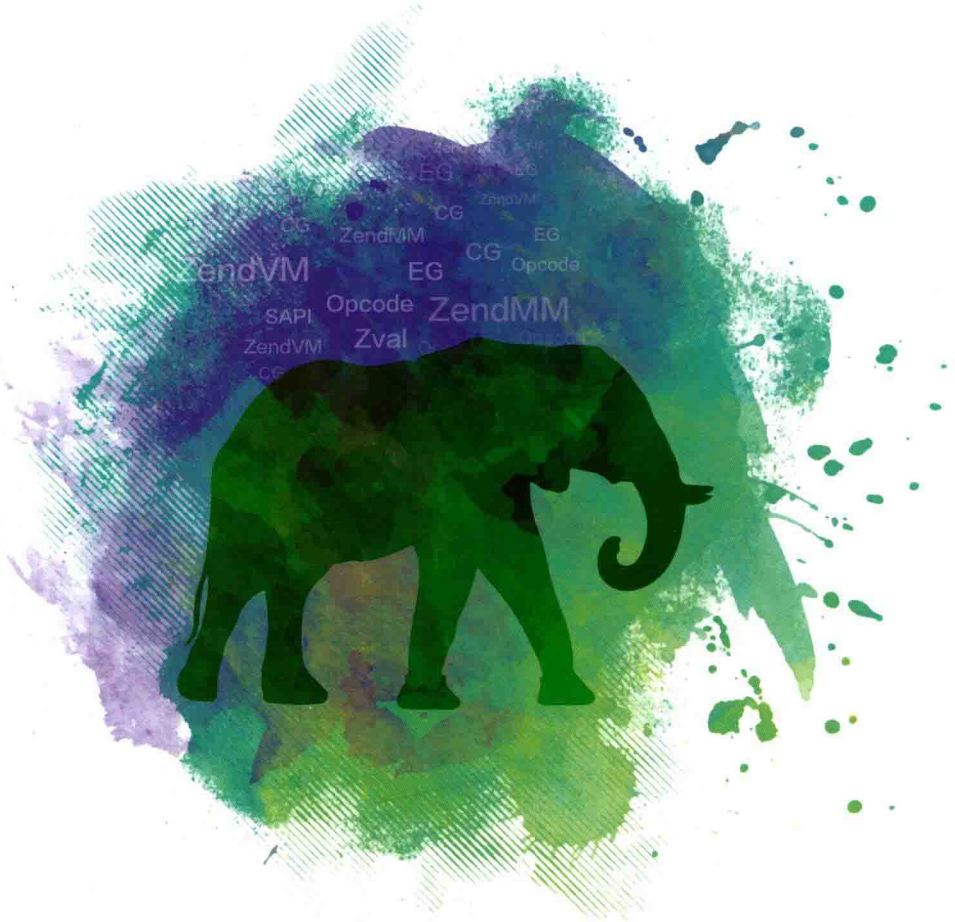
Broadview
www.broadview.com.cn



PHP7内核剖析

基于PHP7版本，围绕PHP的SAPI、数据类型、内存管理、编译与执行、函数、类及基础语法等的实现，深刻揭示PHP底层Zend引擎的实现原理

秦朋 / 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

PHP7内核剖析

专家评价

《PHP7内核剖析》有别于市面上的其他PHP技术图书，它不是介绍PHP如何应用，而是深入讲解PHP语言的底层实现原理。它面向的是具备较多PHP项目经验的中高级的开发者。阅读本书可以帮助开发者了解PHP内核实现，对PHP有更深入、更全面、更清晰的理解，有助于开发者将自身技术水平提升到一个新的层次。

——韩天峰（车轮互联总架构师，Swoole创始人）

此书图文并茂，内容翔实、细致，非常适合对PHP有一定了解，想深入学习PHP运行机制的同学。

——信海龙（阿里巴巴技术专家）

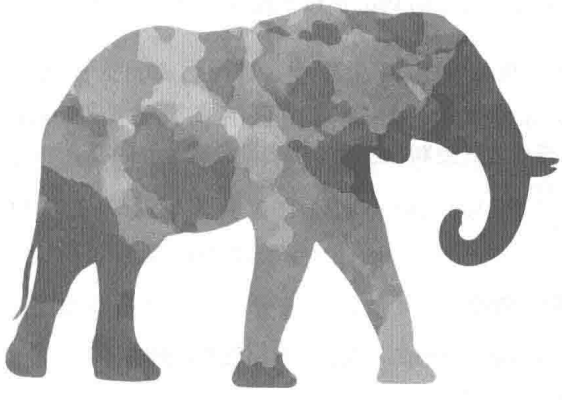
在PHP社区中，关于PHP内核的资料非常有限，大部分PHP开发者停留在使用的阶段，而对PHP的内部实现少有涉猎。该书弥补了这一缺憾，从变量、基础语法的实现到PHP的编译、执行，以及函数、面向对象的实现，非常全面、详细地介绍了PHP7底层的原理，强烈推荐！

——柏强利（滴滴资深开发工程师）

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：陈晓猛
封面设计：李　玲

上架建议：计算机>PHP
ISBN 978-7-121-32810-7
9 787121 328107 >
定价：89.00元



PHP7内核剖析

秦朋 / 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

PHP 作为最流行的语言之一，自第一个版本发布至今的二十几年里经历了多次重大改进，PHP7 版本最大的特色在于其性能上的突破，比 PHP5 快了一倍。目前 PHP7 已经得到了广泛应用，越来越多的项目从 PHP5 迁移到了 PHP7。目前，关于 PHP 内核的资料非常有限，本书以当前最为流行的 PHP7 版本为基础，系统性地、尽可能详细地介绍 PHP 语言底层的实现，旨在帮助更多的开发者进一步理解 PHP，参与到 PHP 的实现中，为未来 PHP 的发展贡献一份力量！全书内容主要包括 PHP 数据类型的实现、PHP 的编译及执行、PHP 内存的管理、函数及面向对象的实现、PHP 基础语法的实现，以及 PHP 扩展的开发。

本书适用于有一定 C 语言基础的 PHP 高级工程师，或者想了解 PHP7 的内部实现、扩展开发的工程师。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

PHP7 内核剖析 / 秦朋著. —北京：电子工业出版社，2017.10
ISBN 978-7-121-32810-7

Ⅰ. ①P… Ⅱ. ①秦… Ⅲ. ①PHP 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2017）第 242680 号

责任编辑：陈晓猛
印　　刷：北京京科印刷有限公司
装　　订：北京京科印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：30.75　　字数：590 千字
版　　次：2017 年 10 月第 1 版
印　　次：2017 年 10 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

序一

PHP 是一门优秀的 Web 开发的编程语言，一直说 PHP 是世界“最优秀”的语言，其他各个语言，包括 Python/Java 等语言都有相应的源码剖析或者内核解读之类的书籍，哪怕 MySQL/Redis 等都有相应的源码解读书籍。但是目前的图书市场，除了零零碎碎的一些 PHP 内核描述的文章，真正关于内核的书籍只有英文的《*Extending and Embedding PHP*》，中文电子版的《TIPI：深入理解 PHP 内核》算是相对比较专业的描述 PHP 内核特性的书籍。

本书从整个 PHP 的内部数据结构到内存管理 GC，从 PHP 脚本的编译执行原理到扩展开发，都有详实、深入的描述，是一本国内难得的描述 PHP 内核的佳作，非常值得推荐。看完以后，对整个 PHP 的内部理解，会更上一个新档次。

秦朋是我曾经在 360 公司的同事，多年前他就表现出对 PHP 技术的巨大兴趣，经过几年的努力和对内核的深入阅读理解，终于编写了本书。我读完内容，感慨万千，对很多 PHP 内部技术细节都理解非常透彻，并且文风通俗易懂，代码翔实清晰，确实是对 PHP 下了很深的功夫，也体现了不俗的技术水平，对秦朋的努力感到骄傲和佩服。希望本书能够给中国 PHP 行业带来新的理解和血液，为 PHP 程序员们带来提高和成长，也让我国 PHP 技术真正提升到一个新的台阶。

——谢华亮（黑夜路人）

序二

认识作者，是在公司内网发现他分享了几个关于 PHP 内核的文章，后来在钉钉上直接找他交流是否能转载到我的 PHP 饭米粒的公众号上，一来二去，就熟悉了。

世面上 PHP 的书籍不少，但对于 PHP 内核分析的书很少，能分析这么透彻就更少了，以前对 PHP5.x 版本做过粗略的分析，对 PHP7 的变化其实了解并不多，当作者把电子初版给我之后，一口气看了前面几章，从 SAPI 到 ZVAL 都写得很透彻，收获颇多。

当由 PC 互联网转战到移动互联网、物联网后，PHP 的优势确实小了，一些新的语言也陆陆续续冒出，也受到了不同程度的热捧，其实这些高并发、非阻塞都不是什么高大上的概念，大多数常用语言都能实现，但目前很多人并没有修练好内功，一但碰到问题，可能就会转向那些可以直接补坑的新事物上，而不是去理解这门语言。可以预料到的是，一但在新的语言上碰到坑，又会转向另一个，周而复始，对于自己，基本没有提高，所以透过现象看本质很有必要，也就能一通百通了。

另一方面，现在的人都比较浮躁，很少有年轻人能够沉下心去做深入的研究，我从业 10 多年，面试的人也众多，大多数人在工作三年左右就会遇到一个瓶颈，主要原因是对业务非常熟悉了，也没有挑战了，就想通过换个环境来找新鲜感。而有些人可以自我蜕变，从日常的业务中找到感兴趣的点深入学习，就如本书的作者一样，这给大多数人也指明了另一种突破的方式。

最后建议 PHPer 都可以精读此书，你就可以知道为什么 PHP 的一个变量类型可以变来变去，也可以知道为什么 PHP 的数组这么强大，深入之后，一定会为你打开一扇新的大门，让你在技术的道路上走得更扎实。

——王晶（滴滴技术专家，Swoole 开发者）

前　言

为什么要写这本书

PHP 作为最流行的语言之一，自第一个版本发布至今的二十几年里经历了多次重大的改进，尤其是 PHP7 版本的发布，其最大的亮点在于性能上的提升，比 PHP5 快了一倍。随着 PHP7 的不断普及，越来越多的项目从 PHP5 迁移到了 PHP7，毫无疑问，PHP7 将成为 PHP 历史上里程碑式的一个版本。我是在大学时代接触到的 PHP，初次相识就被其简洁、易用的语法所吸引了。在工作后的几年里，我一直使用 PHP 作为主要的开发语言。当然，除了 PHP，我也使用过很多其他语言，比如 C、C++、Java、Golang、Python 等，不同的语言有各自的特点、优势，让我印象最深的、也让我最喜欢的有 C、Golang、PHP。

- C

  这是我评价最高的一门语言，其强大的操控能力、简洁的语法、易于理解的处理方式无一不让我折服。编程语言本身只是控制计算机的一种工具，然而很多高级语言过度隔离了人与计算机间的联系，使得编程者并不理解计算机实际的工作机制，只能被编程语言限定在固定范围内，而 C 语言在这一点上做得恰到好处，其没有过度干预我们对计算机的操控，允许我们自由地控制内存、CPU。当然，C 语言也有很多不方便的地方，过于简单的接口使得很多操作不得不通过编写大量的代码来实现。

- Golang

  并发是我对其最大的印象，我们可以用更容易理解的方式来实现并发，但是它的内存控制没有 C 语言那么方便、灵活。

- PHP

  PHP 的底层是 C 语言实现的，这也使得它继承了很多 C 语言的基因，PHP 的简洁、易用、学习成本低等特点成就了它今天的地位。

PHP 的高度封装性与弱类型的特点使得很多操作极其简便，例如 JSON 的解析如果在 Golang 中完成，则需要定义一系列的结构体，然后才能完成解析，而在 PHP 中通过一行代码就可以完成。正是 PHP 底层的强大才得以实现如此简便的操作，那么强大的 PHP 背后到底是什么样子的呢？我想很多 PHPer 都有过这个疑问。然而让人感到沮丧的是，关于 PHP 内核的资料非常有限，已有的这些资料也不全面、系统，多数局限在理论介绍的层面上。后来我就直接去读 PHP 的源码，渐渐地发现，以前很多不理解的问题都在源码中找到了答案。本书主要的出发点是给那些想要了解 PHP 底层实现的读者一些启发，帮助更多的人理解 PHP 的实现，甚至能够参与到 PHP 的开发中，为未来 PHP 的发展贡献一份力量！

本书适合的对象

- 有一定 C 语言基础的读者。
- 想要理解 PHP 内部实现的读者。
- PHP 高级工程师。
- 对虚拟机实现感兴趣的读者。

本书不适合作为 PHP 的入门教程。书中对于基础性的、概念性的东西介绍很少，重点是源码解析。

本书的结构

本书总共分为 10 章，章节之间存在一定的衔接，建议按照先后顺序阅读。其中第 3～第 9 章为 Zend 引擎相关的内容，也是本书的核心章节。

第 1 章介绍 PHP 的基础内容。本章主要介绍 PHP 的历史发展、PHP7 的主要变化，重点讲解 PHP 的构成部分与生命周期的几个阶段。

第 2 章介绍 SAPI。本章选取了 PHP 三种常见的应用场景，介绍三个不同 SAPI 的实现：Cli、Fpm、Embed。SAPI 是 PHP 的接入层，如果只想了解 Zend 引擎的内容，那么可以跳过本章。

第 3 章介绍数据类型。本章主要介绍 PHP 中变量的基础结构 zval，以及不同类型的结构，它们是 PHP 中最基础的、使用最频繁的数据结构，通过本章的内容你将了解 PHP 中变量的内部实现。

第 4 章介绍内存管理。本章主要介绍 PHP 变量自动回收机制的实现，以及 PHP 底层内存

池、线程安全相关的实现。通过本章的内容，你将了解变量的内存是如何进行管理的，为什么PHP中的变量不需要手动申请释放。其中内存池的实现比较独立，它的实现与tcmalloc类似；线程安全只在多线程环境下使用，常见的Fpm、Cli模式不会用到，本书其他章节介绍的内容都是非线程安全的。

第5章介绍PHP的编译与执行。本章介绍PHP代码从编译到执行的整个过程，这也是Zend引擎的核心实现。通过对本章的学习，你将了解PHP代码是如何被Zend引擎识别、执行的。

第6章介绍函数的实现。本章介绍PHP中函数的实现，这也是Zend引擎的核心部分，本章的内容与第5章相关，介绍函数的编译与执行。

第7章介绍面向对象。本章介绍面向对象相关的实现，主要包括类、对象的内部实现。

第8章介绍命名空间。本章介绍PHP中命名空间的实现，这部分内容比较简单，命名空间只涉及编译阶段。

第9章介绍基础语法的实现。本章主要介绍PHP中基础语法的实现，比如条件分支、循环结构、中断跳转、静态变量、常量、全局变量、文件加载等，这些语法涉及PHP的编译、执行，它们是PHP语言的基础组成部分。通过对本章的学习，你可以更全面地掌握PHP语言的实现。

第10章介绍扩展开发。本章的内容偏向应用性，主要介绍扩展开发中常用的一些接口、宏。

勘误与支持

因个人水平有限，以及时间比较仓促，书中难免有不足之处，还望读者批评指正。如果你对本书有比较好的建议或对书中内容有所疑惑，可与我联系。

Email：pangudashu@gmail.com；QQ群：103330909

致谢

首先感谢PHP7的主要开发者鸟哥与PHP社区的其他开发者，正是他们的智慧造就了PHP，期待未来PHP能够有更加广阔的发展空间。在这里尤其要感谢Swoole的创始人韩天峰老师，本项目有幸得到韩老师的推荐，得到了众多人的关注。另外要单独感谢陈晓猛编辑，在他耐心地指导、审稿、修改工作下，最终才有了本书的诞生。

秦朋

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在下载资源处下载。
- **提交勘误**：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32810

目　录

第 1 章 PHP 基础架构

本章将简单介绍 PHP 的基本信息，以及 PHP 的安装、调试，同时将介绍 PHP 生命周期中的几个阶段，它们是 PHP 整个流程中比较关键的几个阶段。

1.1 简介

PHP 是一种非常流行的高级脚本语言，尤其适合 Web 开发，快速、灵活和实用是 PHP 最重要的特点。PHP 自 1995 年由 Lerdorf 创建以来，在全球得到了非常广泛的应用。

PHP 在 1995 年早期以 Personal Home Page Tools（PHP Tools）开始对外发表第一个版本，Lerdorf 写了一些介绍此程序的文档，并且发布了 PHP1.0。在这早期的版本中，提供了访客留言本、访客计数器等简单的功能，之后越来越多的网站开始使用 PHP，并且强烈要求增加一些特性，在新的成员加入开发行列之后，Rasmus Lerdorf 在 1995 年 6 月 8 日将 PHP 公开发布，希望可以通过社群来加速程序开发与寻找错误。这个版本被命名为 PHP2，已经有了今日 PHP 的一些雏型，类似 Perl 的变量命名方式、表单处理功能，以及嵌入到 HTML 中执行的能力。程序语法上也类似 Perl，有较多的限制，不过更简单、更有弹性。PHP/FI 加入了对 MySQL 的支持，从此建立了 PHP 在动态网页开发上的地位。到了 1996 年年底，有 15000 个网站使用了 PHP。

在 1997 年，任职于 Technion IIT 公司的两个以色列程序设计师 Zeev Suraski 和 Andi Gutmans 重写了 PHP 的解析器，成为 PHP3 的基础，而 PHP 也在这个时候改称为 PHP：Hypertext Preprocessor，1998 年 6 月正式发布 PHP3。Zeev Suraski 和 Andi Gutmans 在 PHP3 发布后开始

改写 PHP 的核心，这个在 1999 年发布的解析器被称为 Zend Engine，他们也在以色列的 Ramat Gan 成立了 Zend Technologies 来管理 PHP 的开发。

在 2000 年 5 月 22 日，以 Zend Engine 1.0 为基础的 PHP4 正式发布。2004 年 7 月 13 日发布了 PHP5，PHP5 则使用了第二代的 Zend Engine。PHP 包含了许多新特色：完全实现面向对象，引入 PDO，以及许多性能方面的改进。目前 PHP5.x 仍然是应用非常广泛的一个版本。

PHP 独特的语法混合了 C、Java、Perl 及 PHP 自创新的语法，同时支持面向对象、面向过程，相比 C、Java 等语言具有语法简洁、使用灵活、开发效率高、容易学习等特点。

- 开源免费：PHP 社群有大量活跃的开发者贡献代码。
- 快捷：程序开发快，运行快，技术本身学习快，实用性强。
- 效率高：PHP 消耗相当少的系统资源，自动 gc 机制。
- 类库资源：有大量可用类库供开发者使用。
- 扩展性：允许用户使用 C/C++扩展 PHP。
- 跨平台：可以在 UNIX、Windows、Mac OS 等系统上使用 PHP。

很多人认为 PHP 非常简单，没什么技术含量，这是非常片面的认识，任何语言都有其存在的价值、有其适合的应用领域，正是 PHP 底层的强大才造就了 PHP 语言的简洁、易用，这反而更能够提现出它的优秀所在。

1.2 安装及调试

在学习 PHP 内核之前，首先需要安装 PHP，以方便在学习过程中进行调试。本书使用的 PHP 版本为 7.0.12，下载地址为 http://php.net/distributions/php-7.0.12.tar.gz，下载后使用以下命令进行编译、安装：

```
$ tar zxvf php-7.0.12.tar.gz
$ cd php-7.0.12
$ ./configure --prefix=/usr/local/php7 --enable-debug --enable-fpm
```

--enable-debug 参数为开启 debug 模式，方便我们进行调试。关于调试自然少不了 gdb 了，PHP 内核的实现虽然比较复杂，但是阶段划分比较鲜明，可以通过 gdb 在各个阶段设置断点，然后进行相应的调试。学习内核时，可以使用 Cli 模式，因为它是单线程的，方便调试，这并不影响我们对内核的学习。同时，想要弄清楚 PHP 内核，自然少不了阅读 PHP 的源码，因此本书后面介绍的内容将会非常频繁地列举源码，而对于一些概念性的解释则点到为止。本书主要的目的是引导大家自己去阅读源码、探索 PHP 的实现，而不希望只简单地通过书中的描述来

了解 PHP，所以希望大家在阅读本书时，一定要准备好一份源码以便随时查看和调试。

1.3 PHP7 的变化

PHP7 与 PHP5 版本相比有非常大的变化，尤其是在 Zend 引擎方面。为提升性能，PHP7 对 Zend 进行了深度优化，使得 PHP 的运行速度大大提高，比 PHP5.0～5.6 快了近 5 倍，同时还降低了 PHP 对系统资源的占用。下面介绍 PHP7 比较大的几个变化。

1）抽象语法树

在 PHP 之前的版本中，PHP 代码在语法解析阶段直接生成了 ZendVM 指令，也就是在 zend_language_parser.y 中直接生成 opline 指令，这使得编译器与执行器耦合在一起。编译生成的指令供执行引擎使用，该指令是在语法解析时直接生成的，假如要把执行引擎换成别的，就需要修改语法解析规则；或者如果 PHP 的语法规则变了，但对应的执行指令没有变化，那么也需要对修改语法解析规则。

PHP7 中增加了抽象语法树，首先是将 PHP 代码解析生成抽象语法树，然后将抽象语法树编译为 ZendVM 指令。抽象语法树的加入使得 PHP 的编译器与执行器很好地隔离开，编译器不需要关心指令的生成规则，然后执行器根据自己的规则将抽象语法树编译为对应的指令，执行器同样不需要关心该指令的语法规则是什么样子的。

2）Native TLS

开发过 PHP5.x 版本扩展的读者对 TSRM_CC、TSRM_DC 这两个宏一定不会陌生，它们是用于线程安全的。PHP 中有很多变量需要在不同函数间共享，多线程的环境下不能简单地通过全局变量来实现，为了适应多线程的应用环境，PHP 提供了一个线程安全资源管理器，将全局资源进行了线程隔离，不同的线程之间互不干扰。

使用全局资源需要先获取本线程的资源池，这个过程比较占用时间，因此，PHP5.x 通过参数传递的方式将本线程的资源池传递给其他函数，避免重复查找。这种实现方式使得几乎所有的函数都需要加上接收资源池的参数，也就是 TSRM_DC 宏所加的参数，然后调用其他函数时再把这个参数传下去，不仅容易遗漏，而且这种方式极不优雅。

PHP7 中使用 Native TLS（线程局部存储）来保存线程的资源池，简单地讲就是通过__thread 标识一个全局变量，这样这个全局变量就是线程独享的了，不同线程的修改不会相互影响。具体的实现在本书第 4 章会详细介绍。

3）指定函数参数、返回值类型

PHP7 中可以指定函数参数及返回值的类型，例如：

```
function foo(string $name): array {
```

```
    return [];
}
```

这个函数的参数必须为字符串，返回值必须是数组，否则将会报 error 错误。

4）zval 结构的变化

zval 是 PHP 中非常重要的一个结构，它是 PHP 变量的内部结构，也是 PHP 内核中应用最为普遍的一个结构。在 PHP5.x 中，zval 的结构是下面这个样子的：

```
struct _zval_struct {
    /* Variable information */
    zvalue_value value;     /* value */
    zend_uint refcount__gc;
    zend_uchar type;    /* active type */
    zend_uchar is_ref__gc;
};
```

type 为类型，is_ref__gc 标识该变量是否为引用，value 为变量的具体值，它是一个 union，用来适配不同的变量类型：

```
typedef union _zvalue_value {
    long lval;                  /* long value */
    double dval;                /* double value */
    struct {
        char *val;
        int len;
    } str;
    HashTable *ht;              /* hash table value */
    zend_object_value obj;
    zend_ast *ast;
} zvalue_value;
```

zval 中还有一个比较重要的成员：refcount__gc，它记录变量的引用计数。引用计数是 PHP 实现变量自动回收的基础，也就是记录一个变量有多少个地方在使用的一种机制。PHP5.x 中引用计数是在 zval 中而不是在具体的 value 中，这样一来，导致变量复制时需要复制两个结构，zval、zvalue_value 始终绑定在一起。PHP7 将引用计数转移到了具体的 value 中，这样更合理。因为 zval 只是变量的载体，可以简单地认为是变量名，而 value 才是真正的值，这个改变使得 PHP 变量之间的复制、传递更加简洁、易懂。除此之外，zval 结构的大小也从 24byte 减少到了

16byte，这是PHP7能够降低系统资源占用的一个优化点所在。关于变量的结构及引用计数机制的具体实现，本书将在第3章、第4章详细介绍。

5）异常处理

PHP5.x中很多操作会直接抛出error错误，PHP7中将多数错误改为了异常抛出，这样一来就可以通过try catch捕捉到，例如：

```
try {
    test();
}catch(Throwable $e){
    echo $e->getMessage();
}
```

脚本中调用了一个不存在的函数，PHP5.x中报“PHP Fatal error: Call to undefined function test()”，而PHP7中可以通过Throwable异常类型进行捕获。新的异常处理方式使得错误处理更加可控。

6）HashTable的变化

HashTable，即哈希表，也被称为散列表，它是PHP中强大的array()类型的内部实现结构，也是内核中使用非常频繁的一个结构，函数符号表、类符号表、常量符号表等都是通过HashTable实现的。

PHP7中HashTable有非常大的变化，HashTable结构的大小从72byte减小到了56byte，同时，数组元素Bucket结构也从72byte减小到了32byte。新HashTable的实现在第3章时将详细说明。

7）执行器

execute_data、opline采用寄存器变量存储，执行器的调度函数为execute_ex()，这个函数负责执行PHP代码编译生成的ZendVM指令。在执行期间会频繁地用到execute_data、opline两个变量，在PHP5.x中，这两个变量是由execute_ex()通过参数传递给各指令handler的，在PHP7中不再采用传参的方式，而是将execute_data、opline通过寄存器来进行存储，避免了传参导致的频繁出入栈操作，同时，寄存器相比内存的访问速度更快。这个优化使得PHP的性能有了5%左右的提升，第5章将详细介绍这个特性。

8）新的参数解析方式

PHP5.x通过zend_parse_parameters()解析函数的参数，PHP7提供了另外一种方式，同时保留了原来的方式，但是新的解析方式速度更快，具体使用方式将在第10章介绍。

除了上面介绍的这些变化，PHP7中还有非常多的优化与新的特性，这里不再一一列举。本

书介绍的主要内容是关于 PHP7 的内核实现，后面的章节介绍的内容不会过多地与 PHP 旧版本的实现进行比较。

1.4 PHP 的构成

PHP 的源码下有几个主要目录：SAPI、main、Zend、ext。其中 SAPI 是 PHP 的应用接口层；main 为 PHP 的主要代码，主要是输入/输出、Web 通信，以及 PHP 框架的初始化操作等，比如 fastcgi 协议的解析、扩展的加载、PHP 配置的解析等工作都是由它来完成的，它位于 ZendVM 的上一层；Zend 目录是 PHP 解析器的主要实现，即 ZendVM，它是 PHP 语言的核心实现，PHP 代码的解释、执行就是由 Zend 完成的；ext 是 PHP 的扩展目录；TSRM 为线程安全相关的实现。PHP 各组成部分之间的关系如图 1-1 所示。

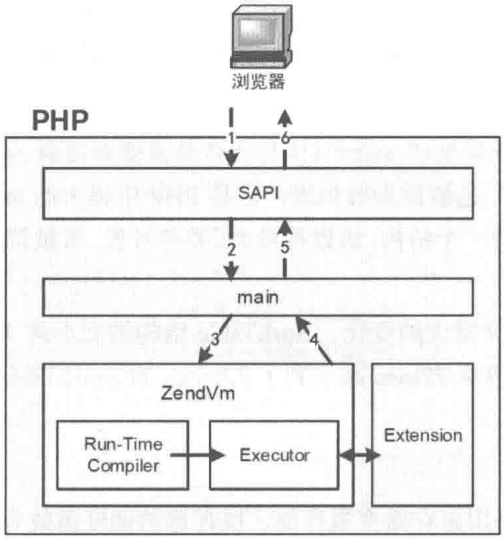


图 1-1 PHP 的基本构成

1）SAPI

PHP 是一个脚本解析器，提供脚本的解析与执行，它的输入是普通的文本，然后由 PHP 解析器按照预先定义好的语法规则进行解析执行。我们可以在不同环境中应用这个解析器，比如命令行下、Web 环境中、嵌入其他应用中使用。为此，PHP 提供了一个 SAPI 层以适配不同的应用环境，SAPI 可以认为是 PHP 的宿主环境。SAPI 也是整个 PHP 框架最外层的一部分，它主要负责 PHP 框架的初始化工作。如果 SAPI 是一个独立的应用程序，比如 Cli、Fpm，那么 main 函数也将定义在 SAPI 中。SAPI 的代码位于 PHP 源码的/sapi 目录下，经常用到的两个 SAPI 是 Cli、Fpm。

2）ZendVM

ZendVM是一个虚拟的计算机，它介于PHP应用与实际计算机中间，我们编写的PHP代码就是被它解释执行的。ZenVM是PHP语言的核心实现，它主要由两部分组成：编译器、执行器。其中编译器负责将PHP代码解释为执行器可识别的指令，执行器负责执行编译器解释出指令。ZendVM的角色等价于Java中的JVM，它们都是抽象出来的虚拟计算机，与C/C++这类编译型语言不同，虚拟机上运行的指令并不是机器指令。虚拟机的一个突出优点是跨平台，只需要按照不同平台编译出对应的解析器就可以实现代码的跨平台执行。本书大部分章节介绍的内容都是关于ZendVM的，如果想要深入理解PHP，那么ZendVM就是最主要的目标了。

3）Extension

扩展是PHP内核提供的一套用于扩充PHP功能的一种方式，PHP社区中有丰富的扩展可供使用，这些扩展为PHP提供了大量实用的功能，PHP中很多操作的函数都是通过扩展提供的。通过扩展，我们可以使用C/C++实现更强大的功能和更高的性能，这也使得PHP与C/C++非常相近，甚至可以在C/C++应用中把PHP嵌入作为第三库使用。扩展分为PHP扩展、Zend扩展，PHP扩展比较常见，而Zend扩展主要应用于ZendVM，它可以做的东西更多，我们所熟知的Opcache就是Zend扩展。

1.5 生命周期

PHP的整个生命周期被划分为以下几个阶段：模块初始化阶段（module startup）、请求初始化阶段（request startup）、执行脚本阶段（execute script）、请求关闭阶段（request shutdown）、模块关闭阶段（module shutdown）。根据不同SAPI的实现，各阶段的执行情况会有一些差异，比如命令行模式下，每次执行一个脚本都会完整地经历这些阶段，而FastCgi模式下则在启动时执行一次模块初始化，然后各个请求只经历请求初始化、执行请求脚本、请求关闭几个阶段，在SAPI关闭时经历模块关闭阶段。各阶段执行的顺序与对应的处理函数如图1-2所示。

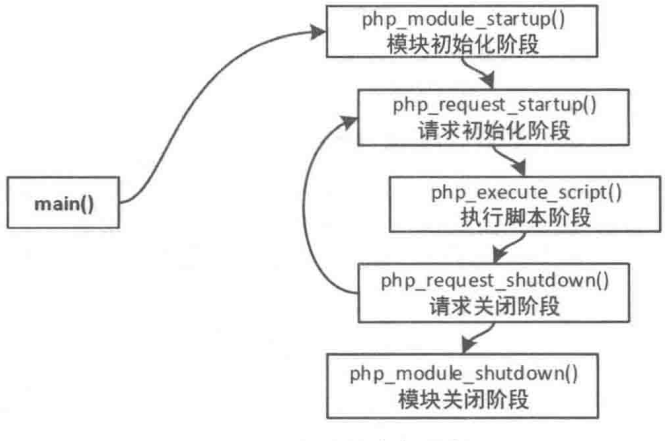


图1-2 PHP的生命周期

1. 模块初始化阶段

这个阶段主要进行 PHP 框架、Zend 引擎的初始化操作。该阶段的入口函数为 php_module_startup()，如图 1-3 所示。这个阶段一般只在 SAPI 启动时执行一次，对于 Fpm 而言，就是在 Fpm 的 master 进程启动时执行的。

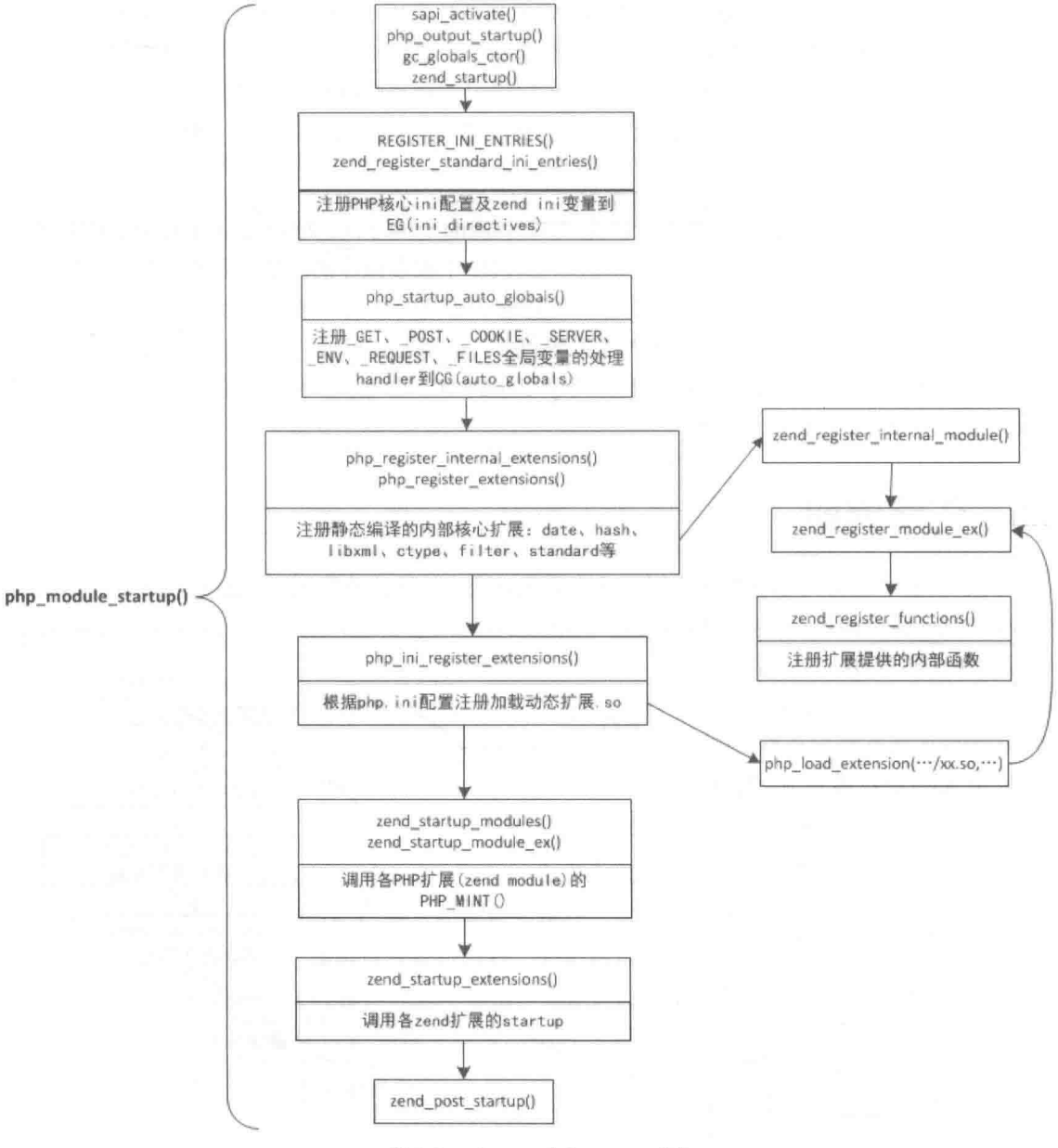


图 1-3 php_module_startup()

该阶段的几个主要处理如下所述。

- 激活 SAPI：sapi_activate()，初始化请求信息 SG(request_info)、设置读取 POST 请求的 handler 等，在 module startup 阶段处理完成后将调用 sapi_deactivate()。
- 启动 PHP 输出：php_output_startup()。
- 初始化垃圾回收器：gc_globals_ctor()，分配 zend_gc_globals 内存。
- 启动 Zend 引擎：zend_startup()，主要操作包括：
  - 启动内存池 start_memory_manager()；
  - 设置一些 util 函数句柄（如 zend_error_cb、zend_printf、zend_write 等）；
  - 设置 Zend 虚拟机编译、执行器的函数句柄 zend_compile_file、zend_execute_ex，以及垃圾回收的函数句柄 gc_collect_cycles；
  - 分配函数符号表（CG(function_table)）、类符号表（CG(class_table)）、常量符号表（EG(zend_constants)）等，如果是多线程的话，还会分配编译器、执行器的全局变量；
  - 注册 Zend 核心扩展：zend_startup_builtin_functions()，这个扩展是内核提供的，该过程将注册 Zend 核心扩展提供的函数，比如 strlen、define、func_get_args、class_exists 等；
  - 注册 Zend 定义的标准常量：zend_register_standard_constants()，比如：E_ERROR、E_WARNING、E_ALL、TRUE、FALSE 等；
  - 注册$GLOBALS 超全局变量的获取 handler；
  - 分配 php.ini 配置的存储符号表：EG(ini_directives)。
- 注册 PHP 定义的常量：PHP_VERSION、PHP_ZTS、PHP_SAPI，等等。
- 解析 php.ini：解析完成后所有的 php.ini 配置保存在 configuration_hash 哈希表中。
- 映射 PHP、Zend 核心的 php.ini 配置：根据解析出的 php.ini，获取对应的配置值，将最终的配置插入 EG(ini_directives)哈希表。
- 注册用于获取$_GET、$_POST、$_COOKIE、$_SERVER、$_ENV、$_REQUEST、$_FILES 变量的 handler。
- 注册静态编译的扩展：php_register_internal_extensions_func()。
- 注册动态加载的扩展：php_ini_register_extensions()，将 php.ini 中配置的扩展加载到 PHP 中。

- 回调各扩展定义的 module starup 钩子函数，即通过 PHP_MINIT_FUNCTION()定义的函数。
- 注册 php.ini 中禁用的函数、类：disable_functions、disable_classes。

2. 请求初始化阶段

该阶段是在请求处理前每一个请求都会经历的一个阶段，对于 Fpm 而言，是在 worker 进程 accept 一个请求且读取、解析完请求数据后的一个阶段。该阶段的处理函数为 php_request_startup()，如图 1-4 所示。

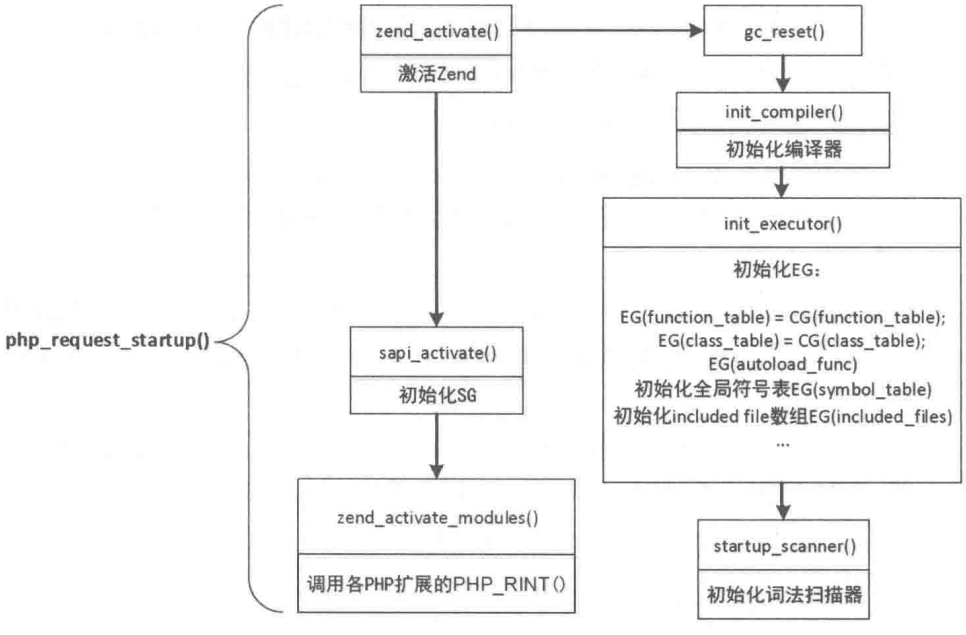


图 1-4 php_request_startup()

主要的处理有以下几个。

- 激活输出：php_output_activate()。
- 激活 Zend 引擎：zend_activate()，主要操作如下所述。
  - 重置垃圾回收器：gc_reset()；
  - 初始化编译器：init_compiler()；
  - 初始化执行器：init_executor()，将 EG(function_table)、EG(class_table)分别指向 CG(function_table)、CG(class_table)，所以在 PHP 的编译、执行期间，EG(function_table)与 CG(function_table)、EG(class_table)与 CG(class_table)是同一个

值；另外还会初始化全局变量符号表 EG(symbol_table)、include 过的文件符号表 EG(included_files)；

- 初始化词法扫描器：startup_scanner()。

- 激活 SAPI：sapi_activate()。
- 回调各扩展定义的 request startup 钩子函数：zend_activate_modules()。

3. 执行脚本阶段

该阶段包括 PHP 代码的编译、执行两个核心阶段，这也是 Zend 引擎最重要的功能。在编译阶段，PHP 脚本将经历从 PHP 源代码到抽象语法树再到 opline 指令的转化过程，最终生成的 opline 指令就是 Zend 引擎可识别的执行指令，这些指令接着被执行器执行，这就是 PHP 代码解释执行的过程，本书介绍的大部分内容都是关于这两个阶段的。这个接口的入口函数为 php_execute_script()，如图 1-5 所示。

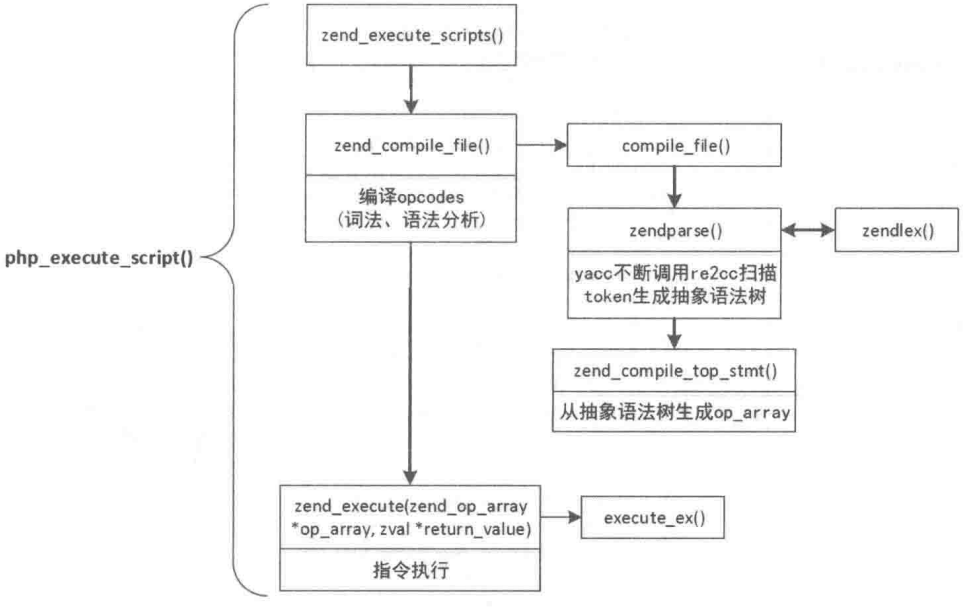


图 1-5 php_execute_script()

4. 请求关闭阶段

在 PHP 脚本解释执行完成后将进入请求关闭阶段，这个阶段将 flush 输出内容、发送 HTTP 应答 header 头、清理全局变量、关闭编译器、关闭执行器等。另外，在该阶段将回调各扩展的 request shutdown 钩子函数。该阶段是请求初始化阶段的相反操作，与请求初始化时的处理一一对应，如图 1-6 所示。

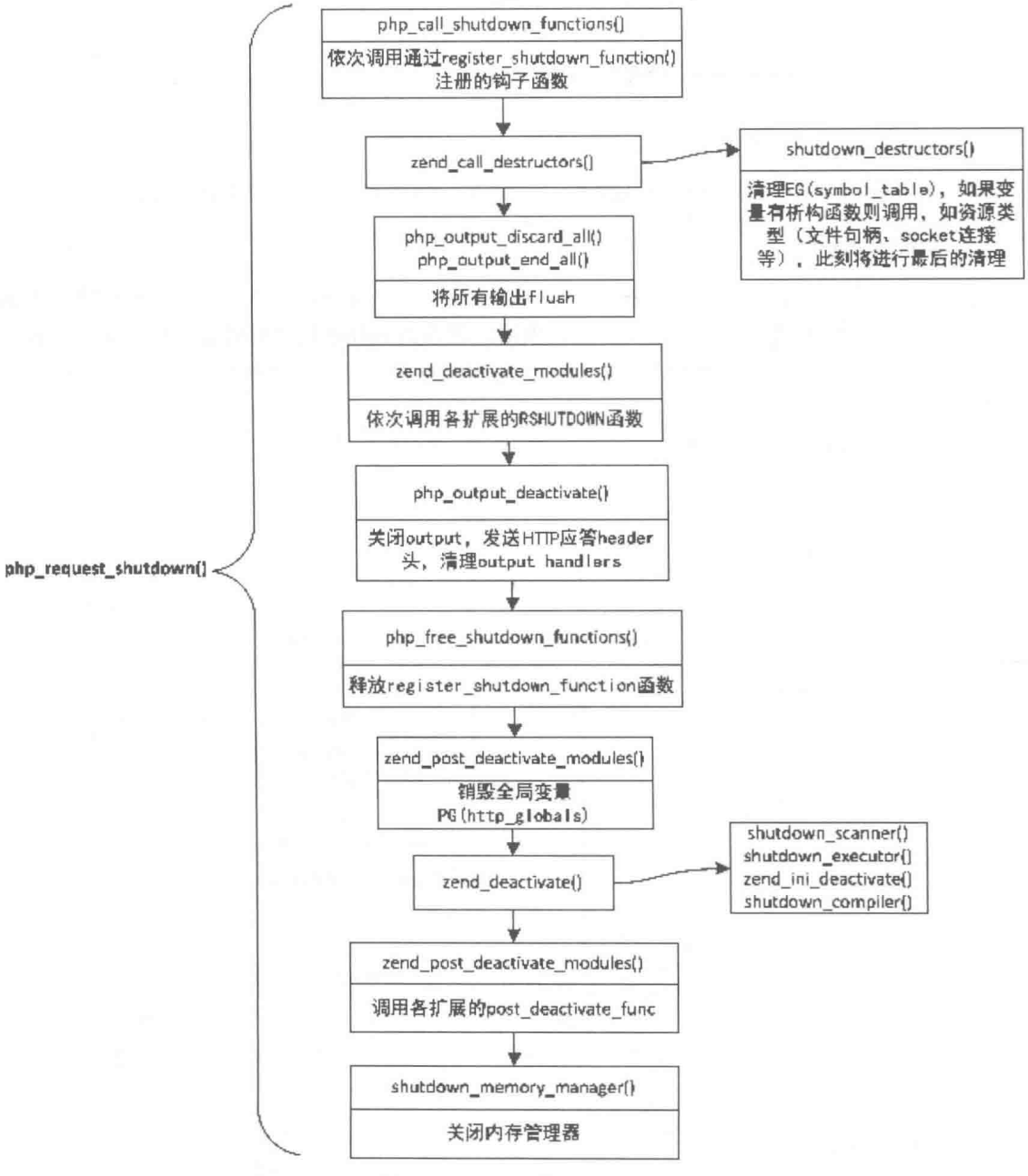


图 1-6 php_request_shutdown()

5. 模块关闭阶段

该阶段在 SAPI 关闭时执行，与模块初始化阶段对应，这个阶段主要进行资源的清理、PHP

各模块的关闭操作，同时，将回调各扩展的 module shutdown 钩子函数。具体的处理函数为 php_module_shutdown()，如图 1-7 所示。

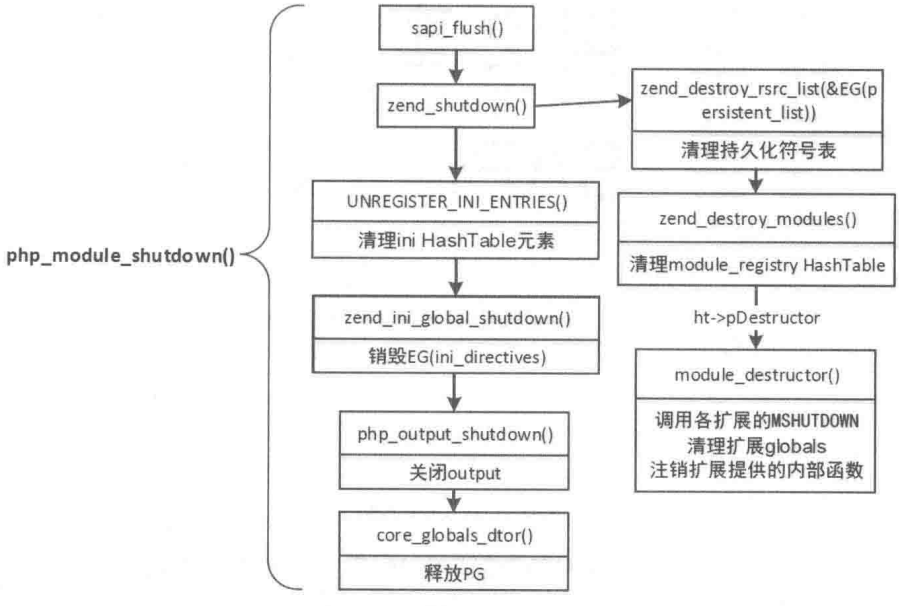


图 1-7 php_module_shutdown()

1.6 小结

本章主要介绍了 PHP7 与旧版本的一些变化，以及学习 PHP7 内核前的准备工作，另外简单介绍了 PHP 生命周期的几个阶段。在接下来的章节中，我们将逐步揭开 PHP 内核的神秘面纱，一点点深入到 PHP 的内部实现中，引导大家深入理解 PHP 的实现。

第2章 SAPI

这一章我们将首先介绍几个 SAPI 的实现，SAPI 是 PHP 框架的接口层，它是进入 PHP 内部的入口。PHP 中实现的 SAPI 有很多，本章选取了 3 个比较典型的 SAPI：Cli、Fpm、Embed，分别介绍 PHP 在不同场景下的应用。其中 Cli、Fpm SAPI 是完整实现的应用程序，它们有定义自己的 main 函数，方便我们从入口开始逐步分析 PHP 的处理，尤其是单进程的 Cli，非常方便调试，本书后面章节基本都是以 Cli 模式为例的。

不同 SAPI 的实现尽管会有差异，但是它们都是围绕着 PHP 的生命周期实现的，在分析 SAPI 的实现时，我们将以 PHP 生命周期的几个阶段为主线，这样更有助于理解。尽管 SAPI 中并不涉及 PHP 具体的内部实现，但它是 PHP 框架最外层的接口，了解清楚它们的实现是探索 PHP 内核的第一步。

2.1 Cli

Cli（Command Line Interface），即命令行接口，用于在命令行下执行 PHP 脚本，就像 Shell 那样，它是执行 PHP 脚本最简便的一种方式。Cli SAPI 最先是随 PHP4.2.0 版本发布的，但当时只是一个实验性的版本，需要在运行 ./configure 时加上 --enable-cli 参数。从 PHP4.3.0 版本开始，Cli SAPI 成为了正式模块，--enable-cli 参数会被默认设置为 on，也可以用参数--disable-cli 来屏蔽。

Cli 模式通过执行编译的 PHP 二进制程序即可启动，它定义了很多命令行参数，不同的参

数对应不同的处理，比如：执行 PHP 脚本文件、直接执行 PHP 代码（-r 参数）、输出 PHP 版本（-v 参数）、输出已安装的扩展（-m 参数）、指定 php.ini 配置（-c 参数）……直接在 PHP 命令后加 PHP 脚本则将执行该脚本。

```
//执行 PHP 脚本
$ php script.php
```

2.1.1 执行流程

Cli 是单进程模式，处理完请求后就直接关闭了，生命周期先后经历 module startup、request startup、execute script、request shutdown、module shutdown，其执行流程比较简单，关键的处理过程如下：

```
main() -> php_cli_startup() -> do_cli() -> php_module_shutdown()
```

Cli SAPI 的 main 函数位于/sapi/cli/php_cli.c 中，执行时首先解析命令行参数，然后初始化 sapi_module_struct，从结构体名称可以看出，它是记录 SAPI 信息的主要结构，这个结构中有几个函数指针，它们是内核定义的操作接口的具体实现，用来告诉内核如何读取、输出数据。

```
static sapi_module_struct cli_sapi_module = {
    "cli",                          /* name */
    "Command Line Interface",       /* pretty name */

    php_cli_startup,                /* startup */
    php_module_shutdown_wrapper,    /* shutdown */
    //请求初始化函数，Cli 没有定义
    NULL,                           /* activate */
    //请求收尾处理函数，Cli 中：fflush(stdout)
    sapi_cli_deactivate,            /* deactivate */
    //输出数据函数，Cli 默认是到标准输出
    sapi_cli_ub_write,              /* unbuffered write */
    sapi_cli_flush,                 /* flush */
    NULL,                           /* get uid */
    NULL,                           /* getenv */
    //错误处理函数
    php_error,                      /* error handler */
    //调用 header()函数的处理 handler，Cli 下是个空函数
```

```
    sapi_cli_header_handler,        /* header handler */
    //发送 header 时的函数
    sapi_cli_send_headers,          /* send headers handler */
    sapi_cli_send_header,           /* send header handler */
    //获取 POST 数据的函数
    NULL,                           /* read POST data */
    //获取 cookie 的函数，Cli 下是个空函数
    sapi_cli_read_cookies,          /* read Cookies */
    //向$_SERVER 中注册变量的函数
    sapi_cli_register_variables,    /* register server variables */
    sapi_cli_log_message,           /* Log message */
    NULL,                           /* Get request time */
    NULL,                           /* Child terminate */

    STANDARD_SAPI_MODULE_PROPERTIES
}
```

具体的处理过程比较简单，这里不再展开。在完成参数的解析及 sapi_module_struct 的基本初始化后，接下来进入 module startup 阶段。

```
if (sapi_module->startup(sapi_module) == FAILURE) {
    exit_status = 1;
    goto out;
}
```

前面 cli_sapi_module 变量中定义的 startup 函数为 php_cli_startup()，这个函数非常简单，直接调用了 php_module_startup()，关于此函数的处理第 1 章已经介绍过，这里不再赘述。

```
static int php_cli_startup(sapi_module_struct *sapi_module)
{
    if (php_module_startup(sapi_module, NULL, 0)==FAILURE) {
        return FAILURE;
    }
    return SUCCESS;
}
```

在 moduel startup 阶段处理完成后，接下来进入请求初始化阶段：

```
zend_first_try {
        if (sapi_module == &cli_sapi_module) {
            exit_status = do_cli(argc, argv);
        } else {
            //内置Web服务器的处理，下一节再单独介绍
            exit_status = do_cli_server(argc, argv);
        }
} zend_end_try();
```

do_cli()将完成请求的处理，此函数一开始对使用到的命令行参数进行解析，如果是一些查询系统信息之类的请求（如-v、-m、-i），则不需要经历 PHP 请求的生命周期，这里会单独处理，下面看一下执行 PHP 脚本请求时的处理。

```
zend_file_handle file_handle;
...
if (script_file) {
    //fopen请求的脚本文件
    if (cli_seek_file_begin(&file_handle, script_file, &lineno) != SUCCESS) {
        ...
    }
}
//输入类型为ZEND_HANDLE_FP，也就是FILE*
file_handle.type = ZEND_HANDLE_FP;
```

PHP 脚本执行时的输入形式有很多种，比如文件路径（filepath）、文件句柄（FILE）、文件描述符（fd）等，zend_file_handle 结构就是用来定义不同输入形式的，这样可以统一 PHP 执行函数的输入参数。

```
typedef struct _zend_file_handle {
    union {
        int             fd; //文件描述符
        FILE            *fp; //文件句柄
        zend_stream     stream; //zend封装的stream
    } handle;
    const char          *filename; //文件路径
    zend_string         *opened_path;
     //用于区分是哪种类型的：ZEND_HANDLE_FILENAME、ZEND_HANDLE_FD、ZEND_HANDLE_
FP、ZEND_HANDLE_STREAM、ZEND_HANDLE_MAPPED
```

```
    zend_stream_type  type;
    zend_bool free_filename;
} zend_file_handle;
```

Cli 中此处使用的是文件句柄，在 Linux 环境下也就是调用 fopen()打开一个文件，这样内核就可以直接读取 PHP 脚本代码了，当然也可以直接把文件路径提供给内核。定义好请求的输入结构后将进行请求初始化操作，即 request startup 阶段，然后开始 PHP 脚本的执行操作。

```
//request starup 阶段
if (php_request_startup()==FAILURE) {
    *arg_excp = arg_free;
     fclose(file_handle.handle.fp);
    PUTS("Could not startup.\n");
    goto err;
}
...
switch (behavior) {
    case PHP_MODE_STANDARD:
        ...
        //执行
        php_execute_script(&file_handle);
        ...
        break;
    case ... //其他执行模式，比如 php -r "php 代码"
}
```

完成脚本的处理后进入 request shutdown 阶段：

```
//do_cli:
out:
    if (request_started) {
        php_request_shutdown((void *) 0);
    }
```

do_cli()完成后回到 main()函数中，进入 module shutdown 阶段，最后进程退出，这就是 Cli 下执行一个脚本的生命周期。

```
//main:
```

```
out:
    if (module_started) {
        php_module_shutdown();
    }
    if (sapi_started) {
        sapi_shutdown();
    }
```

2.1.2 内置 Web 服务器

从 PHP5.4.0 起，Cli SAPI 提供了一个内置的 Web 服务器，这个内置的 Web 服务器主要用于本地开发使用，不可用于线上产品环境。URI 请求会被发送到 PHP 所在的的工作目录（Working Directory）进行处理，除非你使用了-t 参数来自定义不同的目录。

如果请求未指定执行哪个 PHP 文件，则默认执行目录内的 index.php 或者 index.html。如果这两个文件都不存在，服务器会返回 404 错误。

当你在命令行启动这个 Web Server 时，如果指定了一个 PHP 文件，则这个文件会作为一个“路由”脚本，意味着每次请求都会先执行这个脚本。如果这个脚本返回 FALSE ，那么直接返回请求的文件（例如请求静态文件不作任何处理）。

实际上，这个内置的 Web 服务器是一个独立的 SAPI，它有自己的 sapi_module_struct 结构，也就是说 Cli 定义了两个 SAPI。Cli 的这个功能很少使用，所以其具体的实现这里不再展开。

2.2 Fpm

Fpm（FastCGI Process Manager）是 PHP FastCGI 运行模式的一个进程管理器，从它的定义可以看出，Fpm 的核心功能是进程管理，那么它用来管理什么进程呢？这个问题需要从 FastCGI 说起。

FastCGI 是 Web 服务器（如 Nginx、Apache）和处理程序之间的一种通信协议，它是与 HTTP 类似的一种应用层通信协议。注意：它只是一种协议！

前面曾一再强调，PHP 是一个脚本解析器，可以简单地把它理解为一个黑盒函数，输入是 PHP 脚本，输出是脚本的执行结果。除了在命令行下执行脚本，能不能让 PHP 处理 HTTP 请求呢？这种场景下就涉及网络处理，需要接收请求、解析协议，处理完成后再返回处理结果。在网络应用场景下，PHP 并没有像 Golang 那样实现 HTTP 网络库，而是实现了 FastCGI 协议，然后与 Web 服务器配合实现了 HTTP 的处理，Web 服务器来处理 HTTP 请求，然后将解析的结果再通过 FastCGI 协议转发给处理程序，处理程序处理完成后将结果返回给 Web 服务器，Web 服

务器再返回给用户，如图 2-1 所示。

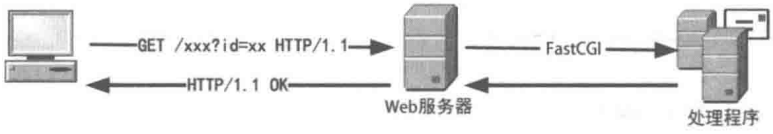


图 2-1 FastCGI 与 Web 服务器

PHP 实现了 FastCGI 协议的处理，但是并没有实现具体的网络处理，比较常用的网络处理模型有以下两种。

- **多进程模型**：由一个主进程和多个子进程组成，主进程负责管理子进程，基本的网络事件由各个子进程处理，Nginx 采用的就是这种模型。
- **多线程模型**：与多进程类似，只是它是线程粒度，这种模型通常会由主线程监听、接收请求，然后交由子线程处理，memcached 就是这种模式；有的也是采用多进程那种模式——主线程只负责管理子线程不处理网络事件，各个子线程监听、接收、处理请求，memcached 使用 UDP 协议时采用的是这种模式。

进程拥有独立的地址空间及资源，而线程则没有，线程之间共享进程的地址空间及资源，所以在资源管理上多进程模型比较简单，而多线程模型则需要考虑不同线程之间的资源冲突，也就是线程安全。

2.2.1 基本实现

Fpm 是一种多进程模型，它由一个 master 进程和多个 worker 进程组成。master 进程启动时会创建一个 socket，但是不会接收、处理请求，而是由 fork 出的 worker 子进程完成请求的接收及处理。

master 进程的主要工作是管理 worker 进程，负责 fork 或杀掉 worker 进程，比如当请求比较多 worker 进程处理不过来时，master 进程会尝试 fork 新的 worker 进程进行处理，而当空闲 worker 进程比较多时则会杀掉部分子进程，避免占用、浪费系统资源。

worker 进程的主要工作是处理请求，每个 worker 进程会竞争地 Accept 请求，接收成功后解析 FastCGI，然后执行相应的脚本，处理完成后关闭请求，继续等待新的连接，这就是一个 worker 进程的生命周期。从 worker 进程的生命周期可以看到：一个 worker 进程只能处理一个请求，只有将一个请求处理完成后才会处理下一个请求。这与 Nginx 的事件模型有很大的区别，Nginx 的子进程通过 epoll 管理套接字，如果一个请求数据还未发送完成则会处理下一个请求，即一个进程会同时连接多个请求，它是非阻塞的模型，只处理活跃的套接字。Fpm 的这种处理模式大大简化了 PHP 的资源管理，使得在 Fpm 模式下不需要考虑并发导致的资源冲突。

master 进程与 worker 进程之间不会直接进行通信，master 通过共享内存获取 worker 进程的信息，比如 worker 进程当前状态、已处理请求数等，master 进程通过发送信号的方式杀掉 worker 进程。

Fpm 可以同时监听多个端口，每个端口对应一个 worker pool，每个 pool 下对应多个 worker 进程，类似 Nginx 中 server 的概念，这些归属不同 pool 的 worker 进程仍由一个 master 管理。worker pool 的结构为 fpm_worker_pool_s，pool 之间构成一个链表。

```
struct fpm_worker_pool_s {
    //指向下一个 worker pool
    struct fpm_worker_pool_s *next;
    //php-fpm.conf 配置:pm、max_children、start_servers...
    struct fpm_worker_pool_config_s *config;
    //监听的套接字
    int listening_socket;
    ...
    //当前 pool 的 worker 链表，每个 worker 对应一个 fpm_child_s 结构
    struct fpm_child_s *children;
    //当前 pool 的 worker 运行总数
    int running_children;
    int idle_spawn_rate;
    int warn_max_children;
    //记录 worker 的运行信息，比如空闲、忙碌 worker 数
    struct fpm_scoreboard_s *scoreboard;
    ...
}
```

在 php-fpm.conf 中通过[pool name]声明一个 worker pool，每个 pool 各自配置监听的地址、进程管理方式、worker 进程数等。

```
[web1]
listen = 127.0.0.1:9000
...
[web2]
listen = 127.0.0.1:9001
...
```

上面这个例子配置了两个 worker pool，分别监听 9000、9001 端口，pool 下的 worker 进程

监听所属 pool 的端口，如图 2-2 所示。

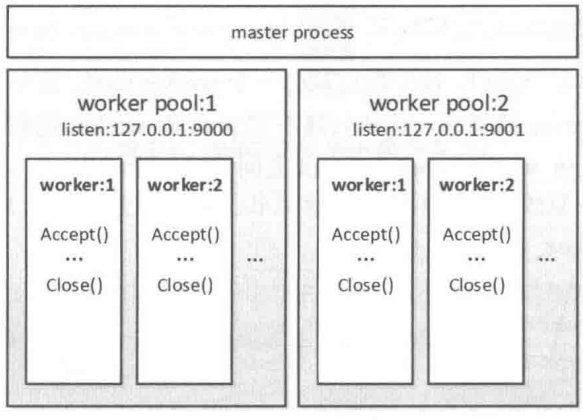


图 2-2 worker pool

2.2.2 Fpm 的初始化

Fpm 的 main 函数位于文件/sapi/fpm/fpm/fpm_main.c 中。Fpm 在启动后首先会进行 SAPI 的注册操作；接着会进入 PHP 生命周期的 module startup 阶段，在这个阶段会调用各个扩展定义的 MINT 钩子函数。然后会进行一系列的初始化操作，最后 master、worker 进程进入不同的处理环节。

```
int main(int argc, char *argv[])
{
    ...
    //注册 SAPI:将全局变量 sapi_module 设置为 cgi_sapi_module
    sapi_startup(&cgi_sapi_module);
    ...
    //执行 php_module_starup()
    if (cgi_sapi_module.startup(&cgi_sapi_module) == FAILURE) {
        return FPM_EXIT_SOFTWARE;
    }
    ...
    //初始化
    if(0 > fpm_init(...)){
        ...
    }
```

```
    ...
    fpm_is_running = 1;
    //后面都是 worker 进程的操作，master 进程不会走到下面
    fcgi_fd = fpm_run(&max_requests);
    parent = 0;
    ...
}
```

fpm_init()方法中将完成以下几个关键操作。

1）fpm_conf_init_main()

解析 php-fpm.conf 配置文件，为每个 worker pool 分配一个 fpm_worker_pool_s 结构，各 worker pool 的配置在解析后保存到 fpm_worker_pool_s->config 中，下面看一下 config 中的几个常用配置。

```
struct fpm_worker_pool_config_s {
    char *name; //pool 名称，即配置：[pool name]
    char *user;  //Fpm 的启动用户，配置：user
    char *group; //配置：group
    char *listen_address; //监听的地址，配置：listen
    ...
    int pm; //进程模型：static、dynamic、ondemand
    int pm_max_children; //最大 worker 进程数
    int pm_start_servers; //启动时初始化的 worker 数
    int pm_min_spare_servers; //最小空闲 worker 数
    int pm_max_spare_servers;  //最大空闲 worker 数
    int pm_process_idle_timeout; //worker 空闲时间
    int pm_max_requests; //worker 处理的最多请求数，超过这个值 worker 将被 kill
    ...
}
```

2）fpm_scoreboard_init_main()

分配用于记录 worker 进程运行信息的结构，此结构分配在共享内存上，master 正是通过这种方式获取 worker 的运行信息。分配时按照 worker pool 的最大 worker 进程数进行分配，每个 worker pool 分配一个 fpm_scoreboard_s 结构。pool 下的每个 worker 进程分配一个 fpm_scoreboard_proc_s 结构，这些结构的地址保存在 fpm_scoreboard_s->procs 数组中，其下标保存在 fpm_child_s->scoreboard_i 中，各结构的对应关系如图 2-3 所示。

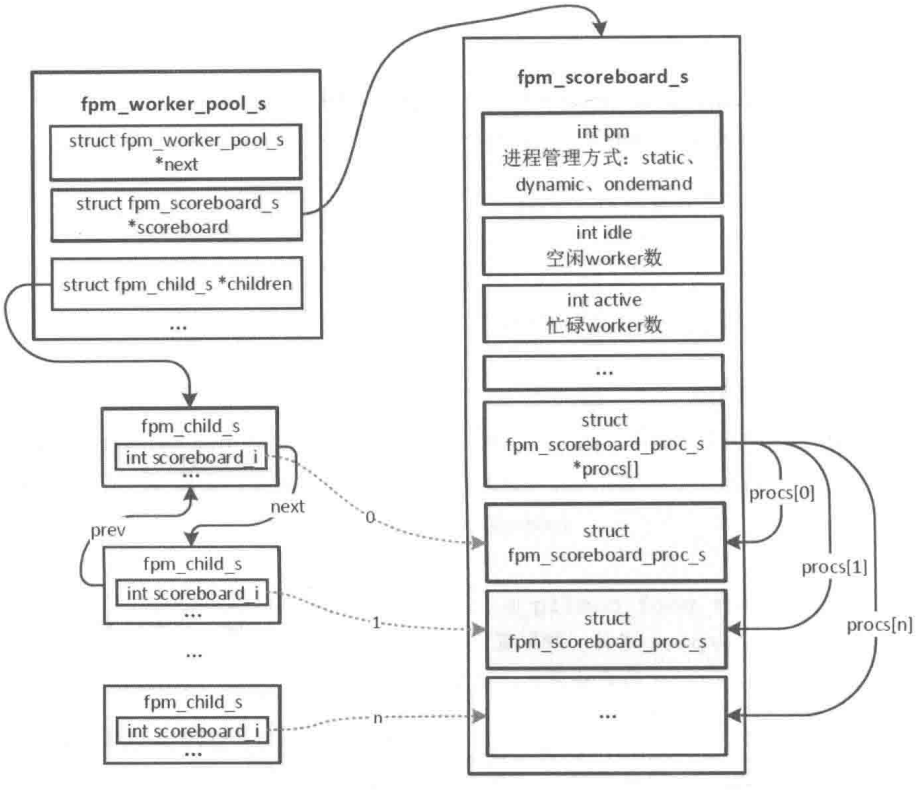


图 2-3　fpm_scoreboard_s 与 fpm_scoreboard_proc_s 结构体

3）fpm_signals_init_main()

这一步会通过 socketpair()创建一个管道，这个管道并不是用于 master 与 worker 进程通信的，它只在 master 进程中使用，具体用途在稍后介绍 event 事件处理时再作说明。同时，设置 master 的信号处理函数为 sig_handler()，当 master 收到 SIGTERM、SIGINT、SIGUSR1、SIGUSR2、SIGCHLD、SIGQUIT 这些信号时将调用 sig_handler()进行处理，在此函数中会把收到的信号写入在 fpm_signals_init_main()中创建的管道。

```
static int sp[2];
int fpm_signals_init_main()
{
    struct sigaction act;
    //创建一个全双工管道
    if (0 > socketpair(AF_UNIX, SOCK_STREAM, 0, sp)) {
        return -1;
    }
```

```
    //注册信号处理 handler
    act.sa_handler = sig_handler;
    sigfillset(&act.sa_mask);
    if (0 > sigaction(SIGTERM,  &act, 0) ||
        0 > sigaction(SIGINT,   &act, 0) ||
        0 > sigaction(SIGUSR1,  &act, 0) ||
        0 > sigaction(SIGUSR2,  &act, 0) ||
        0 > sigaction(SIGCHLD,  &act, 0) ||
        0 > sigaction(SIGQUIT,  &act, 0)) {
        return -1;
    }
    return 0;
}
static void sig_handler(int signo)
{
    static const char sig_chars[NSIG + 1] = {
        [SIGTERM] = 'T',
        [SIGINT]  = 'I',
        [SIGUSR1] = '1',
        [SIGUSR2] = '2',
        [SIGQUIT] = 'Q',
        [SIGCHLD] = 'C'
    };
    char s;
    ...
    s = sig_chars[signo];
    //将信号通知写入管道 sp[1]端
    write(sp[1], &s, sizeof(s));
    ...
}
```

4）fpm_sockets_init_main()

创建每个 worker pool 的 socket 套接字，启动后 worker 将监听此 socket 接收请求。

5）fpm_event_init_main()

启动 master 的事件管理，Fpm 实现了一个事件管理器用于管理 I/O、定时事件，其中 I/O 事件根据不同平台选择 kqueue、epoll、poll、select 等管理，定时事件就是定时器，一定时间后触发某个事件。

fpm_init()中主要的处理就是上面介绍的几个 init 过程，在完成这些初始化操作后接下来就是最关键的 fpm_run()操作了，此环节将 fork 子进程，启动进程管理器，执行后 master 进程将不会返回这个函数，只有各 worker 进程会返回，也就是说 main()函数中调用 fpm_run()之后的操作均是 worker 进程的。

```
int fpm_run(int *max_requests)
{
    struct fpm_worker_pool_s *wp;
    //编译 worker pool
    for (wp = fpm_worker_all_pools; wp; wp = wp->next) {
        //调用 fpm_children_make() fork 子进程
        is_parent = fpm_children_create_initial(wp);
        if (!is_parent) {
            //fork 出的 worker 进程
            goto run_child;
        }
    }
    //master 进程将进入 event 循环，不再往下走
    fpm_event_loop(0);
run_child: //只有 worker 进程会到这里
    *max_requests = fpm_globals.max_requests;
    return fpm_globals.listening_socket; //返回监听的套接字
}
```

由此 fpm_run()的处理来看，在 fork 出 worker 进程后，子进程将返回 main()中，而 master 进程进入 fpm_event_loop()，这是一个事件循环，接下来分别分析 master、worker 后续的处理。

2.2.3 worker——请求处理

worker 进程返回 main()函数后将继续向下执行，此后的流程就是 worker 进程不断 Accept 请求，有请求到达后将读取并解析 FastCGI 协议的数据，解析完成后开始执行 PHP 脚本，执行完成后关闭请求，继续监听等待新的请求到达。关于 FastCGI 协议的解析不是本书的重点，这里不再介绍。各 worker 处理请求的周期如图 2-4 所示。

（1）等待请求：worker 进程阻塞在 fcgi_accept_request()中等待请求。

（2）解析请求：fastcgi 请求到达后被 worker 接收，然后开始接收并解析请求数据，直到 request 数据完全到达。

（3）请求初始化：执行 php_request_startup()，此阶段会调用每个扩展的 PHP_RINIT_FUNCTION()。

（4）执行 PHP 脚本：由 php_execute_script()完成 PHP 脚本的编译、执行操作。

（5）关闭请求：请求完成后执行 php_request_shutdown()，此阶段会调用每个扩展的 PHP_RSHUTDOWN_FUNCTION()，然后进入步骤（1）等待下一个请求。

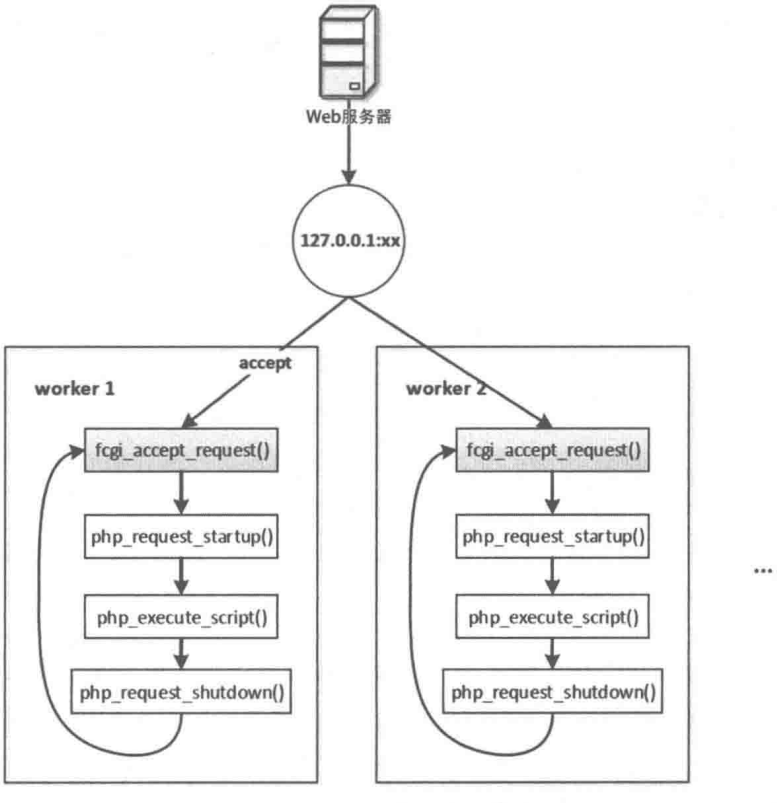


图 2-4 worker 进程的工作周期

具体的处理如下：

```
int main(int argc, char *argv[])
{
    ...
    //worker 在调用后返回监听的 socket fd
    fcgi_fd = fpm_run(&max_requests);
    parent = 0;
    //初始化 fastcgi 请求
```

```
    request = fpm_init_request(fcgi_fd);
    //worker 进程将阻塞在这，等待请求
    while (EXPECTED(fcgi_accept_request(request) >= 0)) {
        SG(server_context) = (void *) request;
        init_request_info();
        //请求开始
        if (UNEXPECTED(php_request_startup() == FAILURE)) {
            ...
        }
        ...
        fpm_request_executing();
        //编译、执行 PHP 脚本
        php_execute_script(&file_handle);
        ...
        //请求结束
        php_request_shutdown((void *) 0);
        ...
    }
    ...
    //worker 进程退出，进入 module shutdown 阶段
    php_module_shutdown();
    ...
}
```

worker 进程的 fpm_scoreboard_proc_s->request_stage 被用于记录 worker 当前所处的阶段，master 进程也是通过这个值来获取 worker 的状态，一次请求过程中这个值将先后被设置为以下值。

- FPM_REQUEST_ACCEPTING：等待请求阶段；
- FPM_REQUEST_READING_HEADERS：读取 fastcgi 请求 header 阶段；
- FPM_REQUEST_INFO：获取请求信息阶段，此阶段是将请求的 method、query stirng、request uri 等信息保存到各 worker 进程的 fpm_scoreboard_proc_s 结构中，此操作需要加锁，因为 master 进程也会操作此结构；
- FPM_REQUEST_EXECUTING：执行 PHP 脚本阶段；
- FPM_REQUEST_END：没有使用；
- FPM_REQUEST_FINISHED：请求处理完成。

通过 gdb 可以清楚地追踪到一个请求的完整处理流程，为了方便找到处理的 worker 进程，可以将 worker 数设置为 1，attach 接管 worker 进程后通过 bt 可以看到，worker 进程阻塞在 fcgi_accept_request()上。

```
$ gdb
(gdb) attach worker 进程 PID
(gdb) bt
#0  0x00000038f72e9650 in __accept_nocancel () from /lib64/libc.so.6
#1  0x0000000000909722 in fcgi_accept_request (req=0x18ea8c0) at
/home/qinpeng/php-7.0.12/main/fastcgi.c:1401
#2  0x0000000000917f46 in main
```

2.2.4 master——进程管理

master 在调用 fpm_run()后不再返回，而是进入一个事件循环中，此后 master 将始终围绕着几个事件进行处理，在具体分析这几个事件之前，首先介绍 Fpm 三种不同的进程管理方式，具体要使用哪种模式可以在 conf 配置中通过 pm 指定，例如：pm = dynamic。

- **静态模式(static)**：这种方式比较简单，在启动时 master 根据 pm.max_children 配置 fork 出相应数量的 worker 进程，也就是 worker 进程数是固定不变的。
- **动态模式（dynamic）**：这种模式比较常用，在 Fpm 启动时会根据 pm.start_servers 配置初始化一定数量的 worker。运行期间如果 master 发现空闲 worker 数低于 pm.min_spare_servers 配置数(表示请求比较多，worker 处理不过来了)则会 fork worker 进程，但总的 worker 数不能超过 pm.max_children；如果 master 发现空闲 worker 数超过了 pm.max_spare_servers（表示闲着的 worker 太多了）则会杀掉一些 worker，避免占用过多资源，master 通过这 4 个值来动态控制 worker 的数量。
- **按需模式（ondemand）**：这种模式很像传统的 cgi，在启动时不分配 worker 进程，等到有请求了后再通知 master 进程 fork worker 进程，也就是来了请求以后再 fork 子进程进行处理。总的 worker 数不超过 pm.max_children，处理完成后 worker 进程不会立即退出，当空闲时间超过 pm.process_idle_timeout 后再退出。

master 进程进入 fpm_event_loop()事件循环，在这个方法中 master 将循环处理 master 注册的几个 I/O 及定时器事件，当有事件触发时将回调具体的 handler 进行处理。

```
void fpm_event_loop(int err)
{
```

```
    //注册 I/O、定时器事件，稍后再作说明
    ...
    //进入事件循环，master 进程将阻塞在此
    while (1) {
        ...
        //等待 I/O 事件
        ret = module->wait(fpm_event_queue_fd, timeout);
        ...
        //检查定时器事件
        ...
    }
}
```

接下来具体看一下 master 注册的几个重要事件。

1）信号事件

前面已经介绍过，fpm_init()阶段时分配了一个 sp 管道，当 master 收到 SIGTERM、SIGINT、SIGUSR1、SIGUSR2、SIGCHLD、SIGQUIT 这些信号时会把对应的信号写到 sp[1]管道中，那么谁来接收这个信号呢？master 在 fpm_event_loop()中注册了此管道可读的事件。

```
static struct fpm_event_s signal_fd_event;
//设置监听的 fd、事件触发类型及回调函数
fpm_event_set(&signal_fd_event, fpm_signals_get_fd(), FPM_EV_READ, &fpm_
got_signal, NULL);
//注册事件
fpm_event_add(&signal_fd_event, 0);
```

fpm_signals_get_fd()返回的是 sp[0]，所以这里就是注册了一个 sp[0]可读的事件，当 sp[0]可读时将回调 fpm_got_signal()进行处理。总体来看，当向 master 发送信号时，首先信号 handler 会把信号通知发送到 sp[1]管道，然后触发 sp[0]可读事件，回调 fpm_got_signal()进行处理，数据流如图 2-5 所示。

fpm_got_signal()根据不同的信号进行相应的处理，不同信号值对应的处理逻辑。

- **SIGINT/SIGTERM/SIGQUIT：**退出 Fpm，在 master 收到退出信号后将向所有的 worker 进程发送退出信号，通知 worker 退出，然后 master 退出。
- **SIGUSR1：**重新加载日志文件，生产环境中通常会根据时间对日志进行切割，切割后会生成一个新的日志文件，如果进程不重新加载文件，则无法继续写入日志，这时就

需要向 master 发送一个 USR1 的信号，告诉 master 重新加载日志文件。

- **SIGUSR2**：重启 Fpm，首先 master 也是会向所有的 worker 进程发送退出信号，等全部 worker 成功退出后，master 会调用 execvp()重新启动一个新的 Fpm，最后旧的 master 退出。
- **SIGCHLD**：这个信号是子进程退出时操作系统发送给父进程的，子进程退出时，操作系统将子进程置为僵尸状态，这个进程称为僵尸进程，它只保留最小的一些内核数据结构，以便父进程查询子进程的退出状态，只有当父进程调用 wait()或者 waitpid()函数查询子进程退出状态后子进程才终止，Fpm 中当 worker 进程因为异常原因（比如 coredump 了）退出而非 master 主动杀掉时，master 将收到此信号，这时父进程将调用 waitpid()保证 worker 退出，然后检查是不是需要重新 fork 新的 worker。

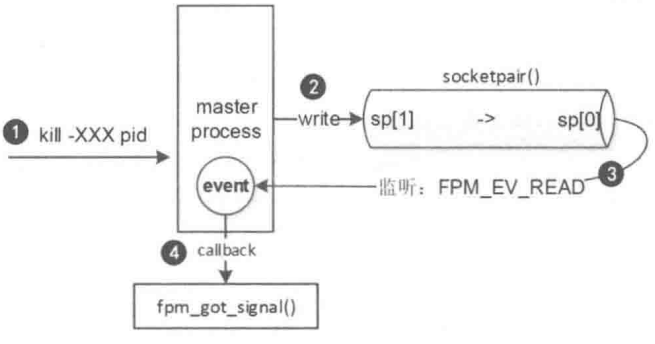


图 2-5 master 信号事件处理

可以通过 gdb 进行调试，通过 attach 命令接管 master 进程，然后向 master 发送不同的信号。

```
$ gdb
(gdb) break fpm_got_signal
(gdb) attach master 进程 PID
(gdb) c
//另起一个终端
$ kill -XX master 进程 PID
```

2）进程检查定时器

fpm_event_loop()中调用 fpm_pctl_perform_idle_server_maintenance_heartbeat()注册了一个定时器：

```
//fpm_event_loop:
if (!err) { //err 传的是 0: fpm_event_loop(0)
```

```
        fpm_pctl_perform_idle_server_maintenance_heartbeat(NULL, 0, NULL);
        //log
    }
```

这个定时器是用来定期检查 worker 进程数的，master 通过这个定时器每隔一定时间检查 worker 的数量，根据不同策略（static/dynamic/ondemand）的配置决定是否需要 fork 或者 kill 进程。

```
    void fpm_pctl_perform_idle_server_maintenance_heartbeat(struct fpm_event_s
*ev, short which, void *arg)
    {
        static struct fpm_event_s heartbeat;
        struct timeval now;
        //定时器触发时回调的逻辑
        ...
        //此后的逻辑只在注册时执行一次，以后不会再到达这里
        //设置定时器，回调函数也是本函数
        fpm_event_set_timer(&heartbeat, FPM_EV_PERSIST, &fpm_pctl_perform_
idle_server_maintenance_heartbeat, NULL);
        //注册定时器，第二个参数是每隔多少 ms 触发
        fpm_event_add(&heartbeat, FPM_IDLE_SERVER_MAINTENANCE_HEARTBEAT);
    }
```

从代码中可以看出，这里注册了一个定时器，其回调函数是 fpm_pctl_perform_idle_server_maintenance_heartbeat()，也就是自己，触发间隔时间为 FPM_IDLE_SERVER_MAINTENANCE_HEARTBEAT（即 1000ms）。定时器触发时的逻辑如下所示。

```
    //fpm_pctl_perform_idle_server_maintenance_heartbeat:
    if (which == FPM_EV_TIMEOUT) { //确认是一个定时事件
        fpm_clock_get(&now);
        //fpm_pctl_can_spawn_children()检查 master 当前状态是否正常，比如 master 收
到了退出信号，这个时候就不需要再进行 fork、kill 操作了
        if (fpm_pctl_can_spawn_children()) {
            //具体的 worker 检查逻辑
            fpm_pctl_perform_idle_server_maintenance(&now);
            ...
        }
        return;
    }
```

static静态模式不会动态地管理worker进程，所以fpm_pctl_perform_idle_server_maintenance()主要是针对dynamic、ondemand两种模式的。检查的过程就是遍历所有的worker pool，根据不同的策略配置进行不同的处理，具体的worker控制策略在一开始已经介绍过了，这里就是具体实现的位置。

检查每个worker pool时，首先根据此pool下所有worker的状态计算出处于空闲、忙碌状态的worker数，这个就是根据每个worker进程 fpm_scoreboard_proc_s->request_stage 状态判断的。

```
//fpm_pctl_perform_idle_server_maintenance:
struct fpm_child_s *last_idle_child = NULL; //空闲时间最久的worker
int idle = 0; //空闲worker数
int active = 0; //忙碌worker数

for (child = wp->children; child; child = child->next) {
    //根据worker进程的fpm_scoreboard_proc_s->request_stage判断
    if (fpm_request_is_idle(child)) {
        //找空闲时间最久的worker
         ...
        idle++;
    }else{
        active++;
    }
}
```

接着根据不同的pm模式分别处理，如果是按需模式（ondemand），则会判断空闲时间最久的那个worker的空闲时间是否达到 pm.process_idle_timeout 阈值，到了则kill掉该worker。

```
//fpm_pctl_perform_idle_server_maintenance:
if (wp->config->pm == PM_STYLE_ONDEMAND) {
    if (!last_idle_child) continue;
    ...
    if (last.tv_sec < now.tv_sec - wp->config->pm_process_idle_timeout) {
    //如果空闲时间最长的worker空闲时间超过了process_idle_timeout则杀掉该worker
        last_idle_child->idle_kill = 1;
        fpm_pctl_kill(last_idle_child->pid, FPM_PCTL_QUIT);
    }
```

```
        continue;
    }
```

从处理过程可以看到，ondemand 模式下每次只 kill 空闲时间最久的那个 worker，如果有多个 worker 都达到了 pm.process_idle_timeout 的阈值，则需要等到下一个周期进行处理。

如果是动态模式（dynamic）模式，首先检查空闲 worker 数是否超过了 pm.max_spare_servers 配置的数量，若超了则杀掉空闲时间最久的那个，同样，这里每个周期只会 kill 一个 worker；接着检查空闲 worker 数是否低于 pm.min_spare_servers，如果低于则需要 fork 更多的 worker 进行补充，但是总的 worker 数不能超过 pm.max_children。在 fork 的过程中会通过 idle_spawn_rate 这个值控制频率，默认一个周期内只会 fork 一个进程，但是如果发现几个周期一直在 fork，则说明 worker 数远远不足，这时就会把 fork 数翻一倍，上限是 32。

```
    //fpm_pctl_perform_idle_server_maintenance:
    if (wp->config->pm != PM_STYLE_DYNAMIC) continue;
    if (idle > wp->config->pm_max_spare_servers && last_idle_child) {
        //空闲 worker 太多了，杀掉
        last_idle_child->idle_kill = 1;
        fpm_pctl_kill(last_idle_child->pid, FPM_PCTL_QUIT);
        wp->idle_spawn_rate = 1;
        continue;
    }
    if (idle < wp->config->pm_min_spare_servers) {
        //空闲 worker 太少了，如果总 worker 数未达到 max 数则 fork
        if (wp->running_children >= wp->config->pm_max_children) {
            //worker 总数已达上限
            continue;
        }
        //确定要 fork 多少个 worker
        children_to_fork = MIN(wp->idle_spawn_rate, wp->config->pm_min_spare_servers
- idle);
        //确保不超过 worker 上限
        children_to_fork = MIN(children_to_fork, wp->config->pm_max_children -
wp->running_children);
        //fork
        fpm_children_make(wp, 1, children_to_fork, 1);
        //将 idle_spawn_rate 翻一倍，这样下个周期内一次 fork 数就会翻倍，如果下个周期内空
闲 worker 数达到 min_spare_servers 则会把这个值重置为 1
```

```
        if (wp->idle_spawn_rate < FPM_MAX_SPAWN_RATE) {
            wp->idle_spawn_rate *= 2;
        }
        continue;
    }
    wp->idle_spawn_rate = 1;
```

3）执行超时检查定时器

php-fpm.conf 中有一个 request_terminate_timeout 的配置项，如果 worker 处理一个请求的总时长超过了这个值，那么 master 会向此 worker 进程发送 kill -TERM 信号杀掉 worker 进程，此配置单位为秒，默认值为 0，表示关闭此机制。

这个功能也是通过定时器实现的，master 每隔一定时间检查所有处理中的 worker，如果发现其处理时间达到阈值则杀掉这个 worker。另外，Fpm 记录的 slow log 也是通过这个定时器完成的。

```
    //fpm_event_loop:
    if (fpm_globals.heartbeat > 0) {
        fpm_pctl_heartbeat(NULL, 0, NULL);
    }
```

fpm_pctl_heartbeat()中定义了一个定时器，每隔 fpm_globals.heartbeat 毫秒触发一次，这个时间是根据 pm.request_terminate_timeout 计算得到的，回调函数也是自己。

```
    //fpm_conf_process_all_pools:
    //设定触发周期
    fpm_globals.heartbeat = fpm_globals.heartbeat ? MIN(fpm_globals.heartbeat,
(wp->config->request_terminate_timeout * 1000) / 3) : (wp->config->request_
terminate_timeout * 1000) / 3;
    //注册定时器（定时器回调 handler）
    void fpm_pctl_heartbeat(struct fpm_event_s *ev, short which, void *arg)
    {
        static struct fpm_event_s heartbeat;
        struct timeval now;

        if (which == FPM_EV_TIMEOUT) {
            //这个函数也是回调函数，回调时将走到这个分支
            fpm_clock_get(&now);
```

```
            fpm_pctl_check_request_timeout(&now);
            return;
        }
        //下面的逻辑只在注册时执行一次
        //注册定时器，回调函数是自己，每隔 fpm_globals.heartbeat 触发一次
        fpm_event_set_timer(&heartbeat, FPM_EV_PERSIST, &fpm_pctl_heartbeat, NULL);
        fpm_event_add(&heartbeat, fpm_globals.heartbeat);
    }
```

定时器触发时将调用 fpm_pctl_check_request_timeout()进行处理，处理逻辑也比较简单，遍历 worker pool，然后逐个判断 worker 处理中的请求是否超时，此判断根据请求处理开始时间 fpm_scoreboard_proc_s->accepted 完成，如果超时则 kill。

```
    //fpm_pctl_check_request_timeout:
    //每个 worker pool
    int terminate_timeout = wp->config->request_terminate_timeout;
    int slowlog_timeout = wp->config->request_slowlog_timeout;
    struct fpm_child_s *child;

    if (terminate_timeout || slowlog_timeout) {
        //遍历当前 worker pool 下的所有 worker
        for (child = wp->children; child; child = child->next) {
            //检查当前 worker 处理的请求是否超时
            fpm_request_check_timed_out(child, now, terminate_timeout, slowlog_
timeout);
        }
    }
```

以上就是 master 进程主要的处理，除了介绍的这些事件，还有一个 I/O 事件没有提到，这个事件只在 ondemand 模式下使用。因为 ondemand 模式下 Fpm 启动时是不会预创建 worker 的，有请求时才会生成子进程，所以有请求到达时需要通知 master 进程进行 fork。这个事件是在 fpm_children_create_initial()时注册的，事件处理函数为 fpm_pctl_on_socket_accept()，具体逻辑这里不再展开，比较好理解。

2.3 Embed

前面介绍的 Cli、Fpm 都是完整的应用程序，它们有定义自己的 main 函数，其应用场景是

固定的，另外两个没有介绍的 SAPI（litespeed、apache2handler）也是配合其他应用使用的。如果我们在自己的第三方程序中也想使用 PHP，则该怎么办呢？比如开发一个路由器，其中想嵌入 PHP 来实现 Web 配置的功能，难道要重新开发一个 SAPI，然后提供给第三方应用使用？PHP 提供了一个用于这类应用场景下的 SAPI，那就是 Embed，它在编译后就是普通的库文件（可以选择编译为静态库、共享库），我们可以在其他 C/C++应用中调用 PHP 提供的 API，甚至可以提供给其他语言使用。

编译 PHP 时通过 --enable-embed=[shared|static] 指定库类型，默认是共享库。编译完成后可以在 PHP 安装位置的/lib 目录下看到生成的库文件，同时在/include/php/sapi 目录下会生成一个存放 Embed 头文件的目录。

2.3.1 实现

首先我们来看一下 Embed 的实现，实际它的逻辑非常简单，只是把 PHP 生命周期的几个处理函数进行了封装，它对外提供了两个 API。

1）php_embed_init()

这个接口主要进行 PHP 框架的初始化操作，比如启动 TSRM、初始化 SAPI、初始化信号处理，另外它还完成了非常重要的两个操作，那就是 php_module_startup()、php_request_startup()。

```
EMBED_SAPI_API int php_embed_init(int argc, char **argv)
{
#ifdef ZTS
    //启动 SAPI
    tsrm_startup(1, 1, 0, NULL);
    (void)ts_resource(0);
    ZEND_TSRMLS_CACHE_UPDATE();
#endif
    //初始化 SAPI
    sapi_startup(&php_embed_module);
    ...
    //进入 module starup 阶段
    if (php_embed_module.startup(&php_embed_module)==FAILURE) {
        return FAILURE;
    }
    ...
    //请求 request startup 阶段
```

```
    if (php_request_startup()==FAILURE) {
        return FAILURE;
    }
    ...
}
```

在第三方应用中嵌入 PHP 时首先需要调用这个接口，然后就可以使用 PHP/Zend 提供的 API 完成 PHP 脚本的执行了。

2）php_embed_shutdown()

此接口与 php_embed_init()对应，主要完成 PHP 框架的关闭收尾工作，包括 request shutdown、module shutdown 两个阶段的操作。

```
EMBED_SAPI_API void php_embed_shutdown(void)
{
    //关闭请求，request shutdown 阶段
    php_request_shutdown((void *) 0);
    //关闭模块，module shutdown 阶段
    php_module_shutdown();
    sapi_shutdown();
#ifdef ZTS
    tsrm_shutdown();
#endif
    ...
}
```

2.3.2 使用

下面我们举个例子具体看一下如何在一个 C 程序中嵌入 PHP，使用的是共享库，这个例子中我们自己定义的 C 程序将调用 PHP 完成一个普通 PHP 脚本的执行。

```
//my_main.c
#include <php/sapi/embed/php_embed.h>

int main(int argc, char **argv)
{
    zend_file_handle file_handle;
```

```
    //PHP 框架初始化
    php_embed_init(argc, argv);
    file_handle.type = ZEND_HANDLE_FILENAME;
    file_handle.filename = "call.php";
    //execute php script
    php_execute_script(&file_handle);
    //关闭 PHP 框架
    php_embed_shutdown();
    return 0;
}
//call.php
function my_func($a, $b){
    return $a + $b;
}
echo my_func(100,200);
```

定义完成后编译，编译时需要通过-I 指定 PHP 的头文件目录，包括 PHP、Zend、SAPI、TSRM 的；通过-l 及-L 指定共享库的目录及名称。另外，还需要通过-rpath 指定运行时共享库的位置。假如 PHP 安装路径为/usr/local/php7，则编译参数为：

```
gcc -o my_php test.c \
    -I/usr/local/php7/include \
    -I/usr/local/php7/include/php \
    -I/usr/local/php7/include/php/include \
    -I/usr/local/php7/include/php/main \
    -I/usr/local/php7/include/php/TSRM \
    -I/usr/local/php7/include/php/Zend \
    -L/usr/local/php7/lib \
    -lphp7 \
    -Wl,--rpath /usr/local/php7/lib
```

编译后执行：

```
$ ./my_php
300
```

> gcc 编译参数-l、-L 只是在**编译时**指定了库文件，与执行时无关，在**执行时**默认会去/usr/lib 目录下搜索.so，另外可以通过环境变量 LD_LIBRARY_PATH 指定搜索目录，也可以将搜索目录添加到/etc/ld.so.conf 配置中。
>
> 除了上面这些方式，还可以在编译时通过-rpath 指定，其语法需要遵循“-Wl,...”，其中-Wl 就是告诉 gcc，后面的内容是传递给 linker 的 option，比如-Wl，--rpath /usr/local/php7/lib。它的优先级要在 LD_LIBRARY_PATH、/etc/ld.so.conf 之上。

2.4 小结

本章我们主要介绍了 PHP 中三种典型 SAPI 的实现：Cli、Fpm、Embed。其中 Cli 是命令行下执行 PHP 脚本的实现；Fpm 是 Web 环境下使用 PHP 的实现，企业级应用中主要配合 Nginx 使用，Cli、Fpm 是使用最为广泛的两个 SAPI。另外我们介绍了如何在第三方应用中嵌入 PHP，这种方式使用得比较少，却是最为灵活的，大大扩展了 PHP 的应用范围，通过 Embed 我们可以在更多的应用场景下使用 PHP。

总的来说，SAPI 的实现并不复杂，它是我们进入 PHP 内核的第一道门。

第 3 章 数据类型

数据类型是一个语言最为基础的部分，它是高级语言抽象出来的一个概念，对于低级的语言来说是没有这个概念的，比如机器语言。对内存、CPU 而言并没有什么类型之分，内存中的数据是没有差别的，高级语言为内存中这些无差异的数据指定了特定的计算方式，比如读取一个 32 位整型，就是告诉 CPU 按照整型的规则连续读取 4 个字节的数据。

数据类型的产生使得程序的编写更加规范、简洁、灵活，它是现代高级语言必不可少的一部分。与强类型语言不同，PHP 中变量的数据类型并不是固定不变的，它可以根据不同的使用场景进行转化，这一章主要介绍 PHP 中数据类型的底层实现。

3.1 变量

变量是最常见的数据类型应用形式，它由三个主要部分组成：变量名、变量值、变量类型，PHP 中变量名与变量值可以简单地对应为：zval、zend_value，这两个概念一定要区分开。PHP 中变量的内存是通过引用计数进行管理的，而且 PHP7 中引用计数转移到了具体的 value 结构中而不再是 zval 中，这是与 PHP 旧版本不同的一个地方，变量之间的传递、赋值通常也针对 zend_value。

PHP 中通过$符号定义一个变量，在定义的同时可以进行初始化，在变量使用前不需要提前声明。事实上普通变量定义的方式包含了两步：变量定义、变量初始化，只定义而不初始化变量也是可以的，比如：

```
$a;
$b = 1;
```

这段代码在执行时会分配两个 zval，也就是这里定义了两个变量，只不过$a 没有值而已，相当于 unset()了。

3.1.1 变量类型

PHP 中的变量类型，也就是数据类型，宏观角度看可分为以下 8 种。

- 标量类型：字符串、整型、浮点型、布尔型。
- 复合类型：数组、对象。
- 特殊类型：资源、NULL。

具体到内部实现上会细分出更多的类型，比如布尔型在内部实际分为 IS_TRUE、IS_FALSE 两种，也有一些基于基础数据类型产生的特殊类型，比如引用。全部类型如下：

```
//file: zend_tyeps.h
/* regular data types */
#define IS_UNDEF                    0
#define IS_NULL                     1
#define IS_FALSE                    2
#define IS_TRUE                     3
#define IS_LONG                     4
#define IS_DOUBLE                   5
#define IS_STRING                   6
#define IS_ARRAY                    7
#define IS_OBJECT                   8
#define IS_RESOURCE                 9
#define IS_REFERENCE                10
/* constant expressions */
#define IS_CONSTANT                 11
#define IS_CONSTANT_AST             12
/* fake types */
#define _IS_BOOL                    13
#define IS_CALLABLE                 14
/* internal types */
```

```
#define IS_INDIRECT                 15
#define IS_PTR                      17
```

3.1.2 内部实现

这一节我们来具体看一下 PHP 中的数据类型在内核中是如何实现的。PHP 中通过 zval 这个结构体来表示一个变量，而不同类型的变量值则通过 zval 嵌入的一个联合体表示，即 zend_value，对 PHP 有所了解的读者对这两个结构体一定不会陌生。通过 zval、zend_value 及不同类型的结构实现了 PHP 基础的数据类型。另外，zval 并不是只有 PHP 变量会使用，它也是内核中的一个通用结构，用于统一函数的参数，很多类型是供内核自己使用的，这里可以简单地了解下，后续碰到的时候再具体看。

```
//file: zend_tyeps.h
typedef struct _zval_struct     zval;
struct _zval_struct {
    //变量值
    zend_value        value;
    union {
        struct {
            //下面这个宏是为了兼容大小字节序，小字节序就是下面的顺序，大字节序则是下面
4 个顺序翻转，忽略即可
            ZEND_ENDIAN_LOHI_4(
                //变量类型
                zend_uchar    type,
                //类型掩码，各类型会有不同的几种属性，内存管理会用到，下一章会介绍
                zend_uchar    type_flags,
                zend_uchar    const_flags,
                 //预留字段，zend 执行过程中会用来记录 call info
                zend_uchar    reserved)
        } v;
        uint32_t type_info;
    } u1;
    union {
        uint32_t     var_flags;
        uint32_t     next;
        uint32_t     cache_slot;
        uint32_t     lineno;
```

```
        uint32_t     num_args;
        uint32_t     fe_pos;
        uint32_t     fe_iter_idx;
    } u2; //一些辅助值
};
```

zval 除了嵌入了一个 zend_value 用来保存具体的变量值，还有两个特殊的 union。

- **u1**：这个结构看起来比较复杂，实际它只是联合了一个结构体 v 和一个 32 位无符号整型 type_info，ZEND_ENDIAN_LOHI_4 这个宏是用来解决字节序问题的，它会根据系统的字节序决定 struct v 中 4 个成员的顺序，忽略即可。v 中定义了 4 个成员，其中 type 用于标识 value 类型，即上一小节列举的 IS_XXX 的类型，type_flags 是类型掩码，用于变量的内存管理，下一章会详细说明，剩下两个后面用到的时候再作说明，这里可以暂时忽略。type_info 实际上是将 v 结构的 4 个成员组合到了一起，v 中的成员各占一个字节，总共 4 个字节，type_info 也是 4 个字节，每个字节对应 v 的一个成员，可以直接通过 type_info 位移获取 v 成员的值。
- **u2**：这个结构纯粹用于一些辅助功能，zval 结构的 value、u1 占用的空间分别为 8byte、4byte，但是加起来却不是 12byte，系统会进行字节对齐，value、u1 将占用 16byte，多的 4byte 将浪费，所以 zval 定义了一个 u2 结构把这 4byte 利用了，最终 zval 结构的大小就是 16byte。这个结构在一些特殊的场景下会使用，比如：next 在散列表解决哈希冲突时会用到，fe_pos 在 foreach 遍历时会用到，cache_slot 在运行时缓存中会用到。

zend_value 的结构比较简单，它是一个联合体，各类型根据自己的类型选择使用不同的成员，其中整型、浮点型的值直接存储在 zend_value 中，其他类型则是指针，指向具体类型的结构。另外，你可能已经注意到 zend_value 中并没有布尔型，这是因为 PHP7 中将布尔型具体拆分为了 true、false 两种类型，它们直接通过 type 类型区分，因此不需要具体的 value，而在 PHP 旧版本中布尔型也是通过整型进行区分的。

```
typedef union _zend_value {
    zend_long        lval;        //整型
    double           dval;        //浮点型
    zend_refcounted  *counted;    //获取不同类型结构的 gc 头部
    zend_string      *str;        //string 字符串
    zend_array       *arr;        //array 数组
    zend_object      *obj;        //object 对象
    zend_resource    *res;        //resource 资源类型
    zend_reference   *ref;        //引用类型，通过&$var_name 定义的
```

```
    zend_ast_ref    *ast;        //下面几个都是内核使用的value
    zval            *zv;         //指向另一个zval
    void            *ptr;        //指针，通用类型
    zend_class_entry *ce;        //类
    zend_function   *func;       //函数
    struct {
        uint32_t w1;
        uint32_t w2;
    } ww;
} zend_value;
```

zend_value 中定义了众多类型的指针，这些类型并不全是变量的类型，有些类型只给内核自己使用，比如 ast、ptr、zv 等。

3.2 字符串

PHP 中并没有使用 char 来表示字符串，而是为字符串单独定义了一个结构：zend_string。在 zend_value 中通过 str 指向具体的结构，zend_string 除了字符串内容，还存储了其他几个信息，具体结构如下：

```
typedef struct _zend_string zend_string;
struct _zend_string {
    zend_refcounted_h gc;
    zend_ulong        h;                /* hash value */
    size_t            len;
    char              val[1];
};
```

该结构有 4 个成员。

- **gc**：变量的引用计数信息，用于内存管理。
- **h**：字符串通过 Times 33 算法计算得到的 Hash Code。
- **len**：字符串长度。
- **val**：字符串内容。

zend_string 的几个成员含义都比较直观，比较特殊的是用于存储字符串内容的成员，这里并没有使用 char *类型，而是使用了一个可变数组 val。val[1]并不表示它只能存储一个字节，在

字符串分配时实际上是类似这样操作的：malloc(sizeof(zend_string) + 字符串长度)，也就是会多分配一些内存，而多出的这块内存的起始位置就是 val，这样就可以直接将字符串内容存储到 val 中，通过 val 进行读取。如果 val 是一个指针 char *，则需要额外分配一次内存，变长结构体不仅可以省一次内存分配，而且更有助于内存管理，free 时直接释放 zend_string 即可。

另外一个需要注意的地方是，val 中多出来的一个字节（结构体中为 val[1]而不是 val[0]），用于存储字符串的最后一个字符："\0"，比如$a = "abc"，则对应的 zend_string 内存结构如图 3-1 所示。

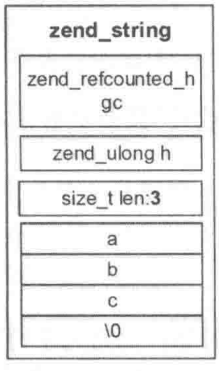


图 3-1 zend_string 内存结构

3.3 数组

数组是 PHP 中非常强大、灵活的一种数据类型，它的底层实现为散列表（HashTable，也称作哈希表），除了我们熟悉的 PHP 用户空间中的 array 类型，内核中也大量使用到这个数据结构，比如内核中用于存储函数、类、常量等的符号表。

散列表是根据关键码值（Key value）而直接进行访问的数据结构，它的 key-value 之间存在一个映射函数，可以根据 key 通过映射函数直接索引到对应的 value 值，它不以关键字的比较为基本操作，直接根据“内存起始位置+偏移值”进行寻址，也就是说，它是直接通过 key 映射到内存地址上去的，从而加快了查找速度。在理想情况下，无须任何比较就可以找到待查关键字，查找的期望时间复杂度为 O(1)。下面看一下散列表的结构：

```
//zend_array、HashTable 的含义是相同的，没有任何区别
typedef struct _zend_array      zend_array;
typedef struct _zend_array HashTable;

struct _zend_array {
    zend_refcounted_h gc;
```

```
    //这个 union 可以先忽略
    union {
        ...
    } u;
    //用于散列函数映射存储元素在 arData 数组中的下标
    uint32_t        nTableMask;
    //存储元素数组，每个元素的结构统一为 Bucket，arData 指向第一个 Bucket
    Bucket          *arData;
    //已用 Bucket 数
    uint32_t        nNumUsed;
    //数组实际存储的元素数
    uint32_t        nNumOfElements;
    //数组的总容量
    uint32_t        nTableSize;
    uint32_t        nInternalPointer;
    //下一个可用的数值索引，如 arr[] = 1;arr["a"] = 2;arr[] = 3;
则 nNextFreeElement = 2;
    zend_long       nNextFreeElement;
    dtor_func_t     pDestructor;
};
```

散列表的结构中有很多成员，比较重要的几个成员的含义如下所述。

- **arData**：散列表中保存存储元素（即 Bucket）的数组，其内存是连续的，arData 指向数组的起始位置。
- **nTableSize**：数组的总容量，即可以容纳的元素数，arData 的内存大小就是根据这个值确定的，它的大小是 2 的幂次方，最小为 8，也就是说，散列表的大小依次按 8、16、32、64……递增。
- **nTableMask**：这个值在散列函数根据 key 的 hash code 映射元素的存储位置时用到，它的值实际就是 nTableSize 的负数，即 nTableMask = -nTableSize，用位运算表示的话则为 nTableMask = ~nTableSize+1。
- **nNumUsed、nNumOfElements**：这两个成员的含义看起来非常相似，但是它们代表的含义是不同的，nNumUsed 是指数组当前使用的 Bucket 数，但是这些 Bucket 并不都是数组有效的元素，因为当我们删除一个数组元素时并不会马上将其从数组中移除，而只是将这个元素的类型标为IS_UNDEF，只有在数组容量超限，需要进行扩容时才会删除；nNumOfElements 则是数组中有效元素的数量，所以 nNumOfElements ≤ nNumUsed。如果数组没有扩容，那么 nNumUsed 将一直是递增的，无论是否删除元素。

- **nNextFreeElement**：这个是给自动确定数值索引使用的，从 0 开始，比如`$a[] = 1`，这时候这个值就增加为 1，下次再有`$a[]`的操作时就使用 1 作为新元素的索引值。
- **pDestructor**：当删除或覆盖数组中的某个元素时，如果提供了这个函数句柄，则在删除或覆盖后调用此函数，对旧元素进行清理。
- **u**：这个结构主要用于一些辅助作用，比如 flags 用来设置散列表的一些属性——是否持久化、是否已经初始化等。

Bucket 的结构比较简单，此结构主要用来保存元素的 key 及 value。除了这两个还有一个整型的 h，它的含义是 hash code：如果元素是数值索引，那么它的值就是数值索引的值；如果是字符串，那么这个值就是根据字符串 key 通过 Time 33 算法计算得到的散列值，h 的值用来映射元素的存储位置。另外，存储的 value 也直接嵌入到了 Bucket 结构中。

```
typedef struct _Bucket {
    zval             val; //存储的具体 value，这里嵌入了一个 zval，而不是一个指针
    zend_ulong       h;   //key 根据 times 33 计算得到的哈希值，或者是数值索引
    zend_string      *key; //存储元素的 key
} Bucket;
```

3.3.1 基本实现

散列表主要由两部分组成：存储元素数组、散列函数。一个简单的散列函数可以采用取模的方式，比如散列表大小为 8，那么在散列表初始化数组时就分配 8 个元素大小的空间，根据 key 的 hash code 与 8 取模得到的值作为该元素在数组中的下标，这样就可以通过 key 映射到存储数组中的具体位置，如图 3-2 所示。

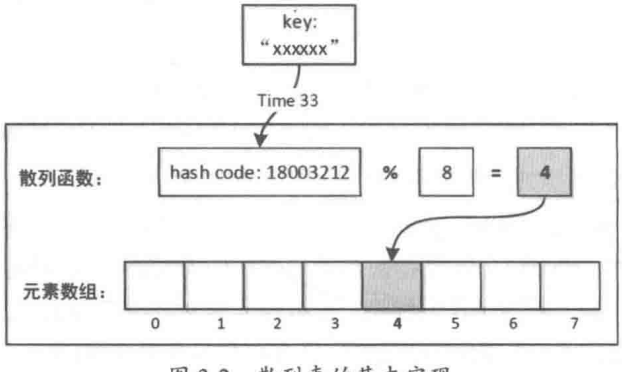


图 3-2 散列表的基本实现

这就是一个散列表的基本实现，但是这种直接以散列函数的输出值作为该元素在存储元素

数组中的下标的方式有一个问题：元素在数组中的位置具有随机性，它是无序的。PHP 中的数组除了散列表具备的特点，还有一个特别的地方，那就是它是有序的，数组中各元素的顺序与其插入顺序一致，那么这个特性是怎么实现的呢？

为了实现散列表的有序性，PHP 中的散列表在散列函数与元素数组之间加了一层映射表，这个映射表也是一个数组，大小与存储元素的数组相同，它存储的元素类型为整型，用于保存元素在实际存储的有序数组中的下标：元素按照先后顺序依次插入实际存储数组，然后将其数组下标按照散列函数散列出来的位置存储在新加的映射表中，如图 3-3 所示。

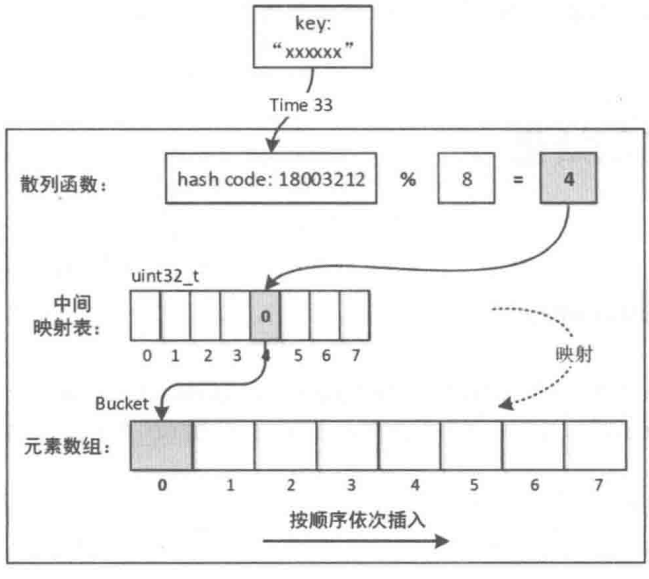


图 3-3　PHP 散列表映射关系

实现原理我们介绍完了，但是在 PHP 数组的结构中并没有发现中间这个映射表，事实上它与 arData 放在一起，在数组初始化时并不仅仅分配用于存储 Bucket 的内存，还会分配相同数量的 uint32_t 大小的空间，这两块空间是一起分配的，然后将 arData 偏移到存储元素数组的位置，而这个中间映射表可以通过 arData 向前访问到，如图 3-4 所示。

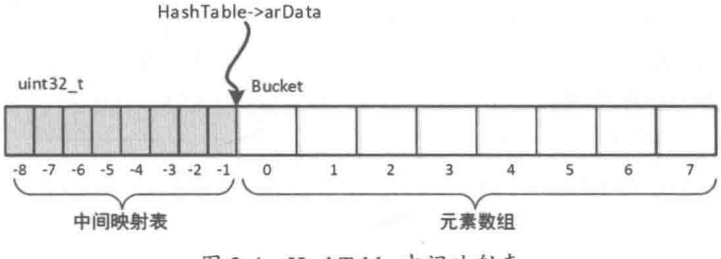


图 3-4　HashTable 中间映射表

3.3.2 散列函数

散列函数的作用是根据 key 映射出元素的存储位置，通常会以取模作为散列函数：key->h % nTableSize。但是 PHP 中使用了另外一种方式，前面曾介绍数组结构中有一个 nTableMask，它的值是 nTableSize 的负数，PHP 中采用如下的方式计算散列值：

```
nIndex = key->h | nTableSize;
```

因为散列表的大小为 2 的幂次方，所以通过与运算可以得到 [-1, nTableMask] 之间的散列值。

3.3.3 数组的初始化

数组初始化的过程主要是对 HashTable 中的成员进行设置，初始化时并不会立即分配 arData 的内存，arData 的内存在插入第 1 个元素时才会分配。初始化操作可以通过 zend_hash_init()宏完成，最后由_zend_hash_init()函数处理。

```
ZEND_API void ZEND_FASTCALL _zend_hash_init(HashTable *ht, uint32_t nSize,
dtor_func_t pDestructor, zend_bool persistent ZEND_FILE_LINE_DC)
{
    //初始化 gc 信息
    GC_REFCOUNT(ht) = 1;
    GC_TYPE_INFO(ht) = IS_ARRAY;
    //设置 flags
    ht->u.flags = (persistent ? HASH_FLAG_PERSISTENT : 0) | HASH_FLAG_APPLY_
PROTECTION | HASH_FLAG_STATIC_KEYS;
    //这里会把数组大小重置为 2 的幂次方
    ht->nTableSize = zend_hash_check_size(nSize);
    //nTableMask 的值也是临时的
    ht->nTableMask = HT_MIN_MASK;
    //临时设置 ht->arData
    HT_SET_DATA_ADDR(ht, &uninitialized_bucket);
    ht->nNumUsed = 0;
    ht->nNumOfElements = 0;
    ht->nInternalPointer = HT_INVALID_IDX;
    ht->nNextFreeElement = 0;
```

```
    ht->pDestructor = pDestructor;
}
```

此时的HashTable只是设置了散列表的大小及其他一些成员的初值，还无法用来存储元素。

3.3.4 插入

插入时首先会检查数组是否已经分配存储空间，因为在初始化并没有实际分配arData的内存，在第一次插入时才会根据nTableSize的大小分配，分配完以后会把HashTable->u.flags打上HASH_FLAG_INITIALIZED掩码，这样下次插入时发现已经分配了就不会再重复操作。

```
#define CHECK_INIT(ht, packed) \
    zend_hash_check_init(ht, packed)

//_zend_hash_add_or_update_i:
if (UNEXPECTED(!(ht->u.flags & HASH_FLAG_INITIALIZED))) {
    //如果数组还没有分配arData内存
    CHECK_INIT(ht, 0);
    goto add_to_hash;
}
```

如果arData没有分配，则最终由zend_hash_real_init_ex()完成内存的分配。分配的内存包括中间映射表及元素数组：nTableSize * (sizeof(Bucket) + sizeof(uint32_t))，分配完以后将HashTable->arData指向第1个Bucket的位置。

```
static void zend_always_inline zend_hash_real_init_ex(HashTable *ht, int
packed)
{
    ...
    //设置nTableMask
    (ht)->nTableMask = -(ht)->nTableSize;
    //分配Bucket数组及映射数组
    HT_SET_DATA_ADDR(ht, pemalloc(HT_SIZE(ht), (ht)->u.flags & HASH_FLAG_
PERSISTENT));
    (ht)->u.flags |= HASH_FLAG_INITIALIZED;
    ...
    //初始化映射数组的value为-1
```

```
    HT_HASH_RESET(ht);
}
```

完成 Bucket 数组的分配后就可以进行插入操作了，插入时首先将元素按照顺序插入 arData，然后将其在 arData 数组中的位置存储到根据 key 的 hash code（即 key->h）与 nTableMask 计算得到的中间映射表中的对应位置。

```
//_zend_hash_add_or_update_i:
add_to_hash:
    HANDLE_BLOCK_INTERRUPTIONS();
    //idx 为 Bucket 在 arData 中的存储位置
    idx = ht->nNumUsed++;
    ht->nNumOfElements++;
    ...
    //找到存储 Bucket，设置 key、value
    p = ht->arData + idx;
    p->key = key;
    ...
    p->h = h = ZSTR_H(key);
    ZVAL_COPY_VALUE(&p->val, pData);
    //计算中间映射表的散列值，idx 将保存在映射数组的 nIndex 位置
    nIndex = h | ht->nTableMask;
    //将映射表中原来的值保存到新 Bucket 中，哈希冲突时会用到
    Z_NEXT(p->val) = HT_HASH(ht, nIndex);
    //保存 idx: ((uint32_t*))(ht->arData)[nIndex] = idx
    HT_HASH(ht, nIndex) = HT_IDX_TO_HASH(idx);

    return &p->val;
```

上面的过程只是最基本的插入，当然插入时还有一些其他逻辑，比如插入的元素已经存在的处理，这种情况下需要根据插入的策略处理，如果想覆盖以前的值，则除了将元素的 value 更新为新的值外，还会调用 pDestructor 设定的函数对旧的元素进行清理，如果不想覆盖则会直接返回，插入失败。

3.3.5 哈希冲突

散列表中不同元素的 key 可能计算得到相同的哈希值，这些具有相同哈希值的元素在插入

散列表时就会发生冲突，因为映射表只能存储一个元素。常见的一种解决方法是把冲突的 Bucket 串成链表，这样一来中间映射表映射出的就不再是一个 Bucket，而是一个 Bucket 链表，查找时需要遍历这个链表，逐个比较 key，从而找到目标元素，PHP 实现的散列表也是采用该方法解决的哈希冲突。

HashTable 中的 Bucket 会记录与它冲突的元素在 arData 数组中的存储位置，本质上这也是一个链表。在设置映射值时，如果发现中间映射表中要设置的位置已经被之前插入的元素占用了（值不等于初始化的-1），那么会把已经存在的值保存到新插入的 Bucket 中，然后将映射表中的值更新为新 Bucket 的存储位置，即每次都会把冲突的元素插到开头。

冲突元素的保存位置并没有直接放在 Bucket 结构中，而是保存到了存储元素 zval 的 u2 结构中，即 Bucket.val.u2.next，所以在插入元素时分为以下两步。

```
//先把旧的值保存到新插入的元素中
Z_NEXT(p->val) = HT_HASH(ht, nIndex);
//再把新元素数组存储位置更新到映射表中
HT_HASH(ht, nIndex) = HT_IDX_TO_HASH(idx); //就是 idx
```

下面举个例子，一个数组有 3 个元素，按照 a、b、c 的顺序插入，假如 a、c 两个 key 冲突了，则 HashTable 的结构如图 3-5 所示。

```
$arr = [];
$arr['a'] = 11;
$arr['b'] = 22;
$arr['c'] = 33;
```

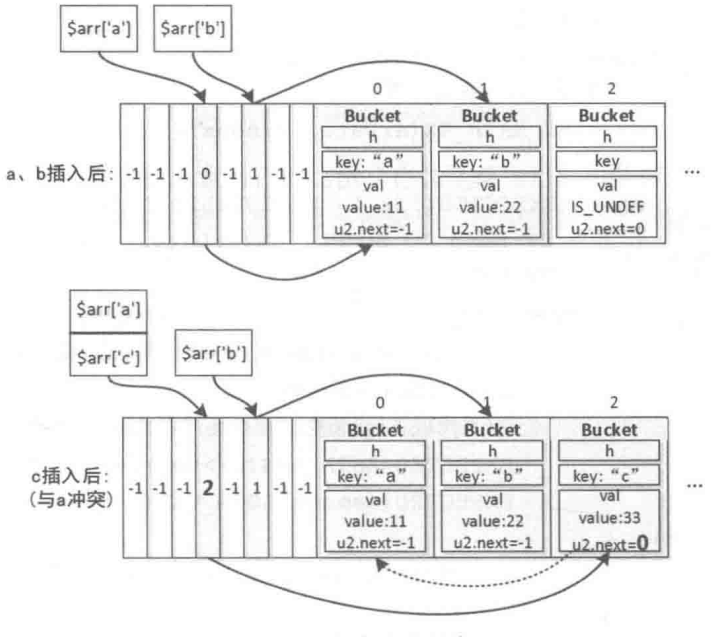


图 3-5　哈希冲突链表

3.3.6 查找

清楚了 HashTable 的实现及哈希冲突的解决方式后，查找的过程就比较简单了：首先根据 key 计算出的 hash code（即 zend_string->h）与 nTableMask 计算得到散列值 nIndex，然后根据散列值从中间映射表中得到存储元素在有序存储数组中的位置 idx，接着根据 idx 从有序存储数组（即 HashTable->arData）中取出 Bucket，最后从取出的 Bucket 开始遍历，判断 Bucket 的 key 是否是要查找的 key，如果是则终止遍历，否则继续根据 zval.u2.next 遍历比较。

```
    //zend_hash_find_bucket:
    //根据 zend_string *key 查找:
    zend_ulong h;
    uint32_t nIndex;
    uint32_t idx;
    Bucket *p, *arData;

    h = zend_string_hash_val(key);
    arData = ht->arData;
    //计算散列值
    nIndex = h | ht->nTableMask;
    //获取 Bucket 存储位置
    idx = HT_HASH_EX(arData, nIndex);
    //遍历
    while (EXPECTED(idx != HT_INVALID_IDX)) {
        p = HT_HASH_TO_BUCKET_EX(arData, idx);
        if (EXPECTED(p->key == key)) { /* check for the same interned string */
            return p;
        } else if (EXPECTED(p->h == h) && //先比较 hash code
                EXPECTED(p->key) &&
                //再比较 key 长度，最后按字符比较是否相同
                EXPECTED(ZSTR_LEN(p->key) == ZSTR_LEN(key)) &&
                EXPECTED(memcmp(ZSTR_VAL(p->key), ZSTR_VAL(key), ZSTR_LEN(key))
== 0)) {
                //比较查找的 key 与 Bucket 的 key 是否匹配
            return p;
        }
        //不匹配则继续遍历
        idx = Z_NEXT(p->val);
    }
```

3.3.7 扩容

数组的容量是有限的，最多可存储 nTableSize 个元素，那么当数组空间已满还要继续插入时该如何处理呢？PHP 的数组实现了自动扩容，在插入前首先会检查是否有空闲空间，当发现空间已满没有位置容纳新元素时就会触发扩容逻辑，扩容以后再执行插入。

```
#define ZEND_HASH_IF_FULL_DO_RESIZE(ht)                \
    if ((ht)->nNumUsed >= (ht)->nTableSize) {          \
        zend_hash_do_resize(ht);                       \
    }

static zend_always_inline zval *_zend_hash_add_or_update_i(HashTable *ht,
zend_string *key, zval *pData, uint32_t flag ZEND_FILE_LINE_DC)
{
    ...
    //检查是否需要扩容
    ZEND_HASH_IF_FULL_DO_RESIZE(ht);
add_to_hash: //插入
    ...
}
```

扩容的过程为：首先检查数组中已经删除的元素所占的比例（也就是那些已经删除但未从存储数组中移除的的元素），如果比例达到阈值则触发重建索引的操作，这个过程会把删除的 Bucket 移除，然后把后面的 Bucket 往前移补上空缺的 Bucket；如果还没有达到阈值，则会分配一个原数组大小 2 倍的新数组，然后把原数组的元素复制到新数组上，最后重建索引。这个阈值并不是一个固定值，它是根据下面的公式判断的：

```
ht->nNumUsed > ht->nNumOfElements + (ht->nNumOfElements >> 5)
```

具体的处理过程：

```
static void ZEND_FASTCALL zend_hash_do_resize(HashTable *ht)
{
    if (ht->nNumUsed > ht->nNumOfElements + (ht->nNumOfElements >> 5)) {
//无须扩容
        //只有到一定阈值才进行 rehash 操作
        zend_hash_rehash(ht); //重建索引数组
```

```
    } else if (ht->nTableSize < HT_MAX_SIZE) {//扩容
        void *new_data, *old_data = HT_GET_DATA_ADDR(ht);
        //扩大为 2 倍，加法要比乘法快
        uint32_t nSize = ht->nTableSize + ht->nTableSize;
        Bucket *old_buckets = ht->arData;
        //新分配 arData 空间，大小为(sizeof(Bucket) + sizeof(uint32_t)) * nSize
        new_data = pemalloc(HT_SIZE_EX(nSize, -nSize), ...);
        ht->nTableSize = nSize;
        ht->nTableMask = -ht->nTableSize;
        //将 arData 指针偏移到 Bucket 数组起始位置
        HT_SET_DATA_ADDR(ht, new_data);
        //将旧的 Bucket 数组复制到新空间
        memcpy(ht->arData, old_buckets, sizeof(Bucket) * ht->nNumUsed);
        //释放旧空间
        pefree(old_data, ht->u.flags & HASH_FLAG_PERSISTENT);

        //重建索引数组：映射表
        zend_hash_rehash(ht);
        ...
    }
    ...
}
```

将旧的数组复制到扩容后的数组时只复制存储的元素，即 HashTable->arData，不会复制中间映射表，因为扩容后旧的映射表已经无法使用了，key-value 的映射关系需要重新计算，这一步骤为重建索引。如果不考虑将已删除的 Bucke 移除，那么重建索引的过程实际上就是将所有元素重新插入了一遍，其处理过程比较简单：

```
//遍历数组，重新设置中间映射表(索引表)
do {
    nIndex = p->h | ht->nTableMask;
    Z_NEXT(p->val) = HT_HASH(ht, nIndex);
    HT_HASH(ht, nIndex) = HT_IDX_TO_HASH(i);
    p++;
} while (++i < ht->nNumUsed);
```

重建索引的过程还会将已删除的 Bucket 移除，移除后会把这个 Bucket 之后的元素全部向前移动一个位置，所以在重建索引后存储数组中元素全部紧密排列在一起。

除了上面介绍的这些操作，数组还有大量的其他操作，比如数组的复制、合并、销毁、重置等，具体的操作定义在 zend_hash.c 中，这里不再一一说明。

3.4　引用

引用并不是一种独立的类型，而是一种指向其他数据类型的结构，类似 C 语言中指针的概念。当修改引用类型的变量时，其修改将反映到实际引用的变量上。在 PHP 中通过&操作符生成一个引用变量，比如$a = &$b，执行时首先为&操作的变量分配一个 zend_reference 结构，这个结构就是引用类型的结构体，它内嵌了一个 zval，此 zval 的 value 指向原来 zval 的 value，然后将原 zval 的类型修改为 IS_REFERENCE，原 zval 的 value 指向新创建的 zend_reference 结构。也就是说，&是将变量的类型转化为了引用，新生成的引用结构指向原来的 value。

```
struct _zend_reference {
    zend_refcounted_h gc;
    zval              val; //指向原来的 value
};
```

下面举个具体例子：

```
$a = date("Y-m-d");
$b = &$a;
```

$a 在 date()函数返回后被赋值为字符串，然后通过&$a 将其转为引用类型并赋值给了另外一个变量$b，转换后的$a 的类型已经不再是 IS_STRING，而变为 IS_REFERENCE，$a 的 value 也转变为 zend_reference 结构，这个结构的 val.value 指向了原来的字符串。最终的结果：$a、$b 指向 zend_reference，其引用计数为 2，然后 zend_reference 指向原 zend_string，其引用计数为 1，也就是$a、$b 间接地指向了实际的 value 值，如图 3-6 所示。

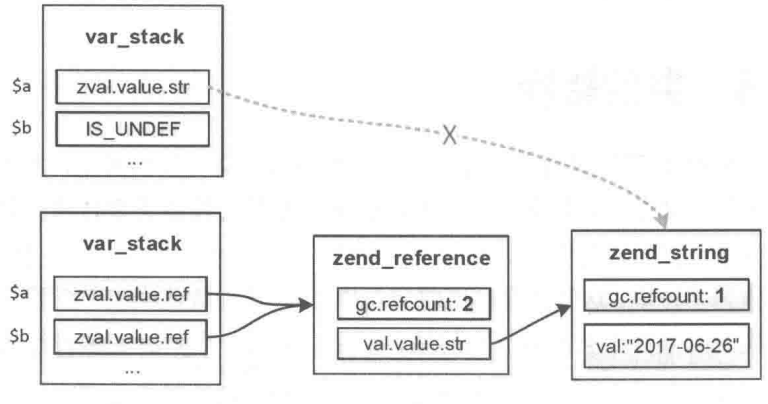


图 3-6　a 与 b 内存引用关系

使用引用时需要注意，引用只能通过&产生，无法通过赋值传递，比

如上面的例子，如果后面再把$b 赋值给其他变量，那么传递给新变量的 value 将是实际引用的值，而不再是引用本身，如图 3-7 所示。这表示 PHP 中的引用只有一级，不会出现一个引用指向另外一个引用的情况 ，也就是没有 C 语言中多级指针的概念。

```
$a = date("Y-m-d");
$b = &$a;
$c = $b; //如果想让$c 也以引用指向$a/$b 引用的值，则：$c = &$b 或 $c = &$a
```

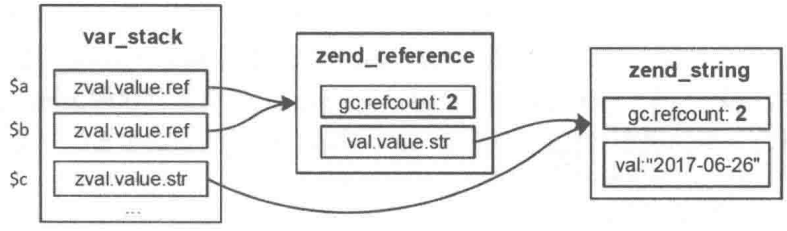


图 3-7 a、b 与 c 内存引用关系

> 上面的例子可以通过 gdb 在 opcode 执行的位置设置断点，然后每执行一条 opcode 看一下 zval 结构的变化：
>
> ```
> $ gdb php
> (gdb) break execute_ex
> (gdb) r test.php
> (gdb) n
> (gdb) p ((zval)((char*)execute_data + 96)).value
> ```
>
> execute_ex()为 opcode 执行的 handler，可以在这个函数中捕获所有 opcode 的执行，局部变量分配在 zend_execute_data，具体的分配规则在第 5 章介绍 PHP 解析及执行时再详细说明。

3.5 类型转换

PHP 是弱类型语言，使用时不需要明确定义变量的类型，Zend 虚拟机在执行 PHP 代码时会根据具体的应用场景进行转换，也就是变量会按照类型转换的规则将不合格的变量转为合格的变量，然后进行操作。比如加法操作：$a = "100" + 200，执行时 Zend 发现相加的其中一个值为字符串，这时就会试图将字符串"100"转为数值类型（整型或浮点型），然后与 200 相加，转换的时候并不会改变原来的值，而是会生成一个新的变量进行处理。

除了自动类型转换，PHP 还提供了一种强制的转换方式。

- **(int)/(integer)**：转换为整型 integer。
- **(bool)/(boolean)**：转换为布尔类型 boolean。
- **(float)/(double)/(real)**：转换为浮点型 float。
- **(string)**：转换为字符串 string。
- **(array)**：转换为数组 array。
- **(object)**：转换为对象 object。
- **(unset)**：转换为 NULL。

无论是自动类型转换还是强制类型转换，这些并非都是万能的，有些类型之间是无法转换的，比如资源类型，无法将任何其他类型转为资源类型。接下来详细看一下不同类型之间的转换规则，这些转换方法定义在 zend_operators.c 中，其中一类是直接对原 value 进行转换，另外一类是不改变原来的值。

3.5.1　转换为 NULL

这类转换比较简单，任意类型都可以转为 NULL，转换时直接将新的 zval 类型设置为 IS_NULL。

```
ZEND_API void ZEND_FASTCALL convert_to_null(zval *op)
{
    if (Z_TYPE_P(op) == IS_OBJECT) {
        if (Z_OBJ_HT_P(op)->cast_object) {
            //如果转化的是一个对象且定义了 case_object 的 handler，则调用
            ...
        }
    }
    //销毁 zval，将类型设置为 NULL
    zval_ptr_dtor(op);
    ZVAL_NULL(op);
}
```

3.5.2　转换为布尔型

当转换为布尔型时，根据原值的 true、false 决定转换后的结果，以下值被认为是 false，除了下面这些情况，其他值通常被认为是 true，比如资源、对象（这里指默认情况下，因为可以

通过扩展改变这个规则)。

- 布尔值 false 本身。
- 整型值 0。
- 浮点型值 0.0。
- 空字符串，以及字符串“0”。
- 空数组。
- NULL。

i_zend_is_true()这个方法用于判断不同类型的 zval 是否为 TRUE，在需要布尔型的使用场景中会调用该方法进行判断，比如 `if($var){}`。在扩展中可以通过 convert_to_boolean()这个函数直接将原 zval 转为 bool 型，转换时的判断逻辑与 i_zend_is_true()一致。

```
static zend_always_inline int i_zend_is_true(zval *op)
{
    int result = 0;

again:
    switch (Z_TYPE_P(op)) {
        case IS_TRUE:
            result = 1;
            break;
        case IS_LONG:
            //非 0 即真
            if (Z_LVAL_P(op)) {
                result = 1;
            }
            break;
        case IS_DOUBLE:
            if (Z_DVAL_P(op)) {
                result = 1;
            }
            break;
        case IS_STRING:
            //非空字符串及"0"外都为 true
            if (Z_STRLEN_P(op) > 1 || (Z_STRLEN_P(op) && Z_STRVAL_P(op)[0] !=
'0')) {
```

```
                result = 1;
            }
            break;
        case IS_ARRAY:
            //非空数组为 true
            if (zend_hash_num_elements(Z_ARRVAL_P(op))) {
                result = 1;
            }
            break;
        case IS_OBJECT:
            //默认情况下始终返回 true
            result = zend_object_is_true(op);
            break;
        case IS_RESOURCE:
            //合法资源就是 true
            if (EXPECTED(Z_RES_HANDLE_P(op))) {
                result = 1;
            }
        case IS_REFERENCE:
            //引用类型判断的是实际引用的值
            op = Z_REFVAL_P(op);
            goto again;
            break;
        default:
            break;
    }
    return result;
}
```

3.5.3 转换为整型

从其他类型转为整型的规则。

- NULL：转为 0。
- 布尔型：false 转为 0，true 转为 1。
- 浮点型：向下取整，比如(int)2.8 => 2。
- 字符串：就是 C 语言 strtoll()的规则，如果字符串以合法的数值开始，则使用该数值，

否则其值为 0（零），合法数值由可选的正负号，后面跟着一个或多个数字（可能有小数点），再跟着可选的指数部分组成。

- 数组：很多操作不支持将一个数组自动转为整型处理，比如 array() + 2，将报 error 错误，但可以强制把数组转为整型，非空数组转为 1，空数组转为 0。
- 对象：与数组类似，很多操作也不支持将对象自动转为整型，但有些操作只会抛一个 warning 警告，还是会把对象转为 1 操作，这个需要看不同操作的处理情况。
- 资源：转为分配给这个资源的唯一编号。

_zval_get_long_func()根据以上规则返回不同类型 zval 转为整型后的值，convert_to_long()直接将原 zval 转为整型，其判断逻辑是相同的。

```
ZEND_API zend_long ZEND_FASTCALL _zval_get_long_func(zval *op)
{
try_again:
    switch (Z_TYPE_P(op)) {
        case IS_NULL:
        case IS_FALSE:
            return 0;
        case IS_TRUE:
            return 1;
        case IS_RESOURCE:
            //资源将转为 zend_resource->handler
            return Z_RES_HANDLE_P(op);
        case IS_LONG:
            return Z_LVAL_P(op);
        case IS_DOUBLE:
            return zend_dval_to_lval(Z_DVAL_P(op));
        case IS_STRING:
            //字符串的转换调用 C 语言的 strtoll()处理
            return ZEND_STRTOL(Z_STRVAL_P(op), NULL, 10);
        case IS_ARRAY:
            //根据数组是否为空转为 0,1
            return zend_hash_num_elements(Z_ARRVAL_P(op)) ? 1 : 0;
        case IS_OBJECT:
            {
                zval dst;
                convert_object_to_type(op, &dst, IS_LONG, convert_to_long);
```

```
            if (Z_TYPE(dst) == IS_LONG) {
                return Z_LVAL(dst);
            } else {
                //默认情况就是1
                return 1;
            }
        }
    case IS_REFERENCE:
        op = Z_REFVAL_P(op);
        goto try_again;
        EMPTY_SWITCH_DEFAULT_CASE()
    }
    return 0;
}
```

3.5.4 转换为浮点型

除字符串类型外，其他类型转换规则与整型基本一致，只是在整型转换结果上加了小数位，字符串转为浮点数由 zend_strtod()完成，这个函数非常长，定义在 zend_strtod.c 中。

3.5.5 转换为字符串

一个值可以通过在其前面加上(string)或用 strval()函数来转变成字符串。在一个需要字符串的表达式中，会自动转换为 string，比如在使用函数 echo 或 print 时，或在一个非 string 类型变量和一个 string 类型变量进行比较时，就会发生这种转换。从其他类型转为字符串的规则。

- NULL/FALSE：转为空字符串。
- TRUE：转为“1”。
- 整型：原样转为字符串，转换时将各位依次除 10 取余。
- 浮点型：原样转为字符串。
- 资源：转为“Resource id #xxx”。
- 数组：转为“Array”，但是报 Notice。
- 对象：不能转换，将报错。

```
ZEND_API zend_string* ZEND_FASTCALL _zval_get_string_func(zval *op)
```

```
{
try_again:
    switch (Z_TYPE_P(op)) {
        case IS_UNDEF:
        case IS_NULL:
        case IS_FALSE:
            //转为空字符串""
            return ZSTR_EMPTY_ALLOC();
        case IS_TRUE:
            //转为"1"
            ...
            return zend_string_init("1", 1, 0);
        case IS_RESOURCE: {
            //转为"Resource id #xxx"
            ...
            len = snprintf(buf, sizeof(buf), "Resource id #" ZEND_LONG_FMT,
(zend_long)Z_RES_HANDLE_P(op));
            return zend_string_init(buf, len, 0);
        }
        case IS_LONG: {
            return zend_long_to_str(Z_LVAL_P(op));
        }
        case IS_DOUBLE: {
            return zend_strpprintf(0, "%.*G", (int) EG(precision), Z_DVAL_P
(op));
        }
        case IS_ARRAY:
            //转为"Array"，但是报 Notice
            zend_error(E_NOTICE, "Array to string conversion");
            return zend_string_init("Array", sizeof("Array")-1, 0);
        case IS_OBJECT: {
            //报 Error 错误
            zval tmp;
            ...
            return ZSTR_EMPTY_ALLOC();
        }
        case IS_REFERENCE:
            op = Z_REFVAL_P(op);
```

```
            goto try_again;
        case IS_STRING:
            return zend_string_copy(Z_STR_P(op));
        EMPTY_SWITCH_DEFAULT_CASE()
    }
    return NULL;
}
```

3.5.6 转换为数组

如果将一个 null、integer、float、string、boolean 和 resource 类型的值转换为数组，则将得到一个仅有一个元素的数组，其下标为 0，该元素为此标量的值。换句话说，`(array)$scalarValue` 与 `array($scalarValue)` 完全一样。

如果一个 object 类型转换为 array，则结果为一个数组，数组元素为该对象的全部属性，包括 public、private、protected，其中 private 的属性转换后的 key 加上了类名作为前缀，protected 属性的 key 加上了“*”作为前缀，但是这个前缀并不是转为数组时单独加上的，而是类编译生成属性 zend_property_info 时就已经加上了。也就是说，这其实是成员属性本身的一个特点，举例来看：

```
class test {
    private $a = 123;
    public $b = "bbb";
    protected $c = "ccc";
}
$obj = new test;
print_r((array)$obj);
```

以上例子将输出：

```
Array
(
    [testa] => 123
    [b] => bbb
    [*c] => ccc
)
```

具体的转换逻辑：

```
ZEND_API void ZEND_FASTCALL convert_to_array(zval *op)
{
    try_again:
    switch (Z_TYPE_P(op)) {
        case IS_ARRAY:
            break;
        case IS_OBJECT:
            ...
            if (Z_OBJ_HT_P(op)->get_properties) {
                //获取所有属性数组
                HashTable *obj_ht = Z_OBJ_HT_P(op)->get_properties(op);
                //将数组内容复制到新数组
                ...
            }
        case IS_NULL:
            ZVAL_NEW_ARR(op);
            //转为空数组
            zend_hash_init(Z_ARRVAL_P(op), 8, NULL, ZVAL_PTR_DTOR, 0);
            break;
        case IS_REFERENCE:
            zend_unwrap_reference(op);
            goto try_again;
        default:
            convert_scalar_to_array(op);
            break;
    }
}

//其他标量类型转 array
static void convert_scalar_to_array(zval *op)
{
    zval entry;
    ZVAL_COPY_VALUE(&entry, op);
    //新分配一个数组，将原值插入数组
    ZVAL_NEW_ARR(op);
    zend_hash_init(Z_ARRVAL_P(op), 8, NULL, ZVAL_PTR_DTOR, 0);
```

```
        zend_hash_index_add_new(Z_ARRVAL_P(op), 0, &entry);
    }
```

3.5.7 转换为对象

如果其他任何类型的值被转换成对象，将会创建一个内置类 stdClass 的实例：如果该值为 NULL，则新的实例为空；array 转换成 object 将以键名成为属性名并具有相对应的值，数值索引的元素也将转为属性，但是无法通过“->”访问，只能遍历获取；对于其他值，会以“scalar”作为属性名。

```
ZEND_API void ZEND_FASTCALL convert_to_object(zval *op)
{
try_again:
    switch (Z_TYPE_P(op)) {
        case IS_ARRAY:
            {
                HashTable *ht = Z_ARR_P(op);
                ...
                //以 key 为属性名，将数组元素复制到对象属性
                object_and_properties_init(op, zend_standard_class_def, ht);
                break;
            }
        case IS_OBJECT:
            break;
        case IS_NULL:
            object_init(op);
            break;
        case IS_REFERENCE:
            zend_unwrap_reference(op);
            goto try_again;
        default: {
            zval tmp;
            ZVAL_COPY_VALUE(&tmp, op);
            object_init(op);
            //以 scalar 作为属性名
            zend_hash_str_add_new(Z_OBJPROP_P(op), "scalar", sizeof("scalar")-1,
&tmp);
```

```
            break;
        }
    }
}
```

3.6 小结

本章主要介绍了 PHP 中数据类型的内部实现，即 zval、zend_value，它们是 PHP 中最基础的两个结构，在内核中随处用到。另外详细介绍了字符串、数组及引用类型的内部实现，其他几种类型（如对象、资源等）在后续章节中再单独说明。 在接下来的一章中我们会继续介绍这些数据类型的内存管理。

第 4 章
内存管理

上一章介绍了 PHP 中数据类型的实现，主要为 zval、zend_value 两个结构体，它们是 PHP 中最基础的两个数据结构。除了这两个结构，还有很多其他类型的结构，基于这些数据结构，PHP 实现自己的变量、常量、静态变量等，其中变量是这些数据类型最常见的表现形式。PHP 中的变量是不需要手动释放的，内核帮我们实现了变量的内存管理，包括内存的分配与回收。本章将重点介绍 PHP 中与内存相关的实现，包括变量的 GC 机制、垃圾回收以及底层的内存池实现，另外还将介绍线程安全相关的实现。

4.1 变量的自动 GC 机制

C/C++语言中，如果想在堆上分配变量，需要手动进行内存的分配与释放，变量的内存管理是一件非常烦琐的事情，稍有不慎就可能导致不可预知的错误。现代高级语言普遍提供了变量的自动 GC 机制，由语言自己进行管理，这使得开发者不需要再去关心变量的分配与释放，将开发者从内存管理的苦海中解脱出来。PHP 同样实现了这种机制，在 PHP 中可以直接通过“$”声明一个变量，使用完也不需要手动销毁，内核自己清楚什么时间该进行释放。

我们先自己思考下如何实现自动 GC，最简单的实现方式：在函数中定义变量时分配一块内存，用于保存 zval 及对应的 value 结构，在函数返回时再将内存释放，如果在函数执行期间该变量作为参数调用了其他函数或者赋值给了其他变量，则把变量复制一份，变量之间互相独立，不会出现冲突。

这种方式是可行的，而且内存管理也很简单，但是，深拷贝带来的一个无法接受的问题是效率，而且内存浪费严重，比如我们定义了一个变量然后赋值给另外一个变量，可能后面都只是只读操作，假如深拷贝的话就会有多余的一份数据。这个问题比较通用的解决方案是：引用计数+写时复制，PHP 变量的内存管理正是基于这两点实现的。当变量赋值、传递时不是直接进行深拷贝，而是多个变量共用同一个 value，引用计数用来记录 value 有多少个变量在使用；当某个变量的 value 发生改变时将无法继续与其他变量共用 value，这个时候就需要进行深拷贝分离 value，这就是写时复制。

4.1.1 引用计数

引用计数用来记录当前有多少 zval 指向同一个 zend_value。当有新的 zval 指向这个 value 时，计数器加 1；当 zval 销毁时，计数器减 1。当引用计数为 0 时，表示此 value 已经没有被任何变量指向，这个时候就可以对 value 进行释放了。

PHP7 中将变量的引用计数保存在了 zend_value 中，也就是不同类型的结构中，这一点与之前的版本不同，旧版本中引用计数保存在 zval 中。回顾下上一章介绍的各种数据类型的结构，这些不同类型的结构体中都有一个相同的成员：gc，这个结构就是用来保存引用计数的，它的类型为 zend_refcounted_h。

```
typedef struct _zend_refcounted_h {
    //引用计数
    uint32_t        refcount;
    union {
        struct {
            ZEND_ENDIAN_LOHI_3(
                //类型
                zend_uchar    type,
                zend_uchar    flags,
                //垃圾回收时用到
                uint16_t      gc_info)
        } v;
        uint32_t type_info;
    } u;
} zend_refcounted_h;
```

举个例子来看：

```
$a = array();   //$a      -> zend_array(refcount=1)
$b = $a;        //$a,$b    -> zend_array(refcount=2)
$c = $b;        //$a,$b,$c -> zend_array(refcount=3)
unset($b);      //$a,$c -> zend_array(refcount=2)  $b = IS_UNDEF
```

并不是所有的类型都会用到引用计数，没有具体 value 结构的类型是不会用到的，比如整型、浮点型、布尔型、NULL，它们的值直接通过 zval 保存，因此这些类型不会共用 value，而是深拷贝，也就是上面我们介绍的那种最简单的内存模型。除了这些不会用到引用计数的类型，还有一些类型在某些特殊情况下也不会使用引用计数，先看个例子：

```
//refcount.php
$a = "hi";
$b = $a;
```

大家可以自己先猜测下字符串“hi”的引用计数，接下来我们通过 gdb 看一下其实际的引用情况。

```
$ gdb /usr/local/php7/bin/php
(gdb) break execute_ex
(gdb) run refcount.php
(gdb) n
```

一直 n，执行完第 2 条 opcode 后停下，也就是$b 赋值完成后，此时打印 value 的 refcount 看一下其具体值是多少。

```
(gdb) p ((zval*)((char*)execute_data+96)).value.str.gc
$5 = {refcount = 0, u = {v = {type = 6 '\006', flags = 2 '\002', gc_info =
0}, type_info = 518}}
```

这里提前介绍一下，PHP 中局部变量的 zval 分配在 zend_execute_data 结构上，也就是我们调试的 execute_ex()函数中的 execute_data 变量，它是运行期间最重要、最关键的一个结构，用于局部变量的分配、保存执行位置、调用上下文切换等。局部变量分配在这个结构体的末尾，在执行前会根据局部变量的数量确定内存的大小，变量按照定义的先后顺序依次分配，96 指的是内存的 offset 值，也就是局部变量区的起始位置，前面的内存被 zend_execute_data 结构体成员占用。当然这个值并不是唯一不变的，在不同机器上它的值可能不一样，笔者使用的机器为 x86_64 GNU/Linux。所以这里的变量 a 就是(zval*)((char*)execute_data+96)，变量 b 为(zval*)((char*)execute_data+112)。

上面我们在 gdb 中打印的值就是 a、b 两个变量 value 的引用计数结构，可以看到其 refcount=0，下面换一个例子：

```
$a = "hi" . time();
$b = $a;
```

后面这个例子 refcount=2，为什么同样是字符串但是它们的引用计数却不一样呢？在 PHP 中除了上面介绍的几种没有 value 结构的类型不会用到引用计数，还有两种特殊情况不会用到：内部字符串（interned string）、不可变数组（immutable array），它们的类型是字符串、数组。

- **interned string：** 内部字符串，在 PHP 中写的函数名、类名、变量名、静态字符串等都是这种类型，第一个例子$a = "hi"，后面的字符串内容是唯一不变的，这些字符串等同于 C 语言中定义在静态变量区的字符串 char *a = "hi"，这些字符串的生命周期为整个请求执行期间，request 完成后会统一销毁释放，自然也就无须在运行期间通过引用计数来管理内存。
- **immutable array：** 不可变数组，这种情况是 opcache 优化出的一种类型，这里不做详细说明。

内部字符串与普通字符串的类型都是 IS_STRING，它们并不是通过 type 进行区分的，而是通过 zend_refcounted_h.u.v.flag 来区分，内部字符串这个值将包含 IS_STR_INTERNED。除了内部字符串与普通字符串之分，flag 还用来表示其他含义，比如持久化字符串，其他类型的 flag 也有各自不同的含义。

上面的例子说明无法通过 value 的类型判断一个变量是否使用引用计数机制，那么如何标识一个 value 是否支持引用计数呢？还记得 zval.u1 中那个类型掩码 type_flag 吗？正是通过这个值标识的，它是一个 bitmap，除了标识 value 是否支持引用计数，还有其他几个标识位用于其他含义。支持引用计数的 value 类型是 zval.u1.type_flag & IS_TYPE_REFCOUNTED。

```
#define IS_TYPE_REFCOUNTED          (1<<2)
```

下面具体列一下哪些类型会使用引用计数机制：

```
|    type              |   refcounted   |
+----------------------+----------------+
|simple types          |                |
|string                |       Y        |
|interned string       |                |
|array                 |       Y        |
```

```
|immutable array    |          |
|object             |     Y    |
|resource           |     Y    |
|reference          |     Y    |
```

4.1.2　写时复制

写时复制的机制在计算机系统中有着非常广泛的应用，它只有在必要的时候（即发生写的时候）才会进行深拷贝，可以很好地提升效率，例如 Linux 操作系统中 fork 子进程时并不会立即复制父进程的地址空间，而是让父子进程共享同一个地址空间，只有在需要写入的时候才会复制地址空间，从而使各个进程拥有各自的地址空间。也就是说，资源的复制是在需要写入的时候才会进行，在此之前，以只读方式共享。

变量使用了引用计数必然会出现其中一个变量修改 value 的情况，这个时候就需要对 value 进行分离了，发生修改的变量会复制一份数据出来进行修改，同时断开原来 value 的指向，指向新的 value。举例来看：

```
$a = array(1,2);
$b = &$a;
$c = $a;
//发生分离
$c[] = 3;
```

其引用计数及分离情况如图 4-1 所示。

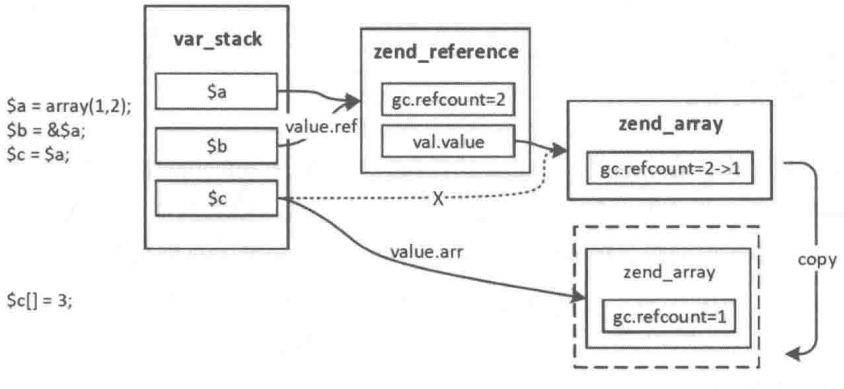


图 4-1　a、b、c 变量间写时复制的过程

然而并不是所有类型的 value 都可以进行复制，比如对象、资源就无法进行复制，也就是

无法进行分离，如果多个变量指向同一个对象，当其中一个变量修改对象时，其修改将反映到所有变量上。事实上只有 string、array 这两种支持 value 的分离，与引用计数相同，这个信息也是通过 zval.u1.type_flag 记录的。

```
#define IS_TYPE_COPYABLE        (1<<4)
```

支持复制的 value 类型：

```
|    type              | copyable |
+----------------------+----------+
|simple types          |          |
|string                |     Y    |
|interned string       |          |
|array                 |     Y    |
|immutable array       |          |
|object                |          |
|resource              |          |
|reference             |          |
```

除了变量的写时复制机制会用到 copyable 这个属性，还有一种情况也会针对这个属性进行处理，那就是变量从 literals 静态数据区复制到局部变量区。比如`$a = array()`，赋值时发现 value：array()支持 copyable，所以会从 literals 中深拷贝一份数据进行赋值。再比如`$a = "hi"`，value："hi"是内部字符串，不支持 copyable，所以不会深拷贝，变量 a 直接指向 literals 中的 value。literal 静态数据区是指在代码中定义的字面量，类似 C 语言中的静态数据区，比如$a = "hi"，编译时就会把字符串"hi"保存在 literals 中，后面介绍 PHP 的编译执行时再详细说明。

4.1.3 回收时机

在自动 GC 机制中，在 zval 断开 value 的指向时如果发现 refcount=0 则会直接释放 value，这就是变量的回收时机，发生断开的两种常见情况为修改变量与函数返回时，修改变量时会断开原有 value 的指向，函数返回时会释放所有的局部变量，也就是把所有局部变量的引用计数减 1。

除了自动 GC，PHP 中也可以通过 unset()函数主动销毁一个变量。

4.2 垃圾回收

通过引用计数 PHP 实现了变量的自动 GC 机制，但是有一种情况是这个机制无法解决的，

从而因变量无法回收导致内存始终得不到释放，造成内存泄漏，这种情况指的是循环引用。简单地讲，循环引用就是变量的内部成员引用了变量自身，比如数组中的某个元素指向了数组，这样一来数组的引用计数中就有一个来自自身成员，当所有的外部引用全部断开时，数组的 refcount 仍然大于 0 而得不到释放，而实际上这种变量不可能再被使用了。举例说明：

```
$a = array(1);
$a[] = &$a;
unset($a);
```

`unset($a)`之前，变量 a 的类型为引用，该引用的 refcount=2，一个来自`$a`，另一个来自`$a[1]`，如图 4-2 所示。

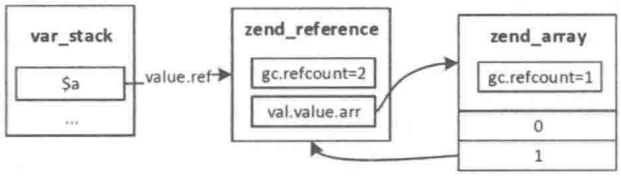


图 4-2　unset($a)前的引用关系

`unset($a)`之后，减少了一次该引用的 refcount，此时已经没有任何外部引用了，但是数组中仍然有一个元素指向该引用，如图 4-3 所示。

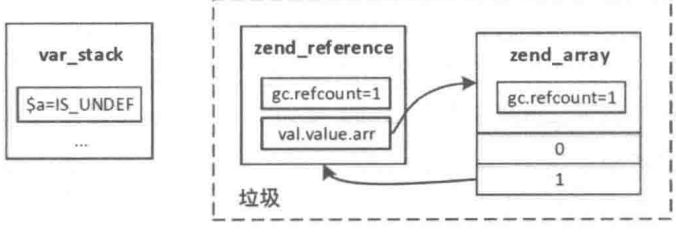


图 4-3　unset($a)后的引用关系

这种因为循环引用而导致的无法释放的变量称之为垃圾，PHP 引入了另外一种机制来对这些垃圾进行回收，也就是垃圾回收器，这里首先明确两个准则：

- 如果一个变量 value 的 refcount 减少到 0，那么此 value 可以被释放掉，不属于垃圾。
- 如果一个变量 value 的 refcount 减少之后大于 0，那么此 value 还不能被释放，此 value 可能成为一个垃圾。

针对第一个情况垃圾回收器不会处理，只有第二种情况垃圾回收器才会将变量收集起来。在 value 的引用计数减少后如果仍然大于 0，那么垃圾回收器就会把可能成为垃圾的 value 收集

起来，等达到一定数量后开始启动垃圾鉴定程序，把真正的垃圾释放掉。

目前垃圾只会出现在 array、object 这两种类型中，数组的情况上面已经介绍了，object 的情况则是成员属性引用对象本身导致的，其他类型不会出现这种变量中的成员引用变量自身的情况，所以垃圾回收器只会处理这两种类型。需要注意的是，垃圾回收器判断是否要收集疑似垃圾时，并不是根据类型进行判断的，而是与前面介绍的是否用到引用计数一样，通过 zval.u1.type_flag 进行标识的，只有包含 IS_TYPE_COLLECTABLE 标识的变量类型才会被收集。

```
#define IS_TYPE_COLLECTABLE          (1<<3)
```

垃圾回收器把收集到的可能垃圾保存到一个 buffer 缓存区中，稍后会介绍这个缓存区。收集的时机是 refcount 减少时，也就是说每次 refcount 减少都会试图收集，比如下面的例子就会触发两次收集动作，第 2 次收集时发现已经收集过了就不再重复收集。

```
$a = array();
$b = $a;
$c = $a;
unset($b);
unset($c);
```

4.2.1 回收算法

等到垃圾回收器收集的可能垃圾达到一定数量后，就会启动垃圾鉴定、回收程序。回收算法的原理很容易理解：既然垃圾是由于成员引用自身导致的，那么就对 value 的所有成员减一遍引用计数，结果如果发现 value 本身 refcount 变为了 0，则就表明其引用全部来自自身成员。具体的回收过程如下所述。

- **步骤（1）**：遍历垃圾回收器的 buffer 缓存区，把当前 value 标为灰色（zend_refcounted_h.gc_info 置为 GC_GREY），然后对当前 value 的成员进行深度优先遍历，把成员 value 的 refcount 减 1，并且也标为灰色。
- **步骤（2）**：重复遍历 buffer，检查当前 value 引用是否为 0，为 0 则表示确实是垃圾，把它标为白色（GC_WHITE），如果不为 0 则排除了引用全部来自自身成员的可能，表示还有外部的引用，并不是垃圾，这时候因为步骤（1）对成员进行了 refcount 减 1 操作，需要再还原回去，对所有成员进行深度遍历，把成员 refcount 加 1，同时标为黑色。
- **步骤（3）**：再次遍历 buffer，将非 GC_WHITE 的节点从 buffer 中删除，最终 buffer 缓

存区中全部为真正的垃圾，最后将这些垃圾释放，回收完成。

4.2.2 具体实现

垃圾回收器主要通过 zend_gc_globals 这个结构对垃圾进行管理，收集到的可能成为垃圾的 value 就保存在这个结构的 buf 中，即垃圾缓存区。

```
typedef struct _zend_gc_globals {
    //是否启用 gc
    zend_bool        gc_enabled;
    //是否在垃圾检查过程中
    zend_bool        gc_active;
    //缓存区是否已满
    zend_bool        gc_full;
    //启动时分配的用于保存可能垃圾的缓存区
    gc_root_buffer   *buf;
    //指向 buf 中最新加入的一个可能垃圾
    gc_root_buffer    roots;
    //指向 buf 中没有使用的 buffer
    gc_root_buffer   *unused;
    //指向 buf 中第一个没有使用的 buffer
    gc_root_buffer   *first_unused;
    //指向 buf 尾部
    gc_root_buffer   *last_unused;
    //待释放的垃圾
    gc_root_buffer    to_free;
    gc_root_buffer   *next_to_free;
    //统计 gc 运行次数
    uint32_t gc_runs;
    //统计已回收的垃圾数
    uint32_t collected;
} zend_gc_globals;
```

buf 用于保存收集到的 value，它是一个数组，在垃圾回收器初始化一次性分配了 10001 个 gc_root_buffer，其中第一个 buffer 被保留，插入 value 时直接取出可用节点即可。root 指向 buf 中最新加入的一个节点，root 是一个双向链表的头部，之所以是一个双向链表，是因为 buf 数组中保存的只是有可能成为垃圾的 value，其中有些 value 在加入之后又被删除了（比如有的 value

在之后的操作中 refcount 变为 0 了，这个时候就需要从 buf 中删除），这样 buf 数组中就会出现一些空隙。first_unused 一开始指向 buf 的第一个位置，有元素插入 roots 时如果 first_unused 还没有到达 buf 的尾部，则返回 first_unused 给最新的元素，然后执行 first_unused++，直到 last_unused。比如现在已经加入了 2 个 gc，则对应的结构如图 4-4 所示。

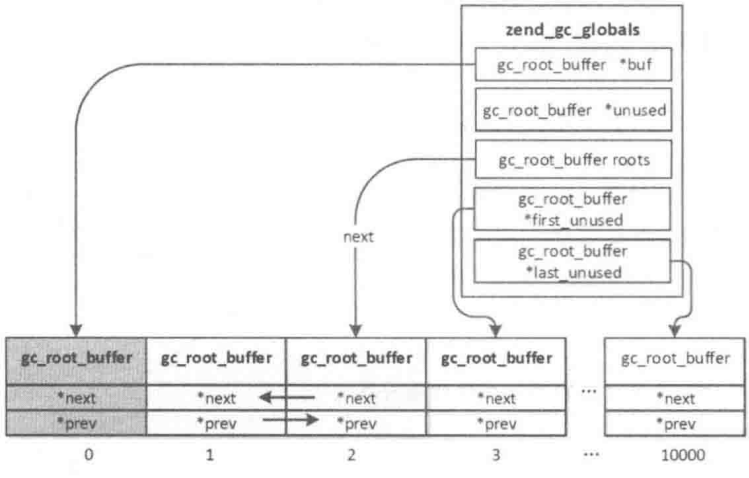


图 4-4 buf 缓存区可用节点

zend_gc_globals 结构中还有一个 unused 成员，它的含义与 first_unused 类似，用来管理 buf 中开始加入后面又删除的节点，这是一个单链表。也就是说，first_unused 是一直往后偏移的，直到 buf 的结尾，buf 中间由于 value 删除而重新空闲的节点则由 unused 串起来。下次有新的 value 插入 roots 时优先使用 unused 的这些节点，其次才是 first_unused 的节点。比如图 4-4 中，如果 buf[1]后来发现不是一个垃圾移除了，则 unused 将指向 buf[1]，如图 4-5 所示。

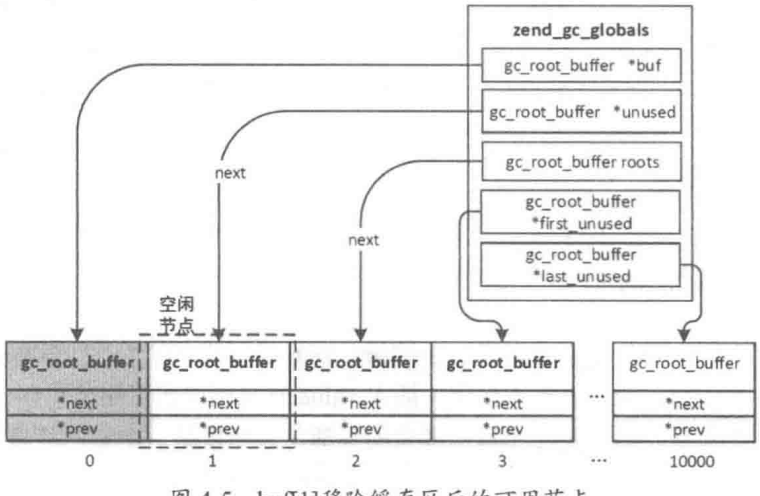


图 4-5 buf[1]移除缓存区后的可用节点

此垃圾回收机制可以通过 php.ini 中的 zend.enable_gc 设置是否开启，默认是开启的，尽管 GC 回收的过程会短暂中断正常逻辑的执行（即垃圾回收停顿），但是关闭这个机制带来的风险更高，除非你有信心自己的代码中不会出现循环引用的情况。如果开启则在 php.ini 解析后调用 gc_init()初始化垃圾回收器：分配 buf 数组内存、设置 first_unused/unused 指针等。

```
ZEND_API void gc_init(void)
{
    if (GC_G(buf) == NULL && GC_G(gc_enabled)) {
        //分配buf缓存区内存，大小为GC_ROOT_BUFFER_MAX_ENTRIES(10001)，其中第1个保留不被使用
        GC_G(buf) = (gc_root_buffer*) malloc(sizeof(gc_root_buffer) * GC_ROOT_BUFFER_MAX_ENTRIES);
        GC_G(last_unused) = &GC_G(buf)[GC_ROOT_BUFFER_MAX_ENTRIES];
        //进行GC_G的初始化，其中：GC_G(first_unused) = GC_G(buf) + 1;从第2个开始，第1个保留
        gc_reset();
    }
}
```

初始化以后就可以进行垃圾收集了，在 Zend 执行过程中如果销毁一个变量就会判断是否需要加入垃圾收集器。销毁一个 zval 会调用 i_zval_ptr_dtor()进行处理。

```
//file:zend_variables.h
static zend_always_inline void i_zval_ptr_dtor(zval *zval_ptr ZEND_FILE_LINE_DC)
{
    //不使用引用计数的类型不需要进行回收
    if (Z_REFCOUNTED_P(zval_ptr)) {
        //refcount减一
        if (!Z_DELREF_P(zval_ptr)) {
            //refcount减一后变为0，不是垃圾，正常回收
            _zval_dtor_func_for_ptr(Z_COUNTED_P(zval_ptr) ZEND_FILE_LINE_RELAY_CC);
        } else {
            //refcount减一后仍然大于0
            GC_ZVAL_CHECK_POSSIBLE_ROOT(zval_ptr);
        }
```

```
    }
}
```

从上面的过程可以很直观地看到，在销毁一个 zval 时首先对 refcount 进行减一操作，如果 refcount 变为 0 则表示不是垃圾，正常释放即可，如果减掉后仍大于 0 则表示可能是一个垃圾，这个时候就会被垃圾回收器收集。

```
//file:zend_gc.h
#define GC_ZVAL_CHECK_POSSIBLE_ROOT(z) \
    gc_check_possible_root((z))

static zend_always_inline void gc_check_possible_root(zval *z)
{
    ZVAL_DEREF(z);
    //判断是否是可收集以及是否已经收集过
    if (Z_COLLECTABLE_P(z) && UNEXPECTED(!Z_GC_INFO_P(z))) {
        gc_possible_root(Z_COUNTED_P(z));
    }
}
```

收集时检查变量类型掩码是否包含 IS_TYPE_COLLECTABLE，即数组、对象，如果是其他类型则不会出现循环引用导致的垃圾，也就没必要收集；除了检查是否可收集，还会检查该变量是否已经被收集过，这个是通过 zend_refcounted_h.v.u.gc_info 来判断的，第一次收集后会把这个值设置为 GC_PURPLE，正是通过这个值来避免重复收集。如果变量是可收集的类型且之前没被收集过则会触发收集操作。

收集时首先会从 buf 中选择一个空闲节点，然后将 value 的 gc 保存到这个节点中，如果没有空闲节点则表明回收器已经满了，这个时候就会触发垃圾鉴定、回收的程序。

```
ZEND_API void ZEND_FASTCALL gc_possible_root(zend_refcounted *ref)
{
    gc_root_buffer *newRoot;
    //插入的节点必须是 GC_BLACK，防止重复插入
    ZEND_ASSERT(EXPECTED(GC_REF_GET_COLOR(ref) == GC_BLACK));

    newRoot = GC_G(unused); //先看一下 unused 中有没有可用的
    if (newRoot) {
        //有的话先用 unused 的，然后将 GC_G(unused)指向单链表的下一个
```

```
        GC_G(unused) = newRoot->prev;
    } else if (GC_G(first_unused) != GC_G(last_unused)) {
        //unused没有可用的，且buf中还有可用的
        newRoot = GC_G(first_unused);
        GC_G(first_unused)++;
    } else {
        //buf缓存区已满，这时需要启动垃圾鉴定、回收程序
        ...
    }
    //将插入的ref标为紫色，防止重复插入
    GC_TRACE_SET_COLOR(ref, GC_PURPLE);
    GC_INFO(ref) = (newRoot - GC_G(buf)) | GC_PURPLE;
    newRoot->ref = ref;
    //插入roots链表头部
    newRoot->next = GC_G(roots).next;
    newRoot->prev = &GC_G(roots);
    GC_G(roots).next->prev = newRoot;
    GC_G(roots).next = newRoot;
}
```

收集的过程比较简单，没有什么复杂的处理，只有一个地方需要特别注意，收集后会设置value的zend_refcounted_h.v.u.gc_info信息，这里除了设置为GC_PURPLE避免后续重复插入，还会设置一个信息：GC_INFO(ref) = (newRoot - GC_G(buf)) | GC_PURPLE，这里同时把该节点在buf数组中的位置保存到了gc_info中。这样做的目的是当后续value的refcount变为了0，需要将其从buf中删除时可以知道该value保存在哪个gc_root_buffer中，如果没有这个信息，在删除value时是无法获取gc_root_buffer位置的。删除的操作通过GC_REMOVE_FROM_BUFFER()宏完成。

```
#define GC_REMOVE_FROM_BUFFER(p) do { \
    zend_refcounted *_p = (zend_refcounted*)(p); \
        if (GC_ADDRESS(GC_INFO(_p))) { \
            gc_remove_from_buffer(_p); \
        } \
} while (0)

#define GC_ADDRESS(v) \
    ((v) & ~GC_COLOR)
```

删除时首先根据 gc_info 取到 gc_root_buffer，然后再从 buf 中移除，删除后把空出来的 gc_root_buffer 插入 unused 单链表尾部。

```
ZEND_API void ZEND_FASTCALL gc_remove_from_buffer(zend_refcounted *ref)
{
    gc_root_buffer *root;
    /*GC_ADDRESS 就是获取节点在缓存区中的位置，因为删除是根据 zend_refcounted
    而缓存链表的节点类型是 gc_root_buffer */
    root = GC_G(buf) + GC_ADDRESS(GC_INFO(ref));
    if (GC_REF_GET_COLOR(ref) != GC_BLACK) {
        GC_TRACE_SET_COLOR(ref, GC_PURPLE);
    }
    GC_INFO(ref) = 0;
    //双向链表的删除操作
    GC_REMOVE_FROM_ROOTS(root);
    ...
}
```

最后，我们再来看一下当 buf 缓存区满了执行垃圾回收的过程，该过程的算法原理及步骤前面已经介绍过了，具体操作在 zend_gc_collect_cycles()中完成，这个过程会有很多递归调用的处理，但是实现原理还是比较容易理解的，各个步骤具体的处理细节比较多，这里不再展开。

```
ZEND_API int zend_gc_collect_cycles(void)
{
    ...
    //(1)遍历 roots 链表，对当前节点 value 的所有成员（如数组元素、成员属性）进行深度优先遍历，把成员 refcount 减 1
    gc_mark_roots();
    //(2)再次遍历 roots 链表，检查各节点当前 refcount 是否为 0，是的话标为白色，表示是垃圾，不是的话需要还原(1)，把 refcount 再加回去
    gc_scan_roots();
    //(3)将 roots 链表中的非白色节点删除，之后 roots 链表中全部是真正的垃圾，将垃圾链表转到 to_free 等待释放
    count = gc_collect_roots(&gc_flags, &additional_buffer);
    ...
    //(4)释放垃圾
    current = to_free.next;
    while (current != &to_free) {
        p = current->ref;
```

```
        GC_G(next_to_free) = current->next;
        if ((GC_TYPE(p) & GC_TYPE_MASK) == IS_OBJECT) {
            //调用 free_obj 释放对象
            obj->handlers->free_obj(obj);
            ...
        } else if ((GC_TYPE(p) & GC_TYPE_MASK) == IS_ARRAY) {
            //释放数组
            zend_array *arr = (zend_array*)p;

            GC_TYPE(arr) = IS_NULL;
            zend_hash_destroy(arr);
        }
        current = GC_G(next_to_free);
    }
    ...
}
```

4.3 内存池

在C语言中，我们通常直接使用malloc进行内存的分配，而频繁地分配、释放内存无疑会产生内存碎片，降低系统性能。在PHP中，变量的分配、释放非常频繁，如果所有的变量都通过malloc的方式进行分配，那么将造成严重的性能问题，作为语言级的应用，这种损耗是无法接受的。为此，PHP自己实现了一套内存池（ZendMM：Zend Memery Manager），用于替换glibc的malloc、free，以解决内存频繁分配、释放的问题。总的来说，内存池技术主要有两方面的作用：减少内存分配及释放的次数、有效控制内存碎片的产生。

PHP内存池的实现参考了tcmalloc的设计，tcmalloc是Google开源的一个非常优秀的内存分配器，尤其是在多线程应用中，它使用C++实现。除了tcmalloc，还有很多内存池的实现，比如ptmalloc、jemalloc等，有兴趣的读者可以自行查阅相关的资料。

内存池是PHP内核中最底层的内存操作，它是非常独立的一个模块，可以移植到其他C语言应用中去。内存池定义了三种粒度的内存块：chunk、page、slot，每个chunk的大小为2MB，page大小为4KB，一个chunk被切割为512个page，而一个或若干个page被切割为多个slot。申请内存时按照不同的申请大小决定具体的分配策略。

- **Huge(chunk)**：申请内存大于2MB，直接调用系统分配，分配若干个chunk。
- **Large(page)**：申请内存大于3092B（即page大小的3/4），小于2044KB（即511个page大小），分配若干个page。

- **Small(slot)**：申请内存小于等于 3092B（即 page 大小的 3/4），内存池提前定义好了 30 种同等大小的内存（8,16,24,32…3072），它们分配在不同的 page 上（不同大小的内存可能会分配在多个连续的 page），申请内存时直接在对应 page 上查找可用的 slot。

内存池通过 zend_mm_heap 结构存储内存池的主要信息，比如大内存链表、chunk 链表、slot 各大小内存链表等，这个结构通过全局变量 alloc_globals 保存，AG 宏操作的就是这个变量。

```
//file: zend_alloc.c
# define AG(v) (alloc_globals.v)
static zend_alloc_globals alloc_globals;

//file: zend_alloc.h
typedef struct _zend_mm_heap zend_mm_heap;
struct _zend_mm_heap {
#if ZEND_MM_STAT
    size_t           size; //当前已用内存数
    size_t           peak; //内存单次申请的峰值
#endif
    /* 小内存分配的可用位置链表，ZEND_MM_BINS 等于 30，即此数组表示的是各种大小内存对
应的链表头部 */
    zend_mm_free_slot *free_slot[ZEND_MM_BINS];
    ...
     //大内存链表
    zend_mm_huge_list *huge_list;
    //指向 chunk 链表头部
    zend_mm_chunk     *main_chunk;
    //缓存的 chunk 链表
    zend_mm_chunk     *cached_chunks;
    //已分配 chunk 数
    int               chunks_count;
     //当前 request 使用 chunk 峰值
    int               peak_chunks_count;
    //缓存的 chunk 数
    int               cached_chunks_count;
     /* chunk 使用均值，每次请求结束后会根据 peak_chunks_count 重新计算：
(avg_chunks_count+peak_chunks_count)/2.0 */
    double            avg_chunks_count;
}
```

大内存分配的实际是若干个 chunk，然后通过一个 zend_mm_huge_list 结构进行管理，大内存之间构成一个单向链表，如图 4-6 所示。

```
typedef struct  _zend_mm_huge_list zend_mm_huge_list;
struct _zend_mm_huge_list {
    void            *ptr;
    size_t           size;
    zend_mm_huge_list *next;
};
```

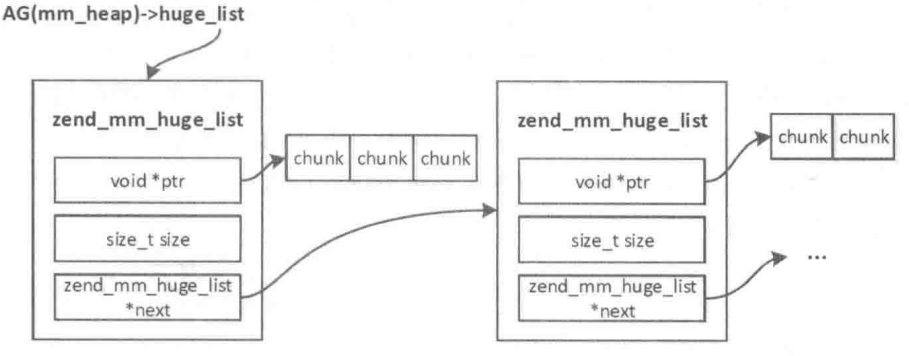


图 4-6 huge_list 链表

chunk 是内存池向系统申请、释放内存的最小粒度，chunk 之间构成双向链表，第一个 chunk 的地址保存于 zend_mm_heap->main_chunk。每个 chunk 的大小为 2MB，它被切割为 512 个 page，所以每个 page 的大小是 4KB，其中第一个 page 的内存用于 chunk 自己的结构体成员，主要记录 chunk 的一些信息，比如前后 chunk 的指针、当前 chunk 上各个 page 的使用情况等。

```
struct _zend_mm_chunk {
    zend_mm_heap     *heap;
    //指向下一个 chunk
    zend_mm_chunk    *next;
    //指向上一个 chunk
    zend_mm_chunk    *prev;
    //当前 chunk 的剩余可用 page 数
    int              free_pages;
    int              free_tail;
    int              num;
    char             reserve[64 - (sizeof(void*) * 3 + sizeof(int) * 3)];
     //heap 结构，只有主 chunk 会用到
    zend_mm_heap     heap_slot;
```

```
        //标识各 page 是否已使用的 bitmap，总大小 512bit，对应 page 总数，每个 page 占一个
bit 位
        zend_mm_page_map   free_map;
         //各 page 的信息：当前 page 使用类型(用于 large 分配还是 small)、占用的 page 数等
        zend_mm_page_info  map[ZEND_MM_PAGES];
    };
```

slot 内存是把若干个 page 按固定大小分割好的内存块，内存池定义了 30 种大小的 slot 内存：8、16、24、32…1792、2048、2560、3072。这些 slot 的大小是有规律的，最小的 slot 大小为 8byte，前 8 个 slot 依次递增 8byte，后面每隔 4 个递增值乘以 2，即 0～7 递增 8byte、8～11 递增 16byte、12～15 递增 32byte、16～19 递增 32byte、20～23 递增 128byte、24～27 递增 256byte、28～29 递增 512byte。每种大小的 slot 占用的 page 数也不相同：slot 0-15 各占 1 个 page、slot 16-29 分别占 5、3、1、1、5、3、2、2、5、3、7、4、5、3 个 page，分配各个规格的 slot 时会按照这个配置申请对应数量的 page，然后进行分割组成链表。所有 slot 的大小、数量、占用的 page 数等信息由 zend_alloc_sizes.h 文件中的 ZEND_MM_BINS_INFO 定义。相同大小的 slot 之间构成单链表，具体结构如下：

```
typedef struct  _zend_mm_free_slot zend_mm_free_slot;
struct _zend_mm_free_slot {
    zend_mm_free_slot *next_free_slot;
};
```

heap、huge、chunk、page、slot 之间的关系如图 4-7 所示。

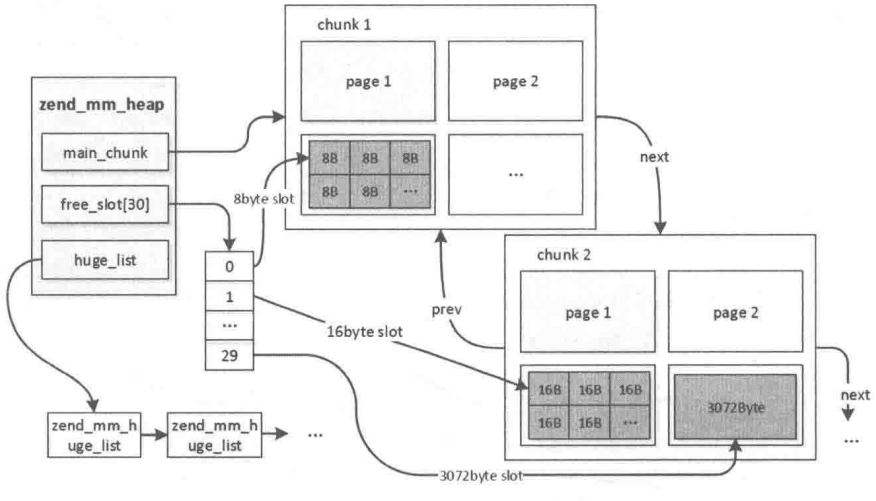


图 4-7　各类型内存结构关系

4.3.1 内存池的初始化

内存池在 php_module_startup()阶段初始化，初始化过程主要是分配 heap 结构，这个过程在 start_memory_manager()中完成，如果是多线程环境，则会为每一个线程分配一个内存池，线程之间互不影响。

```
ZEND_API void start_memory_manager(void)
{
#ifdef ZTS
    ...
#else
    //非线程安全
    alloc_globals_ctor(&alloc_globals);
#endif
}

static void alloc_globals_ctor(zend_alloc_globals *alloc_globals)
{
#ifdef MAP_HUGETLB
    //根据环境变量设定是否开启大页
    tmp = getenv("USE_ZEND_ALLOC_HUGE_PAGES");
    if (tmp && zend_atoi(tmp, 0)) {
        zend_mm_use_huge_pages = 1;
    }
#endif
    ZEND_TSRMLS_CACHE_UPDATE();
    alloc_globals->mm_heap = zend_mm_init();
}
```

初始化时会根据环境变量 USE_ZEND_ALLOC_HUGE_PAGES 设定是否开启内存大页。非线程安全环境下，会将分配的 heap 结构保存到全局变量 alloc_globals 中，也就是 AG()宏。需要注意的是，zend_mm_heap 这个结构并不是单独分配的，它嵌在 chunk 结构体中（即 heap_slot 成员）。也就是说，内存池初始化时是分配了一个 chunk 结构，zend_mm_chunk->heap_slot 作为内存池的 heap 结构，这个 chunk 也是第一个 chunk，即 main_chunk，如图 4-8 所示。

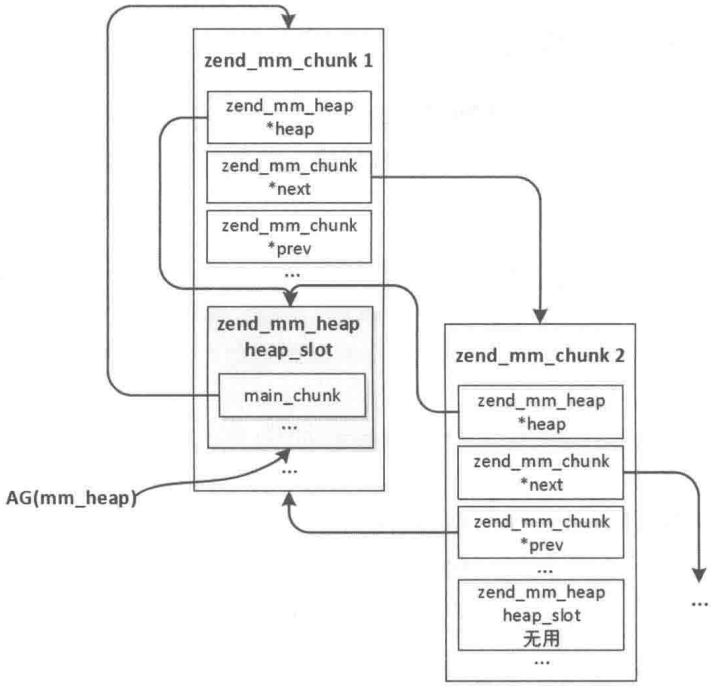


图 4-8 zend_mm_heap 结构的分配

内存池的 heap 结构为什么要嵌在 chunk 中而不是单独分配呢？这是因为每个 chunk 的第一个 page 始终是给 chunk 结构体自己使用的，剩下的 511 个 page 才被用作内存分配，但是 chunk 结构体并不需要一个 page 那么大的内存。也就是说，被占用的这个 page 会有剩余的空间，因此，为了尽可能利用空间，就把 heap 结构嵌在了 chunk 中，这种做法与 zval 结构体中 u2 的起源非常相像。具体的分配过程在 zend_mm_init()中实现：

```
static zend_mm_heap *zend_mm_init(void)
{
    //向系统申请 2MB 大小的 chunk
    zend_mm_chunk *chunk = (zend_mm_chunk*)zend_mm_chunk_alloc_int(ZEND_
MM_CHUNK_SIZE, ZEND_MM_CHUNK_SIZE);
    zend_mm_heap *heap;
    //heap 结构实际是主 chunk 嵌入的一个结构，后面再分配 chunk 的 heap_slot 不再使用
    heap = &chunk->heap_slot;
    ...
    //剩余可用 page 数
    chunk->free_pages = ZEND_MM_PAGES - ZEND_MM_FIRST_PAGE;
    chunk->free_tail = ZEND_MM_FIRST_PAGE;
```

```
    chunk->num = 0;
    //将第一个 page 的 bit 分配标识位设置为 1，表示已经被分配、占用
    chunk->free_map[0] = (Z_L(1) << ZEND_MM_FIRST_PAGE) - 1;
    //第一个 page 的类型为 ZEND_MM_IS_LRUN，即 large 内存
    chunk->map[0] = ZEND_MM_LRUN(ZEND_MM_FIRST_PAGE);
    heap->main_chunk = chunk; //指向主 chunk
    //初始化剩下的一些成员
    ...
    heap->huge_list = NULL; //huge 内存链表
    return heap;
}
```

4.3.2 内存分配

接下来我们详细分析三种粒度内存的分配过程，其中 Huge 大内存的分配过程比较简单，而 Large 与 Small 内存分配都涉及 page 的查找操作，过程稍显复杂。需要使用 emalloc()、emalloc_large()、emalloc_huge()、ecalloc()、erealloc()等方法来向内存池申请内存。以 emalloc()为例，申请时，内存池会按照申请内存的大小自动选择哪种内存进行分配，如图 4-9 所示。

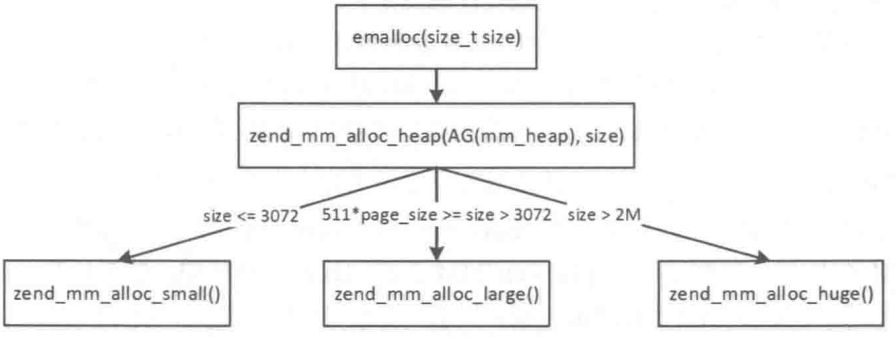


图 4-9 内存分配流程

4.3.2.1 Huge 分配

Huge 是指超过 2MB 大小内存的分配，实际分配时将对齐到 *n* 个 chunk，分配完以后还会分配一个 zend_mm_huge_list 结构，这个结构用于管理所有的 Huge 内存。

```
    static void *zend_mm_alloc_huge(zend_mm_heap *heap, size_t size ZEND_FILE_
LINE_DC ZEND_FILE_LINE_ORIG_DC)
    {
```

```
    //按页大小重置实际要分配的内存
    size_t new_size = ZEND_MM_ALIGNED_SIZE_EX(size, REAL_PAGE_SIZE);

#if ZEND_MM_LIMIT
    //如果有内存使用限制则 check 是否已达上限，达到的话进行 zend_mm_gc 清理后再检查
    //此过程不再展开分析
#endif
    //分配 chunk
    ptr = zend_mm_chunk_alloc(heap, new_size, ZEND_MM_CHUNK_SIZE);
    ...
    //将申请的内存通过 zend_mm_huge_list 插入到链表中
    zend_mm_add_huge_block(heap, ptr, new_size, ...);
    ...
    return ptr;
}
```

除 Huge 分配外，分配 chunk 内存也是 Large、Small 内存分配的基础，它是 ZendMM 向系统申请内存的唯一粒度。在分配 chunk 时有一个非常关键的操作，那就是将内存地址对齐到 chunk 的大小 2MB（即 ZEND_MM_CHUNK_SIZE）。也就是说，分配的 chunk 地址都是 ZEND_MM_CHUNK_SIZE 的整数倍。需要注意的是，这个对齐并不是简单地由系统完成的，而需要内存池在申请内存后自己进行调整。系统返回的地址是随机的，并不一定是 ZEND_MM_CHUNK_SIZE 的倍数，内存池要做的处理是自己移动到对齐的位置上，比如返回的地址 ptr 是 2000，而最近的一个对齐地址是 2048，那么内存池就会把 ptr 移到 2048，从这个位置使用。

ZendMM 处理对齐的方法是：先按实际要申请的内存大小申请一次，如果系统分配的地址恰好是 ZEND_MM_CHUNK_SIZE 的整数倍，那么就不需要调整了，直接返回使用；如果不是 ZEND_MM_CHUNK_SIZE 的整数倍，ZendMM 会把这块内存释放掉，然后按照“实际要申请的内存大小 +ZEND_MM_CHUNK_SIZE”的大小重新申请一块内存，多申请的 ZEND_MM_CHUNK_SIZE 大小的内存是用来调整的，ZendMM 会从系统分配的地址向后偏移到最近一个 ZEND_MM_CHUNK_SIZE 的整数倍位置，调整完以后再把多余的内存释放掉。

```
static void *zend_mm_chunk_alloc_int(size_t size, size_t alignment)
{
    //向系统申请 size 大小的内存
    void *ptr = zend_mm_mmap(size);
    if (ptr == NULL) {
        return NULL;
```

```
    } else if (ZEND_MM_ALIGNED_OFFSET(ptr, alignment) == 0) {
        //判断申请的内存是否为 alignment 的整数倍，是的话直接返回
        return ptr;
    }else{
        //申请的内存不是按照 alignment 对齐的
        size_t offset;
        //将申请的内存释放掉重新申请
        zend_mm_munmap(ptr, size);
        //重新申请一块内存，这里会多申请一块内存，用于截取到 alignment 的整数倍，可以忽略 REAL_PAGE_SIZE
        ptr = zend_mm_mmap(size + alignment - REAL_PAGE_SIZE);
        //offset 为 ptr 距离上一个 alignment 对齐内存位置的大小，注意不能往前移，因为前面的内存都是分配了的
        offset = ZEND_MM_ALIGNED_OFFSET(ptr, alignment);
        if (offset != 0) {
            offset = alignment - offset;
            zend_mm_munmap(ptr, offset);
            //偏移 ptr，对齐到 alignment
            ptr = (char*)ptr + offset;
            alignment -= offset;
        }
        if (alignment > REAL_PAGE_SIZE) {
            zend_mm_munmap((char*)ptr + size, alignment - REAL_PAGE_SIZE);
        }
        return ptr;
    }
}
```

这个过程中用到了一个宏：ZEND_MM_ALIGNED_OFFSET()，这个宏的作用是计算按 alignment 对齐的内存地址距离上一个 alignment 整数倍内存地址的大小，也就是 offset 偏移量。alignment 必须为 2 的 *n* 次方，比如一段 *n**alignment 大小的内存，ptr 为其中一个位置，那么就可以通过位运算计算得到 ptr 在所属 alignment 内存块中的 offset，如图 4-10 所示。

```
#define ZEND_MM_ALIGNED_OFFSET(size, alignment) \
    (((size_t)(size)) & ((alignment) - 1))
```

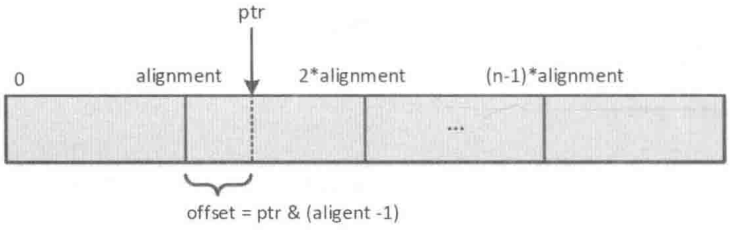


图 4-10 相对对齐值的内存 offset

这个位运算是因为 alignment 为 2^n，所以可以通过 alignment 取到最低位的位置，也就是相对上一个整数倍 alignment 的 offset，如果不用运算的话也可以通过 offset = ptr -（ptr/alignment 取整）*alignment 计算得到，这个更容易理解些，但是不如位运算效率高。

4.3.2.2 Large 分配

当申请的内存大于 3072B、小于 2044KB 时，内存池会选择在 chunk 上查找对应数量的 page 返回。Large 内存申请的粒度是 page，也就是分配 *n* 页连续的 page，所以 Large 分配的过程就转化为在 chunk 上查找 *n* 页连续可用的 page 的过程。

```
static zend_always_inline void *zend_mm_alloc_large(zend_mm_heap *heap,
size_t size ZEND_FILE_LINE_DC ZEND_FILE_LINE_ORIG_DC)
{
    //根据 size 大小计算需要分配多少个 page
    int pages_count = (int)ZEND_MM_SIZE_TO_NUM(size, ZEND_MM_PAGE_SIZE);
    //分配 pages_count 个 page
    void *ptr = zend_mm_alloc_pages(heap, pages_count, ...);
    ...
    return ptr;
}
```

chunk 结构中有两个成员用于记录 page 的分配信息。

- **free_map**：类型为 zend_mm_page_map，实际上就是 zend_ulong free_map[16/8]，这是一个 bitmap，总大小为 64byte，也就是 512bit，它的作用是记录当前 chunk 上 512 个 page 是否已分配，512 个 page 对应 512bit，每个 page 各占一个 bit 位，0 表示未分配，1 表示已分配。以 64 位系统为例，free_map 为 8 个长整型的数组：{0, 0, 0, 0, 0, 0, 0, 0}，分别表示 page 0～63、64～127、128～191、192～255、256～319、320～383、384～447、448～511 的分配与否，比如当前 chunk 的 page 0、page 1 已经分配，则 free_map[0] = 3，即 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000011，如图 4-11 所示。

- **map**：这是一个可容纳 512 个类型为 uint32_t 元素的数组，该数组用于记录各 page 的分配类型及分配的 page 页数，每个 page 对应一个数组成员。Large 内存、Small 内存都会占用 page，正是通过这个数组标识该 page 属于哪种类型的。除了标识 page 的分配类型，还用来记录分配的 page 页数，比如申请分配了 3 个 page，那么就会把 3 记录在起始 page 的 map 中。两者通过 bit 位区分开，最高的两位用于标识 page 的分配类型：01 为 Large（即 0x40000000）、10 为 Small（即 0x80000000），剩余位用于 page 数。比如申请 12KB 的内存（即 3 个 page），内存池分配了 page 1、2、3，则 map[1] = 0x40000000 | 3，如图 4-12 所示。

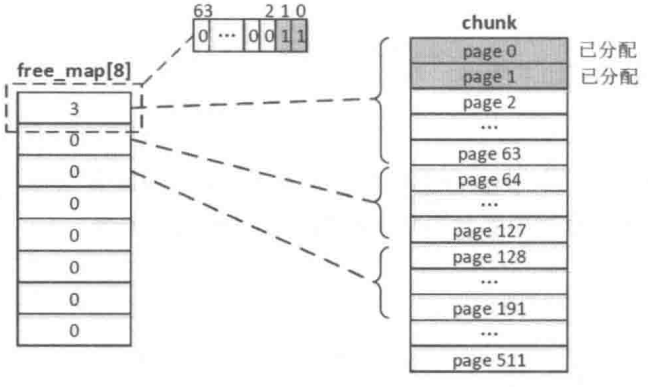


图 4-11　free_map

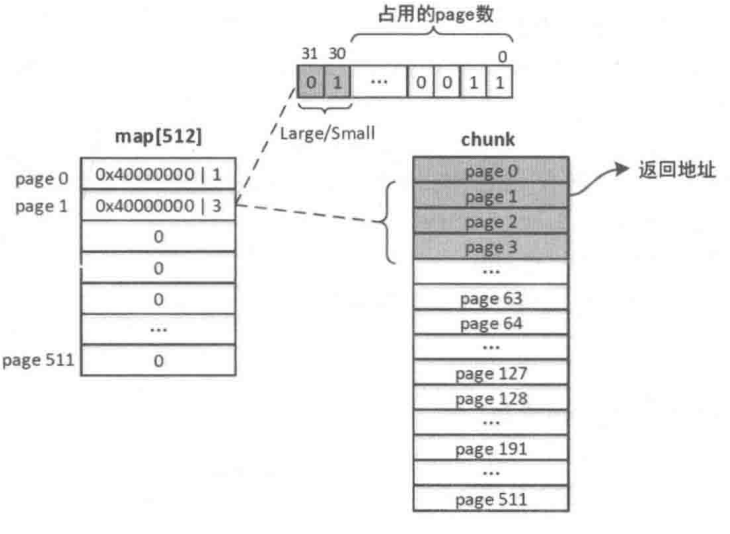


图 4-12　page 的分配类型及页数

page 分配时从第一个 chunk 开始遍历，依次查找各 chunk 是否有满足要求的 page，如果当

前 chunk 没有合适的则进入下一 chunk，如果直到最后都没有找到，则新分配一个 chunk。需要注意的是，查找的过程并不仅仅是 page 页数够了即可，这里有一个准则："申请的 page 页数要尽可能地填满 chunk 的空隙"，也就是说尽可能地与分配了的 page 连在一起，避免中间出现 page 空隙。这样做的目的是为了减少后续分配时的查找次数，提高内存利用率。page 排列越紧凑，那么查找过程的效率就越高，如果未分配的 page 凌乱地分布在 chunk 上，那么查找时就需要花费更多的时间检查这些 page 是否够用。比如未分配 page 与已分配 page 相间分布，如果想再分配 2 页 page，这种情况下就会导致查找了整个 chunk 都没有合适的，因为没有 2 个连续的 page，尽管 chunk 上有大量空闲的 page，而如果这些已分配的 page 全部挨在一起，则只需要查找一次就可以了，在提高查找效率的同时也充分利用了内存空间。这种做法实际上是避免在 chunk 上产生内存碎片。

最优 page 的检索过程如下所述。

step1：

首先从第一个 page 分组（page 0~63）开始检查，如果当前分组无空闲 page（即 free_map[x] = -1）则进入下一分组，直到当前分组有空闲 page，然后进入 step2。

```
//zend_mm_alloc_pages:
int best = -1;
int best_len = ZEND_MM_PAGES;
int free_tail = chunk->free_tail;
zend_mm_bitset *bitset = chunk->free_map;
//将 free_map 复制一份进行操作
zend_mm_bitset tmp = *(bitset++); // zend_mm_bitset tmp = *bitset; bitset++
int i = 0;

  //-1 表示当前 chunk 所有 page 都已分配 11111111 11111111 11111111 11111111
  while (tmp == (zend_mm_bitset)-1) {
    i += ZEND_MM_BITSET_LEN;
    if (i == ZEND_MM_PAGES) { //最后一页
        ...
    }
    //继续检查下一个 page 分组
    tmp = *(bitset++);
}
```

step2：

当前分组有可用 page，首先检查当前 page 分组的 bit 位，找到第一个空闲 page 的位置，记

作 page_num，接着继续向下查找空闲 page，直到遇到第一个已经分配的 page 为止，将最后一个空闲 page 位置记作 end_page_num。注意：查找 end_page_num 时并不局限在当前 page 分组内，会一直向下查找，直到最后一页。查找过程主要依据 free_map，但是这个过程并没有直接遍历各个 bit 位，而是用到了很多不太容易理解的位运算。继续看一下查找过程：

```
//zend_mm_alloc_pages:
//tmp 为当前 page 分组的 bit 位，i 为当前分组第 1 个 page 的页码
//找到第一个空闲 page
page_num = i + zend_mm_bitset_nts(tmp);
tmp &= tmp + 1;
//快速跳过剩余 page 全部可用的分组
while (tmp == 0) {
    i += ZEND_MM_BITSET_LEN;

    if (i >= free_tail || i == ZEND_MM_PAGES) {
        ...
    }
    //当前分组剩下的 page 都是可用的，直接跳到下一分组
    tmp = *(bitset++);
}
//找到第一个已分配 page
len = i + zend_mm_bitset_ntz(tmp) - page_num;
//len 为找到的可用 page 页数
if (len >= pages_count) {
    if (len == pages_count) {
        goto found;
    } else if (len < best_len) {
        best_len = len;
        best = page_num;
    }
}
//把找到的这些 page 标为已分配，注意：此时 tmp 已经过 tmp &= tmp + 1 处理
tmp |= tmp - 1;
```

page_num 至 end_page_num 为找到的可用 page，接着判断找到的 page 页数是否够用，如果不够则把 page_num 至 end_page_num 这些 page 的 bit 位标为 1，也就是已分配，然后回到 step1 继续检索其他 page 分组；如果 page 页数恰好是要申请的页数，那么直接使用，中断检索；如

果找到的 page 页数比申请的页数大则表示可用，但是不是最优的，这个时候会把 page_num 暂存起来，接着回到 step1 继续向后找别的空闲 page，最后比较选择 best_len 最小的，即能够最大程度填满 page 间隔的。

举个例子，当前某个 chunk 的 page 分配情况如图 4-13 中的 A：page 0、1、2、6、9、10 已经分配占用，接下来要申请 2 页 page，首先会找到可用的 page 3、4、5，但是空间比申请页数的多一页，然后把其位置保存到 best，以及页数 best_len，如果后面找不到更合适的就会选用这个，接着会把 page 3、4、5 标为已分配（这里更改的是复制出来的 free_map，并不会影响实际的值），如图 4-13 中的 B。接着继续向下查找，找到可用 page 7、8，其空间正好等于申请的页数，比 page 3 合适，因此最后返回 page 7 的地址，并更新 page 7、8 的实际 free_map。

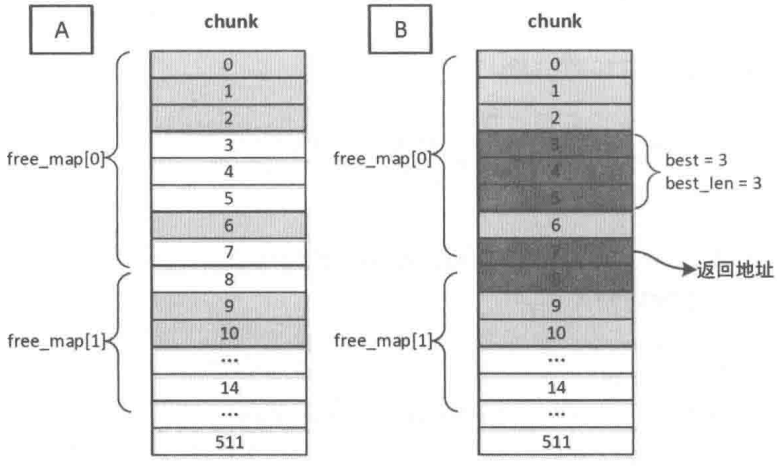


图 4-13　page 的查找过程

上面的过程在查找 end_page_num 时有一步特殊操作：tmp &= tmp + 1，如果 tmp 为 0，表示该 page 分组剩余的 page 全部是空闲的，这个时候直接跳到下一个 page 分组判断。比如 tmp=3，表示前两个 page 已经分配，bit 位：…00000011，tmp + 1 后 bit 位：…00000100，则：…00000011 & …00000100 = 0。

step3：

最后，找到合适的 page 页后设置对应 page 的分配信息，即 free_map、map，然后返回找到的第一页 page 的地址。

```
//zend_mm_alloc_pages:
found:
    chunk->free_pages -= pages_count;
    zend_mm_bitset_set_range(chunk->free_map, page_num, pages_count);
```

```
    chunk->map[page_num] = ZEND_MM_LRUN(pages_count);
    if (page_num == chunk->free_tail) {
        chunk->free_tail = page_num + pages_count;
    }
    return ZEND_MM_PAGE_ADDR(chunk, page_num);
```

4.3.2.3　Small 分配

Small 内存在分配时，首先检查申请规格的内存是否已经分配，如果没有分配或者分配的已经用完了，则申请相应页数的 page，page 的分配过程与 Large 分配完全一致，申请到 page 以后按固定大小将 page 切割为 slot，slot 之间构成单链表，链表头部保存至 AG(mm_heap)->free_slot；如果对应的 slot 已经已经分配，则直接返回 AG(mm_heap)->free_slot。比如 16byte、3072byte 大小的 slot，将分别申请 1 个、3 个 page，然后切割为 256 个 16byte 的 slot，以及 4 个 3072byte 的 slot，如图 4-14 所示。

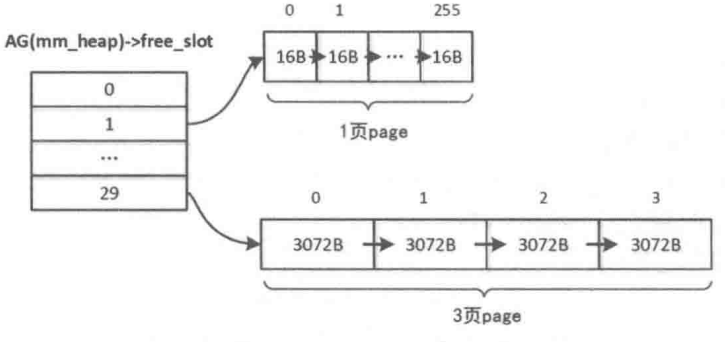


图 4-14　slot[1]与 slot[29]链表

Small 内存的具体的分配过程：

```
    static zend_always_inline void *zend_mm_alloc_small(zend_mm_heap *heap,
size_t size, int bin_num ZEND_FIL
    E_LINE_DC ZEND_FILE_LINE_ORIG_DC)
    {
        ...
        if (EXPECTED(heap->free_slot[bin_num] != NULL)) {
            //有可用 slot
            zend_mm_free_slot *p = heap->free_slot[bin_num];
            heap->free_slot[bin_num] = p->next_free_slot;
            return (void*)p;
        } else {
            //新分配 slot
```

```
            return zend_mm_alloc_small_slow(heap, bin_num ZEND_FILE_LINE_RELAY_CC
ZEND_FILE_LINE_ORIG_RELAY_CC
    );
        }
    }
```

zend_mm_alloc_small_slow()为分配 slot 的操作，过程比较清晰：

```
    //bin_num 为 free_slot 数组下
    static zend_never_inline void *zend_mm_alloc_small_slow(zend_mm_heap *heap,
int bin_num ZEND_FILE_LINE_DC
    ZEND_FILE_LINE_ORIG_DC)
    {
        zend_mm_chunk *chunk;
        int page_num;
        zend_mm_bin *bin;
        zend_mm_free_slot *p, *end;
        //分配固定数量的 page
        bin = (zend_mm_bin*)zend_mm_alloc_pages(heap, bin_pages[bin_num]
ZEND_FILE_LINE_RELAY_CC ZEND_FILE_LI
    NE_ORIG_RELAY_CC);
        chunk = (zend_mm_chunk*)ZEND_MM_ALIGNED_BASE(bin, ZEND_MM_CHUNK_SIZE);
        page_num = ZEND_MM_ALIGNED_OFFSET(bin, ZEND_MM_CHUNK_SIZE) / ZEND_MM_
PAGE_SIZE;
        //设置各页 page 的分配类型
        chunk->map[page_num] = ZEND_MM_SRUN(bin_num);
        if (bin_pages[bin_num] > 1) {
            int i = 1;
            do {
                chunk->map[page_num+i] = ZEND_MM_NRUN(bin_num, i);
                i++;
            } while (i < bin_pages[bin_num]);
        }
        //创建 slot 链表
        end = (zend_mm_free_slot*)((char*)bin + (bin_data_size[bin_num] * (bin_
elements[bin_num] - 1)));
        //heap->free_slot[bin_num]指向第 2 个 slot，第 1 个为当前申请到的 slot
        heap->free_slot[bin_num] = p = (zend_mm_free_slot*)((char*)bin + bin_
```

```
data_size[bin_num]);
    do {
        p->next_free_slot = (zend_mm_free_slot*)((char*)p + bin_data_size
[bin_num]);
        p = (zend_mm_free_slot*)((char*)p + bin_data_size[bin_num]);
    } while (p != end);
    p->next_free_slot = NULL;
    //返回 slot 链表的第一个元素
    return (char*)bin;
}
```

4.3.3 系统内存分配

前面介绍了三种内存分配的过程，实际上内存池只是在系统内存上面做了一些额外工作，从而减少系统内存的分配、释放次数。内存池向系统申请内存的最小粒度是 chunk，通过 mmap() 来申请，开启 HugePage 支持项也是在这个地方用到的。

```
static void *zend_mm_mmap(size_t size)
{
    ...
//hugepage 支持
#ifdef MAP_HUGETLB
    if (zend_mm_use_huge_pages && size == ZEND_MM_CHUNK_SIZE) {
        ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANON | MAP_HUGETLB, -1, 0);
        if (ptr != MAP_FAILED) {
            return ptr;
        }
    }
#endif

    ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON,
-1, 0);
    if (ptr == MAP_FAILED) {
        return NULL;
    }
    return ptr;
}
```

> 关于 Hugepage：简单地讲就是默认的内存是以 4KB 分页的，而虚拟地址和内存地址是需要转换的，而这个转换是要查表的，CPU 为了加速这个查表过程都会内建 TLB(Translation Lookaside Buffer)。显而易见，如果虚拟页越小，表里的条目数也就越多，而 TLB 大小是有限的，条目数越多 TLB 的 Cache Miss 也就会越高，所以如果我们能启用大内存页就能间接降低这个 TLB Cache Miss。

4.3.4 内存释放

内存释放主要通过 efree()来完成，内存池会根据释放的内存地址自动判断属于哪种粒度的内存，从而执行不同的释放逻辑。另外也可以直接使用对应类型的内存释放函数：

```
    #define efree(ptr)          _efree((ptr) ZEND_FILE_LINE_CC ZEND_FILE_LINE_
EMPTY_CC)
    #define efree_large(ptr)    _efree_large((ptr) ZEND_FILE_LINE_CC ZEND_FILE_
LINE_EMPTY_CC)
    #define efree_huge(ptr)     _efree_huge((ptr) ZEND_FILE_LINE_CC ZEND_FILE_
LINE_EMPTY_CC)
```

释放内存时只根据一个内存地址来完成，即：efree(ptr)，那么内存池是如何只根据一个地址就判断出该地址属于哪种内存类型的呢？这是因为 chunk 分配时是按照 ZEND_MM_CHUNK_SIZE（即 2MB）对齐的，也就是 chunk 的起始内存地址一定是 ZEND_MM_CHUNK_SIZE 的整数倍，所以可以根据 chunk 上的任意位置知道 chunk 的起始位置与所在 page。

1）Huge 内存的释放

首先，根据释放地址 ptr 计算该地址相对 chunk 起始位置的内存偏移量，这个值通过宏 ZEND_MM_ALIGNED_OFFSET() 得到，这里是通过位运算计算得到的，比如 ptr = 0x7ffff7c01000，计算得到 offset = 4096。除了位运算还可以这样计算：offset = ptr - (ptr/ZEND_MM_CHUNK_SIZE)*ZEND_MM_CHUNK_SIZE，offset 表示该内存地址距离 chunk 起始位置为 4096 字节。

```
size_t page_offset = ZEND_MM_ALIGNED_OFFSET(ptr, ZEND_MM_CHUNK_SIZE);
```

前面已经介绍过，只有 Huge 内存能够完全使用 chunk，也就是 Huge 内存地址相对 chunk 的 offset 一定等于 0，而 Large、Small 内存因为 chunk 的第 1 个 page 被占用了，所以这两种内存的 offset 不可能为 0。内存池根据 offset 值判断出释放的内存是否为 Huge 类型，如果是则将

占用的 chunk 释放，同时从 AG(mm_heap)->huge_list 链表中删除。

```
    if (UNEXPECTED(page_offset == 0)) {
        if (ptr != NULL) {
            //释放 huge 内存，从 huge_list 中删除
            zend_mm_free_huge(heap, ptr ZEND_FILE_LINE_RELAY_CC ZEND_FILE_LINE_
ORIG_RELAY_CC);
        }
    }
```

Huge 的具体删除过程：

```
    static void zend_mm_free_huge(zend_mm_heap *heap, void *ptr ZEND_FILE_LINE_DC
ZEND_FILE_LINE_ORIG_DC)
    {
        size_t size;

        ZEND_MM_CHECK(ZEND_MM_ALIGNED_OFFSET(ptr, ZEND_MM_CHUNK_SIZE) == 0,
"zend_mm_heap corrupted");
        //将释放的 huge 内存从链表中删除
        size = zend_mm_del_huge_block(heap, ptr ZEND_FILE_LINE_RELAY_CC
ZEND_FILE_LINE_ORIG_RELAY_CC);
        //释放 chunk
        zend_mm_chunk_free(heap, ptr, size);
        ...
    }
```

2）Large 内存的释放

如果计算得到的 offset 不等于 0，则表示该地址是 Large 内存或者 Small 内存，然后根据 offset 值进一步计算出属于第几个 page。这个计算就比较简单了，直接根据 offset 除 page 的大小取整即可，还是上面那个例子，这里计算得到 page_num = 1。得到 page 页码后就可以从 chunk->map 中获取该 page 的分配类型，这个时候就可以完全确定释放地址属于哪种粒度的内存了。

```
    //zend_mm_free_heap:
    zend_mm_chunk *chunk = (zend_mm_chunk*)ZEND_MM_ALIGNED_BASE(ptr, ZEND_MM_
CHUNK_SIZE);
    //计算所属 page
```

```
    int page_num = (int)(page_offset / ZEND_MM_PAGE_SIZE);
    //获取 page 的分配类型及页数
    zend_mm_page_info info = chunk->map[page_num];
    if (EXPECTED(info & ZEND_MM_IS_SRUN)) {
        //Small 内存
    } else {
        //Large 内存
        int pages_count = ZEND_MM_LRUN_PAGES(info);

        ZEND_MM_CHECK(ZEND_MM_ALIGNED_OFFSET(page_offset,  ZEND_MM_PAGE_SIZE)
== 0, "zend_mm_heap corrupted");
        zend_mm_free_large(heap, chunk, page_num, pages_count);
    }
```

如果是 Large 内存，并不会直接释放物理内存，只是将对应 page 的分配信息重新设置为未分配。如果释放 page 后，发现当前 chunk 下所有 page 都是未分配的，则会释放 chunk，释放时优先选择把 chunk 移到 AG(mm_heap)->cached_chunks 缓存队列中，缓存数达到一定值后就不再继续缓存新加入的 chunk，将内存归还系统，避免占用过多的内存资源。在分配 chunk 时，如果发现 cached_chunks 中有缓存的 chunk，则直接取出使用，不再向系统申请。

```
    //page 的释放
    static zend_always_inline void zend_mm_free_pages_ex(zend_mm_heap *heap,
zend_mm_chunk *chunk, int page_num, int pages_count, int free_chunk)
    {
        chunk->free_pages += pages_count;
        //将对应 page 的分配 bit 位设置为 0：未分配
        zend_mm_bitset_reset_range(chunk->free_map, page_num, pages_count);
        chunk->map[page_num] = 0;
        if (chunk->free_tail == page_num + pages_count) {
            /* this setting may be not accurate */
            chunk->free_tail = page_num;
        }
        //如果当前 chunk 所有 page 都是未分配的则释放 chunk
        if (free_chunk && chunk->free_pages == ZEND_MM_PAGES - ZEND_MM_FIRST_
PAGE) {
            zend_mm_delete_chunk(heap, chunk);
        }
    }
```

zend_mm_delete_chunk()完成缓存或者释放 chunk 的操作，cached_chunks 会根据每次 request 请求计算的 chunk 使用均值以保证其维持在一定范围内，具体的过程不再展开。

3）Small 内存的释放

如果根据获取的 page 分配类型发现释放的地址为 Small 内存，则会将释放的 slot 插入到该规格 slot 可用链表的头部，如图 4-15 所示。

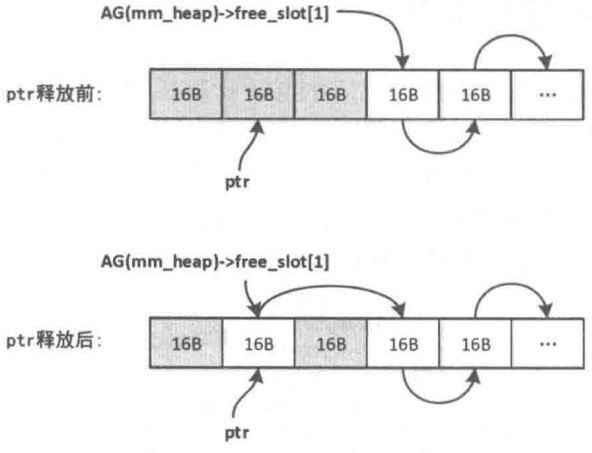


图 4-15　释放 slot

```
    static zend_always_inline void zend_mm_free_small(zend_mm_heap *heap, void
*ptr, int bin_num)
    {
        zend_mm_free_slot *p;
        ...
        //将释放的 slot 插入 free_slot 头部
        p = (zend_mm_free_slot*)ptr;
        p->next_free_slot = heap->free_slot[bin_num];
        heap->free_slot[bin_num] = p;
    }
```

4.4　线程安全

在单线程环境中，我们经常使用全局变量实现多个函数间共享数据，声明在函数之外的变量为全局变量，全局变量为各线程共享，不同的线程引用同一地址空间，如果一个线程修改了全局变量就会影响所有的线程。线程安全指的就是多线程环境下如何安全地获取公共资源。

PHP 的 SAPI 多数是单线程环境，比如 Cli、Fpm、Cgi，每个进程只启动一个主线程，这种模式下是不存在线程安全问题的，但是也有多线程的环境，比如 Apache，或用户自己嵌入 PHP 实现的环境，这种情况下就需要考虑线程安全的问题了，因为 PHP 中使用了很多全局变量，经常使用的 EG、CG 等宏就是用来获取公共资源的。在多线程环境下使用全局变量，将会引起线程之间的冲突，因此，PHP 实现了一个线程安全资源管理器（Thread Safe Resource Mananger，TSRM），用于解决多线程环境下公共资源冲突的问题，实现线程之间安全的操作公共资源。

4.4.1 TSRM 的基本实现

TSRM 相关的实现代码位于 PHP 源码的/TSRM 目录下，它的实现思路比较简单：既然共用资源这么困难，那么就干脆不共用，各线程不再共享同一份全局变量，而是各复制一份，使用数据时各线程各取自己的副本，互不干扰。比如，有个整型的公共资源 global_num，在单线程环境下直接分配一个全局变量即可，但是在多线程环境下就需要进行分离，假如有 3 个线程，那么就会分配三份 global_num，线程之间互不干扰，如图 4-16 所示。

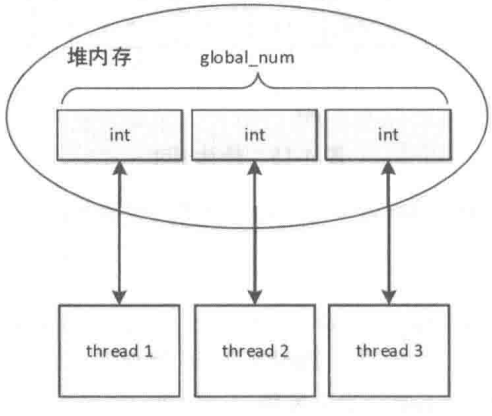


图 4-16 公共资源线程隔离

TSRM 的核心思想就是为不同的线程分配独立的内存空间，如果一个资源会被多线程使用，那么就需要预先向 TSRM 注册资源，TSRM 会为这个资源分配一个唯一的 id，并把这种资源的大小、初始化函数等保存到一个 tsrm_resource_type 结构中，各线程只能通过 TSRM 分配的那个 id 访问这个资源。当线程拿着资源 id 向 TSRM 获取资源时，TSRM 如果发现是第一次请求，则会根据资源注册时指定的资源大小分配一块内存，然后调用初始化函数进行初始化，并把这块资源保存下来供这个线程后续使用。

TSRM 为每个线程分配一个 tsrm_tls_entry 结构，该结构用于保存所有的公共资源。所有线程的 tsrm_tls_entry 结构保存在 tsrm_tls_table 数组中，这是个全局变量，操作这个变量时需要加

锁。tsrm_tls_entry 在 TSRM 初始化时按照预设置的线程数分配，每个线程的 tsrm_tls_entry 结构在这个数组中的位置是根据线程 id 与预设置的线程数（tsrm_tls_table_size）取模得到的。也就是说，有可能多个线程保存在 tsrm_tls_table 的同一位置，所以 tsrm_tls_entry 是个链表。线程在查找自己的 tsrm_tls_entry 时，首先根据 thread_id % tsrm_tls_table_size 得到一个 tsrm_tls_entry，然后需要遍历链表比较 thread_id 确定是否是当前线程的。比如 tsrm_tls_table 的大小设置为 2，现在有 3 个 thread，则存储结构如图 4-17 所示。

```
typedef struct _tsrm_tls_entry tsrm_tls_entry;
struct _tsrm_tls_entry {
    void **storage; //公共资源数组
    int count; //公共资源数，即 storage 数组大小
    THREAD_T thread_id; //所属线程 id
    tsrm_tls_entry *next; //线程之间构成链表
};
```

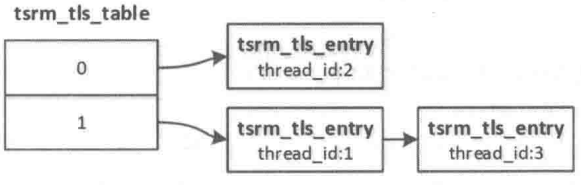


图 4-17　tsrm_tls_table

公共资源使用前必须向 TSRM 注册，注册时需要提供这么几个信息：资源的大小、资源初始化函数、资源清理函数。TSRM 会为注册的资源分配一个 tsrm_resource_type 结构来保存这些信息，当有新线程创建时，就会根据这些信息进行资源分配与初始化，所有资源的 tsrm_resource_type 结构保存在 resource_types_table 数组中，该数组在资源注册时会进行扩容。同时，注册后 TSRM 会为资源分配一个唯一的 id，这个资源 id 保存到全局变量即可，各线程根据同一个资源 id 向 TSRM 获取各自线程的资源。比如当前注册了两个资源：global_array_id、global_num_id，资源类型分别为 zend_array、uint32_t，则对应的 resource_types_table 如图 4-18 所示。TSRM 只会根据 size 大小分配对应的内存，但是并不清楚这块内存具体存储什么类型，所以需要在注册时指定初始化函数 ctor，TSRM 在分配完内存后会调用该函数进行内存的初始化。

```
typedef struct {
    size_t size; //资源的大小
    ts_allocate_ctor ctor; //初始化函数
    ts_allocate_dtor dtor;
    int done;
```

```
} tsrm_resource_type;
```

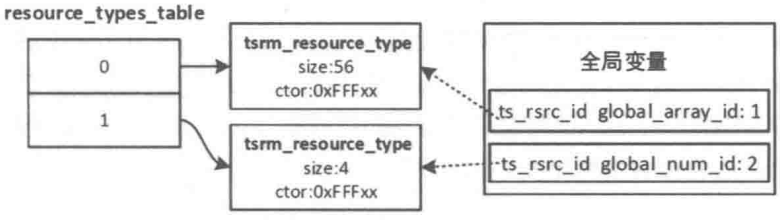


图 4-18 resource_types_table

4.4.1.1 TSRM 初始化

在使用 TSRM 之前需要主动开启，这个步骤在 SAPI 启动时完成，主要工作是分配 tsrm_tls_table、resource_types_table 内存，以及创建线程互斥锁。

```
TSRM_API int tsrm_startup(int expected_threads, int expected_resources, int debug_level, char *debug_filename)
{
    pthread_key_create( &tls_key, 0 );
    //分配 tsrm_tls_table
    tsrm_tls_table_size = expected_threads;
    tsrm_tls_table = (tsrm_tls_entry **) calloc(tsrm_tls_table_size, sizeof(tsrm_tls_entry *));
    ...
    //初始化资源的递增 id，用于资源 id 的分配
    id_count=0;
    //分配资源类型数组：resource_types_table
    resource_types_table_size = expected_resources;
    resource_types_table = (tsrm_resource_type *) calloc(resource_types_table_size, sizeof(tsrm_resource_type));
    ...
    //创建锁
    tsmm_mutex = tsrm_mutex_alloc();
}
```

4.4.1.2 注册资源

TSRM 初始化后各模块就可以进行资源注册了，注册后 TSRM 会给资源分配一个唯一的资源 id，之后对此资源的操作只能依据此 id，接下来我们以 zend_executor_globals 这个全局符号表为例看一下其注册过程。

```
//file:zend_globals.h
#ifdef ZTS
ZEND_API int executor_globals_id;
#endif

//file:zend.c
int zend_startup(zend_utility_functions *utility_functions, char **extensions)
{
    ...
#ifdef ZTS
    //注册资源
    ts_allocate_id(&executor_globals_id, sizeof(zend_executor_globals),
(ts_allocate_ctor) executor_globals_ctor, (ts_allocate_dtor) executor_
globals_dtor);
    ...
#endif
}
```

资源通过 ts_allocate_id()完成注册：

```
TSRM_API ts_rsrc_id ts_allocate_id(ts_rsrc_id *rsrc_id, size_t size,
ts_allocate_ctor ctor, ts_allocate_dtor dtor)
{
    //加锁，保证各线程原子执行此函数
    tsrm_mutex_lock(tsmm_mutex);
    //分配资源 id，即 id_count 当前值，然后把 id_count 加 1
    *rsrc_id = TSRM_SHUFFLE_RSRC_ID(id_count++);
    //检查 resource_types_table 数组当前大小是否已满
    if (resource_types_table_size < id_count) {
        //需要对 resource_types_table 扩容
        resource_types_table = (tsrm_resource_type *) realloc(resource_
types_table, sizeof(tsrm_resource_type)*id_count);
        ...
        //把数组大小修改为新的大小
        resource_types_table_size = id_count;
    }
    //将新注册的资源插入 resource_types_table 数组，下标就是分配的资源 id
```

```
    resource_types_table[TSRM_UNSHUFFLE_RSRC_ID(*rsrc_id)].size = size;
    resource_types_table[TSRM_UNSHUFFLE_RSRC_ID(*rsrc_id)].ctor = ctor;
    resource_types_table[TSRM_UNSHUFFLE_RSRC_ID(*rsrc_id)].dtor = dtor;
    resource_types_table[TSRM_UNSHUFFLE_RSRC_ID(*rsrc_id)].done = 0;
    ...
}
```

此时资源注册并没有完成，因为资源并不是统一时间注册的，假如新注册一个资源时线程已经分配先前注册的资源了，那么线程 tsrm_tls_entry 结构中用于存储资源的 storage 数组就没有空间容纳新的资源了，因此需要对那些已分配 storage 数组的线程进行扩容。扩容的过程比较简单：遍历各线程的 tsrm_tls_entry，检查 storage 当时是否有空闲空间，有的话跳过，没有的话则扩容。

```
//ts_allocate_id():
for (i=0; i<tsrm_tls_table_size; i++) {
    tsrm_tls_entry *p = tsrm_tls_table[i];
    //tsrm_tls_table[i]可能保存着多个线程，需要遍历链表
    while (p) {
        if (p->count < id_count) {
            int j;
            //将 storage 扩容
            p->storage = (void *) realloc(p->storage, sizeof(void *)*id_count);
            //分配并初始化新注册的资源，实际这里只会执行一次，不清楚为什么用循环
            //另外这里不分配内存也可以，可以放到使用时再去分配
            for (j=p->count; j<id_count; j++) {
                p->storage[j] = (void *) malloc(resource_types_table[j].size);
                if (resource_types_table[j].ctor) {
                    //回调初始化函数进行初始化
                    resource_types_table[j].ctor(p->storage[j]);
                }
            }
            p->count = id_count;
        }
        p = p->next;
    }
}
//解锁
tsrm_mutex_unlock(tsmm_mutex);
```

```
return *rsrc_id;
```

4.4.1.3　获取资源

各线程根据资源 id 通过 ts_resource()方法获取到对应的资源。

```
#define ts_resource(id)         ts_resource_ex(id, NULL)
```

比如注册的 zend_executor_globals 这个资源，就可以通过如下的方式获取，取到的 executor_globals 就是属于线程自己的内存结构了。

```
zend_executor_globals *executor_globals;
executor_globals = ts_resource(executor_globals_id);
```

接下来详细看一下资源的获取过程。

Step1：获取线程的 tsrm_tls_entry 结构

首先是获取线程 id，如果是 pthread 则可以通过 pthread_self()方法获取；然后根据线程 id 从 tsrm_tls_table 数组中获取 tsrm_tls_entry，这里需要对取到的 tsrm_tls_entry 遍历比较线程 id，以找到该线程的 tsrm_tls_entry，如果没有找到则表示线程还未分配 tsrm_tls_entry，进入 Step2 进行分配，如果找到了则表示已经分配了 tsrm_tls_entry，进入 Step3 返回 storage 中对应的资源。

```
//ts_resource_ex():
THREAD_T thread_id;
int hash_value;
tsrm_tls_entry *thread_resources;
//获取线程 id
if (!th_id) {
    //查询线程本地存储，暂时忽略
    ...
    thread_id = tsrm_thread_id();//pthread_self(), 当前线程 id
}else{
    thread_id = *th_id;
}
tsrm_mutex_lock(tsmm_mutex);
//获取在 tsrm_tls_table 数组中的存储位置，实际就是 thread_id % tsrm_tls_table_size
hash_value = THREAD_HASH_OF(thread_id, tsrm_tls_table_size);
```

```
thread_resources = tsrm_tls_table[hash_value];
if (!thread_resources) {
    //线程的资源还没分配
    //进入 Step2 分配资源，然后重新调用本函数
    ...
    return ts_resource_ex(id, &thread_id);
} else {
    //遍历查找当前线程的 tsrm_tls_entry
    do {
        //找到了，中断查找，进入 Step3
        if (thread_resources->thread_id == thread_id) {
            break;
        }
        if (thread_resources->next) {
            //继续向后遍历
            thread_resources = thread_resources->next;
        } else {
            //遍历到最后也没找到，与上面的一致，进入 Step2 分配资源，然后重新调用本函数
            ...
            return ts_resource_ex(id, &thread_id);
        }
    } while (thread_resources);
}
```

Step2：分配线程资源

查找线程的 tsrm_tls_entry，如果没有找到，则需要进行资源的分配，这一步通过 allocate_new_resource()完成，分配后会把新分配的 tsrm_tls_entry 插入 tsrm_tls_table 对应链表的末尾，同时为所有的资源分配内存空间。注意：tsrm_tls_entry->storage 保存的资源地址，并不是资源的内存空间。最后，根据注册的资源数以及各资源注册时的配置 tsrm_resource_type 进行内存分配，分配完成后各线程就可以根据资源 id 获取本线程的资源了。

```
static void allocate_new_resource(tsrm_tls_entry **thread_resources_ptr,
THREAD_T thread_id)
{
    //thread_resources_ptr 为 tsrm_tls_table 对应链表末尾的地址，也就是新插入
tsrm_tls_entry 的地址
    //分配 tsrm_tls_entry 结构
```

```
        (*thread_resources_ptr) = (tsrm_tls_entry *) malloc(sizeof(tsrm_tls_entry));
        (*thread_resources_ptr)->storage = NULL;
        //根据已注册资源数分配用于保存资源地址的 storage 数组，注意这里并不是分配资源空间
        if (id_count > 0) {
            (*thread_resources_ptr)->storage = (void **) malloc(sizeof(void *)*
id_count);
        }
        (*thread_resources_ptr)->count = id_count;
        (*thread_resources_ptr)->thread_id = thread_id;
        //将当前线程的 tsrm_tls_entry 保存到线程本地存储，暂时忽略，稍后再作说明
        tsrm_tls_set(*thread_resources_ptr);
        //为全部资源分配空间
        for (i=0; i<id_count; i++) {
            ...
            //根据资源 tsrm_resource_type 获取资源的内存大小
            (*thread_resources_ptr)->storage[i] = (void *) malloc(resource_
types_table[i].size);
            if (resource_types_table[i].ctor) {
                //初始化资源

resource_types_table[i].ctor((*thread_resources_ptr)->storage[i]);
            }
        }
        ...
    }
```

仍以图 4-18 注册的 zend_array、uint32_t 这两种类型的资源为例，假如有 3 个线程，则线程资源分配后的内存空间如图 4-19 所示。

Step3：根据资源 id 获取资源

线程资源只能通过资源 id 获取，这一步比较简单，直接根据资源 id 从 tsrm_tls_entry->storage 数组中获取到资源地址。

```
    //id 就是资源 id
    TSRM_SAFE_RETURN_RSRC(thread_resources->storage, id, thread_resources->
count);
```

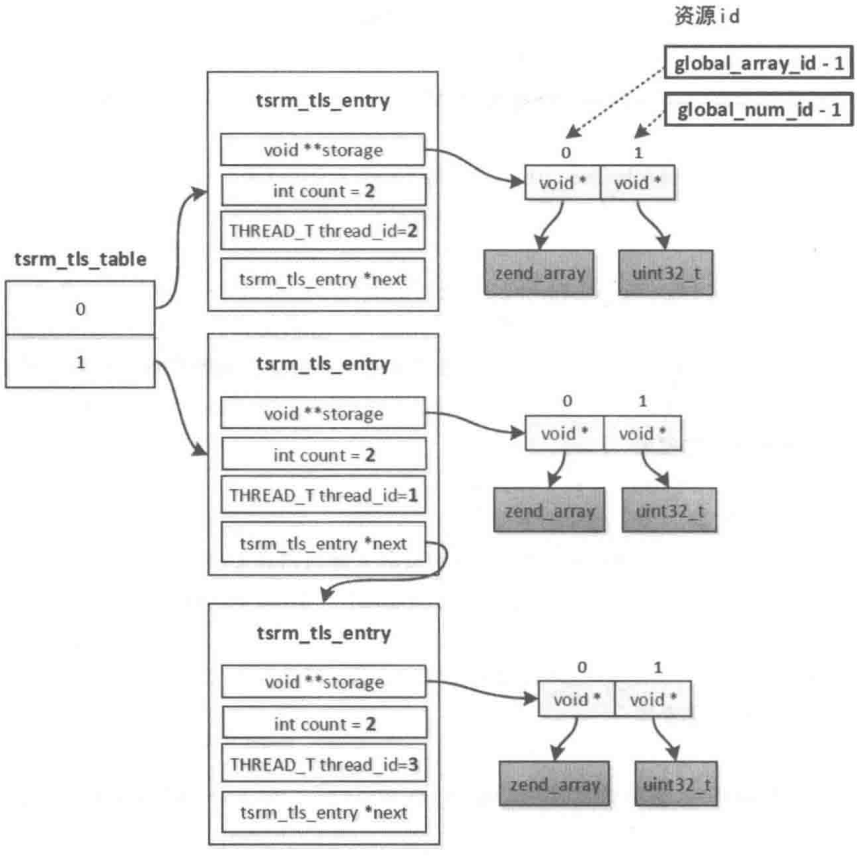


图 4-19 各线程资源的内存空间

TSRM_SAFE_RETURN_RSRC()展开后就是 tsrm_tls_entry->storage[id - 1]。

4.4.2 线程私有数据

上一节我们介绍了获取资源的过程，主要分为三步：第 1 步是获取线程 id；第 2 步是根据线程 id 查找 tsrm_tls_entry，这个过程需要加锁、遍历，比较耗时；第 3 步是根据资源 id 从 tsrm_tls_entry->storage 数组中获取资源。可以看到，整个流程比较长，而且第 2 步需要加锁，同一时刻只允许一个线程进行。另外，资源的访问又是非常频繁的动作，如果每次获取都要经历这三步，那么将严重影响性能，显然这是不可接受的。

TSRM 通过线程私有数据（Thread-Specific Data，TSD）优化了这个问题，TSD 是由 POSIX 线程库维护的，使用同一名称为不同线程保存数据的一种存储形式，即各线程可以根据同名的 key 获取到不同的变量地址。GNU C Library 定义了下面几个函数用于 TSD 的操作：

```
//创建名为 key 的 TSD
int pthread_key_create (pthread_key_t *key, void (*destructor)(void*))
//销毁 TSD
int pthread_key_delete (pthread_key_t key)
//根据 key 设置 TSD
int pthread_setspecific (pthread_key_t key, const void *value)
//根据 key 获取
void *pthread_getspecific (pthread_key_t key)
```

TSRM 在分配线程资源时将各线程 tsrm_tls_entry 的地址保存到 TSD 中，即 allocate_new_resource()，然后在 ts_resource_ex()获取资源时从 TSD 中取出，如图 4-20 所示。

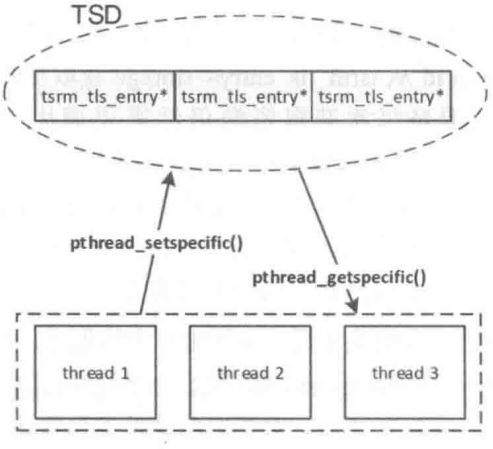


图 4-20 TSD

```
static pthread_key_t tls_key;
# define tsrm_tls_set(what)    pthread_setspecific(tls_key, (void*)(what))
# define tsrm_tls_get()        pthread_getspecific(tls_key)

static void allocate_new_resource(tsrm_tls_entry **thread_resources_ptr,
THREAD_T thread_id)
{
    ...
    //将当前线程的 tsrm_tls_entry 地址保存到 TLS
    tsrm_tls_set(*thread_resources_ptr);
    ...
}
TSRM_API void *ts_resource_ex(ts_rsrc_id id, THREAD_T *th_id)
```

```
{
    ...
    if (!th_id) {
        //从 TLS 获取 tsrm_tls_entry 地址
        thread_resources = tsrm_tls_get();
        ...
    }
    ...
}
```

4.4.3 线程局部存储

通过 TSD，TSRM 将资源的获取缩减为以下两步：调用 pthread_getspecific()获取 tsrm_tls_entry 地址；根据资源 id 从 tsrm_tls_entry->storage 获取资源。尽管通过 TSD 已经大大提高了资源的查询性能，但是每次获取资源仍然需要调用 ts_resource()，然后再调用 pthread_getspecific()，与变量的内存读写效率相比还是差很远，那么有没有什么方法能进一步提升效率呢？从上面的步骤分析，实际耗时的部分是获取 tsrm_tls_entry->storage 数组的环节，而后面根据资源 id 从数组中取值的操作是内存直接寻址，性能上没有问题，因此，这个问题就转化为“如何减少 tsrm_tls_entry->storage 数组的检索次数”。

在介绍 PHP7 的实现之前，我们先来说一下 PHP5 中的做法。PHP5 的解决方式非常简单，它通过参数传递的方式将 tsrm_tls_entry->storage 地址一级级地传给后面调用的函数。也就是说，第一次获取 tsrm_tls_entry->storage 的函数，会把它的地址作为参数传给调用的函数，然后被调用的函数接着再往下传，通过这种方式将 storage 传给所有函数。写过 PHP5 扩展的开发者对这两个宏一定不会陌生：TSRMLS_DC、TSRMLS_CC，或 TSRMLS_D、TSRMLS_C。TSRMLS_DC 用于函数定义，在扩展中定义函数时，需要在参数定义末尾加上 TSRMLS_DC；TSRMLS_CC 用于函数调用，调用其他函数时，需要在传参列表末尾加上这个宏。这两个宏就是用来传递 tsrm_tls_entry->storage 地址的。

```
#define TSRMLS_D   void ***tsrm_ls
#define TSRMLS_C   tsrm_ls
#define TSRMLS_DC   , TSRMLS_D
#define TSRMLS_CC   , TSRMLS_C
```

比如在扩展中定义一个 test()函数，则需要这样定义：

```
void test(int id TSRMLS_DC)
```

```
{
    ...
}
//展开后：
void test(int id , void ***tsrm_ls)
{
    ...
}
```

调用时：

```
void other_func(TSRMLS_DC)
{
    //call test()
    test(1234 TSRMLS_CC);
}
//展开后：
void other_func(void ***tsrm_ls)
{
    test(1234 , tsrm_ls);
}
```

Fpm 中，tsrm_tls_entry->storage 是在 main()函数中获取的，之后就一直向下传递，后面的函数直接通过参数获取到本线程的资源池。

```
//file:sapi/fpm/fpm/fpm_main.c  version:php-5.6.27
int main(int argc, char *argv[])
{
    ...
#ifdef ZTS
    void ***tsrm_ls;
#endif
    ...
#ifdef ZTS
    tsrm_startup(1, 1, 0, NULL);
    tsrm_ls = ts_resource(0);
#endif
    ...
}
```

PHP5 的这种处理方式非常简单，而且效率也很高，但是很不优雅，不管函数用不用 TSRM，都不得不在函数中加上那两个宏，也很容易遗漏。PHP7 并没有沿用这种实现，而是采用线程局部存储来保存各线程的 tsrm_tls_entry->storage。

线程局部存储（Thread-Local Storage，TLS）是一种为每个线程分配一份变量复制的机制，每个线程保存自己的一份副本，线程之间的变量地址是不同的。要定义一个线程局部变量很简单，只需简单地在全局或静态变量的声明中包含__thread 说明符即可，使用时需要注意：__thread 关键词只能独自使用或紧随 static、extern 之后。举个例子：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

__thread int num = 0;

void* worker(void* arg){
    while(1){
        printf("thread:%d %p\n", num, &num);
        sleep(1);
    }
}

int main(void)
{
    pthread_t tid;
    int ret;
    if ((ret = pthread_create(&tid, NULL, worker, NULL)) != 0){
        return 1;
    }
    while(1){
        num = 4;
        printf("main:%d %p\n", num, &num);
        sleep(1);
    }
    return 0;
}
```

这个例子中有两个线程，其中主线程修改了全局变量 num，但是并没有影响另外一个线程，同时根据打印出的 num 的地址可以看出，两个线程的 num 具有不同的地址。

PHP 中通过 ZEND_TSRMLS_CACHE_DEFINE()宏定义线程局部变量，这个变量保存的是 tsrm_get_ls_cache->storage 数组第一个元素的地址，它在 sapi_startup()中完成赋值：

```
//TSRM/TSRM.h
#define TSRMLS_CACHE_DEFINE() TSRM_TLS void *TSRMLS_CACHE = NULL;
#define TSRMLS_CACHE_UPDATE() if (!TSRMLS_CACHE) TSRMLS_CACHE =
tsrm_get_ls_cache()
```

展开后：

```
__thread void *_tsrm_ls_cache = NULL;
//初始化
_tsrm_ls_cache = tsrm_get_ls_cache();
```

EG(xxx)最终展开：((zend_executor_globals) (*(void* **) _tsrm_ls_cache))[executor_globals_id-1]->xxx)。

4.5 小结

本章主要介绍了 PHP 中内存相关的实现，包括变量的内存管理、底层内存池的实现，以及线程安全相关的内容。变量的自动 GC 是 PHP 非常重要的一个特性，主要通过引用计数与写时复制机制来实现，同时介绍了由于循环引用导致的变量无法回收的问题与解决方案。后面介绍了 PHP 实现的内存池，主要用于解决内存频繁申请、释放带来的性能问题以及内存碎片的问题。最后介绍的线程安全并不是 PHP 必用的一个功能，只有在多线程的应用场景下才会使用，但是线程安全是一个非常基础的问题，是多线程应用绕不开的一个问题，PHP 实现了一套通用的线程安全资源管理器，并通过 TLS 优化了线程资源的查找效率。

第 5 章 PHP 的编译与执行

本章将介绍 PHP 内部最核心的两个阶段：编译、执行，这也是 ZendVM 中最重要、最复杂的部分。通过本章的内容，你将弄清楚 PHP 代码是如何被机器识别、执行的。PHP 的编译与执行是两个相对独立的阶段，其中编译的流程涉及词法分析、语法分析、抽象语法树的生成，执行阶段则是根据编译阶段输出的产物（也就是 opline 指令）进行执行。

5.1 语言的编译与执行

语言的本质是信息交流的规则，通过约定俗成的规则让万物有了沟通的能力，语言并没有固定的法则，正因为如此，才在人类的世界里产生了如此之多的语言。尽管不同语言之间差异很大，但语言之间总能建立某种特定的关系，这种特定的关系可以将某种语言表达的信息转为另一种语言可感知的信息，我们把这个过程称为翻译。

编程语言是人类与计算机之间的交流规则，与人类的自然语言一样，也需要一个翻译的过程，因为计算机只认识自己的机器语言，无法理解人类定义的高级编程语言，这个翻译过程称之为编译。通过编程语言，能够让计算机准确地知道应该做什么事情。

根据编译的时机的不同，编程语言可分为编译型、解释型。编译型语言是指在程序在运行前提前编译为计算机可执行的二进制文件，在执行时直接执行机器指令，这种类型的典型代表就是 C、C++、Golang；解释型语言是指程序在运行时由解释器边编译边执行，也称为脚本语言，PHP 正是属于这种类型。无论是哪种类型的语言，它们的本质是一样地，编译型和解释型

语言的不同只是在于编译、执行这些环节的时机不一样。通俗地讲，编译型语言就好比是把饭先提前做好了然后再吃，而解释型语言就好比吃火锅，一边涮一边吃。

5.1.1　编译型语言

编译型语言由编译器负责语言的“翻译”工作，这个步骤是在线下完成的，在执行前编译器根据不同的机器类型将高级语言生成对应的低级机器语言指令，然后将这些机器语言指令按照可执行目标程序的格式存储到磁盘文件中，执行时再按照可执行程序的格式将其加载到内存中的对应位置（数据段、代码段），最后逐条执行机器指令。

编译型语言执行的是机器可直接识别的指令，它最大的优势就是效率高，执行时不需要经过耗时的编译过程。编译型语言在不同平台上需要进行重新编译，因为编译时生成的机器指令是针对特定机器的，比如 Windows 下的编译生成的可执行程序在 Linux 系统中是无法执行的。

以 C 语言为例，Linux 下由 GCC 编译器读取程序的源文件，经过预处理、编译、汇编、链接四个过程，将定义的 C 语言代码编译为可执行的目标程序，如图 5-1 所示。

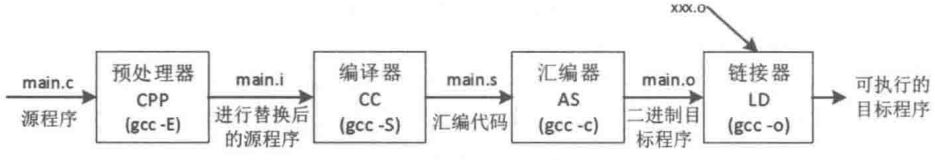


图 5-1　GCC 的编译流程

1）预处理

预处理环节主要对 C 语言程序中以#开头的命令进行替换：替换#include 包含的文件，用实际值替换#define 定义的字符串，根据#if 条件决定要编译的代码。通过 `gcc -o xxx.i -E xxx.c` 命令完成预处理，处理完成后的结果仍然是 C 语言代码。

2）编译

编译过程是将经过预处理后的 C 语言代码转换为汇编语言，通过以下命令完成：`gcc -S xxx.i`。转换后的结果就是汇编代码，仍然是可理解的文本文件。

3）汇编

汇编器将汇编代码生成机器指令，并生成扩展名为.o 的 ELF 可重定位目标文件。可重定位目标文件包括二进制机器指令及数据，由各个数据节（section）组成，包含数据节、代码节、符号表等，ELF 可重定位目标文件大致布局如图 5-2 所示，可以通过 readelf、objdump 命令查看.o 文件的这些信息。

```
$ objdump -x xxx.o
$ readdef -a xxx.o
```

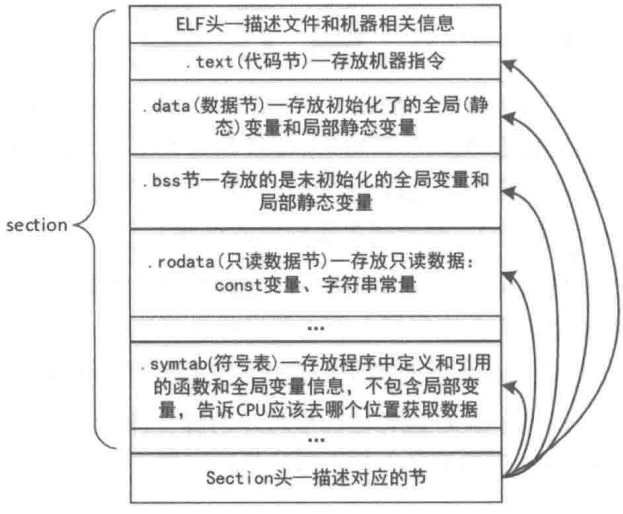


图 5-2 ELF 可重定位目标文件结构

4）链接

链接是生成可执行程序的最后一步，链接器将可重定位目标文件中引用其他文件的符号进行替换，同时，根据上一步骤生成的符号表，把函数、全局变量的引用位置替换为实际的存储位置。比如生成函数调用的指令，需要知道函数代码段的起始位置，这个信息就从符号表中获取。

可执行的二进制程序在执行时首先需要加载到内存中，这个过程会根据生成的 ELF 可执行文件的格式将数据段加载到内存中的对应位置。需要注意的是，ELF 文件中的节并不全部会被 load 到内存中，ELF 包含的在运行期间需要的节（section）被标为“可分配的”（allocable），比如代码节、数据节，但是有些节只是提供给链接器或其他工具使用的，在运行时并不需要，那些节就被标为“不可分配的”（non-allocable），在操作系统加载 ELF 时，只有 allocable 会被加载到内存中去，.symtab 符号表是提供给链接器使用的，因此不会被加载到内存中。

ELF 格式文件提供了两个视角，一个是从链接器的角度，一个是从加载器的角度。链接器把 ELF 文件看成 section 的集合，sections 中包含了链接和重定位的所有重要信息；加载器则把 ELF 文件看成 segment 的集合，segments 中包含了可执行文件需要被加载到内存中的必要信息。每个 segment 可以由一个或多个 section 组成，每个 segment 都有一个长度和一组与之关联的权限（如 read、write、execute），一个进程只有在权限允许且在 segment 中的偏移长度在 segment 指定的长度之内，才能正常引用 segment，否则将会出现 Segmentation fault（即段错误）的异常。

比如下面的例子，“abc”分配在.roread 节，执行时被加载到内存的 Text Segment 段，它是只读的，没有写的权限，因此会触发 Segmentation fault。

```
int main(void)
{
    char *str = "abc";
    str[0] = 'A';
    return 0;
}
```

32 位系统的程序默认内存布局如图 5-3 所示，这就是 ELF 可执行文件被加载到内存中的布局。

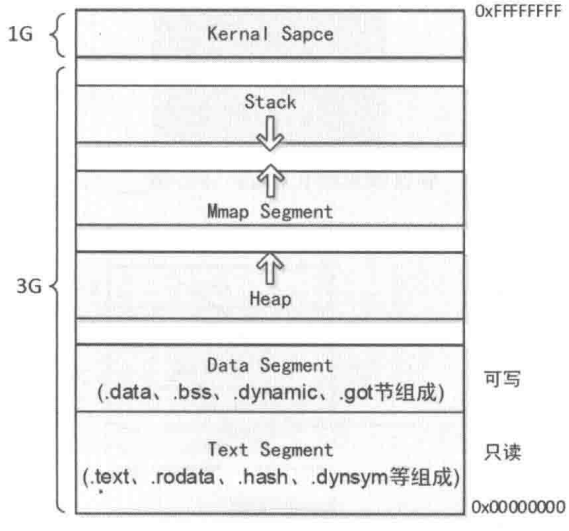


图 5-3　32 位默认内存布局

可执行程序被加载到内存以后，操作系统开始从代码段执行机器指令，在程序执行的过程中有两个重要的寄存器：ebp、esp，分别指向栈底、栈顶。函数在调用时首先将 ebp 入栈保存（push ebp），然后将 ebp 指向 esp，然后函数的参数、局部变量依次入栈，最后是函数返回值。这个过程中 esp 不断向下移，函数调用完成以后再依次出栈，最后将保存的 ebp 出栈，ebp 指回原来位置，通过这两个寄存器界定了函数内部局部变量的作用域。

计算机根据代码段具体的指令操作栈、堆以及数据段的数据，完成数据的计算。这里要把数据、指令两个概念区分开，ebp、esp 是用来控制数据栈的，并不是用来保存指令的，机器当前执行的指令通过 eip 寄存器保存，函数调用会把当前执行位置 push 到栈中保存，然后跳到函数的指令位置执行，执行完以后再回到原位置继续执行。

> 不同架构的 CPU，寄存器名称被添以不同前缀以指示寄存器的大小，例如对于 x86 架构，字母“e”用于名称前缀，指示各寄存器大小为 32 位；对于 x86_64 寄存器，字母“r”用于名称前缀，指示各寄存器大小为 64 位，上面提到的 ebp、esp、eip 在 64 位下对应的就是 rbp、rsp、rip。
>
> 对于参数传递的方式，x86 和 x86_64 定义了不同的处理方式，x86 会把参数压入调用栈中，而 x86_64 则是将函数参数传入通用寄存器，因此在 x86 和 x86_64 下生成的汇编会有些差别。
>
> 默认通过 gcc -S 生成的是 AT&T 格式的汇编，可以通过-masm 指定生成 Intel 格式的汇编：gcc -S -masm=intel main.c。

举个例子：

```
int sum(int a, int b)
{
    return a + b;
}
int main()
{
    int a =100;
    int b = 200;
    int c = sum(a, b);
    return 0;
}
```

具体的执行过程：

（1）如图 5-4 所示，main()并不是程序执行的第一个函数，程序的入口函数为__start()，假设在执行 main 前，rbp、rsp 分别指向 1、2 位置，然后开始进入 main 执行第 1 条指令 push rbp，此时 rsp 指向 3 的位置，同时把 rbp 的地址入栈保存，也就是说，当前 rsp 指向的内存保存的是 rbp 的地址。

（2）接着执行第 2 条指令：`mov rbp, rsp`，将 rbp 指向 rsp，然后继续向下执行，rsp 偏移，为局部变量 a、b、c 分配栈内存，然后执行 `call sum` 调用函数。这条命令首先会把当前指令入栈保存，然后将 rip 指向 sum 的起始位置，开始执行 sum 函数的指令，此时 rsp、rbp 的指向变化与 main 函数开始时相同，如图 5-5 所示。

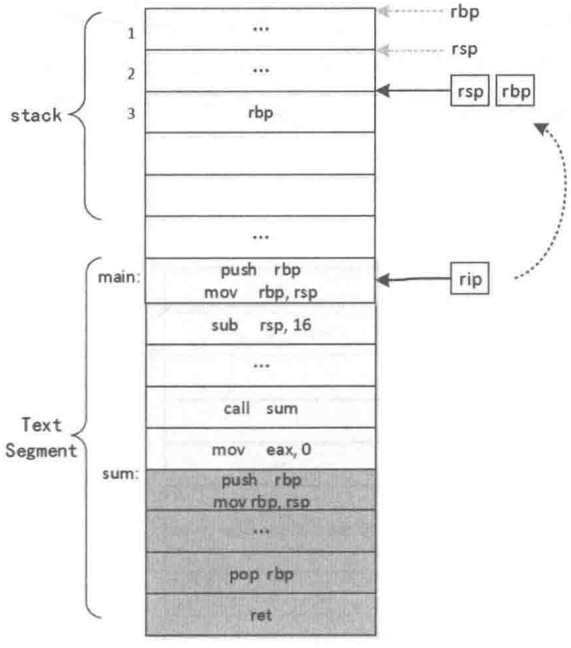


图 5-4　main 开始时的内存

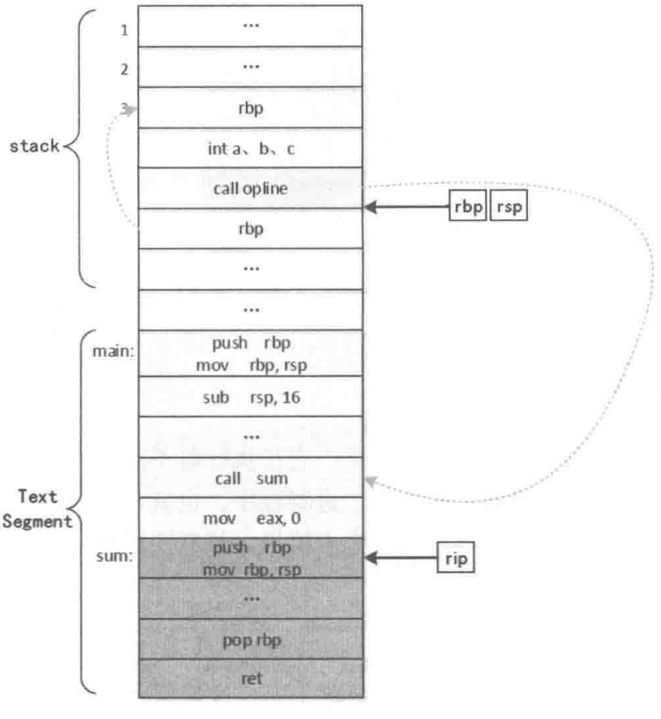


图 5-5　sum 函数调用

（3）sum 函数执行完成后依次出栈，最后 rsp 指向保存在栈上的 rbp 位置，执行 pop rbp，将栈顶的值（即之前 rbp 的位置）取出赋给 rbp，rsp、rbp 还原到 sum 调用前的位置，最后执行 ret 指令，将 rip 指向 main 函数中调用 sum 指令的位置，继续执行 main 后面的指令，如图 5-6 所示。

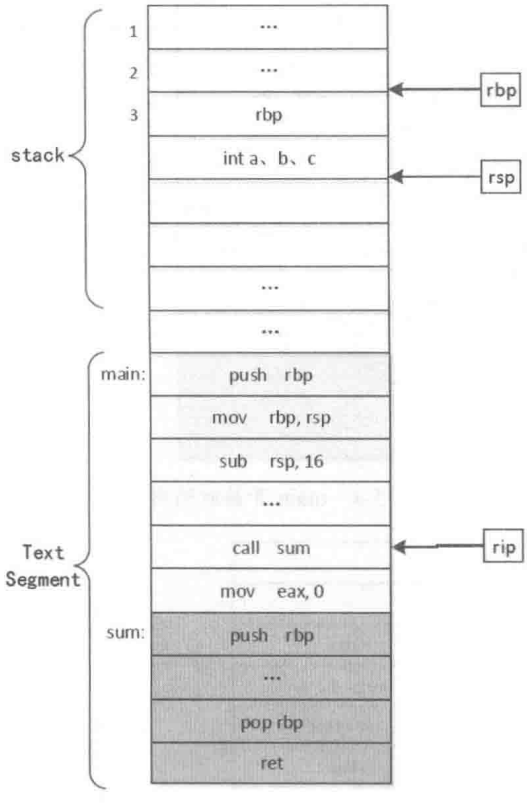


图 5-6 sum 函数调用返回

5.1.2 解释型语言

解释型语言与编译型语言的根本区别在于：它在执行前不需要编译为机器语言，而是由解释器进行解析执行，解释器为机器可识别的二进制程序。也就是说，解释型语言实际上是在语言与实际计算机中间加了一层解释器，也称为虚拟机，然后通过解释型语言控制编译好的解释器执行相应的机器指令，如图 5-7 所示。

图 5-7 解释器

解释型语言与解释器之间的“翻译”并不需要机器语言，而是由解析器自己确定的一种规则，这种规则可以让解释器知道该执行什么样的机器指令。这里一定要明确，解释器并不是将解释型语言编译为机器语言再去执行的，而是解释器本身预先定义好了一些具体的操作，这些操作被编译为机器指令，解释型语言在执行时，告诉解释器该执行哪段机器指令。比如现在编写一个简单的只能处理加法、减法操作的解释器，伪代码如下：

```
int add(int a, int b)
{
    return a + b;
}
int sub(int a, int b)
{
    return a - b;
}
int main( int argc, char *argv[])
{
    cmd = GET_CMD();
    switch (cmd->act){
        case "add":
            add(cmd->param[0], cmd->param[1]);
            break;
        case "sub":
            sub(cmd->param[0], cmd->param[1]);
            break;
    }
}
```

然后通过如下语法执行：

```
add 1 2
sub 4 3
```

解释器编译完成后启动，然后根据输入的解析型语言判断具体执行 add 或 sub，从而完成执行，如图 5-8 所示。

解释型语言与实际计算机之间多了一层解释器，屏蔽了不同平台之间机器语言的差异，因此解释型语言可以方便地运行在不同平台对应的解释器上，由解释器处理不同平台之间的差异，实现跨平台运行。由此换来的代价是运行效率低，与编译型语言直接执行机器指令相比，解释

型语言多了解释器解析的一步。另外，同样的计算，解释型语言往往需要执行更多的指令才能完成，比如加法操作，机器指令只需要一条，而在解释器中可能就需要调用一个函数才能完成。

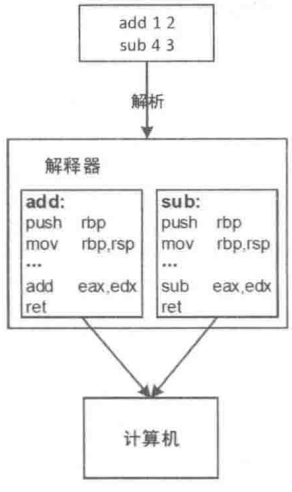


图 5-8 解释器的总体处理过程

5.2 Zend 虚拟机

Zend 虚拟机（ZendVM）是 PHP 语言的解释器，它是 PHP 语言实现的核心，负责 PHP 代码的解析、执行。ZendVM 预先定义好了大量的指令供用户在 PHP 代码中使用，这些指令对应的处理过程被编译为机器指令，执行 PHP 代码时，首先根据定义好的规则确定要执行的指令，然后调用对应的机器指令完成执行。

ZendVM 对于实际计算机而言就是普通的二进制可执行程序，它是编译好的机器指令，这个过程没什么可说的。实际上，PHP 代码的编译、执行过程与编译型语言的处理过程非常相似，只不过 PHP 代码被编译为了 ZendVM 可识别的指令，而不是机器指令，对于 PHP 而言，实际计算机是透明的，因此我们可以把 ZendVM 当作真正的计算机来理解。

ZendVM 由两部分组成：编译器、执行器，其中编译器负责将 PHP 代码解释为 ZendVM 可识别的指令（即 opline），同时生成对应的符号表（函数、类等），执行器负责执行 opcode 对应的机器指令，如图 5-9 所示。

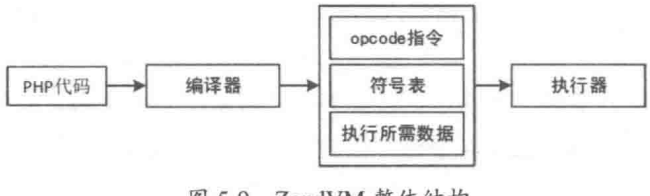


图 5-9 ZendVM 整体结构

5.2.1　opline 指令

opline 是 ZendVM 定义的执行指令，每条指令的编码为 opcode，等价于机器指令。PHP 代码在编译阶段被转化为 ZendVM 可识别的指令，ZendVM 根据不同的指令完成 PHP 代码的运行。opline 的编译是 PHP 编译器最核心的操作，也是编译阶段输出的产物。尽管 opline 指令与机器指令并不一样，但是对于执行它们的机器而言，其含义是相同的，都是控制执行机器工作的命令。

opline 指令的结构为 zend_op：

```
struct _zend_op {
    const void *handler; //指令执行 handler
    znode_op op1;   //操作数 1
    znode_op op2;   //操作数 2
    znode_op result; //返回值
    uint32_t extended_value;
    uint32_t lineno;
    zend_uchar opcode;  //opcode 指令
    zend_uchar op1_type; //操作数 1 类型
    zend_uchar op2_type; //操作数 2 类型
    zend_uchar result_type; //返回值类型
};
```

opline 指令的组成概括一下就是：对何数据，作何处理。前者称之为操作数，也就是指令的操作对象，后者称之为 opcode，即指令的处理动作。下面详细介绍各个字段的含义及用途。

5.2.1.1　opocde

opcode 为指令编码，唯一标识一个指令动作，它是运算符，用于决定做什么事情。目前 PHP 总共定义了 173 条 opcode，所有 PHP 的语法都是基于这些 opcode 实现的，比如赋值、四则运算、循环、条件判断等。ZendVM 的指令集定义在 zend_vm_opcode.h 头文件中。

```
//file: zend_vm_opcode.h
#define ZEND_NOP                    0
#define ZEND_ADD                    1
#define ZEND_SUB                    2
#define ZEND_MUL                    3
#define ZEND_DIV                    4
#define ZEND_MOD                    5
#define ZEND_SL                     6
#define ZEND_SR                     7
...
```

5.2.1.2 操作数

zend_op 结构中有三个 znode_op 类型的成员：op1、op2、result，它们称之为操作数。操作数是运算符作用于的实体，是表达式中的一个组成部分，它规定了指令中进行数字运算的量。也就是说，opcode 指定机器的运算动作，而操作数则是该动作操纵的具体对象，比如汇编指令 add eax, 100，意思是将 eax 寄存器中指向的值加上 100，这里 add 就是运算符，对应 ZendVM 的 opcode，而 eax、100 就是操作数，用来告诉计算机运算操作的对象。

ZendVM 为每条指令定义了三个操作数，当然，并不是所有的指令都会用到三个，有的指令不需要操作数，有的指令只需要一个，有的需要两个，有的则都会用到，这个由各条指令自行决定具体用途，其中 result 这个操作数用于返回值，告诉 ZendVM 运算结果的存储位置。

操作数的结构为 znode_op，实际就是个 32 位整型：

```
typedef union _znode_op {
    uint32_t     constant;
    uint32_t     var;
    uint32_t     num;
    uint32_t     opline_num; /*  Needs to be signed */
    uint32_t     jmp_offset;
} znode_op;
```

比如赋值操作`$a = 123`，操作数 1 用来告诉 VM 变量`$a`的位置，操作数 2 用来保存变量值 123 的位置，执行的时候，ZendVM 从操作数获取到变量`$a`与变量值 123 的存储位置，从而执行对应的动作，如图 5-10 所示。

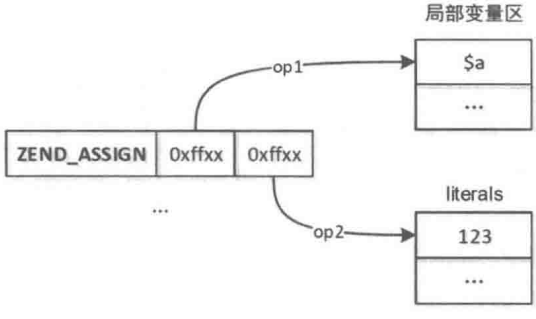


图 5-10 ZEND_ASSIGN 指令

操作数还有类型之分，因为运算操作的对象会有不同类型，比如同样是赋值操作：$b = $a 与 $b = 123，赋的变量值是不同的，一类是变量，一类是常量。因此，ZendVM 会根据操作数的不同类型，获取操作数指定的对象。这个类型就好比 ELF 可执行程序中，有的数据从栈上读取，有的从.data 段读取。操作数类型在 zend_op 中定义，同样有三个：op1_type、op2_type、result_type，分别对应三个操作数。操作数的具体类型有以下几个：

```
//file: zend_compile.h

#define IS_CONST        (1<<0)   //1
#define IS_TMP_VAR      (1<<1)   //2
#define IS_VAR          (1<<2)   //4
#define IS_UNUSED       (1<<3)   //8
#define IS_CV           (1<<4)   //16
```

各种类型的具体含义如下：

- IS_CONST：常量，也称作字面量（literal），也就是直接被写到 PHP 代码中的值，这种类型在我们写程序时会频繁用到，比如$a = 123、$a = "hello"、$a = array()、$a = 3 + $b，其中 123、"hello"、array()、3 就是字面量，它们的值都是固定不变的。字面量在编译阶段会被分配在单独的内存区（类似 ELF 加载到的.rodata 段的数据），执行时如果发现操作数为 IS_CONST 类型，那么就会到字面量存储位置获取数据。
- IS_CV：CV 变量（Compile Variable），也就是 PHP 脚本中通过$声明的变量，这种是 PHP 中最常见、最普通的一种类型了，比如$a = 123、$b = 3 + $c，其中$a、$b、$c 就是 CV 变量。
- IS_VAR：PHP 变量。注意：这里的变量并不是我们在 PHP 脚本中声明的变量，这个变量的含义可以这样理解——PHP 变量没有显式地在 PHP 脚本中定义，不是直接在代码通过$var_name 定义的。这个类型最常见的例子是 PHP 函数的返回值，比如$a = time()，time()的返回值就是 IS_VAR 类型。注意，它的返回值指的不是$a，这里实际是两个指令：函数调用、赋值，也就是 time()执行完以后会把返回值写到一个单独的位置，然后再把它的值赋给 CV 变量$a，而不是直接以$a 作为返回值。除了函数返回值，还有 $a[0]、$$a 这种。
- IS_TMP_VAR：临时变量，或者中间变量，比如$a = "hello~" . time()，这里"hello~" . time() 字符串拼接指令的执行结果就是临时变量，它们主要用于一些操作的中间结果。
- IS_UNUSED：表示操作数没有使用。

关于不同类型操作数所表示的变量的存储位置，下面两节会详细介绍。另外一个需要注意的地方是：result_type 除了上面几种类型，还有一种类型 EXT_TYPE_UNUSED，表示有返回值但是没有使用。比如只调用了函数而没有接收返回值：`time()`，这种类型跟 IS_UNUSED 不同，IS_UNUSED 表示操作数没有使用。

```
//file: zend_compile.h
#define EXT_TYPE_UNUSED (1<<5) //32
```

5.2.1.3 handler

handler 为每条 opcode 对应的实际处理函数，opcode 只是指令的编码，编译时会根据 opcode 为每条 opline 指令设置具体的 handler，执行时调用 handler 进行处理。handler 为 C 语言编写的、编译为机器指令的处理逻辑，默认情况下，handler 就是普通的 C 语言函数。此外，由于操作数有多种类型，同一 opcode 不同类型操作数的处理方式可能会有一些差异，因此每条 opcode 会根据操作数类型定义多个 handler。比如`$a = 123`与`$a = $b`，虽然都是赋值操作，但是一个是 CONST 类型，一个是 CV 类型。每条指令有 2 个操作数，操作数有 5 种类型，因此，每条 opcode 最多可有 5×5 = 25 个 handler。handler 的基本调用过程如图 5-11 所示。

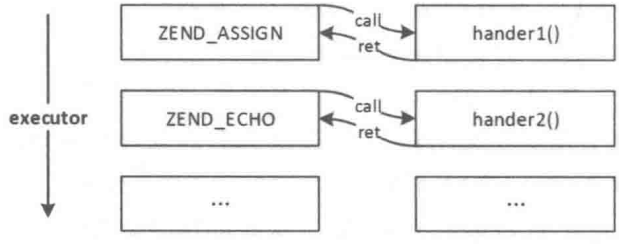


图 5-11 opline 指令处理 handler

5.2.2 zend_op_array

opline 是编译生成的单条指令，所有的指令集合组成了 zend_op_array。除了指令集合，zend_op_array 保存着很多编译生成的关键数据，比如前面介绍的字面量存储区就在 zend_op_array 中。zend_op_array 是编译器的输出，也是执行器的输入，它的角色相当于编译型语言最终编译出的可执行文件。对于 ZendVM 而言，zend_op_array 就是可执行数据，每个 PHP 脚本都会被编译为独立的 zend_op_array 结构。

```
struct _zend_op_array {
    //common 是普通函数或类成员方法对应的 opcodes 快速访问时使用的字段，后面分析 PHP
函数实现的时候会详细讲
    ...
    uint32_t *refcount;

    uint32_t this_var;

    uint32_t last;
    //opcode 指令数组
    zend_op *opcodes;
    //PHP 代码里定义的变量数：op_type 为 IS_CV 的变量，不含 IS_TMP_VAR、IS_VAR 的
```

```
    //编译前此值为 0，然后发现一个新变量这个值就加 1
    int last_var;
    //临时变量数:op_type 为 IS_TMP_VAR、IS_VAR 的变量
    uint32_t T;
    //PHP 变量名数组
    zend_string **vars; //这个数组在 ast 编译期间配合 last_var 来确定各个变量的编
号，非常重要的一步操作
    ...
    //静态变量符号表:通过 static 声明的
    HashTable *static_variables;
    ...
    //字面量数量
    int last_literal;
    //字面量(常量)数组，这些都是在 PHP 代码定义的一些值
    zval *literals;
    //运行时缓存数组大小
    int  cache_size;
    //运行时缓存，主要用于缓存一些操作数以便于快速获取数据，后面单独介绍这个机制
    void **run_time_cache;

    void *reserved[ZEND_MAX_RESERVED_RESOURCES];
};
```

下面介绍几个关键的成员：

- opcodes：就是开始所说的指令集合，这是一个数组，执行器执行时将从该数组的第一条指令开始，直到最后。
- literals：字面量存储区，上一节介绍的操作数类型中，IS_CONST 类型的字面量保存在单独的存储位置，这个位置就是 literals。根据编译时字面量出现的先后顺序，依次分配在 literals 数组中，其中 last_literal 用于记录当前字面量的数量，它的值从 0 开始，依次增大，用于访问字面量的操作数用的就是这个值，执行时就是根据这个操作数，从 literals 数组中获取对应的数据。
- last_var、vars：last_var 这个成员用来统计 CV 变量数，vars 数组用来保存所有的 CV 变量名，也就是 PHP 脚本中通过$声明的变量。编译过程中，如果发现一个 CV 变量，首先会遍历 vars 数组，检查该变量是否已经存在 ，如果不存在则表示该变量第一次出现，这时就会把 last_var 的值分配给它作为这个变量的操作数，然后把它的名称插入 vars 数组。如果 CV 在 vars 数组中存在了，则表示前面已经出现过，这个时候直接使用之前分配的操作数即可。比如$a = 123; echo $a;，首先编译$a = 123，发现$a 是一个 CV 变量，并且在 vars 中没有查到，因此，赋值语句的 op1 的操作数就是 0，即 op1.var

= 0，然后把$a 插入 vars 数组。接下来编译 echo $a 时，首先会查找 vars 数组，发现已经存在了，$a 在 vars 数组中的下标就是操作数，因此 echo 语句的 op1 操作数就直接用$a 之前分配的操作数 0。

- T：这个值记录的是 TMP、VAR 类型操作数的数量，编译时，如果发现需要用到 TMP 或 VAR 操作数，就会使用 T 的值作为操作数，然后把 T 的值加 1。

从 zend_op_array 结构可以看到，它囊括了 PHP 编译过程的所有产物，剩下的成员在后面用到的时候再单独说明，这里不一一介绍了。

从上面对几个成员的介绍，我们可以看到：ZendVM 为每一个 PHP 变量、临时变量、字面量按顺序编了号，这个编号就是每条 opline 指令的操作数，换句话说，在执行时，ZendVM 就是根据这些编号进行相应数据存取的，这一点与 C 程序的实现是一致的。下面看个例子：

```
#include <stdio.h>
int main()
{
    char *name = "pangudashu";

    printf("%s\n", name);
    return 0;
}
```

我们知道指针 name 分配在栈上，而 pangudashu 分配在常量区，那么变量名 name 分配在哪呢？实际上 C 里面是不会存变量名称的，编译时会将变量名替换为偏移量表示：ebp - 偏移量，所有局部变量都是通过相对 ebp 的内存偏移读取的。对应的汇编：

```
.LC0:
    .string "pangudashu"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movq    $.LC0, -8(%rbp)
    movq    -8(%rbp), %rax
    movq    %rax, %rdi
    call    puts
    movl    $0, %eax
```

```
    leave
```

-8(%rbp)就是 name 变量，也就是 rbp - 8。PHP 中的变量也采用类似的方式读取，每个 CV 变量、临时变量、字面量都有自己唯一的编号，它们分配在不同区域上，然后根据各自的编号实现数据访问。其中字面量保存在 literals 数组中，那么 CV 变量、临时变量在哪呢？CV 变量、临时变量是运行时分配的，与字面量不同，每次执行都需要重新分配，执行完以后释放，好比 C 程序调用函数时，每次调用都会在栈上重新分配局部变量，调用完释放，而常量区的数据在程序退出时才释放。

5.2.3　zend_execute_data

C 程序在执行时首先会分配运行栈，局部变量、上下文调用信息都通过栈来保存，eip 寄存器指向指令区，函数调用时首先将 eip 入栈保存，然后移动 ebp、esp 分配新的栈，执行完以后再将保存的 eip 出栈还原，继续执行。从 C 程序执行的过程来看，最重要的两部分为执行栈、eip。同样地，ZendVM 的执行器在执行流程中，通过 zend_execute_data 结构实现了类似 C 程序中执行栈、eip 的功能。首先来看一下这个结构：

```
typedef struct _zend_execute_data zend_execute_data;
struct _zend_execute_data {
    const zend_op       *opline;
    zend_execute_data   *call;
    zval                *return_value;
    zend_function       *func;
    zval                This;
    zend_class_entry    *called_scope;
    zend_execute_data   *prev_execute_data;
    zend_array          *symbol_table;
    void                **run_time_cache;
    zval                *literals;
};
```

ZendVM 执行 opcode 指令前，首先会根据 zend_op_array 信息分配一个 zend_execute_data 结构，这个结构用来保存运行时信息，包括当前执行的指令、局部变量、上下文调用信息等，各成员字段的含义：

- **opline**：当前执行中的指令，等价于 eip 的作用，执行之初 opline 指向 zend_op_array->opcodes 指令集的第一条指令，当执行完一条指令后，该值会更新为下一条指令。
- **return_value**：返回值，执行完以后，会把返回值设置到这个地址。

- **symbol_table**：全局变量符号表，第 9 章介绍全局变量时再作说明。
- **prev_execute_data**：调用上下文，当函数调用或者 include 时，会重新分配一个 zend_execute_data，并把当前执行的 zend_execute_data 保存到被调函数的 zend_execute_data->prev_execute_data，被调函数执行完成后，再根据 prev_execute_data 还原到原来的执行位置，prev_execute_data 等价于 C 程序调用过程中 call、ret 指令的作用。
- **literals**：就是 zend_op_array->literals。

除了上面介绍的这些成员，zend_execute_data 还有一个重要组成部分：动态变量区。前面一节我们介绍了 PHP 编译过程中会为所有的 CV 变量、临时变量编号，这些变量就分配在 zend_execute_data 结构上，它们按照编号依次分配在 zend_execute_data 结构末尾，根据具体的 CV 变量、临时变量数（即 zend_op_array->last_var 和 zend_op_array->T）确定内存大小。这些变量的编号就是用于在 zend_execute_data 上获取对应数据的，zend_op_array 与 zend_execute_data 关系如图 5-12 所示。

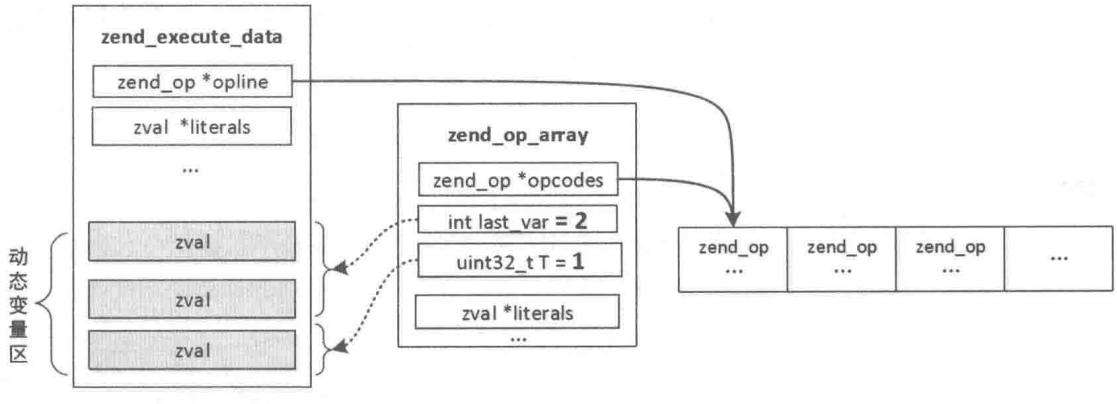


图 5-12 zend_op_array 与 zend_execute_data 的关系

5.2.4 zend_executor_globals

zend_executor_globals 是 PHP 整个生命周期中非常重要的一个数据结构，它是全局符号表，在 main 执行前分配（非 ZTS 下），直到 PHP 退出，PHP 中经常见到的 EG 宏操作的就是这个结构。zend_executor_globals 保存着类、函数符号表，类、函数编译过程中会注册到相应的符号表中，执行时如果发生函数调用、实例化类就会去各自的符号表中查找。另外，zend_executor_globals 还有一个指针指向当前执行的 zend_execute_data。除了这些，zend_executor_globals 还有很多其他用途的成员，后续碰到的时候再展开说明，其中比较重要的几个成员如图 5-13 所示。

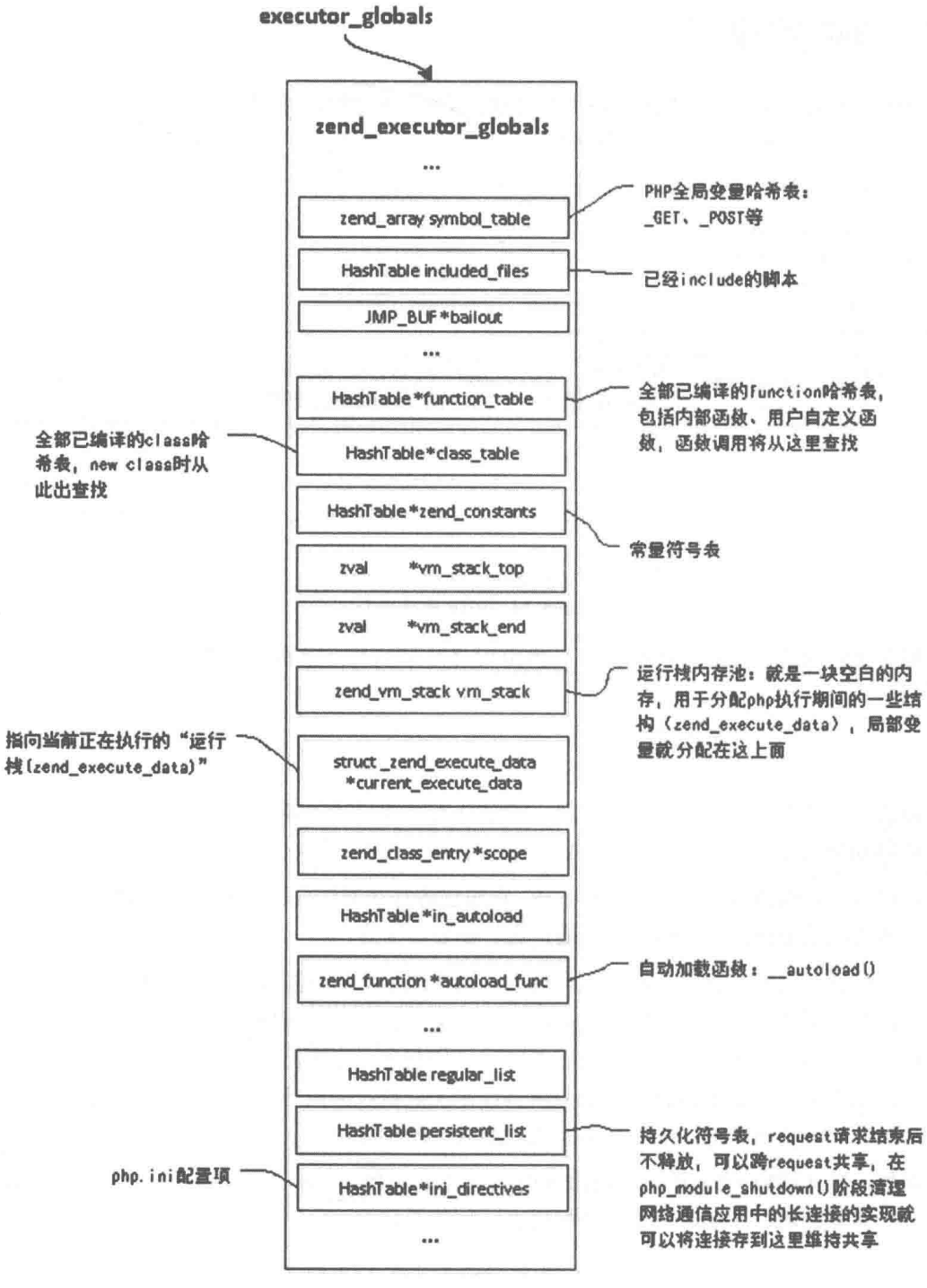


图 5-13　zend_executor_globals 结构

5.3 PHP 的编译

PHP 的编译过程是将 PHP 脚本代码根据定义的语法规则解析为 opcode 指令，在这个过程中先后经历词法分析、语法分析生成抽象语法树，然后再将抽象语法树编译为 opcode 指令，最终输出 zend_op_array，处理过程如图 5-14 所示。

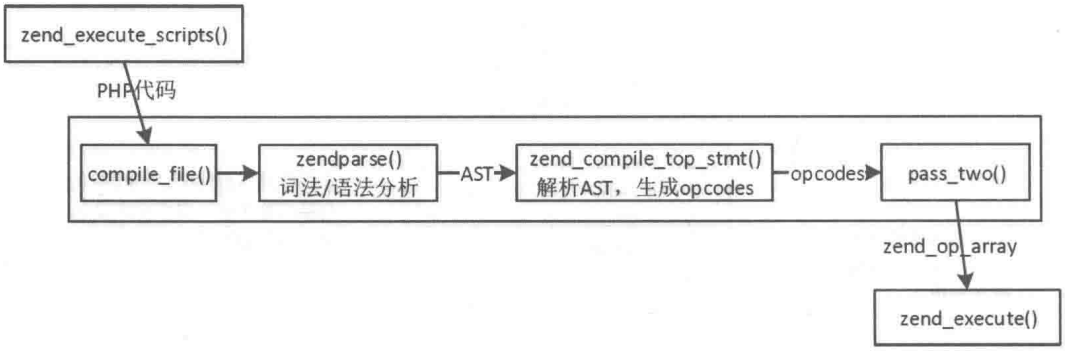


图 5-14 PHP 编译流程

接下来的两节，我们来详细看一下 PHP 编译环节的处理逻辑。

5.3.1 词法、语法解析

词法分析是编译过程的第一个阶段，词法分析器逐行读入源代码，然后按照构词规则，将 PHP 代码切割为定义好的可识别的 token。例如：`$a = 123`，其中`$a` 会被识别为 T_VARIABLE。

语法分析是编译过程的一个逻辑阶段，它的任务是在词法分析的基础上，将单词序列（token）组合成各类语法短语，如语句声明、表达式、函数定义等。

语法分析器需要与词法分析器配合使用，由词法分析器将源代码切割为 token 返回给语法分析器，然后语法分析器根据根据 token 组合检索匹配的语法规则，生成抽象语法树。简单地讲，词法分析器是将源代码去除一些无关符号，切割为一个个不同类型的单词，然后由语法分析器负责来理解这些单词组合，语法分析器预定义好所有的语法规则，理解的过程就是根据单词组合到这些规则中匹配，从而知道源代码要表达的意思。

例如：表达式$a = 3 + 4 - 6，词法分析器将其分割为$a、=、3、+、4、-、6，这些 token 被语法分析器所理解、匹配，用语法分析树表示如图 5-15 所示。

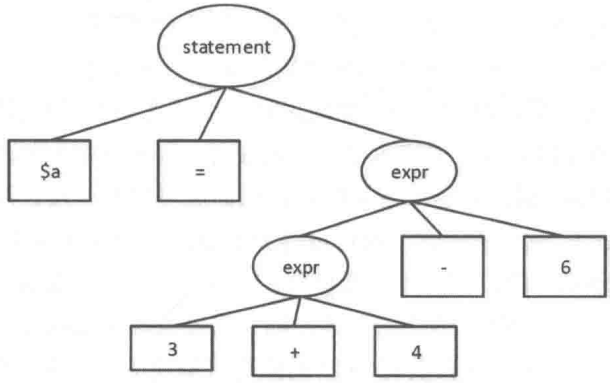


图 5-15　语法分析树

PHP 的词法分析、语法分析过程分别使用 re2c、yacc 完成。

1）re2c

re2c 是一个词法扫描器，它的词法解析规则格式如下：

```
/*!re2c
    token 格式 {
        ...
        return token 类型;
    }
*/
```

token 格式可以按照正则规则编写，后面大括号里的内容为命中定义的 token 后的处理，这里可以按照不同需求进行个性化处理，通常最后会返回 token 类型，以便告诉 yacc 解析出的是什么 token。

2）yacc

yacc 是语法分析器，其配置规则就是进行各种 token、字符组合。token 之间通过空格隔开，在命中处理逻辑中可以通过$0、$1……获取对应的 token 值，返回值可以通过$$设置：

```
规则标识:
    token1 token2
        {...}
;
```

在上面这个例子中，就可以通过$1 获取 token1 的值。

这里不再对 re2c、yacc 作更多解释，想要了解更多的内容推荐看一下《flex 与 bison》这本书。

接下来介绍抽象语法树，在 PHP7 之前的版本中是没有这个概念的，此前在语法分析阶段会直接生成 opcode 指令，这样导致 PHP 的编译器与执行器耦合在一起，假如后面改变 PHP 编译 opcode 的方式，那么语法规则文件也需要随之修改。或者 PHP 的语法规则发生变化了而编译 opcode 指令不需要改变时，也需要修改语法规则文件。因此 PHP7 在语法解析与编译 opcode 指令之间加了一层抽象语法树，将语法规则的解析抽离成单独的一层。这样一来，如果语法规则发生改变而编译 opcode 没有变化时，就可以将新的语法规则编译为原来的抽象语法树节点，从抽象语法树编译为 opcode 的过程并不需要修改，比如现在访问对象的成员属性及方法的规则是“->”，如果要改成 Java 中通过“.”的方式，则只需要修改生成抽象语法树的规则即可，后面的过程并不需要改动。再比如现在要将 PHP 的执行引擎换成全新的一套 VM，但是仍然使用 PHP 的语法，这个时候就不要改动语法解析规则，只需要修改抽象语法树编译为 opcode 的过程，将抽象语法树编译为新 VM 的指令即可。

因此，抽象语法树的作用是非常重要的，通过抽象语法树将 PHP 的编译器与执行器很好地隔离开了。

抽象语法树的结构非常简单，它通过不同节点表示具体的语法，抽象语法树节点可分为四类。

1）普通节点

这类节点作为非叶子节点，通常用于某种语法的根节点，结构为 zend_ast：

```
typedef struct _zend_ast       zend_ast;
struct _zend_ast {
    zend_ast_kind kind; /* 节点类型 */
    zend_ast_attr attr; /* Additional attribute, use depending on node type */
    uint32_t lineno;    /* 行号 */
    zend_ast *child[1]; /* 子节点 */
};
```

zend_ast 最重要的两部分就是节点类型 kind 与子节点 child，需要注意的是，child[1]并不意味着它只有一个子节点，实际上不同 kind 类型 zend_ast 的子节点数是不同的，具体的子节点数根据 kind 类型限定。节点类型：

```
enum _zend_ast_kind {
    ...
    /* 0 个子节点 */
    ZEND_AST_MAGIC_CONST = 0 << ZEND_AST_NUM_CHILDREN_SHIFT,  //0
    END_AST_TYPE,  //1
```

```
    /* 1 个子节点的类型 */
    ZEND_AST_VAR = 1 << ZEND_AST_NUM_CHILDREN_SHIFT, //64
    ZEND_AST_CONST,
    ...
    /* 2 个子节点的类型 */
    ZEND_AST_DIM = 2 << ZEND_AST_NUM_CHILDREN_SHIFT,  //512
    ZEND_AST_PROP,
    ...
};
```

也就是说，这些节点的子节点类型都是固定的，根据 kind 就可知道该节点的子节点数，这类节点最多有 4 个子节点，最少的没有子节点。

2）list 节点

顾名思义，list 节点是多个节点的组合，组成 list 的这些节点通常具有相同的节点类型，比如 `use aa, bb, cc`，导入多个命名空间的语法，就会生成列表节点，编译时循环编译各个节点即可。list 节点与普通节点的结构相比，多了一个记录子节点数量的成员 children：

```
typedef struct _zend_ast_list {
    zend_ast_kind kind;
    zend_ast_attr attr;
    uint32_t lineno;
    uint32_t children;
    zend_ast *child[1];
} zend_ast_list;
```

zend_ast_list 与 zend_ast 节点前三个成员完全一致，因此使用时会把 zend_ast_list 的内存地址转为 zend_ast 插入抽象语法树，使用时再根据 kind 转为 zend_ast_list，这样将所有的节点统一为 zend_ast 节点。

在 list 节点中有一个比较特殊的节点：ZEND_AST_STMT_LIST，值为 133。这个节点类型本身不表示任何语法，只是用来组织各个节点的，相当于一个节点数组，里面的节点之间没有任何关系，抽象语法树的根节点 CG(ast)就是这个类型。比如下面的代码：

```
$a = 123;
$b = $a;
new stdClass();
```

生成的抽象语法树如图 5-16 所示。

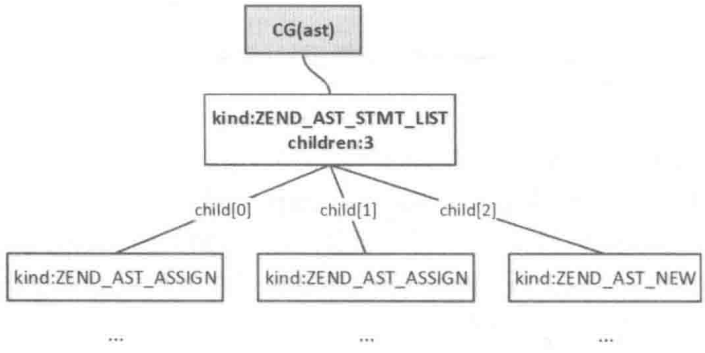


图 5-16 抽象语法树 ZEND_AST_STMT_LIST 节点

3）数据节点

这种节点的结构为 zend_ast_zval，它没有子节点，而多了一个 zval 成员，通常作为叶子节点，用于存储词法分析器切割出的 token 字符，也就是语法的操作对象。

```
enum _zend_ast_kind {
    ZEND_AST_ZVAL = 1 << ZEND_AST_SPECIAL_SHIFT,
    ...
};
typedef struct _zend_ast_zval {
    zend_ast_kind kind;
    zend_ast_attr attr;
    zval val;
} zend_ast_zval;
```

比如$a = 123，其 token 为 a、=、123，“=”为语法动作，通过非叶子节点 ZEND_AST_ASSIGN 表示，同时它也是该语句的根节点，子节点表示具体操作的对象，其中 a 为赋值的操作对象，通过变量节点 ZEND_AST_VAR 表示，而“a”名称则保存在这个节点的子节点中，类型就是 ZEND_AST_ZVAL。同样，值“123”也是 ZEND_AST_ZVAL 类型，该赋值语句的语法节点如图 5-17 所示。

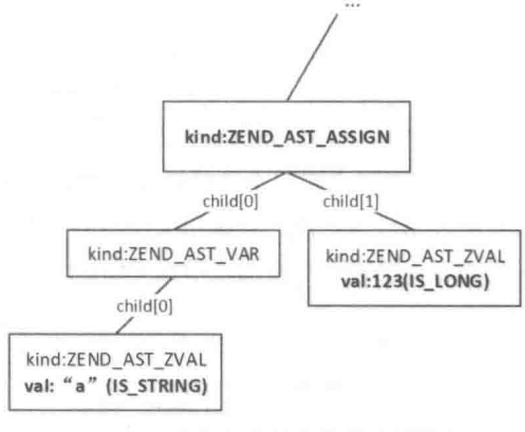


图 5-17 赋值语法的抽象语法树节点

4）声明节点

这类节点用于函数、类、成员方法、闭包的表示，后面几章再介绍。

下面我们看一下 PHP 中具体的实现。PHP 的词法规则文件定义在 Zend/zend_language_scanner.l 中，语法规则文件定义在 Zend/zend_language_parser.y 中，这两个规则文件需要通过执行 re2c、yacc 生成对应的 C 语言代码。编译开始的入口函数为 compile_file()：

```
ZEND_API zend_op_array *compile_file(zend_file_handle *file_handle, int type)
{
    ...
    zend_op_array *op_array = NULL;
    //打开 PHP 脚本文件
    if (open_file_for_scanning(file_handle)==FAILURE) {
        ...
    }else{
        ...
        CG(ast) = NULL;
        if (!zendparse()) {
            ...
        }
        zend_ast_destroy(CG(ast));
        ...
    }
    return op_array;
}
```

首先是打开 PHP 脚本文件，然后调用 zendparse()完成语法分析，zendparse()中将不断调用 zendlex()切割 token，然后匹配语法，生成抽象语法树，zend_language_parser.y 语法规则文件中的 zend_ast_create()方法就是生成语法树节点的操作。在编译过程中使用到一个重要的结构 zend_compiler_globals，它与 zend_executor_globals 一样作为全局变量分配，记录编译时的一些重要信息，其中编译生成的抽象语法树就保存在这个结构中，即 CG(ast)。

使用 re2c、yacc 进行词法、语法分析时，有两个地方需要注意。

（1）语义值（token 值）。

词法解析器解析到的 token 值内容就是语义值，比如$a = 123，其中 a、123 就是语义值。这些值统一通过 zval 存储，zval 在 zendlex()中分配，然后将其地址作为参数传递给 lex_scan()进行 token 扫描，当匹配到某个 token 时，把语义值保存到该地址中，从而传递给语法解析器使用。

```
#define yylex          zendlex

//zend_compile.c
int zendlex(zend_parser_stack_elem *elem)
{
    zval zv;
    int retval;
    ...
again:
    ZVAL_UNDEF(&zv);
    //进行词法扫描，将 zval 地址传入
    retval = lex_scan(&zv);
    if (EG(exception)) {
        //语法错误
        return T_ERROR;
    }
    ...
    if (Z_TYPE(zv) != IS_UNDEF) {
        //如果在分割 token 中有 zval 生成，则将其值复制到 zend_ast_zval 结构中
        elem->ast = zend_ast_create_zval(&zv);
    }

    return retval;
}
```

比如 PHP 中的解析变量的规则$var_name，其词法规则为：

```
int lex_scan(zval *zendlval)
{
...
<ST_IN_SCRIPTING,ST_DOUBLE_QUOTES,ST_HEREDOC,ST_BACKQUOTE,ST_VAR_OFFSET>
"$"{LABEL} {
    //将匹配到的语义值保存在 zval 中
    //只保存{LABEL}内容，不包括$，所以是 yytext+1
    zend_copy_value(zendlval, (yytext+1), (yyleng-1));
    RETURN_TOKEN(T_VARIABLE);
}
```

```
...
}
```

yytext 指向命中的语义值起始位置，类型为 char*，yyleng 为语义值的长度，zend_copy_value 将创建 zend_string 的 value，该语义值返回到 zendlex()中后，会被保存到 ZEND_AST_ZVAL 类型的节点中，该节点的 val 就是解析出的语义值。

（2）语义值类型。

yacc 调用 re2c 解析出的 token 有两个含义：一个是 token 类型、另一个是 token 值。token 类型以 yylex（即 zendlex()）的返回值告诉 yacc，而 token 值就是语义值，这个值一般定义为固定的类型，这个类型就是语义值类型，默认为 int，可以通过 YYSTYPE 定义，而 PHP 中这个类型是 zend_parser_stack_elem 。语法分析器传给词法分析器一个 zend_parser_stack_elem 地址，词法分析器将解析出的 token 值保存到这个地址中，这就是为什么 zendlex 的参数为 zend_parser_stack_elem 指针，以及 zendlex()为什么要把保存于 zval 的 token 值生成一个 ZEND_AST_ZVAL 节点的原因。

```
#define YYSTYPE zend_parser_stack_elem

typedef union _zend_parser_stack_elem {
    zend_ast *ast; //抽象语法树主要结构
    zend_string *str;
    zend_ulong num;
} zend_parser_stack_elem;
```

另外，zend_parser_stack_elem 结构是一个联合体，ast 类型用的比较多，其他两种类型可暂时忽略。在语法解析规则中，可以通过<ast/str/num>指定 token 或 type 使用哪种类型：

```
%token <ast> T_LNUMBER   "integer number (T_LNUMBER)"
%token <ast> T_VARIABLE  "variable (T_VARIABLE)"

%type  <ast>  top_statement  namespace_name  name  statement  function_
declaration_statement
%type <ast> class_declaration_statement trait_declaration_statement
```

仍然以 T_VARIABLE 为例，这里指定该 token 使用 ast，因此，在命中的语法解析规则处理中，传入的类型就是 zend_parser_stack_elem->ast，下面规则$1、$2、$3、$$值的类型就是 zend_ast：

```
simple_variable:
        T_VARIABLE          { $$ = $1; }
```

```
    |   '$' '{' expr '}'    { $$ = $3; }
    |   '$' simple_variable { $$ = zend_ast_create(ZEND_AST_VAR, $2); }
;
```

PHP 的语法解析器从 start 开始调用，首先会创建一个根节点 list，然后层层匹配各个规则，将命中 top_statement 规则生成的语法树节点依次加到这个 list 中，最后将语法树根节点保存到 CG(ast) 完成整个解析操作：

```
start:
    top_statement_list  { CG(ast) = $1; }
;

top_statement_list:
    top_statement_list top_statement { $$ = zend_ast_list_add($1, $2); }
    |   /* empty */ { $$ = zend_ast_create_list(0, ZEND_AST_STMT_LIST); }
;

top_statement:
        statement                       { $$ = $1; }
    |   function_declaration_statement  { $$ = $1; }
    |   class_declaration_statement     { $$ = $1; }
    |   trait_declaration_statement     { $$ = $1; }
    ...
;
```

解析的过程可以简单地理解为：根据拿到的 token 去检索定义好的 token 组合，看是否命中。大致的过程可用以下伪代码表示：

```
void start()
{
    zend_ast *root = top_statement_list();
    CG(ast) = root;
}

zend_ast *top_statement_list()
{
    zend_ast *res;
    zend_ast *root = zend_ast_create_list(0, ZEND_AST_STMT_LIST);
```

```
    if(res = statement()) { zend_ast_list_add(root, res); }
    if(res = function_declaration_statement()) { zend_ast_list_add(root,
res); }
    ...
    return root;
}

zend_ast *statement()
{
    zend_ast *res;
    switch(token_list){
      //命中 do{}while()语法
       case T_DO statement T_WHILE '(' expr ')' ';':
           return zend_ast_create(ZEND_AST_DO_WHILE, $2, $5);
           break;
       case 其他规则:
         ...
    }
}
```

当然，具体的解析过程中会有很多递归调用，要复杂很多。语法解析的过程并不涉及 ZendVM 的核心实现，它是相对独立的一部分内容，如果对这部分内容掌握的不是非常透彻，也不会妨碍对 ZendVM 后续部分的理解，这部分内容建议读者以了解为主，能够根据 PHP 代码知道生成的抽象语法树是什么样即可，具体的生成过程可以不必细究。但如果你想通过修改语法实现新的 PHP 语言特性，那么就需要熟练掌握 re2c、yacc 的使用了，规则的语法可以参考 PHP 现有规则。

5.3.2　抽象语法树编译

抽象语法树的编译就是生成 ZendVM 可识别指令的过程，语法解析过程的产物保存于 CG(ast)，接着 ZendVM 会把抽象语法树进一步编译为 zend_op_array 。根据上一节对 Zend 虚拟机的介绍，在抽象语法树的编译过程中，会计算出当前脚本使用的 CV 变量、临时变量、字面量数，根据先后顺序为这些变量编号，这些编号作为 opline 指令的操作数，在执行时按照该编号获取相应数据。

抽象语法的编译过程主要定义在 Zend/zend_compile.c 文件中，这个过程发生在 compile_file()

完成 zendparse()处理之后，编译的过程就是从 CG(ast)节点开始遍历抽象语法树，根据不同的节点类型编译为对应语法的 opline。在编译前首先分配一个 zend_op_array 结构，然后进行初始化，并把 CG(active_op_array)指向该结构，CG(active_op_array)表示当前正在编译的 zend_op_array，在抽象语法树的编译过程中，会把编译产生的 opline 指令插入到 CG(active_op_array)。

```
//compile_file:
if (!zendparse()) {
    ...
    zend_op_array *original_active_op_array = CG(active_op_array);
    //分配 zend_op_array 内存
    op_array = emalloc(sizeof(zend_op_array));
    //初始化 zend_op_array
    init_op_array(op_array, ZEND_USER_FUNCTION, INITIAL_OP_ARRAY_SIZE);
    CG(active_op_array) = op_array;
    ...
}
```

从上面的过程可以看到，在编译前会把当前 CG(active_op_array)保存下来，这是因为 compile_file()在 PHP 生命周期中并非只会调用一次，include、require 等语句的执行过程也会触发，因此这里会保存下来，编译完成后再还原回去。zend_op_array 初始化完成后从 CG(ast)开始编译：

```
//将抽象语法树编译为 opline 指令
zend_compile_top_stmt(CG(ast));
```

zend_compile_top_stmt()为编译的入口，如果想查看 PHP 代码生成的抽象语法树，那么就可以通过 gdb 设置该函数的断点进行查看。CG(ast)是一个 ZEND_AST_STMT_LIST 类型的 list 节点，list 里的每一个节点都是独立的语法节点，因此 zend_compile_top_stmt()会编译该 list，然后调用 zend_compile_stmt()具体编译各个节点。

```
//file: zend_compile.c
void zend_compile_top_stmt(zend_ast *ast)
{
    if (!ast) {
        return ;
    }
    if (ast->kind == ZEND_AST_STMT_LIST) {
        //将 zend_ast 节点转为实际的 zend_ast_list: list = (zend_ast_list *)ast
```

```
        zend_ast_list *list = zend_ast_get_list(ast);
        uint32_t i;
        //遍历 list
        for (i = 0; i < list->children; ++i) {
            //list 各 child 语句相互独立，递归编译
            zend_compile_top_stmt(list->child[i]);
        }
        return ;
    }
    //非 ZEND_AST_STMT_LIST 节点
    zend_compile_stmt(ast);
    ...
}
```

zend_compile_stmt()主要根据不同的节点类型（kind）进行不同的处理，这里我们不可能把每种类型的处理都讲一遍，下面从一个例子具体看一下几个语法的编译处理。

```
//示例 5.3.2
$a = 123;
$b = "hi~";
echo $a,$b;
```

该示例有两条赋值语句，一条输出语句，最终生成的抽象语法树如图 5-18 所示。

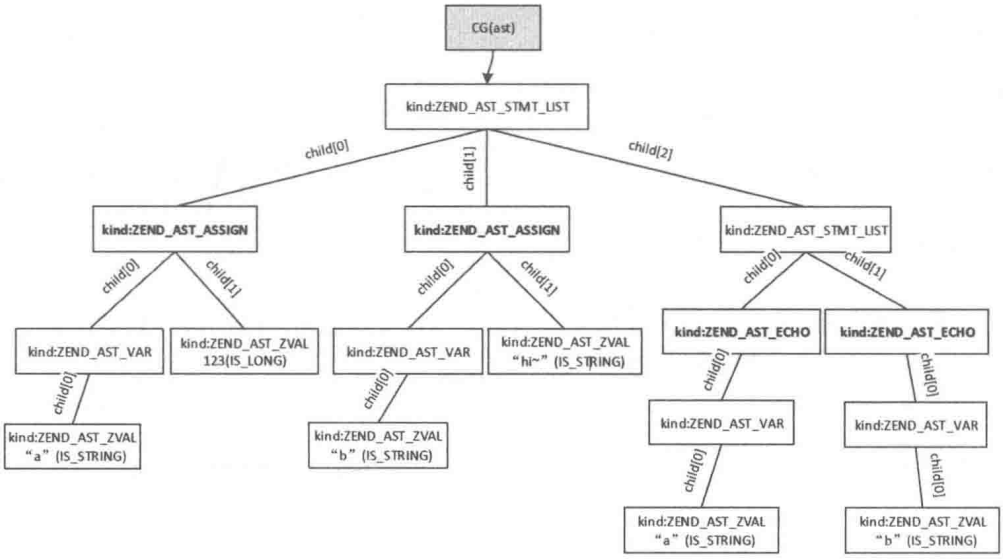


图 5-18　5.3.2 抽象语法树示例

从抽象语法树可以很直观地看到，CG(ast)节点下有 3 个节点，分别是两个 ZEND_AST_ASSIGN 节及一个 ZEND_AST_STMT_LIST 节点，其中 ZEND_AST_STMT_LIST 节点包含两个 ZEND_AST_ECHO 节点。也就是说，`echo $a,$b;`等价于 `echo $a; echo $b;`，下面我们分别看一下 ZEND_AST_ASSIGN、ZEND_AST_ECHO 节点的编译过程。

1. 赋值语句的编译

ZEND_AST_ASSIGN 节点有两个子节点，其中第一个节点保存的是变量名称，第二个节点保存的是变量值表达式，该节点继续调用 zend_compile_expr()处理，最终由 zend_compile_assign()进行编译：

```
void zend_compile_stmt(zend_ast *ast)
{
    ...
    switch (ast->kind) {
        ...
        default:
        {
            znode result;
            //编译表达式
            zend_compile_expr(&result, ast);
            zend_do_free(&result);
        }
    }
    ...
}

void zend_compile_expr(znode *result, zend_ast *ast)
{
    switch (ast->kind) {
        ...
        case ZEND_AST_ASSIGN:
            zend_compile_assign(result, ast);
            return;
        ...
    }
}
```

继续进入 zend_compile_assign()：

```
void zend_compile_assign(znode *result, zend_ast *ast)
{
    zend_ast *var_ast = ast->child[0]; //变量名
    zend_ast *expr_ast = ast->child[1];//变量值表达式

    znode var_node, expr_node;
    zend_op *opline;
    uint32_t offset;
    ...
    switch (var_ast->kind) {
        case ZEND_AST_VAR:
        case ZEND_AST_STATIC_PROP:
            offset = zend_delayed_compile_begin();
            //生成变量名的 znode，这个结构只在此处临时用，所以直接分配在 stack 上
            zend_delayed_compile_var(&var_node, var_ast, BP_VAR_W);
            //递归编译变量值表达式，最终需要得到一个 ZEND_AST_ZVAL 的节点
            zend_compile_expr(&expr_node, expr_ast);
            zend_delayed_compile_end(offset);
            //生成一条 op
            zend_emit_op(result, ZEND_ASSIGN, &var_node, &expr_node);
            return;
        ...
    }
}
```

该过程主要分为三步：编译生成变量名的操作数（即 op1）、编译生成变量值操作数（即 op2）、生成赋值指令。

介绍 zend_op_array 时曾提到，每个 PHP 变量（CV 变量）都会分配一个编号用于变量的存取，这个编号就是在这个地方分配的。CV 变量的编号就是按照顺序分配的，CV 变量保存在 zend_op_array->vars 数组中，生成 CV 变量的操作数时，首先会检查这个数组，如果变量已经存在则直接使用之前分配的编号，如果不存在则按序分配一个编号，然后将变量名称插入 vars 数组。上面的示例中，$a 为第 1 个 CV 变量，$b 为第 2 个，因此$a 的编号（操作数）就是 0，$b 为 1，具体的过程如图 5-19 所示。

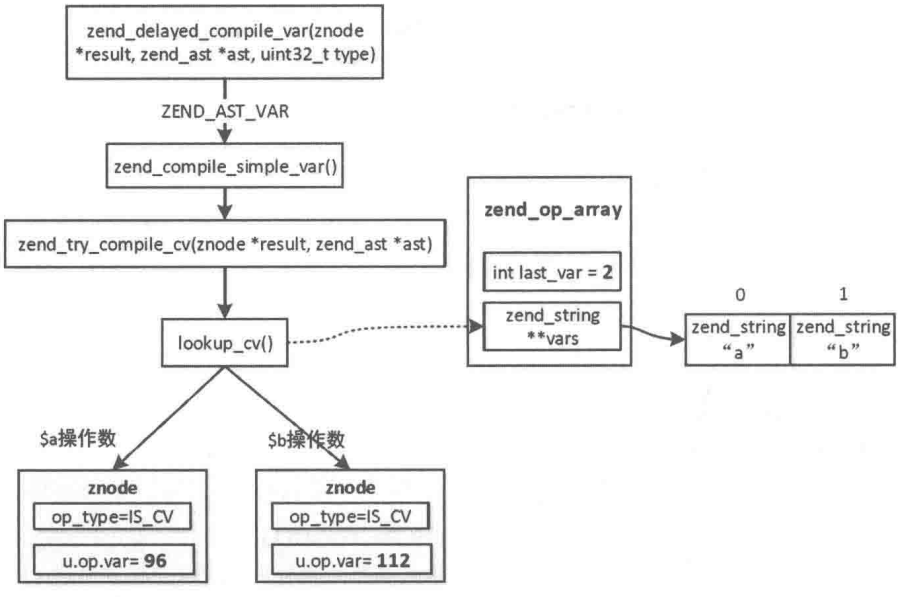


图 5-19 a、b 操作数的生成

zend_try_compile_cv()中生成$a 的操作数，最终的节点为 ZEND_AST_ZVAL，也就是保存变量名"a"的节点。另外需要注意的是，这里生成的操作数结构并不是 znode_op，而是 znode，这个结构只在编译期间使用，在生成 opline 时会被替换为 znode_op。

```
//zend_try_compile_cv:
zend_ast *name_ast = ast->child[0];
if (name_ast->kind == ZEND_AST_ZVAL) {
    zend_string *name = zval_get_string(zend_ast_get_zval(name_ast));
    ...
    result->op_type = IS_CV;
    result->u.op.var = lookup_cv(CG(active_op_array), name);
    ...
}
```

lookup_cv()就是生成操作数的过程，详细看一下这个函数的实现：

```
static int lookup_cv(zend_op_array *op_array, zend_string* name)
{
    int i = 0;
    zend_ulong hash_value = zend_string_hash_val(name);
    //遍历 op_array.vars 检查此变量是否已存在
```

```
        while (i < op_array->last_var) {
            if (ZSTR_VAL(op_array->vars[i]) == ZSTR_VAL(name) ||
                    (ZSTR_H(op_array->vars[i]) == hash_value &&
                     ZSTR_LEN(op_array->vars[i]) == ZSTR_LEN(name) &&
                     memcmp(ZSTR_VAL(op_array->vars[i]), ZSTR_VAL(name),
ZSTR_LEN(name)) == 0)) {
                zend_string_release(name);
                return (int)(zend_intptr_t)ZEND_CALL_VAR_NUM(NULL, i);
            }
            i++;
        }
        //这是一个新变量
        i = op_array->last_var;
        op_array->last_var++;
        if (op_array->last_var > CG(context).vars_size) {
            CG(context).vars_size += 16; /* FIXME */
            op_array->vars = erealloc(op_array->vars, CG(context).vars_size *
sizeof(zend_string*));//扩容vars
        }

        op_array->vars[i] = zend_new_interned_string(name);
        //传NULL时返回的是96 + i*sizeof(zval)
        return (int)(zend_intptr_t)ZEND_CALL_VAR_NUM(NULL, i);
    }
```

这里变量的编号从0、1、2、3…依次递增，但是实际使用中并不是直接使用这个下标，而是转化成了内存偏移量offset，这个是ZEND_CALL_VAR_NUM宏处理的，所以变量的操作数实际是96、112、128…递增的，这个96是根据zend_execute_data大小设定的（不同的平台下对应的值可能不同）。前面介绍ZendVM时曾简单提过：CV变量、临时变量分配在运行时的zend_execute_data上，因此offset是相对zend_execute_data的偏移值，执行时，根据offset和zend_execute_data地址索引要操作的对象。

```
    #define ZEND_CALL_FRAME_SLOT \
        ((int)((ZEND_MM_ALIGNED_SIZE(sizeof(zend_execute_data)) + ZEND_MM_
ALIGNED_SIZE(sizeof(zval)) - 1) / ZEND_MM_ALIGNED_SIZE(sizeof(zval))))
    //将变量编号转为内存偏移
    #define ZEND_CALL_VAR_NUM(call, n) \
```

```
(((zval*)(call)) + (ZEND_CALL_FRAME_SLOT + ((int)(n))))
```

因此，最终生成的a、b 的操作数为 96、112，如图 5-19 所示。

编译完赋值语句的变量名以后，接着就是变量表达式的编译，这个过程将再次调用 zend_compile_expr()编译。示例中的表达式比较简单，是 CONST 类型，如果是其他表达式，比如$a = $b + 3，则会在编译过程中生成其他操作的 opline，也就是先执行$b + 3，然后再把值赋给$a。但是，无论表达式如何复杂，其最终的结果只有一个，zend_compile_expr()会得到最终的操作数。

```
void zend_compile_expr(znode *result, zend_ast *ast)
{
    switch (ast->kind) {
        case ZEND_AST_ZVAL:
            //将变量值复制到 znode.u.constant 中
            ZVAL_COPY(&result->u.constant, zend_ast_get_zval(ast));
            //类型为 IS_CONST，这种 value 后面将会保存在 zend_op_array.literals 中
            result->op_type = IS_CONST;
            return;
        ...
    }
}
```

CONST 类型的变量值（也就是字面量）会被复制到操作数的 znode 结构中，但是这里并没有看到设置操作数的 u.op.val，不要着急，字面量的操作数会在后面分配。

现在，赋值语句的两个操作数都已经编译完成了，接下来就是最终生成 opline 指令的操作了。

```
zend_emit_op(result, ZEND_ASSIGN, &var_node, &expr_node);
```

zend_emit_op()传入三个操作数，以及 opcode，这个过程会在 CG(active_op_array)新生成一条 opline，具体分为以下 3 步。

Step1：分配 opline，设置 opcode

```
//zend_emit_op:
 zend_op *opline = get_next_op(CG(active_op_array));
 opline->opcode = opcode;
```

Step2：设置操作数 op1、op2

如果用到了操作数 1、2，则将临时结构 znode 中操作数结构转移到 zend_op 中，这里有一个特殊处理，如果操作数为 CONST 类型的字面量，则将保存在 znode.u.op.constant 中的字面量值插入 CG(active_op_array)->literals 字面量符号表中，然后更新操作数。

```
//设置 op1
if (op1 == NULL) {
    SET_UNUSED(opline->op1);
} else {
    SET_NODE(opline->op1, op1);
}
//同样的方式设置 op2
...
```

SET_NODE()宏展开如下所示，其中，CONST 类型将调用 zend_add_literal()方法插入字面量，并返回字面量的编号。该编号没有什么规则，就是按照先后插入顺序确定的，示例中 123 这个字面量的操作数就是 0，“hi~”的就是 1。

```
#define SET_NODE(target, src) do { \
        target ## _type = (src)->op_type; \
        if ((src)->op_type == IS_CONST) { \
            target.constant     =     zend_add_literal(CG(active_op_array),
&(src)->u.constant); \
        } else { \
            target = (src)->u.op; \
        } \
    } while (0)
```

Step3：设置返回值操作数

通过 zend_emit_op()方法生成的 opline 指令的返回值类型都是 VAR，前面曾提过，这种类型与 TMP_VAR 临时变量、CV 变量都分配在 zend_execute_data 上，VAR 与 TMP 类型的操作数编号记在 zend_op_array->T 上，因此，这里生成返回值的操作数就是通过 CG(active_op_array)->T 分配的。

```
//zend_emit_op:
if (result) {
    zend_make_var_result(result, opline);
```

```
}

static inline void zend_make_var_result(znode *result, zend_op *opline)
{
    opline->result_type = IS_VAR; //返回值类型固定为 IS_VAR
    //为返回值编号，这个编号记在临时变量 T 上
    opline->result.var = get_temporary_variable(CG(active_op_array));
    GET_NODE(result, opline->result);
}
```

get_temporary_variable()这个方法就是用来生成 VAR、TMP_VAR 类型操作数值的，目前，操作数为 0、1、2…的自增值，并不是像 CV 变量那样转为了内存偏移值。get_temporary_variable()的处理非常简单：

```
static uint32_t get_temporary_variable(zend_op_array *op_array)
{
    return (uint32_t)op_array->T++;
}
```

从赋值语句的具体编译过程可以看到，它是有返回值的，这个返回值操作数是在 zend_compile_stmt()中分配的，但是这个返回值并不被 PHP 使用，编译完以后就被 free 了。zend_do_free()将返回值的操作数类型打上了 EXT_TYPE_UNUSED 标记：

```
//zend_compile_stmt:
default:
{
    znode result;
    zend_compile_expr(&result, ast);
    zend_do_free(&result);
}
```

到此，ZEND_AST_ASSIGN 节点已经编译完成了，其结果就是生成了一条 ZEND_ASSIGN 指令，该指令操作数 op1 为 offset 内存偏移，op2 为字面量编号，返回值为 VAR 类型编号。也就是说，除了 CV 类型的操作数用的是 offset 内存偏移，其他类型用的都是递增编号。

2. echo 语句的编译

ZEND_AST_ECHO 节点由 zend_compile_echo()方法负责编译，这个节点的编译比较简单，主要分为两步：编译生成操作数 1、编译生成 ZEND_ECHO 指令。该指令只用到一个操作数，也没有返回值，用到的这个操作数就是最终要输出的内容，它也通过 zend_compile_expr()编译。

具体处理如下：

```
void zend_compile_echo(zend_ast *ast)
{
    zend_op *opline;
    zend_ast *expr_ast = ast->child[0];

    znode expr_node;
    zend_compile_expr(&expr_node, expr_ast);
    //生成 1 条新的 opcode
    opline = zend_emit_op(NULL, ZEND_ECHO, &expr_node, NULL);
    opline->extended_value = 0;
}
```

示例中输出的是 CV 类型的变量，该操作数最终也通过 zend_try_compile_cv()生成，此时 lookup_cv()将查找到先前分配的 CV 操作数，整体的处理流程如图 5-20 所示。

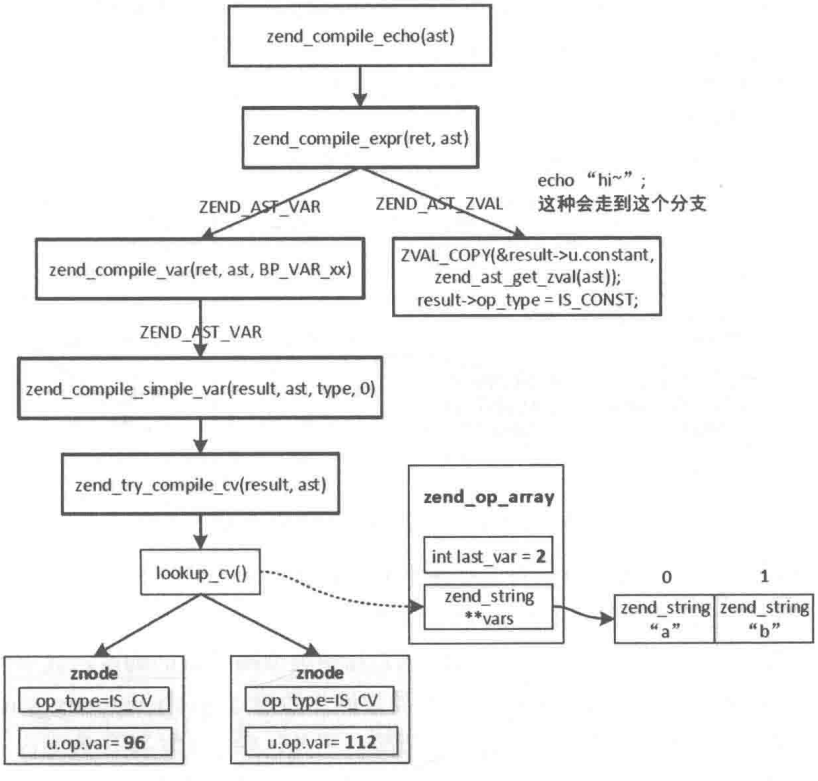


图 5-20　ZEND_AST_ECHO 节点的编译

最终，示例生成的全部 opline 指令与 zend_op_array 结构如图 5-21 所示。

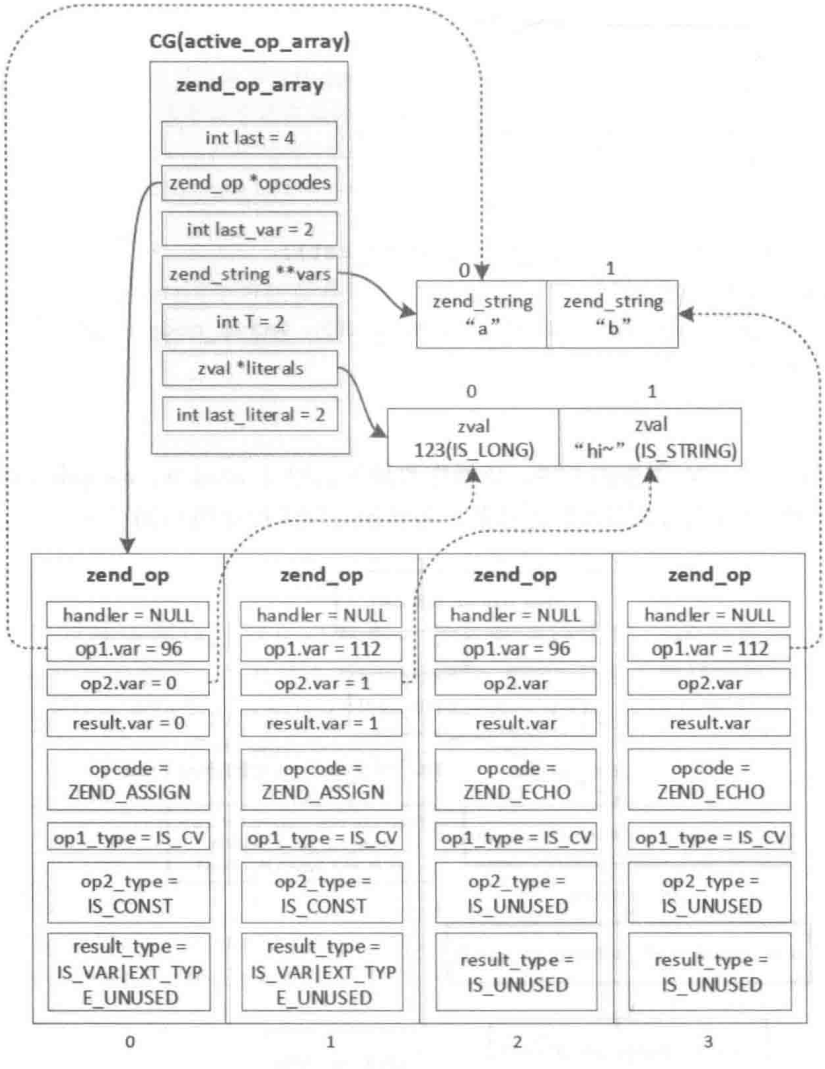


图 5-21　ZEND_ASSIGN、ZEND_ECHO 生成的 opline

从抽象语法树编译为 opline 指令的过程并不复杂，我们只介绍了两种比较简单的语法编译过程，主要目的是通过这两个例子让大家知道编译的基本过程。在整个过程中有三个地方需要特别注意，那就是 CV、VAR/TMP_VAR、CONST 几种类型操作数的确定，其中 CV 类型操作数通过 lookup_cv()方法分配，VAR/TMP_VAR 类型操作数通过 get_temporary_variable()分配，CONST 类型操作数通过 zend_add_literal()方法分配。另外，除了 CV 操作数为内存偏移值，其余两类都是递增数值，即 0、1、2…，在接下来的处理中，将根据该递增值转化为内存偏移值。

5.3.3　pass_two()

完成抽象语法树的编译后，编译阶段并没有结束，最后还会生成一条 ZEND_RETURN 指令，这一步骤通过 zend_emit_final_return()完成。即使我们在脚本中定义了 return，这个地方也会多编译一条 return 指令，只不过不会被执行而已，如果我们在脚本中没有定义 return，那么就需要依靠 ZendVM 为我们添加的这条指令结束执行，默认返回整型 1，因此在生成 ZEND_RETURN 指令时会这个整型添加到字面量数组中。

除了编译生成 ZEND_RETURN 指令外，在 zend_compile_top_stmp()完成后还有一个比较重要的操作：pass_two()，该过程将对一些特殊 opcode 进行处理，部分指令在编译时有些需要的信息暂时无法获得，只有等到抽象语法树编译完成才能获得，因此那些操作就在 pass_two()中完成，比如 goto 语句，具体的内容会在第 9 章介绍。除此之外，pass_two()中还会把 VAR/TMP_VAR、CONST 操作数由递增编号转为内存偏移值。

这里暂时忽略特殊 opcode 的处理，首先介绍 VAR 与 TMP_VAR 类型操作数内存偏移值的分配，在 ZendVM 执行时，CV、VAR、TMP_VAR 三种类型的变量均分配于同一内存区域，即 zend_execute_data，在 5.3.2 节介绍的 CV 操作数就是相对实际存储位置的内存偏移。同样的，VAR、TMP_VAR 类型的操作数也是相对 zend_execute_data 的内存偏移，它们与 CV 内存分布的关系是“先分配 CV 变量，再分配 VAR 与 TMP_VAR 的内存”。因此，它们的内存起始位置在最后一个 CV 之后，它们的分配是从 zend_op_array->last_var 开始的，然后再按照之前分配的递增编号依次确定内存偏移值。

其次是 CONST 类型变量的操作数，这种变量的值保存在 zend_op_array->literals 数组中，因此直接将数组下标转为内存偏移即可，也就是 sizeof(zval)*编号，该过程通过 ZEND_PASS_TWO_UPDATE_CONSTANT()宏完成。

具体处理如下：

```
//file: zend_opcode.c
ZEND_API int pass_two(zend_op_array *op_array)
{
    zend_op *opline, *end;
    ...
    opline = op_array->opcodes;
    end = opline + op_array->last;
    //遍历全部 opline
    while (opline < end) {
        ...
        //将 0、1、2…转为为 16、32、48…(即编号*sizeof(zval))
        if (opline->op1_type == IS_CONST) {
            ZEND_PASS_TWO_UPDATE_CONSTANT(op_array, opline->op1);
```

```
            } else if (opline->op1_type & (IS_VAR|IS_TMP_VAR)) {
                //上面进行相同的处理，不同的是这里的起始值是接着 IS_CV 的
                opline->op1.var = (uint32_t)(zend_intptr_t)ZEND_CALL_VAR_NUM
(NULL, op_array->last_var + opline->op1.var);
            }
            //op2、result 操作数的处理与 op1 相同
            ...
            //设置此 opcode 的处理 handler
            ZEND_VM_SET_OPCODE_HANDLER(opline);
            opline++;
        }
        //标识当前 op_array 已执行过 pass_two()
        op_array->fn_flags |= ZEND_ACC_DONE_PASS_TWO;
        return 0;
    }
```

从上面过程可以很直观地看到，VAR、TMP_VAR 操作数通过 `(uint32_t)(zend_intptr_t)ZEND_CALL_VAR_NUM(NULL, op_array->last_var + opline->op1|2.var)` 计算得到，与 CV 的计算完全一致，5.3.2 节示例中只有两条赋值语句的返回值为 VAR 类型，其最终的操作数就是 128、144。

这里介绍的生成的 CONST 类型的操作数是相对值，也就是相对 zend_op_array->literals 起始位置的字节，执行时需要根据 zend_op_array->literals 与 znode_op.constant（也就是内存偏移）获取对应数据，即 `(zval*)(((char*)(zend_op_array->literals)) + znode_op.constant)`，这是在 64 位系统上的处理方式，但是在 32 位系统上采用的并不是这种相对值的方式，而是绝对值，也就是操作数中直接保存 zend_op_array->literals 数组中对应 zval 的地址，看一下 32 位系统下 znode_op 结构：

```
//32 位系统
typedef union _znode_op {
    uint32_t     constant;
    uint32_t     var;
    uint32_t     num;
    uint32_t     opline_num;
    zend_op      *jmp_addr;
    zval         *zv;
} znode_op;
```

znode_op.zv 就是字面量的地址。除了 CONST 操作数在 32/64 位系统有差异，jmp 指令的跳转值也是如此：在 32 位系统上，跳转指令采用的也是绝对值，也就是根据要跳转到的 znode_op

的地址跳转，即 znode_op.jmp_addr；在 64 位系统上，采用的是相对值，即 znode_op.jmp_offset。在不同平台上，需要注意这两个地方的差异，调试时根据不同平台的实现具体处理。

pass_two()中还有一个重要的处理，那就是根据 opcode 与操作数设置每条指令的处理 handler，这是非常关键的一步，否则是无法执行的。这个过程这里先不介绍，知道这一步是在 pass_two()中完成的即可，等到下一节介绍 ZendVM 执行器时再作说明。

```
ZEND_VM_SET_OPCODE_HANDLER(opline);
```

经过 pass_two()的处理后，5.3.2 节示例中的 opline 指令被最终加工完成，如图 5-22 所示。到目前为止，整个编译过程就全部完成了，最终编译出的 zend_op_array 将传到执行器中执行。

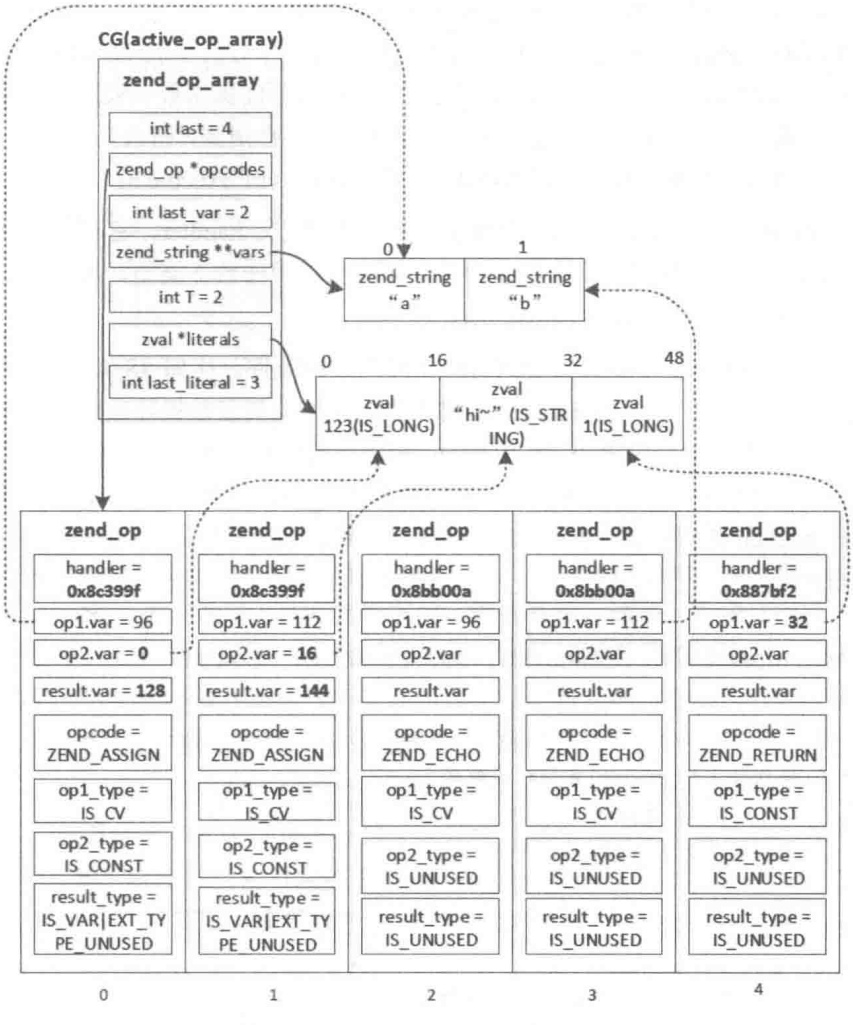


图 5-22　pass_two()处理后的 opline

5.4 PHP 的执行

ZendVM 执行器由指令处理 handler 与调度器组成：指令处理 handler 是每条 opcode 定义的具体处理过程，根据操作数的不同类型，每种 opcode 可定义 25 个 handler，执行时，由执行器调用相应的 handler 完成指令的处理；调度器负责控制指令的执行，以及执行器的上下文切换，由调度器发起 handler 的执行。

5.4.1 handler 的定义

所有 opcode 的处理 handler 定义在 Zend/zend_vm_def.h 中，定义时不需要为每一条 opcode 定义 25 个 handler，Zend 提供了一个脚本用于生成不同操作数类型的 handler，我们只需要在 zend_vm_def.h 为 opcode 定义一个 handler 即可，然后根据不同操作数类型区分处理即可。也就是说，zend_vm_def.h 只是 handler 的定义文件，编译时并不会用到，修改后需要在 Zend 目录下执行 zend_vm_gen.php 脚本生成实际的 handler 文件：zend_vm_execute.h。

每一条 opcode 都需要通过 ZEND_VM_HANDLER()定义 handler，它有四个参数，分别是 opcode 值、opcode 名、可接受的操作数 1 类型、可接受的操作数 2 类型，最后 gen 脚本根据支持的操作数类型生成不同类型组合的 handler。

在 zend_vm_execute.h 中，经常看到有些怪异的判断条件，比如 IS_CONST == IS_CV、IS_CONST == IS_CONST 等，这些就是 gen 脚本根据不同的类型替换导致的。也就是说，我们在 zend_vm_def.h 中兼容不同的操作数，然后 gen 脚本将每一种操作数组合替换生成一个 handler，这就导致生成的 handler 中会有部分逻辑是其他操作数组合的，当前 handler 不会用到。比如下面的例子，该 opcode 的操作数 1 可以接受 CONST、CV 两种类型，handler 定义中针对不同的类型进行处理：

```
ZEND_VM_HANDLER(196, ZEND_TEST_OP, CONST|CV, CONST)
{
    if (OP1_TYPE == IS_CONST){
        //操作数 1 为 const 类型时的处理
    } else if (OP1_TYPE == IS_CV) {
        //操作数 1 为 cv 类型时的处理
    }
}
```

这个例子中将生成两个 handler，分别是 CONST_CONST、CV_CONST 组合，gen 脚本在生成不同组合的 handler 时会把 OP1_TYPE 替换为对应类型，生成的两个 handler 分别是：

```
    //op1:CONST op2:CONST
    static  ZEND_OPCODE_HANDLER_RET  ZEND_FASTCALL  ZEND_TEST_OP_SPEC_CONST_
CONST_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
    {
        if (IS_CONST == IS_CONST){
            //操作数 1 为 const 类型时的处理
        } else if (IS_CONST == IS_CV) {//对本 handler 而言，这个分支是无用的
            //操作数 1 为 cv 类型时的处理
        }
    }
    //op1:CV op2:CONST
    static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_TEST_OP_SPEC_CV_CONST_
HANDLER(ZEND_OPCODE_HANDLER_ARGS)
    {
        if (IS_CV == IS_CONST){ //对本 handler 而言，这个分支是无用的
            //操作数 1 为 const 类型时的处理
        } else if (IS_CV == IS_CV) {
            //操作数 1 为 cv 类型时的处理
        }
    }
```

所以，这些永远不可能成立的判断条件是其他操作数组合用到的，它们实际上是多余的。那么为什么要把每条 opcode 根据不同操作数生成不同的 handler 呢？直接像 zend_vm_def.h 定义的那样，在一个 handler 中兼容不可以吗？这是因为，如果在一个 handler 中根据不同类型兼容处理，那么就会导致这个 handler 的逻辑非常庞大，执行时需要动态地判断，需要把所有类型的判断情况全部走一遍，这在效率上是非常低下的，而拆开后，这些一定成立以及一定不成立的代码会被编译器优化掉。也就是说，这些无用的代码在编译时会被忽略掉，从而大大降低了函数的大小。另外一方面，按操作数粒度拆开后可以方便针对某些特殊操作数进行优化，比如在 JIT 里，就可以针对特定的操作数组合进行 handler 替换。

另外，zend_vm_gen.php 提供了部分宏，用于屏蔽不同类型之间操作的差异，通过这些宏，我们不需要再根据各种操作数类型兼容处理，该脚本会替换为各类型具体的处理方法，当然这仅限于部分操作，并不意味着我们不需要关心不同操作数的处理。比如 GET_OP2_ZVAL_PTR()，用于获取操作数 zval 地址的宏，操作数 2（也就是变量值）的类型会被相应地替换为以下值：

```
$op2_get_zval_ptr = array(
    "ANY"    => "get_zval_ptr(opline->op2_type, opline->op2, execute_data,
```

```
&free_op2, \\1)",
        "TMP"    => "_get_zval_ptr_tmp(opline->op2.var, execute_data, &free_op2)",
        "VAR"    => "_get_zval_ptr_var(opline->op2.var, execute_data, &free_op2)",
        "CONST"  => "EX_CONSTANT(opline->op2)",
        "UNUSED" => "NULL",
        "CV"     => "_get_zval_ptr_cv_\\1(execute_data, opline->op2.var)",
        "TMPVAR" => "_get_zval_ptr_var(opline->op2.var, execute_data, &free_
op2)",
    );
```

生成的 8 个 handler 中，会使用各类型对应的方法。`$a = 123` 这种语句将通过 EX_CONSTANT(opline->op2) 获取变量值 123 的 zval 地址，`$a = $b` 则将通过 _get_zval_ptr_cv_BP_VAR_R(execute_data, opline->op2.var) 获取。关于 zend_vm_gen.php 具体的使用规则这里不再细讲，有兴趣的读者可以根据现有 opcode 的定义规则自行研究。

5.4.2 调度方式

handler 是每条 opcode 对应的 C 语言编写的处理逻辑，那么这个处理逻辑是通过什么样的形式提供的呢？ZendVM 执行器的调度方式有三种不同的实现：CALL、SWITCH、GOTO，默认的方式为 CALL。三种模式效率是不同的，GOTO 最快，如果想选用其他两种，则需要执行 zend_vm_gen.php 脚本重新生成，通过`--with-vm-kind=CALL|SWITCH|GOTO` 参数指定使用的方式，本书介绍的内容均为默认的 CALL 方式。

- **CALL**：这种方式是将各 opcode 定义的 handler 封装为独立的 C 语言函数，执行指令时依次调用不同的函数进行处理。
- **SWITCH**：这种方式是将所有 opcode 的处理逻辑定义在一个函数中，通过 switch 的不同分支进行调度执行。
- **GOTO**：这种方式与 SWITCH 类似，也是将所有 opcode 的处理逻辑定义在一个函数中，与 SWITCH 不同的是，它通过不同的 label 标签区分，执行时根据 opcode 跳转到对应 label 处执行。

假设 opcode 数组如下：

```
int op_array[] = {
    opcode_1,
    opcode_2,
    opcode_3,
```

```
    ...
};
```

不同调度方式的工作过程如下：

```
//CALL 模式
void opcode_1_handler() {...}
void opcode_2_handler() {...}
...
void execute(int []op_array)
{
    void *opcode_handler_list[] = {&opcode_1_handler, &opcode_2_handler, ...};
    while(1){
        void handler = opcode_handler_list[op_array[i]];
        handler(); //call handler
        i++;
    }
}

//GOTO 模式
void execute(int []op_array)
{
    while(1){
        goto opcode_xx_handler_label;
    }

opcode_1_handler_label:
    ...
opcode_2_handler_label:
    ...
...
}

//SWITCH 模式
void execute(int []op_array)
{
    while(1){
        switch(op_array[i]){
```

```
            case opcode_1:
                ...
            case opcode_2:
                ...
        }
        i++;
    }
}
```

CALL 方式下，所有 opcode 定义的 handler 按照固定的排列规则保存在数组中，如果 opcode 用不了 25 个 handler，则可以把无用的 handler 定义为空操作：ZEND_NOP_SPEC_HANDLER。25 个 handler 在数组中的排列规则是：按 CONST、TMP、VAR、UNUSED、CV 的顺序依次两两组合，首先是 CONST 与 5 种类型的组合，接着是 TMP、VAR、UNUSED、CV 与 5 种类型的组合。

```
//file: zend_vm_execute.h
static const void **zend_opcode_handlers;

void zend_init_opcodes_handlers(void)
{
    static const void *labels[] = {
        ZEND_NOP_SPEC_HANDLER,
        ZEND_NOP_SPEC_HANDLER,
        ...
    };
    zend_opcode_handlers = labels;
}
```

这个数组也是 zend_vm_gen.php 自动生成的，根据 opcode 及操作数类型，可通过 zend_vm_get_opcode_handler()方法获取对应的 handler，具体索引方法如下：

```
//file: zend_execute.c
#define _CONST_CODE  0
#define _TMP_CODE    1
#define _VAR_CODE    2
#define _UNUSED_CODE 3
#define _CV_CODE     4
//file: zend_vm_execute.h
```

```
static const void *zend_vm_get_opcode_handler(zend_uchar opcode, const
zend_op* op)
{
        //该数组是以操作数类型作为下标进行索引
        static const int zend_vm_decode[] = {
            _UNUSED_CODE, /* 0                */
            _CONST_CODE,  /* 1 = IS_CONST     */
            _TMP_CODE,    /* 2 = IS_TMP_VAR   */
            _UNUSED_CODE, /* 3                */
            _VAR_CODE,    /* 4 = IS_VAR       */
            _UNUSED_CODE, /* 5                */
            _UNUSED_CODE, /* 6                */
            _UNUSED_CODE, /* 7                */
            _UNUSED_CODE, /* 8 = IS_UNUSED    */
            _UNUSED_CODE, /* 9                */
            _UNUSED_CODE, /* 10               */
            _UNUSED_CODE, /* 11               */
            _UNUSED_CODE, /* 12               */
            _UNUSED_CODE, /* 13               */
            _UNUSED_CODE, /* 14               */
            _UNUSED_CODE, /* 15               */
            _CV_CODE      /* 16 = IS_CV       */
        };
        //根据 op1_type、op2_type、opcode 得到对应的 handler
        return  zend_opcode_handlers[opcode  *  25  +  zend_vm_decode[op->
op1_type] * 5 + zend_vm_decode[op->op2_type]];
    }
```

操作数类型是 1、2、4、8、16，zend_vm_decode 这个数组的作用实际上只是将其映射为 0、1、2、3、4，即 1→0、2→1、4→2、8→3、16→4，因此这个映射数组是 16 个元素，利用其下标进行映射，实际这个数组中只有 1、2、4、8、16 几个元素会用到，其他的都是无用的。上一节介绍的编译过程，最后在 pass_two()中会遍历所有的 opline 指令设置 handler，那个地方就是调用 zend_vm_set_opcode_handler()完成的。

5.4.3　执行流程

zend_execute_data 是 ZendVM 执行过程中非常重要的一个数据结构，它主要有两个作用：

记录运行时信息、分配动态变量内存。在执行开始前，首先会分配一个 zend_execute_data 结构，然后将 zend_execute_data->opline 指向 zend_op_array->opcodes 第一条指令。随 zend_execute_data 结构分配的，还有当前 zend_op_array 中用到的 CV、VAR、TMP_VAR 变量。分配完成后，执行器从 zend_execute_data->opline 开始执行，执行完以后该指针将指向下一条指令继续执行，所有指令执行完以后，再把 zend_execute_data 释放掉。整个过程与 C 程序的执行机制非常相像，zend_execute_data 就扮演了 C 程序执行时 Stack 的角色，PHP 中局部变量的分配与访问也同 C 语言的处理一致。

在分析具体的执行流程前，我们先来明确几个概念。PHP 与 C、Java 这些语言不同，它不需要定义 main 函数，从 PHP 脚本开始位置直接执行，这里我们把 PHP 函数、类之外的代码统称为主代码（main code），我们本节介绍的就是这种类型。在 PHP 代码中定义的函数称为用户自定义函数，与此对应的内核或扩展定义的函数称为内部函数。明确了主代码、用户自定义函数、内部函数的概念后，接下来我们看一下主代码的具体执行过程，执行的入口函数为 zend_execute()：

```
ZEND_API void zend_execute(zend_op_array *op_array, zval *return_value)
{
    zend_execute_data *execute_data;
    ...
    //分配 zend_execute_data
    execute_data = zend_vm_stack_push_call_frame(ZEND_CALL_TOP_CODE,
            (zend_function*)op_array, 0, zend_get_called_scope(EG(current_
execute_data)), zend_get_this_object(EG(current_execute_data)));
    ...
    // execute_data->prev_execute_data = EG(current_execute_data);
    EX(prev_execute_data) = EG(current_execute_data);
    //初始化 execute_data
    i_init_execute_data(execute_data, op_array, return_value);
    //执行 opcode
    zend_execute_ex(execute_data);
    zend_vm_stack_free_call_frame(execute_data);
}
```

下面具体介绍执行过程中的几个重要步骤。

1）分配 zend_execute_data

通过 zend_vm_stack_push_call_frame()方法分配一块用于当前作用域的内存空间，也就是 zend_execute_data，分配时会根据 zend_op_array->last_var、zend_op_array->T 计算出动态变量区

的内存大小，随 zend_execute_data 一起分配。

```
//file: zend_execute.h zend_vm_stack_push_call_frame:
//计算需要分配的内存大小
uint32_t used_stack = zend_vm_calc_used_stack(num_args, func);
```

zend_vm_calc_used_stack()的参数中 num_args 为实际传入参数，在函数分配 zend_execute_data 时用到，func->op_array.num_args 为全部参数数量，所以 MIN(func->op_array.num_args, num_args)等于 num_args，在自定义函数中 used_stack=ZEND_CALL_FRAME_SLOT + func->op_array.last_var + func->op_array.T，而在调用内部函数时则只需要分配实际传入参数的空间即可，内部函数不会有临时变量的概念，具体的情况在下一章介绍函数的相关实现时再展开说明。

```
//zend_execute.h
static zend_always_inline uint32_t zend_vm_calc_used_stack(uint32_t
num_args, zend_function *func)
{
    uint32_t used_stack = ZEND_CALL_FRAME_SLOT + num_args;
    //PHP 用户代码
    if (EXPECTED(ZEND_USER_CODE(func->type))) {
        used_stack += func->op_array.last_var + func->op_array.T - MIN(func->
op_array.num_args, num_args);

    }
    return used_stack * sizeof(zval);
}
//zend_compile.h
#define ZEND_CALL_FRAME_SLOT \
    ((int)((ZEND_MM_ALIGNED_SIZE(sizeof(zend_execute_data)) + ZEND_MM_
ALIGNED_SIZE(sizeof(zval)) - 1) / ZEND_MM_ALIGNED_SIZE(sizeof(zval))))
```

仍然以 5.3.2 节的示例为例：last_var=2，T=2，此时计算出的 used_stack 为 160 字节，其中 zend_execute_data 自己对齐后占 96 字节、CV 变量（a、b）占 32 字节、VAR 变量占 32 字节。zend_execute_data 分配完成后会将 func 成员指向要执行的 zend_op_array，func 类型是 zned_function，这是一个 union，其中一个成员为 zend_op_array 类型，主代码用的就是这个。分配完成后，zend_execute_data 的内存分布以及与 zend_op_array 的关系如图 5-23 所示。

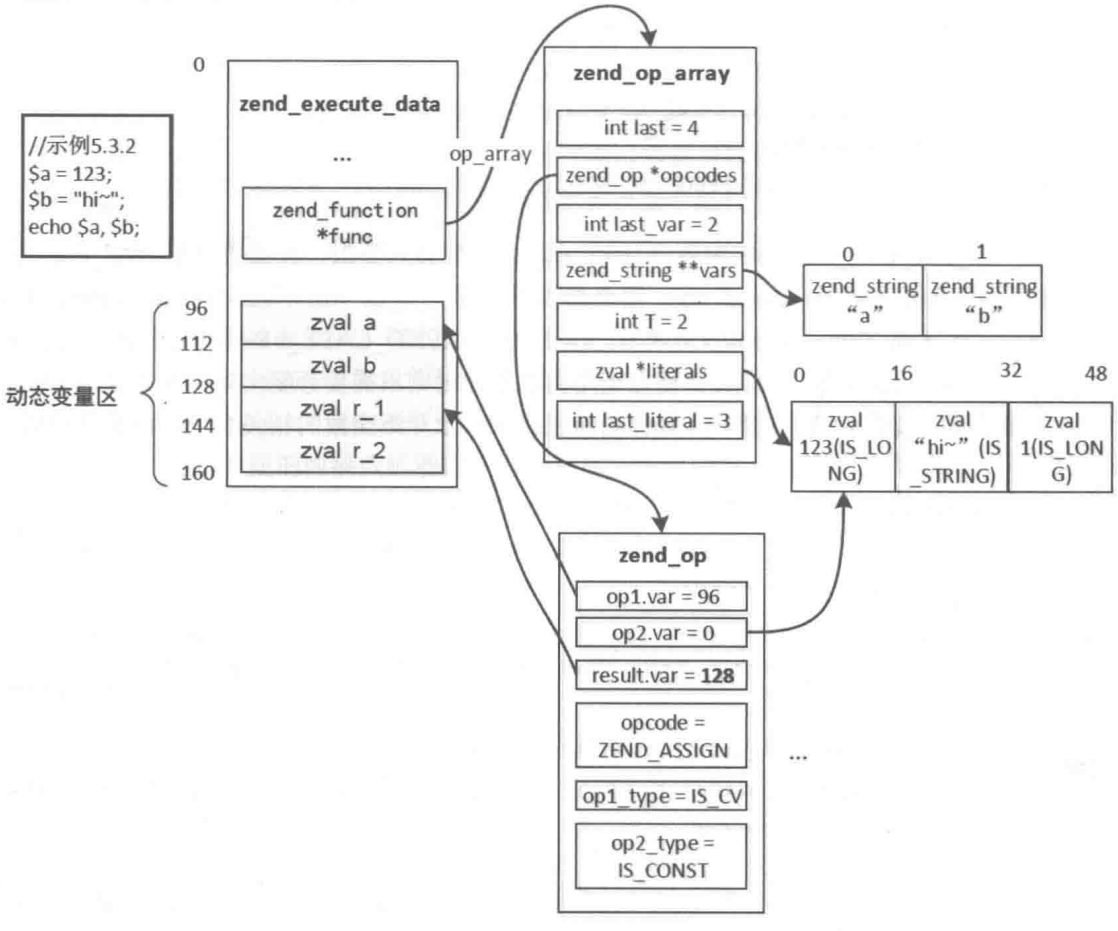


图 5-23 zend_execute_data 动态变量内存分布

在分配 zend_execute_data 内存时，传入了一个 ZEND_CALL_TOP_CODE 参数，这个值是用来标识执行器的调用类型的，因为 PHP 主代码执行、调用用户自定义函数、调用内部函数、include/require/eval 的执行流程都是相似的，都会分配 zend_execute_data，因此，zend_execute_data 通过 This 这个成员，将调用类型保存下来，This 类型是 zval，具体保存在 This.u1.v.reserved 中。

```
typedef enum _zend_call_kind {
    ZEND_CALL_NESTED_FUNCTION,  /* 调用用户自定义函数 */
    ZEND_CALL_NESTED_CODE,      /*  include/require/eval */
    ZEND_CALL_TOP_FUNCTION,     /* 调用内部函数 */
    ZEND_CALL_TOP_CODE          /* 调用主脚本 */
} zend_call_kind;
```

2）初始化 zend_execute_data

这一步主要是初始化 zend_execute_data 中的一些成员，比如 opline、return_value。另外还有一个比较重要的操作：zend_attach_symbol_table()，这是一个哈希表，用于存储全局变量。需要注意的是，主代码中的全部 CV 变量都是全局变量，尽管这些变量相对主代码而言是局部变量，但是对于其他函数，它们就是全局变量，可以在函数中通过 global 关键字访问到，关于全局变量相关的实现，我们在第 9 章会详细说明，这里暂时忽略。

```
//file: zend_execute.c
static  zend_always_inline  void  i_init_execute_data(zend_execute_data
*execute_data, zend_op_array *op_array, zval *return_value)
{
    EX(opline) = op_array->opcodes;
    EX(call) = NULL;
    EX(return_value) = return_value;

    if (UNEXPECTED(EX(symbol_table) != NULL)) {
        ...
        //添加全局变量
        zend_attach_symbol_table(execute_data);
    } else {
        ...
    }
    EX_LOAD_RUN_TIME_CACHE(op_array);
    EX_LOAD_LITERALS(op_array);

    EG(current_execute_data) = execute_data;
    ZEND_VM_INTERRUPT_CHECK();
}
```

经过这一步的处理后，zend_execute_data->opline 指向了第一条指令，同时将 zend_execute_data->literals 指向了 zend_op_array->literals，便于快速访问，完成这些设置后，最后将 EG(current_execute_data)指向 zend_execute_data。现在，执行前的准备工作都已经完成了，zend_execute_data 执行前的状态如图 5-24 所示。

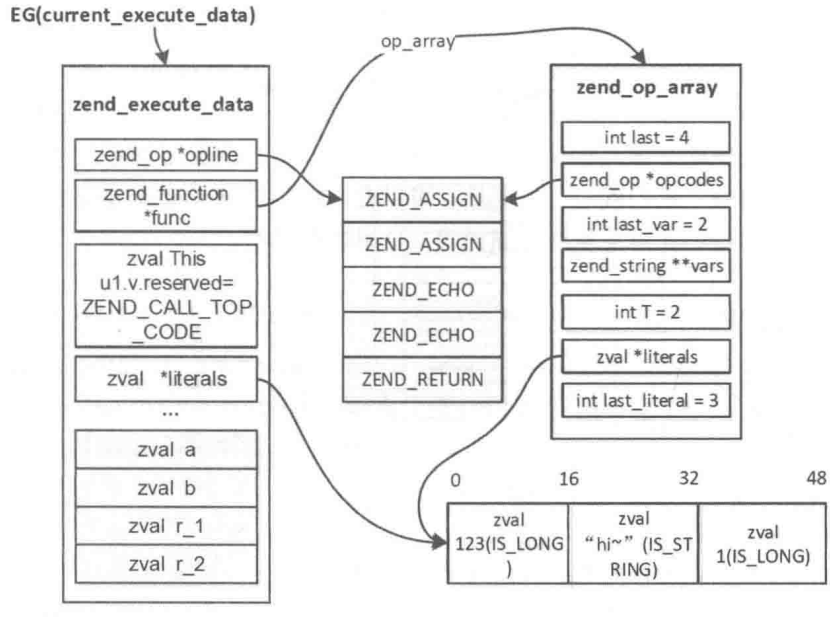


图 5-24　zend_execute_data 执行前状态

3）执行

ZendVM 的执行调度器就是一个 while 循环，在这个循环中依次调用 opline 指令的 handler，然后根据 handler 的返回决定下一步的动作。执行调度器为 zend_execute_ex，这是函数指针，默认为 execute_ex()，可以通过扩展进行覆盖。GCC 低于 4.8 的情况下，这个调度器展开后如下：

```
ZEND_API void execute_ex(zend_execute_data *ex)
{
    zend_execute_data *execute_data = ex;

    while (1) {
        int ret;
        //执行当前指令
        if (UNEXPECTED((ret = ((opcode_handler_t)execute_data->opline->
handler)(execute_data)) != 0)) {
            if (EXPECTED(ret > 0)) {
                execute_data = EG(current_execute_data);
            } else {
                return;
            }
        }
```

```
    }
}
```

执行的第一条 opcode 为 ZEND_ASSIGN，即$a = 123，该指令由 ZEND_ASSIGN_SPEC_CV_CONST_HANDLER()处理，首先根据操作数 1、2 取出赋值的变量与变量值，其中变量值为 CONST 类型，保存在 literals 中，通过 EX_CONSTANT(opline->op2)获取它的值。$a 为 CV 变量，分配在 zend_execute_data 动态变量区，通过_get_zval_ptr_cv_undef_BP_VAR_W()取到这个变量的地址，剩下的处理就很好理解了，将变量值复制到 CV 变量即可，具体的处理过程还会有一些特殊操作，这里不再细说。

```
static int ZEND_ASSIGN_SPEC_CV_CONST_HANDLER(zend_execute_data *execute_data)
{
    //USE_OPLINE
    const zend_op *opline = execute_data->opline;
    ...
    value = EX_CONSTANT(opline->op2);
    variable_ptr = _get_zval_ptr_cv_undef_BP_VAR_W(execute_data, opline->
op1.var);
    ...
        //赋值复制
        value = zend_assign_to_variable(variable_ptr, value, IS_CONST);
    ...
    //ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION()
    execute_data->opline = execute_data->opline + 1;
    return 0;
}
```

执行完成后会更新 opline，指向下一条指令，然后返回到调度器。这里会有下面几个不同的动作，调度器根据不同的返回值决定下一步的动作。

```
#define ZEND_VM_CONTINUE()     return  0
#define ZEND_VM_ENTER()        return  1
#define ZEND_VM_LEAVE()        return  2
#define ZEND_VM_RETURN()       return -1
```

ZEND_VM_CONTINUE()表示继续执行下一条 opcode；ZEND_VM_ENTER()/ZEND_VM_LEAVE()是调用函数时的动作，普通模式下 ZEND_VM_ENTER()实际上就是 return 1，然后

execute_ex()中会将 execute_data 切换到被调函数的结构上。对应的，在函数调用完成后 ZEND_VM_LEAVE()会 return 2，再将 execute_data 切换至原来的结构；ZEND_VM_RETURN()表示执行完成，返回-1 给 execute_ex()，比如 exit，这时候 execute_ex()将退出执行。

这就是执行器的基本执行过程，对于函数调用会重新分配一个 zend_execute_data，然后将调度器切至被调函数的 zend_execute_data，执行完以后再跳回去接着执行下面指令，详细的流程下一章介绍函数时再说明，这里先简单了解下。

最后一条执行的指令是 ZEND_RETURN，这条执行会有几个非常关键的处理，第一个就是设置返回值，zend_execute()执行时传入了一个 return_value，这个过程相当于赋值，ZendVM 会把返回值赋给这个地址；另外一个就是清理动态变量区，对 zend_execute_data 上的 CV、VAR、TMP_VAR 进行清理，需要注意的是，这里清理的不包括全局变量，也就是说主脚本、include 的调用并不会在这一步对全局变量（即主代码中的“局部变量”）进行清理；还有一个处理是切换至调用前的 zend_execute_data，相当于汇编中 ret 指令的作用。不同 call_info 场景下，ZEND_RETURN 的处理也不相同，后面介绍函数及 include 的实现时再展开。

```
    static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL zend_leave_helper_SPEC(ZEND_
OPCODE_HANDLER_ARGS)
    {
        zend_execute_data *old_execute_data;
        //获取 zend_execute_data 的 call_info
        uint32_t call_info = EX_CALL_INFO();

        //调用用户自定义函数
        if (EXPECTED(ZEND_CALL_KIND_EX(call_info) == ZEND_CALL_NESTED_FUNCTION)) {
            ...
        }
        if (EXPECTED((ZEND_CALL_KIND_EX(call_info) & ZEND_CALL_TOP) == 0)) {
            ...
        } else {
            if (ZEND_CALL_KIND_EX(call_info) == ZEND_CALL_TOP_FUNCTION) {
                ...
            } else /* if (call_kind == ZEND_CALL_TOP_CODE) */ {
                zend_array *symbol_table = EX(symbol_table);

                zend_detach_symbol_table(execute_data);
                old_execute_data = EX(prev_execute_data);
                ...
```

```
            //还原 zend_execute_data
            EG(current_execute_data) = EX(prev_execute_data);
        }
        ZEND_VM_RETURN();
    }
}
```

4）释放 zend_execute_data

在所有的指令执行完成后，将调用 zend_vm_stack_free_call_frame()释放 zend_execute_data。为了避免频繁地申请、释放 zend_execute_data，这里会对 zend_execute_data 进行缓存，释放时并不是立即归还底层的内存系统（即 Zend 内存池），而是暂时保留，下次使用时直接分配，缓存位置为 EG(vm_stack)，这部分比较简单，不再展开。

5.4.4　全局 execute_data 和 opline

Zend 执行器在 opcode 的执行过程中，会频繁地用到 execute_data 和 opline 两个变量。普通的处理方式在执行每条 opline 指令的 handler 时，把 execute_data 地址作为参数传给 handler，根据 C 程序执行机制，传参通过入栈的方式传给被调函数，调用结束后再出栈，这种方式下 Zend 执行器展开后如下：

```
ZEND_API void execute_ex(zend_execute_data *ex)
{
    zend_execute_data *execute_data = ex;

    while (1) {
        int ret;

        if (UNEXPECTED((ret = ((opcode_handler_t)execute_data->opline->
handler)(execute_data)) != 0)) {
            if (EXPECTED(ret > 0)) {
                execute_data = EG(current_execute_data);
            } else {
                return;
            }
        }
    }
}
```

关于 execute_ex()上一节已经介绍过了，它是一个大循环，执行前 execute_data->opline 指向第一条指令，执行完以后该指针指向下一条指令，execute_data->opline 类似 eip 寄存器的作用，通过这个循环，ZendVM 完成 opcode 指令的执行。这个处理过程比较简单，并没有不好理解的地方，而且整个过程看起来也都那么顺理成章。

PHP7 针对 execute_data、opline 两个变量地址的存储方式进行了优化，使用全局寄存器变量保存它们的地址，以实现更高效率的读取。这种方式下直接从寄存器读取 execute_data、opline 的地址，比从栈上读取更快，同时省掉了传参的出入栈流程，在性能上大概有 5%的提升。在分析 PHP7 的优化之前，我们先简单介绍下什么是寄存器变量。

寄存器变量存放在 CPU 的寄存器中，使用时，不需要访问内存，直接从寄存器中读写。与存储在内存中的变量相比，寄存器变量具有更快的访问速度。在计算机的存储层次中，寄存器的速度最快，其次是内存，最慢的是硬盘。C 语言中使用关键字 register 来声明局部变量为寄存器变量。需要注意的是，只有局部自动变量和形式参数才能够被定义为寄存器变量，全局变量和局部静态变量都不能被定义为寄存器变量。而且，一个计算机中寄存器数量是有限的，一般为 2 到 3 个，因此寄存器变量的数量不能太多。对于在一个函数中声明的多于 2 到 3 个的寄存器变量，C 编译程序会自动将寄存器变量变为自动变量。受硬件寄存器长度的限制，寄存器变量只能是 char、int 或指针型，而不能使其他复杂数据类型。由于 register 变量使用的是硬件 CPU 中的寄存器，寄存器变量无地址，所以不能使用取地址运算符"&"求寄存器变量的地址。

GCC 在 3.4 版本支持了一项新特性：全局寄存器变量（Global Register Variables），也就是可以把全局变量定义为寄存器变量，从而可以实现函数间共享数据。可以通过下面的语法告诉编译器使用寄存器来保存数据：

```
register int *foo asm ("r12"); //r12、%r12
//或
register int *foo __asm__ ("r12"); //r12、%r12
```

这里 r12 就是指定使用的寄存器，它必须是运行平台上有效的寄存器，这样就可以像使用普通的变量一样使用 foo。但是 foo 同样没有地址，也就是无法通过&获取它的地址，在 gdb 调试时也无法使用 foo 符号，只能使用对应的寄存器获取数据。举个例子来看：

```
//main.c
#include <stdlib.h>

typedef struct _execute_data {
    int ip;
```

```
}zend_execute_data;

register zend_execute_data* execute_data __asm__ ("%r14");

int main(void)
{
    execute_data = (zend_execute_data *)malloc(sizeof(zend_execute_data));
    execute_data->ip = 9999;

    return 0;
}
```

编译：gcc -o main -g main.c，然后通过 gdb 调试寄存器的存储内容。

```
$ gdb main
(gdb) break main
(gdb) r
Starting program: /home/qinpeng/c/php/main

Breakpoint 1, main () at main.c:12
12    execute_data = (zend_execute_data *)malloc(sizeof(zend_execute_data));
(gdb) n
13     execute_data->ip = 9999;
(gdb) n
15     return 0;
```

这时我们就无法再像普通变量那样直接使用 execute_data 访问数据了，只能通过 r14 寄存器读取，直接访问 execute_data 将提示找不到这个符号，这里可以通过 (zend_execute_data *)$r14 获取保存的 zend_execute_data 结构变量的地址：

```
(gdb) p execute_data
Missing ELF symbol "execute_data".
(gdb) info register r14
r14            0x601010 6295568
(gdb) p ((zend_execute_data *)$r14)->ip
$3 = 9999
```

了解完全局寄存器变量，接下来我们再回头看一下 PHP7 中的用法，处理也比较简单，就

是在 execute_ex()执行各 opcode 指令的过程中，不再将 execute_data 作为参数传给 handler，而是通过寄存器保存 execute_data 及 opline 的地址，handler 使用时直接从全局变量（寄存器）读取，执行完再把下一条指令更新到全局变量。

因为 GCC 在 4.8 版本之前的版本对全局寄存器变量的支持并不完善，因此 PHP 在 GCC 4.8+ 以上版本才支持，默认自动开启，可以通过 --disable-gcc-global-regs 编译参数关闭。以 x86_64 为例，execute_data 使用 r14 寄存器，opline 使用 r15 寄存器：

```
//file: zend_execute.c line: 2631
# define ZEND_VM_FP_GLOBAL_REG "%r14"
# define ZEND_VM_IP_GLOBAL_REG "%r15"

//file: zend_vm_execute.h line: 315
register zend_execute_data* volatile execute_data __asm__(ZEND_VM_FP_
GLOBAL_REG);
register const zend_op* volatile opline __asm__(ZEND_VM_IP_GLOBAL_REG);
```

execute_data、opline 定义为全局变量，下面看一下这种方式下调度函数 execute_ex()的变化，展开后：

```
ZEND_API void execute_ex(zend_execute_data ex)
{
    const zend_op orig_opline = opline;
    zend_execute_data *orig_execute_data = execute_data;

    //将当前 execute_data、opline 保存到全局变量
    execute_data = ex;
    opline = execute_data->opline

    while (1) {
        ((opcode_handler_t)opline->handler)();
        if (UNEXPECTED(!opline)) {
            execute_data = orig_execute_data;
            opline = orig_opline;
            return;
        }
    }
}
```

这个时候调用各 opcode 指令的 handler 时就不再传入 execute_data 的参数了，handler 使用时直接从全局变量读取，以赋值 ZEND_ASSIGN 指令为例，handler 展开后：

```
static int ZEND_ASSIGN_SPEC_CV_CONST_HANDLER(void)
{
    ...
    //ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION()
    opline = execute_data->opline + 1;
    return;
}
```

在 handler 中直接访问全局变量 execute_data、opline，这两种处理方式并没有本质上的差异，只是通过全局寄存器变量提升了一些性能。如果你编译的 PHP 使用这种方式，那么在调试 execute_ex()与各个 handler 时就需要从寄存器读取了，无法再直接使用 execute_data、opline。

> automake 编译时的命令是 cc，而不是 gcc，Linux 下 cc 实际指向了 gcc，如果更新 gcc 后发现 PHP 仍然没有支持这个特性，请检查 cc 是否指向了新的 gcc。

5.5　运行时缓存

本节的内容会提前涉及函数、成员属性、成员方法、常量、全局变量等相关内容，这些部分会在后续章节一一介绍，如果有不太明白的地方也可以在了解完后续部分后再看本节的内容。

在本节开始之前，我们先分析一个普通 PHP 语法的例子：

```
class my_class {
    public $id = 123;

    public function test() {
        echo $this->id;
    }
}

$obj = new my_class;
for($i = 0; $i < 10; $i++) {
    $obj->test();
}
```

这个例子定义了一个类，然后多次调用同一个成员方法，这个成员方法的功能很简单：输出一个成员属性。首先简单介绍成员属性的相关实现：成员属性是定义在类中的一种数据，它的基本信息通过 zend_property_info 结构保存，包括成员属性的可见性（public、private、protected）、是否为静态属性、编号等，这些结构通过哈希结构管理，即类的 properties_info 成员。其中普通成员属性属于不同对象独有的数据，在对象实例化时按照各属性的编号依次分配在对象结构中，这个编号保存于 zend_property_info 结构中，在编译时确定，它的实现与局部变量的相同，可以参考 zend_execute_data 上局部变量的分配方式。

如果想访问对象的某个成员，则首先需要知道这个属性在对象中的存储位置，也就是 zend_property_info 结构中的 offset，然后再根据这个位置从对象的内存中索引到对应属性。因此，访问一个成员属性需要经历以下几步：

- step1：根据对象获取实例化类的 zend_class_entry 结构。
- step2：根据属性名称从 zend_class_entry.properties_info 哈希表中找到对应属性的 zend_property_info 结构。
- step3：最后根据属性的 offset 信息从 zend_object 中获取属性值。

那么问题来了：每次执行 test()时难道上面的过程都要完整走一遍吗？我们再仔细看一下这个过程，字面量“id”在“$this->id”此条语句中就是用来索引属性的，不管执行多少次它的任务始终是这个，那么有没有一种办法将“id”与查找到的 zend_class_entry、zend_property_info.offset 建立一种关联关系并保存下来，这样再次执行时直接根据“id”拿到前面关联的这两个数据，从而避免多次重复相同的工作呢？PHP 采用了一种机制来解决这种问题：运行时缓存。

在执行期间，有些操作经常需要根据名称去不同的哈希表中查找常量、函数、类、成员方法、成员属性等，运行时缓存就是用来缓存这些查找结果的，以便再次执行同一指令时直接复用上次缓存的值，无须重复查找，从而提高执行效率。所以，运行时缓存机制是在同一指令执行多次的情况下才会生效，它缓存的是根据操作数获取的数据。需要注意的是，这里的同一指令指的并不是 opcode 值相同的指令，例如：`echo $a; echo $a;`这种就不算，因为这是两条指令。

开始提到的那个例子中会缓存两个数据：zend_class_entry、zend_property_info.offset，再次执行这条指令时就直接取出上次缓存的两个值，不用再重复上面的查找过程。

运行时缓存只用于 CONST 操作数，也就是根据 CONST 操作数索引数据的操作，其他类型都不会用到。这是因为只有 CONST 操作数是固定不变的，其他 CV、VAR 等类型值都不是固定的，既然其值是不固定的，那么缓存的值也就无法固定。比如 `echo $this->$var` 这种，操作数类型是 CV，正常查找时的 zend_property_info 是随`$var` 值而变的，所以操作数与结果之间

没有固定的关联关系，而$this->id 中“id”是固定不变的，它索引到 zend_property_info 也是始终是不变的。

运行时缓存统一分配在一块内存上，各操作数通过不同的位置存取自己的缓存数据，这个位置是在编译时申请的：如果操作数需要用到缓存，则申请一个缓存位置，然后把这个位置保存到 CONST 变量的 zval.u2.cache_slot，在运行时就是根据这个位置到缓存空间中存取数据的。缓存位置的分配比较简单，通过 op_array->cache_size 记录可用位置，编译前它的初始化值为 0，当操作数申请缓存位置时，就将当前值返回，然后增大到申请的大小，最终这个值就是需要的全部缓存大小。在 ZendVM 执行前会根据 cache_size 大小分配内存，各指令根据编译时申请到的缓存位置到此空间存取数据，缓存空间位于 zend_op_array->run_time_cache，在 i_init_func_execute_data()操作中分配。

```
    static zend_always_inline void i_init_func_execute_data(zend_execute_data
*execute_data, zend_op_array *op_array, zval *return_value, int check_this)
    {
        ...
        if (UNEXPECTED(!op_array->run_time_cache)) {
            //分配全部的缓存空间
            op_array->run_time_cache = zend_arena_alloc(&CG(arena), op_array->
cache_size);
            memset(op_array->run_time_cache, 0, op_array->cache_size);
        }
        ...
    }
```

上面那个例子在执行时缓存的生效过程如下：

- 第一次执行 echo $this->id 时首先根据$this 获取示例化类的 zend_class_entry，然后根据“id”向 zend_class_entry.properties_info 哈希表查找属性 zend_property_info，找到后将此结构的 offset 及 zend_class_entry 的地址缓存到 test() 函数的 zend_op_array->run_time_cache 中。
- 再次执行 echo $this->id 时，根据 CONST 变量“id”，得到缓存位置 0，然后去 zend_op_array->run_time_cache 取出缓存的 zend_class_entry、offset。

这个例子中缓存了 16 字节的数据，对应的缓存结构如图 5-25 所示。

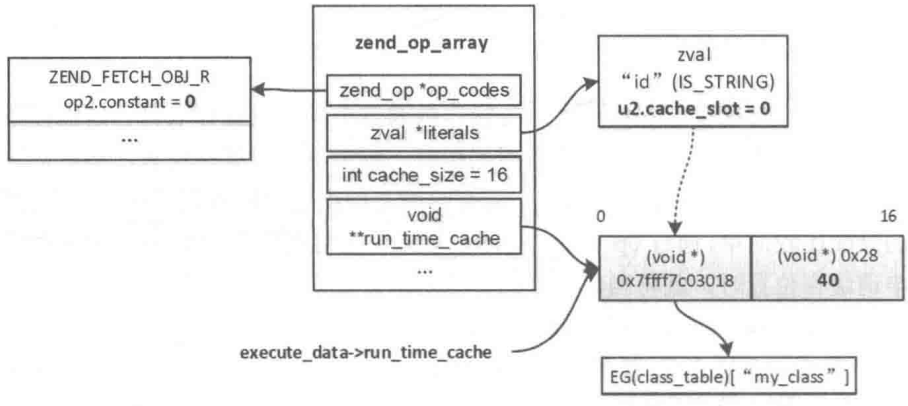


图 5-25 run_time_cache

接下来介绍几种常见的用到缓存的情况。

1）调用函数

例如：

```
function test(){
    time();
}
```

test()函数中调用 time()函数，执行时需要根据"time"到 EG(function_table)中查找 zend_function，这里"time"操作数就会缓存 zend_function。编译时的处理：

```
void zend_compile_call(znode *result, zend_ast *ast, uint32_t type)
{
    ...
    opline = zend_emit_op(NULL, ZEND_INIT_FCALL, NULL, &name_node);
    zend_alloc_cache_slot(opline->op2.constant);
    ...
}
```

zend_alloc_cache_slot()就是申请缓存的存储位置，这个函数申请的一个指针大小的缓存空间，即 8 字节。除了这个函数，还有另外一个相似的：zend_alloc_polymorphic_cache_slot()，这个函数申请的是 16 字节的空间，目前缓存只用到了这两种规格。

```
static inline void zend_alloc_cache_slot(uint32_t literal) {
    zend_op_array *op_array = CG(active_op_array);
```

```
    //op_array->cache_size 为可用位置，申请后向后偏移
    Z_CACHE_SLOT(op_array->literals[literal]) = op_array->cache_size;
    op_array->cache_size += sizeof(void*);
}

#define POLYMORPHIC_CACHE_SLOT_SIZE 2
static inline void zend_alloc_polymorphic_cache_slot(uint32_t literal) {
    zend_op_array *op_array = CG(active_op_array);
    Z_CACHE_SLOT(op_array->literals[literal]) = op_array->cache_size;
    //分配 16 个字节
    op_array->cache_size += POLYMORPHIC_CACHE_SLOT_SIZE * sizeof(void*);
}
```

ZEND_INIT_FCALL 指令需要根据函数名称索引到函数的 zend_function，执行时首先会检查缓存是否存在，如果存在则直接使用，如果不存在则按照正常的逻辑到 EG(function_table)中查找，然后把 zend_function 保存到缓存中，以便下次执行时直接使用。

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_INIT_FCALL_SPEC_CONST_
HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    USE_OPLINE

    zval *fname = EX_CONSTANT(opline->op2);
    zval *func;
    zend_function *fbc;
    zend_execute_data *call;
    //检查缓存是否存在
    fbc = CACHED_PTR(Z_CACHE_SLOT_P(fname));
    if (UNEXPECTED(fbc == NULL)) {
        //如果没有缓存则按普通方式查找
        func = zend_hash_find(EG(function_table), Z_STR_P(fname));
        ...
        fbc = Z_FUNC_P(func);
        //设置缓存
        CACHE_PTR(Z_CACHE_SLOT_P(fname), fbc);
    }
    ...
}
```

CACHE_PTR、CACHED_PTR 两个宏分别用于 8 字节规格缓存的设置、获取，与之对应的还有两个用于 16 字节规则规则的宏：CACHE_POLYMORPHIC_PTR、CACHED_POLYMORPHIC_PTR。

```
//file: zend_execute.h
#define CACHED_PTR(num) \
    ((void**)((char*)EX_RUN_TIME_CACHE() + (num)))[0]

#define CACHE_PTR(num, ptr) do { \
        ((void**)((char*)EX_RUN_TIME_CACHE() + (num)))[0] = (ptr); \
    } while (0)

#define CACHED_POLYMORPHIC_PTR(num, ce) \
    (EXPECTED(((void**)((char*)EX_RUN_TIME_CACHE() + (num)))[0] == (void*)
(ce)) ? \
        ((void**)((char*)EX_RUN_TIME_CACHE() + (num)))[1] : \
        NULL)

#define CACHE_POLYMORPHIC_PTR(num, ce, ptr) do { \
        void **slot = (void**)((char*)EX_RUN_TIME_CACHE() + (num)); \
        slot[0] = (ce); \
        slot[1] = (ptr); \
    } while (0)
```

2）访问常量、静态变量、全局变量

这几种数据都保存于单独的符号表中，访问时需要根据名称到对应的符号表中查找。以常量为例，编译时的处理：

```
void zend_compile_const(znode *result, zend_ast *ast)
{
    ...
    opline = zend_emit_op_tmp(result, ZEND_FETCH_CONSTANT, NULL, NULL);
    opline->op2_type = IS_CONST;
    ...
    //申请缓存位置
    zend_alloc_cache_slot(opline->op2.constant);
}
```

ZEND_FETCH_CONSTANT 指令执行时对缓存的操作不再展开，与函数的处理方式相同。

除了上面介绍的这些操作会用到缓存，还有很多其他的情况，这里不再一一列举。

5.6　Opcache

到目前为止，我们已经基本介绍完了 PHP 的编译与执行的内部实现，一个 request 从请求之初到最终处理完成总是需要经历编译、执行两个阶段，请求结束后关于这个请求的所有数据就被擦除了，下次请求时需要经历同样的过程。在这个过程中我们很容易发现一个问题：当多次请求同一个脚本时，需要重复经历编译的过程。只要脚本的内容没有发生变化，那么多次编译的结果实际上是相同的，这样就导致在重复的工作上花费了大量时间。Opcache 就是用来解决这个问题的，它把编译后的结果缓存下来，再次请求时不需要再重新编译，而是使用缓存的内容，直接进入执行阶段，从而大大提高了 PHP 的性能。

Opcache——一个用于缓存 opcodes 以提升 PHP 性能的 Zend 扩展，它的前身是 Zend Optimizer Plus（简称 O+），目前 Opcache 已经是 PHP 非常重要的一个组成部分，它对于 PHP 性能的提升有着非常显著的作用。PHP 编译生成的 opcodes 指令与 Java 编译生成的字节码的角色是相同的，Java 将字节码保存到了.class 中，而 Opcache 也可以把 opcodes 指令缓存到文件中，从而实现跨生命周期使用，从这个角度看 PHP 与 Java 是非常相似的。

Opcache 最主要的功能是缓存 PHP 代码编译生成的 opcodes 指令，它的工作原理比较容易理解：开启 Opcache 之后，它将 PHP 的编译函数 zend_compile_file 替换为 persistent_compile_file()，接管 PHP 代码的编译过程，当新的请求到达时，将调用 persistent_compile_file()进行编译。此时 Opcache 首先检查是否有该文件的缓存，如果有则取出，然后进行一系列的验证，如果最终发现缓存可用则直接返回，进入执行流程，如果没有缓存或缓存已经失效，则重新调用系统默认的编译器进行编译，然后将编译后的结果缓存下来，供下次使用。整体的处理流程如图 5-26 所示。

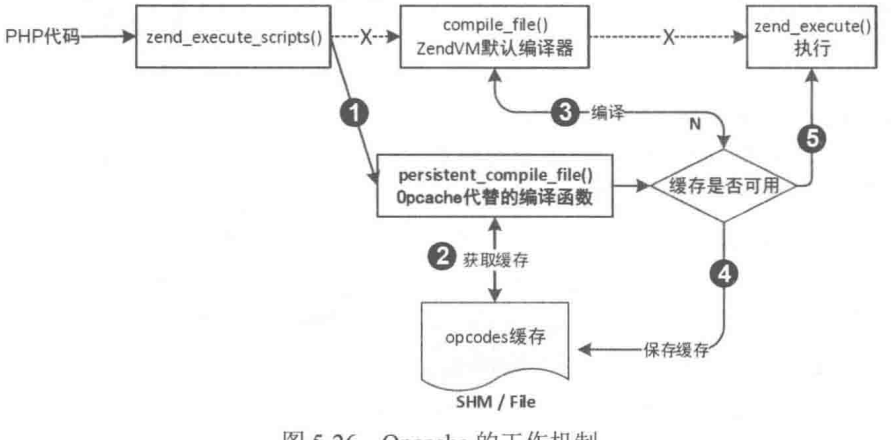


图 5-26　Opcache 的工作机制

1. 初始化

Opcache 是一个 Zend 扩展，在 PHP 的 module startup 阶段将调用 Zend 扩展的 startup 方法进行启动，Opcache 定义的 startup 方法为 accel_startup()，此时 Opcache 将覆盖一些 ZendVM 的函数句柄，其中最重要的一个就是上面提到的 zend_compile_file。具体过程如下：

```
//file: ext/opcache/ZendAccelerator.c
static int accel_startup(zend_extension *extension)
{
    ...
    //分配共享内存
    switch (zend_shared_alloc_startup(ZCG(accel_directives).memory_
consumption)) {
        case ALLOC_SUCCESS:
            //初始化，并分配一个 zend_accel_shared_globals 结构
            if (zend_accel_init_shm() == FAILURE) {
                accel_startup_ok = 0;
                return FAILURE;
            }
            break;
        ...
    }
    ...
    //覆盖编译函数: zend_compile_file
    accelerator_orig_compile_file = zend_compile_file;
    zend_compile_file = persistent_compile_file;
    ...
}
```

同时，该过程还会分配一个 zend_accel_shared_globals 结构，这个结构通过共享内存分配，进程间共享，它保存着所有的 opcodes 缓存信息，以及一些统计数据，ZCSG()宏操作的就是这个结构。这里暂且不关心其具体结构，只看 hash 这个成员，它就是存储 opcodes 缓存的地方。zend_accel_shared_globals->hash 是一个哈希表，但并不是 PHP 内核中的 HashTable，而是单独实现的一个，它的结构为 zend_accel_hash：

```
typedef struct _zend_accel_hash {
    zend_accel_hash_entry **hash_table;
    zend_accel_hash_entry  *hash_entries;
```

```
    uint32_t            num_entries; //已有元素数
    uint32_t            max_num_entries; //总容量
    uint32_t            num_direct_entries;
} zend_accel_hash;
```

其中 hash_table 用于保存具体的 value，它是一个数组，根据 hash_value % max_num_entries 索引存储位置，zend_accel_hash_entry 为具体的存储元素，这是一个单链表，用于解决哈希冲突；max_num_entries 为最大的元素数，也就是可缓存的文件数，这个值可以通过 php.ini 配置 opcache.max_accelerated_files 指定，默认值为 2000，但是实际的 max_num_entries 并不是直接使用的这个配置，而是根据配置值选择了一个固定规格，所有规格如下：

```
static uint prime_numbers[] =
    {5, 11, 19, 53, 107, 223, 463, 983, 1979, 3907, 7963, 16229, 32531, 65407,
130987, 262237, 524521, 1048793 };
```

实际选择的是可以满足 opcache.max_accelerated_files 的最小的一个，比如默认值 2000 最终使用的是 3907，也就是可以缓存 3907 个文件，超过了这个值 Opcache 将不再缓存。

2. 缓存的获取过程

编译一个脚本调用的是 zend_compile_file()，此时将由 Opcache 的 persistent_compile_file() 处理。这里首先介绍 Opcache 中比较重要的几个配置：

- **opcache.validate_timestamps**：是否开启缓存有效期验证，默认值为 1，表示开启，开启之后每隔 opcache.revalidate_freq 秒检查一次文件是否更新了；如果不开启则不会检查，脚本文件修改了只能重启服务才能生效。opcache.revalidate_freq 默认为 2s。
- **opcache.revalidate_path**：验证文件路径，默认值为 0，表示关闭。默认情况下，opcodes 缓存并不是通过完整的文件路径名称进行索引的，而是通过一个根据文件名、当前所在目录、include_path 生成的 key，因此当编译的文件实际已经不存在了但是缓存还在的时候，就会使用已经失效的缓存，如果开启这个选项，将通过完整的文件路径检索缓存，并且检查文件是否存在，而不再使用那个 key。

主要的处理流程如下：

```
zend_op_array *persistent_compile_file(zend_file_handle *file_handle, int
type)
{
    zend_persistent_script *persistent_script = NULL;
    ...
```

```
        //1) 获取缓存
            //如果没有开启 opcache.revalidate_path，则先根据 key 获取缓存
            if (!ZCG(accel_directives).revalidate_path) {
                key = accel_make_persistent_key(file_handle->filename, strlen
(file_handle->filename), &key_length);
                if (!key) {
                    return accelerator_orig_compile_file(file_handle, type);
                }
                //获取缓存
                persistent_script = zend_accel_hash_str_find(&ZCSG(hash), key,
key_length);
            }
            if (!persistent_script) {
                //如果取不到缓存或开启了 opcache.revalidate_path，则根据实际的文件路径
查找缓存
                ...
                bucket = zend_accel_hash_find_entry(&ZCSG(hash), file_handle->
opened_path);
                if (bucket) {
                    persistent_script = (zend_persistent_script *)bucket->data;
                    ...
                }
            }
        ...
        //2) 检查脚本是否更新过，开启 opcache.validate_timestamps 时
        if (persistent_script && ZCG(accel_directives).validate_timestamps) {
            //每隔 opcache.revalidate_freq 秒检查一次文件是否更新过，如果更新了则缓存失效
            if (validate_timestamp_and_record(persistent_script, file_handle)
== FAILURE) {
                ...
                persistent_script = NULL;
            }
        }
        //校验缓存数据是否合法：根据 Adler-32 算法，类似 crc 的一个算法
        if (persistent_script && ZCG(accel_directives).consistency_checks
            && persistent_script->dynamic_members.hits % ZCG(accel_directives).
consistency_checks == 0) {
            unsigned int checksum = zend_accel_script_checksum(persistent_
```

```
script);
            //检查校验和
            if (checksum != persistent_script->dynamic_members.checksum ) {
                ...
                persistent_script = NULL;
            }
        }
        ...
        //3) 返回缓存或重新编译
        if (!persistent_script) { //无缓存可用
            ...
            //调用 ZendVM 默认的编译器进行编译
            persistent_script = opcache_compile_file(file_handle, type, key,
key ? key_length : 0, &op_array);
            if (persistent_script) {
                //将编译结果缓存
                persistent_script = cache_script_in_shared_memory(persistent_
script, key, key ? key_length : 0, &from_shared_memory);
            }
            ...
        } else { //有缓存
            ...
        }
        ...
        return zend_accel_load_script(persistent_script, from_shared_memory);
    }
```

前面曾介绍过，存储脚本缓存的结构为共享内存，进程间共享，但是从 ZCSG(hash)查找脚本缓存时却没有加锁，假如查询过程中有其他进程在改写数据，那么取到的岂不是脏数据吗？事实上，取到缓存后并不是直接就使用了，而是先校验缓存数据是否被改写了，也就是检查 checksum 的过程。checksum 通过 Adler-32 算法计算得到，它是一个比 crc 更高效的算法。如果计算得到的 checksum 与原始的值不同，则表示缓存无效。这是无锁操作比较常见的一种实现方式，360 开源的分布式配置工具 QConf 也采用了这种方式解决并发导致的数据冲突问题。

zend_accel_load_script()根据缓存的内容，把函数符号表、类符号表等复制到系统默认的位置，即 CG(function_table)、CG(class_table)，最后返回 zend_op_array。

3. 缓存的生成

在 persistent_compile_file()中我们简单介绍了缓存的查询过程，如果没有缓存或者缓存失效了，则需要重新编译并缓存结果，其中编译的过程由 opcache_compile_file()完成，编译完成后调用 cache_script_in_shared_memory()进行缓存。首先看一下 Opcache 中缓存的结构 zend_persistent_script，缓存的数据并不仅仅是 zend_op_array，还有函数、类的符号表，zend_persistent_script 具体结构如下：

```
typedef struct _zend_persistent_script {
    zend_string   *full_path;           // 完整的脚本文件路径
    zend_op_array  main_op_array;       //编译生成的 zend_op_array
    HashTable      function_table;
    HashTable      class_table;
    ...
    accel_time_t   timestamp;           //脚本的更新时间
    zend_bool      corrupted;
    zend_bool      is_phar;

    void          *mem;                 //zend_persistent_script 内存的地址
    size_t         size;                //共享内存的大小
    ...

    struct zend_persistent_script_dynamic_members {
        time_t       last_used; //上次使用时间
        zend_ulong       hits; //缓存命中次数
        unsigned int memory_consumption;
        unsigned int checksum; //缓存的校验和
        time_t       revalidate;
    } dynamic_members;
} zend_persistent_script;
```

PHP 脚本在调用 compile_file()编译完成后，将分配一个 zend_persistent_script 结构，然后将编译生成的数据转移到 zend_persistent_script 结构中。被 Opcache 代替的编译过程：

```
    static zend_persistent_script *opcache_compile_file(zend_file_handle
*file_handle, int type, char *key, unsigned int key_length, zend_op_array
```

```
**op_array_p)
    {
        ...
        //分配一个新的 zend_persistent_script 结构
        new_persistent_script = create_persistent_script();
        ...
        //编译前替换 CG(function_table) 、EG(class_table)
        CG(function_table) = &ZCG(function_table);
        EG(class_table) = CG(class_table) = &new_persistent_script->class_
table;
        ZVAL_UNDEF(&EG(user_error_handler));
        ...
        zend_try {
            ...
            //调用 compile_file()编译
            op_array = *op_array_p = accelerator_orig_compile_file(file_handle,
type);
        } zend_catch {
        }
        ...
        //将函数符号表转移到 new_persistent_script->function_table
        zend_accel_move_user_functions(&ZCG(function_table),
&new_persistent_script->function_table);
        new_persistent_script->main_op_array = *op_array;

        efree(op_array);
        ...
        return new_persistent_script;
    }
```

此时生成的 zend_persistent_script 并不在共享内存上，调用 cache_script_in_shared_memory() 进行缓存时会重新复制到共享内存上，以便供其他进程使用。最终被保存到共享内存上的数据有：zend_persistent_script 结构、脚本路径名称、脚本中定义的类、脚本中定义的函数、脚本的 zend_op_array，内存结构如图 5-27 所示。

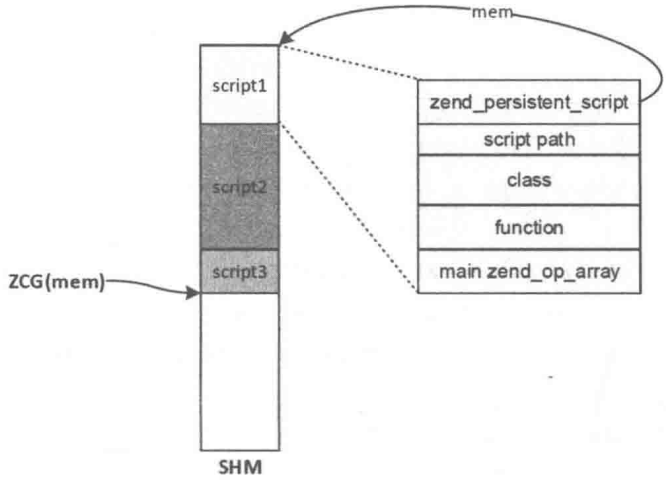


图 5-27 Opcache 脚本缓存的内存分布

```
    //将 PHP 代码的编译结果缓存到共享内存
    static zend_persistent_script *cache_script_in_shared_memory(zend_
persistent_script *new_persistent_script, char *key, unsigned int key_length,
int *from_shared_memory)
    {
        ...
        //加锁
        zend_shared_alloc_lock();
        //检查缓存空间是否够用
        if (zend_accel_hash_is_full(&ZCSG(hash))) {
            ...
        }
        //检查缓存是否存在，因为在其他进程中可能在这之前提前生成缓存了，之前的处理过程并没
有加锁
        bucket = zend_accel_hash_find_entry(&ZCSG(hash), new_persistent_
script->full_path);
        if (bucket) { //存在则不需要缓存了
            ...
            if (!existing_persistent_script->corrupted) {
                ...
                return new_persistent_script;
            }
        }
        //计算所需内存
```

```
        memory_used = zend_accel_script_persist_calc(new_persistent_script,
key, key_length);
        //分配共享内存
        ZCG(mem) = zend_shared_alloc(memory_used);
        ...
        //将 new_persistent_script 拷贝到共享内存中
        new_persistent_script = zend_accel_script_persist(new_persistent_
script, &key, key_length);
        ...
        //计算校验和
        new_persistent_script->dynamic_members.checksum = zend_accel_script_
checksum(new_persistent_script);
        //插入缓存索引表，即 zend_accel_shared_globals->hash
        bucket = zend_accel_hash_update(&ZCSG(hash), ZSTR_VAL(new_persistent_
script->full_path), ZSTR_LEN(new_persistent_script->full_path), 0, new_
persistent_script);
        if (bucket) {
            //这里还会以那个特殊的 key 插入 hash
            if (zend_accel_hash_update(&ZCSG(hash), key, key_length, 1, bucket)) {
                ...
            }
        }
        ...
        return new_persistent_script;
    }
```

具体的实现过程比较复杂，还有非常多的处理，上面介绍的只是关键的几步操作，如果想更深入地掌握 Opcache，还需要仔细研究它的实现。

5.6.1　opcode 优化

Opcache 除了缓存 PHP 代码编译结果的功能，还有一个重要的作用：opcodes 优化。在 Opcache 缓存编译结果之前，首先会对编译生成的指令进行优化，然后再进行缓存，所以说缓存的结果实际是优化后的。PHP5.x 中很多在 Opcache 中的优化已经被直接嵌入到了 PHP7 的编译器中，但是 Opcache 可优化的空间更大，因为 Zend 编译器的优化是在编译过程中，它基于的信息是有限的，而 Opcache 是在编译完成后进行的优化，这个时候取的信息更多，从而能够进行更大的优化。

```
    //将 PHP 代码的编译结果缓存到共享内存
    static zend_persistent_script *cache_script_in_shared_memory(zend_
persistent_script *new_persistent_script, char *key, unsigned int key_length,
int *from_shared_memory)
    {
        ...
        //1) 优化编译指令
        if (!zend_accel_script_optimize(new_persistent_script)) {
            return new_persistent_script;
        }
        //2) 将缓存写入共享内存
        ...
    }
```

那么 Opcache 会对编译指令进行哪些优化呢？具体可分为 11 级优化，通过 opcache.optimization_level 配置指定优化级别，按不同级别设置对应的 bit 位即可，默认配置为 0xffffffff，表示最高级别的优化：

```
#define ZEND_OPTIMIZER_PASS_1      (1<<0)   /* CSE, STRING construction    */
#define ZEND_OPTIMIZER_PASS_2      (1<<1)   /* Constant conversion and jumps */
#define ZEND_OPTIMIZER_PASS_3      (1<<2)   /* ++, +=, series of jumps     */
#define ZEND_OPTIMIZER_PASS_4      (1<<3)   /* INIT_FCALL_BY_NAME -> DO_FCALL */
#define ZEND_OPTIMIZER_PASS_5      (1<<4)   /* CFG based optimization      */
#define ZEND_OPTIMIZER_PASS_6      (1<<5)
#define ZEND_OPTIMIZER_PASS_7      (1<<6)
#define ZEND_OPTIMIZER_PASS_8      (1<<7)
#define ZEND_OPTIMIZER_PASS_9      (1<<8)   /* TMP VAR usage               */
#define ZEND_OPTIMIZER_PASS_10     (1<<9)   /* NOP removal                 */
#define ZEND_OPTIMIZER_PASS_11     (1<<10)  /* Merge equal constants       */
#define ZEND_OPTIMIZER_PASS_12     (1<<11)  /* Adjust used stack           */
#define ZEND_OPTIMIZER_PASS_13     (1<<12)
#define ZEND_OPTIMIZER_PASS_14     (1<<13)
#define ZEND_OPTIMIZER_PASS_15     (1<<14)  /* Collect constants */
```

具体的优化操作为 zend_accel_script_optimize()，这个函数里面将对 main zend_op_array 及函数、类的成员方法的 zend_op_array 一一进行优化：

```
int zend_accel_script_optimize(zend_persistent_script *script)
{
    ...
    //1) main zend_op_array
    zend_accel_optimize(&script->main_op_array, &ctx);
    ...
    //2) 优化各函数的指令
    for (idx = 0; idx < script->function_table.nNumUsed; idx++) {
        p = script->function_table.arData + idx;
        if (Z_TYPE(p->val) == IS_UNDEF) continue;
        op_array = (zend_op_array*)Z_PTR(p->val);
        zend_accel_optimize(op_array, &ctx);
    }
    //3 优化各成员方法的指令
    ...
}
```

zend_op_array 最终由 zend_optimize()完成优化：

```
static void zend_optimize(zend_op_array     *op_array,
                          zend_optimizer_ctx *ctx)
{
    if (op_array->type == ZEND_EVAL_CODE) {
        return;
    }
    //进行 11 级优化
    // pass 1:
    if (ZEND_OPTIMIZER_PASS_1 & OPTIMIZATION_LEVEL) {
        zend_optimizer_pass1(op_array, ctx);
    }
    // pass 2:
    if (ZEND_OPTIMIZER_PASS_2 & OPTIMIZATION_LEVEL) {
        zend_optimizer_pass2(op_array);
    }
    ...
}
```

Pass 1：

（1）将使用持久化常量的操作数替换为实际的常量值，例如：$a = TEST_CONST，其中TEST_CONST 是某个扩展注册的一个常量，正常操作需要首先 fetch 常量，然后再赋值给$a，优化后会直接将 TEST_CONST 替换为实际的 value，这条优化实际在 Zend 中也会进行优化。

（2）在编译时执行一些 CONST 运算，例如$a = 1 + 3，优化为$a = 4。

（3）优化 ADD_STRING、ADD_CHAR 指令，例如$a = "a" . "b"，优化为$a = "ab"，具体优化过程在 zend_optimizer_pass1()中。

Pass 2：

（1）在需要数值类型的操作中，将非数值型的 CONST 转为数值，例如$a = $b + "100"，优化为$a = $b + 100。

（2）针对组合跳转指令的优化，跳转指令是 Zend 中应用非常频繁的一个指令，在条件分支、循环结构、中断结构等语法中均使用了跳转指令。首先看一个简单的例子：

```
if ($a) {
    goto a;
} else {
    echo "no";
}

a:
echo "a";
```

if 分支最后编译一条 ZEND_JMPNZ 指令，这条指令根据条件表达式的结果决定是否需要跳转，如果跳转成立则进入分支执行。也就是说，当$a 为 true 时，进入分支，执行 goto a，然后再跳转到 echo "a"。这种情况将被优化为：如果$a 为 true，则直接跳转到 echo "a"。这个例子的优化实际上就是把 JMPZ(X, L1), JMP(L2)组合的跳转指令优化为 JMPZNZ(X, L1, L2)。除了这个例子，还有很多的组合情况会进行优化，比如 JMPNZ(X, L1), JMP(L2) => JMPZNZ(X, L2, L1)、JMPZ(X, L1), JMP(L1) => NOP, JMP(L1)，等等。

具体的优化过程由 zend_optimizer_pass2()处理。

Pass 3：

（1）将$i = $i+expr 优化为 $i+=expr，$i = $i+expr 会先将$i + expr 的计算结果保存于 TMPVAR，然后再把这个 TMPVAR 赋值给$i，分为两步，而$i += expr 则是直接在$i 上加 expr。

（2）JMP 指令的优化，将 L: JMP L+1 优化为 NOP，也就是跳到下一行的情况。其次还会优化连续跳转的情况，比如：

```
goto A; //优化为 goto B
A:
    goto B;
B:
    //...
```

这种情况实际是将 JMP L1 ... L1: JMP L2，优化为 JMP L2 .. L1: JMP L2，即：

```
JMP L1;
L1:
    JMP L2;
L2:
    ...
```

优化为：

```
JMP L2;
L1:
    JMP L2;
L2:
    ...
```

（3）在可能的地方将$i++优化为++$i，$i++是先把$i 复制一份赋值给该操作的接收变量，再将$i 加 1，如果没有接收变量，则将$i 的副本释放掉，而++$i 则是直接将$i 加 1，不会发生复制。

具体处理过程为 zend_optimizer_pass3()。

剩下还有很多级别的优化，限于篇幅这里不再一一列举，完整的优化处理可以详细看一下 zend_optimize()的处理。

5.6.2　JIT

PHP 是解释执行的，它的编译过程属于动态编译，也就是在运行的时候进行编译，与之相对的是运行前编译的静态编译。动态编译与静态编译除了编译时机的不同，还有一个不同的地

方，那就是静态编译是将代码编译为了机器指令，而动态编译并没有编译为机器指令，而是编译成了解释器可识别的指令。明确了这一点我们再来介绍即时编译，即 just-in-time compilation，简称 JIT。JIT 是动态编译中的一种技术，简单地讲，就是在某段代码第一次执行前进行编译，所以称为即时编译。

与解释执行不同的是，JIT 是将源代码编译为机器指令执行，但 JIT 又与静态编译不同，它是在运行时实时进行的编译，而且 JIT 并不会把所有代码全部编译为机器码，它只会编译频繁执行的代码。说 JIT 比解释快，其实说的是“执行编译后的代码”比“解释器解释执行”要快，并不是说编译比解释的过程快，而且 JIT 编译通常要比解释执行慢一些，如果对只执行一次的代码进行即时编译，其效率反而要比解释执行慢，JIT 之所以快是因为多次执行抵消了编译所占的时间，显得平均效率高而已。

目前，PHP 的 JIT 版本还没有正式发布，还是一个测试版本，JIT 将是下一代 PHP 最大的一个亮点。测试版本代码：https://github.com/zendtech/php-src。

5.7 小结

本章我们介绍了 PHP 内核中最重要的两部分内容：编译、执行，这是 PHP 语言实现的核心，也是不容易学习的两部分。Zend 引擎是一个非常复杂的系统，本章介绍的内容只是很少的一部分，如果想要全面地掌握还需要自己多去研究 Zend 引擎相关的实现。

后面两章将继续介绍 Zend 引擎中关于函数、面向对象的相关实现。

第 6 章 函数

通俗地讲，函数就是一组固定指令的集合，可以被重复执行，它是高级语言抽象出来的一个事物，对于计算机而言，指令集是没有函数概念的。简单地讲，函数将特定的指令集封装在一起，可以多次执行，同时它具有参数、返回值，可以根据不同的值进行处理，然后将处理结果返回。可重用性是函数非常重要的一个特性，我们可以通过函数实现对不同功能操作的封装，在需要这个功能的地方调用对应的函数来完成，避免重复编写相同的逻辑。

函数几乎是所有语言的标配，可见函数的意义所在。本章我们就来介绍 PHP 中函数的相关实现，从函数的编译到执行，完整地讲解函数的实现机制。

6.1 用户自定义函数

根据第 5 章的内容我们知道，PHP 脚本被编译为一条条的 opline 指令，这些指令组成了 zend_op_array，而函数是某些特定指令的集合。因此，很容易想到 PHP 中函数的实现：函数被编译为独立的 opline 指令集合。也就是说，PHP 脚本中定义的函数被编译为一个个独立的 zend_op_array，调用函数时像主代码（main code）那样执行函数自己的指令集。这种实现思路与机器语言是一致的，在 ELF 可执行文件中，指令是按照函数的维度组织在一起的，在代码段中，各函数的指令依次排列，函数的调用就是跳到对应函数的代码段进行执行。

PHP 脚本中通过下面的语法定义一个函数：

```
function my_function($params)
{
    //statement
}
```

PHP 内核中，函数通过 zend_function 结构表示，而不是直接使用 zend_op_array，这是因为 PHP 中还有一种特殊的函数：内部函数。zend_function 实际是一个 union，用来适配两种不同的函数，对于 PHP 脚本中定义的函数（以下称：用户自定义函数）而言，它就是 zend_op_array。

```
typedef union  _zend_function       zend_function;

//zend_compile.h
union _zend_function {
    zend_uchar type;

    struct {
        zend_uchar type;  /* never used */
        zend_uchar arg_flags[3];
        uint32_t fn_flags;
        zend_string *function_name;
        //成员方法所属类，面向对象实现中用到
        zend_class_entry *scope;
        union _zend_function *prototype;
         //参数数量
        uint32_t num_args;
        //必传参数数量
        uint32_t required_num_args;
        //参数信息
        zend_arg_info *arg_info;
    } common;
    //用户自定义函数
    zend_op_array op_array;
    //内部函数
    zend_internal_function internal_function;
};
```

zend_function 结构中还有两个特殊的成员：type、common，它们都是 zend_op_array、zend_internal_function 结构中共有的成员，用于快速获取函数的基本信息。type 用于区分函数类

型，common 则是函数的名称、参数等信息，对于用户自定义函数而言，zend_function.type 实际上取的是 zend_function.op_array.type，zend_function.common.xxx 等价于 zend_function.op_array.xxx。

如果你对 C 语言比较熟悉的话，对于这种用法一定不会陌生，之所以可以这样使用，是因为 zend_function.type、zend_function.common 结构的成员与 zend_op_array 结构的成员的内存分布是一致的。C 语言中的指针只是一块内存的起始位置，并没有特殊的含义，类型只是告诉计算机按照什么样的规则读取内存，将同一内存按照不同类型转换并不改变内存中的数据，只是展示形式不同而已，这种特性使得 C 语言中的类型转换极为灵活，这也是 C 语言中指针的精髓所在。

zend_op_array 在内存中的分布如图 6-1 所示，假设 ptr 指向这块内存，那么就可以将 ptr 转为 zend_function 类型，转完后 ptr->op_array 与转换前的 ptr 是等价的，其中 ptr->type、ptr->common.type、ptr->op_array.type、ptr->internal_function.type 这几种形式读取的是同一块内存。

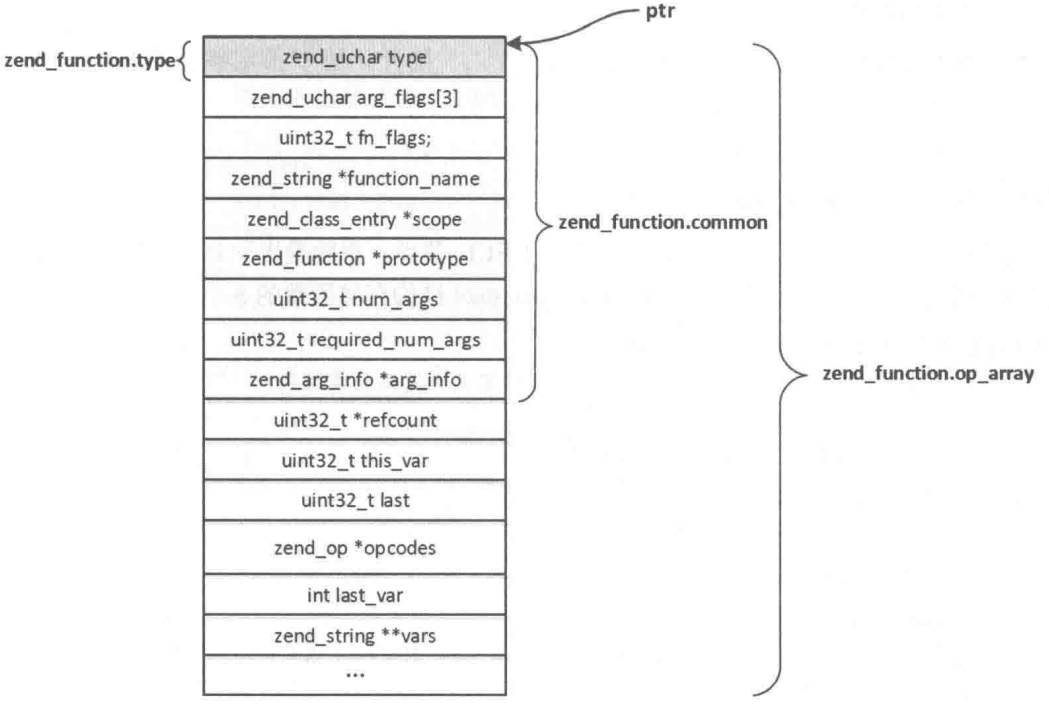


图 6-1　zend_function 结构

函数编译完成后，会插入 EG(function_table)符号表中，使用函数时根据函数名称在该符号表中索引 zend_function，从而获得编译后的函数指令，这也意味着函数的作用域是全局的，编译后可以在任意位置使用函数。接下来的两部分将介绍函数的编译过程。

6.1.1 语法解析

普通函数的语法规则如下：

```
function_declaration_statement:
    function returns_ref T_STRING backup_doc_comment '(' parameter_list ')'
return_type
    '{' inner_statement_list '}'
        { $$ = zend_ast_create_decl(ZEND_AST_FUNC_DECL, $2, $1, $4,
            zend_ast_get_str($3), $6, NULL, $10, $8); }
;
```

从函数的匹配规则可以看出，普通函数由 5 部分组成。

- **returns_ref**：是否返回引用，在函数名前加&，比如 `function &test() { ... }`。
- **T_STRING**：函数名。
- **parameter_list**：参数列表，参数之间以“,”隔开，可以指定参数的类型 `function test(int $id, string $name){ ... }`。
- **return_type**：返回值类型。
- **inner_statement_list**：函数体。

函数的定义被解析为 ZEND_AST_FUNC_DECL 节点，也就是上一章介绍抽象语法树的节点类型时提到的声明节点，该节点通过 zend_ast_decl 结构存储函数的 5 个组成部分：

```
typedef struct _zend_ast_decl {
    //函数就是 ZEND_AST_FUNC_DECL，类的定义也是用的该结构
    zend_ast_kind kind;
    zend_ast_attr attr;
    //函数定义的起始行
    uint32_t start_lineno;
    //函数定义的结束行
    uint32_t end_lineno;
    //标识位，其中一个用来标识返回值是否为引用，如果是则为 ZEND_ACC_RETURN_REFERENCE
    uint32_t flags;
    //词法分析器读取该函数的位置
    unsigned char *lex_pos;
    zend_string *doc_comment;
    //函数名
    zend_string *name;
```

```
    //child 有 4 个子节点，分别是参数列表节点、use 列表节点、函数内部表达式节点、返回值类型节点
    zend_ast *child[4];
} zend_ast_decl;
```

ZEND_AST_FUNC_DECL 节点包含以下 4 个节点。

- **参数列表节点：**该节点为 ZEND_AST_PARAM_LIST，这是一个 list 节点，它包含多个子节点，每一个子节点表示一个参数，参数的节点为 ZEND_AST_PARAM，这种节点又包含 3 个子节点，分别用于参数类型、参数名称、参数默认值。
- **use 导入的变量节点：**匿名函数会用到，这里先忽略。
- **函数体节点：**该节点为 ZEND_AST_STMT_LIST 节点，函数体中的语句均被编译到该节点下。
- **返回值类型节点：**如果在函数参数列表后通过":type"指定了返回值类型，则该类型将保存于这个节点，这个节点会有两种类型，一种为 ZEND_AST_TYPE，这种节点只有返回值为 array、callable 两种会用，另一种为 ZEND_AST_ZVAL，保存的是类型字符串。

比如下面的例子，生成的抽象语法树如图 6-2 所示。

```
//示例 6.1.1 test.php
function my_function(string $a, $b = "php~") :string{
    $ret = $a . $b;
    return $ret;
}
```

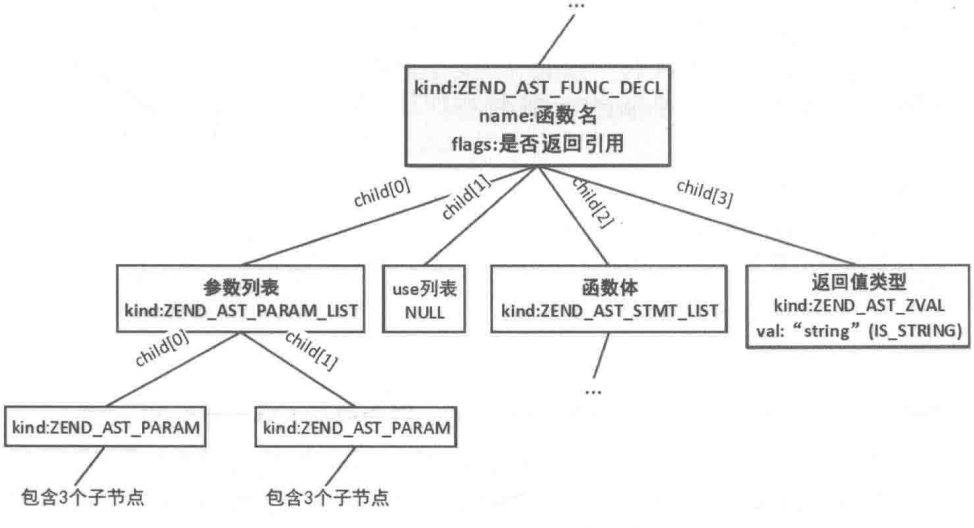


图 6-2　ZEND_AST_FUNC_DECL 节点的抽象语法树

6.1.2 抽象语法树的编译

函数在语法解析阶段被编译为 ZEND_AST_FUNC_DECL 节点，这一节我们再来详细看一下该节点编译为 opline 指令的过程。从抽象语法树的编译入口 zend_compile_top_stmt()开始，依次根据不同的语法树节点进行编译，ZEND_AST_FUNC_DECL 节点最终由 zend_compile_func_decl()方法负责编译。

```
//file: zend_compile.c
void zend_compile_func_decl(znode *result, zend_ast *ast)
{
    zend_ast_decl *decl = (zend_ast_decl *) ast;
    zend_ast *params_ast = decl->child[0]; //参数列表
    zend_ast *uses_ast = decl->child[1]; //use 列表
    zend_ast *stmt_ast = decl->child[2]; //函数内部
    zend_ast *return_type_ast = decl->child[3]; //返回值类型
    zend_bool is_method = decl->kind == ZEND_AST_METHOD; //是否为成员函数

    //这里保存当前正在编译的 zend_op_array: CG(active_op_array)，然后重新为函数生成一个新的 zend_op_array，函数编译完再将旧的还原
    zend_op_array *orig_op_array = CG(active_op_array);
    //新分配 zend_op_array
    zend_op_array *op_array = zend_arena_alloc(&CG(arena), sizeof(zend_op_array));
    //初始化 op_array
    init_op_array(op_array, ZEND_USER_FUNCTION, INITIAL_OP_ARRAY_SIZE);
    //编译 ZEND_DECLARE_FUNCTION 指令
    ...
    //覆盖 CG(active_op_array)
    CG(active_op_array) = op_array;
    //编译各个子节点
    ...
}
```

函数在编译时，会重新分配一个 zend_op_array，然后覆盖 CG(active_op_array)，这样编译函数语句生成的指令就归属与函数自己的 zend_op_arrray 了，zend_op_array 的分配过程与主代码编译前的操作基本相同。函数编译完成后再将原 CG(active_op_array)还原，通过这种方式将函数的 zend_op_array 与其他的隔离开。分配完函数的 zend_op_array 结构后开始依次编译

ZEND_AST_FUNC_DECL 节点的几个子节点，接下来按先后顺序详细看一下每个步骤的处理。

6.1.2.1 编译生成函数声明指令

函数的定义实际上也是一条语句，因此需要生成一条函数声明的指令：ZEND_DECLARE_FUNCTION，需要注意的是，该指令并不是生成在函数的 zend_op_array 中，而是在声明函数的空间内，示例中该指令就是属于主代码(main code)的，编译完该指令后才将 CG(active_op_array)覆盖为函数的。该指令的作用就是将编译后的函数注册到 EG(function_table)。

```
//zend_compile_func_decl:
if (is_method) {
    //成员方法的情况
    zend_bool has_body = stmt_ast != NULL;
    zend_begin_method_decl(op_array, decl->name, has_body);
} else {
    //编译函数声明指令：ZEND_DECLARE_FUNCTION
    zend_begin_func_decl(result, op_array, decl);
}
//覆盖 CG(active_op_array)
CG(active_op_array) = op_array;
```

普通函数由 zend_begin_func_decl()生成该指令：第一步是获取函数的名称，这个地方涉及命名空间，会进行函数名组装，关于命名空间的实现后面会讲到，这里直接认为是脚本中的原始函数名称即可；第二步就是生成 ZEND_DECLARE_FUNCTION，该指令用到两个操作数，类型均为 CONST，其中操作数 1 记录的是转为小写的函数名称，操作数 2 记录的也是一个名称，这个比较特殊，它是一个“\0 + 函数名称 + 文件路径 + lex_pos”组成的字符串；第三步是以第二步生成的那个特殊名称为 key，将函数的 zend_op_array 注册到函数符号表 CG(function_table)中。具体过程如下：

```
//file: zend_compile.c
static void zend_begin_func_decl(znode *result, zend_op_array *op_array,
zend_ast_decl *decl)
{
    zend_ast *params_ast = decl->child[0];
    zend_string *unqualified_name, *name, *lcname;
    zend_op *opline;

    unqualified_name = decl->name;
```

```
        //获取函数名称
        op_array->function_name = name = zend_prefix_with_ns(unqualified_name);
        //函数名称转为小写
        lcname = zend_string_tolower(name);
        ...
        //生成 opline、设置操作数 2
        if (op_array->fn_flags & ZEND_ACC_CLOSURE) {
            //匿名函数
            ...
        } else {
            //生成一条指令，该指令并属于函数，而是函数所在脚本的
            opline = get_next_op(CG(active_op_array));
            opline->opcode = ZEND_DECLARE_FUNCTION;
            //设置操作数 2
            opline->op2_type = IS_CONST;
            //将 CONST 变量注册到 literals
            LITERAL_STR(opline->op2, zend_string_copy(lcname));
        }
        {
            zend_string *key = zend_build_runtime_definition_key(lcname, decl->
lex_pos);
            //设置操作数 1
            opline->op1_type = IS_CONST;
            LITERAL_STR(opline->op1, key);
            //将函数注册到 CG(function_table)
            zend_hash_update_ptr(CG(function_table), key, op_array);
        }

        zend_string_release(lcname);
    }
```

示例中的函数在这里得到的函数名称就是“my_function”，接着将函数名注册到字面量数组 CG(active_op_array)->literals，通过操作数 2 记录存储位置。我们可以使用 gdb 在 zend_begin_func_decl 处设置断点，一步步看一下各语句执行后的结果：

```
$ gdb php
(gdb) break zend_begin_func_decl
(gdb) r test.php
```

```
Breakpoint 1, zend_begin_func_decl (result=0x0, op_array=0x7ffff7c03018,
decl=0x7ffff7c77258) at /home/qinpeng/php-7.0.12/Zend/zend_compile.c:4859
4859        zend_ast *params_ast = decl->child[0];
...
(gdb) p opline->op2
$1 = {constant = 0, var = 0, num = 0, opline_num = 0, jmp_offset = 0}
```

接着就是通过 zend_build_runtime_definition_key()生成那个特殊的名称：

```
//file: zend_compile.c
static zend_string *zend_build_runtime_definition_key(zend_string *name,
unsigned char *lex_pos)
{
    zend_string *result;
    char char_pos_buf[32];
    size_t char_pos_len = zend_sprintf(char_pos_buf, "%p", lex_pos);
    zend_string *filename = CG(active_op_array)->filename;

    result = zend_string_alloc(1 + ZSTR_LEN(name) + ZSTR_LEN(filename) +
char_pos_len, 0);
    sprintf(ZSTR_VAL(result), "%c%s%s%s", '\0', ZSTR_VAL(name), ZSTR_VAL
(filename), char_pos_buf);
    return zend_new_interned_string(result);
}
```

生成这个名称后也把该名称注册到 CG(active_op_array)->literals，通过操作数 1 记录存储位置，然后以这个名称为 key 注册到 CG(function_table)。打印这个 key：

```
(gdb) p key.val@key.len
$8 = {"", "m", "y", "_", "f", "u", "n", "c", "t", "i", "o", "n", "/", "t",
"m", "p", "/", "t", "e", "s", "t", "/", "t", "e", "s", "t", ".", "p", "h", "p",
"0", "x", "7", "f", "f", "f", "f", "7", "f", "5", "8", "0", "9", "2"}
(gdb) p opline->op1
$5 = {constant = 1, var = 1, num = 1, opline_num = 1, jmp_offset = 1}
```

生成的这个 key 为“ my_function/tmp/test/test.php0x7ffff7f58092 ”，最终生成的指令与 CG(function_table)中插入的 Bucket，如图 6-3 所示。至于为什么要在这里把函数注册到 CG(function_table)中将在 6.1.2.4 节得到答案。

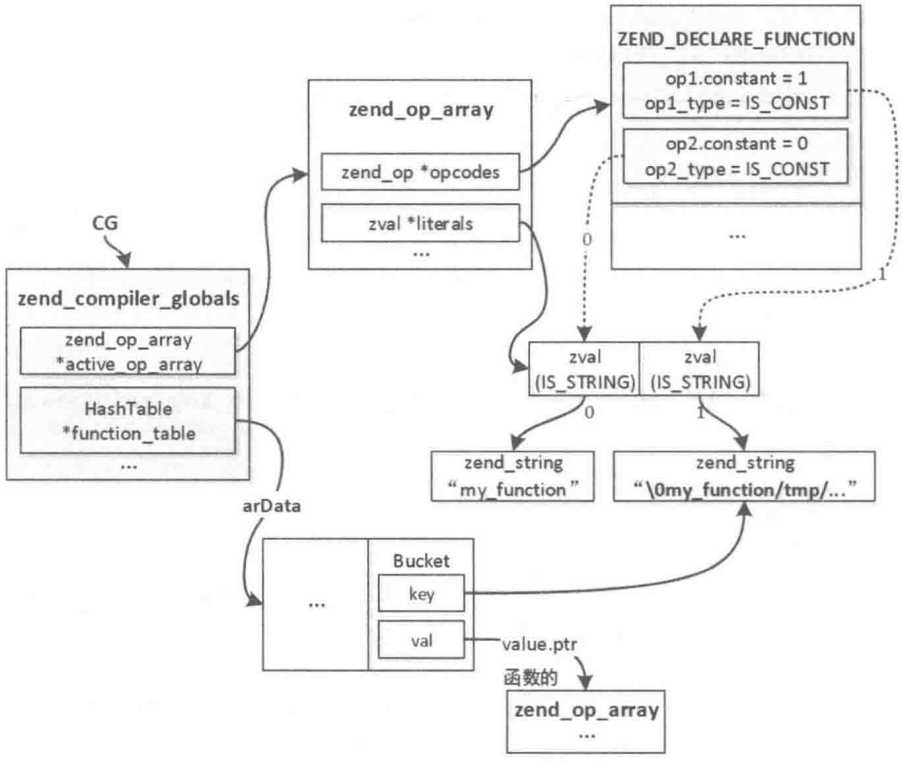


图 6-3 ZEND_AST_FUNC_DECL 指令及 CG(function_table)

6.1.2.2 参数及返回值类型的编译

ZEND_AST_FUNC_DECL 指令编译完成后，将把 CG(active_op_array)指向当前编译函数的 zend_op_array，接着开始编译参数与返回值类型，这两个是在一起编译的。

```
//zend_begin_func_decl:
CG(active_op_array) = op_array;
...
zend_compile_params(params_ast, return_type_ast);
```

在分析参数及返回值的编译前，首先介绍 PHP 中函数的参数、返回值的实现机制。

1）参数

PHP 中，函数的参数被当作普通的局部变量，其内存的分配、访问方式与局部变量没有任何区别，在编译时，参数像普通的局部变量一样分配得到一个内存编号，然后 ZendVM 根据这个值，在函数执行前为参数分配内存，对参数的所有操作均是通过这个编号完成的。在调用函数时，调用方首先会把参数值按顺序复制给被调函数 zend_execute_data 上对应的参数，即传值

过程，然后才开始执行函数指令。示例 6.1.1 中，等价于：

```
function my_function() :string{
    $a = "";
    $b = "php~";
    $ret = $a . $b;
    return $ret;
}
```

由于函数参数的编译在函数体之前，所以参数在 zend_execute_data 上是最先分配的，其次才是普通局部变量，也就是 CV 变量，最后是 VAR、TMPVAR，后面介绍函数的执行时我们再详细介绍其内存间的处理。

参数除了传递数据的基本功能，它还有几个其他属性，比如参数是否为引用、参数类型，这些属性在执行时会被用来校验传参。编译时会为每个参数创建一个 zend_arg_info 结构，这个结构用来保存参数的这些属性以及基本信息，所有的参数的 zend_arg_info 结构按照顺序保存于 zend_op_array->arg_info 数组中。

```
typedef struct _zend_arg_info {
    //参数名
    zend_string *name;
    zend_string *class_name;
    //指定的参数类型，比如 array $param_1
    zend_uchar type_hint;
    //是否引用传参，参数前加&的这个值就是 1
    zend_uchar pass_by_reference;
    //是否允许为 NULL
    zend_bool allow_null;
     //是否为可变参数
    zend_bool is_variadic
} zend_arg_info;
```

is_variadic 表示参数是否为可变参数，这是 PHP5.6 新增的一个特性，与 Golang 的用法相同，这里不作说明。

```
function my_func($a, ...$b) {
    //...
}
```

2）返回值

函数在调用前，调用方会告知被调函数返回值地址，函数只需要将值复制到该地址即可，这样调用方在函数执行结束后就可以得到返回值了。PHP 中可以通过":type"指定函数返回值的类型，如果指定了则在返回前会进行校验，这个属性也是通过 zend_arg_info 结构保存的。需要注意的是，并不是所有的 PHP 类型都可以指定为返回值，可支持以下几种基本类型：array、callable、int、float、string、bool，除此之外还可以指定返回值为类，这种情况返回值需要是该类实例化的对象，例如：

```
class my_class {
}
function my_function():my_class {
    return new my_class;
}
```

编译时，如果函数指定了返回值类型，则也为返回值创建一个 zend_arg_info 结构，该结构与参数的 zend_arg_info 保存在一起，即 zend_op_array->arg_info 数组中，特别的是，返回值的这个结构的保存位置为 zend_op_array->arg_info[-1]，也就是说，zend_op_array->arg_info 指向的是参数开始的位置。返回值指定的类型保存于 zend_arg_info->type_hint，如果返回值指定的是类，则除了通过将 type_hint 设置为 IS_OBJECT，还会把 class_name 设置为指定的类。

了解了函数的参数、返回值的相关实现后，接下来我们具体看下编译的过程。示例 6.1.1 中生成的抽象语法树完整的参数节点如图 6-4 所示。

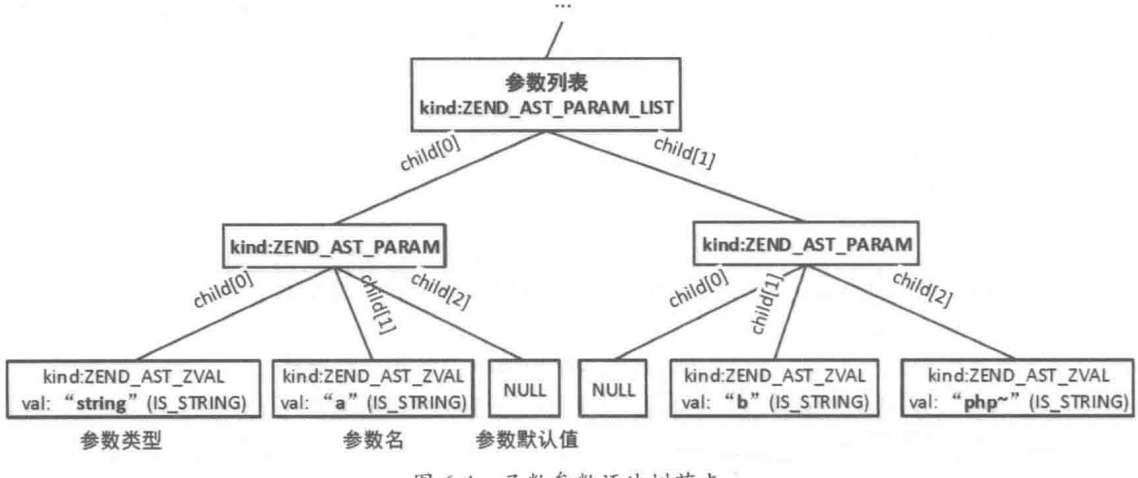


图 6-4 函数参数语法树节点

1. 返回值类型的编译

如果指定了返回值类型或返回引用，则首先编译返回值类型节点，也就是分配 zend_arg_info 结构：

```
//zend_compile_params:
if (return_type_ast) {
    //分配 zend_arg_info 数组，其中 op_array->arg_info[-1] 为返回值类型
    arg_infos = safe_emalloc(sizeof(zend_arg_info), list->children + 1, 0);
    arg_infos->name = NULL;
    //是否返回引用
    arg_infos->pass_by_reference = (op_array->fn_flags & ZEND_ACC_RETURN_
REFERENCE) != 0;
    ...
    //解析返回值类型并设置到 arg_infos->type_hint
    zend_compile_typename(return_type_ast, arg_infos);

    arg_infos++;
    op_array->fn_flags |= ZEND_ACC_HAS_RETURN_TYPE;
} else {
    //如果没有指定返回值类型，则只分配参数的 zend_arg_info 数组即可
    if (list->children == 0) {
        return;
    }
    arg_infos = safe_emalloc(sizeof(zend_arg_info), list->children, 0);
}
```

zend_compile_typename()用于解析类型，前面已经说过了，array、callable 两种类型在语法解析时类型就解析出来了，这两种生成的节点为 ZEND_AST_TYPE，类型直接保存在 attr 中，直接取出即可：

```
//获取返回值类型
static void zend_compile_typename(zend_ast *ast, zend_arg_info *arg_info)
{
    if (ast->kind == ZEND_AST_TYPE) {
        arg_info->type_hint = ast->attr;
    } else {
        ....
```

```
    }
}
```

但是，array、callable 之外的类型则在语法解析时作为字符串保存于节点中，因此这里需要根据字符串解析得到类型，其中 array、callable、int、float、string、bool 这些类型比较简单，直接根据字符串匹配即可：

```
//zend_compile_typename:
zend_string *class_name = zend_ast_get_str(ast);
zend_uchar type = zend_lookup_builtin_type_by_name(class_name);
//检查类型是否为 array、callable、int、float、string、bool，如果是 type>0
if (type != 0) {
    ...
    arg_info->type_hint = type;
}
```

如果指定的类，则会把类名设置到 class_name：

```
//zend_compile_typename:
uint32_t fetch_type = zend_get_class_fetch_type_ast(ast);
if (fetch_type == ZEND_FETCH_CLASS_DEFAULT) {
    //获取类名
    class_name = zend_resolve_class_name_ast(ast);
    zend_assert_valid_class_name(class_name);
} else {
    zend_ensure_valid_class_fetch_type(fetch_type);
    zend_string_addref(class_name);
}

arg_info->type_hint = IS_OBJECT;
arg_info->class_name = class_name;
```

2. 参数编译

编译完返回值类型以后，zend_op_array->arg_info 数组已经分配好了，接下来就是依次编译每一个参数。编译各个参数时，除了设置各参数的 zend_arg_info 信息，还有一个至关重要的操作：生成参数接收指令。根据参数是否有默认值可生成两种不同的指令：ZEND_RECV、ZEND_RECV_INIT，该指令的 result 操作数记录的就是用于参数读取的内存编号，操作数 2 记

录的则是参数的默认值，这条指令的作用主要有三个。

- 检查必传参数：如果参数没有设置默认值，则该参数必传。
- 检查参数类型：如果指定了参数的类型，则该指令将校验实际参数类型。
- 设置参数默认值：如果参数有默认值，且调用函数时未传该参数，则该指令将参数默认值复制给参数。

每个参数节点有三个子节点，分别用于参数类型、参数名、参数默认值，接下来具体看一下参数的编译过程。

```
//zend_compile_params:
//依次编译各个参数
for (i = 0; i < list->children; ++i) {
    zend_ast *param_ast = list->child[i];
    //参数类型节点
    zend_ast *type_ast = param_ast->child[0];
    //参数名节点
    zend_ast *var_ast = param_ast->child[1];
    //参数默认值节点
    zend_ast *default_ast = param_ast->child[2];
    //参数名
    zend_string *name = zend_ast_get_str(var_ast);
    ...
}
```

首先是为参数分配编号，也就是前面提到的用于参数访问的内存编码，从它的处理过程可以看到，与普通局部变量的处理完全一致，也是通过 lookup_cv()申请的 CV 变量编号：

```
//zend_compile_params:
var_node.op_type = IS_CV;
var_node.u.op.var = lookup_cv(CG(active_op_array), zend_string_copy(name));
```

比如示例 6.1.1，参数$a、$b 分配到的编码分别为 0、1，而函数体中的第一个局部变量$ret 之后将被分配为 2。接着，生成接收参数的指令，如果没有定义默认值，则生成 ZEND_RECV 指令，如果定义了默认值，则生成 ZEND_RECV_INIT 指令，参数的默认值为 CONST 类型，保存于 zend_op_array->literals 中。

```
//zend_compile_params:
```

```
if (is_variadic) {
    ...
} else if (default_ast) {
    //参数有默认值：可选参数
    ...
    opcode = ZEND_RECV_INIT;
    default_node.op_type = IS_CONST;
    zend_const_expr_to_zval(&default_node.u.constant, default_ast);
    ...
} else {
    //参数无默认值：必传参数
    opcode = ZEND_RECV;
    default_node.op_type = IS_UNUSED;
    //更新必传参数数
    op_array->required_num_args = i + 1;
}

opline = zend_emit_op(NULL, opcode, NULL, &default_node);
//将参数的 CV 变量编号通过 result 操作数记录
SET_NODE(opline->result, &var_node);
//操作数 1 记录当前参数是第几个参数
opline->op1.num = i + 1;
```

该指令编译完成后，接下来的处理就是设置参数的 zend_arg_info 信息，此过程没什么可讲的。最后还有一个操作是对参数默认值进行校验，如果参数指定了类型且有默认值，则需要检查该默认值是否符合指定的类型，比如指定为 array 类型的参数，其默认值只能是数组或者 null，具体的过程这里不再展开。

另外有一个细节值得注意，在编译各参数过程中，判断参数是否为必传参数依据的是参数默认值，如果没有默认值则为必传参数，同时更新必传参数的数量：op_array->required_num_args = i + 1。这里为什么不是 op_array->required_num_args++ 而是以参数当前是第几个参数来统计的呢？看一个例子就明白了：

```
//将报错，必传参数为 3 个，而不是 1 个
my_test(1);
function my_test($a, $b = 123, $c) {
}
```

参数及返回值类型编译完成后，生成的 opline 指令及参数结构如图 6-5 所示。

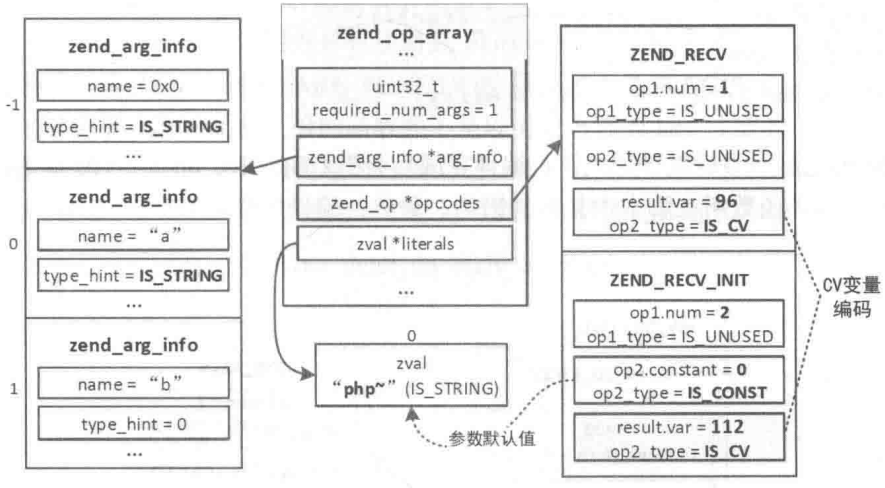


图 6-5　ZEND_RECV、ZEND_RECV_INIT 指令及 arg_info

6.1.2.3　函数体的编译

函数体的编译过程与普通脚本的编译过程无异：

```
//zend_compile_func_decl:
zend_compile_params(params_ast, return_type_ast);
...
//编译函数体
zend_compile_stmt(stmt_ast);
```

示例 6.1.1 中，最后函数生成的指令如图 6-6 所示。

6.1.2.4　pass_two()

第 5 章介绍 PHP 编译时曾介绍过该过程的作用，函数在编译后同样需要该过程的处理，pass_two()中一个重要的操作就是将 literals 中的字面量、VAR、TMPVAR 变量的编号计算转化为内存偏移值，关于该过程的其他处理这里不再赘述。

```
//zend_compile_func_decl:
zend_do_extended_info();
zend_emit_final_return(NULL);

pass_two(CG(active_op_array));
```

```
zend_oparray_context_end(&orig_oparray_context);

zend_stack_del_top(&CG(loop_var_stack));
//还原 CG(active_op_array)
CG(active_op_array) = orig_op_array;
```

pass_two()之后函数就编译完成了，编译完成后会把 CG(active_op_array)还原为函数编译前的内容，继续编译函数所在脚本中其他的语句。最终，编译生成的函数的 zend_op_array 如图 6-6 所示。

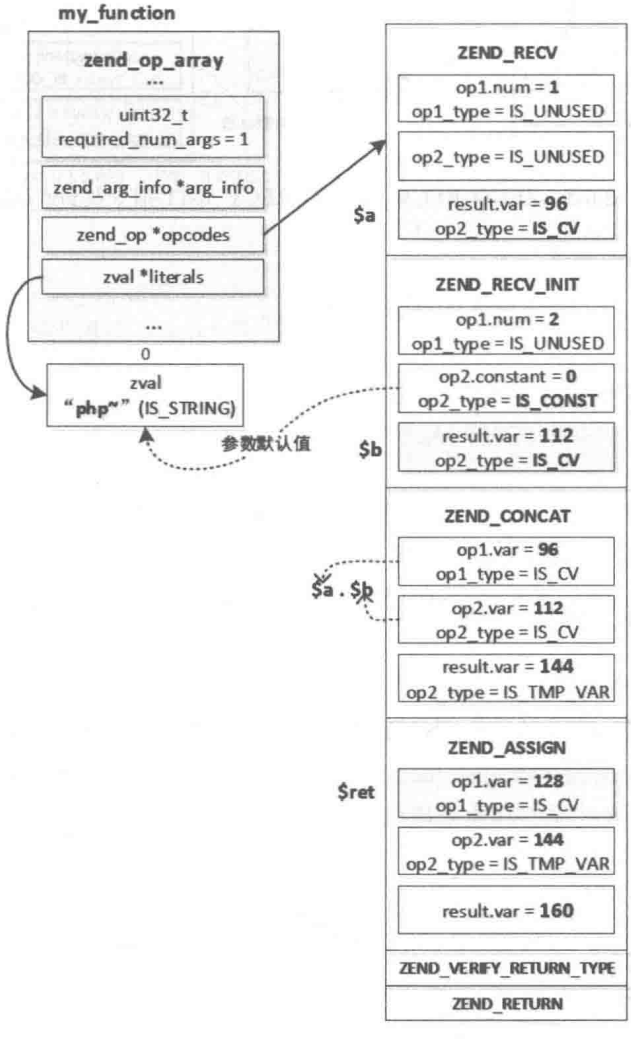


图 6-6 my_function 的 zend_op_array

6.1.2.5 注册函数

事实上，zend_compile_func_decl()完成后函数的编译过程还没有完全结束，函数虽然已经编译完成了，但是还无法使用，因为只有注册到 EG(function_table)中才可以被调用。你可能会问，在函数编译生成 ZEND_DECLARE_FUNCTION 指令时不是向 CG(function_table)中注册过函数了吗？这里需要注意，此前注册的函数并不是以函数名称为 key 注册的，也就是说无法通过函数名称检索到。实际上，那只是临时注册而已，真正注册的操作正是 ZEND_DECLARE_FUNCTION 这条指令来完成，但是该指令不总是等到运行时才会执行。

在函数编译完成后还有一步操作：zend_do_early_binding()。

```
void zend_compile_top_stmt(zend_ast *ast)
{
    ...
    if (ast->kind == ZEND_AST_STMT_LIST) {
        for (i = 0; i < list->children; ++i) {
            zend_compile_top_stmt(list->child[i]);
        }
    }
    //编译各条语法，函数也是在这里编译完成的
    zend_compile_stmt(ast);
    //脚本编译完成后
    if (ast->kind == ZEND_AST_FUNC_DECL || ast->kind == ZEND_AST_CLASS) {
        CG(zend_lineno) = ((zend_ast_decl *) ast)->end_lineno;
        zend_do_early_binding();
    }
}
```

zend_do_early_binding()核心工作完成函数、类的注册，此时注册用的 key 就是用的实际的函数名、类名，注册成功后就直接把 ZEND_DECLARE_FUNCTION 这条指令删除了，如果注册失败的话则保留，等到执行时再注册。在编译阶段，zend_do_early_binding()就已经将函数注册到 EG(function_table)符号表中了，这也是为什么 PHP 中函数不需要在调用前声明的原因，例如：

```
echo my_function();

function my_function() {
    ...
}
```

编译生成的主要指令顺序为 1→2→3，但是实际执行的顺序为 3→1→2，即：

```
1   ZEND_DO_UCALL   //函数调用
2   ZEND_ECHO
3   ZEND_DECLARE_FUNCTION   //函数声明
```

zend_do_early_binding()的具体操作如下：

```
void zend_do_early_binding(void)
{
    ...
    switch (opline->opcode) {
        case ZEND_DECLARE_FUNCTION:
            if (do_bind_function(CG(active_op_array), opline, CG(function_table),
1) == FAILURE) {
                return;
            }
            table = CG(function_table);
            break;
        ...
    }
    //删除临时注册的 key，同时将 opline 指令设置为空操作
    zend_hash_del(table, Z_STR_P(CT_CONSTANT(opline->op1)));
    zend_del_literal(CG(active_op_array), opline->op1.constant);
    zend_del_literal(CG(active_op_array), opline->op2.constant);
    MAKE_NOP(opline);
}
```

函数的注册就是在 do_bind_function()中完成的，ZEND_DECLARE_FUNCTION 指令的两个操作数分别为函数实际名称以及在 CG(function_table)中临时注册的 key。注册的过程就比较简单了：以临时 key 取出函数的 zend_op_array，然后以实际函数名称为 key 再将 zend_op_array 注册到符号表中。

6.2 内部函数

内部函数指的是由内核、扩展提供的 C 语言编写的 function，这类函数不需要经历 opline 的编译过程，所以效率上要高于 PHP 用户自定义的函数，调用时与普通的 C 程序没有差异。

Zend 引擎中定义了很多内部函数供用户在 PHP 中使用，比如：define、defined、strlen、method_exists、class_exists、function_exists……等等，除了 Zend 引擎中定义的内部函数，PHP 扩展中也提供了大量内部函数，我们也可以灵活地通过扩展自行定制。

内部函数的结构不是 zend_op_array，而是 zend_internal_function，它与 zend_op_array 具有相同的 common 成员：

```
//file: zend_complie.h
typedef struct _zend_internal_function {
    /* Common elements */
    zend_uchar type;
    zend_uchar arg_flags[3]; /* bitset of arg_info.pass_by_reference */
    uint32_t fn_flags;
    zend_string* function_name;
    zend_class_entry *scope;
    zend_function *prototype;
    uint32_t num_args;
    uint32_t required_num_args;
    zend_internal_arg_info *arg_info;
    /* END of common elements */
    //函数指针，展开: void (*handler)(zend_execute_data *execute_data, zval
*return_value)
    void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
    struct _zend_module_entry *module;
    void *reserved[ZEND_MAX_RESERVED_RESOURCES];
} zend_internal_function;
```

内部函数与用户自定义函数通过 type 区分，即可以根据函数 zend_function->type 判断出函数的类型：

```
//内部函数
#define ZEND_INTERNAL_FUNCTION          1
//用户自定义函数
#define ZEND_USER_FUNCTION              2
```

内部函数同样注册在 EG(function_table)中，因此，用户自定义函数与内部函数的名称不能相同，否则会有冲突。使用时，内部函数与用户自定义函数并没有区别，只不过两者的处理机

制不同：调用普通函数时，是由执行器执行 zend_op_array->opcodes，而内部函数则调用 zend_internal_function->handler 进行处理。

内部函数的定义非常简单，只需要在 EG(function_table)中注册一个 zend_internal_function 即可，PHP 也为内部函数的定义提供了很多方便的宏与接口，这里暂不说明，第 10 章介绍扩展开发相关的内容时再详细说明。

6.3 函数的调用

前面两节介绍了函数的定义，以及用户自定义函数的编译细节，这一节我们再来分析函数调用的编译过程。

PHP 脚本中，直接通过函数名称即可调用函数，仍以示例 6.1.1 定义的函数为例：

```
//示例 6.3 test.php
function my_function(string $a, $b = "php~") :string{
    $ret = $a . $b;
    return $ret;
}

$p1 = "hello,";
//调用函数
echo my_function($p1);
```

函数调用的语法规则如下：

```
function_call:
        name argument_list
            { $$ = zend_ast_create(ZEND_AST_CALL, $1, $2); }
    | ...
;
```

函数调用被编译为 ZEND_AST_CALL 节点，该节点有两个子节点，分别用于函数名称及参数列表，以上示例生成的节点如图 6-7 所示。

ZEND_AST_CALL 节点由 zend_compile_call()完成编译，这个过程主要生成 3 条指令：函数初始化指令、参数发送指令、函数执行指令，依次看一下这几条指令的编译过程。

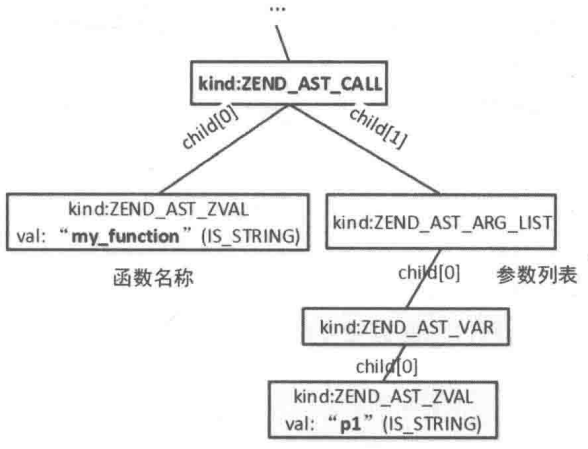


图 6-7 ZEND_AST_CALL 节点

1. 编译函数初始化指令

该指令的作用是：根据函数名称从 EG(function_table)中找到 zend_function，然后分配被调函数的 zend_execute_data 内存。根据函数的定义与调用顺序的不同，该指令有两种：如果函数在定义后调用，则指令为 ZEND_INIT_FCALL；如果函数在定义前调用，则指令为 ZEND_INIT_FCALL_BY_NAME。实际上，这两条指令并没有本质上的区别，这里以 ZEND_INIT_FCALL 为例。

```
void zend_compile_call(znode *result, zend_ast *ast, uint32_t type)
{
    zend_ast *name_ast = ast->child[0];
    zend_ast *args_ast = ast->child[1];
    ...
    //编译函数名称
    zend_bool runtime_resolution = zend_compile_function_name(&name_node,
name_ast);
    ...
    //查找函数，如果找到了则表示函数在调用前定义，否则表示函数尚未定义
    fbc = zend_hash_find_ptr(CG(function_table), lcname);
    if (!fbc || ...) {
        zend_string_release(lcname);
        zend_compile_dynamic_call(result, &name_node, args_ast);
        return;
    }
    ...
```

```
    //生成 ZEND_INIT_FCALL 指令
    opline = zend_emit_op(NULL, ZEND_INIT_FCALL, NULL, &name_node);
    zend_alloc_cache_slot(opline->op2.constant);
    //编译传参
    zend_compile_call_common(result, args_ast, fbc);
}
```

生成的 ZEND_INIT_FCALL 指令只用到了一个操作数：op2，该操作数记录的是调用函数的名称，它是 CONST 类型，通过 gdb 查看生成的指令：

```
$ gdb php
(gdb) break zend_compile_call
(gdb) r test.php
...
(gdb) p *opline
$1 = {handler = 0x0, op1 = {constant = 0, var = 0, num = 0, opline_num = 0,
jmp_offset = 0}, op2 = {constant = 1, var = 1, num = 1, opline_num = 1, jmp_offset
= 1}, result = {constant = 0, var = 0, num = 0, opline_num = 0,  jmp_offset =
0}, extended_value = 0, lineno = 11, opcode = 61 '=', op1_type = 8 '\b', op2_type
= 1 '\001',  result_type = 8 '\b'}
```

2. 编译参数发送指令

该指令的作用是：根据实际传参，将参数的 value 传递给被调函数内接受参数的变量，也就是 zend_execute_data 上的参数，每个参数都会生成这样一条指令。zend_compile_call()中生成 ZEND_INIT_FCALL 指令后，将调用 zend_compile_call_common()继续处理。

```
void zend_compile_call_common(znode *result, zend_ast *args_ast, zend_function
*fbc)
{
    zend_op *opline;
    //opnum_init 获取的就是 ZEND_INIT_FCALL 指令的位置
    uint32_t opnum_init = get_next_op_number(CG(active_op_array)) - 1;
    uint32_t arg_count;
    uint32_t call_flags;
    //忽略这个即可，主要用于 debugger/profiler，cli 模式下通过-e 参数支持
    zend_do_extended_fcall_begin();
    //编译传参
```

```
    arg_count = zend_compile_args(args_ast, fbc);
    ...
}
```

传参指令的类型比较多，根据参数类型的不同有以下几个：

```
//参数为 CONST 常量或 TMPVAR
#define ZEND_SEND_VAL                   65
#define ZEND_SEND_VAL_EX                116
//参数为 CV
#define ZEND_SEND_VAR                   117
#define ZEND_SEND_VAR_EX                 66
//引用传参
#define ZEND_SEND_REF                    67
//参数为 VAR，如：my_function(time());
#define ZEND_SEND_VAR_NO_REF             106
```

_EX 结尾的 opcode 是函数在定义前调用的情况下使用的，zend_compile_args()的具体处理过程这里不再展开，不同的处理情况比较多，但是并不复杂。示例中的参数$p1 生成的指令就是 ZEND_SEND_VAR，其中操作数 1 记录的是参数在被调空间的位置，操作数 2 记录的是参数序号，result 操作数记录的是被调函数接受参数的位置，执行时，首先根据操作数 1 拿到参数值，然后拷贝给 result 操作数指定的变量，如图 6-8 所示。

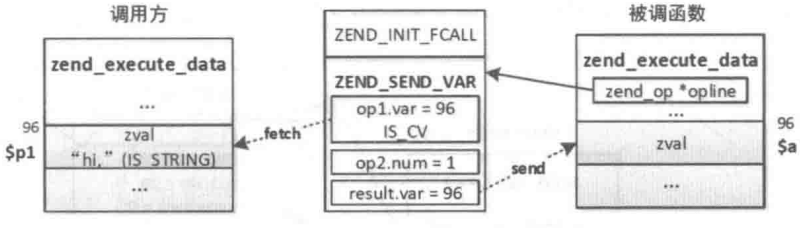


图 6-8 ZEND_SEND_VAR 指令的操作数

3. 编译函数执行指令

zend_compile_call_common()编译完传参指令后，将 ZEND_INIT_FCALL 指令的 extended_value 值更新为实际传参数量，同时计算出函数 zend_execute_data 所需内存，然后将这个值保存到操作数 1 中，关于 zend_execute_data 内存大小的计算，上一章介绍 ZendVM 执行流程时已经说过了。

```
void zend_compile_call_common(znode *result, zend_ast *args_ast, zend_
```

```
function *fbc)
    {
        ...
        opline = &CG(active_op_array)->opcodes[opnum_init];
        //将 ZEND_INIT_FCALL 指令的 extended_value 值更新为实际传参数量
        opline->extended_value = arg_count;

        if (opline->opcode == ZEND_INIT_FCALL) {
            //计算 zend_execute_data 所需内存大小
            opline->op1.num = zend_vm_calc_used_stack(arg_count, fbc);
        }
        ...
        opline = zend_emit_op(result, zend_get_call_op(opline->opcode, fbc),
NULL, NULL);
        opline->op1.num = call_flags;
    }
```

zend_compile_call_common()最后还将生成一条最关键的指令：函数执行，该指令的作用就是将执行器切换到被调函数，执行被调函数的指令。根据调用函数类型的不同，该指令分为：ZEND_DO_ICALL（内部函数）、ZEND_DO_UCALL（调用已定义了的用户自定义函数）、ZEND_DO_FCALL_BY_NAME（调用尚未定义的函数）、ZEND_DO_FCALL（自定义执行器的情况）几种。

示例 6.3 最后编译生成的函数调用的指令如图 6-9 所示。

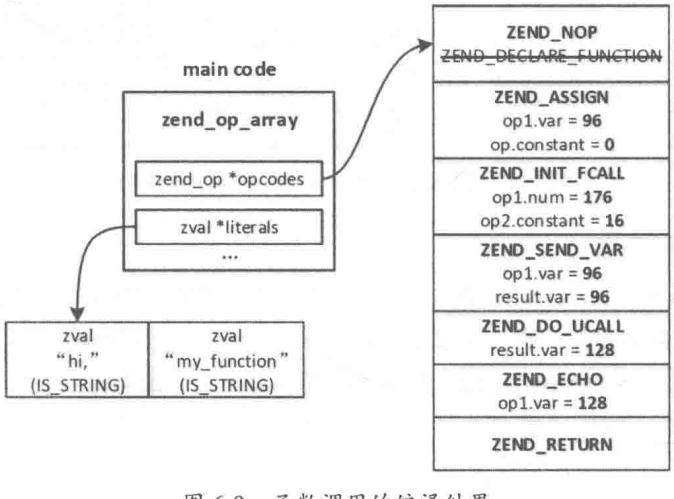


图 6-9 函数调用的编译结果

6.4 函数的执行

函数的执行与普通脚本的执行相比并没有特殊之处，都是由execute_ex()负责调度执行各条指令，不同之处在于，函数的执行过程中多了几个特殊阶段，也就是上一节介绍的函数调用编译出的几条指令的处理。这一节仍以示例6.3为例看一下具体的执行过程。

1. 初始化阶段

这个阶段首先查找到函数的zend_function，接着根据被调函数的zend_op_array新分配zend_execute_data内存，然后将被调用函数的zend_execute_data设置到调用方的zend_execute_data->call，该过程就是ZEND_INIT_FCALL指令所做的工作。

```
ZEND_VM_HANDLER(61, ZEND_INIT_FCALL, ANY, CONST)
{
    USE_OPLINE
    zend_free_op free_op2;
    zval *fname = GET_OP2_ZVAL_PTR(BP_VAR_R);
    zval *func;
    zend_function *fbc;
    zend_execute_data *call;

    fbc = CACHED_PTR(Z_CACHE_SLOT_P(fname));
    if (UNEXPECTED(fbc == NULL)) {
        //查找被调用的函数
        func = zend_hash_find(EG(function_table), Z_STR_P(fname));
        ...
        fbc = Z_FUNC_P(func);
        CACHE_PTR(Z_CACHE_SLOT_P(fname), fbc);
    }
    //分配 zend_execute_data
    //This 中的 call_info 设置为 ZEND_CALL_NESTED_FUNCTION
    call = zend_vm_stack_push_call_frame_ex(
        opline->op1.num, ZEND_CALL_NESTED_FUNCTION,
        fbc, opline->extended_value, NULL, NULL);
    //设置调用上下文
    call->prev_execute_data = EX(call);
    EX(call) = call;
```

```
    ZEND_VM_NEXT_OPCODE();
}
```

该阶段完成后，被调函数的 zend_execute_data 内存分配完毕，其中 prev_execute_data 指向调用方，它的作用就是在函数调用完成后返回调用方。分配 zend_execute_data 的过程第 5 章已经讲过，这里不再赘述，具体的内存结构如图 6-10 所示。

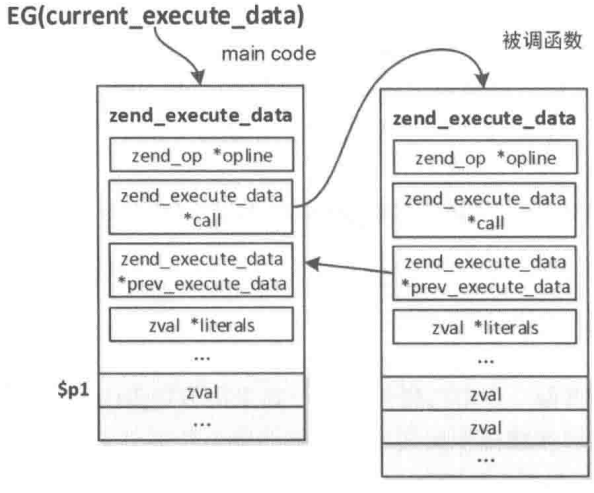


图 6-10　被调函数的初始化 zend_execute_data

2. 参数传递阶段

如果函数没有参数则跳过此步骤，如果有参数，则会将函数所需参数传递给被调函数接收参数的变量，即分配在 zend_execute_data 动态变量区的参数。示例中，传入的参数为$p1，这是一个局部变量，传参的过程就是把$p1 的值复制给 my_function 函数中的$a，如图 6-11 所示。

```
ZEND_VM_HANDLER(117, ZEND_SEND_VAR, VAR|CV, ANY)
{
    USE_OPLINE
    zval *varptr, *arg;
    zend_free_op free_op1;
    //获取传参值
    varptr = GET_OP1_ZVAL_PTR_UNDEF(BP_VAR_R);
    if (OP1_TYPE == IS_CV && UNEXPECTED(Z_TYPE_INFO_P(varptr) == IS_UNDEF)) {
        ...
    }
    //获取接收参数的地址
```

```
    arg = ZEND_CALL_VAR(EX(call), opline->result.var);
    if (OP1_TYPE == IS_CV) {
        ZVAL_OPT_DEREF(varptr);
        //将变量值复制至参数接收的变量
        ZVAL_COPY(arg, varptr);
    } else /* if (OP1_TYPE == IS_VAR) */ {
        ...
    }

    ZEND_VM_NEXT_OPCODE();
}
```

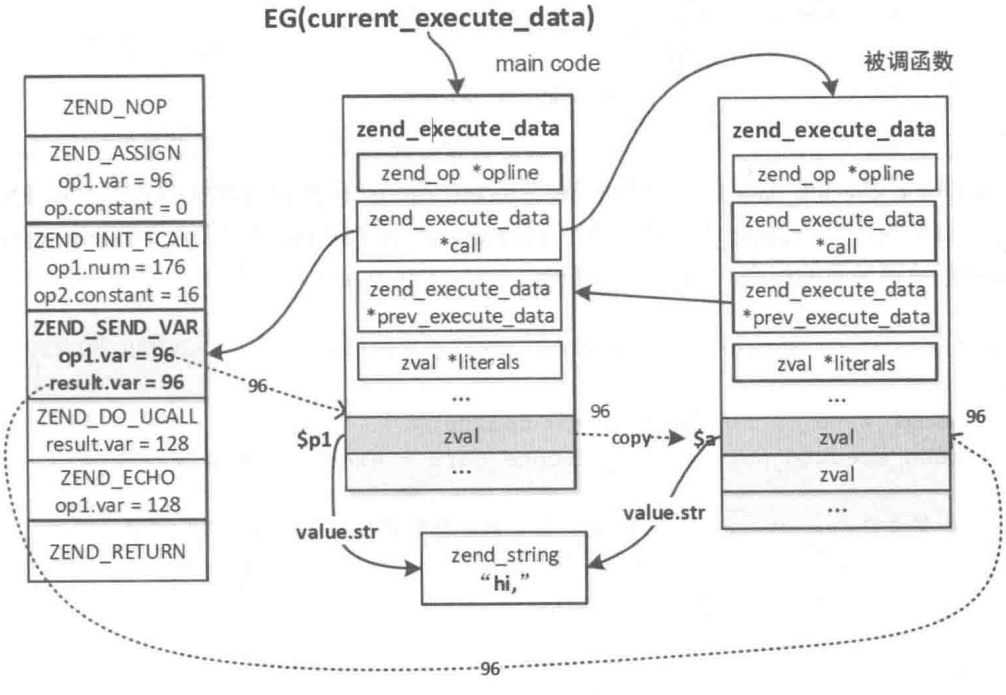


图 6-11　参数传递

3. 函数执行阶段

完成前两步的操作后，接下来就是将执行器切换到被调函数，执行函数指令了。执行器的切换由 ZEND_DO_UCALL 指令完成，切换的过程比较简单：将执行器的 execute_data 指向被调函数，即 execute_data->call，然后返回。

```
ZEND_VM_HANDLER(130, ZEND_DO_UCALL, ANY, ANY)
```

```
{
    USE_OPLINE
    zend_execute_data *call = EX(call);
    zend_function *fbc = call->func;
    zval *ret;
    ...
    //初始化 zend_execute_data，并将 EG(curret_execute_data)指向被调函数
    call->prev_execute_data = execute_data;
    i_init_func_execute_data(call, &fbc->op_array, ret, 0);

    ZEND_VM_ENTER();
    //展开后：
    //execute_data = EG(current_execute_data);
    //opline = EX(opline);
    //return;
}
```

i_init_func_execute_data()会将 EG(curret_execute_data)指向被调函数 ，ZEND_VM_ENTER()将执行器执行的全局 execute_data 更新为 EG(curret_execute_data)，此时，调度函数 execute_ex()将开始执行被调函数的指令，如图 6-12 所示。

```
ZEND_API void execute_ex(zend_execute_data *ex)
{
    const zend_op *orig_opline = opline;
    zend_execute_data *orig_execute_data = execute_data;

    //将当前 execute_data、opline 保存到全局变量
    execute_data = ex;
    opline = execute_data->opline

    while (1) {
        //ZEND_DO_UCALL 执行后返回这里开始执行函数指令
        ((opcode_handler_t)opline->handler)();
        if (UNEXPECTED(!opline)) {
            execute_data = orig_execute_data;
            opline = orig_opline;
            return;
        }
    }
}
```

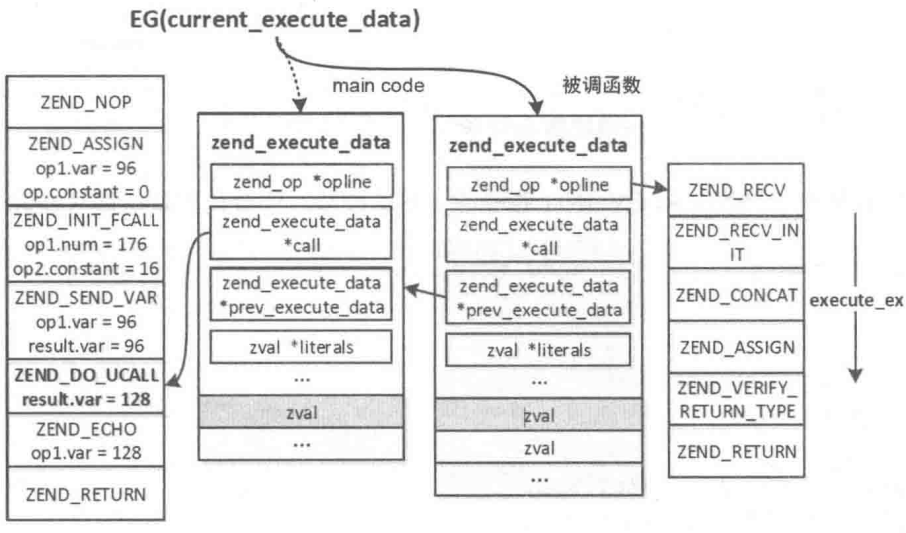


图 6-12 执行器切换

execute_ex()首先执行的就是函数的参数接收指令，该过程有两个作用：如果指定了参数类型，则在此时验证实际传参的类型；使用参数默认值初始化参数。示例中，参数$a指定了类型为 string，该参数在 ZEND_RECV 指令执行时校验；调用函数时，并没有提供参数$b，所以使用该参数的默认值，由 ZEND_RECV_INIT 指令处理。

```
ZEND_VM_HANDLER(63, ZEND_RECV, ANY, ANY)
{
    USE_OPLINE
    //arg_num 为参数序号
    uint32_t arg_num = opline->op1.num;
    //EX_NUM_ARGS 为实际传参数
    if (UNEXPECTED(arg_num > EX_NUM_ARGS())) {
        //必传参数不足
        ...
    } else if (UNEXPECTED((EX(func)->op_array.fn_flags & ZEND_ACC_HAS_TYPE_
HINTS) != 0)) {
        //获取实际传参
        zval *param = _get_zval_ptr_cv_undef_BP_VAR_W(execute_data, opline->
result.var);
        //校验参数类型
        if (UNEXPECTED(!zend_verify_arg_type(EX(func), arg_num, param, NULL,
CACHE_ADDR(opline->op2.num)) || EG(exception))) {
            HANDLE_EXCEPTION();
        }
```

```
    }

    ZEND_VM_NEXT_OPCODE();
}
```

如果没有传参，ZEND_RECV_INIT 指令除了参数校验，还会使用默认值对参数进行赋值：

```
ZEND_VM_HANDLER(64, ZEND_RECV_INIT, ANY, CONST)
{
    USE_OPLINE
    uint32_t arg_num;
    zval *param;

    arg_num = opline->op1.num;
    //result 操作数为函数的参数接收地址
    param = _get_zval_ptr_cv_undef_BP_VAR_W(execute_data, opline->result.var);
    if (arg_num > EX_NUM_ARGS()) { //参数使用默认值
        //将参数默认值复制至参数
        ZVAL_COPY_VALUE(param, EX_CONSTANT(opline->op2));
        ...
    }
    //校验参数类型
    ...
}
```

参数校验、接收完成后，参数$a、$b 的内存分布如图 6-13 所示。

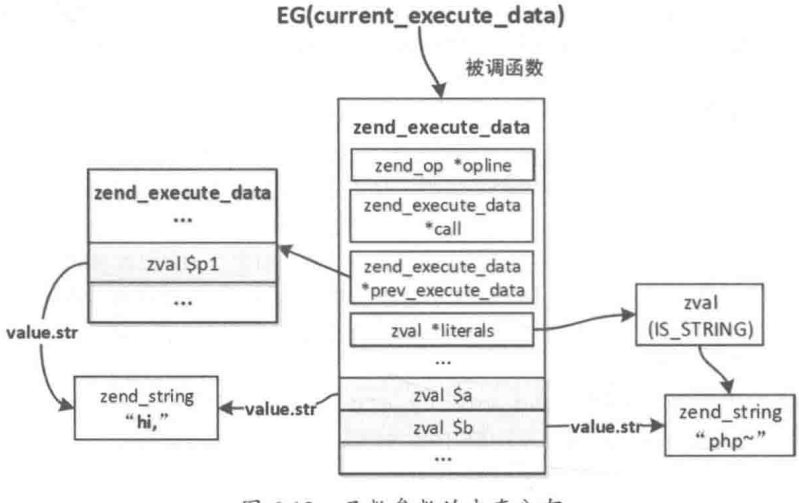


图 6-13　函数参数的内存分布

4. 函数返回阶段

被调用函数执行完毕后，接着将返回到被调用的位置继续执行，该过程主要有三个操作：设置返回值、切换执行器、清理函数局部变量。函数在调用前，将函数的返回值地址传给了被调函数，函数直接将返回值复制至该地址即可。

```
ZEND_VM_HANDLER(62, ZEND_RETURN, CONST|TMP|VAR|CV, ANY)
{
    USE_OPLINE
    zval *retval_ptr;
    zend_free_op free_op1;

    retval_ptr = GET_OP1_ZVAL_PTR_UNDEF(BP_VAR_R);
    if (OP1_TYPE == IS_CV && UNEXPECTED(Z_TYPE_INFO_P(retval_ptr) ==
IS_UNDEF)) {
        ...
    } else if (!EX(return_value)) {
        ...
    } else {
        ...
        ZVAL_DEREF(retval_ptr);
        //复制返回值
        ZVAL_COPY(EX(return_value), retval_ptr);
        ...
    }
    ZEND_VM_DISPATCH_TO_HELPER(zend_leave_helper);
}
```

设置完返回值后，将继续调用 zend_leave_helper_SPEC()返回至调用位置，并清理函数的局部变量、释放 zend_execute_data。

```
ZEND_VM_HELPER(zend_leave_helper, ANY, ANY)
{
    zend_execute_data *old_execute_data;
    uint32_t call_info = EX_CALL_INFO();
    //调用的是用户自定函数
    if (EXPECTED(ZEND_CALL_KIND_EX(call_info) == ZEND_CALL_NESTED_ FUNCTION)) {
        zend_object *object;
```

```
        //释放局部变量
        i_free_compiled_variables(execute_data);
        ...
        zend_vm_stack_free_extra_args_ex(call_info, execute_data);
        old_execute_data = execute_data;
        //将EG(current_execute_data)指向原zend_execute_data，即函数的调用方
        execute_data = EG(current_execute_data) = EX(prev_execute_data);
        ...
        //释放zend_execute_data
        zend_vm_stack_free_call_frame_ex(call_info, old_execute_data);
        ...
        //指向函数调用后的下一条指令
        LOAD_NEXT_OPLINE();
        ZEND_VM_LEAVE();
    }
    ...
}
```

局部变量由 i_free_compiled_variables()完成释放，然后将 execute_data 重新设置为调用函数的 zend_execute_data，也就是被调函数的 zend_execute_data->prev_execute_data。处理完成后，执行调度函数 execute_ex()将从函数调用的下一条指令继续执行，也就是 ZEND_ECHO，如图 6-14 所示。

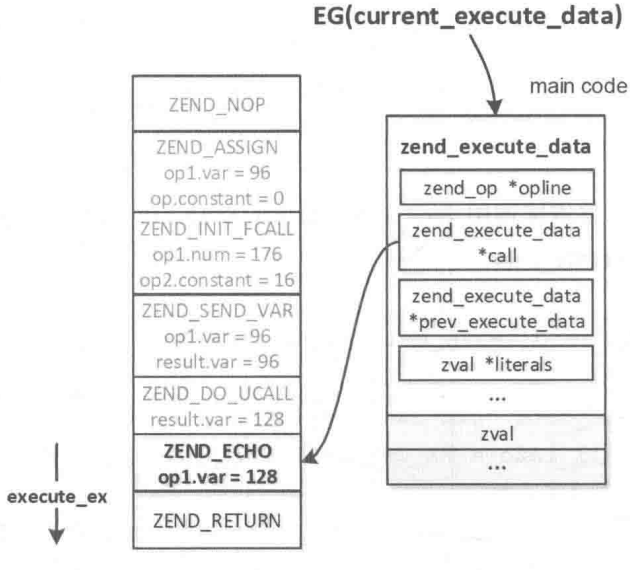


图 6-14 函数返回

6.5 小结

函数对于 PHP 而言是非常重要的一部分，通过对函数的实现机制的了解有助于我们更好地认识 ZendVM。本章介绍了 PHP 中函数的编译、执行相关的实现，实际上，函数主要就是围绕执行时几个关键阶段的处理实现的，即传参、函数调用的切换、返回等，想要深入地理解函数的实现，务必要弄清楚这些阶段所做的工作。

为了便于理解，文中介绍的内容，并没有对一些具体的处理细节进行细致的说明，主要是围绕示例展开说明的。除了介绍的这些内容，还有很多内容没有讲到，比如先调用后定义的函数、匿名函数等，有兴趣的读者可以自行研究。

第 7 章 面向对象

面向对象编程，简称 OOP，是一种程序设计思想。面向对象把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。面向对象一直是软件开发领域内比较热门的话题，它更符合人类看待事物的一般规律。与 Java 不同，PHP 并非纯面向对象的语言，它同时支持面向过程、面向对象，PHP 最初的版本是不支持面向对象的，从 PHP5 开始全面实现了面向对象。

面向对象的特点如下所述。

- 封装：将类中的成员属性和方法结合成一个独立的单位，确保类以外的部分不能随意存取类的内部数据。
- 继承：一个类可以继承并拥有另外一个类的成员属性和成员方法。
- 多态：程序能够处理多种类型对象的能力，PHP 中可以通过继承和接口两种方式实现多态。

本章将详细介绍 PHP 中面向对象的相关实现，主要包括类、对象、继承等部分。

7.1 类

类是现实世界或思维世界中的实体在计算机中的反映，它将某些具有关联关系的数据以及这些数据上的操作封装在一起，是具有相同属性和服务的一组对象的集合。在面向对象中的编程中，类是对象的抽象，对象是类的具体实例。

在 PHP 中类是编译阶段的产物，而对象是运行时产生的，它们归属于不同阶段。一个类可以包含属于自己的常量、变量（称为“属性”）与函数（称为“方法”），PHP 中我们这样定义一个类：

```
class 类名 {
    //常量;
    //成员属性;
    //成员方法;
}
```

除了在 PHP 脚本中定义的类，还有一种类是内核、扩展提供的。在内核中，不管是内部类还是用户自定义类，均通过 zend_class_entry 结构表示，类的常量、成员属性、成员方法均保存在这个结构中，这个结构包含的成员非常多，这里列几个比较重要的：

```
typedef struct _zend_class_entry    zend_class_entry;
struct _zend_class_entry {
    //类的类型：内部类 ZEND_INTERNAL_CLASS(1)、用户自定义类 ZEND_USER_CLASS(2)
    char type;
    //类名，PHP 类不区分大小写，统一为小写
    zend_string *name;
    //父类
    struct _zend_class_entry *parent;
    int refcount;
    //类掩码，如普通类、抽象类、接口，除了这还有别的含义
    uint32_t ce_flags;
    //普通属性数，包括 public、private
    int default_properties_count;
    //静态属性数
    int default_static_members_count;
    //普通属性值数组
    zval *default_properties_table;
    //静态属性值数组
    zval *default_static_members_table;
    zval *static_members_table;
    //成员方法符号表
    HashTable function_table;
    //成员属性基本信息哈希表，key 为成员名，value 为 zend_property_info
```

```
    HashTable properties_info;
    //常量符号表
    HashTable constants_table;

    //以下是构造函授、析构函数、魔术方法的指针
    union _zend_function *constructor;
    union _zend_function *destructor;
    union _zend_function *clone;
    union _zend_function *__get;
    ...
    //自定义的钩子函数，通常在定义内部类时使用，可以灵活地进行一些个性化的操作，用户自定义类不会用到，可忽略
    zend_object* (*create_object)(zend_class_entry *class_type);
    zend_object_iterator *(*get_iterator)(zend_class_entry *ce, zval *object, int by_ref);
    int (*interface_gets_implemented)(zend_class_entry *iface, zend_class_entry *class_type);
    union _zend_function *(*get_static_method)(zend_class_entry *ce, zend_string* method);

    //序列化调用的接口
    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; //实现的接口数
    uint32_t num_traits;
    zend_class_entry **interfaces; //实现的接口

    //union info: 类所在文件、起始行号、所属模块等信息
    ...
}
```

关于 zend_class_entry 中的成员，在后面用到的时候会具体介绍，实际上从 zend_class_entry 的结构就可以大体猜测出类的相关实现了。PHP 脚本中定义的类，在编译阶段将被解析到相应的成员下，例如成员方法会按照函数的编译规则编译为 zend_function，然后注册到 zend_class_entry->function_table 中。类编译完成后，将像函数那样注册到一个全局符号表中：

EG(class_table)。使用时根据类名到该符号表中索引，EG(class_table)保存着全部的类，包括用户自定义类与内部类，如图 7-1 所示。

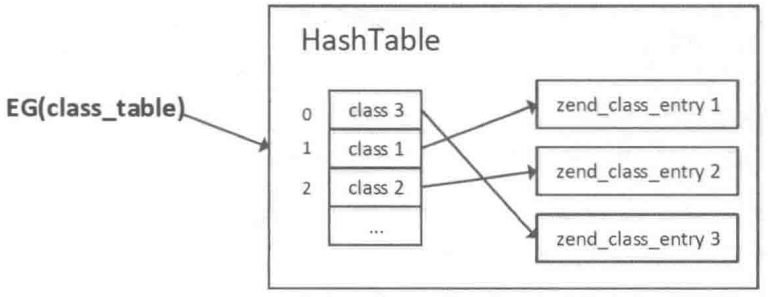


图 7-1　EG(class_table)符号表

7.1.1　常量

PHP 中可以把在类中始终保持不变的值定义为常量，在定义和使用常量的时候不需要使用 $ 符号，常量的值必须是一个定值，比如布尔型、整型、字符串、数组，不能是变量、数学运算的结果或函数调用，它是只读的，无法进行赋值。类中的常量与 PHP 中普通的常量（通过 define() 或 const 定义的）含义是一样的，只不过类中的常量只属于固定的类，而普通的常量是全局的。

常量通过 const 关键字定义，常量名通常采用大写，常量通过 class_name::CONST_NAME 访问，或在类的内部通过 self::CONST_NAME 访问，例如：

```
class my_class {
    //定义一个常量
    const CONST_NAME = const_value;

    public function __construct() {
        //在类内部访问常量
        self::CONST_NAME;
    }
}
//在类外访问类的常量
my_class::CONST_NAME;
```

常量属于类维度的数据，对于类的所有实例化对象是没有任何差异的，这一点与成员属性不同。与普通常量相同，类的常量也通过哈希表存储，类的常量保存在 zend_class_entry->constants_table 中。通常情况下，访问常量时需要根据常量名称索引，但是有些情况下会在编

译阶段将常量直接替换为常量值使用，比如：

```
//示例 1
echo my_class::A1;

class my_class {
    const A1 = "hi";
}

//示例 2
class my_class {
    const A1 = "hi";
}

echo my_class::A1;
```

这两个例子唯一的不同就是常量的使用时机：示例 1 是在定义前使用的，示例 2 是在定义后使用的。PHP 中无论是变量、常量还是函数，都不需要提前声明，那么这两个会有什么不同呢？

具体 debug 一下上面两个例子会发现：示例 2 编译生成的主要指令只有一个 ZEND_ECHO，也就是直接输出值了，并没有涉及常量的查找，进一步查看发现它的操作数为 CONST 变量，值为“hi”，也就是 my_class::A1 的值；而示例 1 首先执行的指令则是 ZEND_FETCH_CONSTANT，查找常量，接着才是 ZEND_ECHO。

事实上这两种情况内核会有两种不同的处理方式：示例 1 这种情况会在运行时根据常量名称索引 zend_class_entry_constants_table，取到常量值以后再执行 echo；而示例 2 则有另外一种处理方式，PHP 代码的编译是顺序的，示例 2 在编译到 `echo my_class::A1` 这行时，首先会尝试检索下是否已经编译了 my_class，如果能在 CG(class_table)中找到，则进一步从类的 contants_table 中查找对应的常量，找到的话则会复制其 value 替换常量，也就是将常量的检索提前到了编译时，通过这种“预处理”优化了耗时的常量检索过程，避免多次执行时的重复检索，同时也可以利用 opcache 避免放到运行时重复检索常量。

7.1.2 成员属性

类的变量成员叫属性。属性声明由关键字 public、protected 或者 private 开头，然后跟一个普通的变量声明来组成。属性中的变量可以初始化，但是初始化的值必须是固定不变的值，不

能是变量，初始化值在编译阶段时就可以得到其值，而不依赖于运行时的信息才能求值，比如 public $time = time()，这样定义一个属性就会触发语法错误。

public、protected、private 用于限制成员属性、方法的访问权限：public 为公有的，可以在任何地方被访问；protected 为受保护的，可以被自身及其子类和父类中访问，在类之外则无法访问；private 为私有的，只能被其定义所在的类访问。按权限大小排序：public > protected > private，类属性必须定义为三者之一，如果用 var 定义，则被视为公有。

成员属性分为两类：普通属性、静态属性。静态属性通过 static 声明，通过 self::$property 或类名$::property 访问；普通属性通过 $this->property 或 $object->property 访问。例如：

```
class my_class {
    //普通属性
    public $property = "normal propterty";
    //静态属性
    public static $static_property = "static property";

    public function __construct() {
        //内部访问
        $this->property;//普通属性
        self::$static_property;//静态属性
    }
}
//外部访问
$obj = new my_class;
$obj->property; //普通属性
my_class::$static_property; //静态属性
```

静态成员属性为各对象共享，与常量类似，而普通成员属性则是各对象独享，对象之间对普通成员属性的修改不会相互影响。与常量的存储方式不同，成员属性的初始化值并不是直接以属性名作为索引的哈希表存储的，而是通过数组保存的，普通属性、静态属性各有一个数组分别存储，即 default_properties_table、default_static_members_table。另外，zend_class_entry 中还有两个成员用于记录这两个数组的长度：default_properties_count、default_static_members_count。编译后类的成员属性的数组如图 7-2 所示。

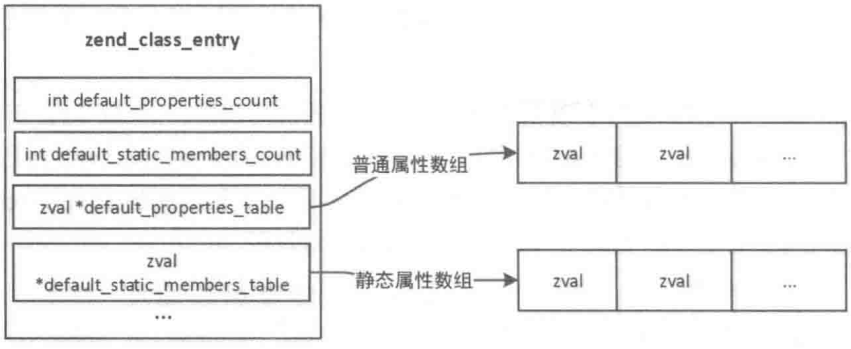


图 7-2 成员属性在类中的存储

看到这里你可能有个疑问：如果使用数组存储，那么访问成员属性时是如何找到的呢？

在解答这个问题之前首先要明确一个问题：静态属性是共享的，所以运行时各对象操作的就是 zend_class_entry->default_static_members_table 数组中的值，也就是说，静态属性值保存在类中，而不是对象中；但是普通成员属性是对象独享的，这就意味着各对象的普通成员属性的值不会保存在类中，它们实际存储在对象结构中，也就是 zend_object，下一节会详细介绍，类中的 default_properties_table 数组中的值只是用于初始化对象的，在实例化对象时会拷贝到对象中。

普通成员属性的存储与局部变量的实现相同，它们分布在 zend_object 上，通过相对 zend_object 的内存偏移进行访问，各属性的内存偏移值在编译时分配，普通成员属性的存取都是根据这个内存 offset 完成的。静态属性相对比较简单，直接根据数组下标访问。了解了普通成员属性与静态成员属性的访问机制，接下来我们再解答成员属性是如何找到的。

实际上，default_properties_table、default_static_members_table 数组只是用来存储属性值的，并不是保存属性信息的，这里说的属性信息包括属性的访问权限（public、protected、private）、属性名、静态属性值的存储下标、非静态属性的内存 offset 等，这些信息通过 zend_property_info 结构存储，该结构通过 zend_class_entry->properties_info 符号表存储，这是一个哈希表，key 就是属性名。看到这里你应该知道属性是如何访问的了吧？

```
typedef struct _zend_property_info {
    uint32_t offset; //普通成员变量的内存偏移值，静态成员变量的数组索引
    uint32_t flags;  //属性掩码，如public、private、protected及是否为静态属性
    zend_string *name; //属性名:并不是原始属性名
    zend_string *doc_comment;
    zend_class_entry *ce; //所属类
} zend_property_info;
```

关键成员的含义：

- **name**：属性名称，特别注意的是这里并不全是原始属性名，private 会在原始属性名前加上类名，protected 则会加上*作为前缀。
- **offset**：普通成员属性的内存偏移值，如 40、56、72…，与 CV 变量的操作数相同，它们分配 在 zend_object 结构上，读取时根据“zend_object 地址 ＋offset”获取；对于静态成员属性则是 default_static_members_table 数组索引，如 0、1、2…
- **flag**：属性标识，有两个含义——一是区分属性是否为静态，静态、非静态属性的结构都存储在这一个符号表中；二是属性权限，即 public、private、protected。

```
//flags 标识位
#define ZEND_ACC_PUBLIC    0x100
#define ZEND_ACC_PROTECTED 0x200
#define ZEND_ACC_PRIVATE   0x400

#define ZEND_ACC_STATIC       0x01
```

在编译时，成员属性将根据属性类型按照属性定义的先后顺序分配一个对应的 offset，用于运行时索引属性的存储位置。读取成员属性分为两步：

- 第 1 步：根据属性名从 zend_class_entry.properties_info 中索引到属性的 zend_property_info 结构。
- 第 2 步：根据 zend_property_info->offset 获取具体的属性值，其中静态成员属性通过 `zend_class_entry.default_static_members_table[offset]` 获取，普通成员属性则通过 `((char*)zend_object) + offset` 获取。

以下面的示例为例，最终生成的属性结构如图 7-3 所示。

```
class my_class {
    public $property_1 = "hi,php~";
    public $property_2 = array();

    public static $property_3 = 110;
}
```

当访问 `self::$property_3` 时，首先根据字符串“property_3”检索 properties_info，找到该属性的 zend_property_info 结构，然后以 zend_property_info->offset 为下标，在 default_static_members_table 数组中读取对应数据，即 default_static_members_table[0]。关于普

通成员属性的操作，下一节介绍对象时再详细说明。

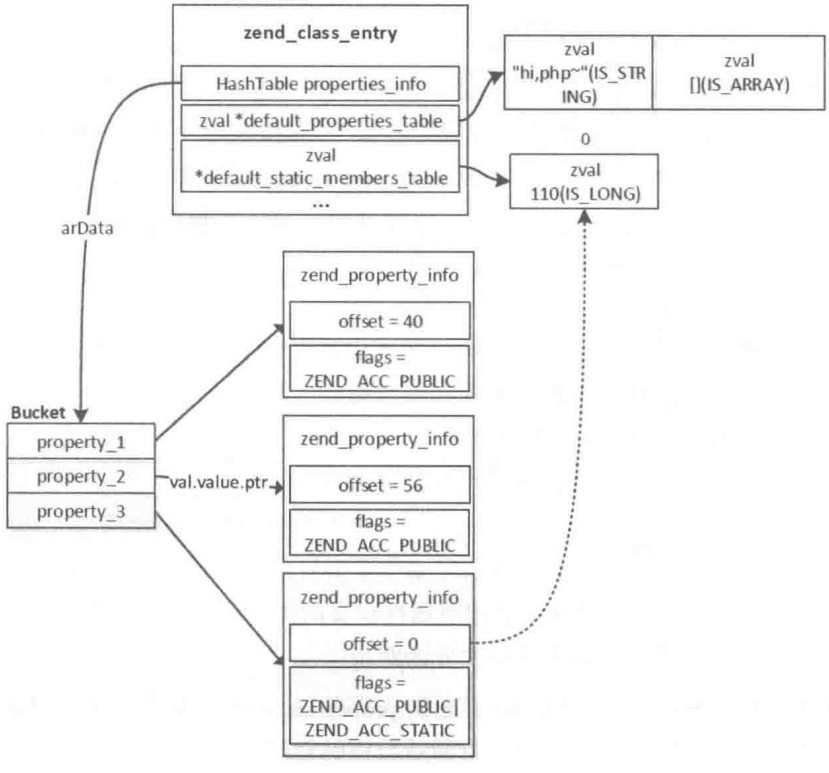


图 7-3 属性信息符号表

7.1.3 成员方法

每个类可以定义若干个属于本类的函数，称之为成员方法。成员方法与普通的函数相比并没有本质上的差别，只不过成员方法封装在类中，是类专有的函数，不能被别的类调用，所以成员方法保存在类中而不是 EG(function_table)全局函数符号表中。

与成员属性一样，成员方法也有权限控制，也有静态、非静态之分，其中静态成员方法不需要通过对象调用，可以直接根据“类名::静态成员方法”调用，或者在类内部通过“self::静态成员方法”调用，而普通非静态成员方法则只能通过对象发起调用。另外，成员方法除了 static 静态之分，还有 abstract、final 两类，分别表示抽象方法、不可被覆盖方法。

```
class my_class {
    static public function static_func() {
        //...
```

```
    }
    public function func() {
        //...
    }
}
//静态方法的调用
my_class::static_func();
//非静态方法的调用
$obj = new my_class;
$obj->func();
```

成员方法的存储结构也是 zend_function，其中 zend_function.common->scope（对用户自定义类而言就是 zend_op_array->scope）为该方法所属类，zend_function.common->fn_flags 用于标识成员方法的权限、类型（如 abstrace、final、static），除了这些，fn_flags 还有很多其他的标识，比如成员方法指定了返回值类型，则 fn_flags 将包含 ZEND_ACC_HAS_RETURN_TYPE 的标识。

```
union _zend_function {
    zend_uchar type;    /* MUST be the first element of this struct! */

    struct {
        zend_uchar type;  /* never used */
        zend_uchar arg_flags[3]; /* bitset of arg_info.pass_by_reference */
        uint32_t  fn_flags;  //方法标识：finnal/static/abstrace、private/
public/protected
        zend_string *function_name;
        zend_class_entry *scope;  //成员方法所属类
        union _zend_function *prototype;
        uint32_t num_args;
        uint32_t required_num_args;
        zend_arg_info *arg_info;
    } common;

    zend_op_array op_array;
    zend_internal_function internal_function;
};
```

编译过程与普通函数相同，不同的是，成员方法最后被注册到所属类的 function_table 符号表中，当调用一个成员方法时，将到 zend_class_entry.function_table 中进行查找。最终编译生成的成员方法符号表如图 7-4 所示。

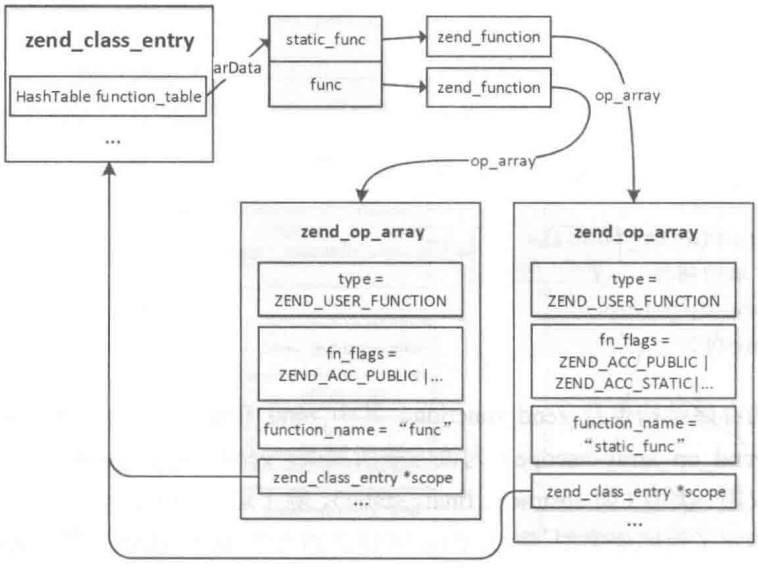


图 7-4 类的成员方法符号表

7.1.4 类的编译

前面我们先介绍了类的相关组成部分，接下来我们通过一个例子简单看一下类的编译过程：

```
//示例：/tmp/user.php
class User {
    const TYPE = 110;

    static $name = "uuu";
    public $uid = 900;

    public function getName(){
        return $this->uid;
    }
}
```

类定义的语法规则：

```
class_declaration_statement:
        class_modifiers T_CLASS { $<num>$ = CG(zend_lineno); }
        T_STRING extends_from implements_list backup_doc_comment '{'
class_statement_list '}'
```

```
            { $$ = zend_ast_create_decl(ZEND_AST_CLASS, $1, $<num>3, $7,
zend_ast_get_str($4), $5, $6, $9, NULL); }
        |   T_CLASS { $<num>$ = CG(zend_lineno); }
            T_STRING extends_from implements_list backup_doc_comment '{'
class_statement_list '}'
            { $$ = zend_ast_create_decl(ZEND_AST_CLASS, 0, $<num>2, $6,
zend_ast_get_str($3), $4, $5, $8, NULL); }
    ;

    //整个类内为 list，每个成员属性、成员方法都是一个子节点
    class_statement_list:
            class_statement_list class_statement
                { $$ = zend_ast_list_add($1, $2); }
        |   /* empty */
                { $$ = zend_ast_create_list(0, ZEND_AST_STMT_LIST); }
    ;

    //类内语法规则：成员属性、成员方法
    class_statement:
            variable_modifiers property_list ';'
                { $$ = $2; $$->attr = $1; }
        |   T_CONST class_const_list ';'
                { $$ = $2; RESET_DOC_COMMENT(); }
        ...
        |   method_modifiers function returns_ref identifier backup_doc_comment
'(' parameter_list ')'
            return_type method_body
                { $$ = zend_ast_create_decl(ZEND_AST_METHOD, $3 | $1, $2, $5,
                    zend_ast_get_str($4), $7, NULL, $10, $9); }
    ;
```

从语法规则可以看出，类被编译为 ZEND_AST_CLASS 节点，该节点有 3 个子节点，分别表示：继承类、实现的接口列表、类的表达式，其中类的表达式是一个 list 节点，每个常量、成员属性、成员方法对应一个子节点，其节点类型分别为 ZEND_AST_CLASS_CONST_DECL、ZEND_AST_PROP_DECL、ZEND_AST_METHOD。

- **ZEND_AST_CLASS_CONST_DECL**：类常量声明节点，类型为 list，因为可以同时声明多个常量，比如 const C1，C2，C3，每个子节点是一个常量，类型为 ZEND_AST_CONST_ELEM，它有两个子节点，分别用于常量名、常量值，具体规则见语法规则文件中的 class_const_list 配置。

- **ZEND_AST_PROP_DECL：** 成员属性声明节点，与常量节点相同，这也是一个 list 节点，可以同时声明多个成员属性，子节点类型为 ZEND_AST_PROP_ELEM，它有三个子节点，其中前两个节点分别表示属性名、属性初始化，第三个节点用于保存注释，忽略即可。需要注意的是，属性的访问权限、是否为静态的信息并没有保存在 ZEND_AST_PROP_ELEM 节点中，而是通过 ZEND_AST_PROP_DECL->attr 保存。具体规则见语法规则文件的 property_list。
- **ZEND_AST_METHOD：** 成员方法声明节点，与函数的声明节点类似：ZEND_AST_FUNC_DECL。每个 ZEND_AST_METHOD 节点表示一个成员方法，它有 4 个子节点，其中第一个节点为参数列表，第二个节点没有使用，第三个节点为函数体，第四个节点为返回值类型。

示例最终生成的抽象语法树如图 7-5 所示。

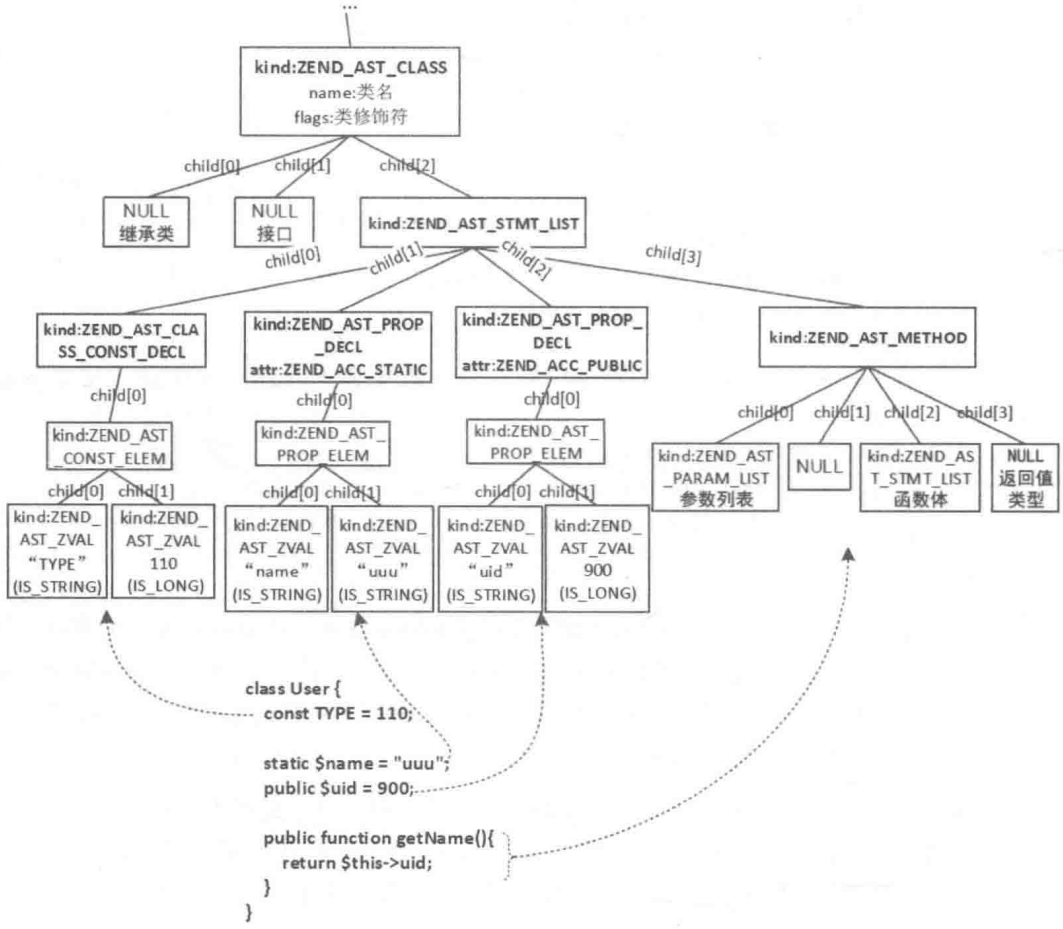


图 7-5 类定义生成的抽象语法树

ZEND_AST_CLASS 节点由 zend_compile_class_decl()函数完成编译，类的各个子节点的编译过程相对比较独立，依次进行对应的编译即可。编译步骤如下：

步骤（1）：分配 zend_class_entry 结构，如果有继承的父类，则编译一条 ZEND_FETCH_CLASS 指令，关于继承的情况，7.3 节再作说明。

步骤（2）：编译一条类声明的 ZEND_DECLARE_CLASS 指令，该指令的作用是将编译后的类注册到 EG(class_table)，与函数的 ZEND_DECLARE_FUNCTION 指令作用相同。同时将类的 zend_class_entry 注册到 CG(class_table)。注意：key 并不是类名，而是“\0 + 类名 + 文件名 + lex_pos”，这个与函数编译时的处理是一样的，此时注册后类还不能被使用。

步骤（3）：将 CG(active_class_entry)指向当前类的 zend_class_entry 结构，然后编译常量、成员属性、成员方法，并把编译结果注册到 CG(active_class_entry)对应的符号表中。

```
void zend_compile_class_decl(zend_ast *ast)
{
    zend_ast_decl *decl = (zend_ast_decl *) ast;
    zend_ast *extends_ast = decl->child[0]; //父类
    zend_ast *implements_ast = decl->child[1]; //实现的接口节点
    zend_ast *stmt_ast = decl->child[2]; //类中声明的常量、属性、方法
    zend_string *name, *lcname;
    //1) 分配 zend_class_entry
    zend_class_entry *ce = zend_arena_alloc(&CG(arena), sizeof(zend_class_
entry));
    zend_op *opline;
    ...
    lcname = zend_new_interned_string(lcname);

    ce->type = ZEND_USER_CLASS; //类型为用户自定义类
    ce->name = name; //类名
    zend_initialize_class_data(ce, 1);
    ...
    if (extends_ast) {
        ...
        //有继承的父类则首先生成一条 ZEND_FETCH_CLASS 的 opcode
        zend_compile_class_ref(&extends_node, extends_ast, 0);
    }
    //2) 生成类声明指令
    opline = get_next_op(CG(active_op_array));
```

```
    zend_make_var_result(&declare_node, opline);
    ...
    //操作数 2 为类的正式名称
    opline->op2_type = IS_CONST;
    LITERAL_STR(opline->op2, lcname);

    if (decl->flags & ZEND_ACC_ANON_CLASS) {
        ...
    }else{
        zend_string *key;

        if (extends_ast) {
            //如果类有继承的父类，则生成 ZEND_DECLARE_INHERITED_CLASS 指令
            opline->opcode = ZEND_DECLARE_INHERITED_CLASS;
            opline->extended_value = extends_node.u.op.var;
        } else {
            //无继承类则生成 ZEND_DECLARE_CLASS 指令
            opline->opcode = ZEND_DECLARE_CLASS;
        }
        //生成类的临时注册 key: \0 + 类名 + 文件名 + lex_pos
        key = zend_build_runtime_definition_key(lcname, decl->lex_pos);

        opline->op1_type = IS_CONST;
        //将这个临时 key 保存到操作数 1 中
        LITERAL_STR(opline->op1, key);
        //以临时 key 为 key，将 zend_class_entry 注册到 CG(class_table)
        zend_hash_update_ptr(CG(class_table), key, ce);
    }
    //3) 编译类的常量、成员属性、成员方法
    CG(active_class_entry) = ce;
    zend_compile_stmt(stmt_ast);
    ...
    CG(active_class_entry) = original_ce;
}
```

这里特别注意生成的这条 ZEND_DECLARE_CLASS 指令，如果类有继承的父类，则将生成另外一条指令 ZEND_DECLARE_INHERITED_CLASS。生成的 ZEND_DECLARE_CLASS 指令有两个操作数：操作数 1 记录的是将类注册到 CG(class_table)中的那个特殊 key，即按照“\0

+ 类名 + 文件名 + lex_pos”生成的 key；操作数 2 记录的则是转为小写的类的实际名称，它们的类型都是 CONST，如图 7-6 所示。

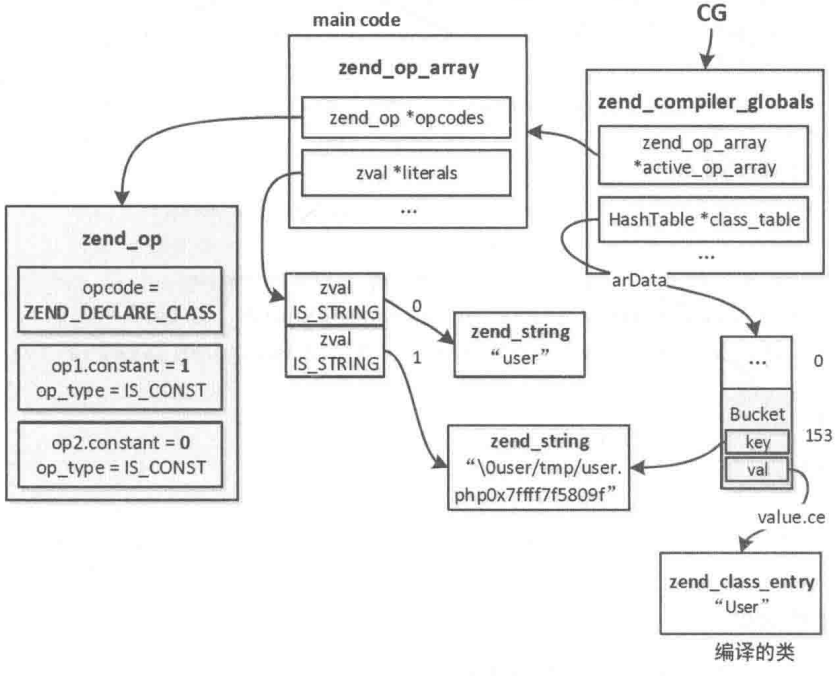


图 7-6　ZEND_DECLARE_CLASS 指令

接下来我们分别看一下常量、成员属性、成员方法的编译过程。

1. 类常量的编译

常量的节点类型为 ZEND_AST_CLASS_CONST_DECL，该节点为 list 节点，如果同时声明了多个常量，则会有多个子节点。需要注意的是，这里说的多个常量是指一个 const 声明多个情况，即通过“,”隔开的常量，并不是多个 const，如果通过多个 const 声明了常量，则将生成多个 ZEND_AST_CLASS_CONST_DECL 节点。具体的编译过程如下：

```
void zend_compile_class_const_decl(zend_ast *ast)
{
    zend_ast_list *list = zend_ast_get_list(ast);
    //获取当前编译的类
    zend_class_entry *ce = CG(active_class_entry);
    uint32_t i;
    //依次编译每个子节点：const C1, C2, C3 = 100;
    for (i = 0; i < list->children; ++i) {
        zend_ast *const_ast = list->child[i];
```

```
        //常量名节点
        zend_ast *name_ast = const_ast->child[0];
        //常量值节点
        zend_ast *value_ast = const_ast->child[1];
         //常量名
        zend_string *name = zend_ast_get_str(name_ast);
        zval value_zv;
        //取出常量值
        zend_const_expr_to_zval(&value_zv, value_ast);

        name = zend_new_interned_string_safe(name);
        //将常量注册到 zend_class_entry->constants_table 哈希表中
        if (zend_hash_add(&ce->constants_table, name, &value_zv) == NULL) {
            ...
        }
        ...
    }
}
```

2. 成员属性的编译

属性节点类型为 ZEND_AST_PROP_DECL，每个属性生成一个节点，与常量节点相同，该节点也是 ist 节点，依次表示声明多个属性的情况，子节点类型为 ZEND_AST_PROP_ELEM。另外，ZEND_AST_PROP_DECL 节点的 attr 保存着属性的访问权限，即 public、protected、private。具体编译过程如下：

```
void zend_compile_prop_decl(zend_ast *ast)
{
    zend_ast_list *list = zend_ast_get_list(ast);
    //属性修饰符: static、public、private、protected
    uint32_t flags = list->attr;
    zend_class_entry *ce = CG(active_class_entry);
    uint32_t i, children = list->children;

    for (i = 0; i < children; ++i) {
        //这个节点类型为: ZEND_AST_PROP_ELEM
        zend_ast *prop_ast = list->child[i];
        zend_ast *name_ast = prop_ast->child[0]; //属性名节点
        zend_ast *value_ast = prop_ast->child[1]; //属性值节点
        zend_ast *doc_comment_ast = prop_ast->child[2]; //备注
```

```
        zend_string *name = zend_ast_get_str(name_ast); //属性名
        zend_string *doc_comment = NULL;
        zval value_zv;
        ...
        //检查该属性是否在当前类中已经定义
        if (zend_hash_exists(&ce->properties_info, name)) {
            zend_error_noreturn(...);
        }
        if (value_ast) {
            //取出默认值
            zend_const_expr_to_zval(&value_zv, value_ast);
        } else {
            //默认值为 null
            ZVAL_NULL(&value_zv);
        }

        name = zend_new_interned_string_safe(name);
        //保存属性
        zend_declare_property_ex(ce, name, &value_zv, flags, doc_comment);
    }
}
```

编译时会检查属性是否已经定义，避免重复。得到属性名、属性值后，接下来就是非常重要的两步操作：创建属性的 zend_property_info 结构并注册到 zend_class_entry->properties_info 符号表中；将属性值复制到属性默认值数组。前面介绍的属性的 offset 就是在这一步分配的：静态属性的 offset 为数组下标——0、1、2…这个值按照顺序依次分配，通过 default_static_members_count 记录当前的静态属性的数量，每编译一个静态属性，这个值就加 1；非静态属性的 offset 为相对 zend_object 结构的内存偏移，非静态属性通过 default_properties_count 记录当前的非静态属性数量，分配时也是根据这个值进行编号的，只是将编号乘以 sizeof(zval)转为了内存偏移值。

```
//file: zend_API.c
ZEND_API int zend_declare_property_ex(zend_class_entry *ce, zend_string
*name, zval *property, int access_type,...)
{
    zend_property_info *property_info, *property_info_ptr;

    if (ce->type == ZEND_INTERNAL_CLASS) {//内部类
        ...
```

```
        }else{
            property_info = zend_arena_alloc(&CG(arena), sizeof(zend_property_
info));
        }

        if (access_type & ZEND_ACC_STATIC) {
            //静态属性
            ...
            //分配属性编号，同变量一样，静态属性的就是数组索引
            property_info->offset = ce->default_static_members_count++;
            //将静态属性值数组扩容
            ce->default_static_members_table = perealloc(ce->default_static_
members_table, sizeof(zval) * ce->default_static_members_count, ce->type ==
ZEND_INTERNAL_CLASS);
            //将新加入的静态属性值复制到 default_static_members_table 数组中

ZVAL_COPY_VALUE(&ce->default_static_members_table[property_info->offset],
property);
            if (ce->type == ZEND_USER_CLASS) {
                ce->static_members_table = ce->default_static_members_table;
            }
        }else{
            //非静态属性
            ...
            //将非静态属性按编号转为相对 zend_obejct 结构的偏移：40、56…
            property_info->offset = OBJ_PROP_TO_OFFSET(ce->default_properties_
count);
            ce->default_properties_count++;
            //扩容
            ce->default_properties_table = perealloc(ce->default_properties_
table, sizeof(zval) * ce->default_properties_count, ce->type == ZEND_INTERNAL_
CLASS);
            //将非静态属性的默认值复制到 default_properties_table 数组
            ZVAL_COPY_VALUE(&ce->default_properties_table[OBJ_PROP_TO_NUM
(property_info->offset)], property);
        }
        ...
        //设置 property_info->name：这里并不是原始的属性名称
        if (access_type & ZEND_ACC_PUBLIC) {
            property_info->name = zend_string_copy(name);
        } else if (access_type & ZEND_ACC_PRIVATE) {
```

```
            //private 属性的名称会被拼上类名
            property_info->name = zend_mangle_property_name(ZSTR_VAL(ce->name),
ZSTR_LEN(ce->name), ZSTR_VAL(name), ZSTR_LEN(name), ce->type & ZEND_INTERNAL_
CLASS);
        } else {
            //protected 属性在名称前加了“*”作为前缀
            ZEND_ASSERT(access_type & ZEND_ACC_PROTECTED);
            property_info->name = zend_mangle_property_name("*", 1, ZSTR_VAL
(name), ZSTR_LEN(name), ce->type & ZEND_INTERNAL_CLASS);
        }
        //将 zend_property_info 注册到 zend_class_entry->properties_info
        property_info->name = zend_new_interned_string(property_info->name);
        property_info->flags = access_type;
        property_info->doc_comment = doc_comment;
        property_info->ce = ce;
        zend_hash_update_ptr(&ce->properties_info, name, property_info);
    }
```

3. 成员方法的编译

成员方法的编译与函数的编译过程基本是相同的，它们都是由同一个函数完成编译的：

```
void zend_compile_stmt(zend_ast *ast)
{
    ...
    switch (ast->kind) {
        ...
        case ZEND_AST_FUNC_DECL: //函数
        case ZEND_AST_METHOD:  //成员方法
            zend_compile_func_decl(NULL, ast);
            break;
        ...
    }
}
```

函数、成员方法都是通过 zend_compile_func_decl()完成编译的，下面看一下成员方法的特殊处理之处：

```
void zend_compile_func_decl(znode *result, zend_ast *ast)
{
```

```
        //参数、函数内语法编译等不看了，与函数的语法编译相同，不清楚请看 3.2.1.3 节
        ...
        if (is_method) {
            zend_bool has_body = stmt_ast != NULL;
            zend_begin_method_decl(op_array, decl->name, has_body);
        } else {
            //函数是在当前空间生成了一条 ZEND_DECLARE_FUNCTION 的 opcode
            //然后在 zend_do_early_binding() 中"执行"了这条指令，即将函数添加到
CG(function_table)
            zend_begin_func_decl(result, op_array, decl);
        }
        ...
    }
```

这个过程之前已经说过，这里不再重复，与普通函数处理不同的地方在于 zend_begin_method_decl()，该方法主要的工作是注册成员方法，将成员方法的 zend_op_array 注册到 zend_class_entry->function_table，另外，该方法还会检查成员方法是否为魔术方法，如果是则还会将 zend_class_entry 结构中对应的魔术方法指向成员方法。

```
    void zend_begin_method_decl(zend_op_array *op_array, zend_string *name,
zend_bool has_body)
    {
        zend_class_entry *ce = CG(active_class_entry);
        ...
        op_array->scope = ce;
        op_array->function_name = zend_string_copy(name);

        lcname = zend_string_tolower(name);
        lcname = zend_new_interned_string(lcname);

        //注册成员方法
        if (zend_hash_add_ptr(&ce->function_table, lcname, op_array) == NULL) {
            zend_error_noreturn(..);
        }
        //后面主要是设置一些构造函数、析构函数、魔术方法指针，以及其他一些可见性、静态非静
态的检查
        ...
    }
```

以上就是类的常量、成员属性、成员方法的编译过程，此时类尚未注册到 EG(class_table)中，还不能被使用，因此，编译完成后还差最后一步：注册。注册的过程与函数的完全相同，也是在编译完类后进行的：

```
void zend_compile_top_stmt(zend_ast *ast)
{
    ...
    if (ast->kind == ZEND_AST_STMT_LIST) {
        for (i = 0; i < list->children; ++i) {
            zend_compile_top_stmt(list->child[i]);
        }
    }
    zend_compile_stmt(ast);
    //脚本编译完成后
    if (ast->kind == ZEND_AST_FUNC_DECL || ast->kind == ZEND_AST_CLASS) {
        CG(zend_lineno) = ((zend_ast_decl *) ast)->end_lineno;
        zend_do_early_binding();
    }
}
```

在类的编译之初曾生成一条 ZEND_DECLARE_CLASS 的指令，前面曾提过，该指令的作用就是注册类，但是这条指令却不是在运行时执行的，而是在 zend_do_early_binding()中完成的，这条指令将调用 do_bind_class()进行类的注册，注册完成后会把前面临时注册的那个 key 从 EG(class_table)中删除。

```
void zend_do_early_binding(void)
{
    ...
    switch (opline->opcode) {
        ...
        case ZEND_DECLARE_CLASS:
            if (do_bind_class(CG(active_op_array), opline, CG(class_table),
1) == NULL) {
                return;
            }
            table = CG(class_table);
            break;
```

```
        case ZEND_DECLARE_INHERITED_CLASS:
            //有继承类的情况
            break;
    }
    //将那个以（类名+file+lex_pos）为 key 的值从 CG(class_table)中删除
    //同时删除两个相关的 literals: key、类名
    zend_hash_del(table, Z_STR_P(CT_CONSTANT(opline->op1)));
    zend_del_literal(CG(active_op_array), opline->op1.constant);
    zend_del_literal(CG(active_op_array), opline->op2.constant);
     //将 ZEND_DECLARE_CLASS 或 ZEND_DECLARE_INHERITED_CLASS 的指令修改为空指令
    MAKE_NOP(opline);
}
```

do_bind_class()注册类的过程就比较简单了：首先根据 ZEND_DECLARE_CLASS 指令的操作数 1 获取那个特殊的 key，然后再根据这个 key 从 CG(class_table)中得到 zend_class_entry，最后再以操作 2 记录的实际的类名为 key，将 zend_class_entry 注册到 CG(class_table)中，相当于把 key 换成了实际的类名。注册完成后，会把 ZEND_DECLARE_CLASS 重置为空指令，同时将它的两个 CONST 操作数删除，如图 7-7 所示。

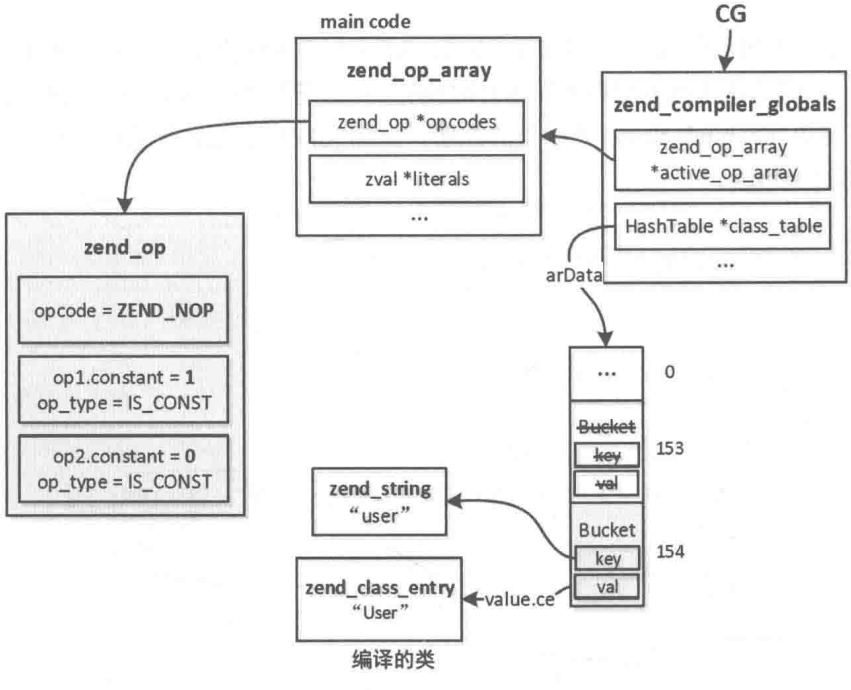


图 7-7 类的注册

7.1.5 内部类

内部类是由内核或扩展直接注册的一种类，它不需要经历编译的过程。内部类的结构也是zend_class_entry，其注册位置也与用户自定义类相同。相比于用户自定义类，内部类的定义比较简单，也更加灵活，可以进行一些个性化的处理，比如我们可以定义创建对象的钩子函数：create_object，从而在对象实例化时调用我们自己定义的函数完成，这样就可以进行很多其他的操作了。

内部类的定义简单的概括就是：创建一个 zend_class_entry 结构，然后插入到 EG(class_table)中。涉及的操作主要有：

- 注册类到符号表；
- 实现继承、接口；
- 定义常量；
- 定义成员属性；
- 定义成员方法。

实际上这些操作与用户自定义类的实现相同，只是内部类直接调用相关 API 完成这些操作，具体的 API 接口本节不再介绍，我们将在后面介绍扩展开发一章中再详细说明。

7.1.6 类的自动加载

在实际使用中，通常会把一个类定义在一个文件中，使用时通过 include 加载进来。这样就带来一个问题：在每个文件的头部都需要包含一个长长的 include 列表，而且当文件名称修改时也需要把每个引用的地方都改一遍。另外还有一个比较棘手的问题是：原则上父类需要在子类定义之前定义，当项目达到一定规模后，很难保证这个原则。为此，PHP 提供了一种类的自动加载机制，当使用未被定义的类时，将自动调用类加载器，这样就可以由类加载器将类所在的文件 include 进来，方便类的统一管理。

类的自动加载实际上就是内核提供了一个钩子函数，实例化类时如果在 EG(class_table)中没有找到对应的类，则会调用这个钩子函数，调用完以后再重新查找一次。这个钩子函数通过EG(autoload_func)指定。

PHP 中提供了两种方式实现自动加载：__autoload()、spl_autoload_register()。

1. __autoload()

这种方式比较简单，用户自定义一个__autoload()函数即可，参数是类名，当实例化一个类时如果没有找到这个类，则会查找用户是否定义了__autoload()函数，如果定义了则调用此函数，

例如：

```
//文件 1: my_class.php
<?php
class my_class {
    public $id = 123;
}

//文件 2: b.php
<?php
function __autoload($class_name){
    //do something...
    include $class_name . '.php';
}

$obj = new my_class();
var_dump($obj);
```

实际上__autoload()是内核默认的加载器，除了这个还可以通过扩展自定义类加载器，然后将 EG(autoload_func)修改为自定义的即可。如果没有自定义的加载器，则在使用时内核会检查是否定义了 __autoload() 函数，如果定义了则将 EG(autoload_func)修改为 __autoload()。类的查找通过 zend_lookup_class_ex()方法完成：

```
//file: zend_execute_API.c
ZEND_API zend_class_entry *zend_lookup_class_ex(zend_string *name, const
zval *key, int use_autoload)
{
    ...
    //首先查找 EG(class_table)，如果找到了则直接返回
    ce = zend_hash_find_ptr(EG(class_table), lc_name);
    if (ce) {
        if (!key) {
            zend_string_release(lc_name);
        }
        return ce;
    }
    //没有找到类
    ...
```

```
        //是否定义了 autoload 函数
        if (!EG(autoload_func)) {
            //检查是否定义了 __autoload 函数: #define ZEND_AUTOLOAD_FUNC_NAME
"__autoload"
            zend_function *func = zend_hash_str_find_ptr(EG(function_table),
ZEND_AUTOLOAD_FUNC_NAME, sizeof(ZEND_AUTOLOAD_FUNC_NAME) - 1);
            if (func) {
                EG(autoload_func) = func;
            } else {
                if (!key) {
                    zend_string_release(lc_name);
                }
                return NULL;
            }
        }
        ...
        //调用 EG(autoload_func)
        fcall_info.function_table = EG(function_table);
        ZVAL_STR_COPY(&fcall_info.function_name,
EG(autoload_func)->common.function_name);
        ...
        if ((zend_call_function(&fcall_info, &fcall_cache) == SUCCESS)
&& !EG(exception)) {
            //调用类加载器后再次查找类
            ce = zend_hash_find_ptr(EG(class_table), lc_name);
        }
        ...
    }
```

2. spl_autoload_register()

相比__autoload()只能定义一个加载器，spl_autoload_register()提供了更加灵活的注册方式，可以支持任意数量的加载器，比如不同 lib 库的文件名规则、保存目录都会不同，相应的，其加载规则也就不同，这种情况就可以通过此函数注册多个加载器，每个加载器按照自己的规则加载类文件。

在实现上，spl 创建了一个队列来保存用户注册的加载器，然后定义了一个 spl_autoload_call 函数到 EG(autoload_func)，当找不到类时，内核回调 spl_autoload_call，这个函数再依次调用用

户注册的加载器，每调用一个就重新检查查找的类是否在 EG(class_table)中已经注册，仍找不到的话继续调用下一个加载器，直到类成功注册为止。spl_autoload_register()函数的参数：

```
bool spl_autoload_register ([ callable $autoload_function [, bool $throw =
true [, bool $prepend = false ]]] )
```

参数$autoload_function 为 callable，可以是函数、成员方法，第 2 个参数$throw 用于设置 autoload_function 无法成功注册时，spl_autoload_register()是否抛出异常，最后一个参数如果为 true 时，spl_autoload_register() 会添加函数到队列之首，而不是队列尾部。例如：

```
function autoload_one($class_name){
    echo "autoload_one->", $class_name, "\n";
}

function autoload_two($class_name){
    echo "autoload_two->", $class_name, "\n";
}

spl_autoload_register("autoload_one");
spl_autoload_register("autoload_two");

$obj = new my_class();
var_dump($obj);
```

这个例子执行时就会将 autoload_one()、autoload_two()都调一遍，假如第一个函数成功注册了 my_class 类则不会再调后面的加载器。

SPL 的实现位于 ext/spl 目录下，具体的实现比较简单，这里不再展开。

7.2 对象

对象是系统中用来描述客观事物的一个实体，一个对象由一组属性和有权对这组属性进行操作的一组服务组成。对象是类的实例，PHP 中要创建一个类的实例，必须使用 new 关键字。在 PHP 内部，对象的结构为 zend_object：

```
typedef struct _zend_object     zend_object;

struct _zend_object {
```

```
    zend_refcounted_h gc; //引用计数
    uint32_t          handle;
    zend_class_entry *ce; //所属类
    const zend_object_handlers *handlers; //对象操作处理函数
    HashTable        *properties;
    zval              properties_table[1]; //普通属性值数组
};
```

几个主要的成员如下所述。

（1）handle。

一次 request 期间对象的编号，每个对象都有一个唯一的编号，与创建先后顺序有关，主要在垃圾回收时使用，下面会详细说明。

（2）ce。

该对象所属的类。

（3）handlers。

对象操作的处理函数：成员属性的读写、成员方法的获取、对象的销毁/克隆，等等。这些操作接口都有默认的函数，也可以通过扩展来自定义，比如自定义 write_property，这样设置一个对象的属性时将调用扩展自己定义的处理函数，让扩展拥有了更高的控制权限。

```
struct _zend_object_handlers {
    int                                offset;
    zend_object_free_obj_t                free_obj; //释放对象
    zend_object_dtor_obj_t                dtor_obj; //销毁对象
    zend_object_clone_obj_t               clone_obj;//复制对象

    zend_object_read_property_t            read_property; //读取成员属性
    zend_object_write_property_t           write_property;//修改成员属性
    ...
}

//默认的处理 handler
ZEND_API zend_object_handlers std_object_handlers = {
    0,
    zend_object_std_dtor,                  /* free_obj */
    zend_objects_destroy_object,           /* dtor_obj */
```

```
    zend_objects_clone_obj,                 /* clone_obj */
    zend_std_read_property,                 /* read_property */
    zend_std_write_property,                /* write_property */
    zend_std_read_dimension,                /* read_dimension */
    zend_std_write_dimension,               /* write_dimension */
    zend_std_get_property_ptr_ptr,          /* get_property_ptr_ptr */
    NULL,                                   /* get */
    NULL,                                   /* set */
    zend_std_has_property,                  /* has_property */
    zend_std_unset_property,                /* unset_property */
    zend_std_has_dimension,                 /* has_dimension */
    zend_std_unset_dimension,               /* unset_dimension */
    zend_std_get_properties,                /* get_properties */
    zend_std_get_method,                    /* get_method */
    NULL,                                   /* call_method */
    zend_std_get_constructor,               /* get_constructor */
    zend_std_object_get_class_name,         /* get_class_name */
    zend_std_compare_objects,               /* compare_objects */
    zend_std_cast_object_tostring,          /* cast_object */
    NULL,                                   /* count_elements */
    zend_std_get_debug_info,                /* get_debug_info */
    zend_std_get_closure,                   /* get_closure */
    zend_std_get_gc,                        /* get_gc */
    NULL,                                   /* do_operation */
    NULL,                                   /* compare */
};
```

如果没有自定义 handlers，那么将由 std_object_handlers 默认的处理函数处理。需要注意的是：const zend_object_handlers *handlers，这里的 handlers 指针加了 const 修饰符，const 修饰的是 handlers 指向的对象，而不是 handlers 指针本身，所以扩展中可以将一个对象的 handlers 修改为另一个 zend_object_handlers 指针，但无法修改 zend_object_handlers 中的值，比如 obj->handlers->write_property = xxx 将报错，而 obj->handlers = xxx 则是可以的。也就是说，如果想自定义 handlers，那么将 handlers 指针整个替换，而不能只修改默认的 handlers 中的一个。所以通常做法是：复制一份 std_object_handlers 替换 handlers，然后替换复制的这个 handlers 结构中需要自定义的 handler。例如：

```
//自定义的 handlers 结构
```

```
    static zend_object_handlers div_object_handlers_date;
    //设置 handler 指针
    memcpy(&div_object_handlers_date, zend_get_std_object_handlers(),
sizeof(zend_object_handlers));
    div_object_handlers_date.dtor_obj = xxx;
    div_object_handlers_date.get_properties = xxx;
    //替换 zend_object 的 handlers
    //zend_object.handlers = &div_object_handlers_date;
```

（4）properties。

普通成员属性哈希表，对象创建之初这个值为 NULL，主要是在动态定义属性时会用到，与 properties_table 有一定的关系，7.4 节将单独说明。

（5）properties_table。

非静态成员属性数组，这个数组就是用来存储普通成员属性值的，非静态成员属性通过 offset 分配在该数组的对应位置上，对象对非静态成员属性的读写操作的就是这个数组。该数组的长度并不是 1，这是一个变长数组，在对象实例化时根据非静态成员属性的具体数量分配相应的大小。这里你可能会有一个疑问：properties_table 数组定义时的长度为什么不是 0 而是 1 呢？这是因为在某些特殊情况下，properties_table 可能会多分配一个元素，这个特殊情况就是类中定义了__get、__set、__unset、__isset 中某一个魔术方法时，多分配的这个元素是用于防止魔术方法循环调用的，后面再详细介绍其作用。

7.2.1 对象的创建

PHP 中通过“new + 类名”创建一个类的实例，我们以下面这个例子为例，看一下一个对象创建的过程。

```
class User {
    public $arr = array(1,2,3);
    public $id = 100;

    public function getArr() {
        return $this->arr;
    }
}
//创建对象
```

```
$user = new User();
```

创建对象的语法规则：

```
new_expr:
        T_NEW class_name_reference ctor_arguments
            { $$ = zend_ast_create(ZEND_AST_NEW, $2, $3); }
    |   T_NEW anonymous_class
            { $$ = $2; }
;
```

从语法规则可以很直观地看出此语法主要有两部分：类名、参数列表。创建对象的语句在语法解析阶段被编译为 ZEND_AST_NEW 节点，该节点由 zend_compile_new()编译为 ZEND_NEW 指令，具体的编译过程这里不再细讲，比较简单。执行时该指令将创建并初始化对象，具体处理过程分为以下几步：

1. 查找类

首先根据实例化的类名从 EG(class_table)中查找这个类，即 zend_fetch_class_by_name()操作，查找时如果没有找到类则会触发上一节介绍的类的自动加载机制，如果找到了则直接返回类的 zend_class_entry 结构。另外，如果类名为 CONST 类型，则会用到运行时缓存机制，避免下次调用重复查找。

```
ZEND_VM_HANDLER(68, ZEND_NEW, CONST|VAR, ANY)
{
    ...
    zend_class_entry *ce;

    SAVE_OPLINE();
    if (OP1_TYPE == IS_CONST) {
        //检查缓存
        ce = CACHED_PTR(Z_CACHE_SLOT_P(EX_CONSTANT(opline->op1)));
        if (UNEXPECTED(ce == NULL)) {
            //从 EG(class_table)查找类
            ce = zend_fetch_class_by_name(Z_STR_P(EX_CONSTANT(opline->op1)),
EX_CONSTANT(opline->op1) + 1, ZEND_FETCH_CLASS_DEFAULT | ZEND_FETCH_CLASS_
EXCEPTION);
            ...
```

```
            //将找到的类设置到缓存中
            CACHE_PTR(Z_CACHE_SLOT_P(EX_CONSTANT(opline->op1)), ce);
        }
    } else {
        ce = Z_CE_P(EX_VAR(opline->op1.var));
    }
    ...
}
```

2. 创建、初始化对象

找到类后接着将创建对象，并进行初始化：

```
//ZEND_VM_HANDLER(68, ZEND_NEW, CONST|VAR, ANY):
if (UNEXPECTED(object_init_ex(&object_zval, ce) != SUCCESS)) {
    HANDLE_EXCEPTION();
}
```

这个过程分为两步：第一步是分配 zend_object 内存，分配时会根据 zend_class_entry->default_properties_count 非静态属性的数量，将非静态成员属性的内存一起分配；第二步是初始化非静态成员属性，将非静态成员属性的默认值拷贝至 zend_object->properties_table。object_init_ex()最终将调用_object_and_properties_init()完成这两步的操作。

```
//file: zend_API.c
ZEND_API int _object_and_properties_init(zval *arg, zend_class_entry
*class_type, HashTable *properties ZEND_FILE_LINE_DC)
{
    ...
    if (class_type->create_object == NULL) {
        //由 ZendVM 默认的 handler 创建、初始化对象
        ZVAL_OBJ(arg, zend_objects_new(class_type));
        ...
        //初始化非静态成员属性
        object_properties_init(Z_OBJ_P(arg), class_type);
    } else {
        //调用自定义的对象创建 handler
        ZVAL_OBJ(arg, class_type->create_object(class_type));
    }
```

```
    return SUCCESS;
}
```

1）创建对象

默认将调用 zend_objects_new()分配 zend_object，具体处理如下：

```
//file: zend_objects.c
ZEND_API zend_object *zend_objects_new(zend_class_entry *ce)
{
    //分配 zend_object
    zend_object *object = emalloc(sizeof(zend_object) + zend_object_
properties_size(ce));

    zend_object_std_init(object, ce);
    //设置对象的操作 handler 为 std_object_handlers
    object->handlers = &std_object_handlers;
    return object;
}
```

分配 zend_object 时需要加上非静态属性所占用的内存大小：zend_object_properties_size()，根据非静态属性的个数确定。如果没有定义__get()、__set()等魔术方法，则占用内存就是属性数*sizeof(zval)，如果定义了这些魔术方法，那么会多分配一个 zval 的空间。该步骤还会设置前面介绍的对象的处理 handlers，默认的就是 std_object_handlers，所以，如果你想在扩展中自定义这些 handlers 的话，需要覆盖类的 create_object 接口，在自定义的创建对象的操作中进行设置。

```
//file: zend_objects_API.h
static zend_always_inline size_t zend_object_properties_size(zend_class_
entry *ce)
{
    //如果定义了__get、__set 等，则 ce_flags & ZEND_ACC_USE_GUARDS
    return sizeof(zval) *
        (ce->default_properties_count -
            ((ce->ce_flags & ZEND_ACC_USE_GUARDS) ? 0 : 1));
}
```

2）初始化非静态成员属性

zend_object 及非静态成员属性的内存分配完后，将进行属性的初始化。非静态成员属性的默认值保存在类的 default_properties_table 数组中，初始化时根据属性的 offset 将各属性的 value

复制至对象的属性。当然这里不是深拷贝，两者当前指向的 value 还是同一份，除非对象试图改写指向的属性值，那时将触发写时复制机制重新复制一份（ZTS 下数组、普通字符串会发生深拷贝）。

```
    //file: zend_API.c
    ZEND_API void object_properties_init(zend_object *object, zend_class_entry
*class_type)
    {
        if (class_type->default_properties_count) {
            //将属性值从 default_properties_table 复制至 properties_table
            zval *src = class_type->default_properties_table;
            zval *dst = object->properties_table;
            zval *end = src + class_type->default_properties_count;

            do {
                //非 ZTS
                ZVAL_COPY(dst, src);
                src++;
                dst++;
            } while (src != end);
            object->properties = NULL;
        }
    }
```

示例最后生成的非静态成员属性的内存结构如图 7-8 所示。

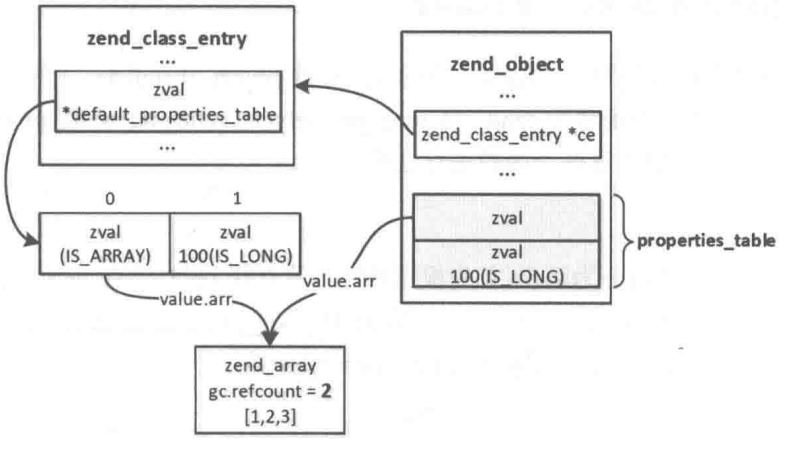


图 7-8 对象非静态成员属性的内存

3. 调用构造方法

如果类定义了构造方法，则在对象创建完成后初始化调用方法，然后将执行构造方法；如果没有则直接跳过构造方法的调用。

```
ZEND_VM_HANDLER(68, ZEND_NEW, CONST|VAR, ANY)
{
    zend_function *constructor;
    ...
    //获取构造方法
    constructor = Z_OBJ_HT(object_zval)->get_constructor(Z_OBJ(object_zval));

    if (constructor == NULL) {
        //跳过
        ...
        ZEND_VM_JMP(OP_JMP_ADDR(opline, opline->op2));
    } else {
        //初始化构造方法
        zend_execute_data *call = zend_vm_stack_push_call_frame(...);
        call->prev_execute_data = EX(call);
        EX(call) = call;
        ...
    }
}
```

7.2.2 非静态成员属性的读写

非静态成员属性的读写处理 handler 分别为 zend_object->handlers 中的 read_property、write_property，默认的处理函数为 zend_std_read_property()、zend_std_write_property()，当通过“→”操作成员属性时，将由这两个函数进行处理。

1. 读取

非静态成员属性的读取分为两步：首先根据属性名称查找 zend_class_entry->properties_info，找到属性的 zend_property_info 结构，并检查是否有该属性的操作权限，然后根据 zend_property_info->offset 从 zend_object->properties_table 数组中获取对应的属性。

```
//file: zend_object_handlers.c
```

```
    zval *zend_std_read_property(zval *object, zval *member, int type, void
**cache_slot, zval *rv)
    {
        zend_object *zobj;
        uint32_t property_offset;

        zobj = Z_OBJ_P(object);

        //根据属性名在 zend_class_entry->zend_property_info 中查找 zend_property_
info，得到属性值在 zend_object 中的存储 offset
        //注意:zend_get_property_offset()会对属性的权限(public、private、protected)
进行验证
        property_offset = zend_get_property_offset(zobj->ce, Z_STR_P(member),
(type == BP_VAR_IS) || (zobj->ce->__get != NULL), cache_slot);

        if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET)) {
            if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OFFSET)) {
                //直接根据 offset 取到属性值: ((zval*)((char*)(zobj) + offset))
                retval = OBJ_PROP(zobj, property_offset);
            } else if (EXPECTED(zobj->properties != NULL)) {
                //动态属性的情况，没有在类中显式定义的属性，后面一节会单独介绍
                ....
            }
        }
        ...
    }
```

如果没有找到属性，则会检查这个类是否定义了__get()、__set()魔术方法，如果定义了则将调用魔术方法进行处理：

```
    //zend_std_read_property:
    //没有找到属性
    //调用魔术方法: __get()
    if (zobj->ce->__get) {
        //将属性插入 guard 哈希表，即 zobj->properties_table[zobj->ce->default_
properties_count]
        zend_long *guard = zend_get_property_guard(zobj, Z_STR_P(member));
        ...
```

```
    //检查该属性是否已经在__get中了
    if(!((*guard) & IN_ISSET)){
        *guard |= IN_ISSET;
        zend_std_call_issetter(&tmp_object, member, &tmp_result);
        *guard &= ~IN_ISSET;
        ...
    }
}
```

如果类定义了__get()方法，则在实例化对象分配 properties_table 时会多分配一个 zval，类型为HashTable，称之为guard，每次调用__get($var)时会把输入的$var名称存入这个哈希表，这样做的目的是防止循环调用。举个例子：

```
public function __get($var) {
    return $this->$var;
}
```

这个例子调用__get()时又访问了一个没有定义的属性，也就是会在__get()方法中递归调用，如果不对请求的$var 进行判断则将一直递归下去，所以在调用__get()前首先会判断当前$var是不是已经在guard哈希表中了，如果是则不会再调用__get()，否则会把$var作为key插入guard哈希表，然后将它的值打上 IN_ISSET 的标记（*guard |= IN_ISSET），调用完__get()再把这个标记擦除（*guard &= ~IN_ISSET）。

这个 HashTable 并不是只提供给__get()用的，其他魔术方法也会用到，其哈希值类型是zend_long，不同的魔术方法占不同的 bit 位；其次，并不是所有的对象都会额外分配这个HashTable，在对象创建时会根据 zend_class_entry->ce_flags 是否包含 ZEND_ACC_USE_GUARDS 确定是否分配，在类编译时如果发现定义了__get()、__set()、__unset()、__isset()方法则会将 ce_flags 打上这个标识。

2. 修改

属性的修改是写操作，该操作同样分为两步：首先也是查找属性，然后修改属性的 value。如果属性没有找到且定义了__set()魔术方法，则调用该方法进行处理，其逻辑与读取的基本一致。

```
//file: zend_object_handlers.c
ZEND_API void zend_std_write_property(zval *object, zval *member, zval
*value, void **cache_slot)
```

```
    {
        zend_object *zobj;
        uint32_t property_offset;

        zobj = Z_OBJ_P(object);

        //与读取属性相同
        property_offset = zend_get_property_offset(zobj->ce, Z_STR_P(member),
(zobj->ce->__set != NULL), cache_slot);

        if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET)) {
            if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OFFSET)) {
                //非静态属性
                variable_ptr = OBJ_PROP(zobj, property_offset);
                if (Z_TYPE_P(variable_ptr) != IS_UNDEF) {
                    goto found; //即: zend_assign_to_variable(variable_ptr, value,
IS_CV);
                }
            } else if (EXPECTED(zobj->properties != NULL)) {
                ...
            }
        }
        ...
        //没有找到属性
        //如果定义了__set()则调用
        if (zobj->ce->__set) {
            //与__get()相同, 也会判断 set 的变量名是否已经在__set()中
            ...
            ZVAL_COPY(&tmp_object, object);
            (*guard) |= IN_SET; //防止循环__set()
            if (zend_std_call_setter(&tmp_object, member, value) != SUCCESS) {
            }
            (*guard) &= ~IN_SET;
        }else if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET)) {
            ...
        }
    }
```

7.2.3 对象的复制

PHP 中普通变量的复制可以通过直接赋值完成，比如：

```
$a = array();
$b = $a;
```

对象无法像数组、字符串等类型那样，通过赋值进行复制，仅仅通过赋值传递对象，它们指向的都是同一个对象，修改时也不会发生深拷贝。比如上面这个例子，我们把$a 赋值给$b，然后如果我们修改$b 的内容，那么这时候会进行 value 分离，$a 的内容是不变的，但是如果是把一个对象赋值给了另一个变量，这两个对象不管哪一个修改，另外一个都随之改变。例如：

```
class my_class
{
    public $arr = array();
}

$a = new my_class;
$b = $a;
$b->arr[] = 1;
var_dump($a === $b);

//输出: bool(true)
```

第 4 章介绍变量的写时复制机制时，曾介绍过 zval 的 type_flag，只有 IS_TYPE_COPYABLE 标识的类型在变量发生分离时才可以进行深拷贝，而 object 类型是没有这个标识的，因此，对象是无法通过这个机制进行分离的。在实际应用中，我们经常把一个对象传给多个函数进行修改，传递的时候并不需要像其他类型那样将参数设置为引用，其原因正是这个。

PHP 提供了另外一个关键词来实现对象的复制：clone。

```
$copy_of_object = clone $object;
```

clone 时会把对象的所有属性都复制一份（当然这里并不会深拷贝），这样 clone 出的对象就与原来的对象完全隔离了，各自修改都不会相互影响，另外如果类中定义了__clone()魔术方法，那么在 clone 时将调用此函数。clone 的实现比较简单，通过 zend_object->clone_obj（即 zend_objects_clone_obj()）进行处理：

```
//zend_objects.c
ZEND_API zend_object *zend_objects_clone_obj(zval *zobject)
{
    zend_object *old_object;
    zend_object *new_object;

    old_object = Z_OBJ_P(zobject);
    //重新分配一个 zend_object
    new_object = zend_objects_new(old_object->ce);

    //复制 properties_table、properties
    //如果定义了 __clone() 则调用此方法
    zend_objects_clone_members(new_object, old_object);

    return new_object;
}
```

7.2.4 对象的比较

当使用比较运算符“==”比较两个对象变量时，比较的原则是：如果两个对象的属性和属性值都相等，而且两个对象是同一个类的实例，那么这两个对象变量相等；而如果使用全等运算符“===”，这两个对象变量一定要指向某个类的同一个实例（即同一个对象）。

PHP 中对象间的“==”比较通过函数 zend_std_compare_objects()进行处理：

```
static int zend_std_compare_objects(zval *o1, zval *o2)
{
    ...

    if (zobj1->ce != zobj2->ce) {
        return 1; /* different classes */
    }
    if (!zobj1->properties && !zobj2->properties) {
        //逐个比较 properties_table 中属性
        ...
    }else{
        //比较 properties
```

```
            return zend_compare_symbol_tables(zobj1->properties, zobj2->properties);
        }
    }
```

“===”的比较通过函数 zend_is_identical()处理，比较简单：

```
ZEND_API int ZEND_FASTCALL zend_is_identical(zval *op1, zval *op2)
{
    switch (Z_TYPE_P(op1)) {
        ...
        case IS_OBJECT:
            //对象必须是同一个
            return (Z_OBJ_P(op1) == Z_OBJ_P(op2) && Z_OBJ_HT_P(op1) ==
Z_OBJ_HT_P(op2));
        ...
    }
}
```

7.2.5 对象的销毁

object 与 string、array 等类型不同，它是个复合类型，所以它的销毁过程更加复杂。销毁一个对象由 zend_objects_store_del() 完成，销毁的主要操作有：清理成员属性，从 EG(objects_store).object_buckets 中删除，释放 zend_object 内存，等等。

```
//file: zend_objects_API.c
ZEND_API void zend_objects_store_del(zend_object *object)
{
    ...
    if (GC_REFCOUNT(object) > 0) {
        GC_REFCOUNT(object)--;
        return;
    }
    ...
    //调用 dtor_obj，默认 zend_objects_destroy_object()
    //接着调用 free_obj，默认 zend_object_std_dtor()
    object->handlers->dtor_obj(object);
    object->handlers->free_obj(object);
```

```
    ...
    ptr = ((char*)object) - object->handlers->offset;
    efree(ptr);
}
```

另外，前面介绍垃圾回收时曾介绍过两种因循环引用导致的无法正常回收的垃圾，一种是数组，另外一种类型就是对象。在减少 refcount 时如果发现 object 的引用计数大于 0，则将加入垃圾回收器。具体的处理过程前面已经详细介绍过了，这里不再赘述。

7.3 继承

继承是面向对象非常重要的三个特性之一，它允许创建分等级层次的类，它允许子类继承父类所有公有或受保护的特征和行为，使得子类对象具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。继承对于功能的设计和抽象是非常有用的，对于类似的对象增加新功能时就无须重新再写这些公用的功能。

PHP 中通过 extends 关键词继承一个父类，一个类只允许继承一个父类，但是可以多级继承。

```
class 父类 {
}

class 子类 extends 父类 {
}
```

如果有继承的父类，则该类的 zend_class_entry->parent 将指向父类的结构。含有继承类的类与不含继承类的类在编译时会有一些差异，含继承类的类，在注册前需要首先找到父类，并继承父类的常量、属性、方法，因此父类必须要在子类注册前注册。本章第一节时曾简单提过，含有继承类的类在编译生成类声明指令时，生成的是 ZEND_DECLARE_INHERITED_CLASS，而非继承类生成的则是 ZEND_DECLARE_CLASS。同时，含有继承类的类在 ZEND_DECLARE_INHERITED_CLASS 前还会生成一条 ZEND_FETCH_CLASS 指令。

```
void zend_compile_class_decl(zend_ast *ast)
{
    zend_ast_decl *decl = (zend_ast_decl *) ast;
    zend_ast *extends_ast = decl->child[0]; //继承类节点，zen_ast_zval 节点，
存的是父类名
```

```
    zend_ast *implements_ast = decl->child[1]; //实现接口节点
    zend_ast *stmt_ast = decl->child[2]; //类中声明的常量、属性、方法
    zend_string *name, *lcname;
    zend_class_entry *ce = zend_arena_alloc(&CG(arena), sizeof(zend_class_entry));
    zend_op *opline;
    ...
    if (extends_ast) {
        ...
        //有继承的父类则首先生成一条 ZEND_FETCH_CLASS 的指令
        zend_compile_class_ref(&extends_node, extends_ast, 0);
    }

    //在当前父空间生成一条 opcode
    opline = get_next_op(CG(active_op_array));
    zend_make_var_result(&declare_node, opline);
    ...
    opline->op2_type = IS_CONST;
    LITERAL_STR(opline->op2, lcname);

    if (decl->flags & ZEND_ACC_ANON_CLASS) {
        ...
    }else{
        zend_string *key;

        if (extends_ast) {
            opline->opcode = ZEND_DECLARE_INHERITED_CLASS; //有继承的类生成的
是这条指令
            opline->extended_value = extends_node.u.op.var;
        } else {
            opline->opcode = ZEND_DECLARE_CLASS; //无继承的类生成的是这条
        }
    }
    ...
}
```

接下来编译类常量、成员属性、成员方法的过程与普通类没有差别，最后一个关键的、不同的步骤在于 zend_do_early_binding()，也就是类注册的环节。普通类的处理比较简单，直接注册到 EG(class_table)即可，但是含有继承类的类不能这样简单地处理，因为它有依赖的父类，

只有父类合法注册后子类才能注册。也就是说，子类注册的前提是 parent 指针不能为空。编译生成的 ZEND_FETCH_CLASS 指令就是用来获取父类的。

```
    //zend_do_early_binding:
    case ZEND_DECLARE_INHERITED_CLASS: //声明子类
    {
        zend_op *fetch_class_opline = opline-1;
        zval *parent_name;
        zend_class_entry *ce;

        parent_name = CT_CONSTANT(fetch_class_opline->op2); //父类名

        //在EG(class_table)中查找父类
        if (((ce = zend_lookup_class_ex(Z_STR_P(parent_name), parent_name + 1,
0)) == NULL) || ...) {
            //没找到父类，有可能父类没有定义，或者父类是在子类之后定义的
            ...
            return;
        }
        //注册继承类
        if (do_bind_inherited_class(CG(active_op_array), opline, CG(class_table),
ce, 1) == NULL) {
            return;
        }
        //清理无用的指令：ZEND_FETCH_CLASS，重置为0，执行时直接跳过
        zend_del_literal(CG(active_op_array),
fetch_class_opline->op2.constant);
        MAKE_NOP(fetch_class_opline);

        table = CG(class_table);
        break;
    }
```

这么单独看可能不太容易理解，我们整体来看一下：

```
    class Parents {
    }
    class Son extends Parent {
    }
```

Parents 类没有父类，因此在编译过程中在 zend_do_early_binding()时注册到了 EG(class_table)。编译 Son 类时，生成了 ZEND_FETCH_CLASS、ZEND_DECLARE_INHERITED_CLASS 两条指令，ZEND_DECLARE_INHERITED_CLASS 在 zend_do_early_binding()环节首先查找父类，此时父类已经成功注册了，因此可以找得到，找到后将调用 do_bind_inherited_class()继承父类的数据，这里面的逻辑我们稍后再看。继承后子类的 parent 指向了父类，此时子类也成功编译完成了，接着将 ZEND_FETCH_CLASS、ZEND_DECLARE_INHERITED_CLASS 指令重置为空指令。

这个过程中并没有看到 ZEND_FETCH_CLASS 指令起到什么作用，这是因为这条指令是在运行时用到的，也就是子类在 zend_do_early_binding()时没有找到父类，这种情况下子类还不能注册到 EG(class_table)中，因为找不到父类则表示这个子类是不完整的、有缺陷的，此时 ZEND_FETCH_CLASS、ZEND_DECLARE_INHERITED_CLASS 指令将保留到运行时执行。例如下面这种情况：

```
//son.php
class Son extends Parent {
}
class Parents {
}
```

这个例子首先定义的是 Son 类，然后才是父类 Parents，zend_do_early_binding()时首先尝试注册的也是 Son 类，这个时候发现找不到 Parents 类，则终止注册；接着注册 Parents，注册成功。编译后生成的指令如图 7-9 所示。

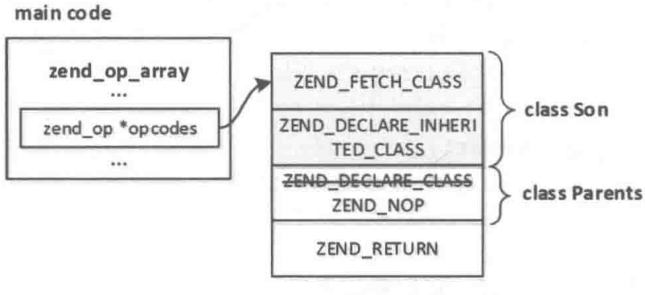


图 7-9 未完成注册的 Son 类的声明指令

执行时首先由 ZEND_FETCH_CLASS 指令获取父类，虽然 Parents 在 Son 之后定义，但是 Parents 在编译阶段提前注册了，所以此时 ZEND_FETCH_CLASS 能够找到父类 Parents：

```
ZEND_VM_HANDLER(109, ZEND_FETCH_CLASS, ANY, CONST|TMPVAR|UNUSED|CV)
```

```
    {
        ...
        if (OP2_TYPE == IS_CONST) {
            //先查找缓存
            zend_class_entry *ce = CACHED_PTR(Z_CACHE_SLOT_P(class_name));

            if (UNEXPECTED(ce == NULL)) {
                //查找父类是否存在
                ce = zend_fetch_class_by_name(Z_STR_P(class_name), EX_CONSTANT
(opline->op2) + 1, opline->extended_value);
                CACHE_PTR(Z_CACHE_SLOT_P(class_name), ce);
            }
            //将找到的父类存储在 result 操作数指定的位置
                Z_CE_P(EX_VAR(opline->result.var)) = ce;
        }
    }
```

成功 fetch 父类后，接下来将继续执行 ZEND_DECLARE_INHERITED_CLASS 继承父类，这个过程仍然是调用 do_bind_inherited_class()完成的，所以，实际上这两条指令的工作就是 zend_do_early_binding()中的，只是执行时机不同而已：

```
    ZEND_VM_HANDLER(140, ZEND_DECLARE_INHERITED_CLASS, ANY, ANY)
    {
        USE_OPLINE

        SAVE_OPLINE();
        //继承并注册子类
        Z_CE_P(EX_VAR(opline->result.var)) = do_bind_inherited_class(&EX(func)
->op_array, opline, EG(class_table), Z_CE_P(EX_VAR(opline->extended_value)), 0);
        ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION();
    }
```

通过上面的过程我们可以看到，有些情况下父类并不需要在子类定义前定义，这种情况能够正常工作的原因就是：后定义的父类在编译阶段先于子类注册了。但是实际使用时并不建议这样做，原则上父类必须先于子类定义，否则有些情况下是会出错的。这种情况下就体现出自动加载机制的价值了，通过自动加载机制，能够在子类在 fetch 父类时首先将父类注册了。下面看几个不同情况下实际指令的执行情况：

```
//情况 1
new A();

class A extends B{}
class B{}
====================
1 ZEND_NEW      => 执行到这报错，因为此时 A 因为找不到 B 尚未编译进 EG(class_table)
2 ZEND_DO_FCALL
3 ZEND_FETCH_CLASS
4 ZEND_DECLARE_INHERITED_CLASS
5 ZEND_DECLARE_CLASS          => 注册 class B
6 ZEND_RETURN

实际执行顺序：5→1→2→3→4→6

//情况 2
class A extends B{}
class B{}

new A();
====================
1 ZEND_FETCH_CLASS
2 ZEND_DECLARE_INHERITED_CLASS => 注册 class A，此时已经可以找到 B
3 ZEND_DECLARE_CLASS           => 注册 class B
4 ZEND_NEW
5 ZEND_DO_FCALL
6 ZEND_RETURN

实际执行顺序：3→1→2→4→5→6，执行到 4 时 A 都已经注册，所以能够正常执行

//情况 3
class A extends B{}
class B extends C{}
class C{}

new A();
====================
```

```
1 ZEND_FETCH_CLASS                  => 找不到B,直接报错
2 ZEND_DECLARE_INHERITED_CLASS
3 ZEND_FETCH_CLASS
4 ZEND_DECLARE_INHERITED_CLASS => 注册class B，此时可以找到C，所以注册成功
5 ZEND_DECLARE_CLASS            => 注册class C
6 ZEND_NEW
7 ZEND_DO_FCALL
8 ZEND_RETURN

实际执行顺序：5→1→2→3→4→6→7→8，执行到1发现还是找不到父类B，报错
```

接下来我们将具体看一下 do_bind_inherited_class()中父子类间常量、属性、方法的继承逻辑，这个过程中子类会将父类的这些数据合并到子类中。

```
ZEND_API zend_class_entry *do_bind_inherited_class(
    const zend_op_array *op_array, //这个是定义类的地方的
    const zend_op *opline, //类声明的opcode: ZEND_DECLARE_INHERITED_CLASS
    HashTable *class_table, //CG(class_table)
    zend_class_entry *parent_ce,  //父类
    zend_bool compile_time) //是否编译时
{
    zend_class_entry *ce;
    zval *op1, *op2;

    if (compile_time) {
       op1 = CT_CONSTANT_EX(op_array, opline->op1.constant);
       op2 = CT_CONSTANT_EX(op_array, opline->op2.constant);
    }else{
       ...
    }
    ...
    //父子类关联
    zend_do_inheritance(ce, parent_ce);
    //注册到CG(class_table)
    ...
}
```

上面这个函数的处理与注册非继承类的 do_bind_class()几乎完全相同，只是多了

zend_do_inheritance()这一步，该过程就是父子类继承的具体处理逻辑：

```
//file: zend_inheritance.c
ZEND_API void zend_do_inheritance(zend_class_entry *ce, zend_class_entry
*parent_ce)
{
    zend_property_info *property_info;
    zend_function *func;
    zend_string *key;
    zval *zv;
    //interface、trait、final 类检查
    ...
    ce->parent = parent_ce;

    zend_do_inherit_interfaces(ce, parent_ce);
    //下面就是继承属性、常量、方法
    ...
}
```

我们根据一个示例具体来看一下常量、属性、成员方法的合并逻辑：

```
class A {
    const A1 = 1;
    public $a1 = array(1);
    private $a2 = 120;

    public function get() {
        echo "A::get()";
    }
}
class B extends A {
    const B1 = 2;

    public $b1 = "ddd";

    public function get() {
        echo "B::get()";
    }
}
```

7.3.1　常量的继承

常量的合并策略比较简单，父类与子类冲突时用子类的常量，不冲突时则将父类的常量合并到子类。

```
//zend_do_inheritance:
//变量父类的常量符号表
ZEND_HASH_FOREACH_STR_KEY_VAL(&parent_ce->constants_table, key, zv) {
    do_inherit_class_constant(key, zv, ce, parent_ce);
} ZEND_HASH_FOREACH_END();

static void do_inherit_class_constant(zend_string *name, zval *zv,
zend_class_entry *ce, zend_class_entry *parent_ce)
{
    //父类定义的常量在子类中没有定义
    if (!zend_hash_exists(&ce->constants_table, name)) {
        ...
        //将父类的常量合并到子类
        _zend_hash_append(&ce->constants_table, name, zv);
    }
}
```

示例中，类 A 的常量：A1 将被合并到类 B 中，在 B 中就可以直接使用 B::A1 了，如图 7-10 所示。

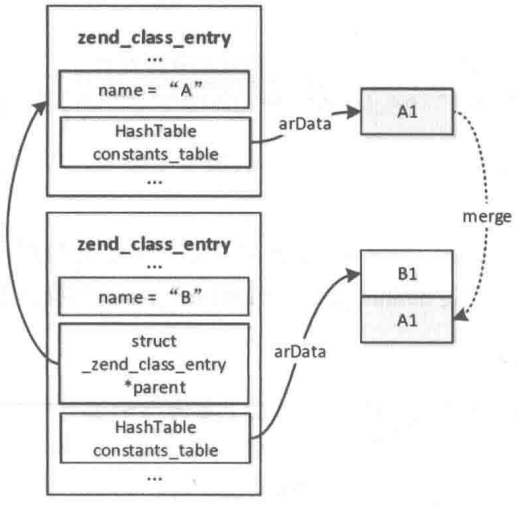


图 7-10　子类继承父类的常量

7.3.2 成员属性的继承

成员属性的继承也是将父类的静态成员属性、非静态成员属性分别合并到子类中去，具体过程分为以下几步：

1. 合并非静态成员属性

首先申请一个父类+子类非静态属性大小的数组，然后先将父类非静态属性拷贝到新数组，然后再将子类的非静态数组接着父类属性的位置复制过去，子类的 default_properties_table 指向合并后的新数组，default_properties_count 更新为新数组的大小，最后将子类旧的数组释放。

```
//zend_do_inheritance:
if (parent_ce->default_properties_count) {
    zval *src, *dst, *end;
    ...
zval *table = pemalloc(sizeof(zval) * (ce->default_properties_count +
parent_ce->default_properties_count), ...);

    ce->default_properties_table = table;

    //复制父类、子类 default_properties_table
    do {
        ...
}while(dst != end);

    //更新 default_properties_count 为合并后的大小
    ce->default_properties_count += parent_ce->default_properties_count;
}
```

2. 合并静态成员属性

与非静态属性相同，新申请一个父类+子类静态属性大小的数组，依次将父类、子类静态属性复制到新数组，然后更新子类 default_static_members_table 指向新数组。

3. 更新子类属性 offset

因为合并后原子类属性整体向后移了，所以子类属性的编号 offset 需要加上前面父类属性的总大小。

```
//zend_do_inheritance:
```

```
    ZEND_HASH_FOREACH_PTR(&ce->properties_info, property_info) {
        if (property_info->ce == ce) {
            if (property_info->flags & ZEND_ACC_STATIC) {
                //静态属性 offset 为数组下标，直接加上父类 default_static_members_
count 即可
                property_info->offset += parent_ce->default_static_members_count;
            } else {
                //非静态属性 offset 为内存偏移值，按 zval 大小递增
                property_info->offset += parent_ce->default_properties_count *
sizeof(zval);
            }
        }
    } ZEND_HASH_FOREACH_END();
```

4. 合并属性信息哈希表

这也是非常关键的一步，上面只是将父类的属性值合并到了子类，但是索引属性用的是properties_info 哈希表，所以需要将父类的属性索引表与子类的索引表合并，这个过程也决定了父类有哪些属性可以被子类继承和覆盖。

```
    //zend_do_inheritance:
    if (zend_hash_num_elements(&parent_ce->properties_info)) {
        //先将 properties_info 扩容
        zend_hash_extend(&ce->properties_info,
            zend_hash_num_elements(&ce->properties_info) +
            zend_hash_num_elements(&parent_ce->properties_info), 0);
        //合并 properties_info
        ZEND_HASH_FOREACH_STR_KEY_PTR(&parent_ce->properties_info, key,
property_info) {
            do_inherit_property(property_info, key, ce);
        } ZEND_HASH_FOREACH_END();
    }
```

具体的合并策略在 do_inherit_property()方法之中，父类的私有属性无法被子类继承，如子类覆盖了父类中的属性，则只能放大属性的权限而不能缩小，比如父类中的属性为 public，则子类中不能覆盖为 protected、private，具体的判断逻辑这里不再展开。

7.3.3 成员方法的继承

与属性一样，子类可以继承父类的公有、受保护的方法，方法的继承比较复杂，因为会有访问控制、抽象类、接口、Trait 等多种限制条件。实现上与前面几种相同，也是将父类的 function_table 合并到子类的 function_table 中：首先将子类 function_table 扩大，以容纳父子类全部方法，然后遍历父类 function_table，逐个判断是否可被子类继承，如果可被继承则插入到子类 function_table 中。

```
if (zend_hash_num_elements(&parent_ce->function_table)) {
    //扩展子类的 function_table 哈希表大小
    zend_hash_extend(&ce->function_table,
            zend_hash_num_elements(&ce->function_table) +
            zend_hash_num_elements(&parent_ce->function_table), 0);

    //遍历父类 function_table，检查是否可被子类继承
    ZEND_HASH_FOREACH_STR_KEY_PTR(&parent_ce->function_table, key, func) {
        //do_inherit_method 检查方法能否被继承
        zend_function *new_func = do_inherit_method(key, func, ce);

        if (new_func) {
            //将父类方法插入子类
            _zend_hash_append_ptr(&ce->function_table, key, new_func);
        }
    } ZEND_HASH_FOREACH_END();
}
```

7.4 动态属性

前面介绍的成员属性都是在类中明确的定义过的，这些属性在实例化时会被复制到对象空间中去。PHP 中除了显式地在类中定义成员属性，还可以动态地创建非静态成员属性，这种属性不需要在类中明确定义，可以直接通过$obj->property_name=xxx 为对象设置一个属性，这种属性称之为动态属性，例如：

```
class my_class {
    public $id = 123;
}
```

```
$obj = new my_class;
$obj->prop_1 = array(1,2,3);
print_r($obj);
```

执行后将输出：

```
my_class Object
(
    [id] => 123
    [prop_1] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
        )
)
```

前面类、对象两节曾介绍，非静态成员属性值在实例化时保存到了对象中，属性的操作按照编译时顺序编好的序号操作，各对象对其非静态成员属性的操作互不干扰，那么动态属性是在运行时创建的，它是如何存储的呢？与普通非静态属性不同，动态创建的属性保存在zend_object->properties 哈希表中，查找的时候首先按照普通属性在 zend_class_entry->properties_info 中找，没有找到再去 zend_object->properties 中继续查找。创建属性时的操作：

```
//zend_object->handlers->write_property:
ZEND_API void zend_std_write_property(zval *object, zval *member, zval
*value, void **cache_slot)
{
    ...
    zobj = Z_OBJ_P(object);
    //先在 zend_class_entry.properties_info 中查找此属性
    property_offset = zend_get_property_offset(zobj->ce, Z_STR_P(member),
(zobj->ce->__set != NULL), cache_slot);

    if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET)) {
        if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OFFSET)) {
            //普通属性，直接根据根据属性 ofsset 取出属性值
        } else if (EXPECTED(zobj->properties != NULL)) { //有动态属性
```

```
            ...
            //从动态属性中查找
            if ((variable_ptr = zend_hash_find(zobj->properties, Z_STR_P
(member))) != NULL) {found:
                zend_assign_to_variable(variable_ptr, value, IS_CV);
                goto exit;
            }
        }
    }

    if (zobj->ce->__set) {
        //定义了__set()魔法函数
    }else if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET)){
        if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OFFSET)) {
            ...
        } else {
            //首次创建动态属性将在这里完成
            if (!zobj->properties) {
                rebuild_object_properties(zobj);
            }
            //将动态属性插入properties
            zend_hash_add_new(zobj->properties, Z_STR_P(member), value);
        }
    }
}
```

上面就是成员属性的修改过程，普通属性根据其 offset 再从对象中取出属性值进行修改，而首次创建动态属性将通过 rebuild_object_properties()初始化 zend_object->properties 哈希表，后面再创建动态属性直接插入此哈希表。rebuild_object_properties()过程并不仅仅是创建一个 HashTable，还会将普通成员属性值插入到这个数组中，与动态属性不同，这里的插入并不是增加原 zend_value 的 refcount，而是创建了一个 IS_INDIRECT 类型的 zval，指向原属性值 zval，具体结构如图 7-11 所示。

这里之所以把原来的普通属性也插入到 properties 数组中，是因为 Zend 中有很多地方需要获取全部的属性，比如将对象转为数组、垃圾回收时等，这种情况下直接获取 zend_object->properties 数组即可，不然的话则需要将动态属性、普通属性合并，对象的处理的 handlers 中，get_properties 操作就是用来获取全部属性的。

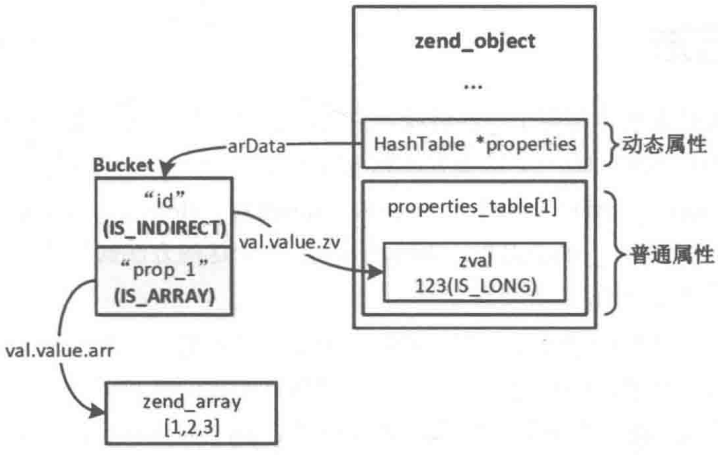


图 7-11 zend_object->properties 动态属性

成员属性的读取通过 zend_object->handlers->read_property（默认为 zend_std_read_property()）函数完成，动态属性的查找过程实际与 write_property 中相同：

```
    zval *zend_std_read_property(zval *object, zval *member, int type, void
**cache_slot, zval *rv)
    {
        ...
        zobj = Z_OBJ_P(object);
        //首先查找 zend_class_entry.properties_info，普通属性可以在这里找到
        property_offset = zend_get_property_offset(zobj->ce, Z_STR_P(member),
(type == BP_VAR_IS) || (zobj->ce->__get != NULL), cache_slot);

        if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET)) {
            if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OFFSET)) {
                //普通属性
                retval = OBJ_PROP(zobj, property_offset);
            } else if (EXPECTED(zobj->properties != NULL)) {
                //动态属性从 zend_object->properties 中查找
                retval = zend_hash_find(zobj->properties, Z_STR_P(member));
                if (EXPECTED(retval)) goto exit;
            }
        }
        ...
    }
```

7.5 魔术方法

PHP 在类的成员方法中预留了一些特殊的方法，它们会在一些特殊的时机被调用（比如创建对象之初、访问成员属性时…），这类方法称为魔术方法，包括：__construct()、__destruct()、__call()、__callStatic()、__get()、__set()、__isset()、__unset()、__sleep()、__wakeup()、__toString()、__invoke()、__set_state()、__clone()和__debugInfo()，关于这些方法的用法这里不作说明，不清楚的读者可以查看官方文档。

魔术方法实际上是 PHP 提供的一些特殊操作时的钩子函数，与普通成员方法无异，它们只是与一些操作的口头约定，并没有什么字段标识它们，比如我们定义了一个函数：my_function()，我们希望在这个函数在处理对象时首先调用其成员方法 my_magic()，那么 my_magic()也可以认为是一个魔术方法。魔术方法与普通成员方法一样保存在 zend_class_entry->function_table 中，另外，针对一些内核常用到的成员方法在 zend_class_entry 中还有一些单独的指针指向具体的成员方法：

```
struct _zend_class_entry {
    ...
    union _zend_function *constructor;
    union _zend_function *destructor;
    union _zend_function *clone;
    union _zend_function *__get;
    union _zend_function *__set;
    union _zend_function *__unset;
    union _zend_function *__isset;
    union _zend_function *__call;
    union _zend_function *__callstatic;
    union _zend_function *__tostring;
    union _zend_function *__debugInfo;
    ...
}
```

在编译成员方法时，如果发现与这些魔术方法名称一致，则除了插入 zend_class_entry->function_table 哈希表，还会设置 zend_class_entry 中对应的指针，这个操作是在成员方法编译时完成的。

```
void zend_begin_method_decl(zend_op_array *op_array, zend_string *name,
zend_bool has_body)
```

```
{
    ...
    //插入类的 function_table 中
    if (zend_hash_add_ptr(&ce->function_table, lcname, op_array) == NULL) {
        zend_error_noreturn(..);
    }
    //设置魔术方法指针
    if (!in_trait && zend_string_equals_ci(lcname, ce->name)) {
        if (!ce->constructor) {
            ce->constructor = (zend_function *) op_array;
        }
    } else if (zend_string_equals_literal(lcname, ZEND_CONSTRUCTOR_FUNC_
NAME)) {
        ce->constructor = (zend_function *) op_array;
    } else if (zend_string_equals_literal(lcname, ZEND_DESTRUCTOR_FUNC_NAME)) {
        ce->destructor = (zend_function *) op_array;
    } else if (...){
        ...
    }
    ...
}
```

比如一个类中定义了__construct()、__get()两个魔术方法，则 zend_class_entry 的结构如图 7-12 所示。

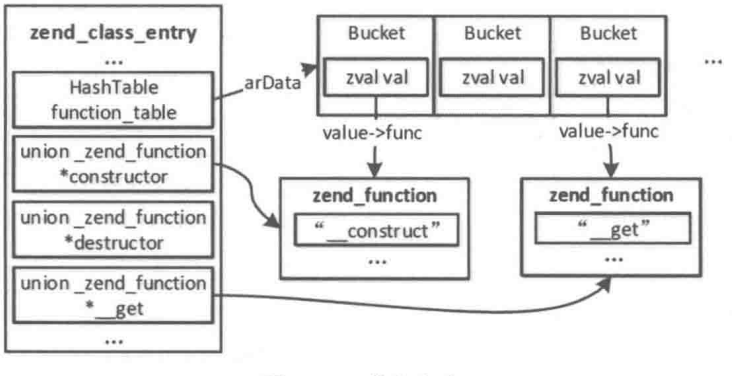


图 7-12 魔术方法

除了这几个其他魔术方法都没有单独的指针指向，比如__sleep()、__wakeup()，这两个主要是 serialize()、unserialize()序列化、反序列化时调用的，它们是在这俩函数中写死的，具体的

实现在 ext/standard/var.c 中。

```
//file: ext/standard/var.c
PHP_FUNCTION(serialize)
{
    zval *struc;
    php_serialize_data_t var_hash;
smart_str buf = {0};

    if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &struc) == FAILURE) {
        return;
    }

    php_var_serialize(&buf, struc, &var_hash);
    ...
}
```

最终由 php_var_serialize_intern()处理，这个函数会根据不同的类型选择不同的处理方式：

```
static void php_var_serialize_intern(smart_str *buf, zval *struc,
php_serialize_data_t var_hash)
{
    ...
    switch (Z_TYPE_P(struc)) {
        case IS_FALSE:
            ...
        case IS_TRUE:
            ...
        case IS_NULL:
            ...
        case IS_LONG:
            ...
    }
}
```

其中类型是对象时将先检查 zend_class_function.function_table 中是否定义了__sleep()，如果有的话则调用：

```
    //case IS_OBJEST:
    ...
    if (ce != PHP_IC_ENTRY && zend_hash_str_exists(&ce->function_table,
"__sleep", sizeof("__sleep")-1)) {
        ZVAL_STRINGL(&fname, "__sleep", sizeof("__sleep") - 1);
        //调用用户自定义的__sleep()方法
        res = call_user_function_ex(CG(function_table), struc, &fname, &retval,
0, 0, 1, NULL);
        ...
        //序列化
    }
```

其他魔术方法与__sleep()类似，都是在一些特殊操作中固定调用的。

7.6 小结

本章介绍了 PHP 中面向对象的相关实现，面向对象是 PHP 非常重要的一个特性，很难找到没有使用到面向对象的应用。本章主要介绍了面向对象中类、对象、继承等相关的内容，这是面向对象中最为基础的部分，除此之外还有很多面向对象的特性没有介绍，比如接口、抽象类等，这些内容需要大家自己去了解。

面向对象本身的实现原理并不复杂，从类、对象的实现来看，它只是将 PHP 中的常量、属性以及方法等数据封装在了一起，是抽象的一层。

第 8 章 命名空间

PHP 在 5.3 版本引用了命名空间的概念，如果你学过 C#和 Java，那命名空间就不算什么新事物。命名空间是在函数、类之上的一层概念，允许不同命名空间下定义相同名称的函数、类，从而解决不同库之间名称冲突的问题， 因此命名空间有着非常重要的意义。

本章将详细介绍命名空间在内核中的实现，主要包括以下两部分内容：

- 名称空间的定义；
- 命名空间的使用。

8.1 概述

什么是命名空间？从广义上来说，命名空间是一种封装事物的方法。在很多地方都可以见到这种抽象概念。例如，在操作系统中目录用来将相关文件分组，对于目录中的文件来说，它就扮演了命名空间的角色。举个例子，文件 foo.txt 可以同时在目录/home/greg 和 /home/other 中存在，但在同一个目录中不能存在两个 foo.txt 文件。另外，在目录/home/greg 外访问 foo.txt 文件时，我们必须将目录名及目录分隔符放在文件名之前得到/home/greg/foo.txt。这个原理应用到程序设计领域就是命名空间的概念。

命名空间主要用来解决两类问题：

- 用户编写的代码与 PHP 内部的或第三方的类、函数、常量、接口名字冲突。

- 为很长的标识符名称创建一个别名的名称，提高源代码的可读性。

PHP 命名空间提供了一种将相关的类、函数、常量和接口组合到一起的途径，不同命名空间的类、函数、常量、接口相互隔离不会冲突，特别注意：PHP 命名空间只能隔离类、函数、常量和接口，不包括全局变量。

8.2　命名空间的定义

命名空间通过关键字 namespace 来声明，如果一个文件中包含命名空间，它必须在其他所有代码之前声明命名空间，除了 declare 关键字，也就是说，除 declare 之外任何代码都不能在 namespace 之前声明。另外，命名空间并没有文件限制，可以在多个文件中声明同一个命名空间，也可以在同一文件中声明多个命名空间。

```
//方式一
//ns_define.php
namespace com\aa;

const MY_CONST = 1234;
function my_func(){ /* ... */ }
class my_class { /* ... */ }
```

另外也可以通过大括号直接将类、函数、常量封装在一个命名空间下，例如：

```
//方式二
namespace com\aa{
    const MY_CONST = 1234;
    function my_func(){ /* ... */ }
    class my_class { /* ... */ }
}
```

但是同一个文件中这两种定义方式不能混用，下面这样的定义将是非法的：

```
namespace com\aa{
    /* ... */
}

namespace com\bb;
/* ... */
```

如果没有定义任何命名空间，所有的类、函数和常量的定义都是在全局空间，与 PHP 引入命名空间概念前一样。

实际上，命名空间的实现非常简单，它只是在名称上进行了补全，其余的没有任何改动。当声明了一个命名空间后，接下来编译类、函数和常量时会把类名、函数名和常量名统一加上命名空间的名称作为前缀存储。也就是说，声明在命名空间中的类、函数和常量的实际名称是被修改过的，这样来看，它们与普通的定义方式没有任何区别，只是这个前缀是内核帮我们自动添加的。

以上面方式一为例，最终“MY_CONST”、“my_func”、“my_class”在 EG(zend_constants)、EG(function_table)、EG(class_table)中的实际存储名称被修改为“com\aa\MY_CONST”、“com\aa\my_func”、“com\aa\my_class”。接下来，我们从命名空间的编译过程入手，分析内核的具体处理。

namespace 语法被编译为 ZEND_AST_NAMESPACE 类型的抽象语法树节点，它有两个子节点：child[0]为命名空间的名称，child[1]为通过大括号方式定义时包裹的语句。方式一示例生成的抽象语法树如图 8-1 所示。

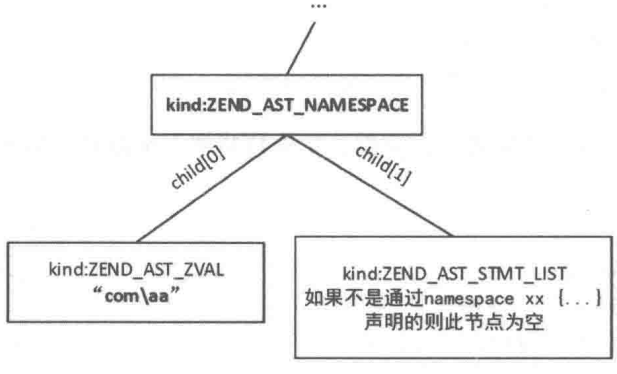


图 8-1 声明命名空间的抽象语法树

ZEND_AST_NAMESPACE 节点由 zend_compile_namespace()负责编译，先来看一下这个函数的编译处理：

```
void zend_compile_namespace(zend_ast *ast)
{
    //name_ast 保存着命名空间的名称
    zend_ast *name_ast = ast->child[0];
    zend_ast *stmt_ast = ast->child[1];
    zend_string *name;
```

```
    zend_bool with_bracket = stmt_ast != NULL;

    //检查声明方式，不允许通过大括号声明与非大括号声明混用
    ...
    //释放上一个编译的命名空间
    if (FC(current_namespace)) {
        zend_string_release(FC(current_namespace));
    }

    if (name_ast) {
        name = zend_ast_get_str(name_ast);

        if (ZEND_FETCH_CLASS_DEFAULT != zend_get_class_fetch_type(name)) {
            zend_error_noreturn(E_COMPILE_ERROR, "Cannot use '%s' as namespace
name", ZSTR_VAL(name));
        }
        //将命名空间名称保存到 FC(current_namespace)
        FC(current_namespace) = zend_string_copy(name);
    } else {
        FC(current_namespace) = NULL;
    }
    //重置 use 导入的命名空间符号表
    zend_reset_import_tables();
    ...
    if (stmt_ast) {
        //如果是通过 namespace xxx { ... }这种方式声明的则直接编译{}中的语句
        zend_compile_top_stmt(stmt_ast);
        zend_end_namespace();
    }
}
```

从上面的编译过程可以看出，命名空间定义的编译过程非常简单，最主要的操作是把 FC(current_namespace)设置为当前定义的命名空间名称，FC()这个宏为 CG(file_context)，file_context 是在编译过程中使用的一个辅助结构，在编译抽象语法树之前分配：

```
typedef struct _zend_file_context {
    zend_declarables declarables;
    znode implementing_class;
```

```
    //当前所属 namespace
    zend_string *current_namespace;
    //是否在 namespace 中
    zend_bool in_namespace;
    //当前 namespace 是否为{}定义
    zend_bool has_bracketed_namespaces;

    //下面这三个值在后面介绍 use 时再说明，这里忽略即可
    HashTable *imports;
    HashTable *imports_function;
    HashTable *imports_const;
} zend_file_context;
```

编译完 namespace 声明语句后，接着按照正常的逻辑编译后面的语句，此后定义的类、函数、常量均属于此命名空间，直到遇到下一个 namespace 的定义，它们的名称就是在编译的时候被加上了命名空间的前缀，接下来继续分析这三种类型编译过程中有何不同之处。

1）类、函数的编译

前面章节曾详细介绍过函数、类的编译过程，主要分为两步：第 1 步是编译函数、类，这个过程将分别生成一条 ZEND_DECLARE_FUNCTION、ZEND_DECLARE_CLASS 的 opcode；第 2 步是在整个脚本编译的最后执行 zend_do_early_binding()，这一步相当于执行 ZEND_DECLARE_FUNCTION、ZEND_DECLARE_CLASS，函数、类正是在这一步注册到 EG(function_table)、EG(class_table)中去的。

在生成 ZEND_DECLARE_FUNCTION、ZEND_DECLARE_CLASS 两条指令时会把函数名、类名的存储位置通过操作数记录下来，然后在 zend_do_early_binding()阶段直接获取函数名、类名作为 key 注册到 EG(function_table)、EG(class_table)中，定义在命名空间中的函数、类的名称修改正是在生成 ZEND_DECLARE_FUNCTION、ZEND_DECLARE_CLASS 时完成的。下面以函数为例看一下具体的处理：

```
//函数的编译方法
void zend_compile_func_decl(znode *result, zend_ast *ast)
{
    ...
    //生成函数声明的 opcode：ZEND_DECLARE_FUNCTION
    zend_begin_func_decl(result, op_array, decl);

    //编译参数、函数体
```

```
    ...
}
static void zend_begin_func_decl(znode *result, zend_op_array *op_array,
zend_ast_decl *decl)
{
    ...
    //获取函数名称
    op_array->function_name = name = zend_prefix_with_ns(unqualified_name);
    lcname = zend_string_tolower(name);

    if (FC(imports_function)) {
        //如果通过 use 导入了其他命名空间则检查函数名称是否已存在
    }
    ....
    //生成一条 opcode: ZEND_DECLARE_FUNCTION
    opline = get_next_op(CG(active_op_array));
    opline->opcode = ZEND_DECLARE_FUNCTION;
    //函数名的存储位置记录在 op2 中
    opline->op2_type = IS_CONST;
    LITERAL_STR(opline->op2, zend_string_copy(lcname));
    ...
}
```

函数名称通过 zend_prefix_with_ns()方法获取：

```
zend_string *zend_prefix_with_ns(zend_string *name) {
    if (FC(current_namespace)) {
        //如果当前是在 namespace 下则拼上 namespace 名称作为前缀
        zend_string *ns = FC(current_namespace);
        return zend_concat_names(ZSTR_VAL(ns), ZSTR_LEN(ns), ZSTR_VAL(name),
ZSTR_LEN(name));
    } else {
        return zend_string_copy(name);
    }
}
```

在 zend_prefix_with_ns()方法中如果发现 FC(current_namespace)不为空，则将函数名加上 FC(current_namespace)作为前缀，接下来向 EG(function_table)注册时就使用修改后的函数名作

为 key，类的情况与函数的处理方式相同，不再赘述。

2）常量的编译

常量也是在编译过程中获取常量名时检查 FC(current_namespace)是否为空，如果不为空表示常量声明在 namespace 下，则为常量名加上 FC(current_namespace)前缀。

```
void zend_compile_const(znode *result, zend_ast *ast)
{
    zend_ast *name_ast = ast->child[0];

    zend_op *opline;

    zend_bool is_fully_qualified;
    zend_string *orig_name = zend_ast_get_str(name_ast);
    //生成带有命名空间的名称
    zend_string *resolved_name = zend_resolve_const_name(orig_name,
name_ast->attr, &is_fully_qualified);
    ...
}
```

zend_resolve_const_name()最终也是调用 zend_prefix_with_ns()补全的名称。

8.3 命名空间的使用

上一节我们知道了定义在命名空间中的类、函数和常量只是加上了 namespace 名称作为前缀，既然是这样，那么在使用时加上同样的前缀是否就可以了呢？答案是肯定的，比如上面那个例子，在 com_CONST 中，可以这么使用：

```
include 'ns_define.php';
echo \com\aa\MY_CONST;
```

这种按照实际类名、函数名、常量名使用的方式很容易理解，与普通的类型没有任何差别，只是名称中多了"\"这种符号而已。这种以"\"开头使用的名称称之为完全限定名称，类似于绝对目录的概念，使用这种名称，PHP 会直接根据"\"之后的名称去对应的符号表中查找，namespace 定义时前面是没有加"\"的，所以查找时也会去掉这个字符。

除了完全限定名称，还有两种形式的名称。

- **非限定名称：**没有加任何 namespace 前缀的普通名称，比如 my_func()。使用这种名称

时如果当前脚本中声明了命名空间，则会被解析为 currentnamespace_func，如果没有声明命名空间则按照原始名称 my_func 解析。

- **部分限定名称**：包含 namespace 前缀，但不是以"\"开始的名称，比如 aa\func()，类似相对路径的概念。这种名称的解析规则：如果当前脚本没有使用 use 导入任何 namespace，那么与非限定名称的解析规则相同，即如果当前有命名空间则会解析为 currentnamespace\aa\my_func，否则解析为 aa\my_func；如果当前脚本使用 use 导入了 namespace，则解析规则就比较复杂了，接下来的一节将单独介绍 use 的用法。

8.3.1　use 导入

使用一个命名空间中的类、函数、常量虽然可以通过完全限定名称的形式访问，但是这种方式需要在每一处使用的地方都加上完整的 namespace 名称，如果将来 namespace 名称变更了就需要所有使用的地方都改一遍，这将是很痛苦的一件事。为此，PHP 提供了一种命名空间导入别名的机制，可以通过 use 关键字将一个命名空间导入或者定义一个别名，在使用时，可以通过导入的 namespace 名称最后一个域或者别名访问，不需要使用完整的名称，别名通过 as 指定。

namespace、use 两个关键词的角色类似 Java、Go 中的 package、import，分别用于命名空间的声明与导入。比如：

```
//ns_define.php
namespace aa\bb\cc\dd;
const MY_CONST = 1234;
```

脚本中，可以采用如下几种类型的名称：

```
//方式 1:
include 'ns_define.php';
use aa\bb\cc\dd;

echo dd\MY_CONST;

//方式 2:
include 'ns_define.php';
use aa\bb\cc;

echo cc\dd\MY_CONST;
```

```
//方式 3:
include 'ns_define.php';
use aa\bb\cc\dd as DD;

echo DD\MY_CONST;

//方式 4:
include 'ns_define.php';
use aa\bb\cc as CC;

echo CC\dd\MY_CONST;
```

这种机制的实现原理也比较简单：编译期间如果发现 use 语句 ，那么就把这个 use 后的命名空间名称插入一个哈希表 FC(imports)，而哈希表的 key 就是定义的别名，如果没有定义别名，则 key 默认使用按"\"分割的最后一节；接下来在使用类、函数和常量时会进行自动补全，把名称按"\"分割，然后以第一节为 key 查找 FC(imports)，如果找到了，则将 FC(imports)中保存的名称与使用时的名称拼接在一起，组成完整的名称。比如方式 2 的情况将以 cc 作为 key，即 FC(imports)["cc"] -> "aa\bb\cc"，使用部分限定名称访问常量：cc\dd\MY_CONST，编译时将名称分割，然后以"cc"为 key 查找 FC(imports)，得到完整的名称"aa\bb\cc"，此时就会拼成完整的名称，如图 8-2 所示。

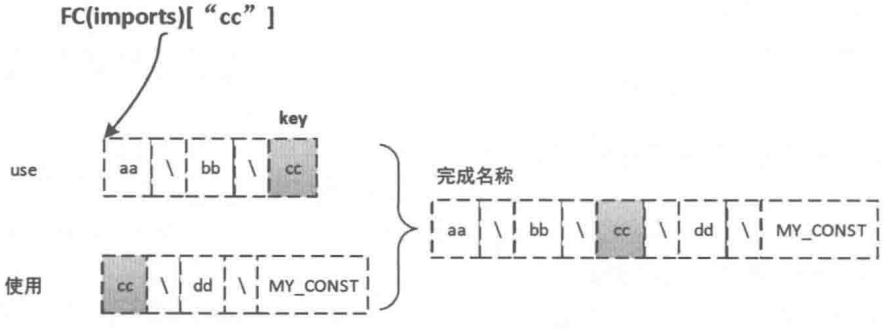


图 8-2 use 名称补全

实际上这种机制是把完整的名称切割缩短，然后缓存下来，使用时再拼接成完整的名称，也就是内核帮我们组装了名称，对内核而言，最终使用的都是包括完整 namespace 的名称。

use 除了上面介绍的用法，还可以导入一个类，导入后再使用类就不需要加 namespace 了，例如：

```
//ns_define.php
namespace aa\bb\cc\dd;
class my_class { /* ... */ }
```

导入类：

```
include 'ns_define.php';
//导入一个类
use aa\bb\cc\dd\my_class;
//直接使用
$obj = new my_class();
var_dump($obj);
```

以上 use 的这两种用法的实现原理是一样的，都是在编译时通过查找 FC(imports)实现的名称补全。从 PHP 5.6 起，use 又提供了两种针对函数、常量的导入，可以通过 use function xxx、use const xxx 语句导入函数、常量，这种用法的实现原理与上面介绍的实际是相同的，只是 use function、use const 在编译时没有导入到 FC(imports)哈希表中，而是分别导入到了 FC(imports_function)、FC(imports_const)中。

```
typedef struct _zend_file_context {
    ...
    //用于保存导入的类或命名空间
    HashTable *imports;
    //用于保存导入的函数
    HashTable *imports_function;
    //用于保存导入的常量
    HashTable *imports_const;
} zend_file_context;
```

此外还有一种比较特殊的用法，如果名称前加了“namespace”关键字，则表示使用当前脚本的命名空间，不使用 use 导入的，这种用法实际上与使用非限定名称的效果一样，比如：

```
//a.php
namespace aa;
const MY_CONST = 1000;

//b.php
namespace bb;
include 'a.php';
use aa;
```

```
const MY_CONST = 2000;
echo namespace\MY_CONST; //将输出 2000，而不是 1000
```

这种情况下 MY_CONST 指明使用当前 namespace 下的，而不是 aa 下的，与下面的用法效果相同：

```
//b.php
namespace bb;
include 'a.php';
use aa;

const MY_CONST = 2000;
echo MY_CONST; //将输出 2000，而不是 1000
```

接下来看一下内核的具体实现，首先是 use 语句的编译，该语句的主要作用是把 use 导入的名称以别名或最后分节为 key 的形式存储到对应的哈希表中，比较特殊的是，常量区分大小写，哈希表中的 value 直接是原样字符串，其他的则转为小写。

```
void zend_compile_use(zend_ast *ast)
{
    zend_string *current_ns = FC(current_namespace);
    //use 的类型
    uint32_t type = ast->attr;
    //根据类型获取存储哈希表：FC(imports)、FC(imports_function)、FC(imports_const)
    HashTable *current_import = zend_get_import_ht(type);
    ...
    //use 可以同时导入多个
    for (i = 0; i < list->children; ++i) {
        zend_ast *use_ast = list->child[i];
        zend_ast *old_name_ast = use_ast->child[0];
        zend_ast *new_name_ast = use_ast->child[1];
        //old_name 为 use 后的 namespace 名称，new_name 为 as 定义的别名
        zend_string *old_name = zend_ast_get_str(old_name_ast);
        zend_string *new_name, *lookup_name;

        if (new_name_ast) {
            //如果有 as 别名则直接使用
            new_name = zend_string_copy(zend_ast_get_str(new_name_ast));
```

```
        } else {
            const char *unqualified_name;
            size_t unqualified_name_len;
            if (zend_get_unqualified_name(old_name, &unqualified_name,
&unqualified_name_len)) {
                //按"\"分割，取最后一节为 new_name
                new_name = zend_string_init(unqualified_name, unqualified_
name_len, 0);
            } else {
                //名称中没有"\": use aa
                new_name = zend_string_copy(old_name);
            }
        }
        //如果是 use const 则大小写敏感，其他用法都转为小写
        if (case_sensitive) {
            lookup_name = zend_string_copy(new_name);
        } else {
            lookup_name = zend_string_tolower(new_name);
        }
        ...
        if (current_ns) {
            //如果当前是在命名空间中则需要检查名称是否冲突
            ...
        }
        //插入 FC(imports/imports_function/imports_const),key 为 lookup_name,
value 为 old_name
        if (!zend_hash_add_ptr(current_import, lookup_name, old_name)) {
            ...
        }
    }
}
```

以下几种不同类型的命名空间导入后的映射表如图 8-3 所示。

```
use aa\bb;                          //导入 namespace
use aa\bb\MY_CLASS;                 //导入类
use function aa\bb\my_func;         //导入函数
use const aa\bb\MY_CONST;           //导入常量
```

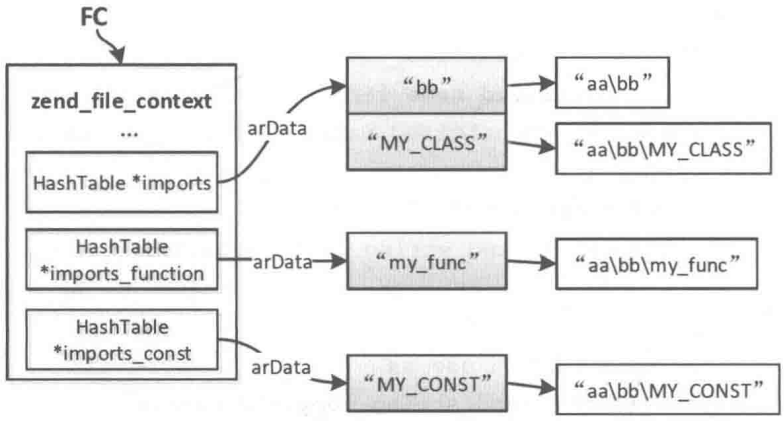


图 8-3 use 命名空间映射表

接下来我们看一下在编译使用类、函数、常量的语句时是如何处理的，使用的语法类型比较多，比如类的使用就有 new、访问静态属性、调用静态方法等，但是不管什么语句都会经历获取类名、函数名、常量名这一步。

1. 类名补全

类名在编译时通过 zend_resolve_class_name()方法进行类名补全，比如实例化类的语句：

```
$a = new my_class;
```

new 语句在语法分析阶段被编译为 ZEND_AST_NEW 节点，然后由 zend_compile_new()函数编译生成 opline，编译的第一步就是获取类名，这个时候就会调用 zend_resolve_class_name()尝试补全类名。

```
void zend_compile_new(znode *result, zend_ast *ast)
{
    ...
    znode class_node, ctor_result;
    ...
    if (zend_is_const_default_class_ref(class_ast)) {
        class_node.op_type = IS_CONST;
        //补全类名，zend_resolve_class_name_ast()将调用 zend_resolve_class_
name()处理
        ZVAL_STR(&class_node.u.constant,
zend_resolve_class_name_ast(class_ast));
    }
```

```
    ...
}
```

不同类型的名称的补全处理规则是不一样的：如果类名没有加任何的命名空间（即非限定名称），则会根据类名查找 FC(imports)，如果找到了则将“命名空间 + 类名”拼成完整类名；如果类名包含部分命名空间（即部分限定名称），则将类名按"\"分割，根据第一节的名称查找 FC(imports)，如果找到了则在类名前拼上完整的命名空间；如果在 FC(imports)中没有找到，或者使用了 namespace 关键字标识，则根据类名所在脚本是否定义了命名空间来决定是否补全。接下来具体看一下不同类型名称的补全规则。

1）使用 namespace 关键字

如果类名为这种规则："namespace\xxx\类名"，表示使用当前命名空间。zend_prefix_with_ns() 处理如下：

```
//zend_resolve_class_name:
if (type == ZEND_NAME_RELATIVE) {
    return zend_prefix_with_ns(name);
}

zend_string *zend_prefix_with_ns(zend_string *name) {
    if (FC(current_namespace)) {
        //当前脚本定义了 namespace
        zend_string *ns = FC(current_namespace);
        return zend_concat_names(ZSTR_VAL(ns), ZSTR_LEN(ns), ZSTR_VAL(name),
ZSTR_LEN(name));
    } else {
        //原始名称，不补全
        return zend_string_copy(name);
    }
}
```

2）完全限定名称

如果使用的是完全限定名称，即实际的类名，则不需要补全：

```
//zend_resolve_class_name:
if (type == ZEND_NAME_FQ || ZSTR_VAL(name)[0] == '\\') {
    if (ZSTR_VAL(name)[0] == '\\') {
        name = zend_string_init(ZSTR_VAL(name) + 1, ZSTR_LEN(name) - 1, 0);
```

```
    } else {
        zend_string_addref(name);
    }
    ...
    return name;
}
```

这里判断是否为完全限定名称采用了两个判断条件：type == ZEND_NAME_FQ、ZSTR_VAL(name)[0] == '\'，这两种有什么区别呢？实际这两个都是指以"\"开头的名称，只是名称的形式不一样导致的两种判断。ZEND_NAME_FQ 这个标识用于在语法分析时，如果发现名称以"\"开头，则会将该节点标上这个标识符，而 ZSTR_VAL(name)[0] == '\'这个判断用于那些名称为字符串的情况，例如：

```
//在语法分析时确定为 ZEND_NAME_FQ
$obj = new \aa\my_class();
//在抽象语法树时根据字符判断
$obj = new "\aa\my_class()";
```

3）部分限定名称

如果当前脚本使用 use 导入了命名空间，且使用的是部分限定名称，则会按照普通用法的规则进行补全，也就是以类名以"\"分割的第一节为 key 检索 FC(imports)，如果有这个 key，再将 value 取出，然后拼上类名，组成完整的类名。

```
//zend_resolve_class_name:
//如果当前脚本有通过 use 导入 namespace
if (FC(imports)) {
    compound = memchr(ZSTR_VAL(name), '\\', ZSTR_LEN(name));
    if (compound) {
        // 1) 部分限定名称
        //名称中包括"\", 比如 new aa\bb\my_class()
        size_t len = compound - ZSTR_VAL(name);
        //根据按"\"分割后的最后一节为 key 查找 FC(imports)
        zend_string *import_name =
            zend_hash_find_ptr_lc(FC(imports), ZSTR_VAL(name), len);
        //如果找到了表示通过 use 导入了 namespace
        if (import_name) {
            //将完整的命名空间拼上类名，组成实际的类名
```

```
            return zend_concat_names(
                ZSTR_VAL(import_name), ZSTR_LEN(import_name), ZSTR_VAL(name)
+ len + 1, ZSTR_LEN(name) - len - 1);
        }
    } else {
        ...
    }
}
```

4）非限定名称

如果使用的非限定名称，也就是普通的类名，这个名称会直接以类名为 key 查找 FC(imports)，检查是否有直接导入的类 ，如果有则拼上命名空间，组成完整类名，部分限定名称的处理是类似的，只是不需要根据"\"分割类名。

```
//zend_resolve_class_name:
//如果当前脚本有通过 use 导入 namespace
if (FC(imports)) {
    compound = memchr(ZSTR_VAL(name), '\\', ZSTR_LEN(name));
    if (compound) {
        ...
    } else {
        //直接根据类名查找
        zend_string *import_name
            = zend_hash_find_ptr_lc(FC(imports), ZSTR_VAL(name), ZSTR_LEN(name));

        if (import_name) {
            return zend_string_copy(import_name);
        }
    }
}
```

2. 函数名补全

函数名在编译时由 zend_resolve_function_name()方法进行补全，比如调用函数的语句：

```
$ret = my_func();
```

调用函数由 zend_compile_call()进行编译，在编译函数体的语句之前，首先就是编译函数名称：

```
void zend_compile_call(znode *result, zend_ast *ast, uint32_t type)
{
    ...
    zend_bool runtime_resolution = zend_compile_function_name(&name_node,
name_ast);
    ...
}
```

zend_compile_function_name()进一步调用 zend_resolve_function_name()获取函数名称。函数名称补全时优先用原始名称去 FC(imports_function)查找，如果没有找到再去 FC(imports)中匹配，按非限定名称的逻辑补全。

```
zend_string *zend_resolve_function_name(zend_string *name, uint32_t type,
zend_bool *is_fully_qualified)
{
    return zend_resolve_non_class_name(
        name, type, is_fully_qualified, 0, FC(imports_function));
}
```

1）使用 namespace 关键字

如果函数名称前加了 namespace 关键字，则表示这是一个相对名称，与类名的处理一致：

```
//zend_resolve_non_class_name:
if (type == ZEND_NAME_RELATIVE) {
    *is_fully_qualified = 1;
    return zend_prefix_with_ns(name);
}
```

2）完全限定名称

完全限定名称不需要进行补全，直接使用脚本中的原始名称：

```
//zend_resolve_non_class_name:
if (ZSTR_VAL(name)[0] == '\\') {
    /* Remove \ prefix (only relevant if this is a string rather than a label) */
    *is_fully_qualified = 1;
    return zend_string_init(ZSTR_VAL(name) + 1, ZSTR_LEN(name) - 1, 0);
}
```

```
    if (type == ZEND_NAME_FQ) {
        *is_fully_qualified = 1;
        return zend_string_copy(name);
    }
```

3）非限定名称

如果是非限定名称，也就是只有函数名称，则首先会查找 FC(imports_function)，由于 FC(imports_function)中保存的是完整的“命名空间+函数名称”，所以如果找到了对应的 key 将直接返回找到的 value：

```
    //zend_resolve_non_class_name:
    //current_import_sub 为传入的 FC(imports_function)
    if (current_import_sub) {
        zend_string *import_name;
        ...
        //以小写名称查找
        import_name = zend_hash_find_ptr_lc(current_import_sub, ZSTR_VAL(name),
ZSTR_LEN(name));
        //返回找到的名称
        if (import_name) {
            *is_fully_qualified = 1;
            return zend_string_copy(import_name);
        }
    }
```

4）部分限定名称

如果在 FC(imports_function)中未找到对应的 key，且为部分限定名称，则会向 FC(imports)中查找，也就是按照普通的 use 导入命名空间的规则进行处理：

```
    compound = memchr(ZSTR_VAL(name), '\\', ZSTR_LEN(name));
    if (compound) {
        *is_fully_qualified = 1;
    }

    if (compound && FC(imports)) {
        size_t len = compound - ZSTR_VAL(name);
        //以“\”分割的第一节为 key 查找 FC(imports)
        zend_string *import_name = zend_hash_find_ptr_lc(FC(imports), ZSTR_
VAL(name), len);
```

```
        if (import_name) {
            return zend_concat_names(
                ZSTR_VAL(import_name), ZSTR_LEN(import_name), ZSTR_VAL(name) +
len + 1, ZSTR_LEN(name) - len - 1);
        }
    }

    return zend_prefix_with_ns(name);
```

3. 常量名补全

常量名的补全逻辑与函数名的几乎完全一样，只是非限定名称时，常量名查找的是FC(imports_const)，且大小写敏感，具体逻辑不再展开。

```
    zend_string *zend_resolve_const_name(zend_string *name, uint32_t type,
zend_bool *is_fully_qualified)
        return zend_resolve_non_class_name(
            name, type, is_fully_qualified, 1, FC(imports_const));
    }
```

8.3.2 动态用法

前面介绍的这些命名空间的使用都是名称为 CONST 类型的情况，所有的处理都是在编译环节完成的，PHP 是动态语言，能否动态使用命名空间呢？举个例子：

```
$class_name = "\aa\bb\my_class";
$obj = new $class_name;
```

类似这样的用法只能用完全限定名称，也就是按照实际存储的名称使用，无法进行自动名称补全。

8.4 小结

本章主要介绍了 PHP 中命名空间的相关实现，命名空间的处理全部发生在编译阶段，并不涉及运行时的逻辑。从命名空间的实现来看，它主要是在类、函数、常量的名称上进行了特殊处理，除此之外没有任何特殊逻辑。

第 9 章
PHP 基础语法的实现

前面几章我们先后了解了 PHP 中变量的基本实现、ZendVM 的编译与执行、函数以及面向对象的实现，这些都是比较大的一些模块。本章我们将从 PHP 语法的角度，具体介绍 PHP 部分语言特性的实现，主要包括以下几个部分的内容：

- 静态变量；
- 常量；
- 全局变量；
- 条件分支——包括 if…else、switch 的实现；
- 循环结构——包括 for、while、do while、foreach 的实现；
- 中断及跳转；
- include/require；
- 异常处理，即 PHP 中 try catch 的实现。

这些语法是 PHP 语言提供的基础功能，是 PHP 中使用最频繁的语法，了解这些基础语法的实现可以帮助我们更好地理解 PHP，同时也希望大家能够从这些已有语法的实现中得到一些启发，能够自己实现一些 PHP 的新 Feature。本章将更多地采用调试追踪的方式带领大家一步步弄清楚各个语法的实现。

9.1 静态变量

静态变量是一种特殊的变量，不管是何种语言，变量的生命周期都是有限的。以 C 语言为例，变量是指分配在函数内部的一种数据，它存储在栈上，其生命周期从函数执行之初开始，至函数返回之时而终，函数执行完以后局部变量就从栈上释放了，下次执行再重新分配、初始化。也就是说，局部变量的值不会保留到下次调用，这是变量的基本属性。PHP 中的变量也具有这个属性，虽然它的变量并没有像 C 语言那样直接分配在栈上，但是 Zend 虚拟机在实际的计算机之上模拟、抽象了一个执行栈：PHP 代码执行时由 Zend 虚拟机分配一块内存，用于记录执行位置、分配局部变量及调用上下文等，每次函数调用都会重新分配。因此，从 Zend 的角度看，PHP 变量的实现与 C 的并没有差别，只不过 Zend 是更高层次的虚拟机，并不是实际的计算机。

静态变量的特殊之处在于：函数调用返回之时并不会释放，它的结果会被保留到下次函数的调用，其生命周期要比局部变量更长。例如：

```
function my_func(){
    static $count = 4;
    $count++;
    echo $count,"\n";
}
my_func();
my_func();
```

在 C 语言中静态变量被分配在静态存储区，在程序启动时分配，直到程序退出时才会释放。在 PHP 中，静态变量并不会像普通变量一样分配在 zend_execute_data 上，而是保存在 zend_op_array->static_variables 中，这是一个哈希结构。静态变量只会初始化一次，而且它的初始化发生在编译阶段而不是执行阶段，上面这个例子中：`static $count = 4`，是在编译阶段发现定义了一个静态变量，然后插进了 zend_op_array->static_variables 中，而不是执行的时候才插入 zend_op_array->static_variables 哈希表，所以这个例子执行的时候会输出 5、6，再次调用 my_func()并没有重置静态变量的值。这个特性也意味着静态变量初始的值不能是变量，比如 `static $count = $xxx`，这样定义将会报错。与 C 语言中的静态变量类似，zend_op_array->static_variables 在请求结束的时候才被释放。

明确了 PHP 静态变量的存储方式，我们再来思考静态变量的访问方式。根据前面章节介绍的内容我们知道，局部变量通过编译时分配的编号进行读写操作，也就是通过 zend_execute_data + offset 获取到对应的变量，而静态变量通过哈希表保存，这就使得其不能像普通变量那样有一个固定的编号。既然保存在哈希表中，很容易想到通过变量名索引的方式进行访问，那么是否如此呢？接下来我们到静态变量的编译、执行过程中寻找答案。

首先看一下静态变量在编译时的处理，语法规则如下：

```
statement:
    ...
    |   T_STATIC static_var_list ';'    { $$ = $2; }
    ...
;

static_var_list:
        static_var_list ',' static_var { $$ = zend_ast_list_add($1, $3); }
    |   static_var { $$ = zend_ast_create_list(1, ZEND_AST_STMT_LIST, $1); }
;

static_var:
        T_VARIABLE    { $$ = zend_ast_create(ZEND_AST_STATIC, $1, NULL); }
    |   T_VARIABLE '=' expr { $$ = zend_ast_create(ZEND_AST_STATIC, $1, $3); }
;
```

从静态变量的语法规则可以看出，静态变量可以同时声明多个，比如 `static $a、$b、$c`，编译时会创建一个 ZEND_AST_STMT_LIST 节点，每个子节点代表一个静态变量，上面的示例只定义了一个，因此该 list 节点只有一个子节点，静态变量的节点类型为 ZEND_AST_STATIC，它有两个子节点，分别用于静态变量名称、变量初始值。最终 my_func()函数生成的抽象语法树如图 9-1 所示。

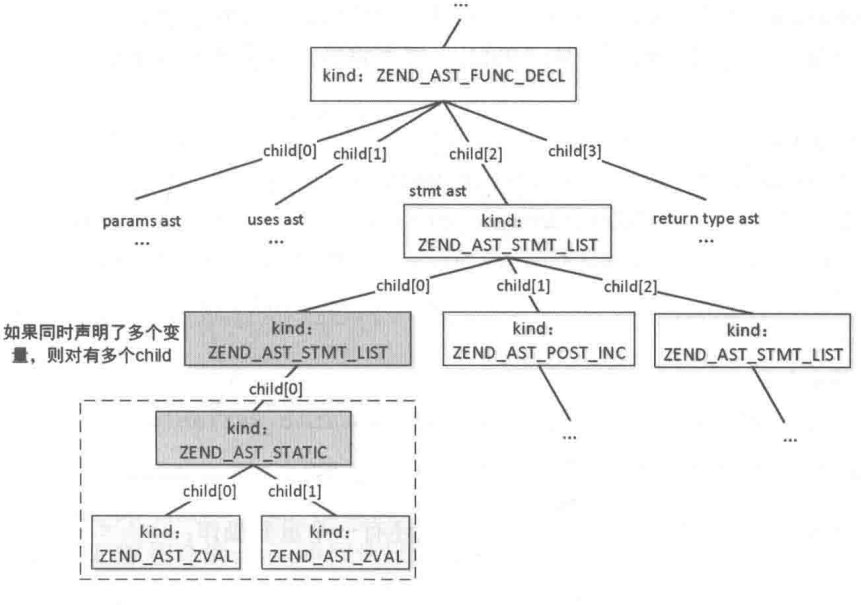


图 9-1　static 语法抽象语法树

ZEND_AST_STATIC 节点由 zend_compile_static_var()方法编译，下面看一下具体的编译过程：

```
void zend_compile_static_var(zend_ast *ast)
{
    zend_ast *var_ast = ast->child[0];
    zend_ast *value_ast = ast->child[1];
    zval value_zv;

    if (value_ast) {
        //定义了初始值
        zend_const_expr_to_zval(&value_zv, value_ast);
    } else {
        //无初始值
        ZVAL_NULL(&value_zv);
    }

    zend_compile_static_var_common(var_ast, &value_zv, 1);
}
```

这里首先对初始化值进行编译，最终得到一个固定值，然后调用 zend_compile_static_var_common()处理。在这个函数中首先判断当前编译的 zend_op_array->static_variables 是否已创建，如果未创建则分配一个 HashTable，接着将静态变量插入这个哈希表：

```
//zend_compile_static_var_common():
if (!CG(active_op_array)->static_variables) {
    ALLOC_HASHTABLE(CG(active_op_array)->static_variables);
    zend_hash_init(CG(active_op_array)->static_variables, 8, NULL, ZVAL_
PTR_DTOR, 0);
}
//插入静态变量
zend_hash_update(CG(active_op_array)->static_variables,
Z_STR(var_node.u.constant), value);
```

插入静态变量哈希表后并没有结束，接下来还有一个重要操作：

```
//生成一条 ZEND_FETCH_W 的 opcode
```

```
    opline = zend_emit_op(&result, by_ref ? ZEND_FETCH_W : ZEND_FETCH_R,
&var_node, NULL);
    opline->extended_value = ZEND_FETCH_STATIC;

    if (by_ref) {
        zend_ast *fetch_ast = zend_ast_create(ZEND_AST_VAR, var_ast);
        //生成一条 ZEND_ASSIGN_REF 的 opcode
        zend_emit_assign_ref_znode(fetch_ast, &result);
    } else {
        ...
    }
```

这个地方生成了两条 opcode：ZEND_FETCH_W、ZEND_ASSIGN_REF，这两条 opcode 是做什么用的呢？接下来我们再去看一下这两条 opcode 的 handler 函数的处理逻辑。

1）ZEND_FETCH_W

这条指令的具体信息如下：

```
    op1_type = IS_CONST     op1.var = 0
    op2_type = IS_UNUSED    op2.var = 0
    result_type = IS_VAR    result.var = 112
    extended_value = ZEND_FETCH_STATIC
```

从指令的操作数可以看到，它用到了 op1、result，具体处理 handler 为 ZEND_FETCH_W_SPEC_CONST_UNUSED_HANDLER()，该 handler 直接调用了 zend_fetch_var_address_helper_SPEC_CONST_UNUSED()进行处理：

```
    static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_FETCH_W_SPEC_CONST_
UNUSED_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
    {
    ZEND_VM_TAIL_CALL(zend_fetch_var_address_helper_SPEC_CONST_UNUSED
    (BP_VAR_W ZEND_OPCODE_HANDLER_ARGS_PASSTHRU_CC));
    }
```

zend_fetch_var_address_helper_SPEC_CONST_UNUSED()的处理逻辑是：首先根据静态变量名“count”从 zend_op_array->static_variables 哈希表中查找到 zval，接着将这个 zval 的地址保存到 ZEND_FETCH_W 指令的 result 返回值操作数中。从这个过程我们知道了 ZEND_FETCH_W 指令的用途：从 static_variables 符号表中根据静态变量名称获取静态变量值，取到的地址被保

存到了 result 指定的存储位置。具体处理如下：

```
    static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL zend_fetch_var_address_
helper_SPEC_CONST_UNUSED(int type ZEND_OPCODE_HANDLER_ARGS_DC)
    {
        zval *varname;
        zval *retval;
        zend_string *name;
        HashTable *target_symbol_table;

        SAVE_OPLINE();
        //从操作数获取静态变量名称
        varname = EX_CONSTANT(opline->op1);
        ...
        //根据 extended_value 获取对应符号表，静态变量获取的就是 execute_data->
static_variables
        target_symbol_table = zend_get_target_symbol_table(execute_data,
opline->extended_value & ZEND_FETCH_TYPE_MASK);
        //根据变量名获取静态变量的 zval
        retval = zend_hash_find(target_symbol_table, name);
        ...
    fetch_var_return:
        ZEND_ASSERT(retval != NULL);
        if (type == BP_VAR_R || type == BP_VAR_IS) {
            ...
        } else {
            //将 retval 保存到 EX_VAR(opline->result.var
            ZVAL_INDIRECT(EX_VAR(opline->result.var), retval);
        }
        ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION();
    }
```

retval 就是静态变量 count 的 zval 地址，最后把这个地址存到了 result 操作数表示的位置，这里并不是将 retval 的值复制给了 result，而是将 result 指向了 retval，具体的可以看一下 ZVAL_INDIRECT()宏的处理。该指令执行完成后，各变量的内存关系如图 9-2 所示。

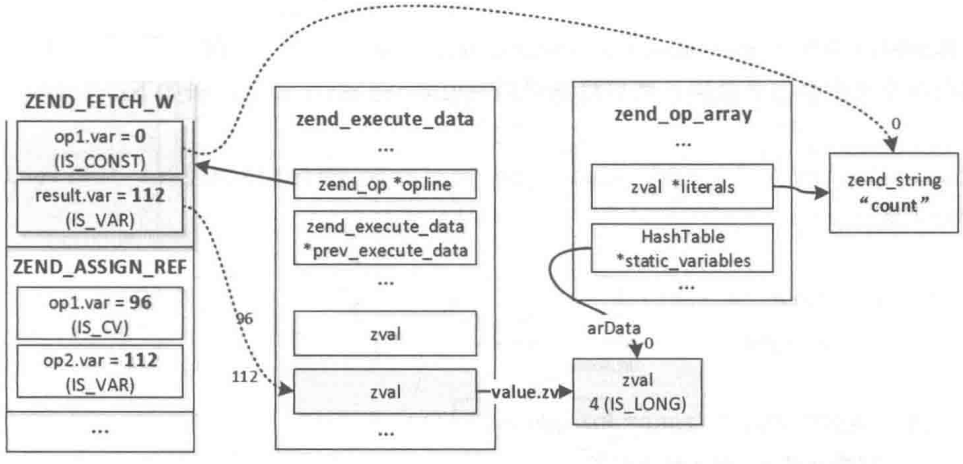


图 9-2　ZEND_FETCH_W 指令的处理

2）ZEND_ASSIGN_REF

其实从图 9-2 就可以猜出 ZEND_ASSIGN_REF 的作用了，这是一条引用赋值指令，该指令负责将 ZEND_FETCH_W 指令获取到的静态变量赋值给局部变量$count，只不过这条指令会将静态变量转为引用类型，然后再赋给$count。因为是引用，所以修改$count 时会直接修改 zend_op_array->static_variables 中对应的静态变量值，这就是为什么要转为引用的原因。具体的处理如下：

```
    static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_ASSIGN_REF_SPEC_CV_VAR_
HANDLER(ZEND_OPCODE_HANDLER_ARGS)
    {
        ...
        zval *variable_ptr;
        zval *value_ptr;
        //获取变量值
        value_ptr = _get_zval_ptr_ptr_var(opline->op2.var, execute_data,
&free_op2);
        //获取要赋值的变量
        variable_ptr = _get_zval_ptr_cv_undef_BP_VAR_W(execute_data, opline->
op1.var);
        ...
            //赋值
            zend_assign_to_variable_reference(variable_ptr, value_ptr);
        ...
        ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION();
    }
```

具体的赋值操作在 zend_assign_to_variable_reference()在中完成，这里将会新创建一个引用，该引用指向原来的静态变量值，然后将该引用赋值给$count，具体逻辑如下：

```
    static inline void zend_assign_to_variable_reference(zval *variable_ptr,
zval *value_ptr)
    {
        zend_reference *ref;
        zval garbage;

        if (EXPECTED(!Z_ISREF_P(value_ptr))) {
            //将 value_ptr 转为引用
            ZVAL_NEW_REF(value_ptr, value_ptr);
        } else if (UNEXPECTED(variable_ptr == value_ptr)) {
            return;
        }

        ref = Z_REF_P(value_ptr);
        GC_REFCOUNT(ref)++;
        //如果 variable_ptr 之前赋了其他值，则将旧值保存到 garbage
        ZVAL_COPY_VALUE(&garbage, variable_ptr);
        //将引用赋值给 variable_ptr
        ZVAL_REF(variable_ptr, ref);
        //断开 variable_ptr 对旧 value 的引用
        zval_ptr_dtor(&garbage);
    }
```

该指令完成后，$count 通过引用指向了静态变量符号表中的值，再次调用函数时指向的还是同一个值，所以静态变量的值可以保留到下次调用，如图 9-3 所示。通过上面两条 opcode 可以确定静态变量的读写过程：首先根据变量名在 static_variables 中取出对应的 zval，然后将它修改为引用类型并赋值给局部变量。也就是说，static $count = 4 包含了两个操作，实际的赋值为$count = & zend_op_array->static_variables["count"]，所以说$count 并不是真正的静态变量，它只是一个指向静态变量的局部变量。

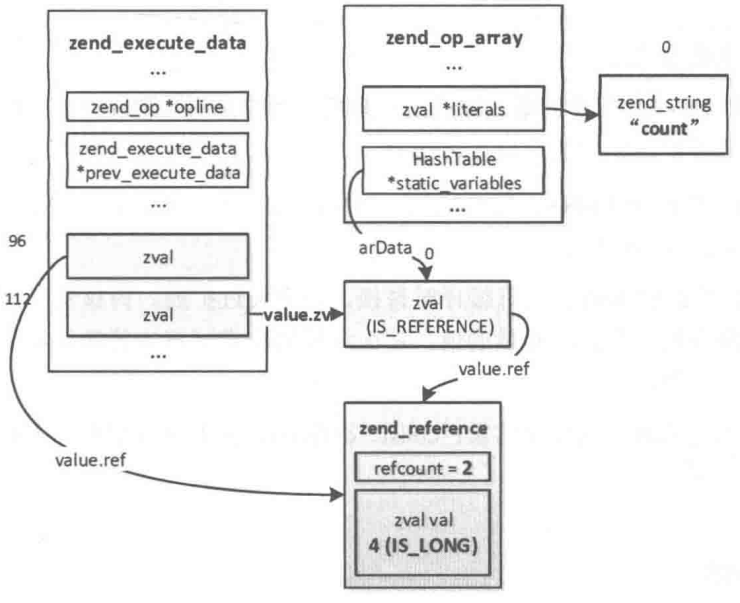


图 9-3　ZEND_ASSIGN_REF 指令的处理

9.2　常量

常量是一个简单值的标识符(名字),如同其名称所暗示的,在脚本执行期间该值不能改变。常量默认为大小写敏感，通常常量标识符总是大写的。常量名和其他任何 PHP 标签遵循同样的命名规则，合法的常量名以字母或下画线开始，后面跟着任何字母、数字或下画线。

在内核中，常量存储在 EG(zend_constants)哈希表中，访问时也是根据常量名直接到哈希表中查找，实现比较简单。常量的数据结构：

```
typedef struct _zend_constant {
    zval value;   //常量值
    zend_string *name; //常量名
    int flags;  //常量标识位
    int module_number; //所属扩展、模块
} zend_constant;
```

常量的几个属性都比较直观，这里只介绍 flags，它的值可以是以下三个中任意的组合：

```
#define CONST_CS              (1<<0)
#define CONST_PERSISTENT       (1<<1)
#define CONST_CT_SUBST         (1<<2)
```

三种 flag 代表的含义。

- **CONST_CS**：大小写敏感，默认是开启的，用户通过 define()定义的始终是区分大小写的，通过扩展定义的可以自由选择。
- **CONST_PERSISTENT**：持久化的，只有通过扩展、内核定义的才支持，这种常量不会在 request 结束时清理掉。
- **CONST_CT_SUBST**：允许编译时替换，只有通过扩展、内核定义的才支持，编译时如果发现有地方在读取常量的值，那么编译器会尝试直接替换为常量值，而不是在执行时再去读取。

PHP 中可以通过两种方式定义常量：const、define()，接下来详细看一下这两种不同方式之间的差异。

9.2.1 const

const 是一个关键字，后面直接加常量名，可以通过下面的语法声明常量：

```
//声明一个常量
const AA = 100;
//声明多个常量，通过逗号隔开
const AA = 100, BB = 200;
```

const 的语法规则：

```
//file: zend_language_parser.y
top_segment:
    ...
    |   T_CONST const_list ';'                    { $$ = $2; }
;
const_list:
        const_list ',' const_decl { $$ = zend_ast_list_add($1, $3); }
    |   const_decl { $$ = zend_ast_create_list(1, ZEND_AST_CONST_DECL, $1); }
;
const_decl:
    T_STRING '=' expr { $$ = zend_ast_create(ZEND_AST_CONST_ELEM, $1, $3); }
;
```

从语法规则可以看出，编译时每一个常量声明生成一个 ZEND_AST_CONST_ELEM 节点，它有两个子节点，分别表示常量名、常量值，该节点会被插入到 ZEND_AST_CONST_DECL 节点中，这是一个 list 节点，编译时会遍历该 list，依次编译每一个 ZEND_AST_CONST_ELEM 节点。具体的编译处理函数为 zend_compile_const_decl()：

```
void zend_compile_const_decl(zend_ast *ast)
{
    zend_ast_list *list = zend_ast_get_list(ast);
    uint32_t i;
    for (i = 0; i < list->children; ++i) {
        zend_ast *const_ast = list->child[i];
        //常量名节点
        zend_ast *name_ast = const_ast->child[0];
        //常量值节点
        zend_ast *value_ast = const_ast->child[1];
        ...
        znode name_node, value_node;
        ...
        //常量值的操作数只能是 CONST
        value_node.op_type = IS_CONST;
        zend_const_expr_to_zval(value_zv, value_ast);
        ...
        zend_emit_op(NULL, ZEND_DECLARE_CONST, &name_node, &value_node);
        ...
    }
}
```

从编译的过程可以看到，常量值的类型只能是 CONST 类型，这就意味着无法通过 const 定义值为变量的常量，比如 `const AA = $a`，将在编译时报错。编译的最后生成了一条 ZEND_DECLARE_CONST 指令，这条指令的主要作用就是注册常量，具体逻辑如下：

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_DECLARE_CONST_SPEC_
CONST_CONST_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    USE_OPLINE

    zval *name;
```

```
    zval *val;
    zend_constant c;

    SAVE_OPLINE();
    name  = EX_CONSTANT(opline->op1);
    val   = EX_CONSTANT(opline->op2);

    ZVAL_COPY_VALUE(&c.value, val);
    ...
    //大小写敏感
    c.flags = CONST_CS; /* non persistent, case sensetive */
    c.name = zend_string_dup(Z_STR_P(name), 0);
    c.module_number = PHP_USER_CONSTANT;
    //注册到EG(zend_constants)
    if (zend_register_constant(&c) == FAILURE) {
    }
    ...
}
```

这个时候一个常量就成功注册完成了，访问常量时也是根据常量名从 EG(zend_constants)中查找。

9.2.2 define()

与 const 不同，define()是一个内部函数，它定义在 Zend/zend_builtin_functions.c 中，通过 define()可以定义值为变量的常量，但是变量的值必须也是标量，比如：

```
$a = arra(1,2);
define('AA', $a);
```

接下来看一下这个函数的实现：

```
ZEND_FUNCTION(define)
{
    zend_string *name;
    zval *val, val_free;
    zend_bool non_cs = 0;
```

```
    int case_sensitive = CONST_CS;
    zend_constant c;
    ...
repeat:
    //针对不同类型的 value 值进行处理
    switch (Z_TYPE_P(val)) {
        ...
    }
    ZVAL_DUP(&c.value, val);
    zval_ptr_dtor(&val_free);
register_constant:
    c.flags = case_sensitive; /* non persistent */
    c.name = zend_string_copy(name);
    c.module_number = PHP_USER_CONSTANT;
    //注册到常量符号表
    if (zend_register_constant(&c) == SUCCESS) {
        RETURN_TRUE;
    } else {
        RETURN_FALSE;
    }
}
```

从 define()的实现可以看到，它支持更多类型的变量值，但是最终被确定为常量值的值必须全部是常量或者资源，也就是说常量值支持：NULL、布尔型、整型、浮点型、字符串、资源，不支持对象。比较特殊的类型的处理：

1）数组

数组元素中不能有非标量成员，比如：

```
$a = new stdClass;
define('AA', array($a));
```

注册时如果常量值为数组，则会遍历数组检查是否含有 NULL、布尔型、整型、浮点型、字符串、资源之外类型的成员，如果数组合法，则将数组深拷贝一份用于注册到 EG(zend_constants)。

```
case IS_ARRAY:
if (!Z_IMMUTABLE_P(val)) {
```

```
        //检查是否含常量不支持的类型成员
        if (!validate_constant_array(Z_ARRVAL_P(val))) {
            RETURN_FALSE;
        } else {
            //这里会进行深拷贝
            copy_constant_array(&c.value, val);
            goto register_constant;
        }}
    break;
```

2）对象

常量值并不支持对象类型，但如果对象的操作 handler 定义了 get、case_object，则会调用该方法，然后将返回值作为常量值，具体应用场景不详。

9.3 全局变量

PHP 中把定义在函数、类之外的变量称之为全局变量，包括 include/require 文件中的，也就是定义在主代码中的变量，这些变量可以在函数、成员方法中通过 global 关键字引入使用。例如下面这个例子中，$id 值在 test()中被修改，修改后的值将直接反映到主代码中的$id。

```
function test() {
    global $name,$id;
    $id++;
}
$name = "php";
$id = 1;
test();
echo $id;
```

9.3.1 全局变量符号表

全局变量与静态变量在实现上有很多相似之处，它们的值都保存在全局符号表中，全局变量保存在 EG(symbol_table)中，这同样是一个哈希表。另外，全局变量的访问也与静态变量的处理方式相同，也是通过局部变量指向全局变量的方式进行访问。不同的地方在于，全局变量并不是通过单独的指令注册到 EG(symbol_table)符号表中，而是在执行前，由内核自动将主代码中的局部变量导入全局变量符号表，即使代码里没有使用全局变量的地方也会导入到该符号

表。导入的过程是在 zend_execute_ex()执行前完成的，再重新看一下 ZendVM 的执行入口：

```
ZEND_API void zend_execute(zend_op_array *op_array, zval *return_value)
{
    ...
    i_init_execute_data(execute_data, op_array, return_value);
    zend_execute_ex(execute_data);
    ...
}
```

i_init_execute_data() 在对 zend_execute_data 初始化的过程中，其中有一步：zend_attach_symbol_table()，该方法就是将局部变量导入全局变量符号表的操作，这个地方将遍历存有局部变量名的 zend_op_array->vars 数组，然后以变量名为 key 插入 EG(symbol_table)，value 指向 zend_execute_data 上对应的局部变量。

```
ZEND_API void zend_attach_symbol_table(zend_execute_data *execute_data)
{
    zend_op_array *op_array = &execute_data->func->op_array;
    HashTable *ht = execute_data->symbol_table;

    if (!EXPECTED(op_array->last_var)) {
        return;
    }

    zend_string **str = op_array->vars;
    zend_string **end = str + op_array->last_var;
    //局部变量数组的起始位置
    zval *var = EX_VAR_NUM(0);

    do{
        zval *zv = zend_hash_find(ht, *str);
        //插入全局变量符号表
        zv = zend_hash_add_new(ht, *str, var);
        //哈希表中 value 指向局部变量的 zval
        ZVAL_INDIRECT(zv, var);
        ...
    }while(str != end);
}
```

上面那个例子完成局部变量的导入后 EG(symbol_table)与 zend_execute_data 的变量关系如

图 9-4 所示，EG(symbol_table)中还有 PHP 自己定义的超全局变量，所以我们脚本中自定义的全局变量在数组中的位置并不是第一个。

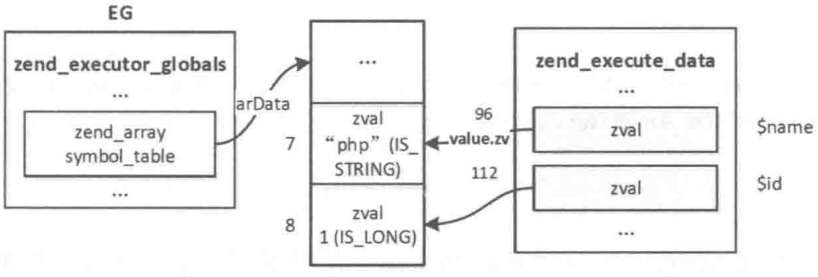


图 9-4 全局变量符号表

9.3.2 全局变量的访问

如果在函数或成员方法中使用全局变量，则需要通过 global 关键字导入，支持同时导入多个，但是不能在导入时赋值，比如 `global $id = 100` 是非法的。全局变量的访问机制与静态变量的实现一致，也是将全局变量转换为引用类型，然后将局部变量指向这个引用。global 语法的编译过程这里不再展开，直接看一下编译生成的 ZEND_BIND_GLOBAL 指令的处理：

```
    static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_BIND_GLOBAL_SPEC_CV_
CONST_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
    {
        zval *value;
        zval *variable_ptr;
        ...
        varname = EX_CONSTANT(opline->op2);
        //根据全局变量名称获取全局变量
        value = zend_hash_find(&EG(symbol_table), Z_STR_P(varname));
        ...
        do {
            zend_reference *ref;

            if (UNEXPECTED(!Z_ISREF_P(value))) {
                //将全局变量转为引用类型，指向原来的 value
                ref = (zend_reference*)emalloc(sizeof(zend_reference));
                GC_REFCOUNT(ref) = 2;
                GC_TYPE_INFO(ref) = IS_REFERENCE;
                ZVAL_COPY_VALUE(&ref->val, value);
```

```
            Z_REF_P(value) = ref;
            Z_TYPE_INFO_P(value) = IS_REFERENCE_EX;
        } else {
            ref = Z_REF_P(value);
            GC_REFCOUNT(ref)++;
        }
        //获取指向全局变量的局部变量
        variable_ptr = _get_zval_ptr_cv_undef_BP_VAR_W(execute_data,
opline->op1.var);
        //如果 variable_ptr 有指向其他值，则释放
        ...
        //将局部变量指向全局变量的引用
        ZVAL_REF(variable_ptr, ref);
    } while (0);
    ...
}
```

经过上面过程的处理后，全局变量符号表中的值就被修改为了引用类型，导入全局变量的函数或成员方法内的局部变量指向了这个引用，所以修改局部变量时会直接修改全局变量的值，具体关系如图 9-5 所示。

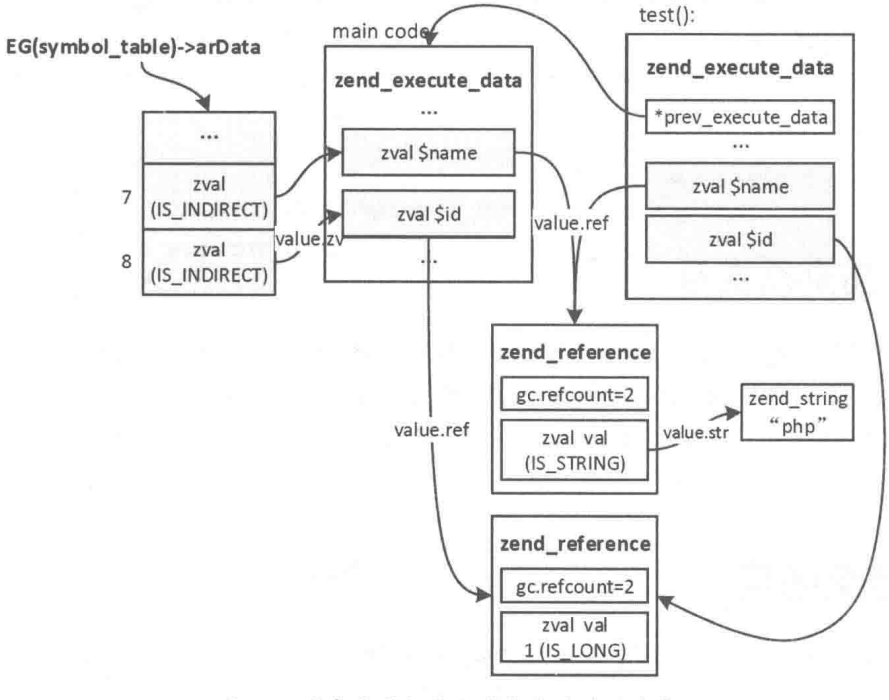


图 9-5　局部变量指向全局变量的引用关系

9.3.3 全局变量的销毁

普通的局部变量如果没有主动销毁，那么在函数结束时会将它们销毁，而全局变量则是在整个请求结束时才会被销毁，换句话说，定义在主代码中的局部变量也是在请求结束时才被释放的，这与函数、成员方法中的局部变量的处理方式不同。销毁过程在 php_request_shutdown() 中触发，具体在函数 shutdown_destructors() 中处理：遍历 EG(symbol_table)，调用 zval_call_destructor()释放各个成员。

```
void shutdown_destructors(void)
{
    if (CG(unclean_shutdown)) {
        EG(symbol_table).pDestructor = zend_unclean_zval_ptr_dtor;
    }
    zend_try {
        uint32_t symbols;
        do {
            symbols = zend_hash_num_elements(&EG(symbol_table));
            //销毁
            zend_hash_reverse_apply(&EG(symbol_table), (apply_func_t) zval_
call_destructor);
        } while (symbols != zend_hash_num_elements(&EG(symbol_table)));
    }
    ...
}
```

9.3.4 超全局变量

全局变量除了通过 global 引入还有一类特殊的类型，它们不需要使用 global 引入而可以直接使用，这些全局变量称为超全局变量。超全局变量实际是 PHP 内部定义的一些全局变量：$GLOBALS、$_SERVER、$_REQUEST、$_POST、$_GET、$_FILES、$_ENV、$_COOKIE、$_SESSION、$argv、$argc。

9.4 分支结构

程序不可能总是顺序执行的，顺序结构的程序虽然能解决计算、输出等问题，但不能做判断再选择，对于要先做判断再选择的问题就要使用分支结构。分支结构用于控制程序在不同条

件下，选择执行不同的分支。分支结构主要依赖两条跳转指令实现：一条用于根据条件真假跳转到对应的分支语句处、一条用于分支语句执行完后跳出分支，明确了这一点再来分析分支结构的实现就比较简单了。

PHP 中通过 if、elseif、else 和 switch 语句实现分支结构，这一节我们就来分析 PHP 中两种分支结构的具体实现。

9.4.1　if

if 语句用法：

```
if(Condition1){
    Statement1;
}elseif(Condition2){
    Statement2;
}else{
    Statement3;
}
```

if 语句由两部分组成：condition（条件）、statement（声明），每个条件分支对应一组这样的组合，最后的 else 比较特殊，它没有条件。编译时也是按照这个逻辑，将 if 语句的各个分支编译为一组组的 condition 和 statement。语法规则如下：

```
if_stmt:
        if_stmt_without_else %prec T_NOELSE { $$ = $1; }
    |   if_stmt_without_else T_ELSE statement
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AST_IF_ELEM,
NULL, $3)); }
;

if_stmt_without_else:
        T_IF '(' expr ')' statement { $$ = zend_ast_create_list(1, ZEND_AST_IF,
zend_ast_create(ZEND_AST_IF_ELEM, $3, $5)); }
    |   if_stmt_without_else T_ELSEIF '(' expr ')' statement
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AST_IF_ELEM, $4,
$6)); }
;
```

从上面的语法规则可以看出，编译 if 语句时首先会创建一个 ZEND_AST_IF 的节点，这个节点是一个 list，用于保存各个分支的 condition、statement。编译每个分支时将创建一个 ZEND_AST_IF_ELEM 的节点，它有两个子节点，分别用来记录：condition、statement，然后把这个节点插入到 ZEND_AST_IF 下。也就是说，每一个 if、elseif、else 都被编译为 ZEND_AST_IF_ELEM 节点。if 语句编译完成后，其语法树类似图 9-6 所示。

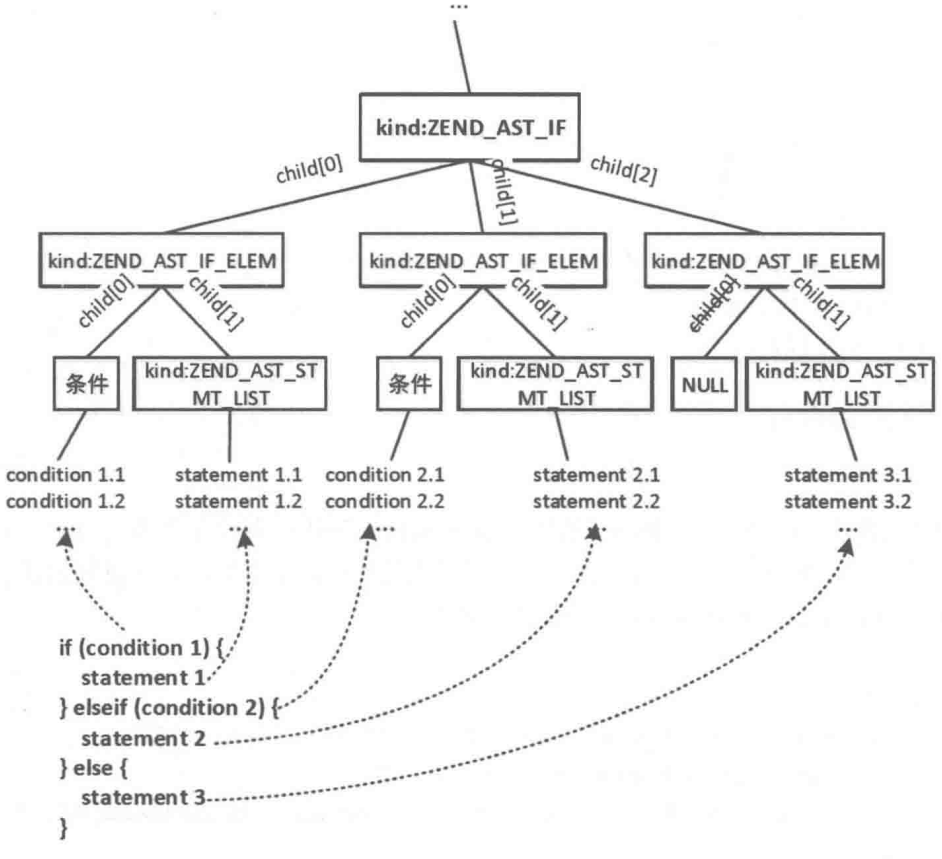


图 9-6　if 语句的抽象语法树

编译 ZEND_AST_IF 节点时，会顺序编译每个分支的 condition、statement。每组 condition 后会编译一条跳转指令，如果 condition 执行最终的结果为假，则该指令将跳到下一个 condition 处继续判断，或者跳出分支结构；statement 编译完之后也会编译一条跳转指令，在分支语句执行结束后，通过这条指令跳出分支。编译过程大致如下所述。

步骤（1）：编译当前分支的 condition 语句，这里可能会有多个条件，但最终会归并为一个 true/false 的结果。

步骤（2）：编译完 condition 后编译一条 ZEND_JMPZ 的指令，如果当前 condition 成立则

直接继续执行本分支 statement，无须进行跳转，如果不成立就需要跳过该分支的 statement，所以这条指令还需要知道该往下跳过多少条指令，而跳过的指令就是本组的 statement，因此这个值需要在编译完本分支 statement 后才能确定，现在还无法确定。编译完该分支的 statement 后，会更新 condition 的 ZEND_JMPZ 指令的跳转值。

步骤（3）：编译当前分支的 statement 列表，其节点类型为 ZEND_AST_STMT_LIST，也就是普通语句的编译。

步骤（4）：编译完 statement 后编译一条 ZEND_JMP 的指令，该指令用于执行完分支语句后跳出分支，ZEND_JMP 需要知道该往下跳过多少指令，而跳过的这些指令是后面所有分支的 conidition、statement 的指令，只有编译完全部分支后才能确定。

步骤（5）：编译完 statement 后再设置步骤（2）中条件不成立时 ZEND_JMPZ 应该跳过的指令数。

步骤（6）：重复上面的过程，依次编译后面的 condition、statement，编译完全部分支后再设置各分支在步骤（4）中 ZEND_JMP 跳出 if 的指令位置。

具体的编译过程在 zend_compile_if()中，过程比较清晰：

```
void zend_compile_if(zend_ast *ast)
{
    zend_ast_list *list = zend_ast_get_list(ast);
    uint32_t i;
    uint32_t *jmp_opnums = NULL;

    //用来保存每个分支在步骤(4)中的 ZEND_JMP
    if (list->children > 1) {
        jmp_opnums = safe_emalloc(sizeof(uint32_t), list->children - 1, 0);
    }
    //依次编译各个条件分支
    for (i = 0; i < list->children; ++i) {
        zend_ast *elem_ast = list->child[i];
        //条件
        zend_ast *cond_ast = elem_ast->child[0];
        //分支声明语句
        zend_ast *stmt_ast = elem_ast->child[1];
```

```
            znode cond_node;
            uint32_t opnum_jmpz;
            if (cond_ast) {
                //1) 编译 condition
                zend_compile_expr(&cond_node, cond_ast);
                //2) 编译 condition 跳转 opcode: ZEND_JMPZ
                opnum_jmpz = zend_emit_cond_jump(ZEND_JMPZ, &cond_node, 0);
            }
            //3) 编译 statement
            zend_compile_stmt(stmt_ast);
            //4) 编译 statement 执行完后跳出 if 的指令: ZEND_JMP(最后一个分支无须这条
opcode)
            if (i != list->children - 1) {
                jmp_opnums[i] = zend_emit_jump(0);
            }
            if (cond_ast) {
                //5) 设置 ZEND_JMPZ 跳过 opcode 数
                zend_update_jump_target_to_next(opnum_jmpz);
            }
        }

        if (list->children > 1) {
            //6) 设置前面各分支 statement 执行完后应跳转的位置
            for (i = 0; i < list->children - 1; ++i) {
                zend_update_jump_target_to_next(jmp_opnums[i]); //设置每组 stmt
最后一条 jmp 跳转为 if 外
            }
            efree(jmp_opnums);
        }
    }
```

if 语句最终编译生成的指令集合如图 9-7 所示。

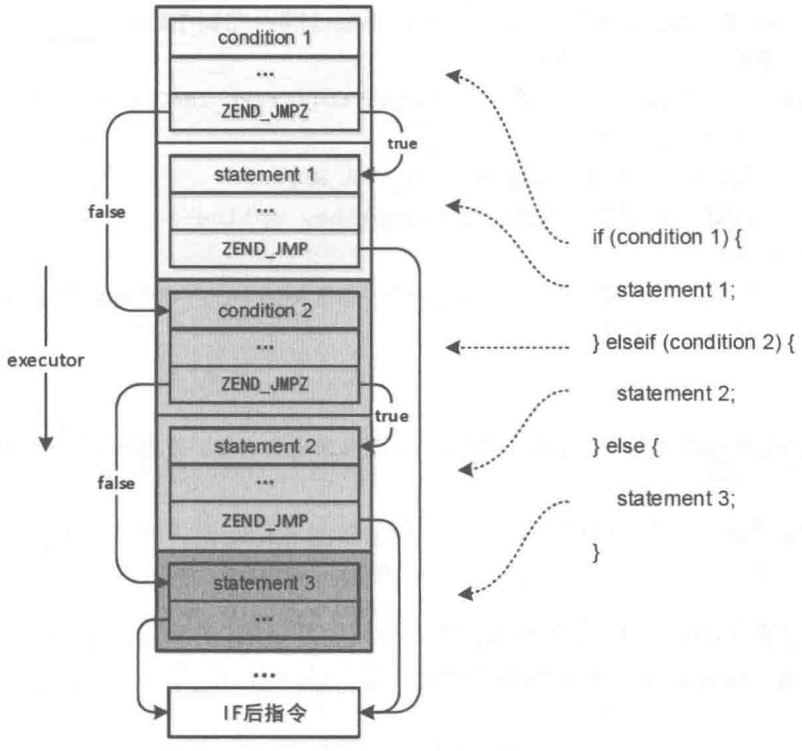


图 9-7　if 分支结构的指令集合

从编译生成的指令集合，可以很直观地看出其执行过程：首先执行第一个分支的条件语句，最后由 ZEND_JMPZ 判断最终的结果，如果为 true，则直接执行下一条指令，也就是第一个条件的分支语句，执行完以后通过 ZEND_JMP 跳到 IF 后指令；如果为 false，则跳到第二个分支的条件语句处，按照同样的逻辑处理。ZEND_JMPZ 指令的处理如下：

```
ZEND_VM_HANDLER(43, ZEND_JMPZ, CONST|TMPVAR|CV, ANY)
{
    USE_OPLINE
    zend_free_op free_op1;
    zval *val;

    val = GET_OP1_ZVAL_PTR_UNDEF(BP_VAR_R);

    if (Z_TYPE_INFO_P(val) == IS_TRUE) {
        //条件为 true：继续执行下一条指令
        ZEND_VM_SET_NEXT_OPCODE(opline + 1);
        ZEND_VM_CONTINUE();
```

```
    } else if (EXPECTED(Z_TYPE_INFO_P(val) <= IS_TRUE)) {
        //条件为 false: 跳过该分支
        if (OP1_TYPE == IS_CV && UNEXPECTED(Z_TYPE_INFO_P(val) == IS_UNDEF)) {
            SAVE_OPLINE();
            GET_OP1_UNDEF_CV(val, BP_VAR_R);
            ZEND_VM_JMP(OP_JMP_ADDR(opline, opline->op2));
        } else {
            ZEND_VM_SET_OPCODE(OP_JMP_ADDR(opline, opline->op2));
            ZEND_VM_CONTINUE();
        }
    }
    //下面是按照弱类型转换的规则对非 bool 型结果进行判断，处理与上面的实际是一样的
    ...
    ZEND_VM_JMP(opline);
}
```

这里需要注意 elseif 与 else if 的差别，上面介绍的是 elseif 的编译，而 else if 则实际上相当于嵌套了一个 if，也就是说一个 if 的分支中包含了另外一个 if，在编译、执行的过程中这两个是有差别的。

```
if (condition 1) {
    ...
} else if(condition 2) {
    ...
}
//等价于:
if (condition 1) {
    ...
} else {
    if (condition 2) {
        ...
    }
}
```

9.4.2 switch

switch 语句与 if 类似，都是条件语句，很多时候需要将一个变量或者表达式与不同的值进

行比较，根据不同的值执行不同的代码，这种场景下用 if、switch 都可以实现，但 switch 相对更加直观。

switch 的语法：

```
switch(expression){
    case value1:
        statement1;
    case value2:
        statement2;
    ...
    default:
        statementn;
}
```

这里并没有将 break 加入到 switch 的语法中，因为 break 并不属于 switch 语句，它属于另外一类单独的语法：中断语法。PHP 中，如果没有在 switch 中加 break，则执行时会从命中的那个 case 开始一直执行到结束。从 switch 的语法可以看出，switch 主要包含两部分：expression、case 列表，case 列表包含多个 case，每个 case 包含 value、statement 两部分。expression 是一个表达式，它在与 case 对比前执行，switch 最终执行时，就是拿 expression 的值逐个与 case 的 value 比较，如果相等则执行命中 case 的 statement。switch 的语法规则：

```
statement:
    ...
    |   T_SWITCH '(' expr ')' switch_case_list { $$ = zend_ast_create(ZEND_
AST_SWITCH, $3, $5); }
    ...
;

switch_case_list:
        '{' case_list '}'                   { $$ = $2; }
    |   '{' ';' case_list '}'                { $$ = $3; }
    |   ':' case_list T_ENDSWITCH ';'         { $$ = $2; }
    |   ':' ';' case_list T_ENDSWITCH ';'     { $$ = $3; }
;

case_list:
        /* empty */ { $$ = zend_ast_create_list(0, ZEND_AST_SWITCH_LIST); }
    |   case_list T_CASE expr case_separator inner_statement_list
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AST_SWITCH_
CASE, $3, $5)); }
```

```
    |   case_list T_DEFAULT case_separator inner_statement_list
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AST_SWITCH_
CASE, NULL, $4)); }
    ;

    case_separator:
            ':'
        |   ';'
    ;
```

从语法解析规则可以看出，switch 最终被解析为一个 ZEND_AST_SWITCH 节点，这个节点主要包含两个子节点：expression、case 列表，其中 expression 节点比较简单，case 列表为 ZEND_AST_SWITCH_LIST 节点，这个节点是一个 list，包含多个 case 子节点，每个 case 节点对应一个 ZEND_AST_SWITCH_CASE 节点，包括 value、statement 两个子节点。switch 语句编译生成的抽象语法树如图 9-8 所示。

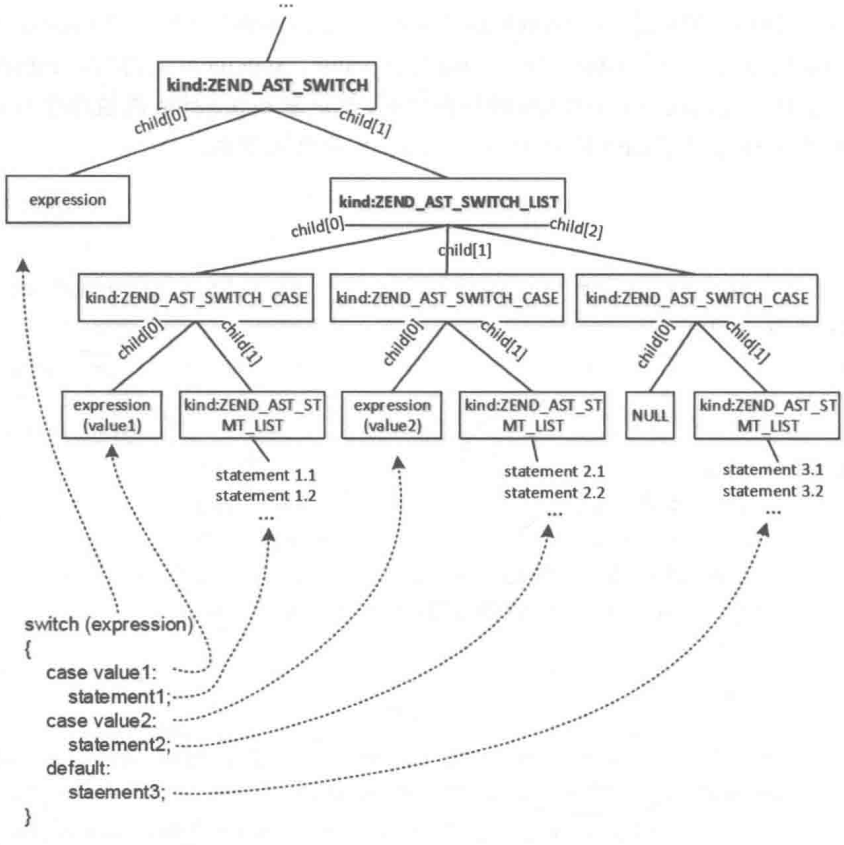


图 9-8 switch 语句的抽象语法树

与 if 不同，switch 不会像 if 那样依次把每个分支编译为一组组的 condition、statement，而是会先编译全部 case 的 value 表达式，再编译全部 case 的 statement，编译过程大致如下所述。

步骤（1）：首先编译 expression，该表达式执行时最终将得到一个 value 值。

步骤（2）：依次编译每个 case，如果 value 是一个表达式则编译 expression，与步骤（1）相同，每个 case 编译生成一条 ZEND_CASE 的指令，该指令用于比较 case 与 expression 的 value 是否相等。除了这条指令，还会编译出一条 ZEND_JMPNZ 的指令，当 ZEND_CASE 指令判断 case 与 expression 的 value 相同时，将通过 ZEND_JMPNZ 指令跳到该 case 的 statement 语句处，但是 statement 在这时还未编译，所以 ZEND_JMPNZ 的跳转值暂不确定。

步骤（3）：为 default 分支编译生成一条 ZEND_JMP 指令。

步骤（4）：编译每个 case 的 statement，编译前首先设置步骤(2)中 ZEND_JMPNZ 的跳转值为当前 statement 起始位置。

具体编译由 zend_compile_switch()完成：

```
void zend_compile_switch(zend_ast *ast)
{
    //switch 表达式节点
    zend_ast *expr_ast = ast->child[0];
    //case 列表节点
    zend_ast_list *cases = zend_ast_get_list(ast->child[1]);
    ...
    uint32_t *jmpnz_opnums, opnum_default_jmp;
    //1) 编译 switch 表达式
    zend_compile_expr(&expr_node, expr_ast);
    ...
    //生成一个 TMPVAR 临时变量，用于 ZEND_CASE 存储比较结果
    case_node.op_type = IS_TMP_VAR;
    case_node.u.op.var = get_temporary_variable(CG(active_op_array));
    //jmpnz_opnums 为每个 case 生成的 ZEND_JMPNZ 指令位置，编译 case 时将指令位置记录在该数组中，等编译 statement 时再更新其跳转值
    jmpnz_opnums = safe_emalloc(sizeof(uint32_t), cases->children, 0);
    //2) 编译各个 case
    for (i = 0; i < cases->children; ++i) {
        zend_ast *case_ast = cases->child[i];
        zend_ast *cond_ast = case_ast->child[0];
```

```
        znode cond_node;
        ...
        //编译 case value 表达式
        zend_compile_expr(&cond_node, cond_ast);
        if (...) { //switch 的 expression 为 true、fase 时，不需要编译 ZEND_CASE
            ...
        } else {
            //生成 ZEND_CASE 指令
            opline = zend_emit_op(NULL, ZEND_CASE, &expr_node, &cond_node);
            SET_NODE(opline->result, &case_node);
            ...
            //生成 ZEND_JMPNZ 指令
            jmpnz_opnums[i] = zend_emit_cond_jump(ZEND_JMPNZ, &case_node, 0);
        }
    }
    //3) 为 default 分支编译一条 ZEND_JMP 指令
    opnum_default_jmp = zend_emit_jump(0);
    //4) 编译各 case 的 statement，并更新各 case ZEND_JMPNZ 指令的跳转值
    for (i = 0; i < cases->children; ++i) {
        ...
        //更新 ZEND_JMPNZ 指令的跳转值
        if (cond_ast) {
            zend_update_jump_target_to_next(jmpnz_opnums[i]);
        } else {
            zend_update_jump_target_to_next(opnum_default_jmp);
        }
        //编译 case 的 statement
        zend_compile_stmt(stmt_ast);
    }
    ...
}
```

最终编译生成的指令集合如图 9-9 所示。

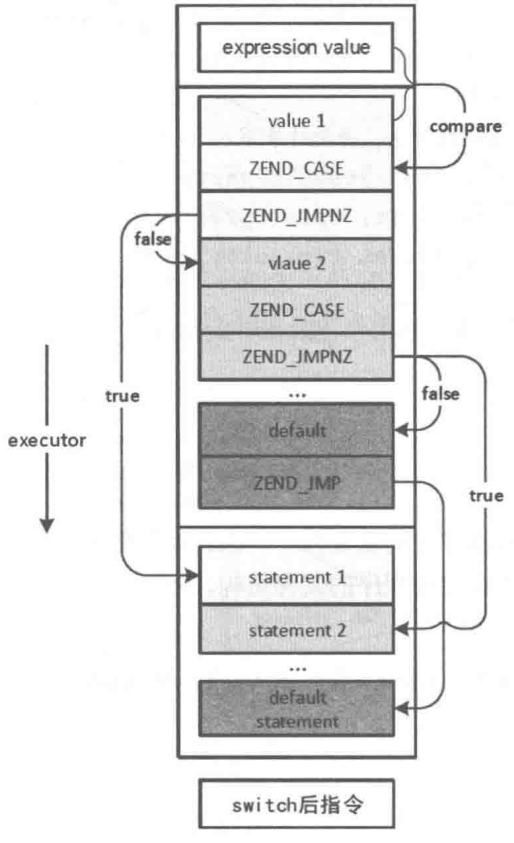


图 9-9　switch 分支结构的指令集合

执行时首先执行 expression 表达式，得到 expr value；然后由 ZEND_CASE 判断 expression 与 case 的 value 是否相等（如果 case 的 value 是表达式，则先执行，用表达式最终的执行结果比较），并把判断结果写入一个变量；接着 ZEND_JMPNZ 指令根据 ZEND_CASE 判断的结果进行处理，如果结果为 true，则表示命中了该 case，ZEND_JMPNZ 将跳转到该 case 的 statement 处执行，如果结果为 false，则表示没有命中该 case，将继续向下执行，接着执行下一个 case 的判断；如果判断一直到了 default，则由 ZEND_JMP 指令跳转到 default 的 statement 处执行。ZEND_JMPNZ 与 if 中的 ZEND_JMPZ 指令类似，这里看一下 ZEND_CASE 的处理：

```
ZEND_VM_HANDLER(48, ZEND_CASE, CONST|TMPVAR|CV, CONST|TMPVAR|CV)
{
    USE_OPLINE
    zend_free_op free_op1, free_op2;
    zval *op1, *op2, *result;
```

```
    op1 = GET_OP1_ZVAL_PTR_UNDEF(BP_VAR_R);
    op2 = GET_OP2_ZVAL_PTR_UNDEF(BP_VAR_R);
    ...
    //比较 op1、op2 指定的变量，并把结果写入 result
    result = EX_VAR(opline->result.var);
    compare_function(result, op1, op2);
    ZVAL_BOOL(result, Z_LVAL_P(result) == 0);
    FREE_OP2();
    ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION();
}
```

9.5 循环结构

实际应用中有许多具有规律性的重复操作，因此在程序中就需要重复执行某些语句。循环结构是在一定条件下反复执行某段程序的流程结构，由循环体、循环终止条件两部分组成，被反复执行的程序被称为循环体。

PHP 中的循环结构有 4 种：while、for、foreach、do while，接下来我们分析这几个结构的具体的实现。

9.5.1 while

while 循环的语法：

```
while(expression) //循环判断条件
{
    statement;//循环体
}
```

while 的结构比较简单，由两部分组成：expression、statement，其中 expression 为循环判断条件，当 expression 为 true 时重复执行 statement。while 语句的语法规则：

```
statement:
    ...
    |   T_WHILE '(' expr ')' while_statement { $$ = zend_ast_create(ZEND_AST_
WHILE, $3, $5); }
    ...
```

```
;

while_statement:
        statement { $$ = $1; }
    |   ':' inner_statement_list T_ENDWHILE ';' { $$ = $2; }
;
```

从 while 语法规则可以看出，在解析时会创建一个 ZEND_AST_WHILE 节点，expression、statement 分别保存在两个子节点中。while 语句编译生成的抽象语法树类似图 9-10。

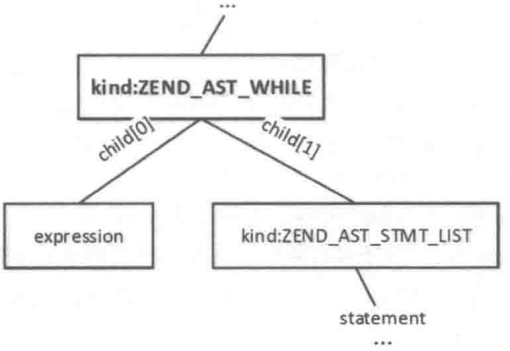


图 9-10　while 语句的抽象语法树

while 编译的过程也比较简单，比较特别的是：while 首先编译的是循环体，然后才是循环判断条件，更像是 do while。编译过程大致如下所述。

步骤（1）：首先编译一条 ZEND_JMP 的 opcode，这条 opcode 用来跳到循环判断条件 expression 的位置，由于 while 是先编译循环体再编译循环条件，所以此时还无法确定具体的跳转值。

步骤（2）：编译循环体 statement，编译完成后后面的指令就是 expression 的，所以更新步骤（1）中 ZEND_JMP 的跳转值。

步骤（3）：编译循环判断条件 expression。

步骤（4）：编译一条 ZEND_JMPNZ 的 opcode，这条 opcode 用于循环判断条件执行完以后跳回循环体的，如果循环条件成立，则通过此 opcode 跳到循环体开始的位置，否则继续往下执行（即跳出循环）。

具体的编译过程：

```
void zend_compile_while(zend_ast *ast)
{
```

```
    zend_ast *cond_ast = ast->child[0];
    zend_ast *stmt_ast = ast->child[1];
    znode cond_node;
    uint32_t opnum_start, opnum_jmp, opnum_cond;

    //1) 编译 ZEND_JMP
    opnum_jmp = zend_emit_jump(0);
    ...
    //2) 编译循环体 statement, opnum_start 为循环体起始位置
    opnum_start = get_next_op_number(CG(active_op_array));
    zend_compile_stmt(stmt_ast);
    //设置 ZEND_JMP opcode 的跳转值
    opnum_cond = get_next_op_number(CG(active_op_array));
    zend_update_jump_target(opnum_jmp, opnum_cond);
    //3) 编译循环条件 expression
    zend_compile_expr(&cond_node, cond_ast);
    //4) 编译 ZEND_JMPNZ, 用于循环条件成立时跳回循环体开始位置: opnum_start
    zend_emit_cond_jump(ZEND_JMPNZ, &cond_node, opnum_start);
    ...
}
```

编译生成的指令集合如图 9-11 所示。

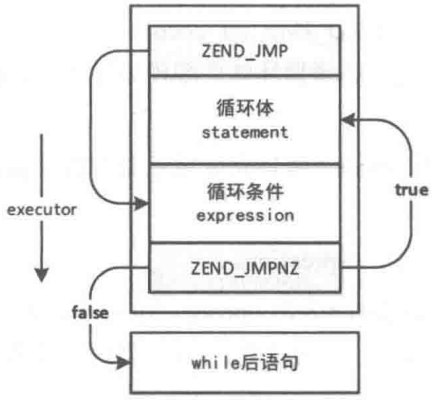


图 9-11 while 循环结构生成的指令集合

运行时首先执行 ZEND_JMP，跳到 while 条件 expression 处开始执行，然后由 ZEND_JMPNZ 对条件的执行结果进行判断。如果条件成立，则跳到循环体 statement 起始位置开始执行；如果条件不成立，则继续向下执行，跳出 while。第一次循环执行以后将不再执行 ZEND_JMP，后

续循环只有靠 ZEND_JMPNZ 控制跳转，循环体执行完成后接着执行循环判断条件，进行下一轮循环的判断。

实际执行时可能会省略 ZEND_JMPNZ 这一步，这是因为很多 while 条件 expression 执行完以后会主动对下一条指令进行判断，如果是 ZEND_JMPNZ 则直接根据条件成立与否进行快速跳转，不需要再由 ZEND_JMPNZ 进行判断，比如：

```
$a = 123;
 while($a > 100)
 {
    echo "yes";
}
```

$a > 100 对应的 opcode：ZEND_IS_SMALLER，执行时发现$a 与 100 类型可以直接比较（都是 long），则直接就能知道循环条件的判断结果，这种情况下将会判断下一条指令是否为 ZEND_JMPNZ，是的话直接设置下一条要执行的 opcode，这样就不需要再单独执行依次 ZEND_JMPNZ 了。这种处理实际上是 expression 中一些指令做了特殊处理，算是一种性能上的优化。如果 gdb 调试 while 时，发现并没有执行 ZEND_JMPNZ，那么基本就是这种情况给优化掉了。

9.5.2　do while

do while 与 while 非常相似，唯一的区别在于 do while 第一次执行时不需要判断循环条件。do while 的语法：

```
do{
    statement;//循环体
}while(expression) //循环判断条件
```

do while 语句也是由两部分组成的：expression、statement。语法规则如下：

```
statement:
    ...
    |   T_DO statement T_WHILE '(' expr ')' ';'
    ...
;
```

do while 编译过程与 while 的基本一致，不同的地方在于 do while 没有 ZEND_JMP 这条指令：

```
void zend_compile_do_while(zend_ast *ast)
{
    zend_ast *stmt_ast = ast->child[0];
    zend_ast *cond_ast = ast->child[1];

    znode cond_node;
    uint32_t opnum_start, opnum_cond;

    //(1)编译循环体 statement, opnum_start 为循环体起始位置
    opnum_start = get_next_op_number(CG(active_op_array));
    zend_compile_stmt(stmt_ast);

    //(2)编译循环判断条件 expression
    opnum_cond = get_next_op_number(CG(active_op_array));
    zend_compile_expr(&cond_node, cond_ast);

    //(3)编译 ZEND_JMPNZ
    zend_emit_cond_jump(ZEND_JMPNZ, &cond_node, opnum_start);
}
```

do while 编译生成的指令集合如图 9-12 所示。

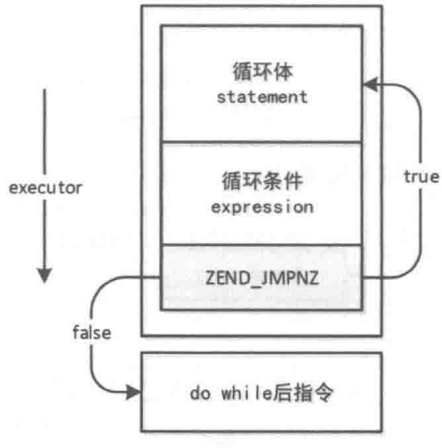


图 9-12　do while 循环结构生成的指令集合

执行时首先执行循环体，然后执行循环判断条件，如果为循环条件为 true，则通过 ZEND_JMPNZ 指令跳回循环体进入下一轮循环，否则结束循环，执行后面的指令。

9.5.3　for

for 循环语法：

```
for (init expr; condition expr; loop expr){
    statement
}
```

for 语句由 4 部分组成：init expr 为初始化表达式，在循环开始前无条件执行一次，后面循环不再执行；condition expr 在每次循环开始前运算，是循环的判断条件，如果值为 true，则继续循环，执行循环体，如果值为 false，则终止循环；loop expr 在每次循环体执行完以后被执行。

for 的语法规则：

```
statement:
    ...
    |  T_FOR '(' for_exprs ';' for_exprs ';' for_exprs ')' for_statement
          { $$ = zend_ast_create(ZEND_AST_FOR, $3, $5, $7, $9); }
    ...
;
```

从语法规则可以看出，for 被编译为 ZEND_AST_FOR 节点，包含 4 个子节点，分别为：init expr、condition expr、loop expr、statement，其中前三个节点为普通的表达式节点，即 ZEND_AST_EXPR_LIST。编译生成的抽象语法树如图 9-13 所示。

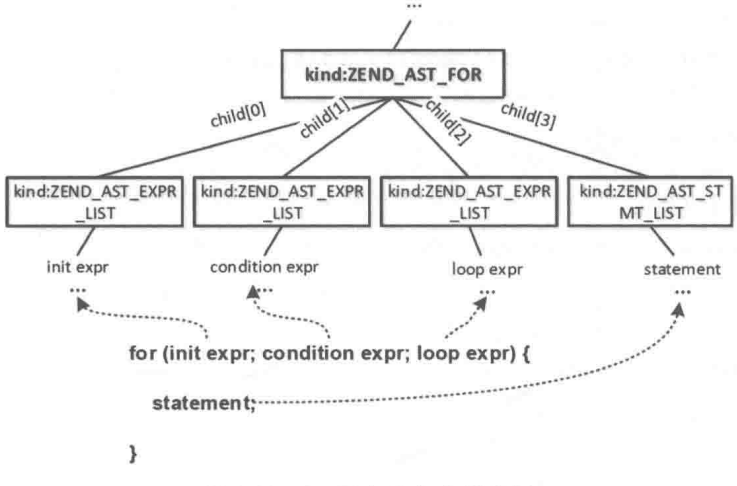


图 9-13　for 语句的抽象语法树

for 语句的编译与 while 类似，也是先编译循环体，再编译循环判断条件，只是多了 init expr、loop expr 两部分，编译过程大致如下所述。

步骤（1）：首先编译初始化表达式：init expr。

步骤（2）：编译一条 ZEND_JMP 指令，该指令用于跳到条件 expression 指令位置，具体跳转值需要编译完循环体后才能确定。

步骤（3）：编译循环体 statement。

步骤（4）：编译 loop expr，然后设置步骤(2)中 ZEND_JMP 的跳转值。

步骤（5）：编译循环条件 condition expr。

步骤（6）：最后编译一条 ZEND_JMPNZ 指令，该指令用于循环条件成立时，跳回循环体起始位置。

具体编译过程：

```
void zend_compile_for(zend_ast *ast)
{
    zend_ast *init_ast = ast->child[0];
    zend_ast *cond_ast = ast->child[1];
    zend_ast *loop_ast = ast->child[2];
    zend_ast *stmt_ast = ast->child[3];

    znode result;
    uint32_t opnum_start, opnum_jmp, opnum_loop;

    //1) 编译 init expression
    zend_compile_expr_list(&result, init_ast);
    zend_do_free(&result);
    //2) 编译 ZEND_JMP
    opnum_jmp = zend_emit_jump(0);
    //3) 编译循环体
     //opnum_start 是循环体起始位置
    opnum_start = get_next_op_number(CG(active_op_array));
    zend_compile_stmt(stmt_ast);
    //4) 编译 loop expression
    opnum_loop = get_next_op_number(CG(active_op_array));
    zend_compile_expr_list(&result, loop_ast);
    zend_do_free(&result);
```

```
    //设置 ZEND_JMP 跳转值
    zend_update_jump_target_to_next(opnum_jmp);
    //5) 编译循环条件 expression
    zend_compile_expr_list(&result, cond_ast);
    zend_do_extended_info();
    //6) 编译 ZEND_JMPNZ
    zend_emit_cond_jump(ZEND_JMPNZ, &result, opnum_start);
}
```

编译生成的指令集合如图 9-14 所示。

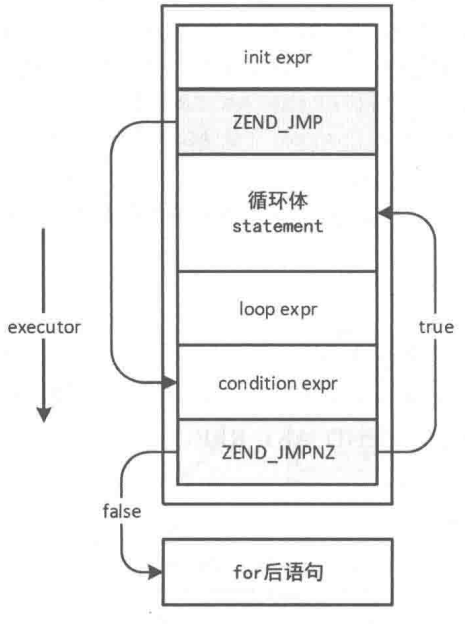


图 9-14　for 循环结构生成的指令集合

执行时首先执行 init expr，然后 ZEND_JMP 指令跳到循环判断条件处，执行条件表达式，ZEND_JMPNZ 指令根据条件表达式的执行结果判断循环条件是否成立，如果成立则跳转到循环体指令位置执行；当循环体执行完成后，继续执行 loop 表达式，然后再次执行循环条件，进入下一轮循环。

9.5.4　foreach

foreach 是 PHP 针对数组、对象提供的一种遍历方式。foreach 的语法：

```
foreach (array_expression as $key => $value){
    statement;
}
```

遍历 arraiy_expression 时，每次循环会把当前单元的值赋给$value，当前单元的键值赋给$key，其中$key 可以省略，$value 前也可以加"&"表示引用单元的值。

语法规则如下：

```
statement:
    ...
    //省略 key 的规则: foreach($array as $v){ ... }
    |  T_FOREACH '(' expr T_AS foreach_variable ')' foreach_statement
          { $$ = zend_ast_create(ZEND_AST_FOREACH, $3, $5, NULL, $7); }
    //有 key 的规则: foreach($array as $k=>$v){ ... }
    |      T_FOREACH  '('  expr  T_AS  foreach_variable  T_DOUBLE_ARROW
foreach_variable ')' foreach_statement
          { $$ = zend_ast_create(ZEND_AST_FOREACH, $3, $7, $5, $9); }
    ...
;
```

foreach 在编译阶段解析为 ZEND_AST_FOREACH 节点，该节点包含 4 个子节点，分别为：遍历的数组或对象、元素的 value、元素的 key、循环体，如果 value 是指向数组或对象成员的引用，则 value 对应的节点类型为 ZEND_AST_REF。foreach 生成的抽象语法树如图 9-15 所示。

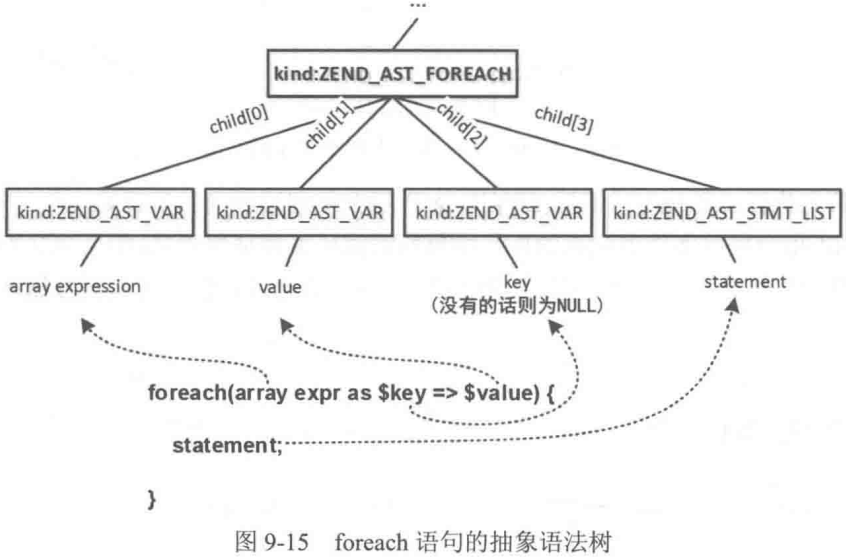


图 9-15　foreach 语句的抽象语法树

相对上面几种常规的循环结构，foreach 的实现略显复杂，`$key`、`$value` 实际上就是两个普通的局部变量，遍历的过程就是对两个局部变量不断赋值、更新的过程。以数组为例，首先将数组复制一份用于遍历，从 arData 第一个元素开始，把 Bucket.val.value 值赋值给`$value`，把 Bucket->key（或 Bucket->h）赋值给`$key`，然后更新迭代位置：将下一个元素的位置更新到数组的 zval.u2.fe_pos 中。下一轮遍历时直接从 zval.u2.fe_pos 开始，这也是遍历前为什么要复制一份变量用于遍历的原因，如果发现 zval.u2.fe_pos 已经到达 arData 末尾了，则结束遍历，并且销毁复制的用于遍历的变量。如果遍历的是对象，则使用 zval.u2.fe_iter_idx 保存迭代位置。例如：

```
//示例 9.5.4
$arr = array(1,2,3);
foreach($arr as $k=>$v){
    echo $v;
}
```

遍历时首先会将`$arr` 复制出一个临时变量，该变量保存遍历的位置，对应的内存结构如图 9-16 所示。

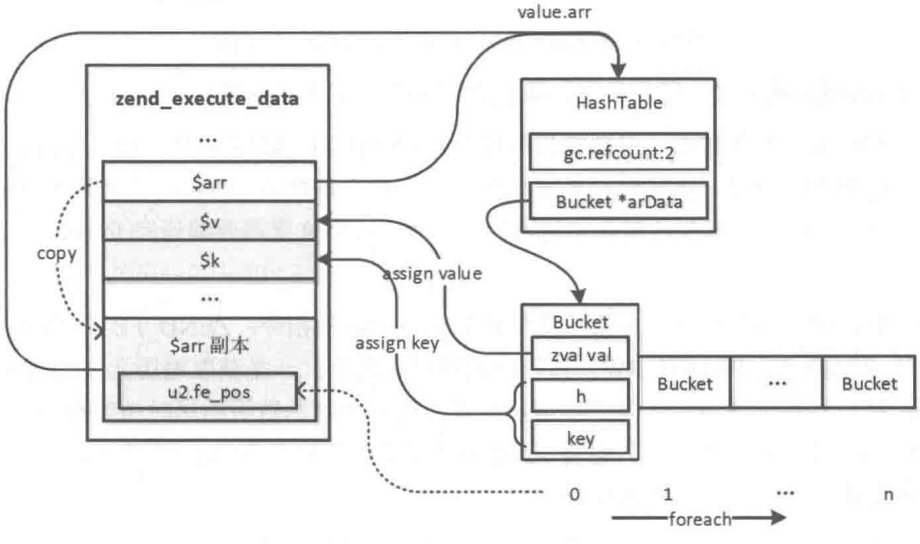


图 9-16　foreach 数组的遍历

如果 value 是引用，则在循环前先将原数组或对象转为引用类型，后面的过程就与上面的一致了，例如：

```
$arr = array(1,2,3);
```

```
foreach($arr as $k=>&$v){
    echo $v;
}
```

这种情况下相关变量的内存结构如图 9-17 所示。

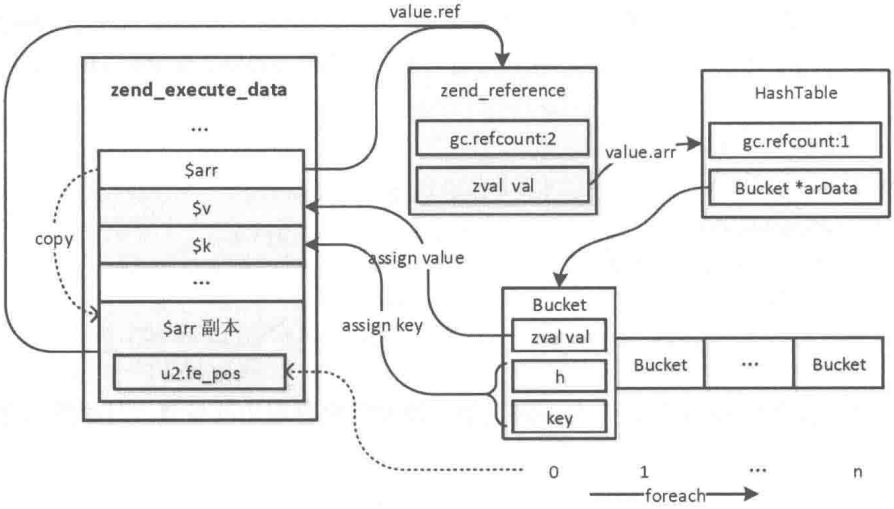


图 9-17　foreach 遍历 value 为引用的内存结构

了解了 foreach 的实现、运行机制，我们再回头看一下其编译过程。

步骤（1）：编译复制数组、对象操作的指令：ZEND_FE_RESET_R，如果 value 是引用则是 ZEND_FE_RESET_RW。执行时如果发现遍历的变量不是数组、对象，则抛出一个 warning，然后跳出循环，所以这条指令还需要知道跳出的位置，这个位置需要编译完 foreach 以后才能确定。

步骤（2）：编译获取数组、对象当前单元 key、value 的指令：ZEND_FE_FETCH_R，如果是引用则是 ZEND_FE_FETCH_RW。该指令的作用有两个：一是获取遍历元素的 key、value，并把 value 赋值给局部变量；二是更新遍历位置，当遍历到达数组结尾时结束遍历。需要注意的是，ZEND_FE_FETCH_R 指令只会将 value 的值赋值给局部变量，key 的值并没有直接赋值，而是将 key 的值保存到了一个 TMPVAR 中。

步骤（3）：如果 foreach 定义了 key 则编译一条赋值指令，作用是将步骤（2）中的 ZEND_FE_FETCH_R 取到的 key 赋值给局部变量。

步骤（4）：编译循环体 statement。

步骤（5）：编译跳回遍历开始位置的指令：ZEND_JMP，一次遍历结束时会跳回步骤（2）编译的指令处进行下一轮遍历。

步骤（6）：设置步骤（1）、（2）中生成的 ZEND_FE_RESET_R、ZEND_FE_FETCH_R 跳过的指令数。

步骤（7）：编译 ZEND_FE_FREE，此操作用于释放步骤（1）复制的数组、对象。

具体的编译过程：

```
void zend_compile_foreach(zend_ast *ast)
{
    zend_ast *expr_ast = ast->child[0];
    zend_ast *value_ast = ast->child[1];
    zend_ast *key_ast = ast->child[2];
    zend_ast *stmt_ast = ast->child[3];
    zend_bool by_ref = value_ast->kind == ZEND_AST_REF;
    ...
    //1) 编译 ZEND_FE_RESET_R 指令
    opnum_reset = get_next_op_number(CG(active_op_array));
    opline = zend_emit_op(&reset_node, by_ref ? ZEND_FE_RESET_RW : ZEND_FE_
RESET_R, &expr_node, NULL);
    //2) 编译 ZEND_FE_FETCH_R 指令
    opnum_fetch = get_next_op_number(CG(active_op_array));
    opline = zend_emit_op(NULL, by_ref ? ZEND_FE_FETCH_RW : ZEND_FE_FETCH_R,
&reset_node, NULL);
    //将$value 编译为局部变量
    if (value_ast->kind == ZEND_AST_VAR &&
        zend_try_compile_cv(&value_node, value_ast) == SUCCESS) {
        SET_NODE(opline->op2, &value_node);
    } else {
        ...
    }

    if (key_ast) {
        //将 ZEND_FE_FETCH_R 指令增加 result 操作数，类型为 IS_TMP_VAR，用于保存 key
        opline = &CG(active_op_array)->opcodes[opnum_fetch];
        zend_make_tmp_result(&key_node, opline);
        //3) 生成 key 赋值的指令
        zend_emit_assign_znode(key_ast, &key_node);
    }
    //4) 编译循环体
```

```
    zend_compile_stmt(stmt_ast);
    //5) 编译 ZEND_JMP 指令
    zend_emit_jump(opnum_fetch);
    //更新 ZEND_FE_RESET_R 指令的跳出值
    opline = &CG(active_op_array)->opcodes[opnum_reset];
    opline->op2.opline_num = get_next_op_number(CG(active_op_array));
    //更新 ZEND_FE_FETCH_R 指令的跳出值
    opline = &CG(active_op_array)->opcodes[opnum_fetch];
    opline->extended_value = get_next_op_number(CG(active_op_array));
    ...
    //6) 编译 ZEND_FE_FREE 指令
    zend_emit_op(NULL, ZEND_FE_FREE, &reset_node, NULL);
}
```

如果 value 不是 CV 变量，则在步骤（2）后会生成其他指令，这里只介绍常规用法，编译生成的指令集合类似图 9-18 所示。

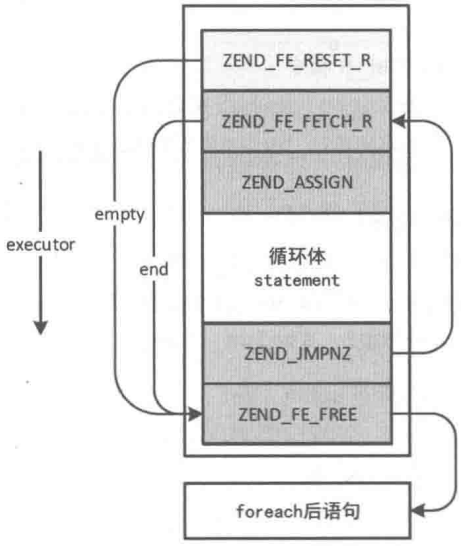


图 9-18 foreach 循环结构生成的指令集合

以示例 9.5.4 为例，运行时的基本流程如下所述。

步骤（1）：执行 ZEND_FE_RESET_R，该指令将$arr 复制一份，并把遍历位置归零。如果遍历的变量不是对象和数组，则跳出 foreach。

```
ZEND_VM_HANDLER(77, ZEND_FE_RESET_R, CONST|TMP|VAR|CV, ANY)
```

```
    {
        ...
        zval *array_ptr, *result;
        ...
        array_ptr = GET_OP1_ZVAL_PTR_DEREF(BP_VAR_R);
        if (EXPECTED(Z_TYPE_P(array_ptr) == IS_ARRAY)) {
            //result 为副本保存位置
            result = EX_VAR(opline->result.var);
            //拷贝
            ZVAL_COPY_VALUE(result, array_ptr);
            if (OP1_TYPE != IS_TMP_VAR && Z_OPT_REFCOUNTED_P(result)) {
                Z_ADDREF_P(array_ptr);
            }
            //将 u2.fe_ps 设置为 0
            Z_FE_POS_P(result) = 0;
            ...
        } else if (OP1_TYPE != IS_CONST && EXPECTED(Z_TYPE_P(array_ptr) ==
IS_OBJECT)) {
            //遍历的是对象
            ...
        } else {
            ...
            //跳出 foreach
            ZEND_VM_JMP(OP_JMP_ADDR(opline, opline->op2));
        }
    }
```

步骤（2）：执行 ZEND_FE_FETCH_R 指令，根据 zval.u2.fe_pos 取出遍历数组中的元素，并复制给$value，如果指定了 key，则把 key 的值复制到一个 TMPVAR 中，然后更新 zval.u2.fe_pos，如果遍历到数组末尾了，则跳到步骤（6）。

```
    ZEND_VM_HANDLER(78, ZEND_FE_FETCH_R, VAR, ANY)
    {
        zval *array;
        zval *value;
        uint32_t value_type;
        HashTable *fe_ht;
        HashPosition pos;
```

```
        Bucket *p;
        //获取遍历的变量
        array = EX_VAR(opline->op1.var);
        //数组的情况:
        fe_ht = Z_ARRVAL_P(array);
        pos = Z_FE_POS_P(array);
        p = fe_ht->arData + pos;
        while (1) {
            if (UNEXPECTED(pos >= fe_ht->nNumUsed)) {
                //遍历结束
                ZEND_VM_C_GOTO(fe_fetch_r_exit);
            }
            value = &p->val;
            ...
        }
        //更新遍历位置
        Z_FE_POS_P(array) = pos + 1;
        if (opline->result_type == IS_TMP_VAR) {
            //将 key 值赋值到 TMPVAR
            if (!p->key) {
                ZVAL_LONG(EX_VAR(opline->result.var), p->h);
            } else {
                ZVAL_STR_COPY(EX_VAR(opline->result.var), p->key);
            }
        }

        if (EXPECTED(OP2_TYPE == IS_CV)) {
            zval *variable_ptr = _get_zval_ptr_cv_undef_BP_VAR_W(execute_data,
opline->op2.var);
            //将 value 赋值给局部变量
            zend_assign_to_variable(variable_ptr, value, IS_CV);
        } else {
            //接收 value 的变量不是 CV 类型
            ...
        }
        ZEND_VM_NEXT_OPCODE();
    }
```

步骤（3）：执行赋值语句：将 key 赋值到指定的变量，在步骤（1）中，ZEND_FE_FETCH_R 指令并没有直接把 key 值赋给局部变量，而是保存到了一个 TMPVAR 中，这一步就是将 TMPVAR 中的值复制给$key。

步骤（4）：执行循环体。

步骤（5）：执行 ZEND_JMPNZ 指令，跳回步骤（2）。

步骤（6）：遍历结束，执行 ZEND_FE_FREE 释放复制的用于遍历的副本。

```
ZEND_VM_HANDLER(127, ZEND_FE_FREE, TMPVAR, ANY)
{
    zval *var;
    USE_OPLINE

    SAVE_OPLINE();
    var = EX_VAR(opline->op1.var);
    if (Z_TYPE_P(var) != IS_ARRAY && Z_FE_ITER_P(var) != (uint32_t)-1) {
        zend_hash_iterator_del(Z_FE_ITER_P(var));
    }
    //销毁
    zval_ptr_dtor_nogc(var);
    ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION();
}
```

9.6　中断及跳转

PHP 中的中断及跳转语句主要有 break、continue、goto，这几种语句的实现基础都是跳转指令，其中 break、continue 可以用于循环、switch 语句，goto 用于跳转到指定标签。

9.6.1　break/continue

break 用于结束当前 for、foreach、while、do-while 或者 switch 结构的执行；continue 用于跳过本次循环中剩余代码，进行下一轮循环。break、continue 是非常相像的，它们都可以接受一个可选数字参数来决定跳过的循环层数，两者的不同点在于 break 是跳到循环结束的位置，而 continue 是跳到循环判断条件的位置，并没有本质上的差别。

上一节我们已经介绍过循环语句的编译，其中在各种循环编译过程中有两个特殊操作：zend_begin_loop()、zend_end_loop()，分别在循环编译前与编译后调用，这两步操作就是为 break、

continue 服务的。在每层循环编译时，都会创建一个 zend_brk_cont_element 的结构，这个结构用于索引循环的判断条件及结束的指令位置：

```
typedef struct _zend_brk_cont_element {
    int start;
    int cont;
    int brk;
    int parent;
} zend_brk_cont_element;
```

cont 记录的是循环判断条件的指令起始位置，brk 记录的是循环结束的位置，parent 记录的是父层循环 zend_brk_cont_element 结构的存储位置。多层嵌套循环会生成一个 zend_brk_cont_element 的链表，每层循环编译结束时更新自己的 zend_brk_cont_element 结构，所以 break、continue 的实现实际上就是根据跳出的层级索引到那一层的 zend_brk_cont_element 结构，然后根据得到的 cont、brk 进行相应的跳转。

各循环的 zend_brk_cont_element 结构保存在 zend_op_array->brk_cont_array 数组中，编译各循环时，依次申请一个 zend_brk_cont_element。zend_op_array->last_brk_cont 记录此数组第一个可用位置，每申请一个元素 last_brk_cont 就相应地增加 1，然后将数组扩容，parent 记录的就是父层循环结构在该数组中的存储位置。

```
static inline void zend_begin_loop(zend_uchar free_opcode, const znode
*loop_var)
{
    zend_brk_cont_element *brk_cont_element;
    int parent = CG(context).current_brk_cont;
    zend_loop_var info = {0};
    //将 CG(context).current_brk_cont 设置为当前循环
    CG(context).current_brk_cont = CG(active_op_array)->last_brk_cont;
    //分配 zend_brk_cont_element
    brk_cont_element = get_next_brk_cont_element(CG(active_op_array));
    brk_cont_element->parent = parent;
    ...
}
zend_brk_cont_element *get_next_brk_cont_element(zend_op_array *op_array)
{
    op_array->last_brk_cont++;
    //扩容
```

```
    op_array->brk_cont_array = erealloc(op_array->brk_cont_array, sizeof
(zend_brk_cont_element)*op_array->last_brk_cont);
    return &op_array->brk_cont_array[op_array->last_brk_cont-1];
}
```

循环编译前调用 zend_begin_loop()申请一个 zend_brk_cont_element 结构，并将该结构的存储位置保存到 CG(context).current_brk_cont，编译完成后，根据这个存储位置取出它的 zend_brk_cont_element，然后更新 cont、brk。

```
static inline void zend_end_loop(int cont_addr)
{
    zend_brk_cont_element *brk_cont_element
        = &CG(active_op_array)->brk_cont_array[CG(context).current_brk_cont];
    //设置循环判断条件起始指令的位置
    brk_cont_element->cont = cont_addr;
    brk_cont_element->brk = get_next_op_number(CG(active_op_array));
    CG(context).current_brk_cont = brk_cont_element->parent;

    zend_stack_del_top(&CG(loop_var_stack));
}
```

举个例子来看：

```
$i = 0;
while(1){
    while(1){
        if($i > 10){
            break 2;
        }
        ++$i
    }
    echo "inner end";
}
echo "outer end";
```

循环编译完以后的 zend_brk_cont_element 结构如图 9-19 所示。

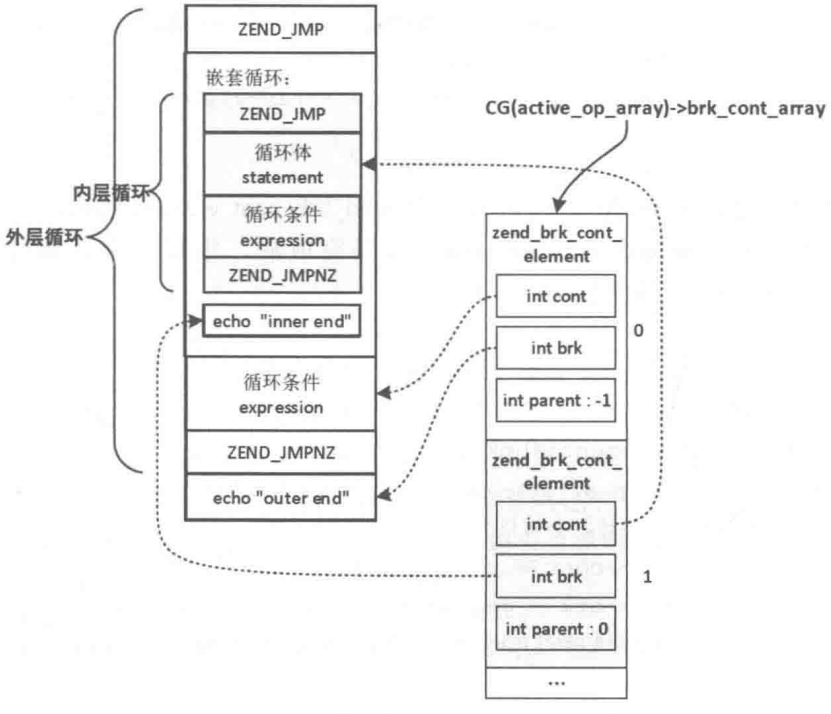


图 9-19　循环结构的 zend_brk_cont_element

有了前面循环结构的准备，break、continue 的编译过程就比较简单了，主要是各生成一条临时指令：ZEND_BRK、ZEND_CONT，这两条并不是 ZendVM 执行的指令，它们最终会被编译为跳转指令。这条 opcode 记录着两个信息。

- **op1**：记录当前循环 zend_brk_cont_element 结构的存储位置，在循环编译过程中，当前循环 zend_brk_cont_element 的位置通过 CG(context).current_brk_cont 保存。
- **op2**：记录要跳出循环的层级，如果 break/continue 没有加数字，则默认为 1。

break、continue 语句的具体编译过程如下：

```
void zend_compile_break_continue(zend_ast *ast)
{
    zend_ast *depth_ast = ast->child[0];

    zend_op *opline;
    int depth;
    //跳出的层级
    if (depth_ast) {
        zval *depth_zv;
```

```
            ...
            depth = Z_LVAL_P(depth_zv);
        } else {
            depth = 1;
        }
        ...
        //生成 ZEND_BRK 或 ZEND_CONT 指令
        opline = zend_emit_op(NULL, ast->kind == ZEND_AST_BREAK ? ZEND_BRK :
ZEND_CONT, NULL, NULL);
        //break、continue 所在循环层
        opline->op1.num = CG(context).current_brk_cont;
        opline->op2.num = depth;  //要跳出的层数
    }
```

zend_compile_break_continue()处理完成后，break、continue 还没有完成最后的编译，因为 break、continue 在循环结构内部，对于循环结构的编译而言，此时处于编译循环体的环节，尚未编译完成。在整个脚本编译完成后，会把 ZEND_BRK、ZEND_CONT 指令修改为 ZEND_JMP，跳转值根据对应循环 zend_brk_cont_element 结构获得，这个操作在 pass_two()中完成。

```
ZEND_API int pass_two(zend_op_array *op_array)
{
    ...
    opline = op_array->opcodes;
    end = opline + op_array->last;
    while (opline < end) {
        switch (opline->opcode) {
            ...
            case ZEND_BRK:
            case ZEND_CONT:
            {
                //计算跳转位置
                uint32_t jmp_target = zend_get_brk_cont_target(op_array, opline);
                ...
                //将指令修改为 ZEND_JMP
                opline->opcode = ZEND_JMP;
                opline->op1.opline_num = jmp_target;
                opline->op2.num = 0;
                //将绝对跳转指令位置修改为相对当前 opcode 的位置
```

```
                ZEND_PASS_TWO_UPDATE_JMP_TARGET(op_array, opline, opline->op1);
            }
            break;
            ...
        }
    }
    ...
}
```

编译 ZEND_JMP 指令时，根据 ZEND_BRK、ZEND_CONT 的 op1、op2 取出 break、continue 语句所在循环的 zend_brk_cont_element 结构，以及要跳过的层级，然后遍历多层循环的 zend_brk_cont_element 链表，找到对应循环的 zend_brk_cont_element 结构。比如上面那个示例，break 2 目标是 break 当前所在循环的父层，首先根据 op1 取出当前循环的结构，也就是 CG(active_op_array)->brk_cont_array[1]，这是一层循环，然后接着 parent 查找父层，这又是一层循环，总共查找 2 层，最终得到 CG(active_op_array)->brk_cont_array[0]。具体查找过程如下：

```
//file: zend_opcode.c
static uint32_t zend_get_brk_cont_target(const zend_op_array *op_array,
const zend_op *opline) {
    //跳出的层级: break n;
    int nest_levels = opline->op2.num;
    //break、continue 所属循环 zend_brk_cont_element 的存储下标
    int array_offset = opline->op1.num;
    zend_brk_cont_element *jmp_to;
    do {
        //从 break/continue 所在循环层开始
        jmp_to = &op_array->brk_cont_array[array_offset];
        if (nest_levels > 1) {
            //如果还没到要跳出的层数则接着跳到上层
            array_offset = jmp_to->parent;
        }
    } while (--nest_levels > 0);

    return opline->opcode == ZEND_BRK ? jmp_to->brk : jmp_to->cont;
}
```

最终执行时的指令如图 9-20 所示。

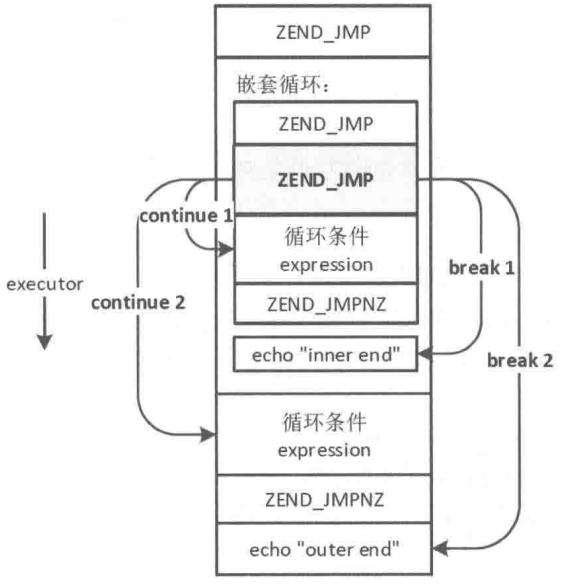


图 9-20　break/continue 指令

> 在多层循环中，直接根据层级数字 break、continue 跳转很不方便，Go 语言中有一种按标签跳转的语法：break/continue + LABEL，PHP 中如何实现类似的功能呢？根据上一节及本节介绍的内容，该特性实现起来并不复杂，本章将在后续的部分介绍如何实现，有兴趣的读者可以自己先想一下思路。

9.6.2　goto

goto 操作符可以用来跳转到程序中的另一位置，该目标位置可以用目标名称加上冒号来标记，也就是标签，goto 后面直接加目标位置即可跳转到对应标签处。PHP 中的 goto 有一定限制，目标位置只能位于同一个文件和作用域，也就是说无法跳出一个函数或类方法，也无法跳入到另一个函数，可以跳出循环但无法跳入循环，多层循环中通常会用 goto 代替多层 break。goto 的语法：

```
goto LABEL;
statement;
LABEL:
    statement;
```

goto 与 label 需要组合使用，其实现与 break、continue 类似，最终也是被优化为 ZEND_JMP。首先看一下定义一个 label 时都有哪些操作，标签的语法规则：

```
statement:
    ...

    |   T_STRING ':' { $$ = zend_ast_create(ZEND_AST_LABEL, $1); }
;
```

标签被编译为 ZEND_AST_LABEL 节点，该节点只有一个子节点，用于保存标签的名称。编译时会把标签插入 CG(context).labels 哈希表中，key 就是标签名称，value 是一个 zend_label 结构，该结构保存着标签后的指令位置，以及标签所在循环：

```
typedef struct _zend_label {
    int brk_cont; //当前标签所在循环
    uint32_t opline_num; //下一条指令位置
} zend_label;
```

brk_cont 用于记录当前标签所在的循环，这个值就是上面介绍的每个循环在 zend_op_array->brk_cont_array 数组中的位置；opline_num 比较容易理解，就是标签下面第一条指令的位置。到这里你应该能猜得到 goto 的处理方式了：首先根据标签名称在 CG(context).labels 查找到标签的 zend_label 结构，然后跳到 zend_label.opline_num 的位置，brk_cont 的作用是用来判断是不是 goto 到了另一层循环中去。标签的具体编译过程：

```
void zend_compile_label(zend_ast *ast)
{
    zend_string *label = zend_ast_get_str(ast->child[0]);
    zend_label dest;

    //编译时会将 label 插入 CG(context).labels 哈希表
    if (!CG(context).labels) {
        ALLOC_HASHTABLE(CG(context).labels);
        zend_hash_init(CG(context).labels, 8, NULL, label_ptr_dtor, 0);
    }

    //设置标签信息：当前所在循环、下一条指令位置
    dest.brk_cont = CG(context).current_brk_cont;
    dest.opline_num = get_next_op_number(CG(active_op_array));
    //将标签插入 CG(context).labels
    if (!zend_hash_add_mem(CG(context).labels, label, &dest, sizeof
```

```
(zend_label))) {
        zend_error_noreturn(E_COMPILE_ERROR, "Label '%s' already defined",
ZSTR_VAL(label));
    }
}
```

goto 的编译过程：

```
void zend_compile_goto(zend_ast *ast)
{
    zend_ast *label_ast = ast->child[0];
    znode label_node;
    zend_op *opline;
    uint32_t opnum_start = get_next_op_number(CG(active_op_array));

    zend_compile_expr(&label_node, label_ast);

    //如果当前在一个循环内则有的情况下是不能简单跳出循环的
    zend_handle_loops_and_finally();
    //编译一条临时指令：ZEND_GOTO
    opline = zend_emit_op(NULL, ZEND_GOTO, NULL, &label_node);
    opline->op1.num = get_next_op_number(CG(active_op_array)) - opnum_start - 1;
    opline->extended_value = CG(context).current_brk_cont;
}
```

goto 初步被编译为 ZEND_GOTO，其中标签名称保存在 op2，extended_value 记录的是 goto 所在循环，如果没有在循环中这个值就等于-1。op1 比较特殊，从上面编译的过程分析，它的值等于 goto 之间的指令数，goto 只编译了一条 ZEND_GOTO，哪来的其他指令呢？上一节介绍的循环结构中有一个比较特殊：foreach，它在遍历前会新生成一个 zval 用于遍历，这个 zval 在循环结束时才被释放，假如 foreach 循环体中执行了 goto，直接像普通跳转一样跳到了别的位置，那么这个 zval 就无法释放了。所以这种情况下在 goto 跳转前需要先执行这些收尾的指令，这些指令就是上面 zend_handle_loops_and_finally()编译的，具体的细节这里不再展开，有兴趣的可以仔细研究 foreach 编译时 zend_begin_loop()的特殊处理。

后面的处理就与 break、continue 一样了，在 pass_two()中 ZEND_GOTO 被重置为 ZEND_JMP，具体的处理过程在 zend_resolve_goto_label()，比较简单，不再赘述。

9.7 include/require

在实际应用中，我们不可能把所有的代码都写到一个文件中，而是会按照一定的标准进行文件划分，include 与 require 的功能就是将其他文件包含进来并且执行，比如在面向对象中，通常会把一个类定义在单独文件中，使用时再 include 进来。include 与 require 没有本质上的区别，唯一的不同在于错误级别，当文件无法被正常加载时 include 会抛出 warning 警告，而 require 则会抛出 error 错误。

在分析 include 的实现过程之前，首先要明确 include 的基本用法及特点：

- 被包含的文件将继承 include 所在行具有的全部变量范围，比如调用文件前面定义了一些变量，那么这些变量就能够在被包含的文件中使用，反之，被包含文件中定义的变量也将从 include 调用处开始可以被调用文件所使用。
- 被包含文件中定义的函数、类在 include 执行之后将可以被随处使用，具有全局作用域。
- include 是在运行时加载文件并执行的，而不是在编译时。

这几个特性可以简单地理解为：include 就是把其他文件的内容复制到了调用文件中，类似 C 语言中的宏（当然执行的时候并不是这样）。举个例子来说明：

```
//a.php
$var_1 = "hi";
$var_2 = array(1,2,3);
include 'b.php';
var_dump($var_2);
var_dump($var_3);

//b.php
$var_2 = array();
$var_3 = 9;
```

执行 `php a.php` 结果显示：`$var_2` 值被修改为 array()了，而 include 文件中新定义的`$var_3`也可以在 a.php 中使用。接下来我们就以这个例子为例，详细介绍 include 具体是如何实现的。

第 5 章曾详细介绍过 Zend 引擎的编译、执行两个阶段的处理，整个过程的输入是一个文件，然后经过 PHP 代码→抽象语法树→opline 指令→execute 一系列过程完成整个处理。编译过程的输入是一个文件，输出是 zend_op_array，输出接着成为执行过程的输入，而 include 的处理过程与之相同，执行 include 时把被包含的文件像主脚本一样编译然后执行，接着再回到调用 include 的位置继续执行。整体的处理过程如图 9-21 所示。

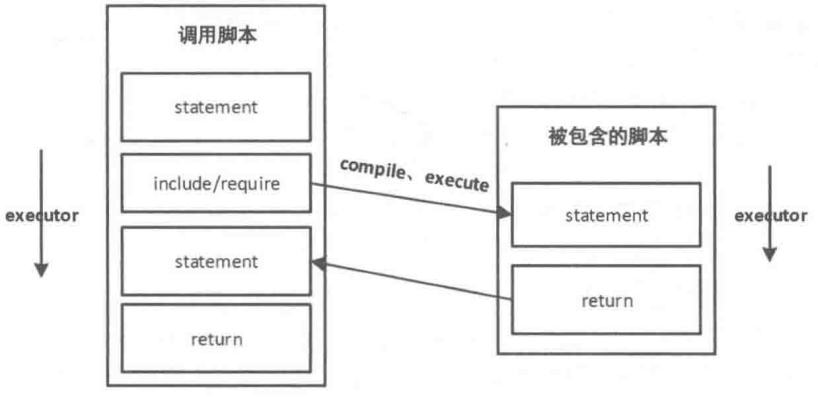


图 9-21　include 的调用、执行过程

include 的编译过程非常简单，只有一条指令：ZEND_INCLUDE_OR_EVAL。上面的示例在执行时将调用 ZEND_INCLUDE_OR_EVAL_SPEC_CONST_HANDLER 进行处理，处理过程就是 PHP 脚本完整的编译和执行流程：

```
    static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_INCLUDE_OR_EVAL_SPEC_
CONST_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
    {
        //include 文件编译的 zend_op_array
        zend_op_array *new_op_array=NULL;
        ...
        //include 的文件名
        inc_filename = EX_CONSTANT(opline->op1);
        ...
        switch (opline->extended_value) {
            ...
            case ZEND_INCLUDE:
            case ZEND_REQUIRE:
                //1) 编译 include 的文件
                new_op_array = compile_filename(opline->extended_value, inc_
filename);
                break;
            ...
        }
        ...
        //2) 下面开始执行流程
        zend_execute_data *call;
```

```
    //分配运行时的 zend_execute_data
    call = zend_vm_stack_push_call_frame(ZEND_CALL_NESTED_CODE,
            (zend_function*)new_op_array, 0, EX(called_scope), Z_OBJ(EX(This)));

    //继承调用文件的全局变量符号表
    if (EX(symbol_table)) {
        call->symbol_table = EX(symbol_table);
    } else {
        call->symbol_table = zend_rebuild_symbol_table();
    }
    //保存当前 zend_execute_data，include 执行完再还原
    call->prev_execute_data = execute_data;
    //执行前初始化
    i_init_code_execute_data(call, new_op_array, return_value);
    //zend_execute_ex 执行器入口，如果没有自定义这个函数则默认为 execute_ex()
    if (EXPECTED(zend_execute_ex == execute_ex)) {
        //将执行器切到新的 zend_execute_data，回忆一下 execute_ex()中的切换过程
        ZEND_VM_ENTER();
    }
    ...
}
```

整个过程比较容易理解，编译的过程不再重复，与之前介绍的没有差别。执行的过程与函数的调用过程非常相似，首先也是重新分配了一个 zend_execute_data，然后将执行器切换到新的 zend_execute_data，执行完以后再切回调用处，如果 include 文件中只定义了函数、类，没有定义全局变量，则执行过程实际直接执行 return，只是在编译阶段将编译出的函数、类注册到 EG(function_table)、EG(class_table)中了，这种情况比较简单，直接跳过。但是如果有全局变量定义处理就比较复杂了，比如上面那个例子，两个文件中都定义了全局变量，这些变量是如何被继承、合并的呢？

include 的执行过程中还有一个关键操作：i_init_code_execute_data()。关于这个函数在前面介绍 zend_execute()时曾提过，这个函数除了一些上下文的设置，还会调用 zend_attach_symbol_table()把当前 zend_op_array 下的变量移到 EG(symbol_table)全局变量符号表中去，这些变量相对自己的作用域是局部变量，但它们定义在函数之外，实际上也是全局变量，可以在函数中通过 global 访问。在执行前会把所有在 PHP 中定义的 CV 变量插入 EG(symbol_table)，value 指向 zend_execute_data 局部变量的 zval。比如上面的示例，a.php 在执行时会把$var_1、$var_2

注册到 EG(symbol_table)，EG(symbol_table)中的 value 指向局部变量，也就是 zend_execute_data 上的局部变量，如图 9-22 所示。

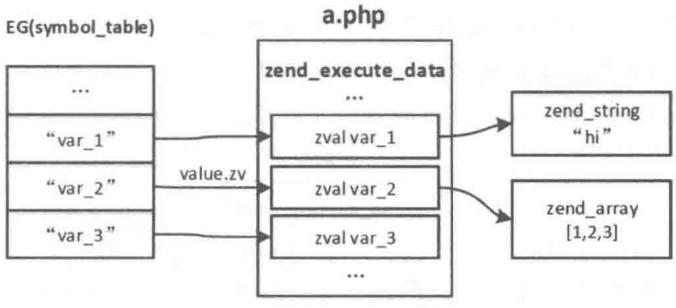


图 9-22　a.php 中局部变量与全局变量的索引关系

而 include 时也会在 i_init_code_execute_data()中调用 zend_attach_symbol_table()进行全局变量注册，这个时候如果发现 var 已经在 EG(symbol_table)中存在了，则会把 EG(symbol_table)符号表中的 value 修改为指向 include 文件的 zend_execute_data 上的局部变量，同时会把原来的 value 赋值给 include 文件中的局部变量。简单地讲，就是被包含文件中的变量会继承、覆盖调用文件中的变量，这就是为什么被包含文件中可以直接使用调用文件中定义的变量的原因。上面的例子，a.php 在 include 'b.php'执行 b.php 时，发现`$var_2`、`$var_3` 已经在 EG(symbol_table)中存在了，则会把 value 复制给 b.php 中的`$var_2`、`$var_3`，也就是把 a.php 中局部变量的值复制给了 b.php 中，同时修改 EG(symbol_table)中的 value，使其指向 b.php 中的`$var_2`、`$var_3`，此时 a.php、b.php 及 EG(symbol_table)的变量关系如图 9-23 所示。

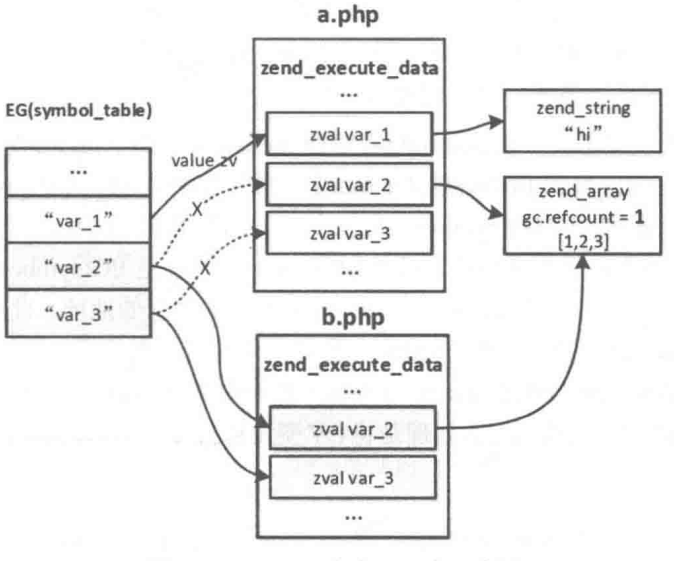


图 9-23　include 脚本继承全局变量

这里的 include 文件中定义的 var_2 实际上是替换了原文件中的变量，从逻辑上讲只有一个 var_2，所以此处 zend_array 的引用是 1 而不是 2。接下来就是执行 include 脚本的执行，执行到 `$var_2 = array()`时，将原 array(1,2,3)引用减 1 变为 0，这时候将其释放，然后将新的 value 赋给`$var_2`，这个过程就是普通变量的赋值过程。注意：此时 a.php 中的`$var_2` 仍然指向被释放掉的 value。如果没有修改继承的变量，则将始终使用 a.php 中的变量，该过程相当于 b.php 中的变量覆盖了 a.php 的局部变量，此时的内存关系如图 9-24 所示。

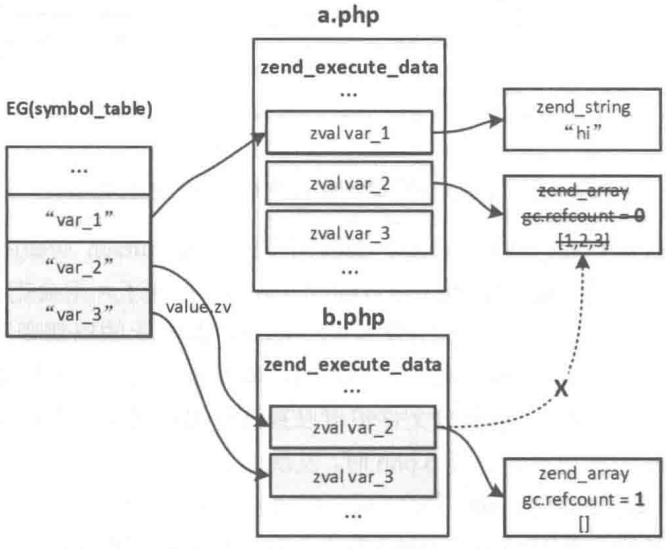


图 9-24 include 脚本覆盖继承的全局变量

看到这里你可能会有一个疑问：`$var_2` 既然被重新修改为新的一个值了，那么为什么调用文件中的`$var_2` 仍然指向释放掉的 value 呢？include 执行完成回到原来的调用文件中后，为什么可以读取到新的`$var_2` 值呢？这个问题的答案在被包含文件执行完成后 return 的过程中。

被包含文件执行完成以后，将执行 return 返回 include 的位置，return 时会把 include 文件中的全局变量从局部变量区移到 EG(symbol_table)中，执行前的 zend_attach_symbol_table()不是已经将局部变量注册到了 EG(symbol_table)中吗？这里的移动是把 value 值更新到 EG(symbol_table)，而不是像原来那样间接地指向 value，也就是 EG(symbol_table)中的 value 直接就是变量的值，而不再是指向 zend_execute_data 上局部变量的地址。此时移动的只是 b.php 中的全局变量，原来 a.php 中定义的、b.php 中没有定义的则不作处理，比如`$var_1`。这个过程由 zend_detach_symbol_table()处理，attach 与 detach 是一对相反的操作：attach 的处理是将 symbol table 中的值复制到 CV 变量；detach 的处理是将 CV 变量的值复制到 symbol table 中。具体的 return 处理：

```
    static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL zend_leave_helper_SPEC
(ZEND_OPCODE_HANDLER_ARGS)
```

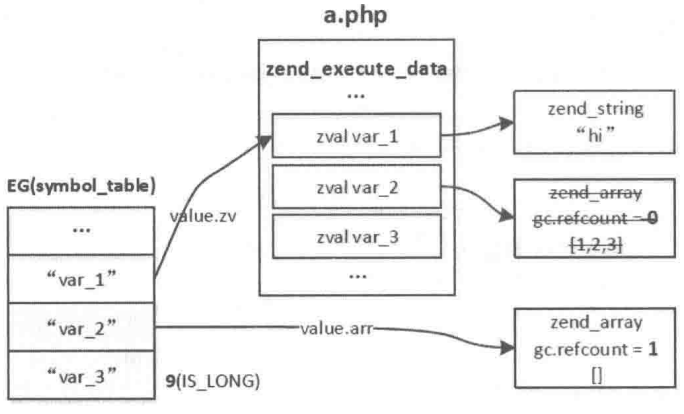
```
{
    ...
    if (EXPECTED((ZEND_CALL_KIND_EX(call_info) & ZEND_CALL_TOP) == 0)) {
        //将 include 文件中定义的变量移到 EG(symbol_table)
        zend_detach_symbol_table(execute_data);
        //释放 zend_op_array
        destroy_op_array(&EX(func)->op_array);

        old_execute_data = execute_data;
        //切回调用文件的 zend_execute_data
        execute_data = EG(current_execute_data) = EX(prev_execute_data);
        //释放 include 文件的 zend_execute_data
        zend_vm_stack_free_call_frame_ex(call_info, old_execute_data);

        //重新 attach
        zend_attach_symbol_table(execute_data);

        LOAD_NEXT_OPLINE();
        ZEND_VM_LEAVE();
    }else{
        //函数、主脚本返回的情况
    }
}
```

zend_detach_symbol_table()操作：

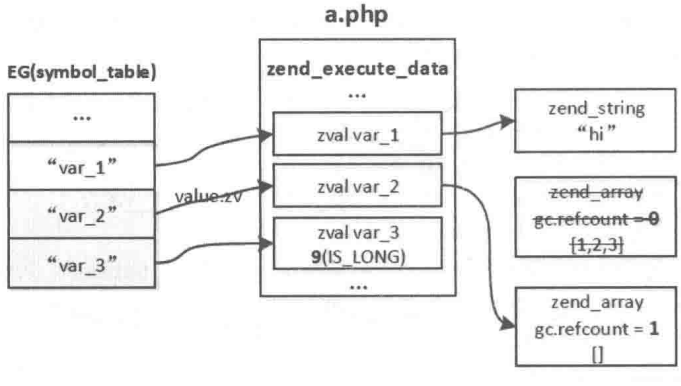
```
ZEND_API void zend_detach_symbol_table(zend_execute_data *execute_data)
{
    zend_op_array *op_array = &execute_data->func->op_array;
    HashTable *ht = execute_data->symbol_table;

    if (EXPECTED(op_array->last_var)) {
        zend_string **str = op_array->vars;
        zend_string **end = str + op_array->last_var;
        zval *var = EX_VAR_NUM(0);
        //将全局变量的值复制至 execute_data->symbol_table
        do {
            if (Z_TYPE_P(var) == IS_UNDEF) {
                zend_hash_del(ht, *str);
            } else {
                zend_hash_update(ht, *str, var);
```

```
                ZVAL_UNDEF(var);
            }
            str++;
            var++;
        } while (str != end);
    }
}
```

return 时，除了还原 a.php 的 zend_execute_data，还会重新执行 zend_attach_symbol_table() 进行全局变量的注册，这个过程的处理与 b.php 在 zend_attach_symbol_table()时逻辑相同。因此，a.php 重新 attach 后，局部变量就继承了 b.php 中修改过的变量的值。b.php 在 detach 后、a.php 在重新 attach 后的变量关系变化如图 9-25 所示。



（b. php在detach后）



（a. php再次attach后）

图 9-25　detach 与 attach 的变量复制

这就是 include 的实现原理，整个过程并不复杂，容易忽视的一个地方就是全局变量的 attach、detach 两个过程，正是通过这两个操作实现了脚本间全局变量的共享。另外，除了 include、require，PHP 中还有两个具有相同用途的关键字：include_once、require_once。它们与 include、require 的区别：在一次请求中同一文件只会被加载一次，第一次加载时会把文件名称插入 EG(included_files)哈希表中，再次加载时检查这个哈希表，如果发现已经加载过则直接跳过。

9.8　异常处理

PHP 的异常处理与其他语言的类似，在程序中可以抛出、捕获一个异常，异常抛出必须只有定义在 try{...}块中才可以被捕获，捕获以后将跳到 catch 块中进行处理，不再执行 try 中抛出异常之后的代码。异常可以在任意位置抛出，然后由最近的一个 try 所捕获，如果在当前执行空间没有进行捕获，那么将调用栈一直往上抛，比如在一个函数内部抛出一个异常，但是函数内没有进行 try，而在函数调用的位置 try 了，那么就由调用处的 catch 捕获。

9.8.1　PHP 中的 try catch

异常捕获及处理的语法：

```
try{
    try statement;
}catch(exception_class_1 $e){
    catch statement 1;
}catch(exception_class_2 $e){
    catch statement 2;
}finally{
    finally statement;
}
```

try 表示要捕获 try statement 中可能抛出的异常；catch 是捕获到异常后的处理，可以定义多个，当 try 中抛出异常时会依次检查各个 catch 的异常类是否与抛出的匹配，如果匹配则由命中的那个 catch 块处理；finally 为最后执行的代码，不管是否有异常抛出都会执行。

try catch 的语法规则：

```
statement:
    ...
    |   T_TRY '{' inner_statement_list '}' catch_list finally_statement
```

```
            { $$ = zend_ast_create(ZEND_AST_TRY, $3, $5, $6); }
    ...
;
catch_list:
        /* empty */
            { $$ = zend_ast_create_list(0, ZEND_AST_CATCH_LIST); }
    |   catch_list T_CATCH '(' name T_VARIABLE ')' '{' inner_statement_list '}'
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AST_CATCH, $4,
$5, $8)); }
;
finally_statement:
        /* empty */ { $$ = NULL; }
    |   T_FINALLY '{' inner_statement_list '}' { $$ = $3; }
;
```

从语法规则可以看出，try-catch-finally 最终编译为一个 ZEND_AST_TRY 节点，该节点有三个子节点：try statement、catch list、finally statement。try statement、finally statement 为 ZEND_AST_STMT_LIST 节点，catch list 包含多个 ZEND_AST_CATCH 节点，每个节点有三个子节点：exception class、exception object 及 catch statement。最终生成的抽象语法树类似图 9-26 的样子。

ZEND_AST_TRY 节点的编译步骤如下所述。

步骤（1）：向所属 zend_op_array 注册一个 zend_try_catch_element 结构，所有 try 都会注册一个这样的结构，这个结构用来记录 try、catch 以及 finally 指令开始的位置，所有的 zend_try_catch_element 结构保存在 zend_op_array->try_catch_array 数组中。

```
typedef struct _zend_try_catch_element {
    uint32_t try_op;     //try 开始的指令位置
    uint32_t catch_op;   //第 1 个 catch 块的指令位置
    uint32_t finally_op; //finally 开始的指令位置
    uint32_t finally_end;//finally 结束的指令位置
} zend_try_catch_element;
```

步骤（2）：编译 try statement，如果定义了 catch 块，则接着编译一条 ZEND_JMP 指令。该指令的作用：当无异常抛出时跳过所有 catch 跳到 finally 或 try catch 后。因为 catch 块是在 try statement 之后编译的，所以具体的跳转值目前还无法确定。

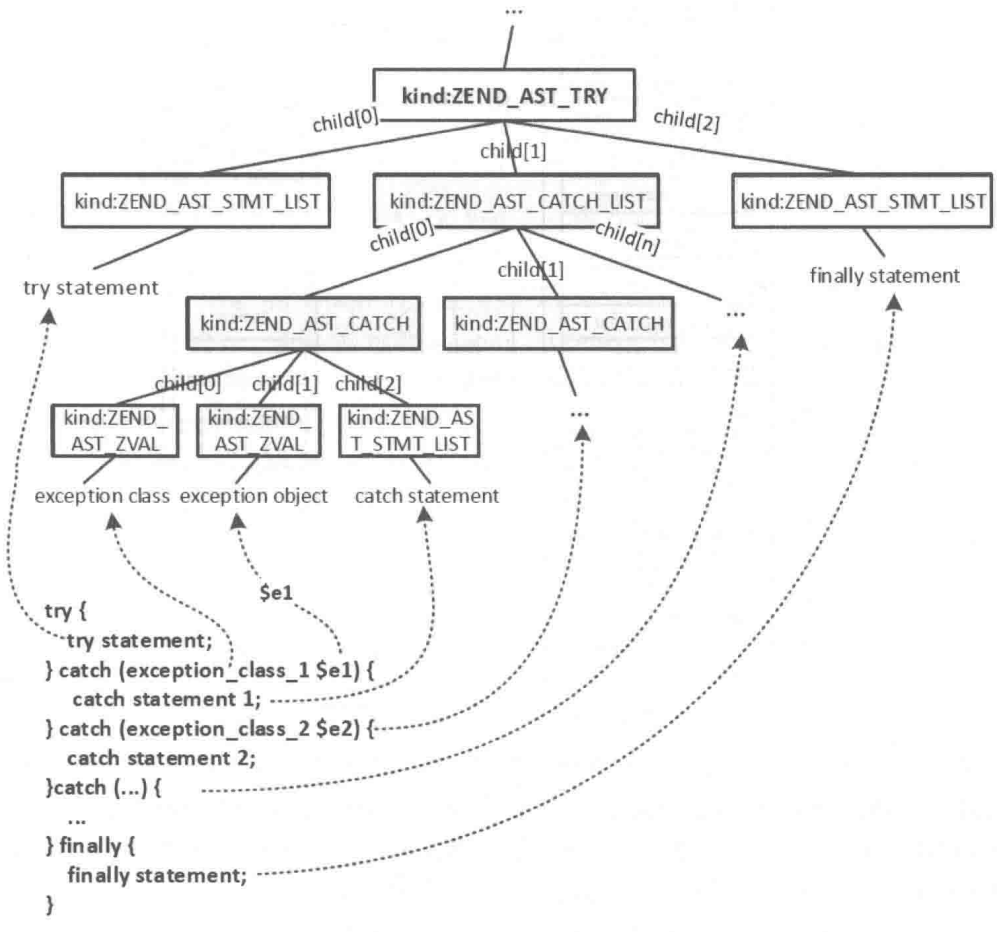


图 9-26　try catch 的抽象语法树

步骤（3）：依次编译各个 catch 块，如果没有定义则跳过此步骤。编译各个 catch 时，首先编译一条 ZEND_CATCH 指令，该指令保存着此 catch 的 exception class、exception object 以及下一个 catch 块开始的位置，编译第 1 个 catch 时，将 ZEND_CATCH 指令的位置更新到 zend_try_catch_element->catch_op，接着编译 catch statement。最后编译一条 ZEND_JMP 指令（最后一个 catch 不需要），执行时如果被这个 catch 捕获异常了，则在执行完 catch statement 后通过 ZEND_JMP 跳到 finally 或 try catch 后。

步骤（4）：更新步骤（2）、步骤（3）中 ZEND_JMP 指令的跳转值。如果没有定义 finally 则结束编译，否则编译 finally 块：首先编译一条 ZEND_FAST_CALL 及 ZEND_JMP 指令，接着编译 finally statement，最后编译一条 ZEND_FAST_RET 指令。

编译完以后的生成的指令结合如图 9-27 所示。

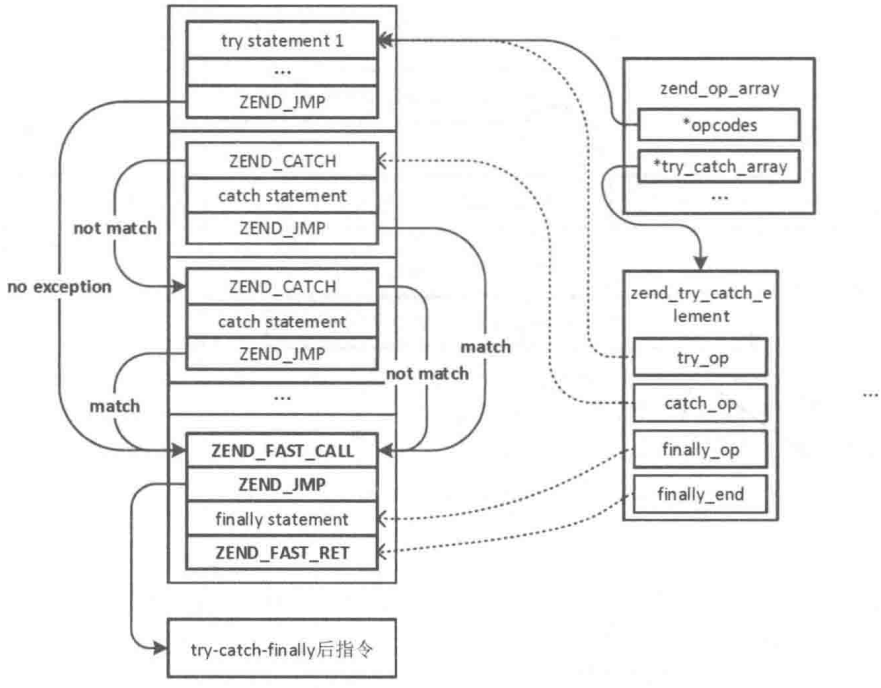


图 9-27　try catch 生成的指令集合

这个步骤生成的三条指令不太好理解，ZEND_FAST_CALL 指令的作用是跳转到 finally statement 指令处，finally statement 指令执行完成后，会通过 ZEND_FAST_RET 指令跳到 ZEND_FAST_CALL 的下一条指令，也就是 ZEND_JMP，这条指令将跳到 try catch 外。ZEND_FAST_CALL 总是跳到 finally statement 处，而 ZEND_FAST_RET 会跳到 ZEND_FAST_CALL 的下一条指令。需要注意的是，ZEND_FAST_CALL 并不仅仅是 finally 中会生成，如果 try statement、catch statement 中有 return 语句，也会生成该指令。也就是说，ZEND_FAST_CALL 的作用是劫持 return，在 return 前先执行 finally statement，执行完后再通过 ZEND_FAST_RET 跳回去执行 return。例如：

```
function test() {
    try {
        $a = "normal return\n";
        return $a;
    } finally {
        echo "finally exec\n";
    }
}
echo test();
```

```
//执行时将输出：
//finally exec
//normal return
```

这个例子在 return 前首先会生成一条 ZEND_FAST_CALL 指令，ZEND_FAST_CALL 将先于 return 执行，它将跳到 `echo "finally exec\n"`处执行，同时会把 ZEND_FAST_CALL 指令的位置保存到一个变量中，执行完 finally statement 后，接着执行 ZEND_FAST_RET，该指令通过 ZEND_FAST_CALL 指令保存的变量得到 ZEND_FAST_CALL 指令的位置，然后跳到 ZEND_FAST_CALL 的下一条指令，也就是 return。这种情况下，finally 中的 ZEND_FAST_CALL、ZEND_JMP 指令就不会被执行了，只有前面没有 return 的情况下，finally 中的这两条指令才会被用到。具体的指令执行顺序如图 9-28 所示。

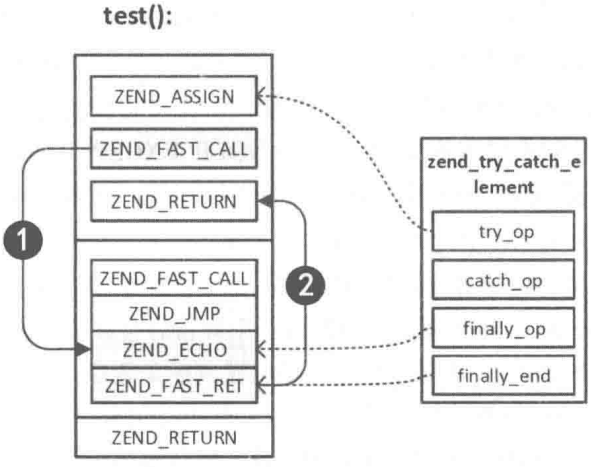


图 9-28　try 中有返回值时 finally 语句的执行

try catch 具体的编译过程为 zend_compile_try()，这里不再展开。了解完 try catch 的编译处理，接下来我们再看一下异常的捕获过程。

异常的抛出通过 throw 关键字来完成：`throw exception_object`，throw 语句被编译为 ZEND_THROW 指令，抛出后将由 ZEND_CATCH 指令检查是否被 catch 捕获。这个过程相对比较复杂，整体的流程如下如下所述。

步骤（1）：检查抛出的是否是 object 对象，如果不是则导致 error 错误。

步骤（2）：将抛出的异常对象保存到 EG(exception)，同时将下一条执行的指令更新为 ZEND_HANDLE_EXCEPTION。

步骤（3）：执行 ZEND_HANDLE_EXCEPTION，首先查找抛出异常的位置是否在某个 try catch 之间，查找的过程就是遍历 zend_op_array->try_catch_array 数组，根据 throw 的位置、try

开始的位置、catch 开始的位置、finally 开始的位置进行比较判断。如果找到了，说明异常是在 try catch 内抛出的，有可能被捕捉到，则进入步骤（4）检查是否被捕获；如果没有找到，则表示当前空间内没有捕获该异常的 try catch，进入步骤（5）处理；如果找到了多个 try catch，则选择最后那个，这种是 try catch 嵌套的情况，首先选择的是最里层的那个。

步骤（4）：当抛出异常的位置有 try catch 进行拦截时，进入该步骤具体判断能否被捕获住。首先跳到第一个 catch 块起始位置（即 zend_try_catch_element->catch_op）执行 ZEND_CATCH 指令，检查抛出的异常对象是否与当前 catch 的类型匹配，也就是检查抛出的异常对象是否是 catch 要拦截的那个类的实例，如果是则表示该异常被当前 catch 成功捕获，接下来就执行该 catch 的 statement，同时将 EG(exception)清空。如果与当前 catch 不匹配，则跳到下一个 catch 的位置继续判断。如果检查到最后一个 catch 仍然不匹配，则表示当前的 try catch 没能捕获住异常，此时会在最后的这个 catch 的位置抛出一个异常，该异常还是原来的那个，只是抛出的位置转移到了另外一个位置，这样做的目的是避免下一轮匹配时重复检查同一个 try catch，更新了异常抛出的位置后，下一轮匹配将不会再命中当前这个没有捕获住的 try catch 了。

步骤（5）：如果在当前空间（比如函数、主代码）无法捕获住异常，则会将异常抛给上一层调用方，在调用方的空间内再次执行 ZEND_HANDLE_EXCEPTION。比如函数中抛出了一个异常，但是函数内没有捕获住，则跳到调用的位置继续捕获，回到步骤（3），如果到最终主脚本也没有被捕获则结束执行并导致 error 错误。

这个过程最复杂的地方在于异常匹配、传递的过程，主要为 ZEND_HANDLE_EXCEPTION、ZEND_CATCH 两条 opcode 之间的调用。当抛出一个异常时会终止后面指令的执行，转向执行 ZEND_HANDLE_EXCEPTION，根据异常抛出的位置判断是否在 try catch 内：如果在则遍历 catch 块检查是否能被捕获，如果被捕获了则执行 catch statement；如果到最后一个 catch 都没有被捕获住，则在最后那个 catch 的位置抛出一个异常，将异常的位置转移，进行下一轮匹配。当所有的 try catch 都没有捕获住异常的时候，将把异常抛给上一层调用方，再次检查，直到被捕获住。这个过程将不断的执行 ZEND_HANDLE_EXCEPTION 及 ZEND_CATCH 指令。

下面根据一个简单的例子具体说明一下：

```
function my_func(){
    //...
    throw new Exception("This is a exception from my_func()");
    echo "hi";
}

try{
    $ret = my_func();
```

```
}catch(ErrorException $e){
    echo "ErrorException";
}catch(Exception $e){
    echo "Exception";
}
```

my_func()中抛出了一个异常，执行到 throw 时将不再执行之后的指令，而是执行 ZEND_HANDLE_EXCEPTION 指令。ZEND_HANDLE_EXCEPTION 首先检查函数 my_func 内是否有 try catch 包裹了该异常，检查后发现没有，然后返回到调用的位置再次执行 ZEND_HANDLE_EXCEPTION，返回时会将调用方接下来要执行的指令修改为 ZEND_HANDLE_EXCEPTION。返回到主代码后，对于主代码而言，异常抛出的位置就是调用 my_func 的位置，接着继续执行 ZEND_HANDLE_EXCEPTION，发现该异常命中了一个 try catch，然后跳到第一个 catch 的位置进行判断，执行 ZEND_CATCH 指令后发现抛出的异常与要捕获的 ErrorException 类型不符，无法进行捕获，跳到下一个 catch 处继续检查，发现与捕获 Exception 的分支匹配，这个时候就捕获成功了，接下来将执行该分支下的指令。具体的执行跳转过程如图 9-29 所示。

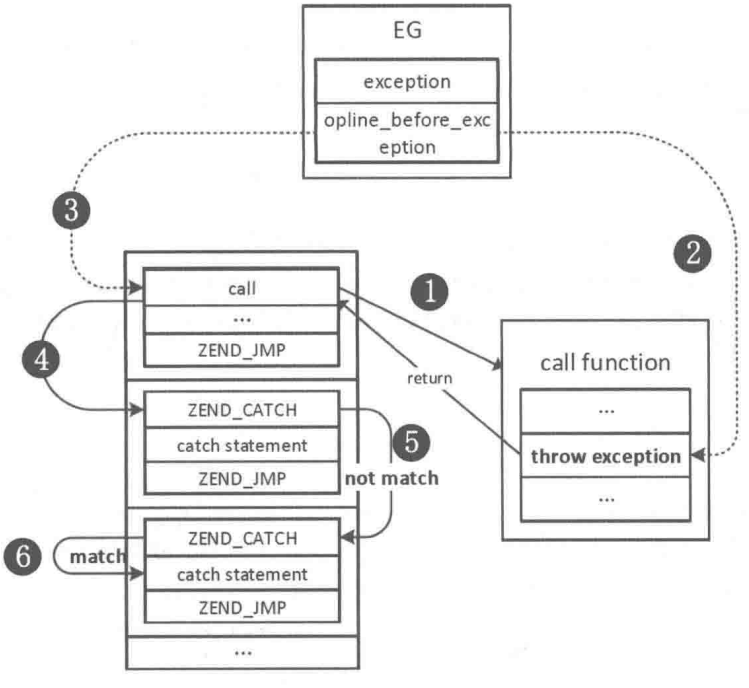


图 9-29　异常的捕获

上面的过程并没有提到 finally 的执行时机，首先要明确 finally 在哪些情况下会执行，命中

catch 的情况比较简单，即在 catch statement 执行完以后跳到 finally 执行，另外一种情况是如果一个异常在 try 中但没有命中任何 catch，那么其 finally 也是会被执行的，这种情况的 finally 实际上是在步骤（3）中执行的，最后一个 catch 检查完以后会更新异常抛出位置 EG(opline_before_exception)，然后会再次执行 ZEND_HANDLE_EXCEPTION，再次检查时就会发现没有命中任何 catch 但命中 finally 了（因为异常位置更新了）。这时候就会将异常对象保存在 finally 块中，然后执行 finally，执行完再将异常对象还原继续捕获。接下来具体看一下处理过程。

```
    ZEND_VM_HANDLER(149, ZEND_HANDLE_EXCEPTION, ANY, ANY)
    {
        //op_num 为异常抛出的位置
        uint32_t op_num = EG(opline_before_exception) - EX(func)->
op_array.opcodes;
        int i;
        uint32_t catch_op_num = 0, finally_op_num = 0, finally_op_end = 0;
        int in_finally = 0;
        ...
        //检查异常是否在 try catch 之间
        for (i = 0; i < EX(func)->op_array.last_try_catch; i++) {
            if (EX(func)->op_array.try_catch_array[i].try_op > op_num) {
                /* further blocks will not be relevant... */
                break;
            }
            in_finally = 0;
            //位于 try catch 之间
            if (op_num < EX(func)->op_array.try_catch_array[i].catch_op) {
                catch_op_num = EX(func)->op_array.try_catch_array[i].catch_op;
            }
            ...
        }
        ...
        if (finally_op_num && (!catch_op_num || catch_op_num >= finally_op_num)) {
            //异常没有在 try catch 之间，但是在 finally 前，这种情况就是当前 try catch 都
没有命中，但是定义了 finally 的情况，此时 finally 是需要执行的
            ...
            //执行 finally statement
            ZEND_VM_SET_OPCODE(&EX(func)->op_array.opcodes[finally_op_num]);
            ZEND_VM_CONTINUE();
```

```
        } else {
            ...
            if (catch_op_num) {
                //检查是否被 catch 住
                ZEND_VM_SET_OPCODE(&EX(func)->op_array.opcodes[catch_op_num]);
                ZEND_VM_CONTINUE();
            } else if (UNEXPECTED((EX(func)->op_array.fn_flags & ZEND_ACC_
GENERATOR) != 0)) {
                ...
            } else {
                //当前空间已经没有 try catch 了，此时需要将异常抛给上一层调用
                ZEND_VM_DISPATCH_TO_HELPER(zend_leave_helper);
            }
        }
    }
```

ZEND_CATCH 检查异常是否被捕获的过程：

```
    ZEND_VM_HANDLER(107, ZEND_CATCH, CONST, CV)
    {
        ...
        //要捕获的异常类型
        catch_ce = zend_fetch_class_by_name(Z_STR_P(EX_CONSTANT(opline->op1)),
EX_CONSTANT(opline->op1) + 1, ZEND_FETCH_CLASS_NO_AUTOLOAD);
        //抛出的异常类型
        ce = EG(exception)->ce;
        ...
        if (ce != catch_ce) {
            //检查异常类型是否匹配
            if (!catch_ce || !instanceof_function(ce, catch_ce)) {
                if (opline->result.num) { //最后一个 catch
                    //在当前位置抛出一个异常，这个时候就是把异常的抛出位置更新成了这个
ZEND_CATCH
                    zend_throw_exception_internal(NULL);
                    HANDLE_EXCEPTION();
                }
                //不是最后一个 catch：跳到下一个 catch 处继续匹配
```

```
ZEND_VM_SET_OPCODE(&EX(func)->op_array.opcodes[opline->extended_value]);
                ZEND_VM_CONTINUE(); /* CHECK_ME */
            }
        }
        //捕获成功
        ...
    }
```

具体的实现过程还有很多额外的处理，这里不再展开，感兴趣的读者可以详细研究一下ZEND_HANDLE_EXCEPTION、ZEND_CATCH 两条指令以及 zend_exception.c 中的具体逻辑。

9.8.2 内核中的异常处理

上一节介绍的异常处理是 PHP 语言层面的实现，在内核中也有一套供内核使用的异常处理模型，也就是 C 语言异常处理的实现，这是内核自己使用的。在内核中经常能够看到下面这样的语句：

```
    zend_try {
        ...
    } zend_catch {
        ...
    } zend_end_try();
```

zend_try、zend_catch、zend_en_try 就是 PHP 内核自己的异常处理操作。C 语言并没有在语言层面提供 try catch 机制，那么 PHP 内核是如何实现的呢？实际上这个功能非常简单，这个主要利用 sigsetjmp()、siglongjmp()两个函数实现堆栈的保存、还原，在 try 的位置通过 sigsetjmp()将当前位置的堆栈保存在一个变量中，异常抛出通过 siglongjmp()跳回原位置。具体看一下这几个宏的定义：

```
#define zend_try                                                \
    {                                                           \
        JMP_BUF *__orig_bailout = EG(bailout);                  \
        JMP_BUF __bailout;                                      \
                                                                \
        EG(bailout) = &__bailout;                               \
        if (SETJMP(__bailout)==0) {
#define zend_catch                                              \
```

```
        } else {                                                    \
            EG(bailout) = __orig_bailout;
#define zend_end_try()                                              \
        }                                                           \
        EG(bailout) = __orig_bailout;                               \
    }


# define JMP_BUF sigjmp_buf
# define SETJMP(a) sigsetjmp(a, 0)
# define LONGJMP(a,b) siglongjmp(a, b)
# define JMP_BUF sigjmp_buf
```

展开后：

```
{
    //保存上一个 zend_try 记录的 JMP_BUF，目的是实现多层嵌套 try
    JMP_BUF *__orig_bailout = EG(bailout);
    JMP_BUF __bailout;

    //将当前堆栈保存在__bailout
    EG(bailout) = &__bailout;
    if (SETJMP(__bailout)==0) {
        //try 中的代码
        ...
        //抛出异常调用 LONGJMP()
    } else { //异常抛出后到这个分支
        EG(bailout) = __orig_bailout;
    }
    EG(bailout) = __orig_bailout;
}
```

PHP 中 exit 语句就是利用这个功能实现的退出，exit 对应的执行指令为 ZEND_EXIT：

```
ZEND_VM_HANDLER(79, ZEND_EXIT, CONST|TMPVAR|UNUSED|CV, ANY)
{
    USE_OPLINE
    ...
```

```
    zend_bailout();
    ZEND_VM_NEXT_OPCODE(); /* Never reached */
}
```

zend_bailout()宏最终就是调用的 LONGJMP()，PHP 脚本中执行 exit 时相当于内核抛出了一个异常，从而退出了执行器。

```
#define zend_bailout()      _zend_bailout(__FILE__, __LINE__)
//file: zend.c
BEGIN_EXTERN_C()
ZEND_API ZEND_COLD void _zend_bailout(char *filename, uint lineno) /* {{{ */
{
    ...
    CG(unclean_shutdown) = 1;
    CG(active_class_entry) = NULL;
    CG(in_compilation) = 0;
    EG(current_execute_data) = NULL;
    LONGJMP(*EG(bailout), FAILURE);
}
```

PHP 脚本最初执行的位置：

```
PHPAPI int php_execute_script(zend_file_handle *primary_file)
{
    ...
    zend_try {
        ...
        retval = (zend_execute_scripts(ZEND_REQUIRE, NULL, 3, prepend_file_p,
primary_file, append_file_p) == SUCCESS);
    } zend_end_try(); //执行中，如果有 exit，则直接跳到这个位置
    ...
}
```

9.9 break/continue LABEL 语法的实现

PHP 中的 break、continue 语句只能根据数字指定操作的循环，循环嵌套比较多的时候维护起来很不方便，需要一层层地去数要操作的循环，例如：

```
//loop1
while(...){
    //loop2
    for(...){
        //loop3
        foreach(...){
            ...
            break 2;
        }

    }
    //loop2 end
    ...
}
```

break 2 表示要中断往上数两层也就是 loop2 这层循环，break 2 之后将从 loop2 end 开始继续执行。

在 Go 语言中，break、continue 有一种比较优雅的语法——可以按照标签指定操作的循环，举个例子来看：

```
func main() {
loop1:
    for i := 0; i < 2; i++ {
        fmt.Println("loop1")
        for j := 0; j < 5; j++ {
            fmt.Println("  loop2")
            if j == 2 {
                break loop1
            }
        }
    }
}
```

break loop1 这种语法在 PHP 中是不支持的，本节我们将介绍如何让 PHP 实现同样的功能。首先需要明确 PHP 中循环、中断以及标签语法的实现，本章之前已经详细介绍过了，不熟悉的读者可以翻回去重新看一下，这里只说两点与本节相关的地方：

- 不管是哪种循环结构，其编译时都生成了一个 zend_brk_cont_element 结构，此结构记录着这个循环 break、continue 要跳转的位置，以及嵌套的父层循环。
- break/continue 编译时分为两个步骤：首先初步编译为临时指令，该指令记录着 break/continue 所在循环层以及要中断的层级（即 break n，默认 n=1）；然后在脚本全部编译完之后的 pass_two()中，根据当前循环层及中断的层级 n 向上查找对应的循环层，最后根据查找到的要中断的循环 zend_brk_cont_element 结构得到对应的跳转位置，生成一条 ZEND_JMP 指令。

仔细研究循环、中断的实现可以发现，这里面的关键就在于找到 break/continue 要中断的那层循环，嵌套循环之间是链表的结构，目前“break + 数字”的处理就是直接从 break/continue 当前循环层向前查找对应的层数，从而找到要操作的那层循环。

标签在内核中通过 HashTable 的结构保存（即 CG(context).labels），key 就是标签名，标签会记录当前指令的位置，我们要实现 break 标签的语法需要根据标签取到循环，因此我们为标签赋予一种新的含义：循环标签。只有标签紧挨着循环的才认为是这种含义，比如：

```
loop1:
for(...){
    break loop1;
}
```

如果标签与循环之间有其他表达式，则认为是普通标签，这种标签是不能进行 break、continue 的：

```
loop1:
$a = 123;
for(...){
    break loop1; //不合法的
}
```

既然要按照标签进行 break、continue，那么很容易想到把中断的循环层级 id 保存到标签中，编译 break/continue 时先查找标签，再查找循环的 zend_brk_cont_element 即可。这样实现的话需要在循环编译时将自己 zend_brk_cont_element 的存储位置保存到标签中，标签的结构需要修改，另外一个问题是标签不会生成任何指令，循环结构无法直接根据上一条指令判断它是不是循环标签 ，所以我们换一种方式实现，具体思路如下所述。

- 循环结构开始编译前先编译一条空指令：ZEND_NOP，用于标识这是一个循环，并把这个循环 zend_brk_cont_element 的存储位置记录在该指令中。

- break 编译时，如果发现是一个标签，则从 CG(context).labels)中取出标签结构，然后判断此标签的下一条指令是否为 ZEND_NOP，如果不是则说明这不是一个循环标签，无法 break、continue，如果是则取出循环结构。
- 得到循环结构之后的处理就比较简单了，但是此时还不能直接编译为 ZEND_JMP，因为循环可能还未编译完成，break 只能编译为临时指令，这里可以把标签标记的循环存储位置记录在临时指令中，然后在 pass_two()中再重新获取，需要对 pass_two()中的逻辑进行改动。为了减少改动，这个地方转化一下实现方式：计算 label 标记的循环相对 break 所在循环的位置，也就是转为现有的 break n，这样一来就无须对 pass_two()进行改动了。

接下来看一下具体的实现，以 for 循环结构为例。

1. 编译循环语句

```
void zend_compile_for(zend_ast *ast)
{
    zend_ast *init_ast = ast->child[0];
    zend_ast *cond_ast = ast->child[1];
    zend_ast *loop_ast = ast->child[2];
    zend_ast *stmt_ast = ast->child[3];

    znode result;
    uint32_t opnum_start, opnum_jmp, opnum_loop;
    zend_op *mark_look_opline;

    //新增：创建一条空 opcode，用于标识接下来的是一个循环结构
    mark_look_opline = zend_emit_op(NULL, ZEND_NOP, NULL, NULL);

    zend_compile_expr_list(&result, init_ast);
    zend_do_free(&result);

    opnum_jmp = zend_emit_jump(0);

    zend_begin_loop(ZEND_NOP, NULL);
    //新增：保存当前循环的 brk，同时为了防止与其他 ZEND_NOP 混淆，把 op1 标为-1
    mark_look_opline->op1.var = -1;
    mark_look_opline->extended_value = CG(context).current_brk_cont;
    ...
}
```

2. 编译中断语句

“break label”将被编译为如图 9-30 所示的语法树节点。

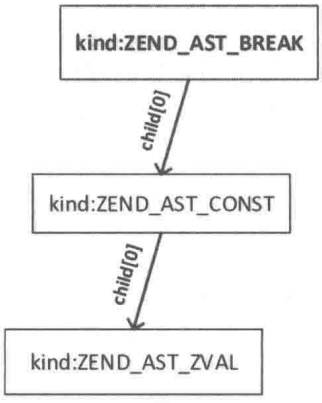


图 9-30

ZEND_AST_BREAK 只有一个子节点，如果是数值，那么这个子节点类型为 ZEND_AST_ZVAL，如果是标签则类型是 ZEND_AST_CONST，ZEND_AST_CONST 也有一个类型为 ZEND_AST_ZVAL 子节点。下面看一下 break/continue 修改后的编译逻辑：

```
void zend_compile_break_continue(zend_ast *ast)
{
    zend_ast *depth_ast = ast->child[0];

    zend_op *opline;
    int depth;

    ZEND_ASSERT(ast->kind == ZEND_AST_BREAK || ast->kind == ZEND_AST_
CONTINUE);

    if (CG(context).current_brk_cont == -1) {
        zend_error_noreturn(E_COMPILE_ERROR, "'%s' not in the 'loop' or
'switch' context",
            ast->kind == ZEND_AST_BREAK ? "break" : "continue");
    }

    if (depth_ast) {
```

```
        switch(depth_ast->kind){
            case ZEND_AST_ZVAL: //break 数值;
                {
                    zval *depth_zv;

                    depth_zv = zend_ast_get_zval(depth_ast);
                    if (Z_TYPE_P(depth_zv) != IS_LONG || Z_LVAL_P(depth_zv) < 1) {
                        zend_error_noreturn(E_COMPILE_ERROR, "'%s' operator
accepts only positive numbers",
                            ast->kind == ZEND_AST_BREAK ? "break" : "continue");
                    }

                    depth = Z_LVAL_P(depth_zv);
                    break;
                }
            case ZEND_AST_CONST://break 标签;
                {
                    //获取 label 名称
                    zend_string *label = zend_ast_get_str(depth_ast->
child[0]);
                    //根据 label 获取标记的循环，以及相对 break 所在循环的位置
                    depth = zend_loop_get_depth_by_label(label);
                    if(depth > 0){
                        goto SET_OP;
                    }
                    break;
                }
            default:
                zend_error_noreturn(E_COMPILE_ERROR, "'%s' operator with
non-constant operand "
                    "is no longer supported", ast->kind == ZEND_AST_BREAK ?
"break" : "continue");
        }
    } else {
        depth = 1;
    }

    if (!zend_handle_loops_and_finally_ex(depth)) {
        zend_error_noreturn(E_COMPILE_ERROR, "Cannot '%s' %d level%s",
```

```
                ast->kind == ZEND_AST_BREAK ? "break" : "continue",
                depth, depth == 1 ? "" : "s");
        }

    SET_OP:
        opline = zend_emit_op(NULL, ast->kind == ZEND_AST_BREAK ? ZEND_BRK :
ZEND_CONT, NULL, NULL);
        opline->op1.num = CG(context).current_brk_cont;
        opline->op2.num = depth;
    }
```

zend_loop_get_depth_by_label()这个函数用来计算标签标记的循环相对 break/continue 所在循环的层级：

```
    int zend_loop_get_depth_by_label(zend_string *label_name)
    {
        zval *label_zv;
        zend_label *label;
        zend_op *next_opline;

        if(UNEXPECTED(CG(context).labels == NULL)){
            zend_error_noreturn(E_COMPILE_ERROR, "can't find label:'%s' or it
not mark a loop", ZSTR_VAL(label_name));
        }

        // 1) 查找 label
        label_zv = zend_hash_find(CG(context).labels, label_name);
        if(UNEXPECTED(label_zv == NULL)){
            zend_error_noreturn(E_COMPILE_ERROR, "can't find label:'%s' or it
not mark a loop", ZSTR_VAL(label_name));
        }

        label = (zend_label *)Z_PTR_P(label_zv);

        // 2) 获取 label 下一条 opcode
        next_opline = &(CG(active_op_array)->opcodes[label->opline_num]);
        if(UNEXPECTED(next_opline == NULL)){
            zend_error_noreturn(E_COMPILE_ERROR, "can't find label:'%s' or it
```

```
not mark a loop", ZSTR_VAL(label_name));
        }

        int label_brk_offset, curr_brk_offset; //标签标识的循环、break 当前所在循环
        int depth = 0; //break 当前循环至标签循环的层级
        zend_brk_cont_element *brk_cont_element;

        if(next_opline->opcode == ZEND_NOP && next_opline->op1.var == -1){
            label_brk_offset = next_opline->extended_value;
            curr_brk_offset = CG(context).current_brk_cont;

            brk_cont_element = &(CG(active_op_array)->brk_cont_array[curr_
brk_offset]);
            //计算标签标记的循环相对位置
            while(1){
                depth++;

                if(label_brk_offset == curr_brk_offset){
                    return depth;
                }

                curr_brk_offset = brk_cont_element->parent;
                if(curr_brk_offset < 0){
                    //label 标识的不是 break 所在循环
                    zend_error_noreturn(E_COMPILE_ERROR, "can't break/conitnue
label:'%s' because it not mark a loop", ZSTR_VAL(label_name));
                }
            }
        }else{
            //label 没有标识一个循环
            zend_error_noreturn(E_COMPILE_ERROR, "can't break/conitnue label:'%s'
because it not mark a loop", ZSTR_VAL(label_name));
        }

        return -1;
    }
```

改动后重新编译 PHP，然后测试新的语法是否生效：

```
//test.php

loop1:
for($i = 0; $i < 2; $i++){
    echo "loop1\n";

    for($j = 0; $j < 5; $j++){
        echo "  loop2\n";
        if($j == 2){
            break loop1;
        }
    }
}

$ php test.php
loop1
  loop2
  loop2
  loop2
```

可以看到，“break loop1”已经能够正常工作了，其他几个循环结构的改动与 for 相同，有兴趣的读者可以自己去尝试一下。

9.10 小结

这一章我们介绍了 PHP 中几种常见语法的实现，这些语法是 PHP 语言最基础的功能，了解这些语法的实现能够帮助我们更好地理解 PHP，同时还可以根据这些已有的语法实现更多的语言特性或者对它们进行改进。

PHP 出现的 Bug 中，很多都是来自这些基础语法，官方的 Bug 列表：https://bugs.php.net。你可以从中找些相关的 Bug，自己尝试去修复，通过这种方式来熟悉 PHP，逐步参与到 PHP 的开发中去。

第 10 章 扩展开发

扩展是 PHP 的重要组成部分，它是 PHP 提供给开发者用于扩展 PHP 语言功能的主要方式，开发者可以使用 C/C++实现自定义的功能，通过扩展嵌入到 PHP 中。灵活的扩展能力使得 PHP 拥有了大量、丰富的第三方组件，这些扩展很好地补充了 PHP 的功能、特性，使得 PHP 在 Web 开发中得以大展身手。扩展可以通过以下几个方面对 PHP 进行扩展。

- 介入 PHP 的编译、执行阶段：可以介入 PHP 框架执行的那 5 个阶段，比如 opcache，就是重定义了编译函数。
- 提供内部函数：可以定义内部函数扩充 PHP 的函数功能，比如 array、date 等操作。
- 提供内部类。
- 实现 RPC 客户端：实现与外部服务的交互，比如 Redis、MySQL 等。
- 提升执行性能：PHP 是解释型语言，在性能方面远不及 C 语言，可以将耗 CPU 的操作用 C 语言代替。

当然扩展可发挥作用的地方并不仅仅这几个，还有很多其他方面。本章将重点介绍扩展开发中常用的一些知识。

10.1 扩展的内部实现

PHP 中扩展通过 zend_module_entry 这个结构来表示，此结构用于保存扩展的基本信息，包

括扩展名、扩展版本、扩展提供的函数列表，以及 PHP 四个执行阶段的 hook 函数等。每一个扩展都需要定义一个此结构的变量，而且这个变量的名称格式必须是{mudule_name}_module_entry，内核正是通过这个结构获取到扩展提供的功能的。

PHP 中的扩展分为两类：PHP 扩展、Zend 扩展，对内核而言这两个分别称之为模块（module）、扩展（extension），本章主要介绍 PHP 扩展，也就是模块。

扩展可以在编译 PHP 时一起编译（即静态编译），也可以单独编译为共享库，共享库的形式需要将库的名称加入到 php.ini 配置中去，在 php_module_startup()阶段 PHP 会根据 php.ini 的配置将对应的扩展共享库加载到 PHP 中。

```
int php_module_startup(sapi_module_struct *sf, zend_module_entry *additional_
modules, uint num_additional_modules)
{
    ...
    //根据 php.ini 注册扩展
    php_ini_register_extensions();
    zend_startup_modules();

    zend_startup_extensions();
    ...
}
```

PHP 解析 php.ini 时，会把配置中的 extension=xxx.so、zend_extension=xxx.so 解析到全局变量 extension_lists，这个变量是一个 php_extension_lists 结构，它的两个成员为类型均为 zend_llist，分别用来保存 php.ini 中配置的 PHP 扩展、Zend 扩展的 so 名称。

```
typedef struct _php_extension_lists {
    zend_llist engine;
    zend_llist functions;
} php_extension_lists;
```

比如在 php.ini 中配置了两个扩展，则 php.ini 解析完成后 extension_lists 结构如图 10-1 所示。

```
extension=mytest1.so
extension=mytest2.so
```

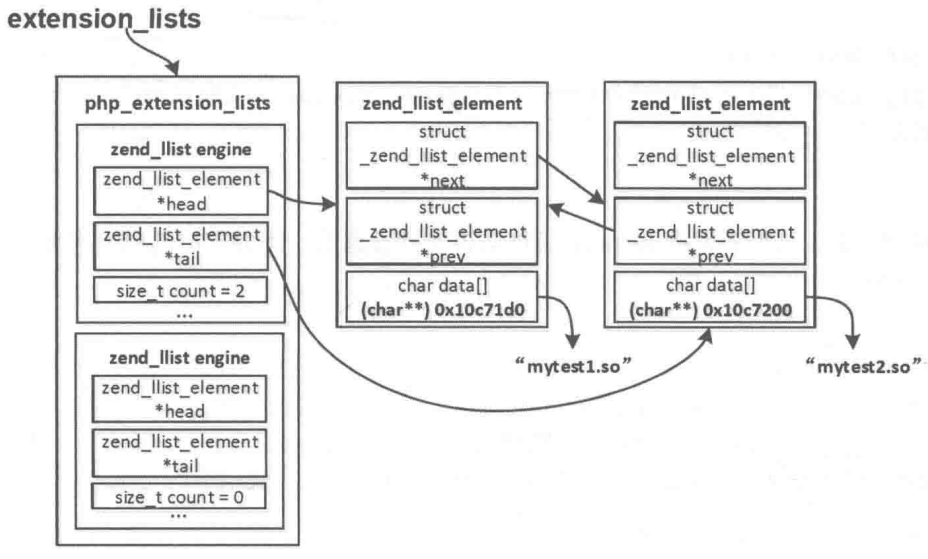


图 10-1　extension_lists 结构

php_ini_register_extensions() 中将依次遍历 php_extension_lists.engine、php_extension_lists.functions，然后分别调用 php_load_zend_extension_cb()、php_load_php_extension_cb()完成 PHP 扩展、Zend 扩展的加载。

```
    //file:main/php_ini.c
    void php_ini_register_extensions(void)
    {
        //注册 zend 扩展
        zend_llist_apply(&extension_lists.engine,
php_load_zend_extension_cb);
        //注册 php 扩展
        zend_llist_apply(&extension_lists.functions,
php_load_php_extension_cb);

        zend_llist_destroy(&extension_lists.engine);
        zend_llist_destroy(&extension_lists.functions);
    }
```

PHP 扩展将调用 php_load_php_extension_cb()进行注册，输入的参数是扩展名称的地址。

```
    //file:main/php_ini.c
    static void php_load_php_extension_cb(void *arg)
```

```
{
#ifdef HAVE_LIBDL
    php_load_extension(*((char **) arg), MODULE_PERSISTENT, 0);
#endif
}
```

HAVE_LIBDL 这个宏根据 dlopen()函数是否支持设置的，因为扩展的共享库就是通过这个函数打开加载的。

```
//file: Zend/Zend.m4
AC_DEFUN([LIBZEND_LIBDL_CHECKS],[
AC_CHECK_LIB(dl, dlopen, [LIBS="-ldl $LIBS"])
AC_CHECK_FUNC(dlopen,[AC_DEFINE(HAVE_LIBDL, 1,[ ])])
])
```

接着就是最关键的操作了，由 php_load_extension()最终完成扩展的注册，该函数定义在 standard 扩展中，具体步骤如下所述。

- dlopen()打开扩展的共享库文件。
- dlsym()获取动态库中 get_module()函数的地址，get_module()是每个扩展都必须提供的一个接口，用于返回扩展 zend_module_entry 结构的地址。
- 调用扩展的 get_module()方法，该方法将返回扩展定义的 zend_module_entry 结构。
- 检查扩展版本是否支持，比如 PHP7 的扩展在 PHP5 下是无法使用的。
- 注册扩展，将扩展添加到 module_registry 中，这是一个全局 HashTable，用于全部扩展的 zend_module_entry 结构。
- 如果扩展提供了内部函数，则将这些函数注册到 EG(function_table)符号表中。

```
//file: ext/standard/dl.c
PHPAPI int php_load_extension(char *filename, int type, int start_now)
{
    void *handle;
    char *libpath;
    zend_module_entry *module_entry;
    zend_module_entry *(*get_module)(void);
    ...
    //调用 dlopen 打开指定的动态连接库文件：xx.so，DL_LOAD()、DL_FETCH_SYMBOL()
这两个宏在 Linux 下展开后就是：dlopen()、dlsym()
```

```
        handle = DL_LOAD(libpath);
        ...
        //调用 dlsym 获取 get_module 的函数指针
        get_module = (zend_module_entry *(*)(void)) DL_FETCH_SYMBOL(handle,
"get_module");
        ...
        //调用扩展的 get_module()函数
        module_entry = get_module();
        ...
        //检查扩展使用的 zend API 是否与当前 PHP 版本一致
        if (module_entry->zend_api != ZEND_MODULE_API_NO) {
            DL_UNLOAD(handle);
            return FAILURE;
        }
        ...
        module_entry->type = type;
        //为扩展编号
        module_entry->module_number = zend_next_free_module();
        module_entry->handle = handle;
        //注册扩展
        if ((module_entry = zend_register_module_ex(module_entry)) == NULL) {
            DL_UNLOAD(handle);
            return FAILURE;
        }
        ...
    }
```

完成扩展的注册后，PHP 将在不同的执行阶段，依次调用每个扩展注册的当前阶段的 hook 函数，具体扩展可定义的 hook 函数稍后再介绍。

10.2 扩展的构成及编译

PHP 扩展通过 zend_module_entry 结构定义扩展相关的信息，包括扩展名称、版本号、各阶段 hook 函数指针、注册的内部函数等，上一节已经介绍扩展的加载过程，内核正是根据这个结构得到扩展的基本信息，并完成扩展的注册。该结构作为全局变量分配即可，名称的格式必须是 extensionname_module_entry。

```
//zend_modules.h
struct _zend_module_entry {
    unsigned short size; //sizeof(zend_module_entry)
    unsigned int zend_api; //ZEND_MODULE_API_NO
    unsigned char zend_debug; //是否开启 debug
    unsigned char zts; //是否开启线程安全
    const struct _zend_ini_entry *ini_entry;
    const struct _zend_module_dep *deps;
    const char *name; //扩展名称，不能重复
    //扩展提供的内部函数列表
    const struct _zend_function_entry *functions;
     //PHP 模块初始化时回调函数，PHP_MINIT_FUNCTION 或 ZEND_MINIT_FUNCTION 定义的
函数
    int (*module_startup_func)(INIT_FUNC_ARGS);
     //PHP 模块关闭时回调函数
    int (*module_shutdown_func)(SHUTDOWN_FUNC_ARGS);
    //请求开始前回调函数
    int (*request_startup_func)(INIT_FUNC_ARGS);
     //请求结束时回调函数
    int (*request_shutdown_func)(SHUTDOWN_FUNC_ARGS);
    //php_info 展示的扩展信息处理函数
    void (*info_func)(ZEND_MODULE_INFO_FUNC_ARGS);
    const char *version; //版本
    ...
    unsigned char type;
    void *handle;
    int module_number; //扩展的唯一编号
    const char *build_id;
};
```

这个结构包含很多成员，但并不是所有的都需要一一定义，常用的需要定义的主要有下面几个。

- **name**：扩展名称，不能与其他扩展冲突。
- **functions**：扩展定义的内部函数 entry。
- **module_startup_func**：PHP 在模块初始化时回调的 hook 函数，可以使扩展介入 module startup 阶段。

- **module_shutdown_func**：在模块关闭阶段回调的函数。
- **request_startup_func**：在请求初始化阶段回调的函数。
- **request_shutdown_func**：在请求结束阶段回调的函数。
- **info_func**：php_info()函数时调用，用于展示一些配置、运行信息。
- **version**：扩展版本。

除了上面这些需要手动设置的成员，剩余的可以通过 STANDARD_MODULE_HEADER、STANDARD_MODULE_PROPERTIES 两个宏分别在前后进行填充。例如：

```
#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"

zend_module_entry mytest_module_entry = {
    STANDARD_MODULE_HEADER,
    "mytest",
    NULL, //mytest_functions,
    NULL, //PHP_MINIT(mytest),
    NULL, //PHP_MSHUTDOWN(mytest),
    NULL, //PHP_RINIT(mytest),
    NULL, //PHP_RSHUTDOWN(mytest),
    NULL, //PHP_MINFO(mytest),
    "1.0.0",
    STANDARD_MODULE_PROPERTIES
};
```

定义完 zend_module_entry 结构，还需要提供一个用来获取该结构地址的接口给内核，这个接口是统一的，并不需要单独定义，在扩展中加入 ZEND_GET_MODULE(extension_name)这个宏即可。

```
//file: Zend/zend_API.h
#define ZEND_GET_MODULE(name) \
    BEGIN_EXTERN_C()\
    ZEND_DLEXPORT zend_module_entry *get_module(void) { return &name##_
module_entry; }\
    END_EXTERN_C()
```

展开后可以看到，这个宏定义了一个 get_module()函数，返回扩展 zend_module_entry 结构的地址，这就是为什么扩展的 zend_module_entry 结构变量名必须是 extensionname_module_entry 这种格式的原因。get_module()就是在上一节介绍的 php_load_extension()注册时调用的。

定义完 zend_module_entry 结构、ZEND_GET_MODULE()两部分，一个扩展就编写完成了。以上面 mytest 这个扩展为例，在源码中加入 ZEND_GET_MODULE(mytest)，然后编译、安装，执行 php -m 就可以看到 mytest 这个扩展了，只不过这个扩展没有提供任何功能而已。

```
zend_module_entry mytest_module_entry = {
    ...
}
ZEND_GET_MODULE(mytest)
```

10.2.1 脚本工具

明确了扩展的内部实现和基本构成后，接下来再来介绍 PHP 提供的几个用于简化扩展开发的脚本工具：ext_skel、phpize、php-config。这些工具主要用于扩展的生成及编译，掌握这些工具的使用是扩展开发的基本要求，同时也需要了解这些工具背后的实际工作及具体实现，而不仅仅局限于使用。

在介绍这几个工具之前，先看一下 PHP 安装后的目录结构，很多脚本、配置都放置在安装后的目录中，假设 PHP 的安装路径为：/usr/local/php7，则此目录的主要结构：

```
|---/usr/local/PHP7
|   |---bin //Php 编译生成的二进制程序目录
|       |---php  //Cli 模式的 PHP
|       |---phpize
|       |---php-config
|       |---...
|   |---etc      //一些 SAPI 的配置
|   |---include //PHP 源码的头文件
|       |---php
|           |---main    //PHP 中的头文件
|           |---Zend    //Zend 头文件
|           |---TSRM    //TSRM 头文件
|           |---ext     //扩展头文件
|           |---sapi    //SAPI 头文件
|           |---include
```

```
|   |---lib //依赖的so库
|       |---php
|           |---extensions  //扩展so保存目录
|           |---build       //编译时的工具、m4配置等，编写扩展是会用到
|               |---acinclude.m4    //PHP自定义的autoconf宏
|               |---libtool.m4      //libtool定义的autoconf宏，acinclude.m4、
libtool.m4会被合成aclocal.m4
|               |---phpize.m4       //PHP核心configure.in配置
|               |---...
|           |---...
|   |---php
|   |---sbin //SAPI编译生成的二进制程序，php-fpm会放在这里
|   |---var  //log、run日志
```

10.2.1.1 ext_skel

这个脚本位于PHP源码/ext目录下，它的作用是用来生成扩展的基本骨架，帮助开发者快速生成一个规范的扩展结构，可以通过以下命令生成一个扩展结构：

```
$ ./ext_skel --extname=扩展名称
```

执行完以后会在ext目录下新生成一个扩展目录，比如extname是mytest，则将生成以下文件：

```
|---mytest
|   |---config.m4        //autoconf规则的编译配置文件
|   |---config.w32       //windows环境的配置
|   |---CREDITS
|   |---EXPERIMENTAL
|   |---include          //依赖库的include头文件，可以不用
|   |---mytest.c         //扩展源码
|   |---php_mytest.h     //头文件
|   |---mytest.php       //用于在PHP中测试扩展是否可用，可以不用
|   |---tests            //测试用例，执行make test时将执行、验证这些用例
|       |---001.phpt
```

这个脚本主要生成了编译需要的配置及扩展的基本结构，初步生成的这个扩展可以成功编译、安装与使用，实际开发中我们可以使用这个脚本生成一个基本结构，然后根据具体的需要逐步完善。

10.2.1.2 php-config

这个脚本为 PHP 源码中的/script/php-config.in，PHP 安装后被移到安装路径的/bin 目录下，并重命名为 php-config，这个脚本主要是获取 PHP 的安装信息的，编译安装 PHP 时，会把以下信息保存到 php-config 脚本中。

- PHP 安装路径。
- PHP 版本。
- PHP 源码的头文件目录：main、Zend、ext、TSRM 中的头文件，编写扩展时会用到这些头文件，这些头文件保存在 PHP 安装位置/include/php 目录下。
- LDFLAGS：外部库路径，比如-L/usr/bib -L/usr/local/lib。
- 依赖的外部库：告诉编译器要链接哪些文件，-lcrypt -lresolv –lcrypt，等等。
- 扩展存放目录：扩展.so 保存位置，安装扩展 make install 时将安装到此路径下。
- 编译的 SAPI：如 Cli、Fpm、Cgi 等。
- PHP 编译参数：编译 PHP 时执行./configure 时带的编译参数。

这些信息在编译 PHP 扩展时会用到，扩展编译执行./configure --with-php-config=xxx 生成 Makefile 时作为参数传入即可，它的作用是提供给 configure.in 获取上面几个配置生成 Makefile。如果没有指定这个参数，将到默认的 PHP 安装路径下搜索，因此，如果在同一台机器安装了不同版本的 PHP，在编译扩展时就需要指定对应版本的 php-config，否则将造成扩展无法编译。

10.2.1.3 phpize

phpize 是扩展编译必用的一个脚本，主要是操作复杂的 autoconf/automake/autoheader/autolocal 等系列命令，用于生成扩展的 configure 文件。整个过程相对比较烦琐，幸运的是并不需要我们手动完成每个步骤，phpize 脚本帮我们完成了绝大多数的工作，我们只需要修改很少的配置即可。

简单的项目我们通常手动编写 Makefile，但是大型软件需要适应不同的系统，编译配置也复杂很多，很难再手动编写，autoconf、automake 就是用于生成可以自动地配置软件源代码包以适应多种 UNIX 类系统的 shell 脚本的工具。GNU auto 系列的工具众多，需要配合使用，经过多个步骤完成，具体步骤如图 10-2 所示。

下面简单介绍不同工具的使用顺序及作用。

（1）autoscan：在源码目录下扫描，生成 configure.scan，然后把这个文件重名为 configure.in，可以在这个文件里对依赖的文件、库进行检查，以及配置一些编译参数等。

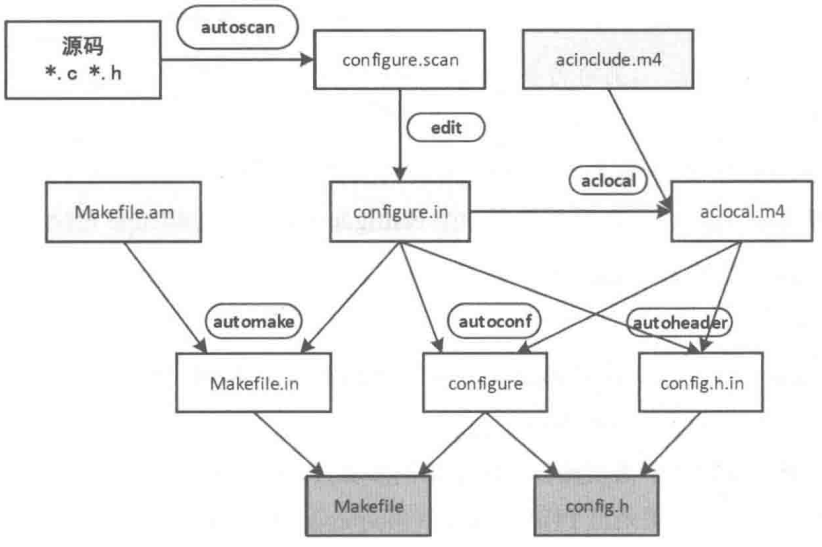


图 10-2 auto 系列的工具使用流程

（2）aclocal： automake 中有很多宏可以在 configure.in 或其他.m4 配置中使用，这些宏必须定义在 aclocal.m4 中，否则将无法被 autoconf 识别，aclocal 可以根据 configure.in 自动生成 aclocal.m4。另外，autoconf 提供的特性不可能满足所有的需求，所以 autoconf 还支持自定义宏，用户可以在 acinclude.m4 中定义自己的宏，然后在执行 aclocal 生成 aclocal.m4 时也会将 acinclude.m4 加载进去。

（3）autoheader： 它可以根据 configure.in、aclocal.m4 生成一个 C 语言"define"声明的头文件模板（config.h.in）供 configure 执行时使用，比如很多程序会通过 configure 提供一些 enable/disable 的参数，然后根据不同的参数决定是否开启某些选项，这种就可以根据编译参数的值生成一个 define 宏，比如`--enabled-xxx` 生成`#define ENABLED_XXX 1`，否则默认生成`#define ENABLED_XXX 0`，代码里直接使用这个宏即可。比如 configure.in 文件内容如下：

```
AC_PREREQ([2.63])
AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])

AC_CONFIG_HEADERS([config.h])

AC_ARG_ENABLE(xxx, "--enable-xxx if enable xxx",[
   AC_DEFINE([ENABLED_XXX], [1], [enabled xxx])
],
[
   AC_DEFINE([ENABLED_XXX], [0], [disabled xxx])
```

```
])

AC_OUTPUT
```

执行 autoheader 后将生成一个 config.h.in 的文件，里面包含#undef ENABLED_XXX，最终执行 ./configure --enable-xxx 后将生成一个 config.h 文件，包含#define ENABLED_XXX 1。

（4）autoconf：将 configure.in 中的宏展开生成 configure、config.h，此过程会用到 aclocal.m4 中定义的宏。

（5）automake：将 Makefile.am 中定义的结构建立 Makefile.in，然后 configure 脚本将生成的 Makefile.in 文件转换为 Makefile。

编写 PHP 扩展时并不需要操作上面全部的步骤，PHP 提供了两个编辑好的配置：configure.in、acinclude.m4，这两个配置是从 PHP 安装路径/lib/php/build 目录下的 phpize.m4、acinclude.m4 复制来的。configure.in 中定义了一些 PHP 内核相关的配置检查项，这个文件会 include 每个扩展各自的配置:config.m4，所以编写扩展时我们只需要在 config.m4 中定义扩展自己的配置就可以了，不需要关心依赖的 PHP 内核相关的配置，在扩展所在目录下执行 phpize 脚本就可以自动生成扩展的 configure、config.h 文件。configure.in 的基本内容如下，了解 autoconf 使用的读者对这个结构应该不会陌生。

```
AC_PREREQ(2.59)
AC_INIT(config.m4)
...
#--with-php-config 参数
PHP_ARG_WITH(php-config,,
[  --with-php-config=PATH  Path to php-config [php-config]], php-config, no)

PHP_CONFIG=$PHP_PHP_CONFIG
...
#加载扩展配置
sinclude(config.m4)
...
#根据编译参数生成 config.h
AC_CONFIG_HEADER(config.h)

AC_OUTPUT()
```

接下来看一下 phpize 脚本中有哪些处理。

（1）phpize_check_configm4：检查扩展的 config.m4 是否存在。

（2）phpize_check_build_files：检查 PHP 安装路径下的 lib/php/build/，这个目录下包含 PHP 自定义的 autoconf 宏文件 acinclude.m4 以及 libtool；检查扩展所在目录。

（3）phpize_print_api_numbers：输出 PHP API Version、Zend Module API No、Zend Extension API No 信息。

（4）phpize_copy_files：将 PHP 安装位置/lib/php/build 目录下的 mkdep.awk、scan_makefile_in.awk、shtool、libtool.m4 四个文件复制到扩展的 build 目录下，然后将 acinclude.m4、Makefile.global、config.sub、config.guess、ltmain.sh、run-tests*.php 文件复制到扩展根目录下，最后将 acinclude.m4、build/libtool.m4 合并到扩展目录下的 aclocal.m4 文件中。

```
#file: /usr/local/php7/bin/phpize
phpize_copy_files()
{
  test -d build || mkdir build

  (cd "$phpdir" && cp $FILES_BUILD "$builddir"/build)
  (cd "$phpdir" && cp $FILES "$builddir")
  #acinclude.m4、libtool.m4 合并到 aclocal.m4
  (cd "$builddir" && cat acinclude.m4 ./build/libtool.m4 > aclocal.m4)
}
```

（5）phpize_replace_prefix：将 PHP 安装位置/lib/php/build/phpize.m4 复制到扩展目录下，将文件中的 prefix 替换为 PHP 安装路径，然后重命名为 configure.in。

```
#file: /usr/local/php7/bin/phpize
phpize_replace_prefix()
{
  $SED \
  -e "s#/usr/local/php7#$prefix#" \
  < "$phpdir/phpize.m4" > configure.in
}
```

（6）phpize_check_shtool：检查/build/shtool。

（7）phpize_check_autotools：检查 autoconf、autoheader 是否已安装。

（8）phpize_autotools：执行 autoconf 生成 configure，然后再执行 autoheader 生成 config.h。

```
#file: /usr/local/php7/bin/phpize
phpize_autotools()
{
  $PHP_AUTOCONF   || exit 1
  $PHP_AUTOHEADER || exit 1
}
```

10.2.2 扩展的编写步骤

可以按照以下步骤编写一个 PHP 扩展。

- 通过 ext 目录下 ext_skel 脚本生成扩展的基本框架./ext_skel –extname。
- 修改 config.m4 配置：设置编译配置参数、设置扩展的源文件、依赖库/函数检查，等等。
- 编写扩展要实现的功能：按照 PHP 扩展的格式以及 PHP 提供的 API 编写功能。
- 生成 configure：扩展编写完成后执行 phpize 脚本生成 configure 及其他配置文件。
- 编译&安装：./configure、make、make install，然后将扩展的.so 路径添加到 php.ini 中。

在扩展的开发过程中，需要经历反复编译、安装的过程：如果有文件的增加或删除以及编译参数的变更，则需要重新执行 phpize；如果只是现有文件内容的更改则只需要重新编译、安装即可。

10.2.3 config.m4

config.m4 是扩展的编译配置文件，它是 autoconf 格式的配置文件，以 dnl 开头的行为注释，编译时 config.4 被 include 到 configure.in 文件中，最终被 autoconf 编译为 configure。编写扩展时我们只需要在 config.m4 中修改配置即可，一个简单的扩展配置只需要包含以下内容：

```
PHP_ARG_WITH(扩展名称, for mytest support,
Make sure that the comment is aligned:
[  --with-扩展名称             Include xxx support])

if test "$PHP_扩展名称" != "no"; then
    PHP_NEW_EXTENSION(扩展名称, 源码文件列表, $ext_shared,, -DZEND_ENABLE_
STATIC_TSRMLS_CACHE=1)
fi
```

PHP 在 acinclude.m4 中基于 autoconf/automake 的宏封装了很多可以直接使用的宏，使用时可以在这个文件中自行搜索，也可以参考其他扩展的用法，下面介绍几个比较常用的宏。

（1）PHP_ARG_WITH(arg_name,check message,help info)：定义一个--with-feature[=arg]这样的编译参数，调用的是 autoconf 的 AC_ARG_WITH，这个宏有 5 个参数，常用的是前三个，分别表示参数名、执行./configure 时的展示信息、执行--help 时的展示信息，第 4 个参数为默认值，如果不定义默认为"no"，通过这个宏定义的参数可以在 config.m4 中通过$PHP_参数名(大写)访问，比如：

```
PHP_ARG_WITH(aaa, aaa-configure, help aa)
dnl 接下来 config.m4 脚本中就可以通过$PHP_AAA 读取到--with-aaa=xxx 设置的值了
```

（2）PHP_ARG_ENABLE(arg_name,check message,help info)：定义一个--enable-feature[=arg] 或 --disable-feature 参数，--disable-feature 等价于 --enable-feature=no，这个宏与 PHP_ARG_WITH 类似，通常情况下如果配置的参数需要额外的 arg 值会使用 PHP_ARG_WITH，而如果不需要 arg 值，只用于开关配置则会使用 PHP_ARG_ENABLE。

（3）AC_MSG_CHECKING()/AC_MSG_RESULT()/AC_MSG_ERROR()：执行./configure 时的输出结果，其中 error 将中断 configure 执行。

（4）AC_DEFINE(variable, value, [description])：定义一个宏，比如 AC_DEFINE(IS_DEBUG, 1, [])，执行 autoheader 时将在头文件中生成：#define IS_DEBUG 1。

（5）PHP_ADD_INCLUDE(path)：添加 include 路径，即 gcc -Iinclude_dir，当 include 一个文件时将先在通过-I 指定的目录下查找，扩展引用了外部库或者扩展下分了多个目录的情况下会用到这个宏。

（6）PHP_CHECK_LIBRARY(library, function [, action-found [, action-not-found [, extra-libs]]])：检查依赖的库中是否存在需要的 function，action-found 为存在时执行的动作，action-not-found 为不存在时执行的动作，比如扩展里使用了线程 pthread 库，检查 pthread_create()函数是否支持，如果没找到则终止./configure 执行：

```
PHP_ADD_INCLUDE(pthread, pthread_create, [], [
    AC_MSG_ERROR([not find pthread_create() in lib pthread])
])
```

（7）AC_CHECK_FUNC(function, [action-if-found], [action-if-not-found])：检查函数是否存在。

（8）PHP_ADD_LIBRARY_WITH_PATH(*LIBNAME*,XXX_DIR/$PHP_LIBDIR,XXX_SHARED_

LIBADD)：添加链接库，即：gcc 编译时的-l、-L 参数。

（9）PHP_NEW_EXTENSION(extname, sources [, shared [, sapi_class [, extra-cflags [, cxx [, zend_ext]]]]])：注册一个扩展，列举扩展的全部*.c 源文件，确定此扩展是动态库还是静态库，每个扩展的 config.m4 中都需要通过这个宏完成扩展的编译配置。

```
PHP_NEW_EXTENSION(mytest, mytest.c user/user.c, $ext_shared,, -DZEND_
ENABLE_STATIC_TSRMLS_CACHE=1)
```

（10）PHP_INSTALL_HEADERS(path [, file ...])：按照扩展的头文件，如果扩展提供了一些方法供其他扩展使用，就可以通过这个宏将头文件安装到 PHP 的 include 目录下。

```
PHP_INSTALL_HEADERS(ext/pdo, [php_pdo.h php_pdo_driver.h php_pdo_error.h])
```

10.3 钩子函数

PHP 为扩展提供了 5 个钩子函数，PHP 执行到不同阶段时回调各个扩展定义的钩子函数，扩展可以通过这些钩子函数介入到PHP生命周期的不同阶段中，这些钩子函数的定义非常简单，PHP 提供了对应的宏，定义完成后只需要设置 zend_module_entry 对应的函数指针即可。

前面已经介绍过PHP生命周期的几个阶段，这几个钩子函数执行的先后顺序：module startup →request startup→编译、执行→request shutdown→post deactivate→module shutdown。

10.3.1 模块初始化阶段

在模块初始化阶段调用的钩子函数通过 zend_module_entry→module_startup_func 指定，通常情况下，此过程只会在 SAPI 启动后执行一次。这个阶段可以进行内部类的注册，如果你的扩展提供了类就可以在此函数中完成注册；除了类，还可以在此函数中注册扩展定义的常量；另外，扩展可以在此阶段覆盖PHP编译、执行的两个函数指针 zend_compile_file、zend_execute_ex，从而可以接管 PHP 的编译、执行，opcache 的实现原理就是替换了 zend_compile_file，使得 PHP 编译时调用的是 opcache 自己定义的编译函数，对编译后的结果进行缓存。

在扩展中可以通过 PHP_MINIT_FUNCTION()或 ZEND_MINIT_FUNCTION()宏定义这个函数，该宏为函数设置了统一参数，并把函数名称加上了 zm_startup_作为前缀：

```
//file: main/php.h:
#define PHP_MINIT_FUNCTION      ZEND_MODULE_STARTUP_D
//file: Zend/zend_API.h:
```

```
#define ZEND_MODULE_STARTUP_D(module)        int ZEND_MODULE_STARTUP_
N(module)(INIT_FUNC_ARGS)
#define ZEND_MODULE_STARTUP_N(module)       zm_startup_##module
#define INIT_FUNC_ARGS       int type, int module_number
```

比如：

```
PHP_MINIT_FUNCTION(mytest)
{
    ...
}
//展开后：
zm_startup_mytest(int type, int module_number)
{
    ...
}
```

定义完成后设置 zend_module_entry→module_startup_func 为扩展自定义的函数即可，这里直接使用 PHP_MINIT()或 ZEND_MINIT()宏获取对应的函数名称。

```
#define PHP_MINIT          ZEND_MODULE_STARTUP_N
#define ZEND_MINIT          ZEND_MODULE_STARTUP_N

#define ZEND_MODULE_STARTUP_N(module)       zm_startup_##module
```

10.3.2 请求初始化阶段

在请求初始化时调用的钩子函数通过 zend_module_entry→request_startup_func 指定，此函数在编译、执行之前回调，fpm 模式下每一个 HTTP 请求就是一个 request，脚本执行前将首先执行这个函数。如果扩展需要针对每一个请求进行处理则可以提供这个函数，比如：对请求进行过滤、根据请求 IP 获取所在城市、对请求/返回数据加解密等。此函数通过 PHP_RINIT_FUNCTION()或 ZEND_RINIT_FUNCTION()宏定义：

```
//file: main/php.h
#define PHP_RINIT_FUNCTION        ZEND_MODULE_ACTIVATE_D
//file: Zend/zend_API.h
#define ZEND_MODULE_ACTIVATE_D(module)        int ZEND_MODULE_ACTIVATE_N
```

```
(module)(INIT_FUNC_ARGS)
    #define ZEND_MODULE_ACTIVATE_N(module)        zm_activate_##module
```

比如：

```
    PHP_RINIT_FUNCTION(mytest)
    {
        ...
    }
    //展开后：
    zm_activate_mytest(int type, int module_number)
    {
        ...
    }
```

获取函数地址的宏：PHP_RINIT()或 ZEND_RINIT()：

```
    #define PHP_RINIT          ZEND_MODULE_ACTIVATE_N
    #define ZEND_RINIT          ZEND_MODULE_ACTIVATE_N

    #define ZEND_MODULE_ACTIVATE_N(module)      zm_activate_##module
```

10.3.3 请求结束阶段

这个阶段对应请求初始化阶段，钩子函数通过 zend_module_entry→request_shutdown_func 指定，此函数在请求结束时被调用，通过 PHP_RSHUTDOWN_FUNCTION()或 ZEND_RSHUTDOWN_FUNCTION()宏定义：

```
    //file: main/php.h
    #define PHP_RSHUTDOWN_FUNCTION    ZEND_MODULE_DEACTIVATE_D
    //file: Zend/zend_API.h
    #define ZEND_MODULE_DEACTIVATE_D(module)    int ZEND_MODULE_DEACTIVATE_N
(module)(SHUTDOWN_FUNC_ARGS)
    #define ZEND_MODULE_DEACTIVATE_N(module)    zm_deactivate_##module
```

比如：

```
PHP_RSHUTDOWN_FUNCTION(mytest)
{
    ...
}
//展开后:
zm_deactivate_mytest(int type, int module_number)
{
    ...
}
```

函数地址通过 PHP_RSHUTDOWN()或 ZEND_RSHUTDOWN()获取：

```
#define PHP_RSHUTDOWN       ZEND_MODULE_DEACTIVATE_N
#define ZEND_RSHUTDOWN      ZEND_MODULE_DEACTIVATE_N

#define ZEND_MODULE_DEACTIVATE_N(module)    zm_deactivate_##module
```

10.3.4 post deactivate 阶段

通过 zend_module_entry→post_deactivate_func 指定，这个函数比较特殊，一般很少会用到，实际上它也是在请求结束之后调用的，它比 request_shutdown_func 更晚执行。

```
//file: main/main.c
void php_request_shutdown(void *dummy)
{
    ...
    //调用各扩展的 request_shutdown_func
    if (PG(modules_activated)) {
        zend_deactivate_modules();
    }
    //关闭输出:发送 http header
    php_output_deactivate();
    //释放超全局变量: $_GET、$_POST...
    ...
    //关闭编译器、执行器
    zend_deactivate();
    //调用每个扩展的 post_deactivate_func
```

```
    zend_post_deactivate_modules();
    ...
}
```

从上面的执行顺序可以看出，request_shutdown_func、post_deactivate_func 是先后执行的，此函数通过 ZEND_MODULE_POST_ZEND_DEACTIVATE_D()宏定义，ZEND_MODULE_POST_ZEND_DEACTIVATE_N()获取函数地址：

```
//file: Zend/zend_API.h
#define ZEND_MODULE_POST_ZEND_DEACTIVATE_D(module)    int ZEND_MODULE_
POST_ZEND_DEACTIVATE_N(module)(void)
#define ZEND_MODULE_POST_ZEND_DEACTIVATE_N(module)    zm_post_zend_
deactivate_##module
```

10.3.5 模块关闭阶段

对应模块初始化阶段，调用的钩子函数通过 zend_module_entry→module_shutdown_func 指定，此阶段可以进行一些资源的清理，通过 PHP_MSHUTDOWN_FUNCTION()或 ZEND_MSHUTDOWN_FUNCTION()定义：

```
PHP_MSHUTDOWN_FUNCTION(mytest)
{
    ...
}
//展开后：
zm_shutdown_mytest(int type, int module_number)
{
    ...
}
```

通过 PHP_MSHUTDOWN()或 ZEND_MSHUTDOWN()获取函数地址：

```
#define PHP_MSHUTDOWN        ZEND_MODULE_SHUTDOWN_N
#define ZEND_MSHUTDOWN       ZEND_MODULE_SHUTDOWN_N

#define ZEND_MODULE_SHUTDOWN_N(module)        zm_shutdown_##module
```

上面详细介绍了各个阶段定义的钩子函数的格式，使用 gdb 调试扩展时可以根据展开后实际的函数名称设置断点。这些钩子实际上已经为扩展构造了一个整体的框架，通过这几个钩子扩展已经能实现很多功能了，后面我们介绍的很多内容都是在这几个函数中完成的，比如内部类的注册、常量注册、资源注册等。如果扩展名称为 mytest，则最终定义的扩展：

```
PHP_MINIT_FUNCTION(mytest)
{
    ...
}
PHP_RINIT_FUNCTION(mytest)
{
    ...
}
PHP_RSHUTDOWN_FUNCTION(mytest)
{
    ...
}
PHP_MSHUTDOWN_FUNCTION(mytest)
{
    ...
}
zend_module_entry mytest_module_entry = {
    STANDARD_MODULE_HEADER,
    "mytest",
    NULL,
    PHP_MINIT(mytest),
    PHP_MSHUTDOWN(mytest),
    PHP_RINIT(mytest),
    PHP_RSHUTDOWN(mytest),
    NULL,
    "1.0.0",
    STANDARD_MODULE_PROPERTIES
};
//获取 mytest_module_entry 地址的接口
ZEND_GET_MODULE(mytest)
```

10.4 全局资源

使用 C 语言开发程序时经常会使用全局变量进行数据存储，以实现函数间的数据共享，但是多线程环境下就不能再使用这种方式了。第 4 章曾详细介绍过线程安全相关的实现，同样的，在扩展中也需要必须通过 TSRM 注册、使用全局资源，除非你的扩展不支持多线程的环境。

如果在扩展中需要使用全局资源来实现函数间数据共享，则可以像内核中的 EG、CG 那样，向 TSRM 注册全局资源。扩展中，通常是定义一个保存全局资源的结构体，然后把这个结构体注册到 TSRM，这个结构可以使用 ZEND_BEGIN_MODULE_GLOBALS(extension_name)、ZEND_END_MODULE_GLOBALS(extension_name)两个宏定义，这两个宏必须成对出现，中间定义扩展中用到“全局变量”即可。比如：

```
ZEND_BEGIN_MODULE_GLOBALS(mytest)
    zend_long  open_cache;
    HashTable  class_table;
ZEND_END_MODULE_GLOBALS(mytest)
```

展开后实际是个结构体：

```
typedef struct _zend_mytest_globals {
    zend_long   open_cache;
    HashTable   class_table;
}zend_mytest_globals;
```

接着创建一个此结构体的全局变量，这时候就会涉及 ZTS 了，如果未开启线程安全，直接创建普通的全局变量即可，如果开启线程安全了，则需要向 TSRM 注册，得到一个唯一的资源 id，这个操作也由专门的宏来完成：ZEND_DECLARE_MODULE_GLOBALS(extension_name)。

```
//file: Zend/zend_API.h
#ifdef ZTS
#define ZEND_DECLARE_MODULE_GLOBALS(module_name)     \
    ts_rsrc_id module_name##_globals_id;
#else
#define ZEND_DECLARE_MODULE_GLOBALS(module_name)    \
    zend_##module_name##_globals module_name##_globals;
#endif
```

还是上面 mytest 的例子：

```
//file: mytest.c
ZEND_DECLARE_MODULE_GLOBALS(mytest)
```

展开后：

```
//ZTS: 此时只是定义资源 id, 并没有向 TSRM 注册
ts_rsrc_id mytest_globals_id;
//非 ZTS
zend_mytest_globals mytest_globals;
```

最后，定义一个像 EG、CG 那样的宏用于访问扩展的全局资源结构体，这一步将使用 ZEND_MODULE_GLOBALS_ACCESSOR(module_name, v)宏完成：

```
#define MYTEST_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(mytest, v)
```

看起来是不是跟 EG、CG 的定义非常像？这个宏展开后：

```
//ZTS
#define MYTEST_G(v) ZEND_TSRMG(mytest_globals_id, zend_mytest_globals *, v)
//非 ZTS
#define MYTEST_G(v) (mytest_globals.v)
```

接下来就可以在扩展中通过：MYTEST_G(opene_cache)、MYTEST_G(class_table)对结构体成员进行读写了。通常会把这个全局资源结构体的定义、访问结构体的宏定义在头文件中，然后把全局变量的声明放到源文件中。

```
//php_mytest.h
#define MYTEST_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(mytest, v)

ZEND_BEGIN_MODULE_GLOBALS(mytest)
    zend_long   open_cache;
    HashTable   class_table;
ZEND_END_MODULE_GLOBALS(mytest)

//mytest.c
ZEND_DECLARE_MODULE_GLOBALS(mytest)
```

至此，一个线程安全的全局资源就定义完成了，可以像全局变量一样方便地使用了。在一个扩展中并不是只能定义一个全局变量结构，数目是没有限制的，定义不同的结构名称即可。

10.5 ini 配置

php.ini 是 PHP 主要的配置文件，解析时 PHP 将在这些地方依次查找该文件：当前工作目录、环境变量 PHPRC 指定目录、编译时指定的路径，在命令行模式下，php.ini 的查找路径可以用-c 参数替代。

该文件的语法非常简单：配置标识符 = 值。空白字符和用分号';'开始的行被忽略，[xxx]行也被忽略；配置标识符大写敏感，通常会用'.'区分不同的节；值可以是数字、字符串、PHP 常量、位运算表达式。

关于 php.ini 的解析过程本书不作介绍，只从应用的角度介绍如何在一个扩展中获取一个配置项。使用时，通常会把 php.ini 的配置解析到上一节介绍的全局资源，从而在使用时直接操作那个变量，要想实现这个解析，需要在扩展中为用到的 php.ini 的配置设置解析规则，配置时需要将各项规则定义在 PHP_INI_BEGIN()、PHP_INI_END()两个宏中间，这两个宏实际生成了一个数组来保存各解析规则。

```
PHP_INI_BEGIN()
    //各项解析规则
    ...
PHP_INI_END();

//file: main/php_ini.h
#define PHP_INI_BEGIN         ZEND_INI_BEGIN
#define PHP_INI_END           ZEND_INI_END
//file: Zend/zend_ini.h
#define ZEND_INI_BEGIN()  static const zend_ini_entry_def ini_entries[] = {
#define ZEND_INI_END()    { NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0, 0, 0} };
```

解析规则就是设置 php.ini 中配置项要解析到的变量，通过 STD_PHP_INI_ENTRY()宏定义：

```
STD_PHP_INI_ENTRY(name,default_value,modifiable,on_modify,property_name,
struct_type,struct_ptr)
```

- **name**：php.ini 中的配置项。

- **default_value**：默认值，注意不管转化后是什么类型，这里必须设置为字符串。
- **modifiable**：可修改等级，ZEND_INI_USER 为可以在 PHP 脚本中修改，ZEND_INI_SYSTEM 为可以在 php.ini、httpd.conf 中修改，还有一个 ZEND_INI_PERDIR 表示可在 php.ini、.htaccess 或 httpd.conf 中设定，ZEND_INI_ALL 表示三种都可以，通常情况下设置为 ZEND_INI_ALL、ZEND_INI_SYSTEM 即可。
- **on_modify**：解析函数，当发现 name 配置将调用该函数完成解析，默认提供了 5 个——OnUpdateBool、OnUpdateLong、OnUpdateLongGEZero、OnUpdateReal、OnUpdateString、OnUpdateStringUnempty，支持可以自定义。
- **property_name**：要解析到的 struct_type 结构中的成员。
- **struct_type**：解析到的结构类型。
- **struct_ptr**：解析到的结构的地址。

也就是说，将 php.ini 中 name 配置项的值，解析到(struct_type*)struct_ptr→property_name，STD_PHP_INI_ENTRY()宏将每一条解析规则生成一个 zend_ini_entry_def 结构来保存上面提供的信息。

```
typedef struct _zend_ini_entry_def {
    const char *name;
    int (*on_modify)(zend_ini_entry *entry, zend_string *new_value, void
*mh_arg1, void *mh_arg2, void *mh_arg3, int stage);
    //映射成员所在结构体的偏移:offsetof(type, member-designator)
    void *mh_arg1;
    //要映射到结构的地址
    void *mh_arg2;
    void *mh_arg3;
    const char *value;//默认值
    void (*displayer)(zend_ini_entry *ini_entry, int type);
    int modifiable;

    uint name_length;
    uint value_length;
} zend_ini_entry_def;
```

> 除了 STD_PHP_INI_ENTRY()这个宏，还有一个类似的宏 STD_PHP_INI_BOOLEAN()，用法一致，差别在于后者会自动把配置添加到 phpinfo()输出中。

比如将 php.ini 中的 mytest.opene_cache 的值解析到上一节的例子：MYTEST_G()结构中的 open_cache，类型为 zend_long，默认值 109，则可以这么定义：

```
PHP_INI_BEGIN()
    STD_PHP_INI_ENTRY("mytest.open_cache", "109", PHP_INI_ALL, OnUpdateLong,
open_cache, zend_mytest_globals, mytest_globals)
PHP_INI_END();
```

展开后：

```
static const zend_ini_entry_def ini_entries[] = {
    {
        "mytest.open_cache",
        OnUpdateLong,
        //获取 open_cache 成员在结构体中的内存偏移，比如
        (void *) XtOffsetOf(zend_mytest_globals, open_cache),
        (void*)&mytest_globals,
        NULL,
        "109",
        NULL,
        PHP_INI_ALL,
        sizeof("mytest.open_cache")-1,
        sizeof("109")-1
    },
    { NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0, 0, 0}
}
```

XtOffsetOf()这个宏在 Linux 环境下展开就是 offsetof()，用来获取一个结构体成员的地址 offset，比如 offset = (void*) XtOffsetOf(zend_mytest_globals, open_cache)，mytest_globals 地址为 ptr，则可以按字节通过 offset 访问 open_cache：(char*)ptr + offset，等价于 ptr->open_cache。

上面只是定义了映射规则，还没有进行注册，此时还无法将 php.ini 配置按照规则解析到指定变量，接下来还需要关键的一步：REGISTER_INI_ENTRIES()，这个宏展开后：zend_register_ini_entries(ini_entries, module_number)，通常会把这一步操作放到模块初始化阶段，也就是放到扩展的 PHP_MINIT_FUNCTION()方法中。在这个阶段，php.ini 已经完成解析，所有的配置项

保存在 configuration_hash 这个哈希表中，zend_register_ini_entries()的处理就是根据配置的解析规则查找这个哈希表，如果找到了对应的 name 则调用此规则指定的 on_modify 函数进行赋值，比如上面的示例将调用 OnUpdateLong()处理，具体的流程如下。

```
    ZEND_API int zend_register_ini_entries(const zend_ini_entry_def *ini_entry,
int module_number)
    {
        zend_ini_entry *p;
        zval *default_value;
        HashTable *directives = registered_zend_ini_directives;
        //遍历解析规则
        while (ini_entry→name) {
            //分配 zend_ini_entry 结构
            p = pemalloc(sizeof(zend_ini_entry), 1);
            //zend_ini_entry 初始化
            ...
            //添加到 registered_zend_ini_directives, EG(ini_directives)也是指向此
HashTable
            if (zend_hash_add_ptr(directives, p→name, (void*)p) == NULL) {
                ...
            }
            //zend_get_configuration_directive()最终将调用 cfg_get_entry()
            //从 configuration_hash 哈希表中查找配置，如果没有找到将使用默认值
            default_value = zend_get_configuration_directive(p→name)
            ...
            if (p→on_modify) {
                //调用定义的赋值 handler 处理
                p→on_modify(p, p→value, p→mh_arg1, p→mh_arg2, p→mh_arg3, ZEND_
INI_STAGE_STARTUP);
            }
        }
    }
```

OnUpdateLong()将配置项的值赋值给指定变量结构成员：

```
    ZEND_API ZEND_INI_MH(OnUpdateLong)
    {
        zend_long *p;
```

```
#ifndef ZTS
    //存储结构的指针
    char *base = (char *) mh_arg2;
#else
    char *base;
    //ZTS 下需要向 TSRM 中获取存储结构的指针
    base = (char *) ts_resource(*((int *) mh_arg2));
#endif
    //指向结构体成员的位置
    p = (zend_long *) (base+(size_t) mh_arg1);
    //将值转为 zend_long
    *p = zend_atol(ZSTR_VAL(new_value), (int)ZSTR_LEN(new_value));
    return SUCCESS;
}
```

这时候就成功将 mytest.open_cache 的值解析到了 MYTEST_G(open_cache)中，如图 10-3 所示。

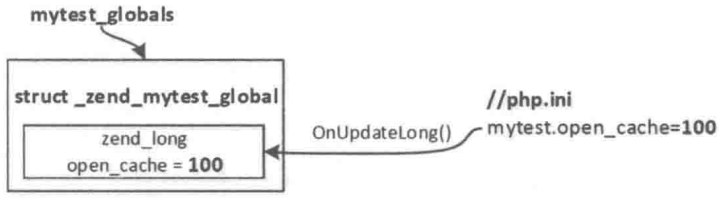


图 10-3 php.ini 解析到指定变量

如果 PHP 提供的几个 on_modify 不能满足需求，则可以自定义 on_modify 函数，通过 ZEND_INI_MH(div_on_modify)宏定义。

```
//file: Zend/zend_ini.h
#define ZEND_INI_MH(name) int name(zend_ini_entry *entry, zend_string
*new_value, void *mh_arg1, void *mh_arg2, void *mh_arg3, int stage)
```

举个例子，将 php.ini 中的配置 mytest.class 插入 MYTESY_G(class_table)哈希表，则可以在扩展中定义这样一个 on_modify：ZEND_INI_MH(OnUpdateAddArray)。

```
PHP_INI_BEGIN()
    STD_PHP_INI_ENTRY("mytest.class", "stdClass", PHP_INI_ALL, OnUpdateAddArray,
class_table, zend_mytest_globals, mytest_globals)
PHP_INI_END();
```

```
ZEND_API ZEND_INI_MH(OnUpdateAddArray)
{
    HashTable   *ht;
    zval    val;
#ifndef ZTS
    char *base = (char *) mh_arg2;
#else
    char *base;
    base = (char *) ts_resource(*((int *) mh_arg2));
#endif

    ht = (HashTable*)(base+(size_t) mh_arg1);
    ZVAL_NULL(&val);
    zend_hash_add(ht, new_value, &val);
}

PHP_MINIT_FUNCTION(mytest)
{
    //将 php.ini 解析到指定结构体
    REGISTER_INI_ENTRIES();
}
```

10.6　函数

通过扩展可以将 C 语言实现的函数提供给 PHP 脚本使用，如同大量 PHP 内置函数一样，这些函数统称为内部函数（internal function），与 PHP 脚本中定义的用户函数不同，它们无须经历用户函数的编译过程，同时执行时也不像用户函数那样每一个指令都调用一次 C 语言编写的 handler 函数，而是直接执行编译好的机器指令，因此，内部函数的执行效率更高。除了性能上的优势，内部函数还可以拥有更高的控制权限，可发挥的作用也更大，能够完成很多用户函数无法实现的功能。

前面介绍 PHP 函数的编译时曾经详细介绍过 PHP 函数的实现，函数通过 zend_function 结构表示，这是一个联合体，用户自定义函数使用 op_array，而内部函数使用 internal_function，两者具有相同的头部用来保存函数的基本信息：函数类型、参数信息、函数名等。

```
typedef struct _zend_internal_function {
```

```
        /* Common elements */
        zend_uchar type;
        zend_uchar arg_flags[3]; /* bitset of arg_info.pass_by_reference */
        uint32_t fn_flags;
        zend_string* function_name;
        zend_class_entry *scope;
        zend_function *prototype;
        uint32_t num_args;
        uint32_t required_num_args;
        zend_internal_arg_info *arg_info;
        /* END of common elements */
        //函数指针，展开：void (*handler)(zend_execute_data *execute_data, zval
*return_value)
        void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
        struct _zend_module_entry *module;
        void *reserved[ZEND_MAX_RESERVED_RESOURCES];
    } zend_internal_function;
```

handler 就是定义的内部函数指针。

10.6.1 内部函数注册

不管是用户自定义函数还是内部函数，它们最终都被注册到 EG(function_table)符号表中，函数被调用时根据函数名称从这个符号表中查找。从内部函数的注册、使用过程可以看出，其定义实际上非常简单，我们只需要定义一个 zend_internal_function 结构，然后注册到 EG(function_table)中即可。

通过扩展提供一个内部函数主要分为两步。

1）定义函数

首先是编写内部函数，可以通过 PHP_FUNCTION()或 ZEND_FUNCTION()宏完成函数的标准声明。PHP 为函数名加了"zif_"作为前缀，gdb 调试时记得加上这个前缀，另外 PHP 调用内部函数会传入两个参数：execute_data、return_value，对于这两个值相信你并不会感到陌生，这两个值在扩展中经常会用到。

```
//file: main/php.h
#define PHP_FUNCTION            ZEND_FUNCTION
//file: Zend/zend_API.h
```

```
#define ZEND_FUNCTION(name)                ZEND_NAMED_FUNCTION(ZEND_FN(name))

#define ZEND_FN(name) zif_##name
#define ZEND_NAMED_FUNCTION(name)      void name(INTERNAL_FUNCTION_PARAMETERS)
//file: Zend/zend.h
#define INTERNAL_FUNCTION_PARAMETERS zend_execute_data *execute_data, zval
*return_value
```

比如要在扩展中定义函数 my_func()，则需要这样定义：

```
PHP_FUNCTION(my_func)
{
    //具体逻辑
    ...
    printf("Hello, I'm my_func\n");
}
```

展开后：

```
void zif_my_func(zend_execute_data *execute_data, zval *return_value)
{
    ...
}
```

（2）注册函数

函数定义完了就需要向 PHP 注册了，这里并不需要扩展自己注册，PHP 提供了一个内部函数注册结构：zend_function_entry，扩展只需要为每个内部函数生成这样一个结构，然后将所有函数的结构数组提供给扩展 zend_module_entry→functions 即可，注册扩展时会自动向 EG(function_table)注册。

```
typedef struct _zend_function_entry {
    const char *fname;  //函数名称
    void (*handler)(INTERNAL_FUNCTION_PARAMETERS); //handler 实现
    const struct _zend_internal_arg_info *arg_info;//参数信息
    uint32_t num_args; //参数数目
    uint32_t flags;
} zend_function_entry;
```

zend_function_entry 结构可以通过 PHP_FE()或 ZEND_FE()生成，然后需要把所有函数的 zend_function_entry 放到一个数组中，这个数组通过全局变量分配即可：

```
const zend_function_entry mytest_functions[] = {
    PHP_FE(my_func,   NULL)
    PHP_FE_END //末尾必须加这个
};
```

展开后：

```
const zend_function_entry mytest_functions[] = {
    {
        "my_func",
        zif_my_func,
        NULL, //arg_info
        0,  //num_args
        0 //flags
    },
    { NULL, NULL, NULL, 0, 0 }
};
```

关于 arg_info 稍后再介绍。最后将 zend_module_entry→functions 设置为 mytest_functions 即可：

```
zend_module_entry mytest_module_entry = {
    STANDARD_MODULE_HEADER,
    "mytest",
    mytest_functions,
    NULL, //PHP_MINIT(mytest),
    NULL, //PHP_MSHUTDOWN(mytest),
    NULL, //PHP_RINIT(mytest),
    NULL, //PHP_RSHUTDOWN(mytest),
    NULL, //PHP_MINFO(mytest),
    "1.0.0",
    STANDARD_MODULE_PROPERTIES
};
```

扩展注册时 php_load_extension()调用 zend_register_module_ex()完成内部函数的注册：

```
//file: Zend/zend_API.c
ZEND_API   zend_module_entry*   zend_register_module_ex(zend_module_entry
*module)
{
    ...
    //注册内部函数
    if (module→functions && zend_register_functions(NULL, module→functions,
NULL, module→type)==FAILURE) {
        EG(current_module) = NULL;
        zend_error(E_CORE_WARNING,"%s: Unable to register functions, unable
to load", module→name);
        return NULL;
    }
}
```

内部函数定义完成后来测试一下函数能否正常工作，编译安装后在 PHP 脚本中调用这个函数：

```
//call function
my_func();
```

在 Cli 模式下执行 php test.php 将输出：

```
Hello, I'm my_func
```

函数已经能够正常工作了，后续的工作就是不断完善 handler 实现扩展自己的功能了。

10.6.2 函数参数解析

先来回顾 PHP 函数参数相关的实现：用户自定义函数在编译时会为每个参数创建一个 zend_arg_info 结构，这个结构用来记录参数的名称、是否引用传参、是否为可变参数等信息，在存储上函数参数与函数内的局部变量并没有差别，可以把参数当作局部变量来对待，它们分配在 zend_execute_data 上；调用函数时首先会进行参数传递，这个过程是按参数次序，依次将参数的 value 从函数调用空间传递到被调函数的 zend_execute_data，函数内部像访问普通局部变量一样通过存储位置访问参数。

内部函数与用户自定义函数最大的不同在于：内部函数就是一个 C 语言定义的函数，它的局部变量是 C 语言中的变量，并不会分配在 zend_execute_data 上，因此，除函数参数以外在

zend_execute_data 上没有其他变量的分配。函数参数是从 PHP 用户空间传到内部函数中的，它们与用户自定义函数完全相同，包括参数的分配方式、传参过程，也是按照参数次序依次分配在 zend_execute_data 上，在内部函数中，可以按照参数顺序从 zend_execute_data 上读取到对应的参数。当然，实际使用时不需要我们自己去取，PHP 提供了一个方法将 zend_execute_data 上的参数解析到指定变量上。

```
//file: Zend/zend_API.h
ZEND_API int zend_parse_parameters(int num_args, const char *type_spec, ...);
```

- num_args，函数调用时实际传递的参数数，通过 ZEND_NUM_ARGS()获取到，即 zend_execute_data→This.u2.num_args。
- type_spec 为参数解析规则，是一个字符串，用来标识解析参数的类型，比如"la"表示第一个参数为整型，第二个为数组，将按照这个解析到指定变量。
- 后面是一个可变参数，用来指定解析到的变量，这个值与 type_spec 配合使用，即 type_spec 用来指定解析的变量类型，可变参数用来指定要解析到的变量地址。

解析的过程也比较容易理解，被调函数执行前已经将参数值从调用空间复制到被调函数的 zend_execute_data 上，所以解析的过程就是按照 type_spec 指定的各个类型获取对应的 value，依次从 zend_execute_data 上获取参数，然后将参数值的地址赋给目标变量。解析时除了整型、浮点型、布尔型是直接硬复制 value，其他解析到的变量只能是指针，因为解析的过程取的是 zend_execute_data 上参数 zval 或具体 value 的地址。比如下面的例子：

```
PHP_FUNCTION(my_func)
{
    zval    *arr;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "a", &arr) == FAILURE){
        RETURN_FALSE;
    }
    ...
}
```

my_func()函数的第一个参数为数组，解析后 arr 实际为 zend_execute_data 上第一个参数 zval 的地址，即 arr = (zval(*))((char*)zend_execute_data + 96)，如图 10-4 所示。

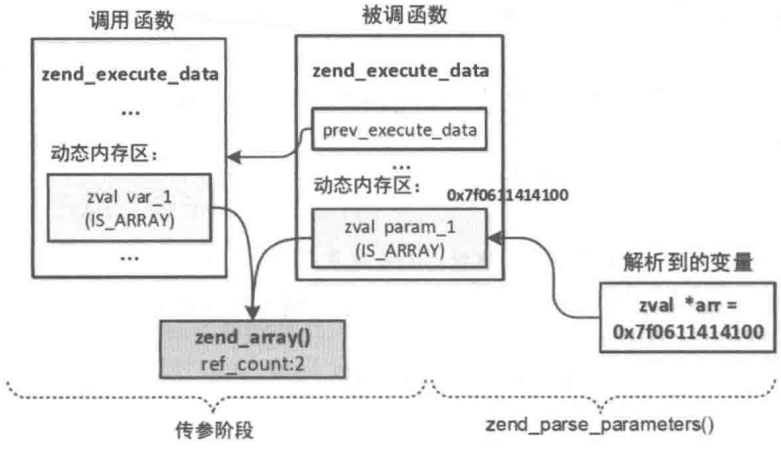


图 10-4　arr 与 zend_execute_data 参数地址的关系

zend_parse_parameters()调用了 zend_parse_va_args()进行处理，首先进行了一些简单的过滤，比如检查必传参数数，然后依次获取 zend_execute_data 的参数，根据不同的类型进行相应的解析。

```
    //va 为提供的要将参数解析到的变量的地址
    static int zend_parse_va_args(int num_args, const char *type_spec, va_list
*va, int flags)
    {
        const  char *spec_walk;
        int min_num_args = -1; //最少参数数
        int max_num_args = 0; //要解析的参数总数
        int post_varargs = 0;
        zval *arg;
        int arg_count; //实际传参数

        //遍历 type_spec 计算出 min_num_args、max_num_args
        for (spec_walk = type_spec; *spec_walk; spec_walk++) {
            ...
        }
        ...
        //检查数目是否合法
        if (num_args < min_num_args || (num_args > max_num_args && max_num_args >=
0)) {
            ...
        }
        //获取实际传参数：zend_execute_data.This.u2.num_args
```

```
    arg_count = ZEND_CALL_NUM_ARGS(EG(current_execute_data));
    ...
    i = 0;
    //逐个解析参数
    while (num_args-- > 0) {
        ...
        //获取第 i 个参数的 zval 地址：arg 就是在 zend_execute_data 上分配的局部变量
        arg = ZEND_CALL_ARG(EG(current_execute_data), i + 1);
        //解析第 i 个参数
        if (zend_parse_arg(i+1, arg, va, &type_spec, flags) == FAILURE) {
            if (varargs && *varargs) {
                *varargs = NULL;
            }
            return FAILURE;
        }
        i++;
    }
}
```

参数解析时，如果实际传参类型与要求的类型不符，则会按照类型转换规则进行对应的转换，不同类型的解析规则会有一些差异。

上面介绍的这种参数解析的方式与 PHP5.x 的方式是一样的，PHP7 中除了支持这种方式，还提供了另外一种更快的解析方式，新的这种方式首先需要通过两个宏将各个参数的解析规则包裹起来，例如：

```
ZEND_PARSE_PARAMETERS_START(2, 3) //最少要传 2 个参数，最多 3 个
    //依次配置各个参数的解析规则
ZEND_PARSE_PARAMETERS_END();
```

ZEND_PARSE_PARAMETERS_START(min_num_args, max_num_args)宏有两个参数：第一个为最少的参数数，即必传参数；第二个为最多可传的参数数，接着就可以在这两个宏之间按照参数的顺序定义解析规则了。下面详细看一下不同类型的两种解析方式：

1）整型：l、L

通过"l"、"L"标识表示解析的参数为整型，解析到的变量类型必须是 zend_long，不能解析其他类型，如果输入的参数不是整型，将按照类型转换规则将其转为整型。例如：

```
zend_long   lval;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "l", &lval){
    ...
}
printf("lval:%d\n", lval);
```

如果在标识符后加"!"，即"l!"、"L!"，表示允许该参数传 NULL，则必须再提供一个 zend_bool 变量的地址，通过这个值可以判断传入的参数是否为 NULL，如果为 NULL，则将要解析到的 zend_long 变量设置为 0，同时 zend_bool 设置为 1。如果不加"!"，传 NULL 时会被解析为 0，无法判断传的是 NULL 还是 0：

```
zend_long   lval; //如果参数为 NULL，则此值被设为 0
zend_bool   is_null; //如果参数为 NULL，则此值为 1，否则为 0

if(zend_parse_parameters(ZEND_NUM_ARGS(), "l!", &lval, &is_null){
    ...
}
```

各标识符具体的解析过程在 zend_parse_arg_impl()中，"l"、"L"的解析过程：

```
//file: zend_API.c zend_parse_arg_impl:
case 'l':
case 'L':
{
    //这里获取解析到的变量取的是 zend_long *，所以只能解析到 zend_long
    zend_long *p = va_arg(*va, zend_long *);
    zend_bool *is_null = NULL;
    //后面加"!"时 check_null 为 1
    if (check_null) {
        is_null = va_arg(*va, zend_bool *);
    }

    if (!zend_parse_arg_long(arg, p, is_null, check_null, c == 'L')) {
        return "integer";
    }
}
```

"l"与"L"的区别在于，当传参不是整型且转为整型后超过了整型的大小范围时，"L"将值调整为整型的最大或最小值，而"l"将报错，比如传的参数是字符串"9223372036854775808"(0x7FFFFFFFFFFFFFFF + 1)，转整型后超过了有符号 int64 的最大值：0x7FFFFFFFFFFFFFFF，所以如果是"L"将解析为 0x7FFFFFFFFFFFFFFF。

新的解析方式可以通过下面的宏完成解析，其中 is_null、check_null、separate 这几个值在不同类型宏中的含义是相同的，后面不再重复。

- Z_PARAM_LONG(dest)、Z_PARAM_LONG_EX(dest, is_null, check_null, separate)：同"l"，其中 dest 为 zend_long，check_null 同"!"，当参数为 NULL 时将设置 is_null，separate 为新的一个标识，用于指定参数的 value 是否分离，定义为 1，那么当参数为 string、array 时将会复制 value。
- Z_PARAM_STRICT_LONG(dest)、Z_PARAM_STRICT_LONG_EX(dest, is_null, check_null, separate)：同"L"。

两种解析方式最终都是通过 zend_parse_arg_long()处理的，只是获取解析到的变量的方式不同而已。

```
static zend_always_inline int zend_parse_arg_long(zval *arg, zend_long
*dest, zend_bool *is_null, int check_null, int cap)
{
    if (check_null) {
        *is_null = 0;
    }
    if (EXPECTED(Z_TYPE_P(arg) == IS_LONG)) {
        //传参为整型，无须转化
        *dest = Z_LVAL_P(arg);
    } else if (check_null && Z_TYPE_P(arg) == IS_NULL) {
        //传参为 NULL
        *is_null = 1;
        *dest = 0;
    } else if (cap) {
        //"L"的情况
        return zend_parse_arg_long_cap_slow(arg, dest);
    } else {
        //"l"的情况
        return zend_parse_arg_long_slow(arg, dest);
    }
    return 1;
}
```

PHP 不支持无符号整型，因此 64 位整型的最大值为 0x7FFFFFFFFFFFFFFF。

2）布尔型：b

"b"标识符表示将参数解析为布尔型，解析到的变量必须是 zend_bool，"b!"的用法与整型的完全相同，也必须再提供一个 zend_bool 的地址用于获取传参是否为 NULL，如果为 NULL，则结果为 0，用于获取是否 NULL 的 zend_bool 为 1。例如：

```
zend_bool  ok;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "b", &ok, &is_null) == FAILURE){
    ...
}
```

新的解析方式则可以通过 Z_PARAM_BOOL(dest)或 Z_PARAM_BOOL_EX(dest, is_null, check_null, separate) 宏解析，其参数与旧的解析方式一一对应。具体解析过程：

```
static zend_always_inline int zend_parse_arg_bool(zval *arg, zend_bool
*dest, zend_bool *is_null, int check_null)
{
    if (check_null) {
        *is_null = 0;
    }
    if (EXPECTED(Z_TYPE_P(arg) == IS_TRUE)) {
        *dest = 1;
    } else if (EXPECTED(Z_TYPE_P(arg) == IS_FALSE)) {
        *dest = 0;
    } else if (check_null && Z_TYPE_P(arg) == IS_NULL) {
        *is_null = 1;
        *dest = 0;
    } else {
        //其他类型，调用 zend_is_true()判断
        return zend_parse_arg_bool_slow(arg, dest);
    }
    return 1;
}
```

3）浮点型：d

"d"标识符表示将参数解析为浮点型，解析的变量类型必须为 double，"d!"与整型、布尔型用法完全相同。例如：

```
double  dval;
if(zend_parse_parameters(ZEND_NUM_ARGS(), "d", &dval) == FAILURE){
    ...
}
```

新的解析方式则可通过Z_PARAM_DOUBLE(dest)或Z_PARAM_DOUBLE_EX(dest, is_null, check_null, separate)宏进行解析，具体的解析过程：

```
static zend_always_inline int zend_parse_arg_double(zval *arg, double *dest,
zend_bool *is_null, int check_null)
{
    if (check_null) {
        *is_null = 0;
    }
    if (EXPECTED(Z_TYPE_P(arg) == IS_DOUBLE)) {
        *dest = Z_DVAL_P(arg);
    } else if (check_null && Z_TYPE_P(arg) == IS_NULL) {
        *is_null = 1;
        *dest = 0.0;
    } else {
        //从其他类型转为浮点型
        return zend_parse_arg_double_slow(arg, dest);
    }
    return 1;
}
```

4）字符串：s、S、p、P

字符串可以解析为两种形式：char*、zend_string*：其中"s"将参数解析为 char*，且需要额外提供一个 size_t 类型的变量用于获取字符串长度；"S"解析为 zend_string*。"s!"、"S!"与整型、布尔型用法不同，字符串时不需要额外提供 zend_bool 的地址，如果参数为 NULL，则解析到的变量将设置为 NULL。除了"s"、"S"，还有两个类似的："p"、"P"，从解析规则来看主要用于解析路径，实际上与普通字符串没什么区别，尚不清楚这两个有什么特殊用法。例如：

```
//解析为 char *字符串
char    *str;
size_t  str_len;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "s", &str, &str_len) == FAILURE){
    ...
}
//解析为 zend_string 地址
zend_string    *str;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "S", &str) == FAILURE){
    ...
}
```

新的解析方式分别通过以下宏进行操作：

- Z_PARAM_STRING(dest, dest_len)、Z_PARAM_STRING_EX(dest, dest_len, check_null, separate)：同"s"，如果 separate 为 1，且参数为普通字符串（非内部字符串），那么将会把传参的 value 深拷贝一份作为参数的 value，相当于传参是深拷贝了一份 value 给了被调函数，而不是增加的引用计数。
- Z_PARAM_STR(dest)、Z_PARAM_STR_EX(dest, check_null, separate)：同"S"。
- Z_PARAM_PATH(dest, dest_len)、Z_PARAM_PATH_EX(dest, dest_len, check_null, separate)：同"p"。
- Z_PARAM_PATH_STR(dest)、Z_PARAM_PATH_STR_EX(dest, check_null, separate)：同“P”。

具体的解析过程为：zend_parse_arg_str()、zend_parse_arg_string()，这里不再展开。

5）数组：a、A、h、H

数组的解析也有两类，一类是解析到 zval 地址，另一类是解析到 HashTable 地址，其中"a"、"A"解析到的变量类型必须是 zval*，"h"、"H"解析到的变量类型必须是 HashTable*。"a!"、"A!"、"h!"、"H!"的用法与字符串一致，也不需要额外提供别的地址，如果传参为 NULL，则对应解析到的变量赋为 NULL。"a"与"A"当传参为数组时没有任何差别，它们的区别在于：如果传参为对象"A"将按照对象解析到 zval，而"a"将报错。"h"与"H"当传参为数组时同样没有差别，当传参为对象时，"H"将调用对象的 get_properties 获取属性数组，然后把这个数组的地址赋值给解析到的变量，而"h"将报错。例如：

```
zval        *arr;   //必须是 zval 指针
HashTable   *ht;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "ah", &arr, &ht) == FAILURE){
    ...
}
```

具体解析过程：

```
//file: Zend/zend_API.c zend_parse_arg_impl:
case 'A':
case 'a':
{
    //解析到 zval*
    zval **p = va_arg(*va, zval **);

    if (!zend_parse_arg_array(arg, p, check_null, c == 'A')) {
        return "array";
    }
}
break;

case 'H':
case 'h':
{
    //解析到 HashTable *
    HashTable **p = va_arg(*va, HashTable **);

    if (!zend_parse_arg_array_ht(arg, p, check_null, c == 'H')) {
        return "array";
    }
}
break;
```

新的解析方式分别通过以下宏进行操作：

- Z_PARAM_ARRAY(dest)、Z_PARAM_ARRAY_EX(dest, check_null, separate)：同"a"，与字符串一样，当 separate 为 1，且参数为普通数组时，也将发生 value 分离。

- Z_PARAM_ARRAY_OR_OBJECT(dest, check_null, separate)、Z_PARAM_ARRAY_OR_OBJECT_EX(dest, check_null, separate)：同"A"，Z_PARAM_ARRAY_OR_OBJECT()这个宏虽然有 3 个参数，但是后面两个并没有什么用，属于定义错误。
- Z_PARAM_ARRAY_HT(dest)、Z_PARAM_ARRAY_HT_EX(dest, check_null, separate)：同"h"。
- Z_PARAM_ARRAY_OR_OBJECT_HT(dest)、Z_PARAM_ARRAY_OR_OBJECT_HT_EX(dest, check_null, separate)：同"H"。

两种类型分别由 zend_parse_arg_array()、zend_parse_arg_array_ht()解析：

```
    static zend_always_inline int zend_parse_arg_array(zval *arg, zval **dest,
int check_null, int or_object)
    {
        if (EXPECTED(Z_TYPE_P(arg) == IS_ARRAY) ||
            (or_object && EXPECTED(Z_TYPE_P(arg) == IS_OBJECT))) {
            *dest = arg;
        } else if (check_null && EXPECTED(Z_TYPE_P(arg) == IS_NULL)) {
            *dest = NULL;
        } else {
            return 0;
        }
        return 1;
    }

    static zend_always_inline int zend_parse_arg_array_ht(zval *arg, HashTable
**dest, int check_null, int or_object)
    {
        if (EXPECTED(Z_TYPE_P(arg) == IS_ARRAY)) {
            *dest = Z_ARRVAL_P(arg);
        } else if (or_object && EXPECTED(Z_TYPE_P(arg) == IS_OBJECT)) {
            //如果参数是对象且标识符为"H"，则获取对象的属性数组
            *dest = Z_OBJ_HT_P(arg)->get_properties(arg);
        } else if (check_null && EXPECTED(Z_TYPE_P(arg) == IS_NULL)) {
            *dest = NULL;
        } else {
            //解析失败，报错
            return 0;
```

```
    }
    return 1;
}
```

6）对象：o、O

通过"o"、"O"标识符将参数解析为对象，解析到的变量类型只能是 zval 的地址，无法解析到 zend_object。"O"要求解析指定类或其子类的对象，类似传参时显式地声明了参数类型的用法：function my_func(MyClass $obj)，如果参数不是指定类的实例则无法解析。"o!"、"O!"与字符串用法相同。例如：

```
zval    *obj;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE){
    ...
}
```

新的解析方式可通过下面的宏处理：

- Z_PARAM_OBJECT(dest)、Z_PARAM_OBJECT_EX(dest, check_null, separate)：同"o"。
- Z_PARAM_OBJECT_OF_CLASS(dest, _ce)、Z_PARAM_OBJECT_OF_CLASS_EX(dest, _ce, check_null, separate)：同"O"。

具体解析过程：

```
static zend_always_inline int zend_parse_arg_object(zval *arg, zval **dest,
zend_class_entry *ce, int check_null)
{
    if (EXPECTED(Z_TYPE_P(arg) == IS_OBJECT) &&
        (!ce || EXPECTED(instanceof_function(Z_OBJCE_P(arg), ce) != 0))) { //
如果是"O"则检查对象是否为指定类的实例
        *dest = arg;
    } else if (check_null && EXPECTED(Z_TYPE_P(arg) == IS_NULL)) {
        *dest = NULL;
    } else {
        return 0;
    }
    return 1;
}
```

7）资源：r

"r"标识符表示将参数解析为资源，与对象相同，只能解析到zval地址，而无法直接解析到zend_resource，"r!"与字符串用法相同。例如：

```
zval    *res;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "r", &res) == FAILURE){
    ...
}
```

新的解析方式可通过 Z_PARAM_RESOURCE(dest)或 Z_PARAM_RESOURCE_EX(dest, check_null, separate)宏进行解析。传参只能是资源类型，无法将其他类型解析为资源，具体的解析过程：

```
static zend_always_inline int zend_parse_arg_resource(zval *arg, zval
**dest, int check_null)
{
    if (EXPECTED(Z_TYPE_P(arg) == IS_RESOURCE)) {
        *dest = arg;
    } else if (check_null && EXPECTED(Z_TYPE_P(arg) == IS_NULL)) {
        *dest = NULL;
    } else { //其他类型无法转为资源
        return 0;
    }
    return 1;
}
```

8）类：C

如果参数是一个类则可以通过"C"解析出zend_class_entry地址：function my_func(stdClass)。这里有个地方比较特殊，解析到的变量可以设定为一个类，这种情况下解析时将判断找到的类与指定的类之间的关系，只有存在父子关系才能解析，如果只是想根据参数获取类型的zend_class_entry地址，记得将解析到的地址初始化为NULL，否则将发生不可预料的错误。例如：

```
zend_class_entry    *ce = NULL; //初始为NULL

if(zend_parse_parameters(ZEND_NUM_ARGS(), "C", &ce) == FAILURE){
```

```
    RETURN_FALSE;
}
```

具体解析过程：

```
case 'C':
{
    zend_class_entry *lookup, **pce = va_arg(*va, zend_class_entry **);
    zend_class_entry *ce_base = *pce;

    if (check_null && Z_TYPE_P(arg) == IS_NULL) {
        *pce = NULL;
        break;
    }
    convert_to_string_ex(arg);
    //根据名称查找类
    if ((lookup = zend_lookup_class(Z_STR_P(arg))) == NULL) {
        *pce = NULL;
    } else {
        *pce = lookup;
    }
    if (ce_base) {
        //如果解析到的变量不是NULL，则检查是否为父子类
        if ((!*pce || !instanceof_function(*pce, ce_base))) {
            *pce = NULL;
            return "";
        }
    }
    ...
}
break;
```

新的解析规则对应 Z_PARAM_CLASS(dest) 或 Z_PARAM_CLASS_EX(dest, check_null, separate) 宏。

9）回调类型(callable)：f

callable 指函数或成员方法，如果参数是函数名称字符串、array（对象/类,成员方法），则可以通过"f"标识符解析出 zend_fcall_info 结构，注意这里并不是 zend_fcall_info *，例如：

```
    zend_fcall_info        callable; //注意，这两个结构不能是指针
    zend_fcall_info_cache   call_cache;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "f", &callable, &call_cache) ==
FAILURE){
        RETURN_FALSE;
    }
```

调用函数时可传入下列类型的参数：

```
    my_func_1("func_name");
    my_func_1(array('class_name', 'static_method'));
    my_func_1(array($object, 'method'));
```

解析出的 zend_fcall_info 可以直接用于函数调用，这个功能可以简化函数调用的操作，否则需要我们自己去做查找函数、检查是否可被调用等工作，关于这个结构稍后介绍函数调用时再作详细说明。

新的解析规则对应 Z_PARAM_FUNC(dest_fci, dest_fcc) 或 Z_PARAM_FUNC_EX(dest_fci, dest_fcc, check_null, separate) 宏。

10）任意类型：z

"z"表示按参数实际类型解析，比如参数为字符串就解析为字符串，参数为数组就解析为数组，这种实际就是将 zend_execute_data 上的参数地址复制到目的变量了，没有做任何转化，"z!"与字符串用法相同。

新的解析方式中，可以通过 Z_PARAM_ZVAL(dest)或 Z_PARAM_ZVAL_EX(dest, check_null, separate)宏替代。

11）其他标识符

除了上面介绍的这些解析符号，还有几个有特殊用法的标识符："|"、"+"、"*"，它们并不是用来表示某种数据类型的。

- |：表示此后的参数为可选参数，可以不传，比如解析规则为"al|b"，则可以传 2 个或 3 个参数，如果是"alb"，则必须传 3 个，否则将报错。新的解析方式中，可以使用 Z_PARAM_OPTIONAL 宏替代。例如：

```
ZEND_PARSE_PARAMETERS_START(2, 3)
    Z_PARAM_STR(name)
```

```
    Z_PARAM_ZVAL(val)
    Z_PARAM_OPTIONAL //表示此后的参数为可选的
    Z_PARAM_BOOL(non_cs)
ZEND_PARSE_PARAMETERS_END();
```

- +、*：用于可变参数，+、*的区别在于 * 表示可以不传可变参数，而 + 表示可变参数至少有一个。可变参数将被解析到 zval 数组，可以通过一个整型参数来获取具体的数量。新的解析方式中，可以使用 Z_PARAM_VARIADIC(spec, dest, dest_num)宏代替，其中 spec 为'*'或'+'，dest 为参数数组的起始位置，dest_num 为参数数量。例如：

```
PHP_FUNCTION(my_func_1)
{
zval *args;
int argc;

if (zend_parse_parameters(ZEND_NUM_ARGS(), "+", &args, &argc) == FAILURE) {
    return;
}
//...
}
```

argc 获取的就是可变参数的数量，args 为参数数组，指向第一个参数，可以通过 args[i]获取其他参数，比如这样传参：

```
my_func_1(array(), 1, false, "ddd");
```

那么传入的 4 个参数就可以在解析后通过 args[0]、args[1]、args[2]、args[3]获取。

10.6.3 引用传参

上一节介绍了如何在内部函数中解析从PHP用户空间传入的参数，这里一种情况没有讲到，那就是引用传参，在 PHP 中可以通过&符号将一个参数转为引用传入，例如：

```
php $a = array();

function my_func(&$a){
$a[] = 1;
}
```

上面这个例子在函数中对$a的修改将反映到原变量上，那么这种用法如何在内部函数中实现呢？用户自定义的函数中，参数通过zend_arg_info结构保存基本的信息，内部函数中也有类似的一个结构用于函数注册时指定参数的一些信息：zend_internal_arg_info。

```
typedef struct _zend_internal_arg_info {
    const char *name;                   //参数名
    const char *class_name;
    zend_uchar type_hint;               //显式声明的类型
    zend_uchar pass_by_reference;       //是否引用传参
    zend_bool allow_null;               //是否允许参数为NULL，类似"!"的用法
    zend_bool is_variadic;              //是否为可变参数
} zend_internal_arg_info;
```

这个结构几乎与zend_arg_info完全一样，不同的地方只在于name、class_name的类型，zend_arg_info这两个成员的类型都是zend_string。如果函数需要使用引用类型的参数或者返回引用，就需要创建函数的参数数组，这个数组通过ZEND_BEGIN_ARG_INFO()或ZEND_BEGIN_ARG_INFO_EX()、ZEND_END_ARG_INFO()宏定义：

```
#define ZEND_BEGIN_ARG_INFO_EX(name, _unused, return_reference, required_
num_args)
#define ZEND_BEGIN_ARG_INFO(name, _unused)
```

- **name**：参数数组名，注册函数PHP_FE(function, arg_info)会用到。
- **unused**：保留值，暂时无用。
- **return_reference**：返回值是否为引用，一般很少会用到。
- **required_num_args**：required参数数。

这两个宏需要与ZEND_END_ARG_INFO()配合使用：

```
ZEND_BEGIN_ARG_INFO_EX(arginfo_my_func_1, 0, 0, 2)
    ...
ZEND_END_ARG_INFO()
```

接着就是在上面两个宏中间定义每一个参数的zend_internal_arg_info，PHP提供的宏有：

```
//pass_by_ref表示是否引用传参，name为参数名称
```

```
    #define ZEND_ARG_INFO(pass_by_ref, name)    { #name, NULL, 0, pass_by_ref,
0, 0 },

    //只声明此参数为引用传参
    #define ZEND_ARG_PASS_INFO(pass_by_ref)     { NULL,  NULL, 0, pass_by_ref,
0, 0 },

    //显式声明此参数的类型为指定类的对象，等价于 PHP 中这样声明：MyClass $obj
    #define ZEND_ARG_OBJ_INFO(pass_by_ref, name, classname, allow_null)
{ #name, #classname, IS_OBJECT, pass_by_ref, allow_null, 0 },

    //显式声明此参数类型为数组，等价于：array $arr
    #define ZEND_ARG_ARRAY_INFO(pass_by_ref, name, allow_null)      { #name,
NULL, IS_ARRAY, pass_by_ref, allow_null, 0 },

    //显式声明为 callable，将检查函数、成员方法是否可调
    #define ZEND_ARG_CALLABLE_INFO(pass_by_ref, name, allow_null)   { #name,
NULL, IS_CALLABLE, pass_by_ref, allow_null, 0 },

    //通用宏，自定义各个字段
    #define ZEND_ARG_TYPE_INFO(pass_by_ref, name, type_hint, allow_null)
{ #name, NULL, type_hint, pass_by_ref, allow_null, 0 },

    //声明为可变参数
    #define ZEND_ARG_VARIADIC_INFO(pass_by_ref, name)    { #name, NULL, 0,
pass_by_ref, 0, 1 },
```

举个例子来看：

```
function my_func_1(&$a, Exception $c){
    ...
}
```

用内核实现则可以这么定义：

```
ZEND_BEGIN_ARG_INFO_EX(arginfo_my_func_1, 0, 0, 1)
    ZEND_ARG_INFO(1, a) //引用
    ZEND_ARG_OBJ_INFO(0, b, Exception, 0) //注意：这里不要把字符串加""
```

```
ZEND_END_ARG_INFO()
```

展开后：

```
static const zend_internal_arg_info name[] = {
    //多出来的这个是给返回值用的
    { (const char*)(zend_uintptr_t)(2), NULL, 0, 0, 0, 0 },
    { "a", NULL, 0, 0, 0, 0 },
    { "b", "Exception", 8, 1, 0, 0 },
}
```

第一个数组元素用于记录必传参数的数量以及返回值是否为引用。定义完这个数组后就需要把这个数组告诉函数：

```
const zend_function_entry mytest_functions[] = {
    PHP_FE(my_func_1,   arginfo_my_func_1)
    PHP_FE(my_func_2,   NULL)
    PHP_FE_END //末尾必须加这个
};
```

引用参数通过 zend_parse_parameters()解析时只能使用"z"解析，不能再直接解析为zend_value了，否则引用将失效：

```
PHP_FUNCTION(my_func_1)
{
    zval    *lval; //必须为zval，定义为zend_long也能解析出，但不是引用
    zval    *obj;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "zo", &lval, &obj) == FAILURE){
        RETURN_FALSE;
    }

    //lval的类型为IS_REFERENCE
    //获取实际引用的zval地址：&(lval.value->ref.val)
    zval *real_val = Z_REFVAL_P(lval);
    Z_LVAL_P(real_val) = 100; //设置实际引用的类型
}
```

```
$a = 90;
$b = new Exception;
my_func_1($a, $b);

echo $a;
```

参数数组与 zend_parse_parameters()有很多功能重合，两者都会生效，zend_internal_arg_info 主要用于传参阶段，为避免混乱，两者应该保持一致；另外，虽然内部函数的参数数组并不强制定义声明，但还是建议声明。

10.6.4 函数返回值

函数的返回值地址在调用内部函数时作为参数传入，即 return_value，如果函数有返回值直接设置此指针即可。这里需要特别注意引用计数的问题，如果把一个用到引用计数类型的变量赋给了返回值，那么就需要考虑是不是需要增加赋给返回值 value 的引用计数，例如：

```
PHP_FUNCTION(my_func_1)
{
    zval    *arr;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "a", &arr) == FAILURE){
        RETURN_FALSE;
    }
    //增加引用计数
    Z_ADDREF_P(arr);
    //设置返回值为数组:
    ZVAL_ARR(return_value, Z_ARR_P(arr));
}
```

此函数接收一个数组，然后直接返回该数组，数组多了一个来自返回值的引用，因此需要增加数组的引用计数。

虽然可以直接设置 return_value，但实际使用时并不建议这么做，PHP 提供了很多专门用于设置返回值的宏，这些宏定义在 zend_API.h 中：

```
//返回布尔型，b: IS_FALSE、IS_TRUE
#define RETURN_BOOL(b)                  { RETVAL_BOOL(b); return; }
```

```
//返回NULL
#define RETURN_NULL()                    { RETVAL_NULL(); return;}

//返回整型，l类型：zend_long
#define RETURN_LONG(l)                   { RETVAL_LONG(l); return; }

//返回浮点值，d类型：double
#define RETURN_DOUBLE(d)                 { RETVAL_DOUBLE(d); return; }

//返回字符串，可返回内部字符串，s类型为：zend_string *
#define RETURN_STR(s)                    { RETVAL_STR(s); return; }

//返回内部字符串，这种变量将不会被回收，s类型为：zend_string *
#define RETURN_INTERNED_STR(s)           { RETVAL_INTERNED_STR(s); return; }

//返回普通字符串，非内部字符串，s类型为：zend_string *
#define RETURN_NEW_STR(s)                { RETVAL_NEW_STR(s); return; }

//复制字符串用于返回，这个会自己加引用计数，s类型为：zend_string *
#define RETURN_STR_COPY(s)               { RETVAL_STR_COPY(s); return; }

//返回char *类型的字符串，s类型为char *，生成的zend_string会将s复制一份，具体操作：zend_string_init()
#define RETURN_STRING(s)                 { RETVAL_STRING(s); return; }

//返回char *类型的字符串，s类型为char *，l为字符串长度，类型为size_t
#define RETURN_STRINGL(s, l)             { RETVAL_STRINGL(s, l); return; }

//返回空字符串
#define RETURN_EMPTY_STRING()            { RETVAL_EMPTY_STRING(); return; }

//返回资源，r类型：zend_resource *
#define RETURN_RES(r)                    { RETVAL_RES(r); return; }

//返回数组，r类型：zend_array *
#define RETURN_ARR(r)                    { RETVAL_ARR(r); return; }

//返回对象，r类型：zend_object *
```

```
#define RETURN_OBJ(r)                    { RETVAL_OBJ(r); return; }

//返回 zval
#define RETURN_ZVAL(zv, copy, dtor)     { RETVAL_ZVAL(zv, copy, dtor); return; }

//返回 false
#define RETURN_FALSE                     { RETVAL_FALSE; return; }

//返回 true
#define RETURN_TRUE                      { RETVAL_TRUE; return; }
```

10.6.5 函数调用

实际应用中，扩展可能需要调用用户自定义的函数或者其他扩展定义的内部函数，前面章节已经介绍过函数的执行过程，这里不再重复，本节只介绍 PHP 提供的用于函数调用的 API 的使用，即 call_user_function()。

```
ZEND_API int call_user_function(HashTable *function_table,
    zval *object,
    zval *function_name,
    zval *retval_ptr,
    uint32_t param_count,
    zval params[]);
```

各参数的含义如下所述。

- **function_table**：函数符号表，函数就是 EG(function_table)，如果是成员方法，则是 zend_class_entry.function_table。
- **object**：调用成员方法的对象。
- **function_name**：调用的函数名称。
- **retval_ptr**：函数返回值地址。
- **param_count**：实际传参数量。
- **params**：参数数组。

从接口的定义看其使用还是很简单的，不需要我们关心执行过程中各阶段复杂的操作，下面从一个具体的例子看一下如何在内部函数中调用 PHP 用户空间的函数。

（1）在PHP中定义了一个普通的函数，将参数$i加上100后返回：

```
function mySum($i){
    return $i+100;
}
```

（2）接下来在扩展中调用这个函数：

```
PHP_FUNCTION(my_func)
{
    zend_long  i;
    zval       call_func_name, call_func_ret, call_func_params[1];
    uint32_t   call_func_param_cnt = 1;
    zend_string *call_func_str;
    char       *func_name = "mySum";

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "l", &i) == FAILURE){
        RETURN_FALSE;
    }

    //分配zend_string:调用完需要释放
    call_func_str = zend_string_init(func_name, strlen(func_name), 0);
    //设置到zval
    ZVAL_STR(&call_func_name, call_func_str);
    //设置参数
    ZVAL_LONG(&call_func_params[0], i);
    //调用
    if(SUCCESS != call_user_function(EG(function_table), NULL, &call_func_
name, &call_func_ret, call_func_param_cnt, call_func_params)){
        zend_string_release(call_func_str);
        RETURN_FALSE;
    }
    zend_string_release(call_func_str);
    RETURN_LONG(Z_LVAL(call_func_ret));
}
```

（3）最后调用这个内部函数：

```
function mySum($i){
    return $i+100;
}
echo my_func(60);
//执行后输出：160
```

call_user_function()并不是只能调用 PHP 脚本中定义的函数，内核或其他扩展注册的函数同样可以通过此函数调用，比如 array_merge()。

```
PHP_FUNCTION(my_func)
{
    zend_array  *arr1, *arr2;
    zval        call_func_name, call_func_ret, call_func_params[2];
    uint32_t    call_func_param_cnt = 2;
    zend_string *call_func_str;
    char        *func_name = "array_merge";

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "hh", &arr1, &arr2) == FAILURE){
        RETURN_FALSE;
    }
    //分配 zend_string
    call_func_str = zend_string_init(func_name, strlen(func_name), 0);
    //设置到 zval
    ZVAL_STR(&call_func_name, call_func_str);

    ZVAL_ARR(&call_func_params[0], arr1);
    ZVAL_ARR(&call_func_params[1], arr2);

    if(SUCCESS != call_user_function(EG(function_table), NULL, &call_func_
name, &call_func_ret, call_func_param_cnt, call_func_params)){
        zend_string_release(call_func_str);
        RETURN_FALSE;
    }
    zend_string_release(call_func_str);
    RETURN_ARR(Z_ARRVAL(call_func_ret));
}
```

然后通过以下脚本测试：

```
$arr1 = array(1,2);
$arr2 = array(3,4);

$arr = my_func($arr1, $arr2);
var_dump($arr);
```

你可能会注意到，上面的例子通过 call_user_function()调用函数时并没有增加两个数组参数的引用计数，但根据前面介绍的内容：函数传参时不会硬拷贝 value，而是增加参数 value 的引用计数，然后在函数 return 阶段再把引用减掉。实际上是 call_user_function()替我们完成了这个工作，下面简单看一下其处理过程。

```
int call_user_function(HashTable *function_table, zval *object, zval
*function_name, zval *retval_ptr, uint32_t param_count, zval params[])
{
    return call_user_function_ex(function_table, object, function_name,
retval_ptr, param_count, params, 1, NULL);
}

int call_user_function_ex(HashTable *function_table, zval *object, zval
*function_name, zval *retval_ptr, uint32_t param_count, zval params[], int
no_separation, zend_array *symbol_table)
{
    zend_fcall_info fci;
    //构建 zend_fcall_info 结构
    fci.size = sizeof(fci);
    fci.function_table = function_table;
    fci.object = object ? Z_OBJ_P(object) : NULL;
    ZVAL_COPY_VALUE(&fci.function_name, function_name);
    fci.retval = retval_ptr;
    fci.param_count = param_count;
    fci.params = params;
    fci.no_separation = (zend_bool) no_separation;
    fci.symbol_table = symbol_table;

    return zend_call_function(&fci, NULL);
}
```

call_user_function()将我们提供的参数组装为 zend_fcall_info 结构，然后调用 zend_call_function()进行处理，还记得 zend_parse_parameters()那个"f"解析符吗？它也是将输入的函数名称解析为一个 zend_fcall_info，可以更方便地调用函数，同时我们也可以自己创建一个 zend_fcall_info 结构，然后使用 zend_call_function()完成函数的调用。

```
int zend_call_function(zend_fcall_info *fci, zend_fcall_info_cache *fci_cache)
{
    ...
    for (i=0; i<fci->param_count; i++) {
        zval *param;
        zval *arg = &fci->params[i];
        ...
        //为参数添加引用
        if (Z_OPT_REFCOUNTED_P(arg)) {
            Z_ADDREF_P(arg);
        }
    }
    ...
    //调用的是用户函数
    if (func->type == ZEND_USER_FUNCTION) {
        //执行
        zend_init_execute_data(call, &func->op_array, fci->retval);
        zend_execute_ex(call);
    }else if (func->type == ZEND_INTERNAL_FUNCTION){ //内部函数
        if (EXPECTED(zend_execute_internal == NULL)) {
            func->internal_function.handler(call, fci->retval);
        } else {
            zend_execute_internal(call, fci->retval);
        }
    }
    ...
}
```

从上面的过程可以看到，函数调用前 zend_call_function()会遍历传入的参数，如果参数的类型用到了引用计数则会增加对应参数的引用计数。

10.7　Zval 的操作

扩展中经常会用到各种类型的 zval，PHP 提供了很多方法或宏用于不同类型 zval 的操作，尽管我们也可以自己操作 zval，但这并不是一个好习惯，因为 zval 有很多其他用途的标识，如果自己去管理这些值将是非常烦琐的一件事，很容易遗漏或者错误使用。因此，在扩展中操作 zval 时应该尽可能地使用内核提供的方法或者宏。

10.7.1　zval 的创建及获取

PHP7 将变量的引用计数转移到了具体的 value 上，所以 zval 更多的是作为统一的传输格式，很多情况下只是临时性使用，比如函数调用时的传参，最终需要的数据是 zval 携带的 zend_value，函数从 zval 取得 zend_value 后就不再关心 zval 了，这种就可以直接在栈上分配 zval。分配完 zval 需要设置其类型、value 等信息，PHP 中定义的 ZVAL_XXX()系列的宏就是用来设置不同类型 zval 的，这些宏的第一个参数 z 均为要设置的 zval 的指针，后面为要设置的 zend_value。对应的，内核提供了 Z_XXX(zval)、Z_XXX_P(zval_p)系列的宏，用于获取不同类型 zval 的 value。

zval 的类型可以通过 Z_TYPE(zval)、Z_TYPE_P(zval_p)两个宏获取，这个类型实际获取的就是 zval.u1.v.type，但是设置时不能只修改这个 type，而是要设置 typeinfo，因为 zval 还有其他的标识需要设置，比如是否使用引用计数、是否可被垃圾回收、是否可被复制等。

```
//file: zend_type.h
static zend_always_inline zend_uchar zval_get_type(const zval* pz) {
    return pz->u1.v.type;
}

//获取 zval 的类型
#define Z_TYPE(zval)                zval_get_type(&(zval))
#define Z_TYPE_P(zval_p)            Z_TYPE(*(zval_p))
```

1）UNDEF 及 NULL

通过 ZVAL_UNDEF(z)将 zval 设置为 UNDEF，ZVAL_NULL(z)设置为 NULL。通过以下宏判断 zval 是否为 UNDEF、NULL：

```
//是否为 undef
#define Z_ISUNDEF(zval)             (Z_TYPE(zval) == IS_UNDEF)
#define Z_ISUNDEF_P(zval_p)          Z_ISUNDEF(*(zval_p))
```

```
//是否为 null
#define Z_ISNULL(zval)              (Z_TYPE(zval) == IS_NULL)
#define Z_ISNULL_P(zval_p)          Z_ISNULL(*(zval_p))
```

2）布尔型

布尔型有两类，一类是老版本的那种方式：布尔型为 IS_BOOL，然后根据 0、1 区分 true、false，这种类型不再使用，另外一类是将 true、false 分别定义为一种类型，即 IS_TRUE、IS_FALSE。内核提供了三个宏定义布尔型：ZVAL_BOOL(z, b)，b 为 IS_TRUE、IS_FALSE；ZVAL_FALSE(z)设置为 false；ZVAL_TRUE(z)设置为 true。

如果想判断一个 zval 是否为布尔型，直接根据 Z_TYPE(zval)、Z_TYPE_P(zval_p)获取其类型进行判断即可。

3）整型、浮点型

整型与浮点型分别通过 ZVAL_LONG(z, l)、ZVAL_DOUBLE(z, d)设置，l、d 为 zend_long、double。对应的，其值可通过 Z_LVAL(zval)/Z_LVAL_P(zval_p)、Z_DVAL(zval)/Z_DVAL_P(zval_p)取到。

4）字符串

将 zval 设置为字符串的情况比较多，因为字符串在内核中分为 char、zend_string 两类，而且还有内部字符串。比较通用的一个宏是 ZVAL_STR(z, s)，s 为 zend_string 类型的地址，这里并不会将 s 的字符串复制一份，只是将 s 地址设置到新的 zval，s 的引用计数不会自动增加，如果需要增加引用计数，则需要手动增加，这个宏支持内部字符串，如果 s 为内部字符串，则只会设置 zval 的 type 类型：

```
#define ZVAL_STR(z, s) do {                                 \
        zval *__z = (z);                                    \
        zend_string *__s = (s);                             \
        Z_STR_P(__z) = __s;                                 \
        /* 支持内部字符串 */          \
        Z_TYPE_INFO_P(__z) = ZSTR_IS_INTERNED(__s) ? \
            IS_INTERNED_STRING_EX :                         \
            IS_STRING_EX;                                   \
    } while (0)
```

另外一个类似的宏为 ZVAL_NEW_STR(z, s)，除了不支持内部字符串，与 ZVAL_STR(z, s)完全一样，这里说的不支持内部字符串是指：如果把内部字符串赋给 zval，这个宏会把该字符串当作普通字符串对待。

除了上面这两个宏，还有一个会自动增加引用计数的宏：ZVAL_STR_COPY(z, s)，与ZVAL_STR(z, s)操作一致，只是会根据 s 的类型决定是否需要增加引用计数。

```
#define ZVAL_STR_COPY(z, s) do {                                    \
        zval *__z = (z);                                            \
        zend_string *__s = (s);                                     \
        Z_STR_P(__z) = __s;                                         \
        /* interned strings support */                              \
        if (ZSTR_IS_INTERNED(__s)) {                                    \
            Z_TYPE_INFO_P(__z) = IS_INTERNED_STRING_EX;             \
        } else {                                                    \
            GC_REFCOUNT(__s)++;                                     \
            Z_TYPE_INFO_P(__z) = IS_STRING_EX;                      \
        }                                                           \
    } while (0)
```

获取字符串的操作也比较多：Z_STR(zval)、Z_STR_P(zval_p)可以获取字符串 zend_string 的地址；Z_STRVAL(zval)、Z_STRVAL_P(zval_p)获取的是 char 字符串，即 zend_string→val；Z_STRLEN(zval)、Z_STRLEN_P(zval_p)获取字符串长度，即 zend_string→len；Z_STRHASH(zval)、Z_STRHASH_P(zval_p)获取字符串的哈希值，即 zend_string→h。

5）数组

如果已经分配了 zend_array 结构，则可以通过 ZVAL_ARR(z, a)赋给 zval，a 就是 zend_array 地址，该宏不会增加 a 的引用计数；如果是新分配一个空数组，则可以通过 ZVAL_NEW_ARR(z)完成，需要注意的是，这个宏这是分配了 zend_array 内存，但是并没有进行数组的初始化，也就是说此时数组还不能使用，关于数组初始化的操作接下来会单独说明。

```
#define ZVAL_NEW_ARR(z) do {                                        \
        zval *__z = (z);                                            \
        zend_array *_arr =                                          \
        (zend_array *) emalloc(sizeof(zend_array));                 \
        Z_ARR_P(__z) = _arr;                                        \
        Z_TYPE_INFO_P(__z) = IS_ARRAY_EX;                           \
    } while (0)
```

数组可以通过这些宏获取 zend_array 地址：Z_ARR(zval)、Z_ARR_P(zval_p)、

Z_ARRVAL(zval)、Z_ARRVAL_P(zval_p)。

6）对象

对象可以通过 ZVAL_OBJ(z, o)宏设置，o 为 zend_object 地址，该宏不会增加 o 的引用计数。获取对象的操作比较多，首先是获取 zend_object 地址的操作，可以通过 Z_OBJ(zval)、Z_OBJ_P(zval_p)取到，另外，还有以下几个宏用于获取对象的其他信息：

```
//获取对象的 zend_object_handlers，即 zend_object→handlers
#define Z_OBJ_HT(zval)              Z_OBJ(zval)→handlers
#define Z_OBJ_HT_P(zval_p)          Z_OBJ_HT(*(zval_p))
// 获取对象对应操作的 handler 的指针 write_property、read_property 等，注意：这个宏取到的值为只读，不要试图修改这个值，如 Z_OBJ_HANDLER(obj, write_property) = xxx，因为对象的 handlers 成员前加了 const 修饰符无法修改
#define Z_OBJ_HANDLER(zval, hf)     Z_OBJ_HT((zval))→hf
#define Z_OBJ_HANDLER_P(zv_p, hf)   Z_OBJ_HANDLER(*(zv_p), hf)
//获取对象的 handle，即 zend_object→handle
#define Z_OBJ_HANDLE(zval)          (Z_OBJ((zval)))→handle
#define Z_OBJ_HANDLE_P(zval_p)      Z_OBJ_HANDLE(*(zval_p))
//获取对象实例化的类，即 zend_object→ce
#define Z_OBJCE(zval)               (Z_OBJ(zval)→ce)
#define Z_OBJCE_P(zval_p)           Z_OBJCE(*(zval_p))
//获取对象的成员数组，返回的是 HashTable 地址
#define Z_OBJPROP(zval)         Z_OBJ_HT((zval))→get_properties(&(zval))
#define Z_OBJPROP_P(zval_p)         Z_OBJPROP(*(zval_p))
```

7）资源

资源可通过 ZVAL_RES(z, r)宏设置，r 为 zend_resource 地址，该宏也不会增加 r 的引用计数。对应的，可通过 Z_RES(zval)、Z_RES_P(zval_p)宏获取 zend_resource 地址，除了这两个宏，还可以通过 Z_RES_HANDLE(zval)、Z_RES_HANDLE_P(zval_p)获取资源的 handle，即 zend_resource→handle。

8）引用

如果已经有 zend_reference 结构了，则可通过 ZVAL_REF(z, r)宏直接设置到 zval，r 为 zend_reference 地址，该宏不会增加 r 的引用计数；如果没有 zend_reference 结构，可以通过 ZVAL_NEW_REF(z, r)创建一个引用，r 为要引用的值：

```
#define ZVAL_NEW_REF(z, r) do {                                 \
```

```
    zend_reference *_ref =                                    \
    (zend_reference *) emalloc(sizeof(zend_reference)); \
    GC_REFCOUNT(_ref) = 1;                                    \
    GC_TYPE_INFO(_ref) = IS_REFERENCE;                        \
    ZVAL_COPY_VALUE(&_ref->val, r);                           \
    Z_REF_P(z) = _ref;                                        \
    Z_TYPE_INFO_P(z) = IS_REFERENCE_EX;                       \
} while (0)
```

可以通过Z_REF(zval)、Z_REF_P(zval_p)获取zend_reference地址，如果要获取引用的实际value，则可使用Z_REFVAL(zval)、Z_REFVAL_P(zval_p)获取。

除了上面这些与PHP变量类型相关的宏，还有一些内核自己使用类型的宏，因为zval是一个通用型的结构，可以用它保存任意类型，比如类、函数等：

```
//获取 indirect 的 zval，指向另一个 zval
#define Z_INDIRECT(zval)          (zval).value.zv
#define Z_INDIRECT_P(zval_p)      Z_INDIRECT(*(zval_p))
//获取 zval 的 zend_class_entry
#define Z_CE(zval)                (zval).value.ce
#define Z_CE_P(zval_p)            Z_CE(*(zval_p))
//获取 zval 的 zend_function
#define Z_FUNC(zval)              (zval).value.func
#define Z_FUNC_P(zval_p)          Z_FUNC(*(zval_p))
//获取 zval 保存的 ptr
#define Z_PTR(zval)               (zval).value.ptr
#define Z_PTR_P(zval_p)           Z_PTR(*(zval_p))
```

10.7.2 变量复制

扩展中经常会遇到将一个变量复制到另外一个变量的场景，这个操作可以通过ZVAL_COPY(z, v)完成，z为目标zval，复制后这个宏会增加对应value的引用计数，除这个宏之外，还有一个功能与之相同，但不会主动增加引用计数的宏：ZVAL_COPY_VALUE(z, v)。

```
#define ZVAL_COPY_VALUE(z, v)                \
    do {                                     \
        zval *_z1 = (z);                     \
        const zval *_z2 = (v);               \
```

```
        zend_refcounted *_gc = Z_COUNTED_P(_z2);          \
        uint32_t _t = Z_TYPE_INFO_P(_z2);                 \
        ZVAL_COPY_VALUE_EX(_z1, _z2, _gc, _t);            \
    } while (0)

#define ZVAL_COPY(z, v)                                   \
    do {                                                  \
        zval *_z1 = (z);                                  \
        const zval *_z2 = (v);                            \
        zend_refcounted *_gc = Z_COUNTED_P(_z2);          \
        uint32_t _t = Z_TYPE_INFO_P(_z2);                 \
        ZVAL_COPY_VALUE_EX(_z1, _z2, _gc, _t);            \
        if ((_t & (IS_TYPE_REFCOUNTED << Z_TYPE_FLAGS_SHIFT)) != 0) { \
            GC_REFCOUNT(_gc)++;                           \
        }                                                 \
    } while (0)
```

从宏的展开结果可以看到，复制时并不会发生硬拷贝，只是将 value 指针复制到了目标 zval，同时还会把变量的类型信息一同复制给了目标变量。具体 value 的复制在 ZVAL_COPY_VALUE_EX()宏中完成。

```
# define ZVAL_COPY_VALUE_EX(z, v, gc, t)                  \
    do {                                                  \
        Z_COUNTED_P(z) = gc;                              \
        Z_TYPE_INFO_P(z) = t;                             \
    } while (0)
```

你可能发现，这里并没有看到具体 value 的复制，实际 value 的复制就是 Z_COUNTED_P(z) = gc。这是因为 zend_value 是一个联合体，因此可以通过联合体中的任意成员获取到对应 value 的内存地址，联合体中的不同成员名称可以看作同一段内存的别名，使用任意一个都可以获取内存的地址。这里使用 zend_value→counted 获取不同类型的 value，这样一来，复制 value 时就不需要关心具体的 value 类型了，否则根据各个类型进行不同的复制操作。

10.7.3 引用计数

在扩展中操作与 PHP 用户空间相关的变量时，需要谨慎考虑是否需要引用计数相关的操作，比如下面这个例子：

```
function test($arr){
    return $arr;
}
$a = array(1,2);
$b = test($a);
```

如果把函数 test()用内部函数实现，这个函数接受了一个 PHP 用户空间传入的数组参数，然后又返回并赋值给了PHP用户空间的另外一个变量，这个时候就需要增加传入数组的refcount。数组由 PHP 用户空间分配，函数调用前 refcount=1，传到内部函数时相当于赋值给了函数的参数，因此 refcount 增加了 1 变为 2，这次增加在函数执行完释放参数时会减掉，等返回并赋值给 $b 后，此时共有两个变量指向这个数组，所以内部函数需要增加 refcount，增加的引用是给返回值的。test()翻译成内部函数：

```
PHP_FUNCTION(test)
{
    zval    *arr;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "a", &arr) == FAILURE){
        RETURN_FALSE;
    }
    //如果注释掉下面这句将导致 core dumped
    Z_TRY_ADDREF_P(arr);
    RETURN_ARR(Z_ARR_P(arr));
}
```

那么在哪些情况下需要考虑设置引用计数呢？一个关键条件是：操作的是与 PHP 用户空间相关的变量，包括对用户空间变量的修改、赋值，要明确的一点是引用计数是用来解决多个变量指向同一个 value 问题的，所以在 PHP 中来回传递 zval 的时候就需要考虑是不是要修改引用计数。下面总结 PHP 中常见的会对引用计数进行操作的情况：

- **变量赋值**：变量赋值是最常见的情况，一个用到引用计数的变量类型在初始赋值时其refcount=1，如果后面把此变量又赋值给了其他变量，那么就需要增加其引用计数。
- **数组操作**：如果把一个变量插入数组中，那么就需要增加这个变量的引用计数，如果要删除一个数组元素，则要相应地减少其引用。
- **函数调用**：传参实际可以当作普通的变量赋值，将调用空间的变量赋值给被调函数空间的变量，函数返回时会销毁函数空间的变量，这时又会减掉传参的引用，这两个过

程由内核完成，不需要扩展自己处理。

- **成员属性**：当把一个变量赋值给对象的成员属性时需要增加引用计数。

PHP 中定义了以下宏用于引用计数的操作，包括获取变量的引用计数以及增加、减少变量的引用计数：

```
//file: zend_types.h
//获取引用数：pz 类型为 zval*
#define Z_REFCOUNT_P(pz)             zval_refcount_p(pz)
//设置引用数
#define Z_SET_REFCOUNT_P(pz, rc)     zval_set_refcount_p(pz, rc)
//增加引用
#define Z_ADDREF_P(pz)               zval_addref_p(pz)
//减少引用
#define Z_DELREF_P(pz)               zval_delref_p(pz)

#define Z_REFCOUNT(z)                Z_REFCOUNT_P(&(z))
#define Z_SET_REFCOUNT(z, rc)        Z_SET_REFCOUNT_P(&(z), rc)
#define Z_ADDREF(z)                  Z_ADDREF_P(&(z))
#define Z_DELREF(z)                  Z_DELREF_P(&(z))

//只对使用了引用计数的变量类型增加引用，建议使用这个
#define Z_TRY_ADDREF_P(pz) do {      \
    if (Z_REFCOUNTED_P((pz))) {      \
        Z_ADDREF_P((pz));            \
    }                                \
} while (0)

#define Z_TRY_DELREF_P(pz) do {      \
    if (Z_REFCOUNTED_P((pz))) {      \
        Z_DELREF_P((pz));            \
    }                                \
} while (0)

#define Z_TRY_ADDREF(z)              Z_TRY_ADDREF_P(&(z))
#define Z_TRY_DELREF(z)              Z_TRY_DELREF_P(&(z))
```

上面这些宏操作的类型都是 zval 或 zval*，如果需要操作具体 value 的引用计数可以使用以下宏：

```
//直接获取 zend_value 的引用，可以直接通过这个宏修改 value 的 refcount
#define GC_REFCOUNT(p)          (p)->gc.refcount
```

另外还有几个常用的宏：

```
//判断 zval 是否用到引用计数机制
#define Z_REFCOUNTED(zval) ((Z_TYPE_FLAGS(zval) & IS_TYPE_REFCOUNTED) != 0)
#define Z_REFCOUNTED_P(zval_p)      Z_REFCOUNTED(*(zval_p))
//根据 zval 获取 value 的 zend_refcounted 头部
#define Z_COUNTED(zval)             (zval).value.counted
#define Z_COUNTED_P(zval_p)         Z_COUNTED(*(zval_p))
```

10.7.4 字符串操作

zend_string 是 PHP 自己封装的一个用于表示字符串的结构，也是使用比较频繁的一种类型，这里单独介绍与 zend_string 操作相关的宏及函数：

```
//file: zend_string.h
//创建 zend_string
zend_string *zend_string_init(const char *str, size_t len, int persistent);
//字符串复制，只增加引用
zend_string *zend_string_copy(zend_string *s);
//字符串拷贝，硬拷贝
zend_string *zend_string_dup(zend_string *s, int persistent);
//将字符串按 len 大小重新分配，会减少 s 的 refcount，返回新的字符串
zend_string *zend_string_realloc(zend_string *s, size_t len, int
persistent);
//延长字符串，与 zend_string_realloc()类似，不同的是 len 不能小于 s 的长度
zend_string *zend_string_extend(zend_string *s, size_t len, int
persistent);
//截断字符串，与 zend_string_realloc()类似，不同的是 len 不能大于 s 的长度
zend_string *zend_string_truncate(zend_string *s, size_t len, int
persistent);
//获取字符串 refcount
```

```
    uint32_t zend_string_refcount(const zend_string *s);
    //增加字符串 refcount
    uint32_t zend_string_addref(zend_string *s);
    //减少字符串 refcount
    uint32_t zend_string_delref(zend_string *s);
    //释放字符串，减少 refcount，为 0 时销毁
    void zend_string_release(zend_string *s);
    //销毁字符串，不管引用计数是否为 0
    void zend_string_free(zend_string *s);
    //比较两个字符串是否相等，区分大小写，memcmp()
    zend_bool zend_string_equals(zend_string *s1, zend_string *s2);
    //比较两个字符串是否相等，不区分大小写
    #define zend_string_equals_ci(s1, s2) \
        (ZSTR_LEN(s1) == ZSTR_LEN(s2) && !zend_binary_strcasecmp(ZSTR_VAL(s1),
ZSTR_LEN(s1), ZSTR_VAL(s2), ZSTR_LEN(s2)))

    //其他宏，zstr 类型为 zend_string*
    #define ZSTR_VAL(zstr)  (zstr)->val //获取字符串
    #define ZSTR_LEN(zstr)  (zstr)->len //获取字符串长度
    #define ZSTR_H(zstr)    (zstr)->h   //获取字符串哈希值
    #define ZSTR_HASH(zstr) zend_string_hash_val(zstr) //计算字符串哈希值
```

除了上面这些，还有很多其他 API 定义在 zend_operators.h 中，例如字符串大小转换、字符串比较等，这里不再列举，使用时可以单独看一下其具体处理逻辑。

10.7.5 数组操作

10.7.5.1 创建数组

创建一个新的 HashTable 分为两步：首先是分配 zend_array 内存，这个可以通过 ZVAL_NEW_ARR()宏分配，也可以自己直接分配；然后初始化数组，通过 zend_hash_init()宏完成，如果不进行初始化数组将无法使用。

```
    //file: zend_hash.h
    #define zend_hash_init(ht, nSize, pHashFunction, pDestructor, persistent) \
        _zend_hash_init((ht), (nSize), (pDestructor), (persistent) ZEND_FILE_
LINE_CC)
```

- **ht**：数组地址 HashTable*，如果内部使用可以直接通过 emalloc 分配。
- **nSize**：初始化大小，只是参考值，这个值会被对齐到 2^n，最小为 8。
- **pHashFunction**：无用，设置为 NULL 即可。
- **pDestructor**：删除或更新数组元素以及数组销毁时会调用这个函数对操作的元素进行处理，比如将一个字符串插入数组，字符串的 refcount 增加，删除时不是简单地将元素的 Bucket 删除就可以了，还需要对其 refcount 进行处理，这个函数就是进行清理工作的。
- **persistent**：是否持久化。注意：持久化数组只能在内部使用，不能传递给 PHP 用户空间，如果为持久化，则会调用系统 malloc 分配内存，同时，数组的 key、arrData 也都是通过 malloc 分配的。另外需要注意的是，如果这里用持久化，那么在 ht 分配的时候也应该使用持久化的方式。

例如：

```
zval        array;
uint32_t    size;

ZVAL_NEW_ARR(&array);
zend_hash_init(Z_ARRVAL(array), size, NULL, ZVAL_PTR_DTOR, 0);
```

这个例子中就不能在 zend_hash_init()时声明为持久化，否则数组在使用时会发生异常，因为 ZVAL_NEW_ARR()分配 zend_array 时是非持久化的，可以使用 ZVAL_NEW_PERSISTENT_ARR(z)代替。

10.7.5.2 插入、更新元素

数组元素的插入、更新主要有三种情况：key 为 zend_string、key 为普通字符串、key 为数值索引。下面是相关的宏及函数，pData 为 zval 地址。

1）key 为 zend_string

```
//file: zend_hash.h
//插入或更新元素，会增加 key 的 refcount
#define zend_hash_update(ht, key, pData) \
        _zend_hash_update(ht, key, pData ZEND_FILE_LINE_CC)

//插入或更新元素，当 Bucket 类型为 indirect 时，将 pData 更新至 indirect 的值，而不是更新 Bucket
#define zend_hash_update_ind(ht, key, pData) \
```

```
        _zend_hash_update_ind(ht, key, pData ZEND_FILE_LINE_CC)

//添加元素，与 zend_hash_update()类似，不同的地方在于如果元素已经存在则不会更新
#define zend_hash_add(ht, key, pData) \
        _zend_hash_add(ht, key, pData ZEND_FILE_LINE_CC)

//直接插入元素，不管 key 存在与否，如果存在也不覆盖原来元素，而是当做哈希冲突处理，所有会出现一个数组中 key 相同的情况，慎用!!!
#define zend_hash_add_new(ht, key, pData) \
        _zend_hash_add_new(ht, key, pData ZEND_FILE_LINE_CC)
```

2）key 为 char 字符串

```
//与上面几个对应，这里的 key 为普通字符串，会自动生成 zend_string 的 key
#define zend_hash_str_update(ht, key, len, pData) \
        _zend_hash_str_update(ht, key, len, pData ZEND_FILE_LINE_CC)
#define zend_hash_str_update_ind(ht, key, len, pData) \
        _zend_hash_str_update_ind(ht, key, len, pData ZEND_FILE_LINE_CC)
#define zend_hash_str_add(ht, key, len, pData) \
        _zend_hash_str_add(ht, key, len, pData ZEND_FILE_LINE_CC)
#define zend_hash_str_add_new(ht, key, len, pData) \
        _zend_hash_str_add_new(ht, key, len, pData ZEND_FILE_LINE_CC)
```

3）key 为数值索引

```
//插入元素，h 为数值
#define zend_hash_index_add(ht, h, pData) \
        _zend_hash_index_add(ht, h, pData ZEND_FILE_LINE_CC)

//与 zend_hash_add_new()类似
#define zend_hash_index_add_new(ht, h, pData) \
        _zend_hash_index_add_new(ht, h, pData ZEND_FILE_LINE_CC)

//更新第 h 个元素
#define zend_hash_index_update(ht, h, pData) \
        _zend_hash_index_update(ht, h, pData ZEND_FILE_LINE_CC)

//使用自动索引值
#define zend_hash_next_index_insert(ht, pData) \
        _zend_hash_next_index_insert(ht, pData ZEND_FILE_LINE_CC)
```

```
#define zend_hash_next_index_insert_new(ht, pData) \
        _zend_hash_next_index_insert_new(ht, pData ZEND_FILE_LINE_CC)
```

10.7.5.3 查找元素

与插入元素对应，查找元素根据 key 的类型分为三种：

```
//根据 zend_string key 查找数组元素
ZEND_API zval* ZEND_FASTCALL zend_hash_find(const HashTable *ht, zend_
string *key);
//根据普通字符串 key 查找元素
ZEND_API zval* ZEND_FASTCALL zend_hash_str_find(const HashTable *ht, const
char *key, size_t len);
//获取数值索引元素
ZEND_API zval* ZEND_FASTCALL zend_hash_index_find(const HashTable *ht,
zend_ulong h);
//判断元素是否存在
ZEND_API zend_bool ZEND_FASTCALL zend_hash_exists(const HashTable *ht,
zend_string *key);
ZEND_API zend_bool ZEND_FASTCALL zend_hash_str_exists(const HashTable *ht,
const char *str, size_t len);
ZEND_API zend_bool ZEND_FASTCALL zend_hash_index_exists(const HashTable *ht,
zend_ulong h);
//获取数组元素数
#define zend_hash_num_elements(ht) \
    (ht)->nNumOfElements
//与 zend_hash_num_elements()类似，会有一些特殊处理
ZEND_API uint32_t zend_array_count(HashTable *ht);
```

10.7.5.4 删除元素

```
//根据 zend_string 类型 key 删除元素
ZEND_API int ZEND_FASTCALL zend_hash_del(HashTable *ht, zend_string *key);
//与 zend_hash_del()类似，不同的地方是如果元素类型为 indirect 则同时销毁 indirect
的值
ZEND_API int ZEND_FASTCALL zend_hash_del_ind(HashTable *ht, zend_string
*key);
//根据 char 字符串 key 删除元素
ZEND_API int ZEND_FASTCALL zend_hash_str_del(HashTable *ht, const char *key,
```

```
size_t len);
    //同上一个，如果元素类型为 indirect 则同时销毁 indirect 的值
    ZEND_API int ZEND_FASTCALL zend_hash_str_del_ind(HashTable *ht, const char
*key, size_t len);
    //根据数值索引删除元素
    ZEND_API int ZEND_FASTCALL zend_hash_index_del(HashTable *ht, zend_ulong
h);
    //删除 bucket
    ZEND_API void ZEND_FASTCALL zend_hash_del_bucket(HashTable *ht, Bucket *p);
```

10.7.5.5 遍历

数组遍历类似 foreach 的用法，在扩展中可以通过如下方式进行遍历：

```
zval *val;
ZEND_HASH_FOREACH_VAL(ht, val) {
    ...
} ZEND_HASH_FOREACH_END();
```

遍历过程中会把数组元素赋值给 val，除了上面这个宏，还有很多其他用于遍历的宏，这里列几个比较常用的：

```
//遍历获取所有的数值索引
#define ZEND_HASH_FOREACH_NUM_KEY(ht, _h) \
    ZEND_HASH_FOREACH(ht, 0); \
    _h = _p->h;

//遍历获取所有的 key
#define ZEND_HASH_FOREACH_STR_KEY(ht, _key) \
    ZEND_HASH_FOREACH(ht, 0); \
    _key = _p->key;

//上面两个的聚合
#define ZEND_HASH_FOREACH_KEY(ht, _h, _key) \
    ZEND_HASH_FOREACH(ht, 0); \
    _h = _p->h; \
    _key = _p->key;

//遍历获取数值索引 key 及 value
```

```
#define ZEND_HASH_FOREACH_NUM_KEY_VAL(ht, _h, _val) \
    ZEND_HASH_FOREACH(ht, 0); \
    _h = _p->h; \
    _val = _z;

//遍历获取 key 及 value
#define ZEND_HASH_FOREACH_STR_KEY_VAL(ht, _key, _val) \
    ZEND_HASH_FOREACH(ht, 0); \
    _key = _p->key; \
    _val = _z;

//遍历获取数值、字符串 key 及 value
#define ZEND_HASH_FOREACH_KEY_VAL(ht, _h, _key, _val) \
    ZEND_HASH_FOREACH(ht, 0); \
    _h = _p->h; \
    _key = _p->key; \
    _val = _z;
```

10.7.5.6 其他操作

数组除了上面介绍的这些基本使用，还有很多其他的操作，可以去 zend_hash.c 文件中详细看下，比如数组合并、导出、排序等。

```
//合并两个数组，将 source 合并到 target，overwrite 为元素冲突时是否覆盖
#define zend_hash_merge(target, source, pCopyConstructor, overwrite)    \
    _zend_hash_merge(target, source, pCopyConstructor, overwrite ZEND_FILE_
LINE_CC)

//导出数组，不会硬拷贝数组元素
ZEND_API HashTable* ZEND_FASTCALL zend_array_dup(HashTable *source);

//排序
#define zend_hash_sort(ht, compare_func, renumber) \
    zend_hash_sort_ex(ht, zend_sort, compare_func, renumber)
```

数组排序：compare_func 为 typedef int (*compare_func_t)(const void*, const void)，需要自己定义比较函数，参数类型为 Bucket，renumber 表示是否更改键值，如果为 1 则会在排序后重新生成各元素的 h。PHP 中的 sort()、rsort()、ksort()等都是基于这个函数实现的。

10.7.5.7 销毁

数组销毁时会把所有的 key、value 释放掉，如果数组没有自定义 pDestructor，则默认调用 i_zval_ptr_dtor()对 value 进行销毁。

```
ZEND_API void ZEND_FASTCALL zend_array_destroy(HashTable *ht);
```

10.8 常量

PHP 提供了很多用于不同类型常量注册的宏，这些宏定义在 zend_constants.h 头文件中，扩展中通常会在 PHP_MINIT_FUNCTION()中定义常量。

```
//注册 NULL 常量
#define REGISTER_NULL_CONSTANT(name, flags)  \
    zend_register_null_constant((name), sizeof(name)-1, (flags), module_number)

//注册 bool 常量
#define REGISTER_BOOL_CONSTANT(name, bval, flags)  \
    zend_register_bool_constant((name), sizeof(name)-1, (bval), (flags),
module_number)

//注册整型常量
#define REGISTER_LONG_CONSTANT(name, lval, flags)  \
    zend_register_long_constant((name), sizeof(name)-1, (lval), (flags),
module_number)

//注册浮点型常量
#define REGISTER_DOUBLE_CONSTANT(name, dval, flags)  \
    zend_register_double_constant((name), sizeof(name)-1, (dval), (flags),
module_number)

//注册字符串常量，str 类型为 char*
#define REGISTER_STRING_CONSTANT(name, str, flags)  \
    zend_register_string_constant((name), sizeof(name)-1, (str), (flags),
module_number)

//注册字符串常量，截取指定长度，str 类型为 char*
#define REGISTER_STRINGL_CONSTANT(name, str, len, flags)  \
```

```
    zend_register_stringl_constant((name), sizeof(name)-1, (str), (len),
(flags), module_number)
```

除了上面这些，还有 REGISTER_NS_XXX 系列的宏用于带 namespace 的常量注册，另外如果这些类型不能满足需求，可以通过 zend_register_constant(zend_constant *c)注册，比如数组类型的常量。

举例来看：

```
PHP_MINIT_FUNCTION(mytest)
{
    ...
    REGISTER_STRING_CONSTANT("MY_CONS_1", "this is a constant", CONST_CS |
CONST_PERSISTENT);
}
```

编译后在 PHP 代码中使用这个常量 echo MY_CONS_1，将输出“this is a constant”。

如果在扩展中需要用到其他扩展或内核定义的常量，则可以通过以下函数获取常量的值：

```
//通过 zend_string 的常量名获取常量
ZEND_API zval *zend_get_constant(zend_string *name);
//通过 char 常量名获取常量
ZEND_API zval *zend_get_constant_str(const char *name, size_t name_len);
```

10.9 面向对象

面向对象是 PHP 重要的一个特性之一，通过扩展可以对类实现更灵活的、更强大的控制，同时与 PHP 脚本中的类相比，通过扩展实现的类不需要编译，有更高的性能。本节我们将介绍如何在扩展中实现面向对象的相关操作。

10.9.1 内部类注册

在扩展中定义一个内部类的方式与函数类似，函数最终注册到 EG(function_table)，而类则最终注册到 EG(class_table)符号表中，注册的过程首先是为类创建一个 zend_class_entry 结构，然后把这个结构插入 EG(class_table)，当然这个过程不需要我们手动操作，PHP 提供了现成的方法和宏帮我们对 zend_class_entry 进行初始化与注册。通常情况下会把内部类的注册放到 module startup 阶段，也就是定义在扩展的 PHP_MINIT_FUNCTION()中。一个简单的类的注册

只需要以下几行：

```
PHP_MINIT_FUNCTION(mytest)
{
    //分配一个 zend_class_entry，这个结构只在注册时使用，所以分配在栈上即可
    zend_class_entry ce;
    //对 zend_class_entry 进行初始化
    INIT_CLASS_ENTRY(ce, "MyClass", NULL);
    //注册
    zend_register_internal_class(&ce);
}
```

这样就成功定义了一个内部类，类名为“MyClass”，只是这个类还没有任何的成员属性、成员方法，定义完成后重新编译、安装扩展，然后在 PHP 脚本中实例化这个类：

```
$obj = new MyClass();

var_dump($obj);
```

类的注册正是通过 zend_register_internal_class()完成的，它的参数非常简单，只有一个 zend_class_entry，我们只需要在扩展中将这个结构定义好，然后交给内核完成注册即可。需要注意的是，注册时传入的 zend_class_entry 并不是最终插入 EG(class_table)的结构，这里只是临时的，内核自己会重新分配，所以直接分配在栈上即可。注册完成后 zend_register_internal_class()将返回实际的 zend_class_entry 地址，扩展可以将这个地址保存下来，后续用到类的地方可以直接使用，不需要再通过 EG(class_table)查找：

```
//注册的类的 zend_class_entry 地址
zend_class_entry* myclass_ce;

PHP_MINIT_FUNCTION(mytest)
{
    ...
    myclass_ce = zend_register_internal_class(&ce);
}
```

另外，内核提供了两个宏用于 zend_class_entry 的初始化，不需要手动设置各个成员：

```
    /**
     * 初始化 zend_class_entry
     * class_container: zend_class_entry 地址
     * class_name: 类名
     * functions: 成员方法数组
     */
    #define INIT_CLASS_ENTRY(class_container, class_name, functions) \
        INIT_OVERLOADED_CLASS_ENTRY(class_container, class_name, functions,
NULL, NULL, NULL)

    /**
     * 初始化 zend_class_entry, 带 namespace
     * class_container: zend_class_entry 地址
     * ns: 命名空间
     * class_name: 类名
     * functions: 成员方法数组
     */
    #define INIT_NS_CLASS_ENTRY(class_container, ns, class_name, functions) \
        INIT_CLASS_ENTRY(class_container, ZEND_NS_NAME(ns, class_name), functions)
```

10.9.2 成员属性

成员属性的操作接口在 zend_API.h 中，主要包括成员属性的声明、修改、获取三个操作。

1. 声明成员属性

PHP 提供了各种标量类型的成员属性声明接口，具体接口如下：

```
    //property 为属性值，可以是任意类型，需要注意：内部类初始化时 value 不能是数组、对象、
资源
    ZEND_API int zend_declare_property(zend_class_entry *ce, const char *name,
size_t name_length, zval *property, int access_type);
    //声明 NULL 型属性
    ZEND_API int zend_declare_property_null(zend_class_entry *ce, const char
*name, size_t name_length, int access_type);
    //声明布尔型属性
    ZEND_API int zend_declare_property_bool(zend_class_entry *ce, const char
*name, size_t name_length, zend_long value, int access_type);
```

```
    //声明整型属性
    ZEND_API int zend_declare_property_long(zend_class_entry *ce, const char
*name, size_t name_length, zend_long value, int access_type);
    //声明浮点型属性
    ZEND_API int zend_declare_property_double(zend_class_entry *ce, const char
*name, size_t name_length, double value, int access_type);
    //声明字符串型属性
    ZEND_API int zend_declare_property_string(zend_class_entry *ce, const char
*name, size_t name_length, const char *value, int access_type);
    //声明带长度的字符串属性
    ZEND_API int zend_declare_property_stringl(zend_class_entry *ce, const char
*name, size_t name_length, const char *value, size_t value_len, int access_type);
```

这些 API 最后一个参数都是 access_type，这个参数是用来指定属性权限以及是否为静态属性的：

```
    //file: zend_compile.h
    #define ZEND_ACC_PUBLIC     0x100
    #define ZEND_ACC_PROTECTED  0x200
    #define ZEND_ACC_PRIVATE    0x400
    //静态
    #define ZEND_ACC_STATIC         0x01
```

通常情况下属性的声明与类的定义放在一起，也就是在 module startup 中完成，例如下面的例子，声明了一个静态整型属性、一个非静态字符串属性：

```
    zend_class_entry *myclass_ce;

    PHP_MINIT_FUNCTION(mytest)
    {
        //分配一个 zend_class_entry，这个结构只在注册时使用，所以分配在栈上即可
        zend_class_entry ce;
        //对 zend_class_entry 进行初始化
        INIT_CLASS_ENTRY(ce, "MyClass", NULL);
        //注册
        myclass_ce = zend_register_internal_class(&ce);

        //声明静态公共属性 prop_1，等价于：static public $prop_1 = 9999
```

```
    zend_declare_property_long(myclass_ce, "prop_1", sizeof("prop_1") - 1,
9999, ZEND_ACC_PUBLIC | ZEND_ACC_STATIC);
    //声明非静态公共属性prop_2，等价于：public $prop_2 = "hello world"
    zend_declare_property_string(myclass_ce, "prop_2", sizeof("prop_2") -
1, "hello word", ZEND_ACC_PUBLIC);
}
```

重新编译后通过以下脚本测试：

```
$obj = new MyClass();
var_dump(MyClass::$prop_1);
var_dump($obj→prop_2);
```

注册属性时有个地方需要注意：属性名称的长度，这里的长度是不包括字符串后的那个""的，因此是sizeof()计算长度减1，否则将无法找到属性。这个地方可以通过ZEND_STRL()宏计算：

```
#define ZEND_STRL(str)    (str), (sizeof(str)-1)
```

另外，内部类声明的属性不能是数组、对象、资源，如果是这几种类型在声明时将会报错，但这并不是指内部类不能使用这种类型，只是默认值不能是这些类型。所以实际使用时通常会把它们定义为NULL，运行时再通过对象将其修改为其他类型。

2. 修改成员属性

```
//1) 修改非静态属性
//通用接口，修改为任意类型
ZEND_API void zend_update_property(zend_class_entry *scope, zval *object,
const char *name, size_t name_length, zval *value);
//将属性修改为NULL类型
ZEND_API void zend_update_property_null(zend_class_entry *scope, zval
*object, const char *name, size_t name_length);
//将属性修改为布尔型
ZEND_API void zend_update_property_bool(zend_class_entry *scope, zval
*object, const char *name, size_t name_length, zend_long value);
//将属性修改为整型
ZEND_API void zend_update_property_long(zend_class_entry *scope, zval
*object, const char *name, size_t name_length, zend_long value);
//将属性修改为浮点型
```

```
    ZEND_API void zend_update_property_double(zend_class_entry *scope, zval
*object, const char *name, size_t name_length, double value);
    //将属性修改为字符串，value 类型为 zend_string
    ZEND_API void zend_update_property_str(zend_class_entry *scope, zval
*object, const char *name, size_t name_length, zend_string *value);
    //同上一个，value 类型为 char
    ZEND_API void zend_update_property_string(zend_class_entry *scope, zval
*object, const char *name, size_t name_length, const char *value);
    ZEND_API void zend_update_property_stringl(zend_class_entry *scope, zval
*object, const char *name, size_t name_length, const char *value, size_t
value_length);
    //2) 修改静态成员属性
    ZEND_API int zend_update_static_property(zend_class_entry *scope, const
char *name, size_t name_length, zval *value);
    ZEND_API int zend_update_static_property_null(zend_class_entry *scope,
const char *name, size_t name_length);
    ZEND_API int zend_update_static_property_bool(zend_class_entry *scope,
const char *name, size_t name_length, zend_long value);
    ZEND_API int zend_update_static_property_long(zend_class_entry *scope,
const char *name, size_t name_length, zend_long value);
    ZEND_API int zend_update_static_property_double(zend_class_entry *scope,
const char *name, size_t name_length, double value);
    ZEND_API int zend_update_static_property_string(zend_class_entry *scope,
const char *name, size_t name_length, const char *value);
    ZEND_API int zend_update_static_property_stringl(zend_class_entry *scope,
const char *name, size_t name_length, const char *value, size_t value_length);
```

3. 获取成员属性

成员属性的获取可通过以下接口获得：

```
    //获取非静态成员属性
    ZEND_API zval *zend_read_property(zend_class_entry *scope, zval *object,
const char *name, size_t name_length, zend_bool silent, zval *rv);
    //获取静态成员属性
    ZEND_API zval *zend_read_static_property(zend_class_entry *scope, const
char *name, size_t name_length, zend_bool silent);
```

10.9.3 成员方法

1. 定义

成员方法的定义与内部函数类似，内核提供了两个宏用于成员方法的定义：

```
#define PHP_METHOD                                    ZEND_METHOD
#define ZEND_METHOD(classname, name)                  ZEND_NAMED_FUNCTION(ZEND_MN
(classname##_##name))
```

定义时指定类名、方法名即可，例如：

```
ZEND_METHOD(MyClass, get)
{
    ...
}
```

2. 注册

在 INIT_CLASS_ENTRY 宏初始类时，第 3 个参数传入的就是成员方法数组，我们直接将所有的成员方法保存到一个数组中，然后设置到 zend_class_entry 中即可，这个数组中成员可通过 PHP_ME(classname, name, arg_info, flags)或 ZEND_ME()宏定义，其中 flags 用于标识该成员方法的权限、final、abstract、static 等信息。

```
zend_class_entry *myclass_ce;

ZEND_METHOD(MyClass, get)
{
    //...
}

zend_function_entry myclass_methods[] = {
    PHP_ME(MyClass, get,        NULL, ZEND_ACC_PUBLIC)
    PHP_FE_END
};

PHP_MINIT_FUNCTION(mytest)
{
```

```
    //分配一个 zend_class_entry，这个结构只在注册时使用，所以分配在栈上即可
    zend_class_entry ce;
    //对 zend_class_entry 进行初始化
    INIT_CLASS_ENTRY(ce, "MyClass", myclass_methods);
    //注册
    myclass_ce = zend_register_internal_class(&ce);
}
```

10.9.4 常量

1. 声明常量

```
//zend_API.h
//通常接口
ZEND_API int zend_declare_class_constant(zend_class_entry *ce, const char *name, size_t name_length, zval *value);
//声明 NULL 型常量
ZEND_API int zend_declare_class_constant_null(zend_class_entry *ce, const char *name, size_t name_length);
//声明整型常量
ZEND_API int zend_declare_class_constant_long(zend_class_entry *ce, const char *name, size_t name_length, zend_long value);
//声明布尔型常量
ZEND_API int zend_declare_class_constant_bool(zend_class_entry *ce, const char *name, size_t name_length, zend_bool value);
//声明浮点型常量
ZEND_API int zend_declare_class_constant_double(zend_class_entry *ce, const char *name, size_t name_length, double value);
//声明字符串常量
ZEND_API int zend_declare_class_constant_stringl(zend_class_entry *ce, const char *name, size_t name_length, const char *value, size_t value_length);
ZEND_API int zend_declare_class_constant_string(zend_class_entry *ce, const char *name, size_t name_length, const char *value);
```

2. 获取常量

扩展中获取类的常量一般较少使用，可以通过下面这个接口获取，其中 name 可以包含“::”，比如 self::常量名、parent::常量名、static::常量名。

```
    //zend_constants.h
    ZEND_API zval *zend_get_constant_ex(zend_string *name, zend_class_entry
*scope, zend_ulong flags);
```

10.9.5 类的实例化

在扩展中可以通过 object_init_ex()创建一个对象：

```
    //zend_API.h
    //arg 为生成的对象，类型为 zval*，ce 为要实例化的类
    #define object_init_ex(arg, ce) _object_init_ex((arg), (ce) ZEND_FILE_
LINE_CC)
```

_object_init_ex()最后将调用类的 create_object 方法创建对象，这个地方我们自定义类的 create_object 的操作，从而接管创建对象的操作，具体的实现方法可以参考默认的 create_object：

```
    ZEND_API zend_object *zend_objects_new(zend_class_entry *ce)
    {
        zend_object *object = emalloc(sizeof(zend_object) + zend_object_
properties_size(ce));

        zend_object_std_init(object, ce);
        //设置对象操作的 handler
        object->handlers = &std_object_handlers;
        return object;
    }
```

这里需要特别注意，如果自定义了 create_object 方法，则需要考虑是否定义对象销毁的 handler：zend_object→handlers→free_obj。默认情况下，当销毁一个对象时会调用默认的方法处理，如果你的 create_object 方法里除 zend_object 外还分配了其他内存，则需要同时自定义 free_obj，否则就会导致 zend_object 之外的内存得不到释放。自定义对象的 hanlders 方法比较简单，直接在创建对象时直接将 handlers 指定为自定义的即可。

10.10 资源

除了 PHP 中定义的那些布尔型、整型、浮点型、数组等类型之外，实际应用中还会经常用到 socket 连接、文件句柄等数据，那么这些类型在 PHP 中是如何实现的呢？答案就是资源。资

源是 PHP 中比较特殊的一种类型，资源类型只能通过扩展、内核定义，无法在用户空间中自定义资源类型，只能使用资源。资源也是 PHP 中最为灵活的一种类型，可以把任何数据都定义为资源类型。

注册新的资源类型时，PHP 内核并不关心资源的具体数据是什么，它只分配给资源一个唯一编号作为区分资源类型的标识，剩下的事情都需要扩展自己控制。资源类型的结构如下：

```
struct _zend_resource {
    zend_refcounted_h gc;
    int               handle; //资源的序号
    int               type; //资源的类型，由内核分配
    void             *ptr; //资源的数据
};
```

资源的数据结构非常简单，handle 为一次请求期间分配的所有资源序号，按资源的创建顺序分配，type 为资源类型，ptr 为资源具体的数据，它可以是任意类型，当创建资源类型的变量时，我们可以把具体的数据保存到 ptr 中，使用时再转为具体的类型。

1. 注册资源类型

首先是注册一种资源类型，注册时只是向内核申请了一个资源类型编号，同时告诉内核该资源的销毁函数句柄，并不需要告诉内核资源具体是什么。注册资源通过 zend_register_list_destructors_ex()方法完成：

```
//zend_list.h
ZEND_API int zend_register_list_destructors_ex(rsrc_dtor_func_t ld,
rsrc_dtor_func_t pld, const char *type_name, int module_number);
```

ld 为资源的销毁函数，pld 为销毁资源类型的回调函数，type_name 为资源类型的名称，module_number 为模块编号。注册成功后该函数返回内核分配的资源类型值，这个值需要保存下来供后续使用，之后对该资源类型的获取、创建资源等操作都会用到这个类型值，它也是不同资源之间唯一的区分标识。下面我们以文件的读写为例，定义一种文件指针的资源类型，也就是实现 C 语言中 fopen、fwrite 的操作。

首先我们定义一个结构用于保存文件指针，然后把这个结构作为一种资源：

```
typedef struct {
    zend_string *file_name;
    FILE    *fp;
```

```
} my_file_t;
```

然后注册资源，资源的注册通常放到 module starup 阶段：

```
static int le_myfile;
PHP_MINIT_FUNCTION(mytest)
{
    le_myfile = zend_register_list_destructors_ex(NULL, NULL, "myfile",
module_number);
}
```

接下来就可以创建并使用这种类型的资源了。zend_register_list_destructors_ex()为所有的资源类型创建了一个 zend_rsrc_list_dtors_entry 结构：

```
typedef struct _zend_rsrc_list_dtors_entry {
    rsrc_dtor_func_t list_dtor_ex; //销毁函数
    rsrc_dtor_func_t plist_dtor_ex; //持久化资源的销毁函数

    const char *type_name; //资源类型描述

    int module_number; //所属模块
    int resource_id;   //资源类型值
} zend_rsrc_list_dtors_entry;
```

内核把所有的资源类型保存到了一个哈希表中：list_destructors，这是一个全局变量。注册后可以通过 gdb 查看到我们注册的资源了：

```
$ gdb php
(gdb) break zm_startup_mytest
(gdb) r
...
(gdb) p list_destructors
$1 = {gc = {refcount = 1, u = {v = {type = 7 '\a', flags = 0 '\000', gc_info
= 0}, type_info = 7}}, u = {v = {flags = 31 '\037', nApplyCount = 0 '\000',
nIteratorsCount = 0 '\000', reserve = 0 '\000'}, flags = 31}, nTableMask =
4294967294, arData = 0x1096c58, nNumUsed = 17, nNumOfElements = 16, nTableSize
= 64, nInternalPointer = 1, nNextFreeElement = 17, pDestructor = 0x83d9dd
<list_destructors_dtor>}
```

```
(gdb) p *((zend_rsrc_list_dtors_entry*)list_destructors→arData[16])
$5 = {list_dtor_ex = 0x0, plist_dtor_ex = 0x0, type_name = 0x7ffff1b6c0bb
"myfile", module_number = 26, resource_id = 16}
```

2. 创建资源

创建资源有两种含义：一种是指创建实际的资源，并不是 zend_resource 结构，这种完全由资源的注册方自己完成；另外一种就是创建 zend_resource 结构，比如我们创建的资源需要传递到 PHP 用户空间，这个时候就需要创建一个 zend_resource 结构，然后把实际的资源保存到 zend_resource→ptr。如果要创建一个资源类型的变量，即 zend_resource，可以通过 zend_register_resource()方法申请，该方法只有两个参数：一个为资源的数据指针，另一个为注册资源时返回的资源类型，申请成功后返回 zend_resource 地址。分配的所有资源变量保存在 EG(regular_list)哈希表中，请求结束后会清理这个哈希表。

```
//rsrc_pointer 为资源的数据，rsrc_type 为资源类型
ZEND_API zend_resource *zend_register_resource(void *rsrc_pointer, int
rsrc_type);
```

我们继续实现文件读写的功能，接下来实现一个打开文件的函数，也就是 C 语言中 fopen，该函数返回一个资源类型，返回后可以通过这个资源读写文件。首先是解析参数，传入的是文件路径；接着打开这个文件；然后分配一个 my_file_t 结构，并将文件指针保存到这个结构；最后注册一个资源变量，并返回。具体逻辑如下：

```
PHP_FUNCTION(my_fopen)
{
    zend_string *path;
    FILE    *fp;
    my_file_s   *myresource;
    zend_resource   *var_resource;
    //1) 解析参数
    if(zend_parse_parameters(ZEND_NUM_ARGS(), "S", &path) == FAILURE){
        RETURN_FALSE;
    }
    //2) 打开文件
    fp = fopen(ZSTR_VAL(path), "w+");
    //3) 创建资源结构
    myresource = emalloc(sizeof(my_file_s));
    myresource→file_name = path;
```

```
    myresource->fp = fp;
    //4) 创建资源变量
    var_resource = zend_register_resource(myresource, le_myfile);
    //返回资源
    RETURN_RES(var_resource);
}
```

重新编译后在 PHP 中调用该函数：

```
$myfp = my_fopen("/tmp/log");
var_dump($myfp);
```

执行后将输出：

```
resource(4) of type (myfile)
```

接着我们继续实现 fwrite 的操作，参数之一就是 my_fopen()返回的资源：

```
PHP_FUNCTION(my_fwrite)
{
    zval    *res;
    zend_string *content;
    my_file_s   *myresource;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "rS", &res, &content) ==
FAILURE){
        RETURN_FALSE;
    }
    //获取 zend_resource->ptr
    myresource = Z_RES_VAL_P(res);
    //写入文件
    fwrite (ZSTR_VAL(content), sizeof(char), content->len, myresource->fp);
}
```

然后调用 my_fwrite()，执行后检查/tmp/log 中是否成功写入了。

```
$myfp = my_fopen("/tmp/log");
```

```
my_fwrite($myfp, "hello world\n");
```

这就是一个资源的基本用法，从上面的例子我们看到，通过资源类型可以灵活地存储任意数据，在 PHP 扩展中，很多 RPC 客户端也是通过资源类型来存储 socket 连接的。关于更多的使用例子大家可以到 ext 目录下搜索，有非常多的范例。

3. 销毁资源

PHP 的变量是自动回收的，资源类型不同于其他的数据，因为资源不仅仅是释放内存，它还需要进行其他的一些清理，比如上面的示例，销毁资源时是需要关闭文件的，再比如保存 socket 连接的资源，在释放时也需要关闭连接。在注册资源类型时可以提供一个析构函数，当资源销毁时，资源管理器会回调该函数进行清理，这个函数类型如下：

```
typedef void (*rsrc_dtor_func_t)(zend_resource *res);
```

我们重新修改下上面的例子，提供一个 myfile_dtor()方法，用于释放资源：

```
static int le_myfile;

void myfile_dtor(zend_resource *res)
{
    my_file_s *myresource;

    myresource = (my_file_s *)(res->ptr);
    //关闭文件
    fclose(myresource->fp);
    //释放资源
    efree(myresource);
}

PHP_MINIT_FUNCTION(mytest)
{
    //指定资源的销毁函数
    le_myfile = zend_register_list_destructors_ex(myfile_dtor, NULL,
"myfile", module_number);
}
```

4. 持久化资源

普通资源的生命周期在 request 结束时也随之结束，持久化资源则可以使资源在 request 结

束时仍然保留，下次请求时还可以继续使用。持久化资源通过 EG(persistent_list)保存，直接将资源插入这个哈希表即可。持久化资源有着非常重要的意义，利用持久化资源可以实现长连接，从而提高系统的性能。持久化资源在 module shutdown 阶段才被释放。

10.11　小结

本章介绍了 PHP 扩展开发中常用的一些知识，主要是一些接口、宏的使用，内容更多地倾向于应用性，原理性的东西介绍的比较少。实际 PHP 扩展的开发并不复杂，尽管扩展开发并不需要对 ZendVM 有很深入的了解，但是如果对 ZendVM 的实现比较熟悉，那么对扩展开发的帮助将非常大，你可以更容易想到从哪个地方入手实现扩展提供的功能，debug 时也更加得心应手。总之一句话，如果你想真正地理解 PHP 的内部实现，仅仅通过扩展是不可能完成的。