PHP 内核介绍及扩展开发指南

zhangdongjin@baidu.com

目录

目录 2
一、 基础知识 4
1.1 PHP 变量的存储 4
1.1.1 zval 结构 4
1.1.2 引用计数 5
1.1.3 zval 状态 5
1.1.4 zval 状态切换 6
1.1.5 参数传递 9
1.2 HashTable 结构 9
1.2.1 数据结构 10
1.2.2 PHP 数组 13
1.2.3 变量符号表 14
1.3 内存和文件 15
二、 Extensions 的编写 17
2.1 Hello World 17
2.1.1 声明导出函数 19
2.1.2 声明导出函数块 20
2.1.3 填写模块信息 21
2.1.4 实现导出函数 23
2.2 使用参数 24
2.2.1 标准方法 25
2.2.2 底层方法 27
2.2.3 引用传递 29
2.2.4 编译检查(TODO) 30
2.3 返回值 30
2.3.1 返回引用 31
2.4 启动和终止函数 33
2.5 调用 PHP 函数 34
2.6 访问 PHP 变量 37
2.6.1 设置 37
2.6.2 获取 38
2.6.3 常量 39
2.7 输出信息 42
三、 高级主题 43
3.1 使用数组 43
3.1.1 关联数组元素 43
3.1.2 非关联数组元素 44
3.2 使用资源 45
3.2.1 注册资源类型 45
3.2.2 注册资源 46
3.2.3 获取资源 47
3.2.4 维护引用计数 48

四、 类和对象(TODO).....50
附录 A. Extension 的编译.....51
附录 B. Extension 的加载过程.....52

一、 基础知识

本章简要介绍一些Zend引擎的内部机制，这些知识和Extensions密切相关，同时也可以帮助我们写出更加高效的PHP代码。

1.1 PHP变量的存储

1.1.1 zval结构

Zend使用zval结构来存储PHP变量的值，该结构如下所示：

```
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;
} zvalue_value;


struct _zval_struct {
    /* Variable information */
    zvalue_value value;     /* value */
    zend_uint refcount;
    zend_uchar type;            /* active type */
    zend_uchar is_ref;
};

typedef struct _zval_struct zval;
```

Zend根据type值来决定访问value的哪个成员，可用值如下：

IS_NULL	N/A
IS_LONG	对应 value.lval
IS_DOUBLE	对应 value.dval
IS_STRING	对应 value.str
IS_ARRAY	对应 value.ht
IS_OBJECT	对应 value.obj
IS_BOOL	对应 value.lval.
IS_RESOURCE	对应 value.lval

根据这个表格可以发现两个有意思的地方：首先是 PHP 的数组其实就是一个 HashTable，这就解释了为什么 PHP 能够支持关联数组了；其次，Resource 就是一个 long 值，它里面存放的通常是个指针、一个内部数组的 index 或者其它什么只有创建者自己才知道的东西，可以将其视作一个 handle。

1.1.2 引用计数

引用计数在垃圾收集、内存池以及字符串等地方应用广泛，Zend 就实现了典型的引用计数。多个 PHP 变量可以通过引用计数机制来共享同一份 zval，zval 中剩余的两个成员 is_ref 和 refcount 就用来支持这种共享。

很明显，refcount 用于计数，当增减引用时，这个值也相应的递增和递减，一旦减到零，Zend 就会回收该 zval。

那么 is_ref 呢？

1.1.3 zval 状态

在 PHP 中，变量有两种——引用和非引用的，它们在 Zend 中都

是采用引用计数的方式存储的。对于非引用型变量，要求变量间互不相干，修改一个变量时，不能影响到其他变量，采用 Copy-On-Write 机制即可解决这种冲突——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 refcount 为 1 的 zval，并递减原 zval 的 refcount，这个过程称为“zval 分离”。然而，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。

可见，有必要指出当前 zval 的状态，以分别应对这两种情况，is_ref 就是这个目的，它指出了当前所有指向该 zval 的变量是否是采用引用赋值的——要么全是引用，要么全不是。此时再修改一个变量，只有当发现其 zval 的 is_ref 为 0，即非引用时，Zend 才会执行 Copy-On-Write。

1.1.4 zval 状态切换

当在一个 zval 上进行的所有赋值操作都是引用或者都是非引用时，一个 is_ref 就足够应付了。然而，世界总不会那么美好，PHP 无法对用户进行这种限制，当我们混合使用引用和非引用赋值时，就必须要进行特别处理了。

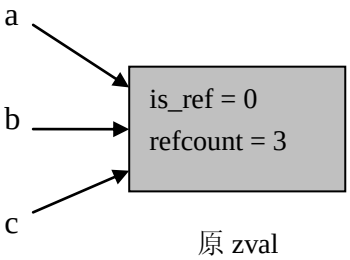
情况 I、看如下 PHP 代码：

```php
<?php
$a = 1;
$b = $a;
$c = $b;
$d = &$c; // 在一堆非引用赋值中，插入一个引用
?>
```

这段代码首先进行了一次初始化，这将创建一个新的 zval，is_ref=0, refcount=1，并将 a 指向这个 zval；之后是两次非引用赋值，正如前面所说，只要把 b 和 c 都指向 a 的 zval 即可；最后一行是个引用赋值，需要 is_ref 为 1，但是 Zend 发现 c 指向的 zval 并不是引用型的，于是为 c 创建单独的 zval，并同时将 d 指向该 zval。

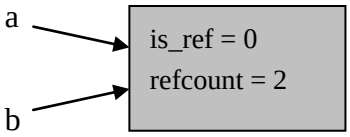
从本质上来说，这也可以看作是一种 Copy-On-Write，不仅仅是 value，is_ref 也是受保护的对象。

整个过程图示如下：

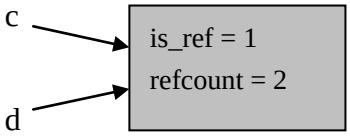


原 zval

引用赋值前



原 zval



新 zval

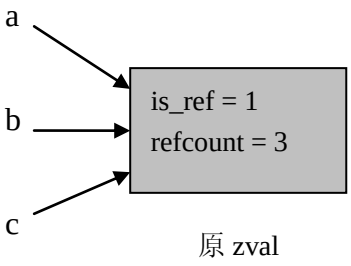
引用赋值后

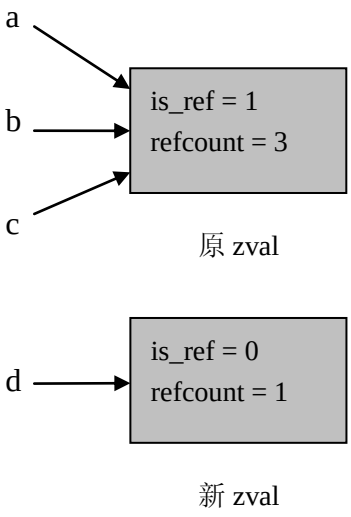
情况 II，看如下 PHP 代码：

```php
<?php
$a = 1;
$b = &$a;
$c = &$b;
$d = $c; // 在一堆引用赋值中，插入一个非引用
?>
```

这段代码的前三句将把 a、b 和 c 指向一个 zval，其 is_ref=1, refcount=3；第四句是个非引用赋值，通常情况下只需要增加引用计数即可，然而目标 zval 属于引用变量，单纯的增加引用计数显然是错误的， Zend 的解决办法是为 d 单独生成一份 zval 副本。

全过程如下所示：





非引用赋值前　　非引用赋值后

1.1.5 参数传递

PHP 函数参数的传递和变量赋值是一样的，非引用传递相当于非引用赋值，引用传递相当于引用赋值，并且也有可能会导致执行 zval 状态切换。这在后面还将提到。

1.2 HashTable 结构

HashTable 是 Zend 引擎中最重要、使用最广泛的数据结构，它被用来存储几乎所有的东西。

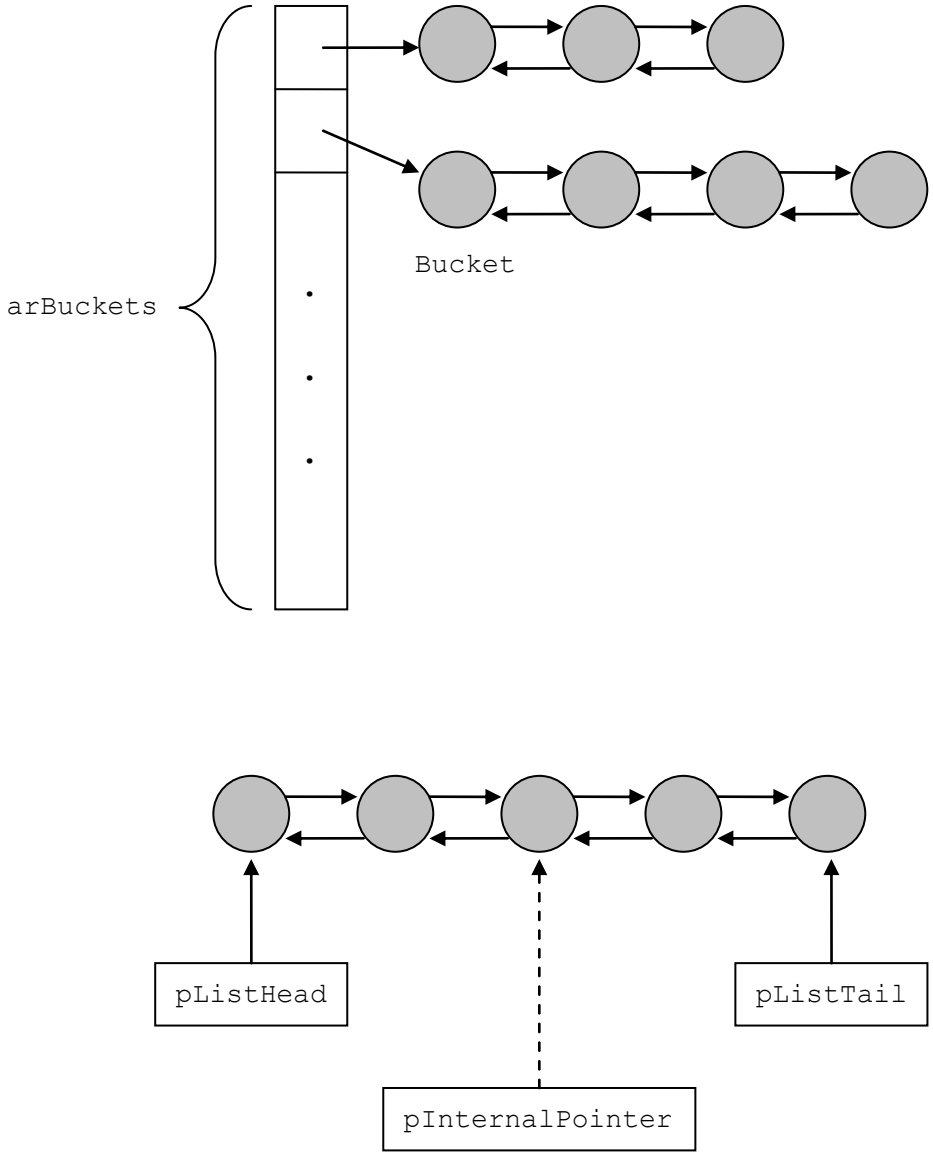
1.2.1 数据结构

HashTable 数据结构定义如下：

```
typedef struct bucket {
    ulong h;                // 存放 hash
    uint nKeyLength;
    void *pData;            // 指向 value，是用户数据的副本
    void *pDataPtr;
    struct bucket *pListNext;   // pListNext 和 pListLast 组成
    struct bucket *pListLast;   // 整个 HashTable 的双链表
    struct bucket *pNext;       // pNext 和 pLast 用于组成某个 hash 对应
    struct bucket *pLast;       // 的双链表
    char arKey[1];              // key
} Bucket;

typedef struct _hashtable {
    uint nTableSize;
    uint nTableMask;
    uint nNumOfElements;
    ulong nNextFreeElement;
    Bucket *pInternalPointer;   /* Used for element traversal */
    Bucket *pListHead;
    Bucket *pListTail;
    Bucket **arBuckets;         // hash 数组
    dtor_func_t pDestructor;    // HashTable 初始化时指定，销毁 Bucket 时调用
    zend_bool persistent;       // 是否采用 C 的内存分配例程
    unsigned char nApplyCount;
    zend_bool bApplyProtection;
#if ZEND_DEBUG
    int inconsistent;
#endif
} HashTable;
```

总的来说，Zend 的 HashTable 是一种链表散列，同时也为线性遍历进行了优化，图示如下：



HashTable 里的两种数据结构

HashTable 中包含两种数据结构，一个链表散列和一个双向链表，前者用于进行快速键-值查询，后者方便线性遍历和排序，一个 Bucket 同时存在于这两个数据结构中。

关于该数据结构的几点解释：

- **链表散列中为什么使用双向链表？**

一般的链表散列只需要按 key 进行操作，只需要单链表就够了。但是，Zend 有时需要从链表散列中删除给定的 Bucket，使用双链表可以非常高效的实现。

- **nTableMask 是干什么的？**

这个值用于 hash 值到 arBuckets 数组下标的转换。当初始化一个 HashTable，Zend 首先为 arBuckets 数组分配 nTableSize 大小的内存，nTableSize 取不小于用户指定大小的最小的 2^n，即二进制的 10*。nTableMask = nTableSize – 1，即二进制的 01*，此时 h & nTableMask 就恰好落在 [0, nTableSize – 1] 里，Zend 就以其为 index 来访问 arBuckets 数组。

- **pDataPtr 是干什么的？**

通常情况下，当用户插入一个键值对时，Zend 会将 value 复制一份，并将 pData 指向 value 副本。复制操作需要调用 Zend 内部例程 emalloc 来分配内存，这是个非常耗时的操作，并且会消耗比 value 大的一块内存（多出的内存用于存放 cookie），如果 value 很小的话，将会造成较大的浪费。考虑到 HashTable 多用于存放指针值，于是 Zend 引入 pDataPtr，当 value 小到和指针一样长时，Zend 就直接将其复制到 pDataPtr 里，并且将 pData 指向 pDataPtr。这就避免了 emalloc 操作，同时也有利于提高 Cache 命中率。

- **arKey 大小为什么只有 1？为什么不使用指针管理 key？**

arKey 是存放 key 的数组，但其大小却只有 1，并不足以放下 key。

在 HashTable 的初始化函数里可以找到如下代码：

```
p = (Bucket *) pemalloc(sizeof(Bucket) - 1 + nKeyLength, ht->persistent);
```

可见，Zend 为一个 Bucket 分配了一块足够放下自己和 key 的内存，上半部分是 Bucket，下半部分是 key，而 arKey “恰好”是 Bucket 的最后一个元素，于是就可以使用 arKey 来访问 key 了。这种手法在内存管理例程中最为常见，当分配内存时，实际上是分配了比指定大小要大的内存，多出的上半部分通常被称为 cookie，它存储了这块内存的信息，比如块大小、上一块指针、下一块指针等，baidu 的 Transmit 程序就使用了这种方法。

不用指针管理 key，是为了减少一次 emalloc 操作，同时也可以提高 Cache 命中率。另一个必需的理由是，key 绝大部分情况下是固定不变的，不会因为 key 变长了而导致重新分配整个 Bucket。这同时也解释了为什么不把 value 也一起作为数组分配了——因为 value 是可变的。

1.2.2 PHP 数组

关于 HashTable 还有一个疑问没有回答，就是 nNextFreeElement 是干什么的？

不同于一般的散列，Zend 的 HashTable 允许用户直接指定 hash 值，而忽略 key，甚至可以不指定 key（此时，nKeyLength 为 0）。同时，HashTable 也支持 append 操作，用户连 hash 值也不用指定，

只需要提供 value，此时，Zend 就用 nNextFreeElement 作为 hash，之后将 nNextFreeElement 递增。

HashTable 的这种行为看起来很奇怪，因为这将无法按 key 访问 value，已经完全不是个散列了。理解问题的关键在于，PHP 数组就是使用 HashTable 实现的——关联数组使用正常的 k-v 映射将元素加入 HashTable，其 key 为用户指定的字符串；非关联数组则直接使用数组下标作为 hash 值，不存在 key；而当在一个数组中混合使用关联和非关联时，或者使用 array_push 操作时，就需要用 nNextFreeElement 了。

再来看 value，PHP 数组的 value 直接使用了 zval 这个通用结构，pData 指向的是 zval*，按照上一节的介绍，这个 zval*将直接存储在 pDataPtr 里。由于直接使用了 zval，数组的元素可以是任意 PHP 类型。

数组的遍历操作，即 foreach、each 等，是通过 HashTable 的双向链表来进行的，pInternalPointer 作为游标记录了当前位置。

1.2.3 变量符号表

除了数组，HashTable 还被用来存储许多其他数据，比如，PHP 函数、变量符号、加载的模块、类成员等。

一个变量符号表就相当于一个关联数组，其 key 是变量名（可见，使用很长的变量名并不是个好主意），value 是 zval*。

在任一时刻 PHP 代码都可以看见两个变量符号表——

symbol_table 和 active_symbol_table——前者用于存储全局变量，称为全局符号表；后者是个指针，指向当前活动的变量符号表，通常情况下就是全局符号表。但是，当每次进入一个 PHP 函数时（此处指的是用户使用 PHP 代码创建的函数），Zend 都会创建函数局部的变量符号表，并将 active_symbol_table 指向局部符号表。Zend 总是使用 active_symbol_table 来访问变量，这样就实现了局部变量的作用域控制。

但如果在函数局部访问标记为 global 的变量，Zend 会进行特殊处理——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。

1.3 内存和文件

程序拥有的资源一般包括内存和文件，对于通常的程序，这些资源是面向进程的，当进程结束后，操作系统或 C 库会自动回收那些我们没有显式释放的资源。

但是，PHP 程序有其特殊性，它是基于页面的，一个页面运行时同样也会申请内存或文件这样的资源，然而当页面运行结束后，操作系统或 C 库也许不会知道需要进行资源回收。比如，我们将 php 作为模块编译到 apache 里，并且以 prefork 或 worker 模式运行 apache。这种情况下 apache 进程或线程是复用的，php 页面分配的内存将永驻内存直到出 core。

为了解决这种问题，Zend 提供了一套内存分配 API，它们的作

用和 C 中相应函数一样，不同的是这些函数从 Zend 自己的内存池中分配内存，并且它们可以实现基于页面的自动回收。在我们的模块中，为页面分配的内存应该使用这些 API，而不是 C 例程，否则 Zend 会在页面结束时尝试 efree 掉我们的内存，其结果通常就是 crush。

emalloc()
efree()
estrdup()
estrndup()
ecalloc()
erealloc()

另外，Zend 还提供了一组形如 VCWD_xxx 的宏用于替代 C 库和操作系统相应的文件 API，这些宏能够支持 PHP 的虚拟工作目录，在模块代码中应该总是使用它们。宏的具体定义参见 PHP 源代码”TSRM/tsrm_virtual_cwd.h”。可能你会注意到，所有那些宏中并没有提供 close 操作，这是因为 close 的对象是已打开的资源，不涉及到文件路径，因此可以直接使用 C 或操作系统例程；同理，read/write 之类的操作也是直接使用 C 或操作系统的例程。

二、 Extensions 的编写

理解了这些运行机制以后，本章着手介绍 Extensions 的编写，但凡写程序的人都知道 hello world，那好，就从 hello world 开始。

2.1 Hello World

这是摘自《PHP 手册》的示例程序：

```
/* include standard header */
#include "php.h"

/* declaration of functions to be exported */
ZEND_FUNCTION(first_module);

/* compiled function list so Zend knows what's in this module */
zend_function_entry firstmod_functions[] =
{
   ZEND_FE(first_module, NULL)
   {NULL, NULL, NULL}
};

/* compiled module information */
zend_module_entry firstmod_module_entry =
{
   STANDARD_MODULE_HEADER,
   "First Module",
   firstmod_functions,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NO_VERSION_YET,
   STANDARD_MODULE_PROPERTIES
};

/* implement standard "stub" routine to introduce ourselves to Zend */
#if COMPILE_DL_FIRST_MODULE
ZEND_GET_MODULE(firstmod)
#endif

/* implement function that is meant to be made available to PHP */
ZEND_FUNCTION(first_module)
{
   long parameter;
   if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l", &parameter)
        == FAILURE)
        return;
    RETURN_LONG(parameter);
}
```

这段代码实现了一个简单的 extension，首先它包含了“php.h”，这是所有 extensions 都需要包含的头文件，它定义、声明了我们可以访问的所有 Zend 数据结构、常量和 API 等。下面对剩余的步骤进行解释。

2.1.1 声明导出函数

```
ZEND_FUNCTION(first_module);
```

ZEND_FUNCTION 宏用于声明一个可在 PHP 代码中调用的函数，其参数即成为 PHP 函数名，因此，这一句声明了一个名为 first_module 的 PHP 函数，将其展开如下：

```
void zif_first_module (INTERNAL_FUNCTION_PARAMETERS);

// 最终展开得：
void zif_first_module (
      int ht,
      zval * return_value,
      zval **return_value_ptr,
      zval * this_ptr,
      int return_value_used
      );
```

可见，ZEND_FUNCTION 就是简单的声明了一个名为 zif_first_module 的 C 函数，zif 可能是”Zend Internal Function”的缩写。函数的原型满足 Zend 引擎对 PHP 函数的调用约定，关于其参数将在后面章节进行解释。

2.1.2 声明导出函数块

声明 C 函数后，Zend 并不知道如何调用，我们需要使用如下的语句来完成 C 函数到 PHP 函数的映射：

```
zend_function_entry firstmod_functions[] =
{
  ZEND_FE(first_module, NULL)
  {NULL, NULL, NULL}
};
```

这创建了一个 zend_function_entry 数组，zend_function_entry 存储了关于如何调用该 PHP 函数的信息，通过它 Zend 引擎就能够理解和调用我们的函数。

其定义如下：

```
typedef struct _zend_function_entry {
    char *fname;
    void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
    struct _zend_arg_info *arg_info;
    zend_uint num_args;
    zend_uint flags;
} zend_function_entry;
```

fname 是 PHP 函数名，是 PHP 代码能够通过它来调用我们的函数；handler 是指向我们在前面声明的 C 函数的函数指针。这两个参数已经足以完成从 C 函数到 PHP 函数的映射。剩余的参数用于告诉 Zend 该 PHP 函数对于函数参数的要求，arg_info 是个数组，它的每一项都描述了对应下标的参数，num_args 是参数的个数，具体将在

后面的章节介绍。

我们可以手动填充一个 zend_function_entry，但更好的办法是使用 Zend 提供的宏 ZEND_FE，因为 Zend 并不保证这个结构以后不会变。ZEND_FE 使用第一个参数作为 PHP 函数名，并且在添加了 zif 前缀后作为 C 函数名；第二个参数用于填充 arg_info，通常使用 NULL。上面的代码将得到这样一个 zend_function_entry 结构：{" first_module,", zif_first_module, NULL, 0, 0}。当然，这并不是说 PHP 函数名必须和 C 函数名有什么关系，也可以通过宏 ZEND_NAMED_FE 来手动指定 PHP 函数名，不过这并不是个好主意。

我们必须为希望导出的每一个 C 函数都创建一个 zend_function_entry 结构，并将其放到一个数组中以备后用，数组最后一项的成员必须全部为 NULL，这用于标记数组的结束。

2.1.3 填写模块信息

下一步需要将我们的模块介绍给 Zend，主要包括我们的模块名和导出的函数，这通过填充一个 zend_module_entry 结构来完成。

```
zend_module_entry firstmod_module_entry =
{
   STANDARD_MODULE_HEADER,
   "First Module",
   firstmod_functions,
   NULL,
   NULL,
   NULL,
   NULL,
   NULL,
   NO_VERSION_YET,
   STANDARD_MODULE_PROPERTIES
};
```

STANDARD_MODULE_HEADER 和 STANDARD_MODULE_PROPERTIES 宏填充了该结构的首尾部分，具体填充了什么并不是我们需要关心的，并且为了兼容后续版本也最好不要手工修改。

第二、三项是模块名称和导出函数，名称可以任意填写，导出函数就是我们在前面准备好的 zend_function_entry 数组。

接下来的五个参数是函数指针，其用法在后面介绍，这里只用 NULL 填充。

下面的参数是一个 C 字符串，用于表示模块版本，如果没有则使用 NO_VERSION_YET，其实就是 NULL。

填写完毕后，需要把这个结构传给 Zend 引擎，这通过下面的语句完成：

```
#if COMPILE_DL_FIRST_MODULE
ZEND_GET_MODULE(firstmod)
#endif
```

宏开关用于判断是否是动态链接的，动态链接时才会执行下面的语句，本文仅介绍动态链接的模块，并不关心静态链接时如何与 Zend 交流信息，因此，可以认为条件总为真。

ZEND_GET_MODULE(firstmod)最后展开得到名为 get_module 的一个函数：

```
zend_module_entry *get_module(void)
{
    return &firstmod_module_entry;
}
```

这个函数就是简单的返回我们填充的 zend_module_entry 结构，这里需要注意的是结构的名称必须是 xxx_module_entry，xxx 是传递给 ZEND_GET_MODULE 的参数。当 Zend 加载我们的模块时，它首先会解析并调用名为 get_module 的函数，这样就可以得到我们的 zend_module_entry，于是，PHP 代码就可以调用模块导出的函数了。

2.1.4 实现导出函数

代码最后一部分实现了我们导出的函数：

```
ZEND_FUNCTION(first_module)
{
    long parameter;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l",
        &parameter) == FAILURE)
        return;
    RETURN_LONG(parameter);
}
```

这里依然要用 ZEND_FUNCTION 来声明函数原型，函数体通过 Zend API 和宏，访问了函数参数并返回一个 long 值——这些都将在后面的章节进行详细介绍。

2.2 使用参数

函数的一个重要部分就是访问参数，但由于 extension 的特殊性，我们无法像通常的函数那样来访问参数。

先来看导出 C 函数的原型：

```
void zif_first_module (
        int ht,
        zval * return_value,
        zval **return_value_ptr,
        zval * this_ptr,
        int return_value_used
        );
```

ht 是用户传入参数的数目，但一般不应直接读取，而是通过宏 ZEND_NUM_ARGS()来获取，这通常用于判断用户是否传入了规定数目的参数。下面介绍如何在我们的 C 函数中访问这些参数。

2.2.1 标准方法

常用的方法是使用下面这个函数，其使用方法类似于 scanf，采用格式化字符串和变长参数列表的方式：

```
int zend_parse_parameters(int num_args TSRMLS_DC, char *type_spec, ...);
```

num_args 指出我希望获取的参数数目，通常使用 ZEND_NUM_ARGS()，因为我们一般会先用 ZEND_NUM_ARGS() 判断用户是否传入了规定数目的参数。TSRMLS_DC 宏用于线程安全，define 和 declare 时必须这样填写，在调用时应该改用 TSRMLS_CC。

type_spec 是格式化字符串，其每个字符代表期望的当前参数的类型，之后应传递相应类型变量的指针来接收值，就像 scanf 那样，可用的字符如下：

格式字符	PHP 参数类型	接收变量类型
l	long	long
d	double	double
s	string	char*和 int
b	boolean	zend_bool
r	resource	zval*
a	array	zval*
z	zval	zval*
o/O/C	类，不予讨论	N/A

这里面，string 是个特例，它需要两个参数，分别获取字符串指

针和长度，这是因为 PHP 没有采用 C 串，不能根据 0 来判断字符串结尾。下面是个示例程序：

```
// 获取一个 long、一个 string 和一个 resource
long l;
char *s;        // 字符串地址
int s_len;      // 字符串长度
zval *res;

// 检查参数数目
if(ZEND_NUM_ARGS() != 3)
    WRONG_PARAM_COUNT; // 该宏输出相应错误信息并退出当前函数

if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
                        "lsr", &l, &s, &s_len, &res) == FAILURE)
     return;
```

由于 PHP 语法不能规定函数原型，因此用户可以传递任意类型的参数，对此，zend_parse_parameters 自动进行了类型检查和转换：在内置标量类型，即 long、double、boolean 和 string 之间，Zend 会自动进行类型转换，我们总能成功取得参数；resource 和 array 则不进行转换，用户传入的参数必须具有指定类型，否则返回错误；zval 作为通用结构，可以用于任何参数类型，Zend 只需要简单的将其写入本地的接收变量。

除了类型格式符外，该函数还支持另外 3 个控制符：

格式字符	意义
|	后面的参数是可选的，如果用户没有传递相应的参数，则本地接收变量保持不变，这用于支持默认参数；
!	前面的那个参数可以是 NULL，仅用于 razoOC，如果用户传递的是 NULL，则本地的接收 zval*被设为 NULL；

/	如果前面那个参数不是引用传递的，则不直接使用传入的 zval，而是执行 Copy-On-Write。这一点将在后面解释。

最后，关于参数的数目也是有要求的。如果没有采用默认参数，即'|'格式符，则 ZEND_NUM_ARGS()、num_args 和格式串指出的参数数目这三者间必须完全匹配，否则 zend_parse_parameters 返回错误；如果使用了默认参数，则 ZEND_NUM_ARGS()应和 num_args 相等，并且应该落在格式串指出的参数数目区间内。

2.2.2 底层方法

大部分情况下，使用标准方法就可以了，但有些函数可能需要处理变参，标准方法对此无能为力(*)。此时，只有使用更加原始的方法——直接获取 zval。Zend 提供了如下的 API：

```
int zend_get_parameters_array_ex(
        int param_count,
        zval ***argument_array
        TSRMLS_DC);
```

param_count 是希望获取的参数数目，这个值不得大于 ZEND_NUM_ARGS()，否则函数出错。argument_array 是一个 zval** 类型的数组，用于接收参数。

这个函数只是简单的返回 zval，为了使用它们，我们需要自己访问其成员。首先是获取参数类型，这可以通过 zval.type 值来判断，可用的 type 见 1.1.1 节。之后是获取该 type 对应的值，我们可以直

接访问 zval 的成员，比如 zval.value.lval 就是 long 值，但更方便的方法是使用 Zend 提供的宏：

宏	展开
Z_LVAL(zval)	(zval).value.lval
Z_DVAL(zval)	(zval).value.dval
Z_STRVAL(zval)	(zval).value.str.val
Z_STRLEN(zval)	(zval).value.str.len
Z_ARRVAL(zval)	(zval).value.ht
Z_RESVAL(zval)	(zval).value.lval
Z_OBJVAL(zval)	(zval).value.obj
Z_BVAL (zval)	((zend_bool)(zval).value.lval)
Z_TYPE(zval)	(zval).type

一个比较特殊的宏是 Z_BVAL，它不是简单的返回值，而是进行了类型转换。另外，这些宏都有相应的 xxx_P 和 xxx_PP 版本，用于访问 zval*和 zval**。

有时，用户传入参数的类型并不是我们期望的，这就需要手动进行类型转换了。为此，Zend 提供了如下几个函数：

convert_to_boolean_ex()
convert_to_long_ex()
convert_to_double_ex()
convert_to_string_ex()
convert_to_array_ex()
convert_to_object_ex()
convert_to_null_ex()

这些函数可将目标 zval 转换成指定类型，它接收 zval**作为参数，为什么不用 zval*呢？这是因为，这些函数有一个额外的步骤，它如果发现传入的 zval 不是引用类型的，并且需要执行类型转换，则会首先执行 Copy-On-Write，并对副本施行转换，因此，为了返回副本必须使用 zval**作为参数。如果 zval 是引用型的，则转换直接作用于目标 zval 结构。

如果无法转换，这些函数就会将 zval 设置为目标类型的虚值，比如 0、FALSE、空串等，因此函数总会成功返回。

这些函数的非 ex 版本不执行 zval 分离，而是直接作用于原 zval，因此参数类型是 zval*。

2.2.3 引用传递

在 1.1.5 节提到，函数参数的传递也是采用的引用计数方式，函数栈中存放的只是 zval**，它很可能和几个变量共享一个 zval。

显然，对于引用型的 zval，我们可以直接进行写入操作；而对于非引用型的 zval，并且其 refcount 大于 1 时，如果要进行写入操作，就必须执行 zval 分离（参见 1.1.3）。refcount 等于 1 的情况是因为 Zend 引擎已经执行了 zval 状态切换（参见 1.1.4 情况 II），我们得到的是自己独占的 zval，可以直接写入。

关于传入的 zval 是否引用，可以通过 zval.is_ref 来判断，或者使用宏 PZVAL_IS_REF(zval*)。对于 zval 分离，可以使用宏 SEPARATE_ZVAL(zval**)，它会自动判断 refcount，并且将新 zval

的地址填充到参数里。

2.2.4 编译检查(TODO)

上面几节介绍了如何在我们的函数中对参数进行检查，也就是运行时检查，这为函数的编写带来了一些负担，代码也不够简洁。为此，Zend 提供了编译时检查机制，允许我们指定函数原型，如果用户不按规定调用，则会报错并且跳过该函数，因此，我们的函数总能得到期望的参数。

2.3 返回值

从 C 函数向 PHP 返回值，并不能使用通常的 return 语句，导出函数的原型也说明了这一点：

```
void zif_first_module (
        int ht,
        zval * return_value,
        zval **return_value_ptr,
        zval * this_ptr,
        int return_value_used
        );
```

因此，Zend 将返回值地址作为参数传给我们，return_value 是 Zend 为我们预先创建的一个标准 zval 结构，相当于一个局部变量，用户获得返回值时就相当于对 return_value 进行赋值操作，我们只需填充它即可；return_value_used 表明用户是否使用了返回值，0 表明没有使用返回值，当函数结束后 return_value 的 refcount 将被减为 0，

并被销毁，因此，这种情况下完全可以不处理返回值；return_value_ptr 用于返回引用，它需要和 zend_function_entry.arg_info 联合使用，通常都是 NULL。

Zend 提供了一组宏用于填充 return_value：

Macro	Description
RETURN_RESOURCE(resource)	resource
RETURN_BOOL(bool)	boolean
RETURN_FALSE	false
RETURN_TRUE	true
RETURN_NULL()	NULL
RETURN_LONG(long)	long
RETURN_DOUBLE(double)	double
RETURN_STRING(string, duplicate)	字符串。string 必须是 C 串，因为 Zend 将调用 strlen()；duplicate 表示是否将传入的 C 串复制一份再赋给 zval，如果传入的 C 串不是用 Zend 例程分配的，应该指定该值
RETURN_STRINGL(string, length, duplicate)	指定字符串长度，而不是使用 strlen()
RETURN_EMPTY_STRING()	空字符串

这些宏将在填充完 return_value 后，执行 return 语句。如果不想 return，可以改用相应 RETURN_xxx 宏的 RETVAL_xxx 版本。

2.3.1 返回引用

默认情况下，return_value_ptr 是 NULL，而当指定返回引用后（参见 2.2.4），zend 将采用*return_value_ptr 作为返回值。初始状态下，return_value 依然指向一个临时 zval，同时 *return_value_ptr = return_value。

通常应该把 return_value 销毁，并且将*return_value_ptr 设为将要返回的 zval*，注意要加加引用计数，因为这相当于将该 zval 赋值给一个用作返回值的临时变量，函数返回后，Zend 会减减引用计数。

示例程序：

```
ZEND_FUNCTION(str_reverse)
{
    if(ZEND_NUM_ARGS()!= 1)
        WRONG_PARAM_COUNT;
    zval **args;
    if(zend_get_parameters_array_ex(ZEND_NUM_ARGS(), &args TSRMLS_CC)
      == FAILURE)
    {
        return;
    }
    convert_to_string(*args);
    char swap;
    char *head = Z_STRVAL_PP(args);
    char *end = head + Z_STRLEN_PP(args) - 1;
    for(; head < end; ++head, --end)
    {
        swap = *end;
        *end = *head;
        *head = swap;
    }
    // 销毁临时 zval
    zval_ptr_dtor(return_value_ptr);
    // 返回传入的参数
    *return_value_ptr = *args;
    // 增加引用计数
    ++(*return_value_ptr)->refcount;
}
```

2.4 启动和终止函数

Zend 允许模块在加载和卸载时收到通知，以进行初始化和清除工作，我们要做的就是把相应函数传递给 Zend，它会在合适的时机自动调用。2.1.3 节里留下的五个 NULL 就是用于这个目的，它们都是函数指针，最后一个用于配合 phpinfo()来显示模块信息，在此忽略，只看其他四个。

Zend 提供了如下四个宏，分别用于声明对应的函数：

宏	意义
ZEND_MODULE_STARTUP_D(module)	在加载模块时调用
ZEND_MODULE_SHUTDOWN_D(module)	在卸载模块时调用
ZEND_MODULE_ACTIVATE_D(module)	一个页面开始运行时调用
ZEND_MODULE_DEACTIVATE_D(module)	一个页面运行完毕时调用

这些宏的用法和 ZEND_FUNCTION 宏一样（参见 2.1.1），展开后就是声明了特定原型的函数，其参数 module 可以是任意的，但最好使用模块名称。这些函数的参数中，对我们有用的是 int module_number，它是模块号，全局唯一，后面会提到其用处。

在声明和实现相应函数时，都应该使用这些宏。最后，需要把这些函数填写到 zend_module_entry 里（参见 2.1.3），可按顺序使用如下的宏，这些宏生成相应的函数名称：

ZEND_MODULE_STARTUP_N(module)
ZEND_MODULE_SHUTDOWN_N(module)
ZEND_MODULE_ACTIVATE_N(module)
ZEND_MODULE_DEACTIVATE_N(module)

2.5 调用 PHP 函数

有时我们需要在模块中调用用户指定的函数，比如我们实现了 sort 这样的函数，并且允许用户指定比较函数。这可以使用如下的 Zend 函数：

```
int call_user_function_ex(
        HashTable *function_table,
        zval **object_pp,
        zval *function_name,
        zval **retval_ptr_ptr,
        zend_uint param_count,
        zval **params[],
        int no_separation,
        HashTable *symbol_table
        TSRMLS_DC)
```

第一个参数是 HashTable，在 1.2.3 节提到 Zend 使用 HashTable 来存储 PHP 函数，function_table 用于指定从哪个 HashTable 中获取函数。通常应该用 CG(function_table)，展开就是 compiler_globals.function_table，compiler_globals 是一个用来存储编译器数据的全局数据结构（与其对应的还有个 EG 宏，即 executor_globals，它用来存储执行器数据）。compiler_globals.function_table 里面存储了所有我们可以在 PHP 页面里面调用的函数，包括 Zend 内建函数、PHP 标准库函数、模块导出的函数以及用户使用 PHP 代码定义的函数。

object_pp 是一个对象，当指定该值时，Zend 会从对象的函数表

中获取函数，这里不予讨论，总是设为 NULL。

function_name 必须是 string 型的 zval，存储我们希望调用的函数的名称。为什么使用 zval 而不是直接用 char*，是因为 Zend 考虑到大部分情况下，我们都是从用户那获得参数，然后再调用 call_user_function_ex 的，这样就可以不作处理直接把用户参数传给该函数。当然，我们也可以手动创建一个 string 型 zval 传给它。

retval_ptr_ptr 用于获取函数的返回值，Zend 执行完指定的函数后，它就将返回值的指针填充到这里。

param_count 和 params 用于指定函数的参数，params 是个 zval ** 这点可能让人感到奇怪，但考虑到该函数的常见用法（见下面的示例）以及 2.2.2 节关于函数参数的介绍，就一点也不奇怪了。

no_separation 用于指定是否在必要时执行 zval 分离（参见 1.1.3），这在写入非引用 zval 时发生。应该总是将其设为 0，表示执行 zval 分离，否则可能破坏数据。

symbol_table 用于指定目标函数的 active_symbol_table（参见 1.2.3），通常应该使用 NULL，这样 Zend 会为目标函数生成一个空的符号表。

说了这么多，该动动手了，下面的程序片段简单实现了 PHP API call_user_func 的功能：

```
ZEND_FUNCTION(call)
{
    int num_args = ZEND_NUM_ARGS();
    if(num_args < 1)
        WRONG_PARAM_COUNT;
    zval ***args = (zval***)emalloc(sizeof(zval**)*num_args);
    zval *ret_zval;
    // 获取传入的参数
    if(zend_get_parameters_array_ex(num_args, args TSRMLS_CC)
         == FAILURE)
    {
        efree(args);
        return;
    }
    // 第一个参数作为函数名，后面的作为函数参数
    if(call_user_function_ex(CG(function_table), NULL, **args,
         &ret_zval, num_args - 1, args + 1, 0, NULL TSRMLS_CC)
         == FAILURE)
    {
        efree(args);
        zend_error(E_ERROR, "Function call failed");
    }
    // 将函数返回值反馈给用户
    *return_value = *ret_zval;
    efree(args);
}
```

2.6 访问 PHP 变量

2.6.1 设置

1.2.3 节提到 Zend 使用 HashTable 来存储全局和局部变量符号，因此访问 PHP 变量，其实就是操作 HashTable。当然，我们不需要手工去做，Zend 提供了一组宏完成这些工作。

PHP 变量的创建共有三步，首先需要创建一个 zval 结构，可使用如下的宏：

```
MAKE_STD_ZVAL(zval*)
```

这个宏先调用 emalloc 分配一块 zval，然后将其 refcount 设为 1、is_ref 设为 0。

之后就是设置 zval 的值，同样，我们不需要直接操作 zval 的成员，Zend 已经提供了如下的宏：

Macro	Description
ZVAL_RESOURCE(zval*, resource)	resource
ZVAL_BOOL(zval*, bool)	boolean
ZVAL_FALSE(zval*)	false
ZVAL_TRUE(zval*)	true
ZVAL_NULL(zval*)	NULL
ZVAL_LONG(zval*, long)	long
ZVAL_DOUBLE(zval*, double)	double
ZVAL_STRING(zval*, string, duplicate)	string 必须是 C 串，因为 Zend 将调用 strlen()；duplicate 表示是否将传入的 C 串复制一份再赋给 zval，如果传入的 C 串不是用 Zend 例程分配的，应该指定该值
ZVAL_STRINGL(zval*, string, length, duplicate)	指定字符串长度，而不是使用 strlen()
ZVAL_EMPTY_STRING(zval*)	空字符串

可能你会发现，这个表格和 2.3 节里面的返回值宏表格很相似，不错，返回值宏就是直接调用的 ZVAL_xxx。

既然有了 zval，下面把它添加到变量符号表里就可以了，可以使用如下的一组宏：

```
ZEND_SET_SYMBOL(symtable, name, var)
ZEND_SET_GLOBAL_VAR(name, var)
```

symtable 用来指定你想插入的符号表，一般使用 EG(active_symbol_table)，表示访问当前调用者的活动符号表。如果想强制访问全局符号表，可以用&EG(symbol_table)，这也正是 ZEND_SET_GLOBAL_VAR(name, var)所做的。这两个宏的最终效果和执行 PHP 赋值语句 name = var 完全一样。

如果只是访问全局变量，可以使用单个宏代替上述三步：

```
SET_VAR_STRING(name, value)
SET_VAR_STRINGL(name, value, length)
SET_VAR_LONG(name, value)
SET_VAR_DOUBLE(name, value)
```

上述宏分别用于创建全局的 string、long 和 double 变量，它们在内部执行了以上三步，当然，最后调用的是 ZEND_SET_GLOBAL_VAR 宏。

2.6.2 获取

如果想获取已有的 PHP 变量，则只能直接访问 HashTable，Zend

并没有提供相应的操作：

```
int zend_hash_find(
        HashTable *ht,
        char *arKey, uint nKeyLength,
        void **pData)
```

这个函数从 HashTable 中查找元素，pData 用于获取结果值，Bucket.pData 将被放到这里（如果找到的话）。函数成功则返回 SUCCESS，否则返回 FAILURE。

下面是个示例：

```
zval **ppzval; // Bucket.pData 里存放的是 zval**
if(zend_hash_find(EG(active_symbol_table),"var", 4,
(void**)&ppzval) == SUCCESS)
    printf("var.refcount = %d\n", (*p)->refcount);
else
    printf("Not Found\n");
```

这段代码从活动符号表中查找名为 var 的变量，需要注意的是 nKeyLength 是 4，必须包括结尾的 0。

获得变量后，拿来读是没有问题的，但是写操作就应该小心对待了。只有当 refcount 为 1 或者 is_ref 为 1，才可以写入；否则应该进行 zval 分离，具体参见 2.2.3 节。

2.6.3 常量

PHP 常量的内部定义如下：

```
typedef struct _zend_constant {
        zval value;
        int flags;
        char *name;
        uint name_len;
        int module_number;
} zend_constant;
```

常量的值依然使用 zval 存储，但这里的 zval 是私有的，不会和其他变量或常量共享，其 refcount 和 is_ref 被忽略。module_number 是模块号，在启动函数中可以获取该值（参见 2.4），当模块被卸载时，Zend 会使用模块号查找和删除所有该模块注册的常量。如果希望在模块被卸载后，常量依然有效，可以将 module_number 设为 0。另一个注意点是，name_len 需要包含结尾的 0。

flags 值可以是如下两个，可以使用”|”联用：

flag	意义
CONST_CS	常量名大小写敏感
CONST_PERSISTENT	持久常量，在创建常量的页面执行结束后，常量依然有效(*)

所有常量都被放在 EG(zend_constants)这张 HashTable 里，其 key 是常量名称，value 是 zend_constant，注意不是 zend_constant*，因此 HashTable 会复制一份 zend_constant 作为 value。

获取一个常量非常简单，只要传递常量名和接受常量值的 zval：

```
int zend_get_constant(char *name, uint name_len, zval *result
        TSRMLS_DC);
```

设置常量稍微复杂一点，需要先填写一个 zend_constant 结构，要注意的是，常量只能是 long、double 和 string。然后使用如下函数

将其加入常量表：

```
int zend_register_constant(zend_constant *c TSRMLS_DC);
```

同时，Zend 也为我们提供了如下的宏，可以直接创建常量：

REGISTER_LONG_CONSTANT(name, value, flags) REGISTER_MAIN_LONG_CONSTANT(name, value, flags)
REGISTER_DOUBLE_CONSTANT(name, value, flags) REGISTER_MAIN_DOUBLE_CONSTANT(name, value, flags)
REGISTER_STRING_CONSTANT(name, value, flags) REGISTER_MAIN_STRING_CONSTANT(name, value, flags)
REGISTER_STRINGL_CONSTANT(name, value, length, flags) REGISTER_MAIN_STRINGL_CONSTANT(name, value, length, flags)

上述宏的 MAIN 版本用于创建 module_number 为 0 的宏，在模块被卸载后，常量依然有效。而非 MAIN 版本则假设存在一个名为 module_number 的 int 变量，并拿来给 zend_constant.module_number 赋值，可见这组宏原本就是为在模块启动函数里调用而设计的。另外，当创建 string 型常量时，Zend 也会 dup 一份字符串，因此可以直接使用 C 串指定常量值。

最后需要指出的是，上述函数和宏都无法改变已有的常量，如果发现已经存在同名常量，则函数失败。如果想修改的话，只能通过 HashTable 操作。

2.7 输出信息

Zend 提供了两个函数用于向浏览器输出信息：

```
int zend_printf(const char *format, ...);
void zend_error(int type, const char *format, ...);
```

zend_printf 用法和 C 的 printf 一样；zend_error 用于输出错误信息，type 可以指定错误的性质，对于不同的错误，Zend 将作不同处理：

错误码	处理
E_ERROR	严重错误，立即终止脚本运行。
E_WARNING	警告， 脚本继续执行。
E_PARSE	解析错误，解析器复位，脚本继续执行。
E_NOTICE	通知，脚本继续执行。该信息默认情况下不予输出，可以修改 php.ini 来启用。

该函数会同时输出出错的文件和行号，类似这样：

```
Fatal error: no memory in /home/wiki/zdj/ext/test.php on line 6
```

三、 高级主题

3.1 使用数组

1.2.2 节曾讲到，PHP 数组本质上就是个 HashTable，因此访问数组就是对 HashTable 进行操作，Zend 为我们提供的一组数组函数也只是对 HashTable 操作进行了简单包装而已。

来看创建数组，由于数组也是存在于 zval 里的，因此要先用 MAKE_STD_ZVAL()宏（参见 2.6.1）创建一个 zval，之后调用如下宏将其转化为一个空数组：

```
array_init(zval*)
```

接下来是朝数组中添加元素，这对关联数组元素和非关联数组元素要采用不同操作。

3.1.1 关联数组元素

关联数组采用 char*作为 key，zval*作为 value，可以使用如下宏将已有的 zval 加入数组或者更新已有元素：

```
int add_assoc_zval(zval *arr, char *key, zval *value)
```

需要特别注意的是，Zend 不会复制 zval，只会简单的储存其指针，并且不关心任何引用计数，因此不能将其他变量的 zval 或者是栈上的 zval 传给它，只能用 MAKE_STD_ZVAL()宏构建。

Zend 为常用的类型定义了相应的 API，以简化我们的操作：

add_assoc_long(zval *array, char *key, long n);
add_assoc_bool(zval *array, char *key, int b);
add_assoc_resource(zval *array, char *key, int r);
add_assoc_double(zval *array, char *key, double d);
add_assoc_string(zval *array, char *key, char *str, int duplicate);
add_assoc_stringl(zval *array, char *key, char *str, uint length, int duplicate);
add_assoc_null(zval *array, char *key);

当函数发现目标元素已经存在时，会首先递减其原 zval 的 refcount，然后才插入新 zval，这就保证了原 zval 引用信息的正确性。这种行为是通过 HashTable.pDestructor（参见 1.2.1）实现的，每次删除一个元素时，HashTable 都将对被删元素调用这个函数指针，而数组为其 HashTable 设置的函数指针就是用来处理被删除 zval 的引用信息。

另外，查看这些函数的源代码可以发现一个有意思的现象，它们没有直接使用 HashTable 操作，而是使用变量符号表操作，可见关联数组和变量符号表就是一种东西。

Zend 没有提供删除和获取数组元素的函数，此类操作只能使用 HashTable 函数或者是 2.6 节的变量符号表操作。

3.1.2 非关联数组元素

非关联数组没有 key，使用 index 作为 hash，相应函数和上面关联数组的十分类似：

add_index_zval(zval *array, uint idx, zval *value);
add_index_long(zval *array, uint idx, long n);
add_index_bool(zval *array, uint idx, int b);
add_index_resource(zval *array, uint idx, int r);
add_index_double(zval *array, uint idx, double d);
add_index_string(zval *array, uint idx, char *str, int duplicate);
add_index_stringl(zval *array, uint idx, char *str, uint length, int duplicate);
add_index_null(zval *array, uint idx);

如果只是想插入值，而不指定 index 的话，可以使用如下函数：

add_next_index_zval(zval *array, zval *value);
add_next_index_long(zval *array, long n);
add_next_index_bool(zval *array, int b);
add_next_index_resource(zval *array, int r);
add_next_index_double(zval *array, double d);
add_next_index_string(zval *array, char *str, int duplicate);
add_next_index_stringl(zval *array, char *str, uint length, int duplicate);
add_next_index_null(zval *array);

3.2 使用资源

3.2.1 注册资源类型

1.1.1 节曾经提到，所谓资源就是内部数据的 handle（但是这句话并不全对），使用资源是比较简单的，首先是注册一个资源类型：

```
int zend_register_list_destructors_ex(
        rsrc_dtor_func_t ld,
        rsrc_dtor_func_t pld,
        char *type_name,
        int module_number);
```

第一个参数是函数指针，当资源不再被使用或者模块将被卸载时，Zend 使用它来销毁资源，稍候再作介绍；第二个参数和第一个类似，只是它被用来销毁持久性资源(*)；type_name 是资源名称，用户可以使用 var_dump 函数来读取；module_number 是模块号，在启动函数中可以获取该值。

注册过程其实就是将我们传入的参数放到一个内部数据结构，然后把这个数据结构放入一个没有使用 key 的 HashTable 里，该函数返回的值，也就是所谓“资源类型 id”，其实就是 HashTable 的 index。

3.2.2 注册资源

注册完资源类型后，就可以注册一个该类型的资源了：

```
ZEND_REGISTER_RESOURCE(
        rsrc_result,
        rsrc_pointer,
        rsrc_type)
```

rsrc_pointer 是个指针类型，就是你的资源的 handle，通常是指向内部数据的指针，当然也可以是 index 或者其它标志符；rsrc_type 是上面获取的资源类型 id；rsrc_result 是个已有的 zval，注册完成后，资源的 id 就被放入该 zval，同时其 type 也被设为 IS_RESOURCE，

通常是传入 return_value，以将资源返回给用户。

在内部，Zend 使用如下数据结构表示一个资源：

```
typedef struct _zend_rsrc_list_entry {
    void *ptr;
    int type;
    int refcount;
} zend_rsrc_list_entry;
```

ptr 和 type 就是我们在上面传入的参数；refcount 是引用计数，由 Zend 维护，当引用减到 0 时，Zend 会销毁该资源。不出所料的是，这个数据结构也被组织在一个 HashTable 里，并且没有使用 key，仅仅使用 index——这就是 zval 里存放的东西。现在资源的整个脉络已经清晰：通过 zval 可以获得资源 id，通过资源 id 可以获得资源 handle 和资源类型 id，通过资源类型 id 可以获得资源的销毁函数。

现在讲一下销毁函数：

```
typedef void (*rsrc_dtor_func_t)(
    zend_rsrc_list_entry *rsrc
    TSRMLS_DC);
```

rsrc 是需要被销毁的资源，我们在函数的实现中可以通过它获得资源的 handle，并且加以处理，比如释放内存块、关闭数据库连接或是关闭文件描述符等。

3.2.3 获取资源

当创建了资源后，用户通常都要调用创建者提供的函数来操作资源，此时我们需要从用户传入的 zval 中取出资源：

```
ZEND_FETCH_RESOURCE(
    rsrc,    rsrc_type,
    passed_id, default_id,
    resource_type_name, resource_type)
```

首个参数用于接收 handle 值，第二个参数是 handle 值的类型，这个函数会扩展成“rsrc = (rsrc_type) zend_fetch_resource(…)”，因此应该保证 rsrc 是 rsrc_type 类型的；passed_id 是用户传入的 zval，这里使用 zval**类型，函数从中取得资源 id；default_id 用来直接指定资源 id，如果该值不是-1，则使用它，并且忽略 passed_id，所以通常应该使用-1；resource_type_name 是资源名称，当获取资源失败时，函数使用它来输出错误信息；resource_type 是资源类型，如果取得的资源不是该类型的，则函数返回 NULL，这用于防止用户传入一个其他类型资源的 zval。

不过，这个宏确实比较难用，用其底层的宏反倒更加容易些：

```
zend_list_find(id, type)
```

id 是要查找的资源 id；type 是 int*类型，用于接收取出的资源的类型，可以用它来判断这是不是我们想要的资源；函数最后返回资源的 handle，失败返回 NULL。

3.2.4 维护引用计数

通常，当用户对资源类型的 PHP 变量执行赋值或是 unset 之类操作时，Zend 会自动维护资源的引用计数。但有时，我们也需要手

动进行，比如我们要复用一个数据库连接或者用户调用我们提供的 close 操作关闭一个文件，此时可以使用如下宏：

```
zend_list_addref(id)
zend_list_delete(id)
```

id 是资源 id，这两个宏分别增加和减少目标资源的引用计数，第二个宏还会在引用计数减到 0 时，调用先前注册的函数销毁资源。

四、 类和对象(TODO)

附录A. Extension 的编译

Extension 的编译是比较简单的，下面是个示例 Makefile：

```
# 扩展搜索目录，模块被放到这里才能被找到和加载
# 可以从 php.ini 中的 extension_dir 命令获取该值
PHP_EXT_HOME=/home/wiki/php5/lib/php/extensions/
# PHP 源代码路径
PHP_SRC=/home/wiki/wikienv/install/php-5.2.3
INCLUDE=-I$(PHP_SRC) -I$(PHP_SRC)/main -I$(PHP_SRC)/TSRM -I$(PHP_SRC)/Zend
CC=gcc

all: first_module.so

first_module.so: first_module.o
    $(CC) -shared -rdynamic -o first_module.so first_module.o

# 注意不要忘了“-DCOMPILE_DL_FIRST_MODULE=1”，否则不会导出 get_module()
first_module.o: first_module.c
    $(CC) -fpic -DCOMPILE_DL_FIRST_MODULE=1 $(INCLUDE) -c first_module.c

clean:
    rm -fr *.so *.o

install: first_module.so
    cp -fp first_module.so $(PHP_EXT_HOME)
```

这将创建一个可动态加载的 Extension 模块，如果想将 Extension 静态编译进 PHP，就需要使用 PHP 本身的编译系统，这里不作介绍。

另外，Zend 提供了一个小脚本来简化创建和编译 Extension 的过程，可以在《PHP 手册》的相关章节找到其使用方法。

目前，Zend 正在开发一个更加完善和标准的 Extension 生成工具，可以在这里找到它：http://pecl.php.net/package/PECL_Gen

附录B. Extension 的加载过程

Extension 有两种加载方式，通过 php.ini 在启动时加载或是通过 dl()函数在运行时加载，前者需在 php.ini 中添加一行命令：extension=myext.so。这两种方式最后都将调用一个名为 php_ld()的内部函数。

php_ld()首先调用系统例程加载动态库，之后解析并执行 get_module()函数来获得模块的 zend_module_entry 结构（参见 2.1.3）。

然后，调用函数 zend_register_module_ex()来注册模块，这个函数先将模块的 zend_module_entry 加入到一个名为 module_registry 的 HashTable 中，然后调用 zend_register_functions 将模块导出的函数加入 CG(function_table)（参见 2.5）这个 HashTable 中，这样我们就可以调用模块导出的函数了。最后，php_ld()将调用模块的启动函数（参见 2.4）来初始化模块。

如果模块是 dl()动态加载的，Zend 还将接着调用模块的激活函数，因为当前显然有页面在运行。

最后要说明的是，通过 dl()加载的模块称为“临时模块”，在页面执行完毕时将被卸载，同时，其创建的常量和资源也将被销毁。