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。

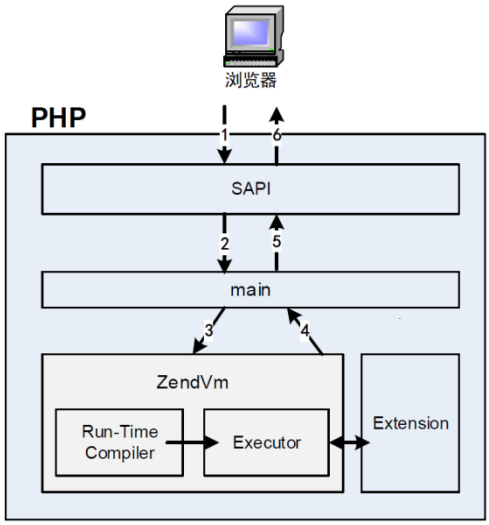

图 1-1 PHP 的基本构成

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。

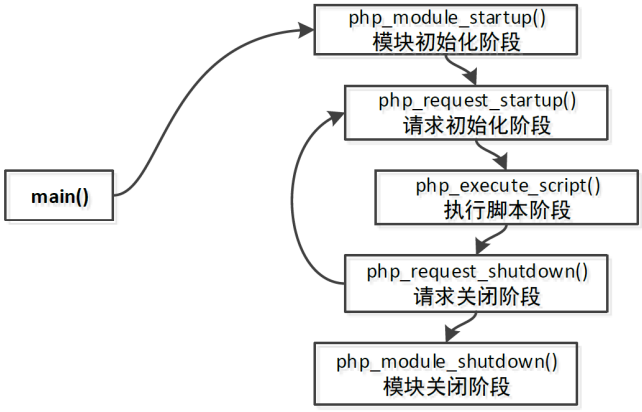

图1-2 PHP的生命周期

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。

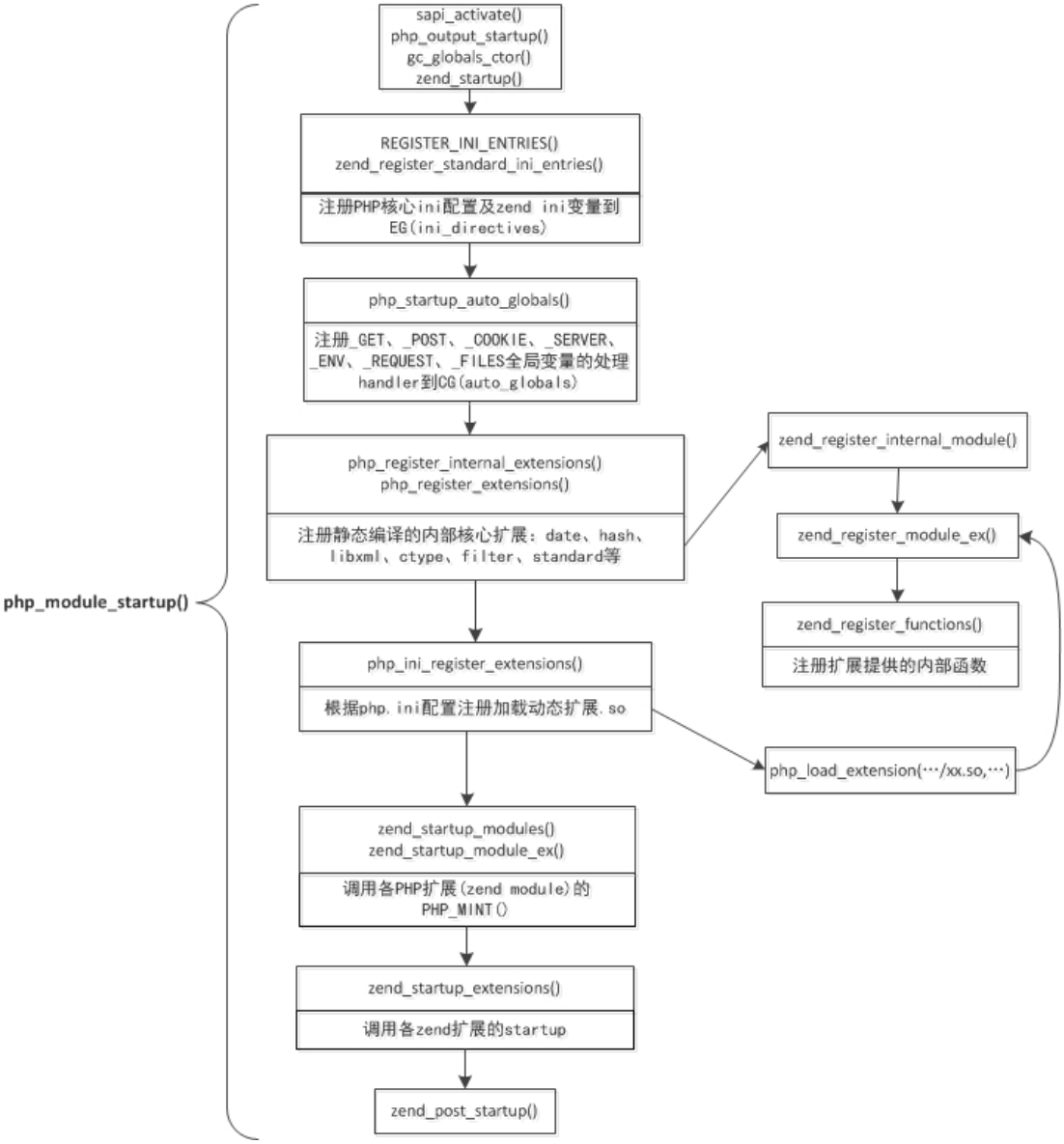

图 1-3　php_module_startup()

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。

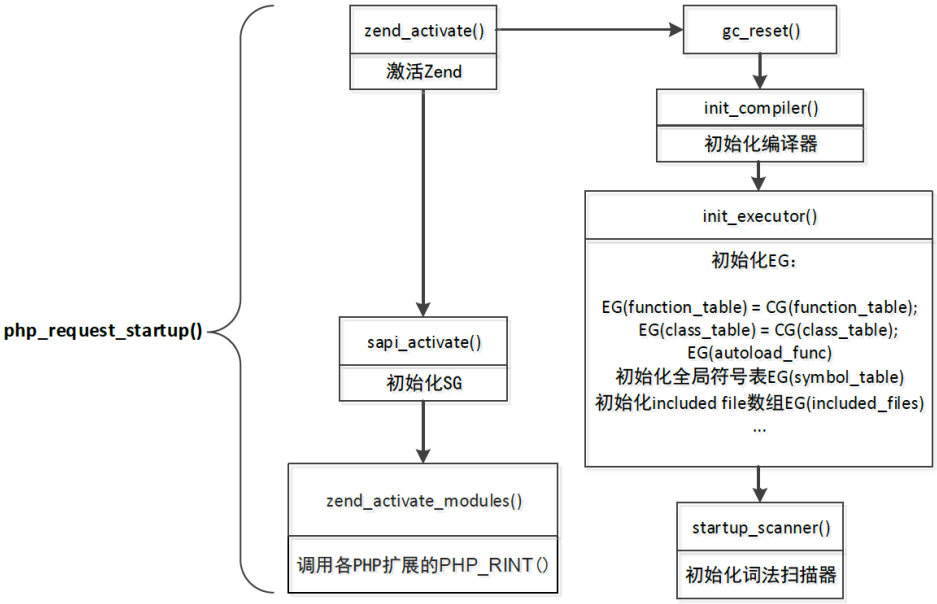

图 1-4　php_request_startup()

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。

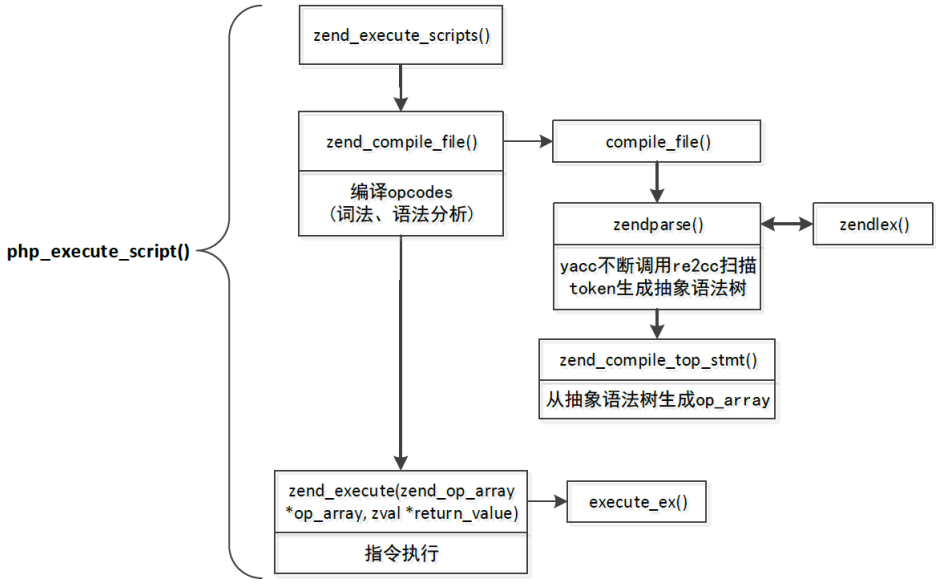

图 1-5　php_execute_script()

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。

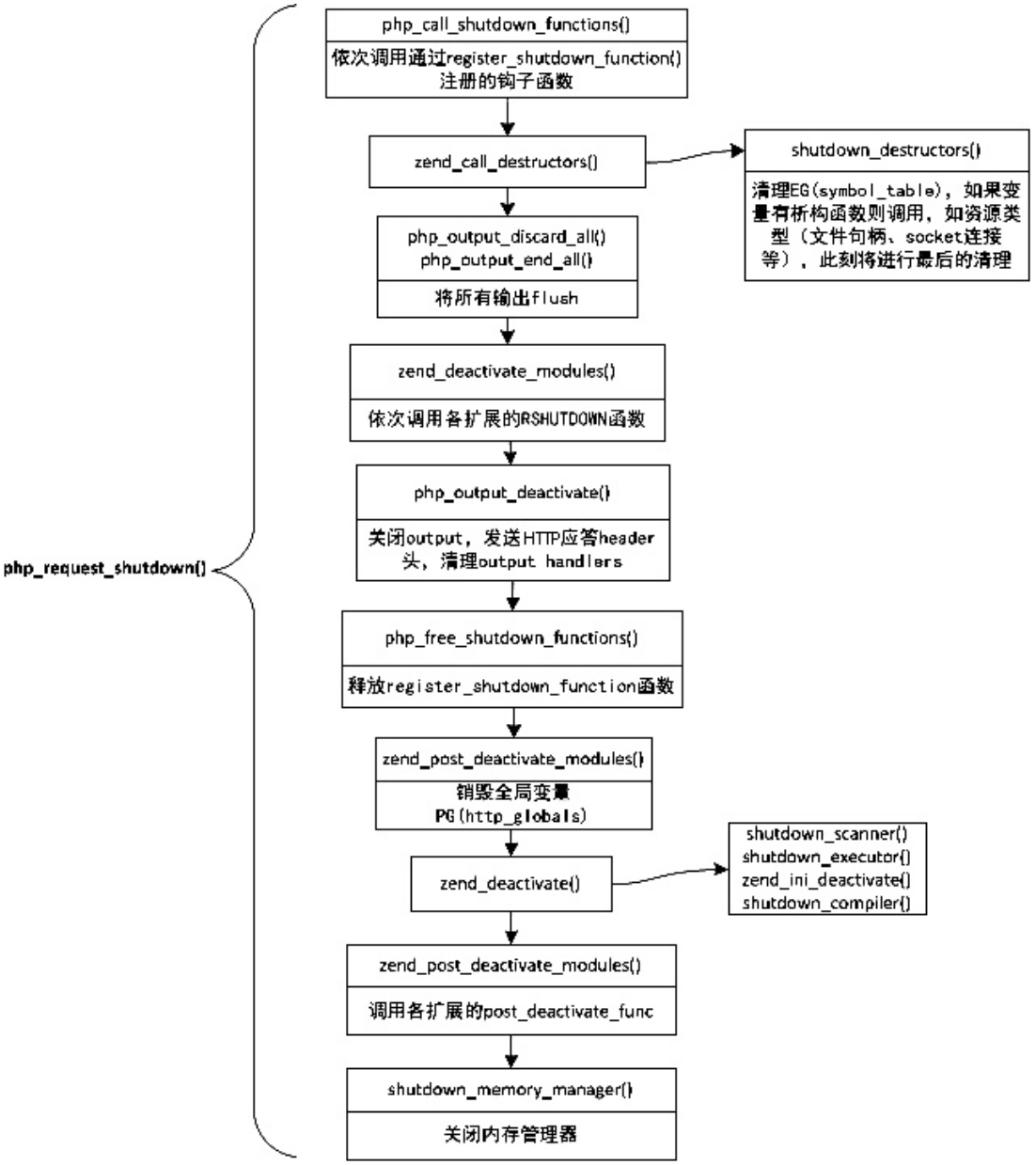

图 1-6　php_request_shutdown()

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。

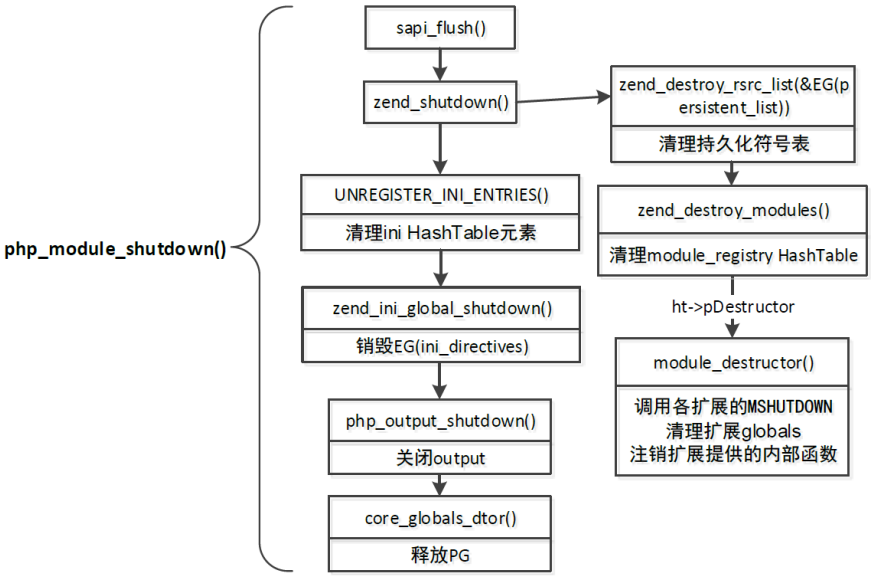

图 1-7 php_module_shutdown()

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。