Operator Overloading in PHP

CismonX

Short Contents

Table of Contents

1 Operator Overloading

Operator overloading is a syntactic sugar available in various programming languages (e.g. C++, Python, Kotlin). This feature contributes to cleaner and more expressive code when performing arithmetic-like operations on objects.

For example, when using a `Complex` class in PHP, you may want to:

```
$a = new Complex(1.1, 2.2);  // 1.1 + 2.2j
$b = new Complex(1.2, 2.3);  // 1.2 + 2.3j

$c = $a * $b / ($a + $b);
```

Instead of:

```
$c = $a->mul($b)->div($a->add($b));
```

Although there is an RFC to provide this feature for PHP, it's not yet taken into consideration, which means we can't simply overload operators in userland PHP.

Fortunately, operator overloading can be achieved in native PHP with a little bit of invocation to several Zend APIs, without having to temper with any internal code of the PHP interpreter. The PECL operator extension already does that for us (the releases are out of date, see the git master branch for PHP7 support).

In this article, we will talk about the details of implementing operator overloading in a native PHP extension. We assume that the readers know the basics of the C/C++ programming language and basics of the Zend implementation of PHP.

2 Opcodes in PHP

Before a PHP script can be executed in Zend VM, it is compiled into arrays of `zend_op`s. Similar to machine codes, a `zend_op` consists of an instruction, at most two operands, and the operation result.

```
struct _zend_op {
    const void *handler;         // Function pointer to opcode handler.
    znode_op    op1;             // First operand.
    znode_op    op2;             // Second operand.
    znode_op    result;          // Instruction result.
    uint32_t    extended_value;  // Extra data corresponding to this opline.
    uint32_t    lineno;          // Line numper of this opline.
    zend_uchar  opcode;          // Instruction code.
    zend_uchar  op1_type;        // Type of first operand.
    zend_uchar  op2_type;        // Type of second operand.
    zend_uchar  result_type;     // Type of instruction result.
};
```

2.1 Operands

A `union znode_op` stores the offset or pointer to the referring target.

```
typedef union _znode_op {
    uint32_t      constant;
    uint32_t      var;
    uint32_t      num;
    uint32_t      opline_num;
#if ZEND_USE_ABS_JMP_ADDR
    zend_op      *jmp_addr;
#else
    uint32_t      jmp_offset;
#endif
#if ZEND_USE_ABS_CONST_ADDR
    zval         *zv;
#endif
} znode_op;
```

As said in zend_compile.h:

> On 64-bit systems, less optimal but more compact VM code leads to better performance. So on 32-bit systems we use absolute addresses for jump targets and constants, but on 64-bit systems relative 32-bit offsets.

The `ZEND_USE_ABS_JMP_ADDR` and `ZEND_USE_ABS_CONST_ADDR` macros are defined to 0 when PHP is compiled on 64-bit systems, thus `znode_op` is always 32 bits in size.

2.2 Instructions

The instruction codes are defined in zend_vm_opcodes.h. Operators are converted to corresponding instruction codes when PHP scripts are compiled.

For example, the following assembly-like code represents `$c = $a + $b` (You can try that yourself using phpdbg):

```
ADD    $a, $b, ~0  # "+" operator
ASSIGN $c, ~0      # "=" operator
```

However, not all operators have a corresponding instruction (e.g. negation, greater than, not less than operators). For example, PHP code `$c = $a > -$b` is compiled to something like:

```
MUL        $b, -1, ~0  # Converted to "$b * (-1)" (since PHP 7.3).
IS_SMALLER ~0, $a, ~1  # Converted to "<".
ASSIGN     $c, ~1
```

2.3 Operand Types

Operand type can be of following:

```
#define IS_UNUSED   0
#define IS_CONST    (1<<0)
#define IS_TMP_VAR  (1<<1)
#define IS_VAR      (1<<2)
#define IS_CV       (1<<3)  // Compiled variable
```

- If the operand is *not* used by the instruction, or the instruction doesn't generate a result, the type of corresponding `znode_op` is `IS_UNUSED`.
- If the operand is a **literal**, its type is `IS_CONST`.
- If the operand is the **temporary value** returned by an expression, its type is `IS_TMP_VAR`.
- If the operand is a **variable** known at **compile time**, its type is `IS_CV`.
- If the operand is a **variable** returned by an **expression**, its type is `IS_VAR`.

For example, the following PHP code:

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

Compiles to:

```
#                         (op1     op2     result) type
ASSIGN     $a, 1       #  CV      CONST   UNUSED
ADD        $a, 1,  ~1  #  CV      CONST   TMP_VAR
FREE       ~1          #  TMP_VAR UNUSED  UNUSED
ADD        $a, 1,  ~2  #  CV      CONST   TMP_VAR
ASSIGN     $b, ~2      #  CV      TMP_VAR UNUSED
ASSIGN_ADD $a, 1       #  CV      CONST   UNUSED
ASSIGN_ADD $a, 1,  @5  #  CV      CONST   VAR
ASSIGN     $b, @5, @6  #  CV      VAR     VAR
ASSIGN     $c, @6      #  CV      VAR     UNUSED
```

We can see that, for an assignment instruction, whether it has a result depends on whether the result is used or not. But for non-assignment instructions, the result is *always* stored in a temporary variable, even when the result is unused, in case it needs to be freed.

3 Opcode Handlers

An opcode handler is a function which executes a `zend_op`. The Zend API provides us with the ability to replace built-in opcode handlers with user-defined ones:

```
ZEND_API int
zend_set_user_opcode_handler(
    zend_uchar            opcode,
    user_opcode_handler_t handler
);
```

Where `opcode` is the instruction code to be overridden, and `handler` is the pointer to the user-defined handler function.

```
typedef int (*user_opcode_handler_t) (zend_execute_data *execute_data);
```

The handler function should accept `execute_data` pointer as argument, and returns an `int` indicating the execution status of the handler function, which could be one of the following values:

```
#define ZEND_USER_OPCODE_CONTINUE   0
#define ZEND_USER_OPCODE_RETURN     1
#define ZEND_USER_OPCODE_DISPATCH   2
#define ZEND_USER_OPCODE_ENTER      3
#define ZEND_USER_OPCODE_LEAVE      4
```

In most cases, we only need the two return values explained below:

- `ZEND_USER_OPCODE_CONTINUE`: The ‘zend_op‘ is successfully executed, and the program should proceed to the next line of opcode.
- `ZEND_USER_OPCODE_DISPATCH`: Fall back to the built-in opcode handler.

Once the handler is set, it will be invoked by the Zend Engine whenever a `znode_op` with a corresponding instruction is about to be executed. Note that multiple calls to `zend_set_user_opcode_handler` replace old handlers with new ones.

To disable a user-defined opcode handler, pass `NULL` to the `handler` argument.

3.1 Handler Implementation

First, we define a general-purpose handler function template in C++. The `handler` argument contains the implementation of the opcode handler. It accepts three `zval` pointers (i.e. two operands and result), and returns a `bool` indicating whether the instruction is executed within this handler.

```
template <typename F>
int op_handler(
    zend_execute_data *execute_data,
    F                  handler
) {
    // ... initialization here
    if (!handler(op1, op2, result)) {
        return ZEND_USER_OPCODE_DISPATCH;
    }
```

```
    // ... clean up here
    return ZEND_USER_OPCODE_CONTINUE;
}
```

Then, we initialize the handler function. Fetch the pointer to the current line of opcode from `execute_data`, and pointers to each operand `zval` from `opline`.

```
const zend_op *opline = EX(opline);
zend_free_op free_op1, free_op2;
zval *op1 = zend_get_zval_ptr(opline,
        opline->op1_type, &opline->op1, execute_data, &free_op1, 0);
zval *op2 = zend_get_zval_ptr(opline,
        opline->op2_type, &opline->op2, execute_data, &free_op2, 0);
zval *result = opline->result_type ? EX_VAR(opline->result.var) : nullptr;
```

A operand may be a reference to another `zval`. We would want to first dereference it before use.

```
if (EXPECTED(op1)) {
    ZVAL_DEREF(op1);
}
if (op2) {
    ZVAL_DEREF(op2);
}
```

Now `handler` can be invoked. Before continuing to the next line of opcode, don't forget to free the operands (if necessary) and increment `EX(opline)`.

```
if (free_op2) {
    zval_ptr_dtor_nogc(free_op2);
}
if (free_op1) {
    zval_ptr_dtor_nogc(free_op1);
}
// No need to free ‘result‘ here.
EX(opline) = opline + 1;
```

Finally, register the handler functions.

```
int
add_handler(
    zend_execute_data *execute_data
) {
    return op_handler(execute_data, [] (auto zv1, auto zv2, auto rv) {
        if (/* Whether we should overload "+" operator */) {
            // ... do something
            return true;
        }
        return false;
    });
}
// Opcode handlers are usually registered on module init.
PHP_MINIT_FUNCTION(my_extension)
```

```
{
    zend_set_user_opcode_handler(ZEND_ADD, add_handler);
}
```

4 Overloading Operators

Now we know that operator overloading in PHP can be achieved by setting user-defined opcode handlers. However, we should be careful with some details when implementing these functions, otherwise the operators may not work properly as expected.

4.1 Binary Operators

Syntax	Instruction
`$a + $b`	`ZEND_ADD`
`$a - $b`	`ZEND_SUB`
`$a * $b`	`ZEND_MUL`
`$a / $b`	`ZEND_DIV`
`$a % $b`	`ZEND_MOD`
`$a ** $b`	`ZEND_POW`
`$a << $b`	`ZEND_SL`
`$a >> $b`	`ZEND_SR`
`$a . $b`	`ZEND_CONCAT`
`$a | $b`	`ZEND_BW_OR`
`$a & $b`	`ZEND_BW_AND`
`$a ^ $b`	`ZEND_BW_XOR`
`$a === $b`	`ZEND_IS_IDENTICAL`
`$a !== $b`	`ZEND_IS_NOT_IDENTICAL`
`$a == $b`	`ZEND_IS_EQUAL`
`$a != $b`	`ZEND_IS_NOT_EQUAL`
`$a < $b`	`ZEND_IS_SMALLER`
`$a <= $b`	`ZEND_IS_SMALLER_OR_EQUAL`
`$a xor $b`	`ZEND_BOOL_XOR`
`$a <=> $b`	`ZEND_SPACESHIP`

A binary operator takes two operands, and always returns a value. Modification of either operand is allowed, provided that the operand type is `IS_CV`.

Note that there is no `ZEND_IS_GREATER` or `ZEND_IS_GREATER_OR_EQUAL` operator as said in Section 2.2 [#2.2], page 2. Although the PECL operator extension does some hack with `extended_value` of `zend_op` to distinguish whether the opcode is compiled from `<` or `>`, it requires patching the PHP source code and may break future compatibility.

The recommended alternative solution is shown below.

```
int
is_smaller_handler(
    zend_execute_data *execute_data
) {
    return op_handler(execute_data, [] (auto zv1, auto zv2, auto rv) {
        if (Z_TYPE_P(zv1) == IS_OBJECT) {
            if (__zobj_has_method(Z_OBJ_P(zv1), "__is_smaller")) {
                // Call `$zv1->__is_smaller($zv2)`.
                return true;
```

```
            }
        } else if (Z_TYPE_P(zv2) == IS_OBJECT) {
            if (__zobj_has_method(Z_OBJ_P(zv2), "__is_greater")) {
                // Call `$zv2->__is_greater($zv1)`.
                return true;
            }
        }
        return false;
    });
}
```

4.2 Binary Assignment Operators

Syntax	Instruction
`$a += $b`	`ZEND_ASSIGN_ADD`
`$a -= $b`	`ZEND_ASSIGN_SUB`
`$a *= $b`	`ZEND_ASSIGN_MUL`
`$a /= $b`	`ZEND_ASSIGN_DIV`
`$a %= $b`	`ZEND_ASSIGN_MOD`
`$a **= $b`	`ZEND_ASSIGN_POW`
`$a <<= $b`	`ZEND_ASSIGN_SL`
`$a >>= $b`	`ZEND_ASSIGN_SR`
`$a .= $b`	`ZEND_ASSIGN_CONCAT`
`$a |= $b`	`ZEND_ASSIGN_BW_OR`
`$a &= $b`	`ZEND_ASSIGN_BW_AND`
`$a ^= $b`	`ZEND_ASSIGN_BW_XOR`
`$a = $b`	`ZEND_ASSIGN`
`$a =& $b`	`ZEND_ASSIGN_REF`

Binary assignment operators behaves similar to non-assignment binary operators, with the exception that it should not generate a result when it is not used (`opline->result_type == IS_UNUSED`). When you overload these operators, make sure that you *never* touch the result `zval` under such circumstances.

The execution result of a binary assignment operator is expected to replace the first operand. However, it is not mandatory and is not done automatically by the Zend Engine.

Code example:

```
int
assign_add_handler(
    zend_execute_data *execute_data
) {
    return op_handler(execute_data, [] (auto zv1, auto zv2, auto rv) {
        if (Z_TYPE_P(zv1) == IS_OBJECT) {
            // ... handle addition.
            __update_value(Z_OBJ_P(zv1), add_result);
            if (rv != nullptr) {
                ZVAL_COPY(rv, zv1);
```

```
            }
            return true;
        }
        return false;
    });
}
```

4.3 Unary Operators

Syntax	Instruction
`~$a`	`ZEND_BW_NOT`
`!$a`	`ZEND_BOOL_NOT`

A unary operator takes one operand (`opline->op1`), and always returns a value. Modification of the operand is allowed, provided that the operand type is `IS_CV`.

There's no opcode for negation operator `-$a` or unary plus operator `+$a`, as said in Section 2.2 [#2.2], page 2, because they are compiled to multiplication by `-1` and `1`. In cases where they are not expected to behave identically, add some logic to the `ZEND_MUL` handler to workaround this.

Note that compatibility issues exists between PHP 7.3 and versions below 7.3.

PHP	Syntax	Instruction	Operand 1	Operand 2
7.3, 7.4	`-$a` or `+$a`	`ZEND_MUL`	`$a`	`-1` or `1`
7.1, 7.2	`-$a` or `+$a`	`ZEND_MUL`	`-1` or `1`	`$a`

Here's a simple example to workaround the negation operator for all major PHP versions.

```
int mul_handler(
    zend_execute_data *execute_data
) {
    return op_handler(execute_data, [] (auto zv1, auto zv2, auto rv) {
        if (Z_TYPE_P(zv1) == IS_OBJECT) {
#if PHP_VERISON_ID >= 70300
            if (Z_TYPE_P(zv2) == IS_LONG && Z_LVAL_P(zv2) == -1) {
                // Handle `-$zv1`.
                return true;
            }
#endif
            // Handle `$zv1 * $zv2`.
            return true;
        } else if (Z_TYPE_P(zv2) == IS_OBJECT) {
#if PHP_VERISON_ID < 70300
            if (Z_TYPE_P(zv1) == IS_LONG && Z_LVAL_P(zv1) == -1) {
                // Handle `-$zv2`.
                return true;
            }
#endif
            // Handle `$zv1 * $zv2`.
```

```
                return true;
            }
            return false;
        });
    }
```

4.4 Unary Assignment Operators

Syntax	Instruction
++$a	ZEND_PRE_INC
$a++	ZEND_POST_INC
--$a	ZEND_PRE_DEC
$a--	ZEND_POST_DEC

Unary assignment operators differ from each other. Post-increment/decrement operators behave identical to non-assignment unary operators, while pre-increment/decrement operators behave identical to binary assignment operators with the exception that they accept only one operand.

This behavior is not hard to understand, as in normal circumstances, the operand of a pre-increment/decrement operator is returned as execution result (with type `IS_VAR`), while a post-increment/decrement operator has to copy itself to a temporary variable and return it as execution result (with type `IS_TMP_VAR`).

5 Notes

Overriding opcode handlers can be used for various purposes other than operator overloading. For example, when you are implementing a profiler, you may want to create a custom handler for `ZEND_INIT_FCALL` and `ZEND_RETURN`.

However, every coin has two sides. Overriding opcode handlers can damage overall performance of your PHP script, as additional handler functions are called every time when executing a hooked opcode.

5.1 Constant Folding

Try compiling the following PHP script:

```
$a = 2 + 3 * (7 + 9);
$b = 'foo' . 'bar';
```

You will get:

```
ASSIGN $a, 50
ASSIGN $b, "foobar"
```

You can see that, the value of `$a` and `$b` is calculated at compile time, and there's neither arithmetic operations nor string concatenation when the script is running.

The PHP compiler does some optimizations (known as constant folding) by recognizing constant expressions and evaluate them with function `zend_const_expr_to_zval`, defined in zend_compile.c. Opcode handlers are fetched with functions like `get_binary_op`, `get_unary_op()` that hard-code the built-in handler functions.

Therefore, if you overload these operators in your extension, the custom handlers won't be invoked.

6 Examples

For a full usable example, you may want to see an implementation of class `Complex`, which is a part of a work-in-progress PHP extension I'm currently working on.

- complex.hh: contains handler functions for class `Complex`.
- complex.cc: contains the implementation of class `Complex`.
- operators.cc: contains the implementation of operator overloading.
- 002-complex-operators.phpt: tests whether operator overloading works for `Complex` class object.