# 一、基本概念

PHP是弱类型语言，声名变量时不需要指定类型，但不指定类型并不等于没有类型。PHP语言中定义的普通数据类型有12种，本篇重点介绍这些类型的底层实现原理，并对其中一些类型展开详细介绍。

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

本篇中的大部分内容来自PHP源码中的/Zend/zend\_types.h文件。

先从最核心的zend\_value数据结构开始。

## 一）zend\_value数据结构

zend\_value是php内核中最核心的数据结构之一，从这里开始认识PHP对数据类型的划分。

zend\_value是一个联合类型（union），而不是结构体（struct），数据结构的定义如下：

|  |
| --- |
| typedef int64\_t zend\_long; // 长整型，原生64位整数  typedef union \_zend\_value { // 占8Bytes，64位内存  zend\_long lval; // 长整型，原生64位整数  double dval; // 双精度（小数）  zend\_refcounted \*counted; // 计数器指针  zend\_string \*str; // 字符串型指针  zend\_array \*arr; // 数组型指针  zend\_object \*obj; // 对象型指针  zend\_resource \*res; // 资源型指针  zend\_reference \*ref; // 引用型指针  zend\_ast\_ref \*ast; // 语句引用型指针  zval \*zv; // 内置变量指针  void \*ptr; // 通用指针  zend\_class\_entry \*ce; // 类指针  zend\_function \*func; // 函数指针  struct { // 兼容32位系统指针  uint32\_t w1;  uint32\_t w2;  } ww;  } zend\_value; |

如上所示，先定义了一个联合类型\_zend\_value，并给它定义了一个别名zend\_value，在程序中通常使用zend\_value而不是\_zend\_value。这是php源码中常见的类型定义方式。

联合类型（union）中的第一个元素是zend\_long类型，zend\_long是原生64位整数（int64\_t）类型的别名，所以这个联合类型的大小是8Bytes，64位。在64位系统中，zend\_value联合类型中所有元素的类型都是64位，这里不会有多余的内存分配。

zend\_value联合类型中最后一个元素ww是为了兼容32位指针而设计，在64位系统中不会用到。

zend\_value的定义中，联合了php中最常见的数据类型，下面以此为线索展开介绍。

zend\_value虽然是最常用的数据类型，但在php源码中几乎见不到它（仅在\Zend\zend\_execute.c文件中有一次引用），这是因为zend\_value类型被包含在zval类型中，一般同zval类型一同创建。zval类型在php源码中随处可见，是使用频率最高的底层类型。

## 二）zval结构体

zval类型是php语言中“变量（variable）”的底层实现，也可以把zval类型称为“变量”，它的定义如下：

|  |
| --- |
| typedef struct \_zval\_struct zval; // zval是\_zval\_struct结构体的别名  struct \_zval\_struct { // 占用8+4+4=16Bytes，128位  zend\_value value; // 占用8Bytes，64位  union { // 联合类型，占用4Bytes，32位  uint32\_t type\_info; // 类型信息，32位整数  struct { // 用struct是打包成和 uint32\_t 相同的大小  ZEND\_ENDIAN\_LOHI\_3( // 要兼容 整数的 高位或低位在前，共4bytes  zend\_uchar type, // 无符号字符，1Byte  zend\_uchar type\_flags, // 无符号字符，1Byte  union { // 这个union是为了兼容旧版本  uint16\_t extra; // 无符号16位整数，2Byte  } u)  } v;  } u1;  union { // 全是32位整数，不同的元素名表示不同的用途  uint32\_t next; // 哈希表中，用于指向相同哈希值的下一个元素  uint32\_t cache\_slot; // opcache中少量用到  uint32\_t opline\_num; // 操作码序号  uint32\_t lineno; // php脚本行号，在解析和编译时用到  uint32\_t num\_args; // 函数调用时的参数数量  uint32\_t fe\_pos; // foreach遍历时的的位置  uint32\_t fe\_iter\_idx; // foreach迭代器序号  uint32\_t property\_guard; // 对象属性保护标记  uint32\_t constant\_flags; // 常量标记  uint32\_t extra; // 额外数据，少量调用  } u2;  }; |

如上所示，zval类型占用16Bytes，128位，包含3个元素。它第一个元素value是zend\_value类型，而不是指针。一般情况下，zend\_value类型实例都是同zval类型实例一同创建和销毁，不会独立存在。

zval类型的第二个元素u1是联合类型（union），用于存放类型信息，它的第一个元素type\_info是一个32位整数。第二个元素v是一个结构体，里面又包含3个元素：前两个元素type和type\_flags各占1个Byte，第三个元素u占2个Bytes，一共占4个Bytes，总大小与type\_info相同。

由于type\_info是一个32位整数，在不同运行环境中存在高位在前、低为在前两种可能。为了与它兼容，u元素中的子元素用ZEND\_ENDIAN\_LOHI\_3()宏程序排序。ZEND\_ENDIAN\_LOHI\_3()宏程序定义如下：

|  |
| --- |
| #ifdef WORDS\_BIGENDIAN  # define ZEND\_ENDIAN\_LOHI\_3(lo, mi, hi) hi; mi; lo;  #else  # define ZEND\_ENDIAN\_LOHI\_3(lo, mi, hi) lo; mi; hi; // 64位windows系统用这个分支  #endif |

zval类型的第三个元素u2也是联合类型，它的所有元素都是32位整数，用于存放扩展信息，不同的元素名称表示它的不同用途。

### 变量（zval）的类型信息

zval结构体可以用来存放各种类型的数据结构，用它来实现的php“变量”可以不指定类型，随时转换成任意类型，例如：

|  |
| --- |
| <?php  $var = true; // IS\_TRUE类型  $var = false; // IS\_FALSE类型  $var = null; // IS\_NULL类型  $var = 5; // IS\_LONG类型  $var = 1.5; // IS\_DOUBLE类型  $var = "str"; // IS\_STRING类型  $var = [1,2,3]; // IS\_ARRAY类型  $var = new stdclass(); // IS\_OBJECT类型  $var = fopen("a.txt",'r'); // IS\_RESOURCE类型  $var = &$var; // IS\_REFERENCE类型  static $a=new stdclass(); // IS\_CONSTANT\_AST类型 |

在以上示例中的php代码中，变量可以随意转换成各种类型，非常方便。像php这样，可以随意转换变量类型的开发语言，被称为弱类型语言，与之相对的是像c语言，java语言等，必须为每个变量指定类型的强类型语言。

PHP不允许直接修改zval.u1.v.type元素，想要更新类型，必须更新zval.u1.type\_info。由于zval.u1.v和zval.u1.type\_info属于同一个联合类型（union），更新zval.u1.type\_info会同时更新zval.u1.v中的3个元素。从zval\_get\_type()函数和Z\_TYPE()宏程序的定义中可以看出这个意图，对zval.u1.v.type元素只提供读取，未提供更新功能。但zval.u1.v.type\_flags和zval.u1.type\_info元素都是可以直接更新的。

在zend\_types.h中，数据类型被分成几个分组，以下是普通数据类型（Regular data types）的常量定义：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **常量名** | **值** | **说明** | **类型名** | **用于更新的宏程序** |
| IS\_UNDEF | 0 | 未定义 |  | ZVAL\_UNDEF() |
| IS\_NULL | 1 | NULL |  | ZVAL\_NULL()  ZVAL\_BOOL() |
| IS\_FALSE | 2 | 假 |  | ZVAL\_FALSE()  ZVAL\_BOOL() |
| IS\_TRUE | 3 | 真 |  | ZVAL\_TRUE() |
| IS\_LONG | 4 | 整数 | zend\_long | ZVAL\_LONG() |
| IS\_DOUBLE | 5 | 小数 | double | ZVAL\_DOUBLE() |
| IS\_STRING | 6 | 字符串 | zend\_string |  |
| IS\_ARRAY | 7 | 数组 | zend\_array/HashTable |  |
| IS\_OBJECT | 8 | 对象 | zend\_object |  |
| IS\_RESOURCE | 9 | 资源 | zend\_resource |  |
| IS\_REFERENCE | 10 | 引用 | zend\_reference | ZVAL\_NEW\_REF() |
| IS\_CONSTANT\_AST | 11 | 常量表达式 | zend\_ast\_ref | ZVAL\_AST() |

PHP提供了一些方法和宏程序来访问zval的类型信息，详情参见“通用宏程序”章节。

## 三）简单类型

如上表所未，普通数据类型有12个。前6种类型只需要一个zval实例就可以完成数据对象的存储，可以称为**简单类型**。

### 数据类型IS\_UNDEF、IS\_NULL、IS\_FALSE、IS\_TRUE

IS\_UNDEF类型表示此zval实例没有被使用。一般使用ZVAL\_UNDEF()宏程序把zval实例更新成IS\_UNDEF类型，例如：

|  |
| --- |
| #define ZVAL\_UNDEF(z) Z\_TYPE\_INFO\_P(z) = IS\_UNDEF; |

如上所示，把zval实例更新成IS\_UNDEF时，只更新zval.u1.type\_info元素，不更新zval.value和zval.u2这两个元素。ZVAL\_NULL()、ZVAL\_FALSE()、ZVAL\_TRUE()、ZVAL\_BOOL()这几个宏程序的定义与ZVAL\_UNDEF()相似，都是只更新zval.u1.type\_info元素。

由于普通类型的值为0-11，更新zval.u1.type\_info元素时，会自动清空zval.u1.v.type\_flags元素和zval.u1.v.u元素。

在基础数据类型的定义中，并没有布尔（bool）类型，真（TRUE）和假（FALSE）被定义成两个不同的类型。在程序设计中，通常中0表示“假”，1表示“真”，但在zval的类型信息中，0和1被用来定义IS\_UNDEF和IS\_NULL类型，用2和3表示IS\_FALSE和IS\_TRUE类型。IS\_FALSE和IS\_TRUE类型非常常用，这样定义使用比较方便。

与这几个类型相关的宏程序逻辑都比较简单，有如下几个：

|  |  |  |
| --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** |
| ZVAL\_UNDEF() | 把zval实例更新成IS\_UNDEF类型  Z\_TYPE\_INFO\_P(z) = IS\_UNDEF; | 大量 |
| Z\_ISUNDEF() | 检验zval实例是否是IS\_UNDEF类型  (Z\_TYPE(zval) == IS\_UNDEF) | 大量 |
| Z\_ISUNDEF\_P() | 通过指针检验zval实例是否是IS\_UNDEF类型  Z\_ISUNDEF(\*(zval\_p)) | 大量 |
| ZVAL\_NULL() | 把zval实例更新成IS\_NULL类型  Z\_TYPE\_INFO\_P(z) = IS\_NULL; | 大量 |
| Z\_ISNULL() | 检验zval实例是否是IS\_NULLL类型  (Z\_TYPE(zval) == IS\_NULL) | 大量 |
| Z\_ISNULL\_P() | 通过指针检验zval实例是否是IS\_NULL类型  Z\_ISNULL(\*(zval\_p)) | 少量 |
| ZVAL\_FALSE() | 把zval实例更新成IS\_FALSE类型  Z\_TYPE\_INFO\_P(z) = IS\_FALSE; | 大量 |
| ZVAL\_TRUE() | 把zval实例更新成IS\_TRUE类型  Z\_TYPE\_INFO\_P(z) = IS\_TRUE; | 大量 |
| ZVAL\_BOOL() | 检验zval实例是否是 IS\_TRUE或IS\_FALSE类型  Z\_TYPE\_INFO\_P(z) = (b) ? IS\_TRUE : IS\_FALSE; | 大量 |

### 数据类型IS\_LONG、IS\_DOUBLE

以上4个类型，都不需要用到zval.value，除了这4个类型以外，其他类型都需要用到zval.value，由于zval.value需要兼容多种类型，php内核中为每个类型提供了对应的宏程序来修改它。

与这两个类型相关的宏程序逻辑都比较简单，有如下几个：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| ZVAL\_LONG() | 把zval实例更新成整数 | 大量 |  |
| Z\_LVAL() | 访问zval实例的整数元素  (zval).value.lval | 大量 | 逻辑简单 |
| Z\_LVAL\_P() | 通过指针访问zval实例的整数元素  Z\_LVAL(\*(zval\_p)) | 大量 | 逻辑简单 |
| ZVAL\_DOUBLE() | 把zval实例更新成小数 | 大量 |  |
| Z\_DVAL() | 访问zval实例的小数元素  (zval).value.dval | 大量 | 逻辑简单 |
| Z\_DVAL\_P() | 通过指针访问zval实例的小数元素  Z\_DVAL(\*(zval\_p)) | 大量 | 逻辑简单 |

使用ZVAL\_LONG()宏程序来把变量（zval实例）赋值成IS\_LONG类型，ZVAL\_LONG()宏程序定义如下：

|  |
| --- |
| #define ZVAL\_LONG(z, l) do { \  zval \*\_\_z = (z); \  Z\_LVAL\_P(\_\_z) = l; \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_LONG; \  } while (0) |

如上所示，把数值存入zval.value元素中，并更新zval.u1.type\_info类型信息。

小数（IS\_DOUBLE）类型的使用相似，使用ZVAL\_DOUBLE()宏程序来更新zval实例，ZVAL\_DOUBLE()宏程序定义如下：

|  |
| --- |
| #define ZVAL\_DOUBLE(z, d) do { \  zval \*\_\_z = (z); \  Z\_DVAL\_P(\_\_z) = d; \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_DOUBLE; \  } while (0) |

PHP内核中处理整数的业务逻辑比较简单，主要使用系统原生的处理方法，相关业务逻辑定义在zend\_long.h文件中。处理小数的业务逻辑非常复杂，主要使用第三方的开源代码进行处理，相关业务逻辑在zend\_strtod.c文件中。

从IS\_STRING开始的后面6种类型，都有自己专属的数据结构，zval实例被当成一个指向数据对象的指针来使用，这6种类型称为**复杂类型**。

## 四）扩展类型（extended types）

复杂类型在zval外有独立的数据对象，为了方便内存回收，需要给它们增加引用数量记录。有的复杂类型可以计数，有的可以自动回收，添加了计数和回收标记的复杂类型，被称为扩展类型（extended types），扩展类型常量定义如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| Z\_TYPE\_FLAGS\_SHIFT | 8 | 类型标记左移位数 |
| IS\_TYPE\_REFCOUNTED | 1（左移8位=256） | 可计数标记 |
| IS\_TYPE\_COLLECTABLE | 2（左移8位=512） | 可回收标记 |
| IS\_INTERNED\_STRING\_EX | IS\_STRING | 内置字符串，不可计数和回收 |
| IS\_STRING\_EX | IS\_STRING|256 | 可计数字符串 |
| IS\_ARRAY\_EX | IS\_ARRAY|256|512 | 可计数、可回收的数组 |
| IS\_OBJECT\_EX | IS\_OBJECT|256|512 | 可计数、可回收的对象 |
| IS\_RESOURCE\_EX | IS\_RESOURCE|256 | 可计数的资源 |
| IS\_REFERENCE\_EX | IS\_REFERENCE|256 | 可计数的引用类型 |
| IS\_CONSTANT\_AST\_EX | IS\_CONSTANT\_AST|256 | 可计数的常量表达式 |

如上所示，内置字符串（IS\_INTERNED\_STRING\_EX）不可以计录引用次数，其他扩展类型都可以记录引用次数。数组（IS\_ARRAY\_EX）和对象（IS\_OBJECT\_EX）可以自动回收。

扩展类型记录在zval实例的u1.type\_info元素中。

## 五）内部类型（internal types）

内部类型是几种特殊的类型，它们只在PHP程序内部使用，不提供PHP代码层面的访问。内部类型有如下几种：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| IS\_INDIRECT | 12 | 间接引用 |
| IS\_PTR | 13 | 通用指针 |
| IS\_ALIAS\_PTR | 14 | 别名指针 |
| \_IS\_ERROR | 15 | 错误 |

相关宏程序比较简单，列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** |
| Z\_INDIRECT() | 访问zval实例的间接引用指针元素  (zval).value.zv | 少量 |
| Z\_INDIRECT\_P() | 通过指针访问zval实例的间接引用指针元素  Z\_INDIRECT(\*(zval\_p)) | 大量 |
| ZVAL\_DEINDIRECT() | 检验zval实例是否是间接引用类型  if (Z\_TYPE\_P(z) == IS\_INDIRECT){ (z) = Z\_INDIRECT\_P(z); } | 少量 |
| ZVAL\_INDIRECT() | 把zval实例更新成间接引用  Z\_INDIRECT\_P(z) = (v); Z\_TYPE\_INFO\_P(z) = IS\_INDIRECT; | 大量 |
| Z\_PTR() | 访问zval实例的通用指针元素  (zval).value.ptr | 大量 |
| Z\_PTR\_P() | 通过指针访问zval实例的通用指针元素  Z\_PTR(\*(zval\_p)) | 大量 |
| ZVAL\_PTR() | 把zval实例更新成通用指针  Z\_PTR\_P(z) = (p); Z\_TYPE\_INFO\_P(z) = IS\_PTR; | 大量 |
| ZVAL\_ALIAS\_PTR() | 把zval实例更新成别名指针  Z\_PTR\_P(z) = (p); Z\_TYPE\_INFO\_P(z) = IS\_ALIAS\_PTR; | 少量 |
| ZVAL\_ERROR() | 把zval实例类型更新成错误  Z\_TYPE\_INFO\_P(z) = \_IS\_ERROR; | 少量 |
| Z\_ISERROR() | 检验zval实例是否是\_IS\_ERROR类型  (Z\_TYPE(zval) == \_IS\_ERROR) | 少量 |
| Z\_ISERROR\_P() | 通过指针检验zval实例是否是\_IS\_ERROR类型  Z\_ISERROR(\*(zval\_p)) | 少量 |
| ZVAL\_FUNC() | 把zval实例更新成函数指针  Z\_FUNC\_P(z) = (f);Z\_TYPE\_INFO\_P(z) = IS\_PTR; | 无 |
| Z\_FUNC() | 访问zval实例的函数指针元素  (zval).value.func | 少量 |
| Z\_FUNC\_P(zval\_p) | 通过指针访问zval实例的函数指针元素  Z\_FUNC(\*(zval\_p)) | 大量 |

通过zval结构体的定义可以看出，zval.value.zv元素的类型是zval指针，zval.value.ptr元素的类型是无类型（void）指针。在这几种内置类型中，间接引用（IS\_INDIRECT）、通用指针（IS\_PTR）、别名指针（IS\_ALIAS\_PTR）这三种类型的用法与复杂类型相同，错误（\_IS\_ERROR）类型的用法与简单类型相同。

## 六）伪类型（Fake types）

伪类型并不是在底层代码中定义的类型，而是为了使用方便而定义的几个虚假的类型。它们没有对应的数据结构，只是在PHP脚本中，用于约束函数或类方法的返回值类型：

|  |  |  |  |
| --- | --- | --- | --- |
| **常量名** | **值** | **说明** | **对应php语句中的**  **类型关键字** |
| IS\_CALLABLE | 12 | 闭包 | callable |
| IS\_ITERABLE | 13 | 可迭代类型 | iterable |
| IS\_VOID | 14 | 不可以有返回值 | void |
| IS\_STATIC | 15 | 本实例 | static |
| IS\_MIXED | 16 | 任意类型 | mixed |
| IS\_NEVER | 17 | 不可以返回 | never |

示例如下：

|  |
| --- |
| // callable，限定函数的返回值类型为闭包  function a12():callable {return function(){};} a12();  // iterable，限定函数的返回值类型为可迭代类型  function a13():iterable {return [];} a13();  // void，限定函数中的return语句后面不可以有值  function a14():void {return;} a14();  // static，限定类方法的返回类型必须是此类的实例  class c1{ function a15():static {return $this;} } (new c1())->a15();  // mixed，限定返回值可以是任何类型（必须要有return语句，并且return语句后面一定要有值）  function a16():mixed {return null;} a16();  // 返回类型为never的函数不可以显示（有return语句）或隐式（没有return语句）地返回  function a17():never {die;} a17(); |

# 二、引用计数与垃圾回收

“引用（reference）”是指一个zval实例指向另一个复杂类型的实例。当一个zval实例指向了一个复杂类型的实例，这个zval实例就引用了这个复杂类型的实例。复杂类型的引用次数是指，一个复杂类型的实例被多少个zval实例指向。

引用计数在垃圾回收机制中非常重要，当一个复杂类型实例被多个变量（zval实例）引用时，删除一个引用它的变量，只需要减少这个复杂类型实例的引用次数。当最后一个引用它的变量被删除时，这个复杂实例就不会再被用到，这时就可以自动回收这个实例，释放它占用的内存了。

在6个复杂类型的结构开头，都会有一个zend\_refcounted\_h类型的gc元素，它是引用计数器，结构定义如下：

|  |
| --- |
| // 垃圾回收计数器，这个结构体主要用法是包装在其他 struct 里  typedef struct \_zend\_refcounted\_h { // 2个32位整数，共8Bytes，64位  uint32\_t refcount; // 引用次数，32位整数，占4Bytes  union { // 用union是为了方便扩展成其他用途，像zval的u1、u2一样。占4Bytes  uint32\_t type\_info; // 类型信息，32位整数  } u;  } zend\_refcounted\_h; |

引用计数器主要用途是用于垃圾回收。zend\_refcounted\_h结构体（struct）主要用法是包装在其他结构体里，不独立使用。所以这是一个头信息，后缀\_h也表明了它的用途。

## zend\_refcounted结构体

在运算过程中，常常需要通过zval实例（或指针）快速访问复杂实类型实例的计数器，这时会用到zend\_refcounted结构体，它的定义如下：

|  |
| --- |
| typedef struct \_zend\_refcounted zend\_refcounted;  struct \_zend\_refcounted {  zend\_refcounted\_h gc;  }; |

在zend\_value结构体中也包含了这个结构体的指针，当一个zval实例指向一个复杂类型实例时，可以通过访问zval.value.counted元素，忽略具体类型直接获取复杂类型实例的引用计数器指针。这种访问方式非常常用，通过Z\_COUNTED()宏程序和Z\_COUNTED\_P()宏程序来实现：

|  |
| --- |
| // 通过zval实例快速访问复杂类型实例的计数器指针  #define Z\_COUNTED(zval) (zval).value.counted  // 通过zval实例指针快速访问复杂类型实例的计数器指针  #define Z\_COUNTED\_P(zval\_p) Z\_COUNTED(\*(zval\_p)) |

## 操作type\_info元素

zend\_refcounted\_h结构体由两个32位整数组成，一共占8Bytes，64位。由于第一个元素的一些用法需要依赖第二个元素，所以先介绍第二个元素。

zend\_refcounted\_h结构体的第二个元素u.type\_info，它的用法与zval的u1元素类似，把一个32位整数分成四段使用，从左到右依次为：1）最左22位，通常用来存放地址索引号，在相关函数和宏程序名中通常写成gc\_info。2）左边第23-28位，共6位，用来存放标记信息，在相关函数和宏程序名中通常写成gc\_flags。3）最右边4位，用来存放类型，在相关函数和宏程序名中通常写成gc\_type。

由于type\_info元素本身设计得比较复杂，与之相关的常量也比较多，为了便于理解，可以把相关的常量和友程序分成如下几组：

### 1）操作gc\_type

gc\_type用于存放实例类型，常用的类型有如下几个，有的类型添加了回收标记：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| GC\_TYPE\_MASK | 0x0000000f | 用于过滤gc\_type  （最右边4位） |
| GC\_NULL | IS\_NULL | GC\_NOT\_COLLECTABLE | 不可回收的null |
| GC\_STRING | IS\_STRING | GC\_NOT\_COLLECTABLE | 不可回收的STRING |
| GC\_ARRAY | IS\_ARRAY | 不可回收的数组 |
| GC\_OBJECT | IS\_OBJECT | 不可回收的对象 |
| GC\_RESOURCE | IS\_RESOURCE | GC\_NOT\_COLLECTABLE | 不可回收的资源 |
| GC\_REFERENCE | IS\_REFERENCE | GC\_NOT\_COLLECTABLE | 不可回收的引用 |
| GC\_CONSTANT\_AST | IS\_CONSTANT\_AST | GC\_NOT\_COLLECTABLE | 不可回收的常量表达式 |

以上常量值中GC\_NOT\_COLLECTABLE常量表示不可回收。与前文介绍的扩展类型不同（扩展类型记录在zval实例的u1.type\_info元素中），以上类型常量记录在复杂类型实例的gc.u.type\_info元素里。

操作gc\_type相关的函数和宏程序逻辑都比较简单，列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** |
| zval\_gc\_type() | 取得回收类型，gc\_type\_info 最右4位  return (gc\_type\_info & GC\_TYPE\_MASK); | 少量 |
| GC\_TYPE() | 取得实例的回收标记（类型信息最右4位）  zval\_gc\_type(GC\_TYPE\_INFO(p)) | 大量 |
| Z\_GC\_TYPE() | 获取zval实例指向对象的回收类型（gc\_type）  GC\_TYPE(Z\_COUNTED(zval)) | 少量 |
| Z\_GC\_TYPE\_P() | 通过指针获取zval实例指向对象的回收类型（gc\_type）  Z\_GC\_TYPE(\*(zval\_p)) | 少量 |
| Z\_GC\_TYPE\_INFO() | 访问zval实例指向对象的整个回收类型信息  // 展开： (zval).value.counted->gc.u.type\_info  GC\_TYPE\_INFO(Z\_COUNTED(zval)) | 少量 |

### 2）操作gc\_flags

与gc\_flags相关的常量有如下几个：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| GC\_FLAGS\_MASK | 0x000003f0 | 用于过滤gc\_flags（右边第5-10位，共6位） |
| GC\_FLAGS\_SHIFT | 0 | gc\_flags的可选值的位移数，0位，不需要位移 |
| GC\_NOT\_COLLECTABLE | 16 (1<<4) | 不可回收标记 |
| GC\_PROTECTED | 32 (1<<5) | 递归保护标记 |
| GC\_IMMUTABLE | 64 (1<<6) | 表示不可修改 |
| GC\_PERSISTENT | 128 (1<<7) | malloc()函数分配内存 |
| GC\_PERSISTENT\_LOCAL | 256 (1<<8) | 这个标记调试时用到 |

操作gc\_flags相关的函数和宏程序逻辑都比较简单，列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** |
| zval\_gc\_flags() | 取得回收标记，gc\_type\_info 右侧 5-10位（共6位）  return (gc\_type\_info >> GC\_FLAGS\_SHIFT) &  (GC\_FLAGS\_MASK >> GC\_FLAGS\_SHIFT); | 少量 |
| GC\_ADD\_FLAGS() | 给 .gc.u.type\_info 添加标记 | 大量 |
| GC\_DEL\_FLAGS() | 删除回收标记 | 大量 |
| GC\_FLAGS() | 取得实例的回收标记  zval\_gc\_flags(GC\_TYPE\_INFO(p)) | 大量 |
| Z\_GC\_FLAGS() | 获取zval实例指向对象的回收标记  GC\_FLAGS(Z\_COUNTED(zval)) | 少量 |
| GC\_IS\_RECURSIVE() | 检验实例是否有【递归保护】标记  (GC\_FLAGS(p) & GC\_PROTECTED) | 大量 |
| GC\_PROTECT\_RECURSION() | 给实例添加【递归保护】标记  GC\_ADD\_FLAGS(p, GC\_PROTECTED) | 大量 |
| GC\_UNPROTECT\_RECURSION() | 给实例删除【递归保护】标记  GC\_DEL\_FLAGS(p, GC\_PROTECTED) | 大量 |
| GC\_TRY\_PROTECT\_RECURSION() | 给可修改的实例添加【递归保护】标记  if (!(GC\_FLAGS(p) & GC\_IMMUTABLE))  GC\_PROTECT\_RECURSION(p); | 大量 |
| GC\_TRY\_UNPROTECT\_RECURSION() | 给可修改的实例删除【递归保护】标记  if (!(GC\_FLAGS(p) & GC\_IMMUTABLE))  GC\_UNPROTECT\_RECURSION(p); | 大量 |
| Z\_IS\_RECURSIVE() | 检验zval实例的目标对象是否有【递归保护】标记  GC\_IS\_RECURSIVE(Z\_COUNTED(zval)) | 少量 |
| Z\_IS\_RECURSIVE\_P() | 通过指针检验zval实例的目标对象是否有【递归保护】标记  Z\_IS\_RECURSIVE(\*(zv)) | 大量 |
| Z\_PROTECT\_RECURSION() | 给zval实例的目标对象添加【递归保护】标记  GC\_PROTECT\_RECURSION(Z\_COUNTED(zval)) | 少量 |
| Z\_PROTECT\_RECURSION\_P() | 通过指针给zval实例的目标对象添加【递归保护】标记  Z\_PROTECT\_RECURSION(\*(zv)) | 大量 |
| Z\_UNPROTECT\_RECURSION() | 给zval实例的目标对象删除【递归保护】标记  GC\_UNPROTECT\_RECURSION(Z\_COUNTED(zval)) | 少量 |
| Z\_UNPROTECT\_RECURSION\_P() | 通过指针给zval实例的目标对象删除【递归保护】标记  Z\_UNPROTECT\_RECURSION(\*(zv)) | 大量 |
| Z\_TYPE\_INFO\_REFCOUNTED() | 通过 type\_info 确定是否可记录引用数，优化版（有任何一个gc\_flags）  (((t) & Z\_TYPE\_FLAGS\_MASK) != 0) | 少量 |
| Z\_OPT\_REFCOUNTED() | 检验zval实例是否有可记录引用数标记  Z\_TYPE\_INFO\_REFCOUNTED(Z\_TYPE\_INFO(zval)) | 少量 |
| Z\_OPT\_REFCOUNTED\_P() | 通过指针检验zval实例是否有可记录引用数标记  Z\_OPT\_REFCOUNTED(\*(zval\_p)) | 大量 |

### 4）操作gc\_info

与gc\_info操作相关的常量有如下几个：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| GC\_INFO\_MASK | 0xfffffc00 | 用于过滤最左边22位，gc\_info |
| GC\_FLAGS\_SHIFT | 0 | gc\_flags的可选值的位移数，0位，不需要位移 |
| GC\_INFO\_SHIFT | 10 | 操作gc\_info时的位移数（跳过gc\_type的4位和gc\_flags的6位） |

操作gc\_info相关的函数和宏程序逻辑都比较简单，列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** |
| zval\_gc\_info() | 取得回收信息，gc\_type\_info 最左侧22位  return (gc\_type\_info >> GC\_INFO\_SHIFT); | 少量 |
| GC\_INFO() | 取得实例的回收信息（类型信息最左侧22位）  zval\_gc\_info(GC\_TYPE\_INFO(p)) | 少量 |
| Z\_GC\_INFO() | 获取zval实例指向对象的回收信息  GC\_INFO(Z\_COUNTED(zval)) | 少量 |

### 5）操作整个type\_info

操作整个type\_info元素的函数和宏程序逻辑都比较简单，列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** |
| GC\_TYPE\_INFO() | 访问实例的回收类型  (p)->gc.u.type\_info | 少量 |
| Z\_GC\_TYPE\_INFO() | 访问zval实例指向对象的整个回收类型信息  // 展开： (zval).value.counted->gc.u.type\_info  GC\_TYPE\_INFO(Z\_COUNTED(zval)) | 少量 |

如上所示，zend\_types.h中定义的zend\_refcounted\_h数据结构和相关算法都比较简单，它们都是为内存垃圾回收（garbage collect）而设计。本篇重点在介绍常用类型的数据结构和相关算法，内存垃圾回收的设计思路可简单概括为“自动回收引用次数为0的数组和对象”，更多内容参见“垃圾回收篇”。

u.type\_info元素的最左2位，在垃圾回收过程中用来存放颜色值（详情请参见“垃圾回收篇”）。

## 操作引用计数元素refcount

zend\_refcounted\_h结构体的第一个元素refcount用来存放引用次数。相关函数和宏程序可以分为两组。

### 1）直接操作复杂类型实例

由于6种复杂类型的开头都有一个计数器，这些宏程序可以兼容所有复杂类型实例，逻辑都比较简单，列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** |
| zend\_gc\_addref() | 通过指针给实例增加1次引用计数  return ++(p->refcount); | 少量 |
| zend\_gc\_addref\_ex() | 通过指针给实例增加引用计数  p->refcount += rc; return p->refcount; | 少量 |
| zend\_gc\_delref() | 通过指针给实例减少1次引用计数  return --(p->refcount); | 少量 |
| zend\_gc\_delref\_ex() | 通过指针给实例减少引用计数  p->refcount -= rc; return p->refcount; | 少量 |
| zend\_gc\_refcount() | 通过指针读取实例的引用计数  return p->refcount; | 少量 |
| zend\_gc\_set\_refcount() | 通过指针更新实例的引用计数  p->refcount = rc; return p->refcount; | 少量 |
| zend\_gc\_try\_addref() | 增加可变实例的引用数  if (!(p->u.type\_info & GC\_IMMUTABLE)) {  ++p->refcount;} | 少量 |
| zend\_gc\_try\_delref() | 减少可变实例的引用数  if (!(p->u.type\_info & GC\_IMMUTABLE)) {  --p->refcount;} | 少量 |
| GC\_ADDREF() | 通过指针给实例增加1次引用计数  zend\_gc\_addref(&(p)->gc) | 大量 |
| GC\_ADDREF\_EX() | 通过指针给实例增加引用计数  zend\_gc\_addref\_ex(&(p)->gc, rc) | 少量 |
| GC\_DELREF() | 通过指针给实例减少1次引用计数  zend\_gc\_delref(&(p)->gc) | 大量 |
| GC\_DELREF\_EX() | 通过指针给实例减少引用计数  zend\_gc\_delref\_ex(&(p)->gc, rc) | 少量 |
| GC\_REFCOUNT() | 读取实例的引用计数  zend\_gc\_refcount(&(p)->gc) | 大量 |
| GC\_SET\_REFCOUNT() | 更新实例的引用计数  zend\_gc\_set\_refcount(&(p)->gc, rc) | 大量 |
| GC\_TRY\_ADDREF() | 增加可变实例的引用数  zend\_gc\_try\_addref(&(p)->gc) | 少量 |
| GC\_TRY\_DELREF() | 减少可变实例的引用数  zend\_gc\_try\_delref(&(p)->gc) | 少量 |

### 2）通过zval实例操作复杂类型实例

zval实例或指针来操作目标对象的refcount元素，相关函数和宏程序逻辑都比较简单，列表如下：

|  |  |  |
| --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** |
| Z\_COUNTED() | 访问zval实例指向的对象，(zval).value.counted | 少量 |
| Z\_COUNTED\_P() | 通过指针访问(zval).value.counted | 大量 |
| zval\_addref\_p() | 通过指针给zal实例指向对象增加1次引用计数  return GC\_ADDREF(Z\_COUNTED\_P(pz)); | 少量 |
| zval\_delref\_p() | 通过指针给zal实例指向对象减少1次引用计数  return GC\_DELREF(Z\_COUNTED\_P(pz)); | 少量 |
| zval\_refcount\_p() | 通过指针读取zval实例指向对象的引用次数  return GC\_REFCOUNT(Z\_COUNTED\_P(pz)); | 少量 |
| zval\_set\_refcount\_p() | 通过指针更新zval实例指向对象的引用计数  return GC\_SET\_REFCOUNT(Z\_COUNTED\_P(pz), rc); | 少量 |
| Z\_ADDREF() | 给zal实例指向对象增加1次引用计数  Z\_ADDREF\_P(&(z)) | 大量 |
| Z\_ADDREF\_P() | 通过指针给zal实例指向对象增加1次引用计数  zval\_addref\_p(pz) | 大量 |
| Z\_DELREF() | 给zal实例指向对象减少1次引用计数  Z\_DELREF\_P(&(z)) | 少量 |
| Z\_DELREF\_P() | 通过指针给zal实例指向对象减少1次引用计数  zval\_delref\_p(pz) | 大量 |
| Z\_REFCOUNTED() | 检验一个zval实例是否是可计数类型  ((Z\_TYPE\_FLAGS(zval) &  IS\_TYPE\_REFCOUNTED) != 0) | 大量 |
| Z\_REFCOUNTED\_P() | 通过指针检验一个对象是否是可计数类型  Z\_REFCOUNTED(\*(zval\_p)) | 大量 |
| Z\_TRY\_ADDREF() | 给zal实例指向对象增加1次引用计数，跳过不可计数类型  Z\_TRY\_ADDREF\_P(&(z)) | 大量 |
| Z\_TRY\_ADDREF\_P() | 通过指针给zal实例指向对象增加1次引用计数，跳过不可计数类型  if (Z\_REFCOUNTED\_P((pz))) { Z\_ADDREF\_P((pz)); } | 大量 |
| Z\_TRY\_DELREF() | 给zal实例指向对象减少1次引用计数，跳过不可计数类型  Z\_TRY\_DELREF\_P(&(z)) | 少量 |
| Z\_TRY\_DELREF\_P() | 通过指针给zal实例指向对象减少1次引用计数，跳过不可计数类型  if (Z\_REFCOUNTED\_P((pz))) { Z\_DELREF\_P((pz)); } | 大量 |
| Z\_REFCOUNT() | 读取zval实例指向对象的引用次数  Z\_REFCOUNT\_P(&(z)) | 大量 |
| Z\_REFCOUNT\_P() | 通过指针读取zval实例指向对象的引用次数  zval\_refcount\_p(pz) | 大量 |
| Z\_SET\_REFCOUNT() | 更新zval实例指向对象的引用计数  Z\_SET\_REFCOUNT\_P(&(z), rc) | 少量 |
| Z\_SET\_REFCOUNT\_P() | 通过指针更新zval实例指向对象的引用计数  zval\_set\_refcount\_p(pz, rc) | 少量 |

# 三、字符串

在任何通用的开发语言里，字符串都是非常重要的底层数据类型，PHP语言也不例外。在PHP语言中，字符串类型的结构体为zend\_string，定义如下：

|  |
| --- |
| typedef struct \_zend\_string zend\_string; // \_zend\_string 结构体的别名  struct \_zend\_string {  zend\_refcounted\_h gc; // 引用计数器 ，8Bytes  zend\_ulong h; // 哈希值。为了用作数组key，给每个字符串加个哈希值，8Bytes  size\_t len; // 长度，8Bytes  char val[1]; // 字符串 1Byte  }; |

在6个复杂类型的结构开头，都会有一个zend\_refcounted\_h类型的gc元素，它是引用计数器，具体信息参见“引用计数器”章节。

zend\_string结构体的最后一个元素是char数组，而不是char指针。这样定义是为了把字符串内容一起打包到zend\_string实例中，和zend\_string实例占用同一块内存，一起分配内存，一起释放内存，管理方便，又可以减少内存碎片。

需要注意的是zend\_string结构体本身带了一个长度为1的字符串数组，在分配内存计算长度时，需要把这个字符减出来，例如，创建存放5个字符的zend\_string实例，实际长度应该是：sizeof(zend\_string)+5-1。

在结构体的最后放一个可伸缩的元素，是PHP内核中常见的设计方式，后续还会遇到。

zend\_string.h中为zend\_string类型定义了一些快捷访问宏程序：

|  |
| --- |
| #define ZSTR\_VAL(zstr) (zstr)->val // 大量使用  #define ZSTR\_LEN(zstr) (zstr)->len // 大量使用  #define ZSTR\_H(zstr) (zstr)->h // 大量使用 |

字符串（zend\_string）相关的大部分方法代码都非常简单。

## 一）基本字符串操作

### 1、创建字符串

创建字符串相关的函数和宏程序列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| \_ZSTR\_STRUCT\_SIZE() | 计算需要创建的zend\_string实例大小 | 大量 |  |
| zend\_string\_alloc() | 调用用 pemalloc()函数创建 zend\_string实例 | 大量 |  |
| zend\_string\_init() | 调用zend\_string\_alloc()函数创建字符串，并复制给定内容 | 大量 | 逻辑简单 |
| zend\_string\_safe\_alloc() | 调用用 safe\_pemalloc()函数创建 zend\_string实例 | 大量 | 逻辑简单 |
| ZSTR\_ALLOCA\_ALLOC() | 调用用 do\_alloca()宏程序创建 zend\_string实例 | 大量 |  |
| zend\_string\_safe\_realloc() | 安全地重新分配内存 | 少量 |  |
| ZSTR\_INIT\_LITERAL() | 调用zend\_string\_init()函数 | 少量 | 逻辑简单 |
| zend\_string\_init\_fast() | 快速创建并初始化zend\_string()实例 | 少量 |  |

如上所示，

ZSTR\_ALLOCA\_ALLOC()宏程序用于分配内存，创建字符串（zend\_string）实例：

|  |
| --- |
| #define ZSTR\_ALLOCA\_ALLOC(str, \_len, use\_heap) do { \  /\* 分配内存创建新字符串，长度对齐到8 \*/ \  (str) = (zend\_string \*)do\_alloca(ZEND\_MM\_ALIGNED\_SIZE\_EX(\_ZSTR\_STRUCT\_SIZE(\_len), 8), (use\_heap)); \  GC\_SET\_REFCOUNT(str, 1); /\* 引用次数1 \*/ \  GC\_TYPE\_INFO(str) = GC\_STRING; /\* 类型为可回收字符串 \*/ \  ZSTR\_H(str) = 0; /\* 哈希值为0 \*/ \  ZSTR\_LEN(str) = \_len; /\* 长度 \*/ \  } while (0) |

如上所示，字符串的长度对齐到8。这里设置的类型不是IS\_STRING，而是GC\_STRING，它带有“不可回收”标记。

\_ZSTR\_STRUCT\_SIZE()宏程序 用于计算整个zend\_string实例的大小：

|  |
| --- |
| // 长度+1是因为字符串的结尾要有一个 \0 字符  #define \_ZSTR\_STRUCT\_SIZE(len) (\_ZSTR\_HEADER\_SIZE + len + 1)  // 计算zend\_string.val元素相对于zend\_string开头的偏移量，val元素前有3个元素24Bytes  #define \_ZSTR\_HEADER\_SIZE XtOffsetOf(zend\_string, val) |

在\_ZSTR\_STRUCT\_SIZE()宏程序的定义中，计算长度的方式是“头信息长度+字符串长度+1” ，而不是sizeof(zend\_string)+len，这是因为，sizeof(zend\_string)值为32（而不是25）。

do\_alloca()宏程序用于分配内存：

|  |
| --- |
| // 分配内存，根据尺寸自动适配分配方式，尺寸阀值为32K  # define do\_alloca(size, use\_heap) do\_alloca\_ex(size, ZEND\_ALLOCA\_MAX\_SIZE, use\_heap)  // 大于limit使用emalloc()函数分配，否则使用alloca()函数分配，内存指针从use\_heap返回  # define do\_alloca\_ex(size, limit, use\_heap) \  ((use\_heap = (UNEXPECTED((size) > (limit)))) ? emalloc(size) : alloca(size))  # define ZEND\_ALLOCA\_MAX\_SIZE (32 \* 1024) // 使用alloca()函数分配的最大内存为32K |

ZSTR\_ALLOCA\_INIT()宏程序用于为指定的字符串创建zend\_string副本：

|  |
| --- |
| #define ZSTR\_ALLOCA\_INIT(str, s, len, use\_heap) do { \  ZSTR\_ALLOCA\_ALLOC(str, len, use\_heap); /\* 创建指定长度（自动对齐）的字符串 \*/ \  memcpy(ZSTR\_VAL(str), (s), (len)); /\* 把字符串s的内容复制到zend\_string里 \*/ \  ZSTR\_VAL(str)[(len)] = '\0'; /\* 添加结尾字符 \*/ \  } while (0) |

zend\_string\_alloc()函数用于创建指定长度的字符串实例，与ZSTR\_ALLOCA\_ALLOC()宏程序不同的是，它不调用do\_alloca()宏程序分配内存，而是使用pemalloc()方法分配内存：

|  |
| --- |
| static zend\_string \*zend\_string\_alloc(size\_t len, bool persistent) {  zend\_string \*ret = (zend\_string \*)pemalloc(  ZEND\_MM\_ALIGNED\_SIZE(\_ZSTR\_STRUCT\_SIZE(len)), persistent);  GC\_SET\_REFCOUNT(ret, 1); // 更新引用次数  // 添加 type\_info， GC\_STRING + 持久标记  GC\_TYPE\_INFO(ret) = GC\_STRING | ((persistent ? IS\_STR\_PERSISTENT : 0) << GC\_FLAGS\_SHIFT);  ZSTR\_H(ret) = 0; // 初始哈希值为0  ZSTR\_LEN(ret) = len; // 长度  return ret; // 返回实例指针  } |

pemalloc()函数和ZEND\_MM\_ALIGNED\_SIZE()宏程序的介绍参见“内存管理篇”。

zend\_string\_init\_fast()函数用于快速创建并初始化字符串：

|  |
| --- |
| sstatic zend\_string \*zend\_string\_init\_fast(const char \*str, size\_t len) {  if (len > 1) { // 长度一个以上调用 zend\_string\_init()函数  return zend\_string\_init(str, len, 0);  } else if (len == 0) {  return zend\_empty\_string; // 返回NULL  } else /\* if (len == 1) \*/ {  return ZSTR\_CHAR((zend\_uchar) \*str); // 返回预先创建好的快捷字符串  }  } |

### 2、更新字符串

更新字符串相关的函数和宏程序列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| zend\_string\_concat2() | 拼接2个字符串（char\*） | 大量 |  |
| zend\_string\_concat3() | 拼接3个字符串（char\*） | 大量 | 逻辑简单 |
| zend\_string\_copy() | 给字符串增加引用次数 | 大量 |  |
| zend\_string\_dup() | 给字符串（zend\_string）创建副本 | 大量 |  |
| zend\_string\_realloc() | 调整zend\_string实例的尺寸（变大或变小） | 大量 |  |
| zend\_string\_safe\_realloc() | 安全地调整zend\_string实例的尺寸 | 少量 | 逻辑类似 |
| zend\_string\_extend() | zend\_string实例增加尺寸  逻辑类似zend\_string\_realloc() | 大量 | 逻辑类似 |
| zend\_string\_truncate() | 截短zend\_string实例  逻辑类似zend\_string\_realloc() | 大量 | 逻辑类似 |
| zend\_string\_addref() | 增加引用次数，保留字返回1，其他返回增加后的引用次数 | 大量 | 逻辑简单 |
| zend\_string\_delref() | 减少引用次数，保留字返回1，其他返减少加后的引用次数 | 少量 | 逻辑简单 |
| zend\_string\_separate() | 给字符串创建副本，separate指的是版本分离 | 少量 |  |

zend\_string\_concat2()方法用于拼接两个字符串（char\*）：

|  |
| --- |
| ZEND\_API zend\_string \*zend\_string\_concat2(const char \*str1, size\_t str1\_len,  const char \*str2, size\_t str2\_len) {  size\_t len = str1\_len + str2\_len; // 长度和  zend\_string \*res = zend\_string\_alloc(len, 0); // 创建新字符串  memcpy(ZSTR\_VAL(res), str1, str1\_len); // 复制第一个字符串  memcpy(ZSTR\_VAL(res) + str1\_len, str2, str2\_len); // 复制第二个字符串  ZSTR\_VAL(res)[len] = '\0'; // 添加结束字符  return res;  } |

zend\_string\_copy()方法比较特别，从字面意思上看，是复制字符串实例，但实际上它只是给原有实例增加了引用次数。zend\_string\_dup()方法才是复制实例操作，它会给实例创建副本：

|  |
| --- |
| static zend\_string \*zend\_string\_copy(zend\_string \*s) {  if (!ZSTR\_IS\_INTERNED(s)) { // 如果不是保留字  GC\_ADDREF(s); // 增加引用次数+1  }  return s;  }  static zend\_string \*zend\_string\_dup(zend\_string \*s, bool persistent){  if (ZSTR\_IS\_INTERNED(s)) { // 如果是保留字，直接返回  return s;  } else { // 不是保留字，创建并返回副本，无哈希值  return zend\_string\_init(ZSTR\_VAL(s), ZSTR\_LEN(s), persistent);  }  } |

zend\_string\_realloc()函数用于调整zend\_string实例大小：

|  |
| --- |
| static zend\_string \*zend\_string\_realloc(zend\_string \*s, size\_t len, bool persistent) {  zend\_string \*ret;  if (!ZSTR\_IS\_INTERNED(s)) { // 如果不是保留字, 并且引用次数是1  if (EXPECTED(GC\_REFCOUNT(s) == 1)) {  // 直接调整大小  ret = (zend\_string \*)perealloc(  s, ZEND\_MM\_ALIGNED\_SIZE(\_ZSTR\_STRUCT\_SIZE(len)), persistent);  ZSTR\_LEN(ret) = len; // 更新长度  zend\_string\_forget\_hash\_val(ret); // 清空哈希值  return ret; // 返回指针  }  }  // 是保留字或引用次数不是1 ， 创建新版本  ret = zend\_string\_alloc(len, persistent);  memcpy(ZSTR\_VAL(ret), ZSTR\_VAL(s), MIN(len, ZSTR\_LEN(s)) + 1); // 复制内容  if (!ZSTR\_IS\_INTERNED(s)) { // 不是保留字  GC\_DELREF(s); // 旧版本减少引用次数  }  return ret; // 返回副本指针  } |

zend\_string\_separate()用于给字符串创建版本分支，separate指的是版本分离：

|  |
| --- |
| static zend\_string \*zend\_string\_separate(zend\_string \*s, bool persistent) {  if (ZSTR\_IS\_INTERNED(s) || GC\_REFCOUNT(s) > 1) { // 如果是保留字或引用次数大于1  // 如果不是保留字（引用数大于1的普通字符串）  if (!ZSTR\_IS\_INTERNED(s)) {  GC\_DELREF(s); // 原字符串引用次数-1  }  return zend\_string\_init(ZSTR\_VAL(s), ZSTR\_LEN(s), persistent); // 创建并返回副本  }  // 不是保留字并且引用次数是1，只删除原始实例的哈希值  zend\_string\_forget\_hash\_val(s);  return s; // 返回原始串（因为只引用了1次，没必要搞两个版本）  } |

### 3、释放字符串

释放字符串相关的函数和宏程序列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| zend\_string\_efree() | 调用efree()函数释放zend\_string实例 | 大量 | 逻辑简单 |
| zend\_string\_free() | 调用pefree()函数释放zend\_string实例 | 大量 | 逻辑简单 |
| zend\_string\_release() | 调用pefree()函数释放zend\_string实例 | 大量 | 逻辑简单 |
| zend\_string\_release\_ex() | 释放zend\_string实例 | 大量 | 逻辑简单 |
| \_str\_dtor() | 调用pefree()函数释放zend\_string实例  销毁保留字哈希表时删除保留字 | 少量 | 逻辑简单 |
| ZSTR\_ALLOCA\_FREE() | free\_alloca()宏程序释放zend\_string实例 | 少量 | 逻辑简单 |

释放字符串实例相关的方法逻辑都比较简单。

### 4、哈希值相关操作

字符串哈希值相关的函数和宏程序列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| zend\_hash\_func() | 调用zend\_inline\_hash\_func()函数，给字符串计算哈希值 | 大量 | 逻辑简单 |
| zend\_string\_hash\_val() | 如果字符串没有哈希值，给它计算哈希值 | 大量 | 逻辑简单 |
| zend\_inline\_hash\_func() | 为字符串计算哈希值 | 少量 |  |
| zend\_string\_forget\_hash\_val() | zend\_string 实例的哈希值更新成0 | 少量 | 逻辑简单 |
| zend\_string\_hash\_func() | 调用zend\_hash\_func()函数，给字符串计算哈希值 | 少量 | 逻辑简单 |

zend\_string\_hash\_func()函数在zend\_string.h文件中，用来计算字符串的哈希值，这个函数使用了DJBX33A算法（由Daniel J. Bernstein发明的，乘以33并相加的算法），这个算法是已知的最好的哈希算法之一，因为它运算非常快，并且哈希值分散均匀。代码注释如下：

|  |
| --- |
| static zend\_ulong zend\_string\_hash\_func(const char \*str, size\_t len) {  zend\_ulong hash = Z\_UL(5381); // 初始哈希值 5381  // 遍历字符串，长度每次-8，指针每次向右跳8Bytes，直到最后不到8Bytes为止  for (; len >= 8; len -= 8, str += 8) {  hash = hash \* Z\_L(33 \* 33 \* 33 \* 33) + // 哈希值 \* pow(33,4)  str[0] \* Z\_L(33 \* 33 \* 33) + // 第1个Byte \* pow(33,3)  str[1] \* Z\_L(33 \* 33) + // 第2个Byte \* pow(33,2)  str[2] \* Z\_L(33) + // 第3个Byte \*33  str[3]; // 第4个Byte  hash = hash \* Z\_L(33 \* 33 \* 33 \* 33) + // 哈希值 \* pow(33,4)  str[4] \* Z\_L(33 \* 33 \* 33) + // 第5个Byte \* pow(33,3)  str[5] \* Z\_L(33 \* 33) + // 第6个Byte \* pow(33,2)  str[6] \* Z\_L(33) + // 第7个Byte \* 33  str[7]; // 第8个Byte  }  if (len >= 4) { // 如果余下的大于等于4Bytes  hash = hash \* Z\_L(33 \* 33 \* 33 \* 33) + // 哈希值 \* pow(33,4)  str[0] \* Z\_L(33 \* 33 \* 33) + // 第1个Byte \* pow(33,3)  str[1] \* Z\_L(33 \* 33) + // 第2个Byte \* pow(33,2)  str[2] \* Z\_L(33) + // 第3个Byte \*33  str[3]; // 第4个Byte  len -= 4; // 长度-4  str += 4; // 指针向右跳4  }  if (len >= 2) { // 如果余下的大于等于2Bytes  if (len > 2) { // 如果余下的是3Bytes  hash = hash \* Z\_L(33 \* 33 \* 33) + // 哈希值 \* pow(33,3)  str[0] \* Z\_L(33 \* 33) + // 第1个Byte \* pow(33,2)  str[1] \* Z\_L(33) + // 第2个Byte \* 33  str[2]; // 第3个Byte  } else { // 余下的是2Bytes  hash = hash \* Z\_L(33 \* 33) + // 哈希值 \* pow(33,2)  str[0] \* Z\_L(33) + // 第1个Byte \* 33  str[1]; // 第2个Byte  }  } else if (len != 0) { // 如果余下的等于1Byte  hash = hash \* Z\_L(33) + \*str; // 加最后一个Byte  }  return hash | Z\_UL(0x8000000000000000); // 第一个位写成1  } |

### 5、字符串比较操作

字符串比较操作相关函数和宏程序列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| zend\_string\_equals() | 调用zend\_string\_equal\_content()函数比较两个zend\_string是否相等，大小写敏感 | 大量 | 逻辑简单 |
| zend\_string\_equals\_ci() | 调用zend\_binary\_strcasecmp()函数比较两个zend\_string是否相同，大小写不敏感 | 大量 | 逻辑简单 |
| zend\_string\_equals\_cstr() | 调用memcmp()函数比较一个zend\_string和一个char\* 是否相等，大小写敏感 | 大量 | 逻辑简单 |
| zend\_string\_equals\_literal() | 调用zend\_string\_equals\_cstr()函数比较一个zend\_string和一个char\* 是否相等，大小写敏感 | 大量 | 逻辑简单 |
| zend\_string\_equals\_literal\_ci() | 调用zend\_binary\_strcasecmp()函数比较一个zend\_string和一个char[] 是否相等，大小写不敏感 | 大量 |  |
| zend\_string\_equal\_val() | 调用memcmp()函数，比较两个zend\_string 是否相同，大小写敏感 | 少量 | 逻辑简单 |
| zend\_string\_starts\_with() | 调用zend\_string\_starts\_with\_cstr()函数，校验参数1（zend\_string\*）是否是参数2（zend\_string\*）开头 | 少量 | 逻辑简单 |
| zend\_string\_starts\_with\_cstr() | 调用memcmp()函数，校验参数1（zend\_string\*）是否是参数2（char \*）开头 | 少量 | 逻辑简单 |
| zend\_string\_starts\_with\_literal() | 调用zend\_string\_starts\_with\_cstr()函数，校验参数1（zend\_string\*）是否是参数2（char \*）开头 | 少量 | 逻辑简单 |

如上所示，使用原生的memcmp()方法来进行大小写敏感的字符串的比较，逻辑简单。

zend\_string\_equals\_ci()方法用于忽略大小写比较两个字符串是否相等，调用zend\_operators.c中的zend\_binary\_strcasecmp()方法，代码如下：

|  |
| --- |
| ZEND\_API int zend\_binary\_strcasecmp(const char \*s1, size\_t len1, const char \*s2, size\_t len2) {  size\_t len;  int c1, c2;  if (s1 == s2) { // 如果是指向同一个地址，返回0  return 0;  }  len = MIN(len1, len2); // 取得最小长度  while (len--) { // 两个字符串从头开始逐个字符比较，忽略大小写  c1 = zend\_tolower\_ascii(\*(unsigned char \*)s1++);  c2 = zend\_tolower\_ascii(\*(unsigned char \*)s2++);  if (c1 != c2) {  return c1 - c2;  }  }  // 如果前面一段全相等，比较长度  return ZEND\_THREEWAY\_COMPARE(len1, len2);  } |

调用调的宏程序注释如下：

|  |
| --- |
| // 比较，返回1，0，-1  #define ZEND\_THREEWAY\_COMPARE(a, b) ((a) == (b) ? 0 : ((a) < (b) ? -1 : 1))  // 在提前准备好的数组里读取每个字符对应的小写字符  #define zend\_tolower\_ascii(c) (zend\_tolower\_map[(unsigned char)(c)]) |

### 6、与变量（zval）相关的操作

字符串与变量相关的操作宏程序列表如下，大部分宏程序都被大量调用：

|  |  |  |  |
| --- | --- | --- | --- |
| **宏程序名** | **说明** | **引用**  **次数** | **备注** |
| Z\_STR() | 访问(zval).value.str | 大量 | 逻辑简单 |
| Z\_STR\_P() | 通过指针访问 (zval).value.str | 大量 | 逻辑简单 |
| Z\_STRLEN() | 访问 (zval).value.str->len | 大量 | 逻辑简单 |
| Z\_STRLEN\_P() | 通过指针访问 (zval).value.str->len | 大量 | 逻辑简单 |
| Z\_STRVAL() | 访问 (zval).value.str->val | 大量 | 逻辑简单 |
| Z\_STRVAL\_P() | 通过指针访问 (zval).value.str->val | 大量 | 逻辑简单 |
| ZVAL\_INTERNED\_STR() | 把zval更新成保留字符串变量 | 大量 |  |
| ZVAL\_NEW\_STR() | 把zval更新成字符串变量 | 大量 | 逻辑类似 |
| ZVAL\_STR() | 把zval更新成符串变量，自动适配保留字符串 | 大量 |  |
| ZVAL\_STR\_COPY() | 把zval更新成符串变量，并增加计数，自动适配保留字符串 | 大量 | 逻辑类似 |
| SEPARATE\_STRING() | 给zval实例指向的 zend\_string实例创建分支，并把指针指向新分支 | 无 |  |
| Z\_STRHASH() | 访问 (zval).value.str->h | 无 | 逻辑简单 |
| Z\_STRHASH\_P() | 通过指针访问 (zval).value.str->h | 无 | 逻辑简单 |

ZVAL\_INTERNED\_STR()宏程序用于把zval更新成保留字符串变量：

|  |
| --- |
| #define ZVAL\_INTERNED\_STR(z, s) do { \  zval \*\_\_z = (z); \  zend\_string \*\_\_s = (s); \  Z\_STR\_P(\_\_z) = \_\_s; /\* 通过指针更新 (zval).value.str \*/ \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_INTERNED\_STRING\_EX; /\*类型为保留字符串\*/ \  } while (0) |

ZVAL\_STR\_COPY()宏程序用于把zval更新成符串变量，并增加计数，自动适配保留字符串：

|  |
| --- |
| // 接收的参数1是zval实例指针，第一个用于接收，第二个是源数据  #define ZVAL\_STR\_COPY(z, s) do { \  zval \*\_\_z = (z); /\* 用于接收的zval实例指针 \*/ \  zend\_string \*\_\_s = (s); /\* zend\_string实例指针 \*/ \  Z\_STR\_P(\_\_z) = \_\_s; /\* zval实例指向zend\_string实例 \*/ \  if (ZSTR\_IS\_INTERNED(\_\_s)) { /\* 保留字 \*/ \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_INTERNED\_STRING\_EX; /\* 类型为保留字 \*/ \  } else { /\* 非保留字 \*/ \  GC\_ADDREF(\_\_s); /\* 增加引用次数 \*/ \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_STRING\_EX; /\* 可计数字符串 \*/ \  } \  } while (0) |

SEPARATE\_STRING()宏程序用于给zval实例指向的 zend\_string实例创建分支，并把指针指向新分支，它在PHP源码中没有被调用过：

|  |
| --- |
| #define SEPARATE\_STRING(zv) do { \  zval \*\_zv = (zv); \  if (Z\_REFCOUNT\_P(\_zv) > 1) { /\* 引用次数 >1 才有必要创建副本 \*/ \  zend\_string \*\_str = Z\_STR\_P(\_zv); /\* 取出zend\_string实例指针 \*/ \  /\* 给zend\_string实例创建副本，本zval实例指向新创建的副本 \*/ \  ZVAL\_NEW\_STR(\_zv, zend\_string\_init( \  ZSTR\_VAL(\_str), ZSTR\_LEN(\_str), 0)); \  GC\_DELREF(\_str); /\* 原字串减少引用 \*/ \  } \  } while (0) |

## 二）保留字符串（interned\_string）

保留字符串（简称“保留字”）是一种特殊的字符串，它的应用非常广泛，例如：常量名，类名，函数名等，都会被创建成保留字。

zend\_string.h中为保留字的访问定义了3个函数原型：

|  |
| --- |
| // 使用zend\_string实例创建保留字  typedef zend\_string \*(\*zend\_new\_interned\_string\_func\_t)(zend\_string \*str);  // 使用字符串（char \*）创建保留字  typedef zend\_string \*(\*zend\_string\_init\_interned\_func\_t)(const char \*str, size\_t size, bool permanent);  // 查找保留字  typedef zend\_string \*(\*zend\_string\_init\_existing\_interned\_func\_t)(const char \*str, size\_t size, bool permanent); |

如上所示，保留字只有创建和查询接口，**它一旦创建后不可以删除和更改，不进行引用计数也不会被垃圾回收机制自动回收，会一直保留在内存当中直到进程结束时自动回收。**

zend\_string.c中定义了3个操作永久保留字的函数变量，它们是外部访问接口：

|  |
| --- |
| // 前两个函数用于创建永久保留字  ZEND\_API zend\_new\_interned\_string\_func\_t zend\_new\_interned\_string;  ZEND\_API zend\_string\_init\_interned\_func\_t zend\_string\_init\_interned;  // 第三个函数用于查找永久保留字  ZEND\_API zend\_string\_init\_existing\_interned\_func\_t zend\_string\_init\_existing\_interned; |

和3个用于操作临时保留字的函数变量：

|  |
| --- |
| // 前两个函数用于创建临时保留字  static zend\_new\_interned\_string\_func\_t interned\_string\_request\_handler =  zend\_new\_interned\_string\_request;  static zend\_string\_init\_interned\_func\_t interned\_string\_init\_request\_handler =  zend\_string\_init\_interned\_request;  // 第三个函数用于查找临时保留字  static zend\_string\_init\_existing\_interned\_func\_t interned\_string\_init\_existing\_request\_handler = zend\_string\_init\_existing\_interned\_request; |

保留字只能创建和查找，不能修改和删除。在使用保留字功能前，需要先进行初始化操作。

### 1、初始化保留字功能

与初始化相关的函数/宏程序有如下几个：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| zend\_init\_interned\_strings\_ht() | 初始化保留字哈希表 | 少量 | 逻辑简单 |
| \_str\_dtor() | 用于销毁保留字哈希表中的元素 | 少量 | 逻辑简单 |
| zend\_interned\_strings\_activate() | 调用zend\_init\_interned\_strings\_ht()函数，初始化临时保留字哈希表 | 少量 | 逻辑简单 |
| zend\_interned\_strings\_init() | 初始化永久保留字功能 | 少量 |  |

zend\_interned\_strings\_init()方法用于初始化永久保留字功能：

|  |
| --- |
| ZEND\_API void zend\_interned\_strings\_init(void) {  char s[2];  unsigned int i;  zend\_string \*str;  // 临时保留字处理方法  interned\_string\_request\_handler = zend\_new\_interned\_string\_request;  interned\_string\_init\_request\_handler = zend\_string\_init\_interned\_request;  interned\_string\_init\_existing\_request\_handler =  zend\_string\_init\_existing\_interned\_request;    zend\_empty\_string = NULL; // 空字符串  zend\_known\_strings = NULL; // 已知字符串  // 初始化永久哈希表，interned\_strings\_permanent是全局变量  zend\_init\_interned\_strings\_ht(&interned\_strings\_permanent, 1);  // 永久保留字处理方法  zend\_new\_interned\_string = zend\_new\_interned\_string\_permanent;  zend\_string\_init\_interned = zend\_string\_init\_interned\_permanent;  zend\_string\_init\_existing\_interned = zend\_string\_init\_existing\_interned\_permanent;    str = zend\_string\_alloc(sizeof("")-1, 1); // 空字符串 保留字  ZSTR\_VAL(str)[0] = '\000'; // 空字符串  // 创建成永久保留字  zend\_empty\_string = zend\_new\_interned\_string\_permanent(str);  s[1] = 0;  for (i = 0; i < 256; i++) { // 256个单字节字符，创建成永久保留字  s[0] = i;  zend\_one\_char\_string[i] = zend\_new\_interned\_string\_permanent(  zend\_string\_init(s, 1, 1));  }  // 常用字符串，全部创建成永久保留字  zend\_known\_strings = pemalloc(sizeof(zend\_string\*) \* ((sizeof(known\_strings) / sizeof(known\_strings[0]) - 1)), 1);  for (i = 0; i < (sizeof(known\_strings) / sizeof(known\_strings[0])) - 1; i++) {  str = zend\_string\_init(known\_strings[i], strlen(known\_strings[i]), 1);  zend\_known\_strings[i] = zend\_new\_interned\_string\_permanent(str);  }  } |

zend\_init\_interned\_strings\_ht()方法用于初始化保留字哈希表，代码比较简单：

|  |
| --- |
| static void zend\_init\_interned\_strings\_ht(HashTable \*interned\_strings, bool permanent) {  // 初始化哈希表，创建1024个备用元素  zend\_hash\_init(interned\_strings, 1024, NULL, \_str\_dtor, permanent);  if (permanent) { // 如果是永久保留字  zend\_hash\_real\_init\_mixed(interned\_strings); // 初始化成混合哈希表  }  } |

关于哈希表的介绍参见“数组篇”。如代码中所示，初始化哈希表时，会传入一个\_str\_dtor()函数，它是保留字的销毁器，用来销毁哈希表中的元素：

|  |
| --- |
| static void \_str\_dtor(zval \*zv) {  zend\_string \*str = Z\_STR\_P(zv); // 找到元素中的字符串（zend\_string实例）指针  pefree(str, GC\_FLAGS(str) & IS\_STR\_PERSISTENT); // 释放 zend\_string实例  } |

zend\_interned\_strings\_activate()函数用于初始化临时保留字哈希表：

|  |
| --- |
| ZEND\_API void zend\_interned\_strings\_activate(void) {  // 初始化非永久保留字哈希表，哈希表在编译时全局变量compiler\_globals里  zend\_init\_interned\_strings\_ht(&CG(interned\_strings), 0);  } |

### 2、创建保留字

与保留字功能相关的常量有以下几个：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| IS\_STR\_INTERNED | GC\_IMMUTABLE (64) | 保留字标记（gc\_flags的【不可修改】标记） |
| IS\_STR\_PERSISTENT | GC\_PERSISTENT (128) | malloc()函数分配的字符串标记  （gc\_flags的【malloc()函数分配内存】标记） |
| IS\_STR\_PERMANENT | 256 | 永久字符串（永久保留字串用到） |

用于创建保留字的函数有4个：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| zend\_new\_interned\_string\_request() | 创建临时保留字，接收参数类型为zend\_string\* | 大量 |  |
| zend\_string\_init\_interned\_request() | 创建临时保留字，接收参数类型为char\* | 大量 | 逻辑类似 |
| zend\_new\_interned\_string\_permanent() | 创建永久保留字，接收参数类型为zend\_string\* | 大量 | 逻辑类似 |
| zend\_string\_init\_interned\_permanent() | 创建永久保留字，接收参数类型为char\* | 大量 | 逻辑类似 |
| zend\_add\_interned\_string() | 向保留字哈希表中添加元素 | 大量 |  |

前4个函数的逻辑类似，只是用于创建永久保留字的函数不查找CG(interned\_strings)哈希表。以zend\_new\_interned\_string\_request()函数为例，代码注释如下：

|  |
| --- |
| static zend\_string\* zend\_new\_interned\_string\_request(zend\_string \*str) {  zend\_string \*ret;  if (ZSTR\_IS\_INTERNED(str)) { // 如果已经是保留字，直接返回  return str;  }  zend\_string\_hash\_val(str); // 计算哈希值  // 检查永久保留字，在这个操作里，只会读，不会修改永久保留字  ret = zend\_interned\_string\_ht\_lookup(str, &interned\_strings\_permanent);  if (ret) { // 如果找到  zend\_string\_release(str); // 把字符串释放掉  return ret; // 返回哈希表里的字符串  }  // 如果没找到，检查非永久保留字  ret = zend\_interned\_string\_ht\_lookup(str, &CG(interned\_strings));  if (ret) { // 如果找到  zend\_string\_release(str); // 把字符串释放掉  return ret; // 返回哈希表里的字符串  }  if (GC\_REFCOUNT(str) > 1) { // 如果字符串引用次数大于1  zend\_ulong h = ZSTR\_H(str); // 取出哈希值  zend\_string\_delref(str); // 减少引用次数1次  str = zend\_string\_init(ZSTR\_VAL(str), ZSTR\_LEN(str), 0); // 创建副本  ZSTR\_H(str) = h; // 给副本写入哈希值  }  // 把str(或者副本）添加到非永久保留字中  ret = zend\_add\_interned\_string(str, &CG(interned\_strings), 0);  return ret; // 返回字符串指针  } |

zend\_add\_interned\_string()方法用于向哈希表中添加保留字：

|  |
| --- |
| static zend\_string \*zend\_add\_interned\_string(zend\_string \*str, HashTable \*interned\_strings, uint32\_t flags) {  zval val; // 临时变量  GC\_SET\_REFCOUNT(str, 1); // str 设置引用次数为1  GC\_ADD\_FLAGS(str, IS\_STR\_INTERNED | flags); // 添加保留字标记和额外标记  ZVAL\_INTERNED\_STR(&val, str); // 字符串绑定到临时变量zval  zend\_hash\_add\_new(interned\_strings, str, &val); // 在保留字哈希表中添加新元素  return str;  } |

### 3、查找保留字

用于查找保留字的函数有4个：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| zend\_string\_init\_existing\_interned\_request() | 查找临时保留字，当查找失败时会创建普通zend\_string实例 | 少量 |  |
| zend\_string\_init\_existing\_interned\_permanent() | 查找永久保留字，当查找失败时会创建普通zend\_string实例 | 少量 | 逻辑类似 |
| zend\_interned\_string\_ht\_lookup\_ex() | 从保留字哈希表中查找字符串，接收参数类型为zend\_string\* | 少量 |  |
| zend\_interned\_string\_ht\_lookup() | 从保留字哈希表中查找字符串，接收参数类型为char \*和哈希值 | 少量 | 逻辑类似 |
| ZSTR\_IS\_INTERNED() | GC\_FLAGS(s) 是否包含 IS\_STR\_INTERNED 标记 | 大量 |  |

前2个方法逻辑类似，以zend\_string\_init\_existing\_interned\_request()函数为例，它用于查找临时保留字，当查找失败时会创建普通zend\_string实例：

|  |
| --- |
| static zend\_string\* zend\_string\_init\_existing\_interned\_request(const char \*str, size\_t size, bool permanent) {  zend\_ulong h = zend\_inline\_hash\_func(str, size); // 计算作为数组key的哈希值  // 在永久保留字列表中查找  zend\_string \*ret = zend\_interned\_string\_ht\_lookup\_ex(h, str, size,  &interned\_strings\_permanent);  if (ret) { // 找到就返回  return ret;  }  // 在临时保留字列表中查找  ret = zend\_interned\_string\_ht\_lookup\_ex(h, str, size, &CG(interned\_strings));  if (ret) { // 找到就返回  return ret;  }  // 如果都没找到，创建普通 zend\_string  ret = zend\_string\_init(str, size, permanent);  ZSTR\_H(ret) = h; // 写入哈希值  return ret; // 返回  } |

zend\_string\_init\_existing\_interned\_permanent()函数用于创建永久保留字，逻辑与zend\_string\_init\_existing\_interned\_request()函数类似，只是不查找CG(interned\_strings)临时保留字哈希表。

zend\_interned\_string\_ht\_lookup\_ex()函数用于从保留字哈希表中查找字符串：

|  |
| --- |
| static zend\_string \*zend\_interned\_string\_ht\_lookup\_ex(zend\_ulong h, const char \*str, size\_t size, HashTable \*interned\_strings) {  uint32\_t nIndex;  uint32\_t idx;  Bucket \*p;  nIndex = h | interned\_strings->nTableMask; // 把哈希值换算成有效索引号  idx = HT\_HASH(interned\_strings, nIndex); // 把索引号换算成存放位置  while (idx != HT\_INVALID\_IDX) { // 遍历整个bucket串  p = HT\_HASH\_TO\_BUCKET(interned\_strings, idx); // 获得bucket  // 如果哈希值和 key 都匹配无误  if ((p->h == h) && zend\_string\_equals\_cstr(p->key, str, size)) {  return p->key; // 返回 key  }  idx = Z\_NEXT(p->val); // 下一个bucket  }  return NULL; // 匹配失败  } |

这个方法用到了一些哈希表相关的概念，如Bucket类型，HT\_HASH\_TO\_BUCKET()宏程序等，详细信息参见“数组篇”。

zend\_interned\_string\_ht\_lookup()方法与zend\_interned\_string\_ht\_lookup\_ex()方法逻辑相同，只是接收的参数不同。

ZSTR\_IS\_INTERNED()宏程序用于检查一个字符串（zend\_string实例）是否是保留字，它在PHP内核中被大量使用：

|  |
| --- |
| #define ZSTR\_IS\_INTERNED(s) (GC\_FLAGS(s) & IS\_STR\_INTERNED) |

### 4、快捷访问宏程序

除了上述函数，PHP内核中还定义了几个用于快速访问保留字的宏程序：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| ZSTR\_CHAR() | 获取 256个单字节字符 的 zend\_string 实例 | 大量 | 逻辑简单 |
| ZSTR\_EMPTY\_ALLOC() | 值为空的zend\_string实例（永久保留字） | 大量 | 逻辑简单 |
| ZSTR\_KNOWN() | 获取常用已知字符串（zend\_string 实例） | 大量 |  |
| \_ZEND\_STR\_DSC() | 用于创建known\_strings[]字符串数组 | 临时 | 逻辑简单 |
| \_ZEND\_STR\_ID() | 用于创建\_zend\_known\_string\_id枚举类型 | 临时 | 逻辑简单 |
| STR\_EMPTY\_ALLOC() | 调用ZSTR\_EMPTY\_ALLOC()宏程序 | 少量 | 逻辑简单 |
| ZEND\_KNOWN\_STRINGS() | 定义常用字符串列表。为known\_strings[]字符串数组和\_zend\_known\_string\_id枚举类型预定义的字符串列表 | 2次 |  |

ZEND\_KNOWN\_STRINGS()宏程序里定义了一些常用的字符串常量：

|  |
| --- |
| #define ZEND\_KNOWN\_STRINGS(\_) \  \_(ZEND\_STR\_FILE, "file") \  \_(ZEND\_STR\_LINE, "line") \  ... |

这里比较特殊的是known\_strings字符串列表和\_zend\_known\_string\_id枚举类型的定义方式：

|  |
| --- |
| static const char \*known\_strings[] = {  #define \_ZEND\_STR\_DSC(id, str) str, // 定义临时宏程序，用完马上删掉  ZEND\_KNOWN\_STRINGS(\_ZEND\_STR\_DSC) // 把列表中的第二列元素添加到 enum 中  #undef \_ZEND\_STR\_DSC // 删除临时宏程序  NULL // 最后一个元素，NULL  };  typedef enum \_zend\_known\_string\_id {  #define \_ZEND\_STR\_ID(id, str) id, // 定义临时宏程序，用完马上删掉  ZEND\_KNOWN\_STRINGS(\_ZEND\_STR\_ID) // 把列表中的第一列元素添加到 enum 中  #undef \_ZEND\_STR\_ID // 删除临时宏程序  ZEND\_STR\_LAST\_KNOWN // 最后一个元素，未知字符串  } zend\_known\_string\_id; |

ZSTR\_KNOWN()用于把\_zend\_known\_string\_id枚举中的元素转换成zend\_known\_strings数组中对应的元素（zend\_string实例）：

|  |
| --- |
| #define ZSTR\_KNOWN(idx) zend\_known\_strings[idx] |

zend\_known\_strings数组中的元素是在前文所述zend\_interned\_strings\_init()方法中添加的。

### 5、保留字与PHP执行过程

在整个PHP进行的运行中，有几个与保留字相关的关键操作，以cli模式为例，运行过程如下：

|  |
| --- |
| main() : /sapi/cli/php\_cli.c #从main函数开始执行  ->php\_cli\_startup() : /sapi/cli/php\_cli.c # 调用sapi\_module->startup(sapi\_module)  ->php\_module\_startup() : /main/main.c  ->zend\_startup() : /Zend/zend.c  **->zend\_interned\_strings\_init(); # 1、初始化保留字，初始化永久保留字哈希表**  **->zend\_interned\_strings\_switch\_storage(1); # 2、切换到使用临时保留字**  ->do\_cli() : /sapi/cli/php\_cli.c # 这一步里执行PHP脚本  ->php\_request\_startup() : /main/main.c  **->zend\_interned\_strings\_activate(); # 3、初始化临时字符串哈希表CG(interned\_strings)**  ... # 执行PHP脚本  ->php\_request\_shutdown()  **->zend\_interned\_strings\_deactivate(); # 4、销毁哈希表CG(interned\_strings)**  ->php\_module\_shutdown() # 关闭各个模块  **->zend\_interned\_strings\_switch\_storage(0); # 5、切换到使用永久保留字**  **->zend\_interned\_strings\_dtor(); # 6、销毁永久保留字**  ... |

如上所示，在整个PHP进程运行的过程中，有5个与保留字符串紧密相关的操作，调用顺序如下：

1、调用zend\_interned\_strings\_init()函数初始化永久保留字哈希表，在这之后才可以使用保留字，默认使用永久保留字。

2、调用zend\_interned\_strings\_switch\_storage()函数，切换到使用临时保留字。

3、调用zend\_interned\_strings\_activate()函数，初始化临时保留字哈希表，在这之后才可以使用临时保留字。这之后开始执行PHP脚本，整个执行过程中都使用临时保留字。

4、调用zend\_interned\_strings\_deactivate()函数，销毁临时保留字哈希表。

5、调用调用zend\_interned\_strings\_switch\_storage()函数，切换到使用永久保留字。

6、调用zend\_interned\_strings\_dtor()函数，销毁永久保留字，之后就不能使用保留字了。

zend\_interned\_strings\_switch\_storage()函数用于切换使用的保留字类型，代码如下：

|  |
| --- |
| ZEND\_API void zend\_interned\_strings\_switch\_storage(bool request) {  if (request) { // php\_module\_startup 调用，切换到使用临时保留字  zend\_new\_interned\_string = interned\_string\_request\_handler;  zend\_string\_init\_interned = interned\_string\_init\_request\_handler;  zend\_string\_init\_existing\_interned = interned\_string\_init\_existing\_request\_handler;  } else { // php\_module\_shutdown 调用，切换到使用永久保留字  zend\_new\_interned\_string = zend\_new\_interned\_string\_permanent;  zend\_string\_init\_interned = zend\_string\_init\_interned\_permanent;  zend\_string\_init\_existing\_interned = zend\_string\_init\_existing\_interned\_permanent;  }  } |

### 6、销毁保留字

如前文所述，保留字不能调用程序接口来销毁，只能在PHP进程结束时统一回收，相关函数如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| zend\_interned\_strings\_dtor() | 销毁永久保留字列表 | 少量 |  |
| zend\_interned\_strings\_deactivate() | 调用zend\_hash\_destroy()函数，销毁临时保留字列表 | 少量 | 逻辑简单 |

zend\_interned\_strings\_dtor()函数用于销毁永久保留字哈希表，并释放常用字符串：

|  |
| --- |
| ZEND\_API void zend\_interned\_strings\_dtor(void) {  zend\_hash\_destroy(&interned\_strings\_permanent); // 销毁保留字哈希表  free(zend\_known\_strings); // 销毁常用字符串  zend\_known\_strings = NULL; // 清空指针  } |

zend\_interned\_strings\_dtor()函数在main.c文件中的php\_module\_shutdown()函数中被调用。

## 三）智能字符串（smart\_str）

智能字符串是PHP内核和扩展中使用频率非常高的功能，它的主要功能是拼接字符串，在一个字符串后面追加各种格式的数据。

使用智能字符串需要用到smart\_str结构体，它zend\_smart\_string\_public.h中定义，代码如下：

|  |
| --- |
| typedef struct { // 共16Bytes  zend\_string \*s; // 字符串指针，8Bytes  size\_t a; // 这是分配的长度，不同于 zend\_string中的长度是实际使用的长度。占8Bytes  } smart\_str; |

注意：在实际使用中，smart\_str实例通常都创建成临时变量，而不是分配内存创建实例。例如：

|  |
| --- |
| // 第一个元素zend\_string指针，必须初始化成0（NULL），否则无法正常分配zend\_string实例  smart\_str id = {0}; |

smart\_str结构非常简单，与其相关的函数和宏程序，大部分业务逻辑都是在操作zend\_string。

### 1、创建智能字符串

创建智能字符串相关的函数和宏程序有如下几个：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| SMART\_STR\_NEW\_LEN() | 计算smart\_str实例的长度 | 少量 |  |
| smart\_str\_erealloc() | 调整smart\_str实例的大小，只保留指定的长度 | 少量 |  |
| smart\_str\_realloc() | 调整smart\_str实例的大小 | 少量 |  |
| smart\_str\_alloc() | 创建smart\_str实例或为已有的智能字符串增加长度（注意是增加，不是调整大小） | 少量 |  |

smart\_str\_alloc()函数用于创建智能字符串或为已有的智能字符串增加长度：

|  |
| --- |
| static size\_t smart\_str\_alloc(smart\_str \*str, size\_t len, bool persistent) {  if (UNEXPECTED(!str->s)) { // 如果没有分配zend\_string  goto do\_smart\_str\_realloc; // 跳到分配  } else { // 如果已经有zend\_string  // 把原有zend\_string实例长度和要求长度加起来，得到需要的长度  len += ZSTR\_LEN(str->s);  if (UNEXPECTED(len >= str->a)) { // 如果需要的长度大于现有长度  do\_smart\_str\_realloc: // 重新分配 zend\_string  if (persistent) { // 调整smart\_str的大小  smart\_str\_realloc(str, len);  } else { // 调整smart\_str的大小，只保留指定的长度  smart\_str\_erealloc(str, len);  }  }  // 现有长度够用时，什么也不做  }  return len; // 返回长度  } |

如上所示，当需要调整大小时，根据情况调用smart\_str\_realloc()函数或smart\_str\_erealloc()函数。smart\_str\_realloc()函数注释如下：

|  |
| --- |
| ZEND\_API void smart\_str\_erealloc(smart\_str \*str, size\_t len) {  if (UNEXPECTED(!str->s)) { // 如果未创建 zend\_string  // 计算长度  str->a = len <= SMART\_STR\_START\_LEN  ? SMART\_STR\_START\_LEN  : SMART\_STR\_NEW\_LEN(len);  str->s = zend\_string\_alloc(str->a, 0); // 创建 zend\_string  ZSTR\_LEN(str->s) = 0; // 长度设置成 0  } else { // 如果已有 zend\_string  str->a = SMART\_STR\_NEW\_LEN(len); // 重新计算长度  // 调整zend\_string 大小  str->s = (zend\_string \*) erealloc2(str->s, str->a + \_ZSTR\_HEADER\_SIZE + 1, \_ZSTR\_HEADER\_SIZE + ZSTR\_LEN(str->s));  }  } |

用于计算大小的常量和宏程序定义如下：

|  |
| --- |
| // ZEND\_MM\_OVERHEAD 生产环境为 0, \_ZSTR\_HEADER\_SIZE 是 zend\_string 结构体中 val 元素的偏移量24, +1因为要多一个结束字符\0。生产环境值为25  #define SMART\_STR\_OVERHEAD (ZEND\_MM\_OVERHEAD + \_ZSTR\_HEADER\_SIZE + 1)  // 最小的智能字符串长度  #define SMART\_STR\_START\_SIZE 256  // 最小的智能字符串长度中，实际用来存储字符串的长度。生产环境为 256-25 = 231  #define SMART\_STR\_START\_LEN (SMART\_STR\_START\_SIZE - SMART\_STR\_OVERHEAD)  // 每个smart\_str页的大小，每次分配或调整尺寸，大小对齐到页  #define SMART\_STR\_PAGE 4096  // 计算smart\_str长度，取得最小的的能存下 len 的 4096 的倍数，然后减掉头信息  #define SMART\_STR\_NEW\_LEN(len) \  (ZEND\_MM\_ALIGNED\_SIZE\_EX(len + SMART\_STR\_OVERHEAD, SMART\_STR\_PAGE) - SMART\_STR\_OVERHEAD) |

smart\_str\_realloc()函数与smart\_str\_erealloc()函数逻辑相似，只是分配内存时调用的函数不同，smart\_str\_realloc()函数使用perealloc()函数分配内存：

|  |
| --- |
| str->s = (zend\_string \*) perealloc(str->s, str->a + \_ZSTR\_HEADER\_SIZE + 1, 1); |

这两个函数的不同是使用erealloc2()函数分配内存可以指定需要复制的数据长度，详情参见“内存管理篇”。

### 2、给智能字符串追加内容

给智能字符串实例追加内容，是智能字符串最重要的功能，为了方便追加各种类型的数据，相关函数比较多：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| smart\_str\_appends\_ex() | 调用smart\_str\_appendl\_ex()函数，为smart\_str实例追加字符串(const char \*) | 少量 | 逻辑简单 |
| smart\_str\_appendl() | 调用smart\_str\_appendl\_ex()函数，smart\_str实例追加字符串常量(const char \*)，带长度参数 | 大量 | 逻辑简单 |
| smart\_str\_0() | 给字符串添加结束标记 | 大量 | 逻辑简单 |
| smart\_str\_appendc() | 调用smart\_str\_appendc\_ex()函数，向smarty\_str实例中追加单个字符 | 大量 | 逻辑简单 |
| smart\_str\_appends() | 调用smart\_str\_appendl\_ex()函数，smart\_str实例追加 字符串常量(const char \*)，不带长度参数 | 大量 | 逻辑简单 |
| smart\_str\_append() | 调用smart\_str\_append\_ex()函数，smart\_str实例追加 zend\_string | 大量 | 逻辑简单 |
| smart\_str\_append\_long() | 调用smart\_str\_append\_long\_ex()函数，smart\_str实例追加 整数 | 大量 | 逻辑简单 |
| smart\_str\_append\_unsigned() | 调用smart\_str\_append\_unsigned\_ex()函数，smart\_str 追加 无符号整数 | 大量 | 逻辑简单 |
| smart\_str\_appendl\_ex() | 给智能字符串追加内容 | 少量 |  |
| smart\_str\_append\_unsigned\_ex() | 给智能字符串追加无符号整数 | 少量 |  |
| smart\_str\_append\_smart\_str() | 调用smart\_str\_append\_smart\_str\_ex()方法，给智能字符串追加内容 | 少量 | 逻辑简单 |
| smart\_str\_appendc\_ex() | 给smarty\_str实例追加单个字符 | 少量 |  |
| smart\_str\_append\_ex() | 给smart\_str实例追加普通字符串 | 少量 | 逻辑简单 |
| smart\_str\_append\_long\_ex() | 给smart\_str 追加整数 | 少量 |  |
| smart\_str\_append\_smart\_str\_ex() | 给smart\_str实例追加smart\_str实例 | 少量 | 逻辑简单 |
| smart\_str\_append\_escaped() | 把原字符串转义后追加到smart\_str实例后面 | 少量 | 逻辑简单 |
| smart\_str\_append\_double() | 给smart\_str实例追加浮点数 | 少量 |  |
| smart\_str\_append\_scalar() | 给smart\_str实例追加标量 | 少量 |  |
| smart\_str\_append\_printf() | 调用zend\_printf\_to\_smart\_str()函数，为smart\_str实例追加任意多个参数 | 少量 |  |
| zend\_compute\_escaped\_string\_len() | 计算字符串转义后多出来的长度 | 少量 |  |
| smart\_str\_append\_escaped\_truncated() | 调用smart\_str\_append\_escaped()函数，为smart\_str实例追加转义后的字符串，如果截断，添加截断标记 | 少量 | 逻辑简单 |

smart\_str\_appendl\_ex()函数用来向智能字符串中追加内容：

|  |
| --- |
| static void smart\_str\_appendl\_ex(smart\_str \*dest, const char \*str, size\_t len, bool persistent) {  size\_t new\_len = smart\_str\_alloc(dest, len, persistent); // 先增加长度  // 直接使用内存复制，把字符串复制过来  memcpy(ZSTR\_VAL(dest->s) + ZSTR\_LEN(dest->s), str, len);  ZSTR\_LEN(dest->s) = new\_len; // 更新zend\_string实例的长度  } |

#### **1）追加整数**

smart\_str\_append\_unsigned\_ex()方法用于向智能字符串中追加无符号整数：

|  |
| --- |
| static void smart\_str\_append\_unsigned\_ex(smart\_str \*dest, zend\_ulong num, bool persistent) {  char buf[32]; // 先创建一个32Bytes的缓冲区，除去1个终止字符，可以放31个数字  // 把 无符号整数 打印到缓冲区中（传入的指针指向缓冲区结尾）  char \*result = zend\_print\_ulong\_to\_buf(buf + sizeof(buf) - 1, num);  // 当成字符串追加到smart\_str里  smart\_str\_appendl\_ex(dest, result, buf + sizeof(buf) - 1 - result, persistent);  } |

zend\_print\_ulong\_to\_buf()函数用于把整数转成字符串类型：

|  |
| --- |
| static char \*zend\_print\_ulong\_to\_buf(char \*buf, zend\_ulong num) {  \*buf = '\0'; // 在结尾先添加终止字符  do { // 遍历每个数字  // 把数字转成字符的方式是从字符'0'开始向后偏移n位，取得ascii码表中的对应字符  \*--buf = (char) (num % 10) + '0'; // 添加最后一位数字，buf指针前移  num /= 10; // 下一位数字  } while (num > 0); // 直到长度用完  return buf;  } |

smart\_str\_append\_long\_ex()方法用于向智能字符串追加有符号整数，它的逻辑与追加无符号整数类似，只是调用zend\_print\_long\_to\_buf()函数来转义要追加的整数：

|  |
| --- |
| static char \*zend\_print\_long\_to\_buf(char \*buf, zend\_long num) {  if (num < 0) { // 负整数  // 先把负整数转成正整数：取反+1  char \*result = zend\_print\_ulong\_to\_buf(buf, ~((zend\_ulong) num) + 1);  \*--result = '-'; // 最后在前面加个负号  return result;  } else { // 正整数  return zend\_print\_ulong\_to\_buf(buf, num);  }  } |

#### **2）追加单个字符**

smart\_str\_appendc\_ex()函数用于向智能字符串中追加单个字符：

|  |
| --- |
| static void smart\_str\_appendc\_ex(smart\_str \*dest, char ch, bool persistent) {  size\_t new\_len = smart\_str\_alloc(dest, 1, persistent); // 先增加长度  ZSTR\_VAL(dest->s)[new\_len - 1] = ch; // 然后直接写入字符  ZSTR\_LEN(dest->s) = new\_len; // 更新 zend\_string 长度  } |

#### **3）追加浮点数**

smart\_str\_append\_double()函数用于向智能字符串追加浮点数（小数）：

|  |
| --- |
| ZEND\_API void smart\_str\_append\_double(  smart\_str \*str, double num, int precision, bool zero\_fraction) {  char buf[ZEND\_DOUBLE\_MAX\_LENGTH]; // 长度1080  // 把小数转成字符串，打印到buffer里。默认保留1位小数，小数点"."，科学计数法标记“E”  zend\_gcvt(num, precision ? precision : 1, '.', 'E', buf);  smart\_str\_appends(str, buf); // 把buffer里的字符串追加到smart\_str后面  // 如果需要在整数后面补.0,并且当前是整数  if (zero\_fraction && zend\_finite(num) && !strchr(buf, '.')) {  smart\_str\_appendl(str, ".0", 2); // 补.0  }  }  define ZEND\_DOUBLE\_MAX\_LENGTH 1080 // 小数最大长度1080 |

zend\_gcvt()函数用于把浮点数转成字符串打针到缓冲区中，相关业务逻辑比较复杂，在zend\_strtod.c文件中，这部分内容不是本篇重点，暂时略过。

#### **4）追加标量（scalar）**

标量（scalar）是指IS\_UNDEF、IS\_NULL、IS\_FALSE、IS\_TRUE、IS\_LONG、IS\_DOUBLE、IS\_STRING七种类型的zval。

给智能字符串追加标量使用smart\_str\_append\_scalar()函数，逻辑很简单：

|  |
| --- |
| ZEND\_API void smart\_str\_append\_scalar(smart\_str \*dest, zval \*value, size\_t truncate) {  switch (Z\_TYPE\_P(value)) { // 根据类型处理  case IS\_UNDEF:  case IS\_NULL:  // -1 是减掉\0结束字符占用的长度  smart\_str\_appendl(dest, "NULL", sizeof("NULL")-1);  break;  case IS\_TRUE:  case IS\_FALSE:  smart\_str\_appends(dest, Z\_TYPE\_P(value) == IS\_TRUE ? "true" : "false");  break;  case IS\_DOUBLE:  // 追加小数，精度使用全局设置EG(precision)，整数要补“.0”  smart\_str\_append\_double(dest, Z\_DVAL\_P(value), (int) EG(precision), true);  break;  case IS\_LONG:  smart\_str\_append\_long(dest, Z\_LVAL\_P(value)); // 追加整数  break;  case IS\_STRING: // 追加字符串  smart\_str\_appendc(dest, '\''); // 前后添加单引号'  // 把转义过的字符串放在引号中间  smart\_str\_append\_escaped\_truncated(dest, Z\_STR\_P(value), truncate);  smart\_str\_appendc(dest, '\'');  break;  EMPTY\_SWITCH\_DEFAULT\_CASE();  }  } |

#### **5）追加需要转义的内容**

smart\_str\_append\_escaped()函数用于给智能字符串追加需要转义的内容，函数代码比较简单，阅读它可以加强对“转义字符”的理解：

|  |
| --- |
| ZEND\_API void smart\_str\_append\_escaped(smart\_str \*str, const char \*s, size\_t l) {  char \*res;  size\_t i, len = zend\_compute\_escaped\_string\_len(s, l); // 计算转义后的长度  smart\_str\_alloc(str, len, 0); // 分配空间  res = &ZSTR\_VAL(str->s)[ZSTR\_LEN(str->s)]; // 找到末尾位置，准备添加  ZSTR\_LEN(str->s) += len; // 给zend\_string实例增加长度  for (i = 0; i < l; ++i) { // 遍历 原字符串中 每一个字符  unsigned char c = s[i]; // 读取一个字符  if (c < 32 || c == '\\' || c > 126) { // 如果当前字符是控制字符 或 反斜线（\）  \*res++ = '\\'; // 先放一个 \，然后指针右移  switch (c) { // 根据当前字符进行操作  // 处理一些特殊功能字符  case '\n': \*res++ = 'n'; break;  case '\r': \*res++ = 'r'; break;  case '\t': \*res++ = 't'; break;  case '\f': \*res++ = 'f'; break;  case '\v': \*res++ = 'v'; break;  case '\\': \*res++ = '\\'; break;  case VK\_ESCAPE: \*res++ = 'e'; break;  // 其他控制字符，转成\x开头的16进制字符串，共占4字节（所以要加3个字符）  default:  \*res++ = 'x'; // 先写一个x，凑成\x，指针右移  // 以下是把一个Byte转成两个16进制字符，先处理左面4个bit  if ((c >> 4) < 10) { // c >> 4 得到左面4个bit, 如果 < 10  \*res++ = (c >> 4) + '0'; // 写数字，然后指针右移  } else { // 如果 >= 10  \*res++ = (c >> 4) + 'A' - 10; // 写A-F，然后指针右移  }  // 再处理右面4个bit  if ((c & 0xf) < 10) { // 右面4个bit , 如果 < 10  \*res++ = (c & 0xf) + '0'; // 写数字，然后 smart\_str 指针右移  } else { // 如果 >=10  \*res++ = (c & 0xf) + 'A' - 10; // 写A-F，然后 smart\_str 指针右移  }  }  } else { // 如果当前字符是普通字符  \*res++ = c; // 把它写入当前位置，然后 smart\_str 指针右移  }  }  } |

转义操作可以把字符归纳为4类：

1）反斜线"\"（ASCII码为92），转义成"\\"。

2）ASCII码在32-126之间（包含32和126）的其他字符，不需要转义。

3）换行、制表符等特殊字符，ASCII码为：10、13、9、12、11、27，分别转义为：\n、\r、\t、\f、\v、\e。

4）其他字符，转成"\x"开头的16进制字符。

zend\_compute\_escaped\_string\_len()函数用于计算普通字符串转义后的长度：

|  |
| --- |
| static size\_t zend\_compute\_escaped\_string\_len(const char \*s, size\_t l) {  size\_t i, len = l;  for (i = 0; i < l; ++i) { // 遍历每个字符  char c = s[i]; // 当前字符  // 需要加一个"\"的字符  if (c == '\n' || c == '\r' || c == '\t' ||  c == '\f' || c == '\v' || c == '\\' || c == VK\_ESCAPE) {  len += 1;  // 其他字符，每个字符转成\x开头的16进制字符串，占4字节，所以要加3个字符  } else if (c < 32 || c > 126) {  len += 3;  }  }  return len; // 返回长度  } |

#### **6）按给定格式打印**

smart\_str\_append\_printf()函数用于按给定格式，把给出的参数列表打印到智能字符串当中：

|  |
| --- |
| ZEND\_API void smart\_str\_append\_printf(smart\_str \*dest, const char \*format, ...) {  va\_list arg;  va\_start(arg, format);  zend\_printf\_to\_smart\_str(dest, format, arg); // php\_printf\_to\_smart\_str函数  va\_end(arg);  } |

它实际上是调用main/spprintf.c中的php\_printf\_to\_smart\_str()函数，代码来自apache snprintf，这部分内容不是本篇重点，暂时略过。

### 3、修改和读取智能字符串

修改和读取智能字符串相关的方法如下，业务逻辑都比较简单：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| smart\_str\_extend\_ex() | 给smart\_str实例增加长度，返回 char \* 结尾位置 | 少量 | 逻辑简单 |
| smart\_str\_extend() | 调用smart\_str\_extend\_ex()函数，给smart\_str实例增加长度 | 少量 | 逻辑简单 |
| smart\_str\_trim\_to\_size\_ex() | 调用zend\_string\_realloc()函数，把zend\_string实例截取到已使用的长度 | 少量 | 逻辑简单 |
| smart\_str\_setl() | 释放zend\_string实例后重新添加内容 | 少量 | 逻辑简单 |
| smart\_str\_trim\_to\_size() | 调用smart\_str\_trim\_to\_size\_ex()函数把smart\_str实例截取到 zend\_string已使用的长度 | 无 | 逻辑简单 |
| smart\_str\_sets() | 调用smart\_str\_setl()函数，释放 zend\_string实例后重新添加内容 | 无 | 逻辑简单 |
| smart\_str\_get\_len() | 获取smart\_str实例的长度 | 少量 | 逻辑简单 |
| smart\_str\_extract() | 调用smart\_str\_extract\_ex()函数，返回并删除 smart\_str实例指向的zend\_string实例 | 大量 | 逻辑简单 |
| smart\_str\_extract\_ex() | 返回 zend\_string实例，并在smart\_str上解除关联 | 少量 |  |

smart\_str\_extract\_ex()函数用于读取智能字符串中的内容：

|  |
| --- |
| static zend\_string \*smart\_str\_extract\_ex(smart\_str \*str, bool persistent) {  if (str->s) { // 如果已分配 zend\_string  zend\_string \*res;  smart\_str\_0(str); // 先把结尾标记好  smart\_str\_trim\_to\_size\_ex(str, persistent); // 把zend\_string截短到已使用长度  res = str->s; // 取出 zend\_string  str->s = NULL; // 清空 zend\_string 指针  return res; // 返回zend\_string指针  } else {  return ZSTR\_EMPTY\_ALLOC(); // 返回空zend\_string的指针  }  } |

### 4、释放智能字符串

释放智能字符串其实是释放zend\_string实例，相关方法如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| smart\_str\_free() | 调用smart\_str\_free\_ex()函数释放smart\_str实例指向的的zend\_string实例 | 大量 | 逻辑简单 |
| smart\_str\_free\_ex() | 释放smart\_str实例指向的zend\_string实例 | 少量 | 逻辑简单 |

## 四）简易智能字符串（smart string）

除了智能字符串smart\_str，PHP内核中还有一个简易版本的智能字符串，这个版本被调用的次数比较少，主要被/main/spprintf.c和/main/spprintf.c文件调用。简易智能字符串（smart string）在zend\_smart\_string.h和zend\_smart\_string\_public.h两个文件中定义，结构体如下：

|  |
| --- |
| typedef struct {  char \*c; // 字符串指针  size\_t len; // 已用长度  size\_t a; // 总长度  } smart\_string; |

相关函数与smart\_str类似，只是没有用到zend\_string，逻辑也比较简单：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| smart\_string\_alloc() | 分配 smart\_string，注意 smart\_string 不是 smart\_str | 少量 |  |
| \_smart\_string\_alloc\_persistent() | 分配内存创建，zend\_smart\_str.c中定义 | 少量 |  |
| \_smart\_string\_alloc() | 分配内存创建，zend\_smart\_str.c中定义 | 少量 |  |
| smart\_string\_0() | 添加结束字串 \0 | 少量 | 逻辑简单 |
| smart\_string\_appendl() | 调用smart\_string\_appendl\_ex()函数，追加字串,需要传入长度 | 少量 | 逻辑简单 |
| smart\_string\_appends() | 调用smart\_string\_appendl()函数，追加字符串(char \*) | 少量 | 逻辑简单 |
| smart\_string\_appendc() | 调用smart\_string\_appendc\_ex()函数，追加 1个字符 | 少量 | 逻辑简单 |
| smart\_string\_free() | 调用smart\_string\_free\_ex()函数，释放 smart\_string 中的字串，不销毁自身 | 少量 | 逻辑简单 |
| smart\_string\_appendl\_ex() | 追加普通字符串(char \*)，需要传入长度 | 少量 | 逻辑简单 |
| smart\_string\_append\_ex() | 追加另一个 smart\_string实例 | 少量 | 逻辑简单 |
| smart\_string\_free\_ex() | 释放 smart\_string实例指向的的字串，不销毁实例自身 | 少量 | 逻辑简单 |
| smart\_string\_appendc\_ex() | 追加 1个字符 | 少量 | 逻辑简单 |
| smart\_string\_append\_long\_ex() | 追加整型数 | 少量 | 逻辑简单 |
| smart\_string\_append\_unsigned\_ex() | 追加无符号整型数 | 少量 | 逻辑简单 |
| smart\_string\_setl() | 直接更新smart\_string实例的len,a,c这3个元素 | 少量 | 逻辑简单 |
| smart\_string\_reset() | 长度更新为0 | 少量 | 逻辑简单 |
| smart\_string\_appends\_ex() | 追加字符串(char \*)，自定义分配方式 | 无 | 逻辑简单 |
| smart\_string\_sets() | 手工分配字串给 smart\_string | 无 | 逻辑简单 |
| smart\_string\_append() | 创建或追加已有的 smart\_string | 无 | 逻辑简单 |
| smart\_string\_append\_long() | 追加整数 | 无 | 逻辑简单 |
| smart\_string\_append\_unsigned() | 追加无符号整数 | 无 | 逻辑简单 |

smart\_string\_alloc()函数用于分配内存：

|  |
| --- |
| static size\_t smart\_string\_alloc(smart\_string \*str, size\_t len, bool persistent) {  // 如果没有分配字符串 或 空间不够用  if (UNEXPECTED(!str->c) || UNEXPECTED(len >= str->a - str->len)) {  if (persistent) { // 原生方法分配  \_smart\_string\_alloc\_persistent(str, len);  } else { // 非原生方法分配  \_smart\_string\_alloc(str, len);  }  }  return str->len + len; // 返回已使用长度，不是总长度  } |

\_smart\_string\_alloc\_persistent()函数：

|  |
| --- |
| ZEND\_API void \_smart\_string\_alloc\_persistent(smart\_string \*str, size\_t len)  {  if (!str->c) { // 如果没分配字符串空间  str->len = 0; // 已用长度是0  if (len <= SMART\_STRING\_START\_LEN) { // <= 255  str->a = SMART\_STRING\_START\_LEN; // 占用的长度是255  } else { // > 255  // 用户的长度是 4096\*n - 1  str->a = ZEND\_MM\_ALIGNED\_SIZE\_EX(len + SMART\_STRING\_OVERHEAD, SMART\_STRING\_PAGE) - SMART\_STRING\_OVERHEAD;  }  str->c = pemalloc(str->a + 1, 1); // 分配内存空间，多一个字节是 \0  } else { // 如果已经分配了内存空间，添加长度  // 如果要求大小大于 最大size，报错  // #define SIZE\_MAX ((size\_t)~0) // 最大的64位无符号整数  if (UNEXPECTED((size\_t) len > SIZE\_MAX - str->len)) {  zend\_error(E\_ERROR, "String size overflow");  }  len += str->len; // 添加长度  // 计算长度  str->a = ZEND\_MM\_ALIGNED\_SIZE\_EX(len + SMART\_STRING\_OVERHEAD, SMART\_STRING\_PAGE) - SMART\_STRING\_OVERHEAD;  str->c = perealloc(str->c, str->a + 1, 1); // 重新分配内存  }  } |

\_smart\_string\_alloc()函数：

|  |
| --- |
| ZEND\_API void \_smart\_string\_alloc(smart\_string \*str, size\_t len) {  if (!str->c) { // 如果还没有分配字符串  str->len = 0; // 使用长度为0  if (len <= SMART\_STRING\_START\_LEN) { // 如果要求的长度小于初始长度  str->a = SMART\_STRING\_START\_LEN; // 使用初始长度作为可用长度  // 普通string 分配内存时要多占 1 字节,\0  str->c = emalloc(SMART\_STRING\_START\_LEN + 1);  } else { // 如果要求的长度大于初始长度  // 长度对齐到 4096 ，减掉头大小  str->a = ZEND\_MM\_ALIGNED\_SIZE\_EX(len + SMART\_STRING\_OVERHEAD, SMART\_STRING\_PAGE) - SMART\_STRING\_OVERHEAD;  // 小于2M - SMART\_STRING\_OVERHEAD  if (EXPECTED(str->a < (ZEND\_MM\_CHUNK\_SIZE - SMART\_STRING\_OVERHEAD))) {  str->c = emalloc\_large(str->a + 1); // 分配大块，多分配一个 \0  } else { // 大于2M  str->c = emalloc(str->a + 1); // 分配巨大块，多分配一个 \0  }  }  } else { // 如果已经分配了字符串空间  if (UNEXPECTED((size\_t) len > SIZE\_MAX - str->len)) { // 如果长度超限  zend\_error(E\_ERROR, "String size overflow"); // 报错：字符串溢出  }  len += str->len; // 更新长度  // 计算大小  str->a = ZEND\_MM\_ALIGNED\_SIZE\_EX(len + SMART\_STRING\_OVERHEAD, SMART\_STRING\_PAGE) - SMART\_STRING\_OVERHEAD;  str->c = erealloc2(str->c, str->a + 1, str->len); // 重新分配内存  }  } |

相关常量也与smart\_str类似，定义如下：

|  |
| --- |
| #define ZEND\_MM\_CHUNK\_SIZE ((size\_t) (2 \* 1024 \* 1024)) // zend\_alloc\_sizes.h中定义，2 MB  // ZEND\_MM\_OVERHEAD 生产环境为 0, SMART\_STRING\_OVERHEAD = 1，就是\0结束字符  #define SMART\_STRING\_OVERHEAD (ZEND\_MM\_OVERHEAD + 1)  #define SMART\_STRING\_START\_SIZE 256 // 初始长度256  // SMART\_STRING\_START\_LEN = 256-1 = 255  #define SMART\_STRING\_START\_LEN (SMART\_STRING\_START\_SIZE -  SMART\_STRING\_OVERHEAD)  #define SMART\_STRING\_PAGE 4096 // 每个页大小4096，4k |

# 四、引用类型

在PHP语言中，不能直接操作内存指针，所以当需要传递一个实例的地址时，需要用到引用类型（zend\_reference），它是PHP内核中的关键类型，应用非常普遍。但它也是PHP内核中结构最简单的类型之一，结构体定义如下：

|  |
| --- |
| typedef struct \_zend\_reference zend\_reference;  struct \_zend\_reference { // 占32Bytes  zend\_refcounted\_h gc; // 共8Bytes，64位  zval val; // 变量，引用结构体中包含一个内置变量，而不是变量指针，16Bytes，128位  zend\_property\_info\_source\_list sources; // 一个指针，占8Bytes  };  typedef union { // 64位系统中， 8bytes，64位  struct \_zend\_property\_info \*ptr; // \_zend\_property\_info在zend\_compile.h中定义  uintptr\_t list; // 8bytes，64位  } zend\_property\_info\_source\_list; |

引用类型的作用是用来指向另一个变量（zval实例），如上所示，最关键的指针元素保存在它自己内部包含的val元素（也是zval实例）中。

在上述结构中，zend\_reference.sources元素在本篇中没有用到，在“面向对象篇”中会有详细的介绍。

引用类型的相关操作比较简单，通过一些宏程序即可完成：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| Z\_ISREF() | 检验zval实例是否是引用类型  (Z\_TYPE(zval) == IS\_REFERENCE) | 大量 | 逻辑简单 |
| Z\_ISREF\_P() | 通过指针检验zval实例是否是引用类型  Z\_ISREF(\*(zval\_p)) | 大量 | 逻辑简单 |
| Z\_OPT\_ISREF() | 检验zval实例是否是引用类型（优化版）  (Z\_OPT\_TYPE(zval) == IS\_REFERENCE) | 少量 | 逻辑简单 |
| Z\_OPT\_ISREF\_P() | 通过指针检验zval实例是否是引用类型（优化版）  Z\_OPT\_ISREF(\*(zval\_p)) | 少量 | 逻辑简单 |
| Z\_REF() | 访问 (zval).value.ref | 少量 | 逻辑简单 |
| Z\_REF\_P() | 通过指针访问 (zval).value.ref  Z\_REF(\*(zval\_p)) | 大量 | 逻辑简单 |
| Z\_REFVAL() | 访问zval实例指向引用对象的val元素  &Z\_REF(zval)->val | 大量 | 逻辑简单 |
| Z\_REFVAL\_P() | 通过指针访问zval实例指向引用对象的val元素  Z\_REFVAL(\*(zval\_p)) | 大量 | 逻辑简单 |
| ZVAL\_COPY\_DEREF() | 如果是引用类型，先追踪到引用目标。然后复制目标实例的value和u1.type\_info两个元素 | 大量 |  |
| ZVAL\_DEREF() | 如果zval实例是引用类型，追踪到引用目标  if (UNEXPECTED(Z\_ISREF\_P(z))) { (z) = Z\_REFVAL\_P(z);} | 大量 | 逻辑简单 |
| ZVAL\_NEW\_REF() | 创建一个引用实例，把给定zval实例中的数据复制到新实例中，并让当前zval实例指向新实例 | 大量 |  |
| ZVAL\_NEW\_PERSISTENT\_REF() | 与ZVAL\_NEW\_REF()宏程序逻辑相似，只是使用原生的malloc()函数创建zend\_reference 实例，并给实例添加【使用原生方法创建】标记 | 少量 | 逻辑类似 |
| ZVAL\_NEW\_EMPTY\_REF() | 创建一个引用实例，并让当前zval实例指向新实例 | 少量 | 逻辑类似 |
| ZVAL\_MAKE\_REF() | 如果当前zval不是引用类型，调用ZVAL\_NEW\_REF()宏程序把它变成引用类型 | 大量 | 逻辑简单 |
| ZVAL\_MAKE\_REF\_EX() | 创建新引用实例，指向给出的zval实例，并设置引用次数 | 少量 |  |
| ZVAL\_OPT\_DEREF() | 如果zval实例是引用类型，追踪到引用目标（优化版）  if (UNEXPECTED(Z\_OPT\_ISREF\_P(z))) { (z) = Z\_REFVAL\_P(z);} | 少量 | 逻辑简单 |
| ZVAL\_REF() | 给zval.ref 添加 zend\_reference 实例指针, 并修改type\_info | 大量 |  |
| ZVAL\_UNREF() | 解除引用，让给出的zval变成它的引用目标 | 大量 |  |

ZVAL\_COPY\_DEREF宏程序的功能是：先追踪到引用实例的目标，然后复制目标实例的value和u1.type\_info两个元素。

|  |
| --- |
| #define ZVAL\_COPY\_DEREF(z, v) do { \  zval \*\_z3 = (v); \  if (Z\_OPT\_REFCOUNTED\_P(\_z3)) { /\* 如果是可计数的类型\*/ \  if (UNEXPECTED(Z\_OPT\_ISREF\_P(\_z3))) { /\* 如果是 引用 类型 \*/ \  \_z3 = Z\_REFVAL\_P(\_z3); /\* 追踪到引用目标 \*/ \  if (Z\_OPT\_REFCOUNTED\_P(\_z3)) { /\* 如果引用目标可计数 \*/ \  Z\_ADDREF\_P(\_z3); /\* 增加引用计数 \*/ \  } \  } else { /\* 如果v不是引用类型 \*/ \  Z\_ADDREF\_P(\_z3); /\* 增加引用计数 \*/ \  } \  } \  /\* 复制zval的value和u1.type\_info两个元素 \*/ \  ZVAL\_COPY\_VALUE(z, \_z3); \  } while (0) |

ZVAL\_NEW\_REF()宏程序的用途是：创建一个引用实例，把给定zval实例中的数据复制到新实例中，并让当前zval实例指向新实例：

|  |
| --- |
| #define ZVAL\_NEW\_REF(z, r) do { \  /\* 创建新的引用实例（自带zval） \*/ \  zend\_reference \*\_ref = (zend\_reference \*) emalloc(sizeof(zend\_reference)); \  GC\_SET\_REFCOUNT(\_ref, 1); /\* 引用数为1 \*/ \  GC\_TYPE\_INFO(\_ref) = GC\_REFERENCE; /\* 引用实例类型为：不可回收的引用类型 \*/ \  ZVAL\_COPY\_VALUE(&\_ref->val, r); /\* 复制zval的value和u1.type\_info两个元素 \*/ \  \_ref->sources.ptr = NULL; /\* 新实例初始指向目标为null \*/ \  Z\_REF\_P(z) = \_ref; /\* 当前 zval 实例指向引用实例 \*/ \  Z\_TYPE\_INFO\_P(z) = IS\_REFERENCE\_EX; /\* zval 类型为 可计数的引用类型 \*/ \  } while (0) |

ZVAL\_MAKE\_REF\_EX()宏程序用于：创建新引用实例，指向给出的zval实例，并设置引用次数。

|  |
| --- |
| #define ZVAL\_MAKE\_REF\_EX(z, refcount) do { \  zval \*\_z = (z); \  /\* 创建引用实例（自带zval） \*/ \  zend\_reference \*\_ref = (zend\_reference \*) emalloc(sizeof(zend\_reference)); \  GC\_SET\_REFCOUNT(\_ref, (refcount)); /\* 自定义引用次数 \*/ \  GC\_TYPE\_INFO(\_ref) = GC\_REFERENCE; /\* 类型是引用 \*/ \  ZVAL\_COPY\_VALUE(&\_ref->val, \_z); /\* 把zval指针复制给 引用实例 \*/ \  \_ref->sources.ptr = NULL; /\* 没有用到这个指针 \*/ \  Z\_REF\_P(\_z) = \_ref; /\* 引用实例关联到原zval \*/ \  Z\_TYPE\_INFO\_P(\_z) = IS\_REFERENCE\_EX; /\* 原zval 类型为引用 \*/ \  } while (0) |

ZVAL\_REF()宏程序用于让给出的zval实例指向目标zend\_reference 实例，并更新zval实例的类型：

|  |
| --- |
| #define ZVAL\_REF(z, r) do { \  zval \*\_\_z = (z); \  Z\_REF\_P(\_\_z) = (r); /\* zval实例指向给出的zend\_reference 实例 \*/ \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_REFERENCE\_EX; /\* zval 类型为 可计数的引用类型 \*/ \  } while (0) |

ZVAL\_UNREF()宏程序用于解除引用，让给出的zval变成它的引用目标：

|  |
| --- |
| #define ZVAL\_UNREF(z) do { \  zval \*\_z = (z); \  zend\_reference \*ref; \  ref = Z\_REF\_P(\_z); /\* 取出zend\_reference实例指针 \*/ \  ZVAL\_COPY\_VALUE(\_z, &ref->val); /\* 把实例中的zval的数据复制给当前zval \*/ \  efree\_size(ref, sizeof(zend\_reference)); /\* 释放zend\_reference实例 \*/ \  } while (0) |

注意：这个操作会改变zval实例的地址，原zend\_reference实例中包含的zval实例被销毁掉了。

## 引用（zend\_reference）和间接引用（IS\_INDIRECT）的区别

引用类型（zend\_reference）是可以在PHP语句中使用的数据结构，间接引用（IS\_INDIRECT）是内部类型，不可以在PHP语句中使用。

在结构上区别如下：

1）引用类型：先从zval实例指向到一个zend\_reference实例，再从zend\_reference实例指向到另一个zval实例。

2）间接引用类型：从一个zval直接指向到另一个zval。

# 五、资源类型

资源类型（zend\_resource）是PHP内核中的关键类型，但它在PHP内核中很少使用，主要用在PHP的扩展中，如SPL扩展、php\_mysql扩展等，都用到资源类型。结构体定义如下：

|  |
| --- |
| struct \_zend\_resource {  zend\_refcounted\_h gc; // 引用计数器  zend\_long handle; // 资源实例编号  int type; // 类型  void \*ptr; // 指针  }; |

在PHP内核中定义的与资源相关的操作比少，只有一些简单的宏程序：

|  |  |  |  |
| --- | --- | --- | --- |
| **宏程序名** | **说明** | **引用**  **次数** | **备注** |
| Z\_RES() | 访问(zval).value.res | 少量 | 逻辑简单 |
| Z\_RES\_P() | 通过指针访问(zval).value.res | 大量 | 逻辑简单 |
| Z\_RES\_HANDLE() | 访问(zval).value.res->handle | 少量 | 逻辑简单 |
| Z\_RES\_HANDLE\_P() | 通过指针访问(zval).value.res->handle | 大量 | 逻辑简单 |
| Z\_RES\_TYPE() | 访问(zval).value.res->type | 少量 | 逻辑简单 |
| Z\_RES\_TYPE\_P() | 通过指针访问(zval).value.res->type | 少量 | 逻辑简单 |
| Z\_RES\_VAL() | 访问(zval).value.res->ptr | 少量 | 逻辑简单 |
| Z\_RES\_VAL\_P() | 通过指针访问(zval).value.res->ptr | 少量 | 逻辑简单 |
| ZVAL\_RES() | 把zval实例更新成资源类型 | 大量 |  |
| ZVAL\_NEW\_RES() | 创建新资源实例并关联到给定的zval实例 | 少量 |  |
| ZVAL\_NEW\_PERSISTENT\_RES() | 创原生分配的建新资源实例并关联到给定的zval实例 | 少量 | 逻辑类似 |

ZVAL\_RES()宏程序用于把zval实例更新成资源类型：

|  |
| --- |
| #define ZVAL\_RES(z, r) do { \  zval \*\_\_z = (z); \  Z\_RES\_P(\_\_z) = (r); 通过指针更新(zval).value.res \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_RESOURCE\_EX; /\* 可计数资源类型 \*/ \  } while (0) |

ZVAL\_NEW\_RES()宏程序用于创建新资源实例并关联到给定的zval实例：

|  |
| --- |
| #define ZVAL\_NEW\_RES(z, h, p, t) do { \  zend\_resource \*\_res = \  /\* 创建 zend\_reosource 实例 \*/ \  (zend\_resource \*) emalloc(sizeof(zend\_resource)); \  zval \*\_\_z; \  GC\_SET\_REFCOUNT(\_res, 1); /\* 引用次数设置成1 \*/ \  GC\_TYPE\_INFO(\_res) = GC\_RESOURCE; /\* 新实例回收类型为：为不可回收的资源 \*/ \  \_res->handle = (h); /\* 把 h,p,t 关联到新资源实例 \*/ \  \_res->type = (t); \  \_res->ptr = (p); \  \_\_z = (z); \  Z\_RES\_P(\_\_z) = \_res; /\* 更新(zval).value.res \*/ \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_RESOURCE\_EX; /\*类型为可计数资源\*/ \  } while (0) |

ZVAL\_NEW\_PERSISTENT\_RES()宏程序与ZVAL\_NEW\_RES()宏程序逻辑类似，只是新资源实例增加了GC\_PERSISTENT标记，表示创建实例时调用系统原生的malloc()函数分配内存。

# 六、常量表达式类型

IS\_CONSTANT\_AST类型，PHP官方给出的解释是“常量表达式”（Constant expressions）。在整个编译（compile）阶段，大部分PHP语句都被运算成了常量和操作码（operation code）,但有些PHP语句需要在运行阶段进行计算，这些语句就会被转换成常量表达式（IS\_CONSTANT\_AST）实例。

由于语句（zend\_ast）结构体本身没有计数器，这里需要用到一个辅助结构体zend\_ast\_ref，对应的数据结构如下：

|  |
| --- |
| typedef struct \_zend\_ast\_ref zend\_ast\_ref;  struct \_zend\_ast\_ref {  zend\_refcounted\_h gc; // 只封装了一个引用计数器  }; |

与IS\_CONSTANT\_AST类型相关宏程序如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| GC\_AST() | 跳过计数器，找到zend\_ast实例的开始位置  ((zend\_ast\*)(((char\*)p) + sizeof(zend\_ast\_ref))) | 少量 | 逻辑简单 |
| Z\_AST() | 访问zval实例找到zend\_ast实例的开始位置  (zval).value.ast | 少量 | 逻辑简单 |
| Z\_AST\_P() | 通过zval实例指针找到zend\_ast实例的开始位置  Z\_AST(\*(zval\_p)) | 大量 | 逻辑简单 |
| Z\_ASTVAL() | 通过zval实例，找到zend\_ast实例的开始位置  GC\_AST(Z\_AST(zval)) | 少量 | 逻辑简单 |
| Z\_ASTVAL\_P() | 通过zval实例指针，找到zend\_ast实例的开始位置  Z\_ASTVAL(\*(zval\_p)) | 大量 | 逻辑简单 |
| Z\_CONSTANT() | 检验zval实例是否是常量表达式类型  (Z\_TYPE(zval) == IS\_CONSTANT\_AST) | 少量 | 逻辑简单 |
| Z\_CONSTANT\_P() | 通过指针检验zval实例是否是常量表达式类型  Z\_CONSTANT(\*(zval\_p)) | 少量 | 逻辑简单 |
| Z\_OPT\_CONSTANT() | 检验zval实例是否是常量表达式类型（优化版本）  (Z\_OPT\_TYPE(zval) == IS\_CONSTANT\_AST) | 少量 | 逻辑简单 |
| Z\_OPT\_CONSTANT\_P() | 通过指针检验zval实例是否是常量表达式类型  Z\_OPT\_CONSTANT(\*(zval\_p)) | 无 | 逻辑简单 |
| ZVAL\_AST() | 把zval实例更新成常量表达式类型 | 少量 | 逻辑简单 |

ZVAL\_AST()宏程序用于把zval实例更新成常量表达式类型：

|  |
| --- |
| #define ZVAL\_AST(z, ast) do { \  zval \*\_\_z = (z); \  Z\_AST\_P(\_\_z) = ast; /\* 更新语句指针 \*/ \  Z\_TYPE\_INFO\_P(\_\_z) = IS\_CONSTANT\_AST\_EX; /\* 可计数常量表达式 \*/ \  } while (0) |

如前文所述，常量表达式类型的应用过程可以分成两个阶段：

## 1）编译阶段

ZVAL\_AST()宏程序只有在zend\_compile.c文件的zend\_const\_expr\_to\_zval()函数中被引用。zend\_const\_expr\_to\_zval()函数在编译静态变量、declare语句，修饰属性（attribute）、有默认值的形式参数（params）、类成员变量、常量、枚举（enum）等PHP语句时，都有可能用到。

zend\_const\_expr\_to\_zval()函数定义中下：

|  |
| --- |
| void zend\_const\_expr\_to\_zval(zval \*result, zend\_ast \*\*ast\_ptr, bool allow\_dynamic) {  ...  if ((\*ast\_ptr)->kind != ZEND\_AST\_ZVAL) {  zval ast\_zv;  ZVAL\_AST(&ast\_zv, zend\_ast\_copy(\*ast\_ptr)); // 把普通的语句实例转成常量表达式实例 |

zend\_ast\_copy()函数用于把普通的语句实例转成常量表达式实例：

|  |
| --- |
| ZEND\_API zend\_ast\_ref \* zend\_ast\_copy(zend\_ast \*ast) {  size\_t tree\_size;  zend\_ast\_ref \*ref;  // 内存大小 = 整个语句树的大小 + 一个引用计数器的大小  tree\_size = zend\_ast\_tree\_size(ast) + sizeof(zend\_ast\_ref);  ref = emalloc(tree\_size); // 开辟内存创建 zend\_ast\_ref 开头的语句树实例  zend\_ast\_tree\_copy(ast, GC\_AST(ref)); // 复制整个语句树到新内存中  GC\_SET\_REFCOUNT(ref, 1); // 引用次数为1  GC\_TYPE\_INFO(ref) = GC\_CONSTANT\_AST; // 类型为 不可回收的常量表达式  return ref;  } |

如上所示，先计算整个语句树的大小，再加上zend\_ast\_ref实例的大小，得到整个常量表达式实例的大小，然后调用zend\_ast\_tree\_copy()函数，把原语句树复制到新实例中。新实例中，zend\_ast\_ref实例在前，语句树实例在后。

## 2）执行阶段

在执行阶段，通常调用zval\_update\_constant\_ex()函数来执行常量表达式，例如：

|  |
| --- |
| if (Z\_TYPE\_P(value) == IS\_CONSTANT\_AST) {  zval\_update\_constant\_ex(value, c->ce); |

zval\_update\_constant\_ex()函数在zend\_execute\_API.c文件中定义，碰到需要计算的常量表达式语句，它调用zend\_ast\_evaluate()函数来进行计算：

|  |
| --- |
| ZEND\_API zend\_result zval\_update\_constant\_ex(zval \*p, zend\_class\_entry \*scope) {  if (Z\_TYPE\_P(p) == IS\_CONSTANT\_AST) {  zend\_ast \*ast = Z\_ASTVAL\_P(p);  if (ast->kind == ZEND\_AST\_CONSTANT) {  ...  } else {  ...  zend\_ast\_ref \*ast\_ref = Z\_AST\_P(p);  zend\_result result = zend\_ast\_evaluate(&tmp, ast, scope); |

更多内容请参看“编译与执行篇”。

# 七、zval类型通用的宏程序和函数

除了上述各种类型的操作，还有一些通用的宏程序和函数，它们适用于任何类型的zval实例，把这部分内容放在最后是因为它依赖前文提到的一些概念，如计数器，引用类型等。

通用宏程序列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| Z\_OPT\_TYPE() | 读取类型信息，优化版本  (Z\_TYPE\_INFO(zval) & Z\_TYPE\_MASK) | 大量 | 逻辑简单 |
| Z\_OPT\_TYPE\_P() | 通过指针读取类型信息，优化版本  Z\_OPT\_TYPE(\*(zval\_p)) | 大量 | 逻辑简单 |
| zval\_get\_type() | 通过指针读取zval->u1.v.type类型信息  return pz->u1.v.type; | 少量 | 逻辑简单 |
| Z\_TYPE() | 调用zval\_get\_type()函数读取类型信息，这个宏程序是只读访问，因为u1.v.type元素不应该单独修改，应当修改整个v1.type\_info | 大量 | 逻辑简单 |
| Z\_TYPE\_P() | 通过zval实例指针读取类型信息  Z\_TYPE(\*(zval\_p)) | 大量 | 逻辑简单 |
| Z\_TYPE\_FLAGS() | 访问类型标记  (zval).u1.v.type\_flags | 少量 | 逻辑简单 |
| Z\_TYPE\_FLAGS\_P() | 通过指针访问类型标记  Z\_TYPE\_FLAGS(\*(zval\_p)) | 少量 | 逻辑简单 |
| Z\_COLLECTABLE() | 检验zval实例的目标对象是否有【可回收】标记  ((Z\_TYPE\_FLAGS(zval)  & IS\_TYPE\_COLLECTABLE) != 0) | 少量 | 逻辑简单 |
| Z\_COLLECTABLE\_P() | 通过指针检验zval实例的目标对象是否有【可回收】标记  Z\_COLLECTABLE(\*(zval\_p)) | 少量 | 逻辑简单 |
| Z\_TYPE\_INFO() | 访问类型信息  (zval).u1.type\_info | 大量 | 逻辑简单 |
| Z\_TYPE\_INFO\_P() | 通过指针访问类型信息  Z\_TYPE\_INFO(\*(zval\_p)) | 大量 | 逻辑简单 |
| Z\_EXTRA() | 访问zval实例的附加信息  (zval).u2.extra | 少量 | 逻辑简单 |
| Z\_EXTRA\_P() | 通过指针访问zval实例的附加信息  Z\_EXTRA(\*(zval\_p)) | 大量 | 逻辑简单 |
| Z\_PROP\_FLAG\_P() | 调用Z\_EXTRA\_P()宏程序，访问zval实例的附加信息 | 少量 | 逻辑简单 |
| ZVAL\_COPY() | 复制zval实例的指针和类型，并给zval实例的目标元素增加引用次数 | 大量 |  |
| ZVAL\_COPY\_VALUE() | 与ZVAL\_COPY()类似，只是不增加引用次数 | 大量 | 逻辑类似 |
| ZVAL\_COPY\_OR\_DUP() | 复制zval实例的指针和类型元素后，针对可计数对象进行一些后续处理，按情况为目标实例创建副本 | 大量 |  |
| ZVAL\_COPY\_VALUE\_EX() | 更新zval实例的指针和类型两个元素 | 少量 |  |
| ZVAL\_DUP() | 复制zval实例的指针和类型两个元素，并给zval实例的目标元素增加引用次数。如果源zval实例的目标实例是数组，给数组创建副本，并使用新创建的副本 | 大量 |  |
| ZVAL\_COPY\_VALUE\_PROP() | 只有这个宏程序是原样复制整个zval实例  \*(z) = \*(v); | 少量 |  |
| ZVAL\_COPY\_OR\_DUP\_PROP() | 调用ZVAL\_COPY\_OR\_DUP()方法复制zval，再复制zval实例的附加信息 | 少量 |  |
| ZVAL\_COPY\_PROP() | 调用ZVAL\_COPY()方法复制zval，再复制zval实例的附加信息 | 少量 |  |
| SEPARATE\_ZVAL() | 1）针对引用类型：先追踪引用目标，如果目标是数组，创建副本并让当前zval关联到副本，非数组只增加引用次数。2）针对数组类型：调用SEPARATE\_ARRAY()宏程序创建副本。 | 少量 |  |

ZVAL\_COPY()宏程序用于复制zval实例的指针（zval.value.counted）和类型（zval.u1.type\_info）两个元素，并给zval实例的目标元素增加引用次数。代码注释如下：

|  |
| --- |
| #define ZVAL\_COPY(z, v) do { \  zval \*\_z1 = (z); \  const zval \*\_z2 = (v); \  zend\_refcounted \*\_gc = Z\_COUNTED\_P(\_z2); /\* 取出v的指针（zval.value.counted）\*/ \  uint32\_t \_t = Z\_TYPE\_INFO\_P(\_z2); /\* 取出v的类型（zval.u1.type\_info） \*/ \  /\* 更新z的指针（zval.value.counted）和类型（zval.u1.type\_info） \*/ \  ZVAL\_COPY\_VALUE\_EX(\_z1, \_z2, \_gc, \_t); \  if (Z\_TYPE\_INFO\_REFCOUNTED(\_t)) { /\* 如果是可计数类型 \*/ \  GC\_ADDREF(\_gc); /\* 目标实例引用次数+1 \*/ \  } \  } while (0) |

ZVAL\_COPY\_VALUE\_EX()宏程序用于更新zval实例的指针（zval.value.counted）和类型（zval.u1.type\_info）两个元素。代码注释如下：

|  |
| --- |
| # define ZVAL\_COPY\_VALUE\_EX(z, v, gc, t) do { \  Z\_COUNTED\_P(z) = gc; /\* 更新z的指针（zval.value.counted） \*/ \  Z\_TYPE\_INFO\_P(z) = t; /\* 更新z的类型（zval.u1.type\_info） \*/ \  } while (0) |

ZVAL\_DUP宏程序用于复制zval实例的指针（zval.value.counted）和类型（zval.u1.type\_info）两个元素，并给zval实例的目标元素增加引用次数，如果源zval的目标实例是数组，给数组创建副本，并使用新创建的副本。代码注释如下：

|  |
| --- |
| #define ZVAL\_DUP(z, v) do { \  zval \*\_z1 = (z); \  const zval \*\_z2 = (v); \  zend\_refcounted \*\_gc = Z\_COUNTED\_P(\_z2); /\* 取出v的指针（zval.value.counted） \*/ \  uint32\_t \_t = Z\_TYPE\_INFO\_P(\_z2); /\* 取出v的类型（zval.u1.type\_info） \*/ \  if ((\_t & Z\_TYPE\_MASK) == IS\_ARRAY) { /\* 如果源实例类型为数组 \*/ \  /\* 给gc创建副本，让z指向新创建的副本 \*/ \  ZVAL\_ARR(\_z1, zend\_array\_dup((zend\_array\*)\_gc));\  } else { /\* 如果源实例类型不是数组 \*/ \  \  if (Z\_TYPE\_INFO\_REFCOUNTED(\_t)) { /\* 如果是可计数类型 \*/ \  GC\_ADDREF(\_gc); /\* 目标实例引用次数+1 \*/ \  } \  /\* 更新z的指针（zval.value.counted）和类型（zval.u1.type\_info） \*/ \  ZVAL\_COPY\_VALUE\_EX(\_z1, \_z2, \_gc, \_t); \  } \  } while (0) |

ZVAL\_COPY\_OR\_DUP()宏程序的逻辑比较复杂，它在复制指针和类型元素后，针对可计数对象进行一些后续处理：1）如果目标实例不是使用malloc()分配或者实例类型是对象，直接增加引用次数。2）如果是使用malloc()分配并且实例类型不是对象，为实例创建副本，并让zval实例指向到新副本。代码注释如下：

|  |
| --- |
| #define ZVAL\_COPY\_OR\_DUP(z, v) do { \  zval \*\_z1 = (z); \  const zval \*\_z2 = (v); \  zend\_refcounted \*\_gc = Z\_COUNTED\_P(\_z2); /\* 取出v的指针（zval.value.counted） \*/ \  uint32\_t \_t = Z\_TYPE\_INFO\_P(\_z2); /\* 取出v的类型（zval.u1.type\_info） \*/ \  /\* 更新z的指针（zval.value.counted）和类型（zval.u1.type\_info） \*/ \  ZVAL\_COPY\_VALUE\_EX(\_z1, \_z2, \_gc, \_t); \  if (Z\_TYPE\_INFO\_REFCOUNTED(\_t)) { /\* 如果是可计数类型 \*/ \  /\* 如果没有用malloc()分配或 或 者类型是对象 \*/ \  if (EXPECTED(!(GC\_FLAGS(\_gc) & GC\_PERSISTENT)  || GC\_TYPE(\_gc) == IS\_OBJECT)) { \  GC\_ADDREF(\_gc); /\* 增加引用次数 \*/ \  } else { /\* 使用malloc()分配，并且不是对象 \*/ \  zval\_copy\_ctor\_func(\_z1); /\* 创建副本，并把zval关联到副本 \*/ \  } \  } \  } while (0) |

zval\_copy\_ctor\_func()函数在zend\_variables.c文件中定义，用于给字符串和数组类型的目标实例创建副本，并把zval关联到副本，代码注释如下：

|  |
| --- |
| // 给字符串和数组类型创建副本，并把zval关联到副本  ZEND\_API void zval\_copy\_ctor\_func(zval \*zvalue) {  if (EXPECTED(Z\_TYPE\_P(zvalue) == IS\_ARRAY)) { // 如果类型是数组  // 给数组创建副本，并把zval关联到副本  ZVAL\_ARR(zvalue, zend\_array\_dup(Z\_ARRVAL\_P(zvalue)));  } else if (EXPECTED(Z\_TYPE\_P(zvalue) == IS\_STRING)) { // 如果类型是字串  // 给字符串创建副本，并把zval关联到副本  ZVAL\_NEW\_STR(zvalue, zend\_string\_dup(Z\_STR\_P(zvalue), 0));  } else { // 不支持其他类型  ZEND\_UNREACHABLE();  }  } |

ZVAL\_COPY\_VALUE\_PROP()、ZVAL\_COPY\_OR\_DUP\_PROP()、ZVAL\_COPY\_PROP()这3个宏程序的逻辑比较简单，但这三个宏程序复制了包含附加信息在内的整个zval实例：

|  |
| --- |
| #define ZVAL\_COPY\_VALUE\_PROP(z, v) do { \  \*(z) = \*(v); /\* 一次复制整个zval实例 \*/ \  } while (0)  #define ZVAL\_COPY\_PROP(z, v) do { \  ZVAL\_COPY(z, v); \  Z\_PROP\_FLAG\_P(z) = Z\_PROP\_FLAG\_P(v); /\* 复制附加信息(zval).u2.extra \*/ \  } while (0)    #define ZVAL\_COPY\_OR\_DUP\_PROP(z, v) do { \  ZVAL\_COPY\_OR\_DUP(z, v); \  Z\_PROP\_FLAG\_P(z) = Z\_PROP\_FLAG\_P(v); /\* 复制附加信息(zval).u2.extra \*/ \  } while (0) |

SEPARATE\_ZVAL()宏程序的功能是：1）针对引用类型：先追踪引用目标，如果目标是数组，创建副本并让当前zval关联到副本，非数组只增加引用次数。2）针对数组类型：调用SEPARATE\_ARRAY()宏程序创建副本。代码注释如下：

|  |
| --- |
| #define SEPARATE\_ZVAL(zv) do { \  zval \*\_zv = (zv); \  if (Z\_ISREF\_P(\_zv)) { /\* 如果是引用类型 \*/ \  zend\_reference \*\_r = Z\_REF\_P(\_zv); /\* zend\_reference实例 \*/ \  ZVAL\_COPY\_VALUE(\_zv, &\_r->val); /\* 解除引用 \*/ \  if (GC\_DELREF(\_r) == 0) { /\* 如果zend\_reference实例只被引用1次 \*/ \  efree\_size(\_r, sizeof(zend\_reference)); /\* 释放引用实例 \*/ \  /\* 否则（还有其他地方引用，不可释放）：如果类型是数组 \*/ \  } else if (Z\_OPT\_TYPE\_P(\_zv) == IS\_ARRAY) { \  ZVAL\_ARR(\_zv, zend\_array\_dup(Z\_ARR\_P(\_zv))); /\* 数组创建新副本，zval指向它 \*/ \  break; /\* 完毕 \*/ \  /\* 否则（不可释放且不是数组）：如果 zv 可计数 \*/ \  } else if (Z\_OPT\_REFCOUNTED\_P(\_zv)) { \  Z\_ADDREF\_P(\_zv); /\* 增加引用次数，不是数组不用真的separate \*/ \  break; /\* 完毕 \*/ \  } \  } \  if (Z\_TYPE\_P(\_zv) == IS\_ARRAY) { /\* 数组这里单独有一个处理 \*/ \  SEPARATE\_ARRAY(\_zv); /\* 数组创建副本 \*/ \  } \  } while (0) |

zval\_add\_ref()函数比较特殊，官方给出的说明是：它是一个复制实例后调用的构造方法，只能在ZVAL\_COPY\_VALUE()后面调用，不能在复制前调用，那样会造成引用泄漏。函数定义如下：

|  |
| --- |
| ZEND\_API void zval\_add\_ref(zval \*p) {  if (Z\_REFCOUNTED\_P(p)) { // 如果是可计数类型  // 如果是引用类型 并且 引用数次是1  if (Z\_ISREF\_P(p) && Z\_REFCOUNT\_P(p) == 1) {  ZVAL\_COPY(p, Z\_REFVAL\_P(p)); // 把引用目标复制到 p  } else { // 否则  Z\_ADDREF\_P(p); // 只增加引用次数  }  }  } |

# 八、销毁实例

在zend\_variables.c中为各种数据类型定义了销毁器：

|  |
| --- |
| static const zend\_rc\_dtor\_func\_t zend\_rc\_dtor\_func[] = {  /\* IS\_UNDEF \*/ (zend\_rc\_dtor\_func\_t)zend\_empty\_destroy,  /\* IS\_NULL \*/ (zend\_rc\_dtor\_func\_t)zend\_empty\_destroy,  /\* IS\_FALSE \*/ (zend\_rc\_dtor\_func\_t)zend\_empty\_destroy,  /\* IS\_TRUE \*/ (zend\_rc\_dtor\_func\_t)zend\_empty\_destroy,  /\* IS\_LONG \*/ (zend\_rc\_dtor\_func\_t)zend\_empty\_destroy,  /\* IS\_DOUBLE \*/ (zend\_rc\_dtor\_func\_t)zend\_empty\_destroy,  // 以上类型是不需要用销毁器销毁的， zend\_empty\_destroy 是空函数，无业务逻辑    /\* IS\_STRING \*/ (zend\_rc\_dtor\_func\_t)zend\_string\_destroy, // 字串  /\* IS\_ARRAY \*/ (zend\_rc\_dtor\_func\_t)zend\_array\_destroy, // 数组  /\* IS\_OBJECT \*/ (zend\_rc\_dtor\_func\_t)zend\_objects\_store\_del, // 对象  /\* IS\_RESOURCE \*/ (zend\_rc\_dtor\_func\_t)zend\_list\_free, // 资源  /\* IS\_REFERENCE \*/ (zend\_rc\_dtor\_func\_t)zend\_reference\_destroy, // 引用  /\* IS\_CONSTANT\_AST \*/ (zend\_rc\_dtor\_func\_t)zend\_ast\_ref\_destroy // 常量表达式  }; |

如上所示，销毁器的函数原型是 zend\_rc\_dtor\_func\_t：

|  |
| --- |
| typedef void (\*zend\_rc\_dtor\_func\_t)(zend\_refcounted \*p); |

12种类型的前6种（IS\_UNDEF、IS\_NULL、IS\_FALSE、IS\_TRUE、IS\_LONG、IS\_DOUBLE）使用的销毁器都是zend\_empty\_destroy()函数，它是一个空函数：

|  |
| --- |
| static void zend\_empty\_destroy(zend\_reference \*ref){} |

这是因为前6种类型并没有单独的实例，它们只是zval实例的不同状态，而销毁器是用来销毁单独的实例的。后6种类型（IS\_STRING、IS\_ARRAY、IS\_OBJECT、IS\_RESOURCE、IS\_REFERENCE、IS\_CONSTANT\_AST）都有单独的数据结构，也有各自的销毁器。

**1）字符串类型**

字符串（IS\_STRING）类型的销毁器是zend\_string\_destroy()函数，它是\_efree()函数的别名：

|  |
| --- |
| #define zend\_string\_destroy \_efree |

字符串类型的另和个常用的销毁器是zval\_ptr\_dtor\_str()函数，它在销毁实例前增加了引用次数判断，代码如下：

|  |
| --- |
| static void zval\_ptr\_dtor\_str(zval \*zval\_ptr) {  // 如果是可计数类型， 并且引用次数-1后为0  if (Z\_REFCOUNTED\_P(zval\_ptr) && !Z\_DELREF\_P(zval\_ptr)) {  efree(Z\_STR\_P(zval\_ptr)); // 释放zval指向的实例  }  } |

\_efree()函数和efree()宏程序的更多介绍参见“内存管理篇”。

zval\_internal\_ptr\_dtor()函数用于销毁使用malloc()函数分配内存创建的字符串实例，这些实例都带有IS\_STR\_PERSISTENT标记。函数代码如下：

|  |
| --- |
| ZEND\_API void zval\_internal\_ptr\_dtor(zval \*zval\_ptr) {  if (Z\_REFCOUNTED\_P(zval\_ptr)) { // 如果是可计数变量  zend\_refcounted \*ref = Z\_COUNTED\_P(zval\_ptr); // 实例  if (GC\_DELREF(ref) == 0) { // 如果-1后引用数是 0  if (Z\_TYPE\_P(zval\_ptr) == IS\_STRING) { // 如果是字串  zend\_string \*str = (zend\_string\*)ref; // 转成字串指针  ZEND\_ASSERT(!ZSTR\_IS\_INTERNED(str)); // 不可以是保留字  ZEND\_ASSERT((GC\_FLAGS(str) & IS\_STR\_PERSISTENT)); // 用malloc()函数分配  free(str); // 释放字串  } else { // 其他情况 抛错：内部zval不可以是 数组，对象、资源和引用类型  zend\_error\_noreturn(E\_CORE\_ERROR, "Internal zval's can't be arrays, objects, resources or reference");  }  }  }  }  #define ZVAL\_INTERNAL\_PTR\_DTOR zval\_internal\_ptr\_dtor // 别名 |

ZEND\_ASSERT()宏程序在调试时生效，非调试环境中没有作用。

zval\_internal\_ptr\_dtor()函数有一个别名ZVAL\_INTERNAL\_PTR\_DTOR。

**2）数组类型**

数组（IS\_ARRAY）类型的销毁器是zend\_array\_destroy()函数，更多介绍参见“数组篇”。

**3）对象类型**

对象（IS\_OBJECT）类型的销毁器是zend\_objects\_store\_del()函数，更多介绍参见“面向对象篇”。

**4）资源类型**

资源（IS\_RESOURCE）类型的销毁器是zend\_list\_free()函数，在zend\_list.c文件中定义。

**5）引用类型**

引用（IS\_REFERENCE）类型的销毁器是zend\_reference\_destroy()函数：

|  |
| --- |
| static void zend\_reference\_destroy(zend\_reference \*ref) {  i\_zval\_ptr\_dtor(&ref->val); // 销毁引用目标  efree\_size(ref, sizeof(zend\_reference)); // 释放zend\_reference实例  } |

i\_zval\_ptr\_dtor()函数在下文中介绍。

**6）常量表达式类型**

资源（IS\_CONSTANT\_AST）类型的销毁器是zend\_ast\_ref\_destroy()函数，在zend\_ast.c文件中定义。

## 通用销毁器

有了每个类型的销毁器，就可以定义一个通用的销毁器，调用它可以销毁所有类型的变量，这个通用的销毁器是rc\_dtor\_func()函数：

|  |
| --- |
| ZEND\_API void rc\_dtor\_func(zend\_refcounted \*p) {  // 通过GC\_TYPE()宏程序获取将要销毁的实例的类型值，把它为索引号，获取并调用销毁器  zend\_rc\_dtor\_func[GC\_TYPE(p)](p);  } |

如上所示，在zend\_rc\_dtor\_func销毁器列表的定义中，销毁器的顺序号与它对应的类型值相同，所以只要使用“将要销毁的实例的类型”作为索引号，就可以找到每个实例的销毁器。

由于销毁实例时常常需要考虑实例的被引用次数，所以实际调用更多的是rc\_dtor\_func()函数的一些衍生函数，以下几个函数在代码中都被大量调用。

i\_zval\_ptr\_dtor()函数用于给实例减少引用次数，引用次数为0时销毁实例，不为0时加入自动回收列表：

|  |
| --- |
| static void i\_zval\_ptr\_dtor(zval \*zval\_ptr) {  if (Z\_REFCOUNTED\_P(zval\_ptr)) { // 如果是可计数类型  zend\_refcounted \*ref = Z\_COUNTED\_P(zval\_ptr); // 获取zval指向的实例  if (!GC\_DELREF(ref)) { // 引用次数-1 后如果为0  rc\_dtor\_func(ref); // 销毁实例  } else { // 不为0，还有引用  gc\_check\_possible\_root(ref); // 添加到自动回收  }  }  } |

gc\_check\_possible\_root()函数的更多介绍参见“垃圾回收篇”。

i\_zval\_ptr\_dtor()函数有两个别名：zval\_ptr\_dtor和ZVAL\_PTR\_DTOR：

|  |
| --- |
| ZEND\_API void zval\_ptr\_dtor(zval \*zval\_ptr) {  i\_zval\_ptr\_dtor(zval\_ptr);  }  #define ZVAL\_PTR\_DTOR zval\_ptr\_dtor |

zval\_ptr\_dtor\_nogc()函数用于给实例减少引用次数，引用次数为0时销毁实例：

|  |
| --- |
| // 函数名中的nogc，表示不把实例放入垃圾回收列表  static void zval\_ptr\_dtor\_nogc(zval \*zval\_ptr) {  // 如果是可计数变量，并且 -1 后引用次数为0  if (Z\_REFCOUNTED\_P(zval\_ptr) && !Z\_DELREF\_P(zval\_ptr)) {  rc\_dtor\_func(Z\_COUNTED\_P(zval\_ptr)); // 销毁zval指向的实例  }  } |

zval\_ptr\_dtor\_nogc()函数有一个用于保持兼容性的别名zval\_dtor，在8.2.5版本源码中没有用到。

# 结语

基础类型种类比较多，相关的算法也比较烦琐，但在设计上并不算复杂。

在zval类型的相关算法中，并没有用于创建zval的函数和宏程序，这是因为在实际使用中，zval实例通常是被批量创建的，例如在哈希表的相关操作中，或者在创建执行数（execute\_data）据时都会批量创建zval实例，具体可参见相关章节。

数组（zend\_array）和对象（zend\_object）类型的介绍参见独立章节“数组篇”与“面向对象”篇。

# 附录

## 一）声名但是没有用到的函数和宏程序

|  |  |  |
| --- | --- | --- |
| **函数名/宏程序名** | **说明** | **备注** |
| I\_REPLACE\_SONAME\_FNNAME\_ZU() | 生成 \_vgr00000ZU\_开头的函数名 | 逻辑简单 |
| IS\_INTERNED() | 调用ZSTR\_IS\_INTERNED()宏程序 | 逻辑简单 |
| STR\_ALLOCA\_ALLOC() | 调用ZSTR\_ALLOCA\_ALLOC()宏程序 | 逻辑简单 |
| STR\_ALLOCA\_FREE() | 调用ZSTR\_ALLOCA\_FREE()宏程序 | 逻辑简单 |
| STR\_ALLOCA\_INIT() | 调用ZSTR\_ALLOCA\_INIT()宏程序 | 逻辑简单 |
| zend\_string\_refcount() | 获取 zend\_string实例的引用次数 | 逻辑简单 |
| ZSTR\_ALLOCA\_INIT() | 创建指定长度（自动对齐）的字符串，并复制给出的字符串 | 逻辑简单 |
| ZSTR\_HASH() | 调用zend\_string\_hash\_val()函数 | 逻辑简单 |

zval\_copy\_ctor()函数，对数组类型创建副本，对其他类型增加引用次数：

|  |
| --- |
| static zend\_always\_inline void zval\_copy\_ctor(zval \*zvalue) {  if (Z\_TYPE\_P(zvalue) == IS\_ARRAY) { // 如果是数组  ZVAL\_ARR(zvalue, zend\_array\_dup(Z\_ARR\_P(zvalue)));  } else if (Z\_REFCOUNTED\_P(zvalue)) { // 如果不是数组，并且可计数  Z\_ADDREF\_P(zvalue); // 添加引用次数  }  } |

zval\_opt\_copy\_ctor()函数是zval\_copy\_ctor()函数的优化版本：

|  |
| --- |
| static zend\_always\_inline void zval\_opt\_copy\_ctor(zval \*zvalue) {  if (Z\_OPT\_TYPE\_P(zvalue) == IS\_ARRAY) { // 如果是数组  ZVAL\_ARR(zvalue, zend\_array\_dup(Z\_ARR\_P(zvalue)));  } else if (Z\_OPT\_REFCOUNTED\_P(zvalue)) { // 如果不是数组，并且可计数  Z\_ADDREF\_P(zvalue); // 添加引用次数  }  } |

## 二）zval结构体的u2元素

在“基本概念”章节中介绍过，zval结构体的u2元素也是联合类型，用于存放一个32位整数。与其相关的宏程序有如下几个：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名/宏程序名** | **说明** | **引用**  **次数** | **备注** |
| Z\_CACHE\_SLOT(zval) | (zval).u2.cache\_slot | 大量 | 逻辑简单 |
| Z\_CACHE\_SLOT\_P(zval\_p) | Z\_CACHE\_SLOT(\*(zval\_p)) | 少量 | 逻辑简单 |
| Z\_CONSTANT\_FLAGS(zval) | (zval).u2.constant\_flags | 少量 | 逻辑简单 |
| Z\_CONSTANT\_FLAGS\_P(zval\_p) | Z\_CONSTANT\_FLAGS(\*(zval\_p)) | 少量 | 逻辑简单 |
| Z\_LINENO(zval) | (zval).u2.lineno | 少量 | 逻辑简单 |
| Z\_LINENO\_P(zval\_p) | Z\_LINENO(\*(zval\_p)) | 少量 | 逻辑简单 |
| Z\_OPLINE\_NUM(zval) | (zval).u2.opline\_num | 少量 | 逻辑简单 |
| Z\_OPLINE\_NUM\_P(zval\_p) | Z\_OPLINE\_NUM(\*(zval\_p)) | 少量 | 逻辑简单 |

以上宏程序主要在编译和运行过程中用到，与基本类型篇关联不大，更多内容请参看“编译与执行篇”。

## 三）用于转换的类型

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| \_IS\_BOOL | 18 | 布尔型 |
| \_IS\_NUMBER | 19 | 数字 |

## 四）其他字符串标记

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| IS\_STR\_CLASS\_NAME\_MAP\_PTR | GC\_PROTECTED | 面向对象中使用 |
| IS\_STR\_VALID\_UTF8 | 512 | utf8有效 |

# 未解疑问

alloca()是什么

为什么sizeof(zend\_string)长度是32不是25？

切换临时保留字和永久保留字的意义何在？看着没差嘛。

Z\_OPT\_TYPE()如何兼容高位和低位？

zval\_add\_ref()更详细的应用，在相关章节中补充。

# Change Log

2024.12.6 创建

2024.12.7 zval类型，转战虚拟机

2024.12.14 下午，zval类型

2024.12.15 zend\_string

2024.12.18 保留字

2024.12.19 smart\_str开头

2024.12.20 smart\_str结束, smart\_string

2024.12.21 保留字完成

2024.12.24 简易字符串基本完成，zend\_string与zval相关宏程序完成，zend\_string基本完成

2024.12.25 资源类型完成，开始引用和计数器

2024.12.26 整理计数器

2024.12.27 计数器相关内容完毕 ，引用相关内容完毕

2024.12.28 通用宏程序完毕，内部类型over，基础类型修改过，IS\_CONSTANT\_AST\_EX碰到问题，如何调试？static $a=new stdclass();

2024.12.29 常量表达式简单介绍完成，主体内容基本完成

2025.1.3 引用和间接引用的比较未完成

2025.1.4 整理未编目部分，引用和间接引用的比较未完成

2025.1.5 基本完成，暂时不花时间打磨细节，因为不如把时间留出来再写一些其他篇章。这一篇已经花了很多心血了，内容和格式，在编写过程中也都调整过，还都是过得去的。给自己点个赞！

2025.2.6 增加销毁器章节。 todo:还可以增加资源列表zend\_list.h部分内容。