第二章 用户代码的执行

**不识庐山真面目，只缘身在此山中。**

PHP做为一种优秀的脚本语言，在当前的互联网应用中可谓风光无限。 从简单的“Hello World!”到各种框架开发，架构设计，性能优化，到编写PHP扩展， PHP编程中涉及的知识结构和跨度蔚为可观。 从这个角度上来看，学会PHP编程的语法可能并不困难， 但如果想真正用好PHP，在不同的场景下发挥PHP最大的性能和效用， 对PHP的理解到达熟悉和精通的程度，就不得不去了解PHP语言的实现， 进一步理解PHP语法的本质，这确实是一件需要更多的精力和时间的事情。

PHP语言经过许多人多年的淬炼，性能不断优化，支持的语法现象与各种特性也越来越多。 导致PHP内核的代码中，涉及知识面比较广泛，具体实现也非常复杂， 从脚本的编译解析到执行以及和Web服务器等的配合，内存管理，语法实现等等。 为了不过早陷入细节的沼泽，我们先从整体上来接触PHP的实现， 先对PHP的整体结构，生命周期，PHP与其它容器（如Apache）的交互， PHP的整个执行过程等进行一个大概的了解，从而有一个整体的概念。

关于PHP是如何一步步从一个朴素的想法发展到今天的模样，可以了解一下PHP的历史： http://en.wikipedia.org/wiki/PHP

从宏观上来看，PHP内核的实现与世界上绝大多数的程序一样，接收输入数据， 做相应处理然后输出（返回）结果（me：input->process->output）。 我们编写的代码就是PHP接收的输入数据，PHP内核对我们编写的代码进行解释和运算， 最后返回相应的运算结果。 然而，PHP与我们自己平时写的一般的C程序有所不同的是， 我们的程序一般用来解决某个具体问题， 而PHP本身实现了把用户的逻辑“翻译”为机器语言来执行的功能， 这也是各种编译语言与承载具体业务逻辑的程序代码的一个明显区别。 于是PHP就多出一个把用户代码“翻译”成具体操作的步骤：**词法分析、语法分析**

当用户代码输入给PHP内核去执行的时候， PHP内核会对PHP代码进行词法分析和语法分析， 词法分析是把PHP代码分割成一个个的“单元”（TOKEN）， 语法分析则将这些“单元”转化为Zend Engine可执行的操作。 然后PHP内部的Zend Engine对这些操作进行顺次的执行。 Zend Engine是PHP内核的核心部分，负责最终操作的执行和结果的返回， 可以理解成为PHP内核中的“发动机”。

于是PHP代码的执行过程可以简单描述为下图：

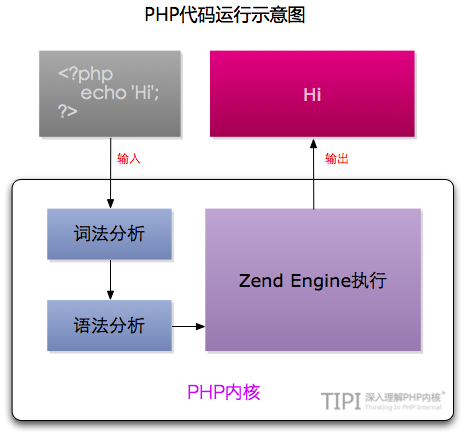


图2.1 单进程SAPI生命周期

接下来，本章会对图中的每一部分展开详细的讨论，主要包括以下内容：

1. PHP内部的生命周期
2. SAPI接口
3. 词法分析与语法分析
4. 什么是Opcodes

# 第一节 生命周期和Zend引擎

## 一切的开始: SAPI接口

SAPI(Server Application Programming Interface)指的是PHP具体应用的编程接口， 就像PC一样，无论安装哪些操作系统，只要满足了PC的接口规范都可以在PC上正常运行， PHP脚本要执行有很多种方式，通过Web服务器，或者直接在命令行下，也可以嵌入在其他程序中。

通常，我们使用Apache或者Nginx这类Web服务器来测试PHP脚本，或者在命令行下通过PHP解释器程序来执行。 脚本执行完后，Web服务器应答，浏览器显示应答信息，或者在命令行标准输出上显示内容。

我们很少关心PHP解释器在哪里。虽然通过Web服务器和命令行程序执行脚本看起来很不一样， 实际上它们的工作流程是一样的。命令行参数传递给PHP解释器要执行的脚本， 相当于通过url请求一个PHP页面。脚本执行完成后返回响应结果，只不过命令行的响应结果是显示在终端上。

脚本执行的开始都是以SAPI接口实现开始的。只是不同的SAPI接口实现会完成他们特定的工作(me：做一些定制化的处理)， 例如Apache的mod\_php SAPI实现需要初始化从Apache获取的一些信息，在输出内容是将内容返回给Apache， 其他的SAPI实现也类似。

me：从SAPI开始，从SAPI结束

下面几个小节将对一些常见的SAPI实现进行更为深入的介绍。

## 开始和结束

PHP开始执行以后会经过两个主要的阶段：处理请求之前的开始阶段和请求之后的结束阶段。 开始阶段有两个过程：第一个过程是模块初始化阶段（MINIT）， 在整个SAPI生命周期内(例如Apache启动以后的整个生命周期内或者命令行程序整个执行过程中)， 该过程只进行一次。第二个过程是模块激活阶段（me:这个过程应该是PHP源代码使用了某个模块的时候才会激活该模块）（RINIT），该过程发生在请求阶段， 例如通过url请求某个页面，则在每次请求之前都会进行模块激活（RINIT请求开始）。 例如PHP注册了一些扩展模块，则在MINIT阶段会回调所有模块的MINIT函数。 模块在这个阶段可以进行一些初始化工作，例如注册常量，定义模块使用的类等等。 模块在实现时可以通过如下宏来实现这些回调函数：

PHP\_MINIT\_FUNCTION(myphpextension)

{

*// 注册常量或者类等初始化操作*

return SUCCESS;

}

请求到达之后PHP初始化执行脚本的基本环境，例如创建一个执行环境，包括保存PHP运行过程中变量名称和值内容的符号表， 以及当前所有的函数以及类等信息的符号表。然后PHP会调用所有模块的RINIT函数， 在这个阶段各个模块也可以执行一些相关的操作，模块的RINIT函数和MINIT回调函数类似：

PHP\_RINIT\_FUNCTION(myphpextension)

{

*// 例如记录请求开始时间*

*// 随后在请求结束的时候记录结束时间。这样我们就能够记录下处理请求所花费的时间了*

return SUCCESS;

}

请求处理完后就进入了结束阶段，一般脚本执行到末尾或者通过调用exit()或die()函数， PHP都将进入结束阶段。和开始阶段对应，结束阶段也分为两个环节，一个在请求结束后停用模块（me：退出模块）(RSHUTDOWN，对应RINIT)， 一个在SAPI生命周期结束（Web服务器退出或者命令行脚本执行完毕退出）时关闭模块(MSHUTDOWN，对应MINIT)。

PHP\_RSHUTDOWN\_FUNCTION(myphpextension)

{

*// 例如记录请求结束时间，并把相应的信息写入到日至文件中。*

return SUCCESS;

}

me：开始和结束：

模块初始化->模块激活->ZEND 引擎处理PHP源代码…-> 模块停用 或者模块退出->关闭模块

PHP\_MINIT\_FUNCTION

PHP\_RINIT\_FUNCTION

PHP\_RSHUTDOWN\_FUNCTION

PHP\_MSHUTDOWN\_FUNCITON

想要了解扩展开发的相关内容，请参考第十三章 扩展开发

### 单进程SAPI生命周期

CLI/CGI(me:通用网管接口)模式的PHP属于单进程的SAPI模式。这类的请求在处理一次请求后就关闭。也就是只会经过如下几个环节： 开始 - 请求开始 - 请求关闭 - 结束 SAPI接口实现就完成了其生命周期。如图2.1所示：

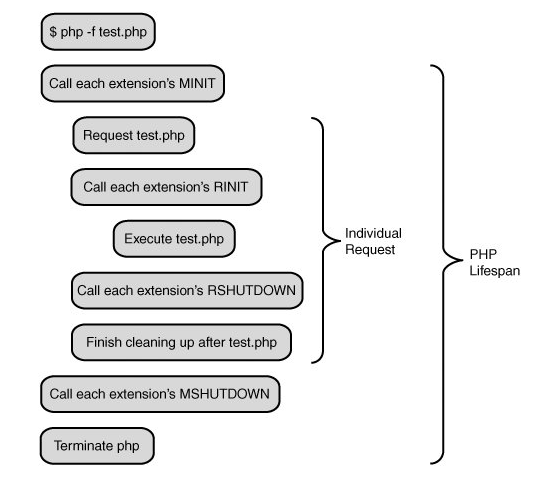


图2.1 单进程SAPI生命周期

如上的图是非常简单，也很好理解。只是在各个阶段之间PHP还做了许许多多的工作。这里做一些补充：

**启动**

在调用每个模块的模块初始化前，会有一个初始化的过程，它包括（me:也是按照如下的顺序执行）：

* **初始化若干全局变量**

这里的初始化全局变量大多数情况下是将其设置为NULL，有一些除外，比如设置zuf（zend\_utility\_functions）， 以zuf.printf\_function = php\_printf为例，这里的php\_printf在zend\_startup函数中会被赋值给zend\_printf作为全局函数指针使用， 而zend\_printf函数通常会作为常规字符串输出使用，比如显示程序调用栈的debug\_print\_backtrace就是使用它打印相关信息。

* **初始化若干常量**

这里的常量是PHP自己的一些常量，这些常量要么是硬编码在程序中,比如PHP\_VERSION，要么是写在配置头文件中， 比如PEAR\_EXTENSION\_DIR，这些是写在config.w32.h文件中。

* **初始化Zend引擎和核心组件**

前面提到的zend\_startup函数的作用就是初始化Zend引擎，这里的初始化操作包括内存管理初始化、 全局使用的函数指针初始化（如前面所说的zend\_printf等），对PHP源文件进行词法分析、语法分析、 中间代码执行的函数指针的赋值，初始化若干HashTable（比如函数表，常量表等等），为ini文件解析做准备， 为PHP源文件解析做准备，注册内置函数（如strlen、define等），注册标准常量（如E\_ALL、TRUE、NULL等）、注册GLOBALS全局变量等。

* **解析php.ini**

php\_init\_config函数的作用是读取php.ini文件，设置配置参数，加载zend扩展并注册PHP扩展函数。此函数分为如下几步： 初始化参数配置表，调用当前模式下的ini初始化配置，比如CLI模式下，会做如下初始化：

INI\_DEFAULT("report\_zend\_debug", "0");

INI\_DEFAULT("display\_errors", "1");

不过在其它模式下却没有这样的初始化操作。接下来的各种操作都是查找ini文件：

1. 判断是否有php\_ini\_path\_override，在CLI模式下可以通过-c参数指定此路径（在php的命令参数中-c表示在指定的路径中查找ini文件）。
2. 如果没有php\_ini\_path\_override，判断php\_ini\_ignore是否为非空（忽略php.ini配置，这里也就CLI模式下有用，使用-n参数）。
3. 如果不忽略ini配置，则开始处理php\_ini\_search\_path（查找ini文件的路径），这些路径包括CWD(当前路径，不过这种不适用CLI模式)、 执行脚本所在目录、环境变量PATH和PHPRC和配置文件中的PHP\_CONFIG\_FILE\_PATH的值。
4. 在准备完查找路径后，PHP会判断现在的ini路径（php\_ini\_file\_name）是否为文件和是否可打开。 如果这里ini路径是文件并且可打开，则会使用此文件， 也就是CLI模式下通过-c参数指定的ini文件的优先级是最高的， 其次是PHPRC指定的文件，第三是在搜索路径中查找php-%sapi-module-name%.ini文件（如CLI模式下应该是查找php-cli.ini文件）， 最后才是搜索路径中查找php.ini文件。

* **全局操作函数的初始化**

php\_startup\_auto\_globals函数会初始化在用户空间所使用频率很高的一些全局变量，如：$\_GET、$\_POST、$\_FILES等。 这里只是初始化，所调用的zend\_register\_auto\_global函数也只是将这些变量名添加到CG(auto\_globals)这个变量表。

php\_startup\_sapi\_content\_types函数用来初始化SAPI对于不同类型内容的处理函数， 这里的处理函数包括POST数据默认处理函数、默认数据处理函数等。

* **初始化静态构建的模块和共享模块(MINIT)**

php\_register\_internal\_extensions\_func函数用来注册静态构建的模块，也就是默认加载的模块， 我们可以将其认为内置模块。在PHP5.3.0版本中内置的模块包括PHP标准扩展模块（/ext/standard/目录， 这里是我们用的最频繁的函数，比如字符串函数，数学函数，数组操作函数等等），日历扩展模块、FTP扩展模块、 session扩展模块等。这些内置模块并不是一成不变的，在不同的PHP模板中，由于不同时间的需求或其它影响因素会导致这些默认加载的模块会变化， 比如从代码中我们就可以看到mysql、xml等扩展模块曾经或将来会作为内置模块出现。

模块初始化会执行两个操作： 1. 将这些模块注册到已注册模块列表（module\_registry），如果注册的模块已经注册过了，PHP会报Module XXX already loaded的错误。 1. 将每个模块中包含的函数注册到函数表（ CG(function\_table) ），如果函数无法添加，则会报 Unable to register functions, unable to load。

在注册了静态构建的模块后，PHP会注册附加的模块，不同的模式下可以加载不同的模块集，比如在CLI模式下是没有这些附加的模块的。

在内置模块和附加模块后，接下来是注册通过共享对象（比如DLL）和php.ini文件灵活配置的扩展。

在所有的模块都注册后，PHP会马上执行模块初始化操作（zend\_startup\_modules）。 它的整个过程就是依次遍历每个模块，调用每个模块的模块初始化函数， 也就是在本小节前面所说的用宏PHP\_MINIT\_FUNCTION包含的内容。

* **禁用函数和类**

php\_disable\_functions函数用来禁用PHP的一些函数。这些被禁用的函数来自PHP的配置文件的disable\_functions变量。 其禁用的过程是调用zend\_disable\_function函数将指定的函数名从CG(function\_table)函数表中删除。

php\_disable\_classes函数用来禁用PHP的一些类。这些被禁用的类来自PHP的配置文件的disable\_classes变量。 其禁用的过程是调用zend\_disable\_class函数将指定的类名从CG(class\_table)类表中删除。

**ACTIVATION**

在处理了文件相关的内容，PHP会调用php\_request\_startup做请求初始化操作。 请求初始化操作，除了图中显示的调用每个模块的请求初始化函数外，还做了较多的其它工作，其主要内容如下：

* **激活Zend引擎**

gc\_reset函数用来重置垃圾收集机制，当然这是在PHP5.3之后才有的。

init\_compiler函数用来初始化编译器，比如将编译过程中放在opcode里的数组清空，准备编译时需要用的数据结构等等。

init\_executor函数用来初始化中间代码执行过程。 在编译过程中，函数列表、类列表等都存放在编译时的全局变量中， 在准备执行过程时，会将这些列表赋值给执行的全局变量中，如：EG(function\_table) = CG(function\_table); 中间代码执行是在PHP的执行虚拟栈中，初始化时这些栈等都会一起被初始化。 除了栈，还有存放变量的符号表(EG(symbol\_table))会被初始化为50个元素的hashtable，存放对象的EG(objects\_store)被初始化了1024个元素。 PHP的执行环境除了上面的一些变量外，还有错误处理，异常处理等等，这些都是在这里被初始化的。 通过php.ini配置的zend\_extensions也是在这里被遍历调用activate函数。

* **激活SAPI**

sapi\_activate函数用来初始化SG(sapi\_headers)和SG(request\_info)，并且针对HTTP请求的方法设置一些内容， 比如当请求方法为HEAD时，设置SG(request\_info).headers\_only=1； 此函数最重要的一个操作是处理请求的数据，其最终都会调用sapi\_module.default\_post\_reader。 而sapi\_module.default\_post\_reader在前面的模块初始化是通过php\_startup\_sapi\_content\_types函数注册了 默认处理函数为main/php\_content\_types.c文件中php\_default\_post\_reader函数。 此函数会将POST的原始数据写入$HTTP\_RAW\_POST\_DATA变量。

在处理了post数据后，PHP会通过sapi\_module.read\_cookies读取cookie的值， 在CLI模式下，此函数的实现为sapi\_cli\_read\_cookies，而在函数体中却只有一个return NULL;

如果当前模式下有设置activate函数，则运行此函数，激活SAPI，在CLI模式下此函数指针被设置为NULL。

* **环境初始化**

这里的环境初始化是指在用户空间中需要用到的一些环境变量初始化，这里的环境包括服务器环境、请求数据环境等。 实际到我们用到的变量，就是$\_POST、$\_GET、$\_COOKIE、$\_SERVER、$\_ENV、$\_FILES。 和sapi\_module.default\_post\_reader一样，sapi\_module.treat\_data的值也是在模块初始化时， 通过php\_startup\_sapi\_content\_types函数注册了默认数据处理函数为main/php\_variables.c文件中php\_default\_treat\_data函数。

以$\_COOKIE为例，php\_default\_treat\_data函数会对依据分隔符，将所有的cookie拆分并赋值给对应的变量。

* **模块请求初始化**

PHP通过zend\_activate\_modules函数实现模块的请求初始化，也就是我们在图中看到Call each extension's RINIT。 此函数通过遍历注册在module\_registry变量中的所有模块，调用其RINIT方法实现模块的请求初始化操作。

**运行**

php\_execute\_script函数包含了运行PHP脚本的全部过程。

当一个PHP文件需要解析执行时，它可能会需要执行三个文件，其中包括一个前置执行文件、当前需要执行的主文件和一个后置执行文件。 非当前的两个文件可以在php.ini文件通过auto\_prepend\_file参数和auto\_append\_file参数设置。 如果将这两个参数设置为空，则禁用对应的执行文件。

对于需要解析执行的文件，通过zend\_compile\_file（compile\_file函数）做词法分析、语法分析和中间代码生成操作，返回此文件的所有中间代码。 如果解析的文件有生成有效的中间代码，则调用zend\_execute（execute函数）执行中间代码。 如果在执行过程中出现异常并且用户有定义对这些异常的处理，则调用这些异常处理函数。 在所有的操作都处理完后，PHP通过EG(return\_value\_ptr\_ptr)返回结果。

**DEACTIVATION（钝化）**

PHP关闭请求的过程是一个若干个关闭操作的集合，这个集合存在于php\_request\_shutdown函数中。 这个集合包括如下内容：

1. 调用所有通过register\_shutdown\_function()注册的函数。这些在关闭时调用的函数是在用户空间添加进来的。 一个简单的例子，我们可以在脚本出错时调用一个统一的函数，给用户一个友好一些的页面，这个有点类似于网页中的404页面。
2. 执行所有可用的\_\_destruct函数。 这里的析构函数包括在对象池（EG(objects\_store）中的所有对象的析构函数以及EG(symbol\_table)中各个元素的析构方法。
3. 将所有的输出刷出去。
4. 发送HTTP应答头。这也是一个输出字符串的过程，只是这个字符串可能符合某些规范。
5. 遍历每个模块的关闭请求方法，执行模块的请求关闭操作，这就是我们在图中看到的Call each extension's RSHUTDOWN。
6. 销毁全局变量表（PG(http\_globals)）的变量。
7. 通过zend\_deactivate函数，关闭词法分析器、语法分析器和中间代码执行器。
8. 调用每个扩展的post-RSHUTDOWN函数。只是基本每个扩展的post\_deactivate\_func函数指针都是NULL。
9. 关闭SAPI，通过sapi\_deactivate销毁SG(sapi\_headers)、SG(request\_info)等的内容。
10. 关闭流的包装器、关闭流的过滤器。
11. 关闭内存管理。
12. 重新设置最大执行时间

**结束**

最终到了要收尾的地方了。

* **flush**

sapi\_flush将最后的内容刷新出去。其调用的是sapi\_module.flush，在CLI模式下等价于fflush函数。

* **关闭Zend引擎**

zend\_shutdown将关闭Zend引擎。

此时对应图中的流程，我们应该是执行每个模块的关闭模块操作。 在这里只有一个zend\_hash\_graceful\_reverse\_destroy函数将module\_registry销毁了。 当然，它最终也是调用了关闭模块的方法的，其根源在于在初始化module\_registry时就设置了这个hash表析构时调用ZEND\_MODULE\_DTOR宏。 而ZEND\_MODULE\_DTOR宏对应的是module\_destructor函数。 在此函数中会调用模块的module\_shutdown\_func方法，即PHP\_RSHUTDOWN\_FUNCTION宏产生的那个函数。

在关闭所有的模块后，PHP继续销毁全局函数表，销毁全局类表、销售全局变量表等。 通过zend\_shutdown\_extensions遍历zend\_extensions所有元素，调用每个扩展的shutdown函数。

### 多进程SAPI生命周期

通常PHP是编译为apache的一个模块来处理PHP请求。Apache一般会采用多进程模式， Apache启动后会fork（复制）出多个子进程，每个进程的内存空间独立，每个子进程都会经过开始和结束环节， 不过每个进程的开始阶段只在进程fork出来以来后进行，在整个进程的生命周期内可能会处理多个请求。 只有在Apache关闭或者进程被结束之后才会进行关闭阶段（mshutdown），在这两个阶段之间会随着每个请求重复请求开始-请求关闭的环节。 如图2.2所示：

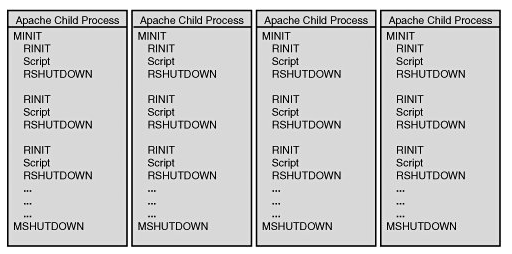


图2.2 多进程SAPI生命周期

### 多线程的SAPI生命周期

多线程模式和多进程中的某个进程类似，不同的是在整个进程的生命周期内会**并行**的重复着 请求开始-请求关闭的环节( me:在一个进程内)

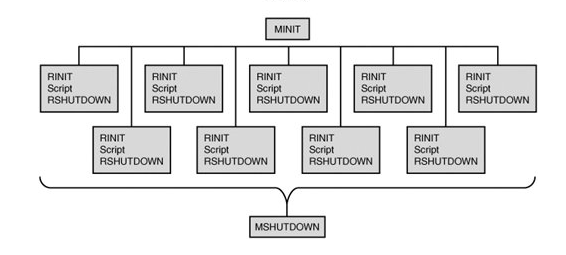


图2.3 多线程SAPI生命周期

## Zend引擎

Zend引擎是PHP实现的核心，提供了语言实现上的基础设施。例如：PHP的语法实现，脚本的编译运行环境， 扩展机制以及内存管理等，当然这里的PHP指的是官方的PHP实现(除了官方的实现， 目前比较知名的有facebook的hiphop实现，不过到目前为止，PHP还没有一个标准的语言规范)， 而PHP则提供了请求处理和其他Web服务器的接口(SAPI)。

目前PHP的实现和Zend引擎之间的关系非常紧密，甚至有些过于紧密了，例如很多PHP扩展都是使用的Zend API， 而Zend正是PHP语言本身的实现，PHP只是使用Zend这个内核来构建PHP语言的，而PHP扩展大都使用Zend API， 这就导致PHP的很多扩展和Zend引擎耦合在一起了，在笔者编写这本书的时候PHP核心开发者就提出将这种耦合解开，

目前PHP的受欢迎程度是毋庸置疑的，但凡流行的语言通常都会出现这个语言的其他实现版本， 这在Java社区里就非常明显，目前已经有非常多基于JVM的语言了，例如IBM的Project Zero就实现了一个基于JVM的PHP实现， .NET也有类似的实现，通常他们这样做的原因无非是因为：他们喜欢这个语言，但又不想放弃原有的平台， 或者对现有的语言实现不满意，处于性能或者语言特性等（HipHop就是这样诞生的）。

很多脚本语言中都会有语言扩展机制，PHP中的扩展通常是通过Pear库或者原生扩展，在Ruby中则这两者的界限不是很明显， 他们甚至会提供两套实现，一个主要用于在无法编译的环境下使用，而在合适的环境则使用C实现的原生扩展， 这样在效率和可移植性上都可以保证。目前这些为PHP编写的扩展通常都无法在其他的PHP实现中实现重用， HipHop的做法是对最为流行的扩展进行重写。如果PHP扩展能和ZendAPI解耦，则在其他语言中重用这些扩展也将更加容易了。

## 参考文献

Extending and Embedding PHP

第二节 SAPI概述

前一小节介绍了PHP的生命周期，在其生命周期的各个阶段，一些与服务相关的操作都是通过SAPI接口实现（me：服务端引用程序接口-处理服务相关的操作）。 这些内置实现的物理位置在PHP源码的SAPI目录。这个目录存放了PHP对各个服务器抽象层的代码， 例如命令行程序的实现，Apache的mod\_php模块实现以及fastcgi的实现等等。

在各个服务器抽象层之间遵守着相同的约定，这里我们称之为SAPI接口。 每个SAPI实现都是一个\_sapi\_module\_struct结构体变量。（SAPI接口）。 在PHP的源码中，当需要调用服务器相关信息时，全部通过SAPI接口中对应方法调用实现， 而这对应的方法在各个服务器抽象层实现时都会有各自的实现。

由于很多操作的通用性，有很大一部分的接口方法使用的是默认方法。

如图2.4所示，为SAPI的简单示意图。

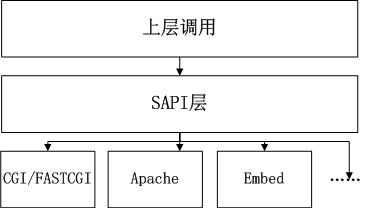


图2.4 SAPI的简单示意图

以cgi模式和apache2服务器为例，它们的启动方法如下：

cgi\_sapi\_module.startup(&cgi\_sapi\_module) *// cgi模式 cgi/cgi\_main.c文件*

apache2\_sapi\_module.startup(&apache2\_sapi\_module);

*// apache2服务器 apache2handler/sapi\_apache2.c文件*

这里的cgi\_sapi\_module是sapi\_module\_struct结构体的静态变量。 它的startup方法指向php\_cgi\_startup函数指针。在这个结构体中除了startup函数指针，还有许多其它方法或字段。 其部分定义如下：

struct \_sapi\_module\_struct {

char \*name; *// 名字（标识用）*

char \*pretty\_name; *// 更好理解的名字（自己翻译的）*

int (\*startup)(struct \_sapi\_module\_struct \*sapi\_module); *// 启动函数*

int (\*shutdown)(struct \_sapi\_module\_struct \*sapi\_module); *// 关闭方法*

int (\*activate)(TSRMLS\_D); *// 激活*

int (\*deactivate)(TSRMLS\_D); *// 停用*

int (\*ub\_write)(const char \*str, unsigned int str\_length TSRMLS\_DC);

*// 不缓存的写操作(unbuffered write)*

void (\*flush)(void \*server\_context); *// flush*

struct stat \*(\*get\_stat)(TSRMLS\_D); *// get uid*

char \*(\*getenv)(char \*name, size\_t name\_len TSRMLS\_DC); *// getenv*

void (\*sapi\_error)(int type, const char \*error\_msg, ...); */\* error handler \*/*

int (\*header\_handler)(sapi\_header\_struct \*sapi\_header, sapi\_header\_op\_enum op,

sapi\_headers\_struct \*sapi\_headers TSRMLS\_DC); */\* header handler \*/*

*/\* send headers handler \*/*

int (\*send\_headers)(sapi\_headers\_struct \*sapi\_headers TSRMLS\_DC);

void (\*send\_header)(sapi\_header\_struct \*sapi\_header,

void \*server\_context TSRMLS\_DC); */\* send header handler \*/*

int (\*read\_post)(char \*buffer, uint count\_bytes TSRMLS\_DC); */\* read POST data \*/*

char \*(\*read\_cookies)(TSRMLS\_D); */\* read Cookies \*/*

*/\* register server variables \*/*

void (\*register\_server\_variables)(zval \*track\_vars\_array TSRMLS\_DC);

void (\*log\_message)(char \*message); */\* Log message \*/*

time\_t (\*get\_request\_time)(TSRMLS\_D); */\* Request Time \*/*

void (\*terminate\_process)(TSRMLS\_D); */\* Child Terminate \*/*

char \*php\_ini\_path\_override; *// 覆盖的ini路径*

...

...

};

其中一些函数指针的说明如下：

* startup 当SAPI初始化时，首先会调用该函数。如果服务器处理多个请求时，该函数只会调用一次。 比如Apache的SAPI，它是以mod\_php5的Apache模块的形式加载到Apache中的， 在这个SAPI中，startup函数只在父进程中创建一次，在其fork的子进程中不会调用。
* activate 此函数会在每个请求开始时调用，它会再次初始化每个请求前的数据结构。
* deactivate 此函数会在每个请求结束时调用，它用来确保所有的数据都，以及释放在activate中初始化的数据结构。
* shutdown 关闭函数，它用来释放所有的SAPI的数据结构、内存等。
* ub\_write 不缓存的写操作(unbuffered write)，它是用来将PHP的数据输出给客户端， 如在CLI模式下，其最终是调用fwrite实现向标准输出输出内容；在Apache模块中，它最终是调用Apache提供的方法rwrite。
* sapi\_error 报告错误用，大多数的SAPI都是使用的PHP的默认实现php\_error。
* flush 刷新输出，在CLI模式下通过使用C语言的库函数fflush实现，在php\_mode5模式下，使用Apache的提供的函数函数rflush实现。
* read\_cookie 在SAPI激活时，程序会调用此函数，并且将此函数获取的值赋值给SG(request\_info).cookie\_data。 在CLI模式下，此函数会返回NULL。
* read\_post 此函数和read\_cookie一样也是在SAPI激活时调用，它与请求的方法相关，当请求的方法是POST时，程序会操作$\_POST、$HTTP\_RAW\_POST\_DATA等变量。
* send\_header 发送头部信息，此方法一般的SAPI都会定制，其所不同的是，有些的会调服务器自带的（如Apache），有些的需要你自己实现（如 FastCGI）。

以上的这些结构在各服务器的接口实现中都有定义。如Apache2的定义：

static sapi\_module\_struct apache2\_sapi\_module = {

"apache2handler",

"Apache 2.0 Handler",

php\_apache2\_startup, */\* startup \*/*

php\_module\_shutdown\_wrapper, */\* shutdown \*/*

...

}

在PHP的源码中实现了很多的实现，比如IIS的实现以及一些非主流的Web服务器实现，其文件结构如图2.5所示：

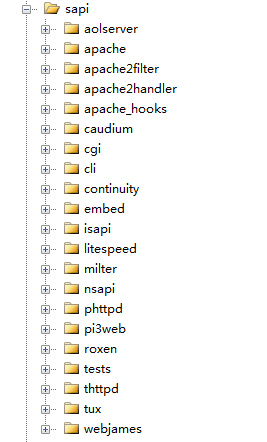


图2.5 SAPI文件结构图

目前PHP内置的很多SAPI实现都已不再维护或者变的有些非主流了，PHP社区目前正在考虑将一些SAPI移出代码库。 社区对很多功能的考虑是除非真的非常必要，或者某些功能已近非常通用了，否则就在PECL库中， 例如非常流行的APC缓存扩展将进入核心代码库中。

整个SAPI类似于一个面向对象中的模板方法模式的应用。 SAPI.c和SAPI.h文件所包含的一些函数就是模板方法模式中的抽象模板， 各个服务器对于sapi\_module的定义及相关实现则是一个个具体的模板。

这样的结构在PHP的源码中有多处使用， 比如在PHP扩展开发中，每个扩展都需要定义一个zend\_module\_entry结构体。 这个结构体的作用与sapi\_module\_struct结构体类似，都是一个类似模板方法模式的应用。 在PHP的生命周期中如果需要调用某个扩展，其调用的方法都是zend\_module\_entry结构体中指定的方法， 如在上一小节中提到的在执行各个扩展的请求初始化时，都是统一调用request\_startup\_func方法， 而在每个扩展的定义时，都通过宏PHP\_RINIT指定request\_startup\_func对应的函数。 以VLD扩展为例：其请求初始化为PHP\_RINIT(vld),与之对应在扩展中需要有这个函数的实现：

PHP\_RINIT\_FUNCTION(vld) {

}

所以， 我们在写扩展时也需要实现扩展的这些接口，同样，当实现各服务器接口时也需要实现其对应的SAPI。