# 一、基本概念

内存的垃圾回收（garbage collect）是程序设计中至关重要的问题。高级编程语言如Java，Php，Python等都有自己的垃圾回收机制，让研发人员可以尽量少地关心内存管理，把主要精力放在程序的业务逻辑实现上。PHP的垃圾回收功能主要在zend\_gc.c文件中实现。

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）全局变量

zend\_gc\_globals结构体是垃圾回收功能的全局变量，它有一个别名zend\_gc\_globals，结构体定义如下：

|  |
| --- |
| typedef struct \_zend\_gc\_globals {  gc\_root\_buffer \*buf; // 回收对象指针列表  bool gc\_enabled; // 垃圾回收功能是否已开启  bool gc\_active; // 是否正在回收中（禁止同时运行两个回收过程）  bool gc\_protected; // 垃圾回收保护，开启后无法向回收指针列表添加元素  bool gc\_full; // 回收指针列表已满（占用内存达到上限）  uint32\_t unused; // 第一个可复用指针的序号  uint32\_t first\_unused; // 第一个空闲指针的序号  uint32\_t gc\_threshold; // 回收阀值  uint32\_t buf\_size; // 指针列表中已分配的指针数量  uint32\_t num\_roots; // 已使用的指针数量  uint32\_t gc\_runs; // 累计回收次数  uint32\_t collected; // 累计回收实例数量  } zend\_gc\_globals; |

相关常量列表如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| GC\_INVALID | 0 | 无效值 |

zend\_gc\_globals结构有一个全局唯一的实例和一个快速访问此实例的宏程序。相关定义如下：

|  |
| --- |
| static zend\_gc\_globals gc\_globals; // 全局唯一实例  #define GC\_G(v) (gc\_globals.v) // 快速访问垃圾回收功能的全局变量 |

### 初始化全局变量

gc\_globals\_ctor()函数用于初始化全局变量，在开启或关闭线程安全选项时，它有不同的实现逻辑。线程安全不是本篇重点内容，此处以关闭线程安全时的业务逻辑为准，函数代码如下：

|  |
| --- |
| void gc\_globals\_ctor(void) {  gc\_globals\_ctor\_ex(&gc\_globals); // 初始化gc全局变量  } |

实际业务逻辑在gc\_globals\_ctor\_ex()函数中：

|  |
| --- |
| static void gc\_globals\_ctor\_ex(zend\_gc\_globals \*gc\_globals) {  gc\_globals->gc\_enabled = 0; // 未启用垃圾回收  gc\_globals->gc\_active = 0; // 没有正在回收  gc\_globals->gc\_protected = 1; // 保护中  gc\_globals->gc\_full = 0; // 使用内存未达上阴  gc\_globals->buf = NULL; // 指针列表为空  gc\_globals->unused = GC\_INVALID; // 无可复用指针  gc\_globals->first\_unused = GC\_INVALID; // 无空闲指针  gc\_globals->gc\_threshold = GC\_INVALID; // 回收阀值，0  gc\_globals->buf\_size = GC\_INVALID; // 已分配指针数量，0  gc\_globals->num\_roots = 0; // 使用中的指针，0个  gc\_globals->gc\_runs = 0; // 累计回收次数，0次  gc\_globals->collected = 0; // 累计回收实例数量，0个  } |

### 销毁全局变量

gc\_globals\_dtor()函数用于销毁全局变量，在非线程安全模式下，函数无业务逻辑。

### 保护状态

gc\_protect()函数用于开启和关闭垃圾回收保护，开启垃圾回收保护后，gc\_possible\_root()函数不会再向指针列表中添加实例指针。gc\_protect()函数代码如下：

|  |
| --- |
| ZEND\_API bool gc\_protect(bool protect) {  bool old\_protected = GC\_G(gc\_protected); // 旧值  GC\_G(gc\_protected) = protect;  return old\_protected; // 返回旧值  } |

gc\_protected()函数用于返回垃圾回收保护状态，代码如下：

|  |
| --- |
| ZEND\_API bool gc\_protected(void) {  return GC\_G(gc\_protected);  } |

### 检查回收情况参数

在程序运行过程中，如果想要查看垃圾回收情况，除了直接访问垃圾回收全局变量，还可以使用zend\_gc\_get\_status()函数，它用于返回全局变量中的几个运行情况参数：

|  |
| --- |
| ZEND\_API void zend\_gc\_get\_status(zend\_gc\_status \*status) {  status->runs = GC\_G(gc\_runs); // 回收次数  status->collected = GC\_G(collected); // 已实例数量大小  status->threshold = GC\_G(gc\_threshold); // 回收临界值：  status->num\_roots = GC\_G(num\_roots); // 指针列表中的元素数量  } |

zend\_gc\_status结构体定义如下：

|  |
| --- |
| typedef struct \_zend\_gc\_status {  uint32\_t runs;  uint32\_t collected;  uint32\_t threshold;  uint32\_t num\_roots;  } zend\_gc\_status; |

## 二）相关常量

垃圾回收模块中经常用到的常量列表如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| GC\_ADDRESS | 0x0fffffu | 掩码，用于获取回收指针的序号  过滤右侧20个bit |
| GC\_COLOR | 0x300000u | 掩码，用于获取颜色值  过滤左侧2个bit |
| GC\_BLACK | 0 | 黑色标记，使用中，或已释放的实例 |
| GC\_WHITE | 0x100000u | 白色标记，是垃圾，待回收的实例 |
| GC\_GREY | 0x200000u | 灰色标记，可能是垃圾的实例 |
| GC\_PURPLE | 0x300000u | 紫色标记，新加入自动回收的实例，或者有特殊销毁器，等待销毁的实例 |
| GC\_INFO\_SHIFT | 10 | 位移数量，zend\_types.h中定义 |
| GC\_INFO\_MASK | 0xfffffc00 | 掩码，用于获取垃圾回收指针序号和颜色值  过滤type\_info（32位整数）左侧22个bit |
| GC\_TYPE\_MASK | 0x0000000f | 掩码，用于获取实例的类型信息  过滤type\_info（32位整数）右侧4个bit |

# 二、自动回收指针列表

垃圾回收功能中的基本单位是实例指针，对应\_gc\_root\_buffer结构体，定义如下：

|  |
| --- |
| typedef struct \_gc\_root\_buffer { // 占用内存8Bits  zend\_refcounted \*ref; // 实例指针  } gc\_root\_buffer; |

如上所示，gc\_root\_buffer是它的别名，结构体里面只有一个zend\_refcounted（引用计数器）类型的指针，在64位系统中，它占用内存的大小是8 Bits。它可以用来指向任何可计数类型的实例，它的作用是用来追踪需要自动回收的实例。

这个结构体里面只有一个元素，但仍然要声名成结构体，这是为了方便把实例串联成链表。

与指针操作相关的常量列表如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| GC\_ROOT | 0x0 | 的垃圾循环中的可用root |
| GC\_DEFAULT\_BUF\_SIZE | (16 \* 1024) | 默认指针数量 16K |
| GC\_BUF\_GROW\_STEP | (128 \* 1024) | 指针列表每次增加大小 128K |
| GC\_MAX\_UNCOMPRESSED | (512 \* 1024) | 指针索引号压缩上限 512K |
| GC\_MAX\_BUF\_SIZE | 0x40000000 | 指针列表最大大小，64M |
| GC\_THRESHOLD\_DEFAULT | (10000 + GC\_FIRST\_ROOT) | 默认回收阀值：10001 |
| GC\_THRESHOLD\_STEP | 10000 | 回收阀值增加步长 |
| GC\_THRESHOLD\_MAX | 1000000000 | 回收阀值最大值 |
| GC\_THRESHOLD\_TRIGGER | 100 | 回收量参考标准，用于决定阀值的调整方向 |

## 一）启用和关闭垃圾回收功能

启用垃圾回收功能需要调用gc\_enable()函数，代码如下：

|  |
| --- |
| // enable参数为0时表示关闭，为1时表示启用垃圾回收功能  ZEND\_API bool gc\_enable(bool enable) {  bool old\_enabled = GC\_G(gc\_enabled); // 旧状态  GC\_G(gc\_enabled) = enable; // 更新状态  // 如果从关闭转到开启，并且没有分配缓存区  if (enable && !old\_enabled && GC\_G(buf) == NULL) {  // 持久分配，大小等于 8\*GC\_DEFAULT\_BUF\_SIZE(16K) 个指针的大小，128K  GC\_G(buf) = (gc\_root\_buffer\*) pemalloc(sizeof(gc\_root\_buffer) \* GC\_DEFAULT\_BUF\_SIZE, 1);  GC\_G(buf)[0].ref = NULL; // 第一个指针指向无效位置  GC\_G(buf\_size) = GC\_DEFAULT\_BUF\_SIZE; // 更新全局变量，缓冲区指针数量，16K  GC\_G(gc\_threshold) = GC\_THRESHOLD\_DEFAULT; // 默认回收阀值10001  gc\_reset(); // 重置全局变量  }  return old\_enabled; // 返回原状态  } |

如上所示，第一次调用垃圾回收功能时会创建用来追踪回收对象的指针缓冲区（指针列表），缓冲区的大小为16K个指针，在64位系统中为128K内存。还要更新全局变量中的指针数量和回收阀值两个元素。再调用gc\_reset()函数重置全局变量，函数代码如下：

|  |
| --- |
| void gc\_reset(void) { // 重置全局变量  if (GC\_G(buf)) { // 如果有指针列表  GC\_G(gc\_active) = 0; // 没有正在回收  GC\_G(gc\_protected) = 0; // 解除回收保护  GC\_G(gc\_full) = 0; // 指针列表未满  GC\_G(unused) = GC\_INVALID; // 没有指针被使用  GC\_G(first\_unused) = GC\_FIRST\_ROOT; // 第一个未使用序号 1  GC\_G(num\_roots) = 0; // 已使用指针数0  GC\_G(gc\_runs) = 0; // 未运行过回收  GC\_G(collected) = 0; // 无回收大小  }  } |

gc\_enabled()函数用于检测垃圾回收功能是否已开启，代码如下：

|  |
| --- |
| ZEND\_API bool gc\_enabled(void) {  return GC\_G(gc\_enabled);  } |

## 二）添加需要自动回收的实例

gc\_possible\_root()函数用于把实例指针添加到自动回收指针列表中，代码如下：

|  |
| --- |
| ZEND\_API void gc\_possible\_root(zend\_refcounted \*ref) {  uint32\_t idx;  gc\_root\_buffer \*newRoot;  if (UNEXPECTED(GC\_G(gc\_protected))) { // 如果开启了gc保护，直接返回  return;  }  if (EXPECTED(GC\_HAS\_UNUSED())) { // 如果有可复用指针  idx = GC\_FETCH\_UNUSED(); // 取得可复用指针序号  // 没有可复用指针，有新的指针可用（且数量未超回收阀值）  } else if (EXPECTED(GC\_HAS\_NEXT\_UNUSED\_UNDER\_THRESHOLD())) {  idx = GC\_FETCH\_NEXT\_UNUSED(); // 获取下一个(新的)可用 root 的索引号  } else { // 如果没有可用指针了，或者超过阀值  gc\_possible\_root\_when\_full(ref); // 按空间已满处理（垃圾回收过程从这里触发）  return; // 完成  }  **// 后半段，有可用指针，使用此指针**  // 回收类型只能是数组或对象  ZEND\_ASSERT(GC\_TYPE(ref) == IS\_ARRAY || GC\_TYPE(ref) == IS\_OBJECT);  ZEND\_ASSERT(GC\_INFO(ref) == 0); // 必须无附加信息，也就是没有经过任何gc相关操作  newRoot = GC\_IDX2PTR(idx); // 第一步，把指针序号换成指针  newRoot->ref = ref; // 第二步，指针指向给出的实例  idx = gc\_compress(idx); // 压缩 指针索引号  GC\_REF\_SET\_INFO(ref, idx | GC\_PURPLE); // 给实例 ref->u.type\_info写入指针索引号和颜色  GC\_G(num\_roots)++; // 已使用指针数+ 1  } |

如上所示，业务逻辑可归纳为如下几种情况：

1）如果开始了gc\_protected，直接返回；

2）如果有可复用的指针，直接使用；

3）如果下一个空闲指针的序号没有超出回收阀值（gc\_threshold），使用空闲的指针；

4）如果下一个空闲指针的序号超出回收阀值，调用gc\_possible\_root\_when\_full()函数进行处理并返回。

每次调用，以上4种情况只会有一种生效。其中第一种情况最简单，其他几种情况分别介绍。

### 情况2：有可复用的指针

第二种情况中使用到的宏程序如下：

|  |
| --- |
| // 检测是否有可复用的指针：可复用指针链表有效（不是GC\_INVALID）  #define GC\_HAS\_UNUSED() (GC\_G(unused) != GC\_INVALID)  // 取得下一个没使用过的指针序号  #define GC\_FETCH\_UNUSED() gc\_fetch\_unused() |

gc\_fetch\_unused()函数用于取得每一个可复用指针的序号，代码如下：

|  |
| --- |
| static uint32\_t gc\_fetch\_unused(void) {  uint32\_t idx;  gc\_root\_buffer \*root;  idx = GC\_G(unused); // 第一个可复用指针的序号  root = GC\_IDX2PTR(idx); // 转成指针（序号未压缩过，不用解压）  GC\_G(unused) = GC\_LIST2IDX(root->ref); // 摘除可复用链表第一个元素，第二个当做头元素  return idx; // 返回指针序号  } |

如上所示，把可复用链表中的第一个指针元素摘除并返回它的序号，GC\_LIST2IDX()宏程序在下文回收指针过程中介绍。

取得可复用指针后，跳到gc\_possible\_root()函数的后半段业务逻辑，应用这个指针。程序中的第一个断言（ZEND\_ASSERT）表示，只有数组和对象可以添加进自动回收；第二个断言表示，ref->u.type\_info（32位整数）的左边22位必须都为0，这表示对象对象从前没有进行过回收相关处理。GC\_TYPE()和GC\_INFO()宏程序的更多介绍参见“类型篇”。

应用指针分为几步操作。

#### **1）使用GC\_IDX2PTR()宏程序通过索引号取得指针的地址。**

相关宏程序定义如下：

|  |
| --- |
| // 用索引号找到指针列表中的指针  #define GC\_IDX2PTR(idx) (GC\_G(buf) + (idx))  // 计算指针在指针列表中的索引号，也就是指针到列表开头的偏移量  #define GC\_PTR2IDX(ptr) ((ptr) - GC\_G(buf)) |

#### **2）更新此指针，让它指向给出的实例**

#### **3）调用gc\_compress()函数压缩指针的索引号**

gc\_compress()函数定义如下：

|  |
| --- |
| static uint32\_t gc\_compress(uint32\_t idx) {  if (EXPECTED(idx < GC\_MAX\_UNCOMPRESSED)) { // 没到压缩尺寸（512K）  return idx; // 不压缩，直接返回  }  // (idx % 512K) | 512K，大小在512K到1M之间，也就是最多只使用右侧10位  return (idx % GC\_MAX\_UNCOMPRESSED) | GC\_MAX\_UNCOMPRESSED;  } |

如上所示，指针索引号经过压缩，取值范围在GC\_MAX\_UNCOMPRESSED的1倍到2倍之间，也就是512K到1M之间，用整数表示时，最多只占20个Bit。

#### **4）更新ref->u.type\_info**

把压缩过的索引号和颜色标记紫色（0x300000u）拼接在一起，索引号占最右侧20个bit，颜色值占次右侧2个bit，共占22个bit。再调用GC\_REF\_SET\_INFO()宏程序更新ref->u.type\_info，把拼接好的数据写进去。GC\_REF\_SET\_INFO()宏程序定义如下：

|  |
| --- |
| #define GC\_REF\_SET\_INFO(ref, info) do { /\* 写入 ref->u.type\_info左边22位 \*/ \  GC\_TYPE\_INFO(ref) = \  /\* 保留自己的右边10位 \*/ \  (GC\_TYPE\_INFO(ref) & (GC\_TYPE\_MASK | GC\_FLAGS\_MASK)) | \  ((info) << GC\_INFO\_SHIFT); /\* 传入的22位写到左边 \*/ \  } while (0) |

如上所示，ref->u.type\_info是32位整数。它的右侧10位用来保存类型信息，与垃圾回收无关，所以要保留这10位不变，再把传入的数据左移10位后拼在一起写入ref->u.type\_info。示意图如下：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ref->u.type\_info，32位整数 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 颜色 | | 中间20位保存压缩过的索引号 | | | | | | | | | | | | | | | | | | | | 右侧10位保存类型 | | | | | | | | | |

#### **5）更新占用指针数**

### 情况3：使用空闲的指针

在使用空闲指针前，先使用GC\_HAS\_NEXT\_UNUSED\_UNDER\_THRESHOLD()宏程序检验空闲指针数量，宏程序定义如下：

|  |
| --- |
| // 验证使用数量是否在回收阀值内，回收阀值初始值为10001  #define GC\_HAS\_NEXT\_UNUSED\_UNDER\_THRESHOLD() \  (GC\_G(first\_unused) < GC\_G(gc\_threshold)) |

如上所示，检测空闲指针编号（first\_unused）是否小于回收阀值（gc\_threshold）。初始回收阀值为10001。当条件成立时使用第一个空闲指针。

取得空闲指针相关的宏程序如下：

|  |
| --- |
| // 验证buf没满：下一个空闲指针的序号 != 指针数量  #define GC\_HAS\_NEXT\_UNUSED() (GC\_G(first\_unused) != GC\_G(buf\_size))  // 取得下一个没使用过的 root指针  #define GC\_FETCH\_NEXT\_UNUSED() gc\_fetch\_next\_unused() |

gc\_fetch\_next\_unused()函数用于取得下一个空闲指针的编号，代码如下：

|  |
| --- |
| static uint32\_t gc\_fetch\_next\_unused(void) {  uint32\_t idx;  idx = GC\_G(first\_unused); // 全局变量中的下一个空闲指针序号  GC\_G(first\_unused) = GC\_G(first\_unused) + 1; // 使用后，序号+1  return idx;  } |

取得空闲指针后，指针的应用与第二种情况相同。

### 情况4：扩展指针列表

第三种情况比较复杂。当空闲指针数量大于回收阀值（gc\_threshold）时，调用gc\_possible\_root\_when\_full()函数进行处理，函数定义如下：

|  |
| --- |
| static void gc\_possible\_root\_when\_full(zend\_refcounted \*ref) {  uint32\_t idx;  gc\_root\_buffer \*newRoot;  if (GC\_G(gc\_enabled) && !GC\_G(gc\_active)) { // 如果 gc 已启用并且现在不在回收中  GC\_ADDREF(ref); // 引用数 +1，防止在其他进程中被回收掉  // gc\_collect\_cycles: 默认是 zend\_gc\_collect\_cycles()  // buf满了的时候，先回收一轮，按回收数量调整可用指针数量和回收阀值  gc\_adjust\_threshold(gc\_collect\_cycles());  if (UNEXPECTED(GC\_DELREF(ref)) == 0) { // 如果 ref 引用数 -1后为 0  rc\_dtor\_func(ref); // 销毁实例  return;  // 如果 ref 还有引用，并且有 gc\_info，说明已经添加进列表  } else if (UNEXPECTED(GC\_INFO(ref))) {  return; // 什么也不做  }  }  // 以下逻辑与gc\_possible\_root()函数相似。  if (GC\_HAS\_UNUSED()) { // 如果有可复用root  idx = GC\_FETCH\_UNUSED(); // 获取第一个可复用root的索引号  } else if (EXPECTED(GC\_HAS\_NEXT\_UNUSED())) { // 否则：如果有可用的新root  idx = GC\_FETCH\_NEXT\_UNUSED(); // 获取第一个新root的索引号  } else { // 其他情况：没用可复用的，也没有新的  gc\_grow\_root\_buffer(); // 先增加buf  if (UNEXPECTED(!GC\_HAS\_NEXT\_UNUSED())) { // 如果增加失败  return; // 直接返回，不报错  }  idx = GC\_FETCH\_NEXT\_UNUSED(); // 获取新root  }  newRoot = GC\_IDX2PTR(idx); // 新root地址  newRoot->ref = ref; // 把 ref添加进去  idx = gc\_compress(idx); // 压缩idx  GC\_REF\_SET\_INFO(ref, idx | GC\_PURPLE); // 给ref标记成紫色  GC\_G(num\_roots)++; // root数量增加  } |

如上所示，业务逻辑分为前后两部分，前半部分主要是进行垃圾回收并调整回收阀值。

#### **1）进行垃圾回收并调整回收阀值**

在多线程处理中，回收过程比较耗时，在过程中ref实例可能会被其他进程销毁，这样会导致后续的操作出错。为了防止这种错误，在进行回收前先给ref增加引用次数，待回收过程结束后再减少引用次数，如果减少1次引用后目标实例已经无引用次数了，可以直接把实例销毁掉。这是内核代码中常见的操作方式。

gc\_collect\_cycles是预先定义好的函数原型：

|  |
| --- |
| ZEND\_API int (\*gc\_collect\_cycles)(void); // zend\_gc.c中定义  gc\_collect\_cycles = zend\_gc\_collect\_cycles; // 在zend.c中赋值 |

默认情况下，使用zend\_gc\_collect\_cycles()函数进行垃圾回收，这个函数是本篇中最复杂的部分，后续展开介绍。函数的返回值是回收的指针数量（int类型），通过它，调用gc\_adjust\_threshold()函数调整回收阀值。

回收阀值的作用是用来**调整回收频率**，如果一次回收的实例过少，可以提升回收阀值降低回收频率，避免频繁回收消耗运算资源；如果一次回收的实例过多，可以降低回收阀值提高回收频率，让垃圾清理更及时，提高内存利用率。gc\_adjust\_threshold()函数的作用是动态调整回收阀值，让回收频率达到最优。函数定义如下：

|  |
| --- |
| static void gc\_adjust\_threshold(int count){  uint32\_t new\_threshold;  if (count < GC\_THRESHOLD\_TRIGGER) { // 如果上次回收数量小于100，提高阈值  if (GC\_G(gc\_threshold) < GC\_THRESHOLD\_MAX) { // 果当前回收阀值小于1000000000  new\_threshold = GC\_G(gc\_threshold) + GC\_THRESHOLD\_STEP; // 阀值 + 10000  if (new\_threshold > GC\_THRESHOLD\_MAX) { // 如果阀值大于最大可用值  new\_threshold = GC\_THRESHOLD\_MAX; // 使用最大可用值1000000000  }  if (new\_threshold > GC\_G(buf\_size)) { // 如果阀值大于指针列表长度  gc\_grow\_root\_buffer(); // 增加指针空间，一次增加128K，增加数量与阀值无关  // 这种情况并没有应用新阀值  }  if (new\_threshold <= GC\_G(buf\_size)) { // 如果阀值小于等于buf数量  GC\_G(gc\_threshold) = new\_threshold; // 直接应用  }  }  // 如果上次回收数量大于等于 100  } else if (GC\_G(gc\_threshold) > GC\_THRESHOLD\_DEFAULT) {  new\_threshold = GC\_G(gc\_threshold) - GC\_THRESHOLD\_STEP; // 阀值减少 10000  if (new\_threshold < GC\_THRESHOLD\_DEFAULT) { // 如果小于默认值  new\_threshold = GC\_THRESHOLD\_DEFAULT; // 使用默认值  }  GC\_G(gc\_threshold) = new\_threshold; // 应用新的 阀值  }  } |

如上所示，代码并不复杂，当一次回收实例数量小于100个并且阀值小于指针列表长度时，提升阀值，阀值最大不超过GC\_THRESHOLD\_MAX；当一次回收实例数量大于等于100个时降低阀值，阀值小值为初始值GC\_THRESHOLD\_DEFAULT。

当一次回收实例数量小于100个并且阀值大于指针列表长度时，不改变阀值，而是调用gc\_grow\_root\_buffer()函数增加指针列表长度，函数定义如下：

|  |
| --- |
| static void gc\_grow\_root\_buffer(void) {  size\_t new\_size;  if (GC\_G(buf\_size) >= GC\_MAX\_BUF\_SIZE) { // 如果指针列表长度超限  if (!GC\_G(gc\_full)) { // 如果没有【已满】标记  // 警告：gc缓存溢出，gc已禁用  zend\_error(E\_WARNING, "GC buffer overflow (GC disabled)\n");  GC\_G(gc\_active) = 1; // 状态为回收中（禁止再回收）  GC\_G(gc\_protected) = 1; // gc保护开启  GC\_G(gc\_full) = 1; // 添加已满标记  return;  }  }  if (GC\_G(buf\_size) < GC\_BUF\_GROW\_STEP) { // 指针数量小于128K  new\_size = GC\_G(buf\_size) \* 2; // 把指针数量翻倍  } else { // 否则 ，增长一个步长的大小 128K  new\_size = GC\_G(buf\_size) + GC\_BUF\_GROW\_STEP;  }  if (new\_size > GC\_MAX\_BUF\_SIZE) { // 如果内存超限  new\_size = GC\_MAX\_BUF\_SIZE; // 使用最大限制值  // 这时不添加【已满】标记，要等下一次增加空间时处理  }  // 调整指针列表大小，分配方式为使用 realloc() 函数  GC\_G(buf) = perealloc(GC\_G(buf), sizeof(gc\_root\_buffer) \* new\_size, 1);  GC\_G(buf\_size) = new\_size; // 更新缓存数量  } |

#### **2）查找找空闲指针并应用它**

gc\_possible\_root\_when\_full()函数的后半部分业务逻辑与gc\_possible\_root()函数相似：先查找可复用的指针，再查找空闲指针；如果都没有，调用gc\_grow\_root\_buffer()函数增加指针列表长度；最后如果有可用指针，让指针指向目标实例并在实例中记录回收信息。

## 三）从自动回收指针列表中移除实例

GC\_REMOVE\_FROM\_BUFFER()宏程序用于从回收列表中移除指针，代码如下：

|  |
| --- |
| #define GC\_REMOVE\_FROM\_BUFFER(p) do { \  zend\_refcounted \*\_p = (zend\_refcounted\*)(p); /\* 实例 \*/ \  if (GC\_TYPE\_INFO(\_p) & GC\_INFO\_MASK) { /\* 获取左侧22位，如果有垃圾回收信息 \*/ \  gc\_remove\_from\_buffer(\_p); /\* 把实例和root解除关联 \*/ \  } \  } while (0) |

它调用gc\_remove\_from\_buffer()函数来进行操作，函数代码如下：

|  |
| --- |
| ZEND\_API void gc\_remove\_from\_buffer(zend\_refcounted \*ref) {  gc\_root\_buffer \*root;  uint32\_t idx = GC\_REF\_ADDRESS(ref); // 取出存在 ref->u.type\_info 中的指针索引号  GC\_REF\_SET\_INFO(ref, 0); // 清空ref->u.type\_info 中的 【地址索引号】和【颜色】  // 如果使用序号>=压缩阀值  if (UNEXPECTED(GC\_G(first\_unused) >= GC\_MAX\_UNCOMPRESSED)) {  gc\_remove\_compressed(ref, idx); // 通过压缩索引号回收指针  return; // 完成  }  // 序号小于压缩阀值，说明序号没有被压缩过  root = GC\_IDX2PTR(idx); // 把idx 转换成 gc\_root\_buffer 指针  gc\_remove\_from\_roots(root); // 删除这个指针  } |

如上所示，先从实例的类型信息（ref->u.type\_info）中取出关联指针的序号，再清除类型信息中的垃圾回收信息（ref->u.type\_info中的最左侧22个bit）。这时已经解除了从实例到垃圾回收指针列表的关联，接下来需要清空关联的指针，让实例和回收列表双方都解除关联。

GC\_REF\_ADDRESS()宏程序用于从实例中取回关联的垃圾回收指针序号，代码如下：

|  |
| --- |
| // 获取指针索引号：32位type\_info的左边第3-22位（共20位），它的右边有10位  #define GC\_REF\_ADDRESS(ref) \  (((GC\_TYPE\_INFO(ref)) & (GC\_ADDRESS << GC\_INFO\_SHIFT)) >> GC\_INFO\_SHIFT) |

### 解压指针索引号

如果当前使用的指针数量大于等于指针压缩阀值，说明指针序号可能被压缩过，这时需要调用gc\_remove\_compressed()函数进行删除操作，函数代码如下：

|  |
| --- |
| static void gc\_remove\_compressed(zend\_refcounted \*ref, uint32\_t idx) {  gc\_root\_buffer \*root = gc\_decompress(ref, idx); // 通过ref和idx获取到实例关联的指针  gc\_remove\_from\_roots(root); // 清空指针  } |

gc\_decompress()函数用于解压指针索引号，通过实例指针和压缩过的索引号，找到指向实例的指针，代码如下：

|  |
| --- |
| static gc\_root\_buffer\* gc\_decompress(zend\_refcounted \*ref, uint32\_t idx) {  gc\_root\_buffer \*root = GC\_IDX2PTR(idx); // 假设没压缩过，直接寻找此编号的指针  if (EXPECTED(GC\_GET\_PTR(root->ref) == ref)) { // 如果实例 和 根地址实例匹配  return root;  }  // 如果不匹配，说明索引号被压缩过  while (1) { // 遍历所有余数相同的索引号  idx += GC\_MAX\_UNCOMPRESSED; // 每次跳跃 GC\_MAX\_UNCOMPRESSED  root = GC\_IDX2PTR(idx); // idx转成指针  if (GC\_GET\_PTR(root->ref) == ref) { // 如果是指向此对象的指针  return root; // 返回 这个指针  }  }  } |

如上所示，索引号可能没有被压缩过，所以先尝试直接匹配；如果不成功，再去指针列表里遍历余数（索引号与GC\_MAX\_UNCOMPRESSED取余）相同的指针，逐个匹配，匹配成功则返回。

GC\_GET\_PTR()宏程序用于修正指针：

|  |
| --- |
| // 指针转成 int 再清空最右2个bit  #define GC\_GET\_PTR(ptr) ((void\*)(((uintptr\_t)(ptr)) & ~GC\_BITS)) |

如上所示，先把指针转成int型，清空最后两位，再转成指针。在使用emalloc()等系列函数分配内存时，所有的内存地址都会被校正到8Bytes的倍数，所以所有指针的最后3个bit都是0（详情参见“内存管理篇”）。在垃圾回收的过程中，指针的最右两个bit会用来保存指针状态，所以使用指针时需要进行校正，后继会展开详细介绍。

### 回收指针列表中的指针

无论是否经过解压，找到指针列表中的元素指针后，调用gc\_remove\_from\_roots()函数来回收指针，函数代码如下：

|  |
| --- |
| static void gc\_remove\_from\_roots(gc\_root\_buffer \*root) {  GC\_LINK\_UNUSED(root); // 删除一个 gc\_root\_buffer （并没真的删除，只是标记成未使用）  GC\_G(num\_roots)--; // 已用 root 数量减少（root总数是不会变的）  } |

如上所示，先调用GC\_LINK\_UNUSED()宏程序回收指针，再把已用指针数量减1。GC\_LINK\_UNUSED()宏程序是gc\_link\_unused()函数的别名，相关代码如下：

|  |
| --- |
| #define GC\_LINK\_UNUSED(root) gc\_link\_unused(root)  // 回收一个 gc\_root\_buffer : 把所有回收的 gc\_root\_buffer 串成一个链，最后回收的会最先复用  static void gc\_link\_unused(gc\_root\_buffer \*root) {  root->ref = GC\_IDX2LIST(GC\_G(unused)); // 把原来的序号编码后，放在正在回收的指针中  // 把当前指针的索引号，放到全局变量里（这样就串成了一个链）  GC\_G(unused) = GC\_PTR2IDX(root); **// 注意：这里序号没有压缩过**  } |

如上所示，被回收的地址需要串成一个链表，等待被复用。链表的第一个元素是最后一个回收的指针，它的序号存放在全局变量GC\_G(unused)中，注意，这个序号没有被压缩过。串联时，指针中存放的不是下一个可用指针的地址，而是经过GC\_IDX2LIST()宏程序处理的指针序号。

GC\_IDX2LIST()宏程序用于把指针索引号编码成一个64位整数（uintptr\_t），相关代码如下：

|  |
| --- |
| // 把idx换算成指针相对于列表开头偏移的Bytes数，这个数字一定是8的倍数，最右一位写成1  #define GC\_IDX2LIST(idx) ((void\*)(uintptr\_t)(((idx) \* sizeof(void\*)) | GC\_UNUSED))  // list转换成idx，只要除以8，最右一个1自动会被丢掉  #define GC\_LIST2IDX(list) (((uint32\_t)(uintptr\_t)(list)) / sizeof(void\*)) |

如上所示，由于指针（gc\_root\_buffer）列表是用malloc()原生方法创建的，而不是用emalloc()函数创建的，地址没有经过校正，没有固定的空闲位，所以不可以直接在指针上添加标记。GC\_IDX2LIST()宏程序和GC\_LIST2IDX()宏程序是专为这种情况设计的，相当于把索引号左移3位，用最右侧空出来的bit存放标记。GC\_UNUSED标记的值为1，只占用最右一个bit。这样找到地址列表中的任何一个地址，只要它的最右一个bit是1，就表示它是被回收可复用的地址。

指针的回收和复用过程并不算复杂，但操作比较精巧，在不开辟新内存空间的情况下添加标记，需要把指针进行转换。这和指针序号的解压与压缩的处理过程类似。

## 四）整理指针列表

gc\_compact()用于整理指针列表，函数采用了二指压缩算法（Two-Finger compaction algorithm）：从指针列表两头同时遍历，找到左侧的空位，把最右侧的元素转移到左侧空位里，直到两个指针交叉碰撞，列表里就没有空位了。代码如下：

|  |
| --- |
| static void gc\_compact(void) {  // 如果已用指针数+1 != 第一个空闲指针序号，说明指列表中的元素不是连续使用的（中间有空指针）  if (GC\_G(num\_roots) + GC\_FIRST\_ROOT != GC\_G(first\_unused)) {  if (GC\_G(num\_roots)) { // 如果指针列表中有有效指针  // free指针用于从左向右找到空闲位置  gc\_root\_buffer \*free = GC\_IDX2PTR(GC\_FIRST\_ROOT);  // scan指针用于从右向左找到可迁移的指针  gc\_root\_buffer \*scan = GC\_IDX2PTR(GC\_G(first\_unused) - 1);  // 没有空指针时，最后一个元素序号，scan指针扫描到这里就不会再有空元素了（因为都填充了）  gc\_root\_buffer \*end = GC\_IDX2PTR(GC\_G(num\_roots));  uint32\_t idx; //  zend\_refcounted \*p;  while (free < scan) { // 只要free和scan指针没有交叉，就一直扫描  while (!GC\_IS\_UNUSED(free->ref)) { // 跳过有效指针，找到最左侧的空位  free++;  }  while (GC\_IS\_UNUSED(scan->ref)) { // 跳过无效指针，找到最右侧的有效元素  scan--;  }  if (scan > free) { // free和scan指针没有交叉，迁移指针  p = scan->ref; // 有效指针的值  free->ref = p; // 有效指针的值复制到左侧空位里  p = GC\_GET\_PTR(p); // 去掉指针里的附加标记，获得真实指针  idx = gc\_compress(GC\_PTR2IDX(free)); // 找到新位置序号，然后压缩  // 把压缩好的索引号和颜色，更新到到实例 ref->u.type\_info 里  GC\_REF\_SET\_INFO(p, idx | GC\_REF\_COLOR(p));  free++; // free 指针右移  scan--; // scan 指针左移  if (scan <= end) { // 如果两个指针交叉或碰撞，结束交换  break;  }  }  }  }  GC\_G(unused) = GC\_INVALID; // 没有可复用指针了，可复用指针序号归0  GC\_G(first\_unused) = GC\_G(num\_roots) + GC\_FIRST\_ROOT; // 更新第一个空闲指针位置  }  } |

GC\_REF\_COLOR()宏程序用于获取颜色值：

|  |
| --- |
| // 获取颜色，32位ref->u.type\_info的1-2两个位  #define GC\_REF\_COLOR(ref) \  (((GC\_TYPE\_INFO(ref)) & (GC\_COLOR << GC\_INFO\_SHIFT)) >> GC\_INFO\_SHIFT) |

## 五）销毁指针列表

root\_buffer\_dtor()函数用于销毁指针列表：

|  |
| --- |
| static void root\_buffer\_dtor(zend\_gc\_globals \*gc\_globals) {  if (gc\_globals->buf) { // 如果创建了缓冲区，释放整个缓冲区  free(gc\_globals->buf);  gc\_globals->buf = NULL;  }  } |

## 六）指针的附加标记

前文中已经介绍过，列表中的指针最右侧两个bit在垃圾回收过程中会被用来保存指针状态。相关常量如下：

|  |  |  |
| --- | --- | --- |
| **常量名** | **值** | **说明** |
| GC\_BITS | 0x3 | 掩码，过滤最右侧2个bit |
| GC\_ROOT | 0x0 | 已使用的指针 |
| GC\_UNUSED | 0x1 | 标记：可复用的指针 |
| GC\_GARBAGE | 0x2 | 标记：垃圾 |
| GC\_DTOR\_GARBAGE | 0x3 | 标记：待销毁的垃圾 |

用于访问gc信息的宏程序，代码如下：

|  |
| --- |
| // 检验是否是已使用的指针  #define GC\_IS\_ROOT(ptr) ((((uintptr\_t)(ptr)) & GC\_BITS) == GC\_ROOT)  // 检验指针有【可复用】标记  #define GC\_IS\_UNUSED(ptr) ((((uintptr\_t)(ptr)) & GC\_BITS) == GC\_UNUSED)  // 检验指针有【垃圾】标记  #define GC\_IS\_GARBAGE(ptr) ((((uintptr\_t)(ptr)) & GC\_BITS) == GC\_GARBAGE)  // 指针是否是【待销毁的垃圾】  #define GC\_IS\_DTOR\_GARBAGE(ptr) ((((uintptr\_t)(ptr)) & GC\_BITS) == GC\_DTOR\_GARBAGE)  // 给指针添加【垃圾】标记  #define GC\_MAKE\_GARBAGE(ptr) ((void\*)(((uintptr\_t)(ptr)) | GC\_GARBAGE))  // 给指针添加【待销毁的垃圾】标记  #define GC\_MAKE\_DTOR\_GARBAGE(ptr) ((void\*)(((uintptr\_t)(ptr)) | GC\_DTOR\_GARBAGE)) |

# 三、临时堆栈

在垃圾回收过程中，需要用到临时堆栈作为辅助。临时堆栈结构体如下：

|  |
| --- |
| #define ZEND\_MM\_OVERHEAD 0 // zend\_alloc.h中定义  // 64位系统中，4096/8-2=510  #define GC\_STACK\_SEGMENT\_SIZE (((4096 - ZEND\_MM\_OVERHEAD) / sizeof(void\*)) - 2)  typedef struct \_gc\_stack gc\_stack; // 堆栈页  struct \_gc\_stack { // 每个堆栈页大小为510+2=512个指针，占4K内存  gc\_stack \*prev; // 指向上一页  gc\_stack \*next; // 指向下一页  zend\_refcounted \*data[GC\_STACK\_SEGMENT\_SIZE]; // 64位系统 510个  }; |

如上所示，堆栈中每页占4K内存，64位系统中每页包含512个指针，除掉prev和next两个指针，还余下510个可用。

## 创建堆栈页

gc\_stack\_next()函数用于获取堆栈的下一页，没有可用页时自动创建。函数代码如下：

|  |
| --- |
| static zend\_never\_inline gc\_stack\* gc\_stack\_next(gc\_stack \*stack) {  if (UNEXPECTED(!stack->next)) { // 如果没有下一页  gc\_stack \*segment = emalloc(sizeof(gc\_stack)); // 创建一个新的堆栈页  segment->prev = stack; // 关联前一页  segment->next = NULL; // 下一页为空  stack->next = segment; // 前一页关联到新堆栈页  }  return stack->next; // 可以返回next了  } |

## 向堆栈中添加指针

GC\_STACK\_PUSH()宏程序用向堆栈中添加实例指针：

|  |
| --- |
| #define GC\_STACK\_PUSH(ref) gc\_stack\_push(&\_stack, &\_top, ref); |

如上所示，它调用gc\_stack\_push()函数进行操作，函数代码如下：

|  |
| --- |
| // 向堆栈页中添加实例指针。p1:堆栈页指针的指针，p2:高度，p3:实例指针  static void gc\_stack\_push(gc\_stack \*\*stack, size\_t \*top, zend\_refcounted \*ref) {  if (UNEXPECTED(\*top == GC\_STACK\_SEGMENT\_SIZE)) { // 如果存满了  (\*stack) = gc\_stack\_next(\*stack); // 转到下一个堆栈  (\*top) = 0; // 从第一个元素开始  }  (\*stack)->data[(\*top)++] = ref; // 把指针存进堆栈，高度增加  } |

如上所示，需要注意的是第一个参数stack是堆栈页指针的指针，这样传递是为了在切换堆栈页时，函数外部的堆栈页指针也跟着改变。同样，更新高度时也通过指针更新了外部的变量。

## 从堆栈中弹出元素

GC\_STACK\_POP()宏程序用于弹出堆栈页中最上一个指针，代码如下：

|  |
| --- |
| #define GC\_STACK\_POP() gc\_stack\_pop(&\_stack, &\_top) |

它调用gc\_stack\_pop()函数进行处理，代码如下：

|  |
| --- |
| // 从堆栈里弹出元素，这里其实没删掉，只是指针把这个位置标记成可用了  static zend\_refcounted\* gc\_stack\_pop(gc\_stack \*\*stack, size\_t \*top) {  if (UNEXPECTED((\*top) == 0)) { // 如果高度是0  if (!(\*stack)->prev) { // 找到前一个堆栈，没有则返回null  return NULL;  } else { // 有前一个堆栈  (\*stack) = (\*stack)->prev; // 切换到前一个堆栈  (\*top) = GC\_STACK\_SEGMENT\_SIZE - 1; // 指针指向最后一个位置  return (\*stack)->data[GC\_STACK\_SEGMENT\_SIZE - 1]; // 返回最后一个元素  }  } else { // 高度不是0  return (\*stack)->data[--(\*top)]; // 先左移指针，然后返回指向的元素  }  } |

如上所示，分为几种情况：

1）高度为0，并且没有上一页，反回NULL；

2）高度为0，并且有上一页，切换到上一页并且返回最上一个元素，高度更新为最大值减1；

3）高度不为0，返回最上一个元素，高度减1。

和gc\_stack\_push()函数一样，更新堆栈页和高度时，通过指针更新了外部变量。

GC\_STACK\_DCL()宏程序是一个辅助宏程序，代码如下：

|  |
| --- |
| #define GC\_STACK\_DCL(init) \  gc\_stack \*\_stack = init; \  size\_t \_top = 0; |

这个宏程序声名\_stack和\_top两个临时变量，留给GC\_STACK\_PUSH()和GC\_STACK\_POP()宏程序使用，这三个宏程序一起实现了堆栈的常用操作，使用它们可以不必关心堆栈的内部实现逻辑。

## 删除堆栈

gc\_stack\_free()函数用于删除整个堆栈，代码如下：

|  |
| --- |
| static void gc\_stack\_free(gc\_stack \*stack) {  gc\_stack \*p = stack->next; // 下一个堆栈  while (p) { // 如果下一个存在  stack = p->next; // stack指向再下一个  efree(p); // 删除下一个  p = stack; // 指向存在的  }  } |

如上所示，它没有删除传入的那个堆栈页，因为第一个堆栈页在使用中是通过临时变量定义的，而不是分配内存创建的。

# 四、缓冲区

在前文中提到对象的get\_gc()方法，为了简化这个方法的实现，垃圾回收模块中提供了一个缓冲区作为辅助。缓冲是一串zval实例，为了方便使用，定义了一个辅助结构体zend\_get\_gc\_buffer来协助管理它，zend\_get\_gc\_buffer结构体定义如下：

|  |
| --- |
| typedef struct {  zval \*cur; // 游标，指向缓冲区空闲位置  zval \*end; // 指向缓冲区开头  zval \*start; // 指向缓冲区结尾  } zend\_get\_gc\_buffer; |

如上所示，结构体中有3个指针，start指向缓冲区开头，end指针缓冲区结尾，cur指向缓冲区的空闲位置。

zend\_get\_gc\_buffer结构体在全局只有一个实例，在运行时全局变量中，定义如下：

|  |
| --- |
| struct \_zend\_executor\_globals {  ...  zend\_get\_gc\_buffer get\_gc\_buffer;  ...  } |

它跟随运行时全局变量executor\_globals一同创建和销毁，通常使用EG()宏程序进行快捷访问。相关的函数都比较简单。

## 重置缓冲区

zend\_get\_gc\_buffer\_create()函数用于重置缓冲区游标，代码如下：

|  |
| --- |
| ZEND\_API zend\_get\_gc\_buffer \*zend\_get\_gc\_buffer\_create(void) {  zend\_get\_gc\_buffer \*gc\_buffer = &EG(get\_gc\_buffer); // 运行时全局变量  gc\_buffer->cur = gc\_buffer->start; // 游标重置到开始位置  return gc\_buffer; // 返回实例指针  } |

## 调整缓冲区大小

zend\_get\_gc\_buffer\_grow()函数用于创建创建区内存，或改变缓冲区大小，代码如下：

|  |
| --- |
| ZEND\_API void zend\_get\_gc\_buffer\_grow(zend\_get\_gc\_buffer \*gc\_buffer) {  size\_t old\_capacity = gc\_buffer->end - gc\_buffer->start; // 旧大小  size\_t new\_capacity = old\_capacity == 0 ? 64 : old\_capacity \* 2; // 容量最小 64，每次翻倍  gc\_buffer->start = erealloc(gc\_buffer->start, new\_capacity \* sizeof(zval)); // 调整内存大小  gc\_buffer->end = gc\_buffer->start + new\_capacity; // 调整结束位置指针  gc\_buffer->cur = gc\_buffer->start + old\_capacity; // 游标指向新添加的第一个zval  } |

## 释放缓冲区

zend\_get\_gc\_buffer\_release()函数用于释放缓冲区，代码如下：

|  |
| --- |
| static void zend\_get\_gc\_buffer\_release(void) {  zend\_get\_gc\_buffer \*gc\_buffer = &EG(get\_gc\_buffer); // 运行时全局变量  efree(gc\_buffer->start); // 释放 zval 列表，但指针组还在  gc\_buffer->start = gc\_buffer->end = gc\_buffer->cur = NULL; // 指针组置空  } |

## 向缓冲区中添加变量

zend\_get\_gc\_buffer\_add\_zval()函数用于向缓冲区中添加变量，代码如下：

|  |
| --- |
| // 给游标（cur）指向的zval赋值， 完全是外部调用  static void zend\_get\_gc\_buffer\_add\_zval(zend\_get\_gc\_buffer \*gc\_buffer, zval \*zv) {  if (Z\_REFCOUNTED\_P(zv)) { // 如果是可计数类型  if (UNEXPECTED(gc\_buffer->cur == gc\_buffer->end)) { // 如果游标指向结尾  zend\_get\_gc\_buffer\_grow(gc\_buffer); // 扩展空闲  }  ZVAL\_COPY\_VALUE(gc\_buffer->cur, zv); // 把 zv 复制给 游标指向的变量  gc\_buffer->cur++; // 游标后移  }  } |

## 向缓冲区中添加对象

zend\_get\_gc\_buffer\_add\_obj()函数用于向缓冲区中添加对象，代码如下：

|  |
| --- |
| static void zend\_get\_gc\_buffer\_add\_obj( zend\_get\_gc\_buffer \*gc\_buffer, zend\_object \*obj) {  if (UNEXPECTED(gc\_buffer->cur == gc\_buffer->end)) { // 如果游标指向结尾  zend\_get\_gc\_buffer\_grow(gc\_buffer); // 扩展空闲  }  ZVAL\_OBJ(gc\_buffer->cur, obj); // 让当前zval指向此对象  gc\_buffer->cur++; // 游标后移  } |

## 计算缓冲区已使用大小

zend\_get\_gc\_buffer\_use()函数用于找到缓冲区开头位置，并计算已使用zval数量，代码如下：

|  |
| --- |
| static void zend\_get\_gc\_buffer\_use(zend\_get\_gc\_buffer \*gc\_buffer, zval \*\*table, int \*n) {  \*table = gc\_buffer->start;  \*n = gc\_buffer->cur - gc\_buffer->start;  } |

如上所示，参数table用于返回缓冲区开头位置，参数n用于返回已使用的zval数量。

# 五、垃圾回收

垃圾回收是本篇中最复杂的部分，开启回收过程的函数是zend\_gc\_collect\_cycles()。业务逻辑可归纳为如下步骤：

1、把新加入的紫色实例递归标记成灰色，并把引用次数-1。

2、处理灰色实例，先把实例标记成白色，如果引用次数大于0，把实例递归标记成黑色。

3、回收指针列表，先把所有黑色实例从指针列表中移除；再对所有白色实例递归进行如下操作：a）添加GC\_GARBAGE（垃圾）标记；b）引用次数加1；c）标记成黑色。

4、锁定zend\_fiber。

5、处理有销毁器的对象。

6、释放临时堆栈象。

7、释放临时堆栈象。

8、释放对象。

9、后续处理。

下面展开详细介绍。

zend\_gc\_collect\_cycles()函数代码量很大，为了便于理解，分块进行介绍。第一块代码如下：

|  |
| --- |
| **// zend\_gc\_collect\_cycles()函数的第1块代码**  ZEND\_API int zend\_gc\_collect\_cycles(void) {  int total\_count = 0;  bool should\_rerun\_gc = 0;  bool did\_rerun\_gc = 0;  rerun\_gc: // 重复回收  if (GC\_G(num\_roots)) { // 如果指针列表不为空  int count;  gc\_root\_buffer \*current, \*last;  zend\_refcounted \*p;  uint32\_t gc\_flags = 0;  uint32\_t idx, end;  gc\_stack stack; // 创建临时堆栈  stack.prev = NULL;  stack.next = NULL;  if (GC\_G(gc\_active)) { // 如果 正在 回收中  return 0; // 返回0  }  GC\_G(gc\_runs)++; // 回收次数 +1  GC\_G(gc\_active) = 1; // 状态更新成回收中  gc\_mark\_roots(&stack); // 步骤1：把实例的元素 递归标记成灰色  gc\_scan\_roots(&stack); // 步骤2：把实例的所有灰色原素标记成白色，再把所有引用次数>0的白色元素 递归标记成黑色    // 步骤3：把黑色元素的 type\_info 清空，并回收所在root，给每个指针添加 垃圾 标记  count = gc\_collect\_roots(&gc\_flags, &stack);  if (!GC\_G(num\_roots)) { // 如果没有有效指针，处理结束  gc\_stack\_free(&stack); // 释放stack  GC\_G(gc\_active) = 0; // 完成回收，状态：不在回收中  goto finish; // 完成  }  ... |

## 步骤一：把新实例递归标记成灰色

gc\_mark\_roots()函数用于把新加入列表的紫色实例递归标记成灰色。代码如下：

|  |
| --- |
| static void gc\_mark\_roots(gc\_stack \*stack) {  gc\_root\_buffer \*current, \*last;  gc\_compact(); // 先整理指针列表  current = GC\_IDX2PTR(GC\_FIRST\_ROOT); // 从第一个开始  last = GC\_IDX2PTR(GC\_G(first\_unused)); // 到最后一个  while (current != last) { // 遍历每一个  if (GC\_IS\_ROOT(current->ref)) { // 如果指针上没有附加标记（不是可复用，没有在回收中）  if (GC\_REF\_CHECK\_COLOR(current->ref, GC\_PURPLE)) { // 如果是新添加的紫色实例  GC\_REF\_SET\_COLOR(current->ref, GC\_GREY); // 把实例标记成灰色  gc\_mark\_grey(current->ref, stack); // 把实例的相关元素递归标记成灰色  }  }  current++; // 下一个指针  }  } |

GC\_REF\_CHECK\_COLOR()宏程序用于检验实例是否是某一颜色（黑、灰等），代码如下：

|  |
| --- |
| #define GC\_REF\_CHECK\_COLOR(ref, color) \  ((GC\_TYPE\_INFO(ref) & (GC\_COLOR << GC\_INFO\_SHIFT)) == ((color) << GC\_INFO\_SHIFT)) |

GC\_REF\_SET\_COLOR()宏程序用于把实例设置成指定的颜色，代码如下：

|  |
| --- |
| #define GC\_REF\_SET\_COLOR(ref, c) do { \  GC\_TYPE\_INFO(ref) = \  /\* 先取出原信息，删除颜色 \*/ \  (GC\_TYPE\_INFO(ref) & ~(GC\_COLOR << GC\_INFO\_SHIFT)) | \  /\* 再把颜色拼进来，这里没用掩码过滤，有可能像上面的方法一样，把前面22位都改了 \*/  ((c) << GC\_INFO\_SHIFT); \  } while (0) |

gc\_mark\_grey()函数用于把数组的所有元素或对象的所有属性递归标记成灰色，函数的业务逻辑比较复杂，可概括成几种情况：1）处理对象；2）处理哈希表；3）处理引用类型

### 第一种情况：处理对象

代码如下：

|  |
| --- |
| **// gc\_mark\_grey()函数的第1块代码**  static void gc\_mark\_grey(zend\_refcounted \*ref, gc\_stack \*stack) {  HashTable \*ht;  Bucket \*p;  zval \*zv;  uint32\_t n;  GC\_STACK\_DCL(stack); // 准备堆栈页 gc\_stack \*\_stack = init; size\_t \_top = 0;  tail\_call: // 处理完一个元素后，切换到下一个元素，跳转到这里处理下一个元素  if (GC\_TYPE(ref) == IS\_OBJECT) { // 如果实例类型是对象  zend\_object \*obj = (zend\_object\*)ref; // 取得对象实例  if (EXPECTED(!(OBJ\_FLAGS(ref) & IS\_OBJ\_FREE\_CALLED))) { // 如果对象没有被释放  zval \*table;  int len;  ht = obj->handlers->get\_gc(obj, &table, &len); // 取得可回收的数据，哈希表  n = len;  zv = table; // 可回收的zval列表指针  **... // 处理哈希表，业务逻辑见下文**  handle\_zvals: **// 跳转点：处理 zval 列表（get\_gc()方法返回，或顺序数组跳转过来）**  **... // 处理zval列表，业务逻辑见下文**  }  // 如果有IS\_OBJ\_FREE\_CALLED标记，就不用处理了 |

如上所示，针对对象的处理比较复杂，一开始要调用obj->handlers->get\_gc()函数取得需要处理的对象属性表，get\_gc()函数在不同的对象类型中可能有不同的实现，但基本业务逻辑相似，最常用的是zend\_std\_get\_gc()函数，在zend\_object\_handlers.c文件中，代码如下：

|  |
| --- |
| ZEND\_API HashTable \*zend\_std\_get\_gc(zend\_object \*zobj, zval \*\*table, int \*n) {  if (zobj->handlers->get\_properties != zend\_std\_get\_properties) { // 如果不使用标准函数  \*table = NULL; // 返回表为null  \*n = 0; // 返回个数为0  return zobj->handlers->get\_properties(zobj); // 调用 get\_properties()函数 ，返回结果  } else { // 使得标准函数  if (zobj->properties) { // 如果有动态属性表  \*table = NULL; // 返回zval列表为空  \*n = 0; // 返回zval个数为0  return zobj->properties; // 返回动态属性表  } else { // 没有动态属性表  \*table = zobj->properties\_table; // 返回列表为属性表  \*n = zobj->ce->default\_properties\_count; // 返回默认属性个数  return NULL; // 返回哈希表指针为空  }  }  } |

如上所示，反回值有几种情况：1）对象使用动态属性表，返回属性表数组，不返回table（zval列表）和n（zval个数）；2）对象使用默认属性表，不返回属性表数组，返回table（zval列表）和n（zval个数）。所以也需要针对这两咱情况分别处理。3）既使用默认属性表，又使用动态属性表，

#### **1）属性列表为数组**

处理哈希表的业务逻辑如下：

|  |
| --- |
| if (UNEXPECTED(ht)) { // 如果有返回数组  GC\_DELREF(ht); // 数组引用次数-1  if (!GC\_REF\_CHECK\_COLOR(ht, GC\_GREY)) { // 如果不是灰色  GC\_REF\_SET\_COLOR(ht, GC\_GREY); // 标记成灰色  for (; n != 0; n--) { // 先处理zval列表，检查后面所有元素  if (Z\_REFCOUNTED\_P(zv)) { // 如果是可计数类型  ref = Z\_COUNTED\_P(zv); // 取得实例  GC\_DELREF(ref); **// 减少引用次数**  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) { // 如果不是灰色  GC\_REF\_SET\_COLOR(ref, GC\_GREY); **// 标记成灰色**  GC\_STACK\_PUSH(ref); **// 向堆栈中压入实例**  }  }  zv++; // 下一个 zval  }  goto handle\_ht; **// 处理完zval列表，再跳转到第二种情况，处理数组**  }  } |

如上所示，当返回结果为数组时，也先处理zval列表，再处理数组。

处理zval列表中的可计数类型实例，业务逻辑分为两步：1）实例引用数-1；2）如果实例不是灰色：把实例标记成灰色，并添加到堆栈中等待处理。

#### **2）属性列表为zval列表**

处理zval列表的业务逻辑如下：

|  |
| --- |
| for (; n != 0; n--) { // 遍历zval列表中每一个元素  if (Z\_REFCOUNTED\_P(zv)) { // 如果是可计数类型  ref = Z\_COUNTED\_P(zv); // 取得实例  GC\_DELREF(ref); **// 引用次数-1**  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) { // 如果颜色不是灰色  GC\_REF\_SET\_COLOR(ref, GC\_GREY); **// 第一个不是灰色的实例，标记灰色，等待处理**  // 第一个元素不入栈，这是因为tail\_call刚好就处理这个元素，不用入栈了  zv++; // 下一个zv  while (--n) { // 检查后面所有元素  if (Z\_REFCOUNTED\_P(zv)) { // 如果是可计数类型  // 这个ref 是重新声名的，只对子域有效，不影响外面的  zend\_refcounted \*ref = Z\_COUNTED\_P(zv);  GC\_DELREF(ref); // 引用次数 -1  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) { // 如果颜色不是灰色  GC\_REF\_SET\_COLOR(ref, GC\_GREY); **// 标记成灰色**  GC\_STACK\_PUSH(ref); **// 向堆栈中压入ref**  }  }  zv++; // 下一个zval  }  goto tail\_call; **// 处理第一个不是灰色的实例**  }  // 如果一直走这里，说明一个灰色的也没有，全都跳过了  }  zv++; // 下一个zval  } |

处理zval列表中的可计数类型实例，业务逻辑分为几步：1）实例引用数-1；2）碰到第一个不是灰色的实例，标记成灰色，准备处理它；3）其他不是灰色的实例：把实例标记成灰色，并添加到堆栈中等待处理。

### 第二种情况：处理数组

代码如下：

|  |
| --- |
| // gc\_mark\_grey()函数的第2块代码  } else if (GC\_TYPE(ref) == IS\_ARRAY) { // 如果实例类型是数组  ht = (zend\_array\*)ref; // 取得数组  handle\_ht:  n = ht->nNumUsed; // 使用元素数量  if (HT\_IS\_PACKED(ht)) { **// 如果 ht 是顺序数组**  zv = ht->arPacked; // 顺序数组开头  goto handle\_zvals; // 跳转到第一种情况：处理zval列表  }  **// 如果 ht是哈希表**  p = ht->arData; // 找到bucket列表  for (; n != 0; n--) { // 一个一个遍历  zv = &p->val; // bucket 的值  if (Z\_TYPE\_P(zv) == IS\_INDIRECT) { // 如果是间接引用  zv = Z\_INDIRECT\_P(zv); // 追踪到引用目标  }  if (Z\_REFCOUNTED\_P(zv)) { // 如果是可计数类型  ref = Z\_COUNTED\_P(zv); // 取得实例  GC\_DELREF(ref); // 引用数量 -1  // 如果颜色不是灰色：检查后面的所有bucket，有可能压栈，所以最后是 tail\_call  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) {  GC\_REF\_SET\_COLOR(ref, GC\_GREY); // 标记成灰色  // 第一个元素不入栈，这是因为tail\_call刚好就处理这个元素，不用入栈了  p++; // 下一个bucket  while (--n) { // 检查后面的所有bucket  zv = &p->val; // 取出值  if (Z\_TYPE\_P(zv) == IS\_INDIRECT) { // 如果类型是间接引用  zv = Z\_INDIRECT\_P(zv); // 取得引用值  }  if (Z\_REFCOUNTED\_P(zv)) { // 如果 zv 是可计数类型  // 取得实例，这个ref 是重新声名的，只对子域有效，和外面的不一样  zend\_refcounted \*ref = Z\_COUNTED\_P(zv);  GC\_DELREF(ref); // 引用次数 -1  // 如果颜色不是灰色  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) {  GC\_REF\_SET\_COLOR(ref, GC\_GREY); // 标记成灰色  // 向堆栈中压入ref实例, \_top是引用返回  GC\_STACK\_PUSH(ref);  }  }  p++; // 下一个bucket  }  goto tail\_call; // 从头开始  }  // 如果一直走这里，没有压栈，就不用 tail\_call  }  p++; // 下一个bucket  } |

如上所示，当实例为顺序数组时，跳转到第一种情况中的handle\_zvals跳转点，处理顺序数组中的zval列表。如果实例是哈希表数组，处理哈希列表中的可计数类型实例，业务逻辑分为几步：1）碰到间接引用类型，先追踪间接引用；2）实例引用数-1；3）碰到第一个不是灰色的实例，标记成灰色，准备处理它；4）其他不是灰色的实例：把实例标记成灰色，并添加到堆栈中等待处理。

### 第三种情况：处理引用类型

代码如下：

|  |
| --- |
| // gc\_mark\_grey()函数的第3块代码  } else if (GC\_TYPE(ref) == IS\_REFERENCE) { // 如果实例类型是引用  if (Z\_REFCOUNTED(((zend\_reference\*)ref)->val)) { // 如果目标是可计数的类型  ref = Z\_COUNTED(((zend\_reference\*)ref)->val); // 取回目标的实例  GC\_DELREF(ref); // 减少引用次数  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) { // 如果颜色不是灰色  GC\_REF\_SET\_COLOR(ref, GC\_GREY); // 标记成灰色  // 紧接着就处理这个，所以不用入栈了  goto tail\_call; // 从头开始  }  }  } |

如上所示，处理引用类型的业务逻辑比较简单：追踪到引用目标实例，如果实例是可计数类型，先减少引用次数，如果实例不是灰色，标记成灰色并递归处理它。

### 处理下一个元素

如果实例不是以上3种类型，或在以上3种类型中没有进入递归处理，说明当前实例已经处理完毕，这时可以从堆栈中弹出一个元素进行处理3。代码如下：

|  |
| --- |
| // gc\_mark\_grey()函数的第4块代码  ref = GC\_STACK\_POP(); // 从堆栈中弹出一个指针  if (ref) { // 如果弹出的元素有效  goto tail\_call; // 从头再来，处理这个元素  }  } |

gc\_mark\_grey()函数的业务逻辑比较复杂，但目的只是把列实例递归标记成灰色。函数中使用堆栈来代替递归调用自身，这样更节省内存，效率也更高。

## 步骤二：处理灰色实例

gc\_scan\_roots()函数用于处理列表中的灰色实例，代码如下：

|  |
| --- |
| static void gc\_scan\_roots(gc\_stack \*stack) {  gc\_root\_buffer \*current = GC\_IDX2PTR(GC\_FIRST\_ROOT); // 第一个指针  gc\_root\_buffer \*last = GC\_IDX2PTR(GC\_G(first\_unused)); // 最后一个已用的指针  while (current != last) { // 遍历每一个指针  if (GC\_IS\_ROOT(current->ref)) { // 如果指针上没有附加标记（不是可复用，没有在回收中）  if (GC\_REF\_CHECK\_COLOR(current->ref, GC\_GREY)) { // 如果实例是灰色  GC\_REF\_SET\_COLOR(current->ref, GC\_WHITE); // 变成白色  gc\_scan(current->ref, stack); // 扫描实例，传入的原素一定是白色  }  }  current++; // 下一个指针  }  } |

### 把白色实例标记成黑色

gc\_scan()函数用于递归处理所有白色实例，此函数的业务逻辑框架与gc\_mark\_grey()函数相似。主要有几点不同：1）开头的处理逻辑：

|  |
| --- |
| static void gc\_scan(zend\_refcounted \*ref, gc\_stack \*stack) {  HashTable \*ht;  Bucket \*p;  zval \*zv;  uint32\_t n;  GC\_STACK\_DCL(stack); // gc\_stack \*\_stack = init; size\_t \_top = 0;  tail\_call: // 尾调用  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_WHITE)) { // 如果实例不是白色  goto next; // 转到下一个  }  if (GC\_REFCOUNT(ref) > 0) { // 如果实例是白色，引用数 > 0  // 如果实例不是黑色（如果是黑色应该在上面跳走了，这里只能是白色）  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_BLACK)) {  GC\_REF\_SET\_BLACK(ref); // 标记成黑色  if (UNEXPECTED(!\_stack->next)) { // 如果没有下一个页堆栈  gc\_stack\_next(\_stack); // 创建新的 stack  }  \_stack->next->prev = NULL; // 打断下一页到本页的联系  // 下一页堆栈一定是空的，否则不会切换到本页  gc\_scan\_black(ref, \_stack->next); // 把实例递归标记成黑色  \_stack->next->prev = \_stack; // 恢复下一页到本页的联系  }  goto next; // 下一个元素  }  ... |

GC\_REF\_SET\_BLACK()宏程序用于把实例设置成黑色，代码如下：

|  |
| --- |
| // 黑色值是0，实例上就是删除颜色值  #define GC\_REF\_SET\_BLACK(ref) do { \  GC\_TYPE\_INFO(ref) &= ~(GC\_COLOR << GC\_INFO\_SHIFT); \  } while (0) |

2）后续业务逻辑处理中，不改变元素的引用次数。

3）找到所有的灰色（而不是其他颜色）的实例，标记成白色（而不是灰色），并递归处理。代码对比示例如下：

|  |
| --- |
| **// gc\_mark\_grey()函数**  GC\_DELREF(ref); // 引用数-1  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) { // 如果不是灰色  GC\_REF\_SET\_COLOR(ref, GC\_GREY); // 标记成灰色  GC\_STACK\_PUSH(ref); // 准备递归处理  }  **// gc\_scan()函数**  if (GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) { // 如果是灰色  GC\_REF\_SET\_COLOR(ref, GC\_WHITE); // 标记成白色  GC\_STACK\_PUSH(ref); // 准备递归处理  } |

### 处理黑色实例

gc\_scan\_black()函数用于把实例递归标记成黑色，函数的逻辑框架与gc\_mark\_grey()函数一致，只是在处理实例时有些不同，代码对比示例如下：

|  |
| --- |
| **// gc\_mark\_grey()函数**  GC\_DELREF(ref); // 引用数-1  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) { // 如果不是灰色  GC\_REF\_SET\_COLOR(ref, GC\_GREY); // 标记成灰色  GC\_STACK\_PUSH(ref); // 入栈，准备递归处理  }  **// gc\_scan\_black()函数**  GC\_ADDREF(ref); // 引用数+1  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_BLACK)) { // 如果不是黑色  GC\_REF\_SET\_BLACK(ref); // 标记成黑色  GC\_STACK\_PUSH(ref); // 入栈，准备递归处理  } |

## 步骤三：回收指针列表

gc\_collect\_roots()函数用于回收指针列表：

|  |
| --- |
| // flags 用于返回 GC\_HAS\_DESTRUCTORS 标记：被回收的对象是否自带析构方法  static int gc\_collect\_roots(uint32\_t \*flags, gc\_stack \*stack) {  uint32\_t idx, end;  zend\_refcounted \*ref;  int count = 0;  gc\_root\_buffer \*current = GC\_IDX2PTR(GC\_FIRST\_ROOT); // 第一个指针  gc\_root\_buffer \*last = GC\_IDX2PTR(GC\_G(first\_unused)); // 最后一个最后一个有效指针  while (current != last) { // 遍历每一个指针，从列表中删除非垃圾元素  if (GC\_IS\_ROOT(current->ref)) { // 如果是普通指针，不是垃圾也不可复用  if (GC\_REF\_CHECK\_COLOR(current->ref, GC\_BLACK)) { // 如果实例是黑色  GC\_REF\_SET\_INFO(current->ref, 0); // 清空 type\_info 左边22位  gc\_remove\_from\_roots(current); // 回收一个指针  }  }  current++; // 下一个  }  gc\_compact(); // 回收完后，整理指针列表  **// 后半部分，收集白色实例**  idx = GC\_FIRST\_ROOT; // 第一个  end = GC\_G(first\_unused); // 最后一个  while (idx != end) { // 遍历  current = GC\_IDX2PTR(idx); // 取得 root 指针  ref = current->ref; // root 连接的计数器  current->ref = GC\_MAKE\_GARBAGE(ref); // 给指针添加 GC\_GARBAGE 标记  if (GC\_REF\_CHECK\_COLOR(ref, GC\_WHITE)) { // 如果是白色节点  GC\_REF\_SET\_BLACK(ref); // 标记成黑色  count += gc\_collect\_white(ref, flags, stack); // 把白色元素 递归标记成黑色  }  idx++; // 下个指针  }  return count; // 返回所有回收的非引用类型元素数量  } |

如上所示，业务逻辑分为前后两部分，前半部分遍历指针列表，把所有黑色（不可回收）的实例从列表中删除。后半部分为回收白色实例。

### 回收白色实例

gc\_collect\_white()函数用于把白色元素递归标记成黑色，函数的逻辑框架与gc\_mark\_grey()函数一致，但有几点不同：1）增加了计数：

|  |
| --- |
| static int gc\_collect\_white(zend\_refcounted \*ref, uint32\_t \*flags, gc\_stack \*stack) {  ...  tail\_call:  if (GC\_TYPE(ref) != IS\_REFERENCE) { // 只要不是引用类型  count++; // 计数+1  }  ... |

2）在处对象和理数组实例的开头增加了一些业务逻辑：

|  |
| --- |
| // 与gc\_mark\_grey()函数的第1块代码相似  if (GC\_TYPE(ref) == IS\_OBJECT) { // 实例是对象  zend\_object \*obj = (zend\_object\*)ref;  if (EXPECTED(!(OBJ\_FLAGS(ref) & IS\_OBJ\_FREE\_CALLED))) { // 对象没有被销毁过  int len;  zval \*table;  if (!GC\_INFO(ref)) { // GC\_INFO()反回0，说明是黑色实例  gc\_add\_garbage(ref); // 添加垃圾标记  }  // 对象没有被销毁过并且（销毁方法不是 zend\_objects\_destroy\_object 或 所属类有析构方法），说明对象有自己的销毁器  if (!(OBJ\_FLAGS(obj) & IS\_OBJ\_DESTRUCTOR\_CALLED)  && (obj->handlers->dtor\_obj != zend\_objects\_destroy\_object  || obj->ce->destructor != NULL)) {  \*flags |= GC\_HAS\_DESTRUCTORS; // 返回标记，有销毁器  }  ... |

和

|  |
| --- |
| // 与gc\_mark\_grey()函数的第1块代码相似  } else if (GC\_TYPE(ref) == IS\_ARRAY) { // 实例是数组  if (!GC\_INFO(ref)) { // GC\_INFO()反回0，说明是黑色实例  gc\_add\_garbage(ref); // 添加垃圾标记  }  ... |

3）找到所有的白色的实例，标记成黑色，并递归处理。代码对比示例如下：

|  |
| --- |
| **// gc\_mark\_grey()函数**  GC\_DELREF(ref); // 引用数-1  if (!GC\_REF\_CHECK\_COLOR(ref, GC\_GREY)) { // 如果不是灰色  GC\_REF\_SET\_COLOR(ref, GC\_GREY); // 标记成灰色  GC\_STACK\_PUSH(ref); // 入栈，准备递归处理  }  **// gc\_scan\_black()函数**  GC\_ADDREF(ref); // 引用数+1  if (GC\_REF\_CHECK\_COLOR(ref, GC\_WHITE)) { // 如果是白色  GC\_REF\_SET\_BLACK(ref); // 标记成黑色  GC\_STACK\_PUSH(ref); // 入栈，准备递归处理  } |

gc\_add\_garbage()函数用于把实例添加进回收列表，并标记成黑色：

|  |
| --- |
| static void gc\_add\_garbage(zend\_refcounted \*ref) {  uint32\_t idx;  gc\_root\_buffer \*buf;  if (GC\_HAS\_UNUSED()) { // 如果buf有可复用空间  idx = GC\_FETCH\_UNUSED(); // 取得 索引号  } else if (GC\_HAS\_NEXT\_UNUSED()) { // 如果有新空间  idx = GC\_FETCH\_NEXT\_UNUSED(); // 取得新指针索引号  } else { // 都没有  gc\_grow\_root\_buffer(); // 增加指针列表大小  if (UNEXPECTED(!GC\_HAS\_NEXT\_UNUSED())) { // 如果增加失败，直接返回，不报错  return;  }  idx = GC\_FETCH\_NEXT\_UNUSED(); // 取得新指针索引号  }  buf = GC\_IDX2PTR(idx); // 通过索引号取得指针  buf->ref = GC\_MAKE\_GARBAGE(ref); // 给指针添加 GC\_GARBAGE 标记  idx = gc\_compress(idx); // 压缩索引号  GC\_REF\_SET\_INFO(ref, idx | GC\_BLACK); // 标记成黑色  GC\_G(num\_roots)++; // 使用root数量增加  } |

## 步骤四：锁定zend\_fiber

代码如下：

|  |
| --- |
| **// zend\_gc\_collect\_cycles()函数的第2块代码**  zend\_fiber\_switch\_block(); // 步骤4：锁定fiber |

关于fiber的更多内容参见相关章节。

## 步骤五：处理有销毁器的对象

代码如下：

|  |
| --- |
| **// zend\_gc\_collect\_cycles()函数的第3块代码**  // 步骤5：如果有析构方法，这里才是真的开始销毁对象，释放内存  if (gc\_flags & GC\_HAS\_DESTRUCTORS) { // 如果有实例有销毁器  should\_rerun\_gc = 1; // 只要有析构方法就要重来  idx = GC\_FIRST\_ROOT; // 第一个元素索引号  current = GC\_IDX2PTR(GC\_FIRST\_ROOT); // 第一个元素位置  while (idx != end) { **// 步骤1.5.1：遍历每一个，打标记**  if (GC\_IS\_GARBAGE(current->ref)) { // 如果有垃圾标记  p = GC\_GET\_PTR(current->ref); // 取得指针  // 如果是对象 并且 对象没有销毁过  if (GC\_TYPE(p) == IS\_OBJECT  && !(OBJ\_FLAGS(p) & IS\_OBJ\_DESTRUCTOR\_CALLED)) {  zend\_object \*obj = (zend\_object \*) p; // 找到对象  // 如果有自己的或者所属类的析构方法  if (obj->handlers->dtor\_obj != zend\_objects\_destroy\_object  || obj->ce->destructor) {  current->ref = GC\_MAKE\_DTOR\_GARBAGE(obj); // 有销毁器的垃圾  GC\_REF\_SET\_COLOR(obj, GC\_PURPLE); **// 颜色标记成紫色**  } else {  GC\_ADD\_FLAGS(obj, IS\_OBJ\_DESTRUCTOR\_CALLED); // 对象已销毁  }  }  }  current++; // 下一个  idx++; // 序号下一个  }  **// 步骤5.2：删除将要销毁的对象的嵌套数据。不会删除对象本身，它们已被标记成紫色**  idx = GC\_FIRST\_ROOT; // 第一个root的索引号  current = GC\_IDX2PTR(GC\_FIRST\_ROOT); // 第一个root的指针  while (idx != end) { // 遍历每一个  if (GC\_IS\_DTOR\_GARBAGE(current->ref)) { // 如果是有销毁器的垃圾  p = GC\_GET\_PTR(current->ref); // 取得回收指针  // 递归所有元素，找到黑色元素，把它和回收指针解除关联  count -= gc\_remove\_nested\_data\_from\_buffer(p, current, &stack);  }  current++; // 下一个root  idx++; // 下一个索引号  }  **// 步骤5.3：真正开始销毁**  idx = GC\_FIRST\_ROOT; // 从第一个开始  while (idx != end) { // 遍历每一个  current = GC\_IDX2PTR(idx); // root指针  if (GC\_IS\_DTOR\_GARBAGE(current->ref)) { // 如果是有销毁器的垃圾  p = GC\_GET\_PTR(current->ref); // 取得指针，这里清空了附加标记  // 把这个指针标记成普通的，没有附加标记，下一轮回收时它就不再是垃圾了  current->ref = p;  // 再次检查销毁器尚未被调用，因为它可以被其他销毁器间接调用  if (!(OBJ\_FLAGS(p) & IS\_OBJ\_DESTRUCTOR\_CALLED)) { // 对象未销毁  zend\_object \*obj = (zend\_object\*)p;  GC\_ADD\_FLAGS(obj, IS\_OBJ\_DESTRUCTOR\_CALLED); // 对象标记已销毁  GC\_ADDREF(obj); // 增加引用次数  obj->handlers->dtor\_obj(obj); // 调用销毁器，销毁对象  GC\_DELREF(obj); // 减少引用次数  }  }  idx++; // 下一个root  }  if (GC\_G(gc\_protected)) { **// 步骤5.4：如果是受保护状态，说明出错了**  zend\_get\_gc\_buffer\_release(); // 释放缓冲区  zend\_fiber\_switch\_unblock(); // 解锁fiber  return 0; // 返回回收数0  }  }  ... |

如上所示，代码主要分4部分：

1）遍历指针列表，找到所有有垃圾标记的指针，如果有销毁器，添加“有销毁器的垃圾”标记，并把实例标记成紫色，否则给实例添加“已销毁”标记；

2）遍历指针列表，找到有销毁器的对象，遍历它们，碰到黑色实例，递归从垃圾回收列表中删除；

3）遍历指针列表，销毁有销毁器的对象；

4）如果开启了垃圾回收保护，中断操作。

gc\_remove\_nested\_data\_from\_buffer()函数用于把实例递归从垃圾回收列表中删除：

|  |
| --- |
| static int gc\_remove\_nested\_data\_from\_buffer(zend\_refcounted \*ref, gc\_root\_buffer \*root, gc\_stack \*stack) {  HashTable \*ht;  Bucket \*p;  zval \*zv;  uint32\_t n;  int count = 0;  GC\_STACK\_DCL(stack); // gc\_stack \*\_stack = init; size\_t \_top = 0;  tail\_call:  if (root) { // 如果指针有效  root = NULL; // 置空  count++; // 数量 +1，对所有的元素递归计数  **// 如果实例中有记录指针地址，并且颜色是黑色**  } else if (GC\_REF\_ADDRESS(ref) != 0  && GC\_REF\_CHECK\_COLOR(ref, GC\_BLACK)) {  GC\_REMOVE\_FROM\_BUFFER(ref); // **把实例和回收指针解除关联，并把实例标为黑色**  count++; // 数量 +1  } else if (GC\_TYPE(ref) == IS\_REFERENCE) { // 如果是引用类型  if (Z\_REFCOUNTED(((zend\_reference\*)ref)->val)) { // 如果目标是可计数类型  ref = Z\_COUNTED(((zend\_reference\*)ref)->val); // 转到目标  goto tail\_call; // 从头开始  }  goto next; // 如果不是可计数类型，下一个  } else { // 其他情况，跳过这个实例，从堆栈中弹出一下个实例进行处理  goto next;  }    if (GC\_TYPE(ref) == IS\_OBJECT) { // 如果类型是对象  zend\_object \*obj = (zend\_object\*)ref; // 转成对象  if (EXPECTED(!(OBJ\_FLAGS(ref) & IS\_OBJ\_FREE\_CALLED))) { // 如果对象没有被释放过  int len;  zval \*table;  ht = obj->handlers->get\_gc(obj, &table, &len); // 取得需要回收的数据  n = len;  zv = table;  if (UNEXPECTED(ht)) { // 如果哈希表存在  // 先处理zval列表  for (; n != 0; n--) { // 遍历zval列表的每一个元素  if (Z\_REFCOUNTED\_P(zv)) { // 如果是可计数类型  ref = Z\_COUNTED\_P(zv); // 从zval中取出实例指针  GC\_STACK\_PUSH(ref); // 实例指针压入堆栈  }  zv++; // 下一个zval  }  // 如果实例中有记录指针序号，并且颜色是黑色  if (GC\_REF\_ADDRESS(ht) != 0 && GC\_REF\_CHECK\_COLOR(ht, GC\_BLACK)) {  **// gc\_remove\_from\_buffer()，把ht和回收指针解除关联，并把实例标为黑色**  GC\_REMOVE\_FROM\_BUFFER(ht);  }  goto handle\_ht; // 处理哈希表  }  handle\_zvals: // 处理zval列表  for (; n != 0; n--) { // 遍历列表中每一个zval  if (Z\_REFCOUNTED\_P(zv)) { // 如果是可计数类型  ref = Z\_COUNTED\_P(zv); // 取得第一个可计数实例，准备处理这个实例  zv++; // 下一个zval  while (--n) { // 遍历后面的每一个zval  if (Z\_REFCOUNTED\_P(zv)) {  zend\_refcounted \*ref = Z\_COUNTED\_P(zv); // 取得实例  GC\_STACK\_PUSH(ref); // 向堆栈中压入实例  }  zv++; // 下一个zval  }  goto tail\_call; // 从头开始  }  zv++; // 下一个zval  }  }  } else if (GC\_TYPE(ref) == IS\_ARRAY) { // 如果类型是数组  ht = (zend\_array\*)ref; // 转成array指针  handle\_ht: // 处理哈希表  n = ht->nNumUsed;  if (HT\_IS\_PACKED(ht)) { // 如果是顺序数组  zv = ht->arPacked; // 取得元素列表开头  goto handle\_zvals; // 按zval列表处理  }  // 如果不是顺序数组，是哈希表  p = ht->arData; // 定位元素列表开头  for (; n != 0; n--) { // 遍历列表  zv = &p->val; // 元素值  if (Z\_TYPE\_P(zv) == IS\_INDIRECT) { // 如果类型是间接引用  zv = Z\_INDIRECT\_P(zv); // 追踪到引用目标  }  if (Z\_REFCOUNTED\_P(zv)) { // 如果是可计数类型  ref = Z\_COUNTED\_P(zv); // 找到第一个可计数实例，准备处理这个实例  p++; // 下一个bucket  while (--n) { // 遍历后面所有，把后面的有效实例压入堆栈  zv = &p->val; // 取得元素值  if (Z\_TYPE\_P(zv) == IS\_INDIRECT) { // 如果是间接引用  zv = Z\_INDIRECT\_P(zv); // 追踪到引用地址  }  if (Z\_REFCOUNTED\_P(zv)) { // 如果是可计数类型  zend\_refcounted \*ref = Z\_COUNTED\_P(zv); // 获得实例  GC\_STACK\_PUSH(ref); // 压入堆栈  }  p++; // 下一个bucket  }  goto tail\_call; // 从头开始  }  p++; // 下一个bucket  }  }  next:  ref = GC\_STACK\_POP(); // 从堆栈中弹出实例指针  if (ref) { // 如果实例指针有效  goto tail\_call; // 处理这个实例  }  return count; // 返回递归累计数量  } |

## 步骤六：释放临时堆栈象

这时候不再需要用到临时堆栈了，可以把临时堆栈释放掉，代码如下：

|  |
| --- |
| **// zend\_gc\_collect\_cycles()函数的第4块代码**  gc\_stack\_free(&stack); // 步骤6：释放 stack |

## 步骤七：释放实例的关联数据

这一步开始释放对象，先释放每个对象的关联数据，代码如下：

|  |
| --- |
| **// zend\_gc\_collect\_cycles()函数的第5块代码**  // 步骤7  idx = GC\_FIRST\_ROOT; // 从第一个指针开始  while (idx != end) { // 遍历每一个  current = GC\_IDX2PTR(idx); // 通过序号取得指针  if (GC\_IS\_GARBAGE(current->ref)) { // 如果指针有垃圾标记  p = GC\_GET\_PTR(current->ref); // 取出不带标记的指针  if (GC\_TYPE(p) == IS\_OBJECT) { // 如果类型是对象， 这个分支比较复杂  zend\_object \*obj = (zend\_object\*)p; // 转成对象  // 删除对象在全局列表中的指针  EG(objects\_store).object\_buckets[obj->handle] = SET\_OBJ\_INVALID(obj);  // 清空type\_info最后4位（类型）  GC\_TYPE\_INFO(obj)=GC\_NULL | (GC\_TYPE\_INFO(obj) & ~GC\_TYPE\_MASK); // 在调用free\_obj()函数前修改指针，因为free\_obj()函数会导致指针重新分配  // 给实例指针添加 GC\_GARBAGE 标记，offset用于校正指针，通常是0（ 参见：\_zend\_object\_handlers 结构体）  current->ref = GC\_MAKE\_GARBAGE(((char\*)obj) - obj->handlers->offset);  if (!(OBJ\_FLAGS(obj) & IS\_OBJ\_FREE\_CALLED)) { // 如果对象没有被释放  GC\_ADD\_FLAGS(obj, IS\_OBJ\_FREE\_CALLED); // 添加已释放标记  GC\_ADDREF(obj); // 增加引用次数，防止重复释放  obj->handlers->free\_obj(obj); // 删除对象关联的数据，对象本身未被释放  GC\_DELREF(obj); // 减少引用次数  }  // 回收对象仓库中的指针  ZEND\_OBJECTS\_STORE\_ADD\_TO\_FREE\_LIST(obj->handle);  } else if (GC\_TYPE(p) == IS\_ARRAY) { // 如果类型是数组  zend\_array \*arr = (zend\_array\*)p; // 数组指针  // 清空type\_info后4位（类型信息）  GC\_TYPE\_INFO(arr) = GC\_NULL | (GC\_TYPE\_INFO(arr) & ~GC\_TYPE\_MASK);  zend\_hash\_destroy(arr); // 销毁数组的数据块，没有销毁数组本身  }  }  idx++; // 下一个指针  } |

SET\_OBJ\_INVALID()宏程序用于把对象仓库中的指针标记成无效，相关代码如下：

|  |
| --- |
| #define OBJ\_BUCKET\_INVALID (1<<0) // 无效对象标记  #define SET\_OBJ\_INVALID(o) ((zend\_object\*)((((zend\_uintptr\_t)(o)) | OBJ\_BUCKET\_INVALID))) |

ZEND\_OBJECTS\_STORE\_ADD\_TO\_FREE\_LIST()宏程序用于回收对象仓库（objects\_store）里面的对象指针，传入的参数h（handle）是对象在全局对象容器（EG(objects\_store).object\_buckets）中的序号，通过它可以直接在全局容器中找到对象指针：

|  |
| --- |
| #define ZEND\_OBJECTS\_STORE\_ADD\_TO\_FREE\_LIST(h) do { \  /\* h是释放的指针元素序号，让这个指针指向原本的第一个空指针，让空指针形成一个链表 \*/ \  SET\_OBJ\_BUCKET\_NUMBER(EG(objects\_store).object\_buckets[(h)], EG(objects\_store).free\_list\_head); \  EG(objects\_store).free\_list\_head = (h); /\* 把新回收的指针序号放在最前面，作为链表开头 \*/ \  } while (0)  // 把要回收的序号n左移1位, 最后一位写成1，然后转成 zend\_object 指针，写到 o 里面  #define SET\_OBJ\_BUCKET\_NUMBER(o, n) do { \  (o) = (zend\_object\*)((((zend\_uintptr\_t)(n)) << 1) | OBJ\_BUCKET\_INVALID); \  } while (0) |

如上所示，对象仓库（objects\_store）和垃圾回收指针列表的结构相似，使用方法也相似。

## 步骤八：释放对象

在这一步前，数组（zend\_array）或对象（zend\_object）实例的关联数据都已经被删除了，这一步要删除实例本身，代码如下：

|  |
| --- |
| **// zend\_gc\_collect\_cycles()函数的第6块代码**  // 步骤8：释放对象（释放内存，这是最后步骤了）  current = GC\_IDX2PTR(GC\_FIRST\_ROOT); // 第一个root的指针  last = GC\_IDX2PTR(end); // 最后一个root的指针  while (current != last) { // 遍历每一个  if (GC\_IS\_GARBAGE(current->ref)) { // 如果指针带了【垃圾】标记  p = GC\_GET\_PTR(current->ref); // 取得清空标记的指针  GC\_LINK\_UNUSED(current); // 回收指针，和实例解除关联  GC\_G(num\_roots)--; // root数量 -1  efree(p); // 释放实例，这里是释放数组（zend\_array）或对象（zend\_object）  }  current++; // 下一个root  } |

如上所示，经过这一步，实例和指针都已经被彻底回收了。

## 步骤九：后续处理

代码如下：

|  |
| --- |
| **// zend\_gc\_collect\_cycles()函数的第7块代码**  zend\_fiber\_switch\_unblock(); // 解锁fiber  GC\_G(collected) += count; // 累计总回收数量  total\_count += count; // 累计本轮回收数量  GC\_G(gc\_active) = 0; // 标记，当前不在回收中  }  gc\_compact(); // 整理buf，没有GC\_G(num\_roots)的话直接到这里了  // 如果需要二次回收（只要有一个实例有特殊的析构方法，就要重来），并且没进行过二次回收  if (should\_rerun\_gc && !did\_rerun\_gc) {  did\_rerun\_gc = 1; // 标记，已经重来过  goto rerun\_gc; // 从头再来  }  finish: // 完成  zend\_get\_gc\_buffer\_release(); // 释放缓冲区  zend\_gc\_root\_tmpvars(); // 步骤5：  return total\_count; // 返回本轮回收数量  } |

# 未解疑问

修改指针时更便捷吗？还是无法修改？

|  |
| --- |
| gc\_root\_buffer \*buf;  zend\_refcounted \*p1=NULL;  \*buf = (gc\_root\_buffer\*)p1; // 常规方法  buf->ref=p1; // 更便捷的方法 |

# 未编目

## zend\_gc\_root\_tmpvars()

zend\_gc\_root\_tmpvars()

|  |
| --- |
| // zend\_gc\_collect\_cycles 一处引用。  // TMPVAR 运算对象通过 zval\_ptr\_dtor\_nogc 来销毁，因为它们通常不能包含回收周期。  // 但仍然会有一些少见的异常可能存在，在这样的用例里，依赖 producing code 来存放值。  // 如果 一个GC运行在值的 rooting 和消费中间，需要结束这样的泄露。为了避免这种情况发生，把所有 活跃 的 TMPVAR 值存放在这里。  /\* TMPVAR operands are destroyed using zval\_ptr\_dtor\_nogc(), because they usually cannot contain  \* cycles. However, there are some rare exceptions where this is possible, in which case we rely  \* on the producing code to root the value. If a GC run occurs between the rooting and consumption  \* of the value, we would end up leaking it. To avoid this, root all live TMPVAR values here. \*/  static void zend\_gc\_root\_tmpvars(void) {  zend\_execute\_data \*ex = EG(current\_execute\_data); // 当前的执行数据  // 只要ex有效就运行，运行完切换到下一个执行数据  for (; ex; ex = ex->prev\_execute\_data) {  zend\_function \*func = ex->func;  if (!func || !ZEND\_USER\_CODE(func->type)) { // 如果不在函数里或当前函数不是用户定义  continue; // 下一个执行数据  }  // 取得两个操作码之前的顺序差  // 当前 ex 的操作码，到当前函数的操作码列表开头，的偏移量  uint32\_t op\_num = ex->opline - ex->func->op\_array.opcodes;  // last\_live\_range？  for (uint32\_t i = 0; i < func->op\_array.last\_live\_range; i++) {  const zend\_live\_range \*range = &func->op\_array.live\_range[i];  if (range->start > op\_num) { // 区域开始位置大于op\_num  break;  }  if (range->end <= op\_num) { // 区域结束位置大于op\_num  continue;  }  uint32\_t kind = range->var & ZEND\_LIVE\_MASK; // 取出变量类型  if (kind == ZEND\_LIVE\_TMPVAR) {  uint32\_t var\_num = range->var & ~ZEND\_LIVE\_MASK;  zval \*var = ZEND\_CALL\_VAR(ex, var\_num); // 取出编号对应的zval  if (Z\_REFCOUNTED\_P(var)) { // 如果是可计数类型  gc\_check\_possible\_root(Z\_COUNTED\_P(var)); // 找个合适的root放进去  }  }  }  }  } |

## 声名但无调用的函数和宏程序

|  |
| --- |
| // ref->u.type\_info中添加紫色标记，全局无调用  #define GC\_REF\_SET\_PURPLE(ref) do { \  GC\_TYPE\_INFO(ref) |= (GC\_COLOR << GC\_INFO\_SHIFT); \  } while (0) |

# Change Log

2025.2.11 创建，基本数据结构

2025.2.12 创建，启用功能，添加指针

2025.2.13 回收指针

2025.2.14 到mark\_grey()函数

2025.2.18 开始垃圾回收章节

2025.2.19 缓冲区，垃圾回收框架。一边整理一加添加，框架基本完成了，需要仔细整理和测试。

2025.2.20 梳理主要回收逻辑，基本完成一遍，先放放，后续再修改