28 Apr 09 用C/C++扩展你的PHP

- 本文地址: http://www.laruence.com/2009/04/28/719.html
- 文章转自: http://blog.csdn.net/taft/archive/2006/02/10/596291.aspx

```
原文出自: <PHP 5 Power Programming> Chapter 15
作者:Andi Gutmans, Stig Sæther Bakken, Derick Rethans
翻译:taft at wjl.cn(http://blog.csdn.net/taft/)
校对:laruence at yahoo.com.cn
最后更新日期:2009/04/29
```

简 介

英文版下载: PHP 5 Power Programming

PHP取得成功的一个主要原因之一是她拥有大量的可用扩展。web开发者无论有何种需求，这种需求最有可能在PHP发行包里找到。PHP发行包包括支持各种数据库，图形文件格式，压缩，XML技术扩展在内的许多扩展。

扩展API的引入使PHP3取得了巨大的进展，扩展API机制使PHP开发社区很容易的开发出几十种扩展。现在，两个版本过去了，API仍然和PHP3时的非常相似。扩展主要的思想是：尽可能的从扩展编写者那里隐藏PHP的内部机制和脚本引擎本身，仅仅需要开发者熟悉API。

有两个理由需要自己编写PHP扩展。第一个理由是：PHP需要支持一项她还未支持的技术。这通常包括包裹一些现成的C函数库，以便提供PHP接口。例如，如果一个叫FooBase的数据库已推出市场，你需要建立一个PHP扩展帮助你从PHP里调用FooBase的C函数库。这个工作可能仅由一个人完成，然后被整个PHP社区共享（如果你愿意的话）。第二个不是很普遍的理由是：你需要从性能或功能的原因考虑来编写一些商业逻辑。

如果以上的两个理由都和你没什么关系，同时你感觉自己没有冒险精神，那么你可以跳过本章。

本章教你如何编写相对简单的PHP扩展，使用一部分扩展API函数。对于大多数打算开发自定义PHP扩展开发者而言，它含概了足够的资料。学习一门编程课程的最好方法之一就是动手做一些极其简单的例子，这些例子正是本章的线索。一旦你明白了基础的东西，你就可以在互联网上通过阅读文档、原代码或参加邮件列表新闻组讨论来丰富自己。因此，本章集中在让你如何开始的话题。在UNIX下一个叫ext_skel的脚本被用于建立扩展的骨架，骨架信息从一个描述扩展接口的定义文件中取得。因此你需要利用UNIX来建立一个骨架。Windows开发者可以使用Windows ext_skel_win32.php代替ext_skel。

然而，本章关于用你开发的扩展编译PHP的指导仅涉及UNIX编译系统。本章中所有的对API的解释与UNIX和Windows下开发的扩展都有联系。

当你阅读完这章，你能学会如何

- 建立一个简单的商业逻辑扩展。
- 建议个C函数库的包裹扩展，尤其是有些标准C文件操作函数比如fopen()

快速开始

本节没有介绍关于脚本引擎基本构造的一些知识，而是直接进入扩展的编码讲解中，因此不要担心你无法立刻获得对扩展整体把握的感觉。假设你正在开发一个网站，需要一个把字符串重复n次的函数。下面是用PHP写的例子：

```php
function self_concat($string, $n){
    $result = "";
    for($i = 0; $i < $n; $i++){
        $result .= $string;
    }
    return $result;
}

self_concat("One", 3) returns "OneOneOne".

self_concat("One", 1) returns "One".
```

假设由于一些奇怪的原因，你需要时常调用这个函数，而且还要传给函数很长的字符串和大值n。这意味着在脚本里有相当巨大的字符串连接量和内存重新分配过程，以至显著地降低脚本执行速度。如果有一个函数能够更快地分配大量且足够的内存来存放结果字符串，然后把$string重复n次，就不需要在每次循环迭代中分配内存。

为扩展建立函数的第一步是写一个函数定义文件，该函数定义文件定义了扩展对外提供的函数原形。该例中，定义函数只有一行函数原形self_concat()：

```
string self_concat(string str, int n)
```

函数定义文件的一般格式是一个函数一行。你可以定义可选参数和使用大量的PHP类型，包括: bool, float, int, array等。

保存为myfunctions.def文件至PHP原代码目录树下。

该是通过扩展骨架(skeleton)构造器运行函数定义文件的时机了。该构造器脚本叫ext_skel，放在PHP原代码目录树的ext/目录下（PHP原码主目录下的README.EXT_SKEL提供了更多的信息）。假设你把函数定义保存在一个叫做myfunctions.def的文件里，而且你希望把扩展取名为myfunctions，运行下面的命令来建立扩展骨架

```
./ext_skel --extname=myfunctions --proto=myfunctions.def
```

这个命令在ext/目录下建立了一个myfunctions/目录。你要做的第一件事情也许就是编译该骨架，以便编写和测试实际的C代码。编译扩展有两种方法：

- 作为一个可装载模块或者DSO（动态共享对象）
- 静态编译到PHP

PHP扩展开发导图

因为第二种方法比较容易上手，所以本章采用静态编译。如果你对编译可装载扩展模块感兴趣，可以阅读PHP原代码根目录下的README.SELF-CONTAINED_EXTENSIONS文件。为了使扩展能够被编译，需要修改扩展目录ext/myfunctions/下的config.m4文件。扩展没有包裹任何外部的C库，你需要添加支持–enable-myfunctions配置开关到PHP编译系统里（–with-extension 开关用于那些需要用户指定相关C库路径的扩展）。可以去掉自动生成的下面两行的注释来开启这个配置。

```
./ext_skel --extname=myfunctions --proto=myfunctions.def
PHP_ARG_ENABLE(myfunctions, whether to enable myfunctions support,

[ --enable-myfunctions               Include myfunctions support])
```

现在剩下的事情就是在PHP原代码树根目录下运行./buildconf，该命令会生成一个新的配置脚本。通过查看./configure –help输出信息，可以检查新的配置选项是否被包含到配置文件中。现在，打开你喜好的配置选项开关和–enable-myfunctions重新配置一下PHP。最后的但不是最次要的是，用make来重新编译PHP。

ext_skel应该把两个PHP函数添加到你的扩展骨架了：打算实现的self_concat()函数和用于检测myfunctions 是否编译到PHP的confirm_myfunctions_compiled()函数。完成PHP的扩展开发后，可以把后者去掉。

```
<?php
print confirm_myfunctions_compiled("myextension");
?>
```

运行这个脚本会出现类似下面的输出：

```
"Congratulations! You have successfully modified ext/myfunctions

config.m4. Module myfunctions is now compiled into PHP."
```

另外，ext_skel脚本生成一个叫myfunctions.php的脚本，你也可以利用它来验证扩展是否被成功地编译到PHP。它会列出该扩展所支持的所有函数。

现在你学会如何编译扩展了，该是真正地研究self_concat()函数的时候了。
下面就是ext_skel脚本生成的骨架结构：

```
/* {{{ proto string self_concat(string str, int n)

*/

PHP_FUNCTION(self_concat)
{

char *str = NULL;

int argc = ZEND_NUM_ARGS();

int str_len;

long n;

if (zend_parse_parameters(argc TSRMLS_CC, "sl", &str, &str_len, &n) == FAILURE)

return;

php_error(E_WARNING, "self_concat: not yet implemented");

}

/* }}} */
```

自动生成的PHP函数周围包含了一些注释，这些注释用于自动生成代码文档和vi、Emacs等编辑器的代码折叠。函数自身的定义使用了宏PHP_FUNCTION()，该宏可以生成一个适合于Zend引擎的函数原型。逻辑本身分成语义各部分，取得调用函数的参数和逻辑本身。

为了获得函数传递的参数，可以使用zend_parse_parameters()API函数。下面是该函数的原型：

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

第一个参数是传递给函数的参数个数。通常的做法是传给它ZEND_NUM_ARGS()。这是一个表示传递给函数参数总个数的宏。第二个参数是为了线程安全，总是传递TSRMLS_CC宏，后面会讲到。第三个参数是一个字符串，指定了函数期望的参数类型，后面紧跟着需要随参数值更新的变量列表。因为PHP采用松散的变量定义和动态的类型判断，这样做就使得把不同类型的参数转化为期望的类型成为可能。例如，如果用户传递一个整数变量，可函数需要一个浮点数，那么zend_parse_parameters()就会自动地把整数转换为相应的浮点数。如果实际值无法转换成期望类型（比如整形到数组形），会触发一个警告。

下表列出了可能指定的类型。我们从完整性考虑也列出了一些没有讨论到的类型。

类型指定符	对应的C类型	描述
l	long	符号整数
d	double	浮点数
s	char *, int	二进制字符串，长度
b	zend_bool	逻辑型（1或0）
r	zval *	资源（文件指针，数据库连接等）
a	zval *	联合数组
o	zval *	任何类型的对象
O	zval *	指定类型的对象。需要提供目标对象的类类型
z	zval *	无任何操作的zval

为了容易地理解最后几个选项的含义，你需要知道zval是Zend引擎的值容器[1]。无论这个变量是布尔型，字符串型或者其他任何类型，其信息总会包含在一个zval联合体中。本章中我们不直接存取zval，而是通过一些附加的宏来操作。下面的是或多或少在C中的zval, 以便我们能更好地理解接下来的代码。

```
typedef union _zval{
     long lval;
     double dval;
     struct {
          char *val;
          int len;
     }str;

     HashTable *ht;
     zend_object_value obj;

}zval;
```

在我们的例子中，我们用基本类型调用zend_parse_parameters()，以本地C类型的方式取得函数参数的值，而不

是用zval容器。

为了让zend_parse_parameters()能够改变传递给它的参数的值，并返回这个改变值，需要传递一个引用。仔细查看一下self_concat()：

```
if (zend_parse_parameters(argc TSRMLS_CC, "sl", &str, &str_len, &n) == FAILURE)
return;
```

注意到自动生成的代码会检测函数的返回值FAILUER(成功即SUCCESS)来判断是否成功。如果没有成功则立即返回，并且由zend_parse_parameters()负责触发警告信息。因为函数打算接收一个字符串l和一个整数n，所以指定"sl"作为其类型指示符。s需要两个参数，所以我们传递参考char * 和 int (str 和 str_len)给zend_parse_parameters()函数。无论什么时候，记得总是在代码中使用字符串长度str_len来确保函数工作在二进制安全的环境中。不要使用strlen()和strcpy()，除非你不介意函数在二进制字符串下不能工作。二进制字符串是包含有nulls的字符串。二进制格式包括图象文件，压缩文件，可执行文件和更多的其他文件。"l"只需要一个参数，所以我们传递给它n的引用。尽管为了清晰起见，骨架脚本生成的C变量名与在函数原型定义文件中的参数名一样；这样做不是必须的，尽管在实践中鼓励这样做。

回到转换规则中来。下面三个对self_concat()函数的调用使str, str_len和n得到同样的值：

```
self_concat("321", 5);

self_concat(321, "5");

self_concat("321", "5");

str points to the string "321", str_len equals 3, and n equals 5.

str 指向字符串"321"，str_len等于3，n等于5。
```

在我们编写代码来实现连接字符串返回给PHP的函数前，还得谈谈两个重要的话题：内存管理、从PHP内部返回函数值所使用的API。

内存管理

用于从堆中分配内存的PHP API几乎和标准C API一样。在编写扩展的时候，使用下面与C对应（因此不必再解释）的API函数：

```
emalloc(size_t size);

efree(void *ptr);

ecalloc(size_t nmemb, size_t size);

erealloc(void *ptr, size_t size);

estrdup(const char *s);

estrndup(const char *s, unsigned int length);
```

在这一点上，任何一位有经验的C程序员应该象这样思考一下："什么？标准C没有strndup()？"是的，这是正确的，因为GNU扩展通常在Linux下可用。estrndup()只是PHP下的一个特殊函数。它的行为与estrdup()相似，但是可以指定字符串重复的次数（不需要结束空字符），同时是二进制安全的。这是推荐使用estrndup()而不是estrdup()的原因。

在几乎所有的情况下，你应该使用这些内存分配函数。有一些情况，即扩展需要分配在请求中永久存在的内存，从而不得不使用malloc()，但是除非你知道你在做什么，你应该始终使用以上的函数。如果没有使用这些内存函数，而相反使用标准C函数分配的内存返回给脚本引擎，那么PHP会崩溃。

这些函数的优点是：任何分配的内存在偶然情况下如果没有被释放，则会在页面请求的最后被释放。因此，真正的内存泄漏不会产生。然而，不要依赖这一机制，从调试和性能两个原因来考虑，应当确保释放应该释放的内存。剩下的优点是在多线程环境下性能的提高，调试模式下检测内存错误等。

还有一个重要的原因，你不需要检查这些内存分配函数的返回值是否为null。当内存分配失败，它们会发出E_ERROR错误，从而决不会返回到扩展。

从PHP函数中返回值

扩展API包含丰富的用于从函数中返回值的宏。这些宏有两种主要风格：第一种是RETVAL_type()形式，它设置了返回值但C代码继续执行。这通常使用在把控制交给脚本引擎前还希望做的一些清理工作的时候使用，然后再使用C的返回声明"return"返回到PHP；后一个宏更加普遍，其形式是RETURN_type()，他设置了返回类型，同时返回控制到PHP。下表解释了大多数存在的宏。

设置返回值并且结束函数	设置返回值	宏返回类型和参数
RETURN_LONG(l)	RETVAL_LONG(l)	整数
RETURN_BOOL(b)	RETVAL_BOOL(b)	布尔数(1或0)
RETURN_NULL()	RETVAL_NULL()	NULL

RETURN_DOUBLE(d)	RETVAL_DOUBLE(d)	浮点数
RETURN_STRING(s, dup)	RETVAL_STRING(s, dup)	字符串。如果dup为1，引擎会调用estrdup()重复s，使用拷贝。如果dup为0，就使用s
RETURN_STRINGL(s, l, dup)	RETVAL_STRINGL(s, l, dup)	长度为l的字符串值。与上一个宏一样，但因为s的长度被指定，所以速度更快。
RETURN_TRUE	RETVAL_TRUE	返回布尔值true。注意到这个宏没有括号。
RETURN_FALSE	RETVAL_FALSE	返回布尔值false。注意到这个宏没有括号。
RETURN_RESOURCE(r)	RETVAL_RESOURCE(r)	资源句柄。

完成self_concat()

现在你已经学会了如何分配内存和从PHP扩展函数里返回函数值，那么我们就能够完成self_concat()的编码：

```
/* {{{ proto string self_concat(string str, int n)

*/

PHP_FUNCTION(self_concat)

}

char *str = NULL;

int argc = ZEND_NUM_ARGS();

int str_len;

long n;

char *result; /* Points to resulting string */

char *ptr; /* Points at the next location we want to copy to */

int result_length; /* Length of resulting string */

if (zend_parse_parameters(argc TSRMLS_CC, "sl", &str, &str_len, &n) == FAILURE)

return;

/* Calculate length of result */

result_length = (str_len * n);

/* Allocate memory for result */

result = (char *) emalloc(result_length + 1);

/* Point at the beginning of the result */

ptr = result;

while (n--) {

/* Copy str to the result */

memcpy(ptr, str, str_len);

/* Increment ptr to point at the next position we want to write to */

ptr += str_len;

}

/* Null terminate the result. Always null-terminate your strings

even if they are binary strings */

*ptr = '\0';

/* Return result to the scripting engine without duplicating it*/

RETURN_STRINGL(result, result_length, 0);

}

/* }}} */
```

现在要做的就是重新编译一下PHP，这样就完成了第一个PHP函数。

让我门检查函数是否真的工作。在最新编译过的PHP树下执行[2]下面的脚本：

```
<?php
for ($i = 1; $i <= 3; $i++){
     print self_concat("ThisIsUseless", $i);
     print "\n";
}
?>
```

你应该得到下面的结果：

```
ThisIsUseless

ThisIsUselessThisIsUseless

ThisIsUselessThisIsUselessThisIsUseless
```

实例小结

你已经学会如何编写一个简单的PHP函数。回到本章的开头，我们提到用C编写PHP功能函数的两个主要的动机。第一个动机是用C实现一些算法来提高性能和扩展功能。前一个例子应该能够指导你快速上手这种类型扩展的开发。第二个动机是包裹三方函数库。我们将在下一步讨论。

包裹第三方的扩展

本节中你将学到如何编写更有用和更完善的扩展。该节的扩展包裹了一个C库，展示了如何编写一个含有多个互相依赖的PHP函数扩展。

动机

也许最常见的PHP扩展是那些包裹第三方C库的扩展。这些扩展包括MySQL或Oracle的数据库服务库，libxml2的XML技术库，ImageMagick 或GD的图形操纵库。

在本节中，我们编写一个扩展，同样使用脚本来生成骨架扩展，因为这能节省许多工作量。这个扩展包裹了标准C函数fopen(), fclose(), fread(), fwrite()和 feof().

扩展使用一个被叫做资源的抽象数据类型，用于代表已打开的文件FILE*。你会注意到大多数处理比如数据库连接、文件句柄等的PHP扩展使用了资源类型，这是因为引擎自己无法直接"理解"它们。我们计划在PHP扩展中实现的C API列表如下：

```
FILE *fopen(const char *path, const char *mode);

int fclose(FILE *stream);

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

int feof(FILE *stream);
```

我们实现这些函数，使它们在命名习惯和简单性上符合PHP脚本。如果你曾经向PHP社区贡献过代码，你被期望遵循一些公共习俗，而不是跟随C库里的API。并不是所有的习俗都写在PHP代码树的CODING_STANDARDS文件里。这即是说，此功能已经从PHP发展的很早阶段即被包含在PHP中，并且与C库API类似。PHP安装已经支持fopen(), fclose()和更多的PHP函数。

以下是PHP风格的API：

```
resource file_open(string filename, string mode)

file_open() //接收两个字符串（文件名和模式），返回一个文件的资源句柄。

bool file_close(resource filehandle)

file_close() //接收一个资源句柄，返回真/假指示是否操作成功。

string file_read(resource filehandle, int size)

file_read() //接收一个资源句柄和读入的总字节数，返回读入的字符串。

bool file_write(resource filehandle, string buffer)

file_write()   //接收一个资源句柄和被写入的字符串，返回真/假指示是否操作成功。

bool file_eof(resource filehandle)

file_eof()     //接收一个资源句柄，返回真/假指示是否到达文件的尾部。
```

因此，我们的函数定义文件——保存为ext/目录下的myfile.def——内容如下：

```
resource file_open(string filename, string mode)

bool file_close(resource filehandle)

string file_read(resource filehandle, int size)

bool file_write(resource filehandle, string buffer)

bool file_eof(resource filehandle)
```

下一步，利用ext_skel脚本在ext/ 原代码目录执行下面的命令：

```
./ext_skel --extname=myfile --proto=myfile.def
```

然后，按照前一个例子的关于编译新建立脚本的步骤操作。你会得到一些包含FETCH_RESOURCE()宏行的编译错误，这样骨架脚本就无法顺利完成编译。为了让骨架扩展顺利通过编译，把那些出错行[3]注释掉即可。

资源

资源是一个能容纳任何信息的抽象数据结构。正如前面提到的，这个信息通常包括例如文件句柄、数据库连接结构和其他一些复杂类型的数据。

使用资源的主要原因是因为：资源被一个集中的队列所管理，该队列可以在PHP开发人员没有在脚本里面显式地释放时可以自动地被释放。

举个例子，考虑到编写一个脚本，在脚本里调用mysql_connect()打开一个MySQL连接，可是当该数据库连接资源不再使用时却没有调用mysql_close()。在PHP里，资源机制能够检测什么时候这个资源应当被释放，然后在当前请求的结尾或通常情况下更早地释放资源。这就为减少内存泄漏赋予了一个“防弹”机制。如果没有这样一个机制，经过几次web请求后，web服务器也许会潜在地泄漏许多内存资源，从而导致服务器当机或出错。

注册资源类型

如何使用资源？Zend引擎让使用资源变地非常容易。你要做的第一件事就是把资源注册到引擎中去。使用这个API函数：

```
int zend_register_list_destructors_ex(rsrc_dtor_func_t ld, rsrc_dtor_func_t pld, char *
```

这个函数返回一个资源类型id，该id应当被作为全局变量保存在扩展里，以便在必要的时候传递给其他资源API。ld：该资源释放时调用的函数。pld用于在不同请求中始终存在的永久资源，本章不会涉及。type_name是一个具有描述性类型名称的字符串，module_number为引擎内部使用，当我们调用这个函数时，我们只需要传递一个已经定义好的module_number变量。

回到我们的例子中来：我们会添加下面的代码到myfile.c原文件中。该文件包括了资源释放函数的定义，此资源函数被传递给zend_register_list_destructors_ex()注册函数（资源释放函数应该提早添加到文件中，以便在调用zend_register_list_destructors_ex()时该函数已被定义）：

```
static void myfile_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC){
     FILE *fp = (FILE *) rsrc->ptr;
     fclose(fp);
}
```

把注册行添加到PHP_MINIT_FUNCTION()后，看起来应该如下面的代码：

```
PHP_MINIT_FUNCTION(myfile){
     /* If you have INI entries, uncomment these lines
     ZEND_INIT_MODULE_GLOBALS(myfile, php_myfile_init_globals,NULL);

     REGISTER_INI_ENTRIES();
     */

     le_myfile = zend_register_list_destructors_ex(myfile_dtor,NULL,"standard-c-file",

     return SUCCESS;
}
```

l 注意到le_myfile是一个已经被ext_skel脚本定义好的全局变量。

PHP_MINIT_FUNCTION()是一个先于模块（扩展）的启动函数，是暴露给扩展的一部分API。下表提供可用函数简要的说明。

函数声明宏	语义

PHP_MINIT_FUNCTION()	当PHP被装载时，模块启动函数即被引擎调用。这使得引擎做一些例如资源类型，注册INI变量等的一次初始化。
PHP_MSHUTDOWN_FUNCTION()	当PHP完全关闭时，模块关闭函数即被引擎调用。通常用于注销INI条目
PHP_RINIT_FUNCTION()	在每次PHP请求开始，请求前启动函数被调用。通常用于管理请求前逻辑。
PHP_RSHUTDOWN_FUNCTION()	在每次PHP请求结束后，请求前关闭函数被调用。经常应用在清理请求前启动函数的逻辑。
PHP_MINFO_FUNCTION()	调用phpinfo()时模块信息函数被呼叫，从而打印出模块信息。

新建和注册新资源 我们准备实现file_open()函数。当我们打开文件得到一个FILE *，我们需要利用资源机制注册它。下面的主要宏实现注册功能：

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

参考表格对宏参数的解释

ZEND_REGISTER_RESOURCE 宏参数

宏参数	参数类型
rsrc_result	zval *, which should be set with the registered resource information. zval * 设置为已注册资源信息
rsrc_pointer	Pointer to our resource data. 资源数据指针
rsrc_type	The resource id obtained when registering the resource type. 注册资源类型时获得的资源id

文件函数

现在你知道了如何使用ZEND_REGISTER_RESOURCE()宏，并且准备好了开始编写file_open()函数。还有一个主题我们需要讲述。

当PHP运行在多线程服务器上，不能使用标准的C文件存取函数。这是因为在一个线程里正在运行的PHP脚本会改变当前工作目录，因此另外一个线程里的脚本使用相对路径则无法打开目标文件。为了阻止这种错误发生，PHP框架提供了称作VCWD（virtual current working directory虚拟当前工作目录）宏，用来代替任何依赖当前工作目录的存取函数。这些宏与被替代的函数具备同样的功能，同时是被透明地处理。在某些没有标准C函数库平台的情况下，VCWD框架则不会得到支持。例如，Win32下不存在chown()，就不会有相应的VCWD_CHOWN()宏被定义。

VCWD列表

标准C库	VCWD宏
getcwd()	VCWD_GETCWD()
fopen()	VCWD_FOPEN
open()	VCWD_OPEN() //用于两个参数的版本
open()	VCWD_OPEN_MODE() //用于三个参数的open()版本
creat()	VCWD_CREAT()
chdir()	VCWD_CHDIR()
getwd()	VCWD_GETWD()
realpath()	VCWD_REALPATH()
rename()	VCWD_RENAME()
stat()	VCWD_STAT()
lstat()	VCWD_LSTAT()
unlink()	VCWD_UNLINK()
mkdir()	VCWD_MKDIR()
rmdir()	VCWD_RMDIR()
opendir()	VCWD_OPENDIR()
popen()	VCWD_POPEN()
access()	VCWD_ACCESS()
utime()	VCWD_UTIME()
chmod()	VCWD_CHMOD()
chown()	VCWD_CHOWN()

编写利用资源的第一个PHP函数

实现file_open()应该非常简单，看起来像下面的样子：

```
PHP_FUNCTION(file_open){
     char *filename = NULL;
     char *mode = NULL;
     int argc = ZEND_NUM_ARGS();
     int filename_len;
     int mode_len;
     FILE *fp;

     if (zend_parse_parameters(argc TSRMLS_CC, "ss", &filename,&filename_len, &mode, &m
          return;
     }

     fp = VCWD_FOPEN(filename, mode);

     if (fp == NULL) {
          RETURN_FALSE;
     }

     ZEND_REGISTER_RESOURCE(return_value, fp, le_myfile);
}
```

你可能会注意到资源注册宏的第一个参数return_value，可此地找不到它的定义。这个变量自动的被扩展框架定义为zval * 类型的函数返回值。先前讨论的、能够影响返回值的RETURN_LONG() 和RETVAL_BOOL()宏确实改变了return_value的值。因此很容易猜到程序注册了我们取得的文件指针fp，同时设置return_value为该注册资源。

访问资源 需要使用下面的宏访问资源（参看表对宏参数的解释）

```
ZEND_FETCH_RESOURCE(rsrc, rsrc_type, passed_id, default_id, resource_type_name, resourc
```

ZEND_FETCH_RESOURCE 宏参数

参数	含义
rsrc	资源值保存到的变量名。它应该和资源有相同类型。
rsrc_type	rsrc的类型，用于在内部把资源转换成正确的类型
passed_id	寻找的资源值(例如zval **)
default_id	如果该值不为-1，就使用这个id。用于实现资源的默认值。
resource_type_name	资源的一个简短名称，用于错误信息。
resource_type	注册资源的资源类型id

使用这个宏，我们现在能够实现file_eof()：

```
PHP_FUNCTION(file_eof){
     int argc = ZEND_NUM_ARGS();
     zval *filehandle = NULL;
     FILE *fp;

     if (zend_parse_parameters(argc TSRMLS_CC, "r", &filehandle) ==FAILURE) {
          return;
     }

     ZEND_FETCH_RESOURCE(fp, FILE *, &filehandle, -1, "standard-c-file",le_myfile);

     if (fp == NULL){
          RETURN_FALSE;
     }

     if (feof(fp) <= 0) {
     /* Return eof also if there was an error */
          RETURN_TRUE;
     }

     RETURN_FALSE;
}
```

删除一个资源

通常使用下面这个宏删除一个资源：

```
int zend_list_delete(int id)
```

传递给宏一个资源id，返回SUCCESS或者FAILURE。如果资源存在，优先从Zend资源列队中删除，该过程中会调用该资源类型的已注册资源清理函数。因此，在我们的例子中，不必取得文件指针，调用fclose()关闭文件，然后再删除资源。直接把资源删除掉即可。

使用这个宏，我们能够实现file_close()：

```
PHP_FUNCTION(file_close){
     int argc = ZEND_NUM_ARGS();
     zval *filehandle = NULL;

     if (zend_parse_parameters(argc TSRMLS_CC, "r", &filehandle) == FAILURE) {
          return;
     }

     if (zend_list_delete(Z_RESVAL_P(filehandle)) == FAILURE) {
          RETURN_FALSE;
     }

     RETURN_TRUE;
}
```

你肯定会问自己Z_RESVAL_P()是做什么的。当我们使用zend_parse_parameters()从参数列表中取得资源的时候，得到的是zval的形式。为了获得资源id，我们使用Z_RESVAL_P()宏得到id，然后把id传递给zend_list_delete()。有一系列宏用于访问存储于zval值（参考表的宏列表）。尽管在大多数情况下zend_parse_parameters()返回与c类型相应的值，我们仍希望直接处理zval，包括资源这一情况。

Zval访问宏

宏	访问对象	C 类型
Z_LVAL, Z_LVAL_P, Z_LVAL_PP	整型值	long
Z_BVAL, Z_BVAL_P, Z_BVAL_PP	布尔值	zend_bool
Z_DVAL, Z_DVAL_P, Z_DVAL_PP	浮点值	double
Z_STRVAL, Z_STRVAL_P, Z_STRVAL_PP	字符串值	char *
Z_STRLEN, Z_STRLEN_P, Z_STRLEN_PP	字符串长度值	int
Z_RESVAL, Z_RESVAL_P,Z_RESVAL_PP	资源值	long
Z_ARRVAL, Z_ARRVAL_P, Z_ARRVAL_PP	联合数组	HashTable *
Z_TYPE, Z_TYPE_P, Z_TYPE_PP	Zval类型	Enumeration (IS_NULL, IS_LONG, IS_DOUBLE, IS_STRING, IS_ARRAY, IS_OBJECT, IS_BOOL, IS_RESOURCE)
Z_OBJPROP, Z_OBJPROP_P, Z_OBJPROP_PP	对象属性 hash（本章不会谈到）	HashTable *
Z_OBJCE, Z_OBJCE_P, Z_OBJCE_PP	对象的类信息	zend_class_entry

用于访问zval值的宏

所有的宏都有三种形式：一个是接受zval s，另外一个接受zval *s，最后一个接受zval **s。它们的区别是在命名上，第一个没有后缀，zval *有后缀_P（代表一个指针），最后一个 zval **有后缀_PP（代表两个指针）。
现在，你有足够的信息来独立完成 file_read()和 file_write()函数。这里是一个可能的实现：

```
PHP_FUNCTION(file_read){
     int argc = ZEND_NUM_ARGS();
     long size;
     zval *filehandle = NULL;
     FILE *fp;
     char *result;
     size_t bytes_read;

     if (zend_parse_parameters(argc TSRMLS_CC, "rl", &filehandle,&size) == FAILURE) {
          return;
     }

     ZEND_FETCH_RESOURCE(fp, FILE *, &filehandle, -1, "standard-cfile", le_myfile);

     result = (char *) emalloc(size+1);

     bytes_read = fread(result, 1, size, fp);

     result[bytes_read] = '\0';

     RETURN_STRING(result, 0);
}

PHP_FUNCTION(file_write){
     char *buffer = NULL;
     int argc = ZEND_NUM_ARGS();
     int buffer_len;
     zval *filehandle = NULL;
     FILE *fp;

     if (zend_parse_parameters(argc TSRMLS_CC, "rs", &filehandle,&buffer, &buffer_len)
          return;
     }

     ZEND_FETCH_RESOURCE(fp, FILE *, &filehandle, -1, "standard-cfile", le_myfile);

     if (fwrite(buffer, 1, buffer_len, fp) != buffer_len) {
          RETURN_FALSE;
     }

     RETURN_TRUE;
}
```

测试扩展

你现在可以编写一个测试脚本来检测扩展是否工作正常。下面是一个示例脚本，该脚本打开文件test.txt，输出文件类容到标准输出，建立一个拷贝test.txt.new。

```
<?php
$fp_in = file_open("test.txt", "r") or die("Unable to open input file\n");

$fp_out = file_open("test.txt.new", "w") or die("Unable to open output file\n");

while (!file_eof($fp_in)) {
     $str = file_read($fp_in, 1024);
     print($str);
     file_write($fp_out, $str);
}

file_close($fp_in);
file_close($fp_out);
?>
```

全局变量

你可能希望在扩展里使用全局C变量，无论是独自在内部使用或访问php.ini文件中的INI扩展注册标记（INI在下一节中讨论）。因为PHP是为多线程环境而设计，所以不必定义全局变量。PHP提供了一个创建全局变量的机制，可以同时应用在线程和非线程环境中。我们应当始终利用这个机制，而不要自主地定义全局变量。用一个宏访问这些全局变量，使用起来就像普通全局变量一样。

用于生成myfile工程骨架文件的ext_skel脚本创建了必要的代码来支持全局变量。通过检查php_myfile.h文件，你应当发现类似下面的被注释掉的一节，

```
ZEND_BEGIN_MODULE_GLOBALS(myfile)

int global_value;
char *global_string;

ZEND_END_MODULE_GLOBALS(myfile)
```

你可以把这一节的注释去掉，同时添加任何其他全局变量于这两个宏之间。文件后部的几行，骨架脚本自动地定义一个MYFILE_G(v)宏。这个宏应当被用于所有的代码，以便访问这些全局变量。这就确保在多线程环境中，访问的全局变量仅是一个线程的拷贝，而不需要互斥的操作。

为了使全局变量有效，最后需要做的是把myfile.c：

```
ZEND_DECLARE_MODULE_GLOBALS(myfile)
```

注释去掉。

你也许希望在每次PHP请求的开始初始化全局变量。另外，做为一个例子，全局变量已指向了一个已分配的内存，在每次PHP请求结束时需要释放内存。为了达到这些目的，全局变量机制提供了一个特殊的宏，用于注册全局变量的构造和析构函数（参考表对宏参数的说明）：

```
ZEND_INIT_MODULE_GLOBALS(module_name, globals_ctor, globals_dtor)
```

表 ZEND_INIT_MODULE_GLOBALS 宏参数

参数	含义
module_name	与传递给ZEND_BEGIN_MODULE_GLOBALS()宏相同的扩展名称。
globals_ctor	构造函数指针。在myfile扩展里，函数原形与void php_myfile_init_globals(zend_myfile_globals *myfile_globals)类似
globals_dtor	析构函数指针。例如，php_myfile_init_globals(zend_myfile_globals *myfile_globals)

你可以在myfile.c里看到如何使用构造函数和ZEND_INIT_MODULE_GLOBALS()宏的示例。

添加自定义INI指令

INI文件(php.ini)的实现使得PHP扩展注册和监听各自的INI条目。如果这些INI条目由php.ini、Apache的htaccess或其他配置方法来赋值，注册的INI变量总是更新到正确的值。整个INI框架有许多不同的选项以实现其灵活性。我们涉及一些基本的（也是个好的开端），借助本章的其他材料，我们就能够应付日常开发工作的需要。

通过在PHP_INI_BEGIN()/PHP_INI_END()宏之间的STD_PHP_INI_ENTRY()宏注册PHP INI指令。例如在我们的例子里，myfile.c中的注册过程应当如下：

```
PHP_INI_BEGIN()

STD_PHP_INI_ENTRY("myfile.global_value", "42", PHP_INI_ALL, OnUpdateInt, global_value,

STD_PHP_INI_ENTRY("myfile.global_string", "foobar", PHP_INI_ALL, OnUpdateString, global

PHP_INI_END()
```

除了STD_PHP_INI_ENTRY()其他宏也能够使用，但这个宏是最常用的，可以满足大多数需要（参看表对宏参数的说明）：

```
STD_PHP_INI_ENTRY(name, default_value, modifiable, on_modify, property_name, struct_typ
```

STD_PHP_INI_ENTRY 宏参数表

参数	含义
name	INI条目名
default_value	如果没有在INI文件中指定，条目的默认值。默认值始终是一个字符串。
modifiable	设定在何种环境下INI条目可以被更改的位域。可以的值是： • PHP_INI_SYSTEM. 能够在php.ini或http.conf等系统文件更改 • PHP_INI_PERDIR. 能够在 .htaccess中更改 • PHP_INI_USER. 能够被用户脚本更改 • PHP_INI_ALL. 能够在所有地方更改

on_modify	处理INI条目更改的回调函数。你不需自己编写处理程序，使用下面提供的函数。包括： • OnUpdateInt • OnUpdateString • OnUpdateBool • OnUpdateStringUnempty • OnUpdateReal
property_name	应当被更新的变量名
struct_type	变量驻留的结构类型。因为通常使用全局变量机制，所以这个类型自动被定义，类似于zend_myfile_globals。
struct_ptr	全局结构名。如果使用全局变量机制，该名为myfile_globals。

最后，为了使自定义INI条目机制正常工作，你需要分别去掉PHP_MINIT_FUNCTION(myfile)中的REGISTER_INI_ENTRIES()调用和PHP_MSHUTDOWN_FUNCTION(myfile)中的UNREGISTER_INI_ENTRIES()的注释。

访问两个示例全局变量中的一个与在扩展里编写MYFILE_G(global_value) 和MYFILE_G(global_string)一样简单。

如果你把下面的两行放在php.ini中，MYFILE_G(global_value)的值会变为99。

```
; php.ini - The following line sets the INI entry myfile.global_value to 99.
myfile.global_value = 99
```

线程安全资源管理宏

现在，你肯定注意到以TSRM（线程安全资源管理器）开头的宏随处使用。这些宏提供给扩展拥有独自的全局变量的可能，正如前面提到的。

当编写PHP扩展时，无论是在多进程或多线程环境中，都是依靠这一机制访问扩展自己的全局变量。如果使用全局变量访问宏（例如MYFILE_G()宏），需要确保TSRM上下文信息出现在当前函数中。基于性能的原因，Zend引擎试图把这个上下文信息作为参数传递到更多的地方，包括PHP_FUNCTION()的定义。正因为这样，在PHP_FUNCTION()内当编写的代码使用访问宏（例如MYFILE_G()宏）时，不需要做任何特殊的声明。然而，如果PHP函数调用其他需要访问全局变量的C函数，要么把上下文作为一个额外的参数传递给C函数，要么提取上下文（要慢点）。

在需要访问全局变量的代码块开头使用TSRMLS_FETCH()来提取上下文。例如：

```
void myfunc(){
     TSRMLS_FETCH();

     MYFILE_G(myglobal) = 2;
}
```

如果希望让代码更加优化，更好的办法是直接传递上下文给函数（正如前面叙述的，PHP_FUNCTION()范围内自动可用）。可以使用TSRMLS_C（C表示调用Call）和TSRMLS_CC（CC边式调用Call和逗号Comma）宏。前者应当用于仅当上下文作为一个单独的参数，后者应用于接受多个参数的函数。在后一种情况中，因为根据取名，逗号在上下文的前面，所以TSRMLS_CC不能是第一个函数参。

在函数原形中，可以分别使用TSRMLS_D和TSRMLS_DC宏声名正在接收上下文。

下面是前一例子的重写，利用了参数传递上下文。

```
void myfunc(TSRMLS_D){
     MYFILE_G(myglobal) = 2;
}

PHP_FUNCTION(my_php_function)
{
     …
     myfunc(TSRMLS_C);
     …
}
```

总 结

现在，你已经学到了足够的东西来创建自己的扩展。本章讲述了一些重要的基础来编写和理解PHP扩展。Zend引擎提供的扩展API相当丰富，使你能够开发面向对象的扩展。几乎没有文档谈几许多高级特性。当然，依靠本章所学的基础知识，你可以通过浏览现有的原码学到很多。

更多关于信息可以在PHP手册的扩展PHP章节http://www.php.net/manual/en/zend.php中找到。另外，你也可以考虑加入PHP开发者邮件列表internals@lists.php.net，该邮件列表围绕开发PHP 本身。你还可以查看一下新的扩展生成工具——PECL_Gen(http://pear.php.net/package/PECL_Gen)，这个工具正在开发之中，比起本章使用的ext_skel有更多的特性。

此外你还可以关注风雪之隅,会有更多相关知识更新.

词汇表

binary safe	二进制安全
context	上下文
extensions	扩展
entry	条目
skeleton	骨架
Thread-Safe Resource Manager TSRM	线程安全资源管理器

```
Contact info:
     Email: taft at wjl.cn / laruence at yahoo.com.cn

http://www.laruence.com
```

---

[1] 可参考译者写的
[2] 译者：可以使用phpdi程序在控制台里执行php文件。
[3] 译者：可以查看到生成的FETCH_RESOURCE()宏参数是一些'???'。

Related Posts:

- 关于做PHP扩展开发的一些资源
- 保证PHP扩展的依赖关系
- 扩展PHP[Extending PHP](一)
- 编写提供对象给PHP使用的Module
- PHP RFC: 让PHP的foreach支持list
- Zend Parameters Parser新增类型描述符介绍
- PLua – Lua for PHP
- Yaf-一个PHP扩展实现的PHP框架
- Apache2中俩种设置PHP的异同
- 深入理解PHP原理之foreach

Tags: c写PHP扩展, PHP, PHP extension, php源码, 开发php扩展, 扩展PHP, 扩展开发

Filed in PHP源码分析, 转载

« 数组非数字键名引号的必要性　　使用CSS实现圈人效果(CSS Sprites) »

39 Responses to “用C/C++扩展你的PHP”

xuanskyer | 21 Sep 2014 00:05

亲测有效！

O(∩_∩)O哈哈~

yx | 22 Aug 2014 18:56

看得不是很懂。。

xiaosl | 04 Jun 2014 18:17

学习

chenchaogang | 04 May 2014 18:04

鸟哥出个源码扩展之类的书吧。

关于php + C++ 调用的问题或多数以什么样子的方式使用 - php - 开发者 | 09 Jun 2013 12:10

[...] 可以看看这篇文章 本文链接: 关于php + C++ 调用的问题或多数以什么样子的方式使用 版权所有: 非特殊声明均为本站原创文章，转载请注明出处：开发者 订阅更新: 您可以通过RSS订阅我们的内容更新 [...]

Flybeta | 10 May 2013 13:05

受教了，留着以后搞～

关于php + C++ 调用的问题或多数以什么样子的方式使用 - php - 开发者问答 | 02 May 2013 17:12

[...] 可以看看这篇文章 本条目发布于 2013 年 4 月 25 日。属于 php 分类，被贴了 php 标签。作者是 admin。 [...]

» 如何用C++实现PHP扩展库 marlon's blog | 19 Apr 2013 23:49

[...] 注意：截图中这四行代码是为了接收相对应的函数参数string ip, 关于zend_parse_parameters函数可以查看这篇文章用C/C++扩展你的PHP，这里有对zend_parse_parameters函数接收不同类型参数以及返回值定义的解释。 [...]

用C/C++扩展你的PHP | pitaya的blog | 08 Nov 2012 15:38

[...] 本文地址: http://www.laruence.com/2009/04/28/719.html [...]

sooiiy | 11 Sep 2012 18:01

在网上看见有讨论使用PHP编写PHP扩展的，我还以为已经实现了。最后郁闷了半天。

simplestbest | 27 Jul 2012 18:23

请教一个问题，PHP是如何实现在扩展库中调用zend_API的。就比如zend_parse_parameters这个函数，在扩展中一般都会调用，但这个函数是定义在zend_API.c中的。我不知道这个函数是否会被编译进php可执行文件，但可以确定的是php可执行程序并不会链接zend api库。而编写PHP扩展时，也不需要链接zend api。查了一些代码，也没看到为新写的扩展设置zend_parse_parameter回调函数。那运行时，php调用扩展，扩展再调用zend_parse_parameter.扩展是如何找到这个符号的？

treesky | 09 Jul 2012 23:34

最近才开始写，希望还不晚，因为想去某公司，但是好像能写扩展者优先

Thiniki's blog » 编写一个PHP扩展的基本步骤 | 02 Apr 2012 10:45

[...] 本文依样画葫芦，参考网上的文章做了个PHP扩展，记录一下主要的步骤。参考原文为：http://www.laruence.com/2009/04/28/719.html [...]

PHP下扩展开发步骤 - 编译人生 | 06 Jan 2012 11:26

[...] 参考来源：http://www.laruence.com/2009/04/28/719.html 标签：PHP，PHP扩展，PHP扩展开发 linux 解压缩命令 jquery 1.7.1 更新说明（中文） [...]

xuebao | 15 Nov 2011 15:40

鸟哥很强。。

卡卡龙 | 30 Oct 2011 04:50

俩年过去了，突然醒过来，真心要像你学习，学习的不仅是技术，更是踏实努力的精神，膜拜一下

Er.Z | 10 Aug 2011 00:43

用到了，才能真正发现文章之给力啊。
不过，你是在2006年……强。

51nosql | 29 May 2011 12:49

鸟哥之作，必属精品，受教了

用C/C++扩展你的PHP | 万维网黑客联盟 | 20 Mar 2011 16:52

[...] 本文地址: http://www.laruence.com/2009/04/28/719.html [...]

jquery学习 | 15 Jan 2011 22:36

哇～～讲解的好详细哦～～不过我水平还 不到家～～暂时还看不懂～～标记一下～～以待以后学习～～

xiaokai | 19 Oct 2010 11:01

曾经看不懂的牛文， 现在终于能看懂一二了。。

MirandaGraves22 | 22 Jul 2010 19:55

That's great that people can get the business loans and that opens up new possibilities.

用C/C++扩展你的PHP | OSMSG | 30 May 2010 10:15

[...] 文章来自http://www.laruence.com/2009/04/28/719.html linux← 每天一个 Linux 命令之 wc 详解 MeeGo 选择 Btrfs 作为默认文件系统 →Leave a comment0 Comments. [...]

Tweets that mention 用C/C++扩展你的PHP | 风雪之隅 -- Topsy.com | 16 Jan 2010 16:42

[...] This post was mentioned on Twitter by cgeek, bunny_car. bunny_car said: http://tinyurl.com/y9twqd8 用C/C++扩展你的PHP | 风雪之隅 [...]

Cherry | 26 Oct 2009 17:27

太棒了，第一次看到教怎么写PHP的扩展，留下了，有空了一定也自己写个，学习学习

Aries | 25 Aug 2009 16:51

这个留下,研究研究!

雪候鸟 | 11 Jun 2009 15:26

呵呵, 当然看过来, 那个是也我开始的时候的唯一老师啊…

reeze | 11 Jun 2009 15:19

很好的入门指导。
不知道博主有没有看过 《Extending and Embedded PHP》 这本是全方位的介绍了扩展的方方面面，值得参考。。

Anonymous | 09 Jun 2009 10:33

那这种方式就是fastcgi模式？使用一个进程对应一个app？

雪候鸟 | 04 Jun 2009 11:42

恩,这样啊, 可以通过另外写一个独立的shell, 来获取每个apache的进程所占用的内存, 继而估算..

Anonymous | 03 Jun 2009 09:16

谢谢您及时的回答。我想问问这个能不能独立出来，而不是每个PHP应用都调用这个函数~

雪候鸟 | 02 Jun 2009 18:19

每个PHP应用? 脚本自己可以获取自己的占用情况: memory_get_usage

Anonymous | 02 Jun 2009 11:16

LZ您好，我想咨询个问题，如果我需要度量每个PHP应用在APACHE中的内存占用，有什么思路？谢谢~

phpzxh | 27 May 2009 13:56

你写的都是有深度的文章，向楼主学习。

zvaly | 22 May 2009 22:03

good~~~

Anonymous | 30 Apr 2009 15:42

這樣做很多問題都很容易解決了

思臣 | 29 Apr 2009 09:47

有深度，学习了！

雪候鸟 | 28 Apr 2009 21:28

有理…照办,呵呵

cc0cc | 28 Apr 2009 20:52

顶，应该把我的那张思维导图帖上来。。。

Leave a Reply

Name

Mail (will not be published)

Website

CAPTCHA Code *

Submit Comment

Notify me of followup comments via e-mail