

Harald Radi (harald.radi@nme.at) – <http://www.nme.at>

1 New Features in the ZendEngine2's OO API

ZendEngine2, the core of the upcoming PHP 5 version, not only has a totally revised userland object model but also a completely redesigned extension API. The new API makes it possible to quickly write simple OO extensions but also to hook deep into the engine inwards to be able to use the full scope of provided functionality. This paper will show you which hooks are available in the Engine and what they are good for.

Furthermore you'll get an impression on how to use the extension API in one of your own extensions. At the end a minimal sample implementation will be introduced and discussed.

1.1 What's new compared to the old Engine

There is a whole bunch of new stuff in the ZendEngine2 that cannot be completely elaborated in this paper. Only the portions interesting for extension developers will be covered in detail. Concepts like protected and private class members, static attributes and syntax modifications are important too, but don't affect extensions.

Writing extensions in general is very much the same as for PHP 4, the only part that has changed API wise is the object overloading API. OO extensions written for PHP 4 will not compile with PHP 5, concepts changed very drastically so that old extensions probably have to be rewritten completely.

1.1.1 Improved object handling

One of the major changes is how objects are treated by the engine now. In PHP 4 dealing with objects was very cumbersome. You really had to take care whether to pass an object by value or explicitly by reference. Furthermore it was impossible at all to assign an object reference to a passed-by-reference function parameter.

In PHP 5 objects are not copied anymore, instead a unique object handle is passed around. That handle is used by the engine to look up the actual object in the global object store, thus there is exactly one instance of an object except it is explicitly cloned.

Objects in PHP 5 are comparable with resources where the zval container also only holds the identifier and not the resource itself.

The change of the object treatment should not have any serious implications in userland as this seems to be the more natural way of treating objects and is also more common in other languages.

1.1.2 Improved object dereferencing

In PHP 4 it was not possible to directly dereference objects returned from methods, e.g. `$object->method()->method()` or even `$object->method()->member->method()`.

The new Engine introduces that feature which hopefully leads to better looking and more straightforward code. It can prevent certain programming errors and is also the natural syntax when interfering with Java or COM objects.

1.1.3 Object cloning

In PHP 4 it was not up to the user to decide whether an object is duplicated or not. Whenever the engine had the impression that it should duplicate an object it did a bitwise copy of the underlying memory structure resulting in an identical replica of the originating object.

Due to the revamped object handling described in 1.1.1 an object will never be duplicated by the engine automatically. To give the user the ability to duplicate an object intentionally the `__clone()` method was introduced.

This method can be overloaded in userland as well as directly in an extension, so instead of copying everything only the relevant parts of an object can be copied.

1.1.4 Object destructors

The lack of the ability to define a destructor for an object was criticised very often in the past. Destructors are useful to clean up temporary files or resources, to log debug messages and lots of other things.

PHP 5 is finally coming up with a solution, destructors are now part of the language specification. When the last reference to an object is destroyed the object's destructor is called before the object is freed from memory.

Important to know is that due to the nature of PHP the destructor might not always be called, there are still a few exceptions. For example, when a fatal error occurs the Engine will bail out without cleaning up everything properly.

1.1.5 Interfaces

As a way to expose multiple functionalities by a class while circumventing multiple inheritance, interfaces were implemented. This is a common methodology well reputed from languages like Java or C#.

Interfaces are a way to derive a class from one or more types without inheriting the type's implementation like you would if you derive from a base class.

Interfaces are especially useful in conjunction with the recently introduced type hints, that enable a developer to enforce that a function parameter is of a given class or implements a specific interface.

1.1.6 Namespaces

Due to PHP's popularity a lot of reusable code was written in the past and is now publicly available via various online code bases. One of the best known projects in this area is probably the PEAR project. The huge amount of functions and classes provided by such libraries made it increasingly difficult to avoid symbol name collisions.

Namespaces in PHP 5 provide a way to avoid these collisions and manage those libraries in distinct namespaces. Unlike other languages PHP 5 does not support nested namespaces, each namespace exists in the global scope and has no relation to any of the other namespaces.

Beside the characters allowed in variable and function names, namespace names may also contain the ':' character as a separator. Even if ':' a semantical meaning at first glance it is only syntactical sugar and a punctuation character like any other.

1.1.7 Exception handling

Exception handling is introduced with PHP 5 and is a very convenient tool when used correctly. That means it should only be used for handling errors and not for controlling the regular program flow.

The implementation of `try/throw/catch` looks similar to other programming languages. Any code which is inside a `try/catch` block can throw an exception passing that exception to the closest catch block that catches exceptions of that type.

1.2 Constructing an object

1.2.1 Internal representation

Internally an object is represented by a `zend_object_value` structure, which contains the object handle and a pointer to a `zend_object_handlers` structure which will be described later.

The handle is an unsigned integer which has to be unique to the objects class. That means it has to be clear enough for you to find the objects underlying data based on that handle.

```
struct _zend_object_value {
    zend_object_handle handle;
    zend_object_handlers *handlers;
};
```

Listing 1: `zend_object_value`

The structure from listing 1 is part of the `zvalue_value` union which is the value member of the well known `zval` structure, the internal representation of each variable in PHP. Now it should be clear to you what will happen whenever a `zval` container gets copied, only the unique handle and the pointer to the list of handlers will be copied, the object data itself remains untouched.

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

Constructing an object

```
    } str;
    HashTable *ht;
    zend_object_value obj;
} zvalue_value;
```

Listing 2: zvalue_value

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

Listing 3: zval

The `zend_object_handlers` structure shown in listing 4 contains all the existing hooks that can be used to react to object manipulations in userland. Just don't be scared right now, you only have to implement a few of the hooks. The remaining hooks are either completely optional or have default handlers that can be assigned instead.

Compared to the limited abilities in PHP 4 this is a huge improvement and also reflects the necessity that most parts of existing OO extensions have to be rewritten when ported from PHP 4 to PHP 5.

```
typedef struct _zend_object_handlers {
    /* general object functions */
    zend_object_add_ref_t          add_ref;
    zend_object_del_ref_t          del_ref;
    zend_object_delete_obj_t       delete_obj;
    zend_object_clone_obj_t        clone_obj;

    /* individual object functions */
    zend_object_read_property_t    read_property;
    zend_object_write_property_t   write_property;
    zend_object_get_property_ptr_t get_property_ptr;
    zend_object_get_property_zval_ptr_t get_property_zval_ptr;
    zend_object_get_t              get;
    zend_object_set_t              set;
    zend_object_has_property_t     has_property;
    zend_object_unset_property_t   unset_property;
    zend_object_get_properties_t   get_properties;
    zend_object_get_method_t       get_method;
    zend_object_call_method_t      call_method;
    zend_object_get_constructor_t  get_constructor;
};
```

```
zend_object_get_class_entry_t    get_class_entry;
zend_object_get_class_name_t    get_class_name;
zend_object_compare_t           compare_objects;
} zend_object_handlers;
```

Listing 4: zend_object_handlers

For a better understanding of the structure all hooks and their signature are discussed here in detail.

add_ref

```
typedef void (*zend_object_add_ref_t)(zval *object TSRMLS_DC);
```

This handler is called whenever a new zval referring to that object is created. It does not create a new object, both zvals still refer to the same object.

del_ref

```
typedef void (*zend_object_del_ref_t)(zval *object TSRMLS_DC);
```

This handler is called whenever a reference to that object is destroyed. This does not mean that the object should be destructed, only one zval less is pointing to that object.

delete_obj

```
typedef void (*zend_object_delete_obj_t)(zval *object TSRMLS_DC);
```

This handler is called after all references to that object are destroyed. It should clean up any resources and free the underlying data structures. After that function was called the objects handle is no longer valid.

clone_obj

```
typedef zend_object_value (*zend_object_clone_obj_t)(zval *object
TSRMLS_DC);
```

This handler is called when a new object identical to an old one should be created. Unlike PHP 4 this only happens when the user explicitly clones an object with the `__clone()` method.

The handler is responsible for copying the underlying data and reinitialize resources a second time. Take care, if you only copy resource handles together with your underlying data structure these resources will get freed twice which most probably results in a segmentation fault.

read_property

```
typedef zval *(*zend_object_read_property_t)(zval *object, zval *member TSRMLS_DC);
```

This handler is called when an object's property is requested. The returned value is read-only and not meant to be changed.

write_property

```
typedef void (*zend_object_write_property_t)(zval *object, zval *member, zval *value TSRMLS_DC);
```

This handler is used to assign property variables or to change them in operations like += or ++ unless `get_property_zval_ptr` is also set.

get_property_ptr

```
typedef zval **(*zend_object_get_property_ptr_t)(zval *object, zval *member TSRMLS_DC);
```

This handler is used to create a pointer to the property of the object, for future read-write access via the `get` and `set` handler. If your object properties are stored as `zval*`, return the real pointer where the property is stored. If they aren't, the best way is to create a proxy object that is responsible for managing that property. If you don't want to implement property referencing for your objects, you can set this handler to `NULL`.

get_property_zval_ptr

```
typedef zval **(*zend_object_get_property_zval_ptr_t)(zval *object, zval *member TSRMLS_DC);
```

This handler is used to obtain a pointer to a modifyable `zval` for operations like += or ++. This should be used only if your object model stores properties as real `zvals` that can be modified from outside. Otherwise this handler should be `NULL` and the engine will use `read_property` and `write_property` instead.

get

```
typedef zval* (*zend_object_get_t)(zval *property TSRMLS_DC);
```

This handler is used to get object value, most probably in combination with the result of the `get_property_ptr` function or when converting object value to one of the basic types.

Get and set handlers are used when engine needs to access the object as a value. E.g., in the following situation:

```
$foo = &$obj->bar;  
$foo = 1;
```

If `$foo` is an object it would be accessed using the set handler.

set

```
typedef void (*zend_object_set_t)(zval **property, zval *value  
TSRMLS_DC);
```

The set handler is the counterpart to the get handler and is responsible for writing a value directly to an object.

has_property

```
typedef int (*zend_object_has_property_t)(zval *object, zval  
*member, int check_empty TSRMLS_DC);
```

This handler checks if that object has a certain property set and optionally if it is empty or not.

unset_property

```
typedef void (*zend_object_unset_property_t)(zval *object, zval  
*member TSRMLS_DC);
```

This handler is used to remove the value for a specified property of that object.

get_properties

```
typedef HashTable *(*zend_object_get_properties_t)(zval *object  
TSRMLS_DC);
```

This handler should return the list of properties of the object as a hash of zvals.

get_method

```
typedef union _zend_function *(*zend_object_get_method_t)(zval  
*object, char *method, int method_len TSRMLS_DC);
```

This handler is used to find a method description by name. It should set the right type, function name and parameter mask for the method. If the type is `ZEND_OVERLOADED_FUNCTION`, the method is called via the `call_method` handler, otherwise the returned function is called directly.

call_method

```
typedef int (*zend_object_call_method_t)(char *method,  
INTERNAL_FUNCTION_PARAMETERS);
```

This handler is a dispatcher function for any method call that couldn't be resolved by `get_method`. It is called with the method name and the full argument list of the actually called method. Parameters are passed like to any other Zend internal function.

get_constructor

```
typedef union _zend_function  
*(*zend_object_get_constructor_t)(zval *object TSRMLS_DC);
```

`get_constructor` performs the same operation as `get_method`, but for the object constructor.

get_class_entry

```
typedef zend_class_entry *(*zend_object_get_class_entry_t)(zval  
*object TSRMLS_DC);
```

This handler returns the `zend_class_entry` structure for the object in case it is a class object and not an ad-hoc object without a class.

get_class_name

```
typedef int (*zend_object_get_class_name_t)(zval *object, char  
**class_name, zend_uint *class_name_len, int parent TSRMLS_DC);
```

This handler is used to retrieve class name of the object.

compare_objects

```
typedef int (*zend_object_compare_t)(zval *object1, zval *object2  
TSRMLS_DC);
```

This handler is used to compare objects of the same class. This is used with the `==` operation whereas `===` compares objects by handles, i.e. it returns true if and only if it's really the same object. Note that objects from different object types cannot be compared.

1.2.2 Ad-Hoc objects

Unlike PHP 4 you can now create objects without a corresponding class entry. This is useful for returning objects from an extension function that mainly contain data without methods. Creating ad-hoc objects is probably quicker and much simpler then, but you

have to be aware that such objects can never be instantiated with the new operator, they can only be used as a return value.

You can create an ad-hoc object by assigning a handle and a pointer to the objects handlers structure to a zval container and set its type to IS_OBJECT.

1.2.3 Class objects

Creating class objects is a bit more difficult, in return you gain a lot more functionality. They can be created with the new operator just like userland objects and they are fully supported by PHP's reflection API.

For each class you want to export you have to create a zend_class_entry structure, which is shown in listing 5. Most members of that structure will be initialized by the INIT_CLASS_ENTRY macro, you have to mind a few things though.

```
struct _zend_class_entry {
    char type;
    char *name;
    zend_uint name_length;
    struct _zend_class_entry *parent;
    int refcount;
    zend_bool constants_updated;
    zend_uint ce_flags;

    HashTable function_table;
    HashTable default_properties;
    HashTable properties_info;
    HashTable class_table;
    HashTable *static_members;
    HashTable constants_table;
    zend_function_entry *builtin_functions;
    struct _zend_class_entry *ns;

    union _zend_function *constructor;
    union _zend_function *destructor;
    union _zend_function *clone;
    union _zend_function *__get;
    union _zend_function *__set;
    union _zend_function *__call;

    /* handlers */
    zend_object_value (*create_object)
        (zend_class_entry *class_type TSRMLS_DC);

    zend_class_entry **interfaces;
    zend_uint num_interfaces;
};
```

```
char *filename;
zend_uint line_start;
zend_uint line_end;
char *doc_comment;
zend_uint doc_comment_len;
};
```

Listing 5: zend_class_entry

The INIT_CLASS_ENTRY macro takes the class entry structure as first and the class name as second parameter. The third parameter is optional and can be set to NULL. If set it has to point to a function_entry[] array which will be used as a list of class member functions.

```
INIT_CLASS_ENTRY(class_container, class_name, functions)
```

After initializing the class entry you have to assign the internal constructor function which is responsible to allocate the necessary memory for the object and the underlying data before it can be accessed. This is NOT the actual constructor of the object, it can be seen as the object allocator.

```
class_entry.create_object = ext_objects_new;
```

This is the minimum initialization for the class entry necessary to successfully create objects from it.

To make the engine aware of that class entry you have to register it with one of the two following functions. The former is used for basic registrations, the latter is used if you want to derive your class entry from an existing one.

```
ZEND_API zend_class_entry
*zend_register_internal_class(zend_class_entry *class_entry
TSRMLS_DC);
```

```
ZEND_API zend_class_entry
*zend_register_internal_class_ex(zend_class_entry *class_entry,
zend_class_entry *parent_ce, char *parent_name TSRMLS_DC);
```

1.3 The object store

Since objects are no longer stored in the zval container itself Zend introduced the object store to keep the actual object data and hand out unique object handles.

The object store API mainly consists of three functions which are discussed below.

The first step you have to do is to register the object data at the object store. This is done with the `zend_objects_store_put` function which also expects pointers to a destructor and a clone function for that data. These pointers are optional and can be set to `NULL`.

The function returns a unique handle that can be directly assigned to the `zend_object_value` structure's handle member.

```
ZEND_API zend_object_handle zend_objects_store_put(void *object,
zend_objects_store_dtor_t dtor, zend_objects_store_clone_t clone
TSRMLS_DC);
```

The `zend_object_store_get_object` function can be used to retrieve the stored data back from the object store based on the `zval` containing the unique object handle.

```
ZEND_API void *zend_object_store_get_object(zval *object
TSRMLS_DC);
```

And last but not least you also get a function to tell the object store to forget a specific instance and clean up the allocated resources. This function will be called from an objects destructor handler.

```
ZEND_API void zend_objects_store_delete_obj(zval *object
TSRMLS_DC);
```

1.4 Example

This is a very small example that exports a meaningless class to the userland which can there be instantiated with the new operator and reflected by all means PHP provides.

```
#ifndef PHP_EXT_H
#define PHP_EXT_H

#include "php.h"

extern zend_module_entry ext_module_entry;
#define phpext_ext_ptr &ext_module_entry

PHP_MINIT_FUNCTION(ext);
PHP_MSHUTDOWN_FUNCTION(ext);
PHP_RINIT_FUNCTION(ext);
PHP_RSHUTDOWN_FUNCTION(ext);
PHP_MINFO_FUNCTION(ext);

ZEND_API zend_object_value
ext_objects_new(zend_class_entry * TSRMLS_DC);
```

Example

```
zend_class_entry *ext_class_entry;

typedef struct _ext_object {
    zend_object zo;
    void *ptr;
} ext_object; /* extends zend_object */

#endif /* PHP_EXT_H */
```

Listing 6: php_ext.h

```
#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"

#include "php_ext.h"

static zend_object_handlers ext_object_handlers;

/* {{{ ext_objects_dtor
*/
static void ext_objects_dtor(void *object,
    zend_object_handle handle TSRMLS_DC)
{
    ext_object *intern = (ext_object *) object;
    efree(intern->ptr);

    zend_objects_destroy_object(object, handle TSRMLS_CC);
}
/* }}} */

/* {{{ ext_objects_clone
*/
static void ext_objects_clone(void *object,
    void **object_clone TSRMLS_DC)
{
    ext_object *intern = (ext_object *) object;
    ext_object **intern_clone = (ext_object **) object_clone;

    *intern_clone = emalloc(sizeof(ext_object));
    (*intern_clone)->zo.ce = intern->zo.ce;
    (*intern_clone)->zo.in_get = 0;
    (*intern_clone)->zo.in_set = 0;

    ALLOC_HASHTABLE((*intern_clone)->zo.properties);
```

New Features in the ZendEngine2's OO API

```
zend_hash_init((*intern_clone)->zo.properties, 0,
              NULL, ZVAL_PTR_DTOR, 0);
zend_hash_copy((*intern_clone)->zo.properties,
              intern->zo.properties,
              (copy_ctor_func_t) zval_add_ref,
              NULL, sizeof(zval *));

/* alloc memory for your custom data structure
   and assign it here
   */
intern->ptr = emalloc(sizeof(underlying_data));
}
/* }}} */

/* {{{ ext_objects_new
   */
ZEND_API zend_object_value ext_objects_new(
    zend_class_entry *class_type TSRMLS_DC)
{
    zend_object_value retval;
    ext_object *intern;
    zval *tmp;

    intern = emalloc(sizeof(ext_object));
    intern->zo.ce = class_type;
    intern->zo.in_get = 0;
    intern->zo.in_set = 0;

    ALLOC_HASHTABLE(intern->zo.properties);
    zend_hash_init(intern->zo.properties, 0,
                  NULL, ZVAL_PTR_DTOR, 0);
    zend_hash_copy(intern->zo.properties,
                  &class_type->default_properties,
                  (copy_ctor_func_t) zval_add_ref,
                  (void *) &tmp, sizeof(zval *));

    retval.handle = zend_objects_store_put(intern,
    ext_objects_dtor, ext_objects_clone TSRMLS_CC);
    retval.handlers = &ext_object_handlers;

    return retval;
}
/* }}} */
```

Example

```
/* {{{ ext_functions
 */
function_entry ext_functions[] = {
    {NULL, NULL, NULL}
};
/* }}} */

/* {{{ ext_class_functions
 */
function_entry ext_class_functions[] = {
    {NULL, NULL, NULL}
};
/* }}} */

/* {{{ ext_module_entry
 */
zend_module_entry ext_module_entry = {
    STANDARD_MODULE_HEADER,
    "ext",
    ext_functions,
    PHP_MINIT(ext),
    NULL,
    NULL,
    NULL,
    PHP_MININFO(ext),
    "1.0",
    STANDARD_MODULE_PROPERTIES
};
/* }}} */

#ifdef COMPILE_DL_EXT
ZEND_GET_MODULE(EXT)
#endif

/* {{{ PHP_MINIT(ext)
 */
PHP_MINIT_FUNCTION(ext)
{
    zend_class_entry _ext_class_entry;
    INIT_CLASS_ENTRY(_ext_class_entry, "ext",
                    ext_class_functions);
    _ext_class_entry.create_object = ext_objects_new;
    ext_class_entry = zend_register_internal_class(
        &_ext_class_entry TSRMLS_CC);
}
```

```
/* copy the standard object handlers to you handler table */
memcpy(&ext_object_handlers, zend_get_std_object_handlers(),
       sizeof(zend_object_handlers));
/* replace the default clone handler with the object store
   clone handler

   */
ext_object_handlers.clone_obj =
    zend_objects_store_clone_obj;

return SUCCESS;
}
/* }}} */

/* {{{ PHP_MININFO(ext)
   */
PHP_MININFO_FUNCTION(ext)
{
    php_info_print_table_start();
    php_info_print_table_header(2, "ext support", "enabled");
    php_info_print_table_row(2, "ext", "present");
    php_info_print_table_end();
}
/* }}} */
```

Listing 7: ext.c

1.5 References

[Ze02a] Zend Engine Version 2.0, Feature Overview and Design, <http://www.zend.com>

[Ze02b] OBJECTS2_HOWTO, cvs.php.net

[Bak02] The namespaces RFC, Stig Sæther Bakken, cvs.php.net

and also to all the PHP folks, especially Zeev Suraski, for answering various questions.