PHP说简单，但是要精通也不是一件简单的事。我们除了会使用之外，还得知道它底层的工作原理。

PHP是一种适用于web开发的动态语言。具体点说，就是一个用C语言实现包含大量组件的软件框架。更狭义点看，可以把它认为是一个强大的UI框架。

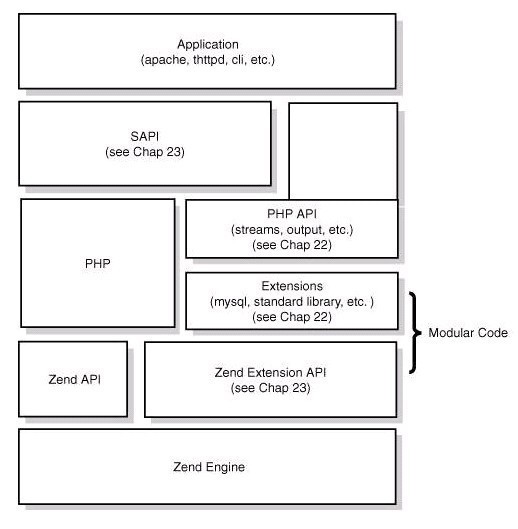
了解PHP底层实现的目的是什么？动态语言要像用好首先得了解它，内存管理、框架模型值得我们借鉴，通过扩展开发实现更多更强大的功能，优化我们程序的性能。

**1. PHP的设计理念及特点**

* 多进程模型：由于PHP是多进程模型，不同请求间互不干涉，这样保证了一个请求挂掉不会对全盘服务造成影响，当然，随着时代发展，PHP也早已支持多线程模型。
* 弱类型语言：和C/C++、Java、C#等语言不同，PHP是一门弱类型语言。一个变量的类型并不是一开始就确定不变，运行中才会确定并可能发生隐式或显式的类型转换，这种机制的灵活性在web开发中非常方便、高效，具体会在后面PHP变量中详述。
* 引擎(Zend)+组件(ext)的模式降低内部耦合。
* 中间层(sapi)隔绝web server和PHP。
* 语法简单灵活，没有太多规范。缺点导致风格混杂，但再差的程序员也不会写出太离谱危害全局的程序。

**2. PHP的四层体系**

PHP的核心架构如下图：



从图上可以看出，PHP从下到上是一个4层体系：

* Zend引擎：Zend整体用纯C实现，是PHP的内核部分，它将PHP代码翻译（词法、语法解析等一系列编译过程）为可执行opcode的处理并实现相应的处理方法、实现了基本的数据结构（如hashtable、oo）、内存分配及管理、提供了相应的api方法供外部调用，是一切的核心，所有的外围功能均围绕Zend实现。
* Extensions：围绕着Zend引擎，extensions通过组件式的方式提供各种基础服务，我们常见的各种内置函数（如array系列）、标准库等都是通过extension来实现，用户也可以根据需要实现自己的extension以达到功能扩展、性能优化等目的（如贴吧正在使用的PHP中间层、富文本解析就是extension的典型应用）。
* Sapi：Sapi全称是Server Application Programming Interface，也就是服务端应用编程接口，Sapi通过一系列钩子函数，使得PHP可以和外围交互数据，这是PHP非常优雅和成功的一个设计，通过sapi成功的将PHP本身和上层应用解耦隔离，PHP可以不再考虑如何针对不同应用进行兼容，而应用本身也可以针对自己的特点实现不同的处理方式。
* 上层应用：这就是我们平时编写的PHP程序，通过不同的sapi方式得到各种各样的应用模式，如通过webserver实现web应用、在命令行下以脚本方式运行等等。

如果PHP是一辆车，那么车的框架就是PHP本身，Zend是车的引擎（发动机），Ext下面的各种组件就是车的轮子，Sapi可以看做是公路，车可以跑在不同类型的公路上，而一次PHP程序的执行就是汽车跑在公路上。因此，我们需要：性能优异的引擎+合适的车轮+正确的跑道。

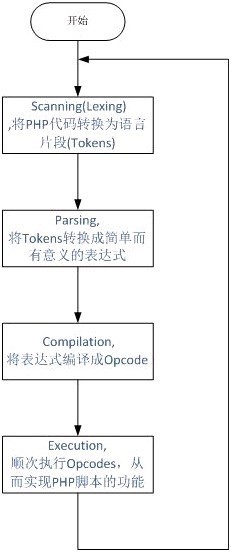
**3. Sapi**

如前所述，Sapi通过通过一系列的接口，使得外部应用可以和PHP交换数据并可以根据不同应用特点实现特定的处理方法，我们常见的一些sapi有：

* apache2handler：这是以apache作为webserver，采用mod\_PHP模式运行时候的处理方式，也是现在应用最广泛的一种。
* cgi：这是webserver和PHP直接的另一种交互方式，也就是大名鼎鼎的fastcgi协议，在最近今年fastcgi+PHP得到越来越多的应用，也是异步webserver所唯一支持的方式。
* cli：命令行调用的应用模式

**4. PHP的执行流程&opcode**

我们先来看看PHP代码的执行所经过的流程。



从图上可以看到，PHP实现了一个典型的动态语言执行过程：拿到一段代码后，经过词法解析、语法解析等阶段后，源程序会被翻译成一个个指令(opcodes)，然后ZEND虚拟机顺次执行这些指令完成操作。PHP本身是用C实现的，因此最终调用的也都是C的函数，实际上，我们可以把PHP看做是一个C开发的软件。

PHP的执行的核心是翻译出来的一条一条指令，也即opcode。

Opcode是PHP程序执行的最基本单位。一个opcode由两个参数(op1,op2)、返回值和处理函数组成。PHP程序最终被翻译为一组opcode处理函数的顺序执行。

常见的几个处理函数：

|  |  |  |
| --- | --- | --- |
| 1 | ZEND\_ASSIGN\_SPEC\_CV\_CV\_HANDLER : 变量分配 （$a=$b） | |
| 2 | ZEND\_DO\_FCALL\_BY\_NAME\_SPEC\_HANDLER：函数调用 |

|  |  |  |
| --- | --- | --- |
| 3 | ZEND\_CONCAT\_SPEC\_CV\_CV\_HANDLER：字符串拼接 $a.$b | |
| 4 | ZEND\_ADD\_SPEC\_CV\_CONST\_HANDLER: 加法运算 $a+2 |

|  |  |
| --- | --- |
| 5 | ZEND\_IS\_EQUAL\_SPEC\_CV\_CONST：判断相等 $a==1 |
| 6 | ZEND\_IS\_IDENTICAL\_SPEC\_CV\_CONST：判断相等 $a===1 | |

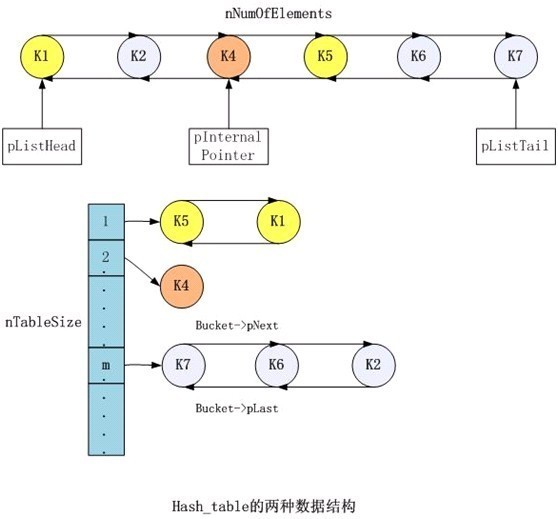
**5. HashTable — 核心数据结构**

HashTable是zend的核心数据结构，在PHP里面几乎并用来实现所有常见功能，我们知道的PHP数组即是其典型应用，此外，在zend内部，如函数符号表、全局变量等也都是基于hash table来实现。

PHP的hash table具有如下特点：

* 支持典型的key->value查询
* 可以当做数组使用
* 添加、删除节点是O（1）复杂度
* key支持混合类型：同时存在关联数组合索引数组
* Value支持混合类型：array (“string”,2332)
* 支持线性遍历：如foreach

Zend hash table实现了典型的hash表散列结构，同时通过附加一个双向链表，提供了正向、反向遍历数组的功能。其结构如下图：



可以看到，在hash table中既有key->value形式的散列结构，也有双向链表模式，使得它能够非常方便的支持快速查找和线性遍历。

* 散列结构：Zend的散列结构是典型的hash表模型，通过链表的方式来解决冲突。需要注意的是zend的hash table是一个自增长的数据结构，当hash表数目满了之后，其本身会动态以2倍的方式扩容并重新元素位置。初始大小均为8。另外，在进行key->value快速查找时候，zend本身还做了一些优化，通过空间换时间的方式加快速度。比如在每个元素中都会用一个变量nKeyLength标识key的长度以作快速判定。
* 双向链表：Zend hash table通过一个链表结构，实现了元素的线性遍历。理论上，做遍历使用单向链表就够了，之所以使用双向链表，主要目的是为了快速删除，避免遍历。Zend hash table是一种复合型的结构，作为数组使用时，即支持常见的关联数组也能够作为顺序索引数字来使用，甚至允许2者的混合。
* PHP关联数组：关联数组是典型的hash\_table应用。一次查询过程经过如下几步（从代码可以看出，这是一个常见的hash查询过程并增加一些快速判定加速查找。）：

|  |  |
| --- | --- |
| 01 | getKeyHashValue h; |
| 02 | index = n & nTableMask; | |

|  |  |  |
| --- | --- | --- |
| 03 | Bucket \*p = arBucket[index]; | |
| 04 | while (p) { |

|  |  |  |
| --- | --- | --- |
| 05 | if ((p->h == h) && (p->nKeyLength == nKeyLength)) { | |
| 06 | RETURN p->data; |

|  |  |
| --- | --- |
| 07 | } |
| 08 | p=p->next; | |

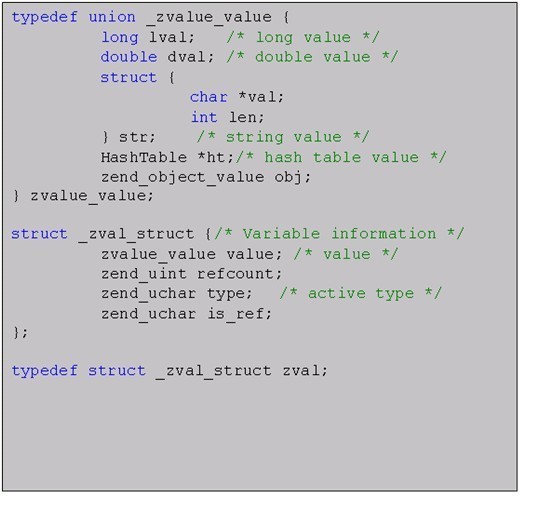
|  |  |
| --- | --- |
| 09 | } |
| 10 | RETURN FALTURE; | |

* PHP索引数组：索引数组就是我们常见的数组，通过下标访问。例如 $arr[0]，Zend HashTable内部进行了归一化处理，对于index类型key同样分配了hash值和nKeyLength(为0)。内部成员变量nNextFreeElement就是当前分配到的最大id，每次push后自动加一。正是这种归一化处理，PHP才能够实现关联和非关联的混合。由于push操作的特殊性，索引key在PHP数组中先后顺序并不是通过下标大小来决定，而是由push的先后决定。例如 $arr[1] = 2; $arr[2] = 3;对于double类型的key，Zend HashTable会将他当做索引key处理

**6. PHP变量**

PHP是一门弱类型语言，本身不严格区分变量的类型。PHP在变量申明的时候不需要指定类型。PHP在程序运行期间可能进行变量类型的隐示转换。和其他强类型语言一样，程序中也可以进行显示的类型转换。PHP变量可以分为简单类型(int、string、bool)、集合类型(array resource object)和常量(const)。以上所有的变量在底层都是同一种结构 zval。

Zval是zend中另一个非常重要的数据结构，用来标识并实现PHP变量，其数据结构如下：



Zval主要由三部分组成：

* type：指定了变量所述的类型（整数、字符串、数组等）
* refcount&is\_ref：用来实现引用计数(后面具体介绍)
* value：核心部分，存储了变量的实际数据

Zvalue是用来保存一个变量的实际数据。因为要存储多种类型，所以zvalue是一个union，也由此实现了弱类型。

PHP变量类型和其实际存储对应关系如下：

|  |  |
| --- | --- |
| 1 | IS\_LONG   -> lvalue |
| 2 | IS\_DOUBLE -> dvalue |

|  |  |  |
| --- | --- | --- |
| 3 | IS\_ARRAY  -> ht | |
| 4 | IS\_STRING -> str |

|  |  |
| --- | --- |
| 5 | IS\_RESOURCE -> lvalue |

引用计数在内存回收、字符串操作等地方使用非常广泛。PHP中的变量就是引用计数的典型应用。Zval的引用计数通过成员变量is\_ref和ref\_count实现，通过引用计数，多个变量可以共享同一份数据。避免频繁拷贝带来的大量消耗。

在进行赋值操作时，zend将变量指向相同的zval同时ref\_count++，在unset操作时，对应的ref\_count-1。只有ref\_count减为0时才会真正执行销毁操作。如果是引用赋值，则zend会修改is\_ref为1。

PHP变量通过引用计数实现变量共享数据，那如果改变其中一个变量值呢？当试图写入一个变量时，Zend若发现该变量指向的zval被多个变量共享，则为其复制一份ref\_count为1的zval，并递减原zval的refcount，这个过程称为“zval分离”。可见，只有在有写操作发生时zend才进行拷贝操作，因此也叫copy-on-write(写时拷贝)

对于引用型变量，其要求和非引用型相反，引用赋值的变量间必须是捆绑的，修改一个变量就修改了所有捆绑变量。

整数、浮点数是PHP中的基础类型之一，也是一个简单型变量。对于整数和浮点数，在zvalue中直接存储对应的值。其类型分别是long和double。

从zvalue结构中可以看出，对于整数类型，和c等强类型语言不同，PHP是不区分int、unsigned int、long、long long等类型的，对它来说，整数只有一种类型也就是long。由此，可以看出，在PHP里面，整数的取值范围是由编译器位数来决定而不是固定不变的。

对于浮点数，类似整数，它也不区分float和double而是统一只有double一种类型。

在PHP中，如果整数范围越界了怎么办？这种情况下会自动转换为double类型，这个一定要小心，很多trick都是由此产生。

和整数一样，字符变量也是PHP中的基础类型和简单型变量。通过zvalue结构可以看出，在PHP中，字符串是由由指向实际数据的指针和长度结构体组成，这点和c++中的string比较类似。由于通过一个实际变量表示长度，和c不同，它的字符串可以是2进制数据（包含\0），同时在PHP中，求字符串长度strlen是O(1)操作。

在新增、修改、追加字符串操作时，PHP都会重新分配内存生成新的字符串。最后，出于安全考虑，PHP在生成一个字符串时末尾仍然会添加\0

常见的字符串拼接方式及速度比较：

假设有如下4个变量：$strA=‘123’; $strB = ‘456’; $intA=123; intB=456;

现在对如下的几种字符串拼接方式做一个比较和说明：

|  |  |
| --- | --- |
| 1 | $res = $strA.$strB和$res = “$strA$strB” |
| 2 | 这种情况下，zend会重新malloc一块内存并进行相应处理，其速度一般 | |

|  |  |
| --- | --- |
| 3 | $strA = $strA.$strB |
| 4 | 这种是速度最快的，zend会在当前strA基础上直接relloc，避免重复拷贝 | |

|  |  |
| --- | --- |
| 5 | $res = $intA.$intB |
| 6 | 这种速度较慢，因为需要做隐式的格式转换，实际编写程序中也应该注意尽量避免 | |

|  |  |
| --- | --- |
| 7 | $strA = sprintf (“%s%s”,$strA.$strB); |
| 8 | 这会是最慢的一种方式，因为sprintf在PHP中并不是一个语言结构，本身对于格式识别和处理就需要耗费比较多时间，另外本身机制也是malloc。不过sprintf的方式最具可读性，实际中可以根据具体情况灵活选择。 | |

PHP的数组通过Zend HashTable来天然实现。

foreach操作如何实现？对一个数组的foreach就是通过遍历hashtable中的双向链表完成。对于索引数组，通过foreach遍历效率比for高很多，省去了key->value的查找。count操作直接调用HashTable->NumOfElements，O(1)操作。对于’123’这样的字符串，zend会转换为其整数形式。$arr[‘123’]和$arr[123]是等价的

资源类型变量是PHP中最复杂的一种变量，也是一种复合型结构。

PHP的zval可以表示广泛的数据类型，但是对于自定义的数据类型却很难充分描述。由于没有有效的方式描绘这些复合结构，因此也没有办法对它们使用传统的操作符。要解决这个问题，只需要通过一个本质上任意的标识符（label）引用指针，这种方式被称为资源。

在zval中，对于resource，lval作为指针来使用，直接指向资源所在的地址。Resource可以是任意的复合结构，我们熟悉的mysqli、fsock、memcached等都是资源。

如何使用资源：

* 注册：对于一个自定义的数据类型，要想将它作为资源。首先需要进行注册，zend会为它分配全局唯一标示。
* 获取一个资源变量：对于资源，zend维护了一个id->实际数据的hash\_tale。对于一个resource，在zval中只记录了它的id。fetch的时候通过id在hash\_table中找到具体的值返回。
* 资源销毁：资源的数据类型是多种多样的。Zend本身没有办法销毁它。因此需要用户在注册资源的时候提供销毁函数。当unset资源时，zend调用相应的函数完成析构。同时从全局资源表中删除它。

资源可以长期驻留，不只是在所有引用它的变量超出作用域之后，甚至是在一个请求结束了并且新的请求产生之后。这些资源称为持久资源，因为它们贯通SAPI的整个生命周期持续存在，除非特意销毁。很多情况下，持久化资源可以在一定程度上提高性能。比如我们常见的mysql\_pconnect ,持久化资源通过pemalloc分配内存，这样在请求结束的时候不会释放。 对zend来说，对两者本身并不区分。

PHP中的局部变量和全局变量是如何实现的？对于一个请求，任意时刻PHP都可以看到两个符号表(symbol\_table和active\_symbol\_table)，其中前者用来维护全局变量。后者是一个指针，指向当前活动的变量符号表，当程序进入到某个函数中时，zend就会为它分配一个符号表x同时将active\_symbol\_table指向a。通过这样的方式实现全局、局部变量的区分。

获取变量值：PHP的符号表是通过hash\_table实现的，对于每个变量都分配唯一标识，获取的时候根据标识从表中找到相应zval返回。

函数中使用全局变量：在函数中，我们可以通过显式申明global来使用全局变量。在active\_symbol\_table中创建symbol\_table中同名变量的引用，如果symbol\_table中没有同名变量则会先创建。