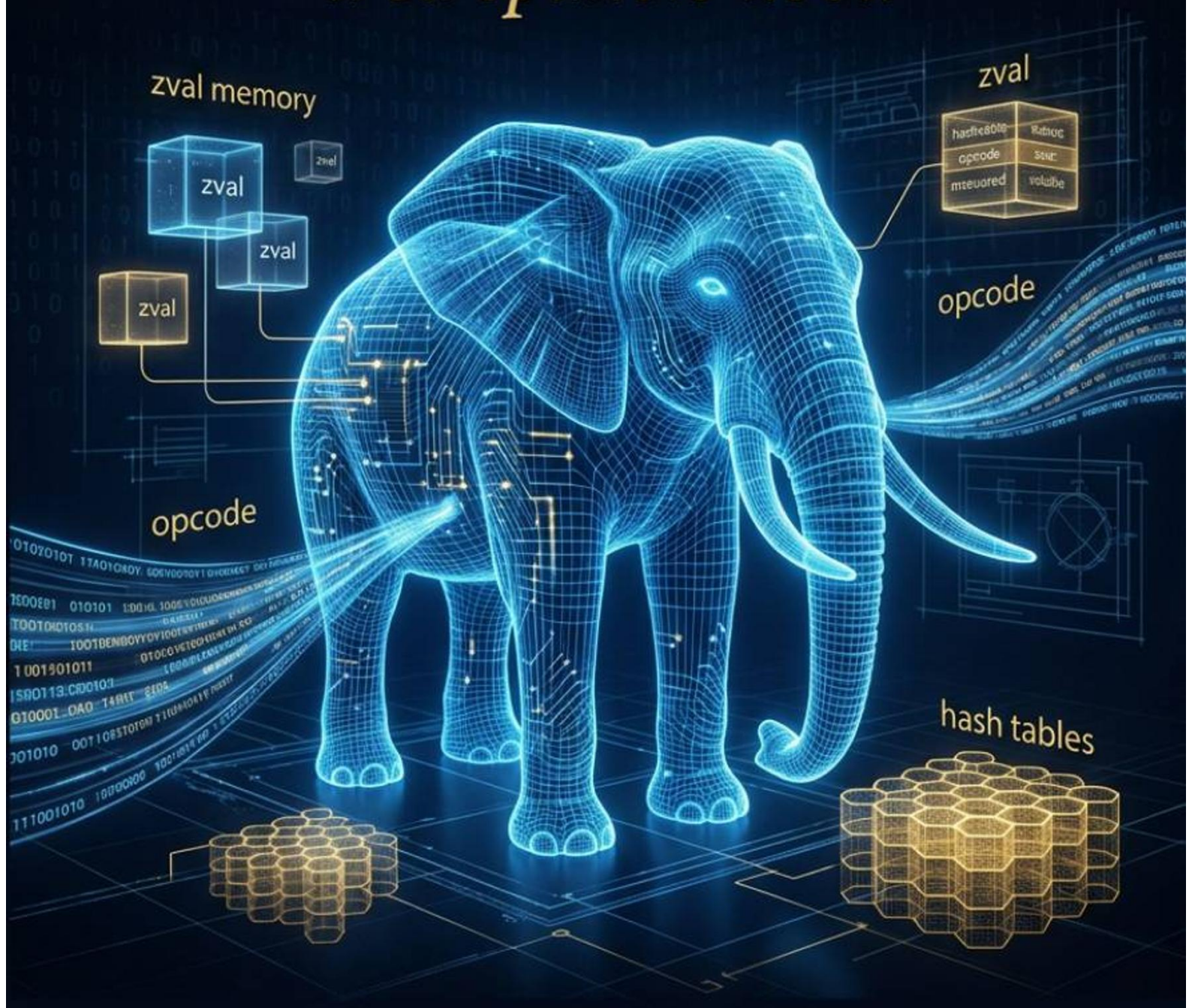
PHP. ПОД КАПОТОМ

ИНТЕРНАЛС

Архитектура, память и за гранью кода



МАКСИМ БАЙТОВИЧ

Максим Байтович

PHP. Под капотом. Архитектура, память и за гранью кода

«Автор»
2026

Байтович М.

PHP. Под капотом. Архитектура, память и за гранью кода /
М. Байтович — «Автор», 2026

Эта книга — не учебник по синтаксису. Она для тех, кто хочет понимать PHP на уровне инженера: как устроен Zend Engine, почему массивы иногда занимают вдвое меньше памяти, чем кажется, и откуда берутся segmentation faults. От жизненного цикла PHP-FPM процесса до прямых вызовов C-библиотек через FFI. От байт-кода и Copy-on-Write до Circuit Breaker и мутационного тестирования.12 глав и 5 военных историй из реальных продакшенов: утечки памяти в long-running процессах, падение сервера из-за файловых сессий, ошибка округления ценой в $47 000 и другие инциденты, которые учат лучше любой теории.Для middle и senior PHP-разработчиков, тимлидов и архитекторов, которые хотят писать надёжный, производительный и антихрупкий код.

Содержание

PHP. Под капотом. Архитектура, память и за гранью кода

Глава 1. Жизненный цикл запроса

В массовом сознании PHP-программистов живёт удобная, но неточная метафора:

«PHP рождается, обслуживает HTTP-запрос и умирает. Всё.»

В этой метафоре нет лжи, но она скрывает механизм, понимание которого отделяет пишущих на PHP от инженеров, работающих с PHP. Начнём с того, что метафору нужно разобрать на три неверных допущения.

Три мифа, которые мы разнесём в первой же главе

Миф 1: PHP — это интерпретатор, выполняющий ваш код строка за строкой.

PHP — транслятор в байт-код. Ваш index.php проходит лексический анализ, парсинг в AST (абстрактное синтаксическое дерево) и компиляцию в опкоды — низкоуровневые инструкции Zend Engine. Интерпретируется уже байт-код, а не исходный текст. Более того, с OPCache этот байт-код кешируется в разделяемой памяти, и повторные запросы вообще не касаются ваших .php-файлов. PHP ближе к Java HotSpot (с JIT в PHP 8+), чем к Bash.

Миф 2: «PHP умирает после каждого запроса».

Не совсем. Умирает контекст запроса. Сам процесс (в случае PHP-FPM) живёт и обслуживает сотни и тысячи запросов. Это означает, что:

Существует фаза инициализации модуля (MINIT), которая выполняется один раз при старте процесса.

Существует фаза инициализации запроса (RINIT), которая выполняется для каждого HTTP-запроса.

Существует фаза завершения запроса (RSHUTDOWN), после которой процесс не умирает, а возвращается в пул ожидания.

Если вы пишете расширение PHP на C, вы обязаны знать эти фазы. Если вы архитектор приложения, вы должны учитывать «утечки» между запросами в long-running режимах.

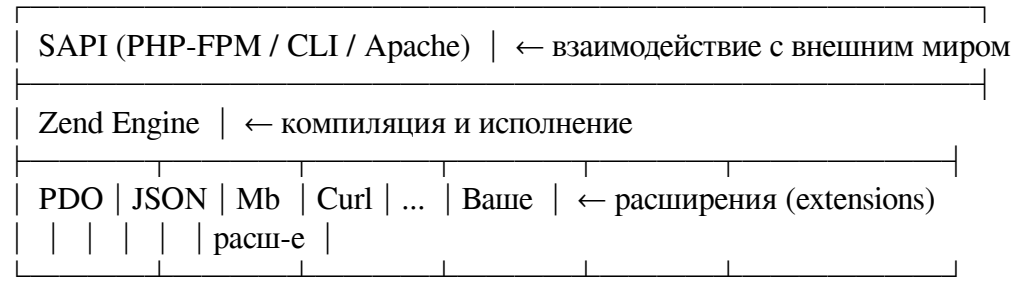
Миф 3: PHP работает только с HTTP.

SAPI (Server Application Programming Interface) — слой абстракции, отделяющий Zend Engine от способа взаимодействия с внешним миром. HTTP-запросы через Apache (mod_php), PHP-FPM (FastCGI) — лишь самые распространённые SAPI. Вы можете запустить PHP из командной строки (CLI SAPI), встроить его в другой сервер (Embed SAPI) или использовать для асинхронных демонов.

Давайте пройдём полный цикл — от нажатия пользователем Enter в браузере до возврата HTML.

1.1. Архитектура процесса: SAPI, ядро, расширения

Представьте три слоя.



SAPI принимает запрос, передаёт его Zend Engine и возвращает ответ. Серверное API ничего не знает о том, как парсить PHP; оно лишь организует вход и выход.

Zend Engine — сердце языка: компиляция в байт-код и виртуальная машина.

Расширения расширяют возможности ядра: от функций вроде array_map до PDO и собственных, написанных на C.

В контексте веб-сервера (PHP-FPM) используется FastCGI SAPI. Он запускает мастер-процесс, управляющий пулом воркеров. Каждый воркер — это долгоживущий процесс, внутри которого крутится бесконечный цикл: accept запроса → выполнение → ответ → ожидание следующего.

1.2. Жизненный цикл процесса PHP-FPM (взгляд изнутри C-расширения)

Для инженера, который пишет на C расширение PHP или отлаживает утечку памяти, жизненный цикл процесса состоит из чётких фаз.

MINIT (Module Initialization)

Выполняется один раз при старте процесса (воркера).

Что здесь происходит:

Регистрация классов, констант, ini-директив расширениями.

Выделение ресурсов, которые должны жить глобально в рамках процесса (например, подключение к shared memory).

Код расширения на C обычно содержит:

c

```
PHP_MINIT_FUNCTION(my_extension) {
    REGISTER_INI_ENTRIES();
    REGISTER_STRING_CONSTANT("MY_VERSION", "1.0", CONST_CS | CONST_PERSISTENT);
```

```
return SUCCESS;
}
```

Ключевой нюанс: MINIT исполняется до того, как пришёл первый запрос. Ресурсы, созданные здесь, не освобождаются до завершения процесса. Если здесь выделить память, которая не освобождается в MSHUTDOWN, получаем классическую утечку.

RINIT (Request Initialization)

Выполняется для каждого HTTP-запроса.

Здесь расширения инициализируют ресурсы, специфичные для запроса: суперглобальные массивы ($_GET, $_POST, $_SERVER), открывают соединения к базе данных, если они не persistent.

Ключевой нюанс: всё, что создано в RINIT, должно быть уничтожено в RSHUTDOWN. PHP предоставляет для этого механизм менеджера памяти с маркерами, но расширения обязаны корректно подчищать за собой, иначе память будет утекать с каждым запросом, убивая воркер медленно, но неизбежно.

Исполнение скрипта

Файл прочитан, скомпилирован (если не взят из OPCache), опкоды выполнены.

RSHUTDOWN (Request Shutdown)

Выполняется после отправки ответа (или фатальной ошибки) для каждого запроса.

Что здесь:

Вызов деструкторов объектов, которые не были освобождены.

Закрытие открытых файловых дескрипторов, созданных в этом запросе.

Сброс буферов вывода.

Освобождение Persistent Database Connections обратно в пул (но не закрытие!).

MSHUTDOWN (Module Shutdown)

Выполняется один раз, когда процесс воркера завершается.

Освобождаются глобальные ресурсы, зарегистрированные в MINIT. Выполняется при плавном перезапуске PHP-FPM (сигнал SIGQUIT), но не при аварийном SIGKILL.

1.3. Модель «ничего общего» (Shared-Nothing)

PHP спроектирован так, что каждый запрос стартует с чистого листа. Переменные, созданные в одном запросе, недоступны в другом. Это архитектурный выбор, а не ограничение.

Преимущества:

Нет гонок за состоянием между запросами.

Утечка памяти в одном запросе изолирована: RSHUTDOWN освободит всё, даже если вы забыли unset().

Простое масштабирование: запросы без состояния легко разбрасывать по пулу процессов.

Недостатки:

Невозможно хранить состояние между запросами в памяти PHP (требуются Redis, Memcached, APCu).

Каждый запрос платит цену инициализации (хотя OPCache и Preloading сильно снижают её).

Важный инженерный вывод: любая технология, превращающая PHP в долгоживущее приложение (Swoole, RoadRunner, ReactPHP), должна вручную эмулировать RSHUTDOWN для предотвращения пересечения состояний запросов.

1.4. PHP — транслятор в байт-код, а не интерпретатор текста

Давайте раз и навсегда разорвём ментальную связку «скриптовый язык == интерпретация строк».

Стадии превращения PHP-файла в исполняемый код

Шаг 1. Лексический анализ (Lexing)

Файл читается и превращается в поток токенов. Можно увидеть своими глазами:
bash

```
php -r 'token_get_all("<?php echo 42;");'
```

Вы получите массив: T_OPEN_TAG, T_ECHO, T_LNUMBER, ;. Это сырьё.

Шаг 2. Парсинг в AST (Abstract Syntax Tree)

Токены преобразуются в дерево, описывающее синтаксическую структуру программы. С PHP 7 у нас появился явный AST как внутренний этап до генерации опкодов.

Можно увидеть AST командой:
bash

```
php -r 'ast\parse_code("<?php echo 42;", $version=50);'
```

Вместо линейного потока токенов мы получаем иерархическое представление: AST_STMT_LIST → AST_ECHO → 42.

Шаг 3. Компиляция в опкоды

AST обходится, и генерируются операции виртуальной машины Zend Engine:

ZEND_ECHO

ZEND_RETURN

Для функции sum($a, $b) { return $a + $b; } опкоды выглядят так:

ZEND_RECV (получить параметры)

ZEND_ADD

ZEND_RETURN

Шаг 4. Выполнение на виртуальной машине

Опкоды выполняются виртуальной машиной — циклом, который последовательно диспетчеризует инструкции. С PHP 8.0 добавлен JIT-компилятор, который транслирует горячие участки опкодов прямо в машинный код процессора, минуя виртуальную машину.

1.5. OPCache: как избежать Шагов 1–3 для миллионов запросов

Поскольку PHP-FPM воркер живёт долго, а код приложения меняется редко, было бы безумием парсить index.php на каждом запросе. OPCache решает это, кешируя скомпилированный байт-код в разделяемой памяти (shared memory).

Трёхуровневое хранилище байт-кода

Memory-mapped файл — выделяется область памяти, доступная всем дочерним процессам PHP-FPM.

Хэш-таблица — ключ: путь к файлу + временная метка, значение: указатель на структуру с опкодами.

Структура с опкодами — содержит не только байт-код, но и interned strings, таблицу классов, функций.

При получении запроса PHP проверяет, есть ли в OPCache актуальный байт-код для index.php. Если да — лексический анализ, парсинг и компиляция пропускаются. Экономия может достигать 90% времени выполнения запроса для фреймворков с тысячами файлов.

Interned Strings

Строки, повторяющиеся в коде (имена классов, названия методов, ключи массивов), хранятся в специальной таблице интернированных строк. Вместо дублирования строки "getUserById" в сотне опкодов, хранится одна копия в shared memory, а опкоды ссылаются на неё по указателю. Это экономит память и ускоряет сравнение строк (сравнение указателей вместо memcmp).

Preloading (PHP 7.4+)

OPCache умеет загружать и компилировать PHP-файлы один раз при старте PHP-FPM сервера и держать их в памяти постоянно. Это шаг к модели Java/.NET, где код фреймворка загружается один раз на весь пул процессов.

В php.ini указывается скрипт:
ini

opcache.preload=/app/preload.php

А внутри preload.php вызывается opcache_compile_file() для ключевых классов. После этого классы Symfony или Doctrine становятся доступны всем воркерам без чтения с диска и парсинга.

Цена preloading'a: любые изменения в preloaded-файлах требуют перезапуска PHP-FPM. Вы жертвуете гибкостью «поменял файл — подхватилось» ради производительности продакшена.

Итог первой главы

Мы разорвали шаблон «PHP — это скриптуха для формочек». Под капотом работает инженерная система с фазами жизни процесса, компиляцией в байт-код, кешированием опкодов и JIT-компиляцией.

PHP-FPM воркер не умирает после запроса — умирает контекст (RINIT → RSHUTDOWN).

PHP — не интерпретатор, а транслятор в байт-код, выполняемый виртуальной машиной с опциональной JIT-компиляцией.

OPCache — не просто «кеш файлов», а сложная система с разделяемой памятью и интернированием строк.

Shared-nothing — архитектурный выбор, дающий изоляцию ценой невозможности хранить состояние в памяти процесса.

В следующей главе мы погрузимся в модель памяти PHP: узнаем, что такое zval, как работает Copy-on-Write, и почему после PHP 7 ваши массивы стали занимать в 2-3 раза меньше памяти.

Глава 2. Модель памяти и Zend Memory Manager

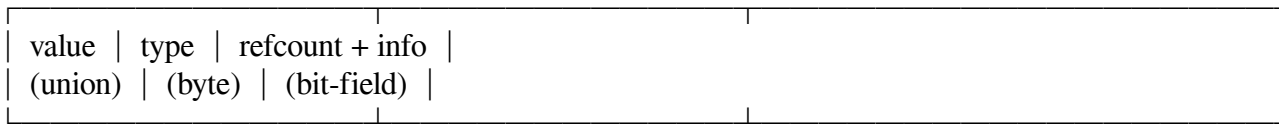
Если первая глава была о времени жизни PHP-процесса, то эта — о пространстве. О том, как PHP хранит ваши переменные, почему $b = $a не всегда означает копирование, и за что мы должны благодарить PHP 7, когда сервер держит не 1000, а 3000 запросов на гигабайте оперативной памяти.

Тема зайдёт глубоко. Но после неё вы никогда не будете смотреть на оператор = как на простое присваивание.

2.1. Zval: атомарная единица всего

В основе системы типов PHP лежит структура на C, называемая zval (Zend Value). Всё, что вы создаёте — числа, строки, массивы, объекты — в какой-то момент становится zval'ом внутри рантайма.

Упрощённо zval выглядит так:
text



value — union, который может хранить long, double, указатель на строку, указатель на объект, указатель на хэш-таблицу (массив) и так далее.

type — тип данных: IS_LONG, IS_DOUBLE, IS_STRING, IS_ARRAY, IS_OBJECT и другие.

refcount — счётчик ссылок (Reference Counting). Сколько переменных ссылаются на этот zval.

До PHP 7 эта структура была большой и тяжёлой. Размер zval'a достигал 48 байт (на 64-битных системах), и каждый zval выделялся отдельно в куче. Это означало, что $a = 1 требовал выделения памяти в куче для целого числа. Умножьте на миллионы переменных во фреймворке — и вот они, гигабайты.

2.2. Великая реформа PHP 7: упаковка и встраивание

Разработчики PHP 7 сделали две революционные вещи.
Во-первых, zval уменьшился до 16 байт.

Как?

value и type объединены: для простых типов (числа) значение хранится прямо в zval, без дополнительного указателя.

refcount сделан частью zval, но для простых типов он не используется (об этом ниже).

Во-вторых, для целых чисел и чисел с плавающей точкой zval больше не выделяется в куче.

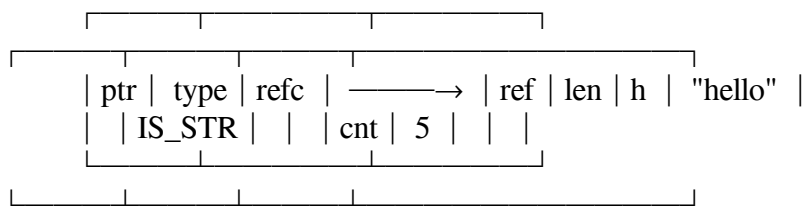
Значение размещается прямо внутри структуры. $a = 42 теперь буквально занимает 16 байт на стеке или в составной структуре, а не отдельный блок в куче.

Что дало это изменение? На реальных приложениях на WordPress и Drupal потребление памяти снизилось на 50–70%. Сервер, который задыхался при 1000 одновременных соединений, теперь держал 3000 без увеличения RAM.

2.3. Строки, массивы и объекты: zval как указатель

Для составных типов (строки, массивы, объекты) zval хранит указатель на отдельную структуру в куче. Рассмотрим на примере строки.

В PHP 7 строка представлена отдельной структурой zend_string:
text



refcnt (в структуре zend_string) — количество zval'ов, указывающих на эту строку.

len — длина строки (в байтах, не символах!).

h — предвычисленный хэш для использования в хэш-таблицах.

Сами символы строки.

Аналогично для массивов (zend_array — хэш-таблица) и объектов (указатель на zend_object).

Ключевое следствие: когда вы пишете $b = $a для строки, zval'ы копируются (16 байт), но zend_string — нет. Оба zval'а указывают на одну и ту же строку в куче, а refcnt строки становится равен 2.

2.4. Copy-on-Write: почему PHP не тормозит, когда вы «копируете» строки

Это, пожалуй, самая важная концепция производительности PHP. PHP не копирует данные, пока вы их не изменяете.

Давайте разберём классический пример:
php

$a = "Hello, World!"; // 1 zval, 1 строка в куче, refcnt строки = 1
$b = $a; // 2 zval'а, 1 строка в куче, refcnt строки = 2

```
// Память под строку не копировалась!
```

В этот момент $a и $b разделяют одну и ту же строку в памяти.

Теперь делаем изменение:
php

```
$b = "Goodbye!"; // Создаётся НОВАЯ строка в куче,
// refcnt старой строки уменьшается до 1,
// $b указывает на новую строку
```

До PHP 7 это работало медленнее из-за дополнительного уровня косвенности через «zval с разделением» (SEPARATE), но в 7-й версии механика стала элегантной и быстрой благодаря встроенным счётчикам ссылок прямо в zend_string.

Copy-on-Write в массивах
CoW работает и для массивов. Ключевое отличие: изменение массива с refcnt > 1 вызывает физическое копирование хэш-таблицы с рекурсивной проверкой элементов. Однако и здесь PHP умён:
php

```
$a = [1, 2, 3]; // массив с refcnt = 1
$b = $a; // refcnt массива = 2
$b[] = 4; // происходит разделение (SEPARATE),
// $a остаётся [1,2,3], $b становится новой копией + новый элемент
```

Проблема, которую нужно знать разработчикам фреймворков: передача массива по значению в функцию формально увеличивает refcnt при вызове, а изменение внутри функции вызывает копирование. Именно поэтому передача по ссылке (&$array) иногда оправдана — но только когда вы действительно планируете модификацию внутри.
2.5. Reference Counting и GC: когда счётчиков недостаточно

Счётчик ссылок великолепно справляется с простыми сценариями, но бессилен перед циклическими ссылками. Классика:
php

```
$a = new stdClass();
$b = new stdClass();
$a->ref = $b;
$b->ref = $a;
unset($a);
unset($b);
// Оба объекта всё ещё живы! refcnt каждого = 1 (второй указывает на первый и наоборот)
```

Счётчик ссылок каждого объекта не равен нулю, хотя из пользовательского кода они недоступны. Это классическая утечка памяти, и до PHP 5.3 она была фатальной для долгоживущих скриптов.
Циклический сборщик мусора (Cycle Collector)

PHP содержит дополнительный механизм — сборщик циклических ссылок. Он работает так:

Все возможные «корни» (zval'ы с флагом возможности цикла) помещаются в специальный буфер.

Когда буфер заполняется (по умолчанию 10 000 элементов), запускается сборщик.

Сборщик моделирует «уменьшение счётчиков»: он вычитает единицу из refcnt для каждой ссылки внутри графа.

Если после вычитания refcnt объекта становится 0 — значит, он жив только за счёт циклических ссылок. Объект помечается как мусор.

Сборщик освобождает память таких объектов.

Эту механику можно отключить: gc_disable(). Для скриптов с миллионами объектов и гарантированным отсутствием циклов это даёт прирост скорости (нет накладных расходов на поддержку буфера GC), но требует абсолютной уверенности в коде. Большинство приложений не должны этого делать никогда.

2.6. Менеджер памяти Zend MM: не просто malloc

PHP не полагается на системный malloc для каждого выделения байта. Вместо этого он использует собственный Zend Memory Manager, который:

Запрашивает большие блоки у ОС (через mmap или malloc), называемые сегментами.

Нарезает эти блоки на мелкие куски под zval'ы, строки, массивы.

Кеширует освобождённые блоки для повторного использования без обращения к ОС.

Это решает несколько проблем:

Фрагментация: системный аллокатор может фрагментировать память, PHP-аллокатор переиспользует блоки одинакового размера.

Скорость: выделение памяти из пула быстрее системного вызова.

Безопасность при завершении: при RSHUTDOWN PHP знает, какая память была выделена в течение запроса, и может одномоментно освободить её всю, даже если расширение или код разработчика «забыли» освободить отдельные блоки. Это не панацея от утечек (память, выделенная как persistent, живёт дольше), но мощная страховка.

Резюме главы

Мы спустились на уровень, где PHP перестаёт быть магией и становится понятной инженерной конструкцией:

Zval — 16-байтный контейнер. Для чисел значение внутри, для сложных типов — указатель.

PHP 7 упаковал zval и встроил простые значения, сократив потребление памяти драматически.

Copy-on-Write предотвращает копирование данных при присваивании; копия создаётся только при модификации.

Reference Counting освобождает память, когда zval больше не нужен.

Cycle Collector разруливает циклические ссылки, которые счётчик не видит.

Zend MM — это собственный аллокатор PHP, эффективный и заточенный под модель «запрос—очистка».

Теперь у нас есть понимание того, где и как живут данные. В следующей главе мы изучим, как Zend Engine превращает ваш PHP-код в эти самые zval'ы и опкоды, и как OPCache хранит их между запросами.

Глава 3. Zend Engine и OPCache

В первой главе мы сказали, что PHP — это транслятор в байт-код. Во второй — как этот байт-код оперирует данными в памяти. Теперь пришло время заглянуть в сердце машины: Zend Engine. Мы разберём, как исходный текст становится исполняемым кодом, и как OPCache позволяет не делать эту работу на каждом запросе.

Это глава о компиляции. Не бойтесь: мы не будем писать компилятор. Мы будем понимать его работу, чтобы писать лучший PHP.

3.1. Четыре фазы превращения текста в действие

Когда PHP получает файл index.php, он проходит четыре стадии, прежде чем будет выполнена первая инструкция echo.

text

Исходный код → Токены → AST → Опкоды → Исполнение

Каждая стадия — это отдельный проход, и каждая может быть изучена и отлажена отдельно.

Фаза 1: Лексический анализ (Lexing)

Задача: превратить поток байтов в поток осмысленных «слов» — токенов.

Лексер (сканер) читает исходный файл символ за символом и группирует их в токены. Каждый токен — это пара: тип + значение.

Напишем простой код:
php

```
<?php
$a = 42;
echo $a;
```

Лексер выдаст поток:
text

```
T_OPEN_TAG ("<?php\n")
T_VARIABLE ("$a")
T_WHITESPACE (" ")
= (оператор присваивания)
T_LNUMBER ("42")
; (конец инструкции)
T_ECHO ("echo")
T_VARIABLE ("$a")
; (конец инструкции)
```

Вы можете увидеть токены самостоятельно:

bash

```
php -r 'print_r(token_get_all(file_get_contents("test.php")));'
```

Что здесь важно инженеру:

Лексер не понимает синтаксис. $a = ; — валидный набор токенов (T_VARIABLE, '=', ';'). Синтаксическая ошибка обнаружится позже.

Лексер очень быстрый. Это конечный автомат. Он не аллоцирует сложные структуры, только токены.

Комментарии и пробелы — тоже токены (T_WHITESPACE, T_COMMENT). Они будут отброшены парсером.

Совет по производительности: лексер — не узкое место. Не пишите «оптимизированный PHP без пробелов и комментариев». OPCache сделает эту работу один раз.

Фаза 2: Синтаксический анализ, порождающий AST

Задача: проверить синтаксис и построить Абстрактное Синтаксическое Дерево (AST).

Парсер читает поток токенов и строит иерархическую структуру, отражающую смысл программы. Именно здесь $a = ; вызовет фатальную ошибку: парсер ожидает выражение после '=', но видит ';'.

Для нашего кода AST будет выглядеть примерно так:
text

```
ZEND_AST_STMT_LIST
 ├─── ZEND_AST_ASSIGN
 │   ├─── ZEND_AST_VAR ($a)
 │   └─── ZEND_AST_ZVAL (42)
 └─── ZEND_AST_STMT_LIST
 └─── ZEND_AST_ECHO
 └─── ZEND_AST_VAR ($a)
```

До PHP 7 парсер генерировал опкоды сразу, без явного AST. Это делало язык менее гибким. Появление явного AST позволило:

Улучшить сообщения об ошибках.

Создавать инструменты статического анализа (PHPStan, Psalm).

Проводить оптимизации на уровне дерева до генерации опкодов.

Вы можете увидеть AST (потребуется расширение ast):
bash

```
php -r '
```

```
$code = "<?php \$a = 42; echo \$a;";
var_dump(ast\parse_code($code, 70));
'
```

Парсер работает методом рекурсивного спуска. Такие монстры, как «LALR(1)-парсер», здесь не нужны: грамматика PHP контекстно-зависима и проще обрабатывается рекурсивным спуском с ручным разрешением неоднозначностей.

Фаза 3: Компиляция в опкоды

Задача: обойти AST и сгенерировать линейную последовательность инструкций для виртуальной машины.

Это сердце Zend Engine. Каждый узел AST транслируется в один или несколько опкодов. Опкод — это низкоуровневая инструкция виртуальной машины, обычно состоящая из:

Кода операции (ZEND_ECHO, ZEND_ASSIGN, ZEND_ADD, ZEND_RETURN, ...)

Операндов (обычно до трёх, ссылающихся на zval'ы)

Для нашего $a = 42; echo $a; компилятор сгенерирует примерно такие инструкции:
text

```
ZEND_ASSIGN $a, 42
ZEND_ECHO $a
ZEND_RETURN null
```

Реальный вывод можно увидеть через расширение vld (Vulcan Logic Dumper) или phpdbg:
bash

```
php -d opcache.opt_debug_level=0x20000 -r '$a = 42; echo $a;' 2>&1 | grep -E "ZEND|opline"
```

Или проще, через встроенную возможность OPCache:
bash

```
php -d opcache.opt_debug_level=0x10000 test.php
```

Компиляция делает несколько важных вещей:

Разрешение имён: переменные, функции, классы связываются с их внутренними представлениями.

Оптимизации времени компиляции: 1 + 2 будет вычислено прямо в опкод ZEND_ASSIGN $a, 3. Это называется constant folding (свёртка констант).

Генерация обработчиков исключений: для try/catch создаются таблицы переходов.

Результат компиляции — массив опкодов (op_array) для каждого файла, функции, метода и даже для «включённого» кода внутри eval().

Фаза 4: Исполнение на виртуальной машине

Задача: выполнить опкоды один за другим.

Виртуальная машина (ВМ) Zend Engine — это классический цикл:
text

пока (есть опкоды) {
прочитать следующий опкод
выполнить обработчик опкода
перейти к следующему опкоду
}

Каждый опкод имеет свой обработчик на C (или машинный код, если JIT скомпилировал его). Например:

Обработчик ZEND_ECHO берёт значение операнда, преобразует в строку и отправляет в буфер вывода.

Обработчик ZEND_ASSIGN берёт два операнда, записывает второй в первый с учётом Copy-on-Write и счётчиков ссылок.

ZEND_ADD складывает два числа, сохраняя результат в третий zval.

Важнейшая деталь: ВМ оперирует zval'ами, которые мы изучили в Главе 2. Каждый опкод создаёт, читает или модифицирует zval'ы. Понимание zval'ов — ключ к пониманию производительности.

С PHP 8.0 виртуальная машина получила JIT-компилятор (Just-In-Time). Он отслеживает «горячие» участки кода (те, что исполняются часто) и транслирует их напрямую в машинный код процессора x86/ARM, минуя интерпретацию опкодов. Это даёт существенный прирост на вычислительных задачах (математика, обработка изображений), но не так заметен на типичных веб-приложениях, где время выполнения уходит на I/O и ожидание базы данных.

3.2. OPCache: кеш, который меняет правила игры

Без кеширования все четыре фазы выполняются для каждого файла, каждого запроса. Для приложения на Symfony с 200+ подключаемых файлов это означало бы сотни тысяч операций парсинга и компиляции в секунду.

OPCache решает эту проблему радикально: он сохраняет результат третьей фазы (опкоды) и переиспользует его.

Что именно кешируется?

OPCache хранит в разделяемой памяти (shared memory) для каждого PHP-файла:

Массив опкодов (op_array) — готовая к исполнению программа.

Таблицу функций и классов, определённых в этом файле.

Interned strings — все строки из кода (имена переменных, функций, классов, констант) сохраняются в общую хэш-таблицу.

Метаданные: время модификации файла, контрольные суммы.

Как работает проверка актуальности?

При запросе PHP должен решить, можно ли использовать закешированный байт-код. Алгоритм проверки:

Берётся путь к файлу.

В shared memory ищется запись по этому пути.

Если запись найдена, сравнивается время модификации файла (mtime) с сохранённым.

Если совпадает — опкоды валидны, файл не читается с диска вообще.

Если не совпадает — файл перекомпилируется и кеш обновляется.

Этот механизм называется stat. При продакшен-деплое, когда все файлы меняются разом через атомарную замену директории, проверку stat можно отключить:
ini

opcache.validate_timestamps = 0

После этого OPCache никогда не проверяет файлы на диске. Производительность становится максимальной, но любое изменение кода требует перезапуска PHP-FPM или сброса кеша через opcache_reset().
Interned Strings: экономия памяти, скрытая от глаз

Представьте, что в сотне файлов вашего приложения встречается строка "getUserById" — название метода. Без интернирования каждый файл хранил бы свою копию этой строки. С OPCache:

Строка сохраняется один раз в общей таблице interned strings.

Каждый опкод, ссылающийся на эту строку, хранит не саму строку, а указатель на неё.

Сравнение строк превращается в сравнение указателей (O(1) вместо O(n)).

Это не микрооптимизация. На больших кодовых базах интернирование экономит мегабайты и десятки мегабайт оперативной памяти.
3.3. Preloading: PHP на стероидах

OPCache решил проблему повторной компиляции, но осталась проблема: каждый процесс PHP-FPM имеет свой собственный кеш опкодов. Когда воркер перезапускается, его кеш пуст, и первый запрос снова платит цену компиляции.

Preloading, введённый в PHP 7.4, решает эту проблему.

Как это работает:

В php.ini указывается preload-скрипт:

ini

```
opcache.preload = /app/preload.php
opcache.preload_user = www-data
```

Этот скрипт выполняется один раз при старте PHP-FPM мастер-процесса, до того, как создаются воркеры.

Внутри скрипта вызывается opcache_compile_file() для всех критически важных файлов:

php

```
<?php
// preload.php
opcache_compile_file('/app/src/Entity/User.php');
opcache_compile_file('/app/src/Service/UserService.php');
// ... сотни классов Symfony/Doctrine
```

Скомпилированные опкоды загружаются в разделяемую память OPCache и становятся доступны всем дочерним процессам.

Что это даёт:

Воркеры рождаются с «горячим» кешем фреймворка.

Память, занятая опкодами, распределяется между всеми воркерами (shared memory).

Экономия 30–50% памяти на воркер для типичного Symfony-приложения.

Первый запрос к новому воркеру не платит цену компиляции сотен файлов.

Цена preloading:

Изменение любого preloaded-файла требует перезапуска PHP-FPM.

Ошибка в preloaded-файле может сделать PHP-FPM неспособным запуститься.

Preloaded-классы нельзя переопределить через механизмы вроде class_alias — они буквально «запекаются» в память.

Требуется тщательная настройка: preload только то, что действительно нужно всегда.

3.4. Практические выводы для разработчика

Что из этого следует для вашего кода?

Пишите чистый, читаемый код. OPCache нивелирует стоимость пробелов, комментариев и длинных имён переменных. Лексический анализ — не узкое место.

Не оптимизируйте константные выражения вручную. 60 * 60 * 24 превращается в 86400 на этапе компиляции. Пишите выразительно: SECONDS_IN_A_DAY.

Файловый автолоад влияет на число компиляций. Каждый уникальный путь к файлу, который загружается через автолоад, требует проверки и потенциальной компиляции. PSR-4 с жёсткой структурой директорий помогает OPCache эффективнее управлять кешем.

Preloading требует дисциплины. Не preload'ьте классы с побочными эффектами (коннекты к БД, чтение конфигов). Preload'ьте чистые определения классов и функций.

JIT — не серебряная пуля. Для типичного веб-приложения выигрыш небольшой. JIT раскрывается на математических вычислениях, машинном обучении на PHP и обработке изображений.

Резюме главы

Zend Engine — это не чёрный ящик, а конвейер:
text

Исходный код → Лексер (токены) → Парсер (AST) → Компилятор (опкоды) → ВМ (исполнение)

OPCache прерывает этот конвейер после компиляции, сохраняя опкоды в разделяемой памяти и делая повторные запросы практически бесплатными с точки зрения CPU.

Preloading идёт дальше: он загружает опкоды в память ещё до прихода первого запроса, делая PHP похожим на «настоящие» долгоживущие приложения.

Теперь мы знаем, как PHP исполняет код. В следующей главе мы погрузимся в то, на чём он его исполняет: строки, бинарная безопасность, и почему strpos может вернуть false, который равен нулю.

Глава 4. Строки, которые мы (не) знаем

Строки в PHP обманчиво просты. Мы используем их каждый день: конкатенируем, обрезаем, ищем подстроки. Но за этим удобным фасадом скрывается инженерная конструкция, полная нюансов. Тот факт, что strpos() может вернуть false, который в нестрогом сравнении равен 0 — лишь вершина айсберга.

Начнём с самых основ и дойдём до оптимизаций на уровне памяти.

4.1. Бинарно-безопасные строки: PHP против C

В языке C строка — это последовательность байтов, заканчивающаяся нулевым байтом (\0). Функция strlen() в C считает байты, пока не встретит \0. Это фундаментальное ограничение: C-строка не может содержать нулевой байт внутри себя.

PHP с самого начала пошёл другим путём. PHP-строки бинарно-безопасны. Это означает:

Строка хранит свою длину явно (в поле len структуры zend_string, которую мы видели в Главе 2).

Нулевой байт — такой же легитимный символ, как и любой другой.

strlen() в PHP возвращает сохранённую длину и не зависит от наличия \0 внутри.

Демонстрация:
php

```
$binary = "Hello\0World";
echo strlen($binary); // 11, не 5!
echo $binary; // "Hello" — терминал обрезает, но строка полная
```

Почему это критически важно? Потому что PHP работает с бинарными данными постоянно. Изображения, загруженные через $_FILES, содержимое зашифрованных данных, сериализованные объекты — всё это бинарные строки с потенциальными нулевыми байтами. Если бы PHP использовал C-строки, каждый такой случай приводил бы к усечению данных.

Конец ознакомительного фрагмента.