# 二、对象

## 一）对象（Object）

### 对象（Object）的数据结构

对象的数据结构比较简单，代码如下：

|  |
| --- |
| typedef struct \_zend\_object zend\_object; // zend\_object是\_zend\_object的别名  struct \_zend\_object {  zend\_refcounted\_h gc; // 引用计数  // handle 是对象在全局对象容器 EG(objects\_store).object\_buckets 中的【索引号】，通过它可以直接在全局容器中获取指针  uint32\_t handle;  zend\_class\_entry \*ce; // 所属类  const zend\_object\_handlers \*handlers; // 对象操作方法（函数）  HashTable \*properties; // 对象的【附加属性】表  zval properties\_table[1]; // 默认属性表  }; |

如上所示，对象数据结构zend\_object是\_zend\_object结构体的别名，它有6个元素：

gc：引用计数器（详情参见“类型篇”）；

handle：是对象在全局对象容器 EG(objects\_store).object\_buckets 中的索引号；

ce：对象所属的类；

handlers：对象的操作方法，这是一个函数集合；

properties：对象的动态属性表，也就是没有在类中声名的属性；

properties\_table：对象的属性表，在类中声名过的属性。

对于以上每个属性的用法，后续逐步展开详细介绍。

类的继承通过zend\_do\_inheritance()函数来实现，代码如下：

1

## 一）对象的基本操作

### 创建对象

zend\_objects\_new()函数用于创建对象，代码如下：

|  |
| --- |
| ZEND\_API zend\_object\* zend\_objects\_new(zend\_class\_entry \*ce) {  // 对象大小是 对象本身 + 属性变量大小（个数\*zval大小）  zend\_object \*object = emalloc(sizeof(zend\_object) + zend\_object\_properties\_size(ce));  \_zend\_object\_std\_init(object, ce); // 初始化标准对象  object->handlers = &std\_object\_handlers; // 绑定标准处理器  return object; // 返回对象  } |

zend\_object\_properties\_size()函数用于计算对象的成员变量表大小，代码如下：

|  |
| --- |
| static size\_t zend\_object\_properties\_size(zend\_class\_entry \*ce) {  // ZEND\_ACC\_USE\_GUARDS 标记说明类有 \_\_get/\_\_set/\_\_unset/\_\_isset 魔术方法  // 占用内存 = zval大小\*默认成员变量数量（没有魔术方法时，数量 -1）  return sizeof(zval) \*  (ce->default\_properties\_count -  ((ce->ce\_flags & ZEND\_ACC\_USE\_GUARDS) ? 0 : 1));  } |

\_zend\_object\_std\_init()函数用于初始化对象：

|  |
| --- |
| static void \_zend\_object\_std\_init(zend\_object \*object, zend\_class\_entry \*ce) {  GC\_SET\_REFCOUNT(object, 1); // 引用次数为1  GC\_TYPE\_INFO(object) = GC\_OBJECT; // object.u.type\_info 为 GC\_OBJECT  object->ce = ce; // 类入口  object->properties = NULL; // 先不创建动态属性表  zend\_objects\_store\_put(object); // 对象添加到全局容器  // 如果有用到魔术方法 \_\_get/\_\_set/\_\_unset/\_\_isset  if (UNEXPECTED(ce->ce\_flags & ZEND\_ACC\_USE\_GUARDS)) {  // 结尾元素设置成 IS\_UNDEF  ZVAL\_UNDEF(object->properties\_table + object->ce->default\_properties\_count);  }  } |

std\_object\_handlers是对象的标准操作方法集合，在后续章节中介绍。

### 初始化对象

zend\_object\_std\_init()函数用于初始化对象，代码如下：

|  |
| --- |
| ZEND\_API void zend\_object\_std\_init(zend\_object \*object, zend\_class\_entry \*ce) {  \_zend\_object\_std\_init(object, ce);  } |

如上所示，它调用前方中介绍过的\_zend\_object\_std\_init()函数进行操作。

### 克隆对象

zend\_objects\_clone\_obj()函数用于克隆对象，代码如下：

|  |
| --- |
| ZEND\_API zend\_object \*zend\_objects\_clone\_obj(zend\_object \*old\_object) {  zend\_object \*new\_object;  new\_object = zend\_objects\_new(old\_object->ce); // 使用原对象的所属类，创建新对象  if (new\_object->ce->default\_properties\_count) { // 如果新对象有默认成员变量  zval \*p = new\_object->properties\_table; // 成员变量表开头  zval \*end = p + new\_object->ce->default\_properties\_count; // 成员变量表结尾  do { // 遍历所有成员变量  ZVAL\_UNDEF(p); // 每一个成员变量设置成无效（IS\_UNDEF）  p++; // 下一个属性  } while (p != end); // 直到结尾  }  zend\_objects\_clone\_members(new\_object, old\_object); // 克隆对象成员变量  return new\_object; // 返回新克隆的对象  } |

zend\_objects\_clone\_members()函数用于克隆对象的成员变量表，复杂。

|  |
| --- |
| ZEND\_API void ZEND\_FASTCALL zend\_objects\_clone\_members(zend\_object \*new\_object, zend\_object \*old\_object) {  if (old\_object->ce->default\_properties\_count) { // 如果类有默认成员变量  zval \*src = old\_object->properties\_table; // 源列表  zval \*dst = new\_object->properties\_table; // 目标列表  zval \*end = src + old\_object->ce->default\_properties\_count; // 源列表结尾位置  do { // 遍历所源列表  i\_zval\_ptr\_dtor(dst); // 销毁目标变量，如果有引用次数，添加到回收列表中  ZVAL\_COPY\_VALUE\_PROP(dst, src); // 直接原样复制zval  zval\_add\_ref(dst); // 目标变量增加引用次数  // 如果目标变量是引用类型  if (UNEXPECTED(Z\_ISREF\_P(dst)) &&  // 如果是调试模式 或 引用目标不是NULL  (ZEND\_DEBUG || ZEND\_REF\_HAS\_TYPE\_SOURCES(Z\_REF\_P(dst)))) {  // 返回此成员变量的信息  zend\_property\_info \*prop\_info = zend\_get\_property\_info\_for\_slot(new\_object, dst);  if (ZEND\_TYPE\_IS\_SET(prop\_info->type)) { // 如果类型有效  // 向zend\_property\_info\_source\_list指向的 zend\_property\_info 列表中添加 zend\_property\_info  // 添加类型信息  ZEND\_REF\_ADD\_TYPE\_SOURCE(Z\_REF\_P(dst), prop\_info);  }  }  src++; // 下一个源元素  dst++; // 下一个目标元素  } while (src != end); // 直到源结尾  // 如果旧成员有属性列表 并且 旧成员没有 clone方法  } else if (old\_object->properties && !old\_object->ce->clone) {  // 如果处理器列表 就是 标记处理器列表  if (EXPECTED(old\_object->handlers == &std\_object\_handlers)) {  // 如果 properties 不是不能变的数组  if (EXPECTED(!(GC\_FLAGS(old\_object->properties) & IS\_ARRAY\_IMMUTABLE))) {  GC\_ADDREF(old\_object->properties); // 引用次数 +1  }  new\_object->properties = old\_object->properties; // 使用新的属性列表  return;  }  }  // 如果有属性列表 并且 列表中有元素  if (old\_object->properties &&  EXPECTED(zend\_hash\_num\_elements(old\_object->properties))) {  zval \*prop, new\_prop;  zend\_ulong num\_key;  zend\_string \*key;  if (!new\_object->properties) { // 如果新对象没有属性列表  // 创建一个元素数量相同的新数组  new\_object->properties = zend\_new\_array(zend\_hash\_num\_elements(old\_object->properties));  zend\_hash\_real\_init\_mixed(new\_object->properties); // 初始化成哈希表  } else { // 如果已经有了属性列表  // 扩展哈希表，第三个参数：类型不是顺序数组  zend\_hash\_extend(new\_object->properties, new\_object->properties->nNumUsed + zend\_hash\_num\_elements(old\_object->properties), 0);  }  // 属性哈希表添加标记：有空索引号  HT\_FLAGS(new\_object->properties) |=  HT\_FLAGS(old\_object->properties) & HASH\_FLAG\_HAS\_EMPTY\_IND;  // 遍历 properties => prop  ZEND\_HASH\_MAP\_FOREACH\_KEY\_VAL(old\_object->properties, num\_key, key, prop) {  if (Z\_TYPE\_P(prop) == IS\_INDIRECT) { // 如果 prop 是间接引用  // new\_prop.zv 添加指针, 并标记成 间接引用 类型  // ？？？  ZVAL\_INDIRECT(&new\_prop, new\_object->properties\_table + (Z\_INDIRECT\_P(prop) - old\_object->properties\_table));  } else { // 其他情况  ZVAL\_COPY\_VALUE(&new\_prop, prop); // 直接复制值  zval\_add\_ref(&new\_prop); // 新 prop 增加引用次数  }  if (EXPECTED(key)) { // 如果有 key  // 新属性添加进哈希表里  \_zend\_hash\_append(new\_object->properties, key, &new\_prop);  } else { // 没有key  // 按顺序号添加进哈希表里 num\_key？  zend\_hash\_index\_add\_new(new\_object->properties, num\_key, &new\_prop);  }  } ZEND\_HASH\_FOREACH\_END();  }  if (old\_object->ce->clone) { // 如果 旧成员有克隆方法  GC\_ADDREF(new\_object); // 新对象引用数量 +1  // 调用新成员的clone方法（所以 \_\_clone 方法是克隆后的附加处理，没有它对象也能正常克隆，测试过）  zend\_call\_known\_instance\_method\_with\_0\_params(new\_object->ce->clone, new\_object, NULL);  // 释放 new\_object  OBJ\_RELEASE(new\_object);  }  } |

### 销毁对象

zend\_object\_std\_dtor()函数用于销毁对象，代码如下：

|  |
| --- |
| ZEND\_API void zend\_object\_std\_dtor(zend\_object \*object) {  zval \*p, \*end;  if (object->properties) { // 如果对象有动态属性列表  // 如果动态属性列表不是不可更改  if (EXPECTED(!(GC\_FLAGS(object->properties) & IS\_ARRAY\_IMMUTABLE))) {  // 如果引用数 -1 后为0  if (EXPECTED(GC\_DELREF(object->properties) == 0)  // 并且属性列表不是 NULL  && EXPECTED(GC\_TYPE(object->properties) != IS\_NULL)) {  zend\_array\_destroy(object->properties); // 销毁属性列表  }  }  }  p = object->properties\_table; // 基本属性列表指针  // 如果类中有定义基本属性  if (EXPECTED(object->ce->default\_properties\_count)) {  end = p + object->ce->default\_properties\_count; // 尾元素指针  do { // 遍历每一个  if (Z\_REFCOUNTED\_P(p)) { // 如果是可计数对象  // 如果p是引用类型  if (UNEXPECTED(Z\_ISREF\_P(p)) &&  // 并且 是调试模式 或 有引用源  (ZEND\_DEBUG || ZEND\_REF\_HAS\_TYPE\_SOURCES(Z\_REF\_P(p)))) {  // 通过slot（p）指针计算slot的序号，返回此序号对应的属性信息  zend\_property\_info \*prop\_info = zend\_get\_property\_info\_for\_slot(object, p);  if (ZEND\_TYPE\_IS\_SET(prop\_info->type)) { // 如果类型有效  // ??  ZEND\_REF\_DEL\_TYPE\_SOURCE(Z\_REF\_P(p), prop\_info);  }  }  i\_zval\_ptr\_dtor(p); // 销毁p或添加到gc列表  }  p++; // 下一个类属性  } while (p != end);  }  // 如果有魔术方法 \_\_get/\_\_set/\_\_unset/\_\_isset  if (UNEXPECTED(object->ce->ce\_flags & ZEND\_ACC\_USE\_GUARDS)) {  if (EXPECTED(Z\_TYPE\_P(p) == IS\_STRING)) { // 如果属性类型是 字串  zval\_ptr\_dtor\_str(p); // 删除字串  } else if (Z\_TYPE\_P(p) == IS\_ARRAY) { // 如果属性类型是 数组  HashTable \*guards;  guards = Z\_ARRVAL\_P(p); // 从zval中取出数组指针  zend\_hash\_destroy(guards); // 销毁数组  FREE\_HASHTABLE(guards); // 释放数组内存  }  }  // 如果对象是弱引用  if (UNEXPECTED(GC\_FLAGS(object) & IS\_OBJ\_WEAKLY\_REFERENCED)) {  zend\_weakrefs\_notify(object); // 发送弱引用通知  }  } |

# 三、对象的标准处理器

对象的标准处理器是组函数，用于处理对象的标准操作，主要由虚拟机调用，函数列表如下：

|  |
| --- |
| ZEND\_API const zend\_object\_handlers std\_object\_handlers = {  0, /\* 偏移量 \*/  zend\_object\_std\_dtor, /\* 释放对象 \*/  zend\_objects\_destroy\_object, /\* 销毁对象 \*/  zend\_objects\_clone\_obj, /\* 克隆对象 \*/  zend\_std\_read\_property, /\* 读取成员变量 \*/  zend\_std\_write\_property, /\* 更新成员变量 \*/  zend\_std\_read\_dimension, /\* 维度读取 \*/  zend\_std\_write\_dimension, /\* 维度更新 \*/  zend\_std\_get\_property\_ptr\_ptr, /\* get\_property\_ptr\_ptr \*/  zend\_std\_has\_property, /\* 检验成员变量是否存在 \*/  zend\_std\_unset\_property, /\* 删除成员变量 \*/  zend\_std\_has\_dimension, /\* 用维度检查数据是否存在 \*/  zend\_std\_unset\_dimension, /\* 用维度删除数据 \*/  zend\_std\_get\_properties, /\* 获取成员变量 \*/  zend\_std\_get\_method, /\* 获取成员方法 \*/  zend\_std\_get\_constructor, /\* 获取构造方法 \*/  zend\_std\_get\_class\_name, /\* 获取类名 \*/  zend\_std\_cast\_object\_tostring, /\* 对象转成字符串 \*/  NULL, /\* 计数，没有标准方法 \*/  zend\_std\_get\_debug\_info, /\* 获取调试信息 \*/  zend\_std\_get\_closure, /\* 获取闭包 \*/  zend\_std\_get\_gc, /\* get\_gc 获取用于垃圾回收的属性表 \*/  NULL, /\* do\_operation方法，没有标准方法 \*/  zend\_std\_compare\_objects, /\* 比较 \*/  NULL, /\* get\_properties\_for，没有标准方法 \*/  }; |

下面依次展开介绍。

## 一）操作对象的属性（成员变量）

操作对象的属性（property，也称为“成员变量”），是对象的标准操作中最复杂的部分。

### 获取对象的属性表

zend\_std\_get\_properties()函数用于获取对象的属性表，代码如下：

|  |
| --- |
| ZEND\_API HashTable \*zend\_std\_get\_properties(zend\_object \*zobj) {  if (!zobj->properties) { // 没有属性表时创建  rebuild\_object\_properties(zobj); // 创建属性哈希表，并添加默认属性 （如果没创建过）  }  return zobj->properties;  } |

rebuild\_object\_properties()函数用于创建属性表，代码如下：

|  |
| --- |
| ZEND\_API void rebuild\_object\_properties(zend\_object \*zobj) {  if (!zobj->properties) { // 没有对象属性表时才创建  zend\_property\_info \*prop\_info; // 属性信息，临时变量  zend\_class\_entry \*ce = zobj->ce; // 类  int i;  zobj->properties = zend\_new\_array(ce->default\_properties\_count); // 创建顺序数组  if (ce->default\_properties\_count) { // 如果有默认的属性  zend\_hash\_real\_init\_mixed(zobj->properties); // 属性表初始化成哈希表  for (i = 0; i < ce->default\_properties\_count; i++) { // 遍历默认属性  prop\_info = ce->properties\_info\_table[i]; // 取得属性信息  if (!prop\_info) { // 如果属性信息为空，跳过  continue;  }  // 如果这个属性信息的类型是 未定义  if (UNEXPECTED(Z\_TYPE\_P(OBJ\_PROP(zobj, prop\_info->offset)) == IS\_UNDEF)) {  // 属性哈希表添加标记：存在空间接引用元素  HT\_FLAGS(zobj->properties) |= HASH\_FLAG\_HAS\_EMPTY\_IND;  }  // 属性哈希表中参加间接引用元素，键名是属性名，值为属性值zval  \_zend\_hash\_append\_ind(zobj->properties, prop\_info->name,  OBJ\_PROP(zobj, prop\_info->offset));  }  }  }  } |

### 获取对象属性的序号

zend\_get\_property\_offset()函数用于通过属性名，取得属性在属性列表中的序号，这个方法在操作对象属性时经常会用到，是非常重要的辅助函数。此函数的业务逻辑比较复杂，要处理多种情况。

#### **1）在缓存中查找**

代码如下：

|  |
| --- |
| // 参数：p1:对象所属类，p2:属性名，p3:是否静默，p4:缓存位置指针，p5:返回的属性信息指针  static uintptr\_t zend\_get\_property\_offset(zend\_class\_entry \*ce, zend\_string \*member,  int silent, void \*\*cache\_slot, zend\_property\_info \*\*info\_ptr) {  zval \*zv;  zend\_property\_info \*property\_info;  uint32\_t flags;  zend\_class\_entry \*scope;  uintptr\_t offset;    // 步骤1，从缓存里快速获取属性信息 。 #define CACHED\_PTR\_EX(slot) (slot)[0]  // 如果缓存位置存的是当前 类指针，说明 cache\_slot 直接可用 （参见 CACHE\_POLYMORPHIC\_PTR\_EX ）  if (cache\_slot && EXPECTED(ce == CACHED\_PTR\_EX(cache\_slot))) { // 如果类匹配  \*info\_ptr = CACHED\_PTR\_EX(cache\_slot + 2); // 向后移两个指针，得到属性信息  // 后移1个，得到偏移量 （参见 CACHE\_POLYMORPHIC\_PTR\_EX ）  return (uintptr\_t)CACHED\_PTR\_EX(cache\_slot + 1);  } |

#### **2）查询属性表**

代码如下：

|  |
| --- |
| **// zend\_get\_property\_offset()函数第2块代码。 步骤2，查询属性信息表**  // 如果所属类属性信息表是空的 或者 属性信息表里找不到这个成员  if (UNEXPECTED(zend\_hash\_num\_elements(&ce->properties\_info) == 0)  || UNEXPECTED((zv = zend\_hash\_find(&ce->properties\_info, member)) == NULL)) {  // 如果成员名字异常  if (UNEXPECTED(ZSTR\_VAL(member)[0] == '\0') && ZSTR\_LEN(member) != 0) {  if (!silent) { // 如果没有要求静默  zend\_bad\_property\_name(); // 报错  }  return ZEND\_WRONG\_PROPERTY\_OFFSET; // 要求静默，返回：属性偏移位置错误  }  dynamic: // 按动态属性处理  if (cache\_slot) { // 如果有缓存，更新缓存  // 第一个位置是类指针，第二位置个写入-1（【动态属性序号】标记）  CACHE\_POLYMORPHIC\_PTR\_EX(cache\_slot, ce,  (void\*)ZEND\_DYNAMIC\_PROPERTY\_OFFSET);  CACHE\_PTR\_EX(cache\_slot + 2, NULL); // 第三个位置写入null  }  return ZEND\_DYNAMIC\_PROPERTY\_OFFSET; // 返回：-1（【动态属性序号】标记）  } |

#### **3）？查询父类**

代码如下：

|  |
| --- |
| **// zend\_get\_property\_offset()函数第3块代码。 步骤3，已经获取到属性信息，要验证权限**  property\_info = (zend\_property\_info\*)Z\_PTR\_P(zv); // 取得属性信息指针  flags = property\_info->flags; // 属性信息附加标记  // 如果属性信息有【来自继承】或【私有】或【受保护】标记  if (flags & (ZEND\_ACC\_CHANGED|ZEND\_ACC\_PRIVATE|ZEND\_ACC\_PROTECTED)) {  if (UNEXPECTED(EG(fake\_scope))) { // 如果当前有临时作用域  scope = EG(fake\_scope); // 优先使用临时作用域  } else { // 没有临时作用域  scope = zend\_get\_executed\_scope(); // 从执行数据串中，获取执行时作用域  }  if (property\_info->ce != scope) { // 关键条件：如果属性所属类 不是 当前域  if (flags & ZEND\_ACC\_CHANGED) { // 情况3.1，如果是继承来的属性  // 查找父类的私有属性信息，ce必须是scope的子类  zend\_property\_info \*p=zend\_get\_parent\_private\_property(scope,ce,member);  // 如果子类有【公开】或【受保护】的属性，不要尝试使用父类的私有静态属性。  // 如果两个都是静态属性,优先使用当前作用域的属性。    // 父类中的属性有效 并且 （ 父类中的属性信息没有static 或 子类的有static ）  if (p && (!(p->flags & ZEND\_ACC\_STATIC) || (flags & ZEND\_ACC\_STATIC))) {  property\_info = p; // 使用父类的属性信息  flags = property\_info->flags; // 使用父类的属性信息标记  goto found; // 按找到有效属性操作，成功情况1  } else if (flags & ZEND\_ACC\_PUBLIC) { // 其他情况， 如果当前属性有public标记  goto found; // 按找到有效属性操作，成功情况2  }  }  if (flags & ZEND\_ACC\_PRIVATE) { // 情况3.2，如果是【私有】  if (property\_info->ce != ce) { // 如果属性信息不属于 要求的类，（也不属于正在执行的作用域，相当于完全没查到）  goto dynamic; // 按动态属性操作， 返回：动态属性标记：-1  } else { // 如果属性信息属于 要求的类。但要求的类不是当前正在执行的作用域（参看上一层判断）。  wrong: // 出错  // 信息可用，但拒绝访问，要报错  if (!silent) { // 非静默  // 报错，不可以访问此属性  zend\_bad\_property\_access(property\_info, ce, member);  }  return ZEND\_WRONG\_PROPERTY\_OFFSET; // 返回错误的 偏移量：0  }  } else { // 情况3.3，不是private，肯定是 protected了  // 检验是否存在继承关系（双向），如果没有  if (UNEXPECTED(!is\_protected\_compatible\_scope(property\_info->ce,scope))) {  goto wrong; // 按出错处理  }  // 成功情况3  }  }  // 成功情况4：如果属性信息所属类 是 正在执行的域，直接到这里，按正常找到操作  } |

zend\_get\_parent\_private\_property()函数用于获取获取父类的私有属性信息，代码如下：

|  |
| --- |
| static zend\_property\_info \*zend\_get\_parent\_private\_property(zend\_class\_entry \*scope,  zend\_class\_entry \*ce, zend\_string \*member) {  zval \*zv;  zend\_property\_info \*prop\_info;  // 如果两个类不同，并且当前域存有效，并且ce是scope的子类  if (scope != ce && scope && is\_derived\_class(ce, scope)) {  zv = zend\_hash\_find(&scope->properties\_info, member); // 通过名字获取类的属性  if (zv != NULL) { // 如果找到了  prop\_info = (zend\_property\_info\*)Z\_PTR\_P(zv); // zval中取出属性信息  // 如果属性是私有的，并且属于父类  if ((prop\_info->flags & ZEND\_ACC\_PRIVATE) && prop\_info->ce == scope) {  return prop\_info; // 返回属性信息  }  }  }  return NULL; // 查找失败，返回空  } |

is\_derived\_class()函数用于验证类的继承关系，代码如下：

|  |
| --- |
| // 验证p1是否是p2的子类  static bool is\_derived\_class(zend\_class\_entry \*child\_class, zend\_class\_entry \*parent\_class) {  child\_class = child\_class->parent; // 找到父类  while (child\_class) { // 如果父类有效  if (child\_class == parent\_class) { // 如果父类和要检测的类相同  return 1; // 返回是  }  child\_class = child\_class->parent; // 一直向上找父类  } // 找到没有父类为止  return 0; // 查找失败，返回否  } |

is\_protected\_compatible\_scope()函数用于检验两个类之间是否存在继承关系，两个类哪个是子类都可以，代码如下：

|  |
| --- |
| static int is\_protected\_compatible\_scope(zend\_class\_entry \*ce, zend\_class\_entry \*scope) {  return scope && (is\_derived\_class(ce, scope) || is\_derived\_class(scope, ce));  } |

#### **4）最后的检验**

代码如下：

|  |
| --- |
| **// zend\_get\_property\_offset()函数第4块代码。 步骤4，最后的检验操作**  found: // 正常找到  if (UNEXPECTED(flags & ZEND\_ACC\_STATIC)) { // 如果是静态属性  if (!silent) { // 如果没有要求静默，报错  // 报错：把静态属性当成非静态属性访问  zend\_error(E\_NOTICE, "Accessing static property %s::$%s as non static", ZSTR\_VAL(ce->name), ZSTR\_VAL(member));  }  return ZEND\_DYNAMIC\_PROPERTY\_OFFSET; // 返回动态偏移量： -1  }    offset = property\_info->offset; // 偏移量  // 如果不是有效的类型 （左侧25位都是0）  if (EXPECTED(!ZEND\_TYPE\_IS\_SET(property\_info->type))) {  property\_info = NULL; // 属性信息为null  } else { // 如果是有效的类型  \*info\_ptr = property\_info; // 放到属性信息指针里返回  } |

#### **5）更新缓存**

代码如下：

|  |
| --- |
| **// zend\_get\_property\_offset()函数第5块代码。 步骤5，更新缓存**  if (cache\_slot) { // 如果有缓存位置  // 前两位置个放类指针 和 顺序号，(slot)[0] = (ce); (slot)[1] = (ptr);  CACHE\_POLYMORPHIC\_PTR\_EX(cache\_slot, ce, (void\*)(uintptr\_t)offset);  CACHE\_PTR\_EX(cache\_slot + 2, property\_info); // 把属性信息指针放到第三个位置  }  return offset; // 返回顺序号  } |

### 读取对象属性

zend\_std\_read\_property()函数是读取成员变量的标准函数，代码如下：

|  |
| --- |
| Z |

## 二）使用按维度索引进行操作

### 使用维度索引进行写入

zend\_std\_write\_dimension()函数用于按维索引度写入，代码如下：

|  |
| --- |
| ZEND\_API void zend\_std\_write\_dimension(zend\_object \*object, zval \*offset, zval \*value) {  zend\_class\_entry \*ce = object->ce; // 类  zval tmp\_offset; // 维度索引  // 只有实现实了 zend\_ce\_arrayaccess 接口，才会有 arrayaccess\_funcs\_ptr  zend\_class\_arrayaccess\_funcs \*funcs = ce->arrayaccess\_funcs\_ptr;  if (EXPECTED(funcs)) { // 如果有实现 zend\_ce\_arrayaccess 接口  if (!offset) { // 如果没有传入维度索引  ZVAL\_NULL(&tmp\_offset); // 维度索引为空  } else { // 有传入维度索引  ZVAL\_COPY\_DEREF(&tmp\_offset, offset); // 维度索引创建副本  }  GC\_ADDREF(object); // 对象增加引用次数  // 调用zf\_offsetset()方法进行写入  zend\_call\_known\_instance\_method\_with\_2\_params(funcs->zf\_offsetset, object, NULL, &tmp\_offset, value);  OBJ\_RELEASE(object); // 减少引用数，引用数为0时释放对象  zval\_ptr\_dtor(&tmp\_offset); // 释放维度索引号  } else {  zend\_bad\_array\_access(ce); // 抛异常：不能把object当成array来访问  }  } |

### 使用维度索引进行读取

zend\_std\_read\_dimension()函数用于按维索引度读取，代码如下：

|  |
| --- |
| ZEND\_API zval \*zend\_std\_read\_dimension(zend\_object \*object, zval \*offset, int type, zval \*rv){  zend\_class\_entry \*ce = object->ce; // 所属类  zval tmp\_offset; // 维度索引  // 只有实现实了 zend\_ce\_arrayaccess 接口，才会有维度操作函数  zend\_class\_arrayaccess\_funcs \*funcs = ce->arrayaccess\_funcs\_ptr;  if (EXPECTED(funcs)) { // 如果有实现 zend\_ce\_arrayaccess 接口  if (offset == NULL) { // 如果没传递维度索引, 例如：echo $a[];  ZVAL\_NULL(&tmp\_offset); // 维度索引为null  } else { // 如果有传递维度索引  ZVAL\_COPY\_DEREF(&tmp\_offset, offset); // 给维度索引创建副本变量  }  GC\_ADDREF(object); // 对象增加引用次数  if (type == BP\_VAR\_IS) { // 如果是要先进行检验，防止报错  // 先调用 zf\_offsetexists() 方法验证维度索引是否存在  zend\_call\_known\_instance\_method\_with\_1\_params(funcs->zf\_offsetexists, object, rv, &tmp\_offset);  if (UNEXPECTED(Z\_ISUNDEF\_P(rv))) { // 如果返回值无效  OBJ\_RELEASE(object); // 减少引用数，引用数为0时释放对象  zval\_ptr\_dtor(&tmp\_offset); // 销毁维度索引  return NULL; // 返回空  }  if (!i\_zend\_is\_true(rv)) { // 如果返回结果 不是 true  OBJ\_RELEASE(object); // 减少引用数，引用数为0时释放对象  zval\_ptr\_dtor(&tmp\_offset); // 销毁维度索引  zval\_ptr\_dtor(rv); // 销毁查询结果  return &EG(uninitialized\_zval); // 返回未初始化 zval  }  zval\_ptr\_dtor(rv); // 释放查询结果  }  // 通过检验或不进行检验，调用zf\_offsetget方法按维度索引读取  zend\_call\_known\_instance\_method\_with\_1\_params(funcs->zf\_offsetget, object, rv, &tmp\_offset);  OBJ\_RELEASE(object); // 减少引用数，引用数为0时释放对象  zval\_ptr\_dtor(&tmp\_offset); // 销毁维度索引  if (UNEXPECTED(Z\_TYPE\_P(rv) == IS\_UNDEF)) { // 如果结果无效  if (UNEXPECTED(!EG(exception))) { // 如果没有异常  // 抛错：此索引号不存在  zend\_throw\_error(NULL, "Undefined offset for object of type %s used as array", ZSTR\_VAL(ce->name));  }  return NULL; // 返回空  }  return rv; // 返回值有效，返回查询到的值  } else { // 不支持 zend\_ce\_arrayaccess 接口的都报错  zend\_bad\_array\_access(ce); // 抛异常：不能把object当成array来访问  return NULL; // 返回空  }  } |

### 检验维度索引是否存在

zend\_std\_has\_dimension()函数用于检验维度索引是否存在，代码如下：

|  |
| --- |
| ZEND\_API int zend\_std\_has\_dimension(zend\_object \*object, zval \*offset, int check\_empty) {  zend\_class\_entry \*ce = object->ce; // 所属类  zval retval, tmp\_offset; // 临时索引号  int result; // 结果  // 只有实现实了 zend\_ce\_arrayaccess 接口，才会有 arrayaccess\_funcs\_ptr  zend\_class\_arrayaccess\_funcs \*funcs = ce->arrayaccess\_funcs\_ptr;  if (EXPECTED(funcs)) { // 如果有实现 zend\_ce\_arrayaccess 接口  ZVAL\_COPY\_DEREF(&tmp\_offset, offset); // 复制维度索引  GC\_ADDREF(object); // 增加对象引用次数  // 调用 zf\_offsetexists() 方法验证维度索引是否存在  zend\_call\_known\_instance\_method\_with\_1\_params(funcs->zf\_offsetexists, object, &retval, &tmp\_offset);  result = i\_zend\_is\_true(&retval); // 结果是否是true  zval\_ptr\_dtor(&retval); // 销毁返回值  // 如果需要检验空，并且返回结果为true，并且没有异常  if (check\_empty && result && EXPECTED(!EG(exception))) {  // 调用 zf\_offsetget 方法读取结果  zend\_call\_known\_instance\_method\_with\_1\_params(funcs->zf\_offsetget, object, &retval, &tmp\_offset);  result = i\_zend\_is\_true(&retval); // 验证元素是否是true  zval\_ptr\_dtor(&retval); // 销毁返回值  }  OBJ\_RELEASE(object); // 减少引用数，引用数为0时释放对象  zval\_ptr\_dtor(&tmp\_offset); // 销毁维度索引  } else {  zend\_bad\_array\_access(ce); // 抛异常：不能把object当成array来访问  return 0; // 返回 否  }  return result; // 返回结果  } |

### 使用维度索引进行删除

zend\_std\_unset\_property()函数用于使用维度索引进行删除，代码如下：

|  |
| --- |
| ZEND\_API void zend\_std\_unset\_dimension(zend\_object \*object, zval \*offset) {  zend\_class\_entry \*ce = object->ce;  zval tmp\_offset;  // 只有实现实了 zend\_ce\_arrayaccess 接口，才会有 arrayaccess\_funcs\_ptr  zend\_class\_arrayaccess\_funcs \*funcs = ce->arrayaccess\_funcs\_ptr;  if (EXPECTED(funcs)) { // 如果有实现 zend\_ce\_arrayaccess 接口  ZVAL\_COPY\_DEREF(&tmp\_offset, offset); // 复制维度索引  GC\_ADDREF(object); // 添加引用计数  // 调用 zf\_offsetunset() 方法进行操作  zend\_call\_known\_instance\_method\_with\_1\_params(funcs->zf\_offsetunset, object, NULL, &tmp\_offset);  OBJ\_RELEASE(object); // 减少引用数，引用数为0时释放对象  zval\_ptr\_dtor(&tmp\_offset); // 销毁维度索引  } else { // 否则报错  zend\_bad\_array\_access(ce); // 抛异常：不能把object当成array来访问  }  } |

## 三）其他操作

### 把对象转换成其他类型

zend\_std\_cast\_object\_tostring()函数用于使用把对象转换成其他类型，代码如下：

|  |
| --- |
| ZEND\_API zend\_result zend\_std\_cast\_object\_tostring(zend\_object \*readobj, zval \*writeobj, int type) {  switch (type) { // 按要求的类型操作  case IS\_STRING: { // 转成字符串类型  zend\_class\_entry \*ce = readobj->ce;  if (ce->\_\_tostring) { // 看所属类是否有 \_\_tostring() 方法，如果有  zval retval;  GC\_ADDREF(readobj); // 增加引用次数  // 调用\_\_tostring()方法  zend\_call\_known\_instance\_method\_with\_0\_params(ce->\_\_tostring, readobj, &retval);  zend\_object\_release(readobj); // 减少引用数，引用数为0时释放对象  if (EXPECTED(Z\_TYPE(retval) == IS\_STRING)) { // 如果返回字符串，转换成功  ZVAL\_COPY\_VALUE(writeobj, &retval); // 把结果复制到 writeobj  return SUCCESS; // 返回成功  }  // 转换结果不是字符串  zval\_ptr\_dtor(&retval); // 销毁转换结果  if (!EG(exception)) { // 如果有异常  // 抛出异常：\_\_tostring()必须返回string 类型  zend\_throw\_error(NULL, "Method %s::\_\_toString() must return a string value", ZSTR\_VAL(ce->name));  }  }  return FAILURE; // 如果没有\_\_tostring方法，返回失败  }  case \_IS\_BOOL: // 布尔类型  ZVAL\_TRUE(writeobj); // 写入 TRUE, 对象转成布尔型永远是true  return SUCCESS; // 返回成功  default: // 其他类型  return FAILURE; // 返回失败  }  } |

如上所示，标准对象转换方法只能把对象转换成字符串和布尔型，对象转换成字符串时需要调用它的\_\_tostring()方法。对象转成布尔型永远是true。

### 获取类型名

zend\_std\_get\_class\_name()函数用于获取对象的所属类名的副本，它是标准处理器中最简单的函数，代码如下：

|  |
| --- |
| ZEND\_API zend\_string \*zend\_std\_get\_class\_name(const zend\_object \*zobj) {  return zend\_string\_copy(zobj->ce->name);  } |

# N、对象仓库

对象仓库（objects store）是一个全局的指针列表，一般情况下，当新对象被初始化时，会调用zend\_objects\_store\_put()函数，把对象存放到对象仓库中，所以对象仓库中应该有全部对象的指针。

对象仓库的结构体是\_zend\_objects\_store，它有一个别名zend\_objects\_store，定义如下：

|  |
| --- |
| typedef struct \_zend\_objects\_store {  zend\_object \*\*object\_buckets; // 指向指针列表的指针  uint32\_t top; // 已使用到的位置  uint32\_t size; // 有效对象数量  int free\_list\_head; // 空指针链表的头元素编号  } zend\_objects\_store; |

这个结构体在全局只有一个实例，随全运行时局变量一起创建和销毁。在运行时全局变量\_zend\_executor\_globals中的引用如下：

|  |
| --- |
| struct \_zend\_executor\_globals {  ...  zend\_objects\_store objects\_store; // 对象仓库  ...  }; |

通常使用EG(objects\_store)来访问对象仓库。

对象仓库相关的常量和宏程序定义如下：

|  |
| --- |
| #define EG\_FLAGS\_OBJECT\_STORE\_NO\_REUSE (1<<1) 【对象容器不可复用】标记  #define OBJ\_BUCKET\_INVALID (1<<0) // 无效指针标记  // 检验对象是否有效（指针不包含无效标记）  #define IS\_OBJ\_VALID(o) (!(((zend\_uintptr\_t)(o)) & OBJ\_BUCKET\_INVALID))  // 给对象指针添加无效标记：转成int，最后一位标记成1 再转回成 指针  #define SET\_OBJ\_INVALID(o) ((zend\_object\*)((((zend\_uintptr\_t)(o)) | OBJ\_BUCKET\_INVALID)))  // 全局只有 zend\_objects\_API.c : zend\_objects\_store\_put 用到  // 把指针（用序号创建的假指针）转成 对象指针在对象容器中的序号  #define GET\_OBJ\_BUCKET\_NUMBER(o) (((zend\_intptr\_t)(o)) >> 1)  // 把要回收的序号n左移1位, 最后一位写成1。然后转成 zend\_object 指针，写到 o 里面。  #define SET\_OBJ\_BUCKET\_NUMBER(o, n) do { \  (o) = (zend\_object\*)((((zend\_uintptr\_t)(n)) << 1) | OBJ\_BUCKET\_INVALID); \  } while (0)  // 回收一个 objects\_store 里面的 元素指针  // 传入的 h( handle ) 是对象在全局对象容器 EG(objects\_store).object\_buckets 中的【索引号】，通过它可以直接在全局容器中获取指针  #define ZEND\_OBJECTS\_STORE\_ADD\_TO\_FREE\_LIST(h) do { \  /\* h是释放的指针元素索引号，让它指向原本的第一个空元素，让空元素形成一个串 \*/ \  SET\_OBJ\_BUCKET\_NUMBER(EG(objects\_store).object\_buckets[(h)], EG(objects\_store).free\_list\_head); \  /\* 把新释放的元素序号放在最前面 \*/ \  EG(objects\_store).free\_list\_head = (h); \  } while (0)  // 释放对象  #define OBJ\_RELEASE(obj) zend\_object\_release(obj) |

## 初始化对象仓库

对象仓库实例跟随运行时全局变量一起创建和销毁，但对象指针列表需要单独创建。zend\_objects\_store\_init()函数用于创建指针列表，初始化对象仓库，代码如下：

|  |
| --- |
| ZEND\_API void zend\_objects\_store\_init(zend\_objects\_store \*objects, uint32\_t init\_size) {  // 大小 = 初始大小 \* 对象指针的尺寸，类型为指针列表  objects->object\_buckets = (zend\_object \*\*) emalloc(init\_size \* sizeof(zend\_object\*));  objects->top = 1; // 跳过 0，使 if 判断时为真  objects->size = init\_size; // 列表中的指针数量  objects->free\_list\_head = -1; // 初始值为 -1，没有可复用的元素  memset(&objects->object\_buckets[0], 0, sizeof(zend\_object\*)); // 第一个指针用 0 填充  } |

## 向仓库中添加对象

zend\_objects\_store\_put()函数用于向对象仓库中添加对象，代码如下：

|  |
| --- |
| ZEND\_API void zend\_objects\_store\_put(zend\_object \*object) {  int handle;  // 如果 有可复用的元素 并且没有全局的 【对象容器不可复用】标记  if (EG(objects\_store).free\_list\_head != -1 && EXPECTED(!(EG(flags) & EG\_FLAGS\_OBJECT\_STORE\_NO\_REUSE))) {  handle = EG(objects\_store).free\_list\_head; // 第一个可复用元素  // 可复用链表的指针指向下一个元素  EG(objects\_store).free\_list\_head =  GET\_OBJ\_BUCKET\_NUMBER(EG(objects\_store).object\_buckets[handle]);  // 如果对象指针仓库已经满了  } else if (UNEXPECTED(EG(objects\_store).top == EG(objects\_store).size)) {  zend\_objects\_store\_put\_cold(object); // 开辟空间并存入新对象指针  return; // 完成  } else { // 如果有可用空间  // 使用新序号，objects\_store 元素数 +1  handle = EG(objects\_store).top++;  }  object->handle = handle; // 元素添加仓库指针序号  EG(objects\_store).object\_buckets[handle] = object; // 对应序号的指针 指向此对象  } |

如上所示，当指针列表已满时，使用zend\_objects\_store\_put\_cold()函数进行处理，函数代码如下：

|  |
| --- |
| static void zend\_objects\_store\_put\_cold(zend\_object \*object) {  int handle;  uint32\_t new\_size = 2 \* EG(objects\_store).size; // 每次重新分配，空间增加 1 倍  // 调整或重新分配空间  EG(objects\_store).object\_buckets = (zend\_object \*\*) erealloc(  EG(objects\_store).object\_buckets, new\_size \* sizeof(zend\_object\*));  EG(objects\_store).size = new\_size; // 新的元素数  handle = EG(objects\_store).top++; // 使用数量 +1  object->handle = handle; // 记录在指针列表中的序号  EG(objects\_store).object\_buckets[handle] = object; // 列表中指定序号的指针指向此对象  } |

## 从仓库中删除对象

zend\_objects\_store\_del()函数用于从对象仓库中删除对象，代码如下：

|  |
| --- |
| ZEND\_API void zend\_objects\_store\_del(zend\_object \*object) {  if (UNEXPECTED(GC\_TYPE(object) == IS\_NULL)) { // 如果已经被释放了  return; // 中断操作  }  if (!(OBJ\_FLAGS(object) & IS\_OBJ\_DESTRUCTOR\_CALLED)) { // 如果没有【销毁中】标记  GC\_ADD\_FLAGS(object, IS\_OBJ\_DESTRUCTOR\_CALLED); // 添加 【销毁中】标记  // 如果有特殊的销毁方法  if (object->handlers->dtor\_obj != zend\_objects\_destroy\_object  || object->ce->destructor) {  zend\_fiber\_switch\_block(); // 锁定fiber  GC\_SET\_REFCOUNT(object, 1); // 引用数为1  object->handlers->dtor\_obj(object); // 调用处理器，销毁对象  GC\_DELREF(object); // 引用次数 -1  zend\_fiber\_switch\_unblock(); // 解除 fiber 锁定  }  }  if (GC\_REFCOUNT(object) == 0) { // 如果引用次数为0  // handle 是对象在全局对象容器中的指针序号，通过它可以直接在全局容器中获取指针  uint32\_t handle = object->handle;  void \*ptr;  // 指针设置成无效  EG(objects\_store).object\_buckets[handle] = SET\_OBJ\_INVALID(object);  if (!(OBJ\_FLAGS(object) & IS\_OBJ\_FREE\_CALLED)) { // 如果没有【已释放】标记  GC\_ADD\_FLAGS(object, IS\_OBJ\_FREE\_CALLED); // 添加 【已释放】标记  GC\_SET\_REFCOUNT(object, 1); // 引用次数为1  object->handlers->free\_obj(object); // 调用处理器释放 对象  }  ptr = ((char\*)object) - object->handlers->offset; // 对象的开头指针，一般 offset是0  GC\_REMOVE\_FROM\_BUFFER(object); // 从gc缓存中删除  efree(ptr); // 释放对象  ZEND\_OBJECTS\_STORE\_ADD\_TO\_FREE\_LIST(handle); // 回收对象仓库里面的元素指针  }  } |

## 销毁仓库中的对象

zend\_objects\_store\_call\_destructors()函数用于给仓库中的所有对象添加“已销毁”标记，碰到有特殊销毁器的对象，调用销毁器销毁它，代码如下：

|  |
| --- |
| ZEND\_API void zend\_objects\_store\_call\_destructors(zend\_objects\_store \*objects) {  EG(flags) |= EG\_FLAGS\_OBJECT\_STORE\_NO\_REUSE; // 添加 【对象容器不可复用】标记  if (objects->top > 1) { // 如果仓库里有对象  zend\_fiber\_switch\_block(); // 锁定 fiber  uint32\_t i;  for (i = 1; i < objects->top; i++) { // 遍历每一个对象  zend\_object \*obj = objects->object\_buckets[i]; // 对象指针  if (IS\_OBJ\_VALID(obj)) { // 如果对象有效  if (!(OBJ\_FLAGS(obj) & IS\_OBJ\_DESTRUCTOR\_CALLED)) { // 没有【已销毁】标记  GC\_ADD\_FLAGS(obj, IS\_OBJ\_DESTRUCTOR\_CALLED); // 添加【已销毁】标记  // 如果有自己的销毁方法，或所属类有析构方法  if (obj->handlers->dtor\_obj != zend\_objects\_destroy\_object  || obj->ce->destructor) {  GC\_ADDREF(obj); // 引用计数 +1  obj->handlers->dtor\_obj(obj); // 调用自字义方法销毁对象  GC\_DELREF(obj); // 引用计数 -1  }  }  }  }  zend\_fiber\_switch\_unblock(); // 解锁 fiber  }  } |

zend\_objects\_store\_mark\_destructed()函数用于把仓库中的所有对象标记成“已销毁”，代码如下：

|  |
| --- |
| ZEND\_API void zend\_objects\_store\_mark\_destructed(zend\_objects\_store \*objects) {  if (objects->object\_buckets && objects->top > 1) { // 如果列表里有对象  zend\_object \*\*obj\_ptr = objects->object\_buckets + 1; // 从第一个开始  zend\_object \*\*end = objects->object\_buckets + objects->top; // 到最后一个  do { // 遍历每一个对象  zend\_object \*obj = \*obj\_ptr;  if (IS\_OBJ\_VALID(obj)) { // 如果对象有效  GC\_ADD\_FLAGS(obj, IS\_OBJ\_DESTRUCTOR\_CALLED); // 添加 【已销毁】 标记  }  obj\_ptr++; // 下一个  } while (obj\_ptr != end);  }  } |

zend\_objects\_store\_free\_object\_storage()函数用于释放仓库里的所有对象：

|  |
| --- |
| ZEND\_API void zend\_objects\_store\_free\_object\_storage(zend\_objects\_store \*objects, bool fast\_shutdown) {  zend\_object \*\*obj\_ptr, \*\*end, \*obj;  if (objects->top <= 1) { // 仓库里没有对象  return;  }  // 结束位置，就是初始化空容器时的位置，开头留1个指针的空间  end = objects->object\_buckets + 1;  // 从最后一个有效对象开始  obj\_ptr = objects->object\_buckets + objects->top;  if (fast\_shutdown) { // 如果要求快速关闭  do { // 倒序遍历  obj\_ptr--; // 游标退 1格  obj = \*obj\_ptr; // 对象指针  if (IS\_OBJ\_VALID(obj)) { // 如果对象有效  if (!(OBJ\_FLAGS(obj) & IS\_OBJ\_FREE\_CALLED)) { // 如果没有【已释放】标记  GC\_ADD\_FLAGS(obj, IS\_OBJ\_FREE\_CALLED); // 添加【已释放】标记  // 如果销毁方法不是 zend\_object\_std\_dtor  if (obj->handlers->free\_obj != zend\_object\_std\_dtor) {  GC\_ADDREF(obj); // 引用数 +1  obj->handlers->free\_obj(obj); // 调用自己的销毁方法  }  // 如果销毁方法是 zend\_object\_std\_dtor，什么也不做  }  }  } while (obj\_ptr != end); // 直到开头位置  } else { // 不是快速关闭  do { // 倒序遍历  obj\_ptr--; // 游标退 1格  obj = \*obj\_ptr; // 对象指针  if (IS\_OBJ\_VALID(obj)) { // 如果对象有效  if (!(OBJ\_FLAGS(obj) & IS\_OBJ\_FREE\_CALLED)) { // 如果对象没有【已释放】标记  GC\_ADD\_FLAGS(obj, IS\_OBJ\_FREE\_CALLED); // 添加【已释放】标记  GC\_ADDREF(obj); // 引用数 -1  obj->handlers->free\_obj(obj); // 释放对象  }  }  } while (obj\_ptr != end); // 直到开头位置  }  } |

## 销毁对象仓库

zend\_objects\_store\_destroy()函数用于销毁对象仓库，代码如下：

|  |
| --- |
| ZEND\_API void zend\_objects\_store\_destroy(zend\_objects\_store \*objects) {  efree(objects->object\_buckets); // 释放对象指针列表  objects->object\_buckets = NULL; // 指针为null  } |

# Change Log

2025.6.18 分离出第二篇。