编写扩展 I：PHP和Zend起步

扩展　教程

by Sara Golemon | Monday, February 28, 2005

介绍

既然您正在阅读本教程，那么您或许对编写PHP语言的扩展感兴趣。如果不是...呃，或许你并不知道这一兴趣，那么我们结束的时候你就会发现它。

本教程假定您基本熟悉PHP语言及其解释器实现所用的语言：**C**.

让我们从指明为什么你想要编写PHP扩展开始。

1. 限于PHP语言本身的抽象程度，它不能直接访问某些库或特定于操作系统的调用。
2. 你想要通过某些不平常的方法定制PHP的行为。
3. 你有一些现成的PHP代码，但是你知道它可以（运行）更快、（占用空间）更小，而且消耗更少的内存。
4. 你有一些不错的代码出售，买家可以使用它，但重要的是不能看到源代码。

这些都是非常正当的理由，但是，在创建扩展之前，你需要首先明白扩展是什么?

扩展是什么?

如果你用过PHP，那么你肯定用到过扩展。除了少数例外，每个用户空间的函数都被组织在不同的扩展中。这些函数中的很多够成了*standard*扩展－总数超过400。PHP本身带有86个扩展（原文写就之时－译注），平均每个含有大约30个函数。数学操作方面大约有2500个函数。似乎这还不够， PECL仓库另外提供了超过100个扩展，而且互联网上可以找到更多。

“除了扩展中的函数，还有什么？”我听到了你的疑问。 “扩展的里面是什么？PHP的‘核心’是什么？”

PHP的核心由两部分组成。最底层是**Zend**引擎（**ZE**）。ZE把人类易读的脚本解析成机器可读的符号，然后在进程空间内执行这些符号。ZE也处理内存管理、变量作用域及调度程序调用。另一部分是**PHP**内核，它绑定了**SAPI**层（Server Application Programming Interface，通常涉及主机环境，如Apache，IIS，CLI，CGI等），并处理与它的通信。它同时对`safe_mode`和`open_basedir`的检测提供一致的控制层，就像流层将`fopen()`、`fread()`和`fwrite()`等用户空间的函数与文件和网络I/O联系起来一样。

生存周期

当给定的SAPI启动时，例如在对`/usr/local/apache/bin/apachectl start`的响应中，PHP由初始化其内核子系统开始。在接近启动例程的末尾，它加载每个扩展的代码并调用其模块初始化例程（`MINIT`）。这使得每个扩展可以初始化内部变量、分配资源、注册资源处理器，以及向ZE注册自己的函数，以便于脚本调用这其中的函数时候ZE知道执行哪些代码。

接下来，PHP等待SAPI层请求要处理的页面。对于CGI或CLI等SAPI，这将立刻发生且只发生一次。对于Apache、IIS或其他成熟的web服务器SAPI，每次远程用户请求页面时都将发生，因此重复很多次，也可能并发。不管请求如何产生，PHP开始于要求ZE建立脚本的运行环境，然后调用每个扩展的请求初始化（`RINIT`）函数。`RINIT`使得扩展有机会设定特定的环境变量，根据请求分配资源，或者执行其他任务，如审核。 session扩展中有个`RINIT`作用的典型示例，如果启用了`session.auto_start`选项，`RINIT`将自动触发用户空间的`session_start()`函数以及预组装`$_SESSION`变量。

一旦请求被初始化了，ZE开始接管控制权，将PHP脚本翻译成符号，最终形成操作码并逐步运行之。如任一操作码需要调用扩展的函数，ZE将会把参数绑定到该函数，并且临时交出控制权直到函数运行结束。

脚本运行结束后，PHP调用每个扩展的请求关闭（`RSHUTDOWN`）函数以执行最后的清理工作（如将session变量存入磁盘）。接下来，ZE执行清理过程（垃圾收集）—有效地对之前的请求期间用到的每个变量执行`unset()`。

一旦完成，PHP继续等待SAPI的其他文档请求或者是关闭信号。对于CGI和CLI等SAPI，没有“下一个请求”，所以SAPI立刻开始关闭。关闭期间，PHP再次遍历每个扩展，调用其模块关闭（`MSHUTDOWN`）函数，并最终关闭自己的内核子系统。

这个过程乍听起来很让人气馁，但是一旦你深入一个运转的扩展，你会逐渐开始了解它。

内存分配

为了避免写的不好的扩展丢失内存，ZE使用附加的标志来执行自己内部的内存管理器以标识持久性。持久分配的内存意味着比单次请求更持久。对比之下，对于在请求期间的非持久分配，不论是否调用释放（内存）函数，都将在请求尾期被释放。例如，用户空间的变量被分配为非持久的，因为请求结束后它们就没用了。

然而，理论上，扩展可以依赖ZE在页面请求结束时自动释放非持久内存，但是不推荐这样做。因为分配的内存将在很长时间保持为未回收状态，与之相关联的资源可能得不到适当的关闭，并且吃饭不擦嘴是坏习惯。稍后你会发现，事实上确保所有分配的数据都被正确清理很容易。

让我们简单地比较传统的内存分配函数（只应当在外部库中使用）和PHP/ZE的持久的以及非持久的内存非配函数。

传统的	非持久的	持久的
`malloc(count)`	`emalloc(count)`	`pemalloc(count, 1)`*
`calloc(count, num)`	`ecalloc(count, num)`	`pecalloc(count, num, 1)`
`strdup(str)`	`estrdup(str)`	`pestrdup(str, 1)`
`strndup(str, len)`	`estrndup(str, len)`	`pemalloc() & memcpy()`
`free(ptr)`	`efree(ptr)`	`pefree(ptr, 1)`
`realloc(ptr, newsize)`	`erealloc(ptr, newsize)`	`perealloc(ptr, newsize, 1)`
`malloc(count * num + extr)`**	`safe_emalloc(count, num, extr)`	`safe_pemalloc(count, num, extr)`

* *`pemalloc()`*族包含一个‘持久’标志以允许它们实现对应的非持久函数的功能。
例如：*`emalloc(1234)`*与*`pemalloc(1234, 0)`*相同。
** *`safe_emalloc()`*和（*PHP 5*中的）*`safe_pemalloc()`*执行附加检测以防整数溢出。

建立构建环境

既然你已经了解了一些PHP和Zend引擎的内部运行理论，我打赌你希望继续深入并开始构建一些东西。在此之前，你需要收集一些必需的构建工具并设定适合于你的目的的环境。

首先，你需要PHP本身及其所需要的构建工具集。如果你不熟悉从源码构建PHP，我建议你看看http://www.php.net/install.unix。（为Windows开发PHP扩展将在以后的文章中提到）。然而，使用PHP的二进制分发包有些冒险，这些版本倾向于忽略*./configure*的两个重要选项，它们在开发过程中很便利。第一个`--enable-debug`。这个选项将把附加的符号信息编译进PHP的执行文件，以便如果发生段错误，你能从中得到一个内核转储文件，使用gdb追踪并发现什么地方以及为什么会发生段错误。另一个选项依赖于你的PHP版本。在PHP 4.3中该选项名为`--enable-experimental-zts`，在PHP 5及以后的版本中为`--enable-maintainer-zts`。这个选项使PHP以为自己执行于多线程环境，并且使你能捕获通常的程序错误，然而它们在非多线程环境中是无害的，却使你的扩展不可安全用于多线程环境。一旦你已经使用这些额外的选项编译了PHP并安装于你的开发服务器（或者工作站）中，你就可以把你的第一个扩展加入其中了。

Hello World

什么程序设计的介绍可以完全忽略必需的**Hello World**程序？此例中，你将制作的扩展导出一个简单函数，它返回一个含有“Hello World”的字符串。用PHP的话你或许这样做：

```
<?php

function hello_world() {
return 'Hello World';
}

?>
```

现在你将把它转入PHP扩展。首先，我们在你的PHP源码树的目录*ext/*中创建一个名为*hello*的目录，并且`chdir`进入该目录。事实上，这个目录可以置于PHP源码树之中或之外的任何地方，但是我希望你把它放在这儿，以例示一个在以后的文章中出现的与此无关的概念。你需要在这儿创建3个文件：包含`hello_world`函数的源码文件，包含引用的头文件，PHP用它们加载你的扩展，以及*phpize*用来准备编译你的扩展的配置文件。

config.m4

```
PHP_ARG_ENABLE(hello, whether to enable Hello
World support,

[ --enable-hello   Enable Hello World support])



if test "$PHP_HELLO" = "yes"; then

  AC_DEFINE(HAVE_HELLO, 1, [Whether you have Hello World])

  PHP_NEW_EXTENSION(hello, hello.c, $ext_shared)

fi
```

php_hello.h

```
#ifndef PHP_HELLO_H
#define PHP_HELLO_H 1

#define PHP_HELLO_WORLD_VERSION "1.0"
#define PHP_HELLO_WORLD_EXTNAME "hello"

PHP_FUNCTION(hello_world);

extern zend_module_entry hello_module_entry;
#define phpext_hello_ptr &hello_module_entry

#endif
```

hello.c

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_hello.h"

static function_entry hello_functions[] = {
```

```
    PHP_FE(hello_world, NULL)
    {NULL, NULL, NULL}
};

zend_module_entry hello_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_HELLO_WORLD_EXTNAME,
    hello_functions,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    PHP_HELLO_WORLD_VERSION,
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_HELLO
ZEND_GET_MODULE(hello)
```

```
#endif



PHP_FUNCTION(hello_world)

{

    RETURN_STRING("Hello World", 1);

}
```

在上面的示例扩展中，你所看到的代码大多是黏合剂，作为将扩展引入PHP的协议语言并且在其间建立会话用于通信。只有最后四行才是你所认为“实际做事的代码”，它们负责与用户空间的脚本交互这一层次。这些代码看起来确实非常像之前看到的PHP代码，而且一看就懂：

1. 声明一个名为hello_world的函数
2. 让该函数返回字符串：“Hello World”
3. ....嗯....1？ 那个1是怎么回事儿？

回忆一下，ZE包含一个复杂的内存管理层，它可以确保分配的资源在脚本退出时被释放。然而，在内存管理领域，两次释放同一块内存是绝对禁止的（big no-no）。这种被称为二次释放（**double freeing**）的做法，是引起段错误的一个常见因素，原因是它使调用程序试图访问不再拥有的内存。类似地，你不应该让ZE去释放一个静态字符串缓冲区（如我们的示例扩展中的“Hello World”），因为它存在于程序空间，而不是被任何进程（process）拥有的数据块。RETURN_STRING()可以假定传入其中的任何字符串都需要被复制以便稍后可被安全地释放；但是由于内部的函数给字符串动态地分配内存、填充并返回并不罕见，第二参数RETURN_STRING()允许我们指定是否需要拷贝字符串的副本。要进一步说明这个概念，下面的代码片段与上面的对应版本等效：

```
PHP_FUNCTION(hello_world)

{

    char *str;



    str = estrdup("Hello World");

    RETURN_STRING(str, 0);

}
```

在这个版本中，你手工为最终将被传回调用脚本的字符串“Hello World”分配内存，然后把这快内存“给予”RETURN_STRING()，用第二参数0指出它不需要制作自己的副本，可以拥有我们的。

构建你的扩展

本练习的最后一步是将你的扩展构建为可动态加载的模块。如果你已经正确地拷贝了上面的代码，只需要在*ext/hello/*中运行3个命令：

```
$ phpize
$ ./configure --enable-hello
$ make
```

每个命令都运行后，可在目录*ext/hello/modules/*中看到文件*hello.so*。现在，你可像其他扩展一样把它拷贝到你的扩展目录（默认是*/usr/local/lib/php/extensions/*，检查你的*php.ini*以确认），把`extension=hello.so`加入你的*php.ini*以使PHP启动时加载它。 对于CGI/CLI，下次运行PHP就会生效；对于web服务器SAPI，如Apache，需要重新启动web服务器。我们现在从命令行尝试下：

```
$ php -r 'echo hello_world();'
```

如果一切正常，你会看到这个脚本输出的`Hello World`，因为你的已加载的扩展中的函数`hello_world()`返回这个字符串，而且`echo`命令原样输出传给它的内容（本例中是函数的结果）。

可以同样的方式返回其他标量，整数值用`RETURN_LONG()`，浮点值用 `RETURN_DOUBLE()`，true/false值用`RETURN_BOOL()`，`RETURN_NULL()`？你猜对了，是`NULL`。我们看下它们各自在实例中的应用，通过在文件*hello.c*中的`function_entry`结构中添加对应的几行`PHP_FE()`，并且在文件结尾添加一些`PHP_FUNCTION()`。

```
static function_entry hello_functions[] =
{
    PHP_FE(hello_world, NULL)
    PHP_FE(hello_long, NULL)
    PHP_FE(hello_double, NULL)
    PHP_FE(hello_bool, NULL)
    PHP_FE(hello_null, NULL)
    {NULL, NULL, NULL}
};

PHP_FUNCTION(hello_long)
{
    RETURN_LONG(42);
}
```

```
PHP_FUNCTION(hello_double)
{
    RETURN_DOUBLE(3.1415926535);
}

PHP_FUNCTION(hello_bool)
{
    RETURN_BOOL(1);
}

PHP_FUNCTION(hello_null)
{
    RETURN_NULL();
}
```

你也需要在头文件*php_hello.h*中函数`hello_world()`的原型声明旁边加入这些函数的原型声明，以便构建进程正确进行：

```
PHP_FUNCTION(hello_world);
PHP_FUNCTION(hello_long);
PHP_FUNCTION(hello_double);
PHP_FUNCTION(hello_bool);
PHP_FUNCTION(hello_null);
```

由于你没有改变文件*config.m4*，这次跳过*phpize*和*./configure*步骤直接跳到*make*在技术上是安全的。然而，此时我要你再次做完全部构建步骤，以确保构建良好。另外，你应该调用*make clean all*而不是简单地在最后一步*make*，确保所有源文件被重建。重复一遍，迄今为止，根据你所做得改变的类型这些（步骤）不是必需的，但是安全比混淆要好。一旦模块构建好了，再次把它拷贝到你的扩展目录，替换旧版本。

此时你可以再次调用PHP解释器， 简单地传入脚本测试刚加入的函数。事实上，为什么不现在就做呢？我会在这儿等待...

完成了？好的。如果用了`var_dump()`而不是`echo`查看每个函数的输出，你或许注意到了`hello_bool()`返回`true`。那就是`RETURN_BOOL()`中的值`1`表现的结果。和在PHP脚本中一样，整数值`0`等于`FALSE`，而其他整数等于`TRUE`。仅仅是作为约定，扩展作者通常用`1`，鼓励你也这么做，但是不要感觉被它限制了。出于另外的可读性目的，也可用宏`RETURN_TRUE`和`RETURN_FALSE`；再来一个`hello_bool()`，这次使用`RETURN_TRUE`：

```
PHP_FUNCTION(hello_bool)

{

    RETURN_TRUE;

}
```

注意这儿没用括号。那样的话，与其他的宏`RETURN_*()`相比，`RETURN_TRUE`和`RETURN_FALSE`的样式有区别（are aberrations），所以确信不要被它误导了（to get caught by this one）。

大概你注意到了，上面的每个范例中，我们都没传入`0`或`1`以表明是否进行拷贝。这是因为，对于类似这些简单的小型标量，不需要分配或释放额外的内存（除了变量容器自身－我们将在第二部分作更深入的考查。）

还有其他的三种返回类型：资源（就像`mysql_connect()`，`fsockopen()`和`ftp_connect()`返回的值的名字一样，但是不限于此），数组（也被称为`HASH`）和对象（由关键字`new`返回）。当我们深入地讲解变量时，会在第二部分看到它们。

初始设置（INI）

Zend引擎提供了两种管理`INI`值的途径。现在我们来看简单一些的，然后当你处理全局数据时再探究更完善但也更复杂的方式。

假设你要在*php.ini*中为你的扩展定义一个值，`hello.greeting`，它保存将在`hello_world()`函数中用到的问候字符串。你需要向*hello.c*和*php_hello.h*中增加一些代码，同时对`hello_module_entry`结构作一些关键性的改变。先在文件*php_hello.h*中靠近用户空间函数的原型声明处增加如下原型：

```
PHP_MINIT_FUNCTION(hello);

PHP_MSHUTDOWN_FUNCTION(hello);



PHP_FUNCTION(hello_world);

PHP_FUNCTION(hello_long);
```

```
PHP_FUNCTION(hello_double);

PHP_FUNCTION(hello_bool);

PHP_FUNCTION(hello_null);
```

现在进入文件*hello.c*，去掉当前版本的`hello_module_entry`，用下面的列表替换它：

```
zend_module_entry hello_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_HELLO_WORLD_EXTNAME,
    hello_functions,
    PHP_MINIT(hello),
    PHP_MSHUTDOWN(hello),
    NULL,
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    PHP_HELLO_WORLD_VERSION,
#endif
    STANDARD_MODULE_PROPERTIES
};
```

```
PHP_INI_BEGIN()

PHP_INI_ENTRY("hello.greeting", "Hello World", PHP_INI_ALL, NULL)

PHP_INI_END()


PHP_MINIT_FUNCTION(hello)
{
    REGISTER_INI_ENTRIES();


    return SUCCESS;
}


PHP_MSHUTDOWN_FUNCTION(hello)
{
    UNREGISTER_INI_ENTRIES();


    return SUCCESS;
}
```

现在，你只需要在文件*hello.c*顶部的那些`#include`旁边增加一个`#include`，这样可以获得正确的支持`INI`的头文件：

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif
```

```
#include "php.h"

#include "php_ini.h"

#include "php_hello.h"
```

最后，你可修改函数hello_world让它使用INI的值：

```
PHP_FUNCTION(hello_world)

{

    RETURN_STRING(INI_STR("hello.greeting"), 1);

}
```

注意，你将要拷贝从INI_STR()返回的值。这是因为，在进入PHP变量堆栈之前（as far as the PHP variable stack is concerned），它都是个静态字符串。实际上，如果你试图修改这个返回的字符串，PHP执行环境会变得不稳定，甚至崩溃。

本节中的第一处改变引入了两个你非常熟悉的函数：MINIT和MSHUTDOWN。正如稍早提到的，这些方法在SAPI初始启动和最终关闭期间被各自调用。它们不会在请求期间和请求之间被调用。本例中它们用来将你的扩展中定义的条目向*php.ini*注册。本系列后面的教程中，你也将看到如何使用MINIT和MSHUTDOWN函数注册资源、对象和流处理器。

函数hello_world()中使用INI_STR()取得hello.greeting条目的当前字符串值。也存在其他类似函数用于取得其他类型的值，长整型、双精度浮点型和布尔型，如下面表格中所示；同时也提供另外的ORIG版本，它们提供在*php.ini*文件中的INI（的原始）设定（在被 *.htaccess*或ini_set()指令改变之前）（原文：provides the value of the referenced INI setting as it was set in php.ini－译注）。

当前值	原始值	类型
INI_STR(name)	INI_ORIG_STR(name)	char * (NULL terminated)
INI_INT(name)	INI_ORIG_INT(name)	signed long
INI_FLT(name)	INI_ORIG_FLT(name)	signed double
INI_BOOL(name)	INI_ORIG_BOOL(name)	zend_bool

传入PHP_INI_ENTRY()的第一个参数含有在*php.ini*文件中用到的名字字符串。为了避免命名空间冲突，你应该使用同函数一样的约定，即是，将你的扩展的名字作为所有值的前缀，就像你对hello.greeting做的一样。仅仅是作为约定，一个句点被用来分隔扩展的名字和更具说明性的初始设定名字。

第二个参数是初始值（默认值？－译注），而且，不管它是不是数字值，都要使用char*类型的字符串。这主要是依据如下事实：*.ini*文件中的原值就是文本－连同其他的一切作为一个文本文件存储。你在后面的脚本中所用到的INI_INT()、INI_FLT()或INI_BOOL()会进行类型转换。

传入的第三个值是访问模式修饰符。这是个位掩码字段，它决定该INI值在何时和何处是可修改的。对于其中的一些，如register_globals，它只是不允许在脚本中用ini_set()改变该值，因为这个设定只在请求启动期间（在脚本能够运行之前）有意义。其他的，如allow_url_fopen，是管理（员才可进行的）设定，你不会希望共享主机环境的用户去修改它，不论是通过ini_set()还是*.htaccess*的指令。该参数的典型值可能是PHP_INI_ALL，表明该值可在任何地方被修改。然后还有PHP_INI_SYSTEM|PHP_INI_PERDIR，表明该设定可在*php.ini*文件中修改，或者通过*.htaccess*文件中的Apache指令修改，但是不能

用ini_set()修改。或者，也可用PHP_INI_SYSTEM，表示该值只能在*php.ini*文件中修改，而不是任何其他地方。

我们现在忽略第四个参数，只是提一下，它允许在初始设定发生改变时－例如使用ini_set()－触发一个方法回调。这使得当设定改变时，扩展可以执行更精确的控制，或是根据新的设定触发一个相关的行为。

全局数值

扩展经常需要在一个特定的请求中由始至终跟踪一个值，而且要把它与可能同时发生的其他请求分开。在非多线程的SAPI中很简单：只是在源文件中声明一个全局变量并在需要时访问它。问题是，由于PHP被设计为可在多线程web服务器（如Apache 2和IIS）中运行，它需要保持各线程使用的全局数值的独立。通过使用TSRM (Thread Safe Resource Management，线程安全的资源管理器) 抽象层－有时称为ZTS（Zend Thread Safety，Zend线程安全），PHP将其极大地简化了。实际上，此时你已经用到了部分TSRM，只是没有意识到。（不要探寻的太辛苦；随着本系列的进行，你将到处看到它的身影。）

如同任意的全局作用域，创建一个线程安全的作用域的第一步是声明它。鉴于本例的目标，你将会声明一个值为0的long型全局数值。每次hello_long()被调用，都将该值增1并返回。在*php_hello.h*文件中的#define PHP_HELLO_H语句后面加入下面的代码段：

```
#ifdef ZTS

#include "TSRM.h"

#endif



ZEND_BEGIN_MODULE_GLOBALS(hello)

    long counter;

ZEND_END_MODULE_GLOBALS(hello)



#ifdef ZTS

#define HELLO_G(v) TSRMG(hello_globals_id, zend_hello_globals *, v)

#else

#define HELLO_G(v) (hello_globals.v)

#endif
```

这次也会使用RINIT方法，所以你需要在头文件中声明它的原型：

```
PHP_MINIT_FUNCTION(hello);

PHP_MSHUTDOWN_FUNCTION(hello);

PHP_RINIT_FUNCTION(hello);
```

现在我们回到文件*hello.c*中并紧接着包含代码块后面加入下面的代码：

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif


#include "php.h"
#include "php_ini.h"
#include "php_hello.h"


ZEND_DECLARE_MODULE_GLOBALS(hello)
```

改变hello_module_entry，加入PHP_RINIT(hello)：

```
zend_module_entry hello_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_HELLO_WORLD_EXTNAME,
    hello_functions,
    PHP_MINIT(hello),
```

```
    PHP_MSHUTDOWN(hello),

    PHP_RINIT(hello),

    NULL,

    NULL,

#if ZEND_MODULE_API_NO >= 20010901

    PHP_HELLO_WORLD_VERSION,

#endif

    STANDARD_MODULE_PROPERTIES

};
```

而且修改你的MINIT函数，附带着另外两个函数，它们在请求启动时执行初始化：

```
static void
php_hello_init_globals(zend_hello_globals *hello_globals)

{

}



PHP_RINIT_FUNCTION(hello)

{

    HELLO_G(counter) = 0;



    return SUCCESS;

}
```

```
PHP_MINIT_FUNCTION(hello)

{

    ZEND_INIT_MODULE_GLOBALS(hello, php_hello_init_globals,
NULL);



    REGISTER_INI_ENTRIES();



    return SUCCESS;

}
```

最后，你可修改hello_long()函数使用这个值:

```
PHP_FUNCTION(hello_long)

{

    HELLO_G(counter)++;



    RETURN_LONG(HELLO_G(counter));

}
```

在你加入*php_hello.h*的代码中，你用到了两个宏－ZEND_BEGIN_MODULE_GLOBALS()和ZEND_END_MODULE_GLOBALS()－用来创建一个名为zend_hello_globals的结构，它包含一个long型的变量。然后有条件地将HELLO_G()定义为从线程池中取得数值，或者从全局作用域中得到－如果你编译的目标是非多线程环境。

在*hello.c*中，你用ZEND_DECLARE_MODULE_GLOBALS()宏来例示zend_hello_globals结构，或者是真的全局（如果此次构建是非线程安全的），或者是本线程的资源池的一个成员。作为扩展作者，我们不需要担心它们的区别，因为Zend引擎为我们打理好这个事情。最后，你在 MINIT中用ZEND_INIT_MODULE_GLOBALS()分配一个线程安全的资源id－现在还不用考虑它是什么。

你可能已经注意到了，php_hello_init_globals()实际上什么也没做，却得多声明个RINIT将变量counter初始化为0。为什么?

关键在于这两个函数何时被调用。php_hello_init_globals()只在开始一个新的进程或线程时被调用；然而， 每个进程都能处理多个请求，所以用这个函数将变量counter初始化为0将只在第一个页面请求时运行。向同一进程发出的后续页面请求将仍会得到以前存储在这儿的counter变量的值，因此不会从0开始计

数。要为每个页面请求把counter变量初始化为0，我们实现RINIT函数， 正如之前看到的，它在每个页面请求之前被调用。此时我们包含php_hello_init_globals()函数是因为稍后你将会用到它，而且在ZEND_INIT_MODULE_GLOBALS()中为这个初始化函数传入NULL将导致在非多线程的平台产生段错误。

初始设置（INI）作为全局数值

回想一下，一个用PHP_INI_ENTRY()声明的*php.ini*值 会作为字符串被解析，并按需用INI_INT()、INI_FLT()和INI_BOOL()转为其他格式。对于某些设定，那么做使得在脚本的执行过程中，当读取这些值时反复做大量不需要的重复工作。幸运的是，可以让ZE将INI值存储为特定的数据类型，并只在它的值被改变时执行类型转换。我们通过声明另一个INI值来尝试下，这次是个布尔值，用来指示变量counter是递增还是递减。开始吧，先把*php_hello.h*中的MODULE_GLOBALS块改成下面的代码:

```
ZEND_BEGIN_MODULE_GLOBALS(hello)

    long counter;

    zend_bool direction;

ZEND_END_MODULE_GLOBALS(hello)
```

接下来，修改PHP_INI_BEGIN()块，声明INI值，像这样:

```
PHP_INI_BEGIN()

    PHP_INI_ENTRY("hello.greeting", "Hello World",
PHP_INI_ALL, NULL)

    STD_PHP_INI_ENTRY("hello.direction", "1", PHP_INI_ALL,
OnUpdateBool, direction, zend_hello_globals, hello_globals)

PHP_INI_END()
```

现在用下面的代码初始化init_globals方法中的设定:

```
static void
php_hello_init_globals(zend_hello_globals *hello_globals)

{

    hello_globals->direction = 1;

}
```

并且最后，在hello_long()中应用这个初始设定来决定是递增还是递减：

```
PHP_FUNCTION(hello_long)

{

    if (HELLO_G(direction)) {

        HELLO_G(counter)++;

    } else {

        HELLO_G(counter)--;

    }



    RETURN_LONG(HELLO_G(counter));

}
```

就是这些。在INI_ENTRY部分指定的OnUpdateBool方法会自动地把*php.ini*、*.htaccess*或者脚本中通过ini_set()提供的值转换为适当的TRUE/FALSE值，这样你就可以在脚本中直接访问它们。STD_PHP_INI_ENTRY的最后三个参数告诉PHP去改变哪个全局变量，我们的扩展的全局（作用域）的结构是什么样子，以及持有这些变量的全局作用域的名字是什么。

核对（代码）完整性

迄今为止，我们的三个文件应该类似于下面的列表。（为了可读性，一些项目被移动和重新组织了。）

config.m4

```
PHP_ARG_ENABLE(hello, whether to enable Hello
World support,

[ --enable-hello   Enable Hello World support])



if test "$PHP_HELLO" = "yes"; then

  AC_DEFINE(HAVE_HELLO, 1, [Whether you have Hello World])
```

```
  PHP_NEW_EXTENSION(hello, hello.c, $ext_shared)
fi
```

php_hello.h

```
#ifndef PHP_HELLO_H
#define PHP_HELLO_H 1

#ifdef ZTS
#include "TSRM.h"
#endif

ZEND_BEGIN_MODULE_GLOBALS(hello)
    long counter;
    zend_bool direction;
ZEND_END_MODULE_GLOBALS(hello)

#ifdef ZTS
#define HELLO_G(v) TSRMG(hello_globals_id, zend_hello_globals *, v)
#else
#define HELLO_G(v) (hello_globals.v)
#endif

#define PHP_HELLO_WORLD_VERSION "1.0"
```

```
#define PHP_HELLO_WORLD_EXTNAME "hello"


PHP_MINIT_FUNCTION(hello);
PHP_MSHUTDOWN_FUNCTION(hello);
PHP_RINIT_FUNCTION(hello);


PHP_FUNCTION(hello_world);
PHP_FUNCTION(hello_long);
PHP_FUNCTION(hello_double);
PHP_FUNCTION(hello_bool);
PHP_FUNCTION(hello_null);


extern zend_module_entry hello_module_entry;
#define phpext_hello_ptr &hello_module_entry


#endif
```

hello.c

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif


#include "php.h"
```

```
#include "php_ini.h"
#include "php_hello.h"

ZEND_DECLARE_MODULE_GLOBALS(hello)

static function_entry hello_functions[] = {
    PHP_FE(hello_world, NULL)
    PHP_FE(hello_long, NULL)
    PHP_FE(hello_double, NULL)
    PHP_FE(hello_bool, NULL)
    PHP_FE(hello_null, NULL)
    {NULL, NULL, NULL}
};

zend_module_entry hello_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_HELLO_WORLD_EXTNAME,
    hello_functions,
    PHP_MINIT(hello),
    PHP_MSHUTDOWN(hello),
    PHP_RINIT(hello),
```

```
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    PHP_HELLO_WORLD_VERSION,
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_HELLO
ZEND_GET_MODULE(hello)
#endif

PHP_INI_BEGIN()
    PHP_INI_ENTRY("hello.greeting", "Hello World",
PHP_INI_ALL, NULL)
    STD_PHP_INI_ENTRY("hello.direction", "1", PHP_INI_ALL,
OnUpdateBool, direction, zend_hello_globals, hello_globals)
PHP_INI_END()

static void php_hello_init_globals(zend_hello_globals *hello_globals)
{
    hello_globals->direction = 1;
}
```

```
PHP_RINIT_FUNCTION(hello)
{
    HELLO_G(counter) = 0;

    return SUCCESS;
}

PHP_MINIT_FUNCTION(hello)
{
    ZEND_INIT_MODULE_GLOBALS(hello, php_hello_init_globals,
NULL);

    REGISTER_INI_ENTRIES();

    return SUCCESS;
}

PHP_MSHUTDOWN_FUNCTION(hello)
{
    UNREGISTER_INI_ENTRIES();

    return SUCCESS;
```

```
}


PHP_FUNCTION(hello_world)
{
    RETURN_STRING("Hello World", 1);
}


PHP_FUNCTION(hello_long)
{
    if (HELLO_G(direction)) {
        HELLO_G(counter)++;
    } else {
        HELLO_G(counter)--;
    }

    RETURN_LONG(HELLO_G(counter));
}


PHP_FUNCTION(hello_double)
{
    RETURN_DOUBLE(3.1415926535);
}
```

```
PHP_FUNCTION(hello_bool)
{
    RETURN_BOOL(1);
}

PHP_FUNCTION(hello_null)
{
    RETURN_NULL();
}
```

下一步是什么?

本教程探究了一个简单PHP扩展的结构，包括导出函数、返回值、声明初始设置（INI）以及在（客户端）请求期间跟踪其内部状态。

下一部分，我们将探究PHP变量的内部结构，以及在脚本环境中如何存储、跟踪和维护它们。在函数被调用时，我们将使用zend_parse_parameters接收来自于程序的参数，以及探究如何返回更加复杂的结果，包括数组、对象和本教程提到的资源等类型。

编写扩展 II：参数、数组和ZVALs

curl 扩展 教程

by Sara Golemon | Sunday, June 5, 2005

介绍

在本系列的第一部分，你了解了PHP扩展的基本结构。你声明了向调用脚本返回静态或者动态值的简单函数，定义INI选项，声明内部数值（全局的）。本教程中，你将看到如何接收从调用脚本传入函数的数值，以及**PHP**和**Zend**引擎如何操作内部的变量。

接收数值

与用户空间的代码不同，内部函数的参数实际上并不是在函数头部声明的，而是将参数列表的地址传入每个函数—不论是否传入了参数—而且，函数可以让Zend引擎将它们转为便于使用的东西。

我们通过定义新函数`hello_greetme()`来看一下，它将接收一个参数然后把它与一些问候的文本一起输出。和以前一样，我们将在三个地方增加代码：

在*php_hello.h*中，靠近其他的函数原型声明处：

```
PHP_FUNCTION(hello_greetme);
```

在*hello.c*中，`hello_functions`结构的底部：

```
    PHP_FE(hello_bool, NULL)
    PHP_FE(hello_null, NULL)
    PHP_FE(hello_greetme, NULL)
    {NULL, NULL, NULL}
};
```

以及*hello.c*底部靠近其他函数的后面：

```
PHP_FUNCTION(hello_greetme)

{

    char *name;

    int name_len;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
            &name, &name_len) == FAILURE) {

        RETURN_NULL();

    }



    php_printf("Hello %s\n", name);



    RETURN_TRUE;

}
```

大多数zend_parse_parameters()块看起来是总是一样的。ZEND_NUM_ARGS()告诉Zend引擎要取得的参数的信息，TSRMLS_CC用来确保线程安全，返回值将被检查是SUCCESS还是FAILURE。通常情况下，zend_parse_parameters()将返回SUCCESS；然而，如果调用脚本试图传入太多或太少的参数，或者传入的参数不能被转为适当的类型，Zend会自动输出一条错误信息并优雅地将控制权还给调用脚本。

本例指定s表明此函数期望只传入一个参数，而且该参数应该被转为string数据类型并装入通过地址传入的char*变量（也就是通过name）。

注意，还有一个int变量通过地址被传入zend_parse_parameters()。这使Zend引擎提供字符串的字节长度，如此二进制安全的函数不再需要依赖strlen(name)确定字符串的长度。实际上使用strlen(name)甚至得不到正确的结果，因为name可能在字符串结束之前包含一个或多个NULL字符。

一旦你的函数确切地得到了name参数，接下来要做的就是把它作为正式问候语的一部分输出。注意，用的是php_printf()而不是更熟悉的printf()。使用这个函数是有重要的理由的。首先，它允许字符串通过PHP的缓冲机制的处理，该机制除了可以缓冲数据，还可执行额外的处理，比如gzip压缩。其次，虽然stdout是极佳的输出目标，使用CLI或CGI时，多数SAPI期望通过特定的pipe或socket传来输出。所以，试图只是通过printf()写入stdout可能导致数据丢失、次序颠倒或者被破坏，因为它绕过了预处理。

最后，函数通过返回TRUE将控制权还给调用程序。你可以没有显式地返回值（默认是NULL）而是让控制到达你的函数的结尾，但这是坏习惯。函数如果不传回任何有意义的结果，应该返回TRUE以说明：“完成任务，一切正常”。

PHP字符串实际可能包含NULL值，所以，输出含有NULL的二进制安全的字符串以及后跟NULL的多个字符的方法是，使用下面的代码块替换php_printf()指令：

```
    php_printf("Hello ");

    PHPWRITE(name, name_len);

    php_printf("\n");
```

这段代码使用php_printf()处理确信没有NULL的字符串，但使用另外的宏－PHPWRITE－处理用户提供的字符串。这个宏接受zend_parse_parameters()提供的长度（name_len）参数以便可以打印name的完整内容，不论它是否含有NULL。

zend_parse_parameters()也会处理可选参数。下一个例子中，你将创建一个函数，它期望一个long（PHP的整数类型）、一个double（浮点）和一个可选的Boolean值。这个函数在用户空间的声明可能像这样：

```
function hello_add($a, $b, $return_long = false) {

    $sum = (int)$a + (float)$b;

    if ($return_long) {
        return intval($sum);
    } else {
        return floatval($sum);
    }
}
```

在C语言中，这个函数类似下面的代码（不要忘记在*php_hello.h*和*hello.c*的hello_functions[]中加入相关条目以启用它）：

```
PHP_FUNCTION(hello_add)
{
    long a;
    double b;
    zend_bool return_long = 0;
```

```
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ld|b",
        &a, &b, &return_long) == FAILURE) {

        RETURN_NULL();

    }



    if (return_long) {

        RETURN_LONG(a + b);

    } else {

        RETURN_DOUBLE(a + b);

    }

}
```

这次你的数据类型字符串读起来像：“我要一个long（l），一个double（d）”。下一个管道字符表示其余的参数是可选的。如果函数调用时没有传入可选参数，那么zend_parse_parameters()将不会改变传给它的对应变量。最后的b当然是用于Boolean。数据类型字符串后面的是a、b和return_long，它们按地址传递，这样zend_parse_parameters()可以将值装入它们。

警告：在32位平台中经常不加区分地使用int和long，但是，当你的代码在64位硬件上编译时，在本该使用一个的地方使用另一个是很危险的。所以记住要把long用于整型，把int用于字符串的长度。

表 *1*显示不同的类型和对应的字母代码，以及可用于zend_parse_parameters()的C类型：

表 **1**：类型和用在**zend_parse_parameters()**中的字母代码

类型	代码	变量类型
Boolean	b	zend_bool
Long	l	long
Double	d	double
String	s	char*, int
Resource	r	zval*
Array	a	zval*
Object	o	zval*
zval	z	zval*

你可能立刻注意到表 *1*中的最后四个类型都是zval*。待会儿你将看到，PHP中实际使用zval数据类型存储所有的用户空间变量。三种“复杂”数据类型，资源、

数组和对象，当它们的数据类型代码被用于zend_parse_parameters()时，Zend引擎会进行类型检查，但是因为在C中没有与它们对应的数据类型，所以不会执行类型转换。

ZVAL

一般而言，zval和PHP用户空间变量是要你费脑筋（wrap your head around）的最困难的概念。它们也将是至关重要的。首先我们考查zval的结构：

```
struct {
    union {
        long lval;
        double dval;
        struct {
            char *val;
            int len;
        } str;
        HashTable *ht;
        zend_object_value obj;
    } value;
    zend_uint refcount;
    zend_uchar type;
    zend_uchar is_ref;
} zval;
```

如你所见，通常每个zval具有三个基本的元素：type、is_ref和refcount。is_ref和refcount将在本教程的稍候讲解；现在让我们关注type。

到如今你应该已经熟悉了PHP的八种数据类型。它们是表1种列出的七种，再加上NULL－虽然实际的字面意义是什么也没有（或许这就是原因），是特殊（unto its own）的类型。给定一个具体的zval，可用三个便利的宏中的一个测试它的类型：Z_TYPE(zval)、Z_TYPE_P(zval*)或Z_TYPE_PP(zval**)。三者之间仅有的功能上的区别在于传入的变量所期望的间接的级别。其他的宏也遵从相同的关于_P和_PP的使用约定，例如你将要看到的宏*VAL。

type的值决定zval的value联合的哪个部分被设置。下面的代码片断演示了一个缩微版本的var_dump()：

```
PHP_FUNCTION(hello_dump)
{
    zval *uservar;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z",
            &uservar) == FAILURE) {
        RETURN_NULL();
    }

    switch (Z_TYPE_P(uservar)) {
        case IS_NULL:
            php_printf("NULL\n");
            break;
        case IS_BOOL:
            php_printf("Boolean: %s\n", Z_LVAL_P(uservar) ? "TRUE" : "FALSE");
            break;
        case IS_LONG:
            php_printf("Long: %ld\n", Z_LVAL_P(uservar));
            break;
        case IS_DOUBLE:
            php_printf("Double: %f\n", Z_DVAL_P(uservar));
            break;
```

```
        case IS_STRING:
            php_printf("String: ");
            PHPWRITE(Z_STRVAL_P(uservar), Z_STRLEN_P(uservar));
            php_printf("\n");
            break;
        case IS_RESOURCE:
            php_printf("Resource\n");
            break;
        case IS_ARRAY:
            php_printf("Array\n");
            break;
        case IS_OBJECT:
            php_printf("Object\n");
            break;
        default:
            php_printf("Unknown\n");
    }


    RETURN_TRUE;
}
```

如你所见，数据类型Boolean和long共享同样的内部元素。如同本系列第一部分中用的RETURN_BOOL()，FALSE用0表示，TRUE用1表示。

当使用zend_parse_parameters()请求一个特定的数据类型时，例如string，Zend引擎检查输入变量的数据类型。如果匹配，Zend只是通过将其传入zval的对应部分来得到正确的数据类型。如果是不同的类型，Zend使用通常的类型转换规则将其转为适当的和／或可能的类型。

修改前面实现的hello_greetme()函数，将它分成小一些的（功能）片断：

```
PHP_FUNCTION(hello_greetme)

{

    zval *zname;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z",
            &zname) == FAILURE) {

        RETURN_NULL();

    }



    convert_to_string(zname);



    php_printf("Hello ");

    PHPWRITE(Z_STRVAL_P(zname), Z_STRLEN_P(zname));

    php_printf("\n");



    RETURN_TRUE;

}
```

这次，zend_parse_parameters()只是获取一个PHP变量（zval），忽略其类型，接着显式地将该变量转为字符串（类似于$zname = (string)$zname;），然后使用zname结构的字符串值调用php_printf()。正如你所猜测的那样，存在其它可用于bool、long和double的convert_to_*()函数。

创建ZVAL

至今为止，你用到的zval已由Zend引擎分配空间，也通过同样的途径释放。然而有时候需要创建你自己的zval。考虑下面的代码段：

```
{
```

```
    zval *temp;


    ALLOC_INIT_ZVAL(temp);


    Z_TYPE_P(temp) = IS_LONG;
    Z_LVAL_P(temp) = 1234;


    zval_ptr_dtor(&temp);
}
```

ALLOC_INIT_ZVAL()，如名所示，为zval*分配内存并把它初始化为一个新变量。那样之后，可用Z_*_P()设置该变量的类型和值。zval_ptr_dtor()处理繁重的清理变量内存工作。

那两个Z_*_P()调用实际可以被归为一条件但的语句：

```
ZVAL_LONG(temp, 1234);
```

对于其他类型也存在相似的宏，并且遵循和本系列第一部分中出现的RETURN_*()相同的语法规则。实际上宏RETURN_*()只是对RETVAL_*()薄薄的一层包装，再深入则是ZVAL_*()。下面的五个版本都是相同的：

```
RETURN_LONG(42);


RETVAL_LONG(42);
return;


ZVAL_LONG(return_value, 42);
return;
```

```
Z_TYPE_P(return_value) = IS_LONG;

Z_LVAL_P(return_value) = 42;

return;
```

```
return_value->type = IS_LONG;

return_value->value.lval = 42;

return;
```

如果你很敏锐，你会思考如何定义它们才能实现在类似`hello_long()`函数中的使用方式。“`return_value`从哪儿来？为什么它不用`ALLOC_INIT_ZVAL()`分配内存？”，你可能会疑惑。

在日常的扩展开发中，你可能不知道`return_value`实际是在每个`PHP_FUNCTION()`原型定义中定义的函数参数。`Zend`引擎给它分配内存并将其初始化为`NULL`，这样即使你的函数没有显式地设置它，返回值仍然是可用的。当你的内部函数执行结束，该值被返回到调用程序，或者被释放—如果调用程序被写为忽略返回值。

数组

因为你之前用过`PHP`，你已经承认了数组作为运载其他变量的变量。这种方式在内部实现上使用了众所周知的`HashTable`。要创建将被返回`PHP`的数组，最简单的方法涉及使用表`2`中列举的函数：

表 **2**：**zval**数组创建函数

PHP语法	**C**语法（**arr**是**zval***）	意义
`$arr = array();`	`array_init(arr);`	初始化一个新数组
`$arr[] = NULL;`	`add_next_index_null(arr);`	向数字索引的数组增加指定类型的值
`$arr[] = 42;`	`add_next_index_long(arr, 42);`	
`$arr[] = true;`	`add_next_index_bool(arr, 1);`	
`$arr[] = 3.14;`	`add_next_index_double(arr, 3.14);`	
`$arr[] = 'foo';`	`add_next_index_string(arr, "foo", 1);`	
`$arr[] = $myvar;`	`add_next_index_zval(arr, myvar);`	
`$arr[0] = NULL;`	`add_index_null(arr, 0);`	向数组中指定的数字索引增加指定类型的值
`$arr[1] = 42;`	`add_index_long(arr, 1, 42);`	
`$arr[2] = true;`	`add_index_bool(arr, 2, 1);`	
`$arr[3] = 3.14;`	`add_index_double(arr, 3, 3.14);`	
`$arr[4] = 'foo';`	`add_index_string(arr, 4, "foo", 1);`	
`$arr[5] = $myvar;`	`add_index_zval(arr, 5, myvar);`	
`$arr['abc'] = NULL;`	`add_assoc_null(arr, "abc");`	

`$arr['def'] = 711;`	`add_assoc_long(arr, "def", 711);`
`$arr['ghi'] = true;`	`add_assoc_bool(arr, "ghi", 1);`
`$arr['jkl'] = 1.44;`	`add_assoc_double(arr, "jkl", 1.44);`
`$arr['mno'] = 'baz';`	`add_assoc_string(arr, "mno", "baz", 1);`
`$arr['pqr'] = $myvar;`	`add_assoc_zval(arr, "pqr", myvar);`

向关联索引的数组增加指定类型的值

同`RETURN_STRING()`宏一样，`add_*_string()`函数的最后一个参数接受`1`或`0`来指明字符串内容是否被拷贝。它们各自都有形如`add_*_stringl()`的对应版本。`l`表示会显式提供字符串长度（而不是让`Zend`引擎调用`strval()`来得到这个值，该函数不是二进制安全的）。

使用二进制安全的形式很简单，只需要在（表示）复制的参数前面指定长度，像这样：

```
add_assoc_stringl(arr, "someStringVar", "baz", 3, 1);
```

使用`add_assoc_*()`函数，数组的关键字假定不包含`NULL`－`add_assoc_*()` 函数自身对于关键字不是二进制安全的。不可使用带有`NULL`的关键字（实际上对象的受保护的和私有的属性已经使用了这种技术），可是如果必须这样做，当我们稍候使用`zend_hash_*()`函数时，你将立刻知道怎样实现。

要实践学到的东西，创建下面的函数，它返回一个数组到调用程序。确定向*`php_hello.h`*和`hello_functions[]`中增加条目以使该函数得到适当地声明。

```
PHP_FUNCTION(hello_array)

{
    char *mystr;

    zval *mysubarray;


    array_init(return_value);


    add_index_long(return_value, 42, 123);


    add_next_index_string(return_value, "I should now be found at index 43", 1);


    add_next_index_stringl(return_value, "I'm at 44!", 10, 1);


    mystr = estrdup("Forty Five");
```

```
    add_next_index_string(return_value, mystr, 0);


    add_assoc_double(return_value, "pi", 3.1415926535);


    ALLOC_INIT_ZVAL(mysubarray);
    array_init(mysubarray);
    add_next_index_string(mysubarray, "hello", 1);
    add_assoc_zval(return_value, "subarray", mysubarray);
}
```

构建扩展并查看`var_dump(hello_array());`的结果：

```
array(6) {
  [42]=>
  int(123)
  [43]=>
  string(33) "I should now be found at index 43"
  [44]=>
  string(10) "I'm at 44!"
  [45]=>
  string(10) "Forty Five"
  ["pi"]=>
  float(3.1415926535)
  ["subarray"]=>
  array(1) {
    [0]=>
    string(5) "hello"
  }
}
```

从数组中取回值意味着使用`ZENDAPI`的`zend_hash`族函数直接从`HashTable`中把它们作为`zval**`抽取出来。我们以接受一个数组为参数的简单函数开始：

```
function hello_array_strings($arr) {
```

```
    if (!is_array($arr)) return NULL;


    printf("The array passed contains %d elements\n", count($arr));


    foreach($arr as $data) {
        if (is_string($data)) echo "$data\n";
    }
}
```

或者，在C中：

```
PHP_FUNCTION(hello_array_strings)
{
    zval *arr, **data;
    HashTable *arr_hash;
    HashPosition pointer;
    int array_count;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a", &arr) == FAILURE) {
        RETURN_NULL();
    }

    arr_hash = Z_ARRVAL_P(arr);
    array_count = zend_hash_num_elements(arr_hash);
```

```
    php_printf("The array passed contains %d elements\n", array_count);



    for(zend_hash_internal_pointer_reset_ex(arr_hash, &pointer);
            zend_hash_get_current_data_ex(arr_hash, (void**) &data, &pointer) == SUCCESS;
            zend_hash_move_forward_ex(arr_hash, &pointer)) {



        if (Z_TYPE_PP(data) == IS_STRING) {

        PHPWRITE(Z_STRVAL_PP(data), Z_STRLEN_PP(data));

      php_printf("\n");

    }

    }

    RETURN_TRUE;

}
```

为了保持函数的简短，只输出了字符串类型的数组元素。你可能会奇怪，为什么不用之前在`hello_greetme()`函数中用过的`convert_to_string()`？我们来看看那样做怎么样；用下面的代码替换上面的`for`循环：

```
for(zend_hash_internal_pointer_reset_ex(arr_hash, &pointer);
zend_hash_get_current_data_ex(arr_hash, (void**) &data, &pointer) == SUCCESS;
zend_hash_move_forward_ex(arr_hash, &pointer)) {



        convert_to_string_ex(data);

        PHPWRITE(Z_STRVAL_PP(data), Z_STRLEN_PP(data));

        php_printf("\n");

}
```

现在重新编译扩展并运行下面的用户空间代码：

```
<?php

$a = array('foo',123);
var_dump($a);
hello_array_strings($a);
var_dump($a);

?>
```

注意，原始数组被改变了！记住，convert_to_*()函数具有与调用set_type()相同的效果。由于处理的数组与传入的是同一个，此处改变它的类型将改变原始变量。要避免则需要首先制作一份zval的副本。为此，再次将for循环改成下面的代码：

```
for(zend_hash_internal_pointer_reset_ex(arr_hash, &pointer);
    zend_hash_get_current_data_ex(arr_hash, (void**) &data, &pointer) == SUCCESS;
    zend_hash_move_forward_ex(arr_hash, &pointer)) {



    zval temp;



    temp = **data;

    zval_copy_ctor(&temp);

    convert_to_string(&temp);

    PHPWRITE(Z_STRVAL(temp), Z_STRLEN(temp));

    php_printf("\n");

    zval_dtor(&temp);

}
```

这次更明显的－temp = **data－只是拷贝了原zval的数据，但由于zval可能含有类似char*字符串或HashTable*数组等额外已分配资源，这些相关的资源需要用zval_copy_ctor()进行复制。之后就是普通的转换、打印，以及最终用zval_dtor()去除这个副本用到的资源。

如果你感到奇怪：为什么首次引入convert_to_string()时（参见hello_greetme()的第二个版本，功能被划分为小片断－译注）没做zval_copy_ctor()？那是因为向函数传入变量会自动地从原始变量分离出zval，拷贝一个副本。这始终只作用于zval的表层（on the base），所以，任何次级资源（例如数组元素和对象属性）在使用前仍然需要进行分离。

既然已经看过了数组的值，我们稍微扩充下此次练习，也来看看（数组的）关键字：

```
for(zend_hash_internal_pointer_reset_ex(arr_hash, &pointer);
    zend_hash_get_current_data_ex(arr_hash, (void**) &data, &pointer) == SUCCESS;
    zend_hash_move_forward_ex(arr_hash, &pointer)) {



    zval temp;

    char *key;

    int key_len;

    long index;



    if (zend_hash_get_current_key_ex(arr_hash, &key, &key_len, &index,
              0, &pointer) == HASH_KEY_IS_STRING) {

        PHPWRITE(key, key_len);

    } else {

        php_printf("%ld", index);

    }



    php_printf(" => ");



    temp = **data;

    zval_copy_ctor(&temp);

    convert_to_string(&temp);

    PHPWRITE(Z_STRVAL(temp), Z_STRLEN(temp));

    php_printf("\n");

    zval_dtor(&temp);
```

```
}
```

记住数组可以具有数字索引、关联字符串关键字或兼有二者。调用zend_hash_get_current_key_ex()使得既可以取得数组当前位置的索引（原文是type－译注），也可以根据返回值确定它的类型，可能为HASH_KEY_IS_STRING、HASH_KEY_IS_LONG或HASH_KEY_NON_EXISTANT。由于zend_hash_get_current_data_ex()能够返回zval**，你可以确定它不会返回HASH_KEY_NON_EXISTANT，所以只需要检测IS_STRING和IS_LONG的可能性。

遍历HashTable还有其他方法。Zend引擎针对这个任务展露了三个非常相似的函数：zend_hash_apply()、zend_hash_apply_with_argument()和zend_hash_apply_with_arguments()。第一种形式仅仅遍历HashTable，第二种形式允许传入单个void*参数，第三种形式通过vararg列表允许数量不限的参数。hello_array_walk()展示了它们各自的行为：

```
static int php_hello_array_walk(zval **element TSRMLS_DC)

{

    zval temp;



    temp = **element;

    zval_copy_ctor(&temp);

    convert_to_string(&temp);

    PHPWRITE(Z_STRVAL(temp), Z_STRLEN(temp));

    php_printf("\n");

    zval_dtor(&temp);



    return ZEND_HASH_APPLY_KEEP;

}



static int php_hello_array_walk_arg(zval **element, char *greeting TSRMLS_DC)

{
```

```
    php_printf("%s", greeting);

    php_hello_array_walk(element TSRMLS_CC);


    return ZEND_HASH_APPLY_KEEP;
}


static int php_hello_array_walk_args(zval **element, int num_args, va_list args,
    zend_hash_key *hash_key)
{
    char *prefix = va_arg(args, char*);
    char *suffix = va_arg(args, char*);
    TSRMLS_FETCH();


    php_printf("%s", prefix);
    php_hello_array_walk(element TSRMLS_CC);
    php_printf("%s\n", suffix);


    return ZEND_HASH_APPLY_KEEP;
}


PHP_FUNCTION(hello_array_walk)
{
    zval *zarray;
```

```
    int print_newline = 1;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a", &zarray) == FAILURE) {

        RETURN_NULL();

    }



    zend_hash_apply(Z_ARRVAL_P(zarray), (apply_func_t)php_hello_array_walk TSRMLS_CC);

    zend_hash_apply_with_argument(Z_ARRVAL_P(zarray),
        (apply_func_arg_t)php_hello_array_walk_arg, "Hello " TSRMLS_CC);

    zend_hash_apply_with_arguments(Z_ARRVAL_P(zarray),
        (apply_func_args_t)php_hello_array_walk_args, 2, "Hello ", "Welcome to my extension!");



    RETURN_TRUE;

}
```

上述代码大多明白易懂，你应该对相关函数的用法足够熟悉了。传入hello_array_walk()的数组被遍历了三次，一次不带参数，一次带单个参数，第三次带两个参数。这次的设计中，walk_arg()和walk_args()函数依赖于不带参的walk()函数处理转换和打印zval的工作，因为这项工作在三者中是通用的。

如同多数用到zend_hash_apply()的地方，在这段代码中，walk()（原文是“apply()”－译注）函数返回ZEND_HASH_APPLY_KEEP。这告诉zend_hash_apply()函数离开HashTable中的（当前）元素，继续处理下一个。这儿也可以返回其他值：ZEND_HASH_APPLY_REMOVE－如名所示，删除当前元素并继续应用到下一个；ZEND_HASH_APPLY_STOP－在当前元素中止数组的遍历并退出zend_hash_apply()函数。

其中不太熟悉的部件大概是TSRMLS_FETCH()。回想第一部分，TSRMLS_*宏是TSRM层的一部分，用于避免各线程的作用域被其他的侵入。因为zend_hash_apply()的多线程版本用了vararg列表，tsrm_ls标记没有传入walk()函数。为了在回调php_hello_array_walk()时找回并使用它，你的函数调用TSRMLS_FETCH()从资源池中找到正确的线程。（注意：该方法比直接传参慢很多，所以非必须不要使用。）

用foreach的形式遍历数组是常见的任务，但是常常需要通过数字索引或关联关键字查找数组中的特定值。下一个函数返回由第一个参数指定的数组的一个值，该值基于第二个参数指定的偏移量或关键字得到。

```
PHP_FUNCTION(hello_array_value)

{
```

```
    zval *zarray, *zoffset, **zvalue;

    long index = 0;

    char *key = NULL;

    int key_len = 0;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "az",
            &zarray, &zoffset) == FAILURE) {

        RETURN_NULL();

    }



    switch (Z_TYPE_P(zoffset)) {

        case IS_NULL:

            index = 0;

            break;

        case IS_DOUBLE:

           index = (long)Z_DVAL_P(zoffset);

            break;

        case IS_BOOL:

        case IS_LONG:

        case IS_RESOURCE:

            index = Z_LVAL_P(zoffset);

            break;

        case IS_STRING:
```

```
            key = Z_STRVAL_P(zoffset);

            key_len = Z_STRLEN_P(zoffset);

            break;

        case IS_ARRAY:

            key = "Array";

            key_len = sizeof("Array") - 1;

            break;

        case IS_OBJECT:

            key = "Object";

            key_len = sizeof("Object") - 1;

            break;

        default:

            key = "Unknown";

            key_len = sizeof("Unknown") - 1;

    }



    if (key && zend_hash_find(Z_ARRVAL_P(zarray),
            key, key_len + 1, (void**)&zvalue) == FAILURE) {

        RETURN_NULL();

    } else if (!key && zend_hash_index_find(Z_ARRVAL_P(zarray),
            index, (void**)&zvalue) == FAILURE) {

        RETURN_NULL();

    }
```

```
    *return_value = **zvalue;

    zval_copy_ctor(return_value);

}
```

该函数开始于switch块，它用和Zend引擎相同的方式处理类型转换。NULL视为0，Boolean据值视为0或1，double转化为long（也进行截断），resource转化为它的数字值。对resource类型的处理是PHP 3的遗留，那时候资源确实只是在查找中用的数字，而不是特殊的类型（unto themselves）。

数组和对象只不过视为字符串字面量“Array”或“Object”，因没有什么转换具有实在的意义。最后插入缺省条件极小心地处理其他情形，以防PHP的未来版本可能引入其他数据类型而使该扩展产生编译问题。

如果函数查找的是关联关键字，那么key只会被设置为非NULL，所以可用它来确定查找是基于关联还是索引。如果因为关键字不存在使选定的查找失败了，函数因此返回NULL表明失败。否则找到的zval被复制到return_value。

符号表作为数组

如果以前用过$GLOBALS数组，你应该知道在PHP脚本的全局作用域声明和使用的每个变量也都存在于这个数组中。回想下，数组在内部是用HashTable表示的，想到个问题：“是否存在特别的地方可以找到GLOBALS数组？”答案是“存在”，就是EG(symbol_table)－Executor Globals结构，它的类型是HashTable（不是HashTable*，留心，只是HashTable）。

已经知道了如何查找数组中关联于关键字的元素，现在又知道了哪儿可以找到全局符号表，应该可以在扩展的代码中查找变量了：

```
PHP_FUNCTION(hello_get_global_var)

{

    char *varname;

    int varname_len;

    zval **varvalue;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
            &varname, &varname_len) == FAILURE) {

        RETURN_NULL();

    }
```

```
    if (zend_hash_find(&EG(symbol_table), varname, varname_len + 1,
            (void**)&varvalue) == FAILURE) {

        php_error_docref(NULL TSRMLS_CC, E_NOTICE, "Undefined variable: %s", varname);

        RETURN_NULL();

    }



    *return_value = **varvalue;

    zval_copy_ctor(return_value);

}
```

现在这些对你来说应该非常熟悉了。这个函数接受一个字符串参数，用它从全局作用域找到一个变量并且返回其副本。

这儿有个新内容`php_error_docref()`。你会发现该函数或是它的近亲遍布PHP源码树的各个角落。第一个参数是个可选的文档引用（缺省是用当前的函数）。其次是到处都出现的`TSRMLS_CC`，后面跟着关于错误的严重级别，最后是`printf()`样式的描述错误信息的格式字符串及相关的参数。让你的函数在失败情形下总是提供一些有意义的错误是很重要的。实际上，现在是个很好的机会，回头向`hello_array_value()`加入一条错误语句。本教程结尾的核对（代码）完整性一节也将包含它们（指错误语句－译注）。

除了全局符号表，Zend引擎也维持一个到局部符号表的引用。由于内部函数没有自己的符号表（为什么需要这个呢？），局部符号表实际上引用了调用当前内部函数的用户函数的局部作用域。看一个简单的函数，它设置了局部作用域的变量：

```
PHP_FUNCTION(hello_set_local_var)

{

    zval *newvar;

    char *varname;

    int varname_len;

    zval *value;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sz",
            &varname, &varname_len, &value) == FAILURE) {
```

```
        RETURN_NULL();
    }


    ALLOC_INIT_ZVAL(newvar);
    *newvar = *value;
    zval_copy_ctor(newvar);
    zend_hash_add(EG(active_symbol_table), varname, varname_len + 1,
            &newvar, sizeof(zval*), NULL);


    RETURN_TRUE;
}
```

这儿绝没有什么新东西。继续前进，构建迄今得到的（代码），针对它运行一些测试脚本。确信得到了期望的结果，确实得到了。

引用计数

迄今为止，我们向HashTables中加入的zval要么是新建的，要么是刚拷贝的。它们都是独立的，只占用自己的资源且只存在于某个HashTable中。作为一个语言设计的概念，创建和拷贝变量的方法是“很好”的，但是习惯了C程序设计就会知道，通过避免拷贝大块的数据—除非绝对必须，来节约内存和CPU时间并不少见。考虑这段用户代码：

```
<?php

$a = file_get_contents('fourMegabyteLogFile.log');
$b = $a;
unset($a);

?>
```

如果执行zval_copy_ctor()（将会对字符串内容执行estrndup()）将$a拷贝给$b，那么这个简短的脚本实际会用掉8M内存来存储同一4M文件的两份相同的副本。在最后一步取消$a只会更糟，因为原始字符串被efree()了。用C做这个将会很简单，大概是这样：b = a; a = NULL;。

幸运的是，Zend引擎稍微聪明些。当创建$a时，会创建一个潜在的string类型的zval，它含有日至文件的内容。这个zval通过调用zend_hash_add()被赋给 $a变量。当$a被拷贝给$b，引擎做类似下面的事情：

```
{
    zval **value;

    zend_hash_find(EG(active_symbol_table), "a", sizeof("a"), (void**)&value);

    ZVAL_ADDREF(*value);

    zend_hash_add(EG(active_symbol_table), "b", sizeof("b"), value, sizeof(zval*));
}
```

当然，实际代码会更复杂，但是这儿的要点是`ZVAL_ADDREF()`。记住`zval`含有四个要素。你已经了解了`type`和`value`；这次处理的是`refcount`。如名所示，`refcount`是特定的`zval`在符号表中、数组中或其他地方被引用次数的计数器。

使用`ALLOC_INIT_ZVAL()`会把`refcount`设为`1`，所以，如果要把它返回或加入`HashTable`一次，你什么也不用去做。在上面的代码中，你从`HashTable`中取得一个`zval`但是没有删除它，所以,它的`refcount`匹配引用它的位置的数量。为了从其他位置引用该值，你需要增加它的引用计数。

当用户空间代码调用`unset($a)`，引擎对该变量执行`zval_ptr_dtor()`。在前面用到的`zval_ptr_dtor()`中，你看不到的事实是，这个调用没有必要销毁该`zval`和它的内容。实际工作是减少`refcount`。如果，且仅仅是如果，`refcount`变成了 0，`Zend`引擎会销毁该`zval`...

第**2**页继续。

编写扩展_II - 参数、数组和ZVALs[继续]

扩展 教程

by Sara Golemon | Sunday, June 5, 2005

拷贝 vs 引用

有两种方法引用zval。第一种，如上文示范的，被称为写复制引用（copy-on-write referencing）。第二种形式是完全引用（full referencing）；当说起“引用”时，用户空间代码的编写者更熟悉这种，以用户空间代码的形式出现类似于：$a = &$b;。

在zval中，这两种类型的区别在于它的is_ref成员的值，0表示写复制引用，非0表示完全引用。注意，一个zval不可能同时具有两种引用类型。所以，如果变量起初是is_ref（即完全引用－译注），然后以拷贝的方式赋给新的变量，那么必将执行一个完全拷贝。考虑下面的用户空间代码：

```
<?php

$a = 1;
$b = &$a;
$c = $a;

?>
```

在这段代码中，为$a创建并初始化了一个zval，将is_ref设为0，将refcount设为1。当$a被$b引用时，is_ref变为1，refcount递增至2。当拷贝至$c时，Zend引擎不能只是递增refcount至3，因为如此则$c变成了$a的完全引用。关闭is_ref也不行，因为如此会使$b看起来像是$a的一份拷贝而不是引用。所以此时分配了一个新的zval，并使用zval_copy_ctor()把原始（zval）的值拷贝给它。原始zval仍为is_ref==1、refcount==2，同时新zval则为is_ref=0、refcount=1。现在来看另一块内容相同的代码块，只是顺序稍有不同：

```
<?php

$a = 1;
$c = $a;
$b = &$a;
```

```
?>
```

最终结果不变，$b是$a的完全引用，并且$c是$a的一份拷贝。然而这次的内部效果稍有区别。如前，开始时为$a创建一个is_ref==0并且refcount=1的新zval。$c = $a;语句将同一个zval赋给$c变量，同时将refcount增至2，is_ref仍是0。当Zend引擎遇到$b = &$a;，它想要只是将is_ref设为1，但是当然不行，因为那将影响到$c。所以改为创建新的zval并用zval_copy_ctor()将原始（zval）的内容拷贝给它。然后递减原始zval的refcount以表明$a不再使用该zval。代替地，（Zend）设置新zval的is_ref为1、refcount为2，并且更新$a和$b变量指向它（新zval）。

核对（代码）完整性

如前，下面是我们的三个主要文件的完整代码：

config.m4

```
PHP_ARG_ENABLE(hello, [whether to enable Hello World support],
    [ --enable-hello   Enable Hello World support])

    if test "$PHP_HELLO" = "yes"; then
      AC_DEFINE(HAVE_HELLO, 1, [Whether you have Hello World])
      PHP_NEW_EXTENSION(hello, hello.c, $ext_shared)
    fi
```

php_hello.h

```
#ifndef PHP_HELLO_H
#define PHP_HELLO_H 1

#ifdef ZTS
#include "TSRM.h"
#endif

ZEND_BEGIN_MODULE_GLOBALS(hello)
```

```
    long counter;
    zend_bool direction;
ZEND_END_MODULE_GLOBALS(hello)

#ifdef ZTS
#define HELLO_G(v) TSRMG(hello_globals_id, zend_hello_globals *, v)
#else
#define HELLO_G(v) (hello_globals.v)
#endif

#define PHP_HELLO_WORLD_VERSION "1.0"
#define PHP_HELLO_WORLD_EXTNAME "hello"

PHP_MINIT_FUNCTION(hello);
PHP_MSHUTDOWN_FUNCTION(hello);
PHP_RINIT_FUNCTION(hello);

PHP_FUNCTION(hello_world);
PHP_FUNCTION(hello_long);
PHP_FUNCTION(hello_double);
PHP_FUNCTION(hello_bool);
PHP_FUNCTION(hello_null);
```

```
PHP_FUNCTION(hello_greetme);

PHP_FUNCTION(hello_add);

PHP_FUNCTION(hello_dump);

PHP_FUNCTION(hello_array);

PHP_FUNCTION(hello_array_strings);

PHP_FUNCTION(hello_array_walk);

PHP_FUNCTION(hello_array_value);

PHP_FUNCTION(hello_get_global_var);

PHP_FUNCTION(hello_set_local_var);


extern zend_module_entry hello_module_entry;

#define phpext_hello_ptr &hello_module_entry


#endif
```

hello.c

```
#ifdef HAVE_CONFIG_H

#include "config.h"

#endif


#include "php.h"

#include "php_ini.h"
```

```
#include "php_hello.h"

ZEND_DECLARE_MODULE_GLOBALS(hello)

static function_entry hello_functions[] = {
    PHP_FE(hello_world, NULL)
    PHP_FE(hello_long, NULL)
    PHP_FE(hello_double, NULL)
    PHP_FE(hello_bool, NULL)
    PHP_FE(hello_null, NULL)
    PHP_FE(hello_greetme, NULL)
    PHP_FE(hello_add, NULL)
    PHP_FE(hello_dump, NULL)
    PHP_FE(hello_array, NULL)
    PHP_FE(hello_array_strings, NULL)
    PHP_FE(hello_array_walk, NULL)
    PHP_FE(hello_array_value, NULL)
    PHP_FE(hello_get_global_var, NULL)
    PHP_FE(hello_set_local_var, NULL)
    {NULL, NULL, NULL}
};

zend_module_entry hello_module_entry = {
```

```
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_HELLO_WORLD_EXTNAME,
    hello_functions,
    PHP_MINIT(hello),
    PHP_MSHUTDOWN(hello),
    PHP_RINIT(hello),
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    PHP_HELLO_WORLD_VERSION,
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_HELLO
    ZEND_GET_MODULE(hello)
#endif

PHP_INI_BEGIN()
    PHP_INI_ENTRY("hello.greeting", "Hello World", PHP_INI_ALL, NULL)
```

```
    STD_PHP_INI_ENTRY("hello.direction", "1", PHP_INI_ALL, OnUpdateBool,\
            direction, zend_hello_globals, hello_globals)

PHP_INI_END()


static void php_hello_init_globals(zend_hello_globals *hello_globals)

{

    hello_globals->direction = 1;

}


PHP_RINIT_FUNCTION(hello)

{

    HELLO_G(counter) = 0;


    return SUCCESS;

}


PHP_MINIT_FUNCTION(hello)

{

    ZEND_INIT_MODULE_GLOBALS(hello, php_hello_init_globals, NULL);


    REGISTER_INI_ENTRIES();


    return SUCCESS;
```

```
}

PHP_MSHUTDOWN_FUNCTION(hello)
{
    UNREGISTER_INI_ENTRIES();

    return SUCCESS;
}

PHP_FUNCTION(hello_world)
{
    RETURN_STRING("Hello World", 1);
}

PHP_FUNCTION(hello_long)
{
    if (HELLO_G(direction)) {
        HELLO_G(counter)++;
    } else {
        HELLO_G(counter)--;
    }
```

```
    RETURN_LONG(HELLO_G(counter));
}

PHP_FUNCTION(hello_double)
{
    RETURN_DOUBLE(3.1415926535);
}

PHP_FUNCTION(hello_bool)
{
    RETURN_BOOL(1);
}

PHP_FUNCTION(hello_null)
{
    RETURN_NULL();
}

PHP_FUNCTION(hello_greetme)
{
    zval *zname;
```

```
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &zname) == FAILURE) {
        RETURN_NULL();
    }

    convert_to_string(zname);

    php_printf("Hello ");
    PHPWRITE(Z_STRVAL_P(zname), Z_STRLEN_P(zname));
    php_printf("\n");

    RETURN_TRUE;
}

PHP_FUNCTION(hello_add)
{
    long a;
    double b;
    zend_bool return_long = 0;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ld|b",
        &a, &b, &return_long) == FAILURE) {
        RETURN_NULL();
    }
```

```
    if (return_long) {
        RETURN_LONG(a + b);
    } else {
        RETURN_DOUBLE(a + b);
    }
}

PHP_FUNCTION(hello_dump)
{
    zval *uservar;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &uservar) == FAILURE) {
    RETURN_NULL();
    }

    switch (Z_TYPE_P(uservar)) {
    case IS_NULL:
        php_printf("NULL\n");
        break;
    case IS_BOOL:
        php_printf("Boolean: %s\n", Z_LVAL_P(uservar) ? "TRUE" : "FALSE");
```

```
            break;
        case IS_LONG:
            php_printf("Long: %ld\n", Z_LVAL_P(uservar));
            break;
        case IS_DOUBLE:
            php_printf("Double: %f\n", Z_DVAL_P(uservar));
            break;
        case IS_STRING:
            php_printf("String: ");
            PHPWRITE(Z_STRVAL_P(uservar), Z_STRLEN_P(uservar));
            php_printf("\n");
            break;
        case IS_RESOURCE:
            php_printf("Resource\n");
            break;
        case IS_ARRAY:
            php_printf("Array\n");
            break;
        case IS_OBJECT:
            php_printf("Object\n");
            break;
        default:
```

```
        php_printf("Unknown\n");
    }

    RETURN_TRUE;
}

PHP_FUNCTION(hello_array)
{
    char *mystr;
    zval *mysubarray;

    array_init(return_value);

    add_index_long(return_value, 42, 123);

    add_next_index_string(return_value, "I should now be found at index 43", 1);

    add_next_index_stringl(return_value, "I'm at 44!", 10, 1);

    mystr = estrdup("Forty Five");
    add_next_index_string(return_value, mystr, 0);

    add_assoc_double(return_value, "pi", 3.1415926535);
```

```
    ALLOC_INIT_ZVAL(mysubarray);
    array_init(mysubarray);
    add_next_index_string(mysubarray, "hello", 1);
    php_printf("mysubarray->refcount = %d\n", mysubarray->refcount);
    mysubarray->refcount = 2;
    php_printf("mysubarray->refcount = %d\n", mysubarray->refcount);
    add_assoc_zval(return_value, "subarray", mysubarray);

    php_printf("mysubarray->refcount = %d\n", mysubarray->refcount);
}

PHP_FUNCTION(hello_array_strings)
{
    zval *arr, **data;
    HashTable *arr_hash;
    HashPosition pointer;
    int array_count;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a", &arr) == FAILURE) {
        RETURN_NULL();
    }
```

```
    arr_hash = Z_ARRVAL_P(arr);

    array_count = zend_hash_num_elements(arr_hash);


    php_printf("The array passed contains %d elements\n", array_count);


    for(zend_hash_internal_pointer_reset_ex(arr_hash, &pointer);
        zend_hash_get_current_data_ex(arr_hash, (void**) &data, &pointer) == SUCCESS;
        zend_hash_move_forward_ex(arr_hash, &pointer)) {


        zval temp;

        char *key;

        int key_len;

        long index;


        if (zend_hash_get_current_key_ex(arr_hash, &key, &key_len,
            &index, 0, &pointer) == HASH_KEY_IS_STRING) {

            PHPWRITE(key, key_len);

        } else {

            php_printf("%ld", index);

        }


        php_printf(" => ");
```

```
        temp = **data;
        zval_copy_ctor(&temp);
        convert_to_string(&temp);
        PHPWRITE(Z_STRVAL(temp), Z_STRLEN(temp));
        php_printf("\n");
        zval_dtor(&temp);
    }

    RETURN_TRUE;
}

static int php_hello_array_walk(zval **element TSRMLS_DC)
{
    zval temp;

    temp = **element;
    zval_copy_ctor(&temp);
    convert_to_string(&temp);
    PHPWRITE(Z_STRVAL(temp), Z_STRLEN(temp));
    php_printf("\n");
    zval_dtor(&temp);
```

```
    return ZEND_HASH_APPLY_KEEP;
}

static int php_hello_array_walk_arg(zval **element, char *greeting TSRMLS_DC)
{
    php_printf("%s", greeting);
    php_hello_array_walk(element TSRMLS_CC);

    return ZEND_HASH_APPLY_KEEP;
}

static int php_hello_array_walk_args(zval **element, int num_args,
    var_list args, zend_hash_key *hash_key)
{
    char *prefix = va_arg(args, char*);
    char *suffix = va_arg(args, char*);
    TSRMLS_FETCH();

    php_printf("%s", prefix);
    php_hello_array_walk(element TSRMLS_CC);
    php_printf("%s\n", suffix);

    return ZEND_HASH_APPLY_KEEP;
```

```
}


PHP_FUNCTION(hello_array_walk)
{
    zval *zarray;
    int print_newline = 1;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a", &zarray) == FAILURE) {
        RETURN_NULL();
    }

    zend_hash_apply(Z_ARRVAL_P(zarray), (apply_func_t)php_hello_array_walk TSRMLS_CC);
    zend_hash_internal_pointer_reset(Z_ARRVAL_P(zarray));
    zend_hash_apply_with_argument(Z_ARRVAL_P(zarray),
        (apply_func_arg_t)php_hello_array_walk_arg, "Hello " TSRMLS_CC);
    zend_hash_apply_with_arguments(Z_ARRVAL_P(zarray),
        (apply_func_args_t)php_hello_array_walk_args, 2, "Hello ",
        "Welcome to my extension!");

    RETURN_TRUE;
}


PHP_FUNCTION(hello_array_value)
```

```
{

    zval *zarray, *zoffset, **zvalue;

    long index = 0;

    char *key = NULL;

    int key_len = 0;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "az",
        &zarray, &zoffset) == FAILURE) {

    RETURN_NULL();

    }



    switch (Z_TYPE_P(zoffset)) {

    case IS_NULL:

        index = 0;

        break;

    case IS_DOUBLE:

        index = (long)Z_DVAL_P(zoffset);

        break;

    case IS_BOOL:

    case IS_LONG:

    case IS_RESOURCE:

        index = Z_LVAL_P(zoffset);

        break;
```

```
    case IS_STRING:
        key = Z_STRVAL_P(zoffset);
        key_len = Z_STRLEN_P(zoffset);
        break;
    case IS_ARRAY:
        key = "Array";
        key_len = sizeof("Array") - 1;
        break;
    case IS_OBJECT:
        key = "Object";
        key_len = sizeof("Object") - 1;
        break;
    default:
        key = "Unknown";
        key_len = sizeof("Unknown") - 1;
    }


    if (key && zend_hash_find(Z_ARRVAL_P(zarray),
        key, key_len + 1, (void**)&zvalue) == FAILURE) {
        php_error_docref(NULL TSRMLS_CC, E_NOTICE, "Undefined index: %s", key);
        RETURN_NULL();
    } else if (!key && zend_hash_index_find(Z_ARRVAL_P(zarray),
        index, (void**)&zvalue) == FAILURE) {
```

```
        php_error_docref(NULL TSRMLS_CC, E_NOTICE, "Undefined index: %ld", index);

        RETURN_NULL();

    }



    *return_value = **zvalue;

    zval_copy_ctor(return_value);

}



PHP_FUNCTION(hello_get_global_var)

{

    char *varname;

    int varname_len;

    zval **varvalue;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
        &varname, &varname_len) == FAILURE) {

        RETURN_NULL();

    }



    if (zend_hash_find(&EG(symbol_table), varname, varname_len + 1,
        (void**)&varvalue) == FAILURE) {

        php_error_docref(NULL TSRMLS_CC, E_NOTICE, "Undefined variable: %s", varname);

        RETURN_NULL();

    }
```

```
    *return_value = **varvalue;
    zval_copy_ctor(return_value);
}

PHP_FUNCTION(hello_set_local_var)
{
    zval *newvar;
    char *varname;
    int varname_len;
    zval *value;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sz",
        &varname, &varname_len, &value) == FAILURE) {
        RETURN_NULL();
    }

    ALLOC_INIT_ZVAL(newvar);
    *newvar = *value;
    zval_copy_ctor(newvar);

    zend_hash_add(EG(active_symbol_table), varname, varname_len + 1,
        &newvar, sizeof(zval*), NULL);
```

```
    RETURN_TRUE;

}
```

下一步是什么?

在本教程—编写扩展系列的第二部分中，你学习了如何接收函数参数，创建并使用了数组，更重要的是了解了`zval`的内部运作方式。第 3 部分将关注资源数据类型并开始处理更复杂的数据结构。

编写扩展_III - 资源

扩展 教程

by Sara Golemon | Thursday, May 11, 2006

介绍

迄今为止，你已经处理了一些熟悉的概念，而且很容易就可以在在用户空间找到它们的对应物。在本教程中，你将深入一个更陌生的数据类型的内部运行机制，该类型对用户空间完全隐藏了内部细节，但是它的行为最终会让你有种似曾相识的感觉。

资源

PHP的`zval`可以描绘广泛的内部数据类型，然而有一种数据类型是它不能充分描绘的－脚本中的指针（pointer）。当指针引用的是不透明的`typedef`时，像值一样表示指针更加困难。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。

要使资源的标识符对`Zend`引擎有意义，必须先向PHP注册其底层的数据类型。你将从在*php_hello.h*中定义一个简单的数据结构开始。你可以把它放在几乎任何地方，但是，为了本次练习，把它放在`#define`语句后面，`PHP_MINIT_FUNCTION`声明前面。你也要定义一个常量，作为资源的名字，例如当调用`var_dump()`时会显示。

```
typedef struct _php_hello_person {
    char *name;
    int name_len;
    long age;
} php_hello_person;

#define PHP_HELLO_PERSON_RES_NAME "Person Data"
```

现在打开*hello.c*，在ZEND_DECLARE_MODULE_GLOBALS语句前面增加一个真正的全局整型数：

```
int le_hello_person;
```

在PHP扩展中，只有很少的几处需要声明真正的全局变量的地方，目录入口标识符（**List entry identifiers**）（le_*）是其中之一。这些值只是由一个查找表用来把资源类型和它们的文本名字以及析构方法相关联，它们不需要是线程安全的。你的扩展将在MINIT步骤为其导出的每个资源生成一个唯一编号。现在把它加入你的扩展，在PHP_MINIT_FUNCTION(hello)的顶部放入下面的代码：

```
le_hello_person = zend_register_list_destructors_ex(NULL, NULL,
    PHP_HELLO_PERSON_RES_NAME, module_number);
```

初始化资源

既然已经注册了资源，你需要用它做些事情。把下面的函数连同hello_functions结构中的匹配项加入*hello.c*，对应的原型声明放入*php_hello.h*：

```
PHP_FUNCTION(hello_person_new)

{

    php_hello_person *person;

    char *name;

    int name_len;

    long age;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sl",
        &name, &name_len, &age) == FAILURE) {

        RETURN_FALSE;

    }



    if (name_len < 1) {

        php_error_docref(NULL TSRMLS_CC, E_WARNING,
```

```
            "No name given, person resource not created.");

        RETURN_FALSE;

    }



    if (age < 0 || age > 255) {

        php_error_docref(NULL TSRMLS_CC, E_WARNING,
            "Nonsense age (%d) given, person resource not created.", age);

        RETURN_FALSE;

    }



    person = emalloc(sizeof(php_hello_person));

    person->name = estrndup(name, name_len);

    person->name_len = name_len;

    person->age = age;



    ZEND_REGISTER_RESOURCE(return_value, person, le_hello_person);

}
```

在分配内存和复制数据以前，该函数对要传入资源的数据进行了一些健壮性检查：是否提供了名字？这个人的年龄是否超出了人类的寿命范围？当然，延缓衰老的研究可能使年龄的数据类型（及其健壮性检查限制）在某天产生类似千年虫的问题，但是假定没人会超过255岁在任何时候都应当是安全的。

一旦函数对入口条件感到满意，所要做的就是分配一些内存并放入它的数据。最后把新注册的资源装入`return_value`。这个函数（指ZEND_REGISTER_RESOURCE－译注）不需要了解数据结构的内部情况；它只需要知道数据的指针地址和相关联的资源类型。

接收资源作为函数参数

从本系列之前的教程开始，你已经知道了如何使用`zend_parse_parameters()` 接收资源参数。现在是时候用它从给定的资源中获取数据了。把下一个函数加入扩展：

```
PHP_FUNCTION(hello_person_greet)
```

```
{
    php_hello_person *person;
    zval *zperson;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r", &zperson) == FAILURE) {
        RETURN_FALSE;
    }

    ZEND_FETCH_RESOURCE(person, php_hello_person*, &zperson, -1, \
        PHP_HELLO_PERSON_RES_NAME, le_hello_person);

    php_printf("Hello ");
    PHPWRITE(person->name, person->name_len);
    php_printf("!\nAccording to my records, you are %d years old.\n", person->age);

    RETURN_TRUE;
}
```

此处功能上的重要部分应该很容易理解。ZEND_FETCH_RESOURCE()需要一个变量来放入指针值。它也需要了解变量的内部类型，以及从哪儿得到资源标识符。

该函数调用中的-1表明利用&zperson标识资源，是种可选方式。如果这儿提供了-1以外的数字值，Zend引擎将尝试使用该编号标识资源，而不是使用zval*参数的数据。如果传入的资源与最后一个参数指定的资源类型不匹配，将会利用倒数第二个参数给出的资源名产生一个错误消息。

skin资源的方式不止一个。实际上下面的四个代码块具有同样的效果：

```
ZEND_FETCH_RESOURCE(person, php_hello_person *, &zperson, -1,\
    PHP_HELLO_PERSON_RES_NAME, le_person_name);
```

```
ZEND_FETCH_RESOURCE(person, php_hello_person *, NULL, Z_LVAL_P(zperson), \
    PHP_HELLO_PERSON_RES_NAME, le_person_name);



person = (php_hello_person *) zend_fetch_resource(&zperson TSRMLS_CC, -1, \
    PHP_HELLO_PERSON_RES_NAME, NULL, 1, le_person_name);

ZEND_VERIFY_RESOURCE(person);



person = (php_hello_person *) zend_fetch_resource(&zperson TSRMLS_CC, -1, \
    PHP_HELLO_PERSON_RES_NAME, NULL, 1, le_person_name);

if (!person) {

    RETURN_FALSE;

}
```

对于不在PHP_FUNCTION()中的情形，最后一对的形式非常有用，因此没有为return_value赋值；或者当出现资源类型不匹配这种非常合理的（原因），并且不想只返回FALSE时。

无论选择哪种方法从参数中获取你的资源数据，结果都是一样的。现在你有一个熟悉的C结构，可以使用同其他C程序完全一样的方式访问它。此时该结构仍然“属于”资源变量，所以你的函数不应该释放指针或是在退出前改变引用计数。那么资源如何被销毁呢？

销毁资源

PHP中大多数创建资源的函数都有对应的函数用于释放资源。例如，mysql_connect()有mysql_close()，mysql_query()有mysql_free_result()，fopen()有fclose()，诸如此类。或许经验告诉你如果只是unset()含有资源值的变量，那么，不论它们附有什么样的真实资源，也都会被释放／关闭。例如：

```
<?php

$fp = fopen('foo.txt','w');
unset($fp);

?>
```

该代码片段的第一行打开一个用于写入的文件－*foo.txt*，并且把流资源赋给变量$fp。当第二行清除$fp时，PHP自动关闭文件－即使fclose()从未被调用。这是怎样做到的呢？

奥秘就在你在MINIT函数中调用的zend_register_resource()中。你传入的两个NULL参数对应清除（或dtor）函数。第一个用于普通资源，第二个用于持久资源。我们现在先关注普通资源，稍后再回到持久资源，但是它们在常规语义上是相同的。像下面一样修改行zend_register_resource：

```
le_hello_person = zend_register_list_destructors_ex(php_hello_person_dtor, NULL,
    PHP_HELLO_PERSON_RES_NAME, module_number);
```

并且在紧邻MINIT方法的上面创建新函数：

```
static void php_hello_person_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC)
{
    php_hello_person *person = (php_hello_person*)rsrc->ptr;

    if (person) {
        if (person->name) {
            efree(person->name);
        }
        efree(person);
    }
}
```

如你所见，只是释放了曾分配并关联于资源的缓冲。当持有你的资源引用的最后一个用户变量超出作用域，该函数将被自动调用以使你的扩展能够释放内存、从远程主机断开或执行其他的最终清理。

强制销毁资源

如果对资源的dtor函数的调用依赖于所有指向它的变量超出作用域，那么类似fclose()或mysql_free_result()这样的函数是怎样在资源的引用仍然存在的情况下完成任务的？回答之前，我希望你实验下面的代码：

```
<?php
$fp = fopen('test', 'w');
```

```
var_dump($fp);
fclose($fp);
var_dump($fp);

?>
```

两次调用var_dump()都能看到资源编码的数值，由此知道资源的引用仍然存在；但是第二次调用var_dump()表明其类型是“unknown”。这是因为Zend引擎在内存中维护的资源查找表不再包含匹配那个编码的文件句柄－所以任何利用那个编码执行 ZEND_FETCH_RESOURCE()的尝试都将失败。

如同很多其他基于资源的函数，fclose()通过使用zend_list_delete()实现这个（目的）。或许明显，或许不明显，该函数从特定的资源表中删除一项。其最简单的应用可能是：

```
PHP_FUNCTION(hello_person_delete)

{

    zval *zperson;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r", &zperson) == FAILURE) {

        RETURN_FALSE;

    }



    zend_list_delete(Z_LVAL_P(zperson));

    RETURN_TRUE;

}
```

当然，这个函数将销毁*任意*资源类型，不管它是我们的person资源、文件句柄、MySQL连接或其他的什么。为了避免给其他的扩展造成潜在的问题以及使得用户空间代码难于调试，首先检验资源类型是好习惯。这很容易做到，利用ZEND_FETCH_RESOURCE()取得资源放入哑变量。继续前进，把它加入你的函数，介于zend_parse_parameters()调用和zend_list_delete()调用之间。

持久资源

如果你用过mysql_pconnect()、popen()或任何其他持久的资源类型，那么你将知道，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。

标准资源和持久资源的两个关键区别是注册时dtor函数的安排，以及使用pemalloc()分配数据而不是emalloc()。

让我们为person资源创建一个能保持持久性的版本。先向MINIT增加另一个zend_register_resource()代码行。不要忘记紧接着le_hello_person定

义le_hello_person_persist变量:

```
PHP_MINIT_FUNCTION(hello)

{

    le_hello_person = zend_register_list_destructor_ex(
                php_hello_person_dtor, NULL, PHP_HELLO_PERSON_RES_NAME, module_number);

    le_hello_person_persist = zend_register_list_destructor_ex (
                NULL, php_hello_person_persist_dtor, PHP_HELLO_PERSON_RES_NAME, module_number);

    ...
```

基本语法是一样的，但这次你在zend_register_resource()的第二个参数指定析构函数而非第一个。二者真正的区别是dtor何时被调用。传入第一个参数的dtor函数在活动请求关闭时被调用，而传入第二个参数的dtor函数直到模块在最终关闭被卸载时才被调用。

由于引用了一个新的资源dtor函数，你将需要定义它。把这个看似熟悉的方法加入*hello.c*中MINIT上面的某处就行:

```
static void php_hello_person_persist_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC)

{

    php_hello_person *person = (php_hello_person*)rsrc->ptr;



    if (person) {

        if (person->name) {

            pefree(person->name, 1);

    }

        pefree(person, 1);

    }

}
```

现在你需要一种方法例示持久版本的person资源。既有的约定是创建一个名字中以“p”作前缀的新函数。向你的扩展中加入该方法:

```
PHP_FUNCTION(hello_person_pnew)

{

    php_hello_person *person;

    char *name;

    int name_len;

    long age;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sl",
        &name, &name_len, &age) == FAILURE) {

        RETURN_FALSE;

    }



    if (name_len < 1) {

        php_error_docref(NULL TSRMLS_CC, E_WARNING, \
            "No name given, person resource not created.");

        RETURN_FALSE;

    }



    if (age < 0 || age > 255) {

        php_error_docref(NULL TSRMLS_CC, E_WARNING, \
            "Nonsense age (%d) given, person resource not created.", age);

        RETURN_FALSE;

    }
```

```
    person = pemalloc(sizeof(php_hello_person), 1);

    person->name = pemalloc(name_len + 1, 1);

    memcpy(person->name, name, name_len + 1);

    person->name_len = name_len;

    person->age = age;



    ZEND_REGISTER_RESOURCE(return_value, person, le_hello_person_persist);

}
```

如你所见，该函数与`hello_person_new()`仅有微小的不同。在实践中，你将看到这类成对的用户空间函数被典型地实现为围绕共同核心的封装函数。看看源代码中其他的资源创建函数对是如何避免这种重复的。

既然你的扩展将创建两类资源，因此需要能处理两种类型。幸运的是，`ZEND_FETCH_RESOURCE`有个姐妹函数胜任这个任务。用下面的代码替换`hello_person_greet()`中当前对`ZEND_FETCH_RESOURCE`的调用：

```
ZEND_FETCH_RESOURCE2(person, php_hello_person*, &zperson, -1, \
    PHP_HELLO_PERSON_RES_NAME , le_hello_person, le_hello_person_persist);
```

这将用合适的数据加载你的`person`变量，无论是否传入持久资源。

这两个`FETCH`宏调用允许你指定很多资源类型，但是也有罕见的需要多于两个的情况。为防万一，这是利用基本函数写的最后一条语句：

```
person = (php_hello_person*) zend_fetch_resource(&zperson TSRMLS_CC, -1, \
    PHP_HELLO_PERSON_RES_NAME, NULL, 2, le_hello_person, le_hello_person_persist);

    ZEND_VERIFY_RESOURCE(person);
```

这儿要注意两件重要的事情。首先，你能看到`FETCH_RESOURCE`宏自动尝试校验资源。展开来说，此种情况下宏`ZEND_VERIFY_RESOURCE`只是转换为：

```
if (!person) {

    RETURN_FALSE;

}
```

当然，你不会总是仅仅因为不能取到一个资源就要你的扩展函数退出，所以你能使用真实的`zend_fetch_resource()`函数尝试取得资源类型，然后应用自己的逻辑处理将返回的`NULL`值。

查找现存的持久资源

持久资源实际上只是相当于你重用它的能力。为了重用它，你需要安全的地方存储它。Zend引擎通过EG(persistent_list)执行器全局作用域（executor global）实现该（目的），它是个包含list_entry结构的HashTable，通常被引擎用于内部。依照下面修改hello_person_pnew()：

```
PHP_FUNCTION(hello_person_pnew)

{

    php_hello_person *person;

    char *name, *key;

    int name_len, key_len;

    long age;

    list_entry *le, new_le;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sl",
        &name, &name_len, &age) == FAILURE) {

        RETURN_FALSE;

    }



    if (name_len < 1) {

        php_error_docref(NULL TSRMLS_CC, E_WARNING, \
            "No name given, person resource not created.");

        RETURN_FALSE;

    }



    if (age < 0 || age > 255) {
```

```
        php_error_docref(NULL TSRMLS_CC, E_WARNING, \
            "Nonsense age (%d) given, person resource not created.", age);

        RETURN_FALSE;

    }


    /* 查找一个已建立的资源 */

    key_len = spprintf(&key, 0, "hello_person_%s_%d\n", name, age);

    if (zend_hash_find(&EG(persistent_list), key, key_len + 1, &le) == SUCCESS) {

        /* 已经存在该person的一个条目 */

        ZEND_REGISTER_RESOURCE(return_value, le->ptr, le_hello_person_persist);

        efree(key);

        return;

    }


    /* 新person，分配一个结构 */

    person = pemalloc(sizeof(php_hello_person), 1);

    person->name = pemalloc(name_len + 1, 1);

    memcpy(person->name, name, name_len + 1);

    person->name_len = name_len;

    person->age = age;


    ZEND_REGISTER_RESOURCE(return_value, person, le_hello_person_persist);
```

```
    /* 在持久列表中存储一个引用 */

    new_le.ptr = person;

    new_le.type = le_hello_person_persist;

    zend_hash_add(&EG(persistent_list), key, key_len + 1, &new_le, sizeof(list_entry), NULL);



    efree(key);

}
```

这个版本的`hello_person_pnew()`首先在`EG(persistent_list)`全局作用域中检测已经存在的`php_hello_person`结构，如果可用就用它而不是浪费时间和资源重新分配。如果还不存在，函数分配一个新的结构装入新数据，并且把该结构加入持久列表中。不论哪种方式，函数都在请求中给你留下一个注册为资源的新的结构。

用于存储指针的持久列表总是位于当前进程或线程中，因此可能同时查找同样数据的两个请求没有任何联系。如果一个进程故意关闭一个持久资源，PHP将处理它，从持久列表中删除那个资源的引用，以使未来的调用不会使用已释放的数据。

核对（代码）完整性

再一次，到本教程结束时你的扩展文件应该为:

config.m4

```
PHP_ARG_ENABLE(hello, [whether to enable Hello World support],

[ --enable-hello Enable Hello World support])



if test "$PHP_HELLO" = "yes"; then

  AC_DEFINE(HAVE_HELLO, 1, [Whether you have Hello World])

  PHP_NEW_EXTENSION(hello, hello.c, $ext_shared)

fi
```

php_hello.h

```
#ifndef PHP_HELLO_H

#define PHP_HELLO_H 1
```

```
#ifdef ZTS
#include "TSRM.h"
#endif

ZEND_BEGIN_MODULE_GLOBALS(hello)
    long counter;
    zend_bool direction;
ZEND_END_MODULE_GLOBALS(hello)

#ifdef ZTS
#define HELLO_G(v) TSRMG(hello_globals_id, zend_hello_globals *, v)
#else
#define HELLO_G(v) (hello_globals.v)
#endif

#define PHP_HELLO_WORLD_VERSION "1.0"
#define PHP_HELLO_WORLD_EXTNAME "hello"

typedef struct _php_hello_person {
    char *name;
    int name_len;
```

```
    long age;
} php_hello_person;


#define PHP_HELLO_PERSON_RES_NAME "Person Data"


PHP_MINIT_FUNCTION(hello);
PHP_MSHUTDOWN_FUNCTION(hello);
PHP_RINIT_FUNCTION(hello);


PHP_FUNCTION(hello_world);
PHP_FUNCTION(hello_long);
PHP_FUNCTION(hello_double);
PHP_FUNCTION(hello_bool);
PHP_FUNCTION(hello_null);
PHP_FUNCTION(hello_greetme);
PHP_FUNCTION(hello_add);
PHP_FUNCTION(hello_dump);
PHP_FUNCTION(hello_array);
PHP_FUNCTION(hello_array_strings);
PHP_FUNCTION(hello_array_walk);
PHP_FUNCTION(hello_array_value);
PHP_FUNCTION(hello_get_global_var);
```

```
PHP_FUNCTION(hello_set_local_var);

PHP_FUNCTION(hello_person_new);

PHP_FUNCTION(hello_person_pnew);

PHP_FUNCTION(hello_person_greet);

PHP_FUNCTION(hello_person_delete);



extern zend_module_entry hello_module_entry;

#define phpext_hello_ptr &hello_module_entry



#endif
```

hello.c

```
#ifdef HAVE_CONFIG_H

#include "config.h"

#endif



#include "php.h"

#include "php_ini.h"

#include "php_hello.h"



int le_hello_person;

int le_hello_person_persist;



ZEND_DECLARE_MODULE_GLOBALS(hello)
```

```
static function_entry hello_functions[] = {
    PHP_FE(hello_world, NULL)
    PHP_FE(hello_long, NULL)
    PHP_FE(hello_double, NULL)
    PHP_FE(hello_bool, NULL)
    PHP_FE(hello_null, NULL)
    PHP_FE(hello_greetme, NULL)
    PHP_FE(hello_add, NULL)
    PHP_FE(hello_dump, NULL)
    PHP_FE(hello_array, NULL)
    PHP_FE(hello_array_strings, NULL)
    PHP_FE(hello_array_walk, NULL)
    PHP_FE(hello_array_value, NULL)
    PHP_FE(hello_get_global_var, NULL)
    PHP_FE(hello_set_local_var, NULL)
    PHP_FE(hello_person_new, NULL)
    PHP_FE(hello_person_pnew, NULL)
    PHP_FE(hello_person_greet, NULL)
    PHP_FE(hello_person_delete, NULL)
    {NULL, NULL, NULL}
};
```

```
zend_module_entry hello_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_HELLO_WORLD_EXTNAME,
    hello_functions,
    PHP_MINIT(hello),
    PHP_MSHUTDOWN(hello),
    PHP_RINIT(hello),
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    PHP_HELLO_WORLD_VERSION,
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_HELLO
    ZEND_GET_MODULE(hello)
#endif
```

```
PHP_INI_BEGIN()

    PHP_INI_ENTRY("hello.greeting", "Hello World", PHP_INI_ALL, NULL)

    STD_PHP_INI_ENTRY("hello.direction", "1", PHP_INI_ALL, OnUpdateBool, \
        direction, zend_hello_globals, hello_globals)

PHP_INI_END()



static void php_hello_init_globals(zend_hello_globals *hello_globals)

{

    hello_globals->direction = 1;

}



PHP_RINIT_FUNCTION(hello)

{

    HELLO_G(counter) = 0;



    return SUCCESS;

}



static void php_hello_person_persist_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC)

{

    php_hello_person *person = (php_hello_person*)rsrc->ptr;



    if (person) {
```

```
        if (person->name) {
            pefree(person->name, 1);
        }
        pefree(person, 1);
    }
}


static void php_hello_person_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC)
{
    php_hello_person *person = (php_hello_person*)rsrc->ptr;

    if (person) {
        if (person->name) {
            efree(person->name);
        }
        efree(person);
    }
}

PHP_MINIT_FUNCTION(hello)
{
    le_hello_person = zend_register_list_destructors_ex(php_hello_person_dtor, NULL,
        PHP_HELLO_PERSON_RES_NAME, module_number);
```

```
    le_hello_person_persist = zend_register_list_destructors_ex (NULL,
        php_hello_person_persist_dtor, PHP_HELLO_PERSON_RES_NAME, module_number);


    ZEND_INIT_MODULE_GLOBALS(hello, php_hello_init_globals, NULL);


    REGISTER_INI_ENTRIES();


    return SUCCESS;
}


PHP_MSHUTDOWN_FUNCTION(hello)
{
    UNREGISTER_INI_ENTRIES();


    return SUCCESS;
}


PHP_FUNCTION(hello_world)
{
    RETURN_STRING("Hello World", 1);
}


PHP_FUNCTION(hello_long)
```

```
{
    if (HELLO_G(direction)) {
        HELLO_G(counter)++;
    } else {
        HELLO_G(counter)--;
    }

    RETURN_LONG(HELLO_G(counter));
}

PHP_FUNCTION(hello_double)
{
    RETURN_DOUBLE(3.1415926535);
}

PHP_FUNCTION(hello_bool)
{
    RETURN_BOOL(1);
}

PHP_FUNCTION(hello_null)
{
```

```
    RETURN_NULL();
}

PHP_FUNCTION(hello_greetme)
{
    zval *zname;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &zname) == FAILURE) {
        RETURN_NULL();
    }

    convert_to_string(zname);

    php_printf("Hello ");
    PHPWRITE(Z_STRVAL_P(zname), Z_STRLEN_P(zname));
    php_printf("\n");

    RETURN_TRUE;
}

PHP_FUNCTION(hello_add)
{
```

```
    long a;

    double b;

    zend_bool return_long = 0;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ld|b",
        &a, &b, &return_long) == FAILURE) {

        RETURN_NULL();

    }



    if (return_long) {

        RETURN_LONG(a + b);

    } else {

        RETURN_DOUBLE(a + b);

    }

}



PHP_FUNCTION(hello_dump)

{

    zval *uservar;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &uservar) == FAILURE) {

    RETURN_NULL();

    }
```

```
    switch (Z_TYPE_P(uservar)) {
    case IS_NULL:
        php_printf("NULL\n");
        break;
    case IS_BOOL:
        php_printf("Boolean: %s\n", Z_LVAL_P(uservar) ? "TRUE" : "FALSE");
        break;
    case IS_LONG:
        php_printf("Long: %ld\n", Z_LVAL_P(uservar));
        break;
    case IS_DOUBLE:
        php_printf("Double: %f\n", Z_DVAL_P(uservar));
        break;
    case IS_STRING:
        php_printf("String: ");
        PHPWRITE(Z_STRVAL_P(uservar), Z_STRLEN_P(uservar));
        php_printf("\n");
        break;
    case IS_RESOURCE:
        php_printf("Resource\n");
        break;
```

```
    case IS_ARRAY:
        php_printf("Array\n");
        break;
    case IS_OBJECT:
        php_printf("Object\n");
        break;
    default:
        php_printf("Unknown\n");
    }

    RETURN_TRUE;
}

PHP_FUNCTION(hello_array)
{
    char *mystr;
    zval *mysubarray;

    array_init(return_value);

    add_index_long(return_value, 42, 123);
    add_next_index_string(return_value, "I should now be found at index 43", 1);
```

```
    add_next_index_stringl(return_value, "I'm at 44!", 10, 1);
    mystr = estrdup("Forty Five");
    add_next_index_string(return_value, mystr, 0);
    add_assoc_double(return_value, "pi", 3.1415926535);

    ALLOC_INIT_ZVAL(mysubarray);
    array_init(mysubarray);

    add_next_index_string(mysubarray, "hello", 1);
    php_printf("mysubarray->refcount = %d\n", mysubarray->refcount);
    mysubarray->refcount = 2;
    php_printf("mysubarray->refcount = %d\n", mysubarray->refcount);
    add_assoc_zval(return_value, "subarray", mysubarray);
    php_printf("mysubarray->refcount = %d\n", mysubarray->refcount);
}

PHP_FUNCTION(hello_array_strings)
{
    zval *arr, **data;
    HashTable *arr_hash;
    HashPosition pointer;
    int array_count;
```

```
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a", &arr) == FAILURE) {
        RETURN_NULL();
    }

    arr_hash = Z_ARRVAL_P(arr);
    array_count = zend_hash_num_elements(arr_hash);

    php_printf("The array passed contains %d elements\n", array_count);

    for(zend_hash_internal_pointer_reset_ex(arr_hash, &pointer);
        zend_hash_get_current_data_ex(arr_hash, (void**) &data, &pointer) == SUCCESS;
        zend_hash_move_forward_ex(arr_hash, &pointer)) {

        zval temp;
        char *key;
        int key_len;
        long index;

        if (zend_hash_get_current_key_ex(arr_hash, &key, &key_len,
            &index, 0, &pointer) == HASH_KEY_IS_STRING) {
            PHPWRITE(key, key_len);
        } else {
            php_printf("%ld", index);
```

```
        }

        php_printf(" => ");

        temp = **data;
        zval_copy_ctor(&temp);
        convert_to_string(&temp);
        PHPWRITE(Z_STRVAL(temp), Z_STRLEN(temp));
        php_printf("\n");
        zval_dtor(&temp);
    }

    RETURN_TRUE;
}

static int php_hello_array_walk(zval **element TSRMLS_DC)
{
    zval temp;
    temp = **element;
    zval_copy_ctor(&temp);
    convert_to_string(&temp);
    PHPWRITE(Z_STRVAL(temp), Z_STRLEN(temp));
    php_printf("\n");
```

```
    zval_dtor(&temp);


    return ZEND_HASH_APPLY_KEEP;
}


static int php_hello_array_walk_arg(zval **element, char *greeting TSRMLS_DC)
{
    php_printf("%s", greeting);
    php_hello_array_walk(element TSRMLS_CC);


    return ZEND_HASH_APPLY_KEEP;
}


static int php_hello_array_walk_args(zval **element, int num_args,
    va_list args, zend_hash_key *hash_key)
{
    char *prefix = va_arg(args, char*);
    char *suffix = va_arg(args, char*);
    TSRMLS_FETCH();


    php_printf("%s", prefix);
    php_hello_array_walk(element TSRMLS_CC);
```

```
    php_printf("%s\n", suffix);


    return ZEND_HASH_APPLY_KEEP;
}


PHP_FUNCTION(hello_array_walk)
{
    zval *zarray;
    int print_newline = 1;


    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a", &zarray) == FAILURE) {
        RETURN_NULL();
    }


    zend_hash_apply(Z_ARRVAL_P(zarray), (apply_func_t)php_hello_array_walk TSRMLS_CC);
    zend_hash_internal_pointer_reset(Z_ARRVAL_P(zarray));
    zend_hash_apply_with_argument(Z_ARRVAL_P(zarray),
        (apply_func_arg_t)php_hello_array_walk_arg, "Hello " TSRMLS_CC);
    zend_hash_apply_with_arguments(Z_ARRVAL_P(zarray),
        (apply_func_args_t)php_hello_array_walk_args, 2, "Hello ", "Welcome to my extension!");


    RETURN_TRUE;
}
```

```
PHP_FUNCTION(hello_array_value)
{
    zval *zarray, *zoffset, **zvalue;
    long index = 0;
    char *key = NULL;
    int key_len = 0;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "az", &zarray, &zoffset) == FAILURE) {
        RETURN_NULL();
    }

    switch (Z_TYPE_P(zoffset)) {
    case IS_NULL:
        index = 0;
        break;
    case IS_DOUBLE:
        index = (long)Z_DVAL_P(zoffset);
        break;
    case IS_BOOL:
    case IS_LONG:
    case IS_RESOURCE:
        index = Z_LVAL_P(zoffset);
```

```
        break;
    case IS_STRING:
        key = Z_STRVAL_P(zoffset);
        key_len = Z_STRLEN_P(zoffset);
        break;
    case IS_ARRAY:
        key = "Array";
        key_len = sizeof("Array") - 1;
        break;
    case IS_OBJECT:
        key = "Object";
        key_len = sizeof("Object") - 1;
        break;
    default:
        key = "Unknown";
        key_len = sizeof("Unknown") - 1;
    }


    if (key && zend_hash_find(Z_ARRVAL_P(zarray), key, key_len + 1,
        (void**)&zvalue) == FAILURE) {
        php_error_docref(NULL TSRMLS_CC, E_NOTICE, "Undefined index: %s", key);
        RETURN_NULL();
    } else if (!key && zend_hash_index_find(Z_ARRVAL_P(zarray),
```

```
        index, (void**)&zvalue) == FAILURE) {

        php_error_docref(NULL TSRMLS_CC, E_NOTICE, "Undefined index: %ld", index);

        RETURN_NULL();

    }



    *return_value = **zvalue;

    zval_copy_ctor(return_value);

}



PHP_FUNCTION(hello_get_global_var)

{

    char *varname;

    int varname_len;

    zval **varvalue;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
        &varname, &varname_len) == FAILURE) {

        RETURN_NULL();

    }



    if (zend_hash_find(&EG(symbol_table), varname, varname_len + 1,
        (void**)&varvalue) == FAILURE) {

        php_error_docref(NULL TSRMLS_CC, E_NOTICE, "Undefined variable: %s", varname);
```

```
        RETURN_NULL();
    }


    *return_value = **varvalue;
    zval_copy_ctor(return_value);
}


PHP_FUNCTION(hello_set_local_var)
{
    zval *newvar;
    char *varname;
    int varname_len;
    zval *value;


    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sz",
        &varname, &varname_len, &value) == FAILURE) {
        RETURN_NULL();
    }


    ALLOC_INIT_ZVAL(newvar);
    *newvar = *value;
    zval_copy_ctor(newvar);
```

```
    zend_hash_add(EG(active_symbol_table), varname, varname_len + 1,
        &newvar, sizeof(zval*), NULL);



    RETURN_TRUE;

}



PHP_FUNCTION(hello_person_new)

{

    php_hello_person *person;

    char *name;

    int name_len;

    long age;



    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sl",
        &name, &name_len, &age) == FAILURE) {

        RETURN_FALSE;

    }



    if (name_len < 1) {

        php_error_docref(NULL TSRMLS_CC, E_WARNING,
            "No name given, person resource not created.");

        RETURN_FALSE;

    }
```

```
    if (age < 0 || age > 255) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
            "Nonsense age (%d) given, person resource not created.", age);
        RETURN_FALSE;
    }

    person = emalloc(sizeof(php_hello_person));
    person->name = estrndup(name, name_len);
    person->name_len = name_len;
    person->age = age;

    ZEND_REGISTER_RESOURCE(return_value, person, le_hello_person);
}

PHP_FUNCTION(hello_person_pnew)
{
    php_hello_person *person;
    char *name, *key;
    int name_len, key_len;
    long age;
    list_entry *le, new_le;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "sl",
```

```
        &name, &name_len, &age) == FAILURE) {

        RETURN_FALSE;

    }



    if (name_len < 1) {

        php_error_docref(NULL TSRMLS_CC, E_WARNING,
            "No name given, person resource not created.");

        RETURN_FALSE;

    }



    if (age < 0 || age > 255) {

        php_error_docref(NULL TSRMLS_CC, E_WARNING,
            "Nonsense age (%d) given, person resource not created.", age);

        RETURN_FALSE;

    }



    /* 查找一个已建立的资源 */

    key_len = spprintf(&key, 0, "hello_person_%s_%d\n", name, age);

    if (zend_hash_find(&EG(persistent_list), key, key_len + 1, &le) == SUCCESS) {

        /* 已经存在该person的一个条目 */

        ZEND_REGISTER_RESOURCE(return_value, le->ptr, le_hello_person_persist);

        efree(key);

        return;
```

```
    }

    /* 新person，分配一个结构 */
    person = pemalloc(sizeof(php_hello_person), 1);
    person->name = pemalloc(name_len + 1, 1);
    memcpy(person->name, name, name_len + 1);
    person->name_len = name_len;
    person->age = age;

    ZEND_REGISTER_RESOURCE(return_value, person, le_hello_person_persist);

    /* 在持久列表中存储一个引用 */
    new_le.ptr = person;
    new_le.type = le_hello_person_persist;
    zend_hash_add(&EG(persistent_list), key, key_len + 1,
        &new_le, sizeof(list_entry), NULL);

    efree(key);
}

PHP_FUNCTION(hello_person_greet)
{
    php_hello_person *person;
```

```
    zval *zperson;


    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r", &zperson) == FAILURE) {
        RETURN_FALSE;
    }


    ZEND_FETCH_RESOURCE2(person, php_hello_person*, &zperson, \
        -1, PHP_HELLO_PERSON_RES_NAME, le_hello_person, le_hello_person_persist);


    php_printf("Hello ");
    PHPWRITE(person->name, person->name_len);
    php_printf("!\nAccording to my records, you are %d years old.\n", person->age);


    RETURN_TRUE;
}


PHP_FUNCTION(hello_person_delete)
{
    zval *zperson;


    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r", &zperson) == FAILURE) {
        RETURN_FALSE;
    }
```

```
    zend_list_delete(Z_LVAL_P(zperson));

    RETURN_TRUE;

}
```

总结

在本教程—编写扩展系列的第三部分中，你学习了把任意的、有时不透明的数据放入PHP用户空间变量的几个简单的必需步骤。在后面的步骤中，你将通过连接第三方库来组合这些技术，以创建在PHP中常看到的粘合库。

在第四部分，我们将关注对象—从PHP 4中可用的简单的带函数的数组，到PHP 5可用的更复杂的重载的OOP结构。