# 一、基本概念

面向对象是PHP内核中最复杂的部分，因为面向对象本身是一套很复杂的程序设计思想，与之相关的类、接口、trait等常用结构以及封装、继承等常用功能在底层实现上都需要考虑诸多情况。学习这部分，可以深入了解面向对象的底层实现原理。

类的编译过程在zend\_compile.c文件中，主要用到zend\_compile\_class\_decl()函数，业务逻辑比较简单，不作为本篇重点。本篇重点介绍类的编译后的继承、接口实现、trait引用以及过程中的权限检查等核心功能。本篇涉及的概念有类（class），接口（interface），trait（特性），枚举（enum）等，其中枚举不是本篇重点内容。

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逻辑，忽略32位系统中的实现逻辑，也不进行差异比较。

## 一）类的数据结构

面向对象编程中，最核心的数据结构是类，它在PHP内核中对应的结构体是\_zend\_class\_entry，它的数据结构比较复杂，代码如下：

|  |
| --- |
| struct \_zend\_class\_entry {  char type; // 类型，用户类或内置类  zend\_string \*name; // 名称  union { // 联合类型  zend\_class\_entry \*parent; // 父类指针  zend\_string \*parent\_name; // 父类名指针  };  int refcount; // 引用数  uint32\_t ce\_flags; // 标记  int default\_properties\_count; // 默认属性数量  int default\_static\_members\_count; // 默认静态属性数量  zval \*default\_properties\_table; // 默认属性列表  zval \*default\_static\_members\_table; // 默认静态属性列表  // 等同于：zval \* static\_numbers\_table\_ptr;  ZEND\_MAP\_PTR\_DEF(zval \*, static\_members\_table);  HashTable function\_table; // 成员方法列表  HashTable properties\_info; // 属性信息  HashTable constants\_table; // 常量列表  // 等同于：zend\_class\_mutable\_data \* mutable\_data\_ptr;  ZEND\_MAP\_PTR\_DEF(zend\_class\_mutable\_data\*, mutable\_data);  zend\_inheritance\_cache\_entry \*inheritance\_cache; // 继承缓存  struct \_zend\_property\_info \*\*properties\_info\_table; // \*\*属性信息表  zend\_function \*constructor; // 构造方法  zend\_function \*destructor; // 析构方法  zend\_function \*clone; // 克隆方法  zend\_function \*\_\_get; // 动态读取属性  zend\_function \*\_\_set; // 动态更新属性  zend\_function \*\_\_unset; // 动态删除属性  zend\_function \*\_\_isset; // 动态更新属性  zend\_function \*\_\_call; // 动态调用成员方法  zend\_function \*\_\_callstatic; // 动态调用静态成员方法  zend\_function \*\_\_tostring; // 转成字符串  zend\_function \*\_\_debugInfo; // 获取调试信息的魔术方法  zend\_function \*\_\_serialize; // 序列化和反序列化的魔术方法  zend\_function \*\_\_unserialize;  // 迭代函数，只有当类实现接口 Iterator 或 IteratorAggregate 时分配此函数  zend\_class\_iterator\_funcs \*iterator\_funcs\_ptr;  // 使用维度访问，只有当类实现接口 ArrayAccess 时分配此函数。例如 $a[1]  zend\_class\_arrayaccess\_funcs \*arrayaccess\_funcs\_ptr;  union { // 联合型，处理器  // 这个是php里new 语句调用的的创建对象的方法  zend\_object\* (\*create\_object)(zend\_class\_entry \*class\_type);  int (\*interface\_gets\_implemented)(zend\_class\_entry \*iface, zend\_class\_entry \*class\_type); /\* a class implements this interface \*/  };  // 获取迭代器的函数  zend\_object\_iterator \*(\*get\_iterator)(zend\_class\_entry \*ce, zval \*object, int by\_ref);  // 获取静态方法的方法  zend\_function \*(\*get\_static\_method)(zend\_class\_entry \*ce, zend\_string\* method);  /\* 序列化和反序列化回调函数 \*/  int (\*serialize)(zval \*object, unsigned char \*\*buffer, size\_t \*buf\_len, zend\_serialize\_data \*data);  int (\*unserialize)(zval \*object, zend\_class\_entry \*ce, const unsigned char \*buf, size\_t buf\_len, zend\_unserialize\_data \*data);  uint32\_t num\_interfaces; // 接口数量  uint32\_t num\_traits; // trait 数量  union { // 实现的接口，联合类型  zend\_class\_entry \*\*interfaces; // 接口指针列表  // zend\_class\_name \*interface\_names; // 接口名  };  zend\_class\_name \*trait\_names; // trait 名称列表  zend\_trait\_alias \*\*trait\_aliases; // trait 别名列表  zend\_trait\_precedence \*\*trait\_precedences; // trait 优先顺序列表  HashTable \*attributes; // 修饰属性  uint32\_t enum\_backing\_type; // enum枚举类型使用  HashTable \*backed\_enum\_table; // enum枚举类型使用  union { // 代码信息，联合类型  struct {  zend\_string \*filename; // 文件名  uint32\_t line\_start; // 开始行  uint32\_t line\_end; // 结束行  zend\_string \*doc\_comment; // 注释文档  } user; // 以上由用户定义  struct {  const struct \_zend\_function\_entry \*builtin\_functions; // 内置函数指针  struct \_zend\_module\_entry \*module; // 模块  } internal; // 以上是内置信息  } info;  }; |

## 二）相关常量

相关常量定义如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **常量值** | **说明** |
| ZEND\_INTERNAL\_CLASS | 1 | PHP内置类 |
| ZEND\_USER\_CLASS | 2 | PHP代码中定义的类 |

类，属性（成员变量），方法，常量 使用的标记常量：

|  |  |  |
| --- | --- | --- |
| **常量名** | **常量值** | **说明** |
| ZEND\_ACC\_PUBLIC | 1 << 0 | 可见性public，  可用于成员方法、属性、成员常量 |
| ZEND\_ACC\_PROTECTED | 1 << 1 | 可见性protected，  可用于成员方法、属性、成员常量 |
| ZEND\_ACC\_PRIVATE | 1 << 2 | 可见性private，  可用于成员方法、属性、成员常量 |
| ZEND\_ACC\_CHANGED | 1 << 3 | 覆盖过父类的私有属性或方法  可用于类的属性和方法 |
| ZEND\_ACC\_STATIC | 1 << 4 | 静态（static）  可用于类的属性和方法 |
| ZEND\_ACC\_PROMOTED | 1 << 5 | 带有可见性或readonly  可用于类属性（可用于类常量，但实际上并未用到） |
| ZEND\_ACC\_FINAL | 1 << 5 | final（不可覆盖）  可用于类和类方法 |
| ZEND\_ACC\_ABSTRACT | 1 << 6 | abstract（抽象）  可用于类和类方法 |
| ZEND\_ACC\_EXPLICIT\_ABSTRACT\_CLASS | 1 << 6 | 抽象类  可用于类 |
| ZEND\_ACC\_READONLY | 1 << 7 | 只读  可用于类属性 |
| ZEND\_ACC\_IMMUTABLE | 1 << 7 | 不可更改  可用于类和类方法 |
| ZEND\_ACC\_HAS\_TYPE\_HINTS | 1 << 8 | 类方法有指定类型的参数 或 类有指定类型的属性  可用于类或类方法 |
| ZEND\_ACC\_TOP\_LEVEL | 1 << 9 | 在顶层定义的类或函数  可用于类、类方法 |
| ZEND\_ACC\_PRELOADED | 1 << 10 | 预加载的类或类方法操作码  可用于类或类方法 |
| ZEND\_CLASS\_CONST\_IS\_CASE | 1 << 6 | 表示枚举中的case  可用于类常量（枚举的case和类常量使用相同数据结构） |
| ZEND\_ACC\_INTERFACE | 1 << 0 | 接口  可用于类（zend\_class\_entry实例） |
| ZEND\_ACC\_TRAIT | 1 << 1 | Trait  可用于类（zend\_class\_entry实例） |
| ZEND\_ACC\_ANON\_CLASS | 1 << 2 | 匿名类  可用于类 |
| ZEND\_ACC\_ENUM | 1 << 28 | enum枚举类型  可用于类（zend\_class\_entry实例） |
| ZEND\_ACC\_LINKED | 1 << 3 | 类已与父类以及相关接口和trait链接完毕 |
| ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS | 1 << 4 | 隐式抽象类（包含抽象方法的类）  可用于类 |
| ZEND\_ACC\_USE\_GUARDS | 1 << 11 | 有魔术方法 \_\_get/\_\_set/\_\_unset/\_\_isset 的类  可用于类 |
| ZEND\_ACC\_CONSTANTS\_UPDATED | 1 << 12 | 类常量已更新  可用于类 |
| ZEND\_ACC\_NO\_DYNAMIC\_PROPERTIES | 1 << 13 | 没有动态属性的类  可用于类 |
| ZEND\_HAS\_STATIC\_IN\_METHODS | 1 << 14 | 包含使用静态变量的类方法  可用于类 |
| ZEND\_ACC\_ALLOW\_DYNAMIC\_PROPERTIES | 1 << 15 | 可能包含动态属性的类  可用于类 |
| ZEND\_ACC\_READONLY\_CLASS | 1 << 16 | 只读类  可用于类 |
| ZEND\_ACC\_RESOLVED\_PARENT | 1 << 17 | 父类已处理完成  可用于类 |
| ZEND\_ACC\_RESOLVED\_INTERFACES | 1 << 18 | 实现的接口已处理完成  可用于类 |
| ZEND\_ACC\_UNRESOLVED\_VARIANCE | 1 << 19 | 带有未处理责任的类  可用于类 |
| ZEND\_ACC\_NEARLY\_LINKED | 1 << 20 | 将要完成链接的类  可用于类 |
| ZEND\_ACC\_CACHED | 1 << 22 | 储存在opcache里（可能只有部分）  可用于类 |
| ZEND\_ACC\_CACHEABLE | 1 << 23 | 延时变异检查时使用的临时标记  可用于类 |
| ZEND\_ACC\_HAS\_AST\_CONSTANTS | 1 << 24 | 包含常量表达式赋值的类常量  可用于类 |
| ZEND\_ACC\_HAS\_AST\_PROPERTIES | 1 << 25 | 包含常量表达式赋值的类属性  可用于类 |
| ZEND\_ACC\_HAS\_AST\_STATICS | 1 << 26 | 包含常量表达式赋值的静态属性  可用于类 |
| ZEND\_ACC\_FILE\_CACHED | 1 << 27 | 来自文件或内存中加载  可用于类 |
| ZEND\_ACC\_NOT\_SERIALIZABLE | 1 << 29 | 不可序列化和反序列化  可用于类 |
| ZEND\_ACC\_DEPRECATED | 1 << 11 | 弃用标记  可用于类方法 |
| ZEND\_ACC\_RETURN\_REFERENCE | 1 << 12 | 使用引用返回  可用于类方法 |
| ZEND\_ACC\_HAS\_RETURN\_TYPE | 1 << 13 | 有规定返回类型  可用于类方法 |
| ZEND\_ACC\_VARIADIC | 1 << 14 | 支持可变参数数量  可用于类方法 |
| ZEND\_ACC\_HAS\_FINALLY\_BLOCK | 1 << 15 | 包含finally代码块  可用于类方法 |
| ZEND\_ACC\_EARLY\_BINDING | 1 << 16 | 在顶层定义的，并创建了ZEND\_DECLARE\_CLASS\_DELAYED操作码  可用于类方法 |
| ZEND\_ACC\_USES\_THIS | 1 << 17 | 闭包中使用$this  可用于类方法 |
| ZEND\_ACC\_CALL\_VIA\_TRAMPOLINE | 1 << 18 | 通过用户函数弹跳调用，其实就是 \_\_call, \_\_callstatic 两个方法  可用于类方法 |
| ZEND\_ACC\_NEVER\_CACHE | 1 << 19 | 禁用行内缓存  可用于类方法 |
| ZEND\_ACC\_TRAIT\_CLONE | 1 << 20 | 来自Trait克隆  可用于类方法 |
| ZEND\_ACC\_CTOR | 1 << 21 | 构造方法  可用于类方法 |
| ZEND\_ACC\_CLOSURE | 1 << 22 | 闭包  可用于类方法 |
| ZEND\_ACC\_FAKE\_CLOSURE | 1 << 23 | 可用于类方法 |
| ZEND\_ACC\_GENERATOR | 1 << 24 | 可用于类方法 |
| ZEND\_ACC\_DONE\_PASS\_TWO | 1 << 25 | 可用于类方法 |
| ZEND\_ACC\_ARENA\_ALLOCATED | 1 << 25 | 可用于类方法 |
| ZEND\_ACC\_HEAP\_RT\_CACHE | 1 << 26 | 可用于类方法 |
| ZEND\_ACC\_USER\_ARG\_INFO | 1 << 26 | Closure::\_\_invoke()使用的标记，整数  可用于类方法 |
| ZEND\_ACC\_COMPILE\_TIME\_EVAL | 1 << 27 | 支持函数在编译时计算  可用于类方法 |
| ZEND\_ACC\_STRICT\_TYPES | 1U << 31 | 操作码使用严格模式的类型  可用于类方法 |

## 三）类的创建和绑定

### 在编译时创建类

PHP代码中定义的类在编译会在编译时被创建成类实例。参见zend\_compile.c文件中的zend\_compile\_class\_decl()函数。创建类实例的代码如下：

|  |
| --- |
| // 创建类  zend\_class\_entry \*ce = zend\_arena\_alloc(&CG(arena), sizeof(zend\_class\_entry)); |

关于zend\_arena\_alloc()函数的介绍参见“内存管理篇”。

### 早期绑定

在zend\_compile\_class\_decl()函数中创建类时，如果在特定条件下，会触发早期绑定，代码如下：

|  |
| --- |
| // 目前不 预先绑定 实现了接口或应用了 trait 的类  /\* We currently don't early-bind classes that implement interfaces or use traits \*/  // 如果没有接口 并且没有 trait 并且 没有 ZEND\_COMPILE\_WITHOUT\_EXECUTION 标记  if (!ce->num\_interfaces && !ce->num\_traits  && !(CG(compiler\_options) & ZEND\_COMPILE\_WITHOUT\_EXECUTION)) {  if (toplevel) { // 如果在顶层  if (extends\_ast) { // 如果有继承语句  zend\_class\_entry \*parent\_ce = zend\_lookup\_class\_ex(  ce->parent\_name, NULL, ZEND\_FETCH\_CLASS\_NO\_AUTOLOAD);  if (parent\_ce  && ((parent\_ce->type != ZEND\_INTERNAL\_CLASS) || !(CG(compiler\_options) & ZEND\_COMPILE\_IGNORE\_INTERNAL\_CLASSES))  // 如果是用户类（通过 zend\_compile\_class\_decl 方法定义的类）  && ((parent\_ce->type != ZEND\_USER\_CLASS) || !(CG(compiler\_options) & ZEND\_COMPILE\_IGNORE\_OTHER\_FILES) || (parent\_ce->info.user.filename == ce->info.user.filename))) {  **if (zend\_try\_early\_bind(ce, parent\_ce, lcname, NULL)) { // 早期绑定**  zend\_string\_release(lcname);  return;  }  } |

如上所示，触发早期绑定需要满足以下条件：

1）当前类没有实现接口，也没有引用trait；

2）编译选项中没有开启ZEND\_COMPILE\_WITHOUT\_EXECUTION；

3）当前类是在顶层声名的；

4）当前类继承了其他类；

5）同时满足以下两个条件：

a）父类不是内置类或没有开启ZEND\_COMPILE\_IGNORE\_INTERNAL\_CLASSES编译选项

b）父类不是用户定义或没有开启ZEND\_COMPILE\_IGNORE\_OTHER\_FILES编译选项或父类和当前类在同一文件中定义。

满足以上条件时，会调用zend\_try\_early\_bind()函数进行早期绑定。

### 创建类的操作码

在zend\_compile\_class\_decl()函数的末尾，如果类不是匿名类，会创建ZEND\_DECLARE\_CLASS操作码，有时还会创建ZEND\_DECLARE\_CLASS\_DELAYED操作码，代码如下：

|  |
| --- |
| // 情况1：如果是匿名类  if (decl->flags & ZEND\_ACC\_ANON\_CLASS) {  **opline->opcode = ZEND\_DECLARE\_ANON\_CLASS; // 操作码：声名匿名类**  // 给 CG(active\_op\_array) 添加缓存大小，添加 一个 void\*指针的大小，返回添加后的缓存大小  opline->extended\_value = zend\_alloc\_cache\_slot();  zend\_make\_var\_result(result, opline); // 把操作码的运行结果返回给 result  if (!zend\_hash\_add\_ptr(CG(class\_table), lcname, ce)) { // 如果添加类失败，报错  zend\_error\_noreturn(E\_ERROR, ...); // 这个错误实现上不会发生  }  // 情况2：如果不是匿名类  } else {  zend\_string \*key = NULL; // 必须创建一个未使用的 RTD key  do { // 一直循环  zend\_tmp\_string\_release(key); // 释放key  // 取得运行时类名  key = zend\_build\_runtime\_definition\_key(lcname, decl->start\_lineno);  // 如果添加到类列表不成功，重来  } while (!zend\_hash\_add\_ptr(CG(class\_table), key, ce));  zend\_add\_literal\_string(&key); // 新生成的RTD key添加到小写类名后面  **opline->opcode = ZEND\_DECLARE\_CLASS; // 操作码：声名类**  // 如果有继承子句，并且当前在最上层 ，并且编译选项有延时绑定，并且没有接口实现，并且没有引用trait  if (extends\_ast && toplevel  && (CG(compiler\_options) & ZEND\_COMPILE\_DELAYED\_BINDING)  // 这里不绑定实现了接口或引用了trait的类  && !ce->num\_interfaces && !ce->num\_traits  ) {  // 添加标记 ZEND\_ACC\_EARLY\_BINDING  CG(active\_op\_array)->fn\_flags |= ZEND\_ACC\_EARLY\_BINDING;  // 延时声名类  **opline->opcode = ZEND\_DECLARE\_CLASS\_DELAYED; // 操作码：声名延时类**  // 给 CG(active\_op\_array) 添加缓存大小，添加 一个 void\*指针的大小，返回添加后的缓存大小  opline->extended\_value = zend\_alloc\_cache\_slot();  // 结果类型是 IS\_UNUSED  opline->result\_type = IS\_UNUSED;  // 结果行号为-1  opline->result.opline\_num = -1;  }  }  } |

如上所示，业务逻辑可以归纳如下：

1）当前类是匿名类：创建ZEND\_DECLARE\_ANON\_CLASS（声名匿名类）操作码；

2）当前类不是匿名类：

a）当满足条件（当前类有继承其他类，并且当前类在顶层定义，并且编译选项中开户了“编译延时绑定”ZEND\_COMPILE\_DELAYED\_BINDING，并且当前类没有实现接口也没有引用trait）时：创建ZEND\_DECLARE\_CLASS\_DELAYED（创建延时类）操作码；

b）不满足以上条件时，只创建ZEND\_DECLARE\_CLASS（声名类）操作码。

### 在执行时绑定类

在PHP虚拟机执行前文中创建的3个操作码时，都会触发绑定类操作，操作码的处理逻辑都在zend\_vm\_def.h文件中。

#### **执行ZEND\_DECLARE\_ANON\_CLASS操作码**

执行ZEND\_DECLARE\_ANON\_CLASS操作码时，会调用zend\_do\_link\_class()函数来链接定类，业务逻辑如下：

|  |
| --- |
| ZEND\_VM\_HANDLER(146, ZEND\_DECLARE\_ANON\_CLASS, ANY, ANY, CACHE\_SLOT) {  ...  if (UNEXPECTED(ce == NULL)) { // 如果当前类不存在  ...  if (!(ce->ce\_flags & ZEND\_ACC\_LINKED)) { // 如果类没有链接过  ...  **ce = zend\_do\_link\_class(ce, (OP2\_TYPE == IS\_CONST) ? Z\_STR\_P(RT\_CONSTANT(opline, opline->op2)) : NULL, rtd\_key);**  ... |

#### **执行**ZEND\_DECLARE\_CLASS\_DELAYED**操作码**

执行ZEND\_DECLARE\_CLASS\_DELAYED操作码时，会调用zend\_bind\_class\_in\_slot()函数来绑定类，业务逻辑如下：

|  |
| --- |
| ZEND\_VM\_HANDLER(145, ZEND\_DECLARE\_CLASS\_DELAYED, CONST, CONST) {  ...  if (ce == NULL) { // 如果当前类不存在  ...  if (zv) { // 如果找到类  ...  **ce = zend\_bind\_class\_in\_slot(zv, lcname, Z\_STR\_P(RT\_CONSTANT(opline, opline->op2)));**  ... |

#### **执行**ZEND\_DECLARE\_CLASS**操作码**

执行ZEND\_DECLARE\_CLASS操作码时，会调用do\_bind\_class()函数来绑定类，业务逻辑如下：

|  |
| --- |
| ZEND\_VM\_HANDLER(144, ZEND\_DECLARE\_CLASS, CONST, ANY){  ...  **do\_bind\_class(RT\_CONSTANT(opline, opline->op1), (OP2\_TYPE == IS\_CONST) ? Z\_STR\_P(RT\_CONSTANT(opline, opline->op2)) : NULL);**  ... |

do\_bind\_class()函数用于绑定类，它的调用路径如下：

|  |
| --- |
| do\_bind\_class() // 绑定类  ->zend\_bind\_class\_in\_slot() // 注册派生类  ->zend\_do\_link\_class() // 链接类 |

如上所示，以上3个操作码最终都要调用zend\_do\_link\_class()函数来链接类。

## 四）小结

类在编译时创建，刚创建完的类没有继承父类，没有实现接口，也没有引用trait，无法直接使用。zend\_try\_early\_bind()函数和do\_bind\_class()函数，分别用于实现早期绑定和绑定，在这个过程中会让类继承父类、实现接口、引用trait并对类进行检验。经过这些操作后类才可以被正常使用。

下面先介绍一些类的最常用的基础操作，再介绍继承、接口实现、trait引用等核心功能，最后介绍zend\_try\_early\_bind()函数和do\_bind\_class()函数，把整个流程串联起来。

# 二、类的常用操作

## 一）检验类名是否合法

zend\_is\_valid\_class\_name()函数用于检验类名是否合法，代码如下：

|  |
| --- |
| // 验证类名有效 /\* 0-9 a-z A-Z \_ \ 0x80-0xff \*/  ZEND\_API bool zend\_is\_valid\_class\_name(zend\_string \*name) {  for (size\_t i = 0; i < ZSTR\_LEN(name); i++) { // 遍历每一个字符  unsigned char c = ZSTR\_VAL(name)[i]; // 取出字符  if (!ZEND\_BIT\_TEST(valid\_chars, c)) { // 检验字符是否合法  return 0; // 有一个字符不合法，就返回false  }  }  return 1; // 全部合法，返回true  } |

ZEND\_BIT\_TEST()宏程序用于验证字符是否在允许的范围内，代码如下：

|  |
| --- |
| /\* 验证字符是否在 0-9 a-z A-Z \_ \ 0x80-0xff 这6段中 \*/  static const uint32\_t valid\_chars[8] = {  0x00000000, 0x03ff0000, 0x97fffffe, 0x07fffffe,  0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,  }; // 个块，每个块占4Bytes,共32Bytes,256个位  // 检验一块内存中的指定bit是否是1。p1:内存（32位整数数组），p2:位置序号  #define ZEND\_BIT\_TEST(bits, bit) \  (((bits)[(bit) / (sizeof((bits)[0])\*8)] >> ((bit) & (sizeof((bits)[0])\*8-1))) & 1)  // (((bits)[(bit) / (4\*8)] >> ((bit) & (4\*8-1))) & 1)  // (((bits)[(bit) / (32)] >> ((bit) & (31))) & 1)  // (((bits)[块号] >> ( (bit)%31 )) & 1)  // (((bits)[块号] >> bit在块中的位置 ) & 1)  // 移动后，这个bit就到了右侧第一位 & 1  // 得到这个bit 是否是 1 |

如上所示，valid\_chars由8个32位整数组成，共占32Bytes，256个bit，每个bit对应一个ascii码序号，bit值为1表示这个字符可用，为0表示这个字符不可用。ZEND\_BIT\_TEST()宏程序用于找到字符对应的bit，来确定这个字符是否可用。

## 二）查找类

在zend\_inheritance.c文件中，用于查找类的函数有lookup\_class()函数和register\_unresolved\_classes()函数。这两个函数都调用lookup\_class\_ex()函数来执行查找，其业务逻辑中涉及的函数比较多，逻辑比较复杂。

**lookup\_class()函数用于，通过类名查找类**，代码如下：

|  |
| --- |
| // 用类名查找类，p1:当前域，p2:类名  static zend\_class\_entry \*lookup\_class(zend\_class\_entry \*scope, zend\_string \*name) {  // 用类名查找类，p1:当前域，p2:类名，p3:不注册延时加载  return lookup\_class\_ex(scope, name, /\* register\_unresolved \*/ false);  } |

**register\_unresolved\_classes()函数用于通过类型（zend\_type实例）查找类**，代码如下：

|  |
| --- |
| // p1:当前域，p2:类型或类型列表  static void register\_unresolved\_classes(zend\_class\_entry \*scope, zend\_type type) {  zend\_type \*single\_type;  ZEND\_TYPE\_FOREACH(type, single\_type) { // 遍历子类型  if (ZEND\_TYPE\_HAS\_LIST(\*single\_type)) { // 如果子类型是列表  register\_unresolved\_classes(scope, \*single\_type); // 递归  continue; // 下一个子类型  }  if (ZEND\_TYPE\_HAS\_NAME(\*single\_type)) { // 如果子类型有名称  // 获取类名，先处理parent和self, 如果都不是，用当前域类名。p1:当前域  zend\_string \*class\_name = resolve\_class\_name(scope, ZEND\_TYPE\_NAME(\*single\_type));  // 用类名查找类，p1:当前域，p2:类名，p3:注册延时加载  lookup\_class\_ex(scope, class\_name, /\* register\_unresolved \*/ true);  }  // 如果没有名称，说明是内置类型，不需要任何操作  } ZEND\_TYPE\_FOREACH\_END();  } |

### resolve\_class\_name()函数

resolve\_class\_name()函数用于在查找类名时匹配parent和self关键字，函数代码如下：

|  |
| --- |
| static zend\_string \*resolve\_class\_name(zend\_class\_entry \*scope, zend\_string \*name) {  // 忽略大小写比较，如果类名是 parent 并且 当前类有父类  if (zend\_string\_equals\_literal\_ci(name, "parent") && scope->parent) {  if (scope->ce\_flags & ZEND\_ACC\_RESOLVED\_PARENT) { // 如果已经完成对父类的继承  return scope->parent->name; // 返回父类的名称（完成继承后才有）  } else { // 没有完成对父类的继承  return scope->parent\_name; // 返回此类中记录的类名  }  } else if (zend\_string\_equals\_literal\_ci(name, "self")) { // 如果类名是self  return scope->name; // 返回当前类名  } else { // 如果都不是，返回原名字  return name;  }  } |

### class\_visible()函数

class\_visible()函数用于检查类是否可以在上下文中访问，函数代码如下：

|  |
| --- |
| static bool class\_visible(zend\_class\_entry \*ce) {  if (ce->type == ZEND\_INTERNAL\_CLASS) { // 如果是内置类  // 如果编译选项没有要求忽略内置类，就可用  return !(CG(compiler\_options) & ZEND\_COMPILE\_IGNORE\_INTERNAL\_CLASSES);  } else { // 不是内置类，是用户定义的类  // 如果编译选项没有要求忽略其他文件 或 此类属于当前文件。这样此类都可以访问。  return !(CG(compiler\_options) & ZEND\_COMPILE\_IGNORE\_OTHER\_FILES)  || ce->info.user.filename == CG(compiled\_filename);  }  } |

### lookup\_class\_ex()函数

lookup\_class\_ex()函数用于根据当前环境，尝试加载类，只在zend\_inheritance.c文件中用到，代码如下：

|  |
| --- |
| // p1:所在域，p2:要查找的类名，p3:是否注册延时加载  static zend\_class\_entry \*lookup\_class\_ex(  zend\_class\_entry \*scope, zend\_string \*name, bool register\_unresolved) {  zend\_class\_entry \*ce;  // 编译选项中的预加载开关  bool in\_preload = CG(compiler\_options) & ZEND\_COMPILE\_PRELOAD;  // 情况1，如果执行器没有激活 并且 不在预加载过程中  if (UNEXPECTED(!EG(active) && !in\_preload)) {  zend\_string \*lc\_name = zend\_string\_tolower(name); // 小写类名  ce = zend\_hash\_find\_ptr(CG(class\_table), lc\_name); // 编译时类表中查找此类  zend\_string\_release(lc\_name); // 释放类名  if (register\_unresolved && !ce) { // 如果有【注册延时加载】并且编译时没有找到此类  // 编译错误：要查找的类必须比scope更早注册  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  return ce; // 返回在编译时类表中找到的类  }  // 如果已到执行阶段或者在预加载过程中  // 用名称尝试加载类。附加要求：允许未链接，不自动加载  ce = zend\_lookup\_class\_ex( name, NULL, ZEND\_FETCH\_CLASS\_ALLOW\_UNLINKED |  ZEND\_FETCH\_CLASS\_NO\_AUTOLOAD);  if (!CG(in\_compilation) || in\_preload) { // 情况2，如果 不在编译阶段 或者 有预加载  if (ce) { // 如果查找成功  return ce; // 返回此类  }  if (register\_unresolved) { // 没有加载成功，需要添加到未完成列表  register\_unresolved\_class(name); // 添加到延时列表  }  } else { // 情况3，在编译阶段和执器也初始化过，没有预加载  if (ce && class\_visible(ce)) { // 有找到并且可见  return ce; // 返回此类  }  // 没找到 或 不可见  // 此类可能没有被注册，也可以是内置类，需要检验一下  if (zend\_string\_equals\_ci(scope->name, name)) { // 如果 当前域 与 要求的名称 相符  return scope; // 返回当前域。是内置类，最后返回  }  }  return NULL; // 没找到，返回null  } |

如上所示，函数的业务逻辑可分为三种情况，按使用到的条件归纳为业务逻辑分支表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **执行器是否激活** | **在编译阶段** | **在预加载阶段** | **处理逻辑** |
| 否 | 是 | 是 | 情况2 |
| 否 | 是 | 否 | 情况1 |
| 否 | 否 | 是 | 情况2 |
| 否 | 否 | 否 | 情况1 |
| 是 | 是 | 是 | 情况2 |
| 是 | 是 | 否 | 情况3 |
| 是 | 否 | 是 | 情况2 |
| 是 | 否 | 否 | 情况2 |

下面分别介绍3种情况的业务逻辑。

#### **情况1、执行器没有激活，并且不在预加载阶段**

这种情况下在编译时类表CG(class\_table)中查找类，如果查找成功，直接返回。

EG(active)是执行时全局变量，声名如下：

|  |
| --- |
| struct \_zend\_executor\_globals {  bool active; // 执行器已激活  } |

它在zend\_execute\_API.c中的init\_executor()函数中被更新成1。init\_executor()函数用于初始化并激活执行器。

编译选项中的ZEND\_COMPILE\_PRELOAD开关表示：正在预加载过程中引用编译器。

延时加载功能只允许在编译阶段和预加载阶段使用，如果不在这两个阶段调用延时加载，会产生error。

为方便理解，业务逻辑可归纳为4个种情况：

|  |  |  |
| --- | --- | --- |
|  | **查找类成功** | **查找类失败** |
| **开启延时加载** | 返回找到的类 | 返回NULL |
| **未开户延时加载** | 返回找到的类 | 报错 |

如果不满足第一种情况，会先调用zend\_lookup\_class\_ex()函数查找类，再根据结果进行操作。

#### **情况2、不在编译阶段，或者在预加载阶段**

在这种情况中，有三种子情况：

1）如果zend\_lookup\_class\_ex()函数查找到有效结果 ，直接返回查找结果；

2）如果zend\_lookup\_class\_ex()函数未查找到有效结果，并要求延时加载类，调用register\_unresolved\_class()函数进行处理；

3）其他情况，返回NULL。

#### **情况3、其他情况（执行器已激活，并且在编译阶段，并且不在预加载阶段）**

在这种情况中，有三种子情况：

1）如果查找到类，并且此类可见，返回查找到的类；

2）如果没查找到类或此类不可见，且此类的名称与当前域名称相同，返回当前域；

3）其他情况，返回NULL。

### zend\_lookup\_class\_ex()函数

zend\_lookup\_class\_ex()函数用于查找类，

|  |
| --- |
| // 查找并返回类。p1:类名，p2:key，p3:附加要求  ZEND\_API zend\_class\_entry \*zend\_lookup\_class\_ex(zend\_string \*name, zend\_string \*key, uint32\_t flags) { |

查找过程分为几步：

第一步，通过类名查找缓存中的类实例，如果找到到，直接应用；

第二步，生成小写类名；

第三步，尝试使用执行时列表中的类；

第四步，尝试使用自动加载。

#### **第1步，通过类名查找缓存中的类实例**

代码如下：

|  |
| --- |
| **// zend\_lookup\_class\_ex()函数的第一块代码，步骤1，查找缓存中的类实例**  zend\_class\_entry \*ce = NULL;  zval \*zv;  zend\_string \*lc\_name;  zend\_string \*autoload\_name;  uint32\_t ce\_cache = 0;  // 步骤1：检查ce缓存，如果有此名称的类缓存 ，并且类缓存有效  if (ZSTR\_HAS\_CE\_CACHE(name) && ZSTR\_VALID\_CE\_CACHE(name)) {  ce\_cache = GC\_REFCOUNT(name); // 取得类名中的缓存指针偏移量  ce = GET\_CE\_CACHE(ce\_cache); // 通过偏移量获取类实例  if (EXPECTED(ce)) { // 如果获取成功  return ce; // 返回类实例  }  } |

ZSTR\_HAS\_CE\_CACHE()宏程序和其他相关宏程序的介绍参见“地图指针与类缓存”章节。

#### **第2步，生成小写类名**

代码如下：

|  |
| --- |
| **// zend\_lookup\_class\_ex()函数的第二块代码，步骤2，生成小写类名**  if (key) { // 如果有传入key  lc\_name = key; // key就是小写名字  } else { // 没有传入key  if (name == NULL || !ZSTR\_LEN(name)) { // 如果没有name或name无效  return NULL; // 返回null  }  if (ZSTR\_VAL(name)[0] == '\\') { // 如果类名是完整路径  lc\_name = zend\_string\_alloc(ZSTR\_LEN(name) - 1, 0); // 创建zend\_string  // 把整个路径和名称转小写，不要开头的反斜线  zend\_str\_tolower\_copy(ZSTR\_VAL(lc\_name), ZSTR\_VAL(name) + 1,  ZSTR\_LEN(name) - 1);  } else { // 类名不是完整路径  lc\_name = zend\_string\_tolower(name); // 名称转小写  }  } |

#### **第3步，尝试使用执行时列表中的类**

代码如下：

|  |
| --- |
| **// zend\_lookup\_class\_ex()函数的第二块代码，步骤3，尝试使用执行时列表中的类**  zv = zend\_hash\_find(EG(class\_table), lc\_name); // 从运行时类列表中用小写名称查找类  if (zv) { // 如果找到元素  if (!key) { // 如果没有key  zend\_string\_release\_ex(lc\_name, 0); // 释放临时生成的小写类名  }  ce = (zend\_class\_entry\*)Z\_PTR\_P(zv); // 取出zval里的 类指针  // 如果类没有完成链接（没有 ZEND\_ACC\_LINKED 标记）  if (UNEXPECTED(!(ce->ce\_flags & ZEND\_ACC\_LINKED))) {  // 如果（有【允未链接的类】标记 或 有【允许新近链接】标记）并且查找到的记录有【新近链接】标记  if ((flags & ZEND\_FETCH\_CLASS\_ALLOW\_UNLINKED) ||  ((flags & ZEND\_FETCH\_CLASS\_ALLOW\_NEARLY\_LINKED) &&  (ce->ce\_flags & ZEND\_ACC\_NEARLY\_LINKED))) {  if (!CG(unlinked\_uses)) { // 如果【使用的未链接类】列表为空  ALLOC\_HASHTABLE(CG(unlinked\_uses)); // 分配内存创建哈希表  zend\_hash\_init(CG(unlinked\_uses), 0, NULL, NULL, 0); // 初始化哈希表  }  // 【使用的未链接类】列表中添加带索引号的空元素 ，索引号为类的指针转成整数  zend\_hash\_index\_add\_empty\_element(CG(unlinked\_uses),  (zend\_long)(zend\_uintptr\_t)ce);  return ce; // 反回类  }  // 类没有完成链接，又不允许使用未链接的类，  return NULL; // 返回null  }    // 如果有类缓存指针 并且 （当前不在编译中 或 类有不可转换标记）  if (ce\_cache &&  (!CG(in\_compilation) || (ce->ce\_flags & ZEND\_ACC\_IMMUTABLE))) {  SET\_CE\_CACHE(ce\_cache, ce); // 在让类缓存指针指向找到的类  }  return ce;  } |

加入CG(unlinked\_uses)未链接列表中的类，会在调用zend\_do\_link\_class()函数链接后从哈希表中删除。

#### **第4步，尝试使用自动加载**

代码如下：

|  |
| --- |
| **// zend\_lookup\_class\_ex()函数的第二块代码，步骤4，自动加载类**  // 编译器不可重复进入，保证只在执行时自动加载类  **// 情况1：如果不允许autoload 或 正在编译**  if ((flags & ZEND\_FETCH\_CLASS\_NO\_AUTOLOAD) || zend\_is\_compiling()) {  if (!key) { // 如果没有传入key，肯定用了小写名  zend\_string\_release\_ex(lc\_name, 0); // 释放小写类名  }  return NULL; // 查找类失败  }  if (!zend\_autoload) { **// 情况2：如果没有自动加载函数**  if (!key) { // 如果没有key，肯定用了小写名  zend\_string\_release\_ex(lc\_name, 0); // 释放小写类名  }  return NULL; // 查找失败  }  // 传给自动加载器之前先检验类名  **// 情况3：如果没有传入key并且类名没有缓存指针 并且 类名无效**  if (!key && !ZSTR\_HAS\_CE\_CACHE(name) && !zend\_is\_valid\_class\_name(name)) {  zend\_string\_release\_ex(lc\_name, 0); // 释放小写名  return NULL; // 查找失败  }  if (EG(in\_autoload) == NULL) { // 如果没有等待自动加载的类表  ALLOC\_HASHTABLE(EG(in\_autoload)); // 创建等待自动加载的类表  zend\_hash\_init(EG(in\_autoload), 8, NULL, NULL, 0); // 初始化等待自动加载的类表  }  **// 情况4：把类名作为键名，添加进等待自动加载的类表，如果类名已经在列表中**  if (zend\_hash\_add\_empty\_element(EG(in\_autoload), lc\_name) == NULL) {  if (!key) { // 如果没有传入key，肯定用了小写名  zend\_string\_release\_ex(lc\_name, 0); // 释放小写类名  }  return NULL; // 查找失败  }  **// 情况5：尝试自动加载**  if (ZSTR\_VAL(name)[0] == '\\') { // 如果类名是绝对路径  // 名称为第一个 \ 后面的部分  autoload\_name = zend\_string\_init(ZSTR\_VAL(name) + 1, ZSTR\_LEN(name) - 1, 0);  } else { // 不是绝对路径  autoload\_name = zend\_string\_copy(name); // 复制一份类名  }  zend\_exception\_save(); // 把异常保存到 previous 链里 ,清空当前异常  // 调用自动加载函数，spl\_perform\_autoload : ext\spl\php\_spl.c  ce = zend\_autoload(autoload\_name, lc\_name);  // 如果有当前异常，把上一个异常存储起来，如果没有，把上一个作为当前异常  zend\_exception\_restore();  zend\_string\_release\_ex(autoload\_name, 0); // 释放autoload用的类名  zend\_hash\_del(EG(in\_autoload), lc\_name); // 在EG(in\_autoload)哈希表里删除此小写类名  if (!key) { // 如果没有传入key，肯定用了小写名  zend\_string\_release\_ex(lc\_name, 0); // 释放小写名  }  if (ce) { // 如果自动加载成功  if (ce\_cache) { // 如果有类缓存指针  // 在 CG(map\_ptr\_base) 指针列表通过偏移量写入 类指针  SET\_CE\_CACHE(ce\_cache, ce);  }  }  return ce; // 返回类  } |

如上所示，第四步业务逻辑可分为5种情况：

情况1：如果不允许autoload 或 正在编译，返回NULL；

情况2：如果没有自动加载函数，返回NULL；

情况3：如果没有传入key并且类名没有缓存指针 并且 类名无效，返回NULL；

情况4：把类名作为键名，添加进等待自动加载的类表，如果类名已经在列表中，返回NULL；

情况5：尝试自动加载。

zend\_autoload()函数不在内核中，在spl扩展中，不是本章重点。

### zend\_fetch\_class\_by\_name()函数

zend\_fetch\_class\_by\_name()函数调用zend\_lookup\_class\_ex()函数来查找类，在查找类失败时会报错，它在Zend/zend\_execute\_API.c中定义，代码如下：

|  |
| --- |
| zend\_class\_entry \*zend\_fetch\_class\_by\_name(zend\_string \*class\_name, zend\_string \*key, uint32\_t fetch\_type) {  // 查找类。p1:类名，p2:key，p3:flags  zend\_class\_entry \*ce = zend\_lookup\_class\_ex(class\_name, key, fetch\_type);  if (!ce) { // 如果没找到  report\_class\_fetch\_error(class\_name, fetch\_type); // 报错：无法找到此类（或接口，trait）  return NULL;  }  return ce;  } |

report\_class\_fetch\_error()函数也在Zend/zend\_execute\_API.c中定义，逻辑简单，略过。

### register\_unresolved\_class()函数

register\_unresolved\_class()函数用于把类名添加到延时自动加载列表中，等待后续处理，代码如下：

|  |
| --- |
| static zend\_always\_inline void register\_unresolved\_class(zend\_string \*name) {  // 稍后会自动载入此类，并处理属性和方法的类型依赖  if (!CG(delayed\_autoloads)) { // 如果没有延时加载表  ALLOC\_HASHTABLE(CG(delayed\_autoloads)); // 创建延时加载哈希表  zend\_hash\_init(CG(delayed\_autoloads), 0, NULL, NULL, 0); // 初始化空哈希表  }  zend\_hash\_add\_empty\_element(CG(delayed\_autoloads), name); // 把此类名添加到列表里  } |

## 四）检查类的继承关系

检查类的继承关系主要用到instanceof\_function()函数和unlinked\_instanceof()函数。

### instanceof\_function()函数

instanceof\_function()函数用于检查一个类是否是另一个类（或接口）的子类，代码如下：

|  |
| --- |
| static zend\_always\_inline bool instanceof\_function(  const zend\_class\_entry \*instance\_ce, const zend\_class\_entry \*ce) {  return instance\_ce == ce || instanceof\_function\_slow(instance\_ce, ce);  } |

如上所示，除了要检查的两个类相同这种情况，其他情况调用instanceof\_function\_slow()函数进行处理。instanceof\_function\_slow()函数用于检查一个类是否是另一个类（或接口）的子类，代码如下：

|  |
| --- |
| // 检验继承或接口实现，p1:子类，p2:可能的父类  ZEND\_API bool instanceof\_function\_slow(const zend\_class\_entry \*instance\_ce,  const zend\_class\_entry \*ce) {  if (ce->ce\_flags & ZEND\_ACC\_INTERFACE) { // 情况1：如果ce是接口  uint32\_t i;  if (instance\_ce->num\_interfaces) { // 如果有实现接口数量  for (i = 0; i < instance\_ce->num\_interfaces; i++) { // 遍历每个实现的接口  if (instance\_ce->interfaces[i] == ce) { // 如果相符  return 1; // 返回是  }  }  }  return 0; // 返回否  } else { // 情况2：如果ce不是接口  while (1) { // 一定是在循环中返回  instance\_ce = instance\_ce->parent; // 追踪到父类  if (instance\_ce == ce) { // 如果找到  return 1; // 返回是  }  if (instance\_ce == NULL) { // 如果到最后也找不到  return 0; // 返回否  }  }  }  } |

如上所示，函数的业务逻辑比较简单，分为两种情况：

情况1：要检查的父类是接口，遍历子类实实现的所有接口，如果有这个接口，返回真，否则返回假；

情况2：要检查的父类不是接口，递归遍历子类的所有父类，如果匹配到要检查的类，返回真，否则返回假。

### unlinked\_instanceof()函数

unlinked\_instanceof()函数用于检查未链接的类的继承关系：

|  |
| --- |
| // 可以安全用在未链接的类上的 Instanceof ，p1:要检查的子类，p2:要检查的父类  static bool unlinked\_instanceof(zend\_class\_entry \*ce1, zend\_class\_entry \*ce2) {  if (ce1 == ce2) { // 情况1：两个类指针相同  return 1; // 返回是  }  if (ce1->ce\_flags & ZEND\_ACC\_LINKED) { // 情况2：要检查的子类链接过  return instanceof\_function(ce1, ce2); // 检验继承或接口实现，单向 ： zend\_operators.h  }  if (ce1->parent) { // 情况3：要检查的子类没有链接过，并且有父类  zend\_class\_entry \*parent\_ce;  if (ce1->ce\_flags & ZEND\_ACC\_RESOLVED\_PARENT) { // 要检查的子类处理过继承  parent\_ce = ce1->parent; // 取出父类  } else { // 要检查的子类没有处理过继承  // 用父类名查找父类，允许未链接，不自动加载  parent\_ce = zend\_lookup\_class\_ex(ce1->parent\_name, NULL,  ZEND\_FETCH\_CLASS\_ALLOW\_UNLINKED | ZEND\_FETCH\_CLASS\_NO\_AUTOLOAD);  }  // 只检查父类链是不够的，当父接口还没有复制过来时，需要一个完整的递归 检查  if (parent\_ce && unlinked\_instanceof(parent\_ce, ce2)) { // 如果父类继承自第二个类  return 1; // 返回真  }  }  if (ce1->num\_interfaces) { // 情况4：如果要检查的子类有实现接口  uint32\_t i;  if (ce1->ce\_flags & ZEND\_ACC\_RESOLVED\_INTERFACES) { // 如果子类已完成接口复制  // 不同于instanceof\_function()，这里需要进行递归检查，因为接口可能没有完成复制  for (i = 0; i < ce1->num\_interfaces; i++) { // 遍历所有接口  if (unlinked\_instanceof(ce1->interfaces[i], ce2)) { // 如果接口继承自要求的父类  return 1; // 返回真  }  }  } else { // 子类未完成接口复制  for (i = 0; i < ce1->num\_interfaces; i++) { // 遍历所有接口  // 查找接口。允许未链接，不自动加载  zend\_class\_entry \*ce = zend\_lookup\_class\_ex(  ce1->interface\_names[i].name, ce1->interface\_names[i].lc\_name,  ZEND\_FETCH\_CLASS\_ALLOW\_UNLINKED | ZEND\_FETCH\_CLASS\_NO\_AUTOLOAD);  // 如果类实现自己，避免递归  // 如果接口有效，并且接口不是第一个类，并且接口继承自要求的父类  if (ce && ce != ce1 && unlinked\_instanceof(ce, ce2)) {  return 1; // 返回真  }  }  }  }  return 0; // 其他情况：返回假  } |

如上所示，业务逻辑分为几种情况：

情况1：要检查的父类和子类是同一个类，返回真；

情况2：要检查的子类已经链接过，调用instanceof\_function()函数进行检查，检查结果为真时返回真；

情况3：要检查的子类没有链接过，找到它的父类，并递归调用unlinked\_instanceof()函数来进行检查，检查结果为真时返回真；

情况4：要检查的子类有实现接口，分成两种情况进行检查：a）如果要检查的子类已经复制过接口信息，遍历每个接口，并调用unlinked\_instanceof()函数进行检查，检查结果为真时返回真；b）如果子类没复制过接口信息，遍历所有接口，调用zend\_lookup\_class\_ex()函数加载接口，并调用unlinked\_instanceof()函数进行检查，检查结果为真时返回真；

情况5：如果以上5种情况都匹配失败，返回假。

# 三、类的继承

## 一）继承的实现

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

|  |
| --- |
| static void zend\_do\_inheritance(zend\_class\_entry \*ce, zend\_class\_entry \*parent\_ce) {  zend\_do\_inheritance\_ex(ce, parent\_ce, 0);  } |

它调用zend\_do\_inheritance\_ex()函数来实现，此函数的业务逻辑比较复杂，下面分块展开介绍。

### 第1步：基本校验

代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第1块代码**  // 参数p1:子类指针，p2:父类指针, p3:是否已经通过继承检查（检查维果为继承成功）  ZEND\_API void zend\_do\_inheritance\_ex(zend\_class\_entry \*ce, zend\_class\_entry \*parent\_ce, bool checked) {  zend\_property\_info \*property\_info;  zend\_function \*func;  zend\_string \*key;    // 步骤1，先检验是否可继承  if (UNEXPECTED(ce->ce\_flags & ZEND\_ACC\_INTERFACE)) { // 如果子类是接口  if (UNEXPECTED(!(parent\_ce->ce\_flags & ZEND\_ACC\_INTERFACE))) { // 父类不是接口  zend\_error\_noreturn(E\_COMPILE\_ERROR,...); // 编译异常：接口只能继承接口  }  // 如果子类不是接口，父类是接口、trait、或有final标记  } else if (UNEXPECTED(parent\_ce->ce\_flags & (ZEND\_ACC\_INTERFACE|ZEND\_ACC\_TRAIT  |ZEND\_ACC\_FINAL))) {  if (parent\_ce->ce\_flags & ZEND\_ACC\_FINAL) { // 如果类有final标记  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...); // 报错：不可以继承final类  }  if ((parent\_ce->ce\_flags & ZEND\_ACC\_INTERFACE) || (parent\_ce->ce\_flags & ZEND\_ACC\_TRAIT)) { // 如果父类是接口 或 trait  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...); // 报错：不可以继承trait 或接口  }  }  // 如果子类和父类一个有readonly一个没有  if (UNEXPECTED((ce->ce\_flags & ZEND\_ACC\_READONLY\_CLASS) != (parent\_ce->ce\_flags & ZEND\_ACC\_READONLY\_CLASS))) {  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...); // 报错父类和子类的【只读】标记必须一致  }  if (ce->parent\_name) { // 如果类已经有父类名  zend\_string\_release\_ex(ce->parent\_name, 0); // 释放掉旧父类名  }  ce->parent = parent\_ce; // 关联父类  ce->ce\_flags |= ZEND\_ACC\_RESOLVED\_PARENT; // 添加标记【已关联父类】 |

这一步验证了以下几种错误情况：

1）子类是接口，父类不是接口；

2）父类是接口、trait、或有final标记；

3）子类和父类一个有readonly一个没有readonly关键字。

### 第2步：继承默认（default）属性（成员变量）

类的默认属性表ce->default\_properties\_table是一串val实例。继承操作代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第2块代码**  if (parent\_ce->default\_properties\_count) { // 如果父类有默认属性  zval \*src, \*dst, \*end;  if (ce->default\_properties\_count) { // 如果子类有默认属性  // 创建属性表，大小是两个类的默认属性数量和，此类是内置类时用malloc()函数分配  zval \*table = pemalloc(sizeof(zval) \* (ce->default\_properties\_count + parent\_ce->default\_properties\_count), ce->type == ZEND\_INTERNAL\_CLASS);  // 子类原属性表的结尾  src = ce->default\_properties\_table + ce->default\_properties\_count;  end = table + parent\_ce->default\_properties\_count; // 新属性表父属性的结尾  dst = end + ce->default\_properties\_count; // 子类新属性表结尾  ce->default\_properties\_table = table; // 应用新默认属性表（旧表解除关联）  do { // 先复制子类属性(后半段，倒序）：从新属性表结尾 到 新属性表，父属性结尾  dst--; // 指针左移  src--;  ZVAL\_COPY\_VALUE\_PROP(dst, src); // 逐个把子类属性复制到新表里  } while (dst != end);  pefree(src, ce->type == ZEND\_INTERNAL\_CLASS); // 释放原属性表（已解除关联）  end = ce->default\_properties\_table; // 结束位置为新属性表的开头  } else { // 如果子类没有默认属性  // 按父类属性表大小分配内存，继承内置类时用malloc()函数分配  end = pemalloc(sizeof(zval) \* parent\_ce->default\_properties\_count, ce->type == ZEND\_INTERNAL\_CLASS);  dst = end + parent\_ce->default\_properties\_count; // 表的结束位置  ce->default\_properties\_table = end; // 新属性表关联到类  }  // 默认属性表中，父属性在前，子类属性在后，default\_properties\_table指针指向父属性结尾位置，也是子类自有属性的开头  // 找到父类属性表的结尾位置  src = parent\_ce->default\_properties\_table + parent\_ce->default\_properties\_count;  if (UNEXPECTED(parent\_ce->type != ce->type)) { // 类型不同，是用户类继承内置类  do { // 同时倒序遍历新旧两个表，复制父类属性  dst--; // 两个同时左移  src--;  // 自动判断 ：持久对象增加引用次数，非持久对象创建副本。并复制 .u2.extra。  ZVAL\_COPY\_OR\_DUP\_PROP(dst, src); // 复制每个类属性 给子类  if (Z\_OPT\_TYPE\_P(dst) == IS\_CONSTANT\_AST) { // 如果此属性来自常量表达式  // 清除 【常量已更新】 标记  ce->ce\_flags &= ~ZEND\_ACC\_CONSTANTS\_UPDATED;  // 添加 【包含表达式属性】标记  ce->ce\_flags |= ZEND\_ACC\_HAS\_AST\_PROPERTIES;  }  continue; // 这个continue 会跳到while判断  } while (dst != end);  } else { // 父类和子类的类型相同，都是用户类  do {  dst--; // 两个同时左移  src--;  ZVAL\_COPY\_PROP(dst, src); // 复制每个类属性 给子类  if (Z\_OPT\_TYPE\_P(dst) == IS\_CONSTANT\_AST) { // 如果此属性来自 常量表达式  // 清除 【常量已更新】 标记  ce->ce\_flags &= ~ZEND\_ACC\_CONSTANTS\_UPDATED;  // 添加 【包含表达式属性】标记  ce->ce\_flags |= ZEND\_ACC\_HAS\_AST\_PROPERTIES;  }  continue; // 这个continue 会跳到while判断  } while (dst != end);  }  // 新类的属性数里，加上父类的属性数  ce->default\_properties\_count += parent\_ce->default\_properties\_count;  } |

如上所示，在复制属性时，分为3种情况：

**1）在复制本类属性到新表中时，用到ZVAL\_COPY\_VALUE\_PROP()宏程序。**

ZVAL\_COPY\_VALUE\_PROP()宏程序是最简单直接的复制zval实例，定义如下：

|  |
| --- |
| #define ZVAL\_COPY\_VALUE\_PROP(z, v) do { \*(z) = \*(v); } while (0) |

**2）复制父类属性时，如果父类是内置类，调用ZVAL\_COPY\_OR\_DUP\_PROP()宏程序。**

ZVAL\_COPY\_OR\_DUP\_PROP()宏程序定义如下：

|  |
| --- |
| #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); } while (0)  #define ZVAL\_COPY\_OR\_DUP(z, v) \  do { \  zval \*\_z1 = (z); \  const zval \*\_z2 = (v); \  zend\_refcounted \*\_gc = Z\_COUNTED\_P(\_z2); /\* 取得实例指针 \*/ \  uint32\_t \_t = Z\_TYPE\_INFO\_P(\_z2); /\* 复制类型信息 \*/ \  ZVAL\_COPY\_VALUE\_EX(\_z1, \_z2, \_gc, \_t); /\* 先复制zval \*/ \  if (Z\_TYPE\_INFO\_REFCOUNTED(\_t)) { /\* 如果对象可计数 \*/ \  /\* 如果实例有 GC\_PERSISTENT 标记 或 者是对象 \*/ \  if (EXPECTED(!(GC\_FLAGS(\_gc) & GC\_PERSISTENT) || GC\_TYPE(\_gc) == IS\_OBJECT)){ \  GC\_ADDREF(\_gc); /\* 增加引用次数 \*/ \  } else { /\* 没有 GC\_PERSISTENT 标记 也不是对象 \*/ \  zval\_copy\_ctor\_func(\_z1); /\* 创建副本，并把zval关联到副本 \*/ \  } \  } \  } while (0) |

如上所示，这个宏程序会自动决定是要创建副本，还是在原实例上增加引用次数。

zval\_copy\_ctor\_func()函数定义如下：

|  |
| --- |
| ZEND\_API void ZEND\_FASTCALL zval\_copy\_ctor\_func(zval \*zvalue) {  if (EXPECTED(Z\_TYPE\_P(zvalue) == IS\_ARRAY)) { // 如果类型是数组  ZVAL\_ARR(zvalue, zend\_array\_dup(Z\_ARRVAL\_P(zvalue))); // 创建副本，并关联到zval  } else if (EXPECTED(Z\_TYPE\_P(zvalue) == IS\_STRING)) { // 如果类型是字串  ZEND\_ASSERT(!ZSTR\_IS\_INTERNED(Z\_STR\_P(zvalue))); // 不可以是内置字串  ZVAL\_NEW\_STR(zvalue, zend\_string\_dup(Z\_STR\_P(zvalue), 0)); // 创建副本，并关联到zval  } else { // 其他类型  ZEND\_UNREACHABLE(); // 其他类型不可以调用这个函数  }  } |

zend\_array\_dup()函数的相关介绍可参见“数组篇”，zend\_string\_dup()函数的相关介绍可参见“类型篇”。

**3）如果父类是用户定义的类，调用ZVAL\_COPY\_PROP()宏程序。**

ZVAL\_COPY\_PROP()宏程序定义如下：

|  |
| --- |
| #define ZVAL\_COPY\_PROP(z, v) \  do { ZVAL\_COPY(z, v); Z\_PROP\_FLAG\_P(z) = Z\_PROP\_FLAG\_P(v); } while (0) |

ZVAL\_COPY()和ZVAL\_COPY\_VALUE\_EX()宏程序的介绍参见“类型篇”。

### 第3步：继承静态（static）属性（成员变量）

代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第3块代码**  if (parent\_ce->default\_static\_members\_count) { // 如果父类有静态属性  zval \*src, \*dst, \*end;  if (ce->default\_static\_members\_count) { // 如果子类有静态属性  // 创建属性表表：子类静态成员数量+父类静态成员数量 ，内置类用malloc()函数分配  zval \*table = pemalloc(sizeof(zval) \* (ce->default\_static\_members\_count + parent\_ce->default\_static\_members\_count), ce->type == ZEND\_INTERNAL\_CLASS);  // 旧表结尾处  src = ce->default\_static\_members\_table + ce->default\_static\_members\_count;  end = table + parent\_ce->default\_static\_members\_count; // 新表父类属性结尾处  dst = end + ce->default\_static\_members\_count; // 新表子类属性结尾处（后半段）  ce->default\_static\_members\_table = table; // 使用新表（旧表解除关联）  do { // 同时倒序遍历新旧两个表，复制子类属性  dst--;  src--;  ZVAL\_COPY\_VALUE(dst, src); // 旧表元素 复制给新表  } while (dst != end);  pefree(src, ce->type == ZEND\_INTERNAL\_CLASS); // 释放旧表  end = ce->default\_static\_members\_table; // 新表的开始位置,也是倒序遍历结束位置  } else { // 子类没有静态属性  // 分配内存创建 zval表：父类静态属性数量，内置类永久分配  end = pemalloc(sizeof(zval) \* parent\_ce->default\_static\_members\_count, ce->type == ZEND\_INTERNAL\_CLASS);  dst = end + parent\_ce->default\_static\_members\_count; // 新表父元素结束位置（前半段）  ce->default\_static\_members\_table = end; // 使用新表（旧表解除关联）  }  // 子类静态属性数，增加父类静态属性数  src = parent\_ce->default\_static\_members\_table  + parent\_ce->default\_static\_members\_count;  do { // 同时倒序遍历新旧两个表，复制父类属性  dst--;  src--;  if (Z\_TYPE\_P(src) == IS\_INDIRECT) { // 如果是间接引用  ZVAL\_INDIRECT(dst, Z\_INDIRECT\_P(src)); // 复制间接引用  } else { // 不是间接引用  ZVAL\_INDIRECT(dst, src); **// 使用间接引用关联到新表**  }  if (Z\_TYPE\_P(Z\_INDIRECT\_P(dst)) == IS\_CONSTANT\_AST) { // 如果是常量表达式  ce->ce\_flags &= ~ZEND\_ACC\_CONSTANTS\_UPDATED; // 清除【常量已更新】  ce->ce\_flags |= ZEND\_ACC\_HAS\_AST\_STATICS; // 添加 【包含静态表达式】标记  }  } while (dst != end);  // 更新 子类的静态属性数  ce->default\_static\_members\_count += parent\_ce->default\_static\_members\_count;  if (!ZEND\_MAP\_PTR(ce->static\_members\_table)) { // 如果没有静态属性指针列表  // 如果是内置类 并且 所属模块为持久模块  if (ce->type == ZEND\_INTERNAL\_CLASS &&  ce->info.internal.module->type == MODULE\_PERSISTENT) {  ZEND\_MAP\_PTR\_NEW(ce->static\_members\_table); // 创建指针列表  }  }  } |

继承静态属性的业务逻辑与静态默认属性相似，Z\_INDIRECT\_P()宏程序的相关介绍参见“类型篇”。

ZEND\_MAP\_PTR\_NEW() 宏程序相关说明参见“地图指针与类缓存”章节。

### 第4步：修正属性信息的偏移量

代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第4块代码**  // 遍历子类属性信息表（属性信息表在编译时创建）  ZEND\_HASH\_MAP\_FOREACH\_PTR(&ce->properties\_info, property\_info) {  if (property\_info->ce == ce) { // 如果属性信息属于子类  if (property\_info->flags & ZEND\_ACC\_STATIC) { // 如果属性信息有 static 标记  **// 计算静态属性信息的偏移量，增加父类的静态属性数量（使用成员个数）**  property\_info->offset += parent\_ce->default\_static\_members\_count;  } else { // 没有static标记  **// 计算属性信息的偏移量，增加父类的属性数量\*zval大小（使用Bytes数量）**  property\_info->offset += parent\_ce->default\_properties\_count \* sizeof(zval);  }  }  } ZEND\_HASH\_FOREACH\_END(); |

如上所示，修正属性信息的偏移量，是要把父类属性的数量计算进来，找到从属性值列表开头到这个属性值的偏移量。对于静态属性和普通属性有不同的处理，静态属性值的偏移量是以成员个数为单位，也就是以zval实例的大小为单位进行偏移；普通属性值的偏移量要转成Bytes数量存储。

### 第5步：继承属性信息

代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第5块代码**  if (zend\_hash\_num\_elements(&parent\_ce->properties\_info)) { // 如果父类有属性信息  // 调整属性信息表的大小，增加空间存放父类属性信息  zend\_hash\_extend(&ce->properties\_info,  zend\_hash\_num\_elements(&ce->properties\_info) +  zend\_hash\_num\_elements(&parent\_ce->properties\_info), 0);  // 遍历父类属性信息表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&parent\_ce->properties\_info, key, property\_info) {  do\_inherit\_property(property\_info, key, ce); // 继承属性信息  } ZEND\_HASH\_FOREACH\_END();  } |

#### **do\_inherit\_property()函数**

do\_inherit\_property()函数用于继承单个属性信息，代码如下：

|  |
| --- |
| static void do\_inherit\_property(zend\_property\_info \*parent\_info, zend\_string \*key, zend\_class\_entry \*ce) {  // 子类属性信息表中查找此属性信息  zval \*child = zend\_hash\_find\_known\_hash(&ce->properties\_info, key);  zend\_property\_info \*child\_info;  if (UNEXPECTED(child)) { **// 如果子类已经有同名属性**  child\_info = Z\_PTR\_P(child); // 取出 属性信息  // 如果父属性有【私有】 或 【更改访问权限】 标记  if (parent\_info->flags & (ZEND\_ACC\_PRIVATE|ZEND\_ACC\_CHANGED)) {  child\_info->flags |= ZEND\_ACC\_CHANGED; // 子属性添加 【更改访问权限】标记  }  if (!(parent\_info->flags & ZEND\_ACC\_PRIVATE)) { // 如果父属性信息没有【私有】标记  // 如果父子属性信息的 static标记不同  if (UNEXPECTED((parent\_info->flags & ZEND\_ACC\_STATIC) !=  (child\_info->flags & ZEND\_ACC\_STATIC))) {  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...); // 报错：不可以更改static标记  }  // 如果父子属性信息的 readonly标记不同  if (UNEXPECTED((child\_info->flags & ZEND\_ACC\_READONLY) !=  (parent\_info->flags & ZEND\_ACC\_READONLY))) {  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...); // 报错：不可以更改readonly标记  }  // 如果子类属性信息的 可见性 大于父类属性信息  if (UNEXPECTED((child\_info->flags & ZEND\_ACC\_PPP\_MASK) >  (parent\_info->flags & ZEND\_ACC\_PPP\_MASK))) {  // 报错：子类属性信息的 可见性 不可以大于父类属性信息  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  // 可见性没问题， 如果子类的属性信息没有static标记，要覆盖父属性值  } else if ((child\_info->flags & ZEND\_ACC\_STATIC) == 0) {  // 把偏移Bytes数量转换成 zval实例 序号  int parent\_num = OBJ\_PROP\_TO\_NUM(parent\_info->offset);  int child\_num = OBJ\_PROP\_TO\_NUM(child\_info->offset);    // 不要把默认属性信信息保存在gc中（它们会被opcache释放掉）  // 销毁默认属性表中的父属性信息，不加入垃圾回收队列  zval\_ptr\_dtor\_nogc(&(ce->default\_properties\_table[parent\_num]));  // 把子属性信息移动到这个位置，直接复制zval实例（子属性原来的位置没用了）  ce->default\_properties\_table[parent\_num] =  ce->default\_properties\_table[child\_num];  ZVAL\_UNDEF(&ce->default\_properties\_table[child\_num]); //清空继承的属性值  child\_info->offset = parent\_info->offset; // 用原来的偏移量  }  // 如果父属性有【私有】标记，上面的static，readonly，可见性验证就全免了    if (UNEXPECTED(ZEND\_TYPE\_IS\_SET(parent\_info->type))) { // 如果父类属性有类型  // 检查属性类型是否兼容（类型双向可继承）  inheritance\_status status=property\_types\_compatible(parent\_info, child\_info);  if (status == INHERITANCE\_ERROR) { // 如果检验结果为：继承出错  // 报错：属性信息不兼容  emit\_incompatible\_property\_error(child\_info, parent\_info);  }  if (status == INHERITANCE\_UNRESOLVED) { // 如果检验结果为：继承未完成  // 添加 兼容性责任。p1:所属类，p2:子属性信息，p3:父属性信息  add\_property\_compatibility\_obligation(ce, child\_info, parent\_info);  }  // 如果子属性有类型 并且 父属性无类型  } else if (UNEXPECTED(ZEND\_TYPE\_IS\_SET(child\_info->type)  && !ZEND\_TYPE\_IS\_SET(parent\_info->type))) {  zend\_error\_noreturn(E\_COMPILE\_ERROR,...); // 报错：子类属性类型要和父类一致  }  }  } else { **// 如果子类没有此属性信息**  \_zend\_hash\_append\_ptr(&ce->properties\_info, key, parent\_info); // 添加属性信息  }  } |

如上所示，业务逻辑分为两种情况：

一、如果子类没有同名属性，直接使用\_zend\_hash\_append\_ptr()函数把属性信息插入到属性信息表ce->properties\_info中；

二、如果子类有同名属性，需要进行几步处理：

1）如果父属性信息有ZEND\_ACC\_PRIVATE（私有）或ZEND\_ACC\_CHANGED（更改访问权限）标记，需要给子属性添加ZEND\_ACC\_CHANGED标记；

2）如果父属性信息没有ZEND\_ACC\_PRIVATE（私有）标记，还要进行以下几项验证：

a）如果父子属性信息的 static标记不同，报错；

b）如果父子属性信息的 readonly标记不同，报错；

c）如果子属性信息的可见性大于父类属性信息的可见性，报错；

d）如果你属性信息有指定类型，需要验证父子属性信息的类型是否兼容，不兼容是时报错；如果继承尚未完成，需要添加“兼容性责任”；

e）如果子属性有指定类型并且父属性无指定类型，报错。

#### **property\_types\_compatible()函数**

property\_types\_compatible()函数用来检验继承时属性类型是否兼容，这里用到inheritance\_status类型，它是一个枚举类型实例的别名，类型定义如下：

|  |
| --- |
| typedef enum {  INHERITANCE\_UNRESOLVED = -1, // 继承未完成，在运行时（runtime）会当成错误来处理  INHERITANCE\_ERROR = 0, // 继承出错  INHERITANCE\_WARNING = 1, // 警告  INHERITANCE\_SUCCESS = 2, // 继承成功  } inheritance\_status; |

property\_types\_compatible()函数代码如下：

|  |
| --- |
| inheritance\_status property\_types\_compatible(  const zend\_property\_info \*parent\_info, const zend\_property\_info \*child\_info) {  // 如果两个信息的纯类型码相同 并且 类型名称（zend\_string指针）相同  if (ZEND\_TYPE\_PURE\_MASK(parent\_info->type) ==  ZEND\_TYPE\_PURE\_MASK(child\_info->type) &&  ZEND\_TYPE\_NAME(parent\_info->type) == ZEND\_TYPE\_NAME(child\_info->type)) {  return INHERITANCE\_SUCCESS; // 返回：继承成功  }    // 如果父类和子类 一个有指定类型，一个没有（父类和子类必须都指定类型）  if (ZEND\_TYPE\_IS\_SET(parent\_info->type) != ZEND\_TYPE\_IS\_SET(child\_info->type)) {  return INHERITANCE\_ERROR; // 返回：继承出错  }  // 进行一个双向类型兼容性检查  // 检查 fe\_type 是否可以作为 proto\_scope 的子类型，两个都可以是复杂类型  inheritance\_status status1 = zend\_perform\_covariant\_type\_check(  child\_info->ce, child\_info->type, parent\_info->ce, parent\_info->type);  // 检查 proto\_scope 是否可以作为 fe\_type 的子类型，两个都可以是复杂类型  inheritance\_status status2 = zend\_perform\_covariant\_type\_check(  parent\_info->ce, parent\_info->type, child\_info->ce, child\_info->type);    // 如果双向都可以继承，返回继承成功  if (status1 == INHERITANCE\_SUCCESS && status2 == INHERITANCE\_SUCCESS) {  return INHERITANCE\_SUCCESS;  }  // 如果有一个失败，返回失败  if (status1 == INHERITANCE\_ERROR || status2 == INHERITANCE\_ERROR) {  return INHERITANCE\_ERROR; // 返回：继承出错  }  return INHERITANCE\_UNRESOLVED; // 其他情况，返回：继承未完成  } |

如上所示，继承类属性时进行的兼容性检查分为以下步骤：

1）检测两个信息的纯类型码是否相同以及类型名称（zend\_string指针）是否相同，如果都相同，直接返回：继承成功（INHERITANCE\_SUCCESS）；

2）如果父类属性信息和子类属性信息一个包含有效的基础类型，一个没有，返回：继承出错（INHERITANCE\_ERROR）；

ZEND\_TYPE\_IS\_SET()宏程序的介绍参见zend\_type章节。

3）调用zend\_perform\_covariant\_type\_check()函数对父类属性信息和子类属性信息进行双向的多兼容性检查，根据检查结果来返回：

a）如果两个检查结果都是继承成功，返回继承成功（INHERITANCE\_SUCCESS）；

b）如果两个检查结果有一个是继承失败，返回继承出错（INHERITANCE\_ERROR）；

c）其他情况，返回继承未完成（INHERITANCE\_UNRESOLVED）。

zend\_perform\_covariant\_type\_check()函数的业务逻辑比较复杂，详细介绍参见“继承中的类型兼容性检查”章节。

add\_property\_compatibility\_obligation()函数用于添加兼容性责任，详细介绍参见“延时变异责任”章节。

### 第6步：继承常量

代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第6块代码**  if (zend\_hash\_num\_elements(&parent\_ce->constants\_table)) { // 如果父类有常量  zend\_class\_constant \*c;  // 先扩展子类常量表大小，准备把父类常量表添加进来  zend\_hash\_extend(&ce->constants\_table,  zend\_hash\_num\_elements(&ce->constants\_table) +  zend\_hash\_num\_elements(&parent\_ce->constants\_table), 0);    // 遍历父类常量表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&parent\_ce->constants\_table, key, c) {  do\_inherit\_class\_constant(key, c, ce); // 逐个继承常量  } ZEND\_HASH\_FOREACH\_END();  } |

#### **do\_inherit\_class\_constant()函数**

do\_inherit\_class\_constant()函数用于继承父类的常量，代码如下：

|  |
| --- |
| static void do\_inherit\_class\_constant(zend\_string \*name, zend\_class\_constant \*parent\_const, zend\_class\_entry \*ce) {  // 从子类常量表中用名称查找常量  zval \*zv = zend\_hash\_find\_known\_hash(&ce->constants\_table, name);  zend\_class\_constant \*c;  if (zv != NULL) { **// 情况1：如果子类中有同名常量**  c = (zend\_class\_constant\*)Z\_PTR\_P(zv); // 取出常量指针  // 如果子类常量的 可见性 大于父类  if (UNEXPECTED((ZEND\_CLASS\_CONST\_FLAGS(c) & ZEND\_ACC\_PPP\_MASK) > (ZEND\_CLASS\_CONST\_FLAGS(parent\_const) & ZEND\_ACC\_PPP\_MASK))) {  // 报错：子类常量的可见性必须小于父类  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  // 如果父类常量有final标记  if (UNEXPECTED((ZEND\_CLASS\_CONST\_FLAGS(parent\_const) & ZEND\_ACC\_FINAL))) {  zend\_error\_noreturn(E\_COMPILE\_ERROR,...); // 报错：不可以覆盖final常量  }  **// 情况2：如果子类没有此常量，并且父类常量没有 private 标记**  } else if (!(ZEND\_CLASS\_CONST\_FLAGS(parent\_const) & ZEND\_ACC\_PRIVATE)) {  // 如果父类的常量类型是【常量表达式】  if (Z\_TYPE(parent\_const->value) == IS\_CONSTANT\_AST) {  // 去掉【已更新常量】标记  ce->ce\_flags &= ~ZEND\_ACC\_CONSTANTS\_UPDATED;  ce->ce\_flags |= ZEND\_ACC\_HAS\_AST\_CONSTANTS; // 添加【有表达式常量】标记  if (ce->parent->ce\_flags & ZEND\_ACC\_IMMUTABLE) { // 如果此类不可更改  // arena 分配新常量  c = zend\_arena\_alloc(&CG(arena), sizeof(zend\_class\_constant));  memcpy(c, parent\_const, sizeof(zend\_class\_constant)); // 复制父类常量  parent\_const = c; // 使用副本进行后继操作  // CONST\_OWNED标记表示 ：此常量应和类一起销毁  Z\_CONSTANT\_FLAGS(c->value) |= CONST\_OWNED;  }  }  if (ce->type & ZEND\_INTERNAL\_CLASS) { // 如果是内置类  c = pemalloc(sizeof(zend\_class\_constant), 1); // 调用malloc()函数分配内存创建常量  memcpy(c, parent\_const, sizeof(zend\_class\_constant)); // 复制父类常量  parent\_const = c; // 使用副本进行后继操作  }  // 添加进常量表里  \_zend\_hash\_append\_ptr(&ce->constants\_table, name, parent\_const);  }  **// 情况3，无需处理的情况：如果子类没有此常量，并且父类常量有 private 标记，不继承它。**  } |

如上所示，函数的业务逻辑比较简单，常量的继承包含3种情况：

1）子类中找到此常量：验证可见性和final标记，发现问题则报错；

2）如果子类中没有此常量，父类中有此常量并且没有private标记：尝试复制父类常量。如果父常量类型是表达式并且类有不可更改（ZEND\_ACC\_IMMUTABLE）标记或子类是内置类，需要给常量创建副本，并应用副本；

3）父类中有此常量，并且有private标记：什么也不做，不继承它。

### 第7步：继承成员方法

代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第7块代码**  if (zend\_hash\_num\_elements(&parent\_ce->function\_table)) { // 遍历父类方法表  // 扩展子类的方法表准备把父类的方法都添加进来  zend\_hash\_extend(&ce->function\_table,  zend\_hash\_num\_elements(&ce->function\_table) +  zend\_hash\_num\_elements(&parent\_ce->function\_table), 0);  if (checked) { // 如果需要检查  // 遍历父类方法表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&parent\_ce->function\_table, key, func) {  do\_inherit\_method(key, func, ce, 0, 1); // 继承方法，需要检查  } ZEND\_HASH\_FOREACH\_END();  } else {  // 遍历父类方法表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&parent\_ce->function\_table, key, func) {  do\_inherit\_method(key, func, ce, 0, 0); // 继承方法，不需要检查  } ZEND\_HASH\_FOREACH\_END();  }  } |

#### **do\_inherit\_method()函数**

do\_inherit\_method()函数用于继承父类的方法，代码如下：

|  |
| --- |
| // p1:方法名，p2:父类，p3:子类，p4:是从从接口继承，p5:是否检查  static void do\_inherit\_method(zend\_string \*key, zend\_function \*parent, zend\_class\_entry \*ce, bool is\_interface, bool checked) {  // 在子类中，按方法名取出方法  zval \*child = zend\_hash\_find\_known\_hash(&ce->function\_table, key);  if (child) { **// 如果子类有同名方法**  zend\_function \*func = (zend\_function\*)Z\_PTR\_P(child); // 取出此方法  // 如果是接口 并且 方法与父方法一致，说明已经继承过了  if (is\_interface && UNEXPECTED(func == parent)) {  return; // 完成，返回  }  if (checked) { // 如果子方法与父方法不同，并且需要检查  // 对方法进行继承检查 。p1:子方法，p2:子类，p3:父方法，p4:父类，p5:?,p6:?,p7:检查可见性，p8,只检查，p9:检查  do\_inheritance\_check\_on\_method\_ex(  func, func->common.scope, parent, parent->common.scope, ce, child,  /\* check\_visibility \*/ 1, 0, checked);  } else { // 如果不需要检查  // 对方法进行继承检查 。p1:子方法，p2:子类，p3:父方法，p4:父类，p5:?,p6:?,p7:检查可见性  do\_inheritance\_check\_on\_method(  func, func->common.scope, parent, parent->common.scope, ce, child,  /\* check\_visibility \*/ 1);  }    } else { **// 如果子类没有同名方法**  // 如果是接口 或 抽象类  if (is\_interface || (parent->common.fn\_flags & (ZEND\_ACC\_ABSTRACT))) {  // 类添加标记：隐式抽象类标记  ce->ce\_flags |= ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS;  }  parent = zend\_duplicate\_function(parent, ce); // 为函数创建副本  if (!is\_interface) { // 如果不是接口  // 快速给哈希表添加键值对，值为指针，不查检已有元素  \_zend\_hash\_append\_ptr(&ce->function\_table, key, parent);  } else { // 如果是接口  // 向哈希表中直接添加指针元素，不检查已有元素，有什么不同？  zend\_hash\_add\_new\_ptr(&ce->function\_table, key, parent);  }  }  } |

如上所示，代码比较简单，业务逻辑分为两种情况：

1）如果子类中有同名成员方法，检查子类成员方法是否与父类的兼容，检查操作的过程比较复杂，需要调用do\_inheritance\_check\_on\_method\_ex()函数，详细介绍参见后续“成员方法的继承检查”章节。

2）如果子类没有同名方法，调用zend\_duplicate\_function()函数为父类的成员方法创建副本，并添加到子类的成员方法表中。

#### **zend\_duplicate\_function()函数**

zend\_duplicate\_function()函数用于为类的成员方法创建副本，代码如下：

|  |
| --- |
| static zend\_function \*zend\_duplicate\_function(zend\_function \*func, zend\_class\_entry \*ce) {  if (UNEXPECTED(func->type == ZEND\_INTERNAL\_FUNCTION)) { // 如果是内置函数  return zend\_duplicate\_internal\_function(func, ce); // 内置函数创建副本  } else { // 不是内置函数，没有真正创建副本，只增加引用次数  if (func->op\_array.refcount) { // 如果操作码有引用次数  (\*func->op\_array.refcount)++; // 增加引用次数  }  if (EXPECTED(func->op\_array.function\_name)) { // 如果有函数名  zend\_string\_addref(func->op\_array.function\_name); // 函数名增加引用次数  }  // 没有函数名时，是匿名函数（闭包）  return func; // 返回原函数，实际上只增加引用次数，没有创建副本  }  } |

如上所示，只有成员方法有内置函数（ZEND\_INTERNAL\_FUNCTION）标记时，才调用zend\_duplicate\_internal\_function()函数创建副本，否则只给成员方法增加引用次数。

#### **zend\_duplicate\_internal\_function()函数**

zend\_duplicate\_internal\_function()函数用于为内置函数创建副本：

|  |
| --- |
| static zend\_function \*zend\_duplicate\_internal\_function(zend\_function \*func, zend\_class\_entry \*ce) {  zend\_function \*new\_function;  if (UNEXPECTED(ce->type & ZEND\_INTERNAL\_CLASS)) { // 如果属于内置类  new\_function = pemalloc(sizeof(zend\_internal\_function), 1); // 创建新内置函数  memcpy(new\_function, func, sizeof(zend\_internal\_function)); // 把函数复制过来  } else { // 不是内置类，但仍然是内置函数  // arena 分配空间，大小是 内置函数大小  new\_function = zend\_arena\_alloc(&CG(arena), sizeof(zend\_internal\_function));  memcpy(new\_function, func, sizeof(zend\_internal\_function)); // 按内置函数大小复制  // 新函数添加标记 由 ARENA 分配  new\_function->common.fn\_flags |= ZEND\_ACC\_ARENA\_ALLOCATED;  }  if (EXPECTED(new\_function->common.function\_name)) { // 新函数有函数名  zend\_string\_addref(new\_function->common.function\_name); // 函数名添加引用次数  }    return new\_function; // 返回新函数  } |

如上所示，如果带有ZEND\_INTERNAL\_FUNCTION标记的成员方法属于内置内，使用pemalloc()函数分配内存创建副本，否则使用zend\_arena\_alloc()函数分配内存创建副本；返回新创建的副本。

### 第8步：继承构造方法和魔术方法

代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第8块代码**  do\_inherit\_parent\_constructor(ce); // 继承父类的构造方法 |

#### **do\_inherit\_parent\_constructor()函数**

do\_inherit\_parent\_constructor()函数用于继承父类的构造方法和魔术方法列表，代码如下：

|  |
| --- |
| static void do\_inherit\_parent\_constructor(zend\_class\_entry \*ce) {  zend\_class\_entry \*parent = ce->parent;  ce->create\_object = parent->create\_object; // create\_object 方法不能改变，一定要继承  // 魔术方法，这些方法在没有的时候才继承  if (EXPECTED(!ce->get\_iterator)) { ce->get\_iterator = parent->get\_iterator; }  if (EXPECTED(!ce->\_\_get)) { ce->\_\_get = parent->\_\_get; }  if (EXPECTED(!ce->\_\_set)) { ce->\_\_set = parent->\_\_set; }  if (EXPECTED(!ce->\_\_unset)) { ce->\_\_unset = parent->\_\_unset; }  if (EXPECTED(!ce->\_\_isset)) { ce->\_\_isset = parent->\_\_isset; }  if (EXPECTED(!ce->\_\_call)) { ce->\_\_call = parent->\_\_call; }  if (EXPECTED(!ce->\_\_callstatic)) { ce->\_\_callstatic = parent->\_\_callstatic; }  if (EXPECTED(!ce->\_\_tostring)) { ce->\_\_tostring = parent->\_\_tostring; }  if (EXPECTED(!ce->clone)) { ce->clone = parent->clone; }  if (EXPECTED(!ce->\_\_serialize)) { ce->\_\_serialize = parent->\_\_serialize; }  if (EXPECTED(!ce->\_\_unserialize)) { ce->\_\_unserialize = parent->\_\_unserialize; }  if (EXPECTED(!ce->serialize)) { ce->serialize = parent->serialize; }  if (EXPECTED(!ce->unserialize)) { ce->unserialize = parent->unserialize; }  if (!ce->destructor) { ce->destructor = parent->destructor; }  if (EXPECTED(!ce->\_\_debugInfo)) { ce->\_\_debugInfo = parent->\_\_debugInfo; }  if (ce->constructor) { // 如果子类有构造方法  // 如果父类有构造方法， 并且 构造方法有final标记  if (parent->constructor && UNEXPECTED(parent->constructor->common.fn\_flags & ZEND\_ACC\_FINAL)) {  zend\_error\_noreturn(E\_ERROR, ...); // 报错：不可以覆盖 final 方法  }  return; // 完成，有构造方法就不用继承了  }  ce->constructor = parent->constructor; // 子类没有构造方法，继承父类的  } |

如上所示，代码毕竟简单，业务逻辑分3部分：

1）继承create\_object()方法；

2）继承魔术方法列表，魔术方法只有在子类没有时才继承。继承魔术方法时也是要检查final标记的，只是不在这部分业务逻辑中；

3）继承构造方法，检查final标记。

### 第9步：实现接口

代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第9块代码**  if (ce->type == ZEND\_INTERNAL\_CLASS) { // 如果子类是内置类  if (parent\_ce->num\_interfaces) { // 如果父类有实现接口  zend\_do\_inherit\_interfaces(ce, parent\_ce); // 继承父类的接口  }  if (ce->ce\_flags & ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS) { // 如果子类是隐式抽象类  ce->ce\_flags |= ZEND\_ACC\_EXPLICIT\_ABSTRACT\_CLASS; // 添加显式抽象类标记  }  } |

如上所示，只有子类是内置类并且父类有实现接口时，才调用zend\_do\_inherit\_interfaces()函数继承父类实现过的接口。

#### **zend\_do\_inherit\_interfaces()函数**

zend\_do\_inherit\_interfaces()函数用于继承父类实现过的接口，代码如下：

|  |
| --- |
| // p1:子类，p2:父类  static void zend\_do\_inherit\_interfaces(zend\_class\_entry \*ce, const zend\_class\_entry \*iface) {  // 接口可以已经在接口列表中了  uint32\_t i, ce\_num, if\_num = iface->num\_interfaces; // 父类的接口数量  zend\_class\_entry \*entry; // 临时变量  ce\_num = ce->num\_interfaces; // 子类的接口数量  // 调整接口列表大小  if (ce->type == ZEND\_INTERNAL\_CLASS) { // 如果子类是内置类  ce->interfaces = (zend\_class\_entry \*\*)  realloc(ce->interfaces, sizeof(zend\_class\_entry \*) \* (ce\_num + if\_num));  } else { // 子类是用户类  ce->interfaces = (zend\_class\_entry \*\*)  erealloc(ce->interfaces, sizeof(zend\_class\_entry \*) \* (ce\_num + if\_num));  }  // 继承还没被继承的接口  while (if\_num--) { // 倒着遍历父类的接口列表  entry = iface->interfaces[if\_num]; // 取得接口  for (i = 0; i < ce\_num; i++) { // 遍历子类已有接口  if (ce->interfaces[i] == entry) { // 如果找到了  break; // 跳出  }  }  if (i == ce\_num) { **// 如果没有在中间跳出，说明在子类中没有这个接口**  ce->interfaces[ce->num\_interfaces++] = entry; // 把接口增加到列表最后  }  }  ce->ce\_flags |= ZEND\_ACC\_RESOLVED\_INTERFACES; // 已解决接口列表  while (ce\_num < ce->num\_interfaces) { // 遍历每个新增的接口  // 调用【接口本身】的interface\_gets\_implemented()方法进行处理  do\_implement\_interface(ce, ce->interfaces[ce\_num++]);  }  } |

#### **do\_implement\_interface()函数**

do\_implement\_interface()函数调用接口的interface\_gets\_implemented()方法进行操作，如果失败则报错，代码如下：

|  |
| --- |
| static inline void do\_implement\_interface(zend\_class\_entry \*ce, zend\_class\_entry \*iface) {  // 当前类不是接口 并且 调用接口的检测方法检测是否可实现此接口  if (!(ce->ce\_flags & ZEND\_ACC\_INTERFACE) && iface->interface\_gets\_implemented && iface->interface\_gets\_implemented(iface, ce) == FAILURE) {  zend\_error\_noreturn(E\_CORE\_ERROR, ...); // 报错：不可以实现接口  }  } |

### 第10步：继承父类的标记

代码如下：

|  |
| --- |
| **// zend\_do\_inheritance\_ex()函数的第10块代码**  ce->ce\_flags |= parent\_ce->ce\_flags & (ZEND\_HAS\_STATIC\_IN\_METHODS | ZEND\_ACC\_HAS\_TYPE\_HINTS | ZEND\_ACC\_USE\_GUARDS | ZEND\_ACC\_NOT\_SERIALIZABLE | ZEND\_ACC\_ALLOW\_DYNAMIC\_PROPERTIES);  } |

如上所示，子类继承父类的以下标记：

1）ZEND\_HAS\_STATIC\_IN\_METHODS：用户类有使用静态变量的成员方法；

2）ZEND\_ACC\_HAS\_TYPE\_HINTS：函数有指定类型的参数 或 类有指定类型的属性；

3）ZEND\_ACC\_USE\_GUARDS：有魔术方法 \_\_get/\_\_set/\_\_unset/\_\_isset 的类，使用 guards；

4）ZEND\_ACC\_NOT\_SERIALIZABLE：类不可序列化或反序列化；

5）ZEND\_ACC\_ALLOW\_DYNAMIC\_PROPERTIES：此类的实例可能有动态属性，不会触发弃用警告。

## 二）成员方法的继承检查

do\_inheritance\_check\_on\_method\_ex()函数用于在继承成员方法时，检查是否可继承，函数的业务检查步骤比较多，但业务逻辑并不复杂。函数的调用路径如下：

|  |
| --- |
| do\_inheritance\_check\_on\_method\_ex()  ->zend\_do\_perform\_implementation\_check() // 成员方法的实现检查  ->perform\_delayable\_implementation\_check() // 延时实现检查  ->add\_compatibility\_obligation() // 添加兼容性责任  ->get\_or\_init\_obligations\_for\_class()  ->variance\_obligation\_dtor()  ->emit\_incompatible\_method\_error() // 报错  ->zend\_do\_perform\_implementation\_check() // 成员方法的实现检查  ->zend\_perform\_covariant\_type\_check() // 继承中的类型兼容性检查 |

函数声名如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第1块代码**  // p1:子方法，p2:子方法所属类，p3:父方法，p4:父方法所属类，p5:当前类，p6:方法在方法表中的zval，p7：检查可见性，p8：只检查，p9：已检查过继承  static inheritance\_status do\_inheritance\_check\_on\_method\_ex(  zend\_function \*child, zend\_class\_entry \*child\_scope,  zend\_function \*parent, zend\_class\_entry \*parent\_scope,  zend\_class\_entry \*ce, zval \*child\_zv,  bool check\_visibility, bool check\_only, bool checked) {  uint32\_t child\_flags;  uint32\_t parent\_flags = parent->common.fn\_flags;  zend\_function \*proto; |

### 第1步：检查父类的私有成员方法

代码如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第2块代码**  // 如果父类的方法有 private 标记 并且 没有 abstract标记 并且 不是 构造方法  if (UNEXPECTED((parent\_flags & ZEND\_ACC\_PRIVATE) && !(parent\_flags & ZEND\_ACC\_ABSTRACT) && !(parent\_flags & ZEND\_ACC\_CTOR))) {  if (!check\_only) { // 如果不是只进行检查，添加标记【覆盖过私有成员属性或方法】  child->common.fn\_flags |= ZEND\_ACC\_CHANGED; // 添加 【更改访问权限】标记  }  // 父方法是私有的，并且不是 abstract方法，所以不用检验继承规则  return INHERITANCE\_SUCCESS; // 返回：继承成功  } |

如上所示，如果父类的成员方法同时满足3个条件：1）有 private 标记；2）没有abstract标记；3）不是构造方法。子类的同名方法可以直接覆盖它，这时直接返回继承成功（INHERITANCE\_SUCCESS）。

### 第2步：检查父类成员方法的final标记

代码如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第3块代码**  // 如果没有检查过 并且 父方法有 final 标记  if (!checked && UNEXPECTED(parent\_flags & ZEND\_ACC\_FINAL)) {  if (check\_only) { // 如果只进行检查不报错  return INHERITANCE\_ERROR; // 返回：继承出错  }  // 报错：不可以覆盖final方法  zend\_error\_at\_noreturn(E\_COMPILE\_ERROR, ...);  } |

### 第3步：检查static标记是否改变

代码如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第4块代码**  child\_flags = child->common.fn\_flags; // 子方法的标记  // 如果没有检查过继承 并且 父方法和子方法的静态标记不同  if (!checked && UNEXPECTED((child\_flags & ZEND\_ACC\_STATIC) != (parent\_flags & ZEND\_ACC\_STATIC))) {  if (check\_only) { // 如果只进行查检不报错  return INHERITANCE\_ERROR; // 返回：继承出错  }  if (child\_flags & ZEND\_ACC\_STATIC) { // 如果子方法有 static 标记  // 编译报错：不可以把非静态方法 变成静态方法  zend\_error\_at\_noreturn(E\_COMPILE\_ERROR,...);  } else { // 如果没有  // 编译报错：不可以把静态方法变成非静态方法  zend\_error\_at\_noreturn(E\_COMPILE\_ERROR,...);  }  } |

### 第4步：检查，不可以把非抽象方法变成抽象方法

代码如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第5块代码**  // 如果没有检查过 并且 子方法添加了 abstract  if (!checked && UNEXPECTED((child\_flags & ZEND\_ACC\_ABSTRACT) > (parent\_flags & ZEND\_ACC\_ABSTRACT))) {  if (check\_only) { // 如果只检查不报错  return INHERITANCE\_ERROR; // 返回：继承出错  }  // 报错：不可以把非抽象 方法 变成抽象方法  zend\_error\_at\_noreturn(E\_COMPILE\_ERROR, ...);  } |

### 第5步：确定要继承的域

代码如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第6块代码**  // 如果不是检查 并且 父方法是 private 或 变改变过  if (!check\_only && (parent\_flags & (ZEND\_ACC\_PRIVATE|ZEND\_ACC\_CHANGED))) {  child->common.fn\_flags |= ZEND\_ACC\_CHANGED; // 添加 【更改访问权限】标记  }  // 函数原型：如果父方法是继承来的，使用继承的原方法，否则使用父方法本身  proto = parent->common.prototype ? parent->common.prototype : parent; |

### 第6步：构造方法的特殊处理

代码如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第7块代码**  if (parent\_flags & ZEND\_ACC\_CTOR) { // 如果父方法是构造方法  if (!(proto->common.fn\_flags & ZEND\_ACC\_ABSTRACT)) { // 如果原型不是抽象方法  return INHERITANCE\_SUCCESS; // 返回：继承成功  }  parent = proto; // 使用原型作为父方法  } |

### 第7步：创建成员方法操作码副本

如果方法原来不属于当前类，并且方法是用户定义的，需要给这个方法创建副本，代码如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第8块代码**  // 如果不是只进行检查 并且子方法和父方法的原型不同 并且有传入子方法在哈希表中的zval  if (!check\_only && child->common.prototype != proto && child\_zv) {  do { // 这个do是为了中间使用break  **// 情况1：如果子方法原来不属于当前类 并且 是用户定义的方法**  if (child->common.scope != ce && child->type == ZEND\_USER\_FUNCTION) {  if (ce->ce\_flags & ZEND\_ACC\_INTERFACE) { // 情况1.1：如果当前类是接口  break; **// 为了跳过child->common.prototype = proto这行**  } else { // 情况1.2：当前类不是接口，还需要复制操作码  // 分配内存创建 op\_array  zend\_function \*new\_function = zend\_arena\_alloc(&CG(arena), sizeof(zend\_op\_array));  // 把子方法的操作码复制过来  memcpy(new\_function, child, sizeof(zend\_op\_array));  // 子方法哈希表中的方法指针指向新创建的op\_array，child也用新的  Z\_PTR\_P(child\_zv) = child = new\_function;  }  }  **// 情况2：如果子方法属于当前类 或 子方法不是用户定义的，无需操作**  child->common.prototype = proto; // 只有情况1.2不走这里  } while (0);  } |

如上所示，当满足3个条件时进行操作：

1）不是只进行检查（!check\_only）；

2）子方法和父方法的原型不同；

3）有传入子方法关联的zval指针。

业务逻辑分为3种情况：

1）如果子方法原来不属于当前类 并且 是用户定义的方法 并且 如果当前类是接口（interface）：这种情况无操作；

2）如果子方法原来不属于当前类 并且 是用户定义的方法 并且 如果当前类不是接口（interface）：为原成员方法的操作码创建副本，并使用新创建的副本。再更新子类成员方法的原型；

3）如果子方法属于当前类 或 子方法不是用户定义的：更新子类成员方法的原型。

### 第8步：检查可见性

代码如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第9块代码**  // 如果没有检查过 并且需要检查可见性 并且 子类的可见性大于父类  if (!checked && check\_visibility  && (child\_flags & ZEND\_ACC\_PPP\_MASK) > (parent\_flags & ZEND\_ACC\_PPP\_MASK)){  if (check\_only) { // 如果只检查不报错  return INHERITANCE\_ERROR; // 返回：继承出错  }  // 报错：子类的访问级别必须低于或等于父类  zend\_error\_at\_noreturn(E\_COMPILE\_ERROR, ...);  } |

### 第9步：实现检查

代码如下：

|  |
| --- |
| **// do\_inheritance\_check\_on\_method\_ex()函数，第10块代码**  if (!checked) { // 如果没进行过继承检查  if (check\_only) { // 如果只检查不报错  // 方法实现检查，p1:方法，p2:作用域，p3:原型，p4:原型作用域  return zend\_do\_perform\_implementation\_check(child, child\_scope, parent, parent\_scope);  }  // 允许延时实现 检查  perform\_delayable\_implementation\_check(ce, child, child\_scope, parent, parent\_scope);  }  return INHERITANCE\_SUCCESS; // 返回：继承成功  } |

zend\_do\_perform\_implementation\_check()函数用于对成员方法进行实现检查，函数业务逻辑比较复杂，perform\_delayable\_implementation\_check()函数用于延时实现检查，下面介绍这两个函数。

## 三）成员方法的实现检查

zend\_do\_perform\_implementation\_check()函数用于检查接口中的方法是否已经被正确实现，函数定义如下：

|  |
| --- |
| // 方法实现检查，p1:方法，p2:作用域，p3:原型，p4:原型作用域  static inheritance\_status zend\_do\_perform\_implementation\_check(  const zend\_function \*fe, zend\_class\_entry \*fe\_scope,  const zend\_function \*proto, zend\_class\_entry \*proto\_scope) {  uint32\_t i, num\_args, proto\_num\_args, fe\_num\_args;  inheritance\_status status, local\_status;  bool proto\_is\_variadic, fe\_is\_variadic; |

函数的业务逻辑比较复杂，包含如下步骤：

### 步骤1：检查调用限制，此函数只有在特定情况下才能调用

代码如下：

|  |
| --- |
| **// zend\_do\_perform\_implementation\_check()函数，第一段代码，步骤1，**  // 只有在接口中定义构造方法 或 者显式声名成抽象时，才检查构造方法  // 断言1：子方法不是构造方法 并且 父类不是接口 并且 父方法不是抽象的  ZEND\_ASSERT(!((fe->common.fn\_flags & ZEND\_ACC\_CTOR)  && ((proto->common.scope->ce\_flags & ZEND\_ACC\_INTERFACE) == 0  && (proto->common.fn\_flags & ZEND\_ACC\_ABSTRACT) == 0)));  // 如果原型方法（父方法）是私有 并且非抽象 ，不需要强制添加标记。私有抽象方法只有在 trait里才会存在。  // 断言2：没有private 或 有abstract。（在普通类里定义私有抽象方法会直接报错，测试过）  ZEND\_ASSERT(!(proto->common.fn\_flags & ZEND\_ACC\_PRIVATE)  || (proto->common.fn\_flags & ZEND\_ACC\_ABSTRACT)); |

### 步骤2：检验必须参数数量，子方法的必须参数不可以比父方法多

代码如下：

|  |
| --- |
| **// zend\_do\_perform\_implementation\_check()函数，第一段代码，步骤2**  // 必须参数数量不能增加  if (proto->common.required\_num\_args < fe->common.required\_num\_args) {  return INHERITANCE\_ERROR; // 返回：继承出错  } |

### 步骤3：检验引用返回标记是否兼容

代码如下：

|  |
| --- |
| **// zend\_do\_perform\_implementation\_check()函数，第一段代码，步骤3**  // 如果原型方法是引用返回 并且 当前方法不是引用返回  if ((proto->common.fn\_flags & ZEND\_ACC\_RETURN\_REFERENCE)  && !(fe->common.fn\_flags & ZEND\_ACC\_RETURN\_REFERENCE)) {  return INHERITANCE\_ERROR; // 返回：继承出错  } |

如上所示，在这一步检验中，如果父方法有引用返回标记，子方法不可以删除这个标记；但是如果父方法没有引用返回标记，子方法可以添加这个标记。

### 步骤4：检验对字典参数的支持是否一致

代码如下：

|  |
| --- |
| **// zend\_do\_perform\_implementation\_check()函数，第一段代码，步骤4**  // 原型是否使用字典参数  proto\_is\_variadic = (proto->common.fn\_flags & ZEND\_ACC\_VARIADIC) != 0;  // 子方法是否使用字典参数  fe\_is\_variadic = (fe->common.fn\_flags & ZEND\_ACC\_VARIADIC) != 0;  // 父方法用字典参数，子方法也要用。否则报错  /\* A variadic function cannot become non-variadic \*/  if (proto\_is\_variadic && !fe\_is\_variadic) {  return INHERITANCE\_ERROR; // 返回：继承错误  } |

### 步骤5：检查参数表是否兼容

代码如下：

|  |
| --- |
| **// zend\_do\_perform\_implementation\_check()函数，第一段代码，步骤5**  // 字典参数不包含在参数数量里  /\* The variadic argument is not included in the stored argument count. \*/  // 原型参数数量 + 字典参数  proto\_num\_args = proto->common.num\_args + proto\_is\_variadic;  fe\_num\_args = fe->common.num\_args + fe\_is\_variadic; // 子方法参数数量 + 字典参数  num\_args = MAX(proto\_num\_args, fe\_num\_args); // 两个里取大的一个  status = INHERITANCE\_SUCCESS; // 默认继承成功  for (i = 0; i < num\_args; i++) { // 遍历每个参数  // 父方法参数信息。参数信息小于参数个数？使用参数信息：（原型是字典？使用返回值信息：null）  zend\_arg\_info \*proto\_arg\_info =  i < proto\_num\_args ? &proto->common.arg\_info[i] :  proto\_is\_variadic ? &proto->common.arg\_info[proto\_num\_args - 1] : NULL;  // 子方法参数信息。参数信息小于参数个数？使用参数信息：（原型是字典？使用返回值信息：null）  zend\_arg\_info \*fe\_arg\_info =  i < fe\_num\_args ? &fe->common.arg\_info[i] :  fe\_is\_variadic ? &fe->common.arg\_info[fe\_num\_args - 1] : NULL;  if (!proto\_arg\_info) { **// 如果没有父方法参数信息**  // 添加一个新的可选参数 就好  /\* A new (optional) argument has been added, which is fine. \*/  continue; // 下一个参数  }  if (!fe\_arg\_info) { **// 如果没有子方法参数信息**  return INHERITANCE\_ERROR; // 返回：继承出错  }  // 检测子参数信息类型 是否可以继承父参数信息  local\_status = zend\_do\_perform\_arg\_type\_hint\_check(  fe\_scope, fe\_arg\_info, proto\_scope, proto\_arg\_info);  if (UNEXPECTED(local\_status != INHERITANCE\_SUCCESS)) { // 如果状态不是 继承成功  if (UNEXPECTED(local\_status == INHERITANCE\_ERROR)) { // 如果是继承出错  return INHERITANCE\_ERROR; // 返回继承出错  }  // 状态必须是继承未完成  ZEND\_ASSERT(local\_status == INHERITANCE\_UNRESOLVED);  status = INHERITANCE\_UNRESOLVED; // 继承未完成（还要继续遍历）  }  // 如果 参数的引用返回限制不一致（测试过，父类引用返回子类不是）  /\* by-ref constraints on arguments are invariant \*/  if (ZEND\_ARG\_SEND\_MODE(fe\_arg\_info) != ZEND\_ARG\_SEND\_MODE(proto\_arg\_info)) {  return INHERITANCE\_ERROR; // 返回：继承出错  }  } |

如上所示，这部分业务逻辑可以归纳为，遍历方法的每个参数，并对每个方法做如下检验：

1）如果父方法没有此参数的信息，跳过这个参数；

2）如果子方法没有此参数的信息，返回：继承出错（INHERITANCE\_ERROR）；

3）检测父子方法的参数信息是否兼容，如果检测结果是INHERITANCE\_ERROR，返回：继承出错；

4）如果父子方法的参数引用返回限制不一致，返回：继承出错。

#### **zend\_do\_perform\_arg\_type\_hint\_check()函数**

zend\_do\_perform\_arg\_type\_hint\_check()函数用于检查子类方法的参数类型，是否与父类的兼容，代码如下：

|  |
| --- |
| static inheritance\_status zend\_do\_perform\_arg\_type\_hint\_check(  zend\_class\_entry \*fe\_scope, zend\_arg\_info \*fe\_arg\_info,  zend\_class\_entry \*proto\_scope, zend\_arg\_info \*proto\_arg\_info) {  // 子类方法的参数类型未定义 或者是 mixed ，直接返回成功  if (!ZEND\_TYPE\_IS\_SET(fe\_arg\_info->type) || ZEND\_TYPE\_PURE\_MASK(fe\_arg\_info->type) == MAY\_BE\_ANY) {  return INHERITANCE\_SUCCESS; // 返回：继承成功  }  // 子类方法的参数已定义类型，并且不是mixed    if (!ZEND\_TYPE\_IS\_SET(proto\_arg\_info->type)) { // 如果父类型不存在  return INHERITANCE\_ERROR; // 返回：继承出错（不可以给子类的函数参数增加类型）  }  // 检查 fe\_type 是否可以作为 proto\_scope 的子类型，两个都可以是复杂类型  return zend\_perform\_covariant\_type\_check(  proto\_scope, proto\_arg\_info->type, fe\_scope, fe\_arg\_info->type);  } |

如上所示，进行以下几步检查：

1）用ZEND\_TYPE\_IS\_SET()宏程序进行检测，如果父子方法的参数类型相同，直接返回继承成功；

2）如果父方法没有此参数的类型信息，返回继承出错；

3）父子方法的参数类型不同，并且父方法也有参数类型要求，需要检查子方法的参数是否与父方法的参数类型兼容，返回调用zend\_perform\_covariant\_type\_check()函数的检查结果。

ZEND\_TYPE\_IS\_SET()宏程序的介绍参见“zent\_type”章节。

zend\_perform\_covariant\_type\_check()函数的详细介绍参见“继承中的类型兼容性检查”章节。

### ?步骤6：检查返回值是否兼容

如果原型中指定的返回值类型，需要检查返回值兼容性；原型中没有返回类型时，不允许增加新的返回类型。代码如下：

|  |
| --- |
| **// zend\_do\_perform\_implementation\_check()函数，第一段代码，步骤6**  if (proto->common.fn\_flags & ZEND\_ACC\_HAS\_RETURN\_TYPE) { // 如果父方法有返回类型  // 如果子方法没有返回类型 （实际上这里不管父接口加不加允null，这里都报错）  if (!(fe->common.fn\_flags & ZEND\_ACC\_HAS\_RETURN\_TYPE)) {  // 如果父方法返回值不允null  if (!ZEND\_ARG\_TYPE\_IS\_TENTATIVE(&proto->common.arg\_info[-1])) {  return INHERITANCE\_ERROR; // 返回：继承出错  }  if (status == INHERITANCE\_SUCCESS) { // 如果状态是 继承成功  return INHERITANCE\_WARNING; // 返回：继承警告  }  return status; // 返回状态  }  // 检查 fe\_type 是否可以作为 proto\_scope 的子类型，两个都可以是复杂类型  local\_status = zend\_perform\_covariant\_type\_check(  fe\_scope, fe->common.arg\_info[-1].type, proto\_scope, proto->common.arg\_info[-1].type);  if (UNEXPECTED(local\_status != INHERITANCE\_SUCCESS)) { // 如果本地状态不是：继承成功  // 如果是继承出错 并且 返回值允null  if (local\_status == INHERITANCE\_ERROR  && ZEND\_ARG\_TYPE\_IS\_TENTATIVE(&proto->common.arg\_info[-1])) {  local\_status = INHERITANCE\_WARNING; // 状态：继承警告  }  return local\_status; // 返回状态  }  }  return status; // 返回状态  } |

如上所示，业务逻辑分为以下几种情况：

1）子方法没有返回类型且父方法返回值不允null，返回状态“继承出错”；

2）子方法没有返回类型且前面检查的状态status结果是“继承成功”，返回状态“继承警告”（INHERITANCE\_WARNING）；

3）如果前面的检查状态不是“继承成功”，并且前面检查的状态local\_status结果是“继承错误”，并且方法原型的返回值允许null，返回状态“继承警告”；

4）如果前面的检查状态不是“继承成功”，并且前面检查的状态local\_status结果不是“继承错误”，返回状态local\_status。

5）其他情况，返回状态status;

### 衍生函数：成员方法的延时实现检查

perform\_delayable\_implementation\_check()函数用于进行延时实现检查：

|  |
| --- |
| static void perform\_delayable\_implementation\_check(  zend\_class\_entry \*ce,  const zend\_function \*fe, zend\_class\_entry \*fe\_scope,  const zend\_function \*proto, zend\_class\_entry \*proto\_scope) {  // 方法实现检查，p1:方法，p2:作用域，p3:原型，p4:原型作用域  inheritance\_status status =  zend\_do\_perform\_implementation\_check(fe, fe\_scope, proto, proto\_scope);  if (UNEXPECTED(status != INHERITANCE\_SUCCESS)) { // 如果未通过检查  if (EXPECTED(status == INHERITANCE\_UNRESOLVED)) { // 如果是 继承未完成  // 添加兼容性责任  add\_compatibility\_obligation(ce, fe, fe\_scope, proto, proto\_scope);  } else { // 如果是 继承出错  ZEND\_ASSERT(status == INHERITANCE\_ERROR || status == INHERITANCE\_WARNING);  // 报错：方法不兼容  emit\_incompatible\_method\_error(fe, fe\_scope, proto, proto\_scope, status);  }  }  } |

如上所示，延时实现性检查，也是先调zend\_do\_perform\_implementation\_check()函数进行实现检查，在检查结果不是“继承成功”时，进行一些附加处理：

1）在碰到未完成的继承状态时，会添加兼容性责任，留待后续处理；

2）其他情况时下报错：父类和子类的成员方法不兼容。

add\_compatibility\_obligation()函数用于添加类方法的兼容性责任，详情参见“变异责任”章节；

# 四、继承中的类型兼容性检查

zend\_perform\_covariant\_type\_check()函数用于继承中的兼容性检查，在实现继承时，使用它来检查子类中用到的类型是否与父类兼容。此函数在zend\_inheritance.c中有3次调用：

1）zend\_do\_perform\_arg\_type\_hint\_check()函数中，检查成员方法参数类型的兼容性；

2）zend\_do\_perform\_implementation\_check()函数中，检查成员方法返回值类型的兼容性；

3）property\_types\_compatible()函数中，检查类属性类型的兼容性。

函数的调用路径如下：

|  |
| --- |
| zend\_perform\_covariant\_type\_check() // 检查类型是否兼容  **->zend\_type\_permits\_self() // 检验一个类型是否是另一个类型的下属类型的子类**  ->resolve\_class\_name() // 查找类名时处理parent和self关键字  ->unlinked\_instanceof() // 可以安全用在未链接的类上的 Instanceof  ->instanceof\_function() // 检查一个类是否是另一个类的子类  ->lookup\_class() // 查找类  **->zend\_is\_intersection\_subtype\_of\_type() // 检查一个类型是否在另一个交叉类型中**  **->zend\_is\_intersection\_subtype\_of\_class()**  ->resolve\_class\_name() // 查找类名时处理parent和self关键字  ->track\_class\_dependency() // 追踪类从属?  ->unlinked\_instanceof() // 可以安全用在未链接的类上的Instanceof  ->instanceof\_function() // 检查一个类是否是另一个类的子类  ->lookup\_class() // 查找类  **->get\_class\_from\_type() // 从zend\_type实例取得类名**  ->resolve\_class\_name() // 查找类名时处理parent和self关键字  **->zend\_is\_class\_subtype\_of\_type() // 检查一个类型是否在另一个组合类型中**  ->resolve\_class\_name() // 查找类名时处理parent和self关键字  **->track\_class\_dependency() // 记录类从属**  ->unlinked\_instanceof() // 可以安全用在未链接的类上的 Instanceof  ->instanceof\_function() // 检查一个类是否是另一个类的子类  ->lookup\_class() // 查找类  **->register\_unresolved\_classes() // 注册未加载完成的类**  **->get\_class\_from\_type() // 从zend\_type实例取得类名** |

如上所示，函数调用很多其他函数来完成检验操作，调用路径比较复杂。

zend\_perform\_covariant\_type\_check()函数的业务逻辑可以划分为5个步骤：

第一步：检查mixed类型；

第二步：处理追加的类型；

第三步：检验类型；

第四步：处理检验结果；

第五步：处理未加载完成的类。

下面分别展开介绍。

zend\_perform\_covariant\_type\_check()函数声名如下：

|  |
| --- |
| // 检查 fe\_type 是否可以作为 proto\_scope 的子类型，两个都可以是复杂类型  // p1:子类，p2:子类型，p3:父类，p4:父类型  static inheritance\_status zend\_perform\_covariant\_type\_check(  zend\_class\_entry \*fe\_scope, zend\_type fe\_type,  zend\_class\_entry \*proto\_scope, zend\_type proto\_type) { |

## 第一步：检查mixed类型

这一步操作的业务逻辑比较简单，代码如下：

|  |
| --- |
| **// zend\_perform\_covariant\_type\_check()函数，第一步：检查mixed**  // 除了void, 所有类型都可以兼容mixed类型  if (ZEND\_TYPE\_PURE\_MASK(proto\_type) == MAY\_BE\_ANY &&  !ZEND\_TYPE\_CONTAINS\_CODE(fe\_type, IS\_VOID)) {  return INHERITANCE\_SUCCESS; // 继承成功  } |

END\_TYPE\_PURE\_MASK(proto\_type)宏程序和ZEND\_TYPE\_CONTAINS\_CODE()宏程序在“zent\_type”章节中介绍。

## 第二步：检查，子类中不可以添加新类型

代码如下：

|  |
| --- |
| **// zend\_perform\_covariant\_type\_check()函数，第二步：处理追加的类型**  uint32\_t fe\_type\_mask = ZEND\_TYPE\_PURE\_MASK(fe\_type); // 取得子类型的纯类型  uint32\_t proto\_type\_mask = ZEND\_TYPE\_PURE\_MASK(proto\_type); // 取得父类型的纯类型  uint32\_t added\_types = fe\_type\_mask & ~proto\_type\_mask; // 追加类型=子类型-父类型  if (added\_types) { // 如果有追加类型  // 如果 新类型 包含static 并且 允许使用self  if ((added\_types & MAY\_BE\_STATIC)  && zend\_type\_permits\_self(proto\_type, proto\_scope, fe\_scope)) {  added\_types &= ~MAY\_BE\_STATIC; // 删除 static 标记  // 示例： class a{ function b():static {} } （测试过）  }  if (added\_types == MAY\_BE\_NEVER) { // 如果新类型是never  return INHERITANCE\_SUCCESS; // 返回继承成功  }  if (added\_types) { // 如果子类添加了新类型  return INHERITANCE\_ERROR; // 返回继承出错  }  } |

added\_types的测试代码如下：

|  |
| --- |
| <?php  class a{ public int $a; }  class a2 extends a{ public int|string $a; } |

如上所示，在子类的成员方法中添加新类型会导致继承失败。

### zend\_type\_permits\_self()函数

zend\_type\_permits\_self()函数用于检验是否可以使用self，代码如下：

|  |
| --- |
| // 检验p3是否是p1的任何下属类型的子类，p2用于查找类名。p1:类型，p2:域，p3:self  static bool zend\_type\_permits\_self(zend\_type type, zend\_class\_entry \*scope,  zend\_class\_entry \*self) {  if (ZEND\_TYPE\_FULL\_MASK(type) & MAY\_BE\_OBJECT) { // 如果类型兼容对象类型  return 1; 返回允许  }  zend\_type \*single\_type;  ZEND\_TYPE\_FOREACH(type, single\_type) { // 遍历子类型  if (ZEND\_TYPE\_HAS\_NAME(\*single\_type)) { // 如果下属类型有名称  // 获取类名，先处理parent和self, 如果都不是，用当前域类名。p1:当前域  zend\_string \*name=resolve\_class\_name(scope, ZEND\_TYPE\_NAME(\*single\_type));  // 用类名查找类，不延时加载。p1:当前域，p2:类名  zend\_class\_entry \*ce = lookup\_class(self, name);  // 如果当前域 是 ce 的子类，那它也是 type的子类  if (ce && unlinked\_instanceof(self, ce)) {  return 1; // 返回：允许  }  }  } ZEND\_TYPE\_FOREACH\_END();  return 0; // 返回：不允许  } |

unlinked\_instanceof()函数的介绍参见“检查类的继承关系”章节。

## 第三步：检验类型

这一步是最关键的检验逻辑，主要思路如下：

1）子类型是联合类型，子类型中有一个元素校验通过就算成功；子类型是交叉类型，子类型中有一个元素校验失败就算失败。

2）因为有上一个条件的限制，所以在遍历校验过程中联合类型只可能报成功，交叉类型只可能报失败；联合类型的失败和交叉类型的成功，都要在遍历完成后才能得到（参看第四步代码）。

代码如下：

|  |
| --- |
| **// zend\_perform\_covariant\_type\_check()函数，第三步：检验类型**  zend\_type \*single\_type;  inheritance\_status early\_exit\_status;  bool have\_unresolved = false; // 没有未完成类型  if (ZEND\_TYPE\_IS\_INTERSECTION(fe\_type)) { // 如果子类型是交叉类型  ... // 验证交叉类型的业务逻辑，见下文  } else { // 子类型不是交叉类型，是联合类型或单个类型  ... // 验证联合类型的业务逻辑，见下文  } |

ZEND\_TYPE\_IS\_INTERSECTION()等相关宏程序的详细介绍参见“zend\_type”章节。

如上所示，业务逻辑主要分为两部分，子类型为交叉类型和子类型为联合类型。

### 子类型为交叉类型

交叉类型的验证业务逻辑如下：

|  |
| --- |
| // 交叉类型实际上是一个类型，不用循环，一次验完  // 步骤1.1: 确定可提前返回的状态。父类型是交叉类型：碰到错误马上返回错误。父类型是联合类型：碰到成功马上返回成功。  early\_exit\_status =  ZEND\_TYPE\_IS\_INTERSECTION(proto\_type) ? INHERITANCE\_ERROR : INHERITANCE\_SUCCESS;    // 这里已经保证了 fe\_type 是交叉类型  // 步骤1.2: 检查交叉类型 fe\_type，是否是 proto\_type 的子类型  inheritance\_status status = zend\_is\_intersection\_subtype\_of\_type(  fe\_scope, fe\_type, proto\_scope, proto\_type);  if (status == early\_exit\_status) { // 步骤1.3: 如果可以马上返回  return status;  }  if (status == INHERITANCE\_UNRESOLVED) { // 步骤1.4: 如果返回状态是未解决  have\_unresolved = true; // 有未加载类  } |

如上所示，验证过程分为几个步骤：

步骤1.1: 确定可提前返回的状态。原型是交叉类型时，碰到“错误”马上返回“错误”，因为交叉类型中每个类型必须完全兼容。原型是联合类型，碰到“成功”马上返回“成功”，因为联合类型中只要有一个类型兼容即可；

步骤1.2: 调用zend\_is\_intersection\_subtype\_of\_type()函数，检查 fe\_type，是否是 proto\_type 的子类型；

步骤1.3: 如果返回状态与前面确定的可提前返回状态相同，直接返回检查结果；

步骤1.4: 如果返回状态是“继承尚未完成”，添加未完成标记，继续后面的操作。

第三步里用到的几个函如下。

#### **zend\_is\_intersection\_subtype\_of\_type()函数**

zend\_is\_intersection\_subtype\_of\_type()函数用于检查交叉类型fe\_type 里，是否有某个类型 是 proto\_type类型的子类型，代码如下：

|  |
| --- |
| // 检查交叉类型 fe\_type，是否是 proto\_type 的子类型，proto\_type可能是交叉或联合类型。  static inheritance\_status zend\_is\_intersection\_subtype\_of\_type(  zend\_class\_entry \*fe\_scope, zend\_type fe\_type,  zend\_class\_entry \*proto\_scope, zend\_type proto\_type) {  bool have\_unresolved = false;  zend\_type \*single\_type;  // 获取纯属类型码  uint32\_t proto\_type\_mask = ZEND\_TYPE\_PURE\_MASK(proto\_type);    // 情况1，父类型包含对象，子类型只要是类就算成功  if (proto\_type\_mask & MAY\_BE\_OBJECT) { // 如果父类型 兼容对象  ZEND\_TYPE\_FOREACH(fe\_type, single\_type) { // 遍历子类型 下属类型  // 从类型获取类名，p1:当前域, p2:类型  zend\_string \*fe\_class\_name = get\_class\_from\_type(fe\_scope, \*single\_type);  if (!fe\_class\_name) { // 如果找不到类名  continue; // 下一个  }  // 用类名查找类，不延时加载。p1:当前域，p2:类名  zend\_class\_entry \*fe\_ce = lookup\_class(fe\_scope, fe\_class\_name);  if (fe\_ce) { // 只要有一个子类型是类，就算成功  track\_class\_dependency(fe\_ce, fe\_class\_name); // 记录类从属  return INHERITANCE\_SUCCESS; // 返回：继承成功  } else { // 没找到  have\_unresolved = true; // 未完成  }  } ZEND\_TYPE\_FOREACH\_END();  }    // 可提前退出的状态。交叉类型有一个出错就可以退出，联合类型有一个成功就可以退出。  inheritance\_status early\_exit\_status =  ZEND\_TYPE\_IS\_INTERSECTION(proto\_type) ? INHERITANCE\_ERROR : INHERITANCE\_SUCCESS;    // 情况2，父类型不包括对象，或子类型暂时没找到对应类  // 遍历父类型中的每个子类型，子类型可以是单个类型或交叉类型，但不会是联合类型  ZEND\_TYPE\_FOREACH(proto\_type, single\_type) {  inheritance\_status status;  // 如果父类型的这个下属类型是交叉类型  if (ZEND\_TYPE\_IS\_INTERSECTION(\*single\_type)) {  // 递归，检查 交叉类型fe\_type，是否是 proto\_type 的子类型  status = zend\_is\_intersection\_subtype\_of\_type(  fe\_scope, fe\_type, proto\_scope, \*single\_type);  } else { // 父类型的这个下属类型是单个类型  // 从类型获取父类的类名，p1:当前域, p2:父类型中的单个类型  zend\_string \*proto\_class\_name = get\_class\_from\_type(proto\_scope, \*single\_type);  if (!proto\_class\_name) { // 如果没有类名  continue; // 下一个子类型（联合类型才有下一个）  }  zend\_class\_entry \*proto\_ce = NULL; // 上面找到单个类型的名称了，还没找到对应类  // 检查交叉类型fe\_type是否是proto\_class\_name的子类型  status = zend\_is\_intersection\_subtype\_of\_class(  fe\_scope, fe\_type, proto\_scope, proto\_class\_name, proto\_ce);  }    // 可提前退出的状态。交叉类型有一个出错就可以退出，联合类型有一个成功就可以退出。  if (status == early\_exit\_status) { // 当前状态 和 提前退出状态相同  return status; // 返回当前状态  }  if (status == INHERITANCE\_UNRESOLVED) { // 如果：状态是 继承未完成  have\_unresolved = true; // 有未加载的类  }  } ZEND\_TYPE\_FOREACH\_END();  if (have\_unresolved) { // 如果有未加载的类  return INHERITANCE\_UNRESOLVED; // 返回：继承未完成  }  // 可提前退出的状态。交叉类型有一个出错就可以退出，联合类型有一个成功就可以退出。  // 提前退出状态是 继承出错（父类型是交叉类型） ？ 继承成功 ：（父类型是联合类型）继承出错  return early\_exit\_status == INHERITANCE\_ERROR ? INHERITANCE\_SUCCESS : INHERITANCE\_ERROR;  } |

#### **get\_class\_from\_type()函数**

get\_class\_from\_type()函数用于从类型获取类名：

|  |
| --- |
| // 从类型获取类名，p1:当前域, p2:类型  static zend\_string \*get\_class\_from\_type(zend\_class\_entry \*scope, zend\_type single\_type) {  if (ZEND\_TYPE\_HAS\_NAME(single\_type)) { // 如果类型有关联的类  // 获取类名，先处理parent和self, 如果都不是，用当前域类名。p1:当前域，p2:类型名  return resolve\_class\_name(scope, ZEND\_TYPE\_NAME(single\_type));  }  return NULL; // 类型没有名称，返回null  } |

ZEND\_TYPE\_NAME()宏程序用于检验类型是否有关联的类，更多介绍参见“zend\_type”章节。

#### **zend\_is\_intersection\_subtype\_of\_class()函数**

zend\_is\_intersection\_subtype\_of\_class()函数用于检查交叉类型fe\_type 里，是否有某个类型 是 proto\_ce类的子类型，代码如下：

|  |
| --- |
| // p1:子类，p2:子类的类型，p3:父类，p4:父类的子类型的类名，p5:父类的子类型的类（传入NULL）  static inheritance\_status zend\_is\_intersection\_subtype\_of\_class(  zend\_class\_entry \*fe\_scope, zend\_type fe\_type,  zend\_class\_entry \*proto\_scope, zend\_string \*proto\_class\_name, zend\_class\_entry \*proto\_ce) {  ZEND\_ASSERT(ZEND\_TYPE\_IS\_INTERSECTION(fe\_type)); // fe\_type必须是交叉类型  bool have\_unresolved = false; // 默认没有 未加载完成的类  zend\_type \*single\_type; // 遍历用的临时变量  // 遍历子类型列表，检查：类型组合中至少有一个类型是 父类型的下属类型  ZEND\_TYPE\_FOREACH(fe\_type, single\_type) { // 遍历交叉类型fe\_type的子类型  zend\_class\_entry \*fe\_ce;  zend\_string \*fe\_class\_name = NULL;  if (ZEND\_TYPE\_HAS\_NAME(\*single\_type)) { // 如果子类型有名称，是用户类  // 获取类名，先处理parent和self, 如果都不是，用当前域类名。p1:当前域  fe\_class\_name =resolve\_class\_name(fe\_scope,ZEND\_TYPE\_NAME(\*single\_type));  // 成功情况1，匹配到同名类：如果与 原型类 小写名称相同  if (zend\_string\_equals\_ci(fe\_class\_name, proto\_class\_name)) {  return INHERITANCE\_SUCCESS; // 返回：继承成功  }  // 用类名查找父类，不延时加载。p1:当前域，p2:类名  if (!proto\_ce) proto\_ce = lookup\_class(proto\_scope, proto\_class\_name);  // 用类名查找子类，不延时加载。p1:当前域，p2:类名  fe\_ce = lookup\_class(fe\_scope, fe\_class\_name);    } else { // 如果没有名称，是内置类型  ZEND\_UNREACHABLE(); // 标准类型不可以在交叉类型里，那样会触发致命编译错误  continue; // 下一个类型  }  if (!fe\_ce || !proto\_ce) { // 如果类 和 原型有一个无效  have\_unresolved = true; // 有未加载完成的类  continue; // 下一个  }  if (unlinked\_instanceof(fe\_ce, proto\_ce)) { // 成功情况2：如果两个类有 继承关系  track\_class\_dependency(fe\_ce, fe\_class\_name); // 记录类的从属关系  track\_class\_dependency(proto\_ce, proto\_class\_name); // 记录类的从属关系  return INHERITANCE\_SUCCESS; // 返回继承成功  }  } ZEND\_TYPE\_FOREACH\_END();  // 有未加载完成的类 ？继承未完成 ： 继承出错  return have\_unresolved ? INHERITANCE\_UNRESOLVED : INHERITANCE\_ERROR;  } |

#### **track\_class\_dependency()函数**

track\_class\_dependency()函数用于记录类的从属关系，代码如下：

|  |
| --- |
| // 追踪类从属  static void track\_class\_dependency(zend\_class\_entry \*ce, zend\_string \*class\_name) {  HashTable \*ht;  // 步骤1，检查是否是当前域  // 如果没有正在链接的类 或 此类就是正在链接的类  if (!CG(current\_linking\_class) || ce == CG(current\_linking\_class)) {  return; // 中断  // 如果正在链接其他类。并且：当前类名是 self 或 parent（还是在操作当前类）  } else if (zend\_string\_equals\_literal\_ci(class\_name, "self")  || zend\_string\_equals\_literal\_ci(class\_name, "parent")) {  return; // 中断  }    // 正在链接的类的 继承缓存  ht = (HashTable\*)CG(current\_linking\_class)->inheritance\_cache;  if (!(ce->ce\_flags & ZEND\_ACC\_IMMUTABLE)) { // 步骤2，如果类不是不可更改的  if (ht) { // 如果有继承缓存  zend\_hash\_destroy(ht); // 销毁继承缓存  FREE\_HASHTABLE(ht); // 释放哈希表  CG(current\_linking\_class)->inheritance\_cache = NULL; // 清空继承缓存指针  }  // 删除可缓存标记  CG(current\_linking\_class)->ce\_flags &= ~ZEND\_ACC\_CACHEABLE;  CG(current\_linking\_class) = NULL; // 清空正在链接的类  return; // 中断  }    // 步骤3，如果类是不可更改的， 记从属关系  if (!ht) { // 如果继承缓存不存在  ALLOC\_HASHTABLE(ht); // 创建继承缓存哈希表  zend\_hash\_init(ht, 0, NULL, NULL, 0); // 初始化哈希表  // 哈希表关联到正在连接的类上  CG(current\_linking\_class)->inheritance\_cache = (zend\_inheritance\_cache\_entry\*)ht;  }  zend\_hash\_add\_ptr(ht, class\_name, ce); // 步骤4，继承缓存中添加此类  } |

如上所示，业务逻辑分为几步：

1）碰到以下几种情况应立即中断操作：a）当前不在类链接过程中；b）当前类正在链接中；c）类名是self或parent；

2）如果当前类不是不可更改的：清空正在链接的类，中断操作；

3）如果类是不可更改的，并且当下正在链接的类没有承继缓存：给正在链接的类创建继承缓存哈希表；

4）把当前类添加到正在链接的类的继承缓存表中去，表中的类会在zend\_try\_early\_bind()函数中处理。

### 子类型为联合类型

子类元素类型为联合类型的检查逻辑比较复杂。因为交叉类型是一个类型，可以直接检查，而联合类型是一个类型列表，需要遍历列表中的每个类型进行检查。

联合类型的验证业务逻辑如下：

|  |
| --- |
| // 步骤2.1: 确定可提前返回的状态。子类型是联合类型，联合类型的每个下属类型都要兼容父类型，否则 继承出错  early\_exit\_status = INHERITANCE\_ERROR;  ZEND\_TYPE\_FOREACH(fe\_type, single\_type) { // 步骤2.2: 遍历子类型  inheritance\_status status;  if (ZEND\_TYPE\_IS\_INTERSECTION(\*single\_type)) { // 如果子类型下属类型是交叉类型  // 这里也已经保证了 single\_type 是交叉类型  // 检查 检查 fe\_type，是否是 proto\_type 的子类型  status = zend\_is\_intersection\_subtype\_of\_type(  fe\_scope, \*single\_type, proto\_scope, proto\_type);  } else { // 子类型下属类型不是交叉类型，是单个类型  // 从子类型的下属类型获取类名，p1:当前域, p2:类型  zend\_string \*fe\_class\_name = get\_class\_from\_type(fe\_scope, \*single\_type);  if (!fe\_class\_name) { // 如果没找到此类，跳过  continue;  }  // 检查子类型的下属类型是否与父类型兼容  status = zend\_is\_class\_subtype\_of\_type(  fe\_scope, fe\_class\_name, proto\_scope, proto\_type);  }    if (status == early\_exit\_status) { // 如果满足快速返回条件（返回状态为继承出错）  return status; // 返回当前状态  }  if (status == INHERITANCE\_UNRESOLVED) { // 如果返回状态是：继承未完成  have\_unresolved = true; // 临时标记，未完成  }  } ZEND\_TYPE\_FOREACH\_END(); |

#### **zend\_is\_class\_subtype\_of\_type()函数**

zend\_is\_class\_subtype\_of\_type()函数代码如下：

|  |
| --- |
| // 检查子类型 是否被父类型兼容  static inheritance\_status zend\_is\_class\_subtype\_of\_type(  zend\_class\_entry \*fe\_scope, zend\_string \*fe\_class\_name,  zend\_class\_entry \*proto\_scope, zend\_type proto\_type) {  zend\_class\_entry \*fe\_ce = NULL;  bool have\_unresolved = 0;  if (ZEND\_TYPE\_FULL\_MASK(proto\_type) & MAY\_BE\_OBJECT) { // 如果父类型允许 object  // 查找子类, 用类名查找类，不延时加载。p1:当前域，p2:类名  if (!fe\_ce) fe\_ce = lookup\_class(fe\_scope, fe\_class\_name);  if (!fe\_ce) { // 如果子类不存在  have\_unresolved = 1; // 有未加载类  } else { // 如果父类型允许object，并且子类型有对应的类存在  track\_class\_dependency(fe\_ce, fe\_class\_name); // 追踪类从属  return INHERITANCE\_SUCCESS; // 返回成功  }  }  zend\_type \*single\_type; // 循环变量  // 父类型是否是交叉类型  bool is\_intersection = ZEND\_TYPE\_IS\_INTERSECTION(proto\_type);  ZEND\_TYPE\_FOREACH(proto\_type, single\_type) { // 遍历父类型，逐个校验它的下属类型  if (ZEND\_TYPE\_IS\_INTERSECTION(\*single\_type)) { // 情况1， 如果父类型下属类型是交叉类型  // 递归检验子类型是否与此下属类型兼容  inheritance\_status subtype\_status = zend\_is\_class\_subtype\_of\_type(  fe\_scope, fe\_class\_name, proto\_scope, \*single\_type);  switch (subtype\_status) { // 根据返回状态操作  case INHERITANCE\_ERROR: // 继承出错  if (is\_intersection) { // 如果父类型是交叉类型，有一个出错就报出错  return INHERITANCE\_ERROR; // 返回继承出错  }  continue; // 如果不是交叉类型，下一个  case INHERITANCE\_UNRESOLVED: // 继承未完成  have\_unresolved = 1; // 未完成  continue; // 下一个  case INHERITANCE\_SUCCESS: // 继承成功  if (!is\_intersection) { // 如果父类型不是交叉类型，有一个成功就报成功  return INHERITANCE\_SUCCESS; // 返回继承 成功  }  continue; // 下一个  // 不会有其他状态，递归时所有的返回一定是上面3个状态之一  EMPTY\_SWITCH\_DEFAULT\_CASE();  }  }    // 情况2，下属类型是交叉类型在上面处理完了，所以到这里的一定是联合类型或单个类型  zend\_class\_entry \*proto\_ce; // 遍历用的临时变量  zend\_string \*proto\_class\_name = NULL; // 遍历用的临时变量  if (ZEND\_TYPE\_HAS\_NAME(\*single\_type)) { // 步骤2.1 如果类型有名称  // 获取类名，先处理parent和self, 如果都不是，用当前域类名。p1:当前域  proto\_class\_name =  resolve\_class\_name(proto\_scope, ZEND\_TYPE\_NAME(\*single\_type));  // 如果子类 和 父类名称相同  if (zend\_string\_equals\_ci(fe\_class\_name, proto\_class\_name)) {  if (!is\_intersection) { // 如果父不是交叉类型，是联合类型  return INHERITANCE\_SUCCESS; // 只要有一个匹配就可以，返回继承成功  }  continue; // 是交叉类型，继续  }  // 如果上刚刚没找到，还要再找，因为 track\_class\_dependency 时可能载入了  // 用类名查找类，不延时加载。p1:当前域，p2:类名  if (!fe\_ce) fe\_ce = lookup\_class(fe\_scope, fe\_class\_name);  // 用类名查找类，不延时加载。p1:当前域，p2:类名  proto\_ce = lookup\_class(proto\_scope, proto\_class\_name);  } else { // 没有名称的是标准（内置）类型  ZEND\_ASSERT(!is\_intersection); // 标准类型，不可以有交叉。父类型只能联合  continue; // 标准类型不校验，跳过  }  if (!fe\_ce || !proto\_ce) { // 步骤2.2， 如果父类 和 当前类 有一个没找到  have\_unresolved = 1; // 未解决，有未加载的类  continue; // 跳过  }  if (unlinked\_instanceof(fe\_ce, proto\_ce)) { // 步骤2.3， 如果子类是继承自父类  track\_class\_dependency(fe\_ce, fe\_class\_name); // 记录类从属  track\_class\_dependency(proto\_ce, proto\_class\_name); // 记录类从属  if (!is\_intersection) { // 如果不是交叉类型。才可能报成功。  return INHERITANCE\_SUCCESS; // 继承成功  }  // 交叉类型还要继续  } else { // 如果没有继承关系  if (is\_intersection) { // 如果没有继承关系，并且父类型是交叉类型  return INHERITANCE\_ERROR; // 继承出错  }  }  } ZEND\_TYPE\_FOREACH\_END();  if (have\_unresolved) { // 如果有未完成的  return INHERITANCE\_UNRESOLVED; // 返回继承未完成  }  // 如果是交叉类型，到这里说明没有碰到失败，继承成功  // 如果是联合类型，到这里说明没有碰到成功，继承出错  return is\_intersection ? INHERITANCE\_SUCCESS : INHERITANCE\_ERROR;  } |

函数的业务逻辑比较复杂，为方便理解，把业务逻辑整理成表格如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **第一层** | **第二层** | **第三层** | **第四层** | **第五层** | **第六层** |
| **步骤1** | 如果父类兼容对象 | 查找父类，如果没找到 | 标记，有未完成的类，继续后面的步骤 | | | |
| 如果找到父类 | 记录类从属，返回继承成功 | | | |
| **步骤2** | 遍历父类型的每个下属类型 | 如果下属类型是交叉类型，递归检查此下属类型，根据结果操作 | 如果递归返回结果是：继承出错 | 如果父类型是交叉类型 | 返回继承出错 | |
| 父类型不是交叉类型 | continue跳到下一个下属类型 | |
| 如果递归返回结果是：继承未完成 | 标记，有未完成的继承 continue跳到下一个下属类型 | | |
| 如果递归返回结果是：继承成功 | 父类型不是交叉类型 | 返回继承成功 | |
| 父类型是交叉类型 | continue跳到下一个下属类型 | |
| 如果下属类型不是交叉类型 | 如果下属类型有绑定类名 | 如果绑定的类名与子类型类名相同 | 如果父类型不是交叉类型 | 返回继承成功 |
| 如果父类型是交叉类型 | continue跳到下一个下属类型 |
| 如果绑定的类名与子类型类名不相同 | 使用绑定的类名加载类 继续后面的操作 | |
| 如果下属类型没有绑定名称 | （下属类型是基本类型，不用处理）  continue跳到下一个下属类型 | | |
| 如果子类型或父类型有一个没加载成功 | continue跳到下一个下属类型 | | |
| 如果子类型是继承自父类型 | 如果父类型不是交叉类型 | 返回继承成功 | |
| 如果父类型是交叉类型 | 继续后面的逻辑 | |
| 如果子类型不是继承自父类型 | 如果父类型是交叉类型 | 返回继承成功失败 | |
| 如果父类型不是交叉类型 | 继续后面的逻辑 | |
| **步骤3** | 如果有未完成的加载 | | 返回：继承未完成 | | | |
| 没有未完成的加载（在中间所有检测中都没有反回） | | 父类型是交叉类型 | 返回成功 | | |
| 父类型不是交叉类型 | 返回失败 | | |

如上所示，表格坚行表示步骤，横行表示逻辑层次。

## 第四步：处理检验结果

代码如下：

|  |
| --- |
| **// zend\_perform\_covariant\_type\_check()函数，第四步：处理检验结果**  if (!have\_unresolved) { // 如果没有未解决的  // 如果早先是 继承出错？返回 继承成功（因为没碰到出错），否则返回 继承出错（因为没碰到成功）  return early\_exit\_status == INHERITANCE\_ERROR ? INHERITANCE\_SUCCESS : INHERITANCE\_ERROR;  } |

## 第五步：处理未加载完成的类

代码如下：

|  |
| --- |
| **// zend\_perform\_covariant\_type\_check()函数，第五步：处理未加载完成的类**  // 查找并解决类型中的类，p1:当前域，p2:类型（或表）  register\_unresolved\_classes(fe\_scope, fe\_type);  register\_unresolved\_classes(proto\_scope, proto\_type);  return INHERITANCE\_UNRESOLVED; // 返回继承未完成  } |

# 五、类实现接口

接口（interface）在PHP内核中没有专门的数据结构，它与类使用同样的数据结构zend\_class\_entry。接口和类（class）一样，在编译时通过zend\_compile.c文件中的zend\_compile\_class\_decl()函数来创建。

类实现接口的业务逻辑在zend/zend\_inheritance.c文件中。

## 一）实现单个接口

类实现接口需要用到zend\_do\_implement\_interface()函数，函数代码如下：

|  |
| --- |
| ZEND\_API void zend\_do\_implement\_interface(zend\_class\_entry \*ce, zend\_class\_entry \*iface) {  uint32\_t i, ignore = 0;  uint32\_t current\_iface\_num = ce->num\_interfaces;  // 父类实现的接口数量  uint32\_t parent\_iface\_num = ce->parent ? ce->parent->num\_interfaces : 0;  zend\_string \*key;  zend\_class\_constant \*c;  ZEND\_ASSERT(ce->ce\_flags & ZEND\_ACC\_LINKED); // 此类必需链接过父类或接口  for (i = 0; i < ce->num\_interfaces; i++) { // 遍历所有接口  if (ce->interfaces[i] == NULL) { // 如果此接口无效  // 接口列表左移一个元素，把空的填上  memmove(ce->interfaces + i, ce->interfaces + i + 1, sizeof(zend\_class\_entry\*) \* (--ce->num\_interfaces - i));  i--; // 再重新检查这个位置  } else if (ce->interfaces[i] == iface) { // 如果找到当下要实现的接口  if (EXPECTED(i < parent\_iface\_num)) { // 如果是父类实现过的接口（父类的接口在前）  ignore = 1; // 忽略标记（父类的接口可以再实现一次,不会报错）  } else { // 不是父类实现过的接口  // 报错 不能实现前面实现过的接口  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  }  }  if (ignore) { // 忽略（父类实现过的接口）  // 检查 尝试重新定义接口常量  /\* Check for attempt to redeclare interface constants \*/  // 遍历接口的所有常量  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&iface->constants\_table, key, c) {  // 继承常量前的检查，p1:类名，p2:父常量，p3:名称  do\_inherit\_constant\_check(ce, c, key);  } ZEND\_HASH\_FOREACH\_END();  } else { // 正常实现此接口  if (ce->num\_interfaces >= current\_iface\_num) { // 如果接口列表不够长了  if (ce->type == ZEND\_INTERNAL\_CLASS) { // 如果 是内置类  // 原生方法进行内存操作，调整原接口列表大小  ce->interfaces = (zend\_class\_entry \*\*) realloc(ce->interfaces, sizeof(zend\_class\_entry \*) \* (++current\_iface\_num));  } else { // 用户类  // zend方法进行内存操作，调整原接口列表大小  ce->interfaces = (zend\_class\_entry \*\*) erealloc(ce->interfaces, sizeof(zend\_class\_entry \*) \* (++current\_iface\_num));  }  }  ce->interfaces[ce->num\_interfaces++] = iface; // 把接口添加到接口列表里  do\_interface\_implementation(ce, iface); // 实现接口 ~  }  } |

如上所示，业务逻辑分为两步：

第一步，检查此接口是否已经被实现过，如果被父类实现过，添加忽略（ignore）标记；如果被此类实现过，报错，不可以重复实现同一个接口。

第二步，如果接口被父类实现过，只调用do\_inherit\_constant\_check()函数进行常量检查；否则把接口添加到列表中，并调用do\_interface\_implementation()函数来实现接口。

### do\_inherit\_constant\_check()函数

do\_inherit\_constant\_check()函数用于检查，接口中的常量是否与此类中的常量兼容，代码如下：

|  |
| --- |
| // p1:当前类，p2:接口中的常量，p3:常量名称  static bool do\_inherit\_constant\_check(zend\_class\_entry \*ce,  zend\_class\_constant \*parent\_constant, zend\_string \*name) {  // 从当前类的常量表中按名称查找  zval \*zv = zend\_hash\_find\_known\_hash(&ce->constants\_table, name);  if (zv == NULL) { // 如果没找到  return true; // 返回真，可以继承此常量  }  zend\_class\_constant \*old\_constant = Z\_PTR\_P(zv); // 如果找到，取出已有常量  // 如果父常量的所属类 和 已有常量不同 并且 父常量 有final标记  if (parent\_constant->ce != old\_constant->ce && (ZEND\_CLASS\_CONST\_FLAGS(parent\_constant) & ZEND\_ACC\_FINAL)) {  // 报错：不可以覆盖 有final 标记的 常量  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  // 如果已有常量所属类 不是父常量的所属类 并且 已有常量不属于此类  if (old\_constant->ce != parent\_constant->ce && old\_constant->ce != ce) {  // 报错：冲突，有两个可继承的常量（可能来自接口和类，测试过）  zend\_error\_noreturn(E\_COMPILE\_ERROR,...);  }  return false; // 其他情况，返回不可继承此常量  } |

如上所示，业务逻辑分为以下几种情况：

1）此类的常量列表中没有同名常量，返回真，可以正常继承；

2）此类的同名常量与父类常量的所属类不同，这种情况下会覆盖父类常量，但如果父类常量有final标记，不可以覆盖，这时会直接报错：不可以覆盖有final标记的常量；

3）此类的同名常量与父类常量的所属类不同，并且此类的同名常量不属于此类（来自接口继承），这时会直接报错：冲突，有两个可继承的常量；

4）其他情况，返回假，不可继承但不报错。

### do\_interface\_implementation()函数

do\_interface\_implementation()函数用于让类实现接口，代码如下：

|  |
| --- |
| static void do\_interface\_implementation(zend\_class\_entry \*ce, zend\_class\_entry \*iface) {  zend\_function \*func;  zend\_string \*key;  zend\_class\_constant \*c;  // 遍历此接口的常量表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&iface->constants\_table, key, c) {  do\_inherit\_iface\_constant(key, c, ce, iface); // 逐个继承常量  } ZEND\_HASH\_FOREACH\_END();  // 遍历 此接口的方法表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&iface->function\_table, key, func) {  do\_inherit\_method(key, func, ce, 1, 0); // 逐个继承方法,p4:方法属于接口, p5:未检查过  } ZEND\_HASH\_FOREACH\_END();  do\_implement\_interface(ce, iface); // 调用【接口本身】的实现方法进行处理  if (iface->num\_interfaces) { // 如果有多个接口  zend\_do\_inherit\_interfaces(ce, iface); // 继承父类的接口  }  } |

如上所示，业务逻辑分为4步：

1）遍历接口中的常量表，调用do\_inherit\_iface\_constant()函数继承接口中的常量；

2）遍历接口中的方法表，调用do\_inherit\_method()函数继承接口中的成员方法；

3）调用接口本身的实现方法进行处理；

4）如果接口继承了其他接口，调用zend\_do\_inherit\_interfaces()函数实现这些父接口。

do\_inherit\_method()函数和zend\_do\_inherit\_interfaces()函数已经在“类的继承”章节中介绍过。

#### **do\_inherit\_iface\_constant()函数**

do\_inherit\_iface\_constant()函数用于继承接口中的常量，代码如下：

|  |
| --- |
| static void do\_inherit\_iface\_constant(zend\_string \*name, zend\_class\_constant \*c,  zend\_class\_entry \*ce, zend\_class\_entry \*iface) {  if (do\_inherit\_constant\_check(ce, c, name)) { // 检查，如果可以继承此常量  zend\_class\_constant \*ct; // 临时变量  if (Z\_TYPE(c->value) == IS\_CONSTANT\_AST) { // 如果值为常量表达式  // 类删除【已更新常量】标记  ce->ce\_flags &= ~ZEND\_ACC\_CONSTANTS\_UPDATED;  // 类添加【有常量语句】标记  ce->ce\_flags |= ZEND\_ACC\_HAS\_AST\_CONSTANTS;  if (iface->ce\_flags & ZEND\_ACC\_IMMUTABLE) { // 如果接口不可更改  // 分配内存创建新常量  ct = zend\_arena\_alloc(&CG(arena), sizeof(zend\_class\_constant));  memcpy(ct, c, sizeof(zend\_class\_constant)); // 把原常量复制过来  c = ct; // 使用新常量  // 添加标记【常量跟随类一起销毁】  Z\_CONSTANT\_FLAGS(c->value) |= CONST\_OWNED;  }  }  if (ce->type & ZEND\_INTERNAL\_CLASS) { // 如果是内置类  ct = pemalloc(sizeof(zend\_class\_constant), 1); // 原生方式分配内存创建常量  memcpy(ct, c, sizeof(zend\_class\_constant)); // 把原常量复制过来  c = ct; // 使用新常量  }  zend\_hash\_update\_ptr(&ce->constants\_table, name, c); // 把新常量添加到常量表里  }  // 不可继承的常量直接跳过了，不会报错  } |

如上所示，对于可以继承的常量，分为3种情况处理：

1）如果接口有不可更改（ZEND\_ACC\_IMMUTABLE）标记，使用zend\_arena\_alloc()函数分配内存，创建常量副本，并把副本关联到类的常量表中。

2）如果当前类是内置类（ZEND\_INTERNAL\_CLASS），使用pemalloc()函数分配内存，创建常量副本，并把副本关联到类的常量表中；

3）其他情况，直接把接口常量关联到类的常量列表中。

## 二）实现多个接口

实现多个接口由zend\_do\_implement\_interfaces()函数来处理，函数声名如下：

|  |
| --- |
| **// zend\_do\_implement\_interfaces()函数，第1块代码**  static void zend\_do\_implement\_interfaces(zend\_class\_entry \*ce, zend\_class\_entry \*\*interfaces) {  zend\_class\_entry \*iface;  // 父类接口数量，这个数不动  uint32\_t num\_parent\_interfaces = ce->parent ? ce->parent->num\_interfaces : 0;  // 自己的接口数量，这个数会增加  uint32\_t num\_interfaces = num\_parent\_interfaces;  zend\_string \*key;  zend\_class\_constant \*c;  uint32\_t i, j; |

函数的业务逻辑比较复杂，分为以下步骤：

### 第1步：过滤掉无效接口，留下有效接口

代码如下：

|  |
| --- |
| **// zend\_do\_implement\_interfaces()函数，第2块代码**  // 遍历每个自己实现的接口（不是继承自父类）  for (i = 0; i < ce->num\_interfaces; i++) {  // 跳过父类实现的接口，获取到自己实现的接口  iface = interfaces[num\_parent\_interfaces + i];  // 如果接口未链接  if (!(iface->ce\_flags & ZEND\_ACC\_LINKED)) {  // 添加从属责任。（从属责任对象关关联到从属类）  add\_dependency\_obligation(ce, iface);  }  // 如果找到的不是接口  if (UNEXPECTED(!(iface->ce\_flags & ZEND\_ACC\_INTERFACE))) {  efree(interfaces); // 释放全部接口  // 报错：不可实现它，它不是接口  zend\_error\_noreturn(E\_ERROR, ...);  return; // 中断  }  // 遍历所有前面的接口，查找重复  for (j = 0; j < num\_interfaces; j++) {  // 如果前面已经实现过此接口  if (interfaces[j] == iface) {    // 如果不是父类接口（是自己实现的接口）  if (j >= num\_parent\_interfaces) {  efree(interfaces); // 释放全部接口  // 不可实现前面已经实现过的接口  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  return; // 中断  }    // 是父类实现过的接口，遍历接口常量，  // zend\_do\_implement\_interface()函数一样，只更新常量  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&iface->constants\_table, key, c) {  // 继承常量前的检查，p1:类名，p2:父常量，p3:名称  do\_inherit\_constant\_check(ce, c, key);  } ZEND\_HASH\_FOREACH\_END();  iface = NULL; // 清空此 重复的 接口  break; // 跳出  }  }    if (iface) { // 如果接口有效  // 放进接口列表里  interfaces[num\_interfaces] = iface;  num\_interfaces++; // 有效接口数量 +1  }  } |

业务逻辑归纳如下，遍历每个自己实现的接口（不是继承自父类），对每个接口进行如下操作：

1）如果接口未链接完毕（ZEND\_ACC\_LINKED），调用add\_dependency\_obligation()函数为它添加“从属责任”；

2）如果碰到接口以外的东西（类，trait等）报错；

3）遍历所有效接口（初始是所有父类实现的接口）进行检查，如果当前接口已经在有效接口中，分两种情况进行操作：a）如果此接口是自己实现的，直接报错：不可以重复实现同一个接口；b）如果此接口是父类实现过的，添加接口中的常量并跳过，避免重复添加相同的接口；

4）把通过以上检验的接口指针更新到接口指针列表中的指定位置。

通过以上检测后，列表中只剩下有效接口。

add\_dependency\_obligation()函数用于添加从属责任，详情参见“变异责任”章节。

### 第2步：处理类名

代码如下：

|  |
| --- |
| **// zend\_do\_implement\_interfaces()函数，第3块代码**  // 如果当前类没被缓存过  if (!(ce->ce\_flags & ZEND\_ACC\_CACHED)) {  // 遍历当前类的所有接口  for (i = 0; i < ce->num\_interfaces; i++) {  // 释放接口名  zend\_string\_release\_ex(ce->interface\_names[i].name, 0);  // 释放小写接口名  zend\_string\_release\_ex(ce->interface\_names[i].lc\_name, 0);  }  // 释放所有接口名  efree(ce->interface\_names);  } |

### 第3步：更新接口数量和标记

代码如下：

|  |
| --- |
| **// zend\_do\_implement\_interfaces()函数，第4块代码**  ce->num\_interfaces = num\_interfaces; // 更新接口数量  ce->interfaces = interfaces; // 应用接口列表  // 标记，已完成接口实现  ce->ce\_flags |= ZEND\_ACC\_RESOLVED\_INTERFACES; |

### 第4步：检测每个接口是否可实现

代码如下：

|  |
| --- |
| **// zend\_do\_implement\_interfaces()函数，第5块代码**  // 遍历所有父类接口  for (i = 0; i < num\_parent\_interfaces; i++) {  // 调用接口的检测方法检测是否可实现此接口  do\_implement\_interface(ce, ce->interfaces[i]);  } |

do\_implement\_interface()函数已经在“类的继承”章节中介绍过。

### 第5步：遍历并实现每个接口

代码如下：

|  |
| --- |
| **// zend\_do\_implement\_interfaces()函数，第6块代码**  // 在循环继承中，可能加入新接口，  // 使用 num\_interfaces 而不是 ce->num\_interfaces 不要重新处理新接口  for (; i < num\_interfaces; i++) { // 继续处理自己实现的接口  // 接口实现  do\_interface\_implementation(ce, ce->interfaces[i]);  }  } |

do\_interface\_implementation()函数已经在“实现单个接口”章节中介绍过。

# 六、类中引用trait

trait使用与类（class）相同的数据结构zend\_class\_entry，与之相关的数据结构还有如下几个：

|  |
| --- |
| typedef struct \_zend\_trait\_method\_reference { // trait 方法引用  zend\_string \*method\_name; // 方法名  zend\_string \*class\_name; // 类名  } zend\_trait\_method\_reference;  typedef struct \_zend\_trait\_precedence { // trait 优先  zend\_trait\_method\_reference trait\_method; // trait 方法引用  uint32\_t num\_excludes; // 排除类数量  zend\_string \*exclude\_class\_names[1]; // 排除类名  } zend\_trait\_precedence;  typedef struct \_zend\_trait\_alias { // trait 别名  zend\_trait\_method\_reference trait\_method; // trait 方法引用  zend\_string \*alias; // 方法别名  uint32\_t modifiers; // 修饰符  } zend\_trait\_alias; |

以上几个结构体都会在后续内容中使用到。

在PHP代码类中引用（use）的trait，需要绑定到类上，使引用的内容生效，绑定trait主要通过zend\_do\_bind\_traits()函数来实现，函数代码如下：

|  |
| --- |
| static void zend\_do\_bind\_traits(zend\_class\_entry \*ce, zend\_class\_entry \*\*traits) {  HashTable \*\*exclude\_tables; // 排除列表  zend\_class\_entry \*\*aliases; // 别名  // 第1步，在类中完成 trait 初始化  zend\_traits\_init\_trait\_structures(ce, traits, &exclude\_tables, &aliases);  // 第2步，绑定trait中的成员方法 p1:类，p2:trait列表，p3:排除列表，p4:trait别名列表  zend\_do\_traits\_method\_binding(ce, traits, exclude\_tables, aliases);  if (aliases) { // 如果有别名列表  efree(aliases); // 释放别名列表  }  if (exclude\_tables) { // 如果有排除列表  efree(exclude\_tables); // 释放排除列表  }    zend\_do\_traits\_constant\_binding(ce, traits); // 第3步，绑定trait中的常量  zend\_do\_traits\_property\_binding(ce, traits); // 第4步，绑定 trait中的属性  } |

函数代码虽然简单，但调用逻辑比较复杂，调用逻辑表如下：

|  |
| --- |
| zend\_do\_bind\_traits()  **->zend\_traits\_init\_trait\_structures() // 第1步：初始化trait**  ->zend\_check\_trait\_usage()  **->zend\_do\_traits\_method\_binding() // 第2步：绑定成员方法**  ->zend\_fixup\_trait\_method()  ->zend\_traits\_copy\_functions()  ->zend\_add\_trait\_method()  ->do\_inheritance\_check\_on\_method()  ->fixup\_trait\_scope()  **->zend\_do\_traits\_constant\_binding() // 第3步：绑定常量**  ->do\_trait\_constant\_check()  ->check\_trait\_property\_or\_constant\_value\_compatibility()  ->find\_first\_constant\_definition()  **->zend\_do\_traits\_property\_binding() // 第4步：绑定属性**  ->check\_trait\_property\_or\_constant\_value\_compatibility()  ->zend\_type\_copy\_ctor()  ->find\_first\_property\_definition()  ->property\_types\_compatible()  ->zend\_perform\_covariant\_type\_check() |

如上所示，业务逻辑主要分成以下几步：

第一步：在类中完成trai初始化；

第二步：绑定trait中的成员方法；

第三步：绑定trait中的常量；

第四步：绑定trait中的属性（成员变量）。

## 一）在类中完成 trait 初始化（创建并返回别名列表）

zend\_traits\_init\_trait\_structures()函数用于构造trait结构，函数声名如下：

|  |
| --- |
| **// zend\_traits\_init\_trait\_structures()函数，第1块代码**  static void zend\_traits\_init\_trait\_structures(zend\_class\_entry \*ce, zend\_class\_entry \*\*traits, HashTable \*\*\*exclude\_tables\_ptr, zend\_class\_entry \*\*\*aliases\_ptr) {  size\_t i, j = 0;  zend\_trait\_precedence \*\*precedences;  zend\_trait\_precedence \*cur\_precedence;  zend\_trait\_method\_reference \*cur\_method\_ref;  zend\_string \*lc\_trait\_name;  zend\_string \*lcname;  HashTable \*\*exclude\_tables = NULL;  zend\_class\_entry \*\*aliases = NULL;  zend\_class\_entry \*trait; |

其业务逻辑主要分为两部分：1）处理优先列表和排除列表；2）处理别名列表

### 第1步：检验优先列表、创建排除列表

代码如下

|  |
| --- |
| **// zend\_traits\_init\_trait\_structures()函数，第2块代码**  if (ce->trait\_precedences) { // 人选  // 创建排除列表，哈希表的列表  exclude\_tables = ecalloc(ce->num\_traits, sizeof(HashTable\*));  i = 0; // 循环变量  precedences = ce->trait\_precedences; // 取出trait优先列表  ce->trait\_precedences = NULL; // 先清空原指针  // 遍历trait优先列表（zend\_trait\_precedence 实例）  while ((cur\_precedence = precedences[i])) {  // 优先的方法  cur\_method\_ref = &cur\_precedence->trait\_method;  // trait名 转小写  lc\_trait\_name = zend\_string\_tolower(cur\_method\_ref->class\_name);  // 在执行时类表里查找  trait = zend\_hash\_find\_ptr(EG(class\_table), lc\_trait\_name);  // 释放小定名称  zend\_string\_release\_ex(lc\_trait\_name, 0);  // 如果没找到 或 trait未完成链接  if (!trait || !(trait->ce\_flags & ZEND\_ACC\_LINKED)) {  // 报错：找不到trait  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  // trait 应用检查：trait是否有效 并且 被添加到类里  zend\_check\_trait\_usage(ce, trait, traits);  // 保证优先的方法确实有效  /\*\* Ensure that the preferred method is actually available. \*/  // 方法名小写  lcname = zend\_string\_tolower(cur\_method\_ref->method\_name);  // 如果方法不存在  if (!zend\_hash\_exists(&trait->function\_table, lcname)) {  // 报错：定义在优先规则中的方法不存在  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  // 遍历优先列表中的排除列表  for (j = 0; j < cur\_precedence->num\_excludes; j++) {  // 排除的 trait 类名  zend\_string\* class\_name = cur\_precedence->exclude\_class\_names[j];  zend\_class\_entry \*exclude\_ce;  uint32\_t trait\_num;  // 排除的 trait 名小写  lc\_trait\_name = zend\_string\_tolower(class\_name);  // 在执行时类表中查找  exclude\_ce = zend\_hash\_find\_ptr(EG(class\_table), lc\_trait\_name);  // 释放小写类名  zend\_string\_release\_ex(lc\_trait\_name, 0);  // 如果没有找到 trait 或没有链接  if (!exclude\_ce || !(exclude\_ce->ce\_flags & ZEND\_ACC\_LINKED)) {  // 报错：找不到trait  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  // trait 应用检查：trait是否有效 并且 被添加到类里  trait\_num = zend\_check\_trait\_usage(ce, exclude\_ce, traits);  if (!exclude\_tables[trait\_num]) { // 如果排除列表中 找不到此哈希表  ALLOC\_HASHTABLE(exclude\_tables[trait\_num]); // 创建哈希表  // 初始化哈希表  zend\_hash\_init(exclude\_tables[trait\_num], 0, NULL, NULL, 0);  }  // 排除列表中添加此 小写 trait 名  if (zend\_hash\_add\_empty\_element(exclude\_tables[trait\_num], lcname) == NULL) {  // 添加失败，报错：无法计算优先trait。它里面的方法被排除了多次。  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  // 为了一致性，保证trait该应 不是在 排除 trait列表中 定义的。  /\* make sure that the trait method is not from a class mentioned in  exclude\_from\_classes, for consistency \*/  if (trait == exclude\_ce) {  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  }  zend\_string\_release\_ex(lcname, 0); // 释放小写trait名  i++; // 下一个  }  // 处理后把优先列表放回去  ce->trait\_precedences = precedences;  } |

如上所示，当存在优先列表时：

一、先创建排除表（exclude\_tables），排除表是一串哈希表指针，指针数量与trait数量相同。

二、然后遍历优先列表，并对每一个优先（zend\_trait\_precedence实例）做如下操作：

1）检验trait有效性，以及trait中存在此优先指定的方法名，检验失败时报错；

2）遍历每一个优先的排除类列表，并对每一个排除类做如下操作：

a）在运行时类表中查找排除类，找不到时报错；

b）为当前类创建排除哈希表（每个类创建一个）；

c）把排除的方法名放到此类的排除哈希表中；

d）检验，当前trait不在排除列表中，否则报错。

三、处理完后要把优先列表重新关联到trait上。

#### **zend\_check\_trait\_usage()函数**

zend\_check\_trait\_usage()函数用于检查trait是否有效并且已经被添加到类里，代码如下：

|  |
| --- |
| static uint32\_t zend\_check\_trait\_usage(zend\_class\_entry \*ce, zend\_class\_entry \*trait, zend\_class\_entry \*\*traits) {  uint32\_t i;  // 如果不是trait  if (UNEXPECTED((trait->ce\_flags & ZEND\_ACC\_TRAIT) != ZEND\_ACC\_TRAIT)) {  // 报错：此类不是 trait， 只有trait可以在 as 和 insteadof 中使用  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  return 0; // 返回 否  }  for (i = 0; i < ce->num\_traits; i++) { // 遍历所有trait  if (traits[i] == trait) { // 如果已经有了  return i; // 返回编号  }  }  // 没找到，报错：trait 还没有添加进类里  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  return 0;  } |

### 第2步：处理别名列表

处理别名列表：

|  |
| --- |
| **// zend\_traits\_init\_trait\_structures()函数，第3块代码**  if (ce->trait\_aliases) { // 如果 trait 别名指针列表  i = 0;  while (ce->trait\_aliases[i]) { // 给所有有效的别名（zend\_trait\_alias）指针计数  i++;  }  aliases = ecalloc(i, sizeof(zend\_class\_entry\*)); // 分配内存创建别名表    i = 0; // 重置循环变量  while (ce->trait\_aliases[i]) { // 遍历属性别名（zend\_trait\_alias）  // 取出一个别名  zend\_trait\_alias \*cur\_alias = ce->trait\_aliases[i];  // 别名指向的 方法引用  cur\_method\_ref = &ce->trait\_aliases[i]->trait\_method;  // 方法名小写  lcname = zend\_string\_tolower(cur\_method\_ref->method\_name);  if (cur\_method\_ref->class\_name) { **// 如果方法引用有所属类**  // 小写trait名  lc\_trait\_name = zend\_string\_tolower(cur\_method\_ref->class\_name);  // 执行时类表中获取 trait  trait = zend\_hash\_find\_ptr(EG(class\_table), lc\_trait\_name);  zend\_string\_release\_ex(lc\_trait\_name, 0); // 释放小写类名  // 如果没找到trait 或者它还没有链接完成  if (!trait || !(trait->ce\_flags & ZEND\_ACC\_LINKED)) {  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...); // 报错：无法找到trait  }  // trait 应用检查：trait是否有效 并且 被添加到类里  zend\_check\_trait\_usage(ce, trait, traits);  aliases[i] = trait; // trait放到别名列表里  // 如果方法名不在这个 trait 里  if (!zend\_hash\_exists(&trait->function\_table, lcname)) {  // 报错：定义的方法不在trait中  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  } else { **// 如果方法引用没有所属类**  trait = NULL;  for (j = 0; j < ce->num\_traits; j++) { // 遍历所有trait  if (traits[j]) { // 如果元素有效  // 如果trait里能找到此方法  if (zend\_hash\_exists(&traits[j]->function\_table, lcname)) {  if (!trait) { // 如果trait还没找到  trait = traits[j]; // 使用此trait  continue; // 继续找  }  // 报错：有多个trait里包含此方法  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  }  }  // 非绝对方法引用 引用了一个不存在的方法  /\* Non-absolute method reference refers to method that does not exist. \*/  // 如果没找到可用的trait（如何测试这种情况？）  if (!trait) {  if (cur\_alias->alias) { // 如果此别名有指定方法名  // 报错：指定的方法不存在  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  } else {  // 报错：方法的修饰符有变化 ，但此方法不存在  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  }  }  aliases[i] = trait; // trait可用，添加要别名列表里  }  zend\_string\_release\_ex(lcname, 0); // 释放小写类名  i++; // 下一个别名  }  }  \*exclude\_tables\_ptr = exclude\_tables; // 引用返回排除列表  \*aliases\_ptr = aliases; // 引用返回新创建的别名列表  } |

如上所示，当存在别名表（ce->trait\_aliases）时：

一、创建别名指针表（aliases），每个元素是一个类指针（zend\_class\_entry\*），列表长度和别名数量相同；

二、遍历别名表（ce->trait\_aliases），对每个别名分2种情况处理：

情况1，名中指定的方法有所属trait名，进行如下操作：

a）检验别名中指定的方法的所属trait是否在执行时类表中，以及trait是否可用，如果检验失败，报错；

b）把所属trait放进别名列表（aliases）中；

c）检验，如果指定的方法不在此trait中，报错。

情况2，名中指定的方法没有所属trait名，进行如下操作：

a）遍历当前类的所有trait，查找指定方法名的所属trait，找到多个时报错；

b）如果一个没找到，也报错；

c）把找到的所属trait放进别名列表（aliases）中。

完成所有操作后，要把排除表（exclude\_tables）和别名表（aliases）通过引用返回。

## 二）绑定trait中的成员方法

绑定成员方法通过zend\_do\_traits\_method\_binding()函数来实现，函数的代码比较简，但调用到的函数比较多。代码如下：

|  |
| --- |
| // 绑定trait方法 p1:类，p2:trait列表，p3:排除列表，p4:trait别名列表  static void zend\_do\_traits\_method\_binding(zend\_class\_entry \*ce, zend\_class\_entry \*\*traits, HashTable \*\*exclude\_tables, zend\_class\_entry \*\*aliases) {  uint32\_t i;  zend\_string \*key;  zend\_function \*fn;  if (exclude\_tables) { // 如果有排除列表  for (i = 0; i < ce->num\_traits; i++) { // 遍历所有trait  if (traits[i]) { // 如果trait有效  // 遍历方法列表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&traits[i]->function\_table, key,fn){  // 复制trait里的方法  zend\_traits\_copy\_functions(key, fn, ce, exclude\_tables[i], aliases);  } ZEND\_HASH\_FOREACH\_END();  if (exclude\_tables[i]) { // 如果排除表中有此哈希表  zend\_hash\_destroy(exclude\_tables[i]); // 销毁此哈希表  FREE\_HASHTABLE(exclude\_tables[i]); // 释放此哈希表  exclude\_tables[i] = NULL; // 指针清空  }  }  }  } else { // 没有排除列表（两种情况业务逻辑类似）  for (i = 0; i < ce->num\_traits; i++) { // 和上面业务逻辑一样，只肖了个销毁哈希表  if (traits[i]) { // 如果 trait 有效  // 遍历 trait 方法列表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&traits[i]->function\_table,key, fn){  zend\_traits\_copy\_functions(key, fn, ce, NULL, aliases); // 复制trait里的方法  } ZEND\_HASH\_FOREACH\_END();  }  }  }  ZEND\_HASH\_MAP\_FOREACH\_PTR(&ce->function\_table, fn) { // 遍历成员方法表  zend\_fixup\_trait\_method(fn, ce);　// 绑定 trait中的成员方法后，给类添加相应的标记  } ZEND\_HASH\_FOREACH\_END();  } |

如上所示，先分两种情况处理：

1）如果有排除表，遍历类的trait表，对每个trait进行如下操作：

a）遍历trait的所有方法，调用zend\_traits\_copy\_functions()函数，尝试把方法添加到类中；

b）清理属于此trait的排除表。

2）如果没有排除表，遍历类的trait表，遍历trait的所有方法，调用zend\_traits\_copy\_functions()函数，尝试把方法添加到类中。

完成上一步所作后，再遍历类的方法表，对每个方法调用zend\_fixup\_trait\_method()函数，添加相关标记。

### zend\_traits\_copy\_functions()函数

zend\_traits\_copy\_functions()函数用于把trait中的方法复制到类中，代码如下：

|  |
| --- |
| // 复制trait里的方法 p1:方法名，p2:方法，p3:类，p4:排除列表，p5:trait别名列表  static void zend\_traits\_copy\_functions(zend\_string \*fnname, zend\_function \*fn, zend\_class\_entry \*ce, HashTable \*exclude\_table, zend\_class\_entry \*\*aliases) {  zend\_trait\_alias \*alias, \*\*alias\_ptr;  zend\_string \*lcname;  zend\_function fn\_copy;  int i;    // 如果有trait别名列表 （zend\_trait\_alias 指针列表）  if (ce->trait\_aliases) {  alias\_ptr = ce->trait\_aliases; // 别名指针列表  alias = \*alias\_ptr; // 第一个别名 （zend\_trait\_alias 实例）  i = 0; // 循环变量  while (alias) { // 遍历每个别名    // 如果有别名 并且 方法的所属域 与此别名相等 并且方法名匹配  if (alias->alias != NULL && fn->common.scope == aliases[i]  && zend\_string\_equals\_ci(alias->trait\_method.method\_name, fnname)  ) {  fn\_copy = \*fn;  if (alias->modifiers) { // 如果有修饰符  // 把原来的修饰符复制过来，不要可见性限制。  fn\_copy.common.fn\_flags = alias->modifiers |  (fn->common.fn\_flags & ~ZEND\_ACC\_PPP\_MASK);  }  lcname = zend\_string\_tolower(alias->alias); // 小写别名  // 添加trait中的方法， p1:类  zend\_add\_trait\_method(ce, alias->alias, lcname, &fn\_copy);  zend\_string\_release\_ex(lcname, 0); // 释放小写名  }  alias\_ptr++; // 下一个别名指针  alias = \*alias\_ptr; // 别名  i++; // 数量+1  }  }  // 如果没有排除列表 或 排除列表里没有此方法  if (exclude\_table == NULL || zend\_hash\_find(exclude\_table, fnname) == NULL) {  // 复制原 方法  memcpy(&fn\_copy, fn, fn->type == ZEND\_USER\_FUNCTION ?  sizeof(zend\_op\_array) : sizeof(zend\_internal\_function));  if (ce->trait\_aliases) { // 如果有别名列表  alias\_ptr = ce->trait\_aliases; // 列表开头  alias = \*alias\_ptr; // 第一个别名指针  i = 0; // 循环变量  while (alias) { // 如果别名存在  // 别名有效 并且 无修饰符 并且 函数的域与此域相同  if (alias->alias == NULL && alias->modifiers != 0  && fn->common.scope == aliases[i]  && zend\_string\_equals\_ci(alias->trait\_method.method\_name, fnname)  ) {  // 清空可见性  fn\_copy.common.fn\_flags = alias->modifiers |  (fn->common.fn\_flags & ~ZEND\_ACC\_PPP\_MASK);  }  alias\_ptr++; // 下一个别名指针  alias = \*alias\_ptr; // 取出指针  i++; // 数量 +1  }  }  // 添加trait中的方法， p1:类  zend\_add\_trait\_method(ce, fn->common.function\_name, fnname, &fn\_copy);  }  } |

如上所示，业务逻辑可以归纳为两部分：

1）处理别名，遍历所有别名表中的方法，如果找到原方法名与传入的名称相符的方法：为方法创建副本，调用zend\_add\_trait\_method()函数把方法副本添加到类里，用别名作为方法名；

2）如果传入的方法名不在排除列表里：

a）为方法创建副本；

b）遍历所有别名表中的方法，如果找到原方法名与传入的名称相符的方法，清空副本的可见性标记；

c）调用zend\_add\_trait\_method()函数把方法副本添加到类里，使用原方法名。

### zend\_add\_trait\_method()函数

zend\_add\_trait\_method()函数用于把trait中的成员方法添加到类中，代码如下：

|  |
| --- |
| // 添加trait中的方法， p1:类  static void zend\_add\_trait\_method(zend\_class\_entry \*ce, zend\_string \*name, zend\_string \*key, zend\_function \*fn) {  zend\_function \*existing\_fn = NULL;  zend\_function \*new\_fn;  // 如果用key找到方法  if ((existing\_fn = zend\_hash\_find\_ptr(&ce->function\_table, key)) != NULL) {  // 如果新旧方法的操作码是同一串，并且可见性相同 并且 方法已经有 trait 标记  if (existing\_fn->op\_array.opcodes == fn->op\_array.opcodes &&  (existing\_fn->common.fn\_flags & ZEND\_ACC\_PPP\_MASK) == (fn->common.fn\_flags & ZEND\_ACC\_PPP\_MASK) &&  (existing\_fn->common.scope->ce\_flags & ZEND\_ACC\_TRAIT) == ZEND\_ACC\_TRAIT) {  return; // 添加过了，直接返回  }  if (fn->common.fn\_flags & ZEND\_ACC\_ABSTRACT) { // 如果方法带有抽象方法标记  // 对方法进行继承检查  do\_inheritance\_check\_on\_method(  // 返回fn的作用域，如果它属于trait，返回p2  existing\_fn, fixup\_trait\_scope(existing\_fn, ce), fn, fixup\_trait\_scope(fn, ce),  ce, NULL, /\* check\_visibility \*/ 0);  // 抽象方法检查通过就好，不用做其他事  return;  }  // 原有方法属于当前类  if (existing\_fn->common.scope == ce) {  // 当前类方法覆盖trait方法  /\* members from the current class override trait methods \*/    // 中断  return;  // 原有方法属于trait 并且 不是抽象的  } else if (UNEXPECTED((existing\_fn->common.scope->ce\_flags & ZEND\_ACC\_TRAIT)  && !(existing\_fn->common.fn\_flags & ZEND\_ACC\_ABSTRACT))) {  // 报错：两个trait 用同名的方法，产生的冲突  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);    // 如果是继承来的方法，要被trait方法覆盖  } else {    // 对方法进行继承检查 。p1:子方法，p2:子类，p3:父方法，p4:父类，p5:?,p6:?,p7:检查可见性  do\_inheritance\_check\_on\_method(  // 返回fn的作用域，如果它属于trait，返回p2  fn, fixup\_trait\_scope(fn, ce), existing\_fn, fixup\_trait\_scope(existing\_fn, ce),  ce, NULL, /\* check\_visibility \*/ 1);  }  }  // 没有旧方法 或 通过了继承检验  // 此方法是内置函数  if (UNEXPECTED(fn->type == ZEND\_INTERNAL\_FUNCTION)) {  // 创建新内置函数  new\_fn = zend\_arena\_alloc(&CG(arena), sizeof(zend\_internal\_function));  // 复制此方法  memcpy(new\_fn, fn, sizeof(zend\_internal\_function));  // 新方法添加 ARENA 标记  new\_fn->common.fn\_flags |= ZEND\_ACC\_ARENA\_ALLOCATED;  } else {  // 创建新内置函数  new\_fn = zend\_arena\_alloc(&CG(arena), sizeof(zend\_op\_array));  // 复制此方法  memcpy(new\_fn, fn, sizeof(zend\_op\_array));  // 新方法添加 【从trait复制】标记  new\_fn->op\_array.fn\_flags |= ZEND\_ACC\_TRAIT\_CLONE;  // 新方法删除“不可更改”标记  new\_fn->op\_array.fn\_flags &= ~ZEND\_ACC\_IMMUTABLE;  }  // 重新分配方法名，这时它是个别名  /\* Reassign method name, in case it is an alias. \*/  // 更新新函数方法名  new\_fn->common.function\_name = name;  // 方法增加引用次数  function\_add\_ref(new\_fn);  // 更新函数表中名称为 key 的方法  fn = zend\_hash\_update\_ptr(&ce->function\_table, key, new\_fn);  // 添加魔术方法,13个，p1:类，p2:函数，p3:魔术方法名  zend\_add\_magic\_method(ce, fn, key);  } |

如上所示，业务逻辑可以归纳如下：

第一步，如果类中已经有同名方法，进行如下操作：

1）如果新旧方法的操作码地址相同，并且方法可见性相同并且类中的方法已经有trait 标记，表示方法已经添加到类中，中断操作，返回；

2）如果新方法带有抽象方法标记：调用do\_inheritance\_check\_on\_method()数据进行检验，然后中断操作，返回；

3）如果类中的方法属性当前类，无需引用trait中的方法，中断操作，返回；

4）如果类中的方法属于其他trait，并且不是抽象方法：报错，冲突；

5）如果不是以上各种情况，说明方法是继承来的，调用do\_inheritance\_check\_on\_method()函数进行继承检查，并准备覆盖这个方法。

第二步，检查新方法是否是内置方法：

1）如果新方法是内置方法：调用zend\_arena\_alloc()函数分配内存创建方法副本，并给副本添加ZEND\_ACC\_ARENA\_ALLOCATED（通过arena分配内存）标记；

2）如果新方法不是内置方法：调用zend\_arena\_alloc()函数分配内存创建方法副本，并给副本添加ZEND\_ACC\_TRAIT\_CLONE（从trait克隆）标记，再删除副本的ZEND\_ACC\_IMMUTABLE（不可更改）标记。

第三步，更新新副本的方法名，使用传入的方法名（别名）；

第四步，给新副本增加引用次数；

第五步，把新副本添加到当前类的方法表中；

第六步，调用zend\_add\_magic\_method()方法进行针对魔术方法的后期处理。

zend\_add\_magic\_method()方法不是本篇重点，暂不介绍。

### do\_inheritance\_check\_on\_method()

do\_inheritance\_check\_on\_method()函数用于对方法进行继承检查：

|  |
| --- |
| // 对方法进行继承检查 。p1:子方法，p2:子类，p3:父方法，p4:父类，p5:?,p6:?,p7:检查可见性  static zend\_never\_inline void do\_inheritance\_check\_on\_method(  zend\_function \*child, zend\_class\_entry \*child\_scope,  zend\_function \*parent, zend\_class\_entry \*parent\_scope,  zend\_class\_entry \*ce, zval \*child\_zv, bool check\_visibility) {  // 对方法进行继承检查  do\_inheritance\_check\_on\_method\_ex(child, child\_scope, parent, parent\_scope, ce, child\_zv, check\_visibility, 0, 0);  } |

do\_inheritance\_check\_on\_method\_ex()函数已经在“成员方法的继承检查”章节中介绍过。

### fixup\_trait\_scope()函数

fixup\_trait\_scope()函数用于处理trait中的self引用，代码如下：

|  |
| --- |
| static zend\_class\_entry \*fixup\_trait\_scope(const zend\_function \*fn, zend\_class\_entry \*ce) {  // trait中的self应被转换成引用它的类，而不是trait本身  /\* self in trait methods should be resolved to the using class, not the trait. \*/    // 如果所属域是 trait ？使用ce ：使用所属域  return fn->common.scope->ce\_flags & ZEND\_ACC\_TRAIT ? ce : fn->common.scope;  } |

### zend\_fixup\_trait\_method()函数

zend\_fixup\_trait\_method()函数用于在绑定后给类添加相应的标记，代码如下：

|  |
| --- |
| static void zend\_fixup\_trait\_method(zend\_function \*fn, zend\_class\_entry \*ce) {  // 如果方法属于trait  if ((fn->common.scope->ce\_flags & ZEND\_ACC\_TRAIT) == ZEND\_ACC\_TRAIT) {  fn->common.scope = ce; // 如果所属域是当前类  if (fn->common.fn\_flags & ZEND\_ACC\_ABSTRACT) { // 如果是抽象方法  // 给类添加隐匿抽象标记  ce->ce\_flags |= ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS;  }  // 如果方法是用户定义 并且 里面有静态变量  if (fn->type == ZEND\_USER\_FUNCTION && fn->op\_array.static\_variables) {  // 类添加标记：方法中有静态变量  ce->ce\_flags |= ZEND\_HAS\_STATIC\_IN\_METHODS;  }  }  } |

如上所示，如果此方法原本是在trait中的，进行如下操作：

1）方法的作用域更新为当前类；

2）如果方法是抽象的，给类添加ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS标记（隐式抽象类）；

3）如果方法是用户定义 并且 里面有静态变量，给类添加ZEND\_HAS\_STATIC\_IN\_METHODS标记（带有包含静态变量的方法）。

## 三）绑定trait中的常量

zend\_do\_traits\_constant\_binding()函数用于绑定trait中的常量，代码如下（因代码行太长，使用横向排版）:

|  |
| --- |
| static void zend\_do\_traits\_constant\_binding(zend\_class\_entry \*ce, zend\_class\_entry \*\*traits) {  size\_t i;  for (i = 0; i < ce->num\_traits; i++) { // 遍历所有trait  zend\_string \*constant\_name;  zend\_class\_constant \*constant;  if (!traits[i]) { // 如果trait无效  continue; // 跳过  }  // 遍历常量表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&traits[i]->constants\_table, constant\_name, constant) {  // trait 常量检查，名称没有被占用，或类型兼容  if (do\_trait\_constant\_check(ce, constant, constant\_name, traits, i)) {  // 创建新类常量  zend\_class\_constant \*ct = NULL;  ct = zend\_arena\_alloc(&CG(arena),sizeof(zend\_class\_constant));  memcpy(ct, constant, sizeof(zend\_class\_constant)); // 复制原类型常量  constant = ct; // 使用新的类常量  // 如果是表达式常量  if (Z\_TYPE(constant->value) == IS\_CONSTANT\_AST) {  // 当前类去掉标记：已更新类常量  ce->ce\_flags &= ~ZEND\_ACC\_CONSTANTS\_UPDATED;  // 类添加标记：包含表达式常量  ce->ce\_flags |= ZEND\_ACC\_HAS\_AST\_CONSTANTS;  }  constant->ce = ce; // 常量使用新类  Z\_TRY\_ADDREF(constant->value); // 常量值增加引用次数  // 常量有文档注释 ？创建注释副本 ： 空  constant->doc\_comment = constant->doc\_comment ? zend\_string\_copy(constant->doc\_comment) : NULL;  // 常量有修饰属性 并且 修饰属性不是不可替代的  if (constant->attributes && (!(GC\_FLAGS(constant->attributes) & IS\_ARRAY\_IMMUTABLE))) {  GC\_ADDREF(constant->attributes); // 修饰添属性添加引用次数  }  // 更新常量表（后面的会覆盖前面的）  zend\_hash\_update\_ptr(&ce->constants\_table, constant\_name, constant);  }  } ZEND\_HASH\_FOREACH\_END();  }  } |

如上所示，遍历此类的所有有效trait，并遍历每个trait的常量表，对每个常量调用do\_trait\_constant\_check()函数进行检查，对检查结果为真的常量做如下操作：

1）分配内存创建常量的副本；

2）更新副本常量的标记；

3）把副本常量的所属类更新成当前类；

4）给常量的值增加引用次数；

5）给常量的文档创建副本；

6）如果常量有修饰属性并且修饰属性不是不可替代的，增加修饰属性的引用次数；

7）更新当前类的常量列表，把新创建的副本链接到常量表中。

### do\_trait\_constant\_check()函数

do\_trait\_constant\_check()函数用于检查trait中的常量是否可用，代码如下：

|  |
| --- |
| // trait 常量检查，名称没有被占用，以及类型兼容。p1:类，p2:trait常量，p3:常量名，p4:trait表，p5:当前trait序号  static bool do\_trait\_constant\_check(zend\_class\_entry \*ce, zend\_class\_constant \*trait\_constant, zend\_string \*name, zend\_class\_entry \*\*traits, size\_t current\_trait) {  // 初始标记掩码：PPP + final  uint32\_t flags\_mask = ZEND\_ACC\_PPP\_MASK | ZEND\_ACC\_FINAL;  // 在类常量列表中用名称查找常量  zval \*zv = zend\_hash\_find\_known\_hash(&ce->constants\_table, name);  if (zv == NULL) { // 如果没找到  return true; // 返回真  }    zend\_class\_constant \*existing\_constant = Z\_PTR\_P(zv); // 已有常量  // 如果trait常量和 已存在的常量 的标记不同 或 trait属性或常量的值不兼容  if ((ZEND\_CLASS\_CONST\_FLAGS(trait\_constant) & flags\_mask) != (ZEND\_CLASS\_CONST\_FLAGS(existing\_constant) & flags\_mask) ||  // 对值更新后，用===比较，值为对象的话必须指针相同  !check\_trait\_property\_or\_constant\_value\_compatibility(ce, &trait\_constant->value, &existing\_constant->value)) {  // 编译错误：trait A 和 B 定义了同样名称的常量，并且不兼容。  zend\_error\_noreturn(E\_COMPILE\_ERROR,...);  }  return false;  } |

如上所示，业务逻辑可归纳如下：

1）在当前类的常量表中查找此常量，如果找不到，返回真；

2）如果找到已有常量，做如下检查：

a）如果trait常量和已存在的常量的标记不同，报错：常量不兼容；

b）使用check\_trait\_property\_or\_constant\_value\_compatibility()函数检查两个常量的兼容性，如果返回值为否，报错：常量不兼容。

c）如果以上两个条件都未报错，返回假，不报错，但不可以使用trait中的常量。

### check\_trait\_property\_or\_constant\_value\_compatibility()函数

check\_trait\_property\_or\_constant\_value\_compatibility()函数用于的常量值进行兼容性检查，代码如下：

|  |
| --- |
| static bool check\_trait\_property\_or\_constant\_value\_compatibility(zend\_class\_entry \*ce,  zval \*op1, zval \*op2) {  bool is\_compatible;  zval op1\_tmp, op2\_tmp;  // 如果值是常量，需要更新它  /\* if any of the values is a constant, we try to resolve it \*/  if (UNEXPECTED(Z\_TYPE\_P(op1) == IS\_CONSTANT\_AST)) { // 如果op1是常量表达式  ZVAL\_COPY\_OR\_DUP(&op1\_tmp, op1); // 更新op1  zval\_update\_constant\_ex(&op1\_tmp, ce); // 更新常量。p1:常量，p2:更新时用到的域  op1 = &op1\_tmp; // 使用更新后的常量  }  if (UNEXPECTED(Z\_TYPE\_P(op2) == IS\_CONSTANT\_AST)) { // 如果op2是常量表达式  ZVAL\_COPY\_OR\_DUP(&op2\_tmp, op2); // 更新op2  zval\_update\_constant\_ex(&op2\_tmp, ce); // 更新常量。p1:常量，p2:更新时用到的域  op2 = &op2\_tmp; // 使用更新后的常量  }  // op1，op2是否相同（===）  is\_compatible = fast\_is\_identical\_function(op1, op2);  if (op1 == &op1\_tmp) { // 如果op1没有创建副本  zval\_ptr\_dtor\_nogc(&op1\_tmp); // 销毁 它  }  if (op2 == &op2\_tmp) { // 如果op2没有创建副本  zval\_ptr\_dtor\_nogc(&op2\_tmp); // 销毁 它  }  return is\_compatible; // 返回是否兼容  } |

如上所示，业务逻辑可归纳如下：

1）如必要，更新两个常量，并把结果放到临时变量中；

2）调用fast\_is\_identical\_function()函数检查两个常量是否相同；

3）清理临时变量；

4）返回第二步的检查结果。

zval\_update\_constant\_ex()函数和fast\_is\_identical\_function()函数不是本篇重点，暂不介绍。

## 四）绑定trait中的属性（成员变量）

zend\_do\_traits\_property\_binding()函数用于绑定trait中的属性，函数声名如下：

|  |
| --- |
| static void zend\_do\_traits\_property\_binding(zend\_class\_entry \*ce, zend\_class\_entry \*\*traits) {  size\_t i;  zend\_property\_info \*property\_info;  zend\_property\_info \*colliding\_prop;  zend\_property\_info \*new\_prop;  zend\_string\* prop\_name;  zval\* prop\_value;  zend\_string \*doc\_comment; |

业务逻辑如下（因代码行太长，使用横向排版）：

|  |
| --- |
| for (i = 0; i < ce->num\_traits; i++) { // 遍历每个trait  if (!traits[i]) { // 跳过无效的  continue;  }  // 遍历trait的属性信息表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&traits[i]->properties\_info, prop\_name, property\_info) {  uint32\_t flags = property\_info->flags; // 属性信息标记  // 用名称在类中查找属性信息，如果找到  if ((colliding\_prop = zend\_hash\_find\_ptr(&ce->properties\_info, prop\_name)) != NULL) {  **// 如果属性信息是 private 并且不属于当前类**  if ((colliding\_prop->flags & ZEND\_ACC\_PRIVATE) && colliding\_prop->ce != ce) {  zend\_hash\_del(&ce->properties\_info, prop\_name); // 类中删除此属性信息  flags |= ZEND\_ACC\_CHANGED; // 添加 修改过 标记  } else { **// 属性不是private 或 属于当前类**  bool is\_compatible = false; // 没查检通过都算不兼容  // ppp + static + readonly 标记  uint32\_t flags\_mask = ZEND\_ACC\_PPP\_MASK | ZEND\_ACC\_STATIC | ZEND\_ACC\_READONLY;  // 类属性信息与trait属性信息 flag相同 并且 检查属性类型是否兼容（双向可继承）  if ((colliding\_prop->flags & flags\_mask) == (flags & flags\_mask) &&  property\_types\_compatible(property\_info, colliding\_prop) == INHERITANCE\_SUCCESS  ) {  zval \*op1, \*op2;  if (flags & ZEND\_ACC\_STATIC) { // 如果trait属性有 static标记  // 取出两个静态成员  op1 = &ce->default\_static\_members\_table[colliding\_prop->offset];  op2 = &traits[i]->default\_static\_members\_table[property\_info->offset];  ZVAL\_DEINDIRECT(op1); // 两个静态成员解引用  ZVAL\_DEINDIRECT(op2);  } else { // 如果trait属性没有 static标记  // 取出两个普通成员  op1 = &ce->default\_properties\_table[OBJ\_PROP\_TO\_NUM(colliding\_prop->offset)];  op2 = &traits[i]->default\_properties\_table[OBJ\_PROP\_TO\_NUM(property\_info->offset)];  }  // 对值更新后，用===比较，值为对象的话必须指针相同  is\_compatible = check\_trait\_property\_or\_constant\_value\_compatibility(ce, op1, op2);  }  if (!is\_compatible) { // 如果不兼容  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...); // 报错：编译错误：  }  continue;  }  }  // 如果类有只读标记 并且 trait中的属性没有只读标记  if ((ce->ce\_flags & ZEND\_ACC\_READONLY\_CLASS) && !(property\_info->flags & ZEND\_ACC\_READONLY)) {  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...); // 只读类不可引用 包含 非只读属性的 trait  }  if (flags & ZEND\_ACC\_STATIC) { // 如果有静态标记  prop\_value = &traits[i]->default\_static\_members\_table[property\_info->offset]; // 从静态成员列表中获取  ZEND\_ASSERT(Z\_TYPE\_P(prop\_value) != IS\_INDIRECT); // 值不可以是间接引用  } else { // 没有静态标记  // 从默认成员列表中获取  prop\_value = &traits[i]->default\_properties\_table[OBJ\_PROP\_TO\_NUM(property\_info->offset)];  }    Z\_TRY\_ADDREF\_P(prop\_value); // 增加引用次数  // 属性信息有文档注释，创建注释副本  doc\_comment = property\_info->doc\_comment ? zend\_string\_copy(property\_info->doc\_comment) : NULL;  // 属性信息中定义的类型  zend\_type type = property\_info->type;  // 复制类型，如果是列表才复制，不是列表只增加引用次数。列表的旧 zend\_type\_list 留给谁了呢？  zend\_type\_copy\_ctor(&type, /\* persistent \*/ 0);  // 声名 类（不是对象）的静态和动态属性。返回属性信息。传入：类、属性名、属性值、权限、注释、类型（zend\_type）  new\_prop = zend\_declare\_typed\_property(ce, prop\_name, prop\_value, flags, doc\_comment, type);  if (property\_info->attributes) { // 如果有修饰属性  new\_prop->attributes = property\_info->attributes; // 复制修饰属性  if (!(GC\_FLAGS(new\_prop->attributes) & IS\_ARRAY\_IMMUTABLE)) { // 如果新属性不是不可更改  GC\_ADDREF(new\_prop->attributes); // 增加引用次数  }  }  } ZEND\_HASH\_FOREACH\_END();  }  } |

如上所示，遍历此类的所有有效trait，并遍历每个trait的属性信息表，对每个属性信息做如下操作：

一、如果类里已经存在同名属性，分两种情况处理：

情况1，如果属性信息有私有（private）标记并且不属于当前类（继承自父类的私有属性信息），在当前类中删除此属性信息；

情况2，如果属性信息不是私有的或者属于当前类，需要满足以下条件属性才可用：

a）类属性信息与trait属性信息的可见性、static标记、readonly标记都相同，

b）并且property\_types\_compatible()函数验证结果为“继承成功”；

c）使用check\_trait\_property\_or\_constant\_value\_compatibility()函数检测类中的属性和trait中的属性的兼容性，结果为真（静态属性需要使用ZVAL\_DEINDIRECT()宏程序追踪引用）。

以上3个条件任何一个不满足，报错：类和trait属性不兼容；

二、检验，如果类有只读标记并且trait属性没有只读标记，报错：只读类不可引用包含非只读属性的trait；

三、给trait属性的值增加引用计数；

四、调用zend\_declare\_typed\_property()函数给当前类创建新属性信息；

五、处理修饰属性（attributes），如果新属性信息包含修饰属性，并且修饰属性不可更改（有IS\_ARRAY\_IMMUTABLE标记），给修饰属性增加引用次数。

property\_types\_compatible()函数已经在“类的继承”章节中介绍过。

ZVAL\_DEINDIRECT()宏程序的详细介绍参见“类型篇”。

zend\_declare\_typed\_property()函数用于给类创建属性信息，代码量大且业务逻辑比较简单，放在附录中供参考。

# 七、类的绑定和链接

## 一）类的绑定

在第章“基本概念”中已经介绍过类的绑定和链接过程，类的绑定通过Zend/zend\_compile.c中的do\_bind\_class()函数来实现，函数代码如下：

|  |
| --- |
| ZEND\_API zend\_result do\_bind\_class(zval \*lcname, zend\_string \*lc\_parent\_name) {  zend\_class\_entry \*ce;  zval \*rtd\_key, \*zv;  rtd\_key = lcname + 1; // rtd\_key是小写名字的下一个变量  // 在执行时类表中查找 rtd\_key  zv = zend\_hash\_find\_known\_hash(EG(class\_table), Z\_STR\_P(rtd\_key));  if (UNEXPECTED(!zv)) { // 如果zv无效， 找不到 rtd\_key  // 在类列表中用小写名查找  ce = zend\_hash\_find\_ptr(EG(class\_table), Z\_STR\_P(lcname));  ZEND\_ASSERT(ce); // 这次必须找到  // 报错：不可以重复定义 类或接口或trait （三种都测试过）  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...);  return FAILURE;  }  // 注册派生类  return zend\_bind\_class\_in\_slot(zv, lcname, lc\_parent\_name) ? SUCCESS : FAILURE;  } |

zend\_bind\_class\_in\_slot()函数用于注册派生类，代码如下：

|  |
| --- |
| // ing3, 构建类，并添加到类表中，p1:类 zval，p2:类名，p3:父类名  ZEND\_API zend\_class\_entry \*zend\_bind\_class\_in\_slot(  zval \*class\_table\_slot, zval \*lcname, zend\_string \*lc\_parent\_name) {  zend\_class\_entry \*ce = Z\_PTR\_P(class\_table\_slot); // 取出类指针  // 类有 ZEND\_ACC\_PRELOADED 标记，并且编译选项没有 ZEND\_COMPILE\_PRELOAD 标记  bool is\_preloaded = (ce->ce\_flags & ZEND\_ACC\_PRELOADED) &&  !(CG(compiler\_options) & ZEND\_COMPILE\_PRELOAD);  bool success; // 操作是否成功    if (EXPECTED(!is\_preloaded)) { // 不是预加载  // 在哈希表中给类重新设置 key（类名）  success = zend\_hash\_set\_bucket\_key(EG(class\_table), (Bucket\*) class\_table\_slot, Z\_STR\_P(lcname)) != NULL;  } else {  // 类添加到执行时类表里  success = zend\_hash\_add\_ptr(EG(class\_table), Z\_STR\_P(lcname), ce) != NULL;  }  if (UNEXPECTED(!success)) { // 如果没有成功  zend\_error\_noreturn(E\_COMPILE\_ERROR, ...); // 报错：无法定义类，因为同名类已存在  return NULL;  }  if (ce->ce\_flags & ZEND\_ACC\_LINKED) { // 如果类已链接完成  // 依次调用每个链接回调 函数  zend\_observer\_class\_linked\_notify(ce, Z\_STR\_P(lcname));  return ce; // 返回类  }  // 链接类 : zend\_inheritance.c  **ce = zend\_do\_link\_class(ce, lc\_parent\_name, Z\_STR\_P(lcname));**  if (ce) { // 如果链接成功  ZEND\_ASSERT(!EG(exception)); // 不能有异常  // 依次调用每个 链接回调 函数  zend\_observer\_class\_linked\_notify(ce, Z\_STR\_P(lcname));  return ce; // 返回类  }  if (!is\_preloaded) { // 不是预加载  // 类表中找到这个类  zval \*zv = zend\_hash\_find(EG(class\_table), Z\_STR\_P(lcname));  // 类表中给这个类 重新设置 key（类名）  zend\_hash\_set\_bucket\_key(EG(class\_table), (Bucket \*) zv, Z\_STR\_P(lcname + 1));  } else { // 是预加载  zend\_hash\_del(EG(class\_table), Z\_STR\_P(lcname)); // 执行时类表中删除这个类  }  return NULL;  } |

如上所示，所有没有ZEND\_ACC\_LINKED（链接完成）标记的类，都需要调用zend\_do\_link\_class()函数进行链接。

zend\_observer\_class\_linked\_notify()函数不是本章重点，暂不介绍。

## 二）类的链接

链接类通过zend\_do\_link\_class()函数来实现，函数声名如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第1块代码**  ZEND\_API zend\_class\_entry \*zend\_do\_link\_class(zend\_class\_entry \*ce, zend\_string \*lc\_parent\_name, zend\_string \*key) {    zend\_class\_entry \*parent = NULL;  zend\_class\_entry \*\*traits\_and\_interfaces = NULL;  zend\_class\_entry \*proto = NULL;  zend\_class\_entry \*orig\_linking\_class;  uint32\_t is\_cacheable = ce->ce\_flags & ZEND\_ACC\_IMMUTABLE; // 获取不可更改标记  uint32\_t i, j;  zval \*zv;  ALLOCA\_FLAG(use\_heap)  SET\_ALLOCA\_FLAG(use\_heap);  ZEND\_ASSERT(!(ce->ce\_flags & ZEND\_ACC\_LINKED)); // 类必须未链接过 |

函数的业务逻辑比较复杂，分为以下步骤：

### 第一步：如果有父类名，加载父类

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第2块代码**  if (ce->parent\_name) {  // 用类名查找类， 允许真正链接过的类 ，允许exception  parent = zend\_fetch\_class\_by\_name(  ce->parent\_name, lc\_parent\_name,  ZEND\_FETCH\_CLASS\_ALLOW\_NEARLY\_LINKED  | ZEND\_FETCH\_CLASS\_EXCEPTION);  if (!parent) { // 如果父类无效  check\_unrecoverable\_load\_failure(ce); // 检查不可恢复的加载失败  return NULL;  }  // is\_cacheable &= (ce)->ce\_flags;  UPDATE\_IS\_CACHEABLE(parent); // 更新缓存标记  } |

zend\_fetch\_class\_by\_name()函数已经在“类的常用操作”章节中介绍过。

#### **check\_unrecoverable\_load\_failure()函数**

check\_unrecoverable\_load\_failure()函数用于检查不可恢复的加载失败，业务逻辑简单，代码如下：

|  |
| --- |
| static void check\_unrecoverable\_load\_failure(zend\_class\_entry \*ce) {  // 如果有 未链接的 uses  if (CG(unlinked\_uses)  // 从未链接的 uses 中 删除此类  && zend\_hash\_index\_del(CG(unlinked\_uses), (zend\_long)(zend\_uintptr\_t)ce) == SUCCESS) {  // 抛错：在有变异从属的继承中  zend\_exception\_uncaught\_error(...);  }  } |

### 第二步：如果有引用trait ，加载trait

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第3块代码**  if (ce->num\_traits || ce->num\_interfaces) {  // 开辟内存创建 trait和接口列表  traits\_and\_interfaces = do\_alloca(sizeof(zend\_class\_entry\*) \* (ce->num\_traits + ce->num\_interfaces), use\_heap);  for (i = 0; i < ce->num\_traits; i++) { // 遍历 所有 trait  // 查找trait，找不到会报错。p1:类名，p2:key，p3:flags  zend\_class\_entry \*trait = zend\_fetch\_class\_by\_name(ce->trait\_names[i].name,  ce->trait\_names[i].lc\_name, ZEND\_FETCH\_CLASS\_TRAIT);  if (UNEXPECTED(trait == NULL)) { // 如果没找到 此trait  free\_alloca(traits\_and\_interfaces, use\_heap); // 释放 临时变量  return NULL; // 返回null  }  // 如果 找到的不是trait  if (UNEXPECTED(!(trait->ce\_flags & ZEND\_ACC\_TRAIT))) {  zend\_error\_noreturn(E\_ERROR, ...); // 报错，不可以引用，它不是trait  free\_alloca(traits\_and\_interfaces, use\_heap); // 释放 临时变量  return NULL; // 返回null  }  for (j = 0; j < i; j++) { // 从临时列表中查找 此 trait  if (traits\_and\_interfaces[j] == trait) { // 如果找到  trait = NULL; // 清空临时变量  break; // 已经在列表中，不重复插入  }  }    traits\_and\_interfaces[i] = trait; // 把 trait 加到临时列表中（只留最后一个）  if (trait) { // 如果trait有效  // 更新缓存标记, is\_cacheable &= (ce)->ce\_flags;  UPDATE\_IS\_CACHEABLE(trait);  }  }  } |

### 第三步：如果有实现接口，加载接口

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第4块代码**  if (ce->num\_interfaces) {  for (i = 0; i < ce->num\_interfaces; i++) { // 遍历 每个接口  // 获取类，找不到会报错。p1:类名，p2:key，p3:flags  // 通过名称获取接口 标记：获取接口 | 允许将要链接完成的类 | 允许exception  zend\_class\_entry \*iface = zend\_fetch\_class\_by\_name(  ce->interface\_names[i].name, ce->interface\_names[i].lc\_name,  ZEND\_FETCH\_CLASS\_INTERFACE |  ZEND\_FETCH\_CLASS\_ALLOW\_NEARLY\_LINKED | ZEND\_FETCH\_CLASS\_EXCEPTION);  if (!iface) { // 如果没找到接口  check\_unrecoverable\_load\_failure(ce); // 检查 不可恢复的加载失败  free\_alloca(traits\_and\_interfaces, use\_heap); // 释放列表  return NULL;  }  // 把接口加入到trait和接口列表中  traits\_and\_interfaces[ce->num\_traits + i] = iface;  if (iface) { // 如果接口有效  // 更新 缓存 标记, is\_cacheable &= (ce)->ce\_flags;  UPDATE\_IS\_CACHEABLE(iface);  }  }  } |

### 第四步：尝试使用继承缓存

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第5块代码**  bool orig\_record\_errors = EG(record\_errors); // 原来是否需要记录错误  // 如果类 是 不可更改 并且 可缓存  if (ce->ce\_flags & ZEND\_ACC\_IMMUTABLE && is\_cacheable) {  // 如果有 zend\_inheritance\_cache\_get 和 zend\_inheritance\_cache\_add 方法  // 这两个方法在 \ext\opcache\ZendAccelerator.c 中定义，默认没有  if (zend\_inheritance\_cache\_get && zend\_inheritance\_cache\_add) {  // 取得继承缓存  zend\_class\_entry \*ret = zend\_inheritance\_cache\_get(ce, parent, traits\_and\_interfaces);  if (ret) { // 如果获取成功  if (traits\_and\_interfaces) { // 如果有trait和接口列表  free\_alloca(traits\_and\_interfaces, use\_heap); // 释放trait和接口列表  }  // 从编译类表中取得此类  zv = zend\_hash\_find\_known\_hash(CG(class\_table), key);  Z\_CE\_P(zv) = ret; // 把zv的指针指向 刚获取到的缓存类  return ret; // 返回缓存类  }  zend\_begin\_record\_errors(); // 开启记录错误信息  } else { // 如果没有两个缓存方法  is\_cacheable = 0; // 不可使用缓存  }  proto = ce; // ce作为原型  } |

如上，zend\_inheritance\_cache\_get()函数用于读取类缓存， zend\_inheritance\_cache\_add()函数用于写入类缓存，数这两个函数在加速器（ext/opcache/ZendAccelerator.c）中定义，不在PHP内核中。缓存部分非本篇重点，暂不详细介绍。

### 第五步：加载本类

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第6块代码**  zend\_try {  if (ce->ce\_flags & ZEND\_ACC\_IMMUTABLE) { // 如果是不可更改  ce = zend\_lazy\_class\_load(ce); // 懒加载当前类  // 从编译类表中获取此类  zv = zend\_hash\_find\_known\_hash(CG(class\_table), key);  Z\_CE\_P(zv) = ce; // 更把类更新到列表元素中  // 如果不是不可更改，并且可使用文件缓存  } else if (ce->ce\_flags & ZEND\_ACC\_FILE\_CACHED) {  ce = zend\_lazy\_class\_load(ce); // 懒加载当前类  ce->ce\_flags &= ~ZEND\_ACC\_FILE\_CACHED; // 删除文件缓存标记  // 从编译类表中获取此类  zv = zend\_hash\_find\_known\_hash(CG(class\_table), key);  Z\_CE\_P(zv) = ce; // 更把类更新到列表元素中  }  if (CG(unlinked\_uses)) { // 如果有未链接的 uses  // 从未链接的引用表中删除此类  zend\_hash\_index\_del(CG(unlinked\_uses), (zend\_long)(zend\_uintptr\_t) ce);  }  orig\_linking\_class = CG(current\_linking\_class); // 原本正在链接的类  // 可缓存时，记录当前类，否则 null  CG(current\_linking\_class) = is\_cacheable ? ce : NULL; |

zend\_lazy\_class\_load()函数用于给类修建副本，并应用副本，详细介绍参见“懒加载类”章节。

CG(unlinked\_uses)哈希表中存放使用到但还未链接完成的类，如果此类在链接完成前被查找过，就会加入这个列表中，具体业务逻辑见zend\_lookup\_class\_ex()函数。

### 第六步：如果此类是枚举，注册成枚举

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第7块代码**  if (ce->ce\_flags & ZEND\_ACC\_ENUM) {  zend\_enum\_register\_funcs(ce);  } |

zend\_enum\_register\_funcs()函数在Zend/zend\_enum.c文件中定义，非本篇重点，暂不介绍。

### 第七步：如果有父类，执行承继

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第8块代码**  if (parent) {  if (!(parent->ce\_flags & ZEND\_ACC\_LINKED)) { // 如果父类没有链接成功  // 添加从属责任。（从属责任对象关关联到从属类）  add\_dependency\_obligation(ce, parent);  }  zend\_do\_inheritance(ce, parent); // 执行继承  } |

add\_dependency\_obligation()函数用于添加从属责任，详情参见“变异责任”章节，zend\_do\_inheritance()函数在“类的继承”章节中介绍过。

### 第八步：如果有引用trait，绑定trait

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第9块代码**  if (ce->num\_traits) {  zend\_do\_bind\_traits(ce, traits\_and\_interfaces); // 绑定trait，p1:类，p2:trait列表  } |

zend\_do\_bind\_traits()函数在“类中引用trait”章节中介绍过。

### 第九步：如果有实现接口, 处理这些接口

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第10块代码**  if (ce->num\_interfaces) {  // 父类接口数量  uint32\_t num\_parent\_interfaces = parent ? parent->num\_interfaces : 0;  // 分配接口列表  zend\_class\_entry \*\*interfaces = emalloc( sizeof(zend\_class\_entry \*) \*  (ce->num\_interfaces + num\_parent\_interfaces));  if (num\_parent\_interfaces) { // 如果有父类接口  // 先把父类接口复制过来  memcpy(interfaces, parent->interfaces,  sizeof(zend\_class\_entry \*) \* num\_parent\_interfaces);  }  // 再复制 traits\_and\_interfaces 中的接口  memcpy(interfaces + num\_parent\_interfaces, traits\_and\_interfaces + ce->num\_traits, sizeof(zend\_class\_entry \*) \* ce->num\_interfaces);  zend\_do\_implement\_interfaces(ce, interfaces); // 实现接口列表中的接口  // 如果子类没有实现接口，并且父类有接口  } else if (parent && parent->num\_interfaces) {  zend\_do\_inherit\_interfaces(ce, parent); // 继承父类的接口  } |

zend\_do\_implement\_interfaces()函数用于实现接口，在“类实现接口”章节中介绍过。

### 第十步：检验是否是抽象类

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第11块代码**  if (!(ce->ce\_flags & (ZEND\_ACC\_INTERFACE|ZEND\_ACC\_TRAIT))  && (ce->ce\_flags & (ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS|ZEND\_ACC\_EXPLICIT\_ABSTRACT\_CLASS))  ) {  // 检验是否需要标记成抽象类，或实现某些抽象方法  zend\_verify\_abstract\_class(ce);  } |

#### **zend\_verify\_abstract\_class()函数**

zend\_verify\_abstract\_class()函数用来检查此类在声名抽象方法时是否正确，核心业务逻辑如下：

|  |
| --- |
| void zend\_verify\_abstract\_class(zend\_class\_entry \*ce) {  zend\_function \*func;  zend\_abstract\_info ai;  // 是否是显式抽象类  bool is\_explicit\_abstract = (ce->ce\_flags & ZEND\_ACC\_EXPLICIT\_ABSTRACT\_CLASS) != 0;  // 不是枚举，才可以是抽象类  bool can\_be\_abstract = (ce->ce\_flags & ZEND\_ACC\_ENUM) == 0;  memset(&ai, 0, sizeof(ai)); // 清空临时抽象信息  ZEND\_HASH\_MAP\_FOREACH\_PTR(&ce->function\_table, func) { // 遍历方法表  if (func->common.fn\_flags & ZEND\_ACC\_ABSTRACT) { // 如果方法是抽象的  // 如果不是显式抽象 或 是private  if (!is\_explicit\_abstract || (func->common.fn\_flags & ZEND\_ACC\_PRIVATE)) {  zend\_verify\_abstract\_class\_function(func, &ai); // 检验抽象类方法  }  }  } ZEND\_HASH\_FOREACH\_END();  if (ai.cnt) { // 如果有抽象类方法  zend\_error\_noreturn(E\_ERROR, ); // 报错：抽象方法使用错误  } else { // 没有抽象类方法  // 删除 隐式抽象类标记  ce->ce\_flags &= ~ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS;  }  } |

zend\_verify\_abstract\_class\_function()函数用于创建错误提示信息，记录，非本篇重点，暂不介绍。

### 第十一步：如果类是枚举类型，检验枚举

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第12块代码**  if (ce->ce\_flags & ZEND\_ACC\_ENUM) {  // 检验 枚举 （属性，魔术方法，接口）  zend\_verify\_enum(ce);  } |

zend\_verify\_enum()函数在Zend/zend\_enum.c文件中定义，非本篇重点，暂不介绍。

### 第十二步：处理 \_\_tostring 方法

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第13块代码**  // Stringable接口通常 是在编译中添加的。但是如果它来自于trait，需要添加此接口  // 如果有\_\_tostring 并且 此类不是trait 并且 没有实现 zend\_ce\_stringable 接口  if (ce->\_\_tostring && !(ce->ce\_flags & ZEND\_ACC\_TRAIT)  && !zend\_class\_implements\_interface(ce, zend\_ce\_stringable)) {  // 进到这里说明 \_\_tostring 是从trait里引用的  ZEND\_ASSERT(ce->\_\_tostring->common.fn\_flags & ZEND\_ACC\_TRAIT\_CLONE);  ce->ce\_flags |= ZEND\_ACC\_RESOLVED\_INTERFACES; // 添加 【完成接口】 标记  ce->num\_interfaces++; // 接口数量 +1  // 调整接口列表 内存大小  ce->interfaces = perealloc(ce->interfaces, sizeof(zend\_class\_entry \*)  \* ce->num\_interfaces, ce->type == ZEND\_INTERNAL\_CLASS);  // 添加接口 zend\_ce\_stringable  ce->interfaces[ce->num\_interfaces - 1] = zend\_ce\_stringable;  **do\_interface\_implementation(ce, zend\_ce\_stringable); // 实现Stringable接口**  }  zend\_build\_properties\_info\_table(ce); // 构造属性信息表  } zend\_catch { // 如果出错  if (!orig\_record\_errors) { // 如果有原来的错误记录  EG(record\_errors) = false; // 停止记录错误信息  zend\_free\_recorded\_errors(); // 释放错误记录  }  zend\_bailout(); // 跳伞  } zend\_end\_try(); |

如上所示，Stringable接口通常 是在编译中添加的。但是如果\_\_tostring方法它来自于trait，前面没有实现过Stringable接口，这时需要添加Stringable接口。

#### **zend\_build\_properties\_info\_table()函数**

zend\_build\_properties\_info\_table()函数用于创建类的属性信息表：

|  |
| --- |
| // 构造属性信息表  void zend\_build\_properties\_info\_table(zend\_class\_entry \*ce) {  zend\_property\_info \*\*table, \*prop;  size\_t size;  if (ce->default\_properties\_count == 0) { // 如果默认属性数量为0，直接返回  return;  }  ZEND\_ASSERT(ce->properties\_info\_table == NULL); // 这时属性信息表还未创建  // 信息表大小 = 信息指针大小 \* 默认属性个数  size = sizeof(zend\_property\_info \*) \* ce->default\_properties\_count;  if (ce->type == ZEND\_USER\_CLASS) { // 如果是用户类  // 使用 arena 分配内存  ce->properties\_info\_table = table = zend\_arena\_alloc(&CG(arena), size);  } else { // 内置类  ce->properties\_info\_table = table = pemalloc(size, 1); // 使用malloc()函数配内存  }  memset(table, 0, size); // 已删除的slots 可能还存在，所以继承时要清理干净  // 如果有父类 并且 父类的属性信息不是0  if (ce->parent && ce->parent->default\_properties\_count != 0) {  // 父类的属性信息表  zend\_property\_info \*\*parent\_table = ce->parent->properties\_info\_table;  // 先把父类的属性信息表复制过来  memcpy(table, parent\_table,  sizeof(zend\_property\_info \*) \* ce->parent->default\_properties\_count );    // 如果子类没有新加属性，这里就完成了  if (ce->default\_properties\_count == ce->parent->default\_properties\_count) {  return;  }  }  // 遍历子类属性信息  ZEND\_HASH\_MAP\_FOREACH\_PTR(&ce->properties\_info, prop) {  // 处理子类属性  // 属性信息属于子类 并且 没有 static 标记  if (prop->ce == ce && (prop->flags & ZEND\_ACC\_STATIC) == 0) {  // 把属性信息添加进列表里  // 从偏移量获取 当前对象的(zval)序号  table[OBJ\_PROP\_TO\_NUM(prop->offset)] = prop;  }  } ZEND\_HASH\_FOREACH\_END();  } |

如上所示，业务逻辑可归纳成如下步骤：

1）如果当前类没有默认属性，直接返回；

2）分配内存创建属性信息表；

3）把父类的属性信息复制到新属性信息表中；

4）遍历此类的所有属性信息，如果此属性信息不是静态属性，把对属性信息的间接引用转成直接引用。

#### **OBJ\_PROP\_TO\_NUM()宏程序**

OBJ\_PROP\_TO\_NUM()宏程序及相关宏程序代码如下：

|  |
| --- |
| // 使用序号属性表中的偏移量  #define OBJ\_PROP\_TO\_OFFSET(num) \  ((uint32\_t)(XtOffsetOf(zend\_object, properties\_table) + sizeof(zval) \* (num)))  // 把计算出的偏移量转成zval偏移数量  #define OBJ\_PROP\_TO\_NUM(offset) \  ((offset - OBJ\_PROP\_TO\_OFFSET(0)) / sizeof(zval))  // 取得一个元素在结构体中的偏量  # define XtOffsetOf(s\_type, field) offsetof(s\_type, field) |

这部分用到zend\_object结构体，更多介绍参见“面向对象篇（二）”。

### 第十三步：处理变异责任

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第14块代码**  EG(record\_errors) = orig\_record\_errors; // 执行时记录 错误信息  if (!(ce->ce\_flags & ZEND\_ACC\_UNRESOLVED\_VARIANCE)) { // 如果没有未处理的变异责任  ce->ce\_flags |= ZEND\_ACC\_LINKED; // 链接完毕  } else { // 有未处理的变异责任  ce->ce\_flags |= ZEND\_ACC\_NEARLY\_LINKED; // 添加【即将链接完毕】标记  if (CG(current\_linking\_class)) { // 如果有正在链接的类  ce->ce\_flags |= ZEND\_ACC\_CACHEABLE; // 添加 cacheable标记  }  load\_delayed\_classes(ce); // 加载延时类  // 如果有未完成的变异责任  if (ce->ce\_flags & ZEND\_ACC\_UNRESOLVED\_VARIANCE) {  **resolve\_delayed\_variance\_obligations(ce); // 处理延时变异责任（主要是检查责任）**  }  if (ce->ce\_flags & ZEND\_ACC\_CACHEABLE) { // 如果有可缓存标记  ce->ce\_flags &= ~ZEND\_ACC\_CACHEABLE; // 删除 ZEND\_ACC\_CACHEABLE 标记  } else { // 否则  CG(current\_linking\_class) = NULL; // 清空当前正在链接的类  }  }  if (!CG(current\_linking\_class)) { // 如果没有正在链接的类  is\_cacheable = 0; // 不可缓存  }  // 还原正在链接的类  CG(current\_linking\_class) = orig\_linking\_class; |

resolve\_delayed\_variance\_obligations()函数用于处理前面添加好的延时责任，详情参见“延时变异责任”章节。

#### **load\_delayed\_classes()函数**

load\_delayed\_classes()函数用于延时加载类，代码如下：

|  |
| --- |
| static void load\_delayed\_classes(zend\_class\_entry \*ce) {  // 延时自动加载列表  HashTable \*delayed\_autoloads = CG(delayed\_autoloads);  // 如果列表无效，直接返回  if (!delayed\_autoloads) {  return;  }  HashPosition pos = 0; // 表位置  zend\_string \*name; // 返回字串key  zend\_ulong idx; // 返回索引号  // 如果当前key有效 （不是 HASH\_KEY\_NON\_EXISTENT ）  while (zend\_hash\_get\_current\_key\_ex(delayed\_autoloads, &name, &idx, &pos)  != HASH\_KEY\_NON\_EXISTENT) {  zend\_string\_addref(name); // 类名称增加引用次数  zend\_hash\_del(delayed\_autoloads, name); // 通过类名称删除此元素  zend\_lookup\_class(name); // 查找并加载此类  zend\_string\_release(name); // 释放名称  if (EG(exception)) { // 如果有异常  // 抛出异常：在继承此类是，自动加载 类 \*\*  zend\_exception\_uncaught\_error( "During inheritance of %s, while autoloading %s", ZSTR\_VAL(ce->name), ZSTR\_VAL(name));  }  }  } |

如上所示，遍历编译时延时自动加载列表CG(delayed\_autoloads)，调用zend\_lookup\_class()函数加载每个类，并把元素从列表中删除。

### 第十四步：尝试使用继承缓存

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第15块代码**  if (is\_cacheable) { // 如果可缓存  HashTable \*ht = (HashTable\*)ce->inheritance\_cache; // 继承缓存表  zend\_class\_entry \*new\_ce;  ce->inheritance\_cache = NULL; // 初始为空  // 添加继承缓存  new\_ce = zend\_inheritance\_cache\_add(ce, proto, parent, traits\_and\_interfaces, ht);  if (new\_ce) { // 如果添加缓存成功  // 找到编译类表中的类  zv = zend\_hash\_find\_known\_hash(CG(class\_table), key);  ce = new\_ce; // 使用缓存返回的新实例  Z\_CE\_P(zv) = ce; // 更新类表中的元素  }  if (ht) { // 如果有继承缓存表  zend\_hash\_destroy(ht); // 销毁继承缓存表  FREE\_HASHTABLE(ht); // 释放继承缓存表  }  }    if (!orig\_record\_errors) { // 如果原来不要求记录错误  zend\_free\_recorded\_errors(); // 释放错误记录  }    if (traits\_and\_interfaces) { // 如果有trait和接口列表  free\_alloca(traits\_and\_interfaces, use\_heap); // 释放trait和接口列表  }    if (ZSTR\_HAS\_CE\_CACHE(ce->name)) { // 如果有类缓存  ZSTR\_SET\_CE\_CACHE(ce->name, ce); // 更新类缓存  }  return ce;  } |

zend\_inheritance\_cache\_add()函数在加速器（ext/opcache/ZendAccelerator.c）中定义，非本篇重点，暂不介绍。

# 八、早期绑定

早期绑定通过zend\_try\_early\_bind()函数来实现，函数声名如下：

|  |
| --- |
| **// zend\_try\_early\_bind()函数，第1块代码**  ZEND\_API zend\_class\_entry \*zend\_try\_early\_bind(zend\_class\_entry \*ce, zend\_class\_entry \*parent\_ce, zend\_string \*lcname, zval \*delayed\_early\_binding) {  inheritance\_status status;  zend\_class\_entry \*proto = NULL;  zend\_class\_entry \*orig\_linking\_class;  // 检查不可更改标记，不可更改的可以被缓存。  uint32\_t is\_cacheable = ce->ce\_flags & ZEND\_ACC\_IMMUTABLE;  UPDATE\_IS\_CACHEABLE(parent\_ce); // is\_cacheable &= (ce)->ce\_flags; |

其中delayed\_early\_binding参数是ce所在的哈希表值（zval）的指针，在操作码缓存功能ext/opcache/zend\_accelerator\_util\_funcs.c文件中调用zend\_try\_early\_bind()函数时用到，在本篇的其他地方没有用到。

函数的业务逻辑比较复杂，分为以下步骤：

## 第一步：尝试从缓存读取类

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第2块代码**  // 如果可以缓存  if (is\_cacheable) {  // 如果有缓存管理方法 ,默认没有。ext\opcache\ZendAccelerator.c 中添加  if (zend\_inheritance\_cache\_get && zend\_inheritance\_cache\_add) {  // 从缓存中获取类入口  zend\_class\_entry \*ret = zend\_inheritance\_cache\_get(ce, parent\_ce, NULL);  if (ret) { // 如果找到了  // 注册已早期绑定的类  if (UNEXPECTED(!register\_early\_bound\_ce(delayed\_early\_binding,  lcname, ret))) {  return NULL; // 不成功返回null  }  zend\_observer\_class\_linked\_notify(ret, lcname); // 依次调用每个链接回调函数  return ret; // 返回缓存中的类入口  }  } else { // 没有缓存管理方法  is\_cacheable = 0; // 不能缓存  }  proto = ce; // 缓存获取失败，ce作为原型  } |

如上，zend\_inheritance\_cache\_get()函数用于读取类缓存， zend\_inheritance\_cache\_add()函数用于写入类缓存，数这两个函数在加速器（ext/opcache/ZendAccelerator.c）中定义，不在PHP内核中。缓存部分非本篇重点，暂不详细介绍。

## 第二步：检验类的继承状态

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第3块代码**  orig\_linking\_class = CG(current\_linking\_class); // 原本正在链接的类  CG(current\_linking\_class) = NULL; // 清空正在链接的类  status = zend\_can\_early\_bind(ce, parent\_ce); // 检查所有属性和方法，在继承上是否有问题  CG(current\_linking\_class) = orig\_linking\_class; // 恢复正在链接的类 |

### zend\_can\_early\_bind()

zend\_can\_early\_bind()函数用于检查所有属性和方法，在继承上是否有问题，代码如下：

|  |
| --- |
| static inheritance\_status zend\_can\_early\_bind(zend\_class\_entry \*ce, zend\_class\_entry \*parent\_ce) {  zend\_string \*key;  zend\_function \*parent\_func;  zend\_property\_info \*parent\_info;  inheritance\_status overall\_status = INHERITANCE\_SUCCESS;  // 遍历 父类方法表  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&parent\_ce->function\_table, key, parent\_func) {  // 在子类方法表中查找此方法  zval \*zv = zend\_hash\_find\_known\_hash(&ce->function\_table, key);  if (zv) { // 如果找到  zend\_function \*child\_func = Z\_FUNC\_P(zv);  // 对方法进行继承检查  inheritance\_status status =  do\_inheritance\_check\_on\_method\_ex(child\_func, child\_func->common.scope,  parent\_func, parent\_func->common.scope,  ce, NULL, /\* check\_visibility \*/ 1, 1, 0);  if (UNEXPECTED(status == INHERITANCE\_WARNING)) { // 如果状态是：继承警告  overall\_status = INHERITANCE\_WARNING;  } else if (UNEXPECTED(status != INHERITANCE\_SUCCESS)) { // 状态不是：继承成功  return status;  }  // 未处理的情况是 INHERITANCE\_SUCCESS：什么也不做  }  } ZEND\_HASH\_FOREACH\_END();  // 遍历所有属性信息  ZEND\_HASH\_MAP\_FOREACH\_STR\_KEY\_PTR(&parent\_ce->properties\_info,  key, parent\_info) {  zval \*zv;  // 如果父属性信息是 private 或 有规定类型  if ((parent\_info->flags & ZEND\_ACC\_PRIVATE) || !ZEND\_TYPE\_IS\_SET(parent\_info->type)) {  continue; // 下一个  }  // 在子类属性信息表中查找这个 信息  zv = zend\_hash\_find\_known\_hash(&ce->properties\_info, key);  if (zv) { // 如果找到  zend\_property\_info \*child\_info = Z\_PTR\_P(zv); // 取出子类属性信息  if (ZEND\_TYPE\_IS\_SET(child\_info->type)) { // 如果它有规定类型  // 检查属性类型是否兼容（双向可继承）  inheritance\_status status = property\_types\_compatible(parent\_info, child\_info);  if (UNEXPECTED(status != INHERITANCE\_SUCCESS)) { // 如果状态不是继承成功  return status; // 中断，返回此状态  }  }  }  } ZEND\_HASH\_FOREACH\_END();  return overall\_status; // 返回方法覆盖状态  } |

如上所示，业务逻辑可以归纳如下：

一、总状态（overall\_status）默认是“继承成功（ INHERITANCE\_SUCCESS）”。

二、遍历父类方法表，对每个有效方法调用do\_inheritance\_check\_on\_method\_ex()函数进行继承检查：

1）只要有一个方法的检查结果是“继承警告（INHERITANCE\_WARNING）”就把总状态（overall\_status）标记成警告；

2）只要有一个方法的检查结果不是“继承警告（INHERITANCE\_WARNING）”和“继承成功（ INHERITANCE\_SUCCESS）”就马上返回这个检查结果。

三、遍历父类属性信息，对每个有效属性进行继承检查：

1）如果父类属性有“私有（private）”标记，跳过这个属性；

2）如果子类也定义了同名属性，并且有规定类型，调用property\_types\_compatible()函数进行属性兼容性检查，碰到检查结果不是“继承成功（ INHERITANCE\_SUCCESS）”的，马上返回这个检查结果。

四、返回总状态overall\_status）。

## 第三步：注册早期绑定

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第4块代码**  if (EXPECTED(status != INHERITANCE\_UNRESOLVED)) { // 如果状态不是未完成  // 如果类是不可更改的  if (ce->ce\_flags & ZEND\_ACC\_IMMUTABLE) {  ce = zend\_lazy\_class\_load(ce); // 懒加载 类  } else if (ce->ce\_flags & ZEND\_ACC\_FILE\_CACHED) { // 如果有文件缓存  ce = zend\_lazy\_class\_load(ce); // 懒加载 类  ce->ce\_flags &= ~ZEND\_ACC\_FILE\_CACHED; // 去掉文件缓存标记  }  // 注册早期绑定类  if (UNEXPECTED(!register\_early\_bound\_ce(delayed\_early\_binding, lcname, ce))) {  return NULL;  } |

如上所示，对于不可更改的类和来自文件缓存的类，需要先调用zend\_lazy\_class\_load()函数创建副本（详细介绍参见“懒加载类”章节），再调用register\_early\_bound\_ce()函数注册已早期绑定的类。

### register\_early\_bound\_ce()函数

register\_early\_bound\_ce()函数用于注册已早期绑定的类，代码如下：

|  |
| --- |
| static zend\_always\_inline bool register\_early\_bound\_ce(zval \*delayed\_early\_binding, zend\_string \*lcname, zend\_class\_entry \*ce) {  if (delayed\_early\_binding) { // 如果有延时早期绑定  if (EXPECTED(!(ce->ce\_flags & ZEND\_ACC\_PRELOADED))) { // 如果类没有预加载标记  // 修改执行时类表中的类名  if (zend\_hash\_set\_bucket\_key(EG(class\_table), (Bucket \*)delayed\_early\_binding, lcname) != NULL) {  Z\_CE\_P(delayed\_early\_binding) = ce; // 延时早期绑定中  return true; // 返回成功  }  } else { // 如果类有预加载标记  // 执行时类表中，添加当前类  if (zend\_hash\_add\_ptr(EG(class\_table), lcname, ce) != NULL) {  return true;  }  }  // 报错：不可定义此类名，因为它已被使用  zend\_error\_noreturn(E\_COMPILE\_ERROR, "Cannot declare %s %s, because the name is already in use", zend\_get\_object\_type(ce), ZSTR\_VAL(ce->name));  return false;  }  // 如果没有 延时早期绑定（有的话已经在上面返回了），把类添加到编译时类表中  if (zend\_hash\_add\_ptr(CG(class\_table), lcname, ce) != NULL) {  return true;  }  return false;  } |

如上所示，按是否有delayed\_early\_binding参数，分两种情况处理：

一、如果有delayed\_early\_binding参数，分成以下几种情况：

1）类有预加载（ZEND\_ACC\_PRELOADED）标记：在执行时类表EG(class\_table)中把对应的预加载类的键名更新成小写类名，并把EG(class\_table)中的类通过delayed\_early\_binding参数进行引用返回，返回真；

2）类没有预加载标记，但类已经在执行时类表EG(class\_table)中，返回真；

3）其他情况：报错，不可定义此类名，因为它已被使用，返回假。

二、如果没有delayed\_early\_binding参数：把此类添加进执行时类表EG(class\_table)中，键名为此类的小写类名。

## 第四步：继承

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第5块代码**  // 记录原本在链接的类  orig\_linking\_class = CG(current\_linking\_class);  // 如果有用到缓存，当前类记录 到正在链接  CG(current\_linking\_class) = is\_cacheable ? ce : NULL;  zend\_try{  if (is\_cacheable) { // 如果用到缓存  zend\_begin\_record\_errors(); // 开始记录错误信息  }  // 继承  zend\_do\_inheritance\_ex(ce, parent\_ce, status == INHERITANCE\_SUCCESS); |

zend\_do\_inheritance\_ex()函数在“类的继承”章节中介绍过。

## 第五步：处理父类的接口

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第5块代码**  if (parent\_ce && parent\_ce->num\_interfaces) { // 如果父类有接口  zend\_do\_inherit\_interfaces(ce, parent\_ce); // 继承父类的接口  } |

zend\_do\_inheritance\_ex()函数在“类的继承-实现接口”章节中介绍过。

## 第六步：构造属性表

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第5块代码**  // 构造属性信息表  zend\_build\_properties\_info\_table(ce); |

zend\_build\_properties\_info\_table()函数已经在“类的绑定和链接”章节中介绍过。

## 第七步：检查抽象类

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第5块代码**  // 如果此类是 是 隐式抽象类 不是 接口、trait、显式抽象类  if ((ce->ce\_flags & (ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS|ZEND\_ACC\_INTERFACE|ZEND\_ACC\_TRAIT|ZEND\_ACC\_EXPLICIT\_ABSTRACT\_CLASS)) == ZEND\_ACC\_IMPLICIT\_ABSTRACT\_CLASS) {  zend\_verify\_abstract\_class(ce); // 检查抽象类  }  // 不可以有未解决变异 标记  ZEND\_ASSERT(!(ce->ce\_flags & ZEND\_ACC\_UNRESOLVED\_VARIANCE));  **ce->ce\_flags |= ZEND\_ACC\_LINKED; // 链接完成**  CG(current\_linking\_class) = orig\_linking\_class; // 恢复原本正在链接的类  } zend\_catch { // 如果捕获到异常  EG(record\_errors) = false; // 不记录错误  zend\_free\_recorded\_errors(); // 释放错误记录  zend\_bailout(); // 跳伞  } zend\_end\_try();    EG(record\_errors) = false; // 不记录错误 |

zend\_verify\_abstract\_class()函数已经在“类的绑定和链接”章节中介绍过。

zend\_free\_recorded\_errors()函数和zend\_bailout()函数不是本篇重点，暂不介绍。

## 第八步：把类写入缓存

代码如下：

|  |
| --- |
| **// zend\_do\_link\_class()函数，第5块代码**  if (is\_cacheable) { // 如果可缓存  HashTable \*ht = (HashTable\*)ce->inheritance\_cache; // 继承缓存哈希表  zend\_class\_entry \*new\_ce; // 临时变量  ce->inheritance\_cache = NULL; // 清空 继承缓存哈希表 指针  // 添加继承缓存  new\_ce = zend\_inheritance\_cache\_add(ce, proto, parent\_ce, NULL, ht);  if (new\_ce) { // 如果新类有效  // 在类表中查找此类  zval \*zv = zend\_hash\_find\_known\_hash(CG(class\_table), lcname);  ce = new\_ce; // 使用新类  Z\_CE\_P(zv) = ce; // 使用此新类替换旧的类  }  if (ht) { // 如果 继承缓存哈希表 有效  zend\_hash\_destroy(ht); // 销毁此哈希表  FREE\_HASHTABLE(ht); // 释放此哈希表  }  }    if (ZSTR\_HAS\_CE\_CACHE(ce->name)) { // 如果此类有缓存  ZSTR\_SET\_CE\_CACHE(ce->name, ce); // 更新缓存  }  zend\_observer\_class\_linked\_notify(ce, lcname); // 依次调用每个 链接回调 函数  return ce; // 返回刚刚加载的类  }  return NULL;  } |

如上，zend\_inheritance\_cache\_add()函数用于写入类缓存，数这两个函数在加速器（ext/opcache/ZendAccelerator.c）中定义，不在PHP内核中。缓存部分非本篇重点，暂不详细介绍。

# 九、变异责任

处理变异责任（variance\_obligation）需要用到variance\_obligation结构体，声名如下：

|  |
| --- |
| typedef struct {  enum {  OBLIGATION\_DEPENDENCY, // 从属  OBLIGATION\_COMPATIBILITY, // 兼容性  OBLIGATION\_PROPERTY\_COMPATIBILITY // 属性兼容  } type;  //  union {  zend\_class\_entry \*dependency\_ce; // 从属类  //  struct {  // 在继承检查过程中，trait 可能使用 临时的 堆栈函数 ，所以使用使用函数的副本比较好。  /\* Traits may use temporary on-stack functions during inheritance checks,  \* so use copies of functions here as well. \*/  zend\_function parent\_fn; // 父函数  zend\_function child\_fn; // 子函数  zend\_class\_entry \*child\_scope; // 子域  zend\_class\_entry \*parent\_scope; // 父域  };  //  struct {  const zend\_property\_info \*parent\_prop; // 父类属性信息  const zend\_property\_info \*child\_prop; // 子类属性信息  };  };  } variance\_obligation; |

目前用到的变异责任分为3种：1）类属性兼的容性责任；2）类方法的兼容性责任；3）从属责任，下面依次介绍。

## 一）类属性兼的容性责任

add\_property\_compatibility\_obligation()函数，在do\_inherit\_property()函数（详情参见“类的继承-继承的实现”章节）中被调用，用于添加类属性的兼容性责任，代码如下：

|  |
| --- |
| // 添加 属性性信息兼容性责任。p1:所属类，p2:子属性信息，p3:父属性信息  static void add\_property\_compatibility\_obligation(  zend\_class\_entry \*ce, const zend\_property\_info \*child\_prop,  const zend\_property\_info \*parent\_prop) {  // 为类获取或初始化 变异责任 哈希表  HashTable \*obligations = get\_or\_init\_obligations\_for\_class(ce);  // 创建新的 变异责任  variance\_obligation \*obligation = emalloc(sizeof(variance\_obligation));  // 类型为属性兼容性  obligation->type = OBLIGATION\_PROPERTY\_COMPATIBILITY;  // 关联子属性信息  obligation->child\_prop = child\_prop;  // 关联父属性信息  obligation->parent\_prop = parent\_prop;  // 把变异责任 添加到责任哈希表中  zend\_hash\_next\_index\_insert\_ptr(obligations, obligation);  } |

如上所示，先调用get\_or\_init\_obligations\_for\_class()函数，创建责任（obligations）哈希表，再创建变异责任实例，把相关信息添加到变异责任实例中，最后把变异责任实例添加到obligations哈希表中。

### get\_or\_init\_obligations\_for\_class()函数

get\_or\_init\_obligations\_for\_class()函数用于为指定类创建“延时变异责任”哈希表：

|  |
| --- |
| // 为类获取或初始化 变异责任 哈希表  static HashTable \*get\_or\_init\_obligations\_for\_class(zend\_class\_entry \*ce) {  HashTable \*ht;  zend\_ulong key;  if (!CG(delayed\_variance\_obligations)) { // 如果 编译时 没有 延时变异责任  // 创建 延时变异责任 哈希表  ALLOC\_HASHTABLE(CG(delayed\_variance\_obligations));  // 初始化 延时变异责任 哈希表  zend\_hash\_init(CG(delayed\_variance\_obligations), 0, NULL, variance\_obligation\_ht\_dtor, 0);  }  // 把类指针转成整数  key = (zend\_ulong) (uintptr\_t) ce;  // 在 延时变异责任 哈希表中查找 此类的 哈希表  ht = zend\_hash\_index\_find\_ptr(CG(delayed\_variance\_obligations), key);  if (ht) { // 如果找到  return ht; // 直接返回  }  ALLOC\_HASHTABLE(ht); // 如果没找到，创建新哈希表  zend\_hash\_init(ht, 0, NULL, variance\_obligation\_dtor, 0); // 初始化新哈希表  // 放到 延时变异责任 哈希表中  zend\_hash\_index\_add\_new\_ptr(CG(delayed\_variance\_obligations), key, ht);  ce->ce\_flags |= ZEND\_ACC\_UNRESOLVED\_VARIANCE; // 为类添加 【未完成变异】 标记  return ht;  } |

如上所示，CG(delayed\_variance\_obligations)是编译时环境变量中的“延时变异责任”哈希表，里面的每一对元素，键名（key）为指定类实例的指针，对应的值为每个类专门创建的哈希表。

CG(delayed\_variance\_obligations)中的元素使用variance\_obligation\_ht\_dtor()函数进行销毁，函数代码如下：

|  |
| --- |
| static void variance\_obligation\_ht\_dtor(zval \*zv) {  zend\_hash\_destroy(Z\_PTR\_P(zv)); // 销毁并释放常量中的哈希表  FREE\_HASHTABLE(Z\_PTR\_P(zv));  } |

新哈希表中的元素使用variance\_obligation\_dtor()函数进行销毁。variance\_obligation\_dtor()函数的代码如下：

|  |
| --- |
| static void variance\_obligation\_dtor(zval \*zv) {  efree(Z\_PTR\_P(zv));  } |

## 二）类方法的兼容性责任

add\_compatibility\_obligation()函数在perform\_delayable\_implementation\_check()函数（详情参见“类的继承-成员方法的实现检查”章节）中被调用，用于添加类方法的兼容性责任：

|  |
| --- |
| static void add\_compatibility\_obligation(  zend\_class\_entry \*ce,  const zend\_function \*child\_fn, zend\_class\_entry \*child\_scope,  const zend\_function \*parent\_fn, zend\_class\_entry \*parent\_scope) {  // 为类获取或初始化 变异责任 哈希表  HashTable \*obligations = get\_or\_init\_obligations\_for\_class(ce);  // 创建变异责任  variance\_obligation \*obligation = emalloc(sizeof(variance\_obligation));  obligation->type = OBLIGATION\_COMPATIBILITY; // 类型为兼容性  // 复制方法，因为它们在traits中会 分配在堆栈里  /\* Copy functions, because they may be stack-allocated in the case of traits. \*/    if (child\_fn->common.type == ZEND\_INTERNAL\_FUNCTION) { // 如果子方法是内置方法  // 复制【内置方法】给 变异责任对象  memcpy(&obligation->child\_fn, child\_fn, sizeof(zend\_internal\_function));  } else { // 不是内置方法  // 复制【操作码】给 变异责任对象（zend\_op\_array 与 zend\_function 部分兼容）  memcpy(&obligation->child\_fn, child\_fn, sizeof(zend\_op\_array));  }  if (parent\_fn->common.type == ZEND\_INTERNAL\_FUNCTION) { // 如果父方法是内置方法  // 复制【内置方法】给 变异责任对象  memcpy(&obligation->parent\_fn, parent\_fn, sizeof(zend\_internal\_function));  } else { // 不是内置方法  // 复制【操作码】给 变异责任对象（zend\_op\_array 与 zend\_function 部分兼容）  memcpy(&obligation->parent\_fn, parent\_fn, sizeof(zend\_op\_array));  }  obligation->child\_scope = child\_scope; // 添加子类域  obligation->parent\_scope = parent\_scope; // 添加父类域  // 新建的责任，添加进责任哈希表里  zend\_hash\_next\_index\_insert\_ptr(obligations, obligation);  } |

如上所示，先调用get\_or\_init\_obligations\_for\_class()函数，创建责任（obligations）哈希表，再创建变异责任实例，把相关信息添加到变异责任实例中，最后把变异责任实例添加到obligations哈希表中。

## 三）从属（继承）责任

add\_dependency\_obligation()函数的类继承和接口的实现过程中被调用，用于添加从属责任，代码如下：

|  |
| --- |
| static void add\_dependency\_obligation(zend\_class\_entry \*ce, zend\_class\_entry \*dependency\_ce) {  // 为类获取或初始化 变异责任 哈希表  HashTable \*obligations = get\_or\_init\_obligations\_for\_class(ce);  // 创建变异责任对象  variance\_obligation \*obligation = emalloc(sizeof(variance\_obligation));  obligation->type = OBLIGATION\_DEPENDENCY; // 类型为 从属  obligation->dependency\_ce = dependency\_ce; // 关联从属类  // 添加到此类的 责任列表中  zend\_hash\_next\_index\_insert\_ptr(obligations, obligation);  } |

如上所示，先调用get\_or\_init\_obligations\_for\_class()函数，创建责任（obligations）哈希表，再创建变异责任实例，把相关信息添加到变异责任实例中，最后把变异责任实例添加到obligations哈希表中。

## 四）处理延时变异责任

resolve\_delayed\_variance\_obligations()函数用于处理前面添加好的延时责任，代码如下：

|  |
| --- |
| // 解决延时变异责任（主要是检查责任）  static void resolve\_delayed\_variance\_obligations(zend\_class\_entry \*ce) {  // 取得编译时 中的 延时变异责任（哈希表）  HashTable \*all\_obligations = CG(delayed\_variance\_obligations), \*obligations;  zend\_ulong num\_key = (zend\_ulong) (uintptr\_t) ce; // 类指针转成整数 ，作为key  // 根据 key 取出 此类的责任哈希表（又是一个哈希表）  obligations = zend\_hash\_index\_find\_ptr(all\_obligations, num\_key);  variance\_obligation \*obligation;  // 遍历责任哈希表  ZEND\_HASH\_FOREACH\_PTR(obligations, obligation) {  check\_variance\_obligation(obligation); // 检查变异责任  } ZEND\_HASH\_FOREACH\_END();  ce->ce\_flags &= ~ZEND\_ACC\_UNRESOLVED\_VARIANCE; // 类删除标记【未完成变异】  ce->ce\_flags |= ZEND\_ACC\_LINKED; // 类添加标记【已链接】  zend\_hash\_index\_del(all\_obligations, num\_key); // 责任表中删除此类  } |

### check\_variance\_obligation()函数

check\_variance\_obligation()函数用于检查和处理变异责任，函数声名如下：

|  |
| --- |
| **// check\_variance\_obligation函数的第1块代码**  static void check\_variance\_obligation(variance\_obligation \*obligation) |

#### **处理从属责任**

代码如下：

|  |
| --- |
| **// check\_variance\_obligation函数的第2块代码**  // 如果责任类型是：从属责任  if (obligation->type == OBLIGATION\_DEPENDENCY) {  // 从属类  zend\_class\_entry \*dependency\_ce = obligation->dependency\_ce;  // 如果类有 未完成变异 标记  if (dependency\_ce->ce\_flags & ZEND\_ACC\_UNRESOLVED\_VARIANCE) {  // 原本正在链接的类  zend\_class\_entry \*orig\_linking\_class = CG(current\_linking\_class);  // 如果从属类有 ZEND\_ACC\_CACHEABLE， 开始链接它  CG(current\_linking\_class) =  (dependency\_ce->ce\_flags & ZEND\_ACC\_CACHEABLE) ? dependency\_ce : NULL;  // 处理解决延时变异责任（间接递归）  resolve\_delayed\_variance\_obligations(dependency\_ce);  // 换回原本正在链接的类  CG(current\_linking\_class) = orig\_linking\_class;  } |

如上所示，当从属类（父类）有未处理的责任时，会调用resolve\_delayed\_variance\_obligations()函数递归处理父类。

#### **处理类方法兼容性责任**

代码如下：

|  |
| --- |
| **// check\_variance\_obligation函数的第3块代码**  // 责任类型是：类方法兼容性责任  } else if (obligation->type == OBLIGATION\_COMPATIBILITY) {  // 方法实现检查，p1:方法，p2:作用域，p3:原型，p4:原型作用域  inheritance\_status status = zend\_do\_perform\_implementation\_check(  &obligation->child\_fn, obligation->child\_scope,  &obligation->parent\_fn, obligation->parent\_scope);  // 如果检查不成功  if (UNEXPECTED(status != INHERITANCE\_SUCCESS)) {  // 报错：方法不兼容  emit\_incompatible\_method\_error(  &obligation->child\_fn, obligation->child\_scope,  &obligation->parent\_fn, obligation->parent\_scope, status);  } |

zend\_do\_perform\_implementation\_check()函数在“成员方法的实现检查”章节中介绍过。

emit\_incompatible\_method\_error()函数用于报错，逻辑简单，暂不介绍。

#### **处理类属性兼容性责任**

代码如下：

|  |
| --- |
| **// check\_variance\_obligation函数的第4块代码**  } else { // 其他情况  // 类型必须是：类属性兼容性责任  ZEND\_ASSERT(obligation->type == OBLIGATION\_PROPERTY\_COMPATIBILITY);  // 检查属性类型是否兼容（双向可继承）  inheritance\_status status =  property\_types\_compatible(obligation->parent\_prop, obligation->child\_prop);    // 如果状态不是继承成功  if (status != INHERITANCE\_SUCCESS) {  // 报错：不兼容的属性  emit\_incompatible\_property\_error(obligation->child\_prop, obligation->parent\_prop);  }  }  } |

emit\_incompatible\_property\_error()函数用于报错，逻辑简单，暂不介绍。

### property\_types\_compatible()函数

property\_types\_compatible()函数用于检查属性是否兼容，代码如下：

|  |
| --- |
| // 检查属性类型是否兼容（双向可继承）  inheritance\_status property\_types\_compatible(  const zend\_property\_info \*parent\_info, const zend\_property\_info \*child\_info) {  // 如果两个信息的纯类型码相同 并且 类型名称相同  if (ZEND\_TYPE\_PURE\_MASK(parent\_info->type) == ZEND\_TYPE\_PURE\_MASK(child\_info->type)  && ZEND\_TYPE\_NAME(parent\_info->type) == ZEND\_TYPE\_NAME(child\_info->type)) {  return INHERITANCE\_SUCCESS; // 继承成功  }    // 如果一个有指定类型，一个没有  if (ZEND\_TYPE\_IS\_SET(parent\_info->type) != ZEND\_TYPE\_IS\_SET(child\_info->type)) {  return INHERITANCE\_ERROR; // 返回：继承出错  }  // 进行一个 双向类型检查 ，来决定 不变性  /\* Perform a covariant type check in both directions to determined invariance. \*/  // 检查 fe\_type 是否可以作为 proto\_scope 的子类型，两个都可以是复杂类型  inheritance\_status status1 = zend\_perform\_covariant\_type\_check(  child\_info->ce, child\_info->type, parent\_info->ce, parent\_info->type);  inheritance\_status status2 = zend\_perform\_covariant\_type\_check(  parent\_info->ce, parent\_info->type, child\_info->ce, child\_info->type);    // 如果双向都可以继承，返回继承成功  if (status1 == INHERITANCE\_SUCCESS && status2 == INHERITANCE\_SUCCESS) {  return INHERITANCE\_SUCCESS;  }  // 只要有一个失败，返回失败  if (status1 == INHERITANCE\_ERROR || status2 == INHERITANCE\_ERROR) {  return INHERITANCE\_ERROR;  }  // 其他情况，继承未完成  ZEND\_ASSERT(status1 == INHERITANCE\_UNRESOLVED || status2 == INHERITANCE\_UNRESOLVED);  return INHERITANCE\_UNRESOLVED; // 返回：继承未完成  } |

ZEND\_TYPE\_开头的宏程序在“zend\_type”章节中介绍。

zend\_perform\_covariant\_type\_check()函数在“继承中的类型兼容性检查”章节中介绍过。

# ？十、懒加载类

zend\_lazy\_class\_load()函数用于，为类速创建一个完整的副本，它在zend\_try\_early\_bind()函数（参见“早期绑定”章节）和zend\_do\_link\_class()函数（参见“类的绑定和链接”章节）中被调用。

函数声名如下：

|  |
| --- |
| **// zend\_lazy\_class\_load()函数，第1块代码**  static zend\_class\_entry \*zend\_lazy\_class\_load(zend\_class\_entry \*pce) {  zend\_class\_entry \*ce;  Bucket \*p, \*end; |

函数的业务逻辑比较复杂，分为以下步骤：

## 第一步：分配内存创建类

代码如下：

|  |
| --- |
| **// zend\_lazy\_class\_load()函数，第2块代码**  ce = zend\_arena\_alloc(&CG(arena), sizeof(zend\_class\_entry));  // 复制原类  memcpy(ce, pce, sizeof(zend\_class\_entry));  ce->ce\_flags &= ~ZEND\_ACC\_IMMUTABLE; // 删除不可更改标记  ce->refcount = 1; // 引用次数  // 继承缓存  ce->inheritance\_cache = NULL; |

## 第二步：创建mutable\_data指针列表

代码如下：

|  |
| --- |
| **// zend\_lazy\_class\_load()函数，第3块代码**  if (CG(compiler\_options) & ZEND\_COMPILE\_PRELOAD) {  // 创建 mutable\_data 指针列表  ZEND\_MAP\_PTR\_NEW(ce->mutable\_data);  // 否则  } else {  // 初始化 mutable\_data 指针  ZEND\_MAP\_PTR\_INIT(ce->mutable\_data, NULL);  } |

ZEND\_MAP\_PTR\_NEW()宏程序和ZEND\_MAP\_PTR\_INIT()宏程序的相关介绍参见“地图指针与类缓存”章节。

## 第三步：复制属性表

代码如下：

|  |
| --- |
| **// zend\_lazy\_class\_load()函数，第4块代码**  if (ce->default\_properties\_table) { // 如果有属性表  // 创建属性表  zval \*dst = emalloc(sizeof(zval) \* ce->default\_properties\_count);  zval \*src = ce->default\_properties\_table; // 原属性表  zval \*end = src + ce->default\_properties\_count; // 原表结尾  ce->default\_properties\_table = dst; // 使用新表  for (; src != end; src++, dst++) { // 遍历原表  ZVAL\_COPY\_VALUE\_PROP(dst, src); // 把元素 复制到新表里  }  } |

ZVAL\_COPY\_VALUE\_PROP()宏程序是最简单直接的复制zval实例，定义如下：

|  |
| --- |
| #define ZVAL\_COPY\_VALUE\_PROP(z, v) do { \*(z) = \*(v); } while (0) |

## 第四步：复制方法表

代码如下：

|  |
| --- |
| **// zend\_lazy\_class\_load()函数，第5块代码**  ce->function\_table.pDestructor = ZEND\_FUNCTION\_DTOR;  // 如果方法表没有初始化过  if (!(HT\_FLAGS(&ce->function\_table) & HASH\_FLAG\_UNINITIALIZED)) {  p = emalloc(HT\_SIZE(&ce->function\_table)); // 分配内存创建方法表  // 把原方法表的数据复制过来  memcpy(p, HT\_GET\_DATA\_ADDR(&ce->function\_table), HT\_USED\_SIZE(&ce->function\_table));  HT\_SET\_DATA\_ADDR(&ce->function\_table, p); // 使用新的方法表数据  p = ce->function\_table.arData; // 指针到 数据段开始位置  end = p + ce->function\_table.nNumUsed; // 元素结束位置  for (; p != end; p++) { // 遍历每个方法  zend\_op\_array \*op\_array, \*new\_op\_array;  op\_array = Z\_PTR(p->val); // 取出元素中的操作码列表指针  // 必须是用户定义函数  ZEND\_ASSERT(op\_array->type == ZEND\_USER\_FUNCTION);  ZEND\_ASSERT(op\_array->scope == pce); // 操作码必须属于此类  ZEND\_ASSERT(op\_array->prototype == NULL); // 原型为空  // 分配内存创建新的操作码列表  new\_op\_array = zend\_arena\_alloc(&CG(arena), sizeof(zend\_op\_array));  Z\_PTR(p->val) = new\_op\_array; // 使用新的操作码列表  memcpy(new\_op\_array, op\_array, sizeof(zend\_op\_array)); // 把操作码复制过来  new\_op\_array->fn\_flags &= ~ZEND\_ACC\_IMMUTABLE; // 删除不可更改标记  new\_op\_array->scope = ce; // 所属域为当前类  // 初始化运行时缓存 指针表  ZEND\_MAP\_PTR\_INIT(new\_op\_array->run\_time\_cache, NULL);  // 初始化 静态变量 指针表  ZEND\_MAP\_PTR\_INIT(new\_op\_array->static\_variables\_ptr, NULL);  // 如果上面处理的是此方法，把处理器的操作码更新成（上文中）新建的副本  zend\_update\_inherited\_handler(constructor);  zend\_update\_inherited\_handler(destructor);  zend\_update\_inherited\_handler(clone);  zend\_update\_inherited\_handler(\_\_get);  zend\_update\_inherited\_handler(\_\_set);  zend\_update\_inherited\_handler(\_\_call);  zend\_update\_inherited\_handler(\_\_isset);  zend\_update\_inherited\_handler(\_\_unset);  zend\_update\_inherited\_handler(\_\_tostring);  zend\_update\_inherited\_handler(\_\_callstatic);  zend\_update\_inherited\_handler(\_\_debugInfo);  zend\_update\_inherited\_handler(\_\_serialize);  zend\_update\_inherited\_handler(\_\_unserialize);  }  } |

### zend\_update\_inherited\_handler()函数

zend\_update\_inherited\_handler()函数用于用于更新指定类方法的操作码表，让它使用新的操作码表：

|  |
| --- |
| // 如果上面处理的是此方法，把处理器的操作码更新成（上文中）新建的副本  #define zend\_update\_inherited\_handler(handler) do { \  if (ce->handler == (zend\_function\*)op\_array) { /\* 如果操作码是此处理器 \*/ \  ce->handler = (zend\_function\*)new\_op\_array; /\* 使用新操作码 \*/ \  } \  } while (0) |

## 第五步：复制静态成员

代码如下：

|  |
| --- |
| **// zend\_lazy\_class\_load()函数，第6块代码**  if (ce->default\_static\_members\_table) {  // 分配内存创建新静态成员表  zval \*dst = emalloc(sizeof(zval) \* ce->default\_static\_members\_count);  // 原静态成员表  zval \*src = ce->default\_static\_members\_table;  // 原静态成员表结尾  zval \*end = src + ce->default\_static\_members\_count;  ce->default\_static\_members\_table = dst; // 使用新静态成员表  for (; src != end; src++, dst++) { // 遍历每个成员  ZVAL\_COPY\_VALUE(dst, src); // 复制到新成员表里  }  }  // 初始化静态成员指针表  ZEND\_MAP\_PTR\_INIT(ce->static\_members\_table, NULL); |

ZVAL\_COPY\_VALUE()宏程序的更多介绍参见“类型篇”。

## 第六步：复制属性信息

代码如下：

|  |
| --- |
| **// zend\_lazy\_class\_load()函数，第7块代码**  if (!(HT\_FLAGS(&ce->properties\_info) & HASH\_FLAG\_UNINITIALIZED)) {  p = emalloc(HT\_SIZE(&ce->properties\_info)); // 创建新属性信息表（顺序哈希表）  // 复制原表数据  // 找到哈希表数据块开头位置。数据块在 arData 的前面，所以通过 arData可以找到数据块开头  memcpy(p, HT\_GET\_DATA\_ADDR(&ce->properties\_info), HT\_USED\_SIZE(&ce->properties\_info));  // 设置Bucket 开头位置。 通过哈希表数据块开头位置，找到 Bucket 列表开头。  HT\_SET\_DATA\_ADDR(&ce->properties\_info, p);  p = ce->properties\_info.arData; // 属性信息 顺序数组数据 部分  end = p + ce->properties\_info.nNumUsed; // 属性信息 顺序数组数据 部分 结尾  for (; p != end; p++) { // 遍历 所有属性信息  zend\_property\_info \*prop\_info, \*new\_prop\_info;  prop\_info = Z\_PTR(p->val); // 取出属性信息  ZEND\_ASSERT(prop\_info->ce == pce); // 属性信息必须属于父类  // 分配内存创建新的属性信息  new\_prop\_info= zend\_arena\_alloc(&CG(arena), sizeof(zend\_property\_info));  Z\_PTR(p->val) = new\_prop\_info; // 使用新创建的属性信息  // 把旧属性信息的内容复制给新 实例  memcpy(new\_prop\_info, prop\_info, sizeof(zend\_property\_info));  new\_prop\_info->ce = ce; // 所属类是新类  if (ZEND\_TYPE\_HAS\_LIST(new\_prop\_info->type)) { // 如果新属性信息有类型列表  zend\_type\_list \*new\_list;  // 取出列表类型  zend\_type\_list \*list = ZEND\_TYPE\_LIST(new\_prop\_info->type);  // 创建新类型列表  new\_list = zend\_arena\_alloc(&CG(arena), ZEND\_TYPE\_LIST\_SIZE(list->num\_types));  // 把旧类型列表的数据复制过来  memcpy(new\_list, list, ZEND\_TYPE\_LIST\_SIZE(list->num\_types));  // 新属性信息的类型列表，使用新建的列表  // new\_prop\_info->type\_\_prt = list  ZEND\_TYPE\_SET\_PTR(new\_prop\_info->type, list);  // ZEND\_TYPE\_FULL\_MASK ：返回zend\_type 的类型码  // 类型码中添加 \_ZEND\_TYPE\_ARENA\_BIT 从ARENA分配  ZEND\_TYPE\_FULL\_MASK(new\_prop\_info->type) |= \_ZEND\_TYPE\_ARENA\_BIT;  }  }  } |

如上所示，如果属性信息哈希表已经初始化过，进行如下操作：

1）创建新的哈希表数据块；

2）从旧数据块中把属性信息复制过来；

3）遍历每个属性信息：给每个属性信息创建副本，如果属性信息有对应的类型列表，为类型列表创建副本，并把新类型列表副本与新属性信息副本关联起来。

## 第七步：复制常量表

代码如下：

|  |
| --- |
| **// zend\_lazy\_class\_load()函数，第8块代码**  if (!(HT\_FLAGS(&ce->constants\_table) & HASH\_FLAG\_UNINITIALIZED)) {  p = emalloc(HT\_SIZE(&ce->constants\_table)); // 创建新常量表  // 复制原表数据  // 找到哈希表数据块开头位置。数据块在 arData 的前面，所以通过 arData可以找到数据块开头  memcpy(p, HT\_GET\_DATA\_ADDR(&ce->constants\_table), HT\_USED\_SIZE(&ce->constants\_table));  // 使用新常量表  // 设置Bucket 开头位置。 通过哈希表数据块开头位置，找到 Bucket 列表开头。  HT\_SET\_DATA\_ADDR(&ce->constants\_table, p);  p = ce->constants\_table.arData; // 数据区指针  end = p + ce->constants\_table.nNumUsed; // 数据区结尾指针  for (; p != end; p++) { // 遍历  zend\_class\_constant \*c, \*new\_c;  c = Z\_PTR(p->val); // 取得元素值下面的 类常量指针  ZEND\_ASSERT(c->ce == pce); // 常量所属类必须为父类  // 创建类常量  new\_c = zend\_arena\_alloc(&CG(arena), sizeof(zend\_class\_constant));  Z\_PTR(p->val) = new\_c; // 使用新类常量  memcpy(new\_c, c, sizeof(zend\_class\_constant)); // 把原类常量的值复制过来  new\_c->ce = ce; // 新常量指向此类  }  }  return ce; // 返回新类  } |

如上所示，如果常量哈希表已经初始化过，进行如下操作：

1）创建新的哈希表数据块；

2）从旧数据块中把常量复制过来；

3）遍历每个常量：给每个常量创建副本，并把副本的所属类更新成当前类。

# 十一、地图指针与类缓存

## 地图指针

地图指针是一个指针列表，里面的每一个指针指向一个类实例，再把指针的偏移量保存到类名（zend\_string实例）的引用次数（zend\_string.gc.refcount）中，这样就创建了从类名到类实例的映射关系，方便从类名直接找到类实例。

用作类名的zend\_string实例都会被创建成保留字串，不会被自动回收，也不需要使用引用次数，所以这样使用不会影响实例的回收。关于zend\_string的更多介绍参见“类型篇”。

在编译时全局变量中，有几个元素与地图指针有关，代码如下：

|  |
| --- |
| struct \_zend\_compiler\_globals {  ...  void \*map\_ptr\_real\_base; // 指针列表数据块的开头指针  void \*map\_ptr\_base; // 开头指针左移1个Byte，用来计算指针偏移量，结尾会多一个1  size\_t map\_ptr\_size; //  size\_t map\_ptr\_last; // 已使用 map指针 数量  ...  }; |

相关常量和宏程序如下：

|  |
| --- |
| #define ZEND\_MAP\_PTR\_KIND\_PTR 0  #define ZEND\_MAP\_PTR\_KIND\_PTR\_OR\_OFFSET 1  #define ZEND\_MAP\_PTR\_KIND ZEND\_MAP\_PTR\_KIND\_PTR\_OR\_OFFSET  // 变量名后面加个 \_\_ptr, 例如：name1 => name1\_\_ptr  #define ZEND\_MAP\_PTR(ptr) ptr ## \_\_ptr  // 定义变量名，前面是类型，例如：ZEND\_MAP\_PTR\_DEF(int,name) => int name\_\_ptr;  #define ZEND\_MAP\_PTR\_DEF(type, name) type ZEND\_MAP\_PTR(name)    // 根据相对CG(map\_ptr\_base)的偏移量（Bytes数）取回指定序号的指针  #define ZEND\_MAP\_PTR\_OFFSET2PTR(offset) ((void\*\*)((char\*)CG(map\_ptr\_base) + offset))  // 返回指针相对于CG(map\_ptr\_base)偏移的byte数量，64位系统中指针长度是8，所以结尾是1  #define ZEND\_MAP\_PTR\_PTR2OFFSET(ptr) ((void\*)(((char\*)(ptr))-((char\*)CG(map\_ptr\_base))))    // 给已经声名的指针赋值，例如：ZEND\_MAP\_PTR\_INIT(name,1) => name\_\_ptr=1;  #define ZEND\_MAP\_PTR\_INIT(ptr, val) do { ZEND\_MAP\_PTR(ptr) = (val); } while (0)  // 声名新指针，并把分配好的指针的偏移量存到指针中（用在已声名的变量上）  #define ZEND\_MAP\_PTR\_NEW(ptr) do { \  ZEND\_MAP\_PTR(ptr) = zend\_map\_ptr\_new(); \  } while (0)  // 地图指针列表中 创建新的指针，并返回新指针相对于 CG(map\_ptr\_base) 的偏移量(32位整数）  # define ZEND\_MAP\_PTR\_NEW\_OFFSET() ((uint32\_t)(uintptr\_t)zend\_map\_ptr\_new())  // 检验是否是地图指针偏移量（最右1位是否是1）  # define ZEND\_MAP\_PTR\_IS\_OFFSET(ptr) (((uintptr\_t)ZEND\_MAP\_PTR(ptr)) & 1L)  // 根据偏移量取回地图指针列表中指定的指针  # define ZEND\_MAP\_PTR\_GET\_IMM(ptr) \  (\*ZEND\_MAP\_PTR\_OFFSET2PTR((uintptr\_t)ZEND\_MAP\_PTR(ptr)))  // 自动适配，如果指针值为偏移量，根据它返回地图列表中的指针；否则直接使用此指针  # define ZEND\_MAP\_PTR\_GET(ptr) ((ZEND\_MAP\_PTR\_IS\_OFFSET(ptr) ? \  ZEND\_MAP\_PTR\_GET\_IMM(ptr) : ((void\*)(ZEND\_MAP\_PTR(ptr)))))    // 通过偏移量给地图指针列表中的指针赋值（有点绕）  # define ZEND\_MAP\_PTR\_SET\_IMM(ptr, val) do { \  /\* 根据偏移量取回指定次序的指针 \*/ \  void \*\*\_\_p = ZEND\_MAP\_PTR\_OFFSET2PTR((uintptr\_t)ZEND\_MAP\_PTR(ptr)); \  \*\_\_p = (val); /\* 给指针的目标指针赋值 \*/ \  } while (0)    // 自动适配赋值  # define ZEND\_MAP\_PTR\_SET(ptr, val) do { \  if (ZEND\_MAP\_PTR\_IS\_OFFSET(ptr)) { /\* 如果是偏移量 \*/ \  ZEND\_MAP\_PTR\_SET\_IMM(ptr, val); /\* 通过偏移量给目标指针赋值 \*/ \  } else { /\* 如果不是偏移量 \*/ \  ZEND\_MAP\_PTR\_INIT(ptr, val); /\* 指针赋值 \*/ \  } \  } while (0)  // 把指针地址-1（向左移1个Byte）  # define ZEND\_MAP\_PTR\_BIASED\_BASE(real\_base) ((void\*)(((uintptr\_t)(real\_base)) - 1)) |

zend\_map\_ptr\_new()函数用于分配内存创建地图指针列表，如果指针列表已满，自动扩展指针列表，代码如下：

|  |
| --- |
| ZEND\_API void \*zend\_map\_ptr\_new(void) {  void \*\*ptr;  if (CG(map\_ptr\_last) >= CG(map\_ptr\_size)) { // 如果指针列表已满  // map\_ptr\_last + 1，对齐到4096  CG(map\_ptr\_size) = ZEND\_MM\_ALIGNED\_SIZE\_EX(CG(map\_ptr\_last) + 1, 4096);  // 使用malloc()函数分配内存，创建指针列表，并记录数据块的开头指针  CG(map\_ptr\_real\_base) = perealloc(CG(map\_ptr\_real\_base), CG(map\_ptr\_size) \* sizeof(void\*), 1);  // 实际地址左移1个Byte  CG(map\_ptr\_base) = ZEND\_MAP\_PTR\_BIASED\_BASE(CG(map\_ptr\_real\_base));  }  ptr = (void\*\*)CG(map\_ptr\_real\_base) + CG(map\_ptr\_last); // 下一个空闲指针  \*ptr = NULL; // 初始化成NULL  CG(map\_ptr\_last)++; // 使用指针数 +1  return ZEND\_MAP\_PTR\_PTR2OFFSET(ptr); // 返回相对于CG(map\_ptr\_base)的偏移数量  } |

zend\_map\_ptr\_extend()函数用于把指针列表扩展到指针大小，代码如下:

|  |
| --- |
| ZEND\_API void zend\_map\_ptr\_extend(size\_t last) {  if (last > CG(map\_ptr\_last)) { // 如果要求的大小，大于已使用大小  void \*\*ptr;  if (last >= CG(map\_ptr\_size)) { // 并且要求的大小大于列表大小  // 校正数量，更新列表大小  CG(map\_ptr\_size) = ZEND\_MM\_ALIGNED\_SIZE\_EX(last, 4096);  // 调整指针列表数据块大小  CG(map\_ptr\_real\_base) = perealloc(CG(map\_ptr\_real\_base), CG(map\_ptr\_size) \* sizeof(void\*), 1);  // 数据块可能重新分配过，所以要更新 CG(map\_ptr\_base)  CG(map\_ptr\_base) = ZEND\_MAP\_PTR\_BIASED\_BASE(CG(map\_ptr\_real\_base));  }  ptr = (void\*\*)CG(map\_ptr\_real\_base) + CG(map\_ptr\_last); // 新分配的空间的开始位置  memset(ptr, 0, (last - CG(map\_ptr\_last)) \* sizeof(void\*)); // 新分配的空间都写成0  CG(map\_ptr\_last) = last; // 更新使用的指针数量  }  } |

如上所示，此函数会把已使用指针数量更新成要求的指针数量。主要在opcache扩展中用到，内核中没有调用。

zend\_map\_ptr\_reset()函数用于重新已使用的地图指针数量，代码如下:

|  |
| --- |
| static size\_t global\_map\_ptr\_last = 0; // 全局变量  ZEND\_API void zend\_map\_ptr\_reset(void) {  CG(map\_ptr\_last) = global\_map\_ptr\_last; // global\_map\_ptr\_last默认值为0  } |

## 类缓存

类缓存是地图指针的其中一个应用场景。

相关常量：

|  |  |  |
| --- | --- | --- |
| 常量名 | 常量值 | 说明 |
| GC\_PROTECTED | (1<<5) | 在遍历时用于递归保护 |
| IS\_STR\_CLASS\_NAME\_MAP\_PTR | GC\_PROTECTED | 对类名保留字串，表示分配过缓存指针 |

zend\_alloc\_ce\_cache()函数用于分配类缓存:

|  |
| --- |
| ZEND\_API void zend\_alloc\_ce\_cache(zend\_string \*type\_name) {  // 如果已经有类缓存，或者类名不是保留字串  if (ZSTR\_HAS\_CE\_CACHE(type\_name) || !ZSTR\_IS\_INTERNED(type\_name)) {  return; // 中断  }  // 如果类名是永久字串，并且已完成启动  if ((GC\_FLAGS(type\_name) & IS\_STR\_PERMANENT) && startup\_done) {  return; // 不要在启动(startup)以外的地方给永久保留字串分配缓存位置  }  // 不给类名 self和parent分配缓存位置  if (zend\_string\_equals\_literal\_ci(type\_name, "self")  || zend\_string\_equals\_literal\_ci(type\_name, "parent")) {  return; // 中断  }  // 类名是保留字串，不用到refcount计数器，用它来存放地图指针偏移量  uint32\_t ret;  do {  ret = ZEND\_MAP\_PTR\_NEW\_OFFSET(); // 分配指针并返回偏移量  } while (ret <= 2); // 不要第一个指针，第一个偏移量是1，和正常保留字串的值相同  // 给类名添加【类名地图指针】标记，表示此类名已经分配过地图指针  GC\_ADD\_FLAGS(type\_name, IS\_STR\_CLASS\_NAME\_MAP\_PTR);  GC\_SET\_REFCOUNT(type\_name, ret); // 把偏移量存到 type\_name->refcount中  } |

如上所示，在分配指针时，会抛弃第一个指针，因为第一个指针的偏移量是1。把偏移量存在引用计数中，普通保留字串的计数也是1，这样会无法区分一个保留字串是否是已经分配过类缓存指针。

|  |
| --- |
| // 检验一个类型是否分配过地图指针  #define ZSTR\_HAS\_CE\_CACHE(s) (GC\_FLAGS(s) & IS\_STR\_CLASS\_NAME\_MAP\_PTR)  // 验证指针是否有效，如果有效，从s->refcount取回类的缓存指针偏移量  #define ZSTR\_GET\_CE\_CACHE(s) ZSTR\_GET\_CE\_CACHE\_EX(s, 1)  // 验证指针是否有效，如果有效，把类的缓存指针偏移量写入s->refcount  #define ZSTR\_SET\_CE\_CACHE(s, ce) ZSTR\_SET\_CE\_CACHE\_EX(s, ce, 1)  // 验证类的缓存指针是否有效，指针序号（偏移量换算成序号）在已使用范围内  #define ZSTR\_VALID\_CE\_CACHE(s) \  EXPECTED((GC\_REFCOUNT(s)-1)/sizeof(void \*) < CG(map\_ptr\_last))  // 获取类的缓存指针，p1:类名，p2:获取前是否先验证类名  #define ZSTR\_GET\_CE\_CACHE\_EX(s, validate) \  ((!(validate) || ZSTR\_VALID\_CE\_CACHE(s)) ? GET\_CE\_CACHE(GC\_REFCOUNT(s)) : NULL)  // 更新类的缓存指针，p1:类名，p2:获取前是否先验证类名  #define ZSTR\_SET\_CE\_CACHE\_EX(s, ce, validate) do { \  if (!(validate) || ZSTR\_VALID\_CE\_CACHE(s)) { \  SET\_CE\_CACHE(GC\_REFCOUNT(s), ce); \  } \  } while (0)  // 通过偏移量取回类的地图指针  #define GET\_CE\_CACHE(ce\_cache) \  (\*(zend\_class\_entry \*\*)ZEND\_MAP\_PTR\_OFFSET2PTR(ce\_cache))  // 让类缓存指针指向类（zend\_class\_entry）实例  #define SET\_CE\_CACHE(ce\_cache, ce) do { \  \*((zend\_class\_entry \*\*)ZEND\_MAP\_PTR\_OFFSET2PTR(ce\_cache)) = ce; \  } while (0) |

# 十二、zend\_type

zend\_type相关功能在zend\_types.h中定义，zend\_type是一个抽象层，来描述类型的信息。它最主要的作用是来做类型的兼容性校验。

在编译PHP脚本时，碰到的所有类型都会创建zend\_type，zend\_compile\_single\_typename()函数中，所有内置类型都会调用ZEND\_TYPE\_INIT\_CODE()创建，非内置类型调用ZEND\_TYPE\_INIT\_CLASS()创建。

zend\_type结构体定义如下：

|  |
| --- |
| typedef struct { // 单个类型  void \*ptr; // 一个泛指针  uint32\_t type\_mask; // 类型掩码，32位无符号整数  } zend\_type;  typedef struct { // 类型列表  uint32\_t num\_types; // 数量  zend\_type types[1]; // 列表，可以在分配内存时增加数量  } zend\_type\_list; |

其中zend\_type\_list被设计成可伸缩结构，types元素可以通过分配内存自由设置长度。

## 相关常量

相关常量定义如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **常量值** | **说明** |
| \_ZEND\_TYPE\_EXTRA\_FLAGS\_SHIFT | 25 | 类型附加标识偏移量 |
| \_ZEND\_TYPE\_MASK | ((1u << 25) - 1) | 类型掩码，用于过滤右侧25位 |
| \_ZEND\_TYPE\_NAME\_BIT | (1u << 24) | 有名称类型标记，第24位 |
| \_ZEND\_TYPE\_LIST\_BIT | (1u << 22) | 列表类型，第22位 |
| \_ZEND\_TYPE\_KIND\_MASK | (\_ZEND\_TYPE\_LIST\_BIT|  \_ZEND\_TYPE\_NAME\_BIT) | 名称或列表类型标记，第22或24位 |
| \_ZEND\_TYPE\_ITERABLE\_BIT | (1u << 21) | 可迭代类型标记，第21位 |
| \_ZEND\_TYPE\_ARENA\_BIT | (1u << 20) | arena分配类型标记，第20位 |
| \_ZEND\_TYPE\_INTERSECTION\_BIT | (1u << 19) | 交叉类型标记，第19位 |
| \_ZEND\_TYPE\_UNION\_BIT | (1u << 18) | 联合类型标记，第18位 |
| \_ZEND\_TYPE\_MAY\_BE\_MASK | ((1u << 18) - 1) | 不可以包含上面定义的标识， 用于过滤最右面18个1 |
| \_ZEND\_TYPE\_NULLABLE\_BIT | 0x2u | 必须和 MAY\_BE\_NULL 有相同的值，2 |

\_ZEND\_TYPE\_MAY\_BE\_MASK掩码可以过滤类型。相关类型常量定义如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **常量值** | **说明** |
| MAY\_BE\_UNDEF | 1 << IS\_UNDEF | 1<<0，兼容无效类型 |
| MAY\_BE\_NULL | 1 << IS\_NULL | 1<<1，兼容NULL |
| MAY\_BE\_FALSE | 1 << IS\_FALSE | 1<<2，兼容false |
| MAY\_BE\_TRUE | 1 << IS\_TRUE | 1<<3，兼容true |
| MAY\_BE\_BOOL | MAY\_BE\_FALSE|MAY\_BE\_TRUE | 1<<2 + 1<<3  兼容true和false |
| MAY\_BE\_LONG | 1 << IS\_LONG | 1<<4，兼容整数 |
| MAY\_BE\_DOUBLE | 1 << IS\_DOUBLE | 1<<5，兼容小数 |
| MAY\_BE\_STRING | 1 << IS\_STRING | 1<<6，兼容字串 |
| MAY\_BE\_ARRAY | 1 << IS\_ARRAY | 1<<7，兼容数组 |
| MAY\_BE\_OBJECT | 1 << IS\_OBJECT | 1<<8，兼容对象 |
| MAY\_BE\_RESOURCE | 1 << IS\_RESOURCE | 1<<9，兼容资源类型 |
| MAY\_BE\_ANY | (MAY\_BE\_NULL|MAY\_BE\_FALSE|  MAY\_BE\_TRUE|MAY\_BE\_LONG|  MAY\_BE\_DOUBLE|MAY\_BE\_STRING|  MAY\_BE\_ARRAY|MAY\_BE\_OBJECT|  MAY\_BE\_RESOURCE) | 掩码，用于过滤  NULL，false，true，  整数，小数，字串，数组，  对象，资源类型 |
| MAY\_BE\_CALLABLE | 1 << IS\_CALLABLE | 1<<12，兼容闭包 |
| MAY\_BE\_VOID | 1 << IS\_VOID | 1<<14，兼容无返回 |
| MAY\_BE\_NEVER | 1 << IS\_NEVER | 1<<17，不可以返回 |
| MAY\_BE\_STATIC | 1 << IS\_STATIC | 1<<15，兼容类自身 |
| \_ZEND\_TYPE\_MAY\_BE\_MASK | (1u << 18) - 1 | 掩码，用于过滤取得以上类型（过滤最右侧18位） |

关于以上类型的更多介绍参见“类型篇”。

zend\_type的相关算法都通过宏程序来实现,在使用zend\_type时，不要直接创建和访问zend\_type实例，而是使用ZEND\_TYPE\_开头的宏程序来进行操作。

## 用于创建类型的宏程序

用于创建类型的宏程序：

|  |
| --- |
| // 创建一个无指针， 带【扩展标记】的 zend\_type实例  #define ZEND\_TYPE\_INIT\_NONE(extra\_flags) { NULL, (extra\_flags) }  // 创建一个无指针，带【类型掩码】的 zend\_type实例  #define ZEND\_TYPE\_INIT\_MASK(\_type\_mask) { NULL, (\_type\_mask) }  // 创建类型。p1:指针，p2:类型种类，p3:是否允null，p4:扩展标记  #define ZEND\_TYPE\_INIT\_PTR(ptr, type\_kind, allow\_null, extra\_flags) \  { (void \*) (ptr), (type\_kind) | ((allow\_null) ? \_ZEND\_TYPE\_NULLABLE\_BIT : 0) | (extra\_flags) }  // 创建类型。p1:指针，p2:类型掩码  #define ZEND\_TYPE\_INIT\_PTR\_MASK(ptr, type\_mask) { (void \*) (ptr), (type\_mask) }  // 创建 联合类型，带指针。p1:指针，p2:扩展标记  #define ZEND\_TYPE\_INIT\_UNION(ptr, extra\_flags) \  { (void \*) (ptr), (\_ZEND\_TYPE\_LIST\_BIT|\_ZEND\_TYPE\_UNION\_BIT) | (extra\_flags) }  // 创建 交叉类型，带指针。p1:指针，p2:扩展标记  #define ZEND\_TYPE\_INIT\_INTERSECTION(ptr, extra\_flags) \  { (void \*) (ptr), (\_ZEND\_TYPE\_LIST\_BIT|\_ZEND\_TYPE\_INTERSECTION\_BIT) | (extra\_flags) }  // 创建带类名的类型，自定义允null, extra\_flags  #define ZEND\_TYPE\_INIT\_CLASS(class\_name, allow\_null, extra\_flags) \  ZEND\_TYPE\_INIT\_PTR(class\_name, \_ZEND\_TYPE\_NAME\_BIT, allow\_null, extra\_flags)  // 和上面一样，用于类型常量  #define ZEND\_TYPE\_INIT\_CLASS\_CONST(class\_name, allow\_null, extra\_flags) \  ZEND\_TYPE\_INIT\_PTR(class\_name, \_ZEND\_TYPE\_NAME\_BIT, allow\_null, extra\_flags)  // 创建 有类名类型，自定义 type\_mask  #define ZEND\_TYPE\_INIT\_CLASS\_CONST\_MASK(class\_name, type\_mask) \  ZEND\_TYPE\_INIT\_PTR\_MASK(class\_name, \_ZEND\_TYPE\_NAME\_BIT | (type\_mask))  // 定义类型 type\_mask ：p1:允许的类型，p2:允null，p3:额外标记  #define ZEND\_TYPE\_INIT\_CODE(code, allow\_null, extra\_flags) \  /\* bool 或 迭代 或 any 或（1 << code）| allow\_null | extra\_flags \*/ \  ZEND\_TYPE\_INIT\_MASK(((code) == \_IS\_BOOL ? MAY\_BE\_BOOL : ( (code) == IS\_ITERABLE ? \_ZEND\_TYPE\_ITERABLE\_BIT : ((code) == IS\_MIXED ? MAY\_BE\_ANY : (1 << (code))))) \  | ((allow\_null) ? \_ZEND\_TYPE\_NULLABLE\_BIT : 0) | (extra\_flags)) |

## 用于读取类型信息的宏程序

用于读取类型信息的宏程序如下：

|  |
| --- |
| // 访问附加指针，类型转换为字串指针  #define ZEND\_TYPE\_NAME(t) ((zend\_string \*) (t).ptr)  // 访问附加指针，类型转换为char指针  #define ZEND\_TYPE\_LITERAL\_NAME(t) ((const char \*) (t).ptr)  // 访问附加指针，类型转换为 type\_list 指针  #define ZEND\_TYPE\_LIST(t) ((zend\_type\_list \*) (t).ptr)  // FULL\_MASK() 包含 MAY\_BE\_\* 格式的类型码，和附加的 元数据字节  // PURE\_MASK() 只包含 MAY\_BE\_\* 格式的类型码  /\* FULL\_MASK() includes the MAY\_BE\_\* type mask, as well as additional metadata bits.  \* The PURE\_MASK() only includes the MAY\_BE\_\* type mask. \*/  // 访问zend\_type 的类型码  #define ZEND\_TYPE\_FULL\_MASK(t) ((t).type\_mask)  // 返回zend\_type 的纯 类型码  #define ZEND\_TYPE\_PURE\_MASK(t) ((t).type\_mask & \_ZEND\_TYPE\_MAY\_BE\_MASK)  // 返回zend\_type 的类型码，去掉NULL  #define ZEND\_TYPE\_FULL\_MASK\_WITHOUT\_NULL(t) \  ((t).type\_mask & ~\_ZEND\_TYPE\_NULLABLE\_BIT)  // 返回zend\_type 的纯 类型码 ，去掉NULL  #define ZEND\_TYPE\_PURE\_MASK\_WITHOUT\_NULL(t) \  ((t).type\_mask & \_ZEND\_TYPE\_MAY\_BE\_MASK & ~\_ZEND\_TYPE\_NULLABLE\_BIT) |

## 用于检验类型的宏程序

用于检验类型的宏程序列表如下

|  |
| --- |
| // 检验：是否存在有效的 基础类型 ,\_ZEND\_TYPE\_MASK 最右侧25位  #define ZEND\_TYPE\_IS\_SET(t) (((t).type\_mask & \_ZEND\_TYPE\_MASK) != 0)  // 检验：是否是复杂类型, 复杂类型是指：（一个列表加一个(union 或 intersection)） 或者 （类名）  #define ZEND\_TYPE\_IS\_COMPLEX(t) ((((t).type\_mask) & \_ZEND\_TYPE\_KIND\_MASK) != 0)  // 检验：是否名称类型（类）  #define ZEND\_TYPE\_HAS\_NAME(t) ((((t).type\_mask) & \_ZEND\_TYPE\_NAME\_BIT) != 0)  // 检验：是否是类型列表  #define ZEND\_TYPE\_HAS\_LIST(t) ((((t).type\_mask) & \_ZEND\_TYPE\_LIST\_BIT) != 0)  // 检验：是否是可迭代类型  #define ZEND\_TYPE\_IS\_ITERABLE\_FALLBACK(t) \  ((((t).type\_mask) & \_ZEND\_TYPE\_ITERABLE\_BIT) != 0)  // 检验：是否是交叉类型  #define ZEND\_TYPE\_IS\_INTERSECTION(t) \  ((((t).type\_mask) & \_ZEND\_TYPE\_INTERSECTION\_BIT) != 0)  // 检验：是否是联合类型  #define ZEND\_TYPE\_IS\_UNION(t) ((((t).type\_mask) & \_ZEND\_TYPE\_UNION\_BIT) != 0)  // 检验：是否是ARENA分配的类型  #define ZEND\_TYPE\_USES\_ARENA(t) ((((t).type\_mask) & \_ZEND\_TYPE\_ARENA\_BIT) != 0)  // 检验：是有效的基础类型，并且指针为空  #define ZEND\_TYPE\_IS\_ONLY\_MASK(t) (ZEND\_TYPE\_IS\_SET(t) && (t).ptr == NULL)  // 检验：p1的类型中 是否包含需要的类型 p2  #define ZEND\_TYPE\_CONTAINS\_CODE(t, code) (((t).type\_mask & (1u << (code))) != 0)  // 检查：是否可以是 null  #define ZEND\_TYPE\_ALLOW\_NULL(t) (((t).type\_mask & \_ZEND\_TYPE\_NULLABLE\_BIT) != 0) |

## 用于遍历类型列表的宏程序

用于遍历类型列表的宏程序：

|  |
| --- |
| // 迭代一个 zend\_type 列表  #define ZEND\_TYPE\_LIST\_FOREACH(list, type\_ptr) do { \  zend\_type \*\_list = (list)->types; \  zend\_type \*\_end = \_list + (list)->num\_types; \  for (; \_list < \_end; \_list++) { \  type\_ptr = \_list;  // for循环语句的结尾大括号  #define ZEND\_TYPE\_LIST\_FOREACH\_END() \  } \  } while (0)  // 这个迭代器可用于任何 zend\_type. 如果是 zend\_type 列表，所有列表元素都会被访问到。  // 如果是个单独的 zend\_type ,就只访问它一个  #define ZEND\_TYPE\_FOREACH(type, type\_ptr) do { \  zend\_type \*\_cur, \*\_end; \  if (ZEND\_TYPE\_HAS\_LIST(type)) { /\* 如果包含列表 \*/ \  zend\_type\_list \*\_list = ZEND\_TYPE\_LIST(type); /\* 取回 zend\_type\_list \*/ \  \_cur = \_list->types; /\* 设置起始位置 \*/ \  \_end = \_cur + \_list->num\_types; /\* 设置结束位置 \*/ \  } else { /\* 如果不包含列表 \*/ \  \_cur = &(type); /\* 设置起始位置 \*/ \  \_end = \_cur + 1; /\* 设置结束位置 \*/ \  } \  do { \  type\_ptr = \_cur; /\* type\_ptr 依次指向每个元素 \*/  // 迭代器后半段。  #define ZEND\_TYPE\_FOREACH\_END() \  } while (++\_cur < \_end); /\* 迭代条件 \*/ \  } while (0) |

## 用于更新类型的宏程序

用于更新类型的宏程序如下：

|  |
| --- |
| // 更新p1的指针 ((t).ptr = (\_ptr))  #define ZEND\_TYPE\_SET\_PTR(t, \_ptr) ((t).ptr = (\_ptr))  // 给类型设置类和type\_mask  #define ZEND\_TYPE\_SET\_PTR\_AND\_KIND(t, \_ptr, kind\_bit) do { \  (t).ptr = (\_ptr); \  (t).type\_mask &= ~\_ZEND\_TYPE\_KIND\_MASK; /\* 删除：名称或列表类型标记 \*/ \  (t).type\_mask |= (kind\_bit); /\* 添加新类型 \*/ \  } while (0)  // 给列表类型设置类和 type\_mask  #define ZEND\_TYPE\_SET\_LIST(t, list) \  ZEND\_TYPE\_SET\_PTR\_AND\_KIND(t, list, \_ZEND\_TYPE\_LIST\_BIT) |

## 辅助宏程序

辅助宏程序如下:

|  |
| --- |
| // 获取一个 包含 num\_types 个元素的 zend\_type列表 的大小  #define ZEND\_TYPE\_LIST\_SIZE(num\_types) \  (sizeof(zend\_type\_list) + ((num\_types) - 1) \* sizeof(zend\_type)) |

# 附录

## 其他函数

### zend\_declare\_typed\_property()函数

zend\_declare\_typed\_property()函数在Zend/zend\_API.c中定义，用于创建带类型的属性信息，代码如下：

|  |
| --- |
| // ing3, 声名 类（不是对象）的静态和动态属性。返回属性信息。传入：类入口、属性名、属性值、权限、注释、类型（zend\_type）  ZEND\_API zend\_property\_info \*zend\_declare\_typed\_property(zend\_class\_entry \*ce, zend\_string \*name, zval \*property, int access\_type, zend\_string \*doc\_comment, zend\_type type) {  zend\_property\_info \*property\_info, \*property\_info\_ptr;  // 步骤1，修改类标记  // 如此类型存在  if (ZEND\_TYPE\_IS\_SET(type)) {  // 添加标记 有类型引用  ce->ce\_flags |= ZEND\_ACC\_HAS\_TYPE\_HINTS;  }  // 如果是内置类  if (ce->type == ZEND\_INTERNAL\_CLASS) {  // 创建持久 属性信息  property\_info = pemalloc(sizeof(zend\_property\_info), 1);  // 非内置类  } else {  // 通过 CG(arena) 创建 属性信息  property\_info = zend\_arena\_alloc(&CG(arena), sizeof(zend\_property\_info));  // 如果 zval类型是 常量语句  if (Z\_TYPE\_P(property) == IS\_CONSTANT\_AST) {  // 添加标记 更新过的常量  ce->ce\_flags &= ~ZEND\_ACC\_CONSTANTS\_UPDATED;  // 如果是static属性  if (access\_type & ZEND\_ACC\_STATIC) {  // 给类添加 包含 静态成员 标记  ce->ce\_flags |= ZEND\_ACC\_HAS\_AST\_STATICS;  // 如果不是 static, 给类  } else {  // 给类添加 包含普通属性 标记  ce->ce\_flags |= ZEND\_ACC\_HAS\_AST\_PROPERTIES;  }  }  }    // 上面是属性信息，下面是处理属性值  // 如果属性类型是 string 并且 值不是保留字  if (Z\_TYPE\_P(property) == IS\_STRING && !ZSTR\_IS\_INTERNED(Z\_STR\_P(property))) {  // 把属性值创建成 保留字 ?  zval\_make\_interned\_string(property);  }    // 如果没有 ppp， 默认是public  if (!(access\_type & ZEND\_ACC\_PPP\_MASK)) {  access\_type |= ZEND\_ACC\_PUBLIC;  }  // 步骤2（最麻烦的在这里），更新类入口 中的属性信息，静态或动态。获取属性信息的offset  // 如果有 static 标记  if (access\_type & ZEND\_ACC\_STATIC) {  // 如果此属性名已存在 并且 有 static 标记  if ((property\_info\_ptr = zend\_hash\_find\_ptr(&ce->properties\_info, name)) != NULL &&  (property\_info\_ptr->flags & ZEND\_ACC\_STATIC) != 0) {  // 使用原有的 序号  property\_info->offset = property\_info\_ptr->offset;  // 销毁此 静态属性 原有的值（准备写入新值）  zval\_ptr\_dtor(&ce->default\_static\_members\_table[property\_info->offset]);  // 从 属性信息列表中删除此属性的信息（这个在最最后重建）  zend\_hash\_del(&ce->properties\_info, name);  // 名称不存在或者没有 static 属性  } else {  // 偏移量 = 静态属性数 +1  property\_info->offset = ce->default\_static\_members\_count++;  // 分配空间扩大静态属性表。这里每次过来都要重新分配，因为一个类不会有太多属性，分配多了浪费。  ce->default\_static\_members\_table = perealloc(ce->default\_static\_members\_table, sizeof(zval) \* ce->default\_static\_members\_count, ce->type == ZEND\_INTERNAL\_CLASS);  }  // 这时候属性已经存在了  // 把属性的值 copy 给类的静态属性表 的对应 元素  ZVAL\_COPY\_VALUE(&ce->default\_static\_members\_table[property\_info->offset], property);  // zval \*static\_members\_table\_\_ptr;  // 如果没有 ce->static\_members\_table\_\_ptr 静态成员表  if (!ZEND\_MAP\_PTR(ce->static\_members\_table)) {  // 如果是内置类 或 属于持久模块  if (ce->type == ZEND\_INTERNAL\_CLASS &&  ce->info.internal.module->type == MODULE\_PERSISTENT) {  // 创建指针 ce->static\_members\_table\_\_ptr =  // CG(map\_ptr\_real\_base) 指针列表中 创建新的指针，并返回新指针相对于 CG(map\_ptr\_base) 的偏移量  ZEND\_MAP\_PTR\_NEW(ce->static\_members\_table);  }  }  // 如果没有 static 标记  } else {  zval \*property\_default\_ptr;  // 如果类的属性信息表中有此属性 并且 属性信息里没有 static 标记  if ((property\_info\_ptr = zend\_hash\_find\_ptr(&ce->properties\_info, name)) != NULL &&  (property\_info\_ptr->flags & ZEND\_ACC\_STATIC) == 0) {  // 复制 offset  property\_info->offset = property\_info\_ptr->offset;  // 从 默认属性表中删除此 属性  zval\_ptr\_dtor(&ce->default\_properties\_table[OBJ\_PROP\_TO\_NUM(property\_info->offset)]);  // 从 属性信息列表中删除此属性信息  zend\_hash\_del(&ce->properties\_info, name);  // 必须是内置类  ZEND\_ASSERT(ce->type == ZEND\_INTERNAL\_CLASS);  // 属性信息列表不可以是空  ZEND\_ASSERT(ce->properties\_info\_table != NULL);  // 更新属性信息列表  ce->properties\_info\_table[OBJ\_PROP\_TO\_NUM(property\_info->offset)] = property\_info;    // 如果属性信息表中没有此属性 或者 有 static 标记（这是怎么回事？）  } else {  // 把序号转成偏移量  property\_info->offset = OBJ\_PROP\_TO\_OFFSET(ce->default\_properties\_count);  // 默认属性数量 +1  ce->default\_properties\_count++;  // 扩大默认属性表  ce->default\_properties\_table = perealloc(ce->default\_properties\_table, sizeof(zval) \* ce->default\_properties\_count, ce->type == ZEND\_INTERNAL\_CLASS);  // 对于 用户定义类，这个在链接过程中处理  /\* For user classes this is handled during linking \*/  // 如果类型为内置类  if (ce->type == ZEND\_INTERNAL\_CLASS) {  // 扩大属性信息表  ce->properties\_info\_table = perealloc(ce->properties\_info\_table, sizeof(zend\_property\_info \*) \* ce->default\_properties\_count, 1);  // 把属性信息添加进去  ce->properties\_info\_table[ce->default\_properties\_count - 1] = property\_info;  }  }  // 默认属性指针，指向当前属性  property\_default\_ptr = &ce->default\_properties\_table[OBJ\_PROP\_TO\_NUM(property\_info->offset)];  // 上面已经创建好了属性实例，现在把值添加进去  ZVAL\_COPY\_VALUE(property\_default\_ptr, property);  // 给此属性添加标记  Z\_PROP\_FLAG\_P(property\_default\_ptr) = Z\_ISUNDEF\_P(property) ? IS\_PROP\_UNINIT : 0;  }  // 如果类型是内置类  if (ce->type & ZEND\_INTERNAL\_CLASS) {  //  /\* Must be interned to avoid ZTS data races \*/  // 如果是持久类  if (is\_persistent\_class(ce)) {  // 属性名创保留字  name = zend\_new\_interned\_string(zend\_string\_copy(name));  }  // 如果是可计数类型  if (Z\_REFCOUNTED\_P(property)) {  // 报错，内置zval不可计数  zend\_error\_noreturn(E\_CORE\_ERROR, "Internal zvals cannot be refcounted");  }  }  // 步骤3，填充信息，构造 属性信息  // 如果 是public  if (access\_type & ZEND\_ACC\_PUBLIC) {  // 名称  property\_info->name = zend\_string\_copy(name);  // 如果 是private  } else if (access\_type & ZEND\_ACC\_PRIVATE) {  // 拼接属性名 类名+属性名 持久  property\_info->name = zend\_mangle\_property\_name(ZSTR\_VAL(ce->name), ZSTR\_LEN(ce->name), ZSTR\_VAL(name), ZSTR\_LEN(name), is\_persistent\_class(ce));  // 必须 是protected  } else {  ZEND\_ASSERT(access\_type & ZEND\_ACC\_PROTECTED);  // 拼接属性名 \*+属性名 持久性跟类走  property\_info->name = zend\_mangle\_property\_name("\*", 1, ZSTR\_VAL(name), ZSTR\_LEN(name), is\_persistent\_class(ce));  }  // 名称 创建保留字  property\_info->name = zend\_new\_interned\_string(property\_info->name);  // 标记  property\_info->flags = access\_type;  // 注释  property\_info->doc\_comment = doc\_comment;  // 修饰属性  property\_info->attributes = NULL;  // 所属类  property\_info->ce = ce;  // 类型（zend\_type）  property\_info->type = type;  // 如果是持久类，处理类型  if (is\_persistent\_class(ce)) {  zend\_type \*single\_type;  // 遍历 zend\_type  ZEND\_TYPE\_FOREACH(property\_info->type, single\_type) {  // TODO Add support and test cases when gen\_stub support added  // 单个type不可以包含列表  ZEND\_ASSERT(!ZEND\_TYPE\_HAS\_LIST(\*single\_type));  // 如果类型有名字  if (ZEND\_TYPE\_HAS\_NAME(\*single\_type)) {  // 名字创建保留字  zend\_string \*name = zend\_new\_interned\_string(ZEND\_TYPE\_NAME(\*single\_type));  // 关联到 single\_type  ZEND\_TYPE\_SET\_PTR(\*single\_type, name);  // 缓存 类入口  zend\_alloc\_ce\_cache(name);  }  } ZEND\_TYPE\_FOREACH\_END();  }  // 步骤4，更新属性信息  // 更新属性信息列表中的 相应条目  zend\_hash\_update\_ptr(&ce->properties\_info, name, property\_info);  // 返回属性信息  return property\_info;  }  /\* }}} \*/ |

### find\_first\_constant\_definition()函数

find\_first\_constant\_definition()函数在报错中使用，代码如下：

|  |
| --- |
| static zend\_class\_entry\* find\_first\_constant\_definition(zend\_class\_entry \*ce, zend\_class\_entry \*\*traits, size\_t current\_trait, zend\_string \*constant\_name, zend\_class\_entry \*colliding\_ce) {  size\_t i;  if (colliding\_ce == ce) { // 如果冲突类就是当前类  for (i = 0; i < current\_trait; i++) { // 遍历前面处理过的trait  // 如果trait有效 并且 trait的常量表中有此常量  if (traits[i] && zend\_hash\_exists(&traits[i]->constants\_table, constant\_name)) {  return traits[i]; // 返回此trait  }  }  }  return colliding\_ce;  } |

### 表格中的字体

主要是调整了行高和首行缩进，行高可以按需要再调整。

主要是调整了行高和首行缩进，行高可以按需要再调整。

# 未解疑问

/\* Don't populate CE\_CACHE for mutable classes during compilation.

\* The class may be freed while persisting. \*/ 这是什么？

成员方法的实现检查大体完成，两个status需要梳理。

继承静态成员变量的业务逻辑与静态默认成员变量类似，区别还是很重要，抽空细化。

# Change Log

2025.2.21 创建，对象仓库。面向对象这部分太复杂了，要一点点抽丝剥茧。按维度操作，完成。

2025.2.22 研究一天zend\_get\_property\_offset()发现，要先搞懂继承，先能懂对象的操作。先把继承的代码先放上，慢慢研究。

2025.2.23 整理继承相关调用列表，完毕，找到头绪了。

2025.2.25 地图指针和类缓存。还是有很多依赖。操作符，api，编译等。先把操作符部分搞定吧。放在面向对象篇里。准备处理这部分。顺便把所有api整理分组一下。要不少时间。

2025.3.5 中间做了月报和qc，耽误一些天，还要准备个人资料。 instanceof\_function()

2025.3.6 zend\_perform\_covariant\_type\_check()相关函数没有太大问题，看继承相关如何梳理，这部分应该是全PHP最复杂的部分了，比哈希表可复杂多了，并且这部分还需要调试，那要先把解析器做好，MY GOD，大工程。继承贯串从解析到执行的所有流程。

2025.3.11 先把解析器修改好了，调试编译类型顺利！其实只是小修改，但流程比较复杂。整理zend\_type相关内容，还要继续消化，这部分很重要，类型这部分可以考虑独立出来。

2025.3.16 先尝试按分组整理，再慢慢注释。

2025.3.18 初步完成查找类部分。

2025.3.19 完成检查类名部分。 查找类基本完成，zend\_perform\_covariant\_type\_check()重点开始。

2025.3.20 继承检查相关。

2025.3.22 兼容性检查很难调试。

2025.3.24 继续完善继承流程。类的ZEND\_ACC\_IMMUTABLE到底是什么东西，它和继承缓存有什么关系？ 忙了一个多月了，还算有模有样。

2025.3.26 前进一点点。zend\_type\_list的编译创建有必要梳理一下。好像真有必要和编译执行一起写呢。

2025.3.30 继承中的类型兼容性判断over。

2025.6.17 empty！从接口的实现检查开始。引用zend\_perform\_covariant\_type\_check()的地方需要加标记。接口的实现检查基本OK。add\_compatibility\_obligation()函数，兼容性责任的后续处理。接口的实现章节，待整理，应放在继承前，方便复用。

完善了一些内容，整理到trait相关，需要先完成大框架，把所有内容整理一遍，最后细化、分章节。本篇预计300页左右。

2025.6.18 zend\_do\_link\_class()简单梳理一遍，这部分没有太难的东东。凡是原代码里有注释的部分还是要好好看一下。zend\_do\_link\_class()是给执行中编译用的，初始编译使用zend\_try\_early\_bind()，所以它特别重要。所以面向对象主要分2部分内容，编译和执行。

把一些内容分离到第二篇。继续整理，一边整理一边理解。

2025.6.19 接口实现不如承继来得直观，还是先从继承开始，因为继承是主要线索，其他操作比如引用trait和接口实现，都是重用继承中的方法。文件名也叫继承。

继承静态成员变量的业务逻辑与静态默认成员变量类似，区别还是很重要，抽空细化。

“类的继承”章节已经打磨完成。

变异责任，有必要单写一节。

继承检查第7步，后面略复杂。

2025.6.20 成员方法的继承检查，over。成员方法的实现检查大体完成，两个status需要梳理。成员方法的实现检查暂时over。zend\_perform\_covariant\_type\_check()早已仔细梳理过。继承告一段落。

最后要把所有方法改成“类方法”属性和成员变量改成“类属性”成员常量改成“类常量”。

|  |
| --- |
| zend\_do\_implement\_interfaces()函数代码中的feature，测试过  interface a{}  class c1 implements a{}  class c2 extends a,a{} // 报错  class c2 extends c1 implements a,a{} // 不报错 |

“类实现接口”章节完毕。

2025.6.21 引用trait第一、二、三、四节完，未细化。

2025.6.22 ZEND\_ACC\_NEARLY\_LINKED

2025.6.24 compile.h中有不少常量弄不明白，先写能弄明白的。

CG(unlinked\_uses) 已解决，是查找时添加的未链接类。“类的绑定和链”接完毕。

2025.6.25 “类的绑定和链”接完毕。“变异责任”章节完成。整个目录整理完成，不知不觉也完成80%了。缓存部分不作为本篇重点，不多介绍。

zend\_build\_properties\_info\_table()函数中的OBJ\_PROP\_TO\_NUM()宏程序。

2025.6.26 有些焦虑，先完成一遍，抽空再细化。