Zend API：深入 PHP 内核

by yAnbiN ben.yan@msn.com

（二）摘要

摘要

知者不言，言者不知。

——老子《道德经》五十六章

有时候，单纯依靠 PHP“本身”是不行的。尽管普通用户很少遇到这种情况，但一些专业性的应用则经常需要将 PHP 的性能发挥到极致（这里的性能是指速度或功能）。由于受到 PHP 语言本身的限制，同时还可能不得不把庞大的库文件包含到每个脚本当中，因此，某些新功能并不是总能被顺利实现，所以我们必须另外寻找一些方法来克服 PHP 的这些缺点。

了解到了这一点，我们就应该接触一下 PHP 的心脏并探究一下它的内核——可以编译成 PHP 并让之工作的 C 代码——的时候了。

（三）概述

概述

“扩展 PHP”说起来容易做起来难。PHP 现在已经发展成了一个具有数兆字节源代码的非常成熟的系统。要想深入这样的一个系统，有很多东西需要学习和考虑。在写这一章节的时候，我们最终决定采用“边学边做”的方式。这也许并不是最科学和专业的方式，但却应该是最有趣和最有效的一种方式。在下面的小节里，你首先会非常快速的学习到如何写一个虽然很基础但却能立即运行的扩展，然后将会学习到有关 Zend API 的高级功能。另外一个选择就是将其作为一个整体，一次性的讲述所有的这些操作、设计、技巧和诀窍等，并且可以让我们在实际动手前就可以得到一副完整的愿景。这看起来似乎是一个更好的方法，也没有死角，但它却枯燥无味、费时费力，很容易让人感到气馁。这就是我们为什么要采用非常直接的讲法的原因。

注意，尽管这一章会尽可能多讲述一些关于 PHP 内部工作机制的知识，但要想真的给出一份在任何时间任何情况下的 PHP 扩展指南，那简直是不可能的。PHP 是如此庞大和复杂，以致于只有你亲自动手实践一下才有可能真正理解它的内部工作机制，因此我们强烈推荐你随时参考它的源代码来进行工作。

Zend 是什么？ PHP 又是什么？

Zend 指的是语言引擎，PHP 指的是我们从外面看到的一套完整的系统。这听起来有点糊涂，但其实并不复杂（见图 3-1 PHP 内部结构图）。为了实现一个 WEB 脚本的解释器，你需要完成以下三个部分的工作：

1、 解释器部分，负责对输入代码的分析、翻译和执行；
2、 功能性部分，负责具体实现语言的各种功能（比如它的函数等等）；
3、 接口部分，负责同 WEB 服务器的会话等功能。

Zend 包括了第一部分的全部和第二部分的局部，PHP 包括了第二部分的局部和第三部分的全部。他们合起来称之为 PHP 包。Zend 构成了语言的核心，同时也包含了一些最基本的 PHP 预定义函数的实现。PHP 则包含了所有创造出语言本身各种显著特性的模块。

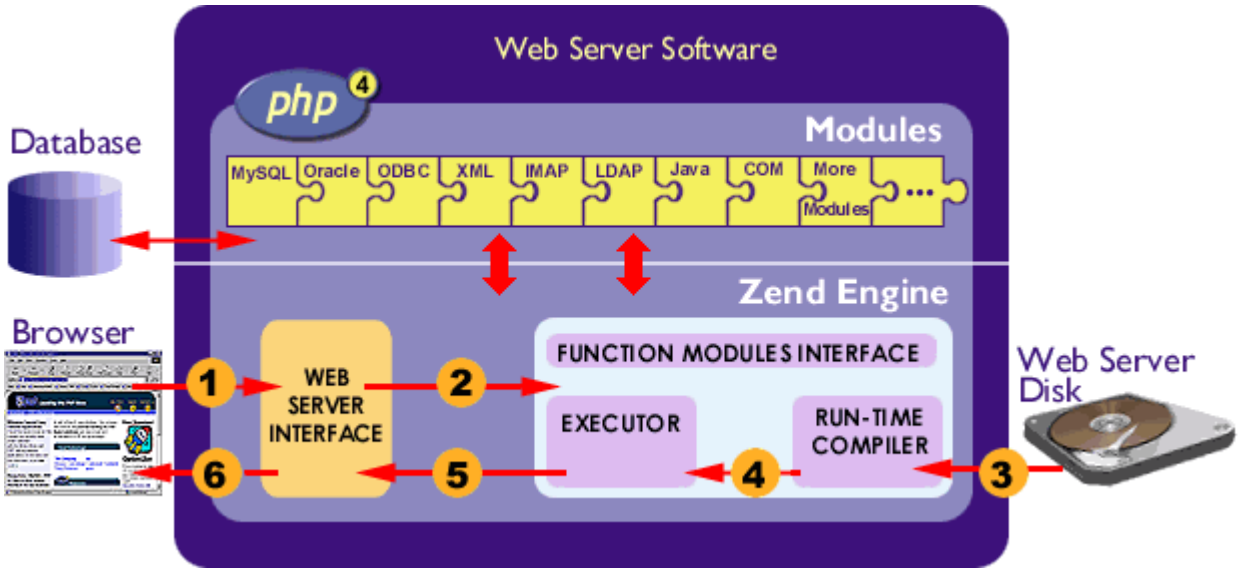


图 3-1　PHP 内部结构图

下面将要讨论 PHP 允许在哪里扩展以及如何扩展。

（四）可扩展性

正如上图（图 3-1 PHP 内部结构图）所示，PHP 主要以三种方式来进行扩展：外部模块，内建模块各 Zend 引擎。下面我们将分别讨论这些方式：

外部模块

外部模块可以在脚本运行时使用 dl() 函数载入。这个函数从磁盘载入一个共享对象并将它的功能与调用该函数的脚本进行绑定并使之生效。脚本终止后，这个外部模块将在内存中被丢弃。这种方式有利有弊，如下表所示：

优点	缺点
外部模块不需要重新对 PHP 进行编译。	共享对象在每次脚本调用时都需要对其进行加载，速度较慢。
PHP 通过“外包”方式来让自身的体积保持很小。	附加的外部模块文件会让磁盘变得比较散乱。
	每个想使用该模块功能的脚本都必须使用 dl() 函数手动加载，或者在 php.ini 文件当中添加一些扩展标签（这并不总是一个恰当的解决方案）。

综上所述，外部模块非常适合开发第三方产品，较少使用的附加的小功能或者仅仅是调试等这些用途。为了迅速开发一些附加功能，外部模块是最佳方式。但对于一些经常使用的、实现较大的，代码较为复杂的应用，那就有些得不偿失了。

第三方可能会考虑在 php.ini 文件中使用扩展标签来创建一个新的外部模块。这些外部模块完全同主 PHP 包分离，这一点非常适合应用于一些商业环境。商业性的发行商可以仅发送这些外部模块而不必再额外创建那些并不允许绑定这些商业模块的 PHP 二进制代码。

内建模块

内建模块被直接编译进 PHP 并存在于每一个 PHP 处理请求当中。它们的功能在脚本开始运行时立即生效。和外部模块一样，内建模块也有各有利弊，列表如下：

优点	缺点
无需专门手动载入，功能即时生效。	修改内建模块时需要重新编译 PHP。
无需额外的磁盘文件，所有功能均内置在 PHP 二进制代码当中。	PHP 二进制文件会变大并且会消耗更多的内存。

Zend 引擎

当然，你也能直接在 Zend 引擎里面进行扩展。如果你需要在语言特性方面做些改动或者是需要在语言核心内置一些特别的功能，那么这就是一种很好的方式。但一般情况下应该尽力避免对 Zend 引擎的修改。这里面的改动会导致和其他代码的不兼容，而且几乎没有人会适应打过特殊补丁的 Zend 引擎。况且这些改动与主 PHP 源代码是不可分割的，因此就有可能在下一次的官方的源代码更新中被覆盖掉。因此，这种方式通常被认为是“不良的习惯”。由于使用极其稀少，本章将不再对此进行赘述。

（五）源码布局

在我们开始讨论具体编码这个话题前，你应该让自己熟悉一下 PHP 的源码树以便可以迅速地对各个源文件进行定位。这也是编写和调试 PHP 扩展所必须具备的一种能力。

下表列出了一些主要目录的内容：

目录	内容
php-src	包含了 PHP 主源文件和主头文件；在这里你可以找到所有的 PHP API 定

	义、宏等内容。(重要). 其他的一些东西你也可以在这里找到。
php-src/ext	这里是存放动态和内建模块的仓库；默认情况下，这些就是被集成于主源码树中的“官方” PHP 模块。自 PHP 4.0 开始，这些 PHP 标准扩展都可以编译为动态可载入的模块。(至少这些是可以的)。
php-src/main	这个目录包含主要的 PHP 宏和定义。 (重要)
php-src/pear	这个目录就是“PHP 扩展与应用仓库”的目录。包含了 PEAR 的核心文件。
php-src/sapi	包含了不同服务器抽象层的代码。
TSRM	Zend 和 PHP 的 “线程安全资源管理器” (TSRM) 目录。
ZendEngine2	包含了 Zend 引擎文件；在这里你可以找到所有的 Zend API 定义与宏等。(重要)

当然，讨论 PHP 包里面全部每一个文件无疑是超出了本章的范围，但你还是应该仔细看一下下面的几个文件：

- php-src/main/php.h, 位于 PHP 主目录。这个文件包含了绝大部分 PHP 宏及 API 定义。
- php-src/Zend/zend.h, 位于 Zend 主目录。这个文件包含了绝大部分 Zend 宏及 API 定义。
- php-src/Zend/zend_API.h, 也位于 Zend 主目录，包含了 Zend API 的定义。

除此之外，你也应该注意一下这些文件所包含的一些文件。举例来说，哪些文件与 Zend 执行器有关，哪些文件又为 PHP 初始化工作提供了支持等等。在阅读完这些文件之后，你还可以花点时间再围绕 PHP 包来看一些文件，了解一下这些文件和模块之间的依赖性——它们之间是如何依赖于别的文件又是如何为其他文件提供支持的。同时这也可以帮助你适应一下 PHP 创作者们代码的风格。要想扩展 PHP，你应该尽快适应这种风格。

扩展规范

Zend 是用一些特定的规范构建的。为了避免破坏这些规范，你应该遵循以下的几个规则：

宏

几乎对于每一项重要的任务，Zend 都预先提供了极为方便的宏。在下面章节的图表里将会描述到大部分基本函数、结构和宏。这些宏定义大多可以在 Zend.h 和 Zend_API.h 中找到。我们建议您在学习完本节之后仔细看一下这些文件。（当然你也可以现在就阅读这些文件，但你可能不会留下太多的印象。）

内存管理

资源管理仍然是一个极为关键的问题，尤其是对服务器软件而言。资源里最具宝贵的则非内存莫属了，内存管理也必须极端小心。内存管理在 Zend 中已经被部分抽象，而且你也应该坚持使用这些抽象，原因显而易见：由于得以抽象，Zend 就可以完全控制内存的分配。Zend 可以确定一块内存是否在使用，也可以自动释放未使用和失去引用的内存块，因此就可以避免内存泄漏。下表列出了一些常用函数：

函数	描述
emalloc()	用于替代 malloc()。
efree()	用于替代 free()。
estrdup()	用于替代 strdup()。
estrndup()	用于替代 strndup()。速度要快于 estrdup() 而且是二进制安全的。如果你在复制之前预先知道这个字符串的长度那就推荐你使用这个函数。
ecalloc()	用于替代 calloc()。
erealloc()	用于替代 realloc()。

emalloc(), estrdup(), estrndup(), ecalloc(), 和 erealloc() 用于申请内部的内存，efree() 则用来释放这些前面这些函数申请的内存。e*() 函数所用到的内存仅对当前本地的处理请求有效，并且会在脚本执行完毕，处理请求终止时被释放。

Zend 还有一个线程安全资源管理器，这可以为多线程 WEB 服务器提供更好的本地支持。不过这需要你为所有的全局变量申请一个局部结构来支持并发线程。但是因为在写本章内容时 Zend 的线程安全模式仍未完成，因此我们无法过多地涉及这个话题。

目录与文件函数

下列目录与文件函数应该在 Zend 模块内使用。它们的表现和对应的 C 语言版本完全一致，只是在线程级提供了虚拟目录的支持。

Zend 函数	对应的 C 函数
V_GETCWD()	getcwd()
V_FOPEN()	fopen()
V_OPEN()	open()
V_CHDIR()	chdir()
V_GETWD()	getwd()
V_CHDIR_FILE()	将当前的工作目录切换到一个以文件名为参数的该文件所在的目录。
V_STAT()	stat()
V_LSTAT()	lstat()

字符串处理

在 Zend 引擎中，与处理诸如整数、布尔值等这些无需为其保存的值而额外申请内存的简单类型不同，如果你想从一个函数返回一个字符串，或往符号表新建一个字符串变量，或做其他类似的事情，那你就必须确认是否已经使用上面的 e*() 等函数为这些字符串申请内存。（你可能对此没有多大的感觉。无所谓，现在你只需在脑子里有点印象即可，我们稍后就会再次回到这个话题）

复杂类型

像数组和对象等这些复杂类型需要另外不同的处理。它们被出存在哈希表中，Zend 提供了一些简单的 API 来操作这些类型。

（六）自动构建系统

PHP 提供了一套非常灵活的自动构建系统（automatic build system），它把所有的模块均放在 Ext 子目录下。每个模块除自身的源代码外，还都有一个用来配置该扩展的 config.m4 文件（详情请参见 http://www.gnu.org/software/m4/manual/m4.html ）。

包括 .cvsignore 在内的所有文件都是由位于 Ext 目录下的 ext_skel 脚本自动生成的，它的参数就是你想创建模块的名称。这个脚本会创建一个与模块名相同的目录，里面包含了与该模块对应的一些的文件。

下面是操作步骤：

:~/cvs/php4/ext:> ./ext_skel –extname=my_module
Creating directory my_module
Creating basic files: config.m4 .cvsignore my_module.c php_my_module.h CREDITS EXPERIMENTAL tests/001.phpt my_module.php [done].

To use your new extension, you will have to execute the following steps:

1. $ cd ..
2. $ vi ext/my_module/config.m4
3. $ ./buildconf
4. $ ./configure –[with|enable]-my_module
5. $ make
6. $ ./php -f ext/my_module/my_module.php
7. $ vi ext/my_module/my_module.c
8. $ make

Repeat steps 3-6 until you are satisfied with ext/my_module/config.m4 and
step 6 confirms that your module is compiled into PHP. Then, start writing
code and repeat the last two steps as often as necessary.

这些指令就会生成前面所说的那些文件。为了能够在自动配置文件和构建程序中包含新增加的模块，你还需要再运行一次 buildconf 命令。这个命令会通过搜索 Ext 目录和查找所有 config.m4 文件来重新生成 configure 脚本。默认情况下的的 config.m4 文件如例 3-1 所示，看起来可能会稍嫌复杂：

例 3.1 默认的 config.m4 文件

dnl $Id: build.xml,v 1.1 2005/08/21 16:27:06 goba Exp $
dnl config.m4 for extension my_module

```
dnl Comments in this file start with the string 'dnl'.
dnl Remove where necessary. This file will not work
dnl without editing.

dnl If your extension references something external, use with:

dnl PHP_ARG_WITH(my_module, for my_module support,
dnl Make sure that the comment is aligned:
dnl [   –with-my_module               Include my_module support])

dnl Otherwise use enable:

dnl PHP_ARG_ENABLE(my_module, whether to enable my_module support,
dnl Make sure that the comment is aligned:
dnl [   –enable-my_module             Enable my_module support])

if test "$PHP_MY_MODULE" != "no"; then
 dnl Write more examples of tests here...

 dnl # –with-my_module -> check with-path
 dnl SEARCH_PATH="/usr/local /usr"       # you might want to change this
 dnl SEARCH_FOR="/include/my_module.h"   # you most likely want to change this
 dnl if test -r $PHP_MY_MODULE/; then # path given as parameter
 dnl    MY_MODULE_DIR=$PHP_MY_MODULE
 dnl else # search default path list
 dnl    AC_MSG_CHECKING([for my_module files in default path])
 dnl    for i in $SEARCH_PATH ; do
 dnl      if test -r $i/$SEARCH_FOR; then
 dnl        MY_MODULE_DIR=$i
 dnl        AC_MSG_RESULT(found in $i)
 dnl      fi
 dnl    done
 dnl fi
 dnl
 dnl if test -z "$MY_MODULE_DIR"; then
 dnl    AC_MSG_RESULT([not found])
 dnl    AC_MSG_ERROR([Please reinstall the my_module distribution])
 dnl fi

 dnl # –with-my_module -> add include path
 dnl PHP_ADD_INCLUDE($MY_MODULE_DIR/include)
```

```
dnl # –with-my_module -> chech for lib and symbol presence
dnl LIBNAME=my_module # you may want to change this
dnl LIBSYMBOL=my_module # you most likely want to change this

dnl PHP_CHECK_LIBRARY($LIBNAME,$LIBSYMBOL,
dnl [
dnl    PHP_ADD_LIBRARY_WITH_PATH($LIBNAME, $MY_MODULE_DIR/lib,
MY_MODULE_SHARED_LIBADD)
dnl    AC_DEFINE(HAVE_MY_MODULELIB,1,[ ])
dnl ],[
dnl    AC_MSG_ERROR([wrong my_module lib version or lib not found])
dnl ],[
dnl    -L$MY_MODULE_DIR/lib -lm -ldl
dnl ])
dnl
dnl PHP_SUBST(MY_MODULE_SHARED_LIBADD)

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

如果你不太熟悉 M4 文件（现在毫无疑问是熟悉 M4 文件的大好时机），那么就可能会有点糊涂。但是别担心，其实非常简单。

注意：凡是带有 dnl 前缀的都是注释，注释是不被解析的。

config.m4 文件负责在配置时解析 configure 的命令行选项。这就是说它将检查所需的外部文件并且要做一些类似配置与安装的任务。

默认的配置文件将会在 configure 脚本中产生两个配置指令：–with-my_module 和 –enable-my_module。当需要引用外部文件时使用第一个选项（就像用 –with-apache 指令来引用 Apache 的目录一样）。第二个选项可以让用户简单的决定是否要启用该扩展。不管你使用哪一个指令，你都应该注释掉另外一个。也就是说，如果你使用了–enable-my_module，那就应该去掉–with-my_module。反之亦然。

默认情况下，通过 ext_skel 创建的 config.m4 都能接受指令，并且会自动启用该扩展。启用该扩展是通过 PHP_EXTENSION 这个宏进行的。如果你要改变一下默认的情况，想让用户明确的使用 –enable-my_module 或 –with-my_module 指令来把扩展包含在 PHP 二进制文件当中，那么将 “if test "$PHP_MY_MODULE" != "no"”改为“if test "$PHP_MY_MODULE" == "yes"”即可。

```
if test "$PHP_MY_MODULE" == "yes"; then dnl
  Action.. PHP_EXTENSION(my_module, $ext_shared)
  fi
```

这样就会导致在每次重新配置和编译 PHP 时都要求用户使用 –enable-my_module 指令。

另外请注意在修改 config.m4 文件后需要重新运行 buildconf 命令。

（七）开始创建扩展

我们先来创建一个非常简单的扩展，这个扩展除了一个将其整形参数作为返回值的函数外几乎什么都没有。下面（“例 3-2 一个简单的扩展”）就是这个样例的代码：

例 3.2 一个简单的扩展

```
/* 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);
}
```

这段代码已经包含了一个完整的 PHP 模块。稍后我们会详细解释这段代码，现在让我们先讨论一下构建过程。（在我们讨论 API 函数前，这可以让心急的人先实验一下。）

模块的编译

模块的编译基本上有两种方法：

1、在 Ext 目录内使用“make” 机制，这种机制也可以编译出动态可加载模块。
2、手动编译源代码。

第一种方法明显受到人们的偏爱。自 PHP 4.0 以来，这也被标准化成了一个的复杂的构建过程。这种复杂性也导致了难于被理解这个缺点。在本章最后我们会更详细的讨论这些内容，但现在还是让我们使用默认的 make 文件吧。

第二种方法很适合那些（因为某种原因而）没有完整 PHP 源码树的或者是很喜欢敲键盘的人。虽然这些情况是比较罕见，但为了内容的完整性我们也会介绍一下这种方法。

使用 make 进行编译

为了能够使用这种标准机制流程来编译这些代码，让我们把它所有的子目录都复制到 PHP 源码树的 Ext 目录下。然后运行 buildconf 命令，这将会创建一个新的包含了与我们的扩展相对应的选项的 configure 脚本。默认情况下，样例中的所有代码都是未激活的，因此你不用担心会破坏你的构建程序。在 buildconf 执行完毕后，再使用 configure –help 命令就会显示出下面的附加模块：

```
–enable-array_experiments     BOOK: Enables array experiments
–enable-call_userland         BOOK: Enables userland module
–enable-cross_conversion      BOOK: Enables cross-conversion module
```

–enable-first_module	BOOK: Enables first module
–enable-infoprint	BOOK: Enables infoprint module
–enable-reference_test	BOOK: Enables reference test module
–enable-resource_test	BOOK: Enables resource test module
–enable-variable_creation	BOOK: Enables variable-creation module

前面样例（“例 3-2 一个简单的扩展”）中的模块（first_module）可以使用–enable-first_module 或 –enable-first_module=yes 来激活。

手动编译

手动编译需要运行以下命令：

动作	命令
编译	cc -fpic -DCOMPILE_DL=1 -I/usr/local/include -I. -I.. -I../Zend -c -o <your_object_file> <your_c_file>
连接	cc -shared -L/usr/local/lib -rdynamic -o <your_module_file> <your_object_file(s)>

编译命令只是简单的让编译器产生一些中间代码(不要忽略了-fpic 参数)，然后又定义了 COMPILE_DL 常量来通知代码这是要编译为一个动态可加载的模块（通常用来测试，我们稍后会讨论它）。这些选项后面是一些编译这些源代码所必须包含的库文件目录。

注意：本例中所有 include 的路径都是都是 Ext 目录的相对路径。如果您是在其他目录编译的这些源文件，那么还要相应的修改路径名。编译所需要的目录有 PHP 目录，Zend 目录和模块所在的目录（如果有必要的话）。

连接命令也是一个非常简单的把模块连接成一个动态模块的命令。

你可以在编译指令中加入优化选项，尽管这些已经在样例中忽略了（不过你还是可以从前面讨论的 make 模版文件中发现一些）。

注意，手动将模块静态编译和连接到 PHP 二进制代码的指令很长很长，因此我们在这里不作讨论。（手动输入那些指令是很低效的。）

（八）使用扩展

根据你所选择的不同的构建过程，你要么把扩展编译进一个新的 PHP 的二进制文件，然后再连接到 Web 服务器（或以 CGI 模式运行），要么将其编译成一个 .so （共享库）文件。如果你将上面的样例文件 first_module.c 编译成了一个共享库，那么编译后的文件应该是 first_module.so。要想使用它，你就必须把他复制到一个 PHP 能访问到的地方。如果仅仅是为了测试的话，简单起见，你可以把它复制到你的 htdocs 目录下，然后用“例 3.3 first_module.so 的一个测试文件”中的代码来进行一下测试。如果

你将其直接编译编译进 PHP 二进制文件的话，那就不用调用 **dl()** 函数了，因为这个模块的函数在脚本一开始运行就生效了。

警告:

为了安全起见，你不应该将你的动态模块放入一个公共目录。即使是一个简单的测试你可以那么做，那也应该把它放进产品环境中的一个隔离的目录。

例 3.3 first_module.so 的一个测试文件

```
<?php

// remove next comment if necessary
// dl("first_module.so");
$param = 2;
$return = first_module($param);

print("We sent '$param' and got '$return'");

?>
```

调用这个测试文件，结果应该输出为：We sent '2' and got '2'。

若有需要，你可以调用 **dl()** 函数来载入一个动态可加载模块。这个函数负责寻找指定的共享库并进行加载使其函数在 PHP 中生效。这个样例模块仅输出了一个函数 **first_module()**，这个函数仅接受一个参数，并将其转换为整数作为函数的结果返回。

如果你已经进行到了这一步，那么，恭喜你，你已经成功创建了你的第一个 PHP 扩展！

（九）故障处理

实际上，在对静态或动态模块进行编译时没有太多故障处理工作要做。唯一可能的问题就是编译器会警告说找不到某些定义或者类似的事情。如果出现这种情况，你应该确认一下所有的头文件都是可用的并且它们的路径都已经在编译命令中被指定。为了确保每个文件都能被正确地定位，你可以先提取一个干净的 PHP 源码树，然后在 Ext 目录使用自动构建工具来创建这些文件。用这种方法就可以确保一个安全的编译环境。假如这样也不行，那就只好试试手动编译了。

PHP 也可能会警告说在你的模块里面有一些未定义的函数。（如果你没有改动样例文件的话这种情况应该不会发生。）假如你在模块中拼错了一些你想访问的外部函数的名字，那么它们就会在符号表中显示为“未能连接的符号”。这样在 PHP 动态加载或连接时，它们就不会运行——在二进制文件中没有相应的符号。为了解决这个问题，你可以在你的模块文件中找一下错误的声明或外部引用。注意，这个问题仅仅发生在动态可加载模块身上。而在静态模块身上则不会发生，因为静态模块在编译时就会抛出这些错误。

（十）关于模块代码的讨论

OK，现在你已经有了一个安全的构建环境，也可以把模块编译进 PHP 了。那么，现在就让我们开始详细讨论一下这里面究竟是如何工作的吧～

模块结构

所有的 PHP 模块通常都包含以下几个部分：

- 包含头文件(引入所需要的宏、API 定义等)；
- 声明导出函数(用于 Zend 函数块的声明)；
- 声明 Zend 函数块；
- 声明 Zend 模块；
- 实现 **get_module()** 函数；
- 实现导出函数。

包含头文件

模块所必须包含的头文件仅有一个 php.h，它位于 main 目录下。这个文件包含了构建模块时所必需的各种宏和 API 定义。

*小提示：*专门为模块创建一个含有其特有信息的头文件是一个很好的习惯。这个头文件应该包含 php.h 和所有导出函数的定义。如果你是使用 ext_skel 来创建模块的话，那么你可能已经有了这个文件，因为这个文件会被 ext_skel 自动生成。

声明导出函数

为了声明导出函数（也就是让其成为可以被 PHP 脚本直接调用的原生函数），Zend 提供了一个宏来帮助完成这样一个声明。代码如下：

```
ZEND_FUNCTION ( my_function );
```

ZEND_FUNCTION 声明了一个使用 Zend 内部 API 来编译的新的 C 函数。这个 C 函数是 void 类型，以 INTERNAL_FUNCTION_PARAMETERS （这是另一个宏）为参数，而且函数名字以 zif_ 为前缀。把上面这句声明展开可以得到这样的代码：

```
void zif_my_function ( INTERNAL_FUNCTION_PARAMETERS );
```

接着再把 INTERNAL_FUNCTION_PARAMETERS 展开就会得到这样一个结果：

```
void zif_my_function( int ht
                    , zval * return_value
                    , zval * this_ptr
```

```
, int return_value_used
, zend_executor_globals * executor_globals
);
```

在解释器（interpreter）和执行器（executor）被分离出 PHP 包后，这里面（指的是解释器和执行器）原有的一些 API 定义及宏也渐渐演变成了一套新的 API 系统：Zend API。如今的 Zend API 已经承担了很多原来（指的是分离之前）本属于 PHP API 的职责，大量的 PHP API 被以别名的方式简化为对应的 Zend API。我们推荐您应该尽可能地使用 Zend API，PHP API 只是因为兼容性原因才被保留下来。举例来说， zval 和 pval 其实是同一类型，只不过 zval 定义在 Zend 部分，而 pval 定义在 PHP 部分（实际上 pval 根本就是 zval 的一个别名）。但由于 INTERNAL_FUNCTION_PARAMETERS 是一个 Zend 宏，因此我们在上面的声明中使用了 zval 。在编写代码时，你也应该总是使用 zval 以遵循新的 Zend API 规范。

这个声明中的参数列表非常重要，你应该牢记于心。（表 3.1 “PHP 调用函数的 Zend 参数”详细介绍了这些参数）

表 3.1 PHP 调用函数的 Zend 参数

参数	说明
ht	这个参数包含了 Zend 参数的个数。但你不应该直接访问这个值，而是应该通过 ZEND_NUM_ARGS() 宏来获取参数的个数。
return_value	这个参数用来保存函数向 PHP 返回的值。访问这个变量的最佳方式也是用一系列的宏。后面我们会有详细说明。
this_ptr	根据这个参数你可以访问该函数所在的对象（换句话说，此时这个函数应该是一个类的“方法”）。推荐使用函数 getThis() 来得到这个值。
return_value_used	这个值主要用来标识函数的返回值是否为脚本所使用。0 表示脚本不使用其返回值，而 1 则相反。通常用于检验函数是否被正确调用以及速度优化方面，这是因为返回一个值是一种代价很昂贵的操作（可以在 array.c 里面看一下是如何利用这一特性的）。
executor_globals	这个变量指向 Zend Engine 的全局设置，在创建新变量时这个这个值会很有用。我们也可以函数中使用宏 TSRMLS_FETCH() 来引用这个值。

声明 Zend 函数块

现在你已经声明了导出函数，除此之外你还必须得将其引入 Zend 。这些函数的引入是通过一个包含有 N 个 zend_function_entry 结构的数组来完成的。数组的每一项都对应于一个外部可见的函数，每一项都包含了某个函数在 PHP 中出现的名字以及在 C 代码中所定义的名字。zend_function_entry 的内部定义如“例 3.4 zend_function_entry 的内部声明”所示：

例 3.4 zend_function_entry 的内部声明

```
typedef struct _zend_function_entry {
  char *fname;
```

```
    void (*handler)(INTERNAL_FUNCTION_PARAMETERS);
    unsigned char *func_arg_types;
} zend_function_entry;
```

字段	说明
fname	指定在 PHP 里所见到的函数名（比如：fopen、mysql_connect 或者是我们样例中的 first_module)。
handler	指向对应 C 函数的句柄。样例可以参考前面使用宏 INTERNAL_FUNCTION_PARAMETERS 的函数声明。
func_arg_types	用来标识一些参数是否要强制性地按引用方式进行传递。通常应将其设定为 NULL。

对于上面的例子，我们可以这样来声明：

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

你可能已经看到了，这个结构的最后一项是 {NULL, NULL, NULL} 。事实上，这个结构的最后一项也必须始终是 {NULL, NULL, NULL} ，因为 Zend Engine 需要靠它来确认这些导出函数的列表是否列举完毕。

注意:

你*不应该*使用一个预定义的宏来代替列表的结尾部分（即{NULL, NULL, NULL}），因为编译器会尽量寻找一个名为 “NULL” 的函数的指针来代替 NULL ！

宏 ZEND_FE（“Zend Function Entry”的简写）将简单地展开为一个 zend_function_entry 结构。不过需要注意，这些宏对函数采取了一种很特别的命名机制：把你的 C 函数前加上一个 zif_ 前缀。比方说，ZEND_FE(first_module) 其实是指向了一个名为 **zif_first_module()** 的 C 函数。如果你想把宏和一个手工编码的函数名混合使用时（这并不是一个好的习惯），请你务必注意这一点。

小提示: 如果出现了一些引用某个名为 **zif_*()** 函数的编译错误，那十有八九与 ZEND_FE 所定义的函数有关。

“表 3.2 可用来定义函数的宏”给出了一个可以用来定义一个函数的所有宏的列表：

表 3.2 可用来定义函数的宏

宏	说明
ZEND_FE(name, arg_types)	定义了一个 zend_function_entry 内字段 name 为

	“name” 的函数。arg_types 应该被设定为 NULL。这个声明需要有一个对应的 C 函数，该这个函数的名称将自动以 zif_ 为前缀。举例来说，ZEND_FE("first_module", NULL) 就引入了一个名为 first_module() 的 PHP 函数，并被关联到一个名为 zif_first_module() 的 C 函数。这个声明通常与 ZEND_FUNCTION 搭配使用。
ZEND_NAMED_FE(php_name, name, arg_types)	定义了一个名为 php_name 的 PHP 函数，并且被关联到一个名为 name 的 C 函数。arg_types 应该被设定为 NULL。 如果你不想使用宏 ZEND_FE 自动创建带有 zif_ 前缀的函数名的话可以用这个来代替。通常与 ZEND_NAMED_FUNCTION 搭配使用。
ZEND_FALIAS(name, alias, arg_types)	为 name 创建一个名为 alias 的别名。arg_types 应该被设定为 NULL。这个声明不需要有一个对应的 C 函数，因为它仅仅是创建了一个用来代替 name 的别名而已。
PHP_FE(name, arg_types)	以前的 PHP API，等同于 ZEND_FE 。仅为兼容性而保留，请尽量避免使用。
PHP_NAMED_FE(runtime_name, name, arg_types)	以前的 PHP API，等同于 ZEND_NAMED_FE 。仅为兼容性而保留，请尽量避免使用。

注意：你不能将 ZEND_FE 和 PHP_FUNCTION 混合使用，也不能将 PHP_FE 和 ZEND_FUNCTION 混合使用。但是将 ZEND_FE ＋ ZEND_FUNCTION 和 PHP_FE ＋ PHP_FUNCTION 一起混合使用是没有任何问题的。当然我们并不推荐这样的混合使用，而是建议你全部使用 ZEND_* 系列的宏。

声明 Zend 模块

Zend 模块的信息被保存在一个名为 zend_module_entry 的结构，它包含了所有需要向 Zend 提供的模块信息。你可以在“例 3.5 zend_module_entry 的内部声明”中看到这个 Zend 模块的内部定义：

例 3.5 zend_module_entry 的内部声明

```
typedef struct _zend_module_entry zend_module_entry;

struct _zend_module_entry {
unsigned short size;
unsigned int zend_api;
unsigned char zend_debug;
unsigned char zts;
char *name;
zend_function_entry *functions;
int (*module_startup_func)(INIT_FUNC_ARGS);
int (*module_shutdown_func)(SHUTDOWN_FUNC_ARGS);
int (*request_startup_func)(INIT_FUNC_ARGS);
```

```
  int (*request_shutdown_func)(SHUTDOWN_FUNC_ARGS);
  void (*info_func)(ZEND_MODULE_INFO_FUNC_ARGS);
  char *version;
 ... // 其余的一些我们不感兴趣的信息
};
```

字段	说明
size, zend_api, zend_debug and zts	通常用 "STANDARD_MODULE_HEADER" 来填充，它指定了模块的四个成员：标识整个模块结构大小的 size ，值为 ZEND_MODULE_API_NO 常量的 zend_api，标识是否为调试版本（使用 ZEND_DEBUG 进行编译）的 zend_debug，还有一个用来标识是否启用了 ZTS （Zend 线程安全，使用 ZTS 或 USING_ZTS 进行编译）的 zts。
name	模块名称（像“File functions”、“Socket functions”、“Crypt”等等）. 这个名字就是使用 **phpinfo()** 函数后在“Additional Modules”部分所显示的名称。
functions	Zend 函数块的指针，这个我们在前面已经讨论过。
module_startup_func	模块启动函数。这个函数仅在模块初始化时被调用，通常用于一些与整个模块相关初始化的工作（比如申请初始化的内存等等）。如果想表明模块函数调用失败或请求初始化失败请返回 FAILURE，否则请返回 SUCCESS。可以通过宏 ZEND_MINIT 来声明一个模块启动函数。如果不想使用，请将其设定为 NULL。
module_shutdown_func	模块关闭函数。这个函数仅在模块卸载时被调用，通常用于一些与模块相关的反初始化的工作（比如释放已申请的内存等等）。这个函数和 **module_startup_func()** 相对应。如果想表明函数调用失败或请求初始化失败请返回 FAILURE，否则请返回 SUCCESS。可以通过宏 ZEND_MSHUTDOWN 来声明一个模块关闭函数。如果不想使用，请将其设定为 NULL。
request_startup_func	请求启动函数。这个函数在每次有页面的请求时被调用，通常用于与该请求相关的的初始化工作。如果想表明函数调用失败或请求初始化失败请返回 FAILURE，否则请返回 SUCCESS。注意：如果该模块是在一个页面请求中被动态加载的，那么这个模块的请求启动函数将晚于模块启动函数的调用(其实这两个初始化事件是同时发生的)。可以使用宏 ZEND_RINIT 来声明一个请求启动函数，若不想使用，请将其设定为 NULL。
request_shutdown_func	请求关闭函数。这个函数在每次页面请求处理完毕后被调用，正好与 **request_startup_func()** 相对应。如果想表明函数调用失败或请求初始化失败请返回 FAILURE，否则请返回 SUCCESS。注意：当在页面请求作为动态模块加载时，这个请求关闭函数先于模块关闭函数的调用(其实这两个反初始化事件是同时发生的)。可以使用宏 ZEND_RSHUTDOWN 来声明

字段	说明
	这个函数，若不想使用，请将其设定为 NULL 。
info_func	模块信息函数。当脚本调用 phpinfo() 函数时，Zend 便会遍历所有已加载的模块，并调用它们的这个函数。每个模块都有机会输出自己的信息。通常情况下这个函数被用来显示一些环境变量或静态信息。可以使用宏 ZEND_MINFO 来声明这个函数，若不想使用，请将其设定为 NULL 。
version	模块的版本号。如果你暂时还不想给某块设置一个版本号的话，你可以将其设定为 NO_VERSION_YET。但我们还是推荐您在此添加一个字符串作为其版本号。版本号通常是类似这样： "2.5-dev", "2.5RC1", "2.5" 或者 "2.5pl3" 等等。
Remaining structure elements	这些字段通常是在模块内部使用的，通常使用宏 STANDARD_MODULE_PROPERTIES 来填充。而且你也不应该将他们设定别的值。STANDARD_MODULE_PROPERTIES_EX 通常只会在你使用了全局启动函数(ZEND_GINIT)和全局关闭函数(ZEND_GSHUTDOWN)时才用到，一般情况请直接使用 STANDARD_MODULE_PROPERTIES 。

在我们的例子当中，这个结构被定义如下：

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

这基本上是你可以设定最简单、最小的一组值。该模块名称为“First Module”，然后是所引用的函数列表，其后所有的启动和关闭函数都没有使用，均被设定为了 NULL。

作为参考，你可以在表 3.3 “所有可声明模块启动和关闭函数的宏”中找到所有的可设置启动与关闭函数的宏。这些宏暂时在我们的例子中还尚未用到，但稍后我们将会示范其用法。你应该使用这些宏来声明启动和关闭函数，因为它们都需要引入一些特殊的变量(INIT_FUNC_ARGS 和 SHUTDOWN_FUNC_ARGS)，而这两个参数宏将在你使用下面这些预定义宏时被自动引入（其实就是图个方便）。如果你是手工声明的函数或是对函数的参数列表作了一些必要的修改，那么你就应该修改你的模块相应的源代码来保持兼容。

表 3.3 所有可声明模块启动和关闭函数的宏

宏	描述
ZEND_MINIT(module)	声明一个模块的启动函数。函数名被自动设定为 zend_minit_<module> (比如：zend_minit_first_module)。通常与 ZEND_MINIT_FUNCTION 搭配使用。
ZEND_MSHUTDOWN(module)	声明一个模块的关闭函数。函数名被自动设定为 zend_mshutdown_<module> (比如：zend_mshutdown_first_module)。通常与 ZEND_MSHUTDOWN_FUNCTION 搭配使用。
ZEND_RINIT(module)	声明一个请求的启动函数。函数名被自动设定为 zend_rinit_<module> (比如：zend_rinit_first_module)。通常与 ZEND_RINIT_FUNCTION 搭配使用。
ZEND_RSHUTDOWN(module)	声明一个请求的关闭函数。函数名被自动设定为 zend_rshutdown_<module> (比如：zend_rshutdown_first_module)。通常与 ZEND_RSHUTDOWN_FUNCTION 搭配使用。
ZEND_MINFO(module)	声明一个输出模块信息的函数，用于 phpinfo()。函数名被自动设定为 zend_info_<module> (比如：zend_info_first_module)。通常与 ZEND_MINFO_FUNCTION 搭配使用。

实现 get_module() 函数

这个函数只用于动态可加载模块。我们先来看一下如何通过宏 ZEND_GET_MODULE 来创建这个函数：

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

这个函数的实现被一条件编译语句所包围。这是很有必要的，因为 **get_module()** 函数仅仅在你的模块想要编译成动态模块时才会被调用。通过在编译命令行指定编译条件：COMPILE_DL_FIRSTMOD （也就是上面我们设置的那个预定义）的打开与否，你就可以决定是编译成一个动态模块还是编译成一个内建模块。如果想要编译成内建模块的话，那么这个 **get_module()** 将被移除。

get_module() 函数在模块加载时被 Zend 所调用，你也可以认为是被你 PHP 脚本中的 **dl()** 函数所调用。这个函数的作用就是把模块的信息信息块传递 Zend 并通知 Zend 获取这个模块的相关内容。

如果你没有在一个动态可加载模块中实现 **get_module()** 函数，那么当你在访问它的时候 Zend 就会向你抛出一个错误信息。

实现导出函数

导出函数的实现是我们构建扩展的最后一步。在我们的 first_module 例子中，函数被实现如下：

```
ZEND_FUNCTION(first_module)
{
   long parameter;

   if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l", &parameter) == FAILURE) {
      return;
   }

   RETURN_LONG(parameter) ;
}
```

这个函数是用宏 ZEND_FUNCTION 来声明的，和前面我们讨论的 Zend 函数块中的 ZEND_FE 声明相对应。在函数的声明之后，我们的代码便开始检查和接收这个函数的参数。在将参数进行转换后将其值返回。（参数的接收和处理我们马上会在下一节中讲到）。

小结

一切基本上就这样了 —— 我们在实现一个模块时不会再遇到其他方面的事了。内建模块也基本上同动态模块差不多。因此，有了前面几节我们所掌握的信息，再在你遇到 PHP 源代码的时候你就有能力去搞定这些小麻烦。

在下面的几个小节里，我们将会学习到如何利用 PHP 内核来创建一个更为强大的扩展！

（十一）接收参数

对于扩展来说，最重要的一件事就是如何接收和处理那些通过函数参数传递而来的数据。大多数扩展都是用来处理某些特定的输入数据的（或者是根据参数来决定进行某些特定的动作），而函数的参数就是 PHP 代码层和 C 代码层之间交换数据的唯一途径。当然，你也可以通过事先定义好的全局变量来交换数据（这个我们稍后会谈到），不过这种习惯可不太好，我们应该尽量避免。

在 PHP 中并不需要做任何显式的函数声明，这也就是我们为什么说 PHP 的调用语法是动态的而且 PHP 从不会检查任何错误的原因。调用语法是否正确完全是留给用户自己的工作。也就是说，在调用一个函数时完全有可能这次用一个参数而下次用 4 个参数，而且两种情况在语法上都是正确的。

取得参数数量

因为 PHP 不但没法根据函数的显式声明来对调用进行语法检查，而且它还支持可变参数，所以我们就不得不在所调用函数的内部来获取参数个数。这个工作可以交给宏 ZEND_NUM_ARGS 来完成。在（PHP4）以前，这个宏（在 PHP3 中应该指的是宏 ARG_COUNT，因为 ZEND_NUM_ARGS 宏是直到 PHP4.0 才出现的，并且其定义一直未变。PHP4 及以后虽也有 ARG_COUNT 宏定义，但却仅仅是为兼容性而保留的，并不推荐使用，译者注）是利用所调用的 C 函数中的变量 ht（就是定义在宏

INTERNAL_FUNCTION_PARAMETERS 里面的那个，HashTable * 类型）来获取参数个数的，而现在变量 ht 就只包含函数的参数个数了（int 类型）。与此同时还定义了一个哑宏：ZEND_NUM_ARGS（直接等于 ht，见 Zend.h）。尽量地采用 ZEND_NUM_ARGS 是个好习惯，因为这样可以保证在函数调用接口上的兼容性。

下面的代码展示了如何判断传入函数的参数个数是否正确：

```
if(ZEND_NUM_ARGS() != 2) WRONG_PARAM_COUNT;
```

如果没有为该函数传入两个参数，那么就会退出该函数并且发出一个错误消息。在这段代码中我们使用了一个工具宏：WRONG_PARAM_COUNT，它主要用来抛出一个类似“Warning: Wrong parameter count for firstmodule() in /home/www/htdocs/firstmod.php on line 5”这样的错误信息。

这个宏会主要负责抛出一个默认的错误信息，然后便返回调用者。我们可以在 zend_API.h 中找到它的定义：

```
ZEND_API void wrong_param_count(void);
#define WRONG_PARAM_COUNT { wrong_param_count(); return; }
```

正如您所见，它调用了一个内部函数 **wrong_param_count()** ，这个函数会输出一个警告信息。至于如何抛出一个自定义的错误信息，可以参见后面的“打印信息”一节。

取回参数

对传入的参数进行解析是一件很常见同时也是颇为乏味的事情，而且同时你还得做好标准化的错误检查和发送错误消息等琐事。不过从 PHP 4.1.0 开始，我们就可以用一个新的参数解析 API 来搞定这些事情。这个 API 可以大大简化参数的接收处理工作，尽管它在处理可变参数时还有点弱。但既然绝大部分函数都没有可变参数，那么使用这个 API 也就理所应当地成为了我们处理函数参数时的标准方法。

这个参数解析函数的声明大致如下：

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

第一个参数 num_args 表明了我们想要接收的参数个数，我们经常使用 ZEND_NUM_ARGS() 来表示对传入的参数“有多少要多少”。第二参数应该总是宏 TSRMLS_CC 。第三个参数 type_spec 是一个字符串，用来指定我们所期待接收的各个参数的类型，有点类似于 printf 中指定输出格式的那个格式化字符串。剩下的参数就是我们用来接收 PHP 参数值的变量的指针。

zend_parse_parameters() 在解析参数的同时会尽可能地转换参数类型，这样就可以确保我们总是能得到所期望的类型的变量。任何一种标量类型都可以转换为另外一种标量类型，但是不能在标量类型与复杂类型（比如数组、对象和资源等）之间进行转换。

如果成功地解析和接收到了参数并且在转换期间也没出现错误，那么这个函数就会返回 SUCCESS，否则返回 FAILURE。如果这个函数不能接收到所预期的参数个数或者不能成功转换参数类型时就会抛出一些类似下面这样的错误信息：

Warning - ini_get_all() requires at most 1 parameter, 2 given
Warning - wddx_deserialize() expects parameter 1 to be string, array given
当然，每个错误信息都会带有错误发生时所在的文件名和行数的。

下面这份清单完整地列举出了我们可以指定接收的参数类型：

- l - 长整形
- d - 双精度浮点类型
- s - 字符串 (也可能是空字节)和其长度
- b - 布尔型
- r - 资源，保存在 zval*
- a - 数组，保存在 zval*
- o - （任何类的）对象，保存在 zval *
- O - （由 class entry 指定的类的）对象，保存在 zval *
- z - 实际的 zval*

下面的一些字符在类型说明字符串（就是那个 char *type_spec）中具有特别的含义：

- | - 表明剩下的参数都是可选参数。如果用户没有传进来这些参数值，那么这些值就会被初始化成默认值。
- / - 表明参数解析函数将会对剩下的参数以 **SEPARATE_ZVAL_IF_NOT_REF()** 的方式来提供这个参数的一份拷贝，除非这些参数是一个引用。
- ! - 表明剩下的参数允许被设定为 NULL（仅用在 a、o、O、r 和 z 身上）。如果用户传进来了一个 NULL 值，则存储该参数的变量将会设置为 NULL。

当然啦，熟悉这个函数的最好的方法就是举个例子来说明。下面我们就来看一个例子：

```
/* 取得一个长整型，一个字符串和它的长度，再取得一个 zval 值。 */
long l;
char *s;
int s_len;
```

```
zval *param;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
                          "lsz", &l, &s, &s_len, &param) == FAILURE) {
   return;
}
/* 取得一个由 my_ce 所指定的类的一个对象，另外再取得一个可选的双精度值。 */
zval *obj;
double d = 0.5;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
                          "O|d", &obj, my_ce, &d) == FAILURE) {
   return;
}

/* 取得一个对象或空值，再取得一个数组。

  如果传递进来一个空对象，则 obj 将被设置为 NULL。 */
zval *obj;
zval *arr;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "O!a", &obj, &arr) == FAILURE) {
   return;
}
/* 取得一个分离过的数组。 */
zval *arr;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a/", &arr) == FAILURE) {
   return;
}

/* 仅取得前 3 个参数（这对可变参数的函数很有用）。 */
zval *z;
zend_bool b;
zval *r;
if (zend_parse_parameters(3, "zbr!", &z, &b, &r) == FAILURE) {
   return;
}
```

注意，在最后的一个例子中，我们直接用了数值 3 而不是 **ZEND_NUM_ARGS()** 来作为想要取得参数的个数。这样如果我们的 PHP 函数具有可变参数的话我们就可以只接收最小数量的参数。当然，如果你想操作剩下的参数，你可以用 **zend_get_parameters_array_ex()** 来得到。.

这个参数解析函数还有一个带有附加标志的扩展版本，这个标志可以让你控制解析函数的某些动作。

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

这个标志（flags）目前仅接受 ZEND_PARSE_PARAMS_QUIET 这一个值，它表示这个函数不输出任何错误信息。这对那些可以传入完全不同类型参数的函数非常有用，但这样你也就不得不自己输出错误信息。

下面就是一个如何既可以接收 3 个长整形数又可以接收一个字符串的例子：

```
long l1, l2, l3;
char *s;
if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET,
                             ZEND_NUM_ARGS() TSRMLS_CC,
                             "lll", &l1, &l2, &l3) == SUCCESS) {
   /* manipulate longs */
} else if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET,
                                    ZEND_NUM_ARGS(), "s", &s, &s_len) == SUCCESS) {
   /* manipulate string */
} else {
   php_error(E_WARNING, "%s() takes either three long values or a string as argument",
             get_active_function_name(TSRMLS_C));
   return;
}
```

我想你通过上面的那些例子就可以基本掌握如何接收和处理参数了。如果你想看更多的例子，请翻阅 PHP 源码包中那些自带的扩展的源代码，那里面包含了你可能遇到的各种情况。

以前的老式的获取参数的的方法（不推荐）

获取函数参数这件事情我们还可以通过 **zend_get_parameters_ex()** 来完成：

```
zval **parameter;

if(zend_get_parameters_ex(1, &parameter) != SUCCESS)
 WRONG_PARAM_COUNT;
```

所有的参数都存储在一个二次指向的 zval 容器里面（其实就是一个 zval* 数组，译者注）。上面的这段代码尝试接收 1 个参数并且将其保存在 parameter 所指向的位置。

zend_get_parameters_ex() 至少需要两个参数。第一个参数表示我们想要接收参数的个数（这个值通常是对应于 PHP 函数参数的个数，由此也可以看出事先对调用语法正确性的检查是多么重要）。第二个参数（包括剩下的所有参数）指向一个二次指向 zval 的指针。（即 ***zval，是不是有点糊涂了？^_^）这些指针是必须的，因为 Zend 内部是使用 **zval 进行工作的。为了能被在我们函数内部定义的 **zval 局部变量所访问，我们就必须在用一个指针来指向它。

zend_get_parameters_ex() 的返回值可以是 SUCCESS 或 FAILURE，分别表示参数处理的成功或失败。如果处理失败，那最大的可能就是由于没有指定一个正确的参数个数。如果处理失败，则应该使用宏 WRONG_PARAM_COUNT 来退出函数。

如果想接收更多的的参数，可以用类似下面一段的代码来处理：

```
zval **param1, **param2, **param3, **param4;

if(zend_get_parameters_ex(4, &param1, &param2, &param3, &param4) != SUCCESS)
   WRONG_PARAM_COUNT;
```

zend_get_parameters_ex() 仅检查你是否在试图访问过多的参数。如果函数有 5 个参数，而你仅仅接收了其中的 3 个，那么你将不会收到任何错误信息，**zend_get_parameters_ex()** 仅返回前三个参数的值。再次调用 **zend_get_parameters_ex()** 也不会获得剩下两个参数的值，而还是返回前三个参数的值。

接收可变（可选）参数

如果你想接收一些可变参数，那用前面我们刚刚讨论的方法就不太合适了，主要是因为我们将不得不为每个可能的参数个数来逐行调用 **zend_get_parameters_ex()**，这太不爽了。

为了解决这个问题，我们可以借用一下 **zend_get_parameters_array_ex()** 这个函数。它可以帮助我们接收不定量的参数并将其保存在我们指定的地方

```
zval **parameter_array[4];
/* get the number of arguments */
argument_count = ZEND_NUM_ARGS();

/* see if it satisfies our minimal request (2 arguments) */
/* and our maximal acceptance (4 arguments) */
if(argument_count < 2 || argument_count > 4)
   WRONG_PARAM_COUNT;

/* argument count is correct, now retrieve arguments */
if(zend_get_parameters_array_ex(argument_count, parameter_array) != SUCCESS)
   WRONG_PARAM_COUNT;
```

让我们来看看这几行代码。首先代码检查了传入参数的个数，确保在我们可接受的范围内；然后就调用 **zend_get_parameters_array_ex()** 把所有有效参数值的指针填入 parameter_array。

我们可以在 fsockopen() 函数（位于 ext/standard/fsock.c ）中找到一个更为漂亮的实现。代码大致如下，你也不用担心还没有弄懂全部的函数，因为我们很快就会谈到它们。

例 3.6 PHP 中带有可变参数的 fsockopen() 函数的实现。

```
pval **args[5];
int *sock=emalloc(sizeof(int));
int *sockp;
int arg_count=ARG_COUNT(ht);
int socketd = -1;
unsigned char udp = 0;
struct timeval timeout = { 60, 0 };
unsigned short portno;
unsigned long conv;
char *key = NULL;
FLS_FETCH();
if (arg_count > 5 || arg_count < 2 || zend_get_parameters_array_ex(arg_count,args)==FAILURE) {
   CLOSE_SOCK(1);
   WRONG_PARAM_COUNT;
}

switch(arg_count) {
   case 5:
        convert_to_double_ex(args[4]);
        conv = (unsigned long) (Z_DVAL_PP(args[4]) * 1000000.0);
        timeout.tv_sec = conv / 1000000;
        timeout.tv_usec = conv % 1000000;
        /* fall-through */
   case 4:
        if (!PZVAL_IS_REF(*args[3])) {
            php_error(E_WARNING,"error string argument to fsockopen not passed by reference");
        }
        pval_copy_constructor(*args[3]);
        ZVAL_EMPTY_STRING(*args[3]);
        /* fall-through */
   case 3:
        if (!PZVAL_IS_REF(*args[2])) {
            php_error(E_WARNING,"error argument to fsockopen not passed by reference");
            return;
        }
        ZVAL_LONG(*args[2], 0);
        break;
}
```

```
convert_to_string_ex(args[0]);
convert_to_long_ex(args[1]);
portno = (unsigned short) Z_LVAL_P(args[1]);

key = emalloc(Z_STRLEN_P(args[0]) + 10);
```

fsockopen() 可以接收 2－5 个参数。在必需的变量声明之后便开始检查参数的数量范围。然后在一个 switch 语句中使用了贯穿（fall-through）法来处理这些的参数。这个 switch 语句首先处理最大的参数个数（即 5），随后依次处理了参数个数为 4 和 3 的情况，最后用 break 关键字跳出 switch 来忽略对其他情况下参数（也就是只含有 2 个参数情况）的处理。这样在经过 switch 处理之后，就开始处理参数个数为最小时（即 2）的情况。

这种像楼梯一样的多级处理方法可以帮助我们很方便地处理一些可变参数。

存取参数

为了存取一些参数，让每个参数都具有一个明确的（C）类型是很有必要的。但 PHP 是一种动态语言，PHP 从不做任何类型检查方面的工作，因此不管你想不想，调用者都可能会把任何类型的数据传到你的函数里。比如说，如果你想接收一个整数，但调用者却可能会给你传递个数组，反之亦然－PHP 可不管这些的。

为了避免这些问题，你就必须用一大套 API 函数来对传入的每一个参数都做一下强制性的类型转换。（见表 3.4 参数类型转换函数）

注意，所有的参数转换函数都以一个 **zval 来作为参数。

表 3.4 参数类型转换函数

函数	说明
convert_to_boolean_ex()	强制转换为布尔类型。若原来是布尔值则保留，不做改动。长整型值 0、双精度型值 0.0、空字符串或字符串‘0’还有空值 NULL 都将被转换为 FALSE（本质上是一个整数 0）。数组和对象若为空则转换为 FALSE，否则转为 TRUE。除此之外的所有值均转换为 TRUE（本质上是一个整数 1）。
convert_to_long_ex()	强制转换为长整型，这也是默认的整数类型。如果原来是空值 NULL、布尔型、资源当然还有长整型，则其值保持不变（因为本质上都是整数 0）。双精度型则被简单取整。包含有一个整数的字符串将会被转换为对应的整数，否则转换为 0。空的数组和对象将被转换为 0，否则将被转换为 1。
convert_to_double_ex()	强制转换为一个双精度型，这是默认的浮点数类型。如果原来是空值 NULL 、布尔值、资源和双精度型则其值保持不变（只变一下变量类型）。包含有一个数字的字符串将被转换成相应的数字，否则被转换为 0.0。空的数组和对象将被转换为 0.0，否则将被转换为 1.0。

convert_to_string_ex()	强制转换为字符串。空值 NULL 将被转换为空字符串。布尔值 TRUE 将被转换为 '1'，FALSE 则被转为一个空字符串。长整型和双精度型会被分别转换为对应的字符串，数组将会被转换为字符串'Array'，而对象则被转换为字符串'Object'。
convert_to_array_ex(value)	强制转换为数组。若原来就是一数组则不作改动。对象将被转换为一个以其属性为键名，以其属性值为键值的数组。（方法将会被转化为一个'scalar'键，键值为方法名？待验证）空值 NULL 将被转换为一个空数组。除此之外的所有值都将被转换为仅有一个元素（下标为 0）的数组，并且该元素即为该值。
convert_to_object_ex(value)	强制转换为对象。若原来就是对象则不作改动。空值 NULL 将被转换为一个空对象。数组将被转换为一个以其键名为属性，键值为其属性值的对象。其他类型则被转换为一个具有'scalar'属性的对象，'scalar'属性的值即为该值本身。
convert_to_null_ex(value)	强制转换为空值 NULL。

在你的参数上使用这些函数可以确保传递给你的数据都是类型安全的。如果提供的类型不是需要的类型，PHP 就会强制性地返回一个相应的伪值（比如空字符串、空的数组或对象、数值 0 或布尔值的 FALSE 等）来确保结果是一个已定义的状态。

下面的代码是从前面讨论过的模块中摘录的，其中用到了这些转换函数：

```
zval **parameter;
if((ZEND_NUM_ARGS() != 1) || (zend_get_parameters_ex(1, &parameter) != SUCCESS))
{
  WRONG_PARAM_COUNT;
}

convert_to_long_ex(parameter);

RETURN_LONG(Z_LVAL_P(parameter));
```

在收到参数指针以后，参数值就被转换成了一个长整型（或整形），转换的结果就是这个函数的返回值。如果想要弄懂如何存取到这个返回值，我们就需要对 zval 有一点点认识。它的定义如下：

例 3.7 zval 类型的定义

```
typedef pval zval;

typedef struct _zval_struct zval;
typedef union _zvalue_value {
long lval; /* long value */
double dval; /* double value */
```

```
struct {
char *val;
int len;
} str;
HashTable *ht; /* hash table value */
struct {
zend_class_entry *ce;
HashTable *properties;
} obj;
} zvalue_value;

struct _zval_struct {
/* Variable information */
zvalue_value value; /* value */
unsigned char type; /* active type */
unsigned char is_ref;
short refcount;
};
```

实际上，pzval（定义在 php.h）就是 zval（定义在 zend.h）的一个别名，都是 _zval_struct 结构的一个别名。_zval_struct 是一个很有趣的结构，它保存了这个结构的真实值 value、类型 type 和引用信息 is_ref。字段 value 是一个 zvalue_value 联合，根据变量类型的不同，你就可以访问不同的联合成员。对于这个结构的描述，可参见“表 3.5 Zend zval 结构”、“表 3.6 Zend zvalue_value 结构”和“表 3.7 Zend 变量类型”。

表 3.5 Zend zval 结构

字段	说明
value	变量内容的联合，参见“表 3.6 Zend zvalue_value 结构”。
type	变量的类型。“表 3.7 Zend 变量类型”给出了一个完整的变量类型列表。
is_ref	0 表示这个变量还不是一个引用。1 表示这个变量还有被别的变量所引用。
refcount	表示这个变量是否仍然有效。每增加一个对这个变量的引用，这个数值就增加 1。反之，每失去一个对这个变量的引用，该值就会减 1。当引用计数减为 0 的时候，就说明已经不存在对这个变量的引用了，于是这个变量就会自动释放。

表 3.6.Zend zvalue_value Structure

字段	说明
lval	如果变量类型为 IS_LONG、IS_BOOLEAN 或 IS_RESOURCE 就用这个属性值。
dval	如果变量类型为 IS_DOUBLE 就用这个属性值。
str	如果变量类型为 IS_STRING 就访问这个属性值。它的字段 len 表示这个字符串的

长度，字段 val 则指向该字符串。由于 Zend 使用的是 C 风格的字符串，因此字符串的长度就必须把字符串末尾的结束符 0×00 也计算在内。

ht	如果变量类型为数组，那这个 ht 就指向数组的哈希表入口。
obj	如果变量类型为 IS_OBJECT 就用这个属性值。

表 3.7 Zend 变量类型

类型常量	说明
IS_NULL	表示是一个空值 NULL；
IS_LONG	是一个长整形（或整形）数；
IS_DOUBLE	是一个双精度的浮点数；
IS_STRING	是一个字符串；
IS_ARRAY	是一个数组；
IS_OBJECT	是一个对象；
IS_BOOL	是一个布尔值；
IS_RESOURCE	是一个资源（关于资源的讨论，我们以后会在适当的时候讨论到它）；
IS_CONSTANT	是一个常量；

想访问一个长整型数，那你就访问 zval.value.lval；想访问一个双精度数，那你就访问 zval.value.dval，依此类推。不过注意，因为所有的值都是保存在一个联合里面，所以如果你用了不恰当的字段去访问，那就可能会得到一个毫无意义的结果。

访问一个数组和对象可能会稍微复杂些，稍后再说。

处理引用传递过来的参数

如果函数里面的参数是通过引用传递进来的，但是你又想去修改它，那就需要多加小心了。

根据我们前面所讨论的知识，我们还没有办法去修改一个经 PHP 函数参数传进来的 zval 。当然你可以修改那些在函数内部创建的局部变量的 zval ，但这并代表你可以修改任何一个指向 Zend 自身内部数据的 zval （也就是那些非局部的 zval）！

这是为什么呢？我想你可能注意到了，我们前面讨论的 API 函数都是类似于 ***_ex()** 这样子的。比如我们用 **zend_get_parameters_ex()** 而不用 **zend_get_parameters()**，用 **convert_to_long_ex()** 而不用 **convert_to_long()** 等等。这些 ***_ex()** 函数被称为新的“扩展”的 Zend API，它们的速度要快于对应的传统 API，但副作用是它们只提供了只读访问机制。

因为 Zend 内部是靠引用机制来运行的，因此不同的变量就有可能引自同一个 value （zval 结构的字段 value）。而修改一个 zval 就要求这个 zval 的 value 必须是独立的，也就是说这个 value 不能被其他 zval 引用。如果有一个 zval 里面的 value 还被其他 zval 引用了，你也同时把这个 value 给修改了，那你也同时就把其他 zval 的 value 给修改了，因为它们的 value 只是简单地指向了这个 value 而已。

zend_get_parameters_ex() 是根本不管这些的，它只是简单地返回一个你所期望的那个 zval 的指针。至于这个 zval 是否还存在其他引用，who care？（所以我们说这些 *_ex() 只提供了只读机制，并没有提供可写机制。你若利用 *_ex() 的结果强行赋值也是可以的，但这样就没法保证数据安全了。译者注。）。而和这个 API 对应的传统 API **zend_get_parameters ()** 就会即时检查 value 的引用情况。如果它发现了对 value 的引用，它就会马上再重新创建一个独立的 zval ，然后把引用的数据复制一份到新的刚刚申请的空间里面，然后返回这个新的 zval 的指针。

这个动作我们称之为“*zval 分离*（或者 *pval 分离*）”。由于 *_ex() 函数并不执行“zval 分离”操作，因此它们虽然快，但是却不能用于进行写操作。

但不管怎样，要想修改参数，写操作是不可避免的。于是 Zend 使用了这样一个特别的方式来处理写操作：无论何时，只要函数的某个参数使用过引用传递的，那它就自动进行 zval 分离。这也就意味着不管什么时间，只要你像下面这样来调用一个 PHP 函数，Zend 就会自动确保传入的是一个独立的 value 并处于“写安全”状态。

my_function(&$parameter);
但这不是一般参数（指不带 & 前缀但也也是引用的参数，译者注）的情况。所有不是直接通过引用（指不带 & 前缀）传递的参数都将只是处在一种“只读”状态（其实这里的“只读”状态可以理解为“写不安全”状态）。

这就要求你确认是否真的在同一个引用打交道，否则你可能会收到你不太想要的结果。我们可以使用宏 PZVAL_IS_REF 来检查一个参数是否是通过引用传递的。这个宏接收一个 zval* 参数。“例 3.8 测试参数是否是引用传递”给出了这样一个例子：

例 3.8 测试参数是否是引用传递

```
zval *parameter;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &parameter) == FAILURE)
   return;

/* check for parameter being passed by reference */
if (!PZVAL_IS_REF(parameter)) {
{
   zend_error(E_WARNING, "Parameter wasn't passed by reference");
   RETURN_NULL();
}

/* make changes to the parameter */
ZVAL_LONG(parameter, 10);
```

确保其他情况下某些参数的写安全

有时候你可能会遇到过这种情况：你想对用 **zend_get_parameters_ex()** 接收的但是没有通过引用传递的一个参数进行写操作。这时你可以用宏 SEPARATE_ZVAL 来手工进行 zval 分离操作。这样可以得到一个新创建的与原来内部数据独立的 zval，但这个 zval 仅在局部有效，它可以被修改或销毁而不影响外部的全局 zval 。

```
zval **parameter;

/* 接收参数 */
zend_get_parameters_ex(1, &parameter);
/* 此时 <parameter> 仍然关联在 Zend 的内部数据缓冲区 */

/* 现在将 <parameter> “写安全”化 */
SEPARATE_ZVAL(parameter);
/* 现在你可以放心大胆去修改 <parameter> 了，无需担心外部的 zval 会受到影响 */
```

因为宏 SEPARATE_ZVAL 通过 **emalloc()** 函数来申请一个新的 zval ，所以这也就意味着如果你不主动去释放这段内存的话，那它就会直到脚本中止时才被释放。如果你大量调用这个宏却没有释放，那它可能会瞬间塞满你的内存。

注意： 因为现在已经很少遇到和需要传统 API（诸如 zend_get_parameters() 等等）了（貌似它有点过时了），所以有关这些 API 本节不再赘述。

当 PHP 脚本与扩展互相交换数据时，我们还需要做一件很重要的事情，那就是创建变量。这一小节将会展示如何处理那些 PHP 脚本所支持的变量类型。

概述

要创建一个能够被 PHP 脚本所访问的“外部变量”，我们只需先创建一个 zval 容器，然后对这个 zval 结构进行必要的填充，最后再把它引入到 Zend 的内部符号表中就可以了。而且几乎所有变量的创建基本上都是这几个步骤：

```
zval *new_variable;
/*  申请并初始化一个新的的 zval 容器  */
MAKE_STD_ZVAL(new_variable);

/* 设置变量的类型和内容，见下 */

/* 将名为 “new_variable_name” 变量引入符号表 */
ZEND_SET_SYMBOL(EG(active_symbol_table), “new_variable_name”, new_variable);

/* 现在就可以在脚本中用 $new_variable_name 来访问这个变量了 */
```

宏 MAKE_STD_ZVAL 通过 ALLOC_ZVAL 来申请一个新的 zval 容器的内存空间并调用 INIT_ZVAL（查看 PHP4/5 的源代码可知此处为原文笔误，实际上应为 INIT_PZVAL，下同。译注）将其初始化。在当前的 Zend Engine 中，INIT_ZVAL 所负责的初始化工作除了将 zval 容器的引用计数（refcount）置为 1 之外，还会把引用标识也清除（即把 is_ref 也置为 0）。而且在以后的 Zend Engine 中还可能会继续扩展这个 INIT_ZVAL 宏操作，因此我们推荐您使用 MAKE_STD_ZVAL 而非简单使用一个 ALLOC_ZVAL 来完成一个变量的创建工作。当然，如果您是想优化一下速度（或者是不想明确地初始化这个 zval 容器），那还是可以只用 ALLOC_ZVAL 来搞定的。不过我们并不推荐这么做，因为这将不能保证数据的完整性。

ZEND_SET_SYMBOL 宏负责将我们新建的变量引入 Zend 内部的符号表。这个宏会首先检查一下这个变量是否已经存在于符号表中，如果已经存在则将其转换为一个引用变量（同时会自动销毁原有的 *zval* 容器）。事实上这个方法经常用在某些速度要求并不苛刻但希望能少用一些内存的情况下。

您可能注意到了 *ZEND_SET_SYMBOL* 是通过宏 EG 来访问 Zend 执行器（executor）的全局结构的。特别的，如果你使用的是 *EG(active_symbol_table)*，那你就可以访问到当前的活动符号表，从而可以处理一些全局或局部变量。其中局部变量可能会依不同的函数而有所不同。.

当然，要是你很在意程序的运行速度并且不在乎那一点点内存的话，那你可以跳过对相同名字变量存在性的检查而直接使用 **zend_hash_update()** 函数强行将这个名字的变量插入符号表。

```
zval *new_variable;

/*  申请并初始化一个新的的 zval 容器  */
MAKE_STD_ZVAL(new_variable);

/* 设置变量的类型和内容，见下 */


/* 将名为 “new_variable_name” 变量引入符号表 */
zend_hash_update(
      EG(active_symbol_table),
      ”new_variable_name”,
      strlen(”new_variable_name”) + 1,
      &new_variable,
      sizeof(zval *),
      NULL
);
```

实际上这段代码也是很多扩展使用的标准方法。

上面这段代码所产生的变量是局部变量，作用范围跟调用函数的上下文相关。如果你想创建一个全局变量那也很简单，方法还是老方法，只需换个符号表就可以了。

```
zval *new_variable;


/*  申请并初始化一个新的的 zval 容器  */
MAKE_STD_ZVAL(new_variable);

/* 设置变量的类型和内容，见下 */

/* 将名为“new_variable_name”变量引入全局符号表 */


ZEND_SET_SYMBOL(&EG(symbol_table), “new_variable_name”, new_variable);
```

注意，现在宏 *ZEND_SET_SYMBOL* 使用的符号表是全局符号表 *EG(symbol_table)*。另外，*active_symbol_table* 是一个指针，而 *symbol_table* 却不是。这就是我们为什么分别使用 *EG(active_symbol_table)* 和 *&EG(symbol_table)* 的原因 — *ZEND_SET_SYMBOL* 需要一个指针作为其参数。

当然，你同样也可以强行更新这个符号表：

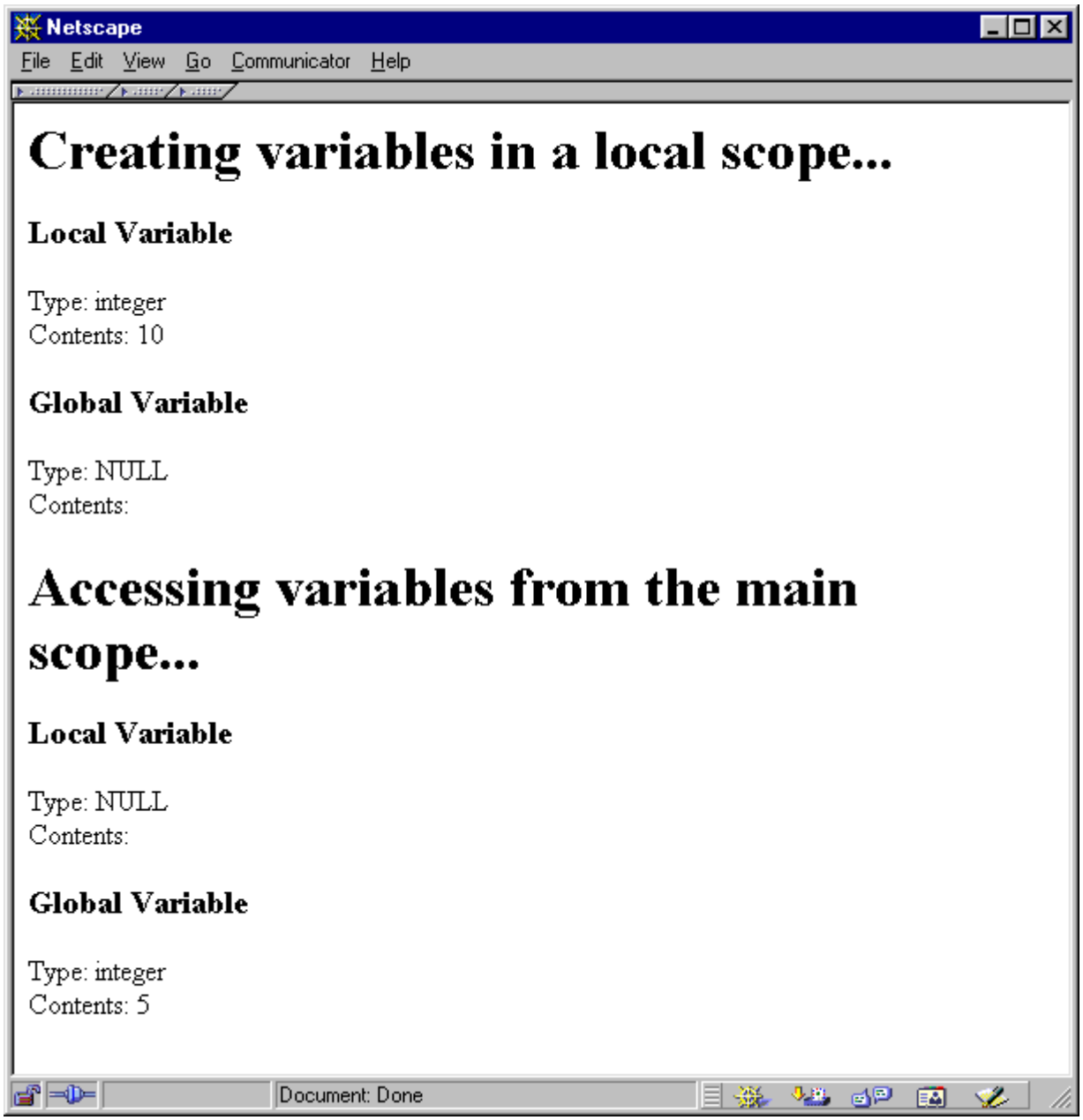
```
zval *new_variable;

/*  申请并初始化一个新的的 zval 容器  */
MAKE_STD_ZVAL(new_variable);

/* 设置变量的类型和内容，见下 */


/* 将名为“new_variable_name”变量引入全局符号表 */
zend_hash_update(
      &EG(symbol_table),
      ”new_variable_name”,
      strlen(”new_variable_name”) + 1,
      &new_variable,
      sizeof(zval *),
      NULL
);
```

例 3.9 “创建不同作用域的变量”向我们展示了创建一个局部变量（local_variable）和一个全局变量（global_variable）的过程。.

注意：你可能会发现在 PHP 函数里似乎还不能直接访问这个全局变量（global_variable），因为你在使用前还必须使用 *global $global_variable;* 声明一下。

例 3.9 创建不同作用域的变量

```
ZEND_FUNCTION(variable_creation)
{
    zval *new_var1, *new_var2;
    MAKE_STD_ZVAL(new_var1);
    MAKE_STD_ZVAL(new_var2);
    ZVAL_LONG(new_var1, 10);
    ZVAL_LONG(new_var2, 5);
    ZEND_SET_SYMBOL(EG(active_symbol_table), “local_variable”, new_var1);
    ZEND_SET_SYMBOL(&EG(symbol_table), “global_variable”, new_var2);
    RETURN_NULL();
}
```



长整型（整数）

现在让我们以长整型变量起点，了解一下如何为一个变量赋值。PHP 中的整数全部是长整型，其值的存储方法也是非常简单的。看一下我们前面讨论过的 zval.value 容器的结构你就会明白，所有的长整型数据都是直接保存在这个联合中的 lval 字段，相应的数据类型（type 字段）为 *IS_LONG*（见 例 3.10 “长整型变量的创建”）。

例 3.10 长整型变量的创建

```
zval *new_long;
MAKE_STD_ZVAL(new_long);
new_long->type = IS_LONG;
new_long->value.lval = 10;
```

或者你也可以直接使用 *ZVAL_LONG* 宏：

```
zval *new_long;
MAKE_STD_ZVAL(new_long);
ZVAL_LONG(new_long, 10);
```

双精度型（浮点数）

PHP 中的浮点数都是双精度型，存储方法和整型差不多，也很简单。它的值是直接放在联合中的 dval 字段，对应数据类型为 *IS_DOUBLE*。

```
zval *new_double;
MAKE_STD_ZVAL(new_double);
new_double->type = IS_DOUBLE;
new_double->value.dval = 3.45;
```

同样你也可以直接使用宏 *ZVAL_DOUBLE*：

```
zval *new_double;
MAKE_STD_ZVAL(new_double);
ZVAL_DOUBLE(new_double, 3.45);
```

字符串

字符串的存储可能会稍费点事。字符串的值是保存在 *zval.value* 容器中的 *str* 结构里面，相应的数据类型为 *IS_STRING*。不过需要注意的是，前面我们已经提到过，所有与 Zend 内部数据结构相关的字符串都必须使用 Zend 自己的内存管理函数来申请空间。这样一来，就不能使用那些静态字符串（因为这种字符串的内存空间是编译器预先分配的）或通过标准函数（比如 *malloc()* 等函数）来申请空间的字符串。

```
zval *new_string;
char *string_contents = “This is a new string variable”;
MAKE_STD_ZVAL(new_string);
```

```
new_string->type = IS_STRING;
new_string->value.str.len = strlen(string_contents);
new_string->value.str.val = estrdup(string_contents);
```

请注意，在这我们使用了 **estrdup()** 函数。当然我们仍可直接使用一个预定义宏 *ZVAL_STRING* 来完成这项工作：

```
zval *new_string;
char *string_contents = "This is a new string variable";
MAKE_STD_ZVAL(new_string);
ZVAL_STRING(new_string, string_contents, 1);
```

ZVAL_STRING 宏的第三个参数指明了该字符串是否需要被复制（使用 **estrdup()** 函数）。值为 1 将导致该字符串被复制，为 0 时则仅仅是简单地将其指向该变量的值容器（即字符串地址，译注）。这项特性将会在你仅仅需要创建一个变量并将其指向一个已经由 Zend 内部数据内存时变得很有用。

如果你想在某一位置截取该字符串或已经知道了这个字符串的长度，那么可以使用宏 *ZVAL_STRINGL(zval, string, length, duplicate)* 来完成这项工作。这个函数会额外需要一个表明该字符串长度地参数。这个宏不但速度上要比 *ZVAL_STRING* 快，而且还是二进制安全的。

如果想创建一个空字符串，那么将其长度置 0 并且把 *empty_string* 作为字符串的内容即可：

```
new_string->type = IS_STRING;
new_string->value.str.len = 0;
new_string->value.str.val = empty_string;
```

当然，我们也专门为您准备了一个相应的宏 *ZVAL_EMPTY_STRING* 来搞定这个步骤：

```
MAKE_STD_ZVAL(new_string);
ZVAL_EMPTY_STRING(new_string);
```

布尔类型

布尔类型变量的创建跟长整型差不多，只是数据类型为 *IS_BOOL*，并且字段 *lval* 所允许的值只能为 0 和 1：

```
zval *new_bool;
MAKE_STD_ZVAL(new_bool);
new_bool->type = IS_BOOL;
new_bool->value.lval = 1;
```

也可以使用宏 *ZVAL_BOOL*（需要另外指定一个值）来完成这件事情，或者干脆直接使用 *ZVAL_TRUE* 或 *ZVAL_FALSE* 直接将其值设定为 *TRUE* 或 *FALSE*。

数组

数组在 Zend 内部是用哈希表（HashTable）来存储的，这个哈希表可以使用一系列的 **zend_hash_*()** 函数来访问。因此我们在创建一个数组时必须先创建一个哈希表，然后再将其保存在 *zval.value* 容器的 *ht* 字段中。

不过针对数组的创建我们现在另有一套非常方便 API 可供使用。为了创建一个数组，我们可先调用一下 **array_init()** 函数：.

```
zval *new_array;
MAKE_STD_ZVAL(new_array);
array_init(new_array);
```

array_init() 函数总是返回 *SUCCESS*。

要给数组增加一个元素，根据实际需要，我们有 N 个函数可供调用。“表 3.8 用于关联数组的 API”、“表 3.9 用于索引数组的 API 第一部分”和“表 3.10 用于索引数组的 API 第二部分”有这些函数的说明。所有这些函数在调用成功时返回 SUCCESS，在调用失败时返回 FAILURE。

表 3.8 用于关联数组的 API

函数	说明
*add_assoc_long(zval *array, char *key, long n);*	添加一个长整型元素。
*add_assoc_unset(zval *array, char *key);*	添加一个 unset 元素。
*add_assoc_bool(zval *array, char *key, int b);*	添加一个布尔值。
*add_assoc_resource(zval *array, char *key, int r);*	添加一个资源。
*add_assoc_double(zval *array, char *key, double d);*	添加一个浮点值。
*add_assoc_string(zval *array, char *key, char *str, int duplicate);*	添加一个字符串。duplicate 用于表明这个字符串是否要被复制到 Zend 的内部内存。
*add_assoc_stringl(zval *array, char *key,*	添加一个指定长度的字符串。其余跟

*char *str, uint length, int duplicate);*	add_assoc_string () 相同。
*add_assoc_zval(zval *array, char *key, zval *value);*	添加一个 zval 结构。这在添加另外一个数组、对象或流等数据时会很有用。

表 3.9 用于索引数组的 API 第一部分

函数	说明
*add_index_long(zval *array, uint idx, long n);*	添加一个长整型元素。
*add_index_unset(zval *array, uint idx);*	添加一个 unset 元素。
*add_index_bool(zval *array, uint idx, int b);*	添加一个布尔值。
*add_index_resource(zval *array, uint idx, int r);*	添加一个资源。
*add_index_double(zval *array, uint idx, double d);*	添加一个浮点值。
*add_index_string(zval *array, uint idx, char *str, int duplicate);*	添加一个字符串。duplicate 用于表明这个字符串是否要被复制到 Zend 的内部内存。
*add_index_stringl(zval *array, uint idx, char *str, uint length, int duplicate);*	添加一个指定长度的字符串。其余跟 add_index_string () 相同。
*add_index_zval(zval *array, uint idx, zval *value);*	添加一个 zval 结构。这在添加另外一个数组、对象或流等数据时会很有用。

表 3.10 用于索引数组的 API 第二部分

函数	说明
*add_next_index_long(zval *array, long n);*	添加一个长整型元素。
*add_next_index_unset(zval *array);*	添加一个 unset 元素。
*add_next_index_bool(zval *array, int b);*	添加一个布尔值。
*add_next_index_resource(zval *array, int r);*	添加一个资源。
*add_next_index_double(zval *array, double d);*	添加一个浮点值。
*add_next_index_string(zval *array, char *str, int duplicate);*	添加一个字符串。duplicate 用于表明这个字符串是否要被复制到 Zend 的内部内存。

*add_next_index_stringl(zval *array, char *str, uint length, int duplicate);*	添加一个指定长度的字符串。其余跟 add_next_index_string () 相同。
*add_next_index_zval(zval *array, zval *value);*	添加一个 zval 结构。 这在添加另外一个数组、对象或流等数据时会很有用。

所有这些函数都是对 Zend 内部 hash API 的一种友好抽象。因此，若你愿意，你大可直接使用那些 hash API 进行操作。比方说，假如你已经有了一个 zval 容器并想把它插入到一个数组，那么你就可以直接使用 **zend_hash_update()** 来把它添加到一个关联数组（例 3.11 给关联数组添加一个元素）或索引数组（例 3.12 给索引数组添加一个元素）。

例 3.11 给关联数组添加一个元素

```
zval *new_array, *new_element;
char *key = “element_key”;
MAKE_STD_ZVAL(new_array);
MAKE_STD_ZVAL(new_element);
array_init(new_array);
ZVAL_LONG(new_element, 10);
if(zend_hash_update(new_array->value.ht, key, strlen(key) + 1, (void *)&new_element,
sizeof(zval *), NULL) == FAILURE)
{
    // do error handling here
}
```

例 3.12 给索引数组添加一个元素

```
zval *new_array, *new_element;
int key = 2;
MAKE_STD_ZVAL(new_array);
MAKE_STD_ZVAL(new_element);
array_init(new_array);
ZVAL_LONG(new_element, 10);
if(zend_hash_index_update(new_array->value.ht, key, (void *)&new_element, sizeof(zval *), NULL)
== FAILURE)
{
```

```
    // do error handling here
}
```

如果还想模拟下 **add_next_index_*()** ，那可以这么做：

```
zend_hash_next_index_insert(ht, zval **new_element, sizeof(zval *), NULL)
```

注意：如果要从函数里面返回一个数组，那就必须首先对预定义变量 return_value （return_value 是我们导出函数中的一个预定义参数，用来存储返回值）使用一下 **array_init()** 函数。不过倒不必对其使用 MAKE_STD_ZVAL 。

提示：为了避免一遍又一遍地书写 new_array->value.ht，我们可以用 *HASH_OF(new_array)* 来代替。而且出于兼容性和风格上的考虑，我们也推荐您这么做。

对象

既然对象可以被转换成数组（反之亦然），那么你可能已经猜到了两者应该具有很多相似之处。实际上，对象就是使用类似的函数进行操作的，所不同的是创建它们时所用的 API。

我们可以调用 **object_init()** 函数来初始化一个对象：

```
zval *new_object;
MAKE_STD_ZVAL(new_object);
if(object_init(new_object) != SUCCESS)
{
    // do error handling here
}
```

可以使用“表 3.11 用于创建对象的 API”来给对象添加一些成员。

表 3.11 用于创建对象的 API

函数	说明
*add_property_long(zval *object, char *key, long l);*	添加一个长整型类型的属性值。
*add_property_unset(zval *object, char *key);*	添加一个 unset 类型的属性值。
*add_property_bool(zval *object, char *key, int b);*	添加一个布尔类型的属性值。

*add_property_resource(zval *object, char *key, long r);*	添加一个资源类型的属性值。
*add_property_double(zval *object, char *key, double d);*	添加一个浮点类型的属性值。
*add_property_string(zval *object, char *key, char *str, int duplicate);*	添加一个字符串类型的属性值。
*add_property_stringl(zval *object, char *key, char *str, uint length, int duplicate);*	添加一个指定长度的字符串类型的属性值，速度要比 *add_property_string()* 函数快，而且是二进制安全的。
*add_property_zval(zval *obect, char *key, zval *container);*	添加一个 *zval* 结构的属性值。 这在添加另外一个数组、对象等数据时会很有用。

资源

资源是 PHP 中一种比较特殊的数据类型。“资源”这个词其实并不特指某些特殊类型的数据，事实上，它指的是一种可以维护任何类型数据信息方法的抽象。所有的资源均保存在一个 Zend 内部的资源列表当中。列表中的每份资源都有一个指向可以表明其种类的类型定义的指针。Zend 在内部统一管理所有对资源的引用。直接访问一个资源是不大可能的，你只能通过提供的 API 来对其进行操作。某个资源一旦失去引用，那就会触发调用相应的析构函数。

举例来说，数据库连接和文件描述符就是一种资源。MySQL 模块中就有其“标准”实现。当然其他模块（比如 Oracle 模块）也都用到了资源。

注意：

实际上，一个资源可以指向函数中任何一种你所感兴趣的数据（比如指向一个结构等等）。并且用户也只能通过某个资源变量来将资源信息传递给相应的函数。

要想创建一个资源你必须先注册一个这个资源的析构函数。这是因为 Zend 需要了解当你把某些数据存到一个资源里后，如果不再需要这份资源时该如何将其释放。这个析构函数会在释放资源（无论是手工释放还是自动释放）时被 Zend 依次调用。析构函数注册后，Zend 会返回一个此种**资源类型句柄**。这个句柄会在以后任何访问此种类型的资源的时候被用到，而且这个句柄绝大部分时间都保存在扩展的全局变量里面。这里你不需要担心线程安全方面的问题，因为你只是需要在模块初始化注册一次就行了。

下面是这个用于注册资源析构函数的 Zend 函数定义：

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

你或许已经注意到了，在该函数中我们需要提供两种不同的资源析构函数：一种是普通资源的析构函数句柄，一种是持久化资源的析构函数句柄。持久化资源一般用于诸如数据库连接等这类情况。在注册资源时，这两个析构函数至少得提供一个，另外一个析构函数可简单地设为 NULL。

zend_register_list_destructors_ex() 接受以下几个参数：

ld	普通资源的析构函数。
ld	持久化资源的析构函数。
type_name	为你的资源指定一个名称。在 PHP 内部为某个资源类型起个名字这是个好习惯（当然名字不能重复）。用户调用 var_dump($resource) 时就可取得该资源的名称。
module_number	这个参数在你模块的 PHP_MINIT_FUNCTION 函数中会自动定义，因此你大可将其忽略。

返回值是表示该**资源类型**的具有唯一性的整数标识符，即**资源类型句柄**。

资源（不论是不是持久化资源）的析构函数都必须具有以下的函数原型：

```
void resource_destruction_handler(zend_rsrc_list_entry *rsrc TSRMLS_DC);
```

参数 *rsrc* 指向一个 zend_rsrc_list_entry 结构：

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

成员 *void *ptr* 才真正指向你的资源。

现在我们就知道该怎么开始了。我们先定义一个将要注册到 Zend 内部的资源类型 my_resource，这个类型的结构很简单，只有两个整数成员：

```
typedef struct {
    int resource_link;
    int resource_type;
} my_resource;
```

接着我们再定义一下这种资源的析构函数。这个析构函数大致上是以下这个样子：

```
void my_destruction_handler(zend_rsrc_list_entry *rsrc TSRMLS_DC) {
    // 先将无类型指针转换为我们的资源类型指针
     my_resource *my_rsrc = (my_resource *) rsrc->ptr;
```

```
    // 现在我们就可以随意处理这些资源了：像关闭文件、释放内存等等。
    // 当然也不要忘了释放资源本身所占用的内存！
    do_whatever_needs_to_be_done_with_the_resource(my_rsrc);
}
```

注意：

有一个很重要的事情必须要提一下：如果你的资源是一个比较复杂的结构，比如包含有你在运行时所申请内存的指针等，那你就必须在释放资源本身前**先**释放它们！

OK。现在我们定义了

1. 我们的资源是什么样子；
2. 我们资源的析构函数是什么样子。

那么，我们还需要做哪些工作呢？我们还需要：

1. 创建一个在整个扩展范围内有效的全局变量用于保存资源类型句柄，这样就可以在每个需要它的函数中都能访问到它；
2. 给我们的资源类型定义一个名称；
3. 完成前面定义的资源析构函数；
4. 最后注册这个析构函数。

```
// 在你扩展的某个地方定义一个表示资源类型的变量
static int le_myresource;

// 给我们的资源起个名字是个很不错的习惯
#define le_myresource_name    ”My type of resource”

[…]

// 现在完成我们资源的析构函数
void my_destruction_handler(zend_rsrc_list_entry *rsrc TSRMLS_DC) {
    my_resource *my_rsrc = (my_resource *) rsrc->ptr;
    do_whatever_needs_to_be_done_with_the_resource(my_rsrc);
}

[…]
```

```
PHP_MINIT_FUNCTION(my_extension) {
    // 注意 'module_number' 已经在 PHP_MINIT_FUNCTION() 函数中被定义过了
    le_myresource = zend_register_list_destructors_ex(my_destruction_handler, NULL,
le_myresource_name, module_number);

    // 然后你可以在这里注册一些附加资源、初始化全局变量、常量等等。
}
```

注册完这种资源的析构函数后，要真正注册一个资源（实例），我们可以使用 **zend_register_resource()** 函数或使用 **ZEND_REGISTER_RESOURE()** 宏。这两个的定义可以在 zend_list.h 中找到。尽管两者的参数定义都是一一对应的，但使用宏通常可以得到更好的前向兼容性：

```
int ZEND_REGISTER_RESOURCE(zval *rsrc_result, void *rsrc_pointer, int rsrc_type);
```

rsrc_result	这是一个初始化过 zval * 容器。
rsrc_pointer	指向所保存的资源。
rsrc_type	这个参数就是你在注册函数析构函数时返回的资源类型句柄。对上面的代码来说就是 le_myresource le_myresource。

返回值就是表示这个资源（实例）的具有唯一性的整数。

那么在我们注册这个资源（实例）时究竟发生了什么事呢？函数会从 Zend 内部某个列表取得一个空闲空间，然后将资源指针及类型保存到这个空间。最后这个空闲空间的索引被简单地保存在给定的 *zval* * 容器里面：

```
rsrc_id = zend_list_insert(rsrc_pointer, rsrc_type);
    if (rsrc_result) {
        rsrc_result->value.lval = rsrc_id;
        rsrc_result->type = IS_RESOURCE;
    }
    return rsrc_id;
```

返回值 *rsrc_id* 就唯一性地标识了我们新注册得到的那个资源。你可以使用宏 *RETURN_RESOURE* 来将其返回给用户：

```
RETURN_RESOURCE(rsrc_id)
```

注意：

如果你想立刻把这个资源返回给用户，那你就应该把 return_value 作为那个 zval * 容器。这也是我们推荐的一种编程实践。

Zend 引擎从现在就会开始跟踪所有对这个资源的引用。一旦对这个资源的引用全都不存在了，那么你在前面为这个资源所注册的析构函数就会被调用。这样做的好处就是你不用担心会在你的模块里面引入内存泄漏－你只需要把你调用脚本中所有需要分配的内存都注册成资源即可。这样一来，一旦脚本认为不再需要它们的时候，Zend 就会找到它们然后再通知你（这就是 callback，译注）。

现在用户已经通过在某处传入到你函数的参数拿到了他的资源。*zval* * 容器中的 *value.lval* 包含了你资源的标识符，然后他就可以宏 *ZEND_FETCH_RESOURCE* 来获取资源了：

ZEND_FETCH_RESOURCE(rsrc, rsrc_type, rsrc_id, default_rsrc_id, resource_type_name, resource_type)

rsrc	这个指针将指向你前面已经声明过的资源。
rsrc_type	这个参数用以表明你你想要把前面参数的那个指针转换为何种类型。比如 myresource * 等等。
rsrc_id	这个是用户传进你函数的那个 *zval *container* 的地址。 假如给出的是 zval *z_resource ，那么此处就应该是 *&z_resource*。
default_rsrc_id	这个参数表明假如没有取到资源时默认指定的资源标识符。通常为 -1。
resource_type_name	所请求的资源类型资源类型名称。当不能找到资源时，就用这个字符串去填充系统由于维护而抛出的错误信息。
resource_type	这个可以取回在注册资源析构函数时返回的资源类型。在本例就是 *le_myresource*。

这个宏没有返回值。这对开发人员可能会方便了点。不过还是要注意添加 TSRM 参数和确认一下是否取回了资源。如果在接收资源时出现了问题，那它就会抛出一个警告信息并且会立刻从当前函数返回，其返回值为 NULL。

如果想从列表强行删除一个资源，可以使用 **zend_list_delete()** 函数。当然也可以强行增加引用计数，如果你知道你正在创建一个指向已分配内存资源的引用（比如说你可能想重用一个默认的数据库连接）。对于这种情况你可以使用函数 **zend_list_addref()** 。想要查找一个已分配内存的资源，请使用 **zend_list_find()** 函数。关于这些操作的完整 API 请参见 *zend_list.h*。

自动创建全局变量的宏

作为我们早期所谈论的一些宏的补充，还有一些宏可以让我们很方便的创建全局变量。了解了它们，我们在引入一些全局标识时就会感觉很爽，不过这个习惯可能会不太好。在“表 3.12 创建全局变量的宏”中描述了完成这些任务所用到的正确的宏。它们不需要申请任何 zval 容器，你只需简单地提供一个变量名和其值即可。

表 3.12 创建全局变量的宏

宏	说明
SET_VAR_STRING(name, value)	新建一个字符串变量。
SET_VAR_STRINGL(name, value, length)	新建一个指定长度的字符串变量。这个宏要比 *SET_VAR_STRING* 快而且还是二进制安全的。
SET_VAR_LONG(name, value)	新建一个长整型变量。
SET_VAR_DOUBLE(name, value)	新建一个双精度变量。

创建常量

Zend 支持创建真正的常量。访问常量时不需要 *$* 前缀，而且常量是全局有效的。比如 *TRUE* 和 *FALSE* 这两个常量。

要想创建一个常量，你可以使用“表 3.13 创建常量的宏”中所列举的宏来完成这项工作。所有的宏在创建常量时都必须指定一个名称和值。

你还可以为常量指定一个特别的标识：

- *CONST_CS* - 这个常量的名称是大小写敏感的；
- *CONST_PERSISTENT* - 这个常量是持久化的。换句话说，当携带这个常量的进程关闭时这个常量在剩下的请求中还依然有效，并不会被“遗忘”。

可以使用二进制的“或（OR）”操作来使用其中的一个或两个标识：

```
// 注册一个长整型常量
REGISTER_LONG_CONSTANT("NEW_MEANINGFUL_CONSTANT", 324, CONST_CS |
CONST_PERSISTENT);
```

我们提供有两种不同类型的宏，分别是 *REGISTER_*_CONSTANT* 和 *REGISTER_MAIN_*_CONSTANT*。第一种类型在创建常量时只会绑定到当前模块。一旦注册这个模块的常量从内存中卸载，那么这个常量也就会随即消逝。第二种类型创建的变量将会独立于该模块，始终保存在符号表中。

表 3.13 创建常量的宏

宏	说明

REGISTER_LONG_CONSTANT(name, value, flags) *REGISTER_MAIN_LONG_CONSTANT(name, value, flags)*	新建一个长整型常量。
REGISTER_DOUBLE_CONSTANT(name, value, flags) *REGISTER_MAIN_DOUBLE_CONSTANT(name, value, flags)*	新建一个双精度型常量。
REGISTER_STRING_CONSTANT(name, value, flags) *REGISTER_MAIN_STRING_CONSTANT(name, value, flags)*	新建一个字符串常量。给定的字符串的空间必须在 Zend 内部内存。
REGISTER_STRINGL_CONSTANT(name, value, length, flags) *REGISTER_MAIN_STRINGL_CONSTANT(name, value, length, flags)*	新建一个指定长度的字符串常量。同样，这个给定的字符串的空间也必须在 Zend 内部内存。

迟早你会遇到把一个 *zval* 容器的内容赋给另外一个 *zval* 容器的情况。不过可别想当然，这事说起来容易做起来可有点难度。因为 *zval* 容器不但包含了类型信息，而且还有对 Zend 内部数据的一些引用。比如，数组以及对象等依据其大小大都或多或少包含了一些哈希表结构。而我们在将一个 *zval* 赋给另外一个 *zval* 时，通常都没有复制这些哈希表本身，复制的只是这些哈希表的引用而已。

为了能够正确复制这些复杂类型的数据，我们可以使用“拷贝构造函数（*copy constructor*）”来完成这项工作。拷贝构造函数在某些为了可以复制复杂类型数据而支持操作符重载的语言中有着代表性的应用。如果你在这种语言中定义了一个对象，那你就可能想为其重载（*Overloading*）一下“＝”操作符，这个操作符通常用于将右值（操作符右边表达式的值）赋给左值（操作符左边表达式的值）。

“重载”就意味着将给予这个操作符另外一种不同的含义，它通常会把这个操作符跟某个函数调用关联起来。当这个操作符作用在一个对象上时，与之关联的函数就将会被调用，同时该操作符的左值和右值也会作为该函数的参数一并传入。这样，这个函数就可以完成“＝”操作符想要完成的事情（一般是某些额外数据的复制）。

这些“额外数据的复制”对 PHP 的 *zval* 容器来说也是很有必要的。对于数组来说，“额外数据的复制”就是指另外再重建和复制那些与该数组有关的哈希表（因为当初我们复制 *zval* 时复制的仅仅是这些哈希表的指针）。而对字符串来说，“额外数据的复制”就意味着我们必须重新为字符串值去申请空间。如此类推。

Zend Engine 会调用一个名为 **zval_copy_ctor()**（在以前的 PHP 版本中这个函数叫做 **pval_copy_constructor()** ）的函数来完成这项工作。

下面这个示例为我们展示了这样一个函数：它接收一个复杂类型的参数，在对其进行一定的修改后把它作为结果返回给 PHP：

```
zval *parameter;
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &parameter) == FAILURE)
   return;
```

```
}
// 在这对参数做一定的修改

……
// 返回修改后的容器
*return_value = *parameter;
zval_copy_ctor(return_value);
```

函数的头一部分没什么可说的，只是一段很平常的接收参数的代码而已。不过在对这个参数进行了某些修改后就变得有趣起来了：先是把 *parameter* 容器值赋给了（预先定义好的）*return_value* 容器，然后为了能够真正复制这个容器，我们便调用了拷贝构造函数。这个拷贝构造函数能够直接处理它的参数，处理成功则返回 *SUCCESS*，否则返回 *FAILURE*。

在这个例子当中如果你忘了调用这个拷贝构造函数，那么 *parameter* 和 *return_value* 就会分别指向同一个 Zend 内部数据，也就是说返回值 *return_value* 非法指向了一个数据结构。当你修改了参数 *parameter* 时这个函数的返回值就可能会受到影响。因此为了创建一个独立的拷贝，我们必须调用这个函数。

在 Zend API 中还有一个与拷贝构造函数相对应的拷贝析构函数：**zval_dtor()**，它做的工作正好与拷贝构造函数相反。

关于扩展内函数到 PHP 脚本的返回值我们前面谈得比较少，这一节我们就来详细说一下。任何函数的返回值都是通过一个名为 *return_value* 的变量传递的。这个变量同时也是函数中的一个参数。这个参数总是包含有一个事先申请好空间的 *zval* 容器，因此你可以直接访问其成员并对其进行修改而无需先对 *return_value* 执行一下 *MAKE_STD_ZVAL* 宏指令。

为了能够更方便从函数中返回结果，也为了省却直接访问 *zval* 容器内部结构的麻烦，ZEND 提供了一大套宏命令来完成相关的这些操作。这些宏命令会自动设置好类型和数值。“表 3.14 从函数直接返回值的宏”和“表 3.15 设置函数返回值的宏”列出了这些宏和对应的说明。

注意：使用“表 3.14 从函数直接返回值的宏”会自动携带结果从当前函数返回。而使用“表 3.15 设置函数返回值的宏”则只是设置了一下函数返回值，并不会马上返回。

表 3.14 从函数直接返回值的宏

宏	说明
RETURN_RESOURCE(resource)	返回一个资源。
RETURN_BOOL(bool)	返回一个布尔值。
RETURN_NULL()	返回一个空值。
RETURN_LONG(long)	返回一个长整数。
RETURN_DOUBLE(double)	返回一个双精度浮点数。
RETURN_STRING(string, duplicate)	返回一个字符串。duplicate 表示这个字符是否使

	用 **estrdup()** 进行复制。
RETURN_STRINGL(string, length, duplicate)	返回一个定长的字符串。其余跟 *RETURN_STRING* 相同。这个宏速度更快而且是二进制安全的。
RETURN_EMPTY_STRING()	返回一个空字符串。
RETURN_FALSE	返回一个布尔值假。
RETURN_TRUE	返回一个布尔值真。

表 3.15 设置函数返回值的宏

宏	说明
RETVAL_RESOURCE(resource)	设定返回值为指定的一个资源。
RETVAL_BOOL(bool)	设定返回值为指定的一个布尔值。
RETVAL_NULL	设定返回值为空值
RETVAL_LONG(long)	设定返回值为指定的一个长整数。
RETVAL_DOUBLE(double)	设定返回值为指定的一个双精度浮点数。
RETVAL_STRING(string, duplicate)	设定返回值为指定的一个字符串，duplicate 含义同 *RETURN_STRING*。
RETVAL_STRINGL(string, length, duplicate)	设定返回值为指定的一个定长的字符串。其余跟 *RETVAL_STRING* 相同。这个宏速度更快而且是二进制安全的。
RETVAL_EMPTY_STRING	设定返回值为空字符串。
RETVAL_FALSE	设定返回值为布尔值假。
RETVAL_TRUE	设定返回值为布尔值真。

如果需要返回的是像数组和对象这样的复杂类型的数据，那就需要先调用 **array_init()** 和 **object_init()**，也可以使用相应的 hash 函数直接操作 *return_value*。由于这些类型主要是由一些杂七杂八的东西构成，所以对它们就没有了相应的宏。

就像我们在脚本中使用 **print()** 函数一样，我们也经常需要从扩展向输出流输出一些信息。在这方面一比如输出警告信息、**phpinfo()** 中对应的信息等一般性任务一PHP 也为我们提供了一系列函数。这一节我们就来详细地讨论一下它们。

zend_printf()

zend_printf() 功能跟 **printf()** 差不多，唯一不同的就是它是向 Zend 的输出流提供信息。

zend_error()

zend_error() 用于创建一个错误信息。这个函数接收两个参数：第一个是错误类型（见 *zend_error.h*），第二个是错误的提示消息。

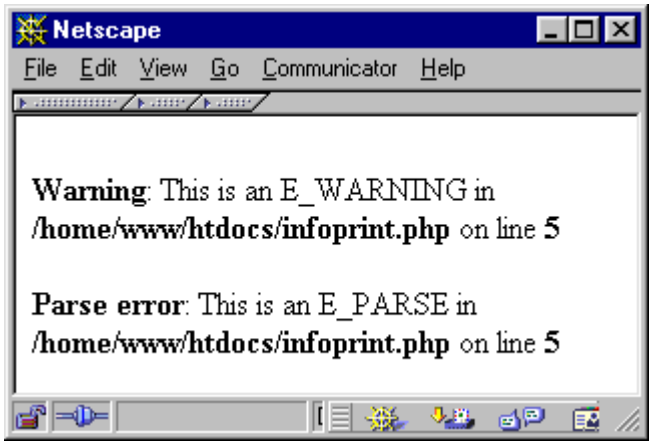
zend_error(E_WARNING, "This function has been called with empty arguments");

“表 3.16 Zend 预定义的错误信息类型”列出了一些可能的值（在 PHP 5.0 及以上版本中又增加了一些错误类型，可参见 *zend_error.h*，译注）。这些值也可以用在 *php.ini* 里面，这样你的错误信息将会依照 *php.ini* 里面的设置，根据不同的错误类型而被选择性地记录。

表 3.16 Zend 预定义的错误信息类型

错误类型	说明
E_ERROR	抛出一个错误，然后立即中止脚本的执行。
E_WARNING	抛出一个一般性的警告。脚本会继续执行。
E_NOTICE	抛出一个通知，脚本会继续执行。注意： 默认情况下 php.ini 会关闭显示这种错误。
E_CORE_ERROR	抛出一个 PHP 内核错误。通常情况下这种错误类型不应该被用户自己编写的模块所引用。
E_COMPILE_ERROR	抛出一个编译器内部错误。通常情况下这种错误类型不应该被用户自己编写的模块所引用。
E_COMPILE_WARNING	抛出一个编译器内部警告。通常情况下这种错误类型不应该被用户自己编写的模块所引用。

图 3.3 在浏览器中显示警告信息



向 phpinfo() 中输出信息

在创建完一个模块之后，你可能就会想往 **phpinfo()** 里面添加一些关于你自己模块的一些信息了（默认是只显示你的模块名）。PHP 允许你用 *ZEND_MINFO()* 函数向 **phpinfo()** 里面添加一段你自己模块

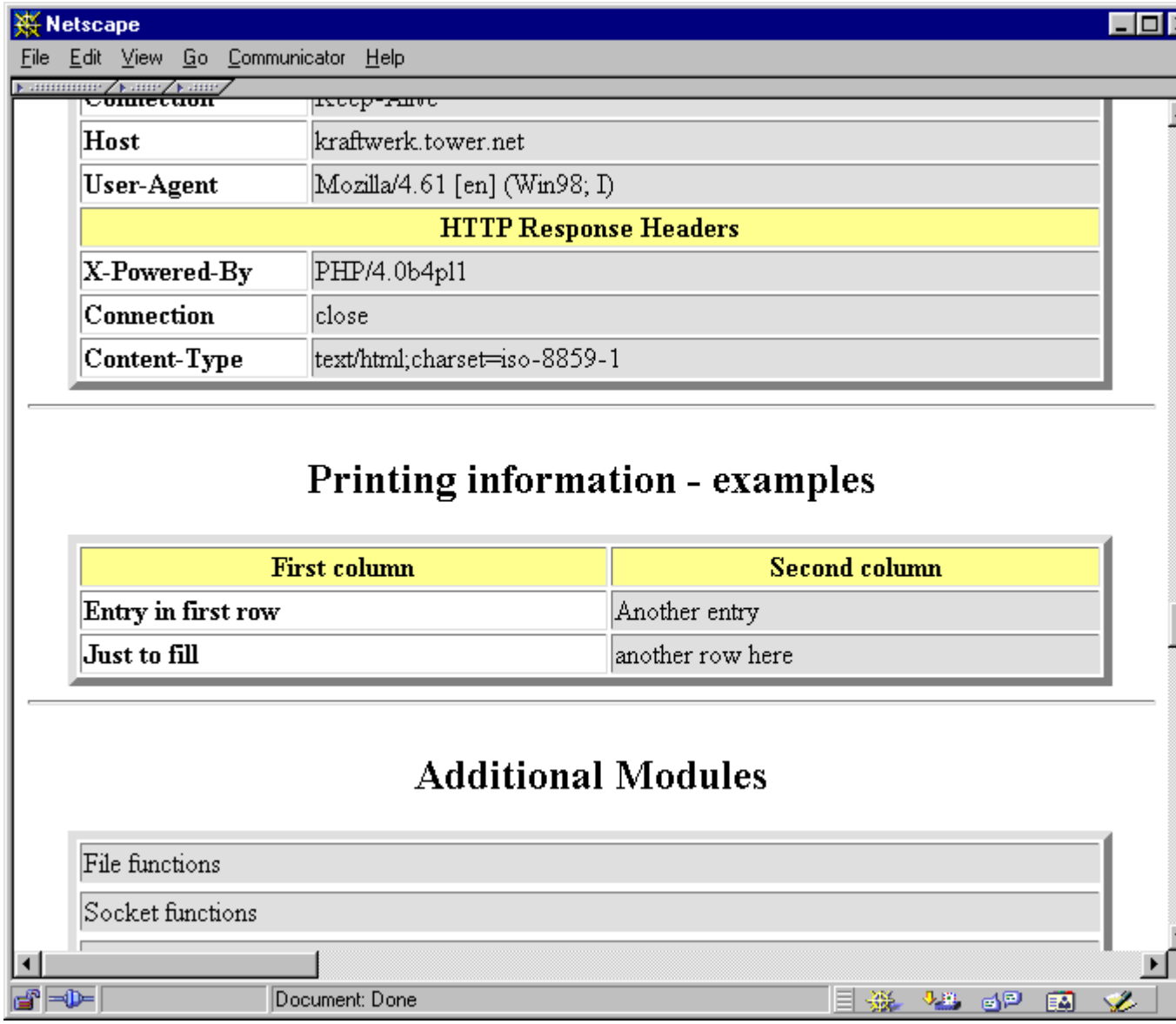
的信息。这个函数应该被放在模块描述块（见前文）部分，这样在脚本调用 **phpinfo()** 时模块的这个函数就会被自动调用。

如果你指定了 *ZEND_MINFO* 函数，**phpinfo()** 会自动打印一个小节，这个小节的头部就是你的模块名。其余的信息就需要你自己去指定一下格式并输出了。

一般情况下，你需要先调用一下 **php_info_print_table_start()**，然后再调用 **php_info_print_table_header()** 和 **php_info_print_table_row()** 这两个标准函数来打印表格具体的行列信息。这两个函数都以表格的列数（整数）和相应列的内容（字符串）作为参数。最后使用 **php_info_print_table_end()** 来结束打印表格。“例 3.13 源代码及其 **在 phpinfo()** 函数中的屏幕显示”向我们展示了某个样例和它的屏幕显示效果。

例 3.13 源代码及其 在 phpinfo() 函数中的屏幕显示

```
php_info_print_table_start();
php_info_print_table_header(2, “First column”, “Second column”);
php_info_print_table_row(2, “Entry in first row”, “Another entry”);
php_info_print_table_row(2, “Just to fill”, “another row here”);
php_info_print_table_end();
```



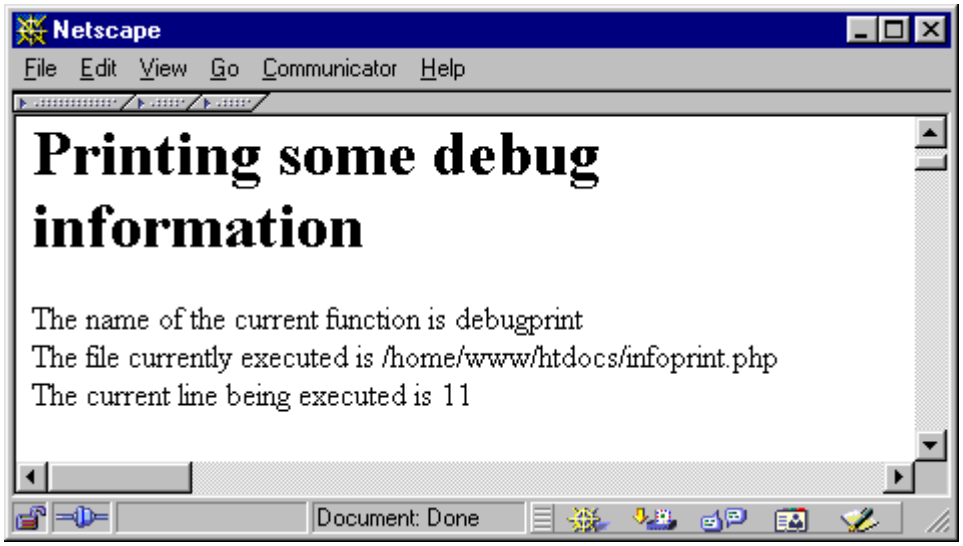
执行时信息

你还可以输出一些执行时信息，像当前被执行的文件名、当前正在执行的函数名等等。当前正在执行的函数名可以通过 **get_active_function_name()** 函数来获取。这个函数没有参数（译注：原文即是如此，事实上是跟后面提到的 **zend_get_executed_filename()** 函数一样需要提交 *TSRMLS_C* 宏参数，译注），返回值为函数名的指针。当前被执行的文件名可以由 **zend_get_executed_filename()** 函数来获得。这个函数需要传入 *TSRMLS_C* 宏参数来访问执行器全局变量。这个执行器全局变量对每个被 Zend 直接调用的函数都是有效的（因为 *TSRMLS_C* 是我们前文讨论过的参数宏 *INTERNAL_FUNCTION_PARAMETERS* 的一部分）。如果你想在其他函数中也访问这个执行器全局变量，那就需要现在那个函数中调用一下宏 *TSRMLS_FETCH()*。

最后你还可以通过 **zend_get_executed_lineno()** 函数来取得当前正在执行的那一行代码所在源文件中的行数。这个函数同样需要访问执行器全局变量作为其参数。关于这些函数的应用，请参阅“例 3.14 输出执行时信息”。

例 3.14 输出执行时信息

```
zend_printf("The name of the current function is %s<br>", get_active_function_name(TSRMLS_C));
zend_printf("The file currently executed is %s<br>", zend_get_executed_filename(TSRMLS_C));
zend_printf("The current line being executed is %i<br>", zend_get_executed_lineno(TSRMLS_C));
```



启动函数和关闭函数会在模块的（载入时）初始化和（卸载时）反初始化时被调用，而且只调用这一次。正如我们在本章前面（见 Zend 模块描述块的说明）所提到的，它们是模块和请求启动和关闭时所发生的事件。

模块启动/关闭函数会在模块加载和卸载时被调用。请求启动/关闭函数会在每次处理一个请求时（也就是在执行一个脚本文件时）被调用。

对于动态加载的扩展而言，模块和请求的启动函数与模块和请求的关闭函数都是同时发生的（严格来说模块启动函数是先于请求启动函数被调用的，译注）。

可以用某些宏来声明和实现这些函数，详情请参阅前面的关于“Zend 模块声明”的讨论。

PHP 还允许你在你的模块里面调用一些一些用户定义的函数，这样在实现某些回调机制（比如在做一些数组的轮循（array walking）、搜索或设计一些简单的事件驱动的程序时）时会很方便。

我们可以通过调用 **call_user_function_ex()** 来调用用户函数。它需要你即将访问函数表的指针、这个对象的指针（假如你访问的是类的一个方法的话），函数名、返回值、参数个数、具体的参数数组和一个是否需要进行 zval 分离的标识（这个函数原型已经“过时”了，至少是从 PHP 4.2 开始这个函数就追加了

一个 *HashTable *symbol_table* 参数。下面所列举的函数原型更像是 **call_user_function ()** 的声明。译注）。

```
ZEND_API int call_user_function_ex(HashTable *function_table, zval *object,
zval *function_name, zval **retval_ptr_ptr,
int param_count, zval **params[],
int no_separation);
```

需要注意的是你不必同时指定 *function_table* 和 *object* 这两个参数，只需要指定其中一个就行了。不过如果你想调用一个方法的话，那你就必须提供一个包含此方法的对象。这时 **call_user_function()** 会自动将函数表设置为当前这个对象的函数表。而对于其他情况，只需要设定一下 *function_table* 而把 *object* 设为 *NULL* 就行了。

一般情况下，默认的函数表是包含有所有函数的“根”函数表。这个函数表是编译器全局变量的一部分，你可以通过 *CG()* 宏来访问它。如果想把编译器全局变量引入你的函数，只需先执行一下 *TSRMLS_FETCH* 宏就可以了。

而调用的函数名是保存在一个 *zval* 容器内的。猛一下你可能会感到好奇，但其实这是很合乎逻辑的。想想看，既然我们在脚本中的大部分时间都是在接收一个函数名作为参数，并且这个参数还是被转换成（或被包含在）一个 *zval* 容器。那还不如现在就直接把这个 *zval* 容器传送给函数，只是这个 *zval* 容器的类型必须为 *IS_STRING*。

下一个参数是返回值 *return_value* 的指针。这个容器的空间函数会自动帮你申请，所以我们无需手动申请，但在事后这个容器空间的销毁释放工作得由我们自己（使用 **zval_dtor()**）来做。

跟在 *return_value* 后面的是一个标识参数个数的整数和一个包含具体参数的数组。最后一个参数 *no_separation* 指明了函数是否禁止进行 zval 分离操作。这个参数应该总是设为 *0*，因为如果设为 *1* 的话那这个函数会节省一些空间但要是其中任何一个参数需要做 zval 分离时都会导致操作失败。

“例 3.15 调用用户函数”向我们展示如何去调用一个脚本中的用户函数。这段代码调用了一个我们模块所提供的 **call_userland()** 函数。模块中的 **call_userland()** 函数会调用脚本中一个名为它的参数的用户函数，并且将这个用户函数的返回值直接作为自己的返回值返回脚本。另外你可能注意到了我们在最后调用了析构函数。这个操作或许没有太大必要（因为这些值都应该是分离过的，对它们的赋值将会很安全），但这么做总没有什么坏处，说不定在某个关键时刻它成为我们的一道“免死金牌”。:D

例 3.15 调用用户函数

```
zval **function_name;
zval *retval;
```

```
if((ZEND_NUM_ARGS() != 1) || (zend_get_parameters_ex(1, &function_name) != SUCCESS))
{
    WRONG_PARAM_COUNT;
}
if((*function_name)->type != IS_STRING)
{
    zend_error(E_ERROR, "Function requires string argument");
}
TSRMLS_FETCH();
if(call_user_function_ex(CG(function_table), NULL, *function_name, &retval, 0, NULL, 0) !=
SUCCESS)
{
    zend_error(E_ERROR, "Function call failed");
}
zend_printf("We have %i as type\n", retval->type);
*return_value = *retval;
zval_copy_ctor(return_value);
zval_ptr_dtor(&retval);
```

调用脚本：

```
<?php
dl("call_userland.so");
function test_function()
{
    echo "We are in the test function!\n";
    return 'hello';
}
$return_value = call_userland("test_function");
echo "Return value: '$return_value'";
?>
```

上例将输出：

We are in the test function! We have 3 as type Return value: 'hello'

PHP4 重写了对初始化文件的支持。现在你可以直接在代码中指定一些初始化选项，然后在运行时读取和改变这些选项值，甚至还可以在选项值改变时接到相关通知。

如果想要为你的模块创建一个 .ini 文件的配置节，可以使用宏 *PHP_INI_BEGIN()* 来标识这个节的开始，并用 *PHP_INI_END()* 表示该配置节已经结束。然后在两者之间我们用 *PHP_INI_ENTRY()* 来创建具体的配置项。

```
PHP_INI_BEGIN()
PHP_INI_ENTRY("first_ini_entry",    "has_string_value", PHP_INI_ALL, NULL)
PHP_INI_ENTRY("second_ini_entry", "2",                  PHP_INI_SYSTEM,
OnChangeSecond)
PHP_INI_ENTRY("third_ini_entry",    "xyz",              PHP_INI_USER, NULL)
PHP_INI_END()
```

PHP_INI_ENTRY() 总共接收 4 个参数：配置项名称、初始值、改变这些值所需的权限以及在值改变时用于接收通知的函数句柄。配置项名称和初始值必须是一个字符串，即使它们是一个整数。

更改这些值所需的权限可以划分为三种：*PHP_INI_SYSTEM* 只允许在 php.ini 中改变这些值；*PHP_INI_USER* 允许用户在运行时通过像 *.htaccess* 这样的附加文件来重写其值；而 *PHP_INI_ALL* 则允许随意更改。其实还有第四种权限：*PHP_INI_PERDIR*，不过我们还暂时不能确定它有什么影响。（本段关于这几种权限的说明与手册中《附录 G php.ini 配置选项》一节的描述略有出入。根据译者自己查到的资料，相比之下还是《附录 G php.ini 配置选项》更为准确些。译注）

第四个参数是初始值被改变时接收通知的函数句柄。一旦某个初始值被改变，那么相应的函数就会被调用。这个函数我们可以用宏 *PHP_INI_MH* 来定义：

```
PHP_INI_MH(OnChangeSecond);                    // handler for ini-entry "second_ini_entry"
// specify ini-entries here
PHP_INI_MH(OnChangeSecond)
{
    zend_printf("Message caught, our ini entry has been changed to %s<br>", new_value);
    return(SUCCESS);
}
```

改变后的新值将会以字符串的形式并通过一个名为 *new_value* 变量传递给函数。要是再注意一下 *PHP_INI_MH* 的定义就会发现，我们实际上用到了不少参数：

```
#define PHP_INI_MH(name) int name(php_ini_entry *entry, char *new_value,
                                  uint new_value_length, void *mh_arg1,
                                  void *mh_arg2, void *mh_arg3)
```

这些定义都可以在 *php_ini.h* 文件里找到。可以发现，我们的通知接收函数可以访问整个配置项、改变后的新值以及它的长度和其他三个可选参数。这几个可选参数可以通过 *PHP_INI_ENTRY1*（携带一个附加

参数）、*PHP_INI_ENTRY2*（携带两个附加参数）、*PHP_INI_ENTRY3*（携带三个附加参数）等宏来加以指定。

关于值改变的通知函数应该被用来本地缓存一些初始花选项以便可以更快地对其访问或被用来从事一个值发生改变时所要求完成的任务。比如要是一个模块对一个主机常量进行了连接，而这时有人改变了主机名，那么就需要自动地关闭原来的连接，并尝试进行新的连接。

可以使用“表 3.17 PHP 中用以访问初始化配置项的宏”来访问初始化配置项：

表 3.17 PHP 中用以访问初始化配置项的宏

宏	说明
INI_INT(name)	将配置项 name 的当前值以长整数返回。
INI_FLT(name)	将配置项 name 的当前值以双精度浮点数返回。
INI_STR(name)	将配置项 name 的当前值以字符串返回。 注意：这个字符串不是复制过的字符串，而是直接指向了内部数据。如果你需要进行进一步的访问的话，那就需要再进行复制一下。
INI_BOOL(name)	将配置项 name 的当前值以布尔值返回。(返回值被定义为 *zend_bool*,也就是说是一个 *unsigned char*)。
INI_ORIG_INT(name)	将配置项 name 的初始值以长整型数返回。
INI_ORIG_FLT(name)	将配置项 name 的初始值以双精度浮点数返回。
INI_ORIG_STR(name)	将配置项 name 的初始值以字符串返回。 注意：这个字符串不是复制过的字符串，而是直接指向了内部数据。如果你需要进行进一步的访问的话，那就需要再进行复制一下。
INI_ORIG_BOOL(name)	将配置项 name 的初始值以布尔值返回。(返回值被定义为 *zend_bool*,也就是说是一个 *unsigned char*)。

最后，你还得把整个初始化配置项引入 PHP。这项工作可以在模块的起始/结束函数中使用宏 *REGISTER_INI_ENTRIES()* 和 *UNREGISTER_INI_ENTRIES()* 来搞定。

```
ZEND_MINIT_FUNCTION(mymodule)
{
    REGISTER_INI_ENTRIES();
}
ZEND_MSHUTDOWN_FUNCTION(mymodule)
{
    UNREGISTER_INI_ENTRIES();
}
```

现在你已经掌握了很多关于 PHP 的知识了。你已经知道了如何创建一个动态加载的模块或被静态连接的扩展。你还知道了在 PHP 和 Zend 的内部变量是如何储存的，以及如何创建和访问这些变量。另外你也知道了很多诸如输出信息文本、自动将变量引入符号表等一系列工具函数的应用。

尽管这一章常常有点“参考”的意味，但我们还是希望它能给你一些关于如何开始编写自己的扩展这方面的知识。限于篇幅，我们不得不省略了很多东西。我们建议你花些时间仔细研究一下它的头文件和一些模块（尤其是 ext/standard 目录下的一些文件以及 MySQL 模块，看一下这些众所周知的函数究竟是怎么实现的），看一下别人是怎么使用这些 API 函数的，尤其是那些本章没有提到的那些函数。

由 *buildconf* 处理的配置文件 *config.m4* 包含了所有在配置过程中所执行的指令。这些指令诸如包含测试包含所需的外部文件，像头文件、库文件等等。PHP 定义了一系列处理这类情况的宏，其中最常用的我们已经在“表 3.18 *config.m4* 中的 M4 宏”列了出来。

表 3.18 *config.m4* 中的 M4 宏

宏	说明
AC_MSG_CHECKING(message)	在执行 *configure* 命令时输出“checking <message>”等信息。
AC_MSG_RESULT(value)	取得 *AC_MSG_CHECKING* 的执行结果，一般情况下 *value* 应为 *yes* 或 *no*。
AC_MSG_ERROR(message)	在执行 *configure* 命令时输出一条错误消息 *message* 并中止脚本的执行。
AC_DEFINE(name,value,description)	向 *php_config.h* 添加一行定义： *#define name value // description* (这对模块的条件编译很有用。)
AC_ADD_INCLUDE(path)	添加一条编译器的包含路径，比如用于模块需

	要为头文件添加搜索路径。
AC_ADD_LIBRARY_WITH_PATH(libraryname,librarypath)	指定一个库的连接路径。
AC_ARG_WITH(modulename,description,unconditionaltest, conditionaltest)	这是一款比较强大的宏，用于将模块的描述 *description* 添加到 *“configure -help”*命令的输出里面。PHP 会检查当前执行的 *configure* 脚本里面有没有 *-with-<modulename>* 这个选项。 如果有则执行 *unconditionaltest* 语句(比如 *-with-myext=yes* 等)，此时，选项的值会被包含在 *$withval* 变量里面。否则就执行 *conditionaltest* 语句。
PHP_EXTENSION(modulename, [shared])	这个是配置你的扩展时 PHP **必定**调用的一个宏。你可以在模块名后面提供第二个参数，用来表明是否将其编译为动态共享模块。这会导致在编译时为你的源码提供一个 *COMPILE_DL_<module name>* 的定义。

下面（见表 3.19 访问 *zval* 容器的 API 宏）是一些引入到 Zend API 里面用于访问 *zval* 容器的 API 宏。

宏	指向
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_LVAL_P(zval)	(*zval).value.lval
Z_DVAL_P(zval)	(*zval).value.dval

Z_STRVAL_P(zval_p)	(*zval).value.str.val
Z_STRLEN_P(zval_p)	(*zval).value.str.len
Z_ARRVAL_P(zval_p)	(*zval).value.ht
Z_LVAL_PP(zval_pp)	(**zval).value.lval
Z_DVAL_PP(zval_pp)	(**zval).value.dval
Z_STRVAL_PP(zval_pp)	(**zval).value.str.val
Z_STRLEN_PP(zval_pp)	(**zval).value.str.len
Z_ARRVAL_PP(zval_pp)	(**zval).value.ht