pre\_dec\_helper\_SPEC\_VAR()，用于处理普通变量的自增操作，例如“--$GLOBALS['a']”。

#### **decrement\_function()函数**

decrement\_function()函数用于对各种类型进行自减运算，代码如下：

|  |
| --- |
| ZEND\_API zend\_result ZEND\_FASTCALL decrement\_function(zval \*op1) {  zend\_long lval;  double dval;  try\_again:  switch (Z\_TYPE\_P(op1)) { // 按类型操作  case IS\_LONG: // 整数，快速自减  fast\_long\_decrement\_function(op1); break;  case IS\_DOUBLE: // 小数，直接算 -1  Z\_DVAL\_P(op1) = Z\_DVAL\_P(op1) - 1; break;  // 字串。和perl一样 只支持字串自减  case IS\_STRING: /\* Like perl we only support string increment \*/  if (Z\_STRLEN\_P(op1) == 0) { // 如果长度为0  zval\_ptr\_dtor\_str(op1); // 清空  ZVAL\_LONG(op1, -1); // 结果为 -1  break;  }  switch (is\_numeric\_str\_function(Z\_STR\_P(op1), &lval, &dval)) { // 转成整数  case IS\_LONG: // 整数  zval\_ptr\_dtor\_str(op1); // 清空op1  if (lval == ZEND\_LONG\_MIN) { // 等于最小整数  double d = (double)lval; // 转成小数  ZVAL\_DOUBLE(op1, d-1); // -1，返回值为小数  } else { // 可以算减法  ZVAL\_LONG(op1, lval-1); // -1  }  break;  case IS\_DOUBLE: // 小数  zval\_ptr\_dtor\_str(op1); // 清空op1  ZVAL\_DOUBLE(op1, dval - 1); // -1  break;  }  // 不是数字的字串，自减无效（测试过）  break;  case IS\_NULL: // null,false,true. 自减无效.（测试过）  case IS\_FALSE:  case IS\_TRUE: break;  case IS\_REFERENCE: // 引用类型  op1 = Z\_REFVAL\_P(op1); // 追踪到引用对象  goto try\_again; // 从头再来  case IS\_OBJECT: // 对象  if (Z\_OBJ\_HANDLER\_P(op1, do\_operation)) { // 如果有 do\_operation  zval op2;  ZVAL\_LONG(&op2, 1); // 临时变量值为整数 1  if (Z\_OBJ\_HANDLER\_P(op1, do\_operation)(ZEND\_SUB, op1, op1, &op2) == SUCCESS) { // 按-1计算  return SUCCESS; // 成功  }  }  ZEND\_FALLTHROUGH; // 失败  case IS\_RESOURCE: // 资源或数组，抛错，不可做自减  case IS\_ARRAY:  zend\_type\_error("Cannot decrement %s", zend\_zval\_type\_name(op1));  return FAILURE;  EMPTY\_SWITCH\_DEFAULT\_CASE()  }  return SUCCESS;  } |

is\_numeric\_str\_function()函数已经在前文中介绍过。

## 四）后自增（ZEND\_POST\_INC）

### 在编译时计算

编译ZEND\_AST\_PRE\_INC语句需要用到zend\_compile\_pre\_incdec()函数，调用路径如下：

|  |
| --- |
| zend\_compile\_stmt() // 编译语句  ->zend\_compile\_expr() // 编译表达式  ->zend\_compile\_expr\_inner() // 根据不同的表达式类型，调用不同的方法处理  ->zend\_compile\_post\_incdec() |

zend\_compile\_expr\_inner()函数中的相关业务逻辑如下：

|  |
| --- |
| static void zend\_compile\_expr\_inner(znode \*result, zend\_ast \*ast) {  ...  switch (ast->kind) { // 支持的类型  ...  case ZEND\_AST\_POST\_INC: // 后自增  case ZEND\_AST\_POST\_DEC: // 后自减  zend\_compile\_post\_incdec(result, ast);  return; |

如上所示，编译前自增（ZEND\_AST\_POST\_INC）语句和前自减（ZEND\_AST\_POST\_DEC）语句需要用到zend\_compile\_post\_incdec()函数。

#### **zend\_compile\_post\_incdec()函数**

zend\_compile\_post\_incdec()函数定义如下：

|  |
| --- |
| static void zend\_compile\_post\_incdec(znode \*result, zend\_ast \*ast) {  zend\_ast \*var\_ast = ast->child[0];  zend\_ensure\_writable\_variable(var\_ast); // 确定变量可写  // 如果是类属性  if (var\_ast->kind == ZEND\_AST\_PROP || var\_ast->kind == ZEND\_AST\_NULLSAFE\_PROP) {  ...  } else if (var\_ast->kind == ZEND\_AST\_STATIC\_PROP) { // 如果是静态属性  ...  } else { // 其他情况  znode var\_node;  // 编译变量，p1:返回值，p2:语句，p3:操作类型，p4:是否引用传递  zend\_op \*opline = zend\_compile\_var(&var\_node, var\_ast, BP\_VAR\_RW, 0);  if (opline && opline->opcode == ZEND\_FETCH\_DIM\_RW) {  opline->extended\_value = ZEND\_FETCH\_DIM\_INCDEC; // 添加扩展信息  }  // 创建操作码 ZEND\_PRE\_INC 或 ZEND\_PRE\_DEC  zend\_emit\_op\_tmp(result, ast->kind == ZEND\_AST\_POST\_INC ? ZEND\_POST\_INC : ZEND\_POST\_DEC,  &var\_node, NULL);  }  } |

如上所示，函数的业务逻辑中囊括了成员属性、静态成员属性、数组元素和其他类型的后自增和自减操作业务逻辑可归纳如下：

一、如果操作目标是成员属性：在面向对象部分介绍；

二、如果操作目标是成员静态属性：在面向对象部分介绍；

三、其他情况：

1）先编译此变量；

2）如果是数组元素，添加ZEND\_FETCH\_DIM\_INCDEC扩展标记；

3）创建操作码ZEND\_POST\_INC或ZEND\_POST\_DEC。

zend\_compile\_var()函数的介绍请在附录中查看。

### “后自增”操作码的处理器

在Zend/zend\_vm\_def.h文件中对ZEND\_POST\_INC操作码的处理器定义如下：

|  |
| --- |
| ZEND\_VM\_HOT\_HANDLER(36, ZEND\_POST\_INC, VAR|CV, ANY){  ... // 取得op1中的变量  if (EXPECTED(Z\_TYPE\_P(var\_ptr) == IS\_LONG)) { // 如果， op1是整数  ZVAL\_LONG(EX\_VAR(opline->result.var), Z\_LVAL\_P(var\_ptr)); // 先保存进结果  fast\_long\_increment\_function(var\_ptr); // 自增运算  ZEND\_VM\_NEXT\_OPCODE(); // 下一个操作码  }  ZEND\_VM\_DISPATCH\_TO\_HELPER(zend\_post\_inc\_helper); // 调用助手  } |

如上所示，业务逻辑比较简单：如果操作对象是整数，调用fast\_long\_increment\_function()进行处理；否则调用助手函数进行处理。此伪函数会根据操作对象的变量类型，被转换成2个处理器函数：

1）ZEND\_POST\_INC\_SPEC\_VAR\_HANDLER()函数，对普通变量进行自增，不接收结果，例如“$GLOBALS['a']++”；

3）ZEND\_POST\_INC\_SPEC\_CV\_HANDLER()函数，对编译变量进行自增，不接收结果，例如“$a++”。

#### **助手函数zend\_post\_inc\_helper**

助手函数zend\_post\_inc\_helper的伪代码如下：

|  |
| --- |
| ZEND\_VM\_HELPER(zend\_post\_inc\_helper, VAR|CV, ANY) {  ... // 取得op1中的变量，失败时报错  do {  if (UNEXPECTED(Z\_TYPE\_P(var\_ptr) == IS\_REFERENCE)) { // 如果原值是引用类型  zend\_reference \*ref = Z\_REF\_P(var\_ptr); // 取出引用实例  var\_ptr = Z\_REFVAL\_P(var\_ptr); // 解引用  if (UNEXPECTED(ZEND\_REF\_HAS\_TYPE\_SOURCES(ref))) { // 如果引用实例 有 类型  // 给带类型的引用目标作 自增 或 自减。p1:引用对象，p2:返回值  zend\_incdec\_typed\_ref(ref, NULL OPLINE\_CC EXECUTE\_DATA\_CC);  break; // 跳过自增方法  }  }  // 不是引用类型，自增运算  increment\_function(var\_ptr);  } while (0);  if (UNEXPECTED(RETURN\_VALUE\_USED(opline))) {  ZVAL\_COPY(EX\_VAR(opline->result.var), var\_ptr); // 运算结果 复制 给结果变量  }  FREE\_OP1(); // 释放操作对象的附加变量  ZEND\_VM\_NEXT\_OPCODE\_CHECK\_EXCEPTION(); // 下一个操作码  } |

# 待编目部分

### 第6步：计算操作码活动区域

zend\_calc\_live\_ranges()函数用于计算操作码的活动区域：

|  |
| --- |
| zend\_calc\_live\_ranges()  ->is\_fake\_def()  ->EX\_VAR\_TO\_NUM()  ->keeps\_op1\_alive()  ->emit\_live\_range()  ->emit\_live\_range\_raw() |

zend\_calc\_live\_ranges()函数：

|  |
| --- |
| // 计算生成 活动区域。p1:操作码组，p2:zend\_needs\_live\_range\_cb  static void zend\_calc\_live\_ranges(zend\_op\_array \*op\_array, f needs\_live\_range) {  uint32\_t opnum = op\_array->last; // 操作码数量  zend\_op \*opline = &op\_array->opcodes[opnum]; // 最后一个操作码  ALLOCA\_FLAG(use\_heap) // bool use\_heap;  uint32\_t var\_offset = op\_array->last\_var; // 变量数量  // 分配一串32位整数，和临时变量个数相同  uint32\_t \*last\_use = do\_alloca(sizeof(uint32\_t) \* op\_array->T, use\_heap);  memset(last\_use, -1, sizeof(uint32\_t) \* op\_array->T); // 所有位都写成1  while (opnum > 0) { // 倒着遍历  opnum--; // 前一个操作码序号  opline--; // 前一个操作码  // 如果结果类型是变量或临时变量，并且操作码会创建结果  if ((opline->result\_type & (IS\_TMP\_VAR|IS\_VAR)) && !is\_fake\_def(opline)) {  //  uint32\_t var\_num = EX\_VAR\_TO\_NUM(opline->result.var) - var\_offset;  //  if (EXPECTED(last\_use[var\_num] != (uint32\_t) -1)) {  // 跳过琐碎的活动区域  if (opnum + 1 != last\_use[var\_num]) { //  uint32\_t num; // 临时变量  num = opnum; // 暂存编号  // 发布活动空间。p1:操作码组，p2:变量编号，p3:开始位置，p4:结束位置，p5:zend\_needs\_live\_range\_cb  emit\_live\_range(op\_array, var\_num, num, last\_use[var\_num], needs\_live\_range);  }  last\_use[var\_num] = (uint32\_t) -1; // 最后使用编号 为 -1  }  }  // 如果第一个操作对象类型是 变量或临时变量  if ((opline->op1\_type & (IS\_TMP\_VAR|IS\_VAR))) {  // 取得这个变量的编号  uint32\_t var\_num = EX\_VAR\_TO\_NUM(opline->op1.var) - var\_offset;  // 如果最后使用是 -1  if (EXPECTED(last\_use[var\_num] == (uint32\_t) -1)) {  // 类型转换，march, 列表读取，copy ,这几种情况返回1。 如果失败  if (EXPECTED(!keeps\_op1\_alive(opline))) {  // 最后使用编号，如果是 ZEND\_OP\_DATA 操作码要 -1  last\_use[var\_num] = opnum - (opline->opcode == ZEND\_OP\_DATA);  }  }  }  // 第二个运算对象类型是 临时变量 或 变量  if (opline->op2\_type & (IS\_TMP\_VAR|IS\_VAR)) {  // 计算变量序号  uint32\_t var\_num = EX\_VAR\_TO\_NUM(opline->op2.var) - var\_offset;  // 如果，操作码是 ZEND\_FE\_FETCH\_R 或 ZEND\_FE\_FETCH\_RW  if (UNEXPECTED(opline->opcode == ZEND\_FE\_FETCH\_R || opline->opcode == ZEND\_FE\_FETCH\_RW)) {  // 如果此位置值 不是 -1  if (last\_use[var\_num] != (uint32\_t) -1) {  // 如果此位置值不是 操作码序号+1  if (opnum + 1 != last\_use[var\_num]) {  // 发布活动空间。p1:操作码组，p2:变量编号，p3:开始位置，p4:结束位置，p5:zend\_needs\_live\_range\_cb  emit\_live\_range(op\_array, var\_num, opnum, last\_use[var\_num], needs\_live\_range);  }  last\_use[var\_num] = (uint32\_t) -1; // 最后使用编号为 -1  }  // 如果此位置值为 -1  } else if (EXPECTED(last\_use[var\_num] == (uint32\_t) -1)) {  // 记录变量编号对象的操作码编号  last\_use[var\_num] = opnum;  }  }  }  // 如果活动区域大于1个  if (op\_array->last\_live\_range > 1) {  zend\_live\_range \*r1 = op\_array->live\_range; // 第一个活动区域  // 最后一个活动区域  zend\_live\_range \*r2 = r1 + op\_array->last\_live\_range - 1;  // 大部分情况下，只要翻转数组就可以了。从两头到中间，两两比较  while (r1 < r2) {  swap\_live\_range(r1, r2); // 把指针小的放在后面  r1++; // 左面的向右移  r2--; // 右向的向左移  }  r1 = op\_array->live\_range; // 区域列表开头位置  r2 = r1 + op\_array->last\_live\_range - 1; // 区域列表结尾位置  while (r1 < r2) { // 遍历这段操作码  if (r1->start > (r1+1)->start) { // 如果前面的区域操作码放在了后面  // 给所有区域排序（快排加插入）  zend\_sort(r1, r2 - r1 + 1, sizeof(zend\_live\_range),  (compare\_func\_t) cmp\_live\_range, (swap\_func\_t) swap\_live\_range);  break; // 排完跳出  }  r1++; // 检查下一个区域  }  }  free\_alloca(last\_use, use\_heap); // 释放临时内存  } |

is\_fake\_def()函数：

|  |
| --- |
| static bool is\_fake\_def(zend\_op \*opline) { // 这些操作码都是修改结果，不创建结果  return opline->opcode == ZEND\_ROPE\_ADD  || opline->opcode == ZEND\_ADD\_ARRAY\_ELEMENT  || opline->opcode == ZEND\_ADD\_ARRAY\_UNPACK;  }  #define EX\_VAR\_TO\_NUM(n) ((uint32\_t)((n) / sizeof(zval) - ZEND\_CALL\_FRAME\_SLOT)) |

keeps\_op1\_alive()函数：

|  |
| --- |
| // switch，match, 列表读取，copy ,这几种情况返回1  static bool keeps\_op1\_alive(zend\_op \*opline) {  // 这些操作码 不使用它们的 op1操作对象，稍后会被其他操作销毁  if (opline->opcode == ZEND\_CASE  || opline->opcode == ZEND\_CASE\_STRICT  || opline->opcode == ZEND\_SWITCH\_LONG  || opline->opcode == ZEND\_SWITCH\_STRING  || opline->opcode == ZEND\_MATCH  || opline->opcode == ZEND\_FETCH\_LIST\_R  || opline->opcode == ZEND\_COPY\_TMP) {  return 1;  }  return 0;  } |

emit\_live\_range()函数：

|  |
| --- |
| // 创建活动区域。p1:操作码组，p2:变量编号，p3:开始位置，p4:结束位置，p5:回调函数  static void emit\_live\_range(zend\_op\_array \*op\_array, uint32\_t var\_num, uint32\_t start, uint32\_t end, zend\_needs\_live\_range\_cb needs\_live\_range) {  // 开始操作码指针  zend\_op \*def\_opline = &op\_array->opcodes[start], \*orig\_def\_opline = def\_opline;  zend\_op \*use\_opline = &op\_array->opcodes[end]; // 结束操作码指针  uint32\_t kind; // 类型  switch (def\_opline->opcode) { // 按操作码处理  ... // 一些操作码的特殊处理    default: // 默认情况  start++; // 开始位置 +1  kind = ZEND\_LIVE\_TMPVAR; // 类型为 临时变量 活动区域  // 如果不需要创建区域（回调函数返回）  if (needs\_live\_range && !needs\_live\_range(op\_array, orig\_def\_opline)) {  return; // 直接返回  }  break; // 跳出  }  emit\_live\_range\_raw(op\_array, var\_num, kind, start, end); // 创建活动区域  } |

emit\_live\_range\_raw()函数用于分配内存创建活动区域实例：

|  |
| --- |
| // 创建活动区域：给新活动区域区域写入 var（+类型）,start,end 3个元素。  // p1:操作码组，p2:变量编号，p3:区域类型，p4:开始位置，p5:结束位置  static void emit\_live\_range\_raw(zend\_op\_array \*op\_array, uint32\_t var\_num, uint32\_t kind, uint32\_t start, uint32\_t end) {  zend\_live\_range \*range; // 活动区域指针  op\_array->last\_live\_range++; // 操作码包含区域数 +1  // 调整内存增加 zend\_live\_range 空间  op\_array->live\_range = erealloc(op\_array->live\_range,sizeof(zend\_live\_range) \* op\_array->last\_live\_range);  range = &op\_array->live\_range[op\_array->last\_live\_range - 1]; // 获得最后一个区域  range->var = EX\_NUM\_TO\_VAR(op\_array->last\_var + var\_num); // 把zval序号转成偏移量  range->var |= kind; // 指针添加类型  range->start = start; // 区域开始序号  range->end = end; // 区域结束序号  } |

zend\_live\_range结构体定义如下：

|  |
| --- |
| typedef struct \_zend\_live\_range { // 操作码活动区域, 4\*3=12Bytes  // 低位用来来存放变量类型 （ZEND\_LIVE\_\*宏）  uint32\_t var; /\* low bits are used for variable type (ZEND\_LIVE\_\* macros) \*/  uint32\_t start; // 活动区域开始位置  uint32\_t end; // 活动区域结束位置  } zend\_live\_range; |

# 附录

## 一）函数

### is\_numeric\_string\_ex()函数

代码如下：

|  |
| --- |
| // 详细说明在 zend\_operators.h里。数字处理想想似乎简单，实际处理起来非常麻烦。下面这些业务逻辑其实只处理了整数 ，小数在 zend\_strtod  // ing3, 验证是否是 数字字串。传入 str字串，length长度，lval接收返回值用的整数，dval接收返回值用的小数，  // allow\_errors是否允许错误，oflow\_info是否溢出，trailing\_data 结尾是否有无效数据  ZEND\_API zend\_uchar ZEND\_FASTCALL \_is\_numeric\_string\_ex(const char \*str, size\_t length, zend\_long \*lval,  double \*dval, bool allow\_errors, int \*oflow\_info, bool \*trailing\_data) /\* {{{ \*/  {  const char \*ptr;  int digits = 0, dp\_or\_e = 0;  double local\_dval = 0.0;  zend\_uchar type;  zend\_ulong tmp\_lval = 0;  int neg = 0;  // 空字串，返回否  if (!length) {  return 0;  }  // 初始化引用返回值，溢出信息。正数溢出，返回1，负数溢出返回-1，无溢出返回0。  if (oflow\_info != NULL) {  \*oflow\_info = 0;  }  // 初始化引用返回值  if (trailing\_data != NULL) {  \*trailing\_data = false;  }  // 跳过开头的空格，这个比 isspace() 快很多  /\* Skip any whitespace  \* This is much faster than the isspace() function \*/  // 如果碰到空字符  while (\*str == ' ' || \*str == '\t' || \*str == '\n' || \*str == '\r' || \*str == '\v' || \*str == '\f') {  // 指针后移  str++;  // 长度-1  length--;  }  // 这时候指针在有效数字的开头  ptr = str;  // 先处理正负号  // 如果字符是 -  if (\*ptr == '-') {  // 负数  neg = 1;  // 下一个字符  ptr++;  // 如果字符是 +  } else if (\*ptr == '+') {  // 下一个字符  ptr++;  }  // #define ZEND\_IS\_DIGIT(c) ((c) >= \'0\' && (c) <= \'9\')  // 如果开头是字符 0-9  if (ZEND\_IS\_DIGIT(\*ptr)) {  // 跳过所有开头的0  /\* Skip any leading 0s \*/  while (\*ptr == '0') {  ptr++;  }  // 计算数字数量。 如果 如果找到小数点或 e(科学计数法)，当成小数处理。  // 其他情况，如果有小数或不需要匹配整个字串， 碰到够长的整数就停止。  /\* Count the number of digits. If a decimal point/exponent is found,  \* it's a double. Otherwise, if there's a dval or no need to check for  \* a full match, stop when there are too many digits for a long \*/  // 类型是整数。逐个字符处理。  for (type = IS\_LONG; !(digits >= MAX\_LENGTH\_OF\_LONG && (dval || allow\_errors)); digits++, ptr++) {  // 检查数字  check\_digits:  // 如果字符是数字  if (ZEND\_IS\_DIGIT(\*ptr)) {  // 计算成整数（前面的\*10，加这最后一位）  tmp\_lval = tmp\_lval \* 10 + (\*ptr) - '0';  // 下一个数字  continue;  // 如果字符是 . 并且 没碰到过小数点和 e （e后面不能再有小数点）  } else if (\*ptr == '.' && dp\_or\_e < 1) {  // 处理小数  goto process\_double;  // 如果字符是 E或e 并且 没碰到过 e  } else if ((\*ptr == 'e' || \*ptr == 'E') && dp\_or\_e < 2) {  // e后面的字符  const char \*e = ptr + 1;  // 如果是正负号  if (\*e == '-' || \*e == '+') {  // 再向右移1个字符  ptr = e++;  }  // 如果碰到数字  if (ZEND\_IS\_DIGIT(\*e)) {  // 处理小数  goto process\_double;  }  }    // 其他情况：碰到无效字符，到此为止，break;  break;  }  // 如果大于最大整数长度（超长会转成小数处理）  if (digits >= MAX\_LENGTH\_OF\_LONG) {  // 如果接收 溢出信息  if (oflow\_info != NULL) {  // 溢出信息：负数 -1，正数 1  \*oflow\_info = \*str == '-' ? -1 : 1;  }  // 长度溢出，值为 -1  dp\_or\_e = -1;  // 处理小数  goto process\_double;  }  // 如果开头是 . 并且下一个字符是数字  } else if (\*ptr == '.' && ZEND\_IS\_DIGIT(ptr[1])) {  // 处理小数  process\_double:  // 类型为小数  type = IS\_DOUBLE;  // 如果有接收用的小数变量，转换小数，否则 如果需要完全匹配的话，检查数字  /\* If there's a dval, do the conversion; else continue checking  \* the digits if we need to check for a full match \*/  // 如果有接收用的小数变量  if (dval) {  // 转成小数  local\_dval = zend\_strtod(str, &ptr);  // 如果没有接收用的小数变量，并且不允许出错 并且长度没有溢出  } else if (!allow\_errors && dp\_or\_e != -1) {  // 如果当前字符是. ,值为1，否则（就是碰到E跳过来的）值为2  dp\_or\_e = (\*ptr++ == '.') ? 1 : 2;  // 检查小数  goto check\_digits;  }  // 开头字符不是数字或 . 直接返回 否  } else {  return 0;  }  // 如果，还没到结尾，简单检查一下后面的字符  if (ptr != str + length) {  // 尾数据，指针指向当前字符  const char \*endptr = ptr;  // 跳过连续的空白  while (\*endptr == ' ' || \*endptr == '\t' || \*endptr == '\n' || \*endptr == '\r' || \*endptr == '\v' || \*endptr == '\f') {  // 指针右移  endptr++;  // 有效长度-1  length--;  }  // 如果指针位置不在字串末尾（说明空格后面还有东西）  if (ptr != str + length) {  // 如果不允许错误  if (!allow\_errors) {  // 返回0 （转换失败）  return 0;  }  // 如果允许错误，并且有接收变量  if (trailing\_data != NULL) {  // 返回：结尾有无效数据  \*trailing\_data = true;  }  }  }  // 如果是整数  if (type == IS\_LONG) {  // # define MAX\_LENGTH\_OF\_LONG 20 ,64位最大长度20  // 如果已经到了最大长度，减掉 \0占一个字符 。  if (digits == MAX\_LENGTH\_OF\_LONG - 1) {  // # define LONG\_MIN\_DIGITS \"9223372036854775808\" ,64位  // 比较大小，检查是否溢出  int cmp = strcmp(&ptr[-digits], long\_min\_digits);  // ！（负数溢出 或 （无溢出 并且带 负号））  if (!(cmp < 0 || (cmp == 0 && \*str == '-'))) {  // 如果接收小数  if (dval) {  // 字串转成小数  \*dval = zend\_strtod(str, NULL);  }  // 如果没有溢出  if (oflow\_info != NULL) {  // 溢出值为：负数 -1，正数 1  \*oflow\_info = \*str == '-' ? -1 : 1;  }  // 返回：小数类型  return IS\_DOUBLE;  }  }  // 如果有接收变量  if (lval) {  // 如果是负数  if (neg) {  // 转成负数  tmp\_lval = -tmp\_lval;  }  // 赋值返回  \*lval = (zend\_long) tmp\_lval;  }  // 返回：整数类型  return IS\_LONG;  // 如果不是整数（是小数）  } else {  // 如果有接收变量  if (dval) {  // 赋值返回  \*dval = local\_dval;  }  // 返回：小数类型  return IS\_DOUBLE;  }  } |

### zend\_compare()函数

zend\_compare()函数用于比较两个变量是否相等，定义如下：

|  |
| --- |
| ZEND\_API int ZEND\_FASTCALL zend\_compare(zval \*op1, zval \*op2) {  int converted = 0;  zval op1\_copy, op2\_copy;  while (1) {  // 按两个变量的类型，适配比较方法  switch (TYPE\_PAIR(Z\_TYPE\_P(op1), Z\_TYPE\_P(op2))) {  // 两个整型：取回zval 里的 long 型，直接比较  case TYPE\_PAIR(IS\_LONG, IS\_LONG):  return Z\_LVAL\_P(op1)>Z\_LVAL\_P(op2)?1:(Z\_LVAL\_P(op1)<Z\_LVAL\_P(op2)?-1:0);  // 一个double 一个 long, 两个都转成 double 进行比较  case TYPE\_PAIR(IS\_DOUBLE, IS\_LONG):  // 转成比较结果 0 1 -1  return ZEND\_NORMALIZE\_BOOL(Z\_DVAL\_P(op1) - (double)Z\_LVAL\_P(op2));  // 同上  case TYPE\_PAIR(IS\_LONG, IS\_DOUBLE):  // 转成比较结果 0 1 -1  return ZEND\_NORMALIZE\_BOOL((double)Z\_LVAL\_P(op1) - Z\_DVAL\_P(op2));    // 两个 double  case TYPE\_PAIR(IS\_DOUBLE, IS\_DOUBLE):  if (Z\_DVAL\_P(op1) == Z\_DVAL\_P(op2)) {  return 0;  } else {  // 结果转成 0，1，-1  // ((n) ? (((n)<0) ? -1 : 1) : 0)  return ZEND\_NORMALIZE\_BOOL(Z\_DVAL\_P(op1) - Z\_DVAL\_P(op2));  }    // 两个 array  case TYPE\_PAIR(IS\_ARRAY, IS\_ARRAY):  // 用 hash\_zval\_compare\_function 函数比较两个哈希表  return zend\_compare\_arrays(op1, op2);  // null 和 false 相等。（测试过，但在javascript 里 null和false 不相等）  case TYPE\_PAIR(IS\_NULL, IS\_NULL):  case TYPE\_PAIR(IS\_NULL, IS\_FALSE):  case TYPE\_PAIR(IS\_FALSE, IS\_NULL):  case TYPE\_PAIR(IS\_FALSE, IS\_FALSE):  case TYPE\_PAIR(IS\_TRUE, IS\_TRUE):  return 0;  // true > null  case TYPE\_PAIR(IS\_NULL, IS\_TRUE):  return -1;  // 同上  case TYPE\_PAIR(IS\_TRUE, IS\_NULL):  return 1;    // 两个 string  case TYPE\_PAIR(IS\_STRING, IS\_STRING):  // 指针目标相同  if (Z\_STR\_P(op1) == Z\_STR\_P(op2)) {  return 0;  }  // 先按数字比较，不能按数字再按字串比较  return zendi\_smart\_strcmp(Z\_STR\_P(op1), Z\_STR\_P(op2));  // string 和 null 比较。空string 和null 相等  case TYPE\_PAIR(IS\_NULL, IS\_STRING):  return Z\_STRLEN\_P(op2) == 0 ? 0 : -1;  // 同上  case TYPE\_PAIR(IS\_STRING, IS\_NULL):  return Z\_STRLEN\_P(op1) == 0 ? 0 : 1;  // string 和 整型比较  case TYPE\_PAIR(IS\_LONG, IS\_STRING):  return compare\_long\_to\_string(Z\_LVAL\_P(op1), Z\_STR\_P(op2));  // 同上  case TYPE\_PAIR(IS\_STRING, IS\_LONG):  return -compare\_long\_to\_string(Z\_LVAL\_P(op2), Z\_STR\_P(op1));  // string 和 双精度比较  case TYPE\_PAIR(IS\_DOUBLE, IS\_STRING):  if (zend\_isnan(Z\_DVAL\_P(op1))) {  return 1;  }  return compare\_double\_to\_string(Z\_DVAL\_P(op1), Z\_STR\_P(op2));  // 同上  case TYPE\_PAIR(IS\_STRING, IS\_DOUBLE):  if (zend\_isnan(Z\_DVAL\_P(op2))) {  return 1;  }    return -compare\_double\_to\_string(Z\_DVAL\_P(op2), Z\_STR\_P(op1));  // object > null  case TYPE\_PAIR(IS\_OBJECT, IS\_NULL):  return 1;  // 同上 null < object  case TYPE\_PAIR(IS\_NULL, IS\_OBJECT):  return -1;    // 其它情况  default:  // 如果有一个是引用，追踪引用对象，然后重来  if (Z\_ISREF\_P(op1)) {  op1 = Z\_REFVAL\_P(op1);  continue;  } else if (Z\_ISREF\_P(op2)) {  op2 = Z\_REFVAL\_P(op2);  continue;  }    // 如果两个都是对象切指针相同，返回相同  if (Z\_TYPE\_P(op1) == IS\_OBJECT  && Z\_TYPE\_P(op2) == IS\_OBJECT  && Z\_OBJ\_P(op1) == Z\_OBJ\_P(op2)) {  return 0;    // 如果op1或op2是对象，调用它的 compare方法  } else if (Z\_TYPE\_P(op1) == IS\_OBJECT) {  return Z\_OBJ\_HANDLER\_P(op1, compare)(op1, op2);  } else if (Z\_TYPE\_P(op2) == IS\_OBJECT) {  return Z\_OBJ\_HANDLER\_P(op2, compare)(op1, op2);  }  // 如果没有转换过  if (!converted) {  // p1 或 p2 有一个是 true false null  // p1是 false 或 null  if (Z\_TYPE\_P(op1) < IS\_TRUE) {  // p2是ture，算p2大，其他情况算相等  return zval\_is\_true(op2) ? -1 : 0;  // p1 是 true  } else if (Z\_TYPE\_P(op1) == IS\_TRUE) {  // p2 是 true 算相等，其他算 p1 大  return zval\_is\_true(op2) ? 0 : 1;  // p2 是 false 或 null,  } else if (Z\_TYPE\_P(op2) < IS\_TRUE) {  // p1是ture，算p1大，其他情况算相等  return zval\_is\_true(op1) ? 1 : 0;  // p2 是 true  } else if (Z\_TYPE\_P(op2) == IS\_TRUE) {  // p1 是 true 算相等，其他算 p2 大  return zval\_is\_true(op1) ? 0 : -1;  // 其他情况  } else {  // 标量转换成数字，结果放在 holder 里，全程不报错，不转换数组，对象转换失败时返回1。  op1 = \_zendi\_convert\_scalar\_to\_number\_silent(op1, &op1\_copy);  // 标量转换成数字，结果放在 holder 里，全程不报错，不转换数组，对象转换失败时返回1。  op2 = \_zendi\_convert\_scalar\_to\_number\_silent(op2, &op2\_copy);  // 如果有异常  if (EG(exception)) {  // 停止比较array（这个异常只能是数组比较造成的？）  return 1; /\* to stop comparison of arrays \*/  }  // 标记成转换过  converted = 1;  // 从头来  }  } else if (Z\_TYPE\_P(op1)==IS\_ARRAY) { // 数组大于其他类型  return 1;  } else if (Z\_TYPE\_P(op2)==IS\_ARRAY) { // 数组大于其他类型  return -1;  // 其他情况，报错：不支持的操作对象  } else {  ZEND\_UNREACHABLE(); // 不应进入这个逻辑分支  zend\_throw\_error(NULL, "Unsupported operand types");  return 1;  }  }  }  } |

### zend\_compile\_var()函数

zend\_compile\_var()函数用于访问变量，代码如下：

|  |
| --- |
| static zend\_op \*zend\_compile\_var(znode \*result, zend\_ast \*ast, uint32\_t type, bool by\_ref) {  // 短路监测点，返回短路操作码堆栈深度  uint32\_t checkpoint = zend\_short\_circuiting\_checkpoint();  zend\_op \*opcode = zend\_compile\_var\_inner(result, ast, type, by\_ref); // 编译变量  // 提交带短路的操作集合  zend\_short\_circuiting\_commit(checkpoint, result, ast);  // 返回操作码  return opcode;  } |

### increment\_string()函数

increment\_string()函数用于对字串进行自增运算，代码如下：

|  |
| --- |
| / perl风格的字串自增  // Perl style string increment  // '0'++ => '1', 'a'++ => 'b'  // 'z'++ => 'aa', 'z9'++ => 'aa0'  // '{9'++ => '{0'  // ing4, 大小写和数字字串的自增。perl风格。  // 这个东西没什么问题，但还不够直观，应该很少用到吧  static void ZEND\_FASTCALL increment\_string(zval \*str) {  int carry=0; // 位置为 字串长度-1  size\_t pos=Z\_STRLEN\_P(str)-1;  char \*s;  zend\_string \*t;  // 关闭编译 警告  int last=0; /\* Shut up the compiler warning \*/  int ch;  if (Z\_STRLEN\_P(str) == 0) { // 如果长度为0  zval\_ptr\_dtor\_str(str); // 销毁字串  ZVAL\_CHAR(str, '1'); // 值更新为 '1'  return; // 返回  }  if (!Z\_REFCOUNTED\_P(str)) { // 如果 zval 不可计数  // 取出里面的string，创建 非永久 zend\_string 副本, 并关联到str  Z\_STR\_P(str) = zend\_string\_init(Z\_STRVAL\_P(str), Z\_STRLEN\_P(str), 0);  // 类型为 string  Z\_TYPE\_INFO\_P(str) = IS\_STRING\_EX;  // 如果 zval 引用计数 > 1  } else if (Z\_REFCOUNT\_P(str) > 1) {  // 只有在分配成功后释放string  /\* Only release string after allocation succeeded. \*/  // 取出 zend\_string  zend\_string \*orig\_str = Z\_STR\_P(str);  // 创建新的字串 副本  Z\_STR\_P(str) = zend\_string\_init(Z\_STRVAL\_P(str), Z\_STRLEN\_P(str), 0);  // 原字串减少引用次数  GC\_DELREF(orig\_str);  // 如果可计数，计数是1（或0）  } else {  // 删除 zend\_string 的哈希值  zend\_string\_forget\_hash\_val(Z\_STR\_P(str));  }  // 取出 字串（可能是副本或原本）  s = Z\_STRVAL\_P(str);  // 遍历  do {  // 倒序遍历  ch = s[pos];  // a-z  if (ch >= 'a' && ch <= 'z') {  // 如果字符是z（直接自增会变成 { ）  if (ch == 'z') {  // 转成 a  s[pos] = 'a';  // 需要进位  carry=1;  // 其他，直接自增（c语言里char型可以 按ascii码序自增，测试过）  } else {  // ascii 码自增  s[pos]++;  // 不需要进位  carry=0;  }  // 最后一个是小写字符  last=LOWER\_CASE;  // A-Z  } else if (ch >= 'A' && ch <= 'Z') {  // 如果字符是Z（直接自增会变成 [ ）  if (ch == 'Z') {  // 转成 Z  s[pos] = 'A';  // 需要进位  carry=1;  // 其他，直接自增（c语言里char型可以 按ascii码序自增，测试过）  } else {  // ascii 码自增  s[pos]++;  // 不需要进位  carry=0;  }  // 最后一个是大写字符  last=UPPER\_CASE;  // 0-9  } else if (ch >= '0' && ch <= '9') {  // 如果字符是Z（直接自增会变成 [ ）  if (ch == '9') {  // 转成 0  s[pos] = '0';  // 需要进位  carry=1;  // 其他，直接自增（c语言里char型可以 按ascii码序自增，测试过）  } else {  // ascii 码自增  s[pos]++;  // 不需要进位  carry=0;  }  // 最后一个是数字  last = NUMERIC;  // 其他字串  } else {  // 无越界  carry=0;  // 自增运算已完成  break;  }  // 如果没有 字符z/Z/9 自增带来的 进位。  if (carry == 0) {  // 自增运算已完成  break;  }  // 一直到头  } while (pos-- > 0);    // 如果有 字符z/Z/9 自增带来的 进位  if (carry) {  // 分配新串，多要1位  t = zend\_string\_alloc(Z\_STRLEN\_P(str)+1, 0);  // 新串最前面空1位，复制全部  memcpy(ZSTR\_VAL(t) + 1, Z\_STRVAL\_P(str), Z\_STRLEN\_P(str));  // 最后加个\0  ZSTR\_VAL(t)[Z\_STRLEN\_P(str) + 1] = '\0';  // 按操作类型  switch (last) {  // 数字，最前面加1  case NUMERIC:  ZSTR\_VAL(t)[0] = '1';  break;  // 大写字母，最前面加A  case UPPER\_CASE:  ZSTR\_VAL(t)[0] = 'A';  break;  // 小写字母，最前面加A  case LOWER\_CASE:  ZSTR\_VAL(t)[0] = 'a';  break;  }  // 释放 str 中的字串  zend\_string\_free(Z\_STR\_P(str));  // 把新字串关联到str  ZVAL\_NEW\_STR(str, t);  }  } |

## 二）宏程序

### ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION()宏程序

ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION()宏程序用于调用两个操作对象的do\_operation()方法，代码如下：

|  |
| --- |
| // 尝试调用op1 和 op2的 do\_operation 方法，如果成功直接返回  #define ZEND\_TRY\_BINARY\_OBJECT\_OPERATION(opcode) \  ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION(opcode) \  else \  ZEND\_TRY\_BINARY\_OP2\_OBJECT\_OPERATION(opcode)  // 尝试调用op1的 do\_operation 方法，如果成功直接返回  #define ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION(opcode) \  /\* 如果op1类型是对象 并且 访问在的方法存在 \*/ \  if (UNEXPECTED(Z\_TYPE\_P(op1) == IS\_OBJECT) \  && UNEXPECTED(Z\_OBJ\_HANDLER\_P(op1, do\_operation))) { \  /\* 调用do\_operation方法，如果成功 \*/\  if (EXPECTED(SUCCESS == Z\_OBJ\_HANDLER\_P(op1, do\_operation)(opcode, result, op1, op2))) { \  return SUCCESS; /\* 返回成功 \*/\  } \  } |

ZEND\_TRY\_BINARY\_OP2\_OBJECT\_OPERATION()宏程序用于调用第二个操作对象的do\_operation()方法，的业务逻辑与ZEND\_TRY\_BINARY\_OP1\_OBJECT\_OPERATION()宏程序相似。

# 未解疑问

创建运行时缓存：ZEND\_MAP\_PTR\_INIT(),RUN\_TIME\_CACHE()如何编目比较好？

InterlockedOr8()底层函数是做什么用的？

ZEND\_FAST\_CONCAT\_SPEC\_CONST\_CONST\_HANDLER()如何才能调用到。

# Change Log

2024.11.23 创建

2024.11.24 前言，执行过程

2024.11.26 分词和语法解析

2024.11.27 hello world

2024.11.28 语句类型

2024.11.29 加法运算

2024.11.29 加法运算,类型转换

2024.12.3 ZVAL\_COPY\_VALUE()，转战基本类型

2024.12.10 编译赋值运算 遇到pass\_two()

2024.12.11 编译赋值运算，pass\_two()，调整目录结构

2024.12.12 pass\_two()完成一半

2024.12.13 赋值，执行，暂时完成，下午转战类型篇

2025.06.26 半年过去了，难以想象！完成了类型篇、哈希表、以及面向对向的一篇，明天继续。

2025.07.18 操作码处理器，有头绪了，把生成器梳理完果然更通透了，停了半年研究其他部分，终于串连上了，不容易。

2025.07.19 减法，乘法，除法，取余数，势如破竹，不过还要更仔细才行。还有编译和执行流程这么复杂，如何更简单地让人理解呢。虽然内容都在里面了，但不够连贯，主要还是流程本身复杂。明天开始幂运算。

2025.07.22 正常进行，二元操作符，每个其实都蛮麻烦的。很多复用的宏程序，可以抽出来放到附录里，要不然不好找，其他篇中也一样。

2025.07.23 $a+=1;这种是赋值操作，和普通的加法不一样。布尔运算over，大多数问题都迎刃而解了。这阶段比较轻松，主要归功于对zend\_vm\_gen.php的学习，再看虚拟机简单多了。

2025.07.24 自增自减操作没那么简单，发现理解有问题。前面还有一些未完善，要修改。内容多了工作也复杂起来了。今天搞完了一元运算符。