Inhoud

1 Hoofdstuk

Hoe PHP werkt – van broncode tot weergave

door Thomas Punt
Onder redactie van Younes Rafie

Er gebeurt veel onder de motorkap wanneer we een stukje PHP-code uitvoeren. In grote lijnen doorloopt de PHP-interpreter bij het uitvoeren van code vier fasen. In dit hoofdstuk zullen we deze fasen vluchtig doornemen en laten zien hoe we de uitvoer van elke fase kunnen bekijken om te zien wat er precies gebeurt. Sommige van de gebruikte extensies maken al deel uit van de PHP-installatie (zoals tokenizer en OPcache), andere moeten handmatig worden geïnstalleerd en ingeschakeld (zoals php-ast en VLD).

In dit hoofdstuk:

Wat lexicale analyse is.

Hoe parsering werkt.

Wat er tijdens de compilatie gebeurt.

Interpretatie.

Fase 1: lexicale analyse

Lexicale analyse (of *tokenising*) is het proces waarbij een tekenreeks (in dit geval PHP-broncode) wordt omgezet in een reeks symbolen (*tokens*). Dit symbool is een benoemde ID (*named identifier*) voor de waarde waarmee die overeenkomt. PHP gebruikt re2c (**re2c.org**) om de lexer (ook wel scanner genoemd) te genereren vanuit het definitiebestand zend_language_scanner.l (**github.com/php/php-src/blob/master/Zend/zend_language_scanner.l**).

We kunnen de gegenereerde lexer bekijken met de PHP-extensie tokenizer (**php.net/manual/en/book.tokenizer.php**):

```
$code = <<<'code'
<?php
$a = 1;
code;
$tokens = token_get_all($code);
foreach ($tokens as $token) {
   if (is_array($token)) {
      echo "Line {$token[2]}: ", token_name($token[0]), " ('{$token[1]}')", PHP_EOL;
   } else {
      var_dump($token);
   }
}
```

De uitvoer is:

```
Line 1: T_OPEN_TAG ('<?php
')
Line 2: T_VARIABLE ('$a')
Line 2: T_WHITESPACE (' ')
string(1) "="
Line 2: T_WHITESPACE (' ')
Line 2: T_LNUMBER ('1')
string(1) ";"
```

De uitvoer bevat enkele opmerkelijke punten. Allereerst zijn niet alle onderdelen van de broncode benoemde symbolen. In plaats daarvan worden sommige symbolen beschouwd als een

symbool op zichzelf (zoals =, ;, :, ?, enzovoort). Het tweede punt is dat de lexer iets meer doet dan alleen het uitvoeren van een stroom symbolen. In de meeste gevallen worden ook de waarde die overeenkomt met het symbool en het regelnummer van het overeenkomende symbool opgeslagen (dat wordt gebruikt voor bijvoorbeeld *stack traces*).

Fase 2: parsering

De parser wordt ook gegenereerd, door Bison (**www.gnu.org/software/bison/**) met een BNF-grammaticabestand (**github.com/php/php-src/blob/master/Zend/zend_language_parser.y**). PHP gebruikt een LALR (1) (look ahead, left-to-right) contextvrije grammatica. Look ahead betekent eenvoudigweg dat de parser in staat is om n symbolen vooruit te kijken (in dit geval 1) om eventuele dubbelzinnigheden op te lossen. Left-to-right wil zeggen dat de stroom van links naar rechts wordt ontleed.

De gegenereerde parser gebruikt de symboolstroom van de lexer als invoer en heeft twee taken. Eerst wordt de geldigheid van de symboolvolgorde gecontroleerd door te kijken of deze overeenkomt met een grammaticaregel in het BNF-grammaticabestand. Dit zorgt ervoor dat er geldige taalconstructies worden gevormd door de symbolen in de stroom. De tweede taak van de parser is het genereren van de abstracte syntaxisboom (AST, *abstract syntax tree*) – een boomstructuur van de broncode die in de volgende fase (compilatie) wordt gebruikt.

Met de extensie php-ast (**github.com/nikic/php-ast**) kunnen we een *vorm* van de AST bekijken die door de parser wordt gemaakt. We zien niet direct de interne AST, want die is niet erg bruikbaar (in termen van consistentie en algemene bruikbaarheid). Daarom voert de extensie php-ast enkele transformaties uit die het werken met de AST prettiger maken.

Laten we eens een minimaal stukje code van de AST bekijken:

```
$code = <<<'code'
<?php
$a = 1;
code;
print_r(ast\parse_code($code, 30));
```

Uitvoer:

```
ast\Node Object (
   [kind] => 132
   [flags] => 0
   [lineno] => 1
   [children] => Array (
      [0] => ast\Node Object (
         [kind] => 517
         [flags] => 0
         [lineno] => 2
         [children] => Array (
            [var] => ast\Node Object (
               [kind] => 256
               [flags] => 0
               [lineno] => 2
               [children] => Array (
                  [name] => a
               )
            )
            [expr] => 1
         )
      )
   )
)
```

De knooppunten van de boom (meestal van het type ast\Node) hebben verschillende eigenschappen:

- `kind` Een geheel getal voor het type knooppunt; elk type heeft een overeenkomstige constante (bijvoorbeeld `AST_STMT_LIST` => 132, `AST_ASSIGN` => 517, `AST_VAR` => 256).
- `flags` Een geheel getal dat overloaded gedrag specificeert (een knooppunt ast\AST_BINARY_OP heeft bijvoorbeeld vlag-

gen voor het onderscheid tussen verschillende binaire bewerkingen).

- `lineno` Het regelnummer, zoals beschreven bij de lexicale analyse.
- `children` Subknooppunten, meestal delen van het knooppunt dat verder uitgesplitst wordt (een functieknooppunt heeft onder andere als kinderen: parameters, returntype, body).

De AST-uitvoer van deze fase is een handige basis voor bijvoorbeeld *static code analysers* zoals Phan (github.com/phan/phan).

Fase 3: compilatie

De compilatiefase gaat in omgekeerde volgorde door de AST en genereert daarbij opcode. In deze fase worden ook enkele optimalisaties uitgevoerd. Deze omvatten het omzetten van sommige functieaanroepen met literale argumenten (zoals `strlen("abc")` naar `int(3)`) en het invouwen van constante wiskundige expressies (zoals 60*60*24 tot `int(86400)`).

We kunnen de opcode-uitvoer in deze fase op een aantal manieren inspecteren, waaronder met OPcache (**php.net/manual/en/book.opcache.php**), VLD (**derickrethans.nl/projects.html#vld**) en PHPDBG (**phpdbg.com**). Vanwege de vriendelijke uitvoer wordt hier VLD gebruikt.

Laten we eens kijken wat de uitvoer is voor het script file.php:

```
if (PHP_VERSION === '7.1.0-dev') {
   echo 'Yay', PHP_EOL;
}
```

De afbeelding toont de uitvoer van de volgende opdracht:

```
php -dopcache.enable_cli=1 -dopcache.optimization_level=0 -dvld.active=1
   -dvld.execute=0 file.php
```

```
line     #* E I O op                            fetch          ext  return  operands
---------------------------------------------------------------------------------------
   3     0  E > > JMPZ                                                    <true>, ->3
   4     1    >   ECHO                                                    'Yay'
         2        ECHO                                                    '%0A'
   7     3    > > RETURN                                                  1
```

Afbeelding 1.1 *De uitvoer van de opdracht.*

De opcodes lijken voldoende op de originele broncode om de basisbewerkingen te kunnen volgen. (Ik ga hier niet in op de details van opcodes, omdat dat een boek op zich zou vergen.) Op opcodeniveau zijn in het script geen optimalisaties toegepast, maar zoals we kunnen zien is in de compilatiefase de voorwaarde met de constante (`PHP_VERSION === '7.1.0-dev'`) omgezet naar true.

OPcache doet meer dan alleen het cachen van opcodes (daarbij de fasen van lexicale analyse, parsering en compilatie omzeilend). Het is ook voorzien van vele verschillende optimalisatieniveaus. Laten we het niveau eens op vier stappen zetten en zien wat eruit komt. U ziet achtereenvolgens de opdracht en de uitvoer:

```
php -dopcache.enable_cli=1 -dopcache.optimization_level=1111 -dvld.active=-1
   -dvld.execute=0 file.php
```

```
line     #* E I O op                            fetch          ext  return  operands
---------------------------------------------------------------------------------------
   4     0  E >   ECHO                                                    'Yay%0A'
   7     1      > RETURN                                                  1
```

Afbeelding 1.2 *De uitvoer van de opdracht.*

We zien dat de voorwaarde is verwijderd en dat de twee ECHO-opdrachten zijn samengevoegd tot één opdracht. Dit is slechts een voorproefje van de vele optimalisaties die OPcache toepast bij het herhaaldelijk doorlopen van de opcodes van een script. Ik zal hier echter niet de verschillende optimalisatieniveaus bespreken.

Fase 4: interpretatie

De laatste fase is de interpretatie van de opcodes. Hier worden de opcodes uitgevoerd op de Zend Engine (ZE) VM. Er is eigenlijk heel weinig te zeggen over deze fase (tenminste, gezien vanaf hoog niveau). De uitvoer is zo'n beetje wat een PHP-script uitvoert met opdrachten zoals `echo`, `print`, `var_dump` enzovoort.

In plaats van het in dit stadium nog ingewikkelder te maken, volgt hier een aardigheidje: PHP is van zichzelf afhankelijk bij het genereren van zijn eigen VM. Dit komt doordat de VM wordt gegenereerd door een PHP-script, omdat dit eenvoudiger te schrijven en gemakkelijker te onderhouden is.

Samenvatting

We hebben kort stilgestaan bij de vier fasen die de PHP-interpreter doorloopt bij het uitvoeren van PHP-code. Dit behelsde het gebruik van verschillende extensies (zoals tokenizer, php-ast, OPcache en VLD) om de output van elke fase te verwerken en weer te geven. Ik hoop dat u met dit verhaal een beter algemeen begrip van de PHP-interpreter hebt gekregen en ook het belang van de extensie OPcache (zowel voor caching als voor optimalisatie) inziet.