DOI: 10.19279/TVZ.PD.2015-3-1-09

EXTENDING PHP WITH MODULES

MODULARNO PROŠIRENJE PHP-A

Davor Lozić, Alen Šimec

Tehničko veleučilište u Zagrebu

Abstract

The purpose of this article is to show how to extend PHP and build modules. Since PHP is built mostly with C, one must be familiar with C or at least with some programming constructs like variables, loops, structures, unions, macros etc. PHP modules are built when standard PHP abilities doesn't fit developer's needs and when a developer wants to create a set of function or a library for which he doesn't want to provide the original source code. Since PHP is an interpreted language, it's hard to hide the original source code. Before creating a module, this paper will also explain how PHP variables are handled internally and what tools one must have for creating PHP modules. Source code in this paper will work on Debian Linux distribution, but with small modifications, it should also work on other Linux distributions.

KeyWords: *Internet, php, modules, source code, variables, Debian Linux*

Sažetak

Cilj ovog članka je pokazati kako proširiti PHP i izraditi module. Kako je PHP napravljen pomoću C-a, potrebno je poznavati C ili barem neke jednostavne programske konstrukte kao što su varijable, petlje, strukture, unije, makroi itd. PHP moduli se rade u slučaju kada programeru osnovna funkcionalnost PHP programskog jezika nije dovoljna ili kada programer želi napraviti set funkcija ili biblioteku funkcija za koju ne želi prikazati izvorni kôd. Kako je PHP interpretirani jezik, teško je skriti originalni izvorni kôd. Prije izrade modula, članak objašnjava i kako su PHP varijable implementirane interno te što je potrebno imati za izradu PHP modula. Izvorni kôd u ovom članku radit će na Debian Linux distribuciji, ali s manjim promjenama trebao bi raditi i na bilo kojoj drugoj distribuciji.

Ključne riječi: *Internet, php, moduli, izvorni kod, varijable, Debian Linux*

1. Introduction

1. Uvod

PHP is an acronym which stands for "PHP: Hypertext Preprocessor" and is widely used in web development on server side [1]. The language is written in C and through history it had many forms and purposes but the PHP version 3 was the first version that closely resembles PHP as it exists today[2].

Also, PHP is a general-purpose, scripting and high-level programming language. General-purpose means the language is designed to be used in a wide variety of application domains. Word "scripting" means the language is not compiled; rather, it is analyzed and executed line by line in runtime. High-level means that most developers does not need to know how the language works with memory, does not do anything "under the hood" and gives a certain level of abstraction from the details. When a developer needs to create a function but does not want to give the source code, function could be written in C, compiled as a module and be added to PHP.

For compiling additional modules, it is crucial to understand some concepts about PHP internals and how it actually works. This paper will start with how PHP variables are represented internally, how to write a simple PHP function in C and then how to compile it and create a module.

2. Zvalue

2. Zvalue (zend value) struktura

In PHP, there are actually eight different types of variables [2]: integers, floating point numbers, strings, Booleans, arrays, objects, resources and null type.

Since C programming language is statically typed, a developer cannot put integers in Boolean variable or strings into floating point variable but PHP can. This is possible with *unions* in C. Actually, every PHP variable is a `_zvalue_value` union and this is how the union is implemented: [4]

```
typedef union _zvalue_value {
   long lval;
   double dval;
   struct {
      char *val;
      int len;
   } str;
   HashTable *ht;
   zend_object_value obj;
} zvalue_value;
```

It is worth noticing that string variables save the start of the string and the length of the string so "\0" values in PHP strings practically doesn't have any special meaning.

Now, the problem is how would PHP know is there integer or string inside union. PHP has `_zval_struct`:

```
struct _zval_struct {
   zvalue_value value;
   zend_uint refcount__gc;
   zend_uchar type;
   zend_uchar is_ref__gc;
};
```

`_zval_struct` has `zvalue_value` inside and `_zval_struct` represents a single PHP variable. Type inside structure tells about variable type inside union, `refcount__gc` tells to garbage collector how many pointers are pointing to the same variable:

```
$variable = 10;
$variable2 = $variable;
$variable3 = $variable2;
```

PHP is smart enough not to create another `_zval_struct` but to increment `refcount__gc` variable.

is`_ref__gc` tells to PHP is the variable maybe just a pointer to another variable:

```
$variable = 10;
$variable2 = &$variable;
```

Now, PHP is really smart when optimizing variables. If the `refcount__gc` is 3 and $variable changes its type and value, PHP doesn't change other 9 variables, instead it just creates another `_zval_struct` only for $variable.

```
$variable = 10;
$variable2 = $variable;
$variable3 = $variable2;
/* only $variable get a new _zval_struct
*/
$variable = "now it's a string";
```

3. PHP-src and PHPize

3. Php izvorni kôd i phpize

First, a developer must get PHP source code and that is very easy thing to do with Git:

```
git clone https://github.com/php/php-src.
git
```

Now, since creating PHP modules could be pretty straightforward in many cases, there is a tool which can create a skeleton for a developer and it is called `ext_skel`. For creating modules, developer needs phpize which can use the created skeleton and your code and then create a module. On Debian Linux (and on many Debian derivatives) you could probably get the tool with:

```
apt-get install php5-devel
```

Now, a developer must go to the downloaded/cloned source code and go to the ext folder and execute `ext_skel` and provide the module name which will, in this paper, be called myext:

```
cd ext
./ext_skel –extname=myext
```

Inside myext folder, there is a config.m4 which needs to be edited. Remove comments so the file

contains this (watch for indentation, "[" needs to be in the same indentation level as "PHP_ARG_ENABLE"):

```
PHP_ARG_ENABLE (myext, whether to enable
myext support,

[  --enable-myext           Enable myext
support])
```

Now, phpize must be executed, which will prepare the module:

```
phpize

./configure --enable-myext
```

Now, inside myext.c there is a function called PHP_FUNCTION (confirm_myext_compiled) and this is could be one of the PHP functions.
A developer can rename or add new functions inside extension but they need to be declared inside myext_functions[]. Now, the function will print "Hello World":

```
PHP_FUNCTION(confirm_myext_compiled){
      php_printf("Hello World\n");
}
```

Compilation and installation step is easy:

```
make
make install
```

Extension needs to be enabled inside php.ini file and the HTTP server needs to get restarted:

```
extension=myext.so
sudo /etc/init.d/apache2 restart
```

Now, inside HTTP server's root folder, there must be a test file, for example, myext.php.

```
<?php
// Hello World
confirm_myext_compiled();
```

What about sending parameters? Function zend_parse_parameters gets parameters and puts them inside zval's. In this example, confirm_myext_compiled is getting a string and instead printing "Hello World", it will print "Hello " and the provided string or a number. Here, binary safety is not a concern so *printf* function should be enough:

```
PHP_FUNCTION(confirm_myext_compiled){
  zval *arg;
if(zend_parse_parameters(ZEND_NUM_ARGS()
TSRMLS_CC, "z", &arg) == FAILURE) {
return;
}

 switch (Z_TYPE_P(arg)){
   case IS_NULL:
    php_printf("Hello NULL\n");
    break;
   case IS_LONG:
    php_printf("Hello %ld\n", Z_
LVAL_P(arg));
    break;
   case IS_DOUBLE:
    php_printf("Hello %.2f\n", Z_
DVAL_P(arg));
    break;
   case IS_STRING:
    php_printf("Hello %s\n", Z_
STRVAL_P(arg));
    break;
   default:
    php_printf("Error type!\n");
 }
}
```

Again, recompiling, installing and restarting a web server:

```
make
make install
/etc/init.d/apache2 restart
```

Now, testing should be easy enough:

```
confirm_myext_compiled(5);
echo "<br />";
confirm_myext_compiled("test");
echo "<br />";
confirm_myext_compiled(0.5);
echo "<br />";
confirm_myext_compiled(null);
echo "<br />";
```

And the result should be as expected:

```
Hello 5
Hello test
Hello 0.50
Hello NULL
```

Function zend_parse_parameters has TSRMLS_CC. TSRM is *Thread Safe Resource*

Manager [5] and it is not a subject in this paper. String "z" could be any of the provided combinations which are telling to PHP what parameters it could expect [3]:

Specifier	C type	Description
b	zend_bool	Boolean
z	zval *	Any variable
L	long	Integer with truncation
d	double	Float
s	char *, int	String (value and length)
l	long	Integer with rollover

It is possible to put strictly *double* inside a double variable instead of zval, with a specifier d.

```
double d;
if(zend_parse_parameters(ZEND_NUM_ARGS()
TSRMLS_CC, "d", &d) == FAILURE) {
return;
}
```

There are some modifiers which can tell more about passed parameters to PHP:

Modifier	Meaning
| (pipe)	All specifiers beyond this point are optional
!	If NULL is passed, leave the C variable unmodified
/	Separate value so that changes made to it by the C implementation do not effect the PHP variable passed to the function.

So, creating a function which gets a Boolean, double and an optional integer is easy:

```
zval_bool b;
double d;
long l;
if(zend_parse_parameters(ZEND_NUM_ARGS()
TSRMLS_CC, "bd|l", &b, &d, &l) ==
FAILURE) {
return;
}
```

4. Benchmarking

4. Test brzine

Hiding source code is not the only advantage when writing extensions in C. C programming language is faster. In this paper, a function which gets two integers and returns a sum, is examined, both C and PHP version of code.
C version, pretty simple:

```
PHP_FUNCTION(confirm_myext_compiled){
  long l1, l2;
  if(zend_parse_parameters(ZEND_NUM_
ARGS() TSRMLS_CC, "ll", &l1, &l2) ==
FAILURE) {
  return;
}


  RETURN_LONG(l1 + l2);
}
```

PHP version, even simpler:

```
function php_version($a, $b){
   return $a + $b;
}
```

On some systems, there is a need for extending PHP timeouts since the test script will be longer than 30 seconds (which is default on many systems):

```
set_time_limit(1000);
```

If this function is not working, inside php.ini `max_execution_time` should be altered. php.ini could be easily found with:

```
php --ini
The whole testing script is here:
<?php
set_time_limit(1000);


function php_version($a, $b){
       return $a + $b;
}


$before_php = microtime(true);
for ($j=0 ; $j<100000000; $j++) {
    php_version(5, 10);
}
```

```
$after_php = microtime(true);
$result_php = $after_php - $before_php;


echo "PHP version: " . $result_php;
echo "<br />";


$before_c = microtime(true);
for ($i=0 ; $i<100000000; $i++) {
    confirm_myext_compiled(5, 10);
}
$after_c = microtime(true);
$result_c = $after_c - $before_c;
echo "C version: " . $result_c;
echo "<br />";


$faster = $result_php - $result_c;
echo "C version is faster for: " .
$faster;
$percentage = (1 - ($result_c / $result_
php)) * 100;
echo "<br />";


echo "That is " . sprintf("%.2f%%",
$percentage) . " faster";
```

The result on my machine:

```
PHP version: 22.995001077652
C version: 19.179035186768
C version is faster for: 3.8159658908844
That is 16.59% faster
```

5. Conclusion

5. Zaključak

Writing modules in C is harder than writing plain PHP functions but, as benchmarking showed, execution time is faster and therefore if a developer needs to write many functions which needs massive calculations, it's definitely a better option. A developer could examine the created .so file inside *modules* folder with *objdump*. In this paper, it's done with the next line:

```
objdump -M intel -D myext.so
```

This is what is generated for `zif_confirm_myext_compiled`:

```
b60: push   rbx
b61: mov    rbx,rsi
b64: lea    rsi,[rip+0x5c]
b6b: xor    eax,eax
b6d: sub    rsp,0x10
b71: lea    rcx,[rsp+0x8]
b76: mov    rdx,rsp
b79: call   9a0 <zend_parse_parameters@
plt>
b7e: cmp    eax,0xffffffff
b81: je     b93 <zif_confirm_myext_
compiled+0x33>
b83: mov    rax,QWORD PTR [rsp]
b87: add    rax,QWORD PTR [rsp+0x8]
b8c: mov    BYTE PTR [rbx+0x14],0x1
b90: mov    QWORD PTR [rbx],rax
b93: add    rsp,0x10
b97: pop    rbx
b98: ret
b99: nop    DWORD PTR [rax+0x0]
```

There are ways to improve generated code but it's still less than code generated while executing plain PHP functions and it's a lot harder to find out how the function works and what it does.

6. References

6. Reference

[1] http://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html

[2] https://line.do/php-evolution/8oq/vertical

[3] https://github.com/sgolemon/phptek2013/blob/master/params/README.md

[4] http://www.phpinternalsbook.com/zvals/basic_structure.html

[6] http://devzone.zend.com/303/extension-writing-part-i-introduction-to-php-and-zend/

AUTORI · AUTHORS

Davor Lozić (dlozic@tvz.hr), student at the Polytechnic of Zagreb, currently working as a senior software engineer at Informatička podrška d.o.o. creating web-based applications with ExtJS and CakePHP. Interested in application security, internal work of operating systems and low-level programming.

Alen Šimec - nepromjenjena biografija nalazi se u časopisu Polytechnic & Design Vol. 1, No. 1, 2013.

Korespondencija:
alen@tvz.hr