

O'REILLY®

# Рецепты PHP

для профессиональных  
разработчиков



**SPRiNT**  
book

Эрик А. Манн

---

# **PHP Cookbook**

*Modern Code Solutions for  
Professional PHP Developers*

*Eric A. Mann*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

---

# Рецепты PHP

*Для профессиональных  
разработчиков*

Эрик А. Манн

**SPRiNT**  
book 2025

ББК 32.988-02-018.1

УДК 004.738.5

M23

### Манн Эрик А.

М23 Рецепты PHP. Для профессиональных разработчиков. — Астана: «Сprint Бук», 2025. — 432 с.: ил.

ISBN 978-601-08-4269-4

В этом сборнике рецептов разработчики на PHP найдут надежные и проверенные решения распространенных задач. PHP — удивительно простой язык программирования, что объясняет, почему на нем написано более 75 % веб-сайтов в Интернете. Но он также невероятно терпим к ошибкам программирования, что может привести к тиражированию сомнительного кода.

Эрик Манн предлагает собственные рецепты использования современных версий PHP для задач, встречающихся в повседневной практике программиста. Вы познакомитесь с паттернами и примерами, которые пригодятся любому разработчику, и сможете быстро находить и решать сложные задачи, не изобретая велосипед.

**16+** (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1098121327 англ.

Authorized Russian translation of the English edition PHP Cookbook  
ISBN 978-1098121327 © 2023 Eric A. Mann.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-601-08-4269-4

© Перевод на русский язык. ТОО «Сprint Бук», 2024

© Издание на русском языке, оформление. ТОО «Сprint Бук», 2024

---

# Краткое содержание

[https://t.me/it\\_boooks/2](https://t.me/it_boooks/2)

Предисловие .....	14
Глава 1. Переменные .....	18
Глава 2. Операторы .....	27
Глава 3. Функции .....	45
Глава 4. Строки .....	78
Глава 5. Числа .....	99
Глава 6. Дата и время .....	121
Глава 7. Массивы .....	143
Глава 8. Классы и объекты .....	182
Глава 9. Безопасность и шифрование .....	229
Глава 10. Работа с файлами .....	262
Глава 11. Потоки .....	274
Глава 12. Обработка ошибок .....	295
Глава 13. Отладка и тестирование .....	307
Глава 14. Настройка производительности .....	336
Глава 15. Пакеты и расширения .....	353
Глава 16. Базы данных .....	366
Глава 17. Асинхронный PHP .....	388
Глава 18. Командная строка PHP .....	415
Об авторе .....	430
Иллюстрация на обложке .....	431

---

# Оглавление

<b>Предисловие .....</b>	<b>14</b>
Для кого эта книга .....	15
Структура издания.....	15
Условные обозначения.....	16
Благодарности .....	17
От издательства.....	17
<b>Глава 1. Переменные .....</b>	<b>18</b>
1.1. Определение констант .....	20
1.2. Создание переменных переменных.....	22
1.3. Обмен значениями между переменными.....	24
<b>Глава 2. Операторы .....</b>	<b>27</b>
Логические операторы.....	27
Побитовые операторы.....	28
Операторы сравнения .....	29
Приведение типов .....	30
2.1. Использование тернарного оператора вместо блока if-else .....	31
2.2. Объединение потенциально нулевых значений .....	33
2.3. Сравнение одинаковых значений .....	35
2.4. Использование оператора spaceship для сортировки значений .....	37
2.5. Подавление сообщений об ошибках с помощью оператора @ .....	40
2.6. Сравнение битов внутри целых чисел .....	41
<b>Глава 3. Функции .....</b>	<b>45</b>
3.1. Доступ к параметрам функций.....	47
3.2. Установка значений параметров функции по умолчанию.....	49
3.3. Использование именованных параметров функций .....	51
3.4. Обеспечение типизации аргументов и возвращаемого значения функции.....	53

3.5. Определение функции с переменным числом аргументов .....	57
3.6. Возвращение нескольких значений .....	59
3.7. Доступ к глобальным переменным внутри функции.....	61
3.8. Управление состоянием внутри функции при многоократных вызовах .....	65
3.9. Определение динамических функций .....	68
3.10. Передача функций в качестве параметров другим функциям .....	69
3.11. Стрелочные функции .....	72
3.12. Создание функции без возвращаемого значения.....	75
3.13. Создание функции, которая не возвращается .....	77
 <b>Глава 4. Строки .....</b>	 78
4.1. Доступ к подстрокам в более крупной строке .....	81
4.2. Извлечение одной строки из другой .....	83
4.3. Замена части строки .....	84
4.4. Обработка строки по одному байту за раз.....	87
4.5. Генерация случайных строк .....	90
4.6. Интерполяция переменных в строке.....	91
4.7. Конкатенация нескольких строк вместе .....	93
4.8. Управление двоичными данными, хранящимися в строках .....	95
 <b>Глава 5. Числа.....</b>	 99
5.1. Проверка числа в переменной.....	100
5.2. Сравнение чисел с плавающей точкой .....	102
5.3. Округление чисел с плавающей точкой .....	104
5.4. Генерация случайных чисел.....	106
5.5. Генерация предсказуемых случайных чисел.....	108
5.6. Генерация взвешенных случайных чисел.....	110
5.7. Вычисление логарифмов .....	113
5.8. Вычисление экспоненты .....	114
5.9. Форматирование чисел как строк.....	115
5.10. Работа с очень большими или очень маленькими числами.....	116
5.11. Конвертация чисел из одной системы счисления в другую .....	119
 <b>Глава 6. Дата и время.....</b>	 121
Объектно-ориентированный подход .....	121
Часовые пояса .....	121

Временные метки Unix.....	122
6.1. Поиск текущей даты и времени .....	122
6.2. Форматирование дат и времени.....	124
6.3. Преобразование дат и времени во временные метки Unix .....	128
6.4. Преобразование временных меток Unix в составные части даты и времени.....	130
6.5. Вычисление разницы между двумя датами .....	131
6.6. Разбор дат и времени из произвольных строк .....	133
6.7. Проверка даты.....	136
6.8. Добавление к дате или вычитание из нее.....	137
6.9. Расчет времени в разных часовых поясах.....	140
 <b>Глава 7. Массивы .....</b>	 143
Типы массивов .....	143
Синтаксис.....	144
7.1. Объединение нескольких элементов по ключу в массиве .....	145
7.2. Инициализация массива с диапазоном чисел.....	148
7.3. Итерация элементов в массиве .....	149
7.4. Удаление элементов из ассоциативных и числовых массивов.....	152
7.5. Изменение размера массива .....	155
7.6. Добавление одного массива к другому .....	158
7.7. Создание массива из фрагмента существующего массива.....	161
7.8. Преобразование между массивами и строками.....	164
7.9. Реверсирование массива .....	167
7.10. Сортировка массива .....	168
7.11. Сортировка массива на основе функции.....	171
7.12. Случайный порядок элементов в массиве.....	173
7.13. Применение функции к каждому элементу массива .....	174
7.14. Сокращение массива до одного значения.....	177
7.15. Итерация по бесконечным или очень большим/ресурсозатратным массивам .....	179
 <b>Глава 8. Классы и объекты.....</b>	 182
Процедурное программирование .....	183
Объектно-ориентированное программирование.....	184

Мультипарадигмальные языки .....	186
Видимость .....	187
8.1. Инстанцирование объектов из пользовательских классов .....	190
8.2. Конструирование объектов для определения значений по умолчанию.....	192
8.3. Определение свойств, доступных только для чтения, в классе .....	194
8.4. Деконструкция объектов для очистки после того, как объект больше не нужен .....	197
8.5. Использование магических методов для предоставления динамических свойств .....	199
8.6. Расширение классов для определения дополнительной функциональности .....	202
8.7. Принуждение классов к определенному поведению .....	204
8.8. Создание абстрактных базовых классов .....	208
8.9. Предотвращение изменений в классах и методах .....	211
8.10. Клонирование объектов .....	216
8.11. Определение статических свойств и методов .....	220
8.12. Интроспекция закрытых свойств или методов внутри объекта.....	223
8.13. Повторное использование произвольного кода между классами .....	225
 <b>Глава 9. Безопасность и шифрование.....</b>	229
Унаследованное шифрование.....	230
Sodium .....	231
Случайности .....	236
9.1. Фильтрация, проверка и очистка пользовательского ввода.....	236
9.2. Защита конфиденциальных учетных данных от попадания в код приложения .....	242
9.3. Хеширование и валидация паролей .....	244
9.4. Шифрование и расшифровка данных.....	247
9.5. Хранение зашифрованных данных в файле .....	253
9.6. Криптографическая подпись сообщения для отправки другому приложению .....	258
9.7. Проверка криптографической подписи .....	260
 <b>Глава 10. Работа с файлами.....</b>	262
Windows или Unix .....	262
10.1. Создание или открытие локального файла .....	263

10.2. Чтение файла в строку .....	265
10.3. Чтение определенного фрагмента файла .....	267
10.4. Изменение файла .....	268
10.5. Одновременная запись в несколько файлов.....	269
10.6. Блокировка файла .....	272
<b>Глава 11. Потоки .....</b>	<b>274</b>
Обертки и протоколы .....	275
Фильтры .....	276
11.1. Потоковая передача данных во временный файл или из него .....	278
11.2. Чтение из потока ввода PHP .....	280
11.3. Запись в поток вывода PHP .....	284
11.4. Чтение из одного потока и запись в другой .....	286
11.5. Компоновка различных обработчиков потока .....	288
11.6. Создание пользовательской обертки потока.....	292
<b>Глава 12. Обработка ошибок .....</b>	<b>295</b>
12.1. Поиск и исправление ошибок синтаксиса .....	295
12.2. Создание и обработка пользовательских исключений .....	297
12.3. Скрытие сообщений об ошибках от конечных пользователей .....	300
12.4. Использование пользовательского обработчика ошибок.....	303
12.5. Регистрация ошибок во внешний поток .....	305
<b>Глава 13. Отладка и тестирование .....</b>	<b>307</b>
13.1. Использование расширения отладчика .....	309
13.2. Написание модульного теста .....	311
13.3. Автоматизация модульных тестов .....	316
13.4. Использование статического анализа кода .....	319
13.5. Запись отладочной информации .....	321
13.6. Выгрузка содержимого переменных в виде строк .....	326
13.7. Использование встроенного веб-сервера для быстрого запуска приложения .....	329
13.8. Использование модульных тестов для обнаружения регрессий в проекте, управляемом системой контроля версий с помощью git-bisect .....	331

---

<b>Глава 14.</b> Настройка производительности.....	336
JIT-компиляция .....	337
Кэширование опкодов.....	338
14.1. Измерение времени выполнения функций.....	339
14.2. Оценка производительности приложения.....	343
14.3. Ускорение работы приложения с помощью кэша опкодов .....	350
<b>Глава 15.</b> Пакеты и расширения .....	353
Стандартные модули.....	354
Библиотеки/Composer .....	355
15.1. Определение проекта Composer.....	355
15.2. Поиск пакетов Composer .....	358
15.3. Установка и обновление пакетов Composer .....	360
15.4. Установка нативных расширений PHP .....	363
<b>Глава 16.</b> Базы данных.....	366
Реляционные базы данных .....	366
База данных «ключ — значение» .....	367
Графовые базы данных .....	368
Документоориентированные базы данных .....	369
16.1. Подключение к базе данных SQLite.....	369
16.2. Использование PDO для подключения к внешнему провайдеру баз данных .....	372
16.3. Очистка пользовательского ввода для запроса к базе данных .....	376
16.4. Имитация данных для интеграционного тестирования .....	379
16.5. Запрос к базе данных SQL с помощью Eloquent ORM .....	384
<b>Глава 17.</b> Асинхронный PH5P .....	388
Библиотеки и среды выполнения.....	389
Асинхронные операции .....	390
17.1. Получение данных из удаленных API асинхронно .....	394
17.2. Ожидание результатов нескольких асинхронных операций.....	396
17.3. Прерывание одной операции для выполнения другой .....	398
17.4. Выполнение кода в отдельном потоке .....	401

17.5. Пересылка сообщений между отдельными потоками .....	406
17.6. Использование файбера для управления содержимым потока.....	411
<b>Глава 18. Командная строка PHP.....</b>	<b>415</b>
18.1. Разбор аргументов программы.....	416
18.2. Чтение интерактивного пользовательского ввода.....	420
18.3. Подсветка текста в консоли .....	421
18.4. Создание консольного приложения с помощью Symfony Console .....	422
18.5. Использование встроенного в PHP цикла REPL.....	427
<b>Об авторе .....</b>	<b>430</b>
<b>Иллюстрация на обложке .....</b>	<b>431</b>

Для Мии

Ева мотивирует меня продолжать писать стихи, даже когда  
я устал и нет вдохновения. Два и два равно четыре, но пять  
дадут вам десять, Уинстон

---

# Предисловие

Практически у каждого разработчика, создающего современные веб-приложения, сложилось свое мнение о PHP. Одни его обожают, другие недолюбливают, однако все они знакомы с его возможностями. Это связано с тем, что на PHP написано примерно 75 % всех веб-сайтов, которые были проанализированы. А учитывая размеры Интернета — это очень много PHP-кода<sup>1</sup>.

Конечно, не весь PHP-код можно назвать качественным. Любой, кто писал на PHP, сталкивался и с хорошим, и с плохим, и с отвратительным кодом. Это удивительно простой в работе язык, что объясняет его доминирующее положение на рынке. К сожалению, это также становится причиной ошибок во многих проектах, содержащих сомнительный код.

В отличие от компилируемых языков, которые обеспечивают строгую типизацию и управление памятью, PHP — интерпретируемый язык, который очень снисходителен к ошибкам программирования. Во многих случаях даже грубые ошибки приводят только к предупреждениям, в то время как PHP продолжит выполнять программу. Это очень удобно для новичков, так как невинная ошибка не обязательно приведет к краху приложения. Но такая снисходительность — это своего рода обоядоострый меч, поскольку даже «плохой код» будет выполняться, а многие разработчики публикуют этот код, который затем зачастую используют ничего не подозревающие новички.

Цель книги — защитить вас от повторного использования плохого кода, помогая понять, как избежать ошибок, допущенных другими разработчиками. Это издание также предлагает готовые шаблоны и примеры, которым может следовать любой программист для решения распространенных задач, связанных с PHP. Описанные в книге методы должны помочь вам быстро определять и решать сложные проблемы, не изобретая велосипед, и, что еще важнее, не поддаваясь искушению копировать и вставлять «плохой код», который вы можете найти в процессе исследований.

---

<sup>1</sup> По состоянию на март 2023 года, согласно статистике W3Techs (<https://oreil.ly/sb24e>), PHP используется на 77,5 % всех веб-сайтов.

## Для кого эта книга

Книга предназначена для всех, кто когда-либо создавал или поддерживал веб-приложение или веб-сайт на PHP. Эта книга не является исчерпывающим обзором всех возможностей языка. Ее предназначение — познакомить вас со специфическими концепциями разработки на PHP. Идеально, если вы уже немного работали с PHP, создали простое приложение или хотя бы пытались повторить один из многочисленных примеров «Hello, World!», которыми пестрит Интернет.

Если вы не знакомы с PHP, но неплохо владеете другим языком программирования, эта книга станет хорошим введением в новый для вас стек технологий. Здесь показано, как выполнять конкретные задачи на PHP. Пожалуйста, сравнивайте каждый блок примеров кода с тем, как бы вы решили ту же задачу на привычном вам языке; это поможет понять разницу между PHP и тем языком, на котором вы обычно работаете.

## Структура издания

Я не ожидаю, что кто-то прочитает эту книгу от корки до корки. Напротив, я надеюсь, что она станет справочником и ее будут использовать во время создания или проектирования новых приложений. Прочитаете ли вы целую главу за раз, чтобы освоить какую-то концепцию, или перейдете к одному или нескольким примерам кода для решения конкретной задачи, зависит только от вас.

Каждый рецепт — это самостоятельное и полностью готовое решение, которое вы можете применить в повседневной работе. Каждая последующая глава опирается на уже пройденный материал и завершается примером кода, который иллюстрирует рассмотренные в ней концепции.

В начале книги рассматриваются основные структурные элементы любого языка: переменные, операторы и функции. В главе 1 вводится понятие переменных и основы работы с данными. В главе 2 эта тема развивается: здесь рассказывается о различных операторах и операциях, поддерживаемых PHP. В главе 3 обе концепции объединяются в процессе определения высокоуровневого функционала и создания простой программы.

Следующие пять глав посвящены системе типизации PHP. Глава 4 охватывает все, в чем вы хотели разобраться, — и даже многое из того, о чём вы и не подозревали, что вам нужно знать, — о работе со строками в PHP. В главе 5 описывается целочисленная и десятичная арифметика с плавающей точкой и вводятся дополнительные строительные блоки, необходимые для создания сложных функций. В главе 6 рассказывается о работе PHP с датой и временем. В главе 7 представлены и объяснены все способы группировки данных в списки. Наконец, в главе 8 говорится о том, как разработчики могут расширять примитивные типы PHP, создавая собственные классы и объекты более высокого уровня.

После обсуждения этих базовых элементов в главе 9 рассматриваются функции шифрования и безопасности PHP, которые помогут создать по-настоящему надежные современные приложения. В главе 10 представлен функционал PHP по работе с файлами и манипулированию ими, а в главе 11 эти знания расширяются за счет рассмотрения более продвинутых интерфейсов потоковой передачи данных в PHP.

Следующие три главы посвящены критически важным концепциям веб-разработки. Глава 12 знакомит с интерфейсами обработки ошибок и исключений PHP. В главе 13 ошибки напрямую связываются с интерактивной отладкой и модульным тестированием. Наконец, в главе 14 показано, как правильно настроить PHP-приложение, чтобы достичь высокой производительности, масштабируемости и стабильности.

PHP — это язык с открытым исходным кодом, и до того, как стать полноценным языком веб-программирования, он представлял собой простой набор скриптов и пользовательских расширений. В главе 15 рассматриваются как встроенные расширения PHP (написанные на языке С и скомпилированные для работы вместе с самим языком), так и сторонние PHP-пакеты, которые могут улучшить функциональность вашего приложения. Далее в главе 16 предоставляется информация о базах данных и некоторых расширениях для работы с ними.

Глава 17 посвящена новой модели потоков, которая была представлена в PHP 8.1, а также асинхронному кодированию в целом. Наконец, в главе 18 я подведу итог знакомству с PHP, рассказав о возможностях командной строки и приложений, написанных для работы с командами как интерфейсом.

## Условные обозначения

Все примеры кода из книги были написаны для запуска как минимум на PHP 8.0.11, если не указано иное (для некоторых функций может потребоваться версия 8.2 или новее). Примеры кода были протестированы в контейнерной среде Linux, однако они должны одинаково хорошо работать на «голом железе» с Linux, Microsoft Windows или Apple macOS.

В книге используются следующие типографские обозначения.

### Курсив

Указывает на новые термины или слова, на которых нужно акцентировать внимание.

### Моноширинный шрифт

Используется в листингах кода, а также в тексте для обозначения таких элементов программы, как имена переменных или функций, базы данных, типы

данных, переменные окружения, операторы и ключевые слова. Им также выделены имена файлов и их расширения.



Этот элемент указывает на совет или предложение.



Этот элемент указывает на примечание.



Этот элемент указывает на предостережение.

## Благодарности

Прежде всего хочу выразить благодарность моей прекрасной жене за то, что она вдохновила меня на написание еще одной книги. Без твоей постоянной любви, поддержки и ободрения я, честно говоря, не был бы там, где нахожусь сейчас, ни в профессиональном, ни в личном плане.

Пока я был погружен в работу, не мог уделять достаточно внимания моим замечательным детям. Спасибо, что терпели мое «отсутствие». Я обязан вам всем!

Отдельная благодарность Крису Лингу, Михалу Шпачеку, Мэтью Турланду и Кендре Эш за их выдающиеся технические обзоры. Они помогли мне объективно осветить некоторые из наиболее важных моментов.

## От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу [comp@sprintbook.kz](mailto:comp@sprintbook.kz) (издательство «SprintBook», компьютерная редакция).

Мы будем рады узнать ваше мнение!

## ГЛАВА 1

---

# Переменные

Основой гибкого приложения является вариативность — способность программы служить нескольким целям в различных контекстах. Переменные — это общий инструмент для достижения такой гибкости в любом языке программирования. Такие именованные объекты ссылаются на конкретное значение, используемое программой. Это может быть число, обычная строка или даже более сложный объект со своими свойствами и методами. Суть в том, что переменные позволяют программе (и разработчику) ссылаться на эти значения и передавать их из одной части программы в другую.

Переменные не обязательно должны быть установлены по умолчанию — вполне допустимо определить пустую переменную, не присваивая ей никакого значения. Представьте стоящую на полке коробку, в которую можно положить подарок на Рождество. Вы можете легко найти эту коробку — переменную, но поскольку внутри ее ничего нет, мало что с ней можно сделать.

Предположим, что переменная называется `$giftbox`. Если бы вы попытались проверить значение этой переменной прямо сейчас, она оказалась бы пустой, так как еще не была задана. На самом деле `empty($giftbox)` вернет `true`, а `isset($giftbox)` вернет `false`. Переменная пуста и еще не определена.



Важно помнить, что любая переменная, которая не была явно определена (или задана), будет восприниматься PHP как `empty()`. Фактически определенная (или заданная) переменная может быть пустой или непустой в зависимости от ее значения, так как любое реальное значение, которое оценивается как `false`, будет считаться пустым.

В широком смысле языки программирования могут быть как строго (сильно), так и нестрого (слабо) типизированными. Строго типизированный язык требует от программиста явного указания типов всех переменных, параметров и возвращаемых функцией значений. Это обеспечивает соответствие каждого

значения ожидаемому типу. В нестрогого типизированных языках, таких как PHP, значения типизируются во время исполнения программы, то есть динамически. Например, разработчики могут хранить целое число (предположим, 42) в переменной, а затем использовать ее как строку в другом месте (скажем, "42"), и PHP автоматически преобразует эту переменную из целого числа в строку в нужный момент.

Главное преимущество слабой типизации заключается в простоте и удобстве разработки. Программисту не нужно заботиться о типах данных на этапе написания кода, поскольку эту задачу на себя берет интерпретатор. Основным же недостатком является то, что не всегда понятно, как будут обрабатываться определенные значения, когда интерпретатор переводит их из одного типа в другой. В табл. 1.1 показаны различные выражения, которые, начиная с PHP 8.0, оцениваются как «пустые» независимо от их базового типа.

**Таблица 1.1.** Пустые выражения в PHP

Выражение	empty(\$x)
\$x = ""	true
\$x = null	true
\$x = []	true
\$x = false	true
\$x = 0	true
\$x = "0"	true

Обратите внимание, что некоторые из этих выражений не являются по-настоящему пустыми, но PHP воспринимает их как таковые. Они считаются ложными, поскольку рассматриваются эквивалентными `false`, хотя они не идентичны `false`. Поэтому важно явно проверять ожидаемые значения типа `null`, `false` или `0` в приложении, а не полагаться на языковые конструкции вроде `empty()`, которые сделают это за вас. В таких случаях вы можете проверить, является ли переменная пустой, и явно сравнить ее с известным, фиксированным значением<sup>1</sup>.

Рецепты в этой главе посвящены основам определения и использования переменных в PHP, а также управления ими.

<sup>1</sup> Операторы сравнения рассматриваются в рецепте 2.3, где приводятся как пример, так и подробное обсуждение проверок на равенство.

## 1.1. Определение констант

### Задача

Вы хотите определить в своей программе конкретную переменную с фиксированным значением, которое не может быть изменено никаким другим кодом.

### Решение

В следующем блоке кода функция `define()` используется для явного определения значения константы с глобальной областью видимости, которое не может быть изменено другим кодом:

```
if (!defined('MY_CONSTANT')) {  
    define('MY_CONSTANT', 5);  
}
```

В качестве альтернативного подхода следующий блок кода использует директиву `const` внутри класса для определения константы, видимой только в пределах этого класса<sup>1</sup>:

```
class MyClass  
{  
    const MY_CONSTANT = 5;  
}
```

### Обсуждение

Если константа определена в приложении, функция `defined()` вернет `true` и сообщит, что вы можете обращаться к ней непосредственно в коде. Если константа еще не определена, интерпретатор PHP проанализирует, что вы делаете, и вместо этого преобразует ссылку на константу в строковый литерал.



Необязательно обозначать имена констант заглавными буквами. Тем не менее это правило, определенное в стандарте PHP Standard Recommendation 1 (PSR-1, [https://oreil.ly/\\_rNMe](https://oreil.ly/_rNMe)), опубликованном PHP Framework Interoperability Group (PHP-FIG, <https://oreil.ly/JHj-l>), рекомендуется соблюдать.

---

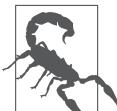
<sup>1</sup> Подробнее о классах и объектах читайте в главе 8.

Например, в следующей строке кода переменной `$x` присвоится значение `MY_CONSTANT` только в том случае, если константа определена. До версии PHP 8.0 неопределенная константа могла привести к тому, что `$x` содержала бы строку "`MY_CONSTANT`":

```
$x = MY_CONSTANT;
```

Если ожидаемое значение `MY_CONSTANT` не является строкой, то ответная реакция PHP на строковый литерал, вероятно, приведет к неожиданным побочным эффектам в вашем приложении. Интерпретатор не обязательно «упадет», но наличие "`MY_CONSTANT`" там, где ожидается целое число, вызовет проблемы. Начиная с PHP 8.0, обращение к еще неопределенному константе приводит к фатальной ошибке.

Наш пример демонстрирует два способа определения констант: `define()` и `const`. Первый создаст глобальную константу, которая будет доступна в любом месте вашего приложения, используя только имя самой константы. Второй привязывает константу к классу, в котором она определяется. В данном случае вместо ссылки на `MY_CONSTANT`, как в первом решении, константа будет упоминаться как `MyClass::MY_CONSTANT`.



PHP определяет несколько констант по умолчанию (<https://oreil.ly/zQ40o>), которые не могут быть перезаписаны пользовательским кодом. Константы в целом фиксированы и не могут быть модифицированы или заменены, поэтому всегда проверяйте, не определена ли константа, прежде чем пытаться ее задать. Попытки переопределить константу приведут к появлению предупреждения. Дополнительные сведения о работе с ошибками и уведомлениями см. в главе 12.

Константы класса по умолчанию общедоступны, то есть любой код в приложении, который обращается к `MyClass`, может ссылаться и на его общедоступные константы. Однако, начиная с версии PHP 7.1.0, можно применить модификатор видимости к константе класса и сделать ее закрытой для экземпляров класса.

## Читайте также

Документация по константам в PHP (<https://oreil.ly/9WBhy>), `defined()` (<https://oreil.ly/jm1au>), `define()` (<https://oreil.ly/9iON9>) и константам классов (<https://oreil.ly/ggaCv>).

## 1.2. Создание переменных переменных

### Задача

Вы хотите динамически ссылаться на определенную переменную, не зная заранее, какая из нескольких связанных переменных понадобится программе.

### Решение

В синтаксисе PHP имя переменной, на которую вы хотите сослаться, всегда предваряется символом \$. Вы также можете сделать имя переменной само по себе переменной. Следующая программа выведет #f00, используя переменную переменной:

```
$red = '#f00';
$c = 'red';

echo $$color;
```

### Обсуждение

Когда PHP интерпретирует ваш код, он воспринимает символ \$ в начале строки как идентификатор переменной, а следующий за ним текст — как имя этой переменной. В примере выше текст после \$ является переменной. PHP будет оценивать переменные переменных справа налево, передавая результат одного вычисления в качестве имени, используемого для следующего вычисления, прежде чем выводить какие-либо данные на экран.

Говоря иначе, пример 1.1 показывает две функционально равнозначные строки кода, за исключением того, что во второй используются фигурные скобки, чтобы явно определить код, который считывается первым.

#### Пример 1.1. Определение переменных переменных

```
$$color;
${$color};
```

Крайняя правая переменная \$color сначала расценивается как литерал "red", это, в свою очередь, означает, что \$\$color и \$red в конечном счете ссылаются на одно и то же значение. Введение фигурных скобок в качестве явных разделителей означает еще более сложное применение.

В примере 1.2 предполагается, что приложение хочет провести A/B-тестирование заголовка для SEO-оптимизации. Маркетинговая команда предоставила два варианта, и разработчики хотят показывать разные заголовки разным посетителям,

но при этом выводить тот же заголовок, когда пользователь возвращается на сайт. Вы можете сделать это, используя IP-адрес посетителя и создав переменную, которая выбирает заголовок на основе IP-адреса.

### Пример 1.2. А/Б-тестирование заголовков

```
$headline0 = 'Десять советов по написанию отличных заголовков';
$headline1 = 'Пошаговый подход к написанию ярких заголовков';

echo ${'headline' . (crc32($_SERVER['REMOTE_ADDR']) % 2)};
```

Функция `crc32()` из предыдущего примера — это удобная утилита, которая вычисляет 32-битную контрольную сумму заданной строки, детерминированно преобразующую строку в целое число. Оператор `%` выполняет операцию деления по модулю над итоговым целым числом, возвращая 0, если контрольная сумма четная, и 1, если нечетная. Затем результат конкатенируется с заголовком строки `headline` в вашей динамической переменной, чтобы функция могла выбрать тот или иной заголовок.



Массив `$_SERVER` — это определяемая системой суперглобальная переменная (<https://oreil.ly/DtQV>), содержащая полезную информацию о сервере, на котором запущен ваш код, и о входящем запросе, который запустил PHP в первую очередь. Точное содержимое этого массива будет отличаться от сервера к серверу, особенно в зависимости от того, использовали ли вы NGINX или Apache HTTP Server (или другой веб-сервер) с PHP, но обычно он содержит такую полезную информацию, как заголовки запроса, пути запроса и имя файла текущего выполняющегося скрипта.

В примере 1.3 построчно представлено применение функции `crc32()`, чтобы проиллюстрировать, как связанное с пользователем значение, например IP-адрес, может быть задействовано для детерминированной идентификации заголовка, служащего в целях SEO.

### Пример 1.3. Проверка контрольных сумм по IP-адресам посетителей

```
$_SERVER['REMOTE_ADDR'] = '127.0.0.1'; ①
crc32('127.0.0.1') = 3619153832; ②
3619153832 % 2 = 0; ③
'headline' . 0 = 'headline0' ④
${'headline0'} = 'Десять советов по написанию отличных заголовков'; ⑤
```

① IP-адрес извлекается из суперглобальной переменной `$_SERVER`. Обратите внимание, что ключ `REMOTE_ADDR` будет присутствовать только при использовании PHP с веб-сервера, а не через CLI.

② Функция `crc32()` преобразует строку с IP-адресом в целочисленную контрольную сумму.

❸ Оператор деления по модулю (%) определяет, является ли контрольная сумма четной или нечетной.

❹ Результат деления добавляется к заголовку `headline`.

❺ Полученная строка `headline0` используется в качестве переменной переменной для определения правильного значения SEO-заголовка.

Можно также вложить переменные переменных более чем на два уровня вглубь. Использование трех символов \$ — как в случае с `$$name` — тоже допустимо, как и `$$some_function()`. Однако для лучшей читаемости кода и его поддержки рекомендуется ограничивать уровни вариативности в именах переменных. В любом случае рассматриваемый инструмент используется довольно редко, но стоит иметь в виду, что многочисленные уровни косвенного обращения к переменным сделают ваш код непонятным и трудным для тестирования и сопровождения, если вдруг что-то сломается.

## Читайте также

Документация по переменным переменных (<https://oreil.ly/wNBh0>).

## 1.3. Обмен значениями между переменными

### Задача

Вы хотите поменять местами значения, хранящиеся в двух переменных, не вводя при этом дополнительных переменных.

### Решение

В следующей строчке кода используется языковая конструкция `list()`, которая служит для повторного присвоения значений переменным:

```
list($blue, $green) = array($green, $blue);
```

Более лаконичный вариант предыдущего решения — использовать для списка и массива короткий синтаксис, который стал доступен в PHP 7.1:

```
[$blue, $green] = [$green, $blue];
```

## Обсуждение

Ключевое слово `list` в PHP не является функцией, хотя и выглядит таковой. Это языковая конструкция, которая позволяет присваивать списку переменных значения за одну операцию. С ее помощью разработчики могут задавать значения сразу нескольким переменным из другой коллекции значений (например, из массива). Она также разрешает разбивать массивы на отдельные переменные.

В современном PHP квадратные скобки [ ] применяют для лаконичного обозначения массивов: запись [1, 4, 5] эквивалентна `array(1, 4, 5)`.



И `list`, и `array` в PHP относятся к языковым конструкциям. Они встроены в язык и являются ключевыми словами, которые служат для управления системой. Некоторые ключевые слова, например `if`, `else` или `echo`, легко отличить от пользовательского кода. В то же время такие языковые конструкции, как `list`, `array` и `exit`, выглядят как функции, но, как и ключевые слова, они встроены в язык и ведут себя несколько иначе, чем обычные функции. Список всех зарезервированных ключевых слов (<https://oreil.ly/OJD13>) можно найти в руководстве по PHP (<https://oreil.ly/OJD13>). Оно также содержит информацию о том, как использовать каждую из языковых конструкций.

Начиная с версии PHP 7.1, разработчики могут применять тот же лаконичный синтаксис для `list()`, создавая более читабельный код. При присвоении значений из массива массиву переменных использование подобного синтаксиса по обе стороны оператора присваивания (=) имеет смысл и уточняет ваше намерение.

Наш пример явно меняет местами значения, хранящиеся в переменных `$green` и `$blue`. Это то, что может сделать инженер при развертывании приложения для перехода с одной версии API на другую. При скользящем развертывании текущая активная среда часто называется «зеленым» развертыванием, а новая, потенциальная замена — «синим», указывая балансировщикам нагрузки и другим зависимым приложениям переключиться с «зеленой» на «синюю» среду и проверить подключение и функциональность, прежде чем подтвердить, что развертывание идет normally.

В примере 1.4 рассмотрим ситуацию, когда приложение использует API с префиксом даты развертывания. Приложение отслеживает, какую версию API оно использует (`$green`), и пытается переключиться на новую среду, чтобы проверить возможность подключения. Если проверка подключения не удается, приложение автоматически переключается обратно на старое окружение.

**Пример 1.4.** Переключение синей/зеленой среды

```
$green = 'https://2021_11.api.application.example/v1';
$blue = 'https://2021_12.api.application.example/v1';

[$green, $blue] = [$blue, $green];

if (connection_fails(check_api($green))) {
    [$green, $blue] = [$blue, $green];
}
```

Конструкцию `list()` также допускается применять для извлечения определенных значений из произвольной группы элементов. Пример 1.5 иллюстрирует, как адрес, хранящийся в виде массива, может использоваться в различных контекстах для извлечения только нужных значений.

**Пример 1.5.** Использование функции `list()` для извлечения элементов массива

```
$address = ['123 S Main St.', 'Anywhere', 'NY', '10001', 'USA'];

// Извлечение каждого элемента в виде именованных переменных
[$street, $city, $state, $zip, $country] = $address;

// Извлечение и именование только штата
[,$state,,] = $address;

// Извлечение только страны
[,,,,$country] = $address;
```

Каждое извлечение в предыдущем примере является независимым и устанавливает только те переменные, которые необходимы<sup>1</sup>. Для более сложных приложений, обрабатывающих значительные объемы данных, установка ненужных переменных может привести к проблемам с производительностью. Хотя `list()` является мощным инструментом для разбивки коллекций вроде массивов, он подходит только для простых случаев, подобных тем, что рассмотрены в предыдущих примерах.

## Читайте также

Документация по `list()` (<https://oreil.ly/bzO7i>), `array()` ([https://oreil.ly/tq1Z\\_](https://oreil.ly/tq1Z_)) и PHP RFC по лаконичному синтаксису `list()` (<https://oreil.ly/Ou98z>).

---

<sup>1</sup> Вспомните, как во введении к этой главе объяснялось, что ссылки на переменные, которые не заданы явно, оцениваются как «пустые». Это означает, что вы можете задавать те значения и переменные, которые вам нужны.

## ГЛАВА 2

# Операторы

В главе 1 мы познакомились с основными строительными блоками PHP — переменными для хранения произвольной информации. Однако сами по себе эти блоки мало что могут. Они бесполезны без своеобразного «клея», удерживающего их вместе. Таким kleem является набор операторов PHP (<https://oreil.ly/Vepfg>). Операторы — это команды, которые сообщают PHP, что делать со значениями: например, преобразовать одни значения в другие.

Чаще всего оператор в PHP представлен одним символом или повторением этого же символа. В некоторых случаях операторы также могут быть целым словом, что помогает определить, что именно пытается выполнить оператор.

В книге мы не пытаемся описать все операторы, используемые в PHP; за исчерпывающими объяснениями каждого из них обращайтесь к руководству по PHP (<https://oreil.ly/YGWyE>). Вместо этого в следующих нескольких разделах мы рассмотрим некоторые из наиболее важных логических, побитовых операторов и операторов сравнения, а затем перейдем к более конкретным задачам, решениям и примерам.

## Логические операторы

Логические операции — это операции PHP, которые создают таблицы истинности и определяют основные критерии группировки и/или/не. В табл. 2.1 перечислены все символьные логические операторы, поддерживаемые PHP.

**Таблица 2.1.** Логические операторы

Выражение	Имя оператора	Результат	Пример
<code>\$x &amp;&amp; \$y</code>	И	true, если и \$x, и \$y имеет значение true	<code>true &amp;&amp; true == true</code>
<code>\$x    \$y</code>	ИЛИ	true, если либо \$x, либо \$y имеет значение true	<code>true    false == true</code>
<code>!\$x</code>	НЕ	true, если \$x имеет значение false (и наоборот)	<code>!true == false</code>

Логические операторы `&&` и `||` имеют аналоги в английском языке: `and` и `or` соответственно. Утверждение `($x and $y)` функционально эквивалентно `($x && $y)`, а `($x or $y) — ($x || $y)`.

В PHP также может использоваться слово `xor` — специальный исключающий оператор или, который возвращает `true`, если одно из двух значений в выражении `true`, но не оба сразу. К сожалению, логическая операция XOR не имеет символьного эквивалента.

## Побитовые операторы

В PHP поддерживаются операции над отдельными битами в целых числах, благодаря чему PHP способен работать не только с веб-приложениями, но и с двоичными файлами и структурами данных! Эти операторы схожи с операторами `and`, `or` и `xor`.

В то время как логические операторы возвращают `true` или `false` на основе сравнения двух целых значений, рассматриваемые в этом разделе операторы фактически выполняют побитовую арифметику над целыми числами и возвращают результат этого вычисления. Если хотите рассмотреть пример того, как это может быть полезно, обратитесь к рецепту 2.6.

В табл. 2.2 показаны различные побитовые операторы в PHP, их назначение и простой пример их работы.

**Таблица 2.2.** Побитовые операторы

Выражение	Имя оператора	Результат	Пример
<code>\$x &amp; \$y</code>	И	Возвращает биты, установленные как в <code>\$x</code> , так и в <code>\$y</code>	<code>5 &amp; 1 == 1</code>
<code>\$x   \$y</code>	ИЛИ	Возвращает биты, установленные либо в <code>\$x</code> , либо в <code>\$y</code>	<code>4   1 == 5</code>
<code>\$x ^ \$y</code>	Исключающее ИЛИ (XOR)	Возвращает биты, установленные только в <code>\$x</code> или <code>\$y</code>	<code>5 ^ 3 == 6</code>
<code>~ \$x</code>	НЕ	Инвертирует биты, установленные в <code>\$x</code>	<code>~ 4 == -5</code>
<code>\$x &lt;&lt; \$y</code>	Сдвиг влево	Сдвиг битов <code>\$x</code> влево на <code>\$y</code> шагов	<code>4 &lt;&lt; 2 == 16</code>
<code>\$x &gt;&gt; \$y</code>	Сдвиг вправо	Сдвиг битов <code>\$x</code> вправо на <code>\$y</code> шагов	<code>4 &gt;&gt; 2 == 1</code>

В PHP максимальное целое число зависит от разрядности процессора, на котором работает приложение. В любом случае с помощью константы `PHP_INT_MAX` вы узнаете, насколько большими могут быть целые числа: на 32-битных машинах это 2 147 483 647,

а на 64-битных — 9 223 372 036 854 775 807. В обоих случаях это число представляется в двоичном виде как строка из единиц, длина которой на единицу меньше разрядности системы. На 32-битной машине число 2 147 483 647 представлено в виде строки из 31 единицы. Старший бит (по умолчанию 0) используется для определения знака целого числа: если он равен 0, целое число положительное, иначе — отрицательное.

На любой машине число 4 в двоичном виде выглядит как 100, с достаточным количеством нулей слева от старшего разряда, чтобы заполнить разрядность процессора. В 32-битной системе это 29 нулей. Чтобы сделать целое число отрицательным, нужно представить его как единицу, за которой следуют 28 нулей, а затем 100.

Для простоты рассмотрим 16-битную систему. Целое число 4 будет представлено как 0000000000000100, а отрицательное — как 1000000000000100. Если применить побитовый оператор `not (~)` к положительному числу 4 в 16-битной системе, все нули превратятся в единицы, и наоборот. В результате вы получите 111111111111011, что в 16-битной системе равно -5.

## Операторы сравнения

Основа любого языка программирования — это уровень контроля, который он предоставляет для ветвления в зависимости от конкретных условий. В PHP значительная часть этой логики осуществляется путем сравнения двух или более значений друг с другом. Именно набор операторов сравнения (<https://oreil.ly/QuPhV>) обеспечивает большую часть расширенной функциональности, используемой для создания сложных приложений.

В табл. 2.3 перечислены операторы сравнения скалярных величин, которые считаются самыми важными для понимания PHP. Остальные операторы («больше», «меньше» и др.) довольно стандартны для языков программирования и не рассматриваются в этой главе.

**Таблица 2.3.** Операторы сравнения

Выражение	Операция	Результат
<code>\$x == \$y</code>	Равно	Возвращает <code>true</code> , если оба значения равны после приведения к одному типу
<code>\$x === \$y</code>	Тождественно равно	Возвращает <code>true</code> , если оба значения равны и имеют один тип
<code>\$x &lt;= \$y</code>	Spaceship	Возвращает 0, если оба значения равны, 1, если <code>\$x</code> больше, или -1, если <code>\$y</code> больше

При работе с объектами операторы равенства и тождества выполняются несколько по-разному. Два объекта считаются равными (`==`), если они имеют одинаковую внутреннюю структуру (атрибуты и значения) и относятся к одному типу (классу). Объекты считаются идентичными (`==`) тогда и только тогда, когда они являются ссылками на один и тот же экземпляр класса. Здесь более строгие требования, чем при сравнении скалярных величин.

## Приведение типов

Хотя само по себе имя типа формально не является оператором, вы можете использовать его для явного приведения значения к необходимому типу. Просто напишите имя типа в круглых скобках перед значением, чтобы принудительно преобразовать его. В примере 2.1 простое целочисленное значение сначала конвертируется в различные типы перед использованием.

### Пример 2.1. Приведение значений к другим типам

```
$value = 1;

$bool = (bool) $value;
$float = (float) $value;
$string = (string) $value;

var_dump([$bool, $float, $string]);

// array(3) {
//   [0]=>
//   bool(true)
//   [1]=>
//   float(1)
//   [2]=>
//   string(1) "1"
// }
```

PHP поддерживает следующие виды приведения типов:

- (`int`) — приведение к `int`;
- (`bool`) — приведение к `bool`;
- (`float`) — приведение к `float`;
- (`string`) — приведение к `string`;
- (`array`) — приведение к `array`;
- (`object`) — приведение к `object`.

Можно также использовать (`integer`) как псевдоним (`int`), (`boolean`) как (`bool`), (`real`) или (`double`) как (`float`), а (`binary`) как (`string`). Эти псевдонимы выполняют те же приведения типов, что и в списке выше, но поскольку они не задействуют имя типа, к которому выполняется приведение, такой подход не рекомендуется.

В этой главе мы рассмотрим способы использования важнейших сравнительных и логических операторов PHP.

## 2.1. Использование тернарного оператора вместо блока if-else

### Задача

Вам нужно создать условие ветвления «или — или», чтобы присвоить переменной определенное значение в одной строке кода.

### Решение

Тернарный оператор (`a ? b : c`) позволяет вложить условие «или — или» и оба возможных значения ветвления в одно утверждение. В следующем примере показано, как определить переменную со значением из суперглобальной переменной `$_GET` и вернуться к значению по умолчанию, если она пуста:

```
$username = isset($_GET['username']) ? $_GET['username'] : 'default';
```

### Обсуждение

Тернарное выражение имеет три аргумента и оценивается слева направо, проверяя истинность крайнего левого утверждения (оценивается ли оно как `true` независимо от типов, участвующих в выражении) и возвращая следующее значение, если `true`, или последнее, если `false`. Вы можете визуализировать это выражение следующим образом:

```
$_значение_ = (_выражение для оценки_) ? (если истинно) : (если ложно);
```

Тернарный шаблон — это простой способ вернуть значение по умолчанию при проверке системных значений или даже параметров из веб-запроса (тех, что хранятся в суперглобальных переменных `$_GET` или `$_POST`). Это также мощный способ переключения логики в шаблонах страниц на основе возврата определенного вызова функции.

Следующий пример предполагает веб-приложение, которое приветствует вошедших в систему пользователей по имени (проверяя состояние их аутентификации вызовом функции `is_logged_in()`) или здоровается с гостем, если пользователь еще не аутентифицирован. Поскольку этот пример закодирован непосредственно в HTML-разметке веб-страницы, использование длинных операторов `if/else` было бы неуместным:

```
<h1>Добро пожаловать, <?php echo is_logged_in() ? $_SESSION['user'] : 'Гость';  
?>!</h1>
```

Тернарные операции также можно упростить, если проверяемое значение одновременно является истинным (оценивается как `true` при приведении к булеву значению) и значением, которое вы хотите получить по умолчанию. Приведенный в подразделе «Решение» код проверяет, установлено ли имя пользователя, и присваивает это значение заданной переменной, если это так. Поскольку непустые строки оцениваются как `true`, решение можно сократить до следующего:

```
$username = $_GET['username'] ?: 'default';
```

Когда тернарный оператор сокращается от формата `a ? b : c` до простого `a ?: c`, PHP оценивает выражение для проверки `a`, как будто оно булево. Если оно истинно, PHP просто вернет само выражение, если ложно — резервное значение `c`.



PHP сравнивает истинность так же, как и «пустоту», о чем говорилось в главе 1. Строки, которые заданы (не пустые и не `null`), целые числа, не равные нулю, и массивы, содержащие хотя бы один элемент, обычно считаются истинными, то есть они оцениваются как `true`, если их привести к булеву значению. Подробнее о способах взаимосвязи типов и их эквивалентности вы можете прочитать в разделе руководства PHP о сравнении типов (<https://oreil.ly/nXsr8>).

Тернарный оператор — это расширенный вариант оператора сравнения, которым, несмотря на лаконичный код, иногда злоупотребляют, создавая слишком сложную логику. Рассмотрим пример 2.2, в котором одна тернарная операция вложена в другую.

### Пример 2.2. Вложенное тернарное выражение

```
$val = isset($_GET['username']) ? $_GET['username'] : (isset($_GET['userid'])  
? $_GET['user_id'] : null);
```

Этот пример следует переписать в виде простого оператора `if/else`, чтобы обеспечить большую ясность относительно того, как код разветвляется. Ничего функционально неверного в коде нет, но вложенные тернарные операторы могут быть

сложны для восприятия или анализа и часто приводят к логическим ошибкам в дальнейшем. Предыдущий тернарный оператор можно переписать так, как показано в примере 2.3.

#### Пример 2.3. Несколько операторов if/else

```
if (isset($_GET['username'])) {  
    $val = $_GET['username'];  
} elseif (isset($_GET['userid'])) {  
    $val = $_GET['userid'];  
} else {  
    $val = null;  
}
```

Хотя пример 2.3 более развернут, чем пример 2.2, вы легко можете отследить, где логика должна разветвляться. Код также более удобен для сопровождения, так как при необходимости можно добавить новую логическую ветвь, которая, в свою очередь, при добавлении в пример 2.2 еще больше усложнила бы и без того непростой тернарный оператор и сделала бы программу более запутанной для сопровождения в долгосрочной перспективе.

### Читайте также

Документация по тернарному оператору (<https://oreil.ly/Y5WCn>) и его вариациям.

## 2.2. Объединение потенциально нулевых значений

### Задача

Вы хотите присвоить переменной определенное значение только, если она установлена и не является `null`, в противном случае использовать статическое значение по умолчанию.

### Решение

При применении оператора объединения с `null` (`??`), как показано ниже, первое значение будет использоваться только в том случае, если оно задано и не является `null`:

```
$username = $_GET['username'] ?? 'not logged in';
```

## Обсуждение

Оператор объединения с `null` в PHP (null-coalescing) — новая функция, появившаяся с выходом PHP 7.0. Это так называемый синтаксический сахар, заменяющий сокращенную версию тернарного оператора `?:`, о котором говорилось в рецепте 2.1.



Синтаксический сахар — это краткая запись для выполнения обычных, но многократно повторяющихся операций в коде. Разработчики языков вводят подобные функции, чтобы сэкономить время и представить рутинные, часто повторяющиеся блоки кода с помощью более простого и лаконичного синтаксиса.

Обе следующие строки кода функционально эквивалентны, но при использовании тернарного оператора компилятор выдаст предупреждение, если оцениваемое выражение не определено, прежде чем вернет значение по умолчанию:

```
$a = $b ?: $c;  
$a = $b ?? $c;
```

Хотя эти два примера функционально идентичны, заметное различие в их поведении возникает, если оцениваемое значение (`$b`) не определено. Однако при использовании оператора объединения с `null` все работает великолепно.

В случае с дискретными переменными различие в функциональности этих операторов не совсем очевидно, но когда оцениваемый компонент представляет собой, скажем, индексированный массив, потенциальное влияние становится более явным. Предположим, что вместо дискретной переменной вы пытаетесь извлечь элемент из суперглобальной переменной `$_GET`, которая содержит параметры запроса. В следующем примере оба оператора — тернарный и объединения с `null` — вернут значение по умолчанию (резервное), но тернарная версия сообщит о неопределенном индексе:

```
$username = $_GET['username'] ?? 'anonymous';  
$username = $_GET['username'] ?: 'anonymous'; // Внимание: неопределенный индекс  
...
```

Если ошибки и уведомления подавляются во время выполнения<sup>1</sup>, то функциональной разницы между операторами не будет. Тем не менее лучше избегать написания кода,зывающего ошибки или предупреждения, поскольку они могут случайно срабатывать у конечного пользователя или засорять системные журналы и затруднять поиск реальных проблем в вашем коде. Хотя краткий тернарный оператор очень полезен, оператор объединения с `null` специально создан для такого рода операций и почти всегда должен использоваться вместо него.

---

<sup>1</sup> Обработка и устранение ошибок, предупреждений и уведомлений подробно рассматриваются в главе 12.

## Читайте также

Анонс нового оператора, когда он только был добавлен в PHP 7.0 ([https://oreil.ly/6vmP\\_](https://oreil.ly/6vmP_)).

## 2.3. Сравнение одинаковых значений

### Задача

Вы хотите сравнить два значения одного типа, чтобы убедиться, что они идентичны.

### Решение

Используйте три знака равенства для сравнения значений без динамического приведения их типов:

```
if ($a === $b) {  
    // ...  
}
```

### Обсуждение

В PHP знак равенства выполняет три функции: одиночный (`=`) присваивает значения переменной, двойной (`==`) определяет, равны ли значения по обе стороны от оператора, и, наконец, тройной (`===`) устанавливает, являются ли значения тождественно равными.

В табл. 2.4 показано, какие значения считаются равными, потому что PHP приводит один тип к другому при оценке выражения.

Таблица 2.4. Равенство значений в PHP

Выражение	Результат	Объяснение
<code>0 == "a"</code>	<code>false</code>	(Только для PHP 8.0 и выше) Стока "a" приводится к целому числу, что означает приведение к 0
<code>"1" == "01"</code>	<code>true</code>	Обе стороны выражения приводятся к целым числам, и <code>1 == 1</code>
<code>100 == "1e2"</code>	<code>true</code>	Правая часть выражения оценивается как экспоненциальное представление 100 и приводится к целому числу



Первый пример в табл. 2.4 оценивается как true в версиях PHP ниже 8.0. В более ранних версиях при сравнении равенства строки (или числовой строки) с числом сначала строка преобразовывалась в число (в нашем случае «а» в 0). В PHP 8.0 это поведение изменилось, и к числам приводятся только числовые строки, поэтому результат первого выражения теперь равен false.

Способность PHP динамически преобразовывать типы во время выполнения представляет интерес, но иногда это совсем не то, что вам нужно. Некоторые методы возвращают false для обозначения ошибки или неудачи, в то время как целое число 0 может быть допустимым результатом функции. Рассмотрим функцию из примера 2.4, возвращающую в виде числа количество книг определенной категории или false, если соединение с базой данных (БД), содержащей эти сведения, не удалось установить.

#### **Пример 2.4.** Подсчет элементов в БД или возврат false

```
function count_books_of_type($category)
{
    $sql = "SELECT COUNT(*) FROM books WHERE category = :category";

    try {
        $dbh = new PDO(DB_CONNECTION_STRING, DB_LOGIN, DB_PASS);
        $statement = $dbh->prepare($sql);

        $statement->execute(array(':category' => $category));
        return $statement->fetchColumn();
    } catch (PDOException $e) {
        return false;
    }
}
```

Если в примере 2.4 все сработает как ожидается, то код вернет в виде целого числа количество книг в определенной категории. В примере 2.5 эта функция применяется для вывода заголовка на веб-странице.

#### **Пример 2.5.** Использование результатов функции, связанной с БД

```
$books_found = count_books_of_type('fiction');

switch ($books_found) {
    case 0:
        echo 'Не найдено ни одной художественной книги';
        break;
    case 1:
        echo 'Найдена одна художественная книга';
        break;
    default:
        echo 'Найдены ' . $books_found . ' художественные книги';
}
```

Внутри оператора `switch` используется нестрогое сравнение типов (знакомый нам оператор `==`). Если `count_books_of_type()` возвращает `false` вместо фактического результата, оператор `switch` взамен уведомления об ошибке выведет сообщение о том, что не было найдено ни одной художественной книги. В данном случае такое поведение считается приемлемым, но, если вашему приложению необходимо отразить существенную разницу между `false` и `0`, нестрогое сравнение не подойдет.

Вместо этого PHP позволяет применять тройной знак равенства (`==`), чтобы проверить, являются ли оба оцениваемых значения тождественными, то есть идентичными по значению и типу. Даже если целое число `5` и строка `"5"` имеют одинаковое значение, оценка `5 == "5"` приведет к `false`, потому что эти два значения разного типа. Таким образом, пока `0 == false` оценивается как `true`, `0 == false` всегда будет `false`.



При работе с объектами, определенными в пользовательских классах или предоставленных PHP, сложнее установить, являются ли два значения тождественно равными. Например, объекты `$obj1` и `$obj2` будут считаться идентичными только в том случае, если они действительно являются одним и тем же экземпляром класса. Подробнее об инстанцировании объектов и классов см. в главе 8.

## Читайте также

Документация PHP по операторам сравнения (<https://oreil.ly/T6GXm>).

# 2.4. Использование оператора spaceship для сортировки значений

## Задача

Вам требуется создать пользовательскую функцию сортировки для упорядочивания произвольного списка объектов с помощью встроенной в PHP функции `usort()` (<https://oreil.ly/xGbc9>).

## Решение

Предположим, вы хотите отсортировать список объектов по нескольким свойствам. В этом случае используйте оператор `spaceship`, или трехстороннего сравнения (`<=`), для определения пользовательской функции сортировки, а затем передайте ее в качестве обратного вызова функции `usort()`.

Рассмотрим следующее определение класса Person, позволяющего создавать записи, содержащие только имя и фамилию:

```
class Person {
    public $firstName;
    public $lastName;

    public function construct($first, $last)
    {
        $this->firstName = $first;
        $this->lastName = $last;
    }
};
```

Затем с помощью этого класса вы можете сформировать перечень людей, например президентов США, добавляя каждого человека в список по очереди, как в примере 2.6.

**Пример 2.6.** Добавление нескольких экземпляров объектов в список

```
$presidents = [];

$presidents[] = new Person('George', 'Washington');
$presidents[] = new Person('John', 'Adams');
$presidents[] = new Person('Thomas', 'Jefferson');
// ...
$presidents[] = new Person('Barack', 'Obama');
$presidents[] = new Person('Donald', 'Trump');
$presidents[] = new Person('Joseph', 'Biden');
```

Оператор spaceship допускается применять для определения способа сортировки этих данных, предполагая, что вы хотите упорядочить их сначала по фамилии, затем по имени, как показано в примере 2.7.

**Пример 2.7.** Сортировка списка президентов с помощью оператора spaceship

```
function presidential_sorter($left, $right)
{
    return [$left->lastName, $left->firstName]
        <=>
        [$right->lastName, $right->firstName];
}

usort($presidents, 'presidential_sorter');
```

В результате предыдущего вызова usort() массив \$presidents будет отсортирован и готов к использованию.

## Обсуждение

Оператор spaceship — это специальное дополнение в PHP 7.0, которое помогает определить связь между значениями по обе стороны от него:

- если первое значение меньше второго, возвращается -1;
- если первое значение больше второго, возвращается +1;
- если оба значения одинаковы, возвращается 0.



Как и оператор равенства в PHP, оператор spaceship предпринимает попытку привести типы каждого значения в сравниваемом выражении к одному и тому же. Можно использовать число для одного значения и строку для другого и получить корректный результат. Рекомендуется быть особенно внимательными при работе с подобными специальными операторами.

Оператор spaceship позволяет легко упорядочить небольшой массив или список примитивных значений (например, дат, символов, целых чисел или чисел с плавающей точкой). В нашем случае, если использовать `usort()`, потребуется функция сортировки, подобная следующей:

```
function sorter($a, $b) {  
    return ($a < $b) ? -1 : (($a > $b) ? 1 : 0);  
}
```

Оператор spaceship упрощает вложенное тернарное выражение из предыдущего блока кода, заменяя оператор `return` на `return $a <= $b`, оставляя возможности функции сортировки нетронутыми.

Более сложные примеры, наподобие того, что приведен в «Решении», потребуют довольно объемных определений функций сортировки. Оператор spaceship упрощает алгоритм сравнения, позволяя разработчикам указывать любую логику в одной легкочитаемой строке.

## Читайте также

Оригинальный RFC для оператора трехстороннего сравнения в PHP (<https://oreil.ly/O1X8R>).

## 2.5. Подавление сообщений об ошибках с помощью оператора @

### Задача

Вы хотите явно игнорировать или подавлять ошибки, вызванные определенным выражением в вашем приложении.

### Решение

Добавьте символ @ перед выражением, чтобы временно установить уровень сообщения об ошибках на 0 для данной строки кода. Это может помочь подавить ошибки, связанные с отсутствием файлов, при попытке открыть их напрямую, как в следующем примере:

```
$fp = @fopen('file_that_does_not_exist.txt', 'r');
```

### Обсуждение

Код из примера выше должен открыть файл `file_that_does_not_exist.txt`. При обычных обстоятельствах вызов `fopen()` вернет `false`, поскольку файла с таким именем не существует, и выдаст предупреждение PHP для диагностики проблемы. Добавление символа @ перед выражением не меняет возвращаемого значения, но полностью подавляет предполагаемое сообщение.



Оператор @ подавляет отчет об ошибках для строки, к которой он применяется. Если разработчик попытается подавить ошибки в операторе `include`, он легко может скрыть любые предупреждения, уведомления или ошибки, вызванные отсутствием включаемого файла (или доступа). Подавление также будет распространяться на все строки кода внутри файла, то есть любые ошибки (синтаксические или иные) во включаемом коде будут проигнорированы. Таким образом, хотя `@include('some-file.php')` является вполне корректным кодом, подавления ошибок в операторах `include` следует избегать!

Рассматриваемый оператор полезен при подавлении ошибок или предупреждений при операциях доступа к файлам (как в примере из «Решения»). Он также полезен для подавления уведомлений при операциях доступа к элементам массива, как в следующем примере, где определенный параметр `GET` может отсутствовать в запросе:

```
$filename = @$_GET['filename'];
```

Переменная `$filename` установится в значение параметра запроса `filename`, если он задан. В противном случае получится литеральный `null`. Если разработчик

опустит оператор @, значение \$filename по-прежнему будет равно null, но PHP выдаст уведомление о том, что индекса filename не существует в массиве.

Начиная с версии PHP 8.0, данный оператор больше не подавляет фатальные ошибки в PHP, которые останавливают выполнение скрипта.

## Читайте также

Официальная документация PHP по операторам контроля ошибок (<https://oreil.ly/bZkLY>).

# 2.6. Сравнение битов внутри целых чисел

## Задача

Вы хотите с помощью простых флагов идентифицировать состояния и поведения в вашем приложении, где к одному элементу может быть применено несколько флагов.

## Решение

Для определения того, какие флаги доступны и установлены, воспользуйтесь битовой маской (<https://oreil.ly/aevr7>) и побитовым оператором. В следующем примере заданы четыре дискретных флага, каждый из которых представлен двоичным числом. Затем эти флаги объединяются с помощью побитового оператора PHP. Таким образом можно определить, какие флаги установлены и какая ветвь программы должна быть выполнена:

```
const FLAG_A = 0b0001; // 1
const FLAG_B = 0b0010; // 2
const FLAG_C = 0b0100; // 4
const FLAG_D = 0b1000; // 8

// Установка составного флага для приложения
$application = FLAG_A | FLAG_B; // 0b0011 или 3

// Установка составного флага для пользователя
$user = FLAG_B | FLAG_C | FLAG_D; // 0b1110 или 14

// Переключение на основе применяемых пользователем флагов
if ($user & FLAG_B) {
    // ...
} else {
    // ...
}
```

## Обсуждение

Каждый флаг в битовой маске имеет уникальное значение, являющееся целочисленной степенью двойки. Такая структура битовой маски позволяет использовать биты в маске как независимые флаги, которые можно установить или снять по отдельности. В приведенном выше коде каждый флаг явно записан в двоичном виде, чтобы проиллюстрировать, какие биты установлены (1), а какие — нет (0).

Например, `FLAG_B` — это целое число 2, которое представлено в двоичной системе как `0010` (установлен третий бит). Аналогично `FLAG_C` — это целое число 4 с двоичным представлением `0100` (второй бит установлен). Чтобы указать, что оба флага установлены, нужно сложить их вместе, чтобы установить и второй, и третий биты: `0110` или целое число 6.

В нашем конкретном случае сложение — это простая модель, которую легко держать в голове, но это не совсем то, что происходит. Чтобы объединить флаги, не обязательно складывать их вместе, достаточно собрать вместе установленные биты. Если объединить `FLAG_A` с самим собой, то результат должен быть только `FLAG_A`; добавление целочисленного представления (1) к самому себе полностью изменит значение флага.

Вместо сложения задействуйте побитовые операции «или» (`|`) и «и» (`&`) для комбинирования битов и фильтрации по заданным флагам. Для объединения двух флагов примените оператор `|`, чтобы создать новое целое число с битами, которые установлены в любом из используемых флагов. Рассмотрим табл. 2.5 для создания составного флага `FLAG_A | FLAG_C`.

**Таблица 2.5.** Составные двоичные флаги с побитовым «или»

Флаг	Двоичное представление	Десятичное представление
<code>FLAG_A</code>	<code>0001</code>	1
<code>FLAG_C</code>	<code>0100</code>	4
<code>FLAG_A   FLAG_C</code>	<code>0101</code>	5

Для сравнения составных флагов с вашими определениями требуется оператор `(&)`, который возвращает новое число с установленными битами по обе стороны операции. Сравнение флага с самим собой всегда будет давать единицу, что в условных проверках выдает `true`. Сравнение двух значений, у которых установлен любой из одинаковых битов, вернет значение больше 0, которое приводит к `true`. Рассмотрим простой пример оценки, где `FLAG_A & FLAG_C` в табл. 2.6.

Вместо того чтобы сравнивать примитивные флаги друг с другом, можно и нужно создавать составные значения, а затем сравнивать их с вашими наборами флагов.

Следующий блок кода демонстрирует элементы контроля доступа в системе управления контентом для публикации новостных статей. Пользователи могут просматривать, создавать, редактировать или удалять статьи. Уровень доступа определяется самой программой и правами, предоставленными учетной записью пользователя:

```
const VIEW_ARTICLES = 0b0001;
const CREATE_ARTICLES = 0b0010;
const EDIT_ARTICLES = 0b0100;
const DELETE_ARTICLES = 0b1000;
```

**Таблица 2.6.** Составные двоичные флаги с побитовым «и»

Флаг	Двоичное представление	Десятичное представление
FLAG_A	0001	1
FLAG_C	0100	4
FLAG_A & FLAG_C	0000	0

Типичный анонимный посетитель, который не авторизовался, получит лишь стандартное разрешение на просмотр содержимого. Вошедшие в систему пользователи могут создавать статьи, но без соответствующего разрешения не имеют право редактировать их. Аналогично редакторам разрешается просматривать и изменять контент (или удалять его), но не создавать его. Наконец, администраторам дозволяется делать все. Каждая из ролей включает в себя все разрешения из предыдущих уровней:

```
const ROLE_ANONYMOUS = VIEW_ARTICLES;
const ROLE_AUTHOR = VIEW_ARTICLES | CREATE_ARTICLES;
const ROLE_EDITOR = VIEW_ARTICLES | EDIT_ARTICLES | DELETE_ARTICLES;
const ROLE_ADMIN = VIEW_ARTICLES | CREATE_ARTICLES | EDIT_ARTICLES
    | DELETE_ARTICLES;
```

Определив составные роли из примитивных разрешений, приложение способно выстроить логику вокруг проверки активной роли пользователя. В то время как разрешения были объединены с помощью оператора `|`, оператор `&` позволит вам переключаться на основе этих флагов, как показано в функциях из примера 2.8.

### Пример 2.8. Использование флагов битовой маски для управления доступом

```
function get_article($article_id)
{
    $role = get_user_role();

    if ($role & VIEW_ARTICLES) {
        // ...
```

```
    } else {
        throw new UnauthorizedException();
    }
}

function create_article($content)
{
    $role = get_user_role();

    if ($role & CREATE_ARTICLES) {
        // ...
    } else {
        throw new UnauthorizedException();
    }
}

function edit_article($article_id, $content)
{
    $role = get_user_role();
    if ($role & EDIT_ARTICLES) {
        // ...
    } else {
        throw new UnauthorizedException();
    }
}

function delete_article($article_id)
{
    $role = get_user_role();
    if ($role & DELETE_ARTICLES) {
        // ...
    } else {
        throw new UnauthorizedException();
    }
}
```

Битовые маски — это мощный способ реализации простых флагов в любом языке программирования. Однако будьте осторожны, если планируете увеличить количество необходимых флагов, потому что каждый новый флаг представляет собой дополнительную степень двойки, что означает быстрый рост размера значения всех флагов. Тем не менее битовые маски широко применяются как приложениями на PHP, так и самим языком. Настройка встроенной в PHP системы отчетов об ошибках, которую мы рассмотрим в главе 12, задействует битовые значения для определения уровня отчетов об ошибках, используемых самим движком.

## Читайте также

Документация по битовым операторам в PHP (<https://oreil.ly/JmF85>).

## ГЛАВА 3

---

# ФУНКЦИИ

[https://t.me/it\\_boooks/2](https://t.me/it_boooks/2)

Любая компьютерная программа вне зависимости от того, на каком языке программирования она написана, строится путем объединения различных компонентов бизнес-логики. Зачастую эти компоненты должны быть в некоторой степени «многоразовыми» и вызываться из разных мест приложения. Самый простой способ сделать эти компоненты модульными и неоднократно используемыми — инкапсулировать их бизнес-логику в функции — специфические конструкции внутри приложения, на которые можно ссылаться в других частях приложения.

В примере 3.1 показана простая программа для изменения регистра текста. Кодирование без применения функций считается императивным программированием, поскольку вы точно определяете, что программа должна выполнить одну команду (или строку кода) за раз.

**Пример 3.1.** Преобразование первого строчного символа в прописной без использования функций (императивное программирование)

```
$str = "это пример";  
  
if (ord($str[0]) >= 97 && ord($str[0]) <= 122) {  
    $str[0] = chr(ord($str[0]) - 32);  
}  
  
echo $str . PHP_EOL; // Это пример  
  
$str = "а это еще один";  
  
if (ord($str[0]) >= 97 && ord($str[0]) <= 122) {  
    $str[0] = chr(ord($str[0]) - 32);  
}  
  
echo $str . PHP_EOL; // А это еще один  
  
$str = "всего 3 примера";  
  
if (ord($str[0]) >= 97 && ord($str[0]) <= 122) {  
    $str[0] = chr(ord($str[0]) - 32);  
}  
  
echo $str . PHP_EOL; // Всего три примера
```



Функции `ord()` и `chr()` являются ссылками на встроенные функции PHP. Функция `ord()` (<https://oreil.ly/kSI-4>) возвращает двоичное значение символа в виде целого числа. Аналогично функция `chr()` (<https://oreil.ly/0KUmf>) преобразует двоичное значение (представленное в виде целого числа) в соответствующий символ.

Когда вы пишете код без определения функций, он дублируется, поскольку вы вынуждены копировать и вставлять одинаковые блоки по всему приложению. Это нарушает один из ключевых принципов разработки программного обеспечения — DRY, или не повторяйся.

Обычно противоположный этому принципу способ описания — WET, или пиши все дважды. Неоднократное написание одного и того же блока кода приводит к двум проблемам:

- код становится очень длинным и сложным для сопровождения;
- если логику внутри повторяющегося блока необходимо изменить, придется каждый раз обновлять те части вашей программы, в которых она содержится.

Вместо того чтобы дублировать логику императивно, как в примере 3.1, удобнее ввести функцию-обертку, а затем вызвать ее напрямую, как в примере 3.2. Определение функций — это один из основных принципов процедурного программирования, который позволяет вам создавать собственные функции и разбивать код на более мелкие и понятные части.

**Пример 3.2.** Преобразование первого строчного символа в строке в прописной с использованием функции (процедурное программирование)

```
function capitalize_string($str)
{
    if (ord($str[0]) >= 97 && ord($str[0]) <= 122) {
        $str[0] = chr(ord($str[0]) - 32);
    }
    return $str;
}

$str = "это пример";

echo capitalize_string($str) . PHP_EOL; // Это пример

$str = "а это еще один";

echo capitalize_string($str) . PHP_EOL; // А это еще один

$str = "всего 3 примера";

echo capitalize_string($str) . PHP_EOL; // Всего 3 примера
```

Определяемые пользователем функции невероятно мощные и довольно гибкие. Функция `capitalize_string()` из примера 3.2 относительно проста: она принимает строковый параметр и возвращает строку. Однако в определении функции нет никаких указаний на то, что параметр `$str` должен быть строкой, — вы можете с тем же успехом передать число или даже массив, как показано ниже:

```
$out = capitalize_string(25); // 25  
$out = capitalize_string(['a', 'b']); // ['A', 'b']
```

Возвращаясь к теме нестрогой типизации (см. главу 1), стоит упомянуть, что по умолчанию PHP попытается проанализировать ваши намерения, когда вы передаете параметр в `capitalize_string()`, и в большинстве случаев вернет что-то полезное. В случае передачи целого числа PHP выдаст предупреждение о том, что вы некорректно обращаетесь к элементам массива, но все равно вернет целое число без сбоев.

Более сложные программы могут добавлять явную информацию о типе как к параметрам функции, так и к возвращаемым ей значениям, чтобы обеспечить проверку безопасности такого использования. Другие функции способны возвращать не один элемент, а несколько. Строгая типизация наглядно продемонстрирована в рецепте 3.4.

В последующих рецептах демонстрируются различные способы использования функций в PHP, а также начинается процесс создания полноценного приложения.

## 3.1. Доступ к параметрам функций

### Задача

Вы хотите получить доступ к значениям, переданным в функцию при ее вызове из другого места программы.

### Решение

Используйте переменные, определенные в сигнатуре функции, в теле самой функции следующим образом:

```
function multiply($first, $second)  
{  
    return $first * $second;  
}  
  
multiply(5, 2); // 10
```

```
$one = 7;  
$two = 5;  
  
multiply($one, $two); // 35
```

## Обсуждение

Имена переменных, заданные в сигнатуре функции, доступны только в пределах самой функции и будут содержать значения, которые вы передаете в функцию при ее вызове. Внутри фигурных скобок — тела функции — вы вправе использовать эти переменные так же, как если бы определили их сами. Однако любые вносимые в них изменения будут доступны только внутри функции и по умолчанию не повлияют ни на что за ее пределами.

Пример 3.3 иллюстрирует, как конкретное имя переменной может использоваться внутри функции и вне ее, ссылаясь при этом на два совершенно независимых значения. Другими словами, изменение значения `$number` в функции повлияет только на значение внутри функции, но не в родительском приложении.

### Пример 3.3. Локальное распределение функций

```
function increment($number)  
{  
    $number += 1;  
    return $number;  
}  
  
$number = 6;  
  
echo increment($number); // 7  
echo $number; // 6
```

По умолчанию PHP передает значения в функции, а не ссылку на переменную. В контексте примера 3.3 это означает, что PHP записывает значение 6 в новую переменную `$number` внутри функции, выполняет вычисление и возвращает результат. Переменная `$number` вне функции остается нетронутой.



По умолчанию PHP передает простые значения (строки, целые числа, булевые выражения, массивы) по значению. Более сложные объекты, однако, всегда передаются по ссылке. В случае с объектами переменная внутри функции указывает на тот же объект, что и переменная вне функции, а не на его копию.

В некоторых ситуациях вам может потребоваться явно передать переменную по ссылке, а не только ее значение. В этом случае необходимо изменить сигнатуру функции, так как это обновление ее определения, а не то, что может быть преоб-

разовано при вызове функции. В примере 3.4 показано, как перестроится функция `increment()`, чтобы передать `$number` по ссылке, а не по значению.

**Пример 3.4.** Передача переменных по ссылке

```
function increment(&$number)
{
    $number += 1;
    return $number;
}

$number = 6;

echo increment($number); // 7
echo $number; // 7
```

На самом деле имя переменной не обязательно должно совпадать с именем как внутри функции, так и за ее пределами. Я использую `$number` в обоих случаях, чтобы проиллюстрировать разницу в области видимости. Если вы сохраните целое число в переменную `$a` и передадите ее в `increment($a)`, результат будет идентичен примеру 3.4.

## Читайте также

Справочная документация PHP по пользовательским функциям (<https://oreil.ly/9c1Nr>) и передаче переменных по ссылке (<https://oreil.ly/ZfOLR>).

## 3.2. Установка значений параметров функции по умолчанию

### Задача

Вы хотите задать значение по умолчанию для параметра функции, чтобы его не требовалось передавать при вызове функции.

### Решение

Присвойте значение по умолчанию непосредственно в сигнатуре функции. Например:

```
function get_book_title($isbn, $error = 'Unable to query')
{
    try {
        $connection = get_database_connection();
        $book = query_isbn($connection, $isbn);
```

```
        return $book->title;
    } catch {
        return $error;
    }
}

get_book_title('978-1-098-12132-7');
```

## Обсуждение

Код из примера выше запрашивает у БД название книги по ее ISBN. Если запрос по какой-либо причине завершится неудачей, функция вернет строку, переданную в параметр `$error`.

Чтобы сделать этот параметр необязательным, в сигнатуре функции задается значение по умолчанию. При вызове функции `get_book_title()` с одним параметром автоматически используется значение по умолчанию `$error`. В качестве альтернативы вы можете передать в эту переменную собственную строку при вызове функции, например `get_book_title(978-1-098-12132-7, Oops!);`.

При определении функции с параметрами по умолчанию все параметры, которым заданы значения по умолчанию, рекомендуется указывать в конце сигнатуры функции. Хотя определение параметров возможно в любом порядке, это затрудняет корректный вызов функции.

Пример 3.5 иллюстрирует потенциальные проблемы, возникающие при размещении необязательных параметров перед обязательными.



Параметры функции с конкретными значениями по умолчанию можно определять в любом порядке. Однако, начиная с версии PHP 8.0, объявление обязательных параметров после необязательных считается устаревшим. Продолжение применения этой практики приведет к ошибке в будущих версиях PHP.

### Пример 3.5. «Неправильный» порядок параметров по умолчанию

```
function brew_latte($flavor = 'unflavored', $shots)
{
    return "Готовим {$shots} порции {$flavor} латте!";
}

brew_latte('ванильного', 2); ❶
brew_latte(3); ❷
```

❶ Корректное исполнение. Возвращает «Готовим 2 порции ванильного латте!»

❷ Вызывает исключение `ArgumentCountError`, поскольку `$shots` не определено.

В некоторых случаях расположение самих параметров в определенном порядке имеет смысл (например, для улучшения читаемости кода). Однако следует помнить, что если какие-либо параметры являются обязательными, то все параметры слева от них также фактически становятся обязательными, даже если вы попытаетесь задать значение по умолчанию.

## Читайте также

Примеры аргументов по умолчанию в Руководстве по PHP (<https://oreil.ly/XVoK1>).

# 3.3. Использование именованных параметров функций

## Задача

Вы хотите передавать аргументы в функцию, основываясь на имени параметра, а не на его положении.

## Решение

В этом случае допустимым вариантом будет прибегнуть к синтаксису именованных аргументов при вызове функции следующим образом:

```
array_fill(start_index: 0, count: 100, value: 50);
```

## Обсуждение

По умолчанию PHP использует позиционные параметры в определениях функций. Приведенный в «Решении» пример ссылается настроенную функцию `array_fill()` (<https://oreil.ly/jdZQH>), имеющую следующую сигнатуру:

```
array_fill(int $start_index, int $count, mixed $value): array
```

В базовом PHP-коде аргументы для `array_fill()` должны передаваться в том же порядке, в котором они заданы: сначала `$start_index`, потом `$count`, а затем `$value`. Хотя сам порядок не является проблемой, понять смысл каждого значения при визуальном просмотре кода может быть непросто. Если использовать основные упорядоченные параметры, пример из «Решения» будет выглядеть так, что без глубокого знакомства с сигнатурой функции не получится определить, какой параметр за что отвечает:

```
array_fill(0, 100, 50);
```

Именованные параметры функции позволяют идентифицировать, какое значение присваивается той или иной внутренней переменной. Они также дают возможность произвольно менять порядок параметров при вызове функции, так как подобный вызов теперь явно указывает, какое значение присваивается тому или иному параметру.

Еще одно ключевое преимущество именованных аргументов состоит в том, что необязательные аргументы можно полностью пропустить при вызове функции. Рассмотрим на примере 3.6 функцию ведения подробного журнала активности, где несколько параметров считаются опциональными, поскольку задают значения по умолчанию.

### Пример 3.6. Функция ведения подробного журнала активности

```
activity_log(  
    string    $update_reason,  
    string    $note      = '',  
    string    $sql_statement = '',  
    string    $user_name   = 'anonymous',  
    string    $ip_address  = '127.0.0.1',  
    ?DateTime $time       = null  
) : void
```

В примере 3.6 при вызове функции с одним аргументом будут использоваться значения по умолчанию: если \$time равно null, значение заменится новым экземпляром DateTime, представляющим «сейчас». Однако иногда вам может потребоваться заполнить один из этих необязательных параметров, не желая явно задавать их все.

Допустим, вы хотите воспроизвести события, ранее зафиксированные в статическом файле журнала. Активность пользователей была анонимной (поэтому для \$user\_name и \$ip\_address достаточно значений по умолчанию), но вам нужно явно установить дату события. Без именованных аргументов вызов будет выглядеть, как показано в примере 3.7.

### Пример 3.7. Вызов функции activity\_log()

```
activity_log(  
    'Тестирование новой системы',  
    '',  
    '',  
    'anonymous',  
    '127.0.0.1',  
    new DateTime('2021-12-20')  
) ;
```

Благодаря именованным аргументам допускается не устанавливать параметры по умолчанию, а явно задать только те, которые вам нужны. Приведенный выше код можно упростить до следующего:

```
activity_log('Тестирование новой системы', time: new DateTime('2021-12-20'));
```

Кроме того, что именованные параметры значительно упрощают использование `activity_log()`, они еще помогают сохранить чистоту кода. Значения по умолчанию для ваших аргументов хранятся непосредственно в определении функции, а не копируются в каждый ее вызов. Если впоследствии вам понадобится изменить значение по умолчанию, достаточно будет отредактировать только определение функции.

## Читайте также

В оригинальном RFC предлагались именованные параметры (<https://oreil.ly/UdoDP>).

# 3.4. Обеспечение типизации аргументов и возвращаемого значения функции

## Задача

Вы хотите заставить свою программу реализовать типобезопасность и избежать нестрогих сравнений типов, свойственных PHP.

## Решение

Добавьте типы вводимого и возвращаемого значений к определениям функций и по желанию строгое объявление типа в верхнюю часть каждого файла, чтобы обеспечить соответствие значений аннотациям типов (и вызвать сообщение о фатальной ошибке, если они не соответствуют). Например:

```
declare(strict_types=1);

function add_numbers(int $left, int $right): int
{
    return $left + $right;
}

add_numbers(2, 3); ①
add_numbers(2, '3'); ②
```

① Это вполне корректная операция, которая вернет целое число 5.

② Хотя `2 + '3'` — это допустимый PHP-код, строка '`3`' нарушает определение типа функции и приведет к фатальной ошибке.

## Обсуждение

PHP поддерживает различные скалярные типы и позволяет разработчикам объявлять как входные параметры функции, так и возвращаемые, чтобы определить допустимые для каждого из них типы значений. Кроме того, разработчики могут указывать собственные классы и интерфейсы в качестве типов или использовать наследование классов в рамках системы типов<sup>1</sup>.

Типы параметров аннотируются путем размещения типа непосредственно перед именем параметра при определении функции. Аналогично типы возвращаемых значений указываются путем добавления к сигнатуре функции символа : и типа, который функция будет возвращать, как показано ниже:

```
function name(type $parameter): return_type
{
    // ...
}
```

В табл. 3.1 перечислены простейшие типы, используемые PHP.

**Таблица 3.1.** Простые одиночные типы в PHP

Тип	Описание
array	Значение должно быть массивом (содержащим любой тип значений)
callable	Значение должно быть вызываемой функцией
bool	Значение должно быть логическим
float	Значение должно быть числом с плавающей точкой
int	Значение должно быть целым числом
string	Значение должно быть строкой
iterable	Значение должно быть массивом или объектом, реализующим Traversable
mixed	Объект может быть любым значением
void	Тип только для возвращаемого значения, указывающий на то, что функция не возвращает значения
never	Тип только для возвращаемого значения, указывающий на то, что функция не возвращается; она либо вызывает exit, либо выбрасывает исключение, либо намеренно зацикливается

---

<sup>1</sup> Пользовательские классы и объекты подробно рассматриваются в главе 8.

Подробную информацию о типах `iterable`, `mixed`, `void` и `never` вы найдете по ссылкам <https://oreil.ly/tITI1>, <https://oreil.ly/V8VOc>, <https://oreil.ly/Izmvp> и <https://oreil.ly/48KVB> соответственно.

Кроме того, для определения типов можно использовать как встроенные, так и пользовательские классы, как показано в табл. 3.2.

**Таблица 3.2.** Типы объектов в PHP

Тип	Описание
Имя класса/ интерфейса	Значение должно быть экземпляром указанного класса или реализацией интерфейса
<code>self</code>	Значение должно быть экземпляром того же класса, что и класс, в котором используется объявление
<code>parent</code>	Значение должно быть экземпляром родительского класса, в котором используется объявление
<code>object</code>	Значение должно быть экземпляром объекта

PHP также позволяет расширять простые скалярные типы, делая их обнуляемыми (`nullable`) или организовывая их в объединение типов. Чтобы преобразовать определенный тип в обнуляемый (`nullable`), необходимо приписать к аннотации типа знак `?`, который указывает компилятору разрешать значения либо указанного типа, либо `null`, как в примере 3.8.

**Пример 3.8.** Функция, использующая обнуляемые параметры

```
function say_hello(?string $message): void
{
    echo 'Привет, ';

    if ($message === null) {
        echo 'мир!';
    } else {
        echo $message . '!';
    }
}

say_hello('читатель'); // Привет, читатель!
say_hello(null); // Привет, мир!
```

Объединение типов дает возможность группировать несколько простых типов в единое объявление с помощью символа `|`. Если переписать объявления типов

в примере из «Решения» для использования с объединением типов, комбинирующим строки и целые числа, то фатальная ошибка, вызываемая передачей строки для сложения, будет устранена. Рассмотрим возможный вариант на примере 3.9, который допускает в качестве параметров либо целые числа, либо строки.

**Пример 3.9.** Адаптация примера из «Решения» для использования объединения типов

```
function add_numbers(int|string $left, int|string $right): int
{
    return $left + $right;
}

add_numbers(2, '3'); // 5
```

Основная проблема этой альтернативы заключается в том, что сложение строк с помощью оператора + не имеет смысла в PHP. Если оба параметра являются числовыми (либо целые числа, либо целые числа, представленные в виде строк), то функция будет работать нормально. Если же хотя бы один из параметров не является числовой строкой, PHP выдаст ошибку `TypeError`, поскольку не знает, как «сложить» две строки вместе. Именно таких ошибок вы надеетесь избежать, добавляя в код объединения типов и обеспечивая строгую типизацию.

По умолчанию PHP использует свою систему типизации для подсказки о том, какие типы могут быть переданы в функции и возвращены. Это полезно для предотвращения передачи некорректных данных в функцию, но обеспечение правильной типизации в значительной степени зависит от дисциплины разработчика или дополнительных инструментов<sup>1</sup>. Вместо того чтобы полагаться на способность человека проверять код, PHP позволяет статически объявлять в каждом файле, что все вызовы должны следовать строгой типизации.

Поместив `declare(strict_types=1);` в начало файла, вы сообщаете компилятору PHP, что все вызовы в этом файле должны соответствовать объявлению типов параметров и возвращаемых значений. Обратите внимание, что эта директива применяется к вызовам внутри файла, в котором она используется, а не к определениям функций в нем. Если вы вызываете функции из другого файла, PHP будет соблюдать объявления типов и тут. Однако размещение такой директивы в вашем файле не заставит другие файлы, что ссылаются на ваши функции, подчиниться системе типизации.

---

<sup>1</sup> PHP CodeSniffer (<https://oreil.ly/G4tHg>) — это популярный инструмент для автоматического сканирования кодовой базы и обеспечения соответствия всего кода определенному стандарту кодирования. Его можно легко расширить, чтобы обеспечить строгое объявление типов во всех файлах.

## Читайте также

Документация PHP по объявлениям типов (<https://oreil.ly/I9D33>) и конструкции declare ([https://oreil.ly/P2jM\\_](https://oreil.ly/P2jM_)).

# 3.5. Определение функции с переменным числом аргументов

## Задача

Вы хотите определить функцию, которая принимает один или несколько аргументов, не зная заранее, сколько значений будет передано.

## Решение

Используйте оператор расширения spread (...) для определения числа переменных или аргументов:

```
function greatest(int ...$numbers): int
{
    $greatest = 0;
    foreach ($numbers as $number) {
        if ($number > $greatest) {
            $greatest = $number;
        }
    }
    return $greatest;
}

greatest(7, 5, 12, 2, 99, 1, 415, 3, 7, 4);
// 415
```

## Обсуждение

Оператор spread автоматически добавляет все параметры, указанные в конкретной позиции или после нее, в массив. Этот массив можно типизировать, добавив к оператору spread объявление типа (подробнее о типизации читайте в рецепте 3.4), чтобы каждый элемент коллекции соответствовал определенному типу. Вызов функции `greatest(2, "five")`; из примера выше, выведет ошибку `TypeError`, поскольку вы явно объявили тип `int` для каждого члена массива `$numbers`.

Ваша функция способна принимать более одного позиционного параметра и при этом задействовать оператор spread для приема неограниченного количества дополнительных аргументов. Функция, определенная в примере 3.10, выводит на экран приветствие для неограниченного числа людей.

**Пример 3.10.** Использование оператора spread

```
function greet(string $greeting, string ...$names): void
{
    foreach($names as $name) {
        echo $greeting . ', ' . $name . PHP_EOL;
    }
}

greet('Привет', 'Тони', 'Стив', 'Ванда', 'Питер');
// Привет, Тони
// Привет, Стив
// Привет, Ванда
// Привет, Питер

greet('Добро пожаловать', 'Элис', 'Боб');
// Добро пожаловать, Элис
// Добро пожаловать, Боб
```

Оператор spread обладает большей универсальностью, чем обыкновенное определение функции. Хотя его можно использовать для упаковки нескольких аргументов в массив, он также подойдет для распаковки массива в несколько аргументов для более традиционного вызова функции. В примере 3.11 показано, как работает распаковка массива с помощью оператора spread для передачи коллекции в функцию, которая не принимает массива.

**Пример 3.11.** Распаковка массива с помощью оператора spread

```
function greet(string $greeting, string $name): void
{
    echo $greeting . ', ' . $name . PHP_EOL;
}

$params = ['Привет', 'мир'];
greet(...$params);
// Привет, мир
```

В некоторых случаях более сложная функция может возвращать несколько значений (об этом мы поговорим в следующем рецепте), поэтому передача возвращаемого значения одной функции в другую становится простой задачей благодаря оператору spread. Фактически любой массив или переменная, реализующие интерфейс Traversable (<https://oreil.ly/jVUvs>) в PHP, могут быть распакованы в вызов функции таким образом.

## Читайте также

Документация PHP по спискам аргументов переменной длины (<https://oreil.ly/9IoHh>).

# 3.6. Возвращение нескольких значений

## Задача

Вы хотите вернуть несколько значений за один вызов функции.

## Решение

Вместо того чтобы возвращать одно значение, верните массив значений и распакуйте их с помощью `list()` вне функции:

```
function describe(float ...$values): array
{
    $min = min($values);
    $max = max($values);
    $mean = array_sum($values) / count($values);

    $variance = 0.0;
    foreach($values as $val) {
        $variance += pow(($val - $mean), 2);
    }
    $std_dev = (float) sqrt($variance/count($values));

    return [$min, $max, $mean, $std_dev];
}

$values = [1.0, 9.2, 7.3, 12.0];
list($min, $max, $mean, $std) = describe(...$values);
```

## Обсуждение

PHP способен возвращать только одно значение при вызове функции, но само это значение может представлять собой массив из нескольких значений. В паре с конструкцией `list()` этот массив может быть легко разбит на отдельные переменные для дальнейшего использования в программе.

Хотя необходимость возвращать множество различных значений возникает не так часто, иногда такая функция пригождается. Один из примеров — веб-аутентификация. Многие современные системы сегодня используют стандарт

JSON Web Tokens (JWT), который представляет собой строки данных, закодированных в формате Base64 и разделенных точками. Каждый компонент JWT является отдельной, дискретной частью: заголовком, описывающим используемый алгоритм, данными в полезной нагрузке токена и верифицируемой подписью на эти данные.

При чтении JWT PHP-приложения часто применяют встроенную функцию `explode()`, которая разбивает строки с помощью разделителя. Функция `explode()` может выглядеть следующим образом:

```
$jwt_parts = explode('.', $jwt);
$header = base64_decode($jwt_parts[0]);
$payload = base64_decode($jwt_parts[1]);
$signature = base64_decode($jwt_parts[2]);
```

Представленный код вполне рабочий, но повторяющиеся ссылки внутри массива усложняют его. Кроме того, разработчикам приходится вручную декодировать каждую часть JWT по отдельности; если забыть вызвать `base64_decode()`, возникнет риск возникновения фатальной ошибки программы.

Альтернативный подход заключается в том, чтобы распаковать и автоматически декодировать JWT внутри функции и вернуть массив компонентов, как показано в примере 3.12.

### Пример 3.12. Декодирование JWT

```
function decode_jwt(string $jwt): array
{
    $parts = explode('.', $jwt);

    return array_map('base64_decode', $parts);
}

list($header, $payload, $signature) = decode_jwt($jwt);
```

Дополнительным преимуществом использования функции для распаковки JWT вместо непосредственного разложения каждого элемента является возможность встраивания автоматической проверки подписи или даже фильтрации JWT на предмет приемлемости, основываясь на алгоритмах шифрования, объявленных в заголовке. Хотя эта логика может быть применена процедурно при обработке JWT, сохранение всего в одном определении функции приводит к созданию более чистого и удобного кода.

Самый большой недостаток при возврате нескольких значений в одном вызове функции заключается в типизации. Тип возвращаемого значения у этих функций —

массив, но PHP не позволяет указать тип элементов в нем. Здесь есть потенциальные обходные пути решения сложившейся проблемы. Например, документирование сигнатуры функции и интеграция с инструментами статического анализа, такими как Psalm (<https://psalm.dev/>) или PHPStan (<https://phpstan.org/>). Однако у нас нет встроенной поддержки типизированных массивов в языке. Таким образом, если вы используете строгую типизацию (а так и должно быть), возврат нескольких значений в рамках одного вызова функции должен быть редким явлением.

## Читайте также

Рецепт 3.5 о передаче переменного числа аргументов и рецепт 1.3 для более подробной информации о конструкции `list()`. Также обратитесь к документации phpDocumentor по типизированным массивам (<https://oreil.ly/RsXGh>), которые могут быть усилены такими инструментами, как Psalm.

# 3.7. Доступ к глобальным переменным внутри функции

## Задача

Ваша функция должна обращаться к глобально определенной переменной из другого места приложения.

## Решение

Используйте ключевое слово `global` перед любыми глобальными переменными, чтобы получить к ним доступ в области видимости функции:

```
$counter = 0;

function increment_counter()
{
    global $counter;

    $counter += 1;
}

increment_counter();

echo $counter; // 1
```

## Обсуждение

PHP подразделяет операции на различные области видимости в зависимости от контекста, в котором определяется переменная. Для большинства программ единая область видимости охватывает все включенные или необходимые файлы. Переменная, определенная в этой глобальной области видимости, доступна везде независимо от того, какой файл выполняется в данный момент, как показано в примере 3.13.

**Пример 3.13.** Переменные, определенные в глобальной области видимости, доступны для включенных сценариев

```
$apple = 'honeycrisp';  
  
include 'someotherscript.php'; ①
```

① Переменная \$apple также определена в этом скрипте и доступна для использования.

Однако пользовательские функции задают собственную область видимости. Переменная, определенная вне такой функции, недоступна в теле функции. Аналогично любая переменная, определенная внутри функции, недоступна за ее пределами. Пример 3.14 иллюстрирует границы родительской области и области видимости функции в программе.

**Пример 3.14.** Локальная и глобальная видимость

```
$a = 1; ①  
  
function example(): void  
{  
    echo $a . PHP_EOL; ②  
    $a = 2; ③  
  
    $b = 3; ④  
}  
  
example();  
  
echo $a . PHP_EOL; ⑤  
echo $b . PHP_EOL; ⑥
```

① Переменная \$a изначально определена в родительской области видимости.

② Внутри области видимости функции переменная \$a еще не определена. Попытка вывести ее значение через echo приведет к предупреждению.

- ❸ Определение переменной с именем `$a` внутри функции не перезапишет значения одноименной переменной вне функции.
- ❹ Определение переменной с именем `$b` внутри функции делает ее доступной внутри функции, но это значение не выходит за пределы области видимости функции.
- ❺ Вывод `$a` за пределы функции, даже после вызова `example()`, выведет начальное значение, которое вы задали, поскольку функция не изменила значения переменной.
- ❻ Поскольку `$b` была определена внутри функции, она не определена в области видимости данного приложения.



Если функция определена таким образом, что принимает переменную по ссылке, ее можно передать в функцию тем же путем. Однако это решение принимается при определении функции и не является флагом среди выполнения, доступным программам, использующим эту функцию, уже после ее завершения. В примере 3.4 показано, как может выглядеть передача по ссылке.

Чтобы обратиться к переменным, определенным за пределами своей области видимости, функция должна объявить эти переменные как глобальные в собственной области видимости. Чтобы ссылаться на родительскую область видимости, вы можете переписать пример 3.15, который представляет собой переосмысление примера 3.14.

**Пример 3.15.** Локальная и глобальная видимость, альтернативный взгляд

```
$a = 1;

function example(): void
{
    global $a, $b; ❶

    echo $a . PHP_EOL; ❷
    $a = 2; ❸

    $b = 3; ❹
}

example();

echo $a . PHP_EOL; ❺
echo $b . PHP_EOL; ❻
```

- ❶ Объявляя `$a` и `$b` глобальными переменными, вы даете функции указание использовать значения из родительской области видимости, а не из своей собственной.

- ❷ Теперь, ссылаясь на глобальную переменную `$a`, вы можете вывести ее значение.
- ❸ Любые изменения `$a` в области видимости функции повлияют на переменную в родительской области видимости.
- ❹ Вы определяете `$b`, но поскольку это глобальная переменная, определение распространится и на родительскую область видимости.
- ❺ Вывод `$a` будет отражать изменения, сделанные в области видимости `example()`, так как вы объявили переменную глобальной.
- ❻ Аналогично `$b` теперь определяется глобально и может быть выведена.

В PHP нет ограничений на количество глобальных переменных, все зависит только от объема памяти, доступной системе. Кроме того, все глобальные переменные допускается перечислять через специальный массив PHP `$GLOBALS`. Этот ассоциативный массив содержит ссылки на все переменные, определенные в глобальной области видимости. Данный инструмент может пригодиться, если вы захотите сослаться на конкретную переменную, не объявляя ее глобальной, как показано в примере 3.16.

**Пример 3.16.** Использование ассоциативного массива `$GLOBALS`

```
$var = 'global';

function example(): void
{
    $var = 'local';

    echo 'Локальная переменная: ' . $var . PHP_EOL;
    echo 'Глобальная переменная: ' . $GLOBALS['var'] . PHP_EOL;
}

example();
// Локальная переменная: local
// Глобальная переменная: global
```



Начиная с версии PHP 8.1, больше невозможно перезаписать весь массив `$GLOBALS`. В предыдущих версиях разрешалось сбросить его в пустой массив (например, во время тестового запуска кода). В дальнейшем вы сможете редактировать только содержимое массива, а не манипулировать коллекцией целиком.

Глобальные переменные представляют собой удобный инструмент для обращения к состоянию всего приложения, однако их чрезмерное использование

чаще всего приводит к путанице и проблемам с поддержкой кода. Некоторые крупные приложения широко применяют глобальные переменные. В их числе WordPress, проект с открытым исходным кодом на базе PHP, который обеспечивает работу более 40 % Интернета<sup>1</sup> (<https://oreil.ly/jztni>). Тем не менее большинство разработчиков приложений согласны с тем, что глобальные переменные следует использовать осторожно (если вообще стоит), чтобы сохранить чистоту и простоту обслуживания систем.

## Читайте также

Документация PHP по области видимости переменных (<https://oreil.ly/tN5tV>) и специальному массиву \$GLOBALS (<https://oreil.ly/z9JJS>).

# 3.8. Управление состоянием внутри функции при многократных вызовах

## Задача

Ваша функция должна отслеживать изменение своего состояния с течением времени.

## Решение

Используйте ключевое слово `static` для определения локальной переменной, которая сохраняет свое состояние между вызовами функций:

```
function increment()
{
    static $count = 0;

    return $count++;
}

echo increment(); // 0
echo increment(); // 1
echo increment(); // 2
```

---

<sup>1</sup> По данным W3Techs ([https://oreil.ly/8Y\\_Zp](https://oreil.ly/8Y_Zp)), по состоянию на март 2023 года, WordPress занимала около 63 % сайтов, использующих системы управления контентом, и более 43 % всех сайтов.

## Обсуждение

Статическая переменная существует только в области видимости функции, в которой она объявлена. Однако, в отличие от обычных локальных переменных, она сохраняет свое значение при каждом возвращении в область видимости функции. Таким образом, функция может зафиксировать свое состояние и отслеживать определенные данные (например, количество вызовов) между независимыми вызовами.

В обычной функции с помощью оператора присвоения (=) переменной присваивается значение. Когда применяется ключевое слово `static`, само присваивание происходит только при первом вызове функции. Последующие вызовы будут ссылаться на предыдущее состояние переменной и позволят программе либо задействовать в работе, либо изменять сохраненное значение.

Одним из наиболее распространенных сценариев использования статических переменных является отслеживание состояния рекурсивной функции. Пример 3.17 демонстрирует функцию, которая рекурсивно вызывает себя фиксированное количество раз перед завершением.

**Пример 3.17.** Использование статической переменной для ограничения глубины рекурсии

```
function example(): void
{
    static $count = 0;

    if ($count >= 3) {
        $count = 0;
        return;
    }

    $count += 1;

    echo 'Выполнение цикла номер ' . $count . PHP_EOL;
    example();
}
```

Ключевое слово `static` также можно применять для отслеживания ресурсов, которые неоднократно могут потребоваться функции, но для использования которых вы, вероятно, захотели бы ввести ограничения. Рассмотрим функцию, регистрирующую сообщения в БД: возможно, у вас не получится передать соединение с базой данных непосредственно в функцию, но вы хотите убедиться, что функция открывает только одно соединение. Такая функция реализуется в примере 3.18.

**Пример 3.18.** Использование статической переменной для хранения соединения с БД

```
function logger(string $message): void
{
    static $dbh = null;
    if ($dbh === null) {
        $dbh = new PDO(DATABASE_DSN, DATABASE_USER, DATABASE_PASSWORD);
    }

    $sql = 'INSERT INTO messages (message) VALUES (:message)';
    $statement = $dbh->prepare($sql);

    $statement->execute([':message' => $message]);
}

logger('Это тест'); ❶
logger('Это еще один тест'); ❷
```

❶ Первый вызов функции `logger()` определит значение статической переменной `$dbh`. В этом случае произойдет подключение к БД с помощью интерфейса PHP Data Objects (PDO) (<https://oreil.ly/do1eJ>). Данный интерфейс представляет собой стандартный объект PHP для доступа к базам данных.

❷ Каждый последующий вызов функции `logger()` будет использовать начальное соединение с БД, сохраненное в `$dbh`.

Обратите внимание, что PHP автоматически управляет памятью и очищает переменные из нее, когда они покидают область видимости. Для обычных переменных внутри функции это означает, что они освобождаются из памяти сразу после завершения функции. Статические и глобальные переменные никогда не очищаются до выхода из программы, так как всегда находятся в области видимости. Будьте внимательны при использовании ключевого слова `static`, чтобы лишний раз не хранить в памяти большие фрагменты данных. В примере 3.18 открывается соединение с базой данных, которое никогда автоматически не закроется созданной вами функцией.

Хотя ключевое слово `static` — это мощный инструмент повторного использования состояния между вызовами функции, его следует применять с осторожностью, чтобы ваше приложение работало стабильно. Во многих случаях лучше явно передавать переменные, представляющие состояние, в функцию. Еще лучше — инкапсулировать состояние функции как часть общего объекта, о чем мы поговорим в главе 8.

## Читайте также

Документация PHP по определению границ переменных, включая ключевое слово `static` (<https://oreil.ly/-yflc>).

## 3.9. Определение динамических функций

### Задача

Необходимо определить анонимную функцию и сослаться на нее как на переменную в своем приложении, чтобы она использовалась или вызывалась только один раз.

### Решение

Определите замыкание, которое можно присвоить переменной и передать в другую функцию при необходимости:

```
$greet = function($name) {
    echo 'Привет, ' . $name . PHP_EOL;
};

$greet('мир!');
// Привет, мир!
```

### Обсуждение

В то время как у большинства функций в PHP есть конкретные имена, язык поддерживает создание неименованных (или анонимных) функций, также называемых замыканиями (*closures*), или лямбдами. Эти функции содержат как простую, так и сложную логику и могут быть напрямую присвоены переменным для использования в других частях программы.

Внутри PHP анонимные функции реализуются с помощью собственного класса *Closure* (<https://oreil.ly/u5qt7>). Этот класс объявлен как *final*, что означает, что ни один класс не может прямо наследовать его. Тем не менее все анонимные функции являются экземплярами данного класса и могут быть использованы в качестве как функций, так и объектов.

По умолчанию замыкания не наследуют область видимости от родительского приложения и, как обычные функции, определяют переменные в собственной области видимости. Переменные из родительской области видимости допускается передавать непосредственно в замыкание с помощью директивы *use* при определении функции. Пример 3.19 иллюстрирует, как переменные из одной области видимости могут быть динамически переданы в другую.

**Пример 3.19.** Обмен переменными между областями видимости с помощью *use()*

```
$some_value = 42;

$foo = function() {
```

```
echo $some_value;  
};  
  
$bar = function() use ($some_value) {  
    echo $some_value;  
};  
  
$foo(); // Предупреждение: неопределенная переменная  
  
$bar(); // 42
```

Анонимные функции используются во многих проектах, чтобы инкапсулировать часть логики для применения к коллекции данных. Следующий рецепт посвящен именно такому сценарию использования.



В старых версиях PHP для достижения подобных целей служила функция `create_function()` (<https://oreil.ly/RRMgO>). Разработчики могли создать анонимную функцию в виде строки и передать этот код в `create_function()`, чтобы превратить его в экземпляр замыкания. К сожалению, такой метод не обходился без функции `eval()`, что считается крайне небезопасной практикой. Хотя в некоторых проектах все еще встречается `create_function()`, сама функция устарела и полностью удалена из языка в PHP 8.0.

## Читайте также

Документация PHP по анонимным функциям (<https://oreil.ly/W0QPL>).

# 3.10. Передача функций в качестве параметров другим функциям

## Задача

Вы хотите определить часть реализации функции и передать ее в качестве аргумента другой функции.

## Решение

Определите замыкание, которое реализует часть необходимой вам логики, и передайте его непосредственно в другую функцию как обычную переменную:

```
$reducer = function(?int $carry, int $item): int {  
    return $carry + $item;  
};
```

```
function reduce(array $array, callable $callback, ?int $initial = null): ?int
{
    $acc = $initial;
    foreach ($array as $item) {
        $acc = $callback($acc, $item);
    }

    return $acc;
}

$list = [1, 2, 3, 4, 5];
$sum = reduce($list, $reducer); // 15
```

## Обсуждение

Многие считают PHP функциональным языком программирования, поскольку функции являются первостепенными элементами языка и могут быть привязаны к именам переменных, переданы в качестве аргументов или даже возвращены из других функций. PHP поддерживает функции как переменные через специальный тип `callable` (<https://oreil.ly/m7skJ>). Многие базовые функции (например, `usort()`, `array_map()` и `array_reduce()`) поддерживают передачу параметра `callable`, который затем используется внутри функции для определения ее общей реализации.

Функция `reduce()`, определенная в примере из «Решения», является пользовательской реализацией встроенной в PHP функции `array_reduce()`. Обе они ведут себя одинаково, и решение можно переписать так, чтобы передать `$reducer` непосредственно в родную функцию PHP без изменения результата:

```
$sum = array_reduce($list, $reducer); // 15
```

Кроме возможности передавать функции в качестве переменных, PHP также позволяет определять частичные реализации функций. Это достигается путем определения функции, возвращающей другую функцию, которую допускается использовать в иных местах программы.

Например, вы можете определить функцию для создания базовой арифметической процедуры, которая умножает любое вводимое число на фиксированное значение, как в примере 3.20. Основная функция возвращает новую функцию при каждом вызове, поэтому вы можете создавать функции для удвоения или утроения произвольных значений и использовать их по своему усмотрению.

### Пример 3.20. Функция умножения с частичным применением

```
function multiplier(int $base): callable
{
    return function(int $subject) use ($base): int {
```

```
        return $base * $subject;
    };
}

$double = multiplier(2);
$triple = multiplier(3);

$double(6); // 12
$double(10); // 20
$triple(3); // 9
$triple(12); // 36
```

Подобное разбиение функций на части называется каррированием (currying, oreil.ly/-a4l). Это практика изменения функции с несколькими входными параметрами в ряд функций, каждая из которых принимает один параметр, причем большинство этих параметров являются функциями сами по себе. Чтобы понять, как это работает в PHP, рассмотрим пример 3.21 и перепишем функцию `multiplier()`.

### Пример 3.21. Каррирование в PHP

```
function multiply(int $x, int $y): int ❶
{
    return $x * $y;
}

multiply(7, 3); // 21

function curried_multiply(int $x): callable ❷
{
    return function(int $y) use ($x): int { ❸
        return $x * $y; ❹
    };
}

curried_multiply(7)(3); // 21 ❺
```

❶ В исходном виде функция принимает два значения, перемножает их и возвращает конечный результат.

❷ При каррировании функции основная задача состоит в том, чтобы каждая функция-компонент принимала только одно значение. Новая функция `curried_multiply()` принимает только один параметр, но возвращает функцию, которая использует его внутри себя.

❸ Внутренняя функция автоматически ссылается на значение, переданное предыдущим вызовом функции (с помощью директивы `use`).

❶ Результирующая функция реализует ту же бизнес-логику, что и базовая форма.

❷ Вызов каррированной функции выглядит как вызов нескольких функций последовательно, но результат остается тем же.

Самым большим преимуществом каррирования, как в примере 3.21, является то, что частично примененная функция может быть передана как переменная и задействована в других местах. Аналогично использованию функции `multiplier()`, вы можете создать функцию удвоения или утроения, частично применив заготовленный множитель следующим образом:

```
$double = curried_multiply(2);
$triple = curried_multiply(3);
```

Частично применяемые, каррированные функции сами являются вызываемыми, но их можно передавать в другие функции в качестве переменных и обращаться к ним позже.

## Читайте также

Подробнее об анонимных функциях в рецепте 3.9.

## 3.11. Стрелочные функции

### Задача

Вы хотите создать простую анонимную функцию, которая ссылается на родительскую область видимости без громоздких деклараций.

### Решение

Используйте синтаксис стрелочной функции (arrow function) PHP, чтобы определить функцию, которая автоматически наследует область видимости своего родителя:

```
$outer = 42;

$anon = fn($add) => $outer + $add;

$anon(5); // 47
```

## Обсуждение

Стрелочные функции были введены в PHP 7.4 как способ написания более лаконичных анонимных функций, как в рецепте 3.9. Стрелочные функции автоматически перехватывают все ссылочные переменные и импортируют их (по значению, а не по ссылке) в область видимости функции.

Код из «Решения» и код из примера 3.22 функционально идентичны, однако последний представляет собой более «подробную» версию.

**Пример 3.22.** Менее лаконичная форма анонимной функции

```
$outer = 42;

$anon = function($add) use ($outer) {
    return $outer + $add;
};

$anon(5);
```

Стрелочные функции всегда возвращают значение: совершенно невозможно вернуть `void`. Эти функции следуют очень специальному синтаксису и неизбежно возвращают результат своего выражения: `fn(аргументы) => выражение`. Такая структура делает стрелочные функции полезными в самых разных ситуациях.

Примером служит краткое внутреннее определение функции, применяемой ко всем элементам массива через встроенную в PHP функцию `array_map()`. Предположим, у вас есть массив строк, представляющих собой целочисленные значения, и вы хотите преобразовать его в массив целых чисел, чтобы обеспечить надлежащую типобезопасность. В примере 3.23 показано, как этого добиться.

**Пример 3.23.** Преобразование массива числовых строк в массив целых чисел

```
$input = ['5', '22', '1093', '2022'];

$output = array_map(fn($x) => intval($x), $input);
// $output = [5, 22, 1093, 2022]
```

Стрелочные функции позволяют использовать только односторочное выражение. Если ваша логика достаточно сложна и требует нескольких выражений, используйте стандартную анонимную функцию (см. рецепт 3.9) или определите именованную функцию в коде. Тем не менее следует отметить, что стрелочная функция сама по себе является выражением, поэтому одна стрелочная функция может возвращать другую.

Способность возвращать стрелочные функции в качестве выражений других стрелочных функций позволяет использовать их в каррированных или частично применяемых функциях для поощрения повторного использования кода. Предположим, вы хотите передать в программу функцию, которая выполняет операции с числами с фиксированным модулем. Вы можете сделать это, определив одну стрелочную функцию для вычислений и обернув ее в другую, которая задает модуль и присваивает конечную, каррированную функцию переменной, которую можно задействовать в другом месте, как в примере 3.24.



Операции по модулю применяются для создания функций, которые всегда возвращают определенный набор целочисленных значений независимо от входного целого числа. Вы берете модуль двух целых чисел, делите их и возвращаете целочисленный остаток. Например, «12 по модулю 3» записывается как 12 % 3 и возвращает 0 (остаток от деления 12 на 3). Аналогично, «15 по модулю 6» записывается как 15 % 6 и возвращает 3 по тому же принципу. Возврат результата операции по модулю никогда не бывает больше самого модуля (3 или 6 в предыдущих двух примерах соответственно). Такая арифметика обычно используется для объединения больших коллекций входных значений или для выполнения криптографических операций, о которых подробнее говорится в главе 9.

#### Пример 3.24. Каррирование функций с помощью стрелочных функций

```
$modulo = fn($x) => fn($y) => $y % $x;  
  
$mod_2 = $modulo(2);  
$mod_5 = $modulo(5);  
  
$mod_2(15); // 1  
$mod_2(20); // 0  
$mod_5(12); // 2  
$mod_5(15); // 0
```

Наконец, стрелочные функции, как и обычные, могут принимать несколько аргументов. Вместо того чтобы передавать одну переменную (или неявно ссылаться на переменные, определенные в родительской области видимости), вы можете так же легко определить функцию с несколькими параметрами и свободно использовать их внутри выражения. Примером может служить следующая тривиальная функция равенства:

```
$eq = fn($x, $y) => $x == $y;  
  
$eq(42, '42'); // true
```

### Читайте также

Подробнее об анонимных функциях в рецепте 3.9 и в документации PHP Manual по стрелочным функциям (<https://oreil.ly/MLURC>).

## 3.12. Создание функции без возвращаемого значения

### Задача

Вам нужно определить функцию, которая после завершения работы не возвращает программе никаких данных.

### Решение

Используйте явные объявления типов и указывайте в качестве типа возвращаемого значения `void`:

```
const MAIL_SENDER = 'wizard@oz.example';
const MAIL SUBJECT = 'Входящее сообщение от Удивительного Волшебника';

function send_email(string $to, string $message): void
{
    $headers = ['От' => MAIL_SENDER];

    $success = mail($to, MAIL SUBJECT, $message, $headers);

    if (!$success) {
        throw new Exception('У человека за кулисами перерыв.');
    }
}

send_email('dorothy@kansas.example', 'Добро пожаловать в Изумрудный город!');
```

### Обсуждение

В примере выше используется штатная функция PHP `mail()` для отправки простого сообщения с фиксированным заголовком указанному получателю. Она возвращает либо `true` (при успехе), либо `false` (при ошибке). В данном случае вам нужно просто создать исключение, если что-то идет не так, но в остальном возвращать данные как обычно.



Иногда по завершении работы функции необходимо вернуть флаг — булево значение, строку или `null`, — чтобы указать, что произошло, и чтобы остальная часть программы могла вести себя соответствующим образом. Функции, которые ничего не возвращают, хотя и относительно редко, но все же встречаются, когда ваша программа общается с внешней стороной, и результат этого взаимодействия не влияет на остальную часть программы. Отправка соединения в очередь сообщений или запись в системный журнал ошибок — оба распространенных сценария использования функции, возвращающей `void`.

В PHP тип возвращаемого значения `void` применяется на этапе компиляции, то есть ваш код выдаст фатальную ошибку, если тело функции вернет хоть что-то, даже если вы еще ничего не выполнили. Пример 3.25 иллюстрирует как допустимое, так и недопустимое использование `void`.

**Пример 3.25.** Допустимые и недопустимые варианты использования `void`

```
function returns_scalar(): void
{
    return 1; ❶
}

function no_return(): void
{
    ❷
}

function empty_return(): void
{
    return; ❸
}

function returns_null(): void
{
    return null; ❹
}
```

❶ Возврат скалярного типа (например, `string`, `integer` или `boolean`) приведет к фатальной ошибке.

❷ Пропуск любого вида возврата в функции допустим.

❸ Отсутствие данных при явном возвращении допустимо.

❹ Несмотря на то что тип `null` — это пустая ссылка, он все равно учитывается как возвращаемое значение и поэтому вызовет критическую ошибку.

В отличие от большинства других типов в PHP, тип `void` может использоваться только для возврата. Его нельзя задействовать в качестве типа параметра в определении функции: попытки сделать это приведут к фатальной ошибке во время компиляции.

## Читайте также

Оригинальный RFC, представляющий тип возврата `void` ([https://oreil.ly/FvRb\\_](https://oreil.ly/FvRb_)) в PHP 7.1.

## 3.13. Создание функции, которая не возвращается

### Задача

Вам нужно определить функцию, которая явно завершает работу и гарантирует, что другие части вашего приложения не будут ждать, что она вернется.

### Решение

Явно задавайте типы и указывайте тип возвращаемого значения `never`. Например:

```
function redirect(string $url): never
{
    header("Location: $url");
    exit();
}
```

### Обсуждение

Некоторые операции в PHP необходимо выполнять в конце текущего процесса. В частности, функция `header()`, предназначенная для отправки заголовков ответа на сервер, должна быть вызвана до вывода тела ответа. Особенно важно соблюдать этот порядок при перенаправлении. Если вы попытаетесь выполнить перенаправление после вывода данных, оно не сработает.

Тип возвращаемого значения `never` в PHP означает, что функция гарантированно завершит программу с помощью `exit()`, `die()` или генерации исключения.

Если функция с типом возвращаемого значения `never` все равно возвращается неявно, как в примере 3.26, PHP выдаст исключение `TypeError`.

**Пример 3.26.** Неявный возврат в функции, которая не должна возвращаться

```
function log_to_screen(string $message): never
{
    echo $message;
}
```

Аналогично если функция с типом `never` явно возвращает значение, PHP сгенерирует исключение `TypeError`. В обеих ситуациях, будь то явный или неявный возврат, это исключение применяется в момент вызова функции, а не во время ее определения.

### Читайте также

Оригинальный RFC, представляющий возвращаемый тип `never` (<https://oreil.ly/wO3zv>) в PHP 8.1.

## ГЛАВА 4

---

# Строки

[https://t.me/it\\_boooks/2](https://t.me/it_boooks/2)

Строки — это один из важнейших строительных блоков данных в PHP. Каждая строка представляет собой упорядоченную последовательность байтов. Строки могут варьироваться от понятных человеку фрагментов текста (например, Быть или не быть) до последовательностей байтов, закодированных как целые числа (например, \110\145\154\154\157\40\127\157\162\154\144\41)<sup>1</sup>. Каждый элемент данных, считываемый или записываемый PHP-приложением, представлен в виде строки.

В PHP строки обычно кодируются в виде ASCII-значений (<https://oreil.ly/Tjsyx>), однако допускается конвертировать их между ASCII и другими форматами (например, UTF-8). Строки могут содержать байты null, если это необходимо, и, по сути, ограничены только доступной PHP памятью.

Самый простой способ создать строку в PHP — использовать одинарные кавычки. Строки, заключенные в одинарные кавычки, рассматриваются как литеральные операторы: в них нет специальных символов или какой-либо интерполяции переменных. Чтобы одинарная кавычка отображалась как часть строки, вы должны ее экранировать, поставив перед ней обратную косую черту, например \'. Фактически единственны два символа, которые периодически требуется экранировать, — это сама кавычка и обратная косая черта. В примере 4.1 показаны строки, заключенные в одинарные кавычки, и соответствующие им выходные данные.



Интерполяция переменных — это практика прямого обращения к переменной по имени внутри строки, позволяющая интерпретатору заменить переменную ее значением во время выполнения. Интерполяция позволяет создавать более гибкие строки, поскольку вы можете написать одну строку, но динамически заменять часть ее содержимого в соответствии с контекстом ее расположения в коде.

---

<sup>1</sup> Эта строка — байтовое представление “Hello World!”, отформатированное в восьмеричной системе счисления.

**Пример 4.1.** Строки с одинарными кавычками

```
print 'Привет, мир!';
// Привет, мир!

print 'Чтобы вывести на экран символы \' и \\, используйте перед ними косую черту \\
\.';
// Чтобы вывести на экран символы ' и \, используйте перед ними косую черту \.

print 'Переменные типа $blue выводятся как литералы。';
// Переменные типа $blue выводятся как литералы。

print '\110\145\154\154\157\40\127\157\162\154\144\41';
// \110\145\154\154\157\40\127\157\162\154\144\41
```

В более сложных строках может потребоваться интерполяция переменных или ссылка на специальные символы, такие как новая строка или табуляция. Для этих случаев в PHP предусмотрены двойные кавычки и различные экранирующие последовательности, как показано в табл. 4.1.

**Таблица 4.1.** Последовательности экранирования строк с двойными кавычками

Последовательность действий	Символ	Пример
\n	Новая строка	"Эта строка заканчивается новой строкой \n"
\r	Возврат каретки	"Эта строка заканчивается возвратом каретки \r"
\t	Табуляция	"Много\tсвободного\tместа"
\\"	Обратная косая черта	"Избегайте символа \\"
\\$	Знак доллара	Билет в кино стоит \\$10
\"	Двойная кавычка	"Некоторые кавычки являются \"парными\""
\0 – \777	Октальное значение символа	"\120\110\120"
\x0 – \xFF	Шестнадцатеричное значение символа	"\x50\x48\x50"

За исключением специальных символов, которые явно экранированы обратным слешем, PHP автоматически подставляет значение любой переменной, переданной в строке с двойными кавычками. Кроме того, PHP будет интерполировать целые выражения в строке с двойными кавычками, если они заключены в фигурные скобки {}, и рассматривать их как переменную. Пример 4.2 показывает, как переменные, сложные или иные, обрабатываются в строках с двойными кавычками.

**Пример 4.2.** Интерполяция переменных в строках с двойными кавычками

```
print "Значение \$var равно $var"; ❶
print "Свойства объектов тоже могут быть интерполированы. {$obj->value}";
print "Выводит значение переменной, возвращенной функцией getVar(): ${getVar()}"; ❸
```

❶ Первая ссылка на \$var экранируется, но вторая будет заменена ее реальным значением. Если \$var = 'apple', в строке будет выведено Значение \$var равно apple.

❷ Использование фигурных скобок позволяет напрямую ссылаться на свойства объекта внутри строки с двойными кавычками, как если бы эти свойства были локально определенными переменными.

❸ Если предположить, что getVar() возвращает имя определенной переменной, то эта строка одновременно выполнит функцию и выведет значение, присвоенное этой переменной.

Строки как с одинарными, так и с двойными кавычками выступают в виде фрагмента текста. Однако часто в программе требуется представить несколько строчек текста (или закодированного двоичного кода) как тип *string*. В подобной ситуации лучшим инструментом, имеющимся в распоряжении разработчика, является Heredoc.

Heredoc — это литеральный блок текста, который начинается с трех угловых скобок (оператор <<<), за которым следует именованный идентификатор, а после него — новая строка. Каждая последующая строчка текста (включая символы новой строки) является частью строки *string*, вплоть до полностью независимой строчки, не содержащей ничего, кроме именованного идентификатора Heredoc и точки с запятой. Пример 4.3 иллюстрирует, как Heredoc может выглядеть в коде.



Идентификаторы, используемые для Heredoc, не обязательно писать в верхнем регистре. Тем не менее это делается, чтобы отличить их от текстового определения строки.

**Пример 4.3.** Определение строки с использованием синтаксиса Heredoc

```
$poem = <<<POEM
Быть или не быть –
вот в чем вопрос
POEM;
```

Heredoc работает так же, как строки с двойными кавычками, и допускает интерполяцию переменных (или специальных символов, таких как экранированные шестнадцатеричные числа) внутри них. Это бывает полезно при кодировании блоков HTML в приложении, так как переменные можно использовать для придания строкам динаминости.

Однако бывают ситуации, когда необходим строковый литерал, а не что-то открытое для интерполяции переменных. В этом вам поможет Nowdoc — альтернатива PHP синтаксису Heredoc для создания строк в одинарных кавычках. Nowdoc выглядит почти так же, как Heredoc, за исключением того, что сам идентификатор заключен в одинарные кавычки, как в примере 4.4.

#### Пример 4.4. Определение строки с использованием синтаксиса Nowdoc

```
$poem = <<< 'POEM'  
Быть или не быть –  
вот в чем вопрос  
POEM;
```

Внутри блоков Heredoc и Nowdoc можно использовать как одинарные, так и двойные кавычки без дополнительного экранирования, однако Nowdoc не будет интерполировать или динамически заменять какие-либо значения, независимо от того, экранированы они или нет.

Следующие примеры призваны проиллюстрировать пользу от применения строк в PHP и то, какие задачи они способны решать.

## 4.1. Доступ к подстрокам в более крупной строке

### Задача

Вы хотите определить, включает ли строка определенную подстроку. Например, вам нужно узнать, содержит ли URL-адрес текст `/secret/`.

### Решение

Используйте функцию `strpos()`:

```
if (strpos($url, '/secret/') !== false) {  
    // Обнаружен секретный фрагмент, запустите дополнительную логику  
    // ...  
}
```

### Обсуждение

Функция PHP `strpos()` сканирует заданную строку и определяет начальную позицию первого вхождения искомой подстроки. Эта функция буквально ищет иголку в стоге сена, поскольку аргументы функции имеют имена `$haystack` и `$needle` (стог сена и иголка соответственно). Если подстрока (`$needle`) не найдена, функция возвращает `false`.

В этом случае важно использовать строгое сравнение, так как `strpos()` вернет 0, если подстрока находится в самом начале исследуемой строки. В рецепте 2.3 говорилось, что при сравнении значений только с двумя знаками равенства будет сделана попытка перестроить типы, преобразовав целое число 0 в булево `false`; чтобы избежать путаницы, всегда используйте операторы строгого сравнения (`==` для проверки равенства или `!=` для неравенства).

Если `$needle` встречается в строке неоднократно, функция `strpos()` возвращает позицию только первого вхождения. Чтобы искать дополнительные вхождения, добавьте смещение позиции в качестве третьего параметра к вызову функции, как показано в примере 4.5. Определение смещения также позволяет искать в оставшейся части строки следующее вхождение подстроки.

#### Пример 4.5. Подсчет всех вхождений подстроки

```
function count_occurrences($haystack, $needle)
{
    $occurrences = 0;
    $offset = 0;
    $pos = 0; ①

    do {
        $pos = strpos($haystack, $needle, $offset);

        if ($pos !== false) { ②
            $occurrences += 1;
            $offset = $pos + 1; ③
        }
    } while ($pos !== false); ④

    return $occurrences;
}

$str = 'Дятел дуб долбил, долбил, продалбливал, да не продолбил и не выдолбил';

print count_occurrences($str, 'долбил'); // 4
print count_occurrences($str, 'рыба'); // 0
```

① Все переменные изначально установлены в 0, чтобы отслеживать появление новых строк.

② Вхождение будет засчитано только в том случае, если строка найдена.

③ Если строка найдена, смещение обновляется, но при этом увеличивается на 1, чтобы не засчитывать повторно уже найденные вхождения.

④ Как только последнее вхождение целевой подстроки будет достигнуто, цикл завершится и возвратит число, равное общему количеству вхождений.

## Читайте также

Документация PHP по функции `strpos()` (<https://oreil.ly/w9Od4>).

# 4.2. Извлечение одной строки из другой

## Задача

Вам нужно извлечь небольшую строку из гораздо большей строки, например доменное имя из адреса электронной почты.

## Решение

Используйте `substr()`, чтобы выбрать часть строки, которую необходимо извлечь:

```
$string = 'eric.mann@cookbook.php';
$start = strpos($string, '@');

$domain = substr($string, $start + 1);
```

## Обсуждение

Функция PHP `substr()` возвращает часть заданной строки, основываясь на начальном смещении (второй параметр) установленной длины. Полная сигнатура функции выглядит следующим образом:

```
function substr(string $string, int $offset, ?int $length = null): string
```

Если параметр `$length` опущен, `substr()` вернет весь оставшийся участок строки. Если параметр `$offset` больше длины входной строки, возвращается пустая строка.

Вы также можете указать отрицательное смещение, чтобы вернуть подмножество, начиная с конца строки, а не с начала, как показано в примере 4.6.

### Пример 4.6. Подстрока с отрицательным смещением

```
$substring = substr('phpcookbook', -3); ❶
$substring = substr('phpcookbook', -2); ❷
$substring = substr('phpcookbook', -8, 4); ❸
```

❶ Возвращает `ook` (три последних символа).

❷ Возвращает `ok` (два последних символа).

❸ Возвращает `cook` (средние четыре символа).

Следует упомянуть о других случаях, связанных со смещением и длиной строки при использовании `substr()`. Бывает, что смещение начинается внутри строки, а `$length` — выходит за ее пределы. PHP улавливает это несоответствие и просто возвращает оставшуюся часть исходной строки, даже если конечный результат меньше заданной длины. В примере 4.7 продемонстрированы некоторые потенциальные результаты работы функции `substr()` при различных длинах.

#### Пример 4.7. Различные длины подстрок

```
$substring = substr('Four score and twenty', 11, 3); ❶
$substring = substr('Four score and twenty', 99, 3); ❷
$substring = substr('Four score and twenty', 20, 3); ❸
```

- ❶ Возвращает `and`.
- ❷ Возвращает пустую строку.
- ❸ Возвращает `y`.

Другой случай — отрицательная длина `$length`, передаваемая в функцию. При запросе подстроки с отрицательной длиной PHP удалит это количество символов из возвращаемой подстроки, как показано в примере 4.8.

#### Пример 4.8. Подстрока с отрицательной длиной

```
$substring = substr('Four score and twenty', 5); ❶
$substring = substr('Four score and twenty', 5, -11); ❷
```

- ❶ Возвращает `score` и `twenty`.
- ❷ Возвращает `score`.

## Читайте также

Документация PHP для функций `substr()` ([https://oreil.ly/z\\_w10](https://oreil.ly/z_w10)) и `strpos()` (<https://oreil.ly/NWcWJ>).

## 4.3. Замена части строки

### Задача

Вы хотите заменить одну часть строки другой строкой. Например, вам требуется замаскировать все цифры номера телефона, кроме последних четырех, прежде чем выводить его на экран.

## Решение

Используйте функцию `substr_replace()`, чтобы заменить компонент существующей строки, основываясь на его позиции:

```
$string = '555-123-4567';
$replace = 'xxx-xxx'

$obfuscated = substr_replace($string, $replace, 0, strlen($replace));
// xxx-xxx-4567
```

## Обсуждение

Функция PHP `substr_replace()`, подобно `substr()`, работает с частью строки, заданной целочисленным смещением определенной длины. В примере 4.9 продемонстрирована полная сигнатура функции.

### Пример 4.9. Полная сигнатурата функции `substr_replace()`

```
function substr_replace(
    array|string $string,
    array|string $replace,
    array|int $offset,
    array|int|null $length = null
): string
```

В отличие от своего аналога `substr()`, функция `substr_replace()` может работать как с отдельными строками, так и с коллекциями строк. Если передать массив строк со скалярными значениями `$replace` и `$offset`, то функция выполнит замену для каждой строки, как показано в примере 4.10.

### Пример 4.10. Замена нескольких подстрок одновременно

```
$phones = [
    '555-555-5555',
    '555-123-1234',
    '555-991-9955'
];

$obfuscated = substr_replace($phones, 'xxx-xxx', 0, 7);

// xxx-xxx-5555
// xxx-xxx-1234
// xxx-xxx-9955
```

В целом разработчики имеют большую свободу действий с параметрами этой функции. Как и в случае с `substr()`, справедливо следующее:

- если значение `$offset` отрицательное, замена начинается с указанного количества символов с конца строки;
- значение `$length` может быть отрицательным, представляя собой количество символов с конца строки, где следует остановить замену;
- если `$length` равно `null`, то внутренне оно станет равным длине входной строки;
- если `$length` равно 0, `$replace` будет вставлено в строку согласно `$offset`, а замена не будет произведена вообще.

Наконец, если `$string` — это массив, все остальные параметры также могут быть заданы в виде массивов. Каждый элемент будет представлять собой параметр строки в той же позиции в `$string`, как показано в примере 4.11.

#### Пример 4.11. Замена нескольких подстрок на параметры массива

```
$phones = [  
    '555-555-5555',  
    '555-123-1234',  
    '555-991-9955'  
];  
  
$offsets = [0, 0, 4];  
  
$replace = [  
    'xxx-xxx',  
    'xxx-xxx',  
    'xxx-xxx'  
];  
  
$lengths = [7, 7, 8];  
  
$obfuscated = substr_replace($phones, $replace, $offsets, $lengths);  
  
// xxx-xxx-5555  
// xxx-xxx-1234  
// 555-xxx-xxx
```



Необязательно, чтобы массивы, передаваемые в `$string`, `$replace`, `$offset` и `$length`, были одинакового размера. PHP не выдаст ошибку или предупреждение, если вы передадите массивы разной длины. Однако это приведет к неожиданному результату во время операции замены. Например, к усечению строки вместо замены ее содержимого. Неплохо бы проверить, что размеры каждого из этих четырех массивов совпадают.

Функция `substr_replace()` удобна, если вы точно знаете, где нужно заменить символы в строке. Однако бывают ситуации, что не известна позиция подстроки, которую требуется пересмотреть, но вы хотите заменить вхождения определенной подстроки. В таких случаях следует использовать `str_replace()` или `str_ireplace()`.

Эти две функции выполняют поиск в заданной строке, чтобы найти вхождение (или множество вхождений) указанной подстроки и заменить ее на что-то другое. Функции идентичны по шаблону вызова, но дополнительная `i` в `str_ireplace()` говорит о том, что поиск ведется без учета регистра. Пример 4.12 иллюстрирует работу обеих функций.

#### Пример 4.12. Поиск и замена в строке

```
$string = 'Орел на горе, перо на орле. Гора под орлом, орел под пером';

$beaver = str_replace('орел', 'бобер', $string); ❶
$ibeaver = str_ireplace('орел', 'бобер', $string); ❷
```

❶ Орел на горе, перо на орле. Гора под орлом, бобер под пером.

❷ Бобер на горе, перо на орле. Гора под орлом, бобер под пером.

И `str_replace()`, и `str_ireplace()` принимают дополнительный необязательный параметр `$count`, который передается по ссылке. Если он указан, эта переменная будет обновлена количеством замен, выполненных функцией. В примере 4.12 возвращаемое значение было бы 1 и 2 соответственно из-за написания «Орел» с заглавной буквы.

## Читайте также

Документация PHP по `substr_replace()` (<https://oreil.ly/-BSkA>), `str_replace()` (<https://oreil.ly/Vm7KH>) и `str_ireplace()` (<https://oreil.ly/8P46w>).

## 4.4. Обработка строки по одному байту за раз

### Задача

Вам нужно обработать строку однобайтовых символов от начала до конца, по одному символу за раз.

### Решение

Перебирайте каждый символ строки как в массиве. В примере 4.13 подсчитывается количество заглавных букв в строке.

**Пример 4.13.** Подсчет заглавных символов в строке

```
$capitals = 0;

$string = 'Сияй, надежда, луч лия, Да на краю воскликну бездны: Жив Бог! – Жива
душа моя!';
for ($i = 0; $i < strlen($string); $i++) {
    if (ctype_upper($string[$i])) {
        $capitals += 1;
    }
}

// $capitals = 5
```

## Обсуждение

Так как в PHP строки не являются массивами, их нельзя перебирать в цикле напрямую. Однако вы можете ссылаться на определенные символы в строке по их целочисленному смещению (начиная с 0) или даже по отрицательному смещению, чтобы начать с конца строки.

Тем не менее доступ к массивам не ограничивается только чтением. У вас также есть возможность заменить один символ в строке, основываясь на его позиции, как показано в примере 4.14.

**Пример 4.14.** Замена одного символа в строке

```
$string = 'Сегодня утром я покормил своего кота';
$string[34] = 'и';

// Сегодня утром я покормил своего кита
```

Можно также напрямую преобразовать строку в массив с помощью функции `str_split()` (<https://oreil.ly/eNxaf>), а затем перебирать все элементы полученного массива. В примере 4.15 показано, как это будет работать.

**Пример 4.15.** Преобразование строки в массив напрямую

```
$capitals = 0;

$string = ' Сияй, надежда, луч лия, Да на краю воскликну бездны: Жив Бог! – Жива
душа моя!';
$array = str_split($string);
foreach ($array as $char) {
    if (ctype_upper($char)) {
```

```
    $capitals += 1;
}
}

// $capitals = 5
```

Код из примера 4.15 имеет недостатки: PHP теперь придется хранить две копии данных — исходную строку и результирующий массив. Это не проблема при работе с небольшими строками, как в рассматриваемом примере, однако, если же ваши строки представляют собой целые файлы на диске, вы быстро исчерпаете доступную PHP память.

PHP облегчает доступ к отдельным байтам (символам) в строке без изменения типа данных. Разделение строки на массив может быть избыточным, если вам не нужен массив символов. Пример 4.16 представляет собой переосмысление примера 4.15, где вместо прямого подсчета заглавных букв в строке используется техника сокращения массива.

#### Пример 4.16. Подсчет заглавных букв в строке с помощью сокращения массива

```
$str = 'Сияй, надежда, луч лия, Да на краю воскликну бездны: Жив Бог! – Жива душа  
моя!';
```

```
$caps = array_reduce(str_split($str), fn($c, $i) => ctype_upper($i) ? $c+1: $c, 0);
```



Хотя примеры 4.15 и 4.16 функционально эквивалентны, второй более лаконичен и, следовательно, более сложен для понимания. Конечно, соблазн преобразовать сложную логику в односторочную функцию велик, но излишний рефакторинг кода ради краткости может быть опасен. Код будет выглядеть элегантно, но со временем его станет сложнее поддерживать.

Способ, представленный в примере 4.16, функционально точен, но все равно требует разбиения строки на массив. Это экономит несколько строк кода в вашей программе, но в любом случае приводит к созданию второй копии данных. Как уже упоминалось, если строки, над которыми вы выполняете итерацию, велики (например, массивные бинарные файлы), такая операция быстро израсходует всю память, доступную PHP.

## Читайте также

Документация PHP по доступу к строкам и их изменению (<https://oreil.ly/8MOWh>), а также документация по `ctype_upper()` (<https://oreil.ly/bQctH>).

## 4.5. Генерация случайных строк

### Задача

Вы хотите сгенерировать строку случайных символов.

### Решение

Используйте встроенную в PHP функцию `random_int()`:

```
function random_string($length = 16)
{
    $characters = '0123456789abcdefghijklmnopqrstuvwxyz';

    $string = '';
    while (strlen($string) < $length) {
        $string .= $characters[random_int(0, strlen($characters) - 1)];
    }
    return $string;
}
```

### Обсуждение

PHP предоставляет функции для генерации криптографически безопасных псевдо-случайных чисел и байтов. В языке нет встроенной функции для создания случайного, понятного человеку текста, но базовые возможности PHP позволяют продуцировать такие строки случайного текста с помощью списков, состоящих из понятных человеку символов, как показано в примере выше.



Криптографически безопасный генератор псевдослучайных чисел — это функция, которая возвращает числа без различимого или предсказуемого шаблона. Даже криминалистическая экспертиза примет выход криптографически безопасного генератора за беспорядочный набор символов.

Действенным и потенциально более простым методом получения случайных строк являются применение функции PHP `random_bytes()` и кодирование двоичного вывода в виде ASCII-текста. Пример 4.17 иллюстрирует два возможных метода использования случайных байтов в качестве строки.

#### Пример 4.17. Создание строки случайных байтов

```
$string = random_bytes(16); ①  
  
$hex = bin2hex($string); ②  
$base64 = base64_encode($string); ③
```

- ❶ Поскольку строка двоичных байтов кодируется в другом формате, количество сгенерированных байтов не будет соответствовать длине конечной строки.
- ❷ Кодировка случайной строки в шестнадцатеричном формате. Обратите внимание, что этот формат удвоит длину строки: 16 байт эквивалентны 32 шестнадцатеричным символам.
- ❸ Использование кодировки Base64 (<https://oreil.ly/NsyVs>) для преобразования необработанных байтов в читаемые символы. Имейте в виду, что данный формат увеличивает длину строки на 33–36 %.

## Читайте также

Документация PHP по функциям `random_int()` (<https://oreil.ly/g3gAR>) и `random_bytes()` (<https://oreil.ly/2Zbio>). Читайте также рецепт 5.4 о генерации случайных чисел.

# 4.6. Интерполяция переменных в строке

## Задача

Вы хотите включить динамическое содержимое в статическую строку.

## Решение

Оберните строку в двойные кавычки и поместите в нее переменную, свойство объекта или даже вызов функции/метода:

```
echo "На улице {$POST['кошки']} кошки и {$POST['собаки']} собаки.";
echo "Длина вашего имени пользователя составляет {strlen($username)} символов.";
echo "Автомобиль окрашен в {$car->color}.";
```

## Обсуждение

Строки с двойными кавычками позволяют использовать в качестве литералов сложные, динамические значения. Любое слово, начинающееся с символа \$, интерпретируется как имя переменной, если только этот символ не экранирован<sup>1</sup>.

Хотя в примере из «Решения» динамическое содержимое заключено в фигурные скобки, в PHP это не является обязательным условием. Простые переменные

<sup>1</sup> См. табл. 4.1, чтобы узнать больше о двухсимвольных последовательностях экранирования.

можно записать в строке с двойными кавычками как есть, и они будут правильно интерполированы. Однако более сложные последовательности принимают трудночитаемый вид без фигурных скобок. Чтобы строка выглядела более разборчивой, всегда заключайте в скобки любое значение, которое вы хотите интерполировать.

К сожалению, интерполяция строк имеет свои ограничения. Приведенный в «Решении» способ извлечения данных из суперглобального массива `$_POST` и их вставки непосредственно в строку является потенциально опасным, так как содержимое генерируется непосредственно пользователем, а строка может быть использована в уязвимых областях. На самом деле подобная интерполяция строк является одним из крупнейших векторов для атак, основанных на принципе внедрения вредоносного кода — инъекциях (injection attack).



Инъекции — это тип атак, когда злоумышленник передает (внедряет) исполняемый или иной вредоносный код в ваше приложение, чтобы оно вело себя некорректно. Более изощренные способы защиты от этого семейства атак рассматриваются в главе 9.

Чтобы защитить вашу строку от потенциально опасного пользовательского ввода, рекомендуется использовать функцию PHP `sprintf()`, которая возвращает отформатированную строку. В примере 4.18 часть примера из «Решения» переписана для защиты от вредоносных данных `$_POST`.

#### Пример 4.18. Получение интерполированной строки

```
echo sprintf('Здесь %d кошек и %d собак.', $_POST['кошек'], $_POST['собак']);
```

Форматирование строк — основная форма санации ввода в PHP. В примере 4.18 явно предполагается, что данные `$_POST` являются числовыми. Токены `%d` в такой строке будут заменены данными, предоставленными пользователем, но PHP явно преобразует данные в целые числа во время замены.

Если бы, например, эта строка помещалась в базу данных, форматирование защищило бы от возможных атак, основанных на внедрении в запрос произвольного SQL-кода. Более полные методы фильтрации и санации пользовательского ввода описаны в главе 9.

## Читайте также

Документация PHP по разбору переменных (<https://oreil.ly/CAj-J>) в двойных кавычках и Heredoc, а также документация по функции `sprintf()` (<https://oreil.ly/DMAg6>).

## 4.7. Конкатенация нескольких строк вместе

### Задача

Вам нужно создать новую строку из двух строк поменьше.

### Решение

Используйте оператор конкатенации строк PHP:

```
$first = 'Быть или не быть –';
$second = 'Вот в чем вопрос';

$line = $first . ' ' . $second;
```

### Обсуждение

В PHP символ . применяется для объединения двух строк. Этот оператор также выполняет приведение типов, чтобы убедиться, что оба значения являются строками, прежде чем они будут объединены (см. пример 4.19).

#### Пример 4.19. Конкатенация строк

```
print 'String ' . 2; ①
print 2 . ' number'; ②
print 'Boolean ' . true; ③
print 2 . 3; ④
```

① Выводит String 2.

② Выводит 2 number.

③ Выводит Boolean 1, поскольку булевые значения приводятся к целым числам, а затем к строкам.

④ Выводит 23.

Оператор конкатенации строк позволяет быстро объединить простые строки. Однако он может стать довольно громоздким, если вы используете его для объединения нескольких строк с пробелами. Рассмотрим пример 4.20, в котором происходит объединение в строку списка слов, каждое из которых разделено пробелом.

#### Пример 4.20. Конкатенация больших групп строк

```
$words = [
    'Whose',
    'woods',
```

```

'these',
'are',
'I',
'think',
'I',
'know'
];

$option1 = $words[0] . ' ' . $words[1] . ' ' . $words[2] . ' ' . $words[3] .
    ' ' . $words[4] . ' ' . $words[5] . ' ' . $words[6] .
    ' ' . $words[7]; ①

$option2 = '';
foreach ($words as $word) {
    $option2 .= ' ' . $word; ②
}
$option2 = ltrim($option2); ③

```

❶ Один из вариантов — конкатенировать каждое слово в коллекции по отдельности, разделяя их пробелами. По мере увеличения списка слов такой код быстро становится громоздким.

❷ Вместо этого можно перебрать коллекцию с помощью цикла и создать конкатенированную строку, не обращаясь к каждому элементу по отдельности.

❸ Во время выполнения цикла могут появиться лишние пробельные символы. Не забудьте удалить их.

Большие, повторяющиеся операции конкатенации лучше заменить встроенными функциями PHP, например `implode()`. Она, в частности, принимает в качестве аргументов массив данных, которые нужно объединить, и символ (или символы), выступающий в роли разделителя. Функция `implode()` возвращает конкатенированную строку.



Некоторые разработчики предпочитают использовать `join()` вместо `implode()`, так как считают, что это более описательное название для операции. На самом деле `join()` — это псевдоним `implode()`, и компилятор PHP их не различает.

Код в примере 4.21 — это переписанный пример 4.20, но с применением функции `implode()`. Обратите внимание насколько при этом упрощается операция.

#### Пример 4.21. Лаконичный подход к конкатенации строк

```
$words = [
    'Whose',
    'woods',
    'these',
```

```
'are',
'I',
'think',
'I',
'know'
];
$string = implode(' ', $words);
```

Запомните порядок параметров для `implode()`. Сначала идет разделитель строк, а затем массив, над которым вы хотите произвести итерацию. В ранних версиях PHP (до версии 8.0) параметры можно было указать в обратном порядке. Такое поведение (указание массива первым, а разделителя — вторым) было отменено в PHP 7.4, а начиная с PHP 8.0, оно приводит к ошибке `TypeError`.

Если вы используете библиотеку, написанную до PHP 8.0, обязательно проверьте, не использует ли она `implode()` или `join()`, прежде чем выпускать ваш проект в свет.

## Читайте также

Документация PHP по функции `implode()` (<https://oreil.ly/bGYt0>).

# 4.8. Управление двоичными данными, хранящимися в строках

## Задача

Вам требуется закодировать данные непосредственно в двоичном формате, а не в ASCII, или прочитать данные в приложении, которые были явно закодированы как двоичные.

## Решение

Используйте функцию `unpack()` для извлечения двоичных данных из строки:

```
$unpacked = unpack('S1', 'Hi'); // [1 => 26952]
```

А для записи двоичных данных в строку — функцию `pack()`:

```
$packed = pack('S13', 72, 101, 108, 108, 111, 44, 32, 119, 111,
114, 108, 100, 33); // 'Hello, world!'
```

## Обсуждение

Обе функции `pack()` и `unpack()` позволяют взаимодействовать с необработанными бинарными строками при условии, что вы знаете формат двоичной строки, с которой работаете. Первым параметром каждой функции является спецификация формата. Она определяется конкретными кодами, как показано в табл. 4.2.

**Таблица 4.2.** Коды строк двоичного формата

Код	Описание
a	Строка с null-заполнением
A	Строка с пробельным заполнением
h	Шестнадцатеричная строка, с нижнего разряда
H	Шестнадцатеричная строка, с верхнего разряда
c	Знаковый char
C	Беззнаковый char
s	Знаковый short (всегда 16 бит, машинный порядок байтов)
S	Беззнаковый short (всегда 16 бит, машинный порядок байтов)
n	Беззнаковый short (всегда 16 бит, порядок байтов big-endian)
v	Беззнаковый short (всегда 16 бит, порядок байтов little-endian)
i	Знаковый integer (машинно-зависимый размер и порядок)
I	Беззнаковый integer (машинно-зависимый размер и порядок)
l	Знаковый long (всегда 32 бита, машинный порядок байтов)
L	Беззнаковый long (всегда 32 бита, машинный порядок байтов)
N	Беззнаковый long (всегда 32 бита, порядок байтов big-endian)
V	Беззнаковый long (всегда 32 бита, порядок байтов little-endian)
q	Знаковый long long (всегда 64 бита, машинный порядок байтов)
Q	Беззнаковый long long (всегда 64 бита, машинный порядок байтов)
J	Беззнаковый long long (всегда 64 бита, порядок байтов big-endian)
P	Беззнаковый long long (всегда 64 бита, порядок байтов little-endian)
f	Float (машинно-зависимый размер и представление)

Код	Описание
g	Float (машинно-зависимый размер, порядок байтов little-endian)
G	Float (машинно-зависимый размер, порядок байтов big-endian)
d	Double (машинно-зависимый размер и представление)
e	Double (машинно-зависимый размер, порядок байтов little-endian)
E	Double (машинно-зависимый размер, порядок байтов big-endian)
x	Байт null
X	Резервирование одного байта
Z	Строка (string) с null-заполнением
@	null-заполнение до абсолютной позиции

При определении форматированных строк можно указать каждый тип байта отдельно или применить необязательный повторяющийся символ. В примерах выше количество байтов явно указано в виде целого числа. Допускается также использовать звездочку (\*), чтобы указать, что тип байта повторяется до конца строки, как показано ниже:

```
$unpacked = unpack('S*', 'Hi'); // [1 => 26952]
$packed = pack('S*', 72, 101, 108, 108, 111, 44, 32, 119, 111,
               114, 108, 100, 33); // 'Hello, world!'
```

Способность PHP конвертировать различные типы байтовых кодировок с помощью функции unpack() также предоставляет простой метод преобразования символов ASCII в их двоичный эквивалент. Функция ord() возвращает значение конкретного символа, но для этого нужно перебирать каждый символ в строке, если вы хотите распаковать их поочередно, как показано в примере 4.22.

#### Пример 4.22. Получение значений символов с помощью функции ord()

```
$ascii = 'PHP Cookbook';

$chars = [];
for ($i = 0; $i < strlen($ascii); $i++) {
    $chars[] = ord($ascii[$i]);
}

var_dump($chars);
```

Благодаря функции unpack() вам не нужно явно перебирать символы в строке. Форматированный символ с ссылается на знаковый char, а C — на беззнаковый.

Вместо того чтобы строить цикл, допускается использовать unpack() напрямую, чтобы получить эквивалентный результат:

```
$ascii = 'PHP Cookbook';
$chars = unpack('C*', $ascii);

var_dump($chars);
```

Как в предыдущем примере unpack(), так и в исходной реализации цикла, в примере 4.22 получается следующий массив:

```
array(12) {
    [1]=>
    int(80)
    [2]=>
    int(72)
    [3]=>
    int(80)
    [4]=>
    int(32)
    [5]=>
    int(67)
    [6]=>
    int(111)
    [7]=>
    int(111)
    [8]=>
    int(107)
    [9]=>
    int(98)
    [10]=>
    int(111)
    [11]=>
    int(111)
    [12]=>
    int(107)
}
```

## Читайте также

Документация PHP по pack() (<https://oreil.ly/0iieT>) и unpack() ([https://oreil.ly/Un\\_aD](https://oreil.ly/Un_aD)).

## ГЛАВА 5

# Числа

Другой фундаментальный строительный блок данных в PHP — это числа. В окружающем нас мире легко встретить различные типы чисел: номер страницы в книге, время на часах, количество шагов на фитнес-браслете. Одни числа неимоверно большие, другие — поразительно маленькие. Они могут быть целыми, дробными или иррациональными, как, например,  $\pi$ .

В PHP числа представляются в одном из двух форматов: как целые числа (тип `int`) или как числа с плавающей точкой (тип `float`). Оба этих типа очень гибкие, но диапазон значений, которые вы можете использовать, зависит от архитектуры процессора вашей системы: 32-битная система имеет более строгие ограничения, чем 64-битная.

PHP определяет несколько констант, которые позволяют программам понимать доступный диапазон чисел в системе. Учитывая, что возможности PHP сильно отличаются в зависимости от версии компилятора (для 32 или 64 бит), разумнее использовать константы, показанные в табл. 5.1, а не подбирать значения в программе. Всегда безопаснее полагаться на настройки операционной системы и языка по умолчанию.

**Таблица 5.1.** Постоянные числовые значения в PHP

Константа	Описание
<code>PHP_INT_MAX</code>	Наибольшее целочисленное значение, поддерживаемое PHP. На 32-битных системах это число равно 2147483647, на 64-битных системах — 9223372036854775807
<code>PHP_INT_MIN</code>	Наименьшее целочисленное значение, поддерживаемое PHP. На 32-битных системах это число равно -2147483648, на 64-битных системах — -9223372036854775808
<code>PHP_INT_SIZE</code>	Размер целых чисел в байтах для данной сборки PHP
<code>PHP_FLOAT_DIG</code>	Количество цифр, которые можно округлить в <code>float</code> и обратно без потери точности

Продолжение ↗

**Таблица 5.1 (продолжение)**

Константа	Описание
PHP_FLOAT_EPSILON	Наименьшее представимое положительное число $x$ , где $x + 1.0 \neq 1.0$
PHP_FLOAT_MIN	Наименьшее представимое положительное число с плавающей точкой
PHP_FLOAT_MAX	Наибольшее представимое число с плавающей точкой
-PHP_FLOAT_MAX	Не отдельная константа, а способ представления наименьшего отрицательного числа с плавающей точкой

Досадное ограничение системы счисления в PHP заключается в том, что вам нужно использовать расширения, такие как BCMath (<https://oreil.ly/qFeO3>) или GNU Multiple Precision Arithmetic Library (GMP) (<https://oreil.ly/u9Mbf>), чтобы выполнять операции над числами на базе операционной системы. С GMP мы подробнее познакомимся в рецепте 5.10.

Некоторые разработчики сталкиваются со множеством проблем при работе с числами в PHP. Следующие рецепты помогут их решить.

## 5.1. Проверка числа в переменной

### Задача

Вы хотите проверить, содержит ли переменная число, даже если она явно указана как строка.

### Решение

Используйте функцию `is_numeric()`, чтобы проверить, может ли переменная быть успешно приведена к числовому значению, например:

```
$candidates = [
    22,
    '15',
    '12.7',
    0.662,
    'бесконечность',
    INF,
    0xDEADBEEF,
    '10e10',
    '15 яблок',
    '2,500'
];
```

```
foreach ($candidates as $candidate) {  
    $numeric = is_numeric($candidate) ? 'является' : 'НЕ является';  
  
    echo "Значение '{$candidate}' {$numeric} числовым." . PHP_EOL;  
}
```

При выполнении кода в консоли будет выведено следующее:

```
Значение '22' является числовым.  
Значение '15' является числовым.  
Значение '12.7' является числовым.  
Значение '0.662' является числовым.  
Значение 'бесконечность' НЕ является числовым.  
Значение 'INF' является числовым.  
Значение '3735928559' является числовым.  
Значение '10e10' является числовым.  
Значение '15 яблок' НЕ является числовым.  
Значение '2,500' НЕ является числовым.
```

## Обсуждение

По своей сути PHP является динамически типизированным языком. Вы с легкостью можете заменить строки на целые числа (и наоборот), и PHP попытается спрогнозировать ваши намерения, динамически преобразуя значения из одного типа в другой по мере необходимости. Хотя вы можете (и, вероятно, должны) обеспечить строгую типизацию, как обсуждалось в рецепте 3.4, зачастую вам придется явно кодировать числа как строки.

В таких ситуациях вы потеряете возможность определять числовые строки с помощью системы типов PHP. Переменная, переданная в функцию в качестве строки `string`, будет недопустимой для математических операций без явного приведения к числовому типу (`int` или `float`). К сожалению, не все строки, содержащие числа, являются числовыми.

Строка `15 яблок` из примера выше содержит число, но не является числовой. Страна `10e10` содержит нечисловые символы, но является допустимым числовым представлением.

Разницу между строками, содержащими числа, и числовыми строками легче заметить на примере пользовательской реализации встроенной в PHP функции `is_numeric()` (см. пример 5.1).

### Пример 5.1. Пользовательская реализация функции `is_numeric()`

```
function string_is_numeric(string $test): bool  
{  
    return $test === (string) floatval($test);  
}
```

Переписанный в примере 5.1 массив `$candidates` из «Решения» точно проверит числовые строки везде, кроме литеральной константы `INF` и сокращения экспоненты `10e10`. Это происходит потому, что функция `floatval()` полностью удаляет из строки любые нечисловые символы, преобразуя их в число с плавающей точкой, перед приведением к строке с помощью `(string)`<sup>1</sup>.

Пользовательская реализация подходит не для всех ситуаций, поэтому безопаснее использовать встроенные функции. Цель функции `is_numeric()` — указать, можно ли надежным способом привести данную строку к числовому типу без потери информации.

## Читайте также

Документация PHP для `is_numeric()` (<https://oreil.ly/jTGcF>).

## 5.2. Сравнение чисел с плавающей точкой

### Задача

Вы хотите проверить равенство двух чисел с плавающей точкой.

### Решение

Определите наибольшую допустимую разницу между двумя числами и оцените ее по соответствующей границе погрешности, называемой `epsilon`:

```
$first = 22.19348234;  
$second = 22.19348230;  
  
$epsilon = 0,000001;  
  
$equal = abs($first - $second) < $epsilon; // true
```

### Обсуждение

Операции над числами с плавающей точкой в современных компьютерах не так точны из-за способа внутреннего представления чисел в компьютерах. Различные точные расчеты, совершаемые вами вручную, могут запутать машины, на которые вы полагаетесь.

---

<sup>1</sup> Подробнее о приведении типов читайте в разделе «Приведение типов» главы 2.

Например, разность двух чисел  $1 - 0.83$ , очевидно, равна  $0.17$ , что довольно просто сосчитать в уме или на бумаге. Однако если попросить компьютер сделать это, то получится странный результат, как показано в примере 5.2.

### Пример 5.2. Операция вычитания с числами с плавающей точкой

```
$a = 0.17;  
$b = 1 - 0.83;  
  
var_dump($a == $b); ❶  
var_dump($a); ❷  
var_dump($b); ❸  
  
❶ bool(false)  
❷ float(0.17)  
❸ float(0.1700000000000004)
```

Когда речь идет об арифметике с числами с плавающей точкой, лучшее, что могут сделать компьютеры, — это вывести приблизительный результат в пределах допустимой погрешности. В итоге сопоставление полученного значения с ожидаемым требует явного определения этой погрешности (`epsilon`) и сравнения с ней, а не с точным значением.

Вместо того чтобы применять один из операторов сравнения (двойной или тройной знак «равно»), достаточно определить функцию для проверки относительного равенства двух чисел с плавающей точкой, как показано в примере 5.3.

### Пример 5.3. Проверка равенства чисел с плавающей точкой

```
function float_equality(float $epsilon): callable  
{  
    return function(float $a, float $b) use ($epsilon): bool  
    {  
        return abs($a - $b) < $epsilon;  
    };  
}  
  
$tight_equality = float_equality(0.000001);  
$loose_equality = float_equality(0.01);  
  
var_dump($tight_equality(1.152, 1.152001)); ❶  
var_dump($tight_equality(0.234, 0.2345)); ❷  
var_dump($tight_equality(0.234, 0.244)); ❸  
var_dump($loose_equality(1.152, 1.152001)); ❹  
var_dump($loose_equality(0.234, 0.2345)); ❺  
var_dump($loose_equality(0.234, 0.244)); ❻
```

❶ `bool(false)`

❷ `bool(false)`

❸ `bool(false)`

❹ `bool(true)`

❺ `bool(true)`

❻ `bool(true)`

## Читайте также

Документация PHP по числам с плавающей точкой ([https://oreil.ly/-311\\_](https://oreil.ly/-311_)).

## 5.3. Округление чисел с плавающей точкой

### Задача

Вы хотите округлить число с плавающей точкой либо до фиксированного количества десятичных знаков, либо до целого числа.

### Решение

Чтобы округлить число с плавающей точкой, используйте функцию `round()`, указывая количество десятичных знаков:

```
$number = round(15.31415, 1);  
// 15.3
```

Чтобы явно округлить число до целого в большую сторону, используйте `ceil()`:

```
$number = ceil(15.3);  
// 16
```

Чтобы явно округлить число до целого в меньшую сторону, используйте `floor()`:

```
$number = floor(15.3);  
// 15
```

### Обсуждение

Все три функции, упомянутые выше, — `round()`, `ceil()` и `floor()` — предназначены для работы с любым числовым значением, но после выполнения они возвращают

результат типа `float`. По умолчанию `round()` округляет до целого числа, но все равно возвращает тип `float`.

Чтобы преобразовать `float` в `int`, оберните саму функцию в `intval()`.

Стоит также отметить, что `round()` всегда округляет вводимое число в сторону от 0, если его дробная часть равна или больше 0,5 по модулю. То есть числа типа 1,4 будут округляться в меньшую сторону, а 1,5 — в большую. Это справедливо и для отрицательных чисел: -1,4 округляется в сторону 0 до -1, а -1,5 — в сторону от 0 до -2.

Вы можете изменить поведение `round()`, передав необязательный третий аргумент (или используя именованные параметры, как показано в рецепте 3.3), чтобы указать режим округления. Этот аргумент принимает одну из четырех констант по умолчанию, определенных PHP, как перечислено в табл. 5.2.

**Таблица 5.2.** Константы режима округления

Константа	Описание
<code>PHP_ROUND_HALF_UP</code>	Округляет положительное число в большую сторону, а отрицательное — в меньшую, превращая 1,5 в 2 и -1,5 в -2 (стремится от нуля)
<code>PHP_ROUND_HALF_DOWN</code>	Округляет положительное число в меньшую сторону, а отрицательное — в большую, превращая 1,5 в 1 и -1,5 в -1 (стремится к нулю)
<code>PHP_ROUND_HALF_EVEN</code>	Округляет число до ближайшего четного значения, превращая 1,5 и 2,5 в 2
<code>PHP_ROUND_HALF_ODD</code>	Округляет число до ближайшего нечетного значения, превращая 1,5 в 1 и 2,5 в 3

Пример 5.4 иллюстрирует эффект каждой константы режима округления при применении к одним и тем же числам.

**Пример 5.4.** Округление чисел с плавающей точкой в PHP с использованием различных режимов

```
echo 'Округление на 1,5' . PHP_EOL;
var_dump(round(1.5, mode: PHP_ROUND_HALF_UP));
var_dump(round(1.5, mode: PHP_ROUND_HALF_DOWN));
var_dump(round(1.5, mode: PHP_ROUND_HALF_EVEN));
var_dump(round(1.5, mode: PHP_ROUND_HALF_ODD));

echo 'Округление на 2,5' . PHP_EOL;
var_dump(round(2.5, mode: PHP_ROUND_HALF_UP));
var_dump(round(2.5, mode: PHP_ROUND_HALF_DOWN));
var_dump(round(2.5, mode: PHP_ROUND_HALF_EVEN));
var_dump(round(2.5, mode: PHP_ROUND_HALF_ODD));
```

В результате выполнения предыдущего примера на консоль будет выведено следующее:

```
Округление на 1,5
float(2)
float(1)
float(2)
float(1)
Округление на 2,5
float(3)
float(2)
float(2)
float(3)
```

## Читайте также

Документация PHP по числам с плавающей точкой (<https://oreil.ly/ONHjD>), функции `round()` (<https://oreil.ly/010CB>), `ceil()` (<https://oreil.ly/i5Rpy>) и функция `floor()` (<https://oreil.ly/VAZ6t>).

## 5.4. Генерация случайных чисел

### Задача

Вы хотите генерировать случайные целые числа в определенном диапазоне.

### Решение

Используйте `random_int()` следующим образом:

```
// Случайное целое число в промежутке от 10 до 225 включительно
$random_number = random_int(10, 225);
```

### Обсуждение

Когда идет речь о математической случайности, чаще всего имеют в виду совершенно непредсказуемую последовательность. В таких ситуациях вы можете положиться на криптографически безопасные генераторы псевдослучайных чисел, встроенные в саму машину. Функция PHP `random_int()` использует эти генераторы чисел на уровне операционной системы, а не реализует собственный алгоритм.

В Windows PHP применит либо `CryptGenRandom()` (<https://oreil.ly/0kVO9>), либо `Cryptography API: Next Generation (CNG)` (<https://oreil.ly/otHP9>) в зависимости от версии языка. В Linux PHP задействует системный вызов `getrandom(2)` (<https://oreil.ly/07DIE>). На любой другой платформе PHP будет опираться на системный интерфейс `/dev/urandom`. Все эти API полностью протестированы и доказали свою криптографическую безопасность, то есть они генерируют числа с достаточной степенью случайности.



В редких ситуациях вам может потребоваться генератор случайных чисел, который выдает предсказуемую серию псевдослучайных значений. В таких случаях можно прибегнуть к помощи алгоритмических генераторов, например `Mersenne Twister`, который более подробно рассматривается в рецепте 5.5.

PHP не поддерживает метод создания случайного числа с плавающей точкой (то есть выбор случайного десятичного числа между 0 и 1). Вместо этого вы можете использовать функцию `random_int()` и свои знания о целых числах в PHP, чтобы написать собственную функцию для подобных целей (см. пример 5.5).

**Пример 5.5.** Пользовательская функция для генерации случайного числа с плавающей точкой

```
function random_float(): float
{
    return random_int(0, PHP_INT_MAX) / PHP_INT_MAX;
}
```

Эта реализация `random_float()` не имеет пределов, поскольку всегда будет генерировать число от 0 до 1 включительно. Это пригодится при создании случайных процентных значений, искусственных данных или для отбора случайно заданных выборок из массивов. Более сложная реализация может включать в себя границы, как показано в примере 5.6, но обычно достаточно возможности выбора между 0 и 1.

**Пример 5.6.** Пользовательская функция для генерации случайного числа с плавающей точкой в заданных границах

```
function random_float(int $min = 0, int $max = 1): float
{
    $rand = random_int(0, PHP_INT_MAX) / PHP_INT_MAX;

    return ($max - $min) * $rand + $min;
}
```

Это обновленное определение `random_float()` просто масштабирует исходное определение до новых пределов. Если оставить границы по умолчанию, функция сведется к изначальному определению.

## Читайте также

Документация PHP по функции `random_int()` (<https://oreil.ly/kLoas>).

# 5.5. Генерация предсказуемых случайных чисел

## Задача

Вы хотите задать случайные числа таким образом, чтобы их последовательность была одинаковой при каждом запуске.

## Решение

Используйте функцию `mt_rand()` после передачи начального значения для генератора псевдослучайных чисел (зерна) в `mt_srand()`, например:

```
function generate_sequence(int $count = 10): array
{
    $array = [];

    for ($i = 0; $i < $count; $i++) {
        $array[] = mt_rand(0, 100);
    }

    return $array;
}

mt_srand(42);
$first = generate_sequence();

mt_srand(42);
$second = generate_sequence();

print_r($first); print_r($second);
```

Оба массива в предыдущем примере будут иметь следующее содержимое:

```
Array
(
    [0] => 38
    [1] => 32
    [2] => 94
    [3] => 55
    [4] => 2
```

```
[5] => 21  
[6] => 10  
[7] => 12  
[8] => 47  
[9] => 30  
)
```

## Обсуждение

При написании любого другого примера о действительно случайных числах лучшее, что можно сделать, — это продемонстрировать, каким может быть результат. Однако если речь идет о выводе `mt_rand()`, то он будет одинаковым на любом компьютере, если вы используете одно и то же зерно.



По умолчанию PHP автоматически задает зерно `mt_rand()` случайным образом. Нет необходимости указывать собственные зерна, если только вашей целью не является детерминированный вывод функции.

Выходные данные одинаковы, потому что `mt_rand()` использует алгоритмический генератор псевдослучайных чисел под названием Вихрь Мерсенна (Mersenne Twister). Этот хорошо известный и часто используемый алгоритм, впервые представленный в 1997 году, также применяется в Ruby и Python.

Учитывая стартовое значение зерна, алгоритм создает начальное состояние, а затем генерирует кажущиеся случайными числа, выполняя над этим состоянием операцию «кручения». Преимущество такого подхода заключается в его детерминированности: при одинаковом значении зерна алгоритм будет создавать одну и ту же последовательность «случайных» чисел каждый раз.



Предсказуемость случайных чисел может быть опасна для некоторых вычислительных операций, в частности для криптографии. Сценарии использования, требующие детерминированной последовательности псевдослучайных чисел, достаточно редки, поэтому `mt_rand()` следует избегать. Если вам нужно генерировать случайные числа, используйте такие функции, как `random_int()` и `random_bytes()`.

Генерация псевдослучайной, но предсказуемой последовательности чисел может пригодиться при создании идентификаторов объектов для БД. Вы легко можете проверить правильность работы вашего кода, запустив его несколько раз и сравнив результат. Недостатком является то, что алгоритмы, подобные Вихрю Мерсенна, могут быть скомпрометированы посторонним человеком.

Учитывая достаточно длинную последовательность кажущихся случайными чисел и зная алгоритм, можно легко провернуть обратную операцию и определить исходное значение зерна. И как только злоумышленник узнает его, он сможет сгенерировать все возможные «случайные» числа, которые ваша система будет использовать в дальнейшем.

## Читайте также

Документация PHP по `mt_rand()` ([https://oreil.ly/niU\\_q](https://oreil.ly/niU_q)) и `mt_srand()` (<https://oreil.ly/xSa53>).

## 5.6. Генерация взвешенных случайных чисел

### Задача

Вы хотите сгенерировать случайные числа, чтобы произвольным образом выбрать некий предмет из коллекции, но при этом нужно, чтобы некоторые предметы имели более высокий приоритет, чем другие. Например, вы хотите выбрать победителя конкурса, но одни участники заработали больше очков, чем другие, и должны иметь больше шансов на победу.

### Решение

Воспользуйтесь `weighted_random_choice()`, как показано в примере 5.7.

#### Пример 5.7. Реализация взвешенного случайного выбора

```
$choices = [
    'Тони' => 10,
    'Стив' => 2,
    'Питер' => 1,
    'Ванда' => 4,
    'Кэрол' => 6
];

function weighted_random_choice(array $choices): string
{
    arsort($choices);

    $total_weight = array_sum(array_values($choices));
    $selection = random_int(1, $total_weight);
```

```

$count = 0;
foreach ($choices as $choice => $weight) {
    $count += $weight;
    if ($count >= $selection) {
        return $choice;
    }
}

throw new Exception('Невозможно сделать выбор!');
}

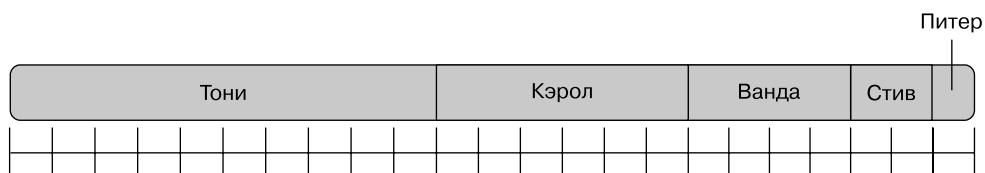
print weighted_random_choice($choices);

```

## Обсуждение

В примере выше каждому возможному варианту присваивается определенный «вес» — рейтинг или оценка, отражающие их значимость или приоритет. Для удобства вы можете ранжировать каждый вариант по весу, начиная с наиболее важного. Затем с помощью случайного числа в пределах общего количества весов вы определяете, на каком из вариантов остановиться.

Это легче всего представить на числовой линии. В нашем примере Тони участвует в отборе с весом 10, а Питер — с весом 1. Это означает, что вероятность победы Тони в десять раз выше, чем у Питера, однако все равно существует вероятность, что ни один из них не будет выбран. На рис. 5.1 показан относительный вес каждого, если упорядочить возможные варианты по весу и нанести их на числовую линию.



**Рис. 5.1.** Потенциальный выбор, упорядоченный и визуализированный по весу

Алгоритм, определенный в `weighted_random_choice()`, проверит, находится ли выбранное случайное число в пределах границ каждого возможного варианта, и, если нет, перейдет к следующему кандидату. Если по какой-либо причине вы не можете сделать выбор, функция вызовет исключение<sup>1</sup>.

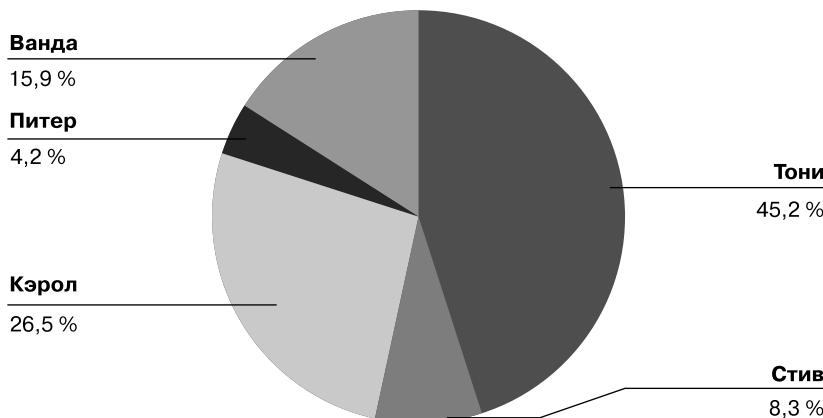
<sup>1</sup> Исключения и обработка ошибок подробно рассматриваются в главе 12.

Убедиться во взвешенном характере этого решения можно, выполнив алгоритм тысячу раз, а затем построив график, демонстрирующий, когда был выбран каждый из вариантов. Пример 5.8 показывает, как такой повторяющийся выбор можно вывести в таблицу, а рис. 5.2 иллюстрирует результат. Оба примера свидетельствуют о том, насколько велика вероятность выбора Тони по сравнению с любым другим кандидатом.

**Пример 5.8.** Повторный подбор взвешенного случайного выбора

```
$output = fopen('output.csv', 'w');
fputcsv($output, ['selected']);

foreach (range(0, 1000) as $i) {
    $selection = weighted_random_choice($choices);
    fputcsv($output, [$selection]);
}
fclose($output);
```



**Рис. 5.2.** Круговая диаграмма, иллюстрирующая относительное количество раз, когда выбирается каждый вариант

Рисунок 5.2 наглядно демонстрирует, что Тони выбирают примерно в десять раз чаще, чем Питера. Это прекрасно согласуется с тем, что его вес равен 10, а вес Питера — 1. Аналогично вес Ванды, равный 4, соотносится с тем, что ее выбирают в два раза чаще, чем Стива, у которого вес равен 2.

Учитывая, что выбор здесь случайный, повторное проведение того же эксперимента приведет к несколько иному процентному соотношению для каждого кандидата. Однако целочисленные веса каждого из них всегда будут приводить к примерно одинаковому распределению.

## Читайте также

Документацию PHP по `random_int()` (<https://oreil.ly/Pq16w>) и `arsort()` (<https://oreil.ly/VZ-Vz>), а также рецепт 5.4, где приведены дополнительные примеры использования `random_int()` на практике.

# 5.7. Вычисление логарифмов

## Задача

Вы хотите вычислить логарифм числа.

## Решение

Для работы с натуральными логарифмами (с основанием  $e$ ) используйте `log()`:

```
$log = log(5);  
// 1.6094379124341
```

Для логарифма с произвольным основанием укажите основание в качестве второго необязательного параметра:

```
$log2 = log(16, 2);  
// 4.0
```

## Обсуждение

PHP поддерживает вычисление логарифмов с помощью встроенного расширения Math. Когда вы вызываете `log()` без указания основания, PHP по умолчанию задействует константу `M_E`, которая закодирована как значение  $e$ , или приблизительно 2,718281828459.

Если вы попытаетесь взять логарифм отрицательного значения, PHP всегда будет возвращать `NAN`, константу (тиปизированную как `float`), которая не является числом (not a number). Если вы попытаетесь передать отрицательное основание, PHP вернет ошибку `ValueError` и выдаст предупреждение.

Любое положительное ненулевое основание поддерживается функцией `log()`. Во многих приложениях основание 10 используется настолько часто, что PHP поддерживает отдельную функцию `log10()` только для этого основания. Функционально это эквивалентно передаче целого числа 10 в качестве основания для `log()`.

## Читайте также

Документация PHP по различным функциям, поддерживаемым расширением Math (<https://oreil.ly/nLOM7>), включая log() (<https://oreil.ly/r-WYo>) и log10() (<https://oreil.ly/7Tn4t>).

## 5.8. Вычисление экспоненты

### Задача

Вы хотите возвести число в произвольную степень.

### Решение

Используйте функцию pow() в PHP следующим образом:

```
// 2^5
$power = pow(2, 5); // 32

// 3^0.5
$power = pow(3, 0.5); // 1.7320508075689

// e^2
$power = pow(M_E, 2); // 7.3890560989306
```

### Обсуждение

Функция pow() в PHP — это эффективный способ возведения любого числа в степень и возвращения целого числа или результата с плавающей точкой. Помимо формы функции, PHP предоставляет специальный оператор для возведения числа в степень: \*\*.

Следующий код эквивалентен использованию pow() в примерах из «Решения»:

```
// 2^5
$power = 2 ** 5; // 32
// 3^0.5
$power = 3 ** 0.5; // 1.7320508075689
// e^2
$power = M_E ** 2; // 7.3890560989306
```



Для обозначения экспоненты обычно используется символ каратки (`^`), однако в PHP он зарезервирован для оператора XOR. Подробнее об этом и других операторах читайте в главе 2.

Хотя возвведение константы `e` в произвольную степень возможно с помощью функции `pow()`, в PHP для этого предусмотрена специальная альтернатива: `exp()`. Операторы `pow(M_E, 2)` и `exp(2)` функционально эквивалентны. Они реализованы с помощью разного кода и из-за того, что PHP использует разные способы внутреннего представления чисел с плавающей точкой, будут возвращать немного различающиеся результаты<sup>1</sup>.

## Читайте также

Документация PHP по функциям `pow()` (<https://oreil.ly/JEsKM>) и `exp()` (<https://oreil.ly/AsgKw>).

# 5.9. Форматирование чисел как строк

## Задача

Вы хотите вывести число с разделителями разрядов, чтобы сделать его более читаемым для конечных пользователей вашего приложения.

## Решение

Используйте функцию `number_format()`, чтобы автоматически добавлять разделители разрядов при преобразовании числа в строку. Например:

```
$number = 25519;  
print number_format($number);  
// 25,519  
  
$number = 64923.12  
print number_format($number, 2);  
// 64,923.12
```

<sup>1</sup> Подробнее о допустимых различиях между числами с плавающей точкой читайте в рецепте 5.2.

## Обсуждение

Встроенная в PHP функция `number_format()` автоматически группирует разряды чисел вместе, а также округляет десятичные знаки до заданной точности. Вы также можете изменить десятичные и тысячные разделители в соответствии с заданным регионом или форматом.

Предположим, вы хотите использовать точки для разделения тысяч и запятую для разделения дробной части (как это принято в датских форматах чисел). Для этого нужно задействовать функцию `number_format()` следующим образом:

```
$number = 525600.23;  
  
print number_format($number, 2, ',', '.');  
// 525.600,23
```

Встроенный класс `NumberFormatter` в PHP предоставляет аналогичные возможности, но позволяет явно определить регион, а не запоминать его формат<sup>1</sup>. Вы можете переписать приведенный выше пример, чтобы использовать `NumberFormatter` с настройкой `da_DK` для идентификации датского формата:

```
$number = 525600.23;  
  
$fmt = new NumberFormatter('da_DK', NumberFormatter::DEFAULT_STYLE);  
print $fmt->format($number);  
// 525.600,23
```

## Читайте также

Документация PHP по функции `number_format()` ([https://oreil.ly/3\\_L6J](https://oreil.ly/3_L6J)) и классу `NumberFormatter` (<https://oreil.ly/IC3a9>).

## 5.10. Работа с очень большими или очень маленькими числами

### Задача

Вам необходимо использовать числа, которые слишком велики (или слишком малы), чтобы их можно было обработать с помощью встроенных в PHP типов.

---

<sup>1</sup> Сам класс `NumberFormatter` является частью расширения PHP `intl` (<https://oreil.ly/B-85H>). Этот модуль не встроен по умолчанию, и может потребоваться его отдельная установка (включение).

## Решение

Используйте библиотеку GMP:

```
$sum = gmp_pow(4096, 100);
print gmp_strval($sum);
// 17218479456385750618067377696052635483579924745448689921733236816400
// 74069124174561939748453723604617328637091903196158778858492729081666
// 10249916098827287173446595034716559908808846798965200551239064670644
// 19056526231345685268240569209892573766037966584735183775739433978714
// 57858778270138079724077247764787455598671274627136289222751620531891
// 4435913511141036261376
```

## Обсуждение

PHP поддерживает два расширения для работы с числами, которые слишком велики или малы для представления встроенными типами. Расширение BCMath (<https://oreil.ly/XhhdH>) — это интерфейс к системному калькулятору, который поддерживает математику произвольной точности. В отличие от встроенных типов PHP, BCMath работает с числами до 2 147 483 647 десятичных цифр при условии, что в системе достаточно памяти.

К сожалению, BCMath кодирует все числа как обычные строки в PHP, что несколько затрудняет его использование в современных приложениях, ориентированных на строгую типизацию<sup>1</sup>.

Расширение GMP — это альтернатива, лишенная подобного недостатка. Здесь числа также хранятся как строки. Однако при обращении к остальной части PHP они обрабатываются как объекты GMP. Это различие помогает понять, работает функция с небольшим числом, закодированным как строка, или с большим, требующим использования расширения.



BCMath и GMP работают с целочисленными значениями и возвращают их, а не с дробными. Если вам нужно выполнить операции над числами с плавающей точкой, то, возможно, вам придется увеличить размер чисел на порядок (умножить на 10), а затем снова уменьшить их после завершения вычислений, чтобы учесть десятичные или дробные числа.

GMP не входит в состав PHP по умолчанию, хотя многие дистрибутивы позволяют легко его добавить. Если вы компилируете PHP из исходного кода, делайте это с помощью опции `--with-gmp`, она добавит поддержку автоматически. Если вы

<sup>1</sup> Ознакомьтесь с рецептом 3.4, чтобы узнать больше о строгой типизации в PHP.

используете менеджер пакетов для установки PHP (например, на Linux), вы можете установить пакет `php-gmp`, чтобы добавить эту поддержку напрямую<sup>1</sup>.

С появлением GMP вам станут доступны любые математические операции над числами неограниченного размера, но с одной оговоркой: вы больше не сможете использовать встроенные операторы PHP и должны использовать функциональный формат, определенный самим расширением. В примере 5.9 представлены некоторые переводы из родных операторов в вызовы функций GMP. Обратите внимание, что возвращаемый тип каждого вызова функции — это объект GMP, поэтому вам потребуется преобразовать его обратно в число или строку с помощью `gmp_intval()` или `gmp_strval()` соответственно.

#### **Пример 5.9.** Различные математические операции и их эквиваленты в GMP-функциях

```
$add = 2 + 5;
$add = gmp_add(2, 5);

$sub = 23 - 2;
$sub = gmp_sub(23, 2);

$div = 15 / 4;
$div = gmp_div(15, 4);

$mul = 3 * 9;
$mul = gmp_mul(3, 9);

$pow = 4 ** 7;
$pow = gmp_pow(4, 7);

$mod = 93 % 4;
$mod = gmp_mod(93, 4);

$eq = 42 == (21 * 2);
$eq = gmp_cmp(42, gmp_mul(21, 2));
```

Последняя строчка кода в примере 5.9 демонстрирует функцию `gmp_cmp()`, которая позволяет сравнивать два значения, обернутые в GMP. Эта функция вернет положительное значение, если первый параметр больше второго, ноль, если они равны, и отрицательное значение, если второй параметр больше первого. По сути, это то же самое, что и оператор spaceship (см. рецепт 2.4), а не операция сравнения равенства, что потенциально дает больше пользы.

## Читайте также

Документация по PHP на GMP (<https://oreil.ly/rtfm3>).

---

<sup>1</sup> Нативные расширения подробно рассматриваются в главе 15.

## 5.11. Конвертация чисел из одной системы счисления в другую

### Задача

Вы хотите перевести число из одной системы счисления в другую.

### Решение

Используйте функцию `base_convert()` следующим образом:

```
// Число по основанию 10 (десятичное)
$decimal = '240';

// Преобразование из основания 10 в основание 16
// $hex = 'f0'
$hex = base_convert($decimal, 10, 16);
```

### Обсуждение

Функция `base_convert()` преобразовывает число с одним основанием в другое, что особенно полезно при работе с шестнадцатеричными или двоичными строками данных. PHP работает только с основаниями от 2 до 36. Основания, превышающие 10, используют символы алфавита для обозначения дополнительных цифр: а равно 10, б равно 11, вплоть до z равно 35.

Обратите внимание, что приведенный в примере выше код передает строку в `base_convert()`, а не целое или дробное число. Это связано с тем, что PHP попытается привести входную строку к числу с соответствующим основанием, прежде чем преобразовать ее в другое основание и вернуть строку. Строки — наилучшее средство представления шестнадцатеричных или восьмеричных чисел в PHP, они достаточно универсальны и могут представлять числа с любым основанием.

В дополнение к более общей функции `base_convert()` PHP поддерживает несколько других функций преобразования, специфичных для систем счисления. Эти функции перечислены в табл. 5.3.



PHP поддерживает две функции для преобразования двоичных данных в шестнадцатеричное представление: `bin2hex()` и `hex2bin()`. Эти функции не предназначены для преобразования строкового представления двоичных данных (например, 11111001) в шестнадцатеричное, вместо этого они будут оперировать двоичными байтами этой строки.

**Таблица 5.3.** Специфические функции преобразования основания

Имя функции	От основания	К основанию
bindec()	Двоичная (кодируется как string)	Десятичная (кодируется как int или, из-за размера, как float)
decbin()	Десятичная (кодируется как int)	Двоичная (кодируется как string)
hexdec()	Шестнадцатеричная (кодируется как string)	Десятичная (кодируется как int или, из-за размера, как float)
dechex()	Десятичная (кодируется как int)	Шестнадцатеричная (кодируется как string)
octdec()	Восьмеричная (кодируется как string)	Шестнадцатеричная (кодируется как int или, из-за размера, как float)
decoct()	Десятичная (кодируется как int)	Восьмеричная (кодируется как string)

Обратите внимание, что, в отличие от `base_convert()`, специализированные функции преобразования оснований часто работают с числовыми типами напрямую. Если вы используете строгую типизацию, это позволит избежать необходимости явного приведения числового типа к строковому перед изменением основания, что потребовалось бы при работе с функцией `base_convert()`.

## Читайте также

Документация PHP по функции `base_convert()` ([https://oreil.ly/NVsk\\_](https://oreil.ly/NVsk_)).

## ГЛАВА 6

---

# Дата и время

Работа с датами и временем — одна из самых сложных задач, которую можно решить на любом языке, а тем более на PHP. Это просто потому, что время относительно — «сейчас» может различаться от пользователя к пользователю и вызывать разное поведение в вашем приложении.

## Объектно-ориентированный подход

Разработчики PHP в основном работают с объектами `DateTime` в коде. Эти объекты «обворачивают» конкретный момент во времени и предоставляют широкий спектр функций. Вы можете вычислить разницу между двумя объектами `DateTime`, конвертировать время между произвольными часовыми поясами или добавить/вычесть интервалы времени из статичного объекта.

Кроме того, PHP поддерживает объект `DateTimeImmutable`, который функционально идентичен `DateTime`, но не может быть изменен напрямую. Большинство методов объекта `DateTime` возвращают тот же самый объект и изменяют его внутреннее состояние. Те же методы для `DateTimeImmutable` оставляют внутреннее состояние, но возвращают новые экземпляры, отражающие результат изменения.



Оба класса даты/времени расширяют абстрактный базовый класс `DateTimeInterface`, что делает эти два класса практически взаимозаменяемыми в рамках функциональности даты и времени в PHP. Там, где вы видите `DateTime` в текущей главе, вы можете использовать экземпляр `DateTimeImmutable` и получить аналогичную, если не идентичную, функциональность.

## Часовые пояса

Одна из самых сложных проблем, с которой сталкивается любой разработчик, — работа с часовыми поясами, особенно когда речь идет о переходе на летнее время. С одной стороны, легко упростить и предположить, что все временные метки в приложении ссылаются на один и тот же часовой пояс. Однако это редко бывает правдой.

К счастью, PHP позволяет работать с часовыми поясами очень просто. В каждый объект `DateTime` встроен часовой пояс, обычно основанный на значении по умолчанию, определенном в системе, на которой работает PHP. Вы также можете явно задать часовой пояс при создании `DateTime`, что сделает регион и время, на которые вы ссылаетесь, совершенно однозначными. Конвертация между часовыми поясами также проста и эффективна и подробно рассматривается в рецепте 6.9.

## Временные метки Unix

Многие компьютерные системы используют временные метки Unix для представления даты и времени. Эти временные метки означают количество секунд, прошедших между эпохой Unix (1 января 1970 года в 00:00:00 GMT) и заданным временем. Они потребляют мало памяти и часто используются в базах данных и программных API. Однако подсчет количества секунд, прошедших с определенного фиксированного момента, не совсем удобен для пользователя, поэтому требуется надежный способ преобразования между времennymi метками Unix и понятными для человека представлениями даты/времени в ваших приложениях.

Встроенные в PHP средства форматирования упрощают это дело. Дополнительные функции, такие как `time()` (<https://oreil.ly/RBqxh>), также позволяют получать временные метки Unix напрямую.

В следующих разделах подробно рассматриваются эти темы, а также некоторые другие задачи, связанные с датой и временем.

### 6.1. Поиск текущей даты и времени

#### Задача

Вы хотите узнать текущие дату и время.

#### Решение

Чтобы вывести текущую дату и время в определенном формате, используйте `date()`. Например:

```
$now = date('r');
```

Вывод функции `date()` зависит от системы, на которой она выполняется, и текущего фактического времени. Используя `r` в качестве строки форматирования, эта функция вернет что-то вроде следующего:

```
Wed, 09 Nov 2022 14:15:12 -0800
```

Аналогично, вновь созданный объект `DateTime` также будет представлять текущую дату и время. Метод `:format()` этого объекта демонстрирует то же поведение, что и `date()`, поэтому следующие два утверждения функционально идентичны:

```
$now = date('r');  
  
$now = (new DateTime())->format('r');
```

## Обсуждение

Функция `date()` в PHP, а также объект `DateTime`, инстанцированный без параметров, автоматически наследуют текущую дату и время системы, на которой они запущены. Дополнительным параметром `r`, передаваемым в обе функции, является символ формата, который определяет, как преобразовать информацию о дате/времени в строку — в данном случае в дату, отформатированную в соответствии с RFC 2822 (<https://oreil.ly/WrB1I>). Подробнее о форматировании даты вы можете узнать в рецепте 6.2.

Функция PHP `getdate()` считается эффективной альтернативой для получения ассоциативного массива всех частей текущей системной даты и времени. Этот массив будет содержать ключи и значения, приведенные в табл. 6.1.

**Таблица 6.1.** Ключевые элементы, возвращаемые функцией `getdate()`

Ключ	Описание значения	Используемые значения
<code>seconds</code>	Секунды	От 0 до 59
<code>minutes</code>	Минуты	От 0 до 59
<code>hours</code>	Часы	От 0 до 23
<code>mday</code>	День месяца	С 1 по 31
<code>wday</code>	Числовое представление дня недели	От 0 (воскресенье) до 6 (суббота)
<code>mon</code>	Месяц	От 1 до 12
<code>year</code>	Год	2024
<code>yday</code>	День года	От 0 до 365
<code>weekday</code>	Текстовое представление дня недели	С воскресенья (Sunday) по субботу (Saturday)
<code>month</code>	Текстовое представление месяца года	С января (January) по декабрь (December)
<code>0</code>	Временная метка Unix	От 0 до 2147483647

В некоторых приложениях вам может понадобиться только день недели, а не полноценный объект `DateTime`. Рассмотрим пример 6.1, в котором показано, как этого можно добиться с помощью `DateTime` или `getdate()`.

**Пример 6.1.** Сравнение DateTime с getdate()

```
print (new DateTime())->format('l') . PHP_EOL;  
  
print getdate()['weekday'] . PHP_EOL;
```

Эти две строки кода функционально эквивалентны. Для такой простой задачи, как «вывести сегодняшнюю дату», подойдет любая из них. Объект `DateTime` предоставляет функциональные возможности для преобразования часовых поясов или прогнозирования будущих дат (оба варианта рассматриваются в других рецептах). Ассоциативный массив, возвращаемый функцией `getdate()`, лишен этой функциональности, но компенсирует данный недостаток за счет простых, легко распознаваемых ключей массива.

## Читайте также

Документация PHP по функциям даты и времени (<https://oreil.ly/rJ9fn>), классу `DateTime` (<https://oreil.ly/t28Zh>) и функции `get date()` (<https://oreil.ly/Kv7l8>).

## 6.2. Форматирование дат и времени

### Задача

Вы хотите вывести дату в строку в определенном формате.

### Решение

Используйте метод `::format()` для объекта `DateTime`, чтобы задать формат возвращаемой строки, как показано в примере 6.2.

**Пример 6.2.** Примеры формата даты и времени

```
$birthday = new DateTime('2017-08-01');  
  
print $birthday->format('l, F j, Y') . PHP_EOL; ❶  
print $birthday->format('n/j/y') . PHP_EOL; ❷  
print $birthday->format(DateTime::RSS) . PHP_EOL; ❸
```

❶ Tuesday, August 1, 2017 (Вторник, 1 августа 2017 г.).

❷ 8/1/17.

❸ Tue, 01 Aug 2017 00:00:00 +0000 (Вт, 01 Авг 2017 00:00:00 +0000).

## Обсуждение

Как функция `date()`, так и метод `::format()` объекта `DateTime` принимают различные входные строки, которые в итоге определяют окончательную структуру строки, создаваемой PHP. Каждая строка формата состоит из отдельных символов, которые представляют определенные части даты или времени, как показано в табл. 6.2.

**Таблица 6.2.** Символы формата PHP

Символ	Описание	Используемые значения
<b>День</b>		
d	Числовое представление дня месяца (начиная с 01)	От 01 до 31
D	Текстовое представление дня недели (трехбуквенное)	С понедельника (Mon) по субботу (Sun)
j	Числовое представление дня месяца (начиная с 1)	От 1 до 31
l	Текстовое представление дня недели (полное наименование)	С воскресенья (Sunday) по субботу (Saturday)
N	Числовое представление дня недели по ISO 8601	От 1 (понедельник) до 7 (воскресенье)
S	Английский порядковый суффикс для дня месяца (двухсимвольный)	st, nd, rd или th. Хорошо сочетается с j
w	Числовое представление дня недели	От 0 (воскресенье) до 6 (суббота)
z	Числовое представление дня года (начиная с 0)	От 0 до 365
<b>Месяц</b>		
F	Текстовое представление месяца (полное наименование)	С января (January) по декабрь (December)
m	Числовое представление месяца (начиная с 01)	От 01 до 12
M	Текстовое представление месяца (трехбуквенное)	С января (Jan) по декабрь (Dec)
n	Числовое представление месяца (начиная с 1)	От 1 до 12
t	Количество дней в данном месяце	От 28 до 31

Продолжение ↗

**Таблица 6.2 (продолжение)**

Символ	Описание	Используемые значения
<b>Год</b>		
L	Високосный год	1 — високосный, 0 — не високосный
o	Год нумерации недель по ISO 8601. Имеет то же значение, что и Y, за исключением того, что если неделя, согласно ISO, относится к предыдущему или следующему году, то вместо нее используется этот год	1999 или 2003
Y	Числовое представление года (в четырехзначном формате)	1999 или 2003
U	Числовое представление года (в двухзначном формате)	99 или 03
<b>Время</b>		
a	Обозначение времени суток как «до полудня» или «после полудня» (в нижнем регистре)	am или pm
A	Обозначение времени суток как «до полудня» или «после полудня» (в верхнем регистре)	AM или PM
g	12-часовой формат времени (начиная с 1)	от 1 до 12
G	24-часовой формат времени (начиная с 0)	от 0 до 23
h	12-часовой формат времени (начиная с 01)	от 01 до 12
H	24-часовой формат времени (начиная с 00)	от 00 до 23
i	Минуты (начиная с 00)	от 00 до 59
s	Секунды (начиная с 00)	от 00 до 59
u	Микросекунды	654321
v	Миллисекунды	654
<b>Часовой пояс</b>		
e	Идентификатор часового пояса	UTC, GMT, Atlantic/Azores
I	Летнее/зимнее время	1 — летнее время, 0 — зимнее время
O	Время по Гринвичу (GMT) без двоеточия между часами и минутами	+0200
P	Время по Гринвичу (GMT) с двоеточием между часами и минутами	+02:00

Символ	Описание	Используемые значения
p	То же, что и P, но вместо +00:00 возвращается Z	+02:00
T	Аббревиатура часового пояса, если она известна; в противном случае — смещение по Гринвичу.	EST, MDT, +05
Z	Смещение часового пояса в секундах	От -43200 до 50400
<b>Другие</b>		
U	Временная метка Unix	От 0 до 2147483647

Комбинация этих символов в строке формата определяет, как именно PHP будет преобразовывать заданную конструкцию даты/времени в строку.

Аналогично PHP определяет несколько предопределенных констант, представляющих широко известные и часто используемые форматы. В табл. 6.3 приведены некоторые из наиболее полезных.

**Таблица 6.3.** Предопределенные константы даты

Константа	Класс константы	Формат символов	Пример
DATE_ATOM	DateTime::ATOM	Y-m-d \TH:i:sP	2023-08-01T13:22:14-08:00
DATE_COOKIE	Date Time::COOKIE	l, d-M-Y H:i:s T	Tuesday, 01-Aug-2023 13:22:14 GMT-0800
DATE_ISO8601 (К сожалению, DATE_ISO8601 несовместим со стандартом ISO 8601. Если необходимо, используйте DATE_ATOM)	Date Time::ISO8601	Y-m-d \TH:i:sO	2013-08-01T21:21:14\+0000
DATE_RSS	DateTime::RSS	D, d M Y H:i:s O	Tue, 01 Aug 2023 13:22:14 -0800

## Читайте также

Полная документация по символам формата (<https://oreil.ly/oQpYP>) и предопределенным константам DateTime (<https://oreil.ly/XJiZy>).

## 6.3. Преобразование дат и времени во временные метки Unix

### Задача

Вы хотите преобразовать определенную дату или время во временную метку Unix и затем преобразовать ее в локальную дату или время.

### Решение

Чтобы преобразовать заданную дату/время во временную метку, используйте символ формата U (см. табл. 6.2) с `DateTime::format()` следующим образом:

```
$date = '2023-11-09T13:15:00-0700';
$dateObj = new DateTime($date);

echo $dateObj->format('U');
// 1699560900
```

Чтобы преобразовать заданную временную метку в объект `DateTime`, также используйте символ формата U, но с `DateTime::createFromFormat()`, как показано ниже:

```
$timestamp = '1648241792';
$dateObj = DateTime::createFromFormat('U', $timestamp);

echo $dateObj->format(DateTime::ISO8601);
// 2022-03-25T20:56:32+0000
```

### Обсуждение

Метод `::createFromFormat()` является статической инверсией метода `DateTime::format()`. Оба метода используют идентичные строки формата для указания используемого формата<sup>1</sup>, но представляют собой противоположные преобразования между отформатированной строкой и исходным состоянием объекта `DateTime`. В примере выше явно используется символ формата U, чтобы сообщить PHP, что входные данные являются временной меткой Unix.

Если входная строка не соответствует вашему формату, PHP вернет литеральное значение `false`, как в следующем примере:

```
$timestamp = '2023-07-23';
$dateObj = DateTime::createFromFormat('U', $timestamp);

echo gettype($dateObj); // false
```

---

<sup>1</sup> Строки формата и доступные символы формата рассматриваются в рецепте 6.2.

При разборе пользовательского ввода полезно проверить возврат `::createFromFormat()`, чтобы убедиться, что введенная дата была валидной. Подробнее о проверке дат см. в рецепте 6.7.

Вместо того чтобы работать с полным объектом `DateTime`, вы можете взаимодействовать с частями даты/времени напрямую. Функция PHP `mktime()` (<https://oreil.ly/YFKz0>) всегда возвращает временную метку Unix, а единственным необходимым параметром является час.

Предположим, что вам нужна временная метка Unix, представляющая 4 июля 2023 года в полдень по Гринвичу (без смещения часового пояса). Это можно сделать двумя способами, как показано в примере 6.3.

#### Пример 6.3. Создание временной метки напрямую

```
$date = new DateTime('2023-07-04T12:00:00');
$timestamp = $date->format('U')  
①
$timestamp = mktime(month: 7, day: 4, year: 2023, hour: 12); ②
```

① Этот вывод будет в точности равен 1688472000.

② Этот вывод окажется близок к 1688472000, но будет отличаться в последних трех цифрах.

Хотя более простой пример выглядит элегантно и позволяет избежать инстанцирования объекта только для того, чтобы превратить его обратно в число, в нем есть важная проблема.

Если не указать параметр (в данном случае минуты или секунды), то `mktime()` по умолчанию следует использовать текущие системные значения для этих параметров. Если бы вы запустили этот пример в 3:05 пополудни, результат мог бы быть таким: 1688472300.

При обратном преобразовании в `DateTime` эта временная метка Unix преобразуется в 12:05:00, а не в 12:00:00, что представляет собой (потенциально незначительное) отклонение от ожидаемого приложением результата.

Если вы решили воспользоваться интерфейсом `mktime()`, не забудьте предоставить значение для каждого компонента даты/времени или же создавайте свое приложение таким образом, чтобы незначительные отклонения были ожидаемы и обрабатывались.

## Читайте также

Документация по `DateTime::createFromFormat()` (<https://oreil.ly/otv8q>).

## 6.4. Преобразование временных меток Unix в составные части даты и времени

### Задача

Вы хотите извлечь определенную составную часть даты или времени (день или час) из временной метки Unix.

### Решение

Передайте временную метку Unix в качестве параметра в `getdate()` и ссылайтесь на соответствующие ключи в полученном ассоциативном массиве. Например:

```
$date = 1688472300;
$time_parts = getdate($date);

print $time_parts['weekday'] . PHP_EOL; // Вторник
print $time_parts['hours'] . PHP_EOL; // 12
```

### Обсуждение

Единственный параметр, который вы можете передать в `getdate()`, — это временная метка Unix. Если он опущен, PHP воспользуется текущей системной датой и временем. Когда вы указываете временную метку, PHP анализирует ее внутренне и позволяет извлечь все ожидаемые элементы даты и времени.

Кроме того, вы можете передать временную метку в конструктор экземпляра `DateTime` двумя способами, чтобы создать из него полный объект.

1. Добавление символа @ перед временной меткой указывает PHP интерпретировать ввод как временную метку Unix, например `new DateTime('@1688472300')`.
2. Вы можете использовать символ формата U при импорте временной метки в `DateTime`, например `DateTime::createFromFormat('U', '1688472300')`.

В любом случае, как только временная метка будет должным образом разобрана и загружена в объект `DateTime`, вы сможете использовать его метод `::format()` для извлечения любого компонента по вашему желанию. Пример 6.4 — это альтернативная реализация задачи из «Решения», в которой используется `DateTime`, а не `getdate()`.

**Пример 6.4.** Извлечение составных частей даты и времени из временных меток Unix

```
$date = '1688472300';

$parsed = new DateTime("@{$date}");
print $parsed->format('l') . PHP_EOL;
print $parsed->format('g') . PHP_EOL;

$parsed2 = DateTime::createFromFormat('U', $date);
print $parsed2->format('l') . PHP_EOL;
print $parsed2->format('g') . PHP_EOL;
```

Любой из подходов в примере 6.4 является допустимой заменой функции `getdate()`, которая также дает преимущество в виде полноценного экземпляра `DateTime`. Вы можете вывести дату (или время) в любом формате, напрямую манипулировать базовым значением или даже конвертировать часовые пояса, если это необходимо. Каждый из этих возможных вариантов использования `DateTime` рассматривается в дальнейших разделах.

## Читайте также

Рецепт 6.1 для подробной информации о функции `getdate()`. О том, как манипулировать объектами `DateTime`, читайте далее в рецепте 6.8. Чтобы узнать, как можно напрямую управлять часовыми поясами, см. рецепт 6.9.

# 6.5. Вычисление разницы между двумя датами

## Задача

Вы хотите узнать, сколько времени прошло между двумя датами или временем.

## Решение

Инкапсулируйте каждую дату/время в объект `DateTime`. Воспользуйтесь методом `::diff()` для вычисления относительной разницы между одной и другой датами `DateTime`. Результатом будет объект `DateInterval`, как показано ниже:

```
$firstDate = new DateTime('2002-06-14');
$secondDate = new DateTime('2022-11-09');

$interval = $secondDate->diff($firstDate);

print $interval->format('%y лет %d дней %m месяца');
// 20 лет 25 дней 4 месяца
```

## Обсуждение

Метод `::diff()` объекта `DateTime` эффективно вычитает одну дату/время (аргумент, переданный в метод) из другой (представленной самим объектом). В результате получается представление относительная продолжительность времени между двумя объектами.



Метод `::diff()` игнорирует переход на летнее время. Чтобы правильно учесть потенциальную разницу в один час, присущую этой системе, целесообразно сначала преобразовать оба объекта даты/времени в UTC.

Важно также отметить, что, хотя в примере из «Решения» это может показаться похожим, метод `::format()` объекта `DateInterval` принимает совершенно другой набор символов формата по сравнению с `DateTime`. Каждый символ формата должен быть предварен знаком `%`, но сама строка формата может включать символы без форматирования (например, годы и месяцы в рассматриваемом примере).

Доступные символы формата перечислены в табл. 6.4. Во всех случаях, кроме символов формата `a` и `r`, использование строчных букв для символа формата возвращает числовое значение без ведущего нуля. Перечисленные символы формата в верхнем регистре возвращают как минимум две цифры с ведущим нулем. Помните, что каждый символ формата должен начинаться с `%`.

**Таблица 6.4.** Символы формата `DateInterval`

Символ	Описание	Пример
<code>%</code>	Символ <code>%</code>	<code>%</code>
<code>Y</code>	Годы	<code>03</code>
<code>M</code>	Месяцы	<code>02</code>
<code>D</code>	Дни	<code>09</code>
<code>H</code>	Часы	<code>08</code>
<code>I</code>	Минуты	<code>01</code>
<code>S</code>	Секунды	<code>04</code>
<code>F</code>	Микросекунды	<code>007705</code>
<code>R</code>	Знак минус ( <code>-</code> ) при отрицательном числе, плюс ( <code>+</code> ) при положительном	<code>- или +</code>
<code>r</code>	Знак минус ( <code>-</code> ) при отрицательном числе, отсутствие знака при положительном	<code>-</code>
<code>a</code>	Общее количество дней	<code>548</code>

## Читайте также

Полная документация по классу DateInterval (<https://oreil.ly/r0FBV>).

# 6.6. Разбор дат и времени из произвольных строк

## Задача

Вам нужно преобразовать произвольную строку, заданную пользователем, в допустимый объект DateTime для дальнейшего использования или манипулирования.

## Решение

Используйте мощную функцию PHP strtotime() для преобразования текстовой записи во временную метку Unix, а затем передайте ее в конструктор нового объекта DateTime. Например:

```
$entry = strtotime('last Wednesday');
$parsed = new DateTime("@{$entry}");

$entry = strtotime('now + 2 days');
$parsed = new DateTime("@{$entry}");

$entry = strtotime('June 23, 2023');
$parsed = new DateTime("@{$entry}");
```

## Обсуждение

Сила strtotime() заключается в поддерживаемых PHP форматах импорта даты и времени ([https://oreil.ly/2f4o\\_](https://oreil.ly/2f4o_)). К ним относятся форматы, которые используют компьютеры (например, YYYY-MM-DD для года, месяца и дня). Они также включают относительные указатели и сложные, составные форматы.



Принято предварять временную метку Unix литеральным символом @ при передаче ее в конструктор DateTime, что само по себе является производным от составных форматов даты/времени, поддерживаемых PHP.

Относительные форматы являются наиболее мощными и поддерживают понятные человеку строки, например такие:

- yesterday (вчера);
- first day of (первый день);

- now (сейчас);
- ago (ранее).

Вооружившись этими форматами, вы сможете разобрать с помощью PHP практически любую строку. Однако существуют некоторые ограничения. В примере из «Решения» я использовал `now + 2 days`, чтобы указать «через два дня». Пример 6.5 демонстрирует, что последнее приводит к ошибке парсера в PHP, хотя на английском языке все читается правильно.

#### **Пример 6.5.** Ограничения при разборе strtotime()

```
$date = strtotime('2 days from now');

if ($date === false) {
    throw new InvalidArgumentException('Ошибка при разборе строки!');
}
```

Важно отметить, что, каким бы умным ни был компьютер, его возможности всегда ограничены качеством ввода, предоставляемого конечными пользователями. Невозможно предусмотреть все возможные способы указания даты или времени; функция `strtotime()` близка к этому, но вам придется обрабатывать и ошибки ввода.

Еще одним потенциальным способом анализа дат, введенных пользователем, является функция PHP `date_parse()`. В отличие от `strtotime()`, эта функция ожидает достаточно хорошо отформатированную входную строку. Кроме того, она по-другому обрабатывает относительное время. Пример 6.6 иллюстрирует несколько строк, которые могут быть разобраны функцией `date_parse()`.

#### **Пример 6.6.** Примеры использования date\_parse()

```
$first = date_parse('January 4, 2022'); ①

$second = date_parse('Feb 14'); ②

$third = date_parse('2022-11-12 5:00PM'); ③

$fourth = date_parse('1-1-2001 + 12 years'); ④
```

① Анализирует 4 января 2022 года.

② Анализирует 14 февраля, но поле «год» равно null.

③ Анализирует дату и время, но без указания часового пояса.

④ Анализирует дату и сохраняет дополнительное относительное поле.

Вместо того чтобы возвращать временную метку, функция `date_parse()` извлекает из входной строки соответствующие части даты/времени и сохраняет их в ассоциативном массиве с ключами:

- `year`;
- `month`;
- `day`;
- `hour`;
- `minute`;
- `second`;
- `fraction`.

Кроме того, при передаче в строке временной спецификации (например, `+12 years` в примере 6.6) в массив будет добавлен ключ `relative` с информацией об относительном смещении.

Все это полезно для определения того, является ли введенная пользователем дата действительной. Функция `date_parse()` также возвращает предупреждения и ошибки при возникновении проблем с разбором, что еще больше упрощает проверку валидности даты. Подробнее о проверке валидности даты читайте в рецепте 6.7.

Повторное рассмотрение примера 6.5, но с использованием функции `date_parse()`, показывает, почему у PHP возникают проблемы с разбором выражения `2 days from now` как относительной даты:

```
$date = date_parse('2 days from now');

if ($date['error_count'] > 0) {
    foreach ($date['errors'] as $error) {
        return $error . PHP_EOL;
    }
}
```

Данный код выведет `The time zone could not be found in the data base`, что говорит о том, что PHP пытается разобрать дату, но не может определить, что на самом деле означает `from` в выражении `from now`. На самом деле, если проанализировать сам массив `$date`, можно увидеть, что он возвращает ключ `relative`. Это относительное смещение правильно представляет указанные два дня, то есть функция `date_parse()` (и даже `strtotime()`) смогла прочитать относительное смещение даты (`2 days`), но остановилась на последней части.

Эта дополнительная ошибка обеспечивает еще один контекст для отладки и, возможно, может послужить основой для какого-то сообщения об ошибке, которое приложение должно предоставить конечному пользователю. В любом случае это более полезно, чем просто возврат функцией `strtotime()` значения `false`.

## Читайте также

Документация по функциям `date_parse()` (<https://oreil.ly/2CECz>) и `strtotime()` (<https://oreil.ly/S7qkH>).

# 6.7. Проверка даты

## Задача

Вы хотите убедиться, что дата действительна. Например, вы хотите удостовериться, что дата рождения, предоставленная пользователем, является реальной датой в календаре, а не чем-то вроде 32 ноября 2022 года.

## Решение

Используйте функцию PHP `checkdate()` следующим образом:

```
$entry = 'November 32, 2022';
$parsed = date_parse($entry);

if (!checkdate($parsed['month'], $parsed['day'], $parsed['year'])) {
    throw new InvalidArgumentException('Указанный дату не существует!');
}
```

## Обсуждение

Функция `date_parse()` уже упоминалась в рецепте 6.6, однако сейчас мы рассмотрим ее в совокупности с `checkdate()`. Эта вторая функция сверяет введенную дату с календарем.

Она проверяет, находится ли месяц (первый параметр) в диапазоне от 1 до 12 и год (третий параметр) — в диапазоне от 1 до 32 767 (максимально возможное значение 2-байтового целого числа в PHP) и что количество дней действительно для данного месяца и года.

Функция `checkdate()` корректно обрабатывает месяцы с 28, 30 или 31 днем. В примере 6.7 показано, что она также учитывает високосные годы, проверяя возможность существования 29 февраля в соответствующие годы.

**Пример 6.7.** Проверка високосного года

```
$valid = checkdate(2, 29, 2024); // true  
  
$invalid = checkdate(2, 29, 2023); // false
```

**Читайте также**

Документация PHP по функции `checkdate()` (<https://oreil.ly/T2io8>).

## 6.8. Добавление к дате или вычитание из нее

### Задача

Вы хотите применить конкретное смещение (прибавление или вычитание) к фиксированной дате. Например, вам нужно рассчитать будущую дату, добавив несколько дней к текущей.

### Решение

Используйте методы `::add()` или `::sub()` объекта `DateTime` для добавления или вычитания значения `DateInterval` соответственно, как показано в примере 6.8.

**Пример 6.8.** Простое добавление `DateTime`

```
$date = new DateTime('December 25, 2023');  
  
// Когда заканчиваются 12 дней Рождества?  
$twelve_days = new DateInterval('P12D');  
$date->add($twelve_days);  
  
print 'Праздники заканчиваются ' . $date->format('F j, Y');  
  
// Праздники заканчиваются 6 января 2024 года
```

### Обсуждение

Методы `::add()` и `::sub()` объекта `DateTime` изменяют сам объект путем добавления или вычитания заданного интервала. Интервалы указываются с помощью обозначения периода, который идентифицирует количество времени, представленного этим интервалом. В табл. 6.5 показаны символы формата, используемые для обозначения интервала.

**Таблица 6.5.** Обозначения периодов, используемые DateInterval

Символ	Описание
<b>Обозначения периодов</b>	
Y	Годы
M	Месяцы
D	Дни
W	Недели
<b>Обозначения времени</b>	
H	Часы
M	Минуты
S	Секунды

Каждый форматированный период интервала даты начинается с буквы P. Затем следует количество лет/месяцев/дней/недель в этом периоде. Любые элементы времени в длительности имеют префикс в виде буквы T.



Для обозначения периодов месяцев и минут применяется буква M. Это может привести к путанице при попытке определить 15 минут против 15 месяцев в обозначении времени. Если вы планируете использовать минуты, убедитесь, что в вашей длительности правильно указан префикс T, чтобы избежать ошибки в вашем приложении.

Например, период в 3 недели и 2 дня будет представлен как P3W2D. Период в 4 месяца 2 часа и 10 секунд будет представлен как P4MT2H10S. Аналогично период в 1 месяц 2 часа и 30 минут будет представлен как P1MT2H30M.

## Изменяемость

Обратите внимание, что в примере 6.8 исходный объект `DateTime` преобразовывается сам по себе при вызове `::add()`. В простом примере это нормально. Если же вы пытаетесь вычислить несколько дат, смешенных от одной и той же начальной даты, изменяемость (мутабельность) объекта `DateTime` вызывает проблемы.

Вместо этого вы можете воспользоваться почти идентичным объектом `DateTimeImmutable`. Этот класс реализует тот же интерфейс, что и `DateTime`, но методы `::add()` и `::sub()` будут возвращать новые экземпляры класса, а не изменять внутреннее состояние самого объекта.

Рассмотрим сравнение обоих типов объектов в примере 6.9.

**Пример 6.9.** Сравнение DateTime и DateTimeImmutable

```
$date = new DateTime('December 25, 2023');
$christmas = new DateTimeImmutable('December 25, 2023');

// Когда заканчиваются 12 дней Рождества?
$twelve_days = new DateInterval('P12D');
$date->add($twelve_days); ❶
$end = $christmas->add($twelve_days); ❷

print 'Праздники заканчиваются ' . $date->format('F j, Y') . PHP_EOL;
print 'Праздники заканчиваются ' . $end->format('F j, Y') . PHP_EOL;

// Когда наступит следующее Рождество?
$next_year = new DateInterval('P1Y');
$date->add($next_year);
$next_christmas = $christmas->add($next_year);

print 'Следующее Рождество наступит ' . $date->format('F j, Y') . PHP_EOL;
print 'Следующее Рождество наступит ' . $next_christmas->format('F j, Y') . PHP_EOL; ❸

print 'Это Рождество наступит ' . $christmas->format('F j, Y') . PHP_EOL; ❹
```

❶ Поскольку `$date` — изменяемый объект, вызов его метода `::add()` приведет к прямому изменению объекта.

❷ Поскольку `$christmas` неизменяем, вызов `::add()` вернет новый объект, который должен быть сохранен в переменной.

❸ Вывод данных из результирующего объекта, полученного в результате добавления времени в `DateTimeImmutable`, представит правильные данные, поскольку новый объект был создан с правильной датой и временем.

❹ Даже после вызова `::add()` объект `DateTimeImmutable` всегда будет содержать те же данные, поскольку он, по сути, является неизменяемым.

Преимущество неизменяемых объектов в том, что вы можете рассматривать их как константы и быть уверены, что никто не перепишет календарь без вашего ведома. Единственным недостатком является использование памяти. Поскольку `DateTime` преобразует один объект, память не обязательно увеличивается по мере внесения изменений. Однако каждый раз, когда вы «модифицируете» объект `DateTimeImmutable`, PHP создает новый объект и потребляет дополнительную память.

В типичном веб-приложении затраты памяти здесь будут незначительными. Нет никаких причин не использовать объект `DateTimeImmutable`.

## Более простая модификация

Аналогичным образом и `DateTime`, и `DateTimeImmutable` реализуют метод `::modify()`, который работает с понятными человеку строками, а не с объектами интервала. Это позволяет находить относительные даты, такие как «прошлую пятницу» или «следующая неделя» от заданного объекта.

Хорошим примером является День благодарения, который в США выпадает на четвертый четверг ноября. Вы можете легко вычислить точную дату в конкретном году с помощью функции из примера 6.10.

**Пример 6.10.** Поиск даты празднования Дня благодарения с помощью `DateTime`

```
function findThanksgiving(int $year): DateTime
{
    $november = new DateTime("1 ноября, {$year}");
    $november->modify('четвертый четверг');

    return $november;
}
```

Такую же функциональность можно реализовать с помощью неизменяемых объектов даты, как показано в примере 6.11.

**Пример 6.11.** Поиск даты празднования Дня благодарения с помощью `DateTimeImmutable`

```
function findThanksgiving(int $year): DateTimeImmutable
{
    $november = new DateTimeImmutable("1 ноября, {$year}");
    return $november->modify('четвертый четверг');
}
```

## Читайте также

Документация по `DateInterval` (<https://oreil.ly/KvluE>).

# 6.9. Расчет времени в разных часовых поясах

## Задача

Вы хотите определить конкретное время в нескольких часовых поясах.

## Решение

Используйте метод `::setTimezone()` класса `DateTime` для изменения часового пояса:

```
$now = new DateTime();
(now->setTimezone(new DateTimeZone('America/Los_Angeles'));
```

```
print $now->format(DATE_RSS) . PHP_EOL;  
  
$now->setTimezone(new DateTimeZone('Europe/Paris'));  
  
print $now->format(DATE_RSS) . PHP_EOL;
```

## Обсуждение

Часовые пояса — одна из самых неприятных вещей, о которых приходится беспокоиться разработчикам приложений. К счастью, PHP позволяет сравнительно легко конвертировать время из одного часового пояса в другой. Метод `::setTimezone()`, используемый в примере выше, показывает, как произвольный объект `DateTime` может быть преобразован из одного часового пояса в другой, просто указав желаемый часовой пояс.



Помните, что и `DateTime`, и `DateTimeImmutable` реализуют метод `::setTimezone()`. Разница между их реализациями заключается в том, что `DateTime` изменяет состояние базового объекта, а `DateTimeImmutable` всегда возвращает новый объект.

Важно знать, какие часовые пояса доступны для использования в коде. Список слишком длинный, чтобы его перечислять, но разработчики могут воспользоваться функцией `DateTimeZone::listIdentifiers()`, чтобы просмотреть все доступные именованные часовые пояса. Если вашему приложению требуется какой-то определенный регион, вы можете дополнительно сократить список, используя одну из предустановленных групп констант, поставляемых вместе с классом.

Например, `DateTimeZone::listIdentifiers(DateTimeZone::AMERICA)` возвращает массив, в котором перечислены все часовые пояса, доступные на территории Америки. В конкретной тестовой системе этот массив содержит список из 145 часовых поясов, каждый из которых указывает на крупный город для определения местного часового пояса. Вы можете сгенерировать список возможных идентификаторов часовых поясов для каждой из следующих региональных констант:

- `DateTimeZone::AFRICA;`
- `DateTimeZone::AMERICA;`
- `DateTimeZone::ARCTIC;`
- `DateTimeZone::ARCTIC;`
- `DateTimeZone::ASIA;`
- `DateTimeZone::ATLANTIC;`
- `DateTimeZone::AUSTRALIA;`
- `DateTimeZone::EUROPE;`
- `DateTimeZone::INDIAN;`

- `DateTimeZone::PACIFIC`;
- `DateTimeZone::UTC`;
- `DateTimeZone::ALL`.

Аналогично можно использовать побитовые операторы для создания объединений из этих констант, чтобы получить списки всех часовых поясов в двух или более регионах. Например, `DateTimeZone::ANTARCTICA | DateTimeZone::ARCTIC` будет представлять все часовые пояса вблизи Южного или Северного полюсов.

Базовый класс `DateTime` позволяет создавать объект с определенным часовым поясом, а не принимать системные настройки по умолчанию. Просто передайте экземпляр `DateTimeZone` в качестве необязательного второго параметра в конструктор, и новый объект будет автоматически настроен на правильный часовой пояс.

Например, время `2022-12-15T17:35:53`, отформатированное в соответствии с ISO 8601 ([https://oreil.ly/rip\\_R](https://oreil.ly/rip_R)), означает «17:35 15 декабря 2022 года», но не содержит информации о часовом поясе. При инстанцировании объекта `DateTime` можно указать, что это время, например, в Токио:

```
$date = new DateTime('2022-12-15T17:35:53', new DateTimeZone('Asia/Tokyo'));  
  
echo $date->format(DateTime::ATOM);  
// 2022-12-15T17:35:53+09:00
```

Если сведения о часовом поясе отсутствуют в разбираемой строке времени, то его указание проясняет ситуацию. Если бы вы не добавили идентификатор часового пояса в предыдущем примере, PHP принял бы за основу тот, что настроен в системе<sup>1</sup>.

Если в строке `datetime` есть информация о часовом поясе, PHP проигнорирует любой часовой пояс, указанный во втором параметре, и разберет строку так, как указано.

## Читайте также

Документация по методу `::setTimezone()` (<https://oreil.ly/dk2gQ>) и классу `DateTimeZone` (<https://oreil.ly/MkdHB>).

---

<sup>1</sup> Вы можете проверить текущую настройку часового пояса для вашей системы с помощью функции `date_default_timezone_get()`.

# ГЛАВА 7

---

## Массивы

Массивы — это упорядоченные таблицы, конструкции, которые связывают определенные значения с легко идентифицируемыми ключами. Эти таблицы являются эффективным средством построения как простых списков, так и более сложных коллекций объектов. Кроме того, ими легко манипулировать: добавление или удаление элементов из массива не требует особых усилий и поддерживается множеством функциональных интерфейсов.

### Типы массивов

В PHP есть две формы массивов — числовые и ассоциативные. Когда вы определяете массив без явного задания ключей, PHP внутренне присваивает целочисленный индекс каждому элементу массива. Массивы индексируются, начиная с 0, и автоматически увеличиваются с шагом в 1.

Ассоциативные массивы могут иметь ключи в виде строк или целых чисел, но обычно используется первый вариант. Строковые ключи — это эффективный способ «поиска» определенного значения, хранящегося в массиве.

По структуре массивы реализованы как хеш-таблицы, что позволяет эффективно использовать прямые связи между ключами и значениями. Например:

```
$colors = [];
$colors['яблоко'] = 'красное';
$colors['груша'] = 'зеленая';
$colors['банан'] = 'желтый';

$numbers = [22, 15, 42, 105];

echo $colors['груша']; // зеленая
echo $numbers[2]; // 42
```

Однако, в отличие от более простых хеш-таблиц, массивы PHP также реализуют итерируемый интерфейс, позволяющий вам перебирать все их элементы

поочередно. Итерация достаточно очевидна, когда ключи числовые, но даже в ассоциативных массивах элементы имеют фиксированный порядок, потому что они хранятся в памяти. В рецепте 7.3 подробно описаны различные способы воздействия на каждый элемент в обоих типах массивов.

Вы также можете столкнуться с объектами или классами, которые выглядят и ведут себя как массивы, но на самом деле ими не являются. Фактически любой объект, реализующий интерфейс `ArrayAccess` ([https://oreil.ly/kdN4\\_](https://oreil.ly/kdN4_)), может применяться в качестве массива<sup>1</sup>. Эти более продвинутые реализации расширяют границы возможного использования массивов, превращая их в нечто большее, чем простые списки и хеш-таблицы.

## Синтаксис

PHP поддерживает два различных синтаксиса для определения массивов. Те, кто уже работал с PHP, знакомы с конструкцией `array()` (<https://oreil.ly/v75i9>), которая позволяет буквально определить массив во время выполнения программы следующим образом:

```
$movies = array('451 по Фаренгейту', 'Без жалости', 'Черная пантера');
```

Альтернативный и более простой синтаксис определения массива — квадратные скобки. Пример выше можно переписать следующим образом:

```
$movies = ['451 по Фаренгейту', 'Без жалости', 'Черная пантера'];
```

Оба формата пригодны для создания вложенных массивов (когда массив содержит другой массив) и считаются взаимозаменяемыми:

```
$array = array(1, 2, array(3, 4), [5, 6]);
```

Хотя смешение и сочетание синтаксисов, как в предыдущем примере, возможно, я настоятельно рекомендую в рамках вашего приложения придерживаться одной формы. Во всех фрагментах кода в этой главе массивы будут определяться с помощью краткого синтаксиса (квадратные скобки).

Все массивы в PHP отображаются от ключей к значениям. В предыдущих примерах массивы просто указывали значения и позволяли PHP автоматически присваивать ключи. Такие массивы считаются числовыми, поскольку в роли ключей выступают целые числа, начиная с 0. Более сложные массивы, такие как вложенные конструкции (см. пример 7.1), присваивают и значения, и ключи. Это делается путем перехода от ключа к значению с помощью оператора стрелочной функции (`=>`).

---

<sup>1</sup> Наследование классов рассматривается в главе 8, а интерфейсы объектов — в рецепте 8.7.

**Пример 7.1.** Ассоциативный массив с вложенными значениями

```
$array = array(
    'a' => 'A',
    'b' => ['b', 'B'],
    'c' => array('c', ['c', 'K'])
);
```

Хотя это и не является синтаксическим требованием, многие среды разработки (IDE) автоматически выравнивают стрелочные операторы в литералах многострочных массивов. Это облегчает чтение кода и является стандартом, которого придерживаюсь и я.

Следующие рецепты иллюстрируют различные способы работы, с помощью которых разработчики могут взаимодействовать с массивами (как числовыми, так и ассоциативными) для выполнения распространенных задач в PHP.

## 7.1. Объединение нескольких элементов по ключу в массиве

### Задача

Вы хотите связать несколько элементов с одним ключом массива.

### Решение

Превратите каждое значение массива в самостоятельный массив, например:

```
$cars = [
    'быстрые'      => ['феррари', 'ламборгини'],
    'медленные'     => ['хонда', 'тойота'],
    'электрические' => ['ривиан', 'tesla'],
    'большие'       => ['хаммер']
];
```

### Обсуждение

PHP не предъявляет никаких требований к типу данных, используемых для значений в массиве. Однако ключи должны быть либо строками, либо целыми числами. Кроме того, каждый ключ в массиве должен быть уникальным. Если вы попытаетесь задать несколько значений для одного и того же ключа, вы перезапишете существующие данные (см. пример 7.2).

**Пример 7.2.** Перезапись данных массива путем присваивания

```
$basket = [];

$basket['цвет']    = 'коричневый';
$basket['размер'] = 'большой';
$basket['вид']    = 'яблоко';
$basket['вид']    = 'апельсин';
$basket['вид']    = 'ананас';

print_r($basket);

// Array
// (
//     [цвет]    => коричневый
//     [размер]  => большой
//     [вид]     => ананас
// )
```

Поскольку PHP допускает только одно значение для уникального ключа в массиве, запись новых данных в этот ключ перезапишет его значение точно так же, как если бы вы переназначили переменную в вашем приложении. Если вам необходимо хранить несколько значений в одном ключе, используйте вложенный массив.

В примере из «Решения» показано, как каждый ключ может указывать на свой собственный массив. Однако PHP не требует, чтобы это было так. Все ключи, кроме одного, могут указывать на скаляр, а ключ, которому нужно несколько элементов, — на массив. В примере 7.3 продемонстрировано использование вложенного массива для хранения нескольких элементов, чтобы случайно не перезаписать одно значение, хранящееся в определенном ключе.

**Пример 7.3.** Запись массива в ключ

```
$basket = [];

$basket['цвет']    = 'коричневый';
$basket['размер'] = 'большой';
$basket['вид']    = [];
$basket['вид'][]  = 'яблоко';
$basket['вид'][]  = 'апельсин';
$basket['вид'][]  = 'ананас';

print_r($basket);

// Array
// (
//     [цвет]    => коричневый
//     [размер]  => большой
//     [вид]     => Array
//                 (

```

```
//          [0] => яблоко
//          [1] => апельсин
//          [2] => ананас
//      )
// )
```

```
echo $basket['вид'][2]; // ананас
```

Чтобы использовать элементы вложенного массива, вы выполняете цикл по ним так же, как и по родительскому массиву. Например, если вы хотите вывести все данные, хранящиеся в массиве `$basket` из примера 7.3, вам потребуются два цикла, как показано в примере 7.4.

#### Пример 7.4. Доступ к данным массива в цикле

```
foreach ($basket as $key => $value) { ❶
    if (is_array($value)) { ❷
        echo "{$key} => [" . PHP_EOL;

        foreach ($value as $item) { ❸
            echo "\t{$item}" . PHP_EOL;
        }

        echo ']' . PHP_EOL;
    } else {
        echo "{$key}: {$value}" . PHP_EOL;
    }
}

// цвет: коричневый
// размер: большой
// вид => [
//     яблоко
//     апельсин
//     ананас
// ]
```

❶ Родительский массив является ассоциативным, и вам нужны как его ключи, так и значения.

❷ Используется одна ветвь логики для вложенных массивов, другая — для скаляров.

❸ Поскольку вложенный массив является числовым, ключи игнорируются, а итерации выполняются только по значениям.

## Читайте также

В рецепте 7.3 приведены дополнительные примеры итерации по массивам.

## 7.2. Инициализация массива с диапазоном чисел

### Задача

Вы хотите построить массив последовательных целых чисел.

### Решение

Используйте функцию `range()` следующим образом:

```
$array = range(1, 10);  
print_r($array);
```

```
// Array  
// (  
//     [0] => 1  
//     [1] => 2  
//     [2] => 3  
//     [3] => 4  
//     [4] => 5  
//     [5] => 6  
//     [6] => 7  
//     [7] => 8  
//     [8] => 9  
//     [9] => 10  
// )
```

### Обсуждение

Функция `range()` в PHP автоматически перебирает заданную последовательность, присваивая значение ключу, основанному на определении этой последовательности. По умолчанию, как показано в примере выше, функция проходит последовательно по одному элементу за раз. Но этим поведение функции не ограничивается — если передать ей третий параметр, то размер шага изменится.

Перебор всех четных целых чисел от 2 до 100 можно выполнить следующим образом:

```
$array = range(2, 100, 2);
```

Аналогичным образом можно перебрать все нечетные целые числа от 1 до 100, изменив начальную точку последовательности на 1. Например:

```
$array = range(1, 100, 2);
```

Начальный и конечный параметры `range()` (первые два параметра соответственно) могут быть целыми числами, числами с плавающей точкой или даже строками. Такая

гибкость позволяет делать удивительные вещи в коде. Например, вместо подсчета натуральных чисел (целых) допускается создать массив чисел с плавающей точкой:

```
$array = range(1, 5, 0.25);
```

При передаче строковых символов в функцию `range()` PHP начнет перечислять символы ASCII. Вы можете использовать эту функциональность для быстрого создания массива, представляющего английский алфавит, как показано в примере 7.5.



PHP будет использовать все печатные символы ASCII, основываясь на их десятичном представлении, для выполнения запроса к функции `range()`. Это эффективный способ перечисления печатных символов, но вы должны помнить, что специальные символы, такие как `=`, `?` и `)`, попадают в таблицу ASCII, особенно если ваша программа ожидает алфавитно-цифровые значения в массиве.

### Пример 7.5. Создание массива алфавитных символов

```
$uppers = range('A', 'Z'); ①  
$lowers = range('a', 'z'); ②  
$special = range('!', ')'); ③
```

① Возвращает все заглавные символы от A до Z

② Возвращает все строчные символы от a до z

③ Возвращает массив специальных символов: `[!, ", #, $, %, &, ', (, )]`

## Читайте также

Документация PHP по функции `range()` ([https://oreil.ly/qH\\_iW](https://oreil.ly/qH_iW)).

## 7.3. Итерация элементов в массиве

### Задача

Вы хотите выполнить действие над каждым элементом массива.

### Решение

Для числовых массивов используйте конструкцию `foreach`:

```
foreach ($array as $value) {  
    // Воздействует на каждое значение $value  
}
```

Для ассоциативных массивов используйте `foreach()` с необязательными ключами:

```
foreach ($array as $key => $value) {  
    // Воздействует на каждое значение $value и/или $key  
}
```

## Обсуждение

В PHP есть концепция итерируемых объектов, и внутренне именно это является массивом. Другие структуры данных также могут реализовывать итерируемое поведение<sup>1</sup>, но любое итерируемое выражение может быть передано в `foreach` и будет возвращать содержащиеся в нем элементы по одному за раз в цикле.



PHP не удаляет значение переменной, используемой в цикле `foreach`, когда вы выходите из цикла. Вы все еще можете явно ссылаться на последнее значение, сохраненное в `$value` в примерах из «Решения», в программе вне цикла!

Самое важное, что нужно помнить, — это то, что `foreach` является языковой конструкцией, а не функцией. Как конструкция, `foreach` действует на заданное выражение и применяет определенный цикл к каждому элементу в этом выражении. По умолчанию этот цикл не изменяет содержимого массива. Если вы хотите сделать значения массива изменяемыми, вы должны передать их в цикл по ссылке, предварительно добавив к имени переменной символ `&`, как показано ниже:

```
$array = [1, 2, 3];  
  
foreach ($array as &$value) {  
    $value += 1;  
}  
  
print_r($array); // 2, 3, 4
```



Версии PHP до 8.0 предоставляли функцию `each()`, которая поддерживала указатель на массив и возвращала текущую пару «ключ/значение» массива перед перемещением этого указателя. Данная функция считалась устаревшей в PHP 7.2 и полностью удалена в релизе 8.0, но вы наверняка найдете примеры ее использования в книгах и Интернете. Обновите все вхождения `each()` до реализации `foreach`, чтобы обеспечить совместимость вашего кода.

Альтернативным подходом к использованию цикла `foreach` является создание явного цикла `for` по ключам массива. Для числовых массивов это наиболее простой

---

<sup>1</sup> Примеры очень больших итерируемых структур данных см. в рецепте 7.15.

способ, поскольку их ключи уже являются инкрементируемыми целыми числами, начиная с 0. Итерация по числовому массиву относительно проста и выполняется следующим образом:

```
$array = ['красный', 'зеленый', 'синий'];

$arrayLength = count($array);
for ($i = 0; $i < $array_length; $i++) {
    echo $array[$i] . PHP_EOL;
}
```



Хотя вызов `count()` для определения верхней границы цикла `for` можно поместить непосредственно в выражение, лучше хранить длину массива вне самого выражения. В противном случае `count()` будет вызываться заново на каждой итерации цикла, чтобы проверить, что вы все еще находитесь в границах массива. Для небольших массивов это незначительно; однако, когда вы начнете работать с большими коллекциями, повторные проверки `count()` вызовут проблемы с производительностью.

Итерация ассоциативного массива с помощью цикла `for` имеет свою специфику. Вместо перебора элементов массива напрямую происходит итерация ключей массива. Затем каждый ключ используется для извлечения соответствующего значения из массива, например:

```
$array = [
    'os'    => 'linux',
    'mfr'   => 'system76',
    'name'  => 'thelio',
];

$keys = array_keys($array);
$arrayLength = count($keys);
for ($i = 0; $i < $arrayLength; $i++) {
    $key = $keys[$i];
    $value = $array[$key];

    echo "{$key} => {$value}" . PHP_EOL;
}
```

## Читайте также

Документация PHP по конструкциям языка `foreach` (<https://oreil.ly/lmeAe>) и `for` (<https://oreil.ly/chSRT>).

## 7.4. Удаление элементов из ассоциативных и числовых массивов

### Задача

Вы хотите удалить один или несколько элементов из массива.

### Решение

Удалите элемент, указав его ключ или числовой индекс непосредственно с помощью функции `unset()`:

```
unset($array['key']);  
  
unset($array[3]);
```

Удалите несколько элементов за один раз, передав несколько ключей или индексов в `unset()` следующим образом:

```
unset($array['first'], $array['second']);  
  
unset($array[3], $array[4], $array[5]);
```

### Обсуждение

В PHP функция `unset()` фактически стирает все ссылки на ячейку памяти, содержащую указанную переменную. В нашем контексте эта переменная является элементом массива, поэтому ее устранение приводит к исключению этого элемента из самого массива. В ассоциативном массиве это выражается в удалении указанного ключа и связанного с ним значения.

В числовом массиве `unset()` делает гораздо больше. Она и убирает указанный элемент, и эффективно преобразует числовой массив в ассоциативный с целочисленными ключами. Это, скорее всего, именно то поведение, которое вы ожидаете в первую очередь (см. пример 7.6).

#### Пример 7.6. Сброс элементов в числовом массиве

```
$array = range('a', 'z');  
  
echo count($array) . PHP_EOL; ❶  
echo $array[12] . PHP_EOL; ❷  
echo $array[25] . PHP_EOL; ❸
```

```
unset($array[22]);
echo count($array) . PHP_EOL; ④
echo $array[12] . PHP_EOL; ⑤
echo $array[25] . PHP_EOL; ⑥
```

- ❶ По умолчанию в массиве представлены все английские буквы от a до z, поэтому эта строка выводит 26.
- ❷ 13-я буква в алфавите — m (помните, что массивы начинаются с индекса 0).
- ❸ 26-я буква в алфавите — z.
- ❹ После удаления элемента размер массива уменьшился до 25!
- ❺ 13-й буквой алфавита по-прежнему является m.
- ❻ 26-я буква в алфавите по-прежнему z. Кроме того, этот индекс все еще действителен, поскольку удаление элемента не приводит к переиндексации массива.

Обычно вы можете игнорировать индексы числовых массивов, поскольку они задаются PHP автоматически. Поэтому поведение функции `unset()`, неявно преобразующей эти индексы в числовые ключи, вызывает некоторое удивление. При использовании числового массива попытка получить доступ к индексу, превышающему длину массива, приводит к ошибке. Однако после использования `unset()` с массивом и уменьшения его размера часто получается массив с числовыми ключами, превышающими размер массива, как это было показано в примере 7.6.

Если вы хотите вернуться к числовому массиву после удаления элемента, вы можете полностью переиндексировать массив. Функция PHP `array_values()` возвращает новый массив с числовым индексом, который содержит только значения указанного массива. Например:

```
$array = ['первый', 'второй', 'третий', 'четвертый']; ❶
unset($array[2]); ❷
$array = array_values($array); ❸
```

- ❶ Массив по умолчанию имеет числовые индексы: [0 => первый, 1 => второй, 2 => третий, 3 => четвертый].
- ❷ Сброс элемента удаляет его из массива, но оставляет индексы (ключи) неизменными: [0 => первый, 1 => второй, 3 => четвертый].
- ❸ Вызов `array_values()` дает вам новый массив с совершенно новыми, правильно увеличивающимися числовыми индексами: [0 => первый, 1 => второй, 2 => четвертый].

Еще один способ удаления элементов из массива — функция `array_splice()`<sup>1</sup>. Она стирает часть массива и заменяет ее чем-то другим<sup>2</sup>. Рассмотрим пример 7.7, в котором функция `array_splice()` используется для замены элементов массива ничем, тем самым удаляя их.

### Пример 7.7. Удаление элементов массива с помощью `array_splice()`

```
$celestials = [
    'Солнце',
    'Меркурий',
    'Венера',
    'Земля',
    'Марс',
    'Пояс астероидов',
    'Юпитер',
    'Сатурн',
    'Уран',
    'Нептун',
    'Плутон',
    'Вояджер 1 и 2',
];
array_splice($celestials, 0, 1); ①
array_splice($celestials, 4, 1); ②
array_splice($celestials, 8); ③
print_r($celestials);

// Array
// (
//     [0] => Меркурий
//     [1] => Венера
//     [2] => Земля
//     [3] => Марс
//     [4] => Юпитер
//     [5] => Сатурн
//     [6] => Уран
//     [7] => Нептун
// )
```

① Сперва исключается «Солнце», чтобы очистить список планет Солнечной системы.

---

<sup>1</sup> Не путайте `array_splice()` с `array_slice()`. Эти две функции имеют совершенно разное применение, и последняя рассматривается в рецепте 7.7.

<sup>2</sup> Функция `array_splice()` также вернет элементы, извлеченные из целевого массива, если вам понадобится использовать эти данные для какой-либо другой операции. Подробнее об этом поведении см. в рецепте 7.7.

❷ После удаления «Солнца» индексы всех объектов сдвигаются. Вам все еще нужно удалить «Пояс астероидов» из списка, поэтому используйте его новый смещенный индекс.

❸ Наконец, обрежьте массив, убрав из него все, начиная с «Плутона» и до конца массива.

В отличие от `unset()`, измененный массив, созданный с помощью `array_splice()`, не сохраняет числовых индексов/ключей в числовых массивах! Это может быть хорошим способом избежать необходимости дополнительного вызова `array_values()` после исключения элемента из массива. Это также эффективный способ удаления непрерывных элементов из массива с числовыми индексами без необходимости явного указания каждого элемента.

## Читайте также

Документация по `unset()` (<https://oreil.ly/-ebRG>), `array_splice()` (<https://oreil.ly/g-M9G>) и `array_values()` (<https://oreil.ly/9FvTV>).

# 7.5. Изменение размера массива

## Задача

Вы хотите увеличить или уменьшить массив.

## Решение

Добавьте элементы в конец массива с помощью `array_push()`:

```
$array = ['яблоко', 'банан', 'кокос'];
array_push($array, 'виноград');

print_r($array);

// Array
// (
//     [0] => яблоко
//     [1] => банан
//     [2] => кокос
//     [3] => виноград
// )
```

Удалите элементы из массива с помощью функции `array_splice()`:

```
$array = ['яблоко', 'банан', 'кокос', 'виноград'];
array_splice($array, 1, 2);
```

```
print_r($array);

// Array
// (
//     [0] => яблоко
//     [1] => виноград
// )
```

## Обсуждение

В отличие от многих других языков, PHP не требует объявлять размер массива. Массивы динамичны — вы можете добавлять или удалять из них данные, когда захотите, без каких-либо реальных проблем.

В первом примере из «Решения» просто добавляется один элемент в конец массива. Хотя этот подход простой, он не самый эффективный. Вместо этого вы можете напрямую поместить отдельный элемент в массив следующим образом:

```
$array = ['яблоко', 'банан', 'кокос'];
$array[] = 'виноград';
```

```
print_r($array);

// Array
// (
//     [0] => яблоко
//     [1] => банан
//     [2] => кокос
//     [3] => виноград
// )
```

Ключевое различие между этими двумя примерами заключается в вызове функции. В PHP вызовы функций требуют больше ресурсов, чем языковые конструкции (например, операторы присваивания). Предыдущий пример немного эффективнее, но только если он используется несколько раз в приложении.

Если вы хотите добавить несколько элементов в конец массива, то функция `array_push()` будет более полезной. Она принимает и добавляет сразу несколько элементов, что позволяет избежать многократного присваивания. Пример 7.8 иллюстрирует разницу между подходами.

**Пример 7.8.** Добавление нескольких элементов с помощью `array_push()` в сравнении с присваиванием

```
$first = ['яблоко'];
array_push($first, 'банан', 'кокос', 'виноград');

$second = ['яблоко'];
$second[] = 'банан';
```

```
$second[] = 'кокос';
$second[] = 'виноград';

echo 'Массивы ' . ($first === $second ? 'одинаковые' : 'разные');

// Массивы одинаковые
```

Если же вы хотите добавлять элементы в начало массива, а не в конец, используйте `array_unshift()`:

```
$array = ['виноград'];
array_unshift($array, 'яблоко', 'банан', 'кокос');

print_r($array);

// Array
// (
//     [0] => яблоко
//     [1] => банан
//     [2] => кокос
//     [3] => виноград
// )
```



PHP сохраняет порядок элементов, переданных в `array_unshift()`, при добавлении их в целевой массив. Первый параметр станет первым элементом, второй — вторым и так далее, пока вы не достигнете исходного первого элемента массива.

Помните, что массивы в PHP не имеют заданного размера и ими можно легко манипулировать различными способами. Все предыдущие функциональные примеры (`array_push()`, `array_splice()` и `array_unshift()`) хорошо работают с числовыми массивами и не изменяют порядок или структуру их числовых индексов. Вы можете с тем же успехом добавить элемент в конец числового массива, направившись к новому индексу. Например:

```
$array = ['яблоко', 'банан', 'кокос'];
$array[3] = 'виноград';
```

Пока индекс, на который ссылается ваш код, непрерывен с остальной частью массива, предыдущий пример будет работать безупречно. Однако если вы не ведете подсчет и создаете разрыв в индексе, то фактически превращаете свой числовой массив в аналогичный, только с числовыми ключами.

Хотя все функции, используемые в этом рецепте, работают с ассоциативными массивами, они в основном взаимодействуют с числовыми ключами и будут вести себя странно, если приводить их с нечисловыми ключами. Разумно использовать эти функции только с числовыми массивами и манипулировать размерами ассоциативных массивов непосредственно на основе их ключей.

## Читайте также

Документация по `array_push()` (<https://oreil.ly/DhVgq>), `array_splice()` (<https://oreil.ly/eLoTZ>) и `array_unshift()` (<https://oreil.ly/BYisR>).

## 7.6. Добавление одного массива к другому

### Задача

Вы хотите объединить два массива в один новый массив.

### Решение

Используйте `array_merge()` следующим образом:

```
$first = ['a', 'b', 'c'];
$second = ['x', 'y', 'z'];

$merged = array_merge($first, $second);
```

Кроме того, вы можете использовать оператор `spread (...)` для прямого объединения массивов. Вместо вызова `array_merge()` предыдущий пример превращается в следующий:

```
$merged = [...$first, ...$second];
```

Оператор `spread` работает как с числовыми, так и с ассоциативными массивами.

### Обсуждение

Функция `array_merge()` в PHP — это очевидный способ объединить два массива в один. Однако она немного отличается по поведению для числовых и ассоциативных массивов.



Любое обсуждение объединения массивов неизбежно будет сводиться к термину «комбинирование». Обратите внимание, что `array_combine()` (<https://oreil.ly/wcM69>) сама по себе является функцией в PHP. Однако она не объединяет два массива, как показано в этом рецепте. Вместо этого она создает новый массив, используя два указанных: первый — для ключей, а второй — для значений нового массива. Это полезная функция, но не то, что можно использовать для слияния двух массивов.

Для числовых массивов все элементы второго массива добавляются к элементам первого. Функция игнорирует индексы обоих, и вновь созданный массив имеет непрерывные индексы (начиная с 0), как если бы вы построили его напрямую.

Для ассоциативных массивов ключи (и значения) второго массива добавляются к ключам (и значениям) первого. Если оба массива имеют одинаковые ключи, значения второго будут перезаписывать значения первого. В примере 7.9 показано, как данные одного массива перезаписывают данные другого.

#### **Пример 7.9.** Перезапись данных ассоциативного массива с помощью array\_merge()

```
$first = [
    'название' => 'Практическое руководство',
    'автор'      => 'Боб Миллс',
    'год'        => 2018
];
$second = [
    'год'      => 2023,
    'регион'  => 'США'
];

$merged = array_merge($first, $second);
print_r($merged);

// Array
// (
//     [название] => Практическое руководство
//     [автор]      => Боб Миллс
//     [год]        => 2023
//     [регион]    => США
// )
```

Бывают случаи, когда при объединении двух или более массивов необходимо сохранить данные, содержащиеся в дублирующих ключах. Здесь имеет смысл использовать array\_merge\_recursive(). В отличие от предыдущего примера, эта функция создаст массив, содержащий данные, определенные в повторяющихся ключах, а не перезапишет одно значение другим. Пример 7.10 иллюстрирует, как это происходит.

#### **Пример 7.10.** Объединение массивов с повторяющимися ключами

```
$first = [
    'название' => 'Практическое руководство',
    'автор'      => 'Боб Миллс',
    'год'        => 2018
];
```

```
$second = [
    'год'      => 2023,
    'регион'  => 'США'
];

$merged = array_merge_recursive($first, $second);
print_r($merged);

// Array
// (
//     [название] => Практическое руководство
//     [автор] => Боб Миллс
//     [год] => Array ()
//         (
//             [0] => 2018
//             [1] => 2023
//         )
//     [
//         [регион] => США
//     )
// )
```

Хотя в предыдущих примерах объединены только два массива, нет верхнего предела для количества массивов, которые вы можете объединить с помощью `array_merge()` или `array_merge_recursive()`. При объединении более чем двух массивов одновременно не забывайте о том, как обрабатываются повторяющиеся ключи, чтобы избежать потенциальной потери данных.

Третий и последний способ объединить два массива в один — это оператор сложения (`+`). На бумаге это выглядит как сложение двух массивов. На самом деле он добавляет любой новый ключ из второго массива к ключам первого. В отличие от `array_merge()`, эта операция не перезаписывает данные. Если во втором массиве есть ключи, дублирующие ключи первого массива, они игнорируются и используются данные из первого массива.

Этот оператор также явно работает с ключами массивов, что означает, что он не очень подходит для числовых массивов. Два одинаковых по размеру числовых массива рассматриваются как ассоциативные и будут иметь одинаковые ключи, потому что у них одинаковые индексы. Это означает, что данные второго массива будут полностью проигнорированы!

## Читайте также

Документация для `array_merge()` (<https://oreil.ly/s38Xa>) и `array_merge_recursive()` (<https://oreil.ly/aFQxS>).

## 7.7. Создание массива из фрагмента существующего массива

### Задача

Вы хотите выбрать фрагмент существующего массива и использовать его независимо.

### Решение

Используйте функцию `array_slice()`, чтобы выбрать последовательность элементов из существующего массива следующим образом:

```
$array = range('A', 'Z');
$slice = array_slice($array, 7, 4);

print_r($slice);

// Array
// (
//     [0] => H
//     [1] => I
//     [2] => J
//     [3] => K
// )
```

### Обсуждение

Функция `array_slice()` быстро извлекает непрерывную последовательность элементов из указанного массива на основе заданного смещения (позиции внутри массива) и длины элементов, которые необходимо извлечь. В отличие от `array_splice()`, она копирует последовательность элементов из массива, оставляя исходный массив неизменным.

Важно понимать полную сигнатуру функции, чтобы оценить ее возможности:

```
array_slice(
    array $array,
    int   $offset,
    ?int  $length = null,
    $bool $preserve_keys = false
): array
```

Обязательны только первые два параметра — целевой массив и начальное смещение. Если смещение положительное (или 0), то новая последовательность начнется с этой позиции от начала массива. Если смещение отрицательное, последовательность начнется с такого же количества позиций от конца массива.



Смещение массива явно указывает на позицию внутри массива, а не на ключи или индексы. Функция `array_slice()` работает с ассоциативными массивами так же легко, как и с числовыми массивами, потому что она использует относительные позиции элементов в массиве для определения новой последовательности и игнорирует фактические ключи массива.

Если вы задаете необязательный аргумент `$length`, он определяет максимальное количество элементов в новой последовательности. Обратите внимание, что новая последовательность ограничена количеством элементов в исходном массиве, поэтому, если длина превысит конец массива, ваша последовательность окажется короче, чем вы ожидали. В примере 7.11 приведен краткий пример подобного поведения.

#### Пример 7.11. Использование `array_slice()` со слишком коротким массивом

```
$array = range('a', 'e');
$newArray = array_slice($array, 4, 100);

print_r($newArray);

// Array
// (
//     [0] => e
// )
```

Если задано отрицательное значение длины, то последовательность остановится на указанном расстоянии от конца целевого массива. Если длина не определена (или равна `null`), то последовательность будет включать все элементы — от исходного смещения до конца целевого массива.

Последний параметр, `$preserve_keys`, указывает PHP, нужно ли сбрасывать целочисленные индексы фрагментов массива. По умолчанию PHP вернет заново проиндексированный массив с целочисленными ключами, начинающимися с 0. Пример 7.12 показывает, как меняется поведение функции в зависимости от этого параметра.



Функция `array_slice()` всегда сохраняет ключи строк в ассоциативном массиве независимо от значения `$preserve_keys`.

**Пример 7.12.** Поведение при сохранении ключей в array\_slice()

```
$array = range('a', 'e');

$standard = array_slice($array, 1, 2);
print_r($standard);

// Array
// (
//     [0] => b
//     [1] => c
// )

$preserved = array_slice($array, 1, 2, true);
print_r($preserved);

// Array
// (
//     [1] => b
//     [2] => c
// )
```

Помните, что числовые массивы в PHP можно рассматривать как ассоциативные с целочисленными ключами, которые начинаются с 0 и последовательно увеличиваются. Учитывая это, легко понять, как `array_slice()` ведет себя с ассоциативными массивами со строковыми и целочисленными ключами: она работает на основе позиции, а не ключа, как показано в примере 7.13.

**Пример 7.13.** Использование `array_slice()` для массива со смешанными типами ключей

```
$array = ['a' => 'яблоко', 'b' => 'банан', 25 => 'кола', 'd' => 'пончик'];
print_r(array_slice($array, 0, 3));
```

```
// Array
// (
//     [a] => яблоко
//     [b] => банан
//     [0] => кола
// )

print_r(array_slice($array, 0, 3, true));

// Array
// (
//     [a] => яблоко
//     [b] => банан
//     [25] => кола
// )
```

В рецепте 7.4 вы познакомились с функцией `array_splice()` для удаления последовательности элементов из массива. Удобно, что эта функция использует сигнатуру метода, аналогичную `array_slice()`:

```
array_splice(  
    array &$array,  
    int $offset,  
    ? int $length = null,  
    mixed $replacement = []  
) : array
```

Ключевое различие между этими функциями заключается в том, что одна из них изменяет исходный массив, а другая — нет. Вы можете использовать `array_slice()` для работы с подмножеством более крупной последовательности в изоляции или вместо этого полностью отделить две последовательности друг от друга. В любом случае функции демонстрируют схожее поведение и случаи использования.

## Читайте также

Документация по `array_slice()` (<https://oreil.ly/9iBvj>) и `array_splice()` (<https://oreil.ly/k-h7n>).

## 7.8. Преобразование между массивами и строками

### Задача

Вы хотите преобразовать строку в массив или объединить элементы массива в строку.

### Решение

Используйте `str_split()` для преобразования строки в массив:

```
$string = 'To be or not to be';  
$array = str_split($string);
```

Используйте функцию `join()`, чтобы объединить элементы массива в строку:

```
$array = ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'];  
$string = join('', $array);
```

## Обсуждение

Функция `str_split()` — это мощный способ преобразовать любую строку символов в массив одинаковых по размеру фрагментов. По умолчанию она разбивает строку на одиночные символы, но вы можете с тем же успехом разбить строку на любое количество символов. Последняя часть в последовательности гарантированно будет иметь указанную длину. В примере 7.14 делается попытка разбить строку на фрагменты по пять символов, однако следует отметить, что последний фрагмент получился короче заданного условия.

**Пример 7.14.** Использование `str_split()` с произвольными размерами блоков

```
$string = 'To be or not to be';
$array = str_split($string, 5);
var_dump($array);

// array(4) {
//   [0]=>
//   string(5) "To be"
//   [1]=>
//   string(5) " or n"
//   [2]=>
//   string(5) "ot to"
//   [3]=>
//   string(3) " be"
// }
```



Помните, что `str_split()` работает с байтами. Когда речь идет о строках с много-байтовой кодировкой, следует использовать функцию `mb_str_split()` (<https://oreil.ly/ocQi1>).

В некоторых случаях вам может понадобиться разделить строку на слова, а не на символы. Функция PHP `explode()` позволяет указать разделитель, по которому производится разбиение. Это удобно для разбиения предложения на массив составляющих его слов, как показано в примере 7.15.

**Пример 7.15.** Разбиение строки на массив слов

```
$string = 'To be or not to be';
$words = explode(' ', $string);

print_r($words);

// Array
// (
//   [0] => To
```

```
// [1] => be
// [2] => or
// [3] => not
// [4] => to
// [5] => be
// )
```



Несмотря на то что `explode()` функционирует аналогично `str_split()`, она не может разбить строку с пустым разделителем (первый параметр функции). Если вы попытаетесь передать пустую строку, то получите ошибку `ValueError`. Если вам необходимо работать с массивом символов, используйте `str_split()`.

Для объединения массива строк в одну используется функция `join()`, которая, по сути, является лишь псевдонимом `implode()`. Однако она гораздо мощнее, чем простое обратное преобразование `str_split()`, поскольку вы можете задать разделитель, который будет помещен между вновь объединенными фрагментами кода.

Разделитель необязателен, но долгое наследие `implode()` в PHP привело к появлению двух несколько неочевидным сигнатурам функций:

```
implode(string $separator, array $array): string
```

```
implode(array $array): string
```

Если вам нужно просто объединить массив символов в строку, вы можете сделать это с помощью эквивалентных методов, показанных в примере 7.16.

#### **Пример 7.16.** Создание строки из массива символов

```
$array = ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'];
$option1 = implode($array);
$option2 = implode('', $array);
echo 'Эти два варианта ' . ($option1 === $option2 ? 'одинаковые' : 'разные');

// Эти два варианта одинаковые
```

Поскольку допускается явно указать разделитель — «клей», используемый для соединения каждого фрагмента текста, — возможности `implode()` практически неограниченны. Предположим, что ваш массив — это список слов, а не символов. Вы можете использовать `implode()`, чтобы соединить их в виде списка, разделенного запятыми, как в следующем примере:

```
$fruit = ['яблоко', 'апельсин', 'груша', 'персик'];
echo implode(', ', $fruit);

// яблоко, апельсин, груша, персик
```

## Читайте также

Документация по `implode()` (<https://oreil.ly/mpdcI>), `explode()` ([https://oreil.ly/PScj\\_](https://oreil.ly/PScj_)) и `str_split()` (<https://oreil.ly/2dTMD>).

# 7.9. Реверсирование массива

## Задача

Вы хотите изменить порядок элементов в массиве.

## Решение

Используйте `array_reverse()` следующим образом:

```
$array = ['пять', 'четыре', 'три', 'два', 'один', 'ноль'];  
  
$reversed = array_reverse($array);
```

## Обсуждение

Функция `array_reverse()` создает новый массив, в котором каждый элемент расположен в обратном порядке, по сравнению с изначальным массивом. По умолчанию эта функция не сохраняет числовые ключи исходного массива, а вместо этого заново индексирует каждый элемент. Нечисловые ключи (в ассоциативных массивах) остаются неизменными в результате переиндексации, однако их порядок, как и ожидалось, меняется на обратный. Пример 7.17 демонстрирует, как ассоциативные массивы переупорядочиваются с помощью `array_reverse()`.

### Пример 7.17. Реверсирование ассоциативных массивов

```
$array = ['a' => 'A', 'b' => 'B', 'c' => 'C'];  
  
$reversed = array_reverse($array);  
  
print_r($reversed);  
  
// Array  
// (  
//     [c] => C  
//     [b] => B  
//     [a] => A  
// )
```

Поскольку в ассоциативных массивах допускаются числовые ключи, переиндексация может привести к неожиданным результатам. К счастью, такой шаблон можно отключить, передав необязательный логический параметр в качестве второго

аргумента при реверсировании массива. В примере 7.18 показано, как это поведение индексации влияет на такие массивы (и как его можно отключить).

**Пример 7.18.** Реверсирование ассоциативного массива с числовыми ключами

```
$array = ['a' => 'A', 'b' => 'B', 42 => 'C', 'd' => 'D'];
print_r(array_reverse($array)); ❶

// Array
// (
//     [d] => D
//     [0] => C
//     [b] => B
//     [a] => A
// )

print_r(array_reverse($array, true)); ❷
// Array
// (
//     [d] => D
//     [42] => C
//     [b] => B
//     [a] => A
// )
```

❶ По умолчанию значение второго параметра равно `false`, это означает, что числовые ключи не будут сохранены после обращения массива.

❷ Передача `true` в качестве второго параметра все равно преобразует массив, но сохранит числовые ключи в новом массиве.

## Читайте также

Документация по `array_reverse()` (<https://oreil.ly/mI5eG>).

## 7.10. Сортировка массива

### Задача

Вы хотите отсортировать элементы массива.

### Решение

Чтобы отсортировать элементы на основе стандартных правил сравнения в PHP, используйте `sort()` следующим образом:

```
$states = ['Орегон', 'Калифорния', 'Аляска', 'Вашингтон', 'Гавайи'];
sort($states);
```

## Обсуждение

Встроенная система сортировки PHP основана на алгоритме быстрой сортировки Quicksort. По умолчанию он использует правила, определенные операторами сравнения PHP, чтобы задать порядок следования каждого элемента в массиве<sup>1</sup>. Однако вы можете произвести сортировку с использованием других правил, передавая флаг в качестве необязательного второго параметра `sort()`. Доступные флаги сортировки описаны в табл. 7.1.

**Таблица 7.1.** Флаги типов сортировки

Флаг	Описание
<code>SORT_REGULAR</code>	Обычное сравнение элементов, используются операции сравнения по умолчанию
<code>SORT_NUMERIC</code>	Числовое сравнение элементов
<code>SORT_STRING</code>	Строковое сравнение элементов
<code>SORT_LOCALE_STRING</code>	Сравнение элементов как строк на основе текущего системного языка
<code>SORT_NATURAL</code>	Сравнение элементов как строки, используя «естественный порядок»
<code>SORT_FLAG_CASE</code>	Можно объединять с <code>SORT_STRING</code> или <code>SORT_NATURAL</code> с помощью побитового оператора OR для сортировки строк без учета регистра

Флаги типов сортировки полезны, когда сортировка по умолчанию приводит к отсортированному массиву, который не имеет смысла. Например, сортировка массива целых чисел, как если бы они были строками, приведет к некорректному результату. Использование флага `SORT_NUMERIC` гарантирует, что целые числа будут отсортированы в правильном порядке. Пример 7.19 демонстрирует, чем отличаются эти два типа сортировки.

**Пример 7.19.** Сортировка целых чисел с помощью обычного и числового типа сортировки

```
$numbers = [1, 10, 100, 5, 50, 500];
sort($numbers, SORT_STRING);
print_r($numbers);

// Array
// (
//     [0] => 1
//     [1] => 10
//     [2] => 100
```

<sup>1</sup> Дополнительные сведения об операторах сравнения и их использовании см. в разделе «Операторы сравнения».

```
//      [3] => 5
//      [4] => 50
//      [5] => 500
// )

sort($numbers, SORT_NUMERIC);
print_r($numbers);

// Array
// (
//      [0] => 1
//      [1] => 5
//      [2] => 10
//      [3] => 50
//      [4] => 100
//      [5] => 500
// )
```

Функция `sort()` игнорирует ключи и индексы массива и сортирует элементы массива исключительно по их значениям. Таким образом, попытка использовать `sort()` для сортировки ассоциативного массива приведет к уничтожению ключей в этом массиве. Если вы хотите сохранить ключи в массиве и при этом отсортировать его по значениям, воспользуйтесь функцией `asort()`.

Для этого вызовите `asort()` точно так же, как и `sort()`; вы даже можете применить флаги из табл. 7.1. Однако в результирующем массиве останутся те же ключи, что и раньше, хотя элементы будут расположены в другом порядке. Например:

```
$numbers = [1, 10, 100, 5, 50, 500];
asort($numbers, SORT_NUMERIC); print_r($numbers);

// Array
// (
//      [0] => 1
//      [3] => 5
//      [1] => 10
//      [4] => 50
//      [2] => 100
//      [5] => 500
// )
```

И `sort()`, и `asort()` создают массивы, отсортированные по возрастанию. Если требуется получить массив в порядке убывания, у вас есть два варианта:

- отсортировать массив по возрастанию, а затем реверсировать его, как показано в рецепте 7.9;
- применить функции `rsort()` или `arsort()` для числовых и ассоциативных массивов соответственно.

Чтобы уменьшить общую сложность кода, последний вариант часто предпочтительнее. Функции имеют те же сигнатуры, что и `sort()` и `asort()`, но просто меняют порядок расположения элементов в результирующем массиве.

## Читайте также

Документация по функциям `arsort()` (<https://oreil.ly/G14ve>), `asort()` (<https://oreil.ly/jkl5w>), `rsort()` (<https://oreil.ly/Z6p49>) и `sort()` (<https://oreil.ly/sHWtt>).

# 7.11. Сортировка массива на основе функции

## Задача

Вам требуется отсортировать массив на основе пользовательской функции или сравнения.

## Решение

Используйте `usort()` с пользовательским обратным вызовом сортировки:

```
$bonds = [
    ['имя' => 'Шон',      'фамилия' => 'Коннери'],
    ['имя' => 'Дэниел',   'фамилия' => 'Крейг'],
    ['имя' => 'Пирс',     'фамилия' => 'Броснан'],
    ['имя' => 'Роджер',   'фамилия' => 'Мур'],
    ['имя' => 'Тимоти',   'фамилия' => 'Далтон'],
    ['имя' => 'Джордж',   'фамилия' => 'Лэзенби'],
];

function sorter(array $a, array $b) {
    return [$a['фамилия'], $a['имя']] <=> [$b['фамилия'], $b['имя']];
}

usort($bonds, 'sorter');

foreach ($bonds as $bond) {
    echo "{$bond['фамилия']}. {$bond['имя']} {$bond['фамилия']}". PHP_EOL;
}
```

## Обсуждение

Функция `usort()` использует определяемую пользователем функцию в качестве операции сравнения в своем алгоритме сортировки. Вы можете передать любую вызываемую функцию в качестве второго параметра, и каждый элемент массива

будет проверен через эту функцию, чтобы определить его правильный порядок. В нашем примере обратный вызов упоминается по имени, но вы можете с тем же успехом передать и анонимную функцию.

Здесь используется новый оператор PHP spaceship для проведения сложного сравнения между элементами массива<sup>1</sup>. В данном конкретном случае требуется отсортировать актеров, игравших Джеймса Бонда, сначала по фамилии, затем по имени. Эту же функцию можно использовать для любой коллекции имён.

Более типичный пример — применение пользовательской сортировки дат в PHP. Даты относительно легко сортировать, поскольку они являются частью непрерывного ряда. Но можно определить пользовательское поведение, которое нарушает эти ожидания. В примере 7.20 сделана попытка отсортировать массив дат сначала по дню недели, потом по году, затем по месяцу.

#### Пример 7.20. Пользовательская сортировка, применяемая к датам

```
$dates = [
    new DateTime('2022-12-25'),
    new DateTime('2022-04-17'),
    new DateTime('2022-11-24'),
    new DateTime('2023-01-01'),
    new DateTime('2022-07-04'),
    new DateTime('2023-02-14'),
];

function sorter(DateTime $a, DateTime $b) {
    return
        [$a->format('N'), $a->format('Y'), $a->format('j')]
        <=>
        [$b->format('N'), $b->format('Y'), $b->format('j')];
}

usort($dates, 'sorter');

foreach ($dates as $date) {
    echo $date->format('l, F jS, Y') . PHP_EOL;
}

// Monday, July 4th, 2022
// Tuesday, February 14th, 2023
// Thursday, November 24th, 2022
// Sunday, April 17th, 2022
// Sunday, December 25th, 2022
// Sunday, January 1st, 2023
```

---

<sup>1</sup> Оператор spaceship подробно рассматривается в рецепте 2.4, где также приводится пример использования usort().

Как и многие другие функции взаимодействия с массивами, рассмотренные в этой главе, `usort()` игнорирует ключи/индексы массива и в процессе работы переиндексирует массив. Если вам требуется сохранить их, воспользуйтесь функцией `uasort()`. Эта функция имеет ту же сигнатуру, что и `usort()`, но оставляет ключи массива нетронутыми после сортировки.

Ключи массива часто содержат важную информацию о данных в массиве, поэтому их сохранение во время сортировки иногда может оказаться критически важным. Кроме того, вы можете захотеть отсортировать массив по ключам, а не по значению каждого элемента. В таких случаях следует применить функцию `uksort()`.

Функция `uksort()` отсортирует массив по ключам, используя определенную вами функцию. Например, `uasort()`, она учитывает ключи и оставляет их на месте после сортировки.

## Читайте также

Документация по `usort()` (<https://oreil.ly/TuK1L>), `uasort()` (<https://oreil.ly/igH5E>) и `uksort()` (<https://oreil.ly/MEyff>).

# 7.12. Случайный порядок элементов в массиве

## Задача

Вы хотите перемешать элементы массива так, чтобы их порядок был случайным.

## Решение

Используйте функцию `shuffle()`:

```
$array = range('a', 'e');
shuffle($array);
```

## Обсуждение

Функция `shuffle()` работает с существующим массивом, который передается в функцию по ссылке. Она полностью игнорирует ключи массива и сортирует значения элементов по порядку, обновляя массив сразу. После перетасовки ключи массива заново индексируются, начиная с 0.



Хотя при перемешивании ассоциативного массива не высветится ошибка, вся информация о ключах будет потеряна во время операции. Перемешивать следует только числовые массивы.

Внутри `shuffle()` используется генератор псевдослучайных чисел Mersenne Twister для определения нового, кажущегося случайным порядка для каждого элемента в массиве. Этот генератор не подходит, когда требуется истинная случайность (например, в криптографии или сценариях безопасности), но это эффективный способ быстро перетасовать содержимое массива.

## Читайте также

Документация по функции `shuffle()` (<https://oreil.ly/AkcpO>).

## 7.13. Применение функции к каждому элементу массива

### Задача

Вы хотите преобразовать массив, применив функцию для изменения каждого элемента массива по очереди.

### Решение

Чтобы изменить массив, используйте функцию `array_walk()`:

```
$values = range(2, 5);

array_walk($values, function(&$value, $key) {
    $value *= $value;
});

print_r($values);

// Array
// (
//     [0] => 4
//     [1] => 9
//     [2] => 16
//     [3] => 25
// )
```

## Обсуждение

Перебор коллекций данных — обычное требование для PHP-приложений. Например, вам может потребоваться использовать коллекции для определения повторяющихся задач. Или выполнить определенную операцию над каждым элементом коллекции, например возвести значения в квадрат, как показано выше.

Функция `array_walk()` позволяет задать нужное преобразование и применить его к значению каждого элемента массива. Функция обратного вызова (второй параметр) принимает три аргумента: значение и ключ для элемента массива и необязательный аргумент `$arg`, который задается при первоначальном вызове `array_walk()` и передается при каждом использовании обратного вызова. Это эффективный способ передать постоянное значение в обратный вызов, как показано в примере 7.21.

### Пример 7.21. Вызов `array_walk()` с дополнительным аргументом

```
function mutate(&$value, $key, $arg)
{
    $value *= $arg;
}

$values = range(2, 5);

array_walk($values, 'mutate', 10);

print_r($values);

// Array
// (
//     [0] => 20
//     [1] => 30
//     [2] => 40
//     [3] => 50
// )
```

Использование функции `array_walk()` для модификации массива требует передачи его значений по ссылке в обратный вызов (обратите внимание на дополнительный символ & перед именем аргумента). Эту функцию также можно использовать, чтобы просто пройтись по каждому элементу массива и выполнить какую-либо другую функцию, не изменяя исходный массив. На самом деле это наиболее распространенное применение данной функции.

Помимо перехода по каждому элементу массива, вы можете пройтись по листьям узлов вложенного массива с помощью `array_walk_recursive()`. В отличие от предыдущих примеров, `array_walk_recursive()` будет обходить вложенные массивы, пока не найдет элемента, не относящегося к массиву, прежде чем применить

указанную вами функцию обратного вызова. В примере 7.22 наглядно продемонстрирована разница между рекурсивными и нерекурсивными вызовами функций для вложенного массива. В частности, если вы имеете дело с вложенным массивом, `array_walk()` выдаст ошибку и вообще ничего не сделает.

**Пример 7.22.** Сравнение `array_walk()` с `array_walk_recursive()`

```
$array = [
    'even' => [2, 4, 6],
    'odd'  => 1,
];

function mutate(&$value, $key, $arg)
{
    $value *= $arg;
}

array_walk_recursive($array, 'mutate', 10);
print_r($array);

// Array
// (
//     [even] => Array
//         (
//             [0] => 20
//             [1] => 40
//             [2] => 60
//         )
//     [
//         [odd] => 10
//     ]
// )

array_walk($array, 'mutate', 10);

// PHP Warning: Uncaught TypeError: Unsupported operand types: array * int
```

Иногда может возникнуть потребность создать новую копию измененного массива, не потеряв при этом его исходного состояния. В таких обстоятельствах более безопасным выбором оказывается `array_map()`, чем `array_walk()`. Вместо того чтобы изменять исходный массив, `array_map()` позволяет применить функцию к каждому элементу исходного массива и вернуть совершенно новый массив. Преимущество заключается в том, что для дальнейшего использования вам будут доступны как исходный, так и модифицированный массивы. В следующем примере используется та же логика, что и в «Решении», без изменения исходного массива:

```
$values = range(2, 5);

$mutated = array_map(function($value) {
    return $value * $value;
}, $values);
```

```
print_r($mutated);

// Array
// (
//     [0] => 4
//     [1] => 9
//     [2] => 16
//     [3] => 25
// )
```

Вот некоторые ключевые различия между этими двумя семействами функций массивов:

- `array_walk()` ожидает сначала массив, а затем обратный вызов;
- `array_map()` ожидает, что первым будет обратный вызов, а вторым — массив;
- `array_walk()` возвращает логический флаг, а `array_map()` возвращает новый массив;
- `array_map()` не передает ключи в обратный вызов;
- `array_map()` не передает дополнительные аргументы в обратный вызов;
- у `array_map()` не существует рекурсивной формы.

## Читайте также

Документация по `array_map()` ([https://oreil.ly/fzU\\_0](https://oreil.ly/fzU_0)), `array_walk()` (<https://oreil.ly/OTpL4>) и `array_walk_recursive()` (<https://oreil.ly/qCt7G>).

# 7.14. Сокращение массива до одного значения

## Задача

Вы хотите итеративно свести коллекцию значений к единственному значению.

## Решение

Используйте `array_reduce()` с обратным вызовом, например:

```
$values = range(0, 10);

$sum = array_reduce($values, function($carry, $item) {
    return $carry + $item;
}, 0);

// $sum = 55
```

## Обсуждение

Функция `array_reduce()` проходит по каждому элементу массива и изменяет свое внутреннее состояние, чтобы в итоге прийти к единому ответу. Функция из примера выше проходит через каждый элемент списка чисел и прибавляет их к начальному значению 0, возвращая конечную сумму всех чисел, о которых идет речь.

Функция обратного вызова принимает два параметра. Первый — это значение, которое вы передаете с последней операции. Второй — значение текущего элемента в массиве, над которым выполняется итерация. Что бы ни возвращал обратный вызов, будет передано в обратный вызов в качестве параметра `$carry` для следующего элемента массива.

Когда только начинаете, вы передаете необязательное начальное значение (по умолчанию `null`) в обратный вызов в качестве параметра `$carry`. Если операция сокращения, которую вы применяете к массиву, является простой, допускается указать лучшее начальное значение, как сделано в примере из «Решения».

Самый большой недостаток функции `array_reduce()` заключается в том, что она не работает с ключами массива. Чтобы задействовать какие-либо ключи в массиве в операции сокращения, необходимо определить собственную версию функции.

В примере 7.23 показано, как выполнить итерацию по массиву, возвращенному функцией `array_keys()`, чтобы использовать ключи и значения элементов в процессе сокращения. Вы передаете и массив, и обратный вызов в замыкание, обрабатываемое `array_reduce()`, чтобы одновременно ссылаться на элемент в массиве, определенном этим ключом, и применять к нему собственные функции. В основной программе вы вольны сокращать ассоциативный массив так же, как и числовой, — за исключением того, что в обратном вызове у вас будет дополнительный аргумент, содержащий ключ каждого элемента.

### Пример 7.23. Ассоциативная альтернатива `array_reduce()`

```
function array_reduce_assoc(
    array $array,
    callable $callback,
    mixed $initial = null
): mixed
{
    return array_reduce(
        array_keys($array),
        function($carry, $item) use ($array, $callback) {
            return $callback($carry, $array[$item], $item);
        },
        $initial
    );
}
```

```
$array = [1 => 10, 2 => 10, 3 => 5];

$sumMultiples = array_reduce_assoc(
    $array,
    function($carry, $item, $key) {
        return $carry + ($item * $key);
    }, 0
);

// $sumMultiples = 45
```

Предыдущий код вернет сумму ключей \$array, умноженную на их соответствующие значения:  $1 * 10 + 2 * 10 + 3 * 5 = 45$ .

## Читайте также

Документация по `array_reduce()` ([https://oreil.ly/iu\\_XM](https://oreil.ly/iu_XM)).

# 7.15. Итерация по бесконечным или очень большим/ресурсозатратным массивам

## Задача

Вам нужно перебрать список элементов, который слишком велик для хранения в памяти или слишком медленно генерируется.

## Решение

Используйте генератор, чтобы выдавать программе по одному фрагменту данных за раз, как показано ниже:

```
function weekday()
{
    static $day = 'Понедельник';

    while (true) {
        yield $day;

        switch($day) {
            case 'Понедельник':
                $day = 'Вторник';
                break;
            case 'Вторник':
                $day = 'Среда';
```

```
        break;
    case 'Среда':
        $day = 'Четверг';
        break;
    case 'Четверг':
        $day = 'Пятница';
        break;
    case 'Пятница':
        $day = 'Понедельник';
        break;
    }
}
}

$weekdays = weekday();
foreach ($weekdays as $day) {
    echo $day . PHP_EOL;
}
```

## Обсуждение

Генераторы — это эффективный инструмент для обработки больших массивов данных в PHP. В приведенном примере генератор выводит дни недели (с понедельника по пятницу) в виде бесконечного ряда. Он не поместится в памяти, доступной PHP, но конструкция генератора позволяет создавать его по частям за раз.

Вместо того чтобы создавать слишком большой массив, вы генерируете первый фрагмент данных и возвращаете его с помощью ключевого слова `yield` тому, кто вызвал генератор. Это замораживает состояние генератора и возвращает контроль над исполнением обратно основному приложению. В отличие от обычной функции, которая возвращает данные один раз, генератор предоставляет их неоднократно, пока они остаются действительными.

В примере из «Решения» ключевое слово `yield` находится внутри бесконечного цикла `while`, поэтому перечисление дней недели будет продолжаться бесконечно. Чтобы генератор завершил свою работу, воспользуйтесь пустым оператором `return` в конце (или просто прервите цикл с неявным возвратом).



Возврат данных из генератора отличается от обычного вызова функции. Чаще всего данные возвращаются при использовании ключевого слова `yield`, а генератор завершается с пустым оператором `return`. Однако, если генератор имеет окончательный возврат, вы должны получить доступ к этим данным с помощью функции `::getReturn()` для объекта генератора. Этот дополнительный вызов метода часто кажется странным, поэтому, если ваш генератор не должен возвращать данные вне его типичной операции `yield`, старайтесь избегать этого.

Поскольку генератор способен предоставлять данные бесконечно, вы можете выполнять итерации по этим данным благодаря стандартному циклу `foreach`. Аналогичным образом допускается использовать ограниченный цикл `for`, чтобы избежать бесконечной серии. В следующем коде задействован такой ограниченный цикл и оригинальный генератор из «Решения»:

```
$weekdays = weekday();
for ($i = 0; $i < 14; $i++) {
    echo $weekdays->current() . PHP_EOL;
    $weekdays->next();
}
```

Хотя генератор определен как функция, PHP распознает его как генератор и преобразует в экземпляр класса `Generator` ([https://oreil.ly/R\\_geQ](https://oreil.ly/R_geQ)). Этот класс дает вам доступ к методам `:current()` и `:next()` и позволяет перебирать сгенерированные данные по одной единице за раз.

Управление внутри приложения передается туда и обратно между основной программой и оператором `yield` генератора. При первом обращении к генератору он выполняет операции до ключевого слова `yield`, а затем возвращает управление (и, возможно, данные) основному приложению. Последующие обращения к генератору начинаются после ключевого слова `yield`. Для того чтобы генератор снова начал работу и вернулся к началу, необходимы циклы.

## Читайте также

Обзор генераторов (<https://oreil.ly/cR4-V>).

## ГЛАВА 8

---

# Классы и объекты

Самые ранние версии PHP не поддерживали определения классов или объектно-ориентированное программирование. PHP 4 стал первой реальной попыткой внедрения объектного интерфейса<sup>1</sup>. Однако по-настоящему объектно-ориентированное программирование, которое мы знаем сегодня, пришло в PHP только с версией 5.

Классы определяются с помощью ключевого слова `class`, за которым следует полное описание констант, свойств и методов, присущих классу. В примере 8.1 представлена базовая конструкция класса в PHP, включающая константное значение, свойство и вызываемые методы.

### Пример 8.1. Базовый класс PHP со свойствами и методами

```
class Foo
{
    const SOME_CONSTANT = 42;

    public string $hello = 'привет';

    public function __construct(public string $world = 'мир') {}

    public function greet(): void
    {
        echo sprintf('%s %s', $this->hello, $this->world);
    }
}
```

Объект может быть инстанцирован с помощью ключевого слова `new` и имени класса; создание объекта похоже на вызов функции. Любые параметры, передаваемые в эту функцию, при инстанцировании прозрачно передаются в конструктор класса

---

<sup>1</sup> PHP 3 включал в себя некоторую примитивную объектную функциональность, но большинство разработчиков не считали язык объектно-ориентированным до выхода версии 4.0.

(метод `__construct()`) для определения начального состояния объекта. Пример 8.2 иллюстрирует, как можно создать экземпляр класса, определенного в примере 8.1, со значениями свойств по умолчанию и без них.

### Пример 8.2. Инстанцирование базового класса PHP

```
$first = new Foo; ❶
$second = new Foo('вселенная'); ❷

$first->greet(); ❸
$second->greet(); ❹

echo Foo::SOME_CONSTANT; ❺
```

❶ Инстанцирование объекта без передачи параметра все равно вызовет конструктор, но при этом будут использованы параметры по умолчанию. Если в сигнатуре функции не указаны параметры по умолчанию, это приведет к ошибке.

❷ Передача параметра во время инстанцирования предоставит этот параметр конструктору.

❸ Выводит привет мир, используя настройки конструктора по умолчанию.

❹ Выводит привет вселенная в консоль.

❺ На константы ссылаются непосредственно из имени класса. В результате будет выведен литерал 42 в консоль.

Конструкторы и свойства рассматриваются в рецептах 8.1 и 8.2.

## Процедурное программирование

Большинство разработчиков впервые знакомятся с PHP через его процедурные интерфейсы. Примеры процедур, простые скрипты, учебники — все они обычно используют функции и переменные, определенные в глобальной области видимости. Это неплохо, но ограничивает гибкость программ, которые вы можете создавать.

Процедурное программирование часто приводит к созданию приложений без статичных данных. У вас практически нет возможности отслеживать, что происходило ранее между вызовами функций, поэтому вы передаете некоторую ссылку на состояние приложения по всему коду. И опять же, это не обязательно плохо. Единственный недостаток заключается в том, что крупные приложения становятся трудно анализировать и понимать.

## Объектно-ориентированное программирование

Альтернативной парадигмой становится использование объектов в качестве контейнеров состояния. Обычный практический пример — рассматривать объекты как способы определения предметов. Автомобиль — это объект. Как и автобус. И велосипед. Это дискретные предметы, которые имеют характеристики (такие как цвет, количество колес и тип привода) и возможности (такие как ехать, останавливаться и поворачивать).

В мире программирования это один из самых простых способов описания объектов. В PHP вы создаете объекты, сначала определив класс, чтобы описать тип объекта. Класс описывает свойства (характеристики) и методы (возможности), которыми будет обладать объект данного типа.

Как и вещи в реальном мире, объекты в контексте программирования могут наследоваться от более примитивных описаний типов. Автомобиль, автобус и велосипед — это все типы транспортных средств, поэтому все они могут происходить от определенного типа. Пример 8.3 демонстрирует, как в PHP может быть организовано наследование объектов такого типа.

### Пример 8.3. Абстракция класса в PHP

```
abstract class Vehicle
{
    abstract public function go(): void;
    abstract public function stop(): void;
    abstract public function turn(Direction $direction): void;
}

class Car extends Vehicle
{
    public int $wheels = 4;
    public string $driveType = 'gas';

    public function __construct(public Color $color) {}
    public function go(): void
    {
        // ...
    }

    public function stop(): void
    {
        // ...
    }
}
```

```
public function turn(Direction $direction): void
{
    // ...
}

class Bus extends Vehicle
{
    public int $wheels = 4;
    public string $driveType = 'diesel';

    public function __construct(public Color $color) {}

    public function go(): void
    {
        // ...
    }

    public function stop(): void
    {
        // ...
    }

    public function turn(Direction $direction): void
    {
        // ...
    }
}

class Bicycle extends Vehicle
{
    public int $wheels = 2;
    public string $driveType = 'direct';

    public function __construct(public Color $color) {}

    public function go(): void
    {
        // ...
    }

    public function stop(): void
    {
        // ...
    }

    public function turn(Direction $direction): void
    {
        // ...
    }
}
```

При инстанцировании объекта создается типизированная переменная, которая представляет собой как начальное состояние, так и методы для манипулирования этим состоянием. Наследование объектов дает возможность использовать один или несколько типов в качестве альтернативы друг другу в другом коде. В примере 8.4 показано, как три типа транспортных средств, описанных в примере 8.2, могут использоваться взаимозаменяя благодаря наследованию.

#### Пример 8.4. Взаимозаменяемость при наследовании

```
function commute(Vehicle $vehicle) ①
{
    // ...
}

function exercise(Bicycle $vehicle) ②
{
    // ...
}
```

❶ Все три подтипа транспортных средств можно использовать в качестве допустимых замен `Vehicle` в вызовах функций. Это означает, что для передвижения вы можете задействовать `Bus`, `Car` или `Bicycle`, и любой выбор будет одинаково актуальным.

❷ В некоторых случаях могут потребоваться более точная формулировка и прямое использование дочернего типа. Ни `Bus`, ни `Car`, ни любой другой подкласс класса `Vehicle` не подойдет для выполнения упражнений, кроме `Bicycle`.

Наследование классов подробно рассматривается в рецептах 8.6, 8.7 и 8.8.

## Мультипарадигмальные языки

PHP считается мультипарадигмальным языком, поскольку вы можете писать код, следуя любой из предыдущих парадигм. Программа на PHP может быть чисто процедурной или же сфокусирована на определении объектов и пользовательских классов. В конце концов, она может использовать смесь обеих парадигм.

Система управления контентом (CMS) WordPress с открытым исходным кодом — один из самых популярных PHP-проектов в интернете<sup>1</sup>. Она написана для актив-

---

<sup>1</sup> На момент написания книги WordPress использовался на 43 % всех сайтов (<https://oreil.ly/tEaN8>).

ного использования объектов для таких абстракций, как объекты базы данных или удаленные запросы. Однако WordPress также имеет долгую историю процедурного программирования — большая часть кодовой базы все еще находится под сильным влиянием этого стиля. WordPress — это ключевой пример не только успеха самого PHP, но и гибкости языка, поддерживающего различные парадигмы.

Не существует единственно верного ответа на вопрос, как должно быть собрано приложение. Большинство из них представляют собой гибриды подходов, которые выигрывают от сильной поддержки PHP нескольких парадигм. Тем не менее даже в приложениях, где преобладает процедурный подход, вы, скорее всего, увидите несколько объектов, поскольку именно так стандартная библиотека языка реализует большую часть своей функциональности (<https://oreil.ly/krXrW>).

В главе 6 было показано использование как функциональных, так и объектно-ориентированных интерфейсов для системы дат в PHP. Обработка ошибок, которая более подробно рассматривается в главе 13, в значительной степени опирается на внутренние классы `Exception` и `Error`. Ключевое слово `yield` в процедурной реализации автоматически создает экземпляры класса `Generator`.

Даже если вы никогда не определяете класс в своей программе напрямую, велика вероятность, что вы будете использовать класс, определенный либо самим PHP, либо сторонней библиотекой, которая требуется вашей программе<sup>1</sup>.

## Видимость

Классы также вводят в PHP концепцию видимости. Свойства, методы и даже константы могут быть определены с дополнительным модификатором видимости, чтобы изменить уровень доступа к ним из других частей приложения. Все, что объявлено как `public`, доступно любому другому классу или функции в вашем приложении. Методы и свойства могут быть объявлены как `protected`, что делает их доступными только для экземпляра самого класса или классов, которые наследуют его. Наконец, ключевое слово `private` означает, что доступ к члену класса могут получить только экземпляры самого класса.



По умолчанию все, что неявно определено как `private` или `protected`, автоматически становится `public`, поэтому некоторые разработчики пропускают объявления видимости членов.

<sup>1</sup> Библиотеки и расширения подробно рассматриваются в главе 15.

Хотя видимость членов может быть непосредственно переопределена с помощью рефлексии<sup>1</sup>, это, как правило, надежный способ уточнить, какие части интерфейса класса должны использоваться другими элементами кода. В примере 8.5 показано, как можно использовать каждый модификатор видимости для создания сложного приложения.

**Пример 8.5.** Обзор видимости членов класса

```
class A
{
    public      string $name = 'Боб';
    public      string $city = 'Портленд';
    protected   int    $year = 2023;
    private     float  $value = 42.9;

    function hello(): string
    {
        return 'привет';
    }

    public function world(): string
    {
        return 'мир';
    }

    protected function universe(): string
    {
        return 'вселенная';
    }

    private function abyss(): string
    {
        return 'пустота';
    }
}

class B extends A
{
    public function getName(): string
    {
        return $this->name;
    }

    public function getCity(): string
    {
        return $this->city;
    }
}
```

---

<sup>1</sup> Подробнее об API Reflection см. в рецепте 8.12.

```

public function getYear(): int
{
    return $this->year;
}

public function getValue(): float
{
    return $this->value;
}

}

$first = new B;
echo $first->getName() . PHP_EOL; ①
echo $first->getCity() . PHP_EOL; ②
echo $first->getYear() . PHP_EOL; ③
echo $first->getValue() . PHP_EOL; ④

$second = new A;
echo $second->hello() . PHP_EOL; ⑤
echo $second->world() . PHP_EOL; ⑥
echo $second->universe() . PHP_EOL; ⑦
echo $second->abyss() . PHP_EOL; ⑧

```

- ① Выводит Боб.
- ② Выводит Портленд.
- ③ Выводит 2023.
- ④ Возвращает Warning, так как свойство `::$value` является закрытым и недоступным.
- ⑤ Выводит привет.
- ⑥ Выводит мир.
- ⑦ Выбрасывает Error, так как метод `::universe()` защищен и недоступен за пределами экземпляра класса.
- ⑧ Этот код даже не будет выполнен из-за ошибки, возникшей в предыдущей строке. Если предыдущая строка не выдала ошибку, то эта выдаст, поскольку метод `::abyss()` является закрытым и недоступен за пределами экземпляра класса.

Следующие рецепты иллюстрируют предыдущие концепции и охватывают некоторые из наиболее распространенных сценариев использования и реализации объектов в PHP.

## 8.1. Инстанцирование объектов из пользовательских классов

### Задача

Вы хотите определить пользовательский класс и создать на его основе новый экземпляр объекта.

### Решение

Определите класс, его свойства и методы с помощью ключевого слова `class`, а затем используйте `new`, чтобы создать его экземпляр:

```
class Pet
{
    public string $name;
    public string $species;
    public int $happiness = 0;

    public function __construct(string $name, string $species)
    {
        $this->name = $name;
        $this->species = $species;
    }

    public function pet()
    {
        $this->happiness += 1;
    }
}

$dog = new Pet('Fido', 'golden retriever');
$dog->pet();
```

### Обсуждение

Пример выше иллюстрирует несколько ключевых характеристик объектов:

- объекты могут иметь свойства, которые определяют внутреннее состояние самого объекта;
- они могут иметь определенную видимость. В приведенном примере объекты публичные, то есть к ним может обращаться любой код в приложении<sup>1</sup>;

---

<sup>1</sup> См. раздел «Видимость» выше в этой главе, чтобы узнать больше о видимости в классах.

- магический метод `__construct()` может принимать параметры только при первом инстанцировании объекта. Эти параметры используются для определения начального состояния объекта;
- методы могут иметь видимость, аналогичную свойствам объекта.

Этот вариант определения класса многие разработчики используют по умолчанию со временем PHP 5, когда впервые появились настоящие объектно-ориентированные примитивы. Однако пример 8.6 демонстрирует новый и несложный способ определения простого объекта, подобного тому, что приведен в «Решении». Вместо того чтобы самостоятельно объявлять и затем напрямую присваивать свойства, которые несут в себе состояние объекта, PHP 8 (и более поздние версии) позволяет определить все в самом конструкторе.

#### Пример 8.6. Продвижение конструктора в PHP 8

```
class Pet
{
    public int $happiness = 0;

    public function __construct(
        public string $name,
        public string $species
    ) {}

    public function pet()
    {
        $this->happiness += 1;
    }
}

$dog = new Pet('Fido', 'golden retriever');
$dog->pet();
```

Пример из «Решения» и пример 8.6 функционально эквивалентны и приведут к созданию объектов с одинаковой внутренней структурой во время выполнения. Однако способность PHP передавать аргументы конструктора в свойства объекта значительно сокращает количество повторяющегося кода, который необходимо набирать при определении класса.

Каждый аргумент конструктора также допускает те же типы видимости (`public/protected/private`), что и свойства объекта<sup>1</sup>. Сокращенный синтаксис означает, что вам не нужно объявлять свойства, затем определять параметры, а потом сопоставлять параметры с этими свойствами при инстанцировании объекта.

<sup>1</sup> Подробнее о том, как эти свойства можно сделать доступными только для чтения, см. в рецепте 8.3.

## Читайте также

Документация по классам и объектам (<https://oreil.ly/TfrNb>), а также оригинальный RFC по продвижению конструкторов (<https://oreil.ly/nzD0s>).

## 8.2. Конструирование объектов для определения значений по умолчанию

### Задача

Вы хотите определить значения по умолчанию для свойств вашего объекта.

### Решение

Определите значения по умолчанию для аргументов конструктора следующим образом:

```
class Example
{
    public function __construct(
        public string $someString = 'default',
        public int     $someNumber = 5
    ) {}
}

$first  = new Example;
$second = new Example('overridden');
$third  = new Example('hitchhiker', 42);
$fourth = new Example(someNumber: 10);
```

### Обсуждение

Функция-конструктор в определении класса ведет себя примерно так же, как и любая другая функция в PHP, за исключением того, что она не возвращает значения. Вы можете определить аргументы по умолчанию так же, как и в стандартной функции. Разрешается даже ссылаться на имена аргументов конструктора, чтобы принимать значения по умолчанию для одних параметров, а другие определять позже в сигнатуре функции.

В примере выше свойства класса определены явно с помощью продвижения конструктора для краткости, но и более старое определение конструктора вполне допустимо:

```
class Example
{
    public string $someString;
    public int $someNumber;

    public function __construct(
        string $someString = 'default',
        int    $someNumber = 5
    )
    {
        $this->someString = $someString;
        $this->someNumber = $someNumber;
    }
}
```

Аналогично, если не использовать продвижение конструктора, можно инициализировать свойства объекта напрямую, присвоив им значение по умолчанию при их определении. При этом обычно оставляют эти параметры вне конструктора и управляют ими из другого места программы, как показано в следующем примере:

```
class Example
{
    public string $someString = 'default';
    public int $someNumber = 5;
}

$test = new Example;
$test->someString = 'overridden';
$test->someNumber = 42;
```



Как будет рассмотрено в рецепте 8.3, нельзя инициализировать свойство класса с модификатором `readonly` (только для чтения) непосредственно значением по умолчанию. Это эквивалентно константе класса, и такой синтаксис запрещен.

## Читайте также

Рецепт 3.2 — о параметрах функции по умолчанию, рецепт 3.3 — об именованных параметрах функции, а также документация по конструкторам и деструкторам (<https://oreil.ly/WJvYY>).

## 8.3. Определение свойств, доступных только для чтения, в классе

### Задача

Требуется определить класс таким образом, чтобы свойства, заданные при инстанцировании, оставались неизменяемыми после создания объекта.

### Решение

Используйте ключевое слово `readonly` для типизированного свойства:

```
class Book
{
    public readonly string $title;

    public function __construct(string $title)
    {
        $this->title = $title;
    }
}

$book = new Book('PHP Cookbook');
```

Если вы используете продвижение конструктора, поместите ключевое слово вместе с типом свойства внутри конструктора:

```
class Book
{
    public function __construct(public readonly string $title) {}
}

$book = new Book('PHP Cookbook');
```

### Обсуждение

Ключевое слово `readonly` было введено в PHP 8.1 и направлено на снижение количества кода, которое первоначально требовалось для достижения той же функциональности. С помощью этого ключевого слова свойство может быть инициализировано значением только один раз и только во время инстанцирования объекта.



Свойства, доступные только для чтения, не могут иметь значения по умолчанию. Это сделало бы их функционально эквивалентными константам класса, которые уже существуют, поэтому такая функциональность недоступна, а синтаксис не поддерживается. Однако свойства, продвигаемые через конструктор, могут использовать значения по умолчанию в определении аргумента, поскольку они оцениваются во время выполнения.

Это ключевое слово применимо только к типизированным свойствам. Типы обычно необязательны в PHP (за исключением случаев строгой типизации<sup>1</sup>) для обеспечения гибкости, поэтому возможно, что свойство вашего класса не может быть установлено каким-то типом. В подобных ситуациях используйте тип `mixed`, чтобы установить свойство «только для чтения» без других ограничений по типу.



На момент написания книги объявления `readonly` не поддерживаются для статических свойств.

Поскольку свойство `readonly` инстанцируется только один раз, его нельзя удалить или изменить другим последующим кодом. Весь код в примере 8.7 приведет к возникновению исключения `Error`.

#### Пример 8.7. Попытки изменить свойство `readonly` приводят к ошибке

```
class Example
{
    public readonly string $prop;
}

class Second
{
    public function __construct(public readonly int $count = 0) {}
}

$first = new Example; ①
$first->prop = 'test'; ②

$test = new Second; ③
$test->count += 1; ④
$test->count++; ⑤
++$test->count;
unset($test->count); ⑥
```

<sup>1</sup> Подробнее о строгой типизации см. в рецепте 3.4.

- ❶ Объект `Example` будет иметь неинициализированное свойство `::$prop`, к которому нельзя получить доступ (обращение к свойству до инициализации вызывает исключение `Error`).
- ❷ Поскольку объект уже инстанцирован, попытка записи в свойство, доступное только для чтения, приводит к ошибке.
- ❸ Свойство `::$count` доступно только для чтения, поэтому вы не можете присвоить ему новое значение без ошибки.
- ❹ Поскольку свойство `::$count` доступно только для чтения, вы не можете увеличивать его напрямую.
- ❺ Нельзя увеличить или уменьшить значение свойства только для чтения.
- ❻ Нельзя удалить свойство только для чтения.

Однако свойства внутри класса могут быть другими классами. В таких случаях объявление свойства `readonly` означает, что оно не может быть перезаписано или отменено, но это не влияет на свойства дочернего класса. Например:

```
class First
{
    public function __construct(public readonly Second $inner) {}
}

class Second
{
    public function __construct(public int $counter = 0) {}
}

$test = new First(new Second);
$test->inner->counter += 1; ❶
$test->inner = new Second; ❷
```

- ❶ Увеличение внутреннего счетчика выполнится успешно, поскольку свойство `::$counter` само по себе не объявлено как «доступное только для чтения».
- ❷ Свойство `::$inner` доступно только для чтения и не может быть переопределено. Попытки сделать это приводят к исключению `Error`.

## Читайте также

Документация по свойствам (<https://oreil.ly/P-AwN>), доступным только для чтения (<https://oreil.ly/P-AwN>).

## 8.4. Деконструкция объектов для очистки после того, как объект больше не нужен

### Задача

Определение вашего класса обертывает ресурсозатратный объект, который необходимо тщательно очистить, когда тот выходит за пределы области видимости.

### Решение

Определите деструктор класса для очистки после удаления объекта из памяти следующим образом:

```
class DatabaseHandler
{
    // ...
    public function __destruct()
    {
        dbo_close($this->dbh);
    }
}
```

### Обсуждение

Когда объект выходит из области видимости, PHP автоматически собирает весь «мусор» из памяти или других ресурсов, которые этот объект использовал для представления. Однако могут возникнуть ситуации, когда вам потребуется принудительно выполнить определенное действие, когда объект выходит из области видимости. Это может быть освобождение обработчика базы данных, как показано в примере выше, явная запись события в файл или, возможно, удаление временного файла из системы, как продемонстрировано в примере 8.8.

#### Пример 8.8. Удаление временного файла в деструкторе

```
class TempLogger
{
    private string $filename;
    private mixed $handle;

    public function __construct(string $name)
    {
        $this->filename = sprintf('tmp_%s_%s.tmp', $name, time());
        $this->handle = fopen($this->filename, 'w');
    }
}
```

```
public function writeLog(string $line): void
{
    fwrite($this->handle, $line . PHP_EOL);
}

public function getLogs(): Generator
{
    $handle = fopen($this->filename, 'r');
    while(($buffer = fgets($handle, 4096)) !== false) {
        yield $buffer;
    }
    fclose($handle);
}

public function __destruct()
{
    fclose($this->handle);
    unlink($this->filename);
}
}

$logger = new TempLogger('тест'); ❶
$logger->writeLog('Это тест'); ❷
$logger->writeLog('И еще');

foreach($logger->getLogs() as $log) { ❸
    echo $log;
}

unset($logger); ❹
```

❶ Объект автоматически создаст в текущем каталоге файл с именем, похожим на `tmp_test_1650837172.tmp`.

❷ Каждая новая запись в журнале фиксируется в виде новой строки во временном файле журнала.

❸ При доступе к журналу будет создан второй обработчик того же файла, но уже для чтения. Объект передает этот обработчик через генератор, который перечисляет каждую строку в файле.

❹ Когда регистратор выходит из области видимости (или явно удаляется), деструктор закрывает открытый файловый обработчик и автоматически удаляет файл.

Этот более сложный пример демонстрирует, как должен быть написан деструктор и как он будет вызван. PHP ищет метод `::__destruct()` для любого объекта,

когда он выходит из области видимости, и вызывает его в этот момент. Деструктор явно удаляет ссылку на объект, вызывая `unset()`, чтобы стереть его из программы. С тем же успехом можно было бы установить переменную, ссылка на объект которой равна `null`, с тем же результатом.

В отличие от конструкторов объектов, деструкторы не принимают никаких параметров. Если вашему объекту необходимо воздействовать на какое-либо внешнее состояние во время очистки после себя, убедитесь, что на это состояние есть ссылка через свойство самого объекта. В противном случае у вас не будет доступа к этой информации.

## Читайте также

Документация по конструкторам и деструкторам (<https://oreil.ly/fJMGm>).

# 8.5. Использование магических методов для предоставления динамических свойств

## Задача

Вы хотите определить пользовательский класс без предопределения свойств, которые он поддерживает.

## Решение

Используйте магические геттеры и сеттеры для обработки динамически определяемых свойств:

```
class Magical
{
    private array $_data = [];

    public function __get(string $name): mixed
    {
        if (isset($this->_data[$name])) {
            return $this->_data[$name];
        }

        throw new Error(sprintf('Свойство `%s` не определено', $name));
    }
}
```

```
public function __set(string $name, mixed $value)
{
    $this->_data[$name] = $value;
}
}

$first = new Magical;
$first->custom = 'hello';
$first->another = 'world';

echo $first->custom . ' ' . $first->another . PHP_EOL;

echo $first->unknown; // Ошибка
```

## Обсуждение

Когда вы ссылаетесь на свойство несуществующего объекта, PHP прибегает к помощи набора магических методов, чтобы заполнить пробелы в реализации. Геттер автоматически используется при попытке сослаться на свойство, в то время как соответствующий сеттер — при присваивании значения свойству.



Перегрузка свойств с помощью магических методов действует только на инстанцированных объектах. Она не работает на статическом определении класса.

Внутренне вы целиком контролируете поведение получения и установки данных. В примере из «Решения» данные хранятся в закрытом ассоциативном массиве. Вы можете доработать этот пример, полностью реализовав магические методы для обработки `isset()` и `unset()`. Пример 8.9 демонстрирует, как с помощью магических методов повторить стандартное определение класса, но без необходимости заранее объявлять все свойства.

**Пример 8.9.** Полное определение объекта с использованием магических методов

```
class Basic
{
    public function __construct(
        public string $word,
        public int $number
    ) {}
}
```

```
class Magic
{
    private array $_data = [];

    public function __get(string $name): mixed
    {
        if (isset($this->_data[$name])) {
            return $this->_data[$name];
        }

        throw new Error(sprintf('Свойство `%s` не определено', $name));
    }

    public function __set(string $name, mixed $value)
    {
        $this->_data[$name] = $value;
    }

    public function __isset(string $name): bool
    {
        return array_key_exists($name, $this->_data);
    }

    public function __unset(string $name): void
    {
        unset($this->_data[$name]);
    }
}

$basic = new Basic('test', 22);

$magic = new Magic;
$magic->word = 'test';
$magic->number = 22;
```

В примере 8.9 два объекта функционально эквивалентны тогда и только тогда, когда единственными динамическими свойствами, используемыми в экземпляре `Magic`, являются те, которые уже определены в `Basic`. Именно этот динамический характер делает подход столь ценным, даже если определения классов будут невероятно громоздкими. Вы можете обернуть удаленный API в класс, реализующий магические методы, чтобы предоставить данные этого API вашему приложению в объектно-ориентированном стиле.

## Читайте также

Документация по магическим методам (<https://oreil.ly/1ZtIE>).

## 8.6. Расширение классов для определения дополнительной функциональности

### Задача

Вы хотите задать класс, который добавляет функциональность к существующему определению класса.

### Решение

Используйте ключевое слово `extends` для определения дополнительных методов или переопределения существующих функций следующим образом:

```
class A
{
    public function hello(): string
    {
        return 'привет';
    }
}

class B extends A
{
    public function world(): string
    {
        return 'мир';
    }
}

$instance = new B();
echo "{$instance->hello()} {$instance->world()}";
```

### Обсуждение

Наследование объектов — распространенное понятие для любого языка высокого уровня. Это способ создания новых объектов на основе других, часто более простых определений объектов. Пример выше иллюстрирует, как класс может наследовать определения методов от родительского класса, что является основной функциональностью модели наследования PHP.



PHP не поддерживает наследования от нескольких родительских классов. Чтобы получить реализацию кода из нескольких источников, PHP использует трейты, которые рассматриваются в рецепте 8.13.

Фактически дочерний класс наследует все публичные и защищенные методы, свойства и константы от своего родительского класса (класса, который он расширяет). Закрытые методы, свойства и константы никогда не наследуются дочерним классом<sup>1</sup>.

Дочерний класс также может переопределить реализацию определенного метода своего родителя. На практике это делается для изменения внутренней логики конкретного метода, но сигнатура метода, открываемая дочерним классом, должна совпадать с той, которую определяет родитель. В примере 8.10 показано, как дочерний класс переопределяет реализацию родительского метода.

**Пример 8.10.** Переопределение реализации родительского метода

```
class A
{
    public function greet(string $name): string
    {
        return 'Доброе утро, ' . $name;
    }
}

class B extends A
{
    public function greet(string $name): string
    {
        return 'Приветик, ' . $name;
    }
}

$first = new A();
echo $first->greet('Элис'); ❶

$second = new B();
echo $second->greet('Боб'); ❷
```

❶ Выводит Доброе утро, Элис.

❷ Выводит Приветик, Боб.

Однако переопределенный дочерний метод не теряет все сведения о реализации родителя. Внутри класса вы ссылаетесь на переменную `$this`, чтобы обратиться к конкретному экземпляру объекта. Аналогично вы можете указывать ключевое слово `parent`, чтобы сослаться на родительскую реализацию функции. Например:

```
class A
{
    public function hello(): string
```

<sup>1</sup> Подробнее о видимости свойств и методов читайте в разделе «Видимость» выше в этой главе.

```
{  
    return 'привет';  
}  
}  
  
class B extends A  
{  
    public function hello(): string  
    {  
        return parent::hello() . ' мир';  
    }  
}  
  
$instance = new B();  
echo $instance->hello();
```

## Читайте также

Документация и обсуждение модели (<https://oreil.ly/nsAM3>) наследования объектов (<https://oreil.ly/nsAM3>) в PHP.

## 8.7. Принуждение классов к определенному поведению

### Задача

Вы хотите определить методы класса, который будет использоваться в других частях вашего приложения, но при этом оставить реализацию фактических методов на усмотрение других разработчиков.

### Решение

Определите интерфейс объекта и воспользуйтесь им в своем приложении:

```
interface ArtifactRepository  
{  
    public function create(Artifact $artifact): bool;  
    public function get(int $artifactId): ?Artifact;  
    public function getAll(): array;  
    public function update(Artifact $artifact): bool;  
    public function delete(int $artifactId): bool;  
}
```

```
class Museum
{
    public function __construct(
        protected ArtifactRepository $repository
    ) {}

    public function enumerateArtifacts(): Generator
    {
        foreach($this->repository->getAll() as $artifact) {
            yield $artifact;
        }
    }
}
```

## Обсуждение

Интерфейс похож на определение класса, за исключением того, что он задает только сигнатуры конкретных методов, а не их реализацию. Однако интерфейс определяет тип, который можно использовать в других частях вашего приложения: если класс непосредственно реализует данный интерфейс, то экземпляр этого класса можно использовать так, как если бы он был того же типа, что и сам интерфейс.



Существует несколько ситуаций, когда у вас могут быть два класса, реализующие одни и те же методы и передающие приложению одни и те же сигнатуры. Однако, если эти классы неявно реализуют один и тот же интерфейс (о чем свидетельствует ключевое слово `implements`), их нельзя использовать как взаимозаменяемые в строго типизированном приложении.

Реализация должна содержать ключевое слово `implements`, чтобы сообщить компилятору PHP, что происходит. Пример из «Решения» показывает, как определяется интерфейс и как другая часть кода может использовать этот интерфейс. В примере 8.11 показано, как интерфейс `ArtifactRepository` может быть реализован с использованием массива в памяти для хранения данных.

### Пример 8.11. Явная реализация интерфейса

```
class MemoryRepository implements ArtifactRepository
{
    private array $_collection = [];

    private function nextKey(): int
    {
        $keys = array_keys($this->_collection);
        $max = array_reduce($keys, function($c, $i) {
            return max($c, $i);
        }, 0);
    }
}
```

```
        return $max + 1;
    }

    public function create(Artifact $artifact): bool
    {
        if ($artifact->id === null) {
            $artifact->id = $this->nextKey();
        }

        if (array_key_exists($artifact->id, $this->_collection)) {
            return false;
        }

        $this->_collection[$artifact->id] = $artifact;
        return true;
    }

    public function get(int $artifactId): ?Artifact
    {
        return $this->_collection[$artifactId] ?? null;
    }

    public function getAll(): array
    {
        return array_values($this->_collection);
    }

    public function update(Artifact $artifact): bool
    {
        if (array_key_exists($artifact->id, $this->_collection)) {
            $this->_collection[$artifact->id] = $artifact;
            return true;
        }

        return false;
    }

    public function delete(int $artifactId): bool
    {
        if (array_key_exists($artifactId, $this->_collection)) {
            unset($this->_collection[$artifactId]);
            return true;
        }
        return false;
    }
}
```

Во всем приложении любой метод может объявить тип параметра, используя сам интерфейс. Класс `Museum` из примера в «Решении» принимает в качестве единственного параметра конкретную реализацию `ArtifactRepository`. Затем этот класс может работать, зная, как будет выглядеть открытый API хранилища. Коду

неважно, как реализован каждый метод, важно лишь, чтобы он точно соответствовал сигнатуре конкретного интерфейса.

Определение класса может реализовывать множество различных интерфейсов одновременно. Это позволяет использовать сложный объект в различных ситуациях разными частями кода. Обратите внимание, если два или более интерфейса определяют одно и то же имя метода, их сигнатуры должны быть идентичными, как показано в примере 8.12.

**Пример 8.12.** Реализация нескольких интерфейсов одновременно

```
interface A
{
    public function foo(): int;
}

interface B
{
    public function foo(): int;
}

interface C
{
    public function foo(): string;
}

class First implements A, B
{
    public function foo(): int ❶
    {
        return 1;
    }
}

class Second implements A, C
{
    public function foo(): int|string ❷
    {
        return 'nope';
    }
}
```

❶ Поскольку А и В определяют одну и ту же сигнатуру метода, эта реализация является допустимой.

❷ Поскольку А и С определяют разные типы возвращаемых значений, то даже с помощью объединения типов невозможно определить класс, реализующий оба интерфейса. Попытка сделать это приводит к фатальной ошибке.

Помните также, что интерфейсы чем-то похожи на классы, поэтому, как и классы, они могут быть расширены<sup>1</sup>. Это делается с помощью ключевого слова `extends`, и в результате получается интерфейс, представляющий собой композицию двух или более интерфейсов, как показано в примере 8.13.

### Пример 8.13. Составные интерфейсы

```
interface A ❶
{
    public function foo(): void;
}

interface B extends A ❷
{
    public function bar(): void;
}

class C implements B
{
    public function foo(): void
    {
        // ... фактическая реализация
    }

    public function bar(): void
    {
        // ... фактическая реализация
    }
}
```

❶ Любой класс, реализующий А, должен определить метод `foo()`.

❷ Любой класс, реализующий В, должен реализовать `bar()` и `foo()` из А.

## Читайте также

Документация по интерфейсам объектов (<https://oreil.ly/A8hkg>).

## 8.8. Создание абстрактных базовых классов

### Задача

Вы хотите, чтобы класс реализовывал определенный интерфейс, но при этом определял и другую специфическую функциональность.

---

<sup>1</sup> См. рецепт 8.6, чтобы узнать больше о наследовании и расширении классов.

## Решение

Вместо реализации интерфейса определите абстрактный базовый класс, который можно расширять, например:

```
abstract class Base
{
    abstract public function getData(): string;

    public function printData(): void
    {
        echo $this->getData();
    }
}

class Concrete extends Base
{
    public function getData(): string
    {
        return bin2hex(random_bytes(16));
    }
}

$instance = new Concrete;
$instance->printData(); ❶
```

❶ Выводит что-то вроде `6ec2aff42d5904e0cce15536d8548dc`

## Обсуждение

Абстрактный класс одновременно похож на интерфейс и определение обычного класса. В нем есть некоторые нереализованные методы наряду с конкретными реализациями. Как и в случае с интерфейсом, вы не можете инстанцировать абстрактный класс напрямую — сначала придется расширить его и реализовать все определенные в нем абстрактные методы. Однако, как и в случае с классом, вы автоматически получаете доступ к любым публичным или защищенным членам базового класса в дочерней реализации<sup>1</sup>.

Ключевое различие между интерфейсами и абстрактными классами заключается в том, что последние могут связывать с собой свойства и определения методов. Абстрактные классы — это, по сути, классы, которые являются лишь неполными реализациями. Интерфейс не имеет свойств, он просто определяет функциональный интерфейс, которому должен соответствовать любой реализующий объект.

<sup>1</sup> Подробнее о наследовании классов см. в рецепте 8.6.

Еще один нюанс заключается в том, что вы можете реализовать несколько интерфейсов одновременно, но расширять можно только один класс за раз. Это ограничение само по себе помогает определить, когда использовать абстрактный базовый класс, а когда — интерфейс, но вы также можете смешивать и сочетать оба варианта!

Абстрактный класс также может определять частные члены (те, что не наследуются никаким дочерним классом), которые в противном случае используются доступными методами, как показано ниже:

```
abstract class A
{
    private string $data = 'this is a secret'; ①

    abstract public function viewData(): void;

    public function getData(): string
    {
        return $this->data; ②
    }
}

class B extends A
{
    public function viewData(): void
    {
        echo $this->getData() . PHP_EOL; ③
    }
}

$instance = new B();
$instance->viewData(); ④
```

❶ Делает ваши данные закрытыми, поэтому они доступны только в контексте A.

❷ Поскольку `::getData()` определяется классом A, свойство `$data` все еще доступно.

❸ Хотя метод `::viewData()` определен в области видимости B, он обращается к публичному методу из области видимости A. Ни один код в B не будет иметь прямого доступа к закрытым членам A.

❹ Выводит в консоль сообщение `this is a secret`.

## Читайте также

Документация и обсуждение абстракции классов (<https://oreil.ly/FMkcT>).

## 8.9. Предотвращение изменений в классах и методах

### Задача

Вы хотите, чтобы никто не смог изменить реализацию вашего класса или расширить его дочерним классом.

### Решение

Используйте ключевое слово `final`, чтобы указать, что класс является нерасширяемым:

```
final class Immutable
{
    // Определение класса
}
```

Или используйте ключевое слово `final`, чтобы пометить конкретный метод как неизменяемый:

```
class Mutable
{
    final public function fixed(): void
    {
        // Определение метода
    }
}
```

### Обсуждение

Ключевое слово `final` — это способ явного предотвращения расширения объектов, подобно механизмам, рассмотренным в предыдущих двух рецептах. Это полезно, когда вы хотите обеспечить использование конкретной реализации метода или целого класса во всей кодовой базе.

Отметив метод как `final`, вы гарантируете, что любые расширения класса не смогут переопределить реализацию этого метода. Код из следующего примера завершится фатальной ошибкой из-за попытки класса `Child` переопределить `final` метод класса `Base`:

```
class Base
{
    public function safe()
```

```
{  
    echo 'safe() внутри класса Base' . PHP_EOL;  
}  
  
final public function unsafe()  
{  
    echo 'unsafe() внутри класса Base' . PHP_EOL;  
}  
}  
  
class Child extends Base  
{  
    public function safe()  
    {  
        echo 'safe() внутри класса Child' . PHP_EOL;  
    }  
  
    public function unsafe()  
    {  
        echo 'unsafe() внутри класса Child' . PHP_EOL;  
    }  
}
```

В предыдущем примере простое исключение определения `unsafe()` из дочернего класса позволит коду выполняться, как и ожидалось. Однако если вы хотите предотвратить расширение базового класса, добавьте ключевое слово `final` в само определение класса:

```
final class Base  
{  
    public function safe()  
    {  
        echo 'safe() внутри класса Base' . PHP_EOL;  
    }  
  
    public function unsafe()  
    {  
        echo 'unsafe() внутри класса Base' . PHP_EOL;  
    }  
}
```

Ключевое слово `final` следует использовать в коде только тогда, когда переопределение конкретного метода или реализации класса грозит привести к поломке вашего приложения. На практике это встречается довольно редко, но полезно при создании гибкого интерфейса. Например, когда ваше приложение представляет интерфейс

или конкретные реализации этого интерфейса<sup>1</sup>. Тогда ваш API будет построен таким образом, чтобы принимать любую допустимую реализацию интерфейса, но вы, возможно, захотите запретить наследование своих конкретных реализаций (опять же потому, что это может сломать ваше приложение). Пример 8.14 демонстрирует, как эти зависимости могут быть построены в реальном приложении.

**Пример 8.14.** Интерфейсы и конкретные классы

```
interface DataAbstraction ①
{
    public function save();
}

final class DBImplementation implements DataAbstraction ②
{
    public function __construct(string $databaseConnection)
    {
        // Подключение к базе данных
    }
    public function save()
    {
        // Сохранение некоторых данных
    }
}

final class FileImplementation implements DataAbstraction ③
{
    public function __construct(string $filename)
    {
        // Открытие файла для записи
    }

    public function save()
    {
        // Запись в файл
    }
}

class Application
{
    public function __construct(
        protected DataAbstraction $datalayer ④
    ) {}
}
```

---

<sup>1</sup> Подробнее об интерфейсах см. в рецепте 8.7.

- ❶ Приложение описывает интерфейс, который должен реализовать любой слой абстракции данных.
- ❷ Одна конкретная реализация явно хранит данные в базе данных.
- ❸ Другая реализация использует плоские (flat) файлы для хранения данных.
- ❹ Неважно, какую реализацию вы используете, лишь бы она реализовывала базовый интерфейс. Вы можете использовать как предоставленные (`final`) классы, так и определить собственную реализацию.

В некоторых ситуациях вы можете столкнуться с классом `final`, который все же необходимо расширить. В таких случаях единственным доступным средством является декоратор. Декоратор — это класс, который принимает другой класс в качестве свойства конструктора и «украшает» его методы дополнительной функциональностью.



В некоторых случаях декораторы не позволяют вам обойти конечную природу класса. Это происходит, если указание типов и строгая типизация требуют передачи экземпляра именно этого класса в функцию или другой объект в приложении.

Предположим, что в библиотеке вашего приложения определен класс `Note`, реализующий метод `::publish()`, который публикует определенные данные в социальных сетях. Вы хотите, чтобы этот метод также создавал статический PDF-артефакт указанных данных, и должны расширить сам класс, как показано в примере 8.15.

**Пример 8.15.** Типичное расширение класса без ключевого слова `final`

```
class Note
{
    public function publish()
    {
        // Публикация данных из заметки в Twitter ...
    }
}

class StaticNote extends Note
{
    public function publish()
    {
        parent::publish();

        // Также создание статического PDF-файла с данными заметки ...
    }
}
```

```
$note = new StaticNote(); ❶
$note->publish(); ❷
```

❶ Вместо того чтобы инстанцировать объект `Note`, допускается инстанцировать `StaticNote` напрямую.

❷ При вызове метода `::publish()` объекта используются оба определения класса.

Если класс `Note` напротив является `final`, вы не сможете расширить его напрямую. В примере 8.16 показано, как можно создать новый класс, который декорирует класс `Note` и косвенно расширяет его функциональность.

**Пример 8.16.** Настройка поведения конечного класса с помощью декоратора

```
final class Note
{
    public function publish()
    {
        // Публикация данных из заметки в Twitter ...
    }
}

final class StaticNote
{
    public function __construct(private Note $note) {}

    public function publish()
    {
        $this->note->publish();

        // Также создание статического PDF-файла с данными заметки ...
    }
}

$note = new StaticNote(new Note()); ❶
$note->publish(); ❷
```

❶ Вместо того чтобы инстанцировать `StaticNote` напрямую, вы используете этот класс для обертывания (или декорирования) обычного экземпляра `Note`.

❷ При вызове метода `::publish()` объекта используются оба определения класса.

## Читайте также

Документация по последнему ключевому слову (<https://oreil.ly/k2ZGz>).

## 8.10. Клонирование объектов

### Задача

Вы хотите создать отдельную копию объекта.

### Решение

Используйте ключевое слово `clone`, например:

```
$dolly = clone $roslin;
```

### Обсуждение

По умолчанию PHP копирует объекты по ссылке, когда они присваиваются новой переменной. Эта ссылка означает, что новая переменная фактически указывает на тот же объект в памяти. Пример 8.17 демонстрирует: хотя может показаться, что вы создали копию объекта, на самом деле вы имеете дело с двумя ссылками на одни и те же данные.

**Пример 8.17.** Оператор присваивания копирует объект по ссылке

```
$obj1 = (object) [❶
    'propertyOne' => 'some',
    'propertyTwo' => 'data',
];
$obj2 = $obj1; ❷

$obj2->propertyTwo = 'changed'; ❸

var_dump($obj1); ❹
var_dump($obj2);
```

❶ Этот конкретный синтаксис является допустимым сокращением, введенным в PHP 5.4, которое динамически соединяет новый ассоциативный массив с экземпляром встроенного класса `stdClass`.

❷ Попытка скопировать первый объект в новый экземпляр с помощью оператора присваивания.

❸ Внесение изменения во внутреннее состояние «скопированного» объекта.

❹ Осмотр исходного объекта показывает, что его внутреннее состояние изменилось. Обе переменные `$obj1` и `$obj2` указывают на одно и то же место в памяти: вы просто скопировали ссылку на объект, а не сам объект!

Вместо копирования ссылки объекта ключевое слово `clone` копирует объект в новую переменную по значению. То есть все свойства копируются в новый экземпляр того же класса, который имеет все методы исходного объекта. Пример 8.18 иллюстрирует, как два объекта теперь полностью разделены.

**Пример 8.18.** Ключевое слово `clone` копирует объект по значению

```
$obj1 = (object) [
    'propertyOne' => 'some',
    'propertyTwo' => 'data',
];
$obj2 = clone $obj1; ①

$obj2->propertyTwo = 'changed'; ②

var_dump($obj1); ③
var_dump($obj2); ④
```

① Вместо строгого присваивания используется ключевое слово `clone` для создания копии объекта по значению.

② Снова вносятся изменения во внутреннее состояние копии.

③ Проверка состояния исходного объекта не выявила никаких изменений.

④ Однако клонированный и измененный объект демонстрируют ранее внесенное изменение свойства.

Стоит отметить, что, как и в предыдущих примерах, `clone` — это неглубокое клонирование данных. Эта операция не распространяется на более сложные свойства, такие как вложенные объекты. Даже при правильном использовании `clone` можно остаться с двумя разными переменными, ссылающимися на один и тот же объект в памяти. Пример 8.19 иллюстрирует, что происходит, если копируемый объект сам содержит более сложный объект.

**Пример 8.19.** Неглубокое клонирование сложных структур данных

```
$child = (object) [
    'name' => 'child',
];
$parent = (object) [
    'name' => 'parent',
    'child' => $child
];
$clone = clone $parent;
```

```
if ($parent === $clone) { ❶
    echo 'Родитель и клон – это один и тот же объект!' . PHP_EOL;
}

if ($parent == $clone) { ❷
    echo 'Родитель и клон имеют одинаковые данные!' . PHP_EOL;
}

if ($parent->child === $clone->child) { ❸
    echo 'У родителя и клона общий потомок!' . PHP_EOL;
}
```

❶ Строгое сравнение считается `true` только в том случае, если состояния по обе стороны оператора ссылаются на один и тот же объект. В данном случае вы правильно клонировали свой объект и создали совершенно новый экземпляр, так что это сравнение `false`.

❷ Свободное сравнение типов между объектами считается `true`, если значения по обе стороны оператора одинаковы, даже между дискретными экземплярами. Это утверждение оценивается как `true`.

❸ Поскольку `clone` — неглубокая операция, свойство `::$child` у обоих ваших объектов указывает на один и тот же дочерний объект в памяти. Это утверждение оценивается как `true`!

Для поддержки более глубокого клонирования клонируемый класс должен реализовать магический метод `__clone()`, который указывает PHP, что делать при применении `clone`. Если такой метод существует, PHP будет вызывать его автоматически при закрытии экземпляра класса. В примере 8.20 показано, как это может быть реализовано при работе с динамическими классами.



Невозможно динамически определять методы для экземпляров `stdClass`. Если вы хотите поддерживать глубокое клонирование объектов в вашем приложении, то должны либо определить класс напрямую, либо использовать анонимный класс, как показано в примере 8.20.

### Пример 8.20. Глубокое клонирование объектов

```
$parent = new class {
    public string $name = 'parent';
    public stdClass $child;

    public function __clone()
{
```

```
$this->child = clone $this->child;  
}  
};  
$parent->child = (object) [  
    'name' => 'child'  
];  
  
$clone = clone $parent;  
  
if ($parent === $clone) { ❶  
    echo 'Родитель и клон – это один и тот же объект!' . PHP_EOL;  
}  
  
if ($parent == $clone) { ❷  
    echo 'Родитель и клон имеют одинаковые данные!' . PHP_EOL;  
}  
  
if ($parent->child === $clone->child) { ❸  
    echo 'У родителя и клона общий потомок!' . PHP_EOL;  
}  
  
if ($parent->child == $clone->child) { ❹  
    echo 'Родитель и клон имеют одинаковые дочерние данные!' . PHP_EOL;  
}
```

❶ Объекты являются разными ссылками, поэтому это выражение оценивается как `false`.

❷ Родительский и клонированный объекты имеют одинаковые данные; `true`.

❸ Свойства `::$child` также были клонированы внутри, поэтому свойства ссылаются на разные экземпляры объектов; `false`.

❹ Оба свойства `::$child` содержат одинаковые данные, так что это значение — `true`.

В большинстве приложений вы будете работать с пользовательскими определениями классов, а не с анонимными классами. В этом случае вы все равно можете реализовать магический метод `clone()`, чтобы указать PHP, как клонировать более сложные свойства вашего объекта, если это необходимо.

## Читайте также

Документация по ключевому слову `clone` (<https://oreil.ly/LqOE2>).

## 8.11. Определение статических свойств и методов

### Задача

Вы хотите определить метод или свойство класса, которое будет доступно всем экземплярам этого класса.

### Решение

Используйте ключевое слово `static` для определения свойств или методов, доступных вне экземпляра объекта:

```
class Foo
{
    public static int $counter = 0;

    public static function increment(): void
    {
        self::$counter += 1;
    }
}
```

### Обсуждение

Статические члены класса доступны для любой части вашего кода (при условии надлежащего уровня видимости) непосредственно из определения класса, независимо от того, существует ли экземпляр этого класса в виде объекта. Статические свойства полезны, поскольку они ведут себя, по сути, как глобальные переменные, но привязаны к конкретному определению класса. Пример 8.21 иллюстрирует отличие вызова глобальной переменной от доступа к статическому свойству класса в другой функции.

**Пример 8.21.** Статические свойства в сравнении с глобальными переменными

```
class Foo
{
    public static string $name = 'Foo';
}

$bar = 'Bar';
```

```
function demonstration()
{
    global $bar; ❶
    echo Foo::$name . $bar; ❷
}
```

❶ Чтобы получить доступ к глобальной переменной в другой области видимости, вы должны явно обратиться к глобальной области видимости. Учитывая, что в более узкой области видимости могут существовать отдельные переменные, совпадающие по имени с глобальными, на практике это, вероятно, приведет к путанице.

❷ Однако доступ к свойству класса можно получить напрямую, указав имя самого класса.

Статические методы позволяют вызывать функциональность, привязанную к классу, до непосредственного создания объекта этого класса. Один из распространенных примеров — формирование объектов значений, которые должны представлять сериализованные данные, где было бы сложно создать объект с нуля напрямую.

Пример 8.22 демонстрирует класс, где вы должны создать экземпляр, десериализуя некоторые фиксированные данные. Конструктор недоступен за пределами внутренней области видимости класса, поэтому единственным средством создания объекта является статический метод.

### Пример 8.22. Инстанцирование объекта статического метода

```
class BinaryString
{
    private function __construct(private string $bits) {} ❶

    public static function fromHex(string $hex): self
    {
        return new self(hex2bin($hex)); ❷
    }

    public static function fromBase64(string $b64): self
    {
        return new self(base64_decode($b64));
    }

    public function __toString(): string ❸
}
```

```
{  
    return bin2hex($this->bits);  
}  
}  
  
$rawData = '48656c6c6f20776f726c6421';  
$binary = BinaryString::fromHex($rawData); ④
```

❶ Доступ к закрытому конструктору возможен только из самого класса.

❷ Внутри статического метода вы все равно можете создать новый экземпляр объекта с помощью специального ключевого слова `self` для ссылки на класс. Это позволит вам получить доступ к вашему `private` конструктору.

❸ Магический метод `__toString()` вызывается всякий раз, когда PHP пытается привести объект к строке напрямую (то есть когда вы пытаетесь вывести его на консоль).

❹ Вместо того чтобы создавать объект с помощью ключевого слова `new`, воспользуйтесь специально разработанным статическим методом десериализации.

На статические методы и свойства распространяются те же ограничения видимости, что и на их нестатические аналоги. Обратите внимание, что пометка любого из них как `private` означает, что они будут ссылаться только друг на друга или на нестатические методы внутри самого класса.

Поскольку статические методы и свойства не привязаны непосредственно к экземпляру объекта, то для доступа к ним вы не можете задействовать обычные объектно-ориентированные аксессоры. Вместо этого используйте непосредственно имя класса и оператор `(::)`, например `Foo::$bar` для свойств или `Foo::bar()` для методов. Внутри самого определения класса допускается указать `self` как сокращение имени класса или `parent` как сокращение имени родительского класса (если речь идет о наследовании).



Если у вас есть доступ к объекту, который является экземпляром класса, вы можете использовать имя этого объекта, а не класса, чтобы получить доступ к его статическим членам. Например, с помощью `$foo::bar()` вы получите доступ к статическому методу `bar()` в определении класса для объекта с именем `$foo`. Хотя это и сработает, другим разработчикам будет сложнее понять, с каким определением класса вы работаете, поэтому редко когда стоит прибегать к такому синтаксису.

## Читайте также

Документация по ключевому слову `static` (<https://oreil.ly/tlxjn>).

## 8.12. Интроспекция закрытых свойств или методов внутри объекта

### Задача

Вы хотите перечислить свойства или методы объекта и использовать его закрытые члены.

### Решение

Используйте PHP Reflection API для перечисления свойств и методов. Например:

```
$reflected = new ReflectionClass('SuperSecretClass');

$methods = $reflected->getMethods(); ❶
$properties = $reflected->getProperties(); ❷
```

### Обсуждение

PHP Reflection API предоставляет разработчикам широкие возможности для анализа всех компонентов их приложений. Этот API позволяет получать информацию о методах, свойствах, константах, аргументах функций и многом другом. Кроме того, вы также можете игнорировать уровень доступа к каждому элементу и напрямую вызывать закрытые методы объектов. В примере 8.23 показано, как с помощью Reflection API вызвать приватный метод.

**Пример 8.23.** Использование Reflection для обхода конфиденциальности класса

```
class Foo
{
    private int $counter = 0; ❶

    public function increment(): void
    {
        $this->counter += 1;
    }

    public function getCount(): int
    {
        return $this->counter;
    }
}

$instance = new Foo;
```

```
$instance->increment(); ❸  
$instance->increment(); ❹  
  
echo $instance->getCount() . PHP_EOL; ❺  
  
$instance->counter = 0; ❻  
  
$reflectionClass = new ReflectionClass('Foo');  
$reflectionClass->getProperty('counter')->setValue($instance, 0); ❻  
  
echo $instance->getCount() . PHP_EOL; ❾
```

- ❶ Класс имеет одно закрытое свойство для ведения внутреннего счетчика.
- ❷ Вы хотите немного увеличить счетчик относительно его значения по умолчанию. Сейчас это значение равно 1.
- ❸ Дополнительное приращение устанавливает счетчик в 2.
- ❹ В этот момент вывод состояния счетчика подтвердит, что его значение равно 2.
- ❺ Попытка напрямую взаимодействовать со счетчиком приведет к ошибке, поскольку свойство является закрытым.
- ❻ Благодаря Reflection вы можете взаимодействовать с членами объекта независимо от уровня их конфиденциальности.
- ❾ Показывает, что счетчик действительно был сброшен в 0.

Reflection (отражение) — это действительно мощный способ обхода модификаторов видимости в API, предоставляемом классом. Однако его использование в рабочем приложении обычно указывает на проблемы с дизайном интерфейса или системы. Если вашему коду нужен доступ к закрытому члену класса, то либо этот член должен быть изначально открыт, либо вам стоит создать соответствующий метод-аксессор.

Единственное стоящее применение Reflection — это проверка и изменение внутреннего состояния объекта. В приложении такое поведение должно ограничиваться открытым API класса. Однако при тестировании может возникнуть необходимость изменить состояние объекта между тестовыми прогонами способом, который API не поддерживает во время обычной работы<sup>1</sup>. Эти редкие обстоятельства могут потребовать сброса внутренних счетчиков или вызова приватных методов очистки, находящихся внутри класса. Именно тогда Reflection доказывает свою полезность.

---

<sup>1</sup> Тестирование и отладка подробно рассматриваются в главе 13.

В обычных условиях Reflection в сочетании с такими функциями, как `var_dump()` (<https://oreil.ly/HXVwr>), помогает понять работу классов, определенных в импортированном коде. Это может пригодиться для анализа сериализованных объектов или сторонних интеграций, однако не следует отправлять такую интроспекцию в рабочую среду.

## Читайте также

Обзор API Reflection (<https://oreil.ly/C49RP>) в PHP.

# 8.13. Повторное использование произвольного кода между классами

## Задача

Вы хотите разделить определенную функциональность между несколькими классами, не прибегая к расширению класса.

## Решение

Импортируйте механизм множественного наследования Trait с помощью оператора `use`, например:

```
trait Logger
{
    public function log(string $message): void
    {
        error_log($message);
    }
}

class Account
{
    use Logger; ❶

    public function __construct(public int $accountNumber)
    {
        $this->log("Создан аккаунт {$accountNumber}.");
    }
}
```

```
class User extends Person
{
    use Logger; ②

    public function authenticate(): bool
    {
        // ...
        $this->log("Пользователь {$userId} вошел в систему.");
        // ...
    }
}
```

❶ Класс Account импортирует функцию ведения журнала из вашего свойства Logger и получает возможность применять его методы, как если бы они были присущи его собственному определению.

❷ Аналогично класс User имеет доступ к методам Logger на своем уровне, несмотря на то что он расширяет базовый класс Person с дополнительной функциональностью.

## Обсуждение

Как обсуждалось в рецепте 8.6, класс в PHP может происходить только от одного базового класса. Это называется одиночным наследованием и характерно для языков, подобных PHP. К счастью, PHP предоставляет дополнительный механизм для повторного использования кода — трейт (Trait). Трейт позволяет инкапсулировать часть функциональности в отдельное определение, подобное классу, которое можно легко импортировать без нарушения одиночного наследования.

Трейт похож на класс, но его нельзя инстанцировать напрямую. Вместо этого методы, определенные трейтом, импортируются в определение другого класса с помощью оператора use. Это дает возможность повторно применять код между классами, не имеющими общего дерева наследования.

Трейты позволяют определять общие методы (с различной видимостью) и свойства, используемые в разных определениях. Вы также вправе переопределить видимость по умолчанию трейта в классе, который его импортирует. В примере 8.24 показано, как открытый метод, определенный в трейте, может быть импортирован в другой класс в качестве защищенного или даже закрытого метода.

### Пример 8.24. Переопределение видимости методов в трейте

```
trait Foo
{
    public function bar()
    {
        echo 'Hello World!';
    }
}
```

```
class A
{
    use Foo { bar as protected; } ❶
}

class B
{
    use Foo { bar as private; } ❷
}
```

❶ Этот синтаксис импортирует все методы, определенные в Foo, но явно сделает метод `::bar()` защищенным в рамках класса A. То есть только экземпляры класса A (или его потомки) смогут вызывать данный метод.

❷ Аналогично класс B меняет видимость своего импортированного метода `::foo()` на закрытый, чтобы только экземпляры B могли обращаться к этому методу напрямую.

Трейты могут использовать другие трейты так же легко, как и классы. Аналогично нет ограничений на количество трейтов, импортируемых как другими трейтами, так и определениями классов.

Если класс, импортирующий один или несколько трейтов, определяет метод, также названный в трейте, то версия класса получит приоритет и будет использоваться по умолчанию (см. пример 8.25).

#### Пример 8.25. Приоритет метода в трейтах

```
trait Foo
{
    public function bar(): string
    {
        return 'FooBar';
    }
}

class Bar
{
    use Foo;
    public function bar(): string
    {
        return 'BarFoo';
    }
}

$instance = new Bar;
echo $instance->bar(); // BarFoo
```

В некоторых случаях возникает необходимость импортировать несколько трейтов, описывающих один и тот же метод. В таких ситуациях вы можете явно указать,

какую версию метода хотите использовать в своем конечном классе, при определении оператора use:

```
trait A
{
    public function hello(): string
    {
        return 'Hello';
    }
    public function world(): string
    {
        return 'Universe';
    }
}

trait B
{
    public function world(): string
    {
        return 'World';
    }
}

class Demonstration
{
    use A, B {
        B::world insteadof A;
    }
}

$instance = new Demonstration;
echo "{$instance->hello()} {$instance->world()}!"; // Hello World!
```

Как и определения классов, трейты могут определять свойства или даже статические члены. С их помощью можно абстрагировать определения операционной логики в блоки кода многократного использования и обмениваться этой логикой между классами в вашем приложении.

## Читайте также

Документация по трейтам (<https://oreil.ly/sykOE>).

## ГЛАВА 9

---

# Безопасность и шифрование

PHP — удивительно простой в использовании язык, потому что среда его выполнения довольно снисходительна к ошибкам. Даже если вы ошибетесь, PHP зачастую продолжит выполнение вашей программы. К сожалению, эта особенность воспринимается некоторыми разработчиками как основной недостаток, поскольку она позволяет огромному количеству некорректного кода работать так, как будто он правильный.

Хуже того, большая часть этого «плохого» кода попадает в учебные материалы, что побуждает программистов копировать его и внедрять в свои проекты, продолжая эту цепочку до бесконечности. Такой щадящий принцип работы среди выполнения и богатая история языка привели ко мнению, что PHP небезопасен. На самом деле, любой язык программирования небезопасен, если его применять без должной осмотрительности.

PHP поддерживает возможность быстрой и легкой фильтрации вредоносного ввода и очистки пользовательских данных. В контексте веб-разработки подобная функция имеет критическое значение для защиты пользовательской информации от вредоносного влияния или атак. PHP также предоставляет безопасные функции для хеширования и проверки паролей при аутентификации.



Функции хеширования и проверки паролей, используемые в PHP по умолчанию, включают в себя алгоритмы безопасного хеширования и их надежные реализации, работающие непрерывно. Это защищает ваше приложение от низшевых атак, связанных с временными данными, направленных на извлечение конфиденциальной информации. Попытка реализовать хеширование (или проверку) самостоятельно, скорее всего, подвергнет ваше приложение рискам, которые PHP уже учел.

PHP также упрощает для разработчиков криптографию — как шифрование, так и подписи. Встроенные высокоуровневые интерфейсы защищают вас от тех ошибок, которые легко допустить в других языках<sup>1</sup>.

Фактически PHP — один из самых простых языков, в котором разработчики могут использовать сильную современную криптографию, не прибегая к сторонним расширениям или реализациям!

## Унаследованное шифрование

Ранние версии PHP поставлялись с расширением mcrypt (<https://oreil.ly/I7stH>). Оно открывало низкоуровневую библиотеку mcrypt, предоставляя разработчикам доступ к различным блочным шифрам и хеш-алгоритмам. Mcrypt было удалено в PHP 7.2 в пользу более нового расширения Sodium, но первое по-прежнему можно установить вручную через библиотеку PHP Extension Community Library (PECL)<sup>2</sup>.



Хотя библиотека mcrypt все еще доступна, она не обновлялась уже более десяти лет. Ее не стоит применять в новых проектах. Для любых потребностей в шифровании используйте либо связку PHP с OpenSSL, либо собственное расширение sodium.

PHP также поддерживает библиотеку OpenSSL через соответствующее расширение (<https://oreil.ly/cYNB2>). Это полезно при построении системы, которая должна взаимодействовать с устаревшими криптографическими библиотеками. Однако расширение для PHP не раскрывает полного функционала OpenSSL; рекомендую для начала изучить функции (<https://oreil.ly/Y8xEK>) и возможности OpenSSL.

В любом случае новые интерфейсы sodium обладают широким спектром криптографических операций в PHP и в целом предпочтительнее, чем OpenSSL или mcrypt.

---

<sup>1</sup> Баг в OpenJDK Psychic Signatures, произошедший в 2022 году (<https://oreil.ly/uvYXZ>), показал, как ошибка в криптографической аутентификации может подвергнуть не только приложения, но и всю языковую реализацию потенциальным угрозам со стороны злоумышленников. Речь идет об ошибке в реализации, что еще раз подчеркивает, насколько важно полагаться на надежные, проверенные, хорошо протестированные примитивы при использовании криптографических систем.

<sup>2</sup> Подробнее о нативных расширениях см. в главе 15.

## Sodium

PHP официально добавил Sodium (<https://oreil.ly/gH1Va>) (также известное как Libsodium) в качестве основного расширения в версии 7.2, выпущенной в конце 2017 года<sup>1</sup>. Эта библиотека с открытым исходным кодом поддерживает высокоуровневые абстракции для шифрования, криптографических подписей, хеширования паролей и многое другое. Сама по себе она является ответвлением более раннего проекта — библиотеки Networking and Cryptography Library (NaCl) (<https://nacl.cr.yp.to/>) Дэниела Дж. Бернштейна.

Оба проекта предоставляют простые и удобные инструменты для работы с шифрованием. Характер их интерфейсов направлен на обеспечение безопасности криптографии и превентивного избегания проблем, возникающих при взаимодействии с другими низкоуровневыми инструментами. Наличие хорошо определенных интерфейсов помогает разработчикам сделать правильный выбор алгоритмов и настроек по умолчанию, поскольку эти решения (и потенциальные ошибки) полностью абстрагированы и реализованы в виде безопасных, простых функций, предназначенных для повседневного использования.



Единственная проблема Sodium и его интерфейсов заключается в их избыточности. Каждая функция имеет префикс `sodium_`, а каждая константа — `SODIUM_`. Высокий уровень описательности в именах функций и констант позволяет легко понять, что происходит в коде. Однако это также приводит к чрезмерно длинным именам функций, например `sodium_crypto_sign_keypair_from_secretkey_and_publickey()`.

Хотя sodium поставляется в качестве основного расширения для PHP, оно полностью совместимо со всеми языками — от .NET до Go, от Java до Python (<https://oreil.ly/L9JPp>).

В отличие от многих других криптографических библиотек, Sodium ориентирована в первую очередь на аутентифицированное шифрование. Каждый фрагмент данных автоматически сопоставляется с тегом аутентификации, который библиотека может использовать для проверки целостности исходного текста. Если тег отсутствует или недействителен, библиотека выдает ошибку, предупреждая разработчика о том, что связанный с ним открытый текст ненадежен.

Подобное использование аутентификации неуникально: режим Galois/Counter Mode (GCM) в Advanced Encryption Standard (AES) делает фактически то же

---

<sup>1</sup> Полное описание процесса, в ходе которого это расширение было добавлено в ядро PHP, можно найти в оригинальном RFC (<https://oreil.ly/6X4AF>).

самое. Однако другие библиотеки часто возлагают на разработчика задачу аутентификации и проверки тега. Существует множество учебников, книг и тем на Stack Overflow, которые указывают на правильную реализацию AES, но не учитывают проверку подлинности GCM-тега, прикрепленного к сообщению! Sodium абстрагирует аутентификацию, и проверку, обеспечивая четкую и лаконичную реализацию, как показано в примере 9.1.

**Пример 9.1.** Аутентифицированное шифрование и расшифровка в sodium

```
$nonce = random_bytes(SODIUM_CRYPTO_SECRETBOX_NONCEBYTES); ①

$key = random_bytes(SODIUM_CRYPTO_SECRETBOX_KEYBYTES); ②

$message = 'Это суперсекретное сообщение!';

$ciphertext = sodium_crypto_secretbox($message, $nonce, $key); ③

$output = bin2hex($nonce . $ciphertext); ④

// Декодирование и расшифровка обратны предыдущим шагам

getBytes = hex2bin($input); ⑤
$nonce = substr($bytes, 0, SODIUM_CRYPTO_SECRETBOX_NONCEBYTES);
$ciphertext = substr($bytes, SODIUM_CRYPTO_SECRETBOX_NONCEBYTES);

$plaintext = sodium_crypto_secretbox_open($ciphertext, $nonce, $key); ⑥

if ($plaintext === false) { ⑦
    throw new Exception('Unable to decrypt!');
}
```

❶ Алгоритмы шифрования детерминированы, то есть одинаковые входные данные всегда приводят к одинаковым результатам. Чтобы шифрование одних и тех же данных с помощью одного и того же ключа было разным, необходимо каждый раз использовать случайное одноразовое число nonce для инициализации алгоритма.

❷ При симметричном шифровании используется один общий ключ, который применяется как для шифрования, так и для расшифровки данных. Хотя ключ в нашем примере случайный, стоит хранить его где-то за пределами приложения для надежности.

❸ Процесс шифрования крайне прост: sodium выбирает алгоритм и режим шифрования за вас, и все, что от вас требуется, — это сообщение, число nonce и симметричный ключ. Все остальное делает библиотека!

❹ При экспорте зашифрованного значения (для отправки другому лицу или для хранения на диске) важно отслеживать как nonce, так и соответствующий ему шифротекст.Nonce не является секретным, поэтому хранить его открыто вместе с зашифрованным значением безопасно (и рекомендуется). Преобразование необ-

работанных байтов из двоичной системы в шестнадцатеричную — эффективный способ подготовки данных для API-запроса или хранения в поле базы данных.

❸ Поскольку результат шифрования закодирован в шестнадцатеричном формате, перед расшифровкой необходимо декодировать все обратно в необработанные байты, а затем разделить nonce и компоненты шифротекста.

❹ Чтобы извлечь значение открытого текста из зашифрованного поля, предоставьте шифротекст с соответствующим ему nonce и исходным ключом шифрования. Библиотека снова извлечет необработанные байты открытого текста и возвратит их вам.

❺ Внутренняя библиотека шифрования также добавляет (и проверяет) аутентификационный тег в каждое зашифрованное сообщение. Если при расшифровке тег не проходит проверку, то вместо оригинального открытого текста библиотека Sodium вернет лiteralное значение `false`. Это признак того, что сообщение было изменено (намеренно или случайно) и ему не следует доверять.

Sodium также предоставляет эффективное средство обработки криптографии с открытым ключом. В этой парадигме для шифрования используется один ключ (известный или открытый), а для расшифровки — совершенно другой ключ, известный только получателю сообщения. Эта двухкомпонентная система ключей идеально подходит для обмена данными между двумя отдельными сторонами по потенциально ненадежному каналу связи (например, при обмене банковской информацией между пользователем и его банком через общедоступный Интернет). Фактически HTTPS-соединения, через которые работают большинство веб-сайтов в современном Интернете, применяют криптографию с открытым ключом под капотом браузера.

В старых системах, таких как RSA, для безопасного обмена информацией необходимо отслеживать относительно большие криптографические ключи. В 2022 году минимальный рекомендуемый размер ключа для RSA составлял 3072 бита; во многих ситуациях разработчики по умолчанию используют 4096 бит для сохранения их надежности в будущем. Управление ключами такого размера может быть затруднительным. Кроме того, традиционный RSA способен зашифровать только 256 байт данных. Если вы захотите зашифровать сообщение большего размера, вам придется сделать следующее.

1. Создайте 256-битный случайный ключ.
2. Используйте этот 256-битный ключ для симметричного шифрования сообщения.
3. Примените RSA для шифрования симметричного ключа.
4. Поделитесь зашифрованным сообщением и ключом, который его защищает.

Это вполне работоспособное решение, но его шаги могут запросто создать излишние трудности для команды программистов, создающих проект, в который случайно попало шифрование. К счастью, sodium практически полностью исправляет эту проблему!

Интерфейсы открытых ключей Sodium работают на основе криптографии с эллиптической кривой (ECC), а не RSA. В RSA используются простые числа и возведение в степень для создания известных (открытых) и неизвестных (закрытых) компонентов системы двух ключей, применяемой для шифрования. ECC вместо этого прибегает к геометрии и особым формам арифметики на четко определенной эллиптической кривой. Если в RSA открытый и закрытый компоненты — это числа, служащие для экспоненцирования, то в ECC такие компоненты — это координаты  $x$  и  $y$  на геометрической кривой.

При использовании ECC 256-битный ключ имеет стойкость, эквивалентную ключу RSA с длиной 3 072 бита (<https://oreil.ly/o2kne>). Кроме того, выбор криптографических примитивов для sodium означает, что его ключи — это просто числа (а не координаты  $x$  и  $y$ , как в большинстве других реализаций ECC); 256-битный ключ ECC для sodium — это просто 32-байтовое целое число!

Кроме того, sodium полностью освобождает разработчиков от необходимости «создавать случайный симметричный ключ и отдельно шифровать его», делая асимметричное шифрование таким же простым в PHP, как пример 9.1 для симметричного шифрования. Пример 9.2 иллюстрирует, как именно работает эта форма шифрования, а также обмен ключами между участниками.

### Пример 9.2. Асимметричное шифрование и расшифровка в sodium

```
$bobKeypair = sodium_crypto_box_keypair(); ❶
$bobPublic  = sodium_crypto_box_publickey($bobKeypair); ❷
$bobSecret  = sodium_crypto_box_secretkey($bobKeypair);

$nonce = random_bytes(SODIUM_CRYPTO_SECRETBOX_NONCEBYTES); ❸

$message = 'Нападение на рассвете';

$alicePublic = '...'; ❹

$keyExchange = sodium_crypto_box_keypair_from_secretkey_and_publickey( ❺
    $bobSecret,
    $alicePublic
);

$ciphertext = sodium_crypto_box($message, $nonce, $keyExchange); ❻

$output = bin2hex($nonce . $ciphertext); ❼

// Расшифровка сообщения обращает процесс обмена ключами вспять.
$keyExchange2 = sodium_crypto_box_keypair_from_secretkey_and_publickey( ❾
    $aliceSecret,
    $bobPublic
);
```

```
$plaintext = sodium_crypto_box_open($ciphertext, $nonce, $keyExchange2); ❾
```

```
if ($plaintext === false) { ❿
    throw new Exception('Unable to decrypt!');
}
```

❶ На практике обе стороны генерируют пары открытых/закрытых ключей локально и распространяют свои открытые ключи напрямую. Функция `sodium_crypto_box_keypair()` каждый раз создает случайную пару ключей, поэтому, по идеи, вам предстоит сделать это только один раз, пока секретный ключ остается закрытым.

❷ Открытый и секретный компоненты пары ключей могут быть извлечены по отдельности. Это облегчает извлечение и передачу третьей стороне только открытого ключа, но также делает секретный ключ отдельно доступным для последующей операции обмена ключами.

❸ Как и в случае с симметричным шифрованием, для каждой операции асимметричного шифрования необходимо случайное однократно используемое число `nonce`.

❹ Открытый ключ Алисы был передан по прямому каналу связи или стал известен иным способом.

❺ Обмена ключами для согласования нового ключа здесь не происходит: просто объединяется секретный ключ Боба с открытым ключом Алисы, чтобы Боб смог зашифровать сообщение, предназначенное только для Алисы.

❻ Опять же, `sodium` сам выбирает алгоритмы и режимы шифрования. Все, что от вас потребуется, — это предоставить данные, `nonce` и ключи, а библиотека сделает все остальное.

❼ При отправке сообщения полезно объединить `nonce` и шифротекст, а затем закодировать необработанные байты так, чтобы их было удобнее отправлять по HTTP-каналу. Обычно выбирают шестнадцатеричную кодировку, но кодировка Base64 также подходит.

❽ Алисе как принимающей стороне нужно объединить свой секретный ключ с открытым ключом Боба, чтобы расшифровать сообщение, которое мог зашифровать только Боб.

❾ Извлечение открытого текста так же просто, как и шифрование на шаге 6!

❿ Как и при симметричном шифровании, данная операция аутентифицируется. Если по какой-либо причине шифрование не удалось (например, открытый ключ Боба оказался недействительным) или аутентификационный тег не прошел проверку, `sodium` возвращает `false`, указывающий на недостоверность сообщения.

## Случайности

В мире шифрования выбор надлежащего источника случайности имеет решающее значение для защиты любых данных. В старых руководствах часто упоминается функция PHP `mt_rand()` (<https://oreil.ly/HeBSd>), которая представляет собой генератор псевдослучайных чисел, основанный на алгоритме Mersenne Twister.

К сожалению, хотя вывод этой функции кажется случайным для стороннего наблюдателя, она не является криптографически безопасным источником случайности. Вместо этого воспользуйтесь функциями PHP `random_bytes()` ([https://oreil.ly/\\_eYh6](https://oreil.ly/_eYh6)) и `random_int()` (<https://oreil.ly/YWQs8>) для критически важных задач. Обе эти функции работают с криптографически безопасным источником случайности, встроенным в вашу локальную операционную систему.



Криптографически безопасный генератор псевдослучайных чисел (CSPRNG) — это генератор, выход которого неотличим от случайного шума. Такие алгоритмы, как Mersenne Twister, «достаточно случайны», чтобы обмануть человека и заставить его думать, что они безопасны. На самом деле их легко можно предсказать или даже взломать, основываясь на серии предыдущих выводов. Если злоумышленник сможет достоверно предсказать выход генератора случайных чисел, он дешифрует все, что вы пытаетесь защитить с помощью этого генератора!

В следующих рецептах рассматриваются некоторые из наиболее важных концепций безопасности и шифрования в PHP. Вы узнаете о проверке ввода, правильном хранении паролей и использовании интерфейса sodium PHP.

### 9.1. Фильтрация, проверка и очистка пользовательского ввода

#### Задача

Вам нужно проверить конкретное значение, предоставленное ненадежным пользователем, прежде чем использовать его в другом месте вашего приложения.

#### Решение

С помощью функции `filter_var()` проверьте, соответствует ли значение определенному ожиданию:

```
$email = $_GET['email'];

$filtered = filter_var($email, FILTER_VALIDATE_EMAIL);
```

## Обсуждение

Фильтрация позволяет либо проверять, соответствуют ли данные определенному формату или типу, либо очищать любые данные, не прошедшие проверку. Существенное различие между этими двумя вариантами — валидацией и санитизацией соответственно — заключается в том, что санитизация удаляет недопустимые символы из значения, в то время как валидация явно возвращает `false`, если конечный, санированный ввод — некорректного типа.

В примере выше ввод недоверенного пользователя явно проверяется на соответствие допустимому адресу электронной почты. Пример 9.3 демонстрирует поведение такой проверки на различных вариантах ввода.

### Пример 9.3. Проверка электронной почты

```
function validate(string $data): mixed
{
    return filter_var($data, FILTER_VALIDATE_EMAIL);
}

validate('blah@example.com'); ①
validate('1234'); ②
validate('1234@example.com<test>'); ③
```

① Возвращает `blah@example.com`.

② Возвращает `false`.

③ Возвращает `false`.

Альтернативой предыдущему примеру может служить санитизация пользовательского ввода, при которой недопустимые символы удаляются из ввода. В результате такой очистки получается строка, состоящая только из символов, допустимых для использования в контексте, определенном в вашем приложении (например, только буквы, цифры и знаки препинания). Однако нет никакой гарантии, что эта строка будет корректным или приемлемым вводом с точки зрения логики вашего приложения или внешних стандартов. Так, в примере 9.4 правильно санируются все возможные строки ввода, даже если два результата представляют собой недопустимые адреса электронной почты.

### Пример 9.4. Тестирование санитизации ввода электронной почты

```
function sanitize(string $data): mixed
{
    return filter_var($data, FILTER_SANITIZE_EMAIL);
}

sanitize('blah@example.com'); ①
sanitize('1234'); ②
sanitize('1234@example.com<test>'); ③
```

- ❶ Возвращает `blah@example.com`.
- ❷ Возвращается `1234`.
- ❸ Возвращает `1234@example.comtest`.

Необходимость санитизации или проверки входных данных во многом зависит от того, для чего вы собираетесь использовать полученные данные. Если вы стремитесь предотвратить попадание недопустимых символов в механизм хранения данных, то правильным подходом может быть санитизация. Если же вы хотите убедиться, что данные не выходят за рамки ожидаемого набора символов и являются допустимыми для ввода, валидация данных — более надежный инструмент.

Оба подхода одинаково хорошо поддерживаются функцией `filter_var()`, основанной на различных типах фильтров в PHP. В частности, PHP позволяет использовать фильтры валидации (перечислены в табл. 9.1) и санитизации (перечислены в табл. 9.2), а также фильтры, относящиеся к другой категории (табл. 9.3). Функция `filter_var()` также поддерживает дополнительный третий параметр для флагов, которые позволяют более детально контролировать общий результат работы фильтра.

**Таблица 9.1.** Фильтры проверки, поддерживаемые PHP

ID	Опции	Флаги	Описание
<code>FILTER_VALIDATE_BOOLEAN</code>	<code>default</code>	<code>FILTER_NULL_ON_FAILURE</code>	Возвращает <code>true</code> для значений <code>1</code> , <code>true</code> , <code>on</code> и <code>yes</code> , иначе — <code>false</code>
<code>FILTER_VALIDATE_DOMAIN</code>	<code>default</code>	<code>FILTER_FLAG_HOSTNAME,</code> <code>FILTER_NULL_ON_FAILURE</code>	Проверяет соответствие длины доменного имени различным RFC
<code>FILTER_VALIDATE_EMAIL</code>	<code>default</code>	<code>FILTER_FLAG_EMAIL_UNICODE,</code> <code>FILTER_NULL_ON_FAILURE</code>	Проверяет адрес электронной почты на соответствие стандарту RFC 822 ( <a href="https://oreil.ly/iHPPaR">https://oreil.ly/iHPPaR</a> )
<code>FILTER_VALIDATE_FLOAT</code>	<code>default,</code> <code>decimal,</code> <code>min_range,</code> <code>max_range</code>	<code>FILTER_FLAG_ALLOW_THOUSANDS,</code> <code>FILTER_NULL_ON_FAILURE</code>	Проверяет значение как <code>float</code> , опционально из указанного диапазона, и преобразует в этот тип в случае успеха

ID	Опции	Флаги	Описание
FILTER_VALIDATE_INT	default, max_range, min_range	FILTER_FLAG_ALLOW_OCTAL, FILTER_FLAG_ALLOW_HEX, FILTER_NULL_ON_FAILURE	Проверяет значение как integer, дополнительно из указанного диапазона, и преобразует в этот тип в случае успеха
FILTER_VALIDATE_IP	default	FILTER_FLAG_IPV4, FILTER_FLAG_IPV6, FILTER_FLAG_NO_PRIV_RANGE, FILTER_FLAG_NO_RES_RANGE, FILTER_NULL_ON_FAILURE	Проверяет значение на соответствие корректному IP-адресу
FILTER_VALIDATE_MAC	default	FILTER_NULL_ON_FAILURE	Проверяет значение на соответствие корректному MAC-адресу
FILTER_VALIDATE_REGEXP	default, regexp	FILTER_NULL_ON_FAILURE	Проверяет значение на соответствие регулярному выражению regexp, совместимому с Perl
FILTER_VALIDATE_URL	default	FILTER_FLAG_SCHEME_REQUIRED, FILTER_FLAG_HOST_REQUIRED, FILTER_FLAG_PATH_REQUIRED, FILTER_FLAG_QUERY_REQUIRED, FILTER_NULL_ON_FAILURE	Проверяет значение на соответствие корректному URL-адресу по правилам стандарта RFC 2396 ( <a href="https://oreil.ly/KiLd3">https://oreil.ly/KiLd3</a> )

Таблица 9.2. Фильтры очистки, поддерживаемые PHP

ID	Флаги	Описание
FILTER_SANITIZE_EMAIL	FILTER_NULL_ON_FAILURE	Удаляет все символы, кроме букв, цифр и знаков !#\$%& '*+-=?^_`{ }~@.[ ]
FILTER_SANITIZE_ENCODED	FILTER_FLAG_STRIP_LOW, FILTER_FLAG_STRIP_HIGH, FILTER_FLAG_STRIP_BACKTICK, FILTER_FLAG_ENCODE_HIGH, FILTER_FLAG_ENCODE_LOW	Кодирует строку в формат URL и, если нужно, удаляет или кодирует специальные символы
FILTER_SANITIZE_ADD_SLASHES		Применяет функцию addslashes()

Продолжение ↗

**Таблица 9.2 (продолжение)**

ID	Флаги	Описание
FILTER_SANITIZE_NUMBER_FLOAT	FILTER_FLAG_ALLOW_FRACTION, FILTER_FLAG_ALLOW_THOUSANDS, FILTER_FLAG_ALLOW_SCIENTIFIC	Удаляет все символы, кроме цифр, знаков «плюс» и «минус», а также, по желанию, точек, запятых, прописных и строчных букв <i>Es</i>
FILTER_SANITIZE_NUMBER_INT		Удаляет все символы, кроме цифр и знаков «плюс» и «минус»
FILTER_SANITIZE_SPECIAL_CHARS	FILTER_FLAG_STRIP_LOW, FILTER_FLAG_STRIP_HIGH, FILTER_FLAG_STRIP_BACKTICK, FILTER_FLAG_ENCODE_HIGH	Кодирует символы ' " < > & и символы с ASCII-кодом меньше 32 в HTML-сущности и, если нужно, удаляет или кодирует другие специальные символы
FILTER_SANITIZE_FULL_SPECIAL_CHARS	FILTER_FLAG_NO_ENCODE_QUOTES	Эквивалент вызова функции <code>htmlspecialchars()</code> с параметром ENT_QUOTES
FILTER_SANITIZE_URL		Удаляет все символы, кроме тех, которые допустимы в URL-адресах

**Таблица 9.3.** Различные фильтры, поддерживаемые PHP

ID	Опции	Флаги	Описание
FILTER_CALLBACK	Функция или метод, которые принадлежат типу <code>callable</code>	Все флаги игнорируются	Вызывает определяемую пользователем функцию для фильтрации данных

Фильтры проверки также принимают массив опций во время выполнения. Это дает вам возможность задавать определенные диапазоны (для числовых проверок) и даже резервные значения по умолчанию, если конкретный пользовательский ввод не прошел проверку.

Например, вы создаете корзину для онлайн-магазина, где пользователь может указать количество товаров, которые он хочет приобрести. Очевидно, что это должно быть значение больше нуля и меньше, чем совокупный объем имеющихся запасов.

Подход, подобный тому, что показан в примере 9.5, принудительно установит целое значение между определенными границами, в противном случае оно вернется к 1. Таким образом, пользователь не сможет случайно заказать больше товаров, чем у вас есть, отрицательное или дробное количество товаров или задать какую-то нечисловую сумму.

**Пример 9.5.** Валидация целочисленного значения с пределами и значением по умолчанию

```
function sanitizeQuantity(mixed $orderSize): int
{
    return filter_var(
        $orderSize,
        FILTER_VALIDATE_INT,
        [
            'options' => [
                'min_range' => 1,
                'max_range' => 25,
                'default'   => 1,
            ]
        ]
    );
}

echo sanitizeQuantity(12) . PHP_EOL; ①
echo sanitizeQuantity(-5) . PHP_EOL; ②
echo sanitizeQuantity(100) . PHP_EOL; ③
echo sanitizeQuantity('banana') . PHP_EOL; ④
```

① Количество проверяется и возвращается 12.

② Отрицательные целые числа не проходят проверку, поэтому по умолчанию возвращается значение 1.

③ Вводимое число выходит за рамки допустимого диапазона, поэтому по умолчанию возвращается значение 1.

④ При вводе нечисловых данных всегда возвращается значение 1.

## Читайте также

Документация по расширению фильтрации данных ([https://oreil.ly/UX\\_Hs](https://oreil.ly/UX_Hs)) в PHP.

## 9.2. Защита конфиденциальных учетных данных от попадания в код приложения

### Задача

Приложению требуется пароль или ключ API, но нужно избежать хранения этих конфиденциальных данных в коде или системе контроля версий.

### Решение

Сохраните учетные данные в переменной окружения, доступной серверу, где запускается приложение. Затем обращайтесь к этой переменной в коде. Например:

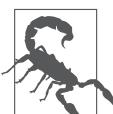
```
$db = new PDO($database_connection, getenv('DB_USER'), getenv('DB_PASS'));
```

### Обсуждение

Распространенной ошибкой, которую совершают многие начинающие разработчики, является жесткое кодирование учетных данных для чувствительных систем в константах или иных местах в коде приложения. Хотя это действие упрощает доступ к учетным данным для логики приложения, оно также создает серьезный риск для безопасности вашего приложения.

Вы можете случайно воспользоваться учетными данными, которые предназначены для доступа к реальным, живым системам или сервисам в производственной среде, из-под аккаунта разработчика. Злоумышленники способны обнаружить учетные данные, опубликованные в открытом хранилище. Сотрудники могут злоупотреблять своими знаниями об учетных данных вне их предполагаемого использования.

Самые надежные учетные данные — это те, о которых люди не знают и к которым они не имеют к ним доступа. Рекомендуется хранить эти учетные данные только в рабочей среде и создавать отдельные учетные записи для разработки и тестирования. Применение переменных окружения в коде делает приложение достаточно гибким, так как позволяет запускать его в любой среде, используя не жестко закодированные учетные данные, а переменные окружения.



Встроенная в PHP функция `phpinfo()` автоматически перечисляет все переменные окружения для отладки. Как только вы начнете эксплуатировать системное окружение для хранения конфиденциальных учетных данных, избегайте использования таких подобных диагностических инструментов, как `phpinfo()`, в общедоступных частях вашего приложения!

Метод заполнения переменных окружения отличается в различных системах. В системах с Apache переменные окружения можно задать с помощью ключевого слова `SetEnv` в директиве `<VirtualHost>` следующим образом:

```
<VirtualHost myhost>
...
SetEnv DB_USER "database"
SetEnv DB_PASS "password1234"
...
</VirtualHost>
```

В системах на базе NGINX вы настраиваете переменные окружения для PHP, только если PHP запущен как FastCGI-процесс. Подобно `SetEnv` в Apache, это делается с помощью ключевого слова в директиве `location` в конфигурации NGINX:

```
location / {
...
    fastcgi_param DB_USER database
    fastcgi_param DB_PASS password1234
...
}
```

Отдельно стоит отметить, что в системах, работающих на Docker, переменные окружения устанавливаются либо в файлах Compose (для Docker Swarm), либо в конфигурации развертывания системы (для Kubernetes). Во всех этих ситуациях вы определяете учетные данные в самой среде, а не в приложении.

Дополнительным вариантом является использование PHP dotenv (<https://oreil.ly/a4TQp>)<sup>1</sup>. Этот сторонний пакет позволяет определить конфигурацию окружения в плоском файле с расширением `.env` и автоматически заполнить как переменные окружения, так и суперглобальный массив `$_SERVER`. Самое большое преимущество подобного подхода заключается в том, что дот-файлы (файлы с префиксом `.`) обычно скрыты на сервере и их легко исключить из системы контроля версий. Вы можете задействовать `.env` локально для определения учетных данных, необходимых для разработки, и держать отдельный `.env` на сервере для доступа к системам или сервисам в эксплуатационной среде.

В обоих случаях вам вообще не придется напрямую управлять конфигурационным файлом Apache или NGINX!

Файл `.env`, определяющий учетные данные базы данных, используемые в примере из «Решения», будет выглядеть следующим образом:

```
DB_USER=database
DB_PASS=password1234
```

---

<sup>1</sup> Загрузка пакетов PHP через Composer подробно рассматривается в рецепте 15.3.

В коде вашего приложения вы загрузите зависимость библиотеки и вызовете ее загрузчик следующим образом:

```
$dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
$dotenv->load();
```

После загрузки библиотеки вы можете применить функцию `getenv()`, чтобы ссылаться на переменные окружения везде, где вам нужен доступ.

## Читайте также

Документация по `getenv()` (<https://oreil.ly/t6ncZ>).

## 9.3. Хеширование и валидация паролей

### Задача

Вы хотите аутентифицировать пользователей с помощью известных только им паролей и предотвратить хранение конфиденциальных данных в вашем приложении.

### Решение

Воспользуйтесь функцией `password_hash()` для хранения защищенных хешей паролей:

```
$hash = password_hash($password, PASSWORD_DEFAULT);
```

Для проверки того, что введенный пользователем пароль соответствует заданному, хранимому в памяти хешу, примените функцию `password_verify()`:

```
if (password_verify($password, $hash)) {
    // Создание действительной сессии для пользователя ...
}
```

### Обсуждение

Хранить пароли в открытом виде — всегда плохая идея. Если ваше приложение или хранилище данных будет скомпрометировано, то пароли станут доступны злоумышленникам. Чтобы обезопасить пользователей и защитить их от возможных злоупотреблений в случае взлома, вы должны всегда хешировать пароли при их хранении в базе данных.

Удобно, что PHP поставляется со встроенной функцией для безопасного хранения паролей — `password_hash()`. Она принимает пароль в виде простого текста и автоматически генерирует из него детерминированный, случайный на первый взгляд хеш. Вместо непосредственно пароля в базе данных хранится этот хеш. В момент аутентификации пользователя сравнивается введенный пароль с хешированным (при помощи безопасной функции сравнения, например `hash_equals()`), и если они совпадают, пользователю разрешается вход к системе.

PHP обычно поддерживает три алгоритма хеширования (см. табл. 9.4). На момент написания книги алгоритмом по умолчанию является `bcrypt` (основан на шифре Blowfish), но во время выполнения вы можете выбрать другой, передав второй параметр в функцию `password_hash()`.

**Таблица 9.4.** Алгоритмы хеширования паролей

Константа	Описание
PASSWORD_DEFAULT	Будет выбран алгоритм <code>bcrypt</code> по умолчанию. Обратите внимание, алгоритм может измениться на более сильный, когда такой добавится в PHP
PASSWORD_BCRYPT	Будет выбран алгоритм <code>CRIPT_BLOWFISH</code>
PASSWORD_ARGON2I	Будет выбран алгоритм хеширования Argon2i (доступен только в том случае, если PHP был скомпилирован с поддержкой Argon2)
PASSWORD_ARGON2ID	Будет выбран алгоритм хеширования Argon2id (доступен только в том случае, если PHP был скомпилирован с поддержкой Argon2)

Каждый алгоритм хеширования поддерживает набор опций, которые определяют сложность вычисления хеша на сервере. Например, алгоритм `bcrypt` поддерживает целочисленный коэффициент «стоимости» — чем он выше, тем сложнее вычисление. Алгоритмы семейства Argon2 поддерживают два коэффициента: один — для затрат памяти, другой — для времени вычисления хеша.



Увеличение стоимости вычисления хеша — это средство защиты приложения от атак методом перебора аутентификации. Говоря максимально упрощенно, если на вычисление хеша уходит 1 секунда, то легитимной стороне для аутентификации потребуется не менее 1 секунды. Однако злоумышленник может попытаться пройти аутентификацию не чаще одного раза в секунду. Это делает атаки методом перебора относительно затратными как по времени, так и по вычислительным ресурсам.

Когда вы только создаете свое приложение, рекомендуется протестировать серверную среду, в которой оно будет запускаться, и установить соответствующие

коэффициенты стоимости. Определение коэффициентов требует тестирования производительности функции `password_hash()` в реальной рабочей среде, как показано в примере 9.6. Этот сценарий протестирует производительность системы хеширования с постепенно увеличивающимися факторами затрат и определит коэффициент, который позволит достичь желаемого времени.

**Пример 9.6.** Тестирование коэффициентов стоимости для `password_hash()`

```
$timeTarget = 0.5; // 500 миллисекунд

$cost = 8;
do {
    $cost++;
    $start = microtime(true);
    password_hash('test', PASSWORD_BCRYPT, ['cost' => $cost]);
    $end = microtime(true);
} while(($end - $start) < $timeTarget);

echo "Соответствующий коэффициент затрат: {$cost}" . PHP_EOL;
```

Результат работы функции `password_hash()` должен работать и поддерживаться в будущих версиях PHP без необходимости каких-либо изменений или конвертации. Вместо того чтобы просто генерировать хеш, функция также внутренне создает модификатор входа хеш-функции (соль), чтобы сделать хеш уникальным. Затем функция возвращает строку, представляющую собой следующее:

- используемый алгоритм;
- факторы сложности или другие опции;
- генерированную случайную соль;
- полученный хеш.

На рис. 9.1 показан пример вывода такой строки.



Уникальная случайная соль генерируется PHP каждый раз, когда вы вызываете `password_hash()`. Это приводит к созданию разных хешей для одинаковых паролей. Таким образом, хешированные значения не могут быть применены для определения учетных записей, использующих одинаковые пароли.

\$2y\$10\$6z7GKa9kpDN7KC3ICQ1Hi.fd0/to7Y/x36WUKNP0IndHdkdR9Ae3K

Алгоритм	Соль	Хешированный пароль
Опции алгоритма (например, сложность)		

**Рис. 9.1.** Пример вывода `password_hash()`

Преимущество функции `password_hash()` заключается в том, что вам не нужно хранить закодированные данные в приложении. В будущем вы можете изменить алгоритм хеширования или модифицировать коэффициенты стоимости. Кодируя параметры, изначально предназначенные для генерации хеша, PHP может воссоздать хеш для проведения сравнения.

Когда пользователь входит в систему, он предоставляет только свой пароль. Приложение пересчитывает хеш этого пароля и сравнивает новое вычисленное значение с тем, которое хранится в базе данных. Учитывая, что информация, необходимая для вычисления хеша, хранится вместе с ним, задача становится относительно простой.

Однако вместо того, чтобы самостоятельно реализовывать сравнение, воспользуйтесь возможностями PHP: функция `password_verify()` самостоятельно обеспечивает безопасное и надежное сравнение хешей.

## Читайте также

Документация по `password_hash()` (<https://oreil.ly/gZwBC>) и `password_verify()` (<https://oreil.ly/gf3O9>).

# 9.4. Шифрование и расшифровка данных

## Задача

Вы хотите защитить конфиденциальные данные с помощью шифрования и гарантированно расшифровать их впоследствии.

## Решение

Используйте функцию `sodium_crypto_secretbox()` для шифрования данных с помощью симметричного (он же общий) ключа, как показано в примере 9.7.

### Пример 9.7. Симметричное шифрование с помощью sodium

```
$key = hex2bin('faae9fa60060e32b3bbe5861c2ff290f' .
    '2cd4008409aeb7c59cb3bad8a8e89512'); ❶

$message = 'Look to my coming on the first light of ' .
    'the fifth day, at dawn look to the east.';

$nonce = random_bytes(SODIUM_CRYPTO_SECRETBOX_NONCEBYTES); ❷

$ciphertext = sodium_crypto_secretbox($message, $nonce, $key); ❸
$output = bin2hex($nonce . $ciphertext); ❹
```

- ❶ Ключ должен быть случайным значением длины SODIUM\_CRYPTO\_SECRETBOX\_KEYBYTES (32 байта). Для создания новой случайной строки вы можете воспользоваться функцией `sodium_crypto_secretbox_keygen()`. Только не забудьте сохранить ее где-нибудь, чтобы была возможность расшифровать сообщение.
- ❷ Каждая операция шифрования должна использовать понсе. Функция PHP `random_bytes()` может надежно создать nonce соответствующей длины с помощью встроенных констант.
- ❸ Операция шифрования применяет как случайное число понсе, так и фиксированный секретный ключ для защиты сообщения и в качестве результата возвращает необработанные байты.
- ❹ Часто требуется обмениваться зашифрованной информацией по сетевому протоколу, поэтому кодирование исходных байтов в шестнадцатеричном виде может сделать их более удобными для передачи на различные устройства, платформы или среды. Для расшифровки данных вам также понадобится понсе, поэтому вы поместите его вместе с шифротекстом.

Если нужно расшифровать данные, используйте `sodium_crypto_secretbox_open()` для их извлечения и проверки, как показано в примере 9.8.

#### Пример 9.8. Симметричная расшифровка с помощью sodium

```
$key = hex2bin('faae9fa60060e32b3bbe5861c2ff290f' .
    '2cd4008409aeb7c59cb3bad8a8e89512'); ❶

$encrypted = '8b9225c935592a5e95a9204add5d09db' .
    'b7b6473a0aa59c107b65f7d5961b720e' .
    '7fc285bd94de531e05497143aee854e2' .
    '918ba941140b70c324efb27c86313806' .
    'e04f8e79da037df9e7cb24aa4bc0550c' .
    'd7b2723cbb560088f972a408ffc973a6' .
    '2be668e1ba1313e555ef4a95f0c1abd6' .
    'f3d73921fafdd372' ; ❷

$raw = hex2bin($encrypted);

$nonce = substr($raw, 0, SODIUM_CRYPTO_SECRETBOX_NONCEBYTES); ❸
$ciphertext = substr($raw, SODIUM_CRYPTO_SECRETBOX_NONCEBYTES);

$plaintext = sodium_crypto_secretbox_open($ciphertext, $nonce, $key);
if ($plaintext === false) { ❹
    echo 'Ошибка при расшифровке сообщения!' . PHP_EOL;
} else {
    echo $plaintext . PHP_EOL;
}
```

- ❶ Для расшифровки используйте тот же ключ, что и для шифрования.
- ❷ Здесь применяется шифротекст из примера 9.7, который извлекается из строки в виде шестнадцатеричного кода.
- ❸ Поскольку вы объединили шифротекст и nonce, вы должны разделить эти два компонента для использования в функции `sodium_crypto_secretbox_open()`.
- ❹ Вся операция шифрования/расшифровки проходит аутентификацию. Если в исходных данных что-то изменилось, этап аутентификации будет провален и функция вернет `false`, означающее факт манипуляции. Если она выдаст что-нибудь другое, то расшифровка прошла успешно и вы можете доверять полученному результату!

## Обсуждение

Семейство функций `secretbox`, предоставляемое библиотекой `sodium`, реализует аутентифицированное шифрование/расшифровку с использованием фиксированного симметричного ключа. Каждый раз, когда шифруете сообщение, вы должны делать это с помощью случайного числа nonce, чтобы полностью защитить конфиденциальность зашифрованного сообщения.



Само по себе nonce не является секретным или конфиденциальным значением. Однако необходимо следить за тем, чтобы оно не было повторно использовано с одним и тем же ключом симметричного шифрования. Одноразовые числа nonce предназначены для добавления случайности в алгоритм шифрования, так что одно и то же значение, зашифрованное дважды с помощью одного и того же ключа, может дать разные шифротексты. Повторное использование nonce с определенным ключом ставит под угрозу безопасность зашифрованных данных.

«Симметрия» заключается в том, что и для шифрования, и для расшифровки сообщения используется один ключ. Это наиболее удобно, когда система отвечает за обе операции, например, когда PHP-приложение должно зашифровать данные для хранения в базе данных, а затем расшифровать их при извлечении.

Для защиты данных применяется потоковый шифр XChaCha20 (<https://oreil.ly/DSUAQ>). Он задействует 32-байтный (256-битный) ключ и 24-байтный (192-битный) nonce. Однако разработчикам не нужно отслеживать эту информацию, поскольку она надежно скрыта за функциями и константами `secretbox`. Теперь достаточно создать или открыть «ящик» с подходящими для сообщения ключом и числом nonce.

Еще одно преимущество этого подхода — аутентификация. При каждой операции шифрования генерируется аутентификационный тег сообщения с помощью алгоритма Poly1305 (<https://oreil.ly/tSgmq>).

В процессе расшифровки sodium проверит, совпадает ли тег с защищенными данными. Если нет, возможно, сообщение было либо случайно, либо намеренно изменено. В любом случае шифротекст будет недостоверным (как и любой расшифрованный открытый текст), и открывающая функция вернет `false`.

## Асимметричное шифрование

Симметричное шифрование удобнее всего, когда все процессы выполняет одна и та же сторона. Во многих современных технологических средах эти стороны могут быть независимыми и взаимодействовать через менее надежные каналы связи. В этом случае требуется другая, более надежная форма шифрования, где каждая из сторон может создавать пары открытых и закрытых ключей и обмениваться ими для согласования ключа шифрования.

Обмен ключами — сложная тема. К счастью, библиотека sodium предоставляет простые интерфейсы для выполнения этой операции на PHP. В примерах далее две стороны будут выполнять следующие действия.

1. Создание пар открытых/закрытых ключей.
2. Обмениваться своими открытыми ключами напрямую.
3. Использовать эти асимметричные ключи для согласования симметричного ключа.
4. Обмениваться зашифрованными данными.

Пример 9.9 иллюстрирует, как каждая сторона генерирует собственные пары открытых и закрытых ключей. Хотя код приведен в одном блоке, каждая сторона будет создавать ключи независимо и обмениваться только своими открытыми ключами.

### Пример 9.9. Создание асимметричного ключа

```
$aliceKeypair = sodium_crypto_box_keypair();
$alicePublic  = sodium_crypto_box_publickey($aliceKeypair); ❶
$alicePrivate = sodium_crypto_box_secretkey($aliceKeypair); ❷

$bethKeypair = sodium_crypto_box_keypair(); ❸
$bethPublic  = sodium_crypto_box_publickey($bethKeypair);
$bethPrivate = sodium_crypto_box_secretkey($bethKeypair);
```

❶ Алиса создает пару ключей и извлекает из нее отдельно свои открытый и закрытый ключи. Она передает открытый ключ Бет.

❷ Алиса хранит свой закрытый ключ в секрете. Этот ключ она будет использовать для шифрования данных для Бет и расшифровки данных от нее. Аналогичным образом Алиса будет применять свой закрытый ключ для шифрования данных для любого другого человека, который поделился с ней своим открытым ключом.

❸ Бет независимо от Алисы делает то же самое и передает свой открытый ключ.

После того как Алиса и Бет поделились своими открытыми ключами, они могут общаться между собой конфиденциально. Семейство функций `cryptobox` в `sodium` задействует эти асимметричные ключи для вычисления симметричного ключа, который можно использовать для конфиденциальной связи. Данный симметричный ключ не раскрывается напрямую ни одной из сторон, но позволяет им легко общаться друг с другом.

Обратите внимание, что все сообщения, которые Алиса шифрует для Бет, может расшифровать только Бет. Даже Алиса не способна расшифровать их, поскольку у нее нет закрытого ключа Бет! В примере 9.10 показано, как Алиса шифрует простое сообщение, используя как свой закрытый ключ, так и открытый ключ Бет.

#### Пример 9.10. Асимметричное шифрование

```
$message = 'Следуй за белым кроликом';
$nonce = random_bytes(SODIUM_CRYPTO_BOX_NONCEBYTES);
$encryptionKey = sodium_crypto_box_keypair_from_secretkey_and_publickey(
    $alicePrivate,
    $bethPublic
); ❶

$ciphertext = sodium_crypto_box($message, $nonce, $encryptionKey); ❷

$toBeth = bin2hex($nonce . $ciphertext); ❸
```

❶ Ключ шифрования, применяемый здесь, фактически представляет собой пару ключей, состоящую из информации, полученной из индивидуальных пар ключей Алисы и Бет.

❷ Как и в случае с симметричным шифрованием, вы используете nonce, чтобы внести в зашифрованный результат эффект случайности.

❸ Вы объединяете случайное одноразовое число nonce и зашифрованный текст, чтобы Алиса могла отправить их Бет одновременно.

Пары ключей представляют собой точки на эллиптической кривой, в частности на Curve25519. Первоначальная операция, к которой прибегает `sodium` для асимметричного шифрования, — это обмен ключами между двумя такими точками для определения фиксированного, но секретного числа. Такая операция использует

алгоритм обмена ключами X25519 (<https://oreil.ly/OAqeF>) и создает число, основанное на закрытом ключе Алисы и открытом ключе Бет.

Это число служит ключом для системы поточного шифрования XSalsa20 (того же самого, что применяется для симметричного шифрования), чтобы зашифровать сообщение. Как и в случае симметричного шифрования, рассмотренного в семействе функций *secret box*, *cryptobox* использует метку аутентификации сообщения Poly1305 для защиты сообщения от фальсификации или повреждения.

На данном этапе Бет с помощью собственного закрытого ключа, открытого ключа Алисы и сообщения nonce может самостоятельно воспроизвести все эти шаги для расшифровки послания. Она выполняет аналогичный обмен ключами X25519, чтобы получить тот же самый общий ключ, а затем применяет его для расшифровки.

К счастью, библиотека sodium скрывает от нас процесс обмена ключами и их получения, что делает асимметричное расшифрование относительно простым. См. пример 9.11.

### Пример 9.11. Асимметричное расшифрование

```
$fromAlice = hex2bin($toBeth);
$nonce = substr($fromAlice, 0, SODIUM_CRYPTO_BOX_NONCEBYTES); ❶
$ciphertext = substr($fromAlice, SODIUM_CRYPTO_BOX_NONCEBYTES);

$decryptionKey = sodium_crypto_box_keypair_from_secretkey_and_publickey(
    $bethPrivate,
    $alicePublic
); ❷

$decrypted = sodium_crypto_box_open($ciphertext, $nonce, $decryptionKey); ❸
if ($decrypted === false) { ❹
    echo 'Ошибка при расшифровке сообщения!' . PHP_EOL;
} else {
    echo $decrypted . PHP_EOL;
}
```

**❶** Аналогично операции *secretbox*, сначала вы извлекаете nonce и шифротекст из шестнадцатеричной кодировки, предоставленной Алисой.

**❷** Бет при помощи своего закрытого ключа и открытого ключа Алисы создает пару ключей, подходящую для расшифровки сообщения.

**❸** Операции расшифровки и обмена ключами абстрагированы и предоставляют простой «открытый» интерфейс для чтения сообщения.

**❹** Как и в случае с симметричным шифрованием, асимметричные интерфейсы sodium будут проверять аутентификационный тег Poly1305 на послании, прежде чем расшифровать и вернуть открытый текст.

Оба механизма шифрования хорошо поддерживаются и аккуратно абстрагированы функциональными интерфейсами sodium. Это позволяет избежать часто встречающихся ошибок при реализации старых механизмов (например, AES или RSA). Libsodium (библиотека уровня C, на основе которой работает расширение sodium) широко поддерживается и в других языках, обеспечивая надежную совместимость между PHP, Ruby, Python, JavaScript и даже языками более низкого уровня, такими как C и Go.

## Читайте также

Документация по `sodium_crypto_secretbox()` (<https://oreil.ly/3IZZM>), `sodium_crypto_secretbox_open()` (<https://oreil.ly/qNDqx>), `sodium_crypto_box()` (<https://oreil.ly/-apZN>) и `sodium_crypto_box_open()` ([https://oreil.ly/5IT\\_D](https://oreil.ly/5IT_D)).

# 9.5. Хранение зашифрованных данных в файле

## Задача

Вам нужно зашифровать (или расшифровать) файл, который слишком велик, чтобы поместиться в памяти.

## Решение

Используйте Push Streaming Interfaces — функциональный интерфейс в библиотеке Sodium для шифрования одного фрагмента файла за раз (пример 9.12).

### Пример 9.12. Потоковое шифрование sodium

```
define('CHUNK_SIZE', 4096);

$key = hex2bin('67794ec75c56ba386f944634203d4e86' .
               '37e43c97857e3fa482bb9dfec1e44e70');
[$state, $header] = sodium_crypto_secretstream_xchacha20poly1305_init_push($key); ❶

$input = fopen('plaintext.txt', 'rb'); ❷
$output = fopen('encrypted.txt', 'wb');

fwrite($output, $header); ❸

$fileSize = fstat($input)['size']; ❹

for ($i = 0; $i < $fileSize; $i += (CHUNK_SIZE - 17)) { ❺
    $plain = fread($input, (CHUNK_SIZE - 17));
    $cipher = sodium_crypto_secretstream_xchacha20poly1305_push($state, $plain); ❻
```

```
    fwrite($output, $cipher);
}

sodium_memzero($state); ⑦

fclose($input); ⑧
fclose($output);
```

❶ Инициализация потоковой операции дает два значения — заголовок и текущее состояние потока. Сам заголовок содержит случайное число понсе и необходимо для расшифровки всего, что зашифровано с помощью потока.

❷ Открытие входного и выходного файлов в виде бинарных потоков. Работа с файлами — один из немногих случаев, когда вам нужно задействовать необработанные байты, а не шестнадцатеричное или Base64-кодирование для обертывания зашифрованного вывода.

❸ Чтобы гарантировать возможность расшифровки файла, сначала сохраните заголовок фиксированной длины для последующего извлечения.

❹ Прежде чем перебирать фрагменты байтов в файле, необходимо определить, насколько велик входной файл.

❺ Потоковое шифрование в sodium строится на основе аутентификационных тегов Poly1305 (длиной 17 байт). В результате вы считываете на 17 байт меньше, чем стандартный блок размером 4096 байт, поэтому на выходе в файл будет записано в общей сложности 4079 байт.

❻ API sodium шифрует открытый текст и автоматически обновляет переменную состояния (\$state передается по ссылке и обновляется на месте).

❼ После завершения шифрования память переменной состояния обнуляется. Система очистки памяти PHP автоматически удалит несуществующие ссылки, но вы должны убедиться, что никакие ошибки кодирования в других частях системы не приведут к утечке.

❽ Наконец, закройте все дескрипторы файлов, поскольку шифрование завершено.

Чтобы расшифровать файл, воспользуйтесь интерфейсами потоковой передачи sodium, как показано в примере 9.13.

### Пример 9.13. Расшифровка потока sodium

```
define('CHUNK_SIZE', 4096);

$key = hex2bin('67794ec75c56ba386f944634203d4e86' .
               '37e43c97857e3fa482bb9dfec1e44e70');
```

```
$input = fopen('encrypted.txt', 'rb');
$output = fopen('decrypted.txt', 'wb');

$header = fread($input, SODIUM_CRYPTO_SECRETSTREAM_XCHACHA20POLY1305_HEADERBYTES); ❶

$state = sodium_crypto_secretstream_xchacha20poly1305_init_pull($header, $key); ❷

$fileSize = fstat($input)['size'];
try {
    for (
        $i = SODIUM_CRYPTO_SECRETSTREAM_XCHACHA20POLY1305_HEADERBYTES;
        $i < $fileSize;
        $i += CHUNK_SIZE
    ) { ❸
        $cipher = fread($input, CHUNK_SIZE);

        [$plain, ] = sodium_crypto_secretstream_xchacha20poly1305_pull(
            $state,
            $cipher
        ); ❹

        if ($plain === false) { ❺
            throw new Exception('Ошибка при расшифровке файла!');
        }
        fwrite($output, $plain);
    }
} finally {
    sodium_memzero($state); ❻
    fclose($input);
    fclose($output);
}
```

❶ Прежде чем приступить к расшифровке, необходимо явно извлечь значение заголовка из файла, который вы до этого зашифровали.

❷ Вооружившись заголовком и ключом шифрования, вы можете начать процесс расшифровки данных с учетом всех соответствующих параметров и настроек, заданных при шифровании.

❸ Помните, что в начале файла находится заголовок, который пропускается, так как он не является частью зашифрованных данных. В фрагментах данных по 4096 байт 4079 байт — это зашифрованный текст, 17 байт — аутентификационный тег.

❹ Операция расшифровки потока возвращает кортеж из двух элементов — открытого текста и необязательного тега состояния (например, если ключ шифрования устарел или скомпрометирован и требуется его обновление).

- ❸ Если аутентификационный тег на зашифрованном сообщении не проходит проверку, функция вернет `false`, что указывает на сбой аутентификации. Если это произойдет, немедленно прекратите расшифровку.
- ❹ Опять же, по завершении операции необходимо обнулить память, хранящую состояние потока, а также закрыть дескрипторы файлов.

## Обсуждение

Интерфейсы потоковых шифров, предоставляемые sodium, не являются реальными потоками для PHP<sup>1</sup>. Точнее, это потоковые шифры, работающие как блочные шифры с внутренним счетчиком. XChaCha20 — это шифр, используемый семейством функций `sodium_crypto_secretstream_xchacha20poly1305_*`() для передачи данных в зашифрованный поток и их извлечения из него в этом рецепте. Реализация в PHP явно разбивает длинное сообщение (файл) на серию связанных сообщений, каждое из которых шифруется с помощью базового шифра и индивидуально помечается, но в определенном порядке.

Эти сообщения нельзя обрезать, удалить, переупорядочить или каким-либо образом изменить без распознавания этих действий операцией расшифровки. Также не существует практического ограничения на общее количество сообщений, которые могут быть зашифрованы в рамках этого потока, что означает, что нет предела размеру файла, который можно пропустить через примеры выше.

В «Решении» задействован фрагмент размером 4096 байт (4 Кбайт), но в целом допускается использовать 1024, 8096 или любое другое количество байт. Единственным ограничением здесь является объем памяти, доступный PHP: итерация по меньшим частям файла будет потреблять меньше памяти во время шифрования и расшифровки. Пример 9.12 иллюстрирует, как `sodium_crypto_secretstream_xchacha20poly1305_push()` шифрует один фрагмент данных за раз, «проталкивая» его через алгоритм шифрования и обновляя внутреннее состояние алгоритма. Парная функция `sodium_crypto_secretstream_xchacha20poly1305_pull()` делает то же самое в обратном направлении, извлекая соответствующий открытый текст обратно из потока и обновляя состояние алгоритма.

Еще один способ увидеть это в действии — низкоуровневая функция-примитив `sodium_crypto_stream_xchacha20_xor()`. Она напрямую использует алгоритм шифрования XChaCha20 для генерации потока байтов, кажущихся случайными,

---

<sup>1</sup> Потоки PHP, подробно рассмотренные в главе 11, представляют собой эффективное средство для работы с большими фрагментами данных без исчерпания доступной системной памяти.

на основе заданного ключа и случайного понсе. Затем выполняется операция XOR между этим потоком байтов и заданным сообщением для получения шифротекста<sup>1</sup>. Пример 9.14 иллюстрирует один из способов применения этой функции — шифрование телефонных номеров в базе данных.

**Пример 9.14.** Простое потоковое шифрование для защиты данных

```
function savePhoneNumber(int $userId, string $phone): void
{
    $db = getDatabase();
    $statement = $db->prepare(
        'INSERT INTO phones (user, number, nonce) VALUES (?, ?, ?)';
    );

    $key = hex2bin(getenv('ENCRYPTION_KEY'));
    $nonce = random_bytes(SODIUM_CRYPTO_STREAM_XCHACHA20_NONCEBYTES);

    $encrypted = sodium_crypto_stream_xchacha20_xor($phone, $nonce, $key);

    $statement->execute([$userId, bin2hex($encrypted), bin2hex($nonce)]);
}
```

Преимущество использования криптографического потока подобным образом в том, что длина шифротекста точно соответствует длине открытого текста. Однако это также означает отсутствие аутентификационного тега (то есть шифротекст может быть поврежден или изменен третьей стороной, что поставит под угрозу надежность любого расшифрованного шифротекста).

В результате вы вряд ли будете напрямую использовать пример 9.14. Однако он иллюстрирует, как работает более подробная функция `sodium_crypto_secretstream_xchacha20poly1305_push()`. Обе функции применяют один и тот же алгоритм, но вариант «секретного потока» генерирует собственное число понсе и отслеживает свое внутреннее состояние при многократном обращении (чтобы зашифровать несколько кусков данных). При работе с более простой версией XOR вам придется управлять этим состоянием и повторными вызовами вручную!

## Читайте также

Документация по `sodium_crypto_secretstream_xchacha20poly1305_init_push()` (<https://oreil.ly/chGJC>), `sodium_crypto_secretstream_xchacha20poly1305_init_pull()` (<https://oreil.ly/ogGvJ>) и `sodium_crypto_stream_xchacha20_xor()` (<https://oreil.ly/yQBBC>).

<sup>1</sup> Подробнее об операторах и, в частности, о XOR читайте в главе 2.

## 9.6. Криптографическая подпись сообщения для отправки другому приложению

### Задача

Вам нужно подписать сообщение или часть данных перед отправкой их в другое приложение, чтобы оно могло подтвердить вашу подпись на данных.

### Решение

Используйте функцию `sodium_crypto_sign()`, чтобы прикрепить криптографическую подпись к простому тексту, как показано в примере 9.15.

**Пример 9.15.** Криптографические подписи на сообщениях

```
$signSeed = hex2bin('eb656c282f46b45a814fcc887977675d' .  
                     'c627a5b1507ae2a68faecee147b77621'); ❶  
$signKeys = sodium_crypto_sign_seed_keypair($signSeed);  
  
$signSecret = sodium_crypto_sign_secretkey($signKeys);  
$signPublic = sodium_crypto_sign_publickey($signKeys);  
  
$message = 'Hello world!';  
$signed = sodium_crypto_sign($message, $signSecret);
```

❶ На практике зерно подписи должно быть случайным значением, хранящимся в секрете. Это также может быть безопасный хеш, полученный из известного пароля.

### Обсуждение

Криптографические подписи позволяют убедиться, что конкретное сообщение (или строка данных) поступило из заданного источника. Пока закрытый ключ, используемый для подписания сообщения, хранится в секрете, любой человек, имеющий доступ к открытому ключу, способен подтвердить, что информация поступила от владельца ключа.

Аналогично подписать данные может только хранитель этого ключа, помогая таким образом проверить, что именно он подписал сообщение. Это также говорит о том, что если подпись была сделана с использованием ключа, его владельцу нельзя

будет утверждать, что это сделал кто-то другой, без того чтобы аннулировать ключ (и все подписи, сделанные с его помощью).

В примере выше подпись вычисляется на основе секретного ключа и содержимого сообщения. Затем байты подписи добавляются к самому сообщению, и оба элемента вместе передаются любой стороне, желающей проверить подпись.

Также допускается сгенерировать отдельную подпись, что фактически означает создание необработанных байтов подписи без их конкатенации с сообщением. Это удобно, если сообщение и подпись должны быть отправлены независимо друг от друга для проверки третьей стороной, например как разные элементы в запросе API.



Хотя необработанные байты отлично подходят для информации, хранящейся на диске или в базе данных, они могут вызвать проблемы при работе с удаленными API. Имеет смысл кодировать весь пакет (подпись и сообщение) в Base64 перед отправкой его удаленной стороне. В противном случае при отправке этих двух компонентов вместе вам, возможно, захочется закодировать подпись отдельно (например, в шестнадцатеричном виде).

Также вместо `sodium_crypto_sign()`, как в примере из «Решения», допускается использовать `sodium_crypto_sign_detached()`, как в примере 9.16.

#### Пример 9.16. Создание отделенной подписи сообщения

```
$signSeed = hex2bin('eb656c282f46b45a814fcc887977675d' .
                     'c627a5b1507ae2a68faecee147b77621');
$signKeys = sodium_crypto_sign_seed_keypair($signSeed);

$signSecret = sodium_crypto_sign_secretkey($signKeys);
$signPublic = sodium_crypto_sign_publickey($signKeys);

$message = 'Hello world!';
$signature = sodium_crypto_sign_detached($message, $signSecret);
```

Подписи всегда будут иметь длину 64 байта независимо от того, присоединены ли они к подписываемому тексту.

## Читайте также

Документация по `sodium_crypto_sign()` (<https://oreil.ly/3eqTz>).

## 9.7. Проверка криптографической подписи

### Задача

Вы хотите проверить подпись на фрагменте данных, который прислала третья сторона.

### Решение

Используйте `sodium_crypto_sign_open()` для проверки подписи сообщения, как показано в примере 9.17.

#### Пример 9.17. Проверка криптографической подписи

```
$signPublic = hex2bin('d58c47ddb986dc2632aa5395e8962d3' .  
                      'e636ee236b38a8dc880e409c19374a5f');  
  
$message = sodium_crypto_sign_open($signed, $signPublic); ❶  
  
if ($message === false) { ❷  
    throw new Exception('Недопустимая подпись в сообщении!');  
}
```

❶ Данные в переменной `$signed` представляют собой объединение необработанной подписи и открытого сообщения, как возвращаемом значении функции `sodium_crypto_sign()`.

❷ Если подпись недействительна, функция возвращает `false` в качестве ошибки, иначе — открытый текст.

### Обсуждение

Проверка подписи очень проста, когда речь идет о лiteralном возврате из `sodium_crypto_sign()`. Достаточно передать данные и открытый ключ подписывающей стороны в `sodium_crypto_sign_open()`, и вы получите либо логическую ошибку, либо исходный открытый текст.

Если вы работаете с веб-API, велика вероятность, что сообщение и подпись были переданы вам отдельно (например, если кто-то использовал `sodium_crypto_sign_detached()`). В этом случае вам нужно объединить подпись и сообщение, прежде чем передавать их в `sodium_crypto_sign_open()`, как показано в примере 9.18.

**Пример 9.18.** Проверка отсоединенной подписи

```
$signPublic = hex2bin('d58c47ddb986dc2632aa5395e8962d3' .
    'e636ee236b38a8dc880e409c19374a5f');

$signature = hex2bin($_POST['signature']);
	payload = $signature . $_POST['message'];

$message = sodium_crypto_sign_open($payload, $signPublic);

if ($message === false) {
    throw new Exception('Недопустимая подпись в сообщении!');
}
```

**Читайте также**

Документация по `sodium_crypto_sign_open()` (<https://oreil.ly/UG5ja>).

## ГЛАВА 10

---

# Работа с файлами

Одна из наиболее распространенных философий разработки Unix и Linux заключается в том, что «все есть файл». Это означает, что независимо от ресурса, с которым вы взаимодействуете, операционная система обращается с ним так, как если бы это был локальный файл. Это касается и удаленных запросов к другим системам, и обработки результатов процессов, запущенных на машине.

PHP одинаково относится к запросам, процессам и ресурсам, но вместо того, чтобы считать все файлом, язык воспринимает все ресурсом потока. О потоках данных мы подробнее поговорим в главе 11, однако сперва важно обсудить то, как PHP работает с ними в памяти.

При обращении к файлу PHP не обязательно считывает все его данные в память. Вместо этого он создает в памяти ресурс `resource`, который ссылается на местоположение файла на диске и выборочно буферизует байты из этого файла в памяти. Затем PHP получает доступ к этим буферизованным байтам или манипулирует ими непосредственно как потоком. Однако для работы с рецептами из этой главы необязательно обладать знаниями основ управления потоками.

Методы PHP `file_open()`, `file_get_contents()` и тому подобные используют потоковую обертку `file://`. Однако помните, что если в PHP все является потоком, то вы можете с тем же успехом применять и другие потоковые протоколы, включая `php://` и `http://`.

## Windows или Unix

PHP свободно функционирует как в операционных системах Windows, так и в Unix (включая Linux и macOS). Важно понимать, что файловая система, лежащая в основе Windows, сильно отличается от Unix. Windows не считает «все файлом» и иногда неожиданно учитывает регистр в именах файлов и каталогов.

Как вы увидите в рецепте 10.6, разница между парадигмами операционных систем также приводит к небольшим различиям в поведении функций. В частности, бл-

кировка файлов будет работать по-другому, если ваша программа запущена под Windows из-за особенностей в вызовах операционной системы.

Следующие рецепты охватывают наиболее распространенные операции с файловой системой: от открытия и манипулирования файлами до блокировки доступа к ним других процессов.

## 10.1. Создание или открытие локального файла

### Задача

Вам нужно открыть файл для чтения или записи в локальной файловой системе.

### Решение

Используйте функцию `fopen()`, чтобы открыть файл и работать с его содержимым:

```
$fp = fopen('document.txt', 'r');
```

### Обсуждение

Внутри PHP открытый файл представлен в виде потока. Вы можете читать данные из потока или записывать их в любую позицию внутри него, основываясь на положении текущего указателя файла. В примере из «Решения» мы открыли поток только для чтения (попытка записи в этот поток будет неудачной) и поместили указатель в начало файла.

В примере 10.1 показано, как можно прочитать из файла необходимое количество байт, а затем закрыть поток, передав ссылку на него в `fclose()`.

#### Пример 10.1. Чтение байтов из буфера

```
while (($buffer = fgets($fp, 4096)) !== false) { ❶
    echo $buffer; ❷
}

fclose($fp); ❸
```

❶ Функция `fgets()` считывает одну строку из указанного ресурса, останавливаясь либо при достижении символа новой строки, либо после считывания заданного

количества байт (4096) из базового потока. Если данных для чтения нет, функция возвращает `false`.

**❷** Как только данные будут занесены в переменную, вы сможете делать с ними все, что захотите. В данном случае — вывести эту единственную строку на консоль.

**❸** После использования содержимого файла необходимо явно закрыть и очистить созданный ресурс.

Помимо чтения файла, `fopen()` позволяет произвольно записывать, добавлять, перезаписывать или усекать файл. Каждая операция определяется режимом, переданным в качестве второго параметра: в примере «Решения» передан `r` для обозначения режима «только для чтения». Дополнительные режимы описаны в табл. 10.1.

**Таблица 10.1.** Режимы работы с файлами, доступные для `fopen()`

Режим	Описание
<code>r</code>	Открывает файл только для чтения; помещает указатель в начало файла
<code>w</code>	Открывает файл только для записи; помещает указатель в начало файла и обрезает файл до нулевой длины. Если файла не существует, пытается его создать
<code>a</code>	Открывает файл только для записи; помещает указатель в конец файла. Если файла не существует, пытается его создать. В данном режиме функция <code>fseek()</code> не применима, записи всегда добавляются в конец
<code>x</code>	Создает и открывает только для записи; помещает указатель в начало файла. Если файл уже существует, вызов <code>fopen()</code> завершится неудачей, вернув <code>false</code> и выдав ошибку уровня <code>E_WARNING</code> . Если файла не существует, пытается его создать
<code>c</code>	Открывает файл только для записи. Если файла не существует, он создается. Если файл уже существует, он не обрезается (в отличие от <code>w</code> ), а вызов к этой функции не вызывает ошибку (как в случае с <code>x</code> ). Указатель файла помещается в начало файла
<code>e</code>	Устанавливает флаг <code>close-on-exec</code> (закрыть при запуске) на открытый файловый дескриптор

Ко всем режимам из табл. 10.1, кроме `e`, вы можете добавить символ `+`, чтобы открыть файл для чтения и записи, а не для одной из операций.

Функция `fopen()` работает не только с локальными файлами. По умолчанию предполагается, что вы хотите взаимодействовать с локальной файловой системой, поэтому от вас не требуется явно указывать обработчик протокола `file://`. Однако вы можете с тем же успехом обращаться к удаленным файлам, используя обработчики `http://` или `ftp://`, как показано ниже:

```
$fp = fopen('https://eamann.com/', 'r');
```



Хотя включение удаленных файлов возможно, во многих ситуациях оно может быть опасным, поскольку вы не контролируете содержимое, возвращаемое удаленной файловой системой. Обычно рекомендуется отключать удаленный доступ к файлам, активировав опцию `allow_url_include` в конфигурации системы. Для подробностей обратитесь к документации (<https://oreil.ly/-gXR->).

Дополнительный третий параметр позволяет `fopen()` искать файл в системном пути `include` (<https://oreil.ly/3S1lo>), если это необходимо. По умолчанию PHP ищет только в локальном каталоге (или использует абсолютный путь, если он указан). Загрузка файлов из системного пути `include` способствует повторному применению кода, поскольку вы можете указывать отдельные классы или конфигурационные файлы, не дублируя их в проекте.

## Читайте также

Документация по файловой системе PHP (<https://oreil.ly/oGJTp>), в частности по функции `fopen()` (<https://oreil.ly/7yQG->).

## 10.2. Чтение файла в строку

### Задача

Вы хотите загрузить весь файл в переменную для дальнейшего использования в вашем приложении.

### Решение

Примените функцию `file_get_contents()` следующим образом:

```
$config = file_get_contents('config.json');

if ($config !== false) {
    $parsed = json_decode($config);

    // ...
}
```

### Обсуждение

Функция `file_get_contents()` открывает файл для чтения, считывает все данные в переменную, а затем закрывает его, позволяя таким образом использовать эти данные как строку. Функционально это эквивалентно чтению файла в строку вручную с помощью функции `fread()`, как в примере 10.2.

**Пример 10.2.** Реализация функции `file_get_contents()` вручную с помощью функции `fread()`

```
function fileGetContents(string $filename): string|false
{
    $buffer = '';
    $fp = fopen($filename, 'r');

    try {
        while (!feof($fp)) {
            $buffer .= fread($fp, 4096);
        }
    } catch(Exception $e) {
        $buffer = false;
    } finally {
        fclose($fp);
    }

    return $buffer;
}

$config = fileGetContents('config.json');
```

Несмотря на возможности, продемонстрированные в примере 10.2, лучше сосредоточиться на написании простых программ и задействовать функции, предоставляемые языком, для выполнения сложных операций. Функция `file_get_contents()` реализована на языке С и обеспечивает высокий уровень производительности для вашего приложения. Она безопасна для работы с бинарными файлами и использует функции отображения памяти, реализуемые вашей операционной системой, для достижения максимальной эффективности.

Как и `fread()`, функция `file_get_contents()` способна считывать в память как локальные, так и удаленные файлы. Она также может искать файлы в системном пути `include`, если вы зададите значение необязательного второго параметра как `true`.

Функция, называемая `file_put_contents()`, позволяет автоматически записывать данные в файл без необходимости явно открывать файл и управлять процессом записи. Ниже показано, как объект может быть закодирован в JSON и записан в статический файл:

```
$config = new Config(** ... **/);
$serialized = json_encode($config);

file_put_contents('config.json', $serialized);
```

## Читайте также

Документация по `file_get_contents()` (<https://oreil.ly/5pRBt>) и `file_put_contents()` (<https://oreil.ly/4W0rG>).

## 10.3. Чтение определенного фрагмента файла

### Задача

Вы хотите прочитать некоторый набор байтов из определенной позиции в файле.

### Решение

Используйте `fopen()` для создания ресурса, `fseek()` для изменения положения указателя в файле и `fread()` для чтения данных из этой позиции:

```
$fp = fopen('document.txt', 'r');
fseek($fp, 32, SEEK_SET);

$data = fread($fp, 32);
```

### Обсуждение

По умолчанию `fopen()` в режиме чтения открывает файл как ресурс и помещает указатель в начало файла. При считывании байтов из файла указатель сдвигается вперед до тех пор, пока не достигнет конца файла. Вы можете использовать функцию `fseek()`, чтобы установить указатель в произвольное положение внутри ресурса, по умолчанию — это начало файла.

Третий параметр в примере из «Решения» — `SEEK_SET` — указывает PHP, куда добавить смещение. У вас есть три варианта:

- `SEEK_SET` (по умолчанию) устанавливает указатель в начало файла;
- `SEEK_CUR` добавляет смещение к текущей позиции указателя;
- `SEEK_END` добавляет смещение в конец файла. Это полезно для чтения последних байтов в файле, если в качестве второго параметра задать отрицательное смещение.

Предположим, вы хотите прочитать последние байты в длинном лог-файле из PHP. Вы сделаете это аналогично тому, как считывали произвольные байты в примере «Решения», но с отрицательным смещением, следующим образом:

```
$fp = fopen('log.txt', 'r');
fseek($fp, -4096, SEEK_END);

echo fread($fp, 4096);

fclose($fp);
```

Обратите внимание, что даже если длина лог-файла в предыдущем фрагменте меньше 4096 байт, PHP не станет читать дальше начала файла. Вместо этого интерпретатор поместит указатель в начало файла и будет считывать байты с этой позиции. Аналогично вы не сможете прочитать файл дальше конца независимо от того, сколько байт указано в вызове `fread()`.

## Читайте также

В рецепте 10.1 рассказывается о функции `fopen()`. Обратитесь также к документации по функциям `fread()` (<https://oreil.ly/Gb2m5>) и `fseek()` (<https://oreil.ly/Tl6gs>).

## 10.4. Изменение файла

### Задача

Вы хотите изменить определенную часть файла.

### Решение

Откройте файл для чтения и записи с помощью функции `fopen()`, затем с помощью функции `fseek()` переместите указатель на позицию, которую вы хотите обновить, и перезапишите определенное количество байтов, начиная с этого места. Например:

```
$fp = fopen('resume.txt', 'r+');
fseek($fp, 32);

fwrite($fp, 'New data', 8);

fclose($fp);
```

### Обсуждение

Как и в рецепте 10.3, функция `fseek()` применяется для смещения указателя в файле. Затем используется функция `fwrite()` для записи определенного набора байтов в файл в месте, где расположен указатель, перед закрытием ресурса.

Третий параметр в `fwrite()` указывает PHP, сколько байтов нужно записать. По умолчанию система запишет все данные, переданные во втором параметре, но вы можете ограничить объем записываемых данных, указав количество байтов. В примере из «Решения» длина записи установлена равной длине данных, что

избыточно. Более реалистичный пример этой функциональности выглядел бы следующим образом.

```
$contents = 'the quick brown fox jumped over the lazy dog';
fwrite($fp, $contents, 9);
```

Обратите внимание, что в рецепте к типичному режиму чтения добавляется знак «плюс»; это открывает файл для чтения и записи. Работа с файлом в других режимах приводит к совершенно иному поведению:

- `w` (режим записи), с возможностью чтения или без нее, обрежет файл, прежде чем вы сделаете с ним что-либо еще;
- `a` (режим добавления), с возможностью или без возможности чтения, заставит указатель файла переместиться в конец файла. Вызовы `fseek()` не переместят указатель, и ваши новые данные всегда будут добавляться в файл.

## Читайте также

В рецепте 10.3 вы найдете дополнительную информацию о произвольном вводе-выводе с файлами в PHP.

# 10.5. Одновременная запись в несколько файлов

## Задача

Например, вам нужно записать данные и в локальную файловую систему, и в консоль.

## Решение

Откройте несколько ссылок на ресурсы с помощью функции `fopen()` и записывайте данные в них, используя цикл:

```
$fps = [
    fopen('data.txt', 'w'),
    fopen('php://stdout', 'w')
];

foreach ($fps as $fp) {
    fwrite($fp, 'Колеса автобуса крутятся туда-сюда');
}
```

## Обсуждение

PHP, как правило, является однопоточной системой, которая должна выполнять операции последовательно<sup>1</sup>. Хотя в примере выше выводятся данные для двух файлов, он сначала записывает в один, а затем в другой. На практике этот процесс происходит достаточно быстро и может быть приемлемым для некоторых сценариев, но он не обеспечивает истинную «одновременность».

Даже с учетом подобного ограничения вы можете с легкостью записывать одни и те же данные в несколько файлов, что сильно упрощает работу. Вместо того чтобы использовать процедурный подход с конечным числом файлов, как в примере из «Решения», вы можете даже вынести такую операцию в класс, как показано в примере 10.3.

**Пример 10.3.** Простой класс для абстрагирования нескольких файловых операций

```
class MultiFile
{
    private array $handles = [];

    public function open(
        string $filename, string $mode = 'w',
        bool $use_include_path = false,
        $context = null
    ): mixed
    {
        $fp = fopen($filename, $mode, $use_include_path, $context);
        if ($fp !== false) {
            $this->handles[] = $fp;
        }
        return $fp;
    }

    public function write(string $data, ?int $length = null): int|false
    {
        $success = true;
        $bytes = 0;

        foreach($this->handles as $fp) {
            $out = fwrite($fp, $data, $length);
        }
    }
}
```

---

<sup>1</sup> В главе 17 подробно рассматриваются параллельные и асинхронные операции, чтобы объяснить способы выхода из однопоточной парадигмы.

```
        if ($out === false) {
            $success = false;
        } else {
            $bytes = $out;
        }
    }

    return $success ? $bytes : false;
}

public function close(): bool
{
    $return = true;

    foreach ($this->handles as $fp) {
        $return = $return && fclose($fp);
    }

    return $return;
}
}
```

Класс, определенный в примере 10.3, позволяет легко привязать операцию записи к нескольким файловым дескрипторам и очищать их по мере необходимости. Вместо того чтобы открывать каждый файл по очереди и вручную перебирать их, вы просто инстанцируете класс, добавляете свои файлы и приступаете к работе. Например:

```
$writer = new MultiFile();
$writer->open('data.txt');
$writer->open('php://stdout');

$writer->write("Греби, греби, греби веслами\nГлавным образом по течению.");

$writer->close();
```

PHP эффективно управляет ресурсными указателями, что позволяет вам записывать данные во множество файлов или потоков с минимальными затратами. Абстракции, как в примере 10.3, помогают сфокусироваться на бизнес-логике вашего приложения, в то время как PHP будет управлять дескрипторами файлов (и памятью) за вас.

## Читайте также

Документация по потоку `stdout` в PHP (<https://oreil.ly/i0kSI>).

## 10.6. Блокировка файла

### Задача

Вы хотите запретить другому процессу PHP манипулировать файлом во время работы вашего скрипта.

### Решение

Используйте функцию `flock()`:

```
$fp = fopen('myfile.txt', 'r');

if (flock($fp, LOCK_EX)) {
    // ... Читайте все, что вам нужно

    flock($fp, LOCK_UN);
} else {
    echo 'Не удалось заблокировать файл!';
    exit(1);
}
```

### Обсуждение

Часто требуется открыть файл для чтения или записи таким образом, чтобы другие скрипты не могли его изменять, пока с ним кто-то работает. Самый безопасный способ сделать это — явно заблокировать файл.



PHP в Windows использует принудительную блокировку, которая обеспечивается самой операционной системой. Если файл заблокирован, ни один процесс не в состоянии открыть его. В системах на базе Unix (включая Linux и macOS) PHP вместо этого прибегает к рекомендательной блокировке. В данном режиме операционная система может пропускать блокировки между различными процессами. Хотя несколько PHP-скриптов, как правило, соблюдают блокировку, другие процессы могут полностью ее игнорировать.

Явная блокировка файла запрещает другим процессам как читать этот файл, так и записывать в него в зависимости от типа блокировки. PHP поддерживает два вида блокировок: разделяемую (`LOCK_SH`) и эксклюзивную (`LOCK_EX`). Первая разрешает чтение, вторая вообще не позволяет другим процессам обращаться к файлу.

Если бы вы запустили код из «Решения» дважды на одной машине (предварительно активировав функцию `sleep()`, которая откладывает исполнение программы

на определенное время), второй процесс приостановился бы и ждал разблокировки перед выполнением (см. пример 10.4).

**Пример 10.4.** Иллюстрация длительной блокировки файла

```
$fp = fopen('myfile.txt', 'r');

echo 'Блокировка...' . PHP_EOL;
if (flock($fp, LOCK_EX)) {
    echo 'Ожидание ...' . PHP_EOL;
    for($i = 0; $i < 3; $i++) {
        sleep(10);
        echo ' Zzz ...' . PHP_EOL;
    }
}

echo 'Разблокировка ...' . PHP_EOL;
flock($fp, LOCK_UN);
} else {
    echo 'Не удалось заблокировать файл!';
    exit(1);
}
```

Рисунок 10.1 иллюстрирует работу блокировки при запуске предыдущей программы в двух отдельных терминалах. Первое выполнение получит блокировку файла и продолжит работу, как и ожидалось. Второе будет ждать, пока блокировка станет доступной, а затем продолжит работу.



**Рис. 10.1.** Два процесса не могут получить одну и ту же блокировку на один файл

## Читайте также

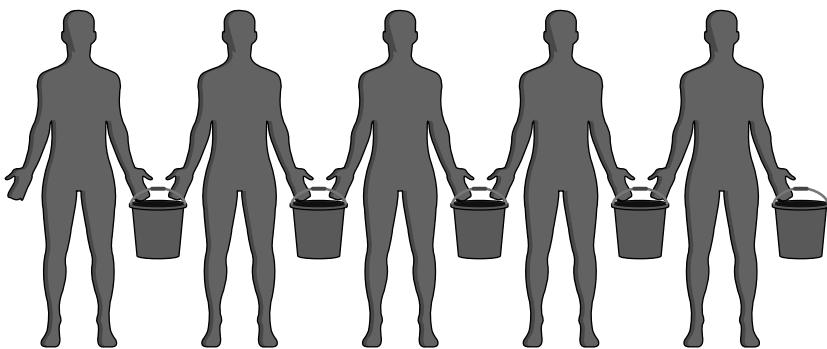
Документация по `flock()` (<https://oreil.ly/BRB05>).

## ГЛАВА 11

---

# Потоки

Потоки в PHP — это общие интерфейсы для ресурсов данных, которые можно записывать и считывать линейным, непрерывным образом. Потоки представлены коллекцией объектов, называемых «ведрами» (buckets). Каждое ведро представляет собой кэшированный объект, фрагмент данных из базового потока. На рис. 11.1 наглядно показан принцип работы потоков, который подобен старомодной бригаде с ведрами.



**Рис. 11.1.** Бригада из людей, поочередно передающих «ведра» с данными от одного к другому

Пожарные караулы, помимо прочего, нужны были для доставки воды из реки, ручья, озера или колодца к очагу пожара. Когда невозможно было использовать шланги для подачи воды, люди выстраивались в линию и передавали ведра друг другу, чтобы бороться с огнем. Один человек наполнял ведро воды у источника, а затем передавал его следующему в очереди. Люди в очереди не двигались, но ведро с водой передавалось от человека к человеку по очереди, пока последний не выливал воду на огонь. Этот процесс продолжался до тех пор, пока огонь не был потушен или не заканчивался источник воды.

Внутренняя структура потока в PHP чем-то напоминает бригаду с ведрами, поскольку данные передаются по одному фрагменту (ведру) за раз через любой компонент кода, который их обрабатывает.

Аналогом этого шаблона являются генераторы<sup>1</sup>. Вместо того чтобы загружать в память сразу весь массив данных, генераторы разбивают его на более мелкие части и взаимодействуют с одним фрагментом данных за раз. Это позволяет PHP-приложению управлять данными, которые в противном случае исчерпали бы всю системную память. Потоки предоставляют аналогичную функциональность, но работают с непрерывными данными, а не коллекциями или массивами дискретных единиц информации.

## Обертки и протоколы

В PHP потоки реализуются с помощью оберток, которые регистрируются в системе для взаимодействия с определенным протоколом. Наиболее распространенные обертки — это те, что дают доступ к файлам или HTTP URL, такие как `file://` и `http://` соответственно. Каждая обертка поддерживает различные типы данных, но все они обеспечивают одну и ту же базовую функциональность. В табл. 11.1 перечислены обертки и протоколы, которые используются в PHP.

**Таблица 11.1.** Обертки и протоколы, поддерживаемые PHP

Протокол	Описание
<code>file://</code>	Доступ к локальной файловой системе
<code>http://</code>	Доступ к удаленным URL-адресам по протоколу HTTP(S)
<code>ftp://</code>	Доступ к удаленным файловым системам по протоколу FTP(S)
<code>php://</code>	Доступ к различным локальным потокам ввода-вывода (память, <code>stdin</code> , <code>stdout</code> и т. д.)
<code>zlib://</code>	Сжатие
<code>data://</code>	Сырые данные (в соответствии с RFC 2397 ( <a href="https://oreil.ly/EBJv6">https://oreil.ly/EBJv6</a> ))
<code>glob://</code>	Поиск путей, соответствующих шаблону
<code>phar://</code>	Манипулирование PHP-архивами
<code>ssh2://</code>	Подключение через защищенную оболочку
<code>rar://</code>	Сжатие файлов RAR
<code>ogg://</code>	Аудиопотоки

Каждая обертка создает ресурс потока `stream`, который позволяет читать или записывать данные линейным образом с дополнительной возможностью «перехода»

<sup>1</sup> Подробнее о генераторах читайте в рецепте 7.15.

к произвольному месту внутри потока. Поток `file://`, например, разрешает произвольный доступ к байтам на диске.

Аналогично протокол `php://` обеспечивает доступ к чтению/записи различных потоков байтов, хранящихся в локальной памяти системы.

## Фильтры

Фильтры потоков в PHP предоставляют конструкцию, позволяющую динамически манипулировать байтами в потоке во время чтения или записи. Простым примером может служить автоматическое преобразование каждого символа в строке в верхний или нижний регистр. Это достигается путем создания пользовательского класса, расширяющего класс `php_user_filter`, и регистрации этого класса в качестве фильтра для использования компилятором, как показано в примере 11.1.

### Пример 11.1. Пользовательский фильтр

```
class StringFilter extends php_user_filter
{
    private string $mode;

    public function filter($in, $out, &$consumed, bool $closing): int
    {
        while ($bucket = stream_bucket_make_writeable($in)) { ❶
            switch($this->mode) {
                case 'lower':
                    $bucket->data = strtolower($bucket->data);
                    break;
                case 'upper':
                    $bucket->data = strtoupper($bucket->data);
                    break;
            }
            $consumed += $bucket->datalen; ❷

            stream_bucket_append($out, $bucket); ❸
        }

        return PSFS_PASS_ON; ❹
    }

    public function onCreate(): bool
    {
        switch($this->filtername) { ❺
            case 'str_tolower':
                $this->mode = 'lower';
                return true;
            case 'str_toupper':
                $this->mode = 'upper';
                return true;
        }
    }
}
```

```

        default:
            return false;
    }
}

stream_filter_register('str.*', 'StringFilter'); ❶

$fp = fopen('document.txt', 'w');
stream_filter_append($fp, 'str.toupper'); ❷

fwrite($fp, 'Hello' . PHP_EOL); ❸
fwrite($fp, 'World' . PHP_EOL);

fclose($fp);

echo file_get_contents('document.txt'); ❹

```

❶ Ресурс `$in`, передаваемый в фильтр, должен быть сначала доступен для записи, прежде чем вы сможете что-либо с ним сделать.

❷ При обращении к данным всегда обновляйте переменную вывода `$consumed`, чтобы PHP мог отслеживать, сколько байтов обработано.

❸ Ресурс `$out` изначально пуст, и вам нужно записывать в него кэшированные объекты (ведра), чтобы другие фильтры (или просто сам PHP) могли продолжать работать с потоком.

❹ Флаг `PSFS_PASS_ON` сообщает PHP, что фильтрация прошла успешно и данные доступны в ресурсе, определенном `$out`.

❺ Этот конкретный фильтр может действовать на любой флаг `str.`, но намеренно считывает только два имени фильтров для преобразования текста в верхний или нижний регистр. Включив определенное имя фильтра, вы можете перехватывать и фильтровать только нужные вам операции, позволяя другим фильтрам определять собственные `str.` функции.

❻ Определить фильтр недостаточно, вы должны явно зарегистрировать его, чтобы PHP знал, какой класс инстанцировать при фильтрации потока.

❼ После определения и регистрации фильтра вы должны добавить (или дописать) пользовательский фильтр в список фильтров, прикрепленных к текущему ресурсу потока.

❽ При подключенном фильтре все данные, записанные в поток, будут проходить через фильтр.

❾ Повторное открытие файла показывает, что ваши входные данные действительно были преобразованы в верхний регистр. Обратите внимание, что `file_get_contents()` считывает весь файл в память, а не работает с ним как с потоком.

Метод `filter()` любого пользовательского фильтра должен возвращать один из трех флагов:

- `PSFS_PASS_ON` — обработка успешно завершена, данные готовы для передачи следующему фильтру;
- `PSFS_FEED_ME` — фильтрация успешно завершена, но фильтру необходимо больше данных (либо из базового потока, либо из фильтра, расположенного непосредственно перед ним в стеке), чтобы продолжить свою работу;
- `PSFS_ERR_FATAL` — фильтр столкнулся с ошибкой.

Метод `onCreate()` открывает доступ к трем внутренним переменным из базового класса `php_user_filter`, как если бы они были свойствами дочернего класса:

- `::filtername` — имя фильтра, указанное в `stream_filter_append()` или `stream_filter_prepend()`;
- `::params` — дополнительные параметры, передаваемые в фильтр при его добавлении или дописывании в стек фильтров;
- `::stream` — фактический ресурс потока, который фильтруется.

Фильтры потоков — это мощное средство управления данными, поступающими в систему или выходящими из нее. В следующих рецептах мы рассмотрим различные варианты использования потоков в PHP, включая как обертки потоков, так и фильтры.

## 11.1. Потоковая передача данных во временный файл или из него

### Задача

Вы хотите задействовать временный файл для хранения данных, используемых в других частях программы.

### Решение

Для хранения данных примените поток `php://temp`, как если бы это был файл:

```
$fp = fopen('php://temp', 'rw');

while (true) {
    // Получение данных из некоторого источника

    fputs($fp, $data);
```

```
    if ($endOfFile) {
        break;
    }
}
```

Чтобы получить эти данные повторно, перемотайте поток в начало, а затем заново считайте данные:

```
rewind($fp);

while (true) {
    $data = fgets($fp);
    if ($data === false) {
        break;
    }

    echo $data;
}

fclose($fp);
```

## Обсуждение

Вообще, PHP поддерживает два различных временных потока данных. В примере выше применяется поток `php://temp`, но можно было бы с тем же успехом использовать `php://memory`. Для потоков данных, которые полностью помещаются в памяти, эти две обертки взаимозаменяемы. По умолчанию они будут задействовать системную память для хранения данных потока. Однако, когда поток превысит объем памяти, доступный приложению, `php://temp` направит данные во временный файл на диске.

В обоих случаях предполагается, что данные, записанные в поток, являются эфемерными. Как только вы закроете поток, они станут недоступны. Аналогично вы не можете создать новый ресурс потока, указывающий на те же данные. Пример 11.2 показывает, как PHP использует различные временные файлы для потоков даже при работе с одной и той же оберткой потока.

### Пример 11.2. Уникальность временных потоков

```
$fp = fopen('php://temp', 'rw');

fputs($fp, 'Hello world!'); ①

rewind($fp); ②
echo fgets($fp) . PHP_EOL; ③

$fp2 = fopen('php://temp', 'rw'); ④
fputs($fp2, 'Goodnight Moon.'); ⑤
```

```
rewind($fp); ⑥  
rewind($fp2);  
  
echo fgets($fp2) . PHP_EOL; ⑦  
echo fgets($fp) . PHP_EOL; ⑧
```

- ❶ Запись одной строки во временный поток.
- ❷ Перемотка обработчика потока для считывания данных из него.
- ❸ Считывание данных из потока выводит на консоль сообщение `Hello world!`
- ❹ Генерация нового обработчика потока создает совершенно новый поток, несмотря на идентичную обертку протокола.
- ❺ Запись уникальных данных в этот новый поток.
- ❻ Перемотка обоих потоков для надежности.
- ❼ Вывод второго потока, чтобы доказать его уникальность. Отображение сообщения `Goodnight Moon`.
- ❽ Вывод `Hello world!` на консоль для доказательства того, что исходный поток по-прежнему работает, как ожидалось.

В любом случае временный поток полезен, когда вам нужно сохранить какие-то данные во время работы приложения и вы не желаете явно сохранять их на диске.

## Читайте также

Документация по `fopen()` (<https://oreil.ly/LR8pa>) и оберткам потоков ввода-вывода PHP (<https://oreil.ly/6Debr>).

## 11.2. Чтение из потока ввода PHP

### Задача

Вы хотите прочитать необработанные данные из PHP.

### Решение

Воспользуйтесь потоком `php://stdin` для чтения стандартного потока ввода (`stdin`) ([https://oreil.ly/\\_BxI](https://oreil.ly/_BxI)) следующим образом:

```
$stdin = fopen('php://stdin', 'r');
```

## Обсуждение

Как и любое другое приложение, PHP имеет прямой доступ к входным данным, передаваемым ему командами и другими приложениями. В окружении, где используется текстовый ввод и вывод, это может быть другая команда, литеральный ввод в терминале или данные из другого приложения. Однако в контексте веб-приложений информация обычно передается через веб-запросы и обрабатывается веб-сервером перед PHP-приложением. В таком случае для доступа к литеральным данным стоит прибегнуть к потоку `php://input+`.



В консольных приложениях вы также можете напрямую применить константу `STDIN` (<https://oreil.ly/wgDpd>). PHP сам откроет для вас поток, то есть вам не придется создавать новую переменную ресурса.

Простое консольное приложение может принимать данные из ввода, обрабатывать их, а затем сохранять в файле. Из рецепта 9.5 вы узнали, как шифровать и расшифровывать файлы с помощью симметричных ключей в Libsodium. Если предположить, что у вас есть ключ шифрования (закодированный в шестнадцатеричном формате), представленный в виде переменной окружения, то программа из примера 11.3 будет использовать этот ключ для шифрования любых данных, передаваемых на вход, и сохранять их в выходном файле.

### Пример 11.3. Шифрование `stdin` с помощью Libsodium

```
if (empty($key = getenv('ENCRYPTION_KEY'))) { ❶
    throw new Exception('Ключ шифрования не предоставлен!');
}

$key = hex2bin($key);
if (strlen($key) !== SODIUM_CRYPTO_STREAM_XCHACHA20_KEYBYTES) { ❷
    throw new Exception('Предоставлен неверный ключ шифрования!');
}

$in = fopen('php://stdin', 'r'); ❸
$filename = sprintf('encrypted-%s.bin', uniqid()); ❹
$out = fopen($filename, 'w'); ❺

[$state, $header] = sodium_crypto_secretstream_xchacha20poly1305_init_push($key); ❻

fwrite($out, $header);

while (!feof($in)) {
    $text = fread($in, 8175);

    if (strlen($text) > 0) {
        $cipher = sodium_crypto_secretstream_xchacha20poly1305_push($state, $text);
    }
}
```

```
        fwrite($out, $cipher);
    }
}

sodium_memzero($state);

fclose($in);
fclose($out);

echo sprintf('Написано %s' . PHP_EOL, $filename);
```

- ❶ Поскольку вы хотите задействовать переменную окружения для хранения ключа шифрования, сначала проверьте, существует ли эта переменная.
- ❷ Также проверьте правильность размера ключа, прежде чем использовать его для шифрования.
- ❸ В этом примере читаем байты непосредственно из `stdin`.
- ❹ Храните зашифрованные данные в файле с динамическим именем. Обратите внимание, что на практике функция `uniqid()` использует временные метки, из-за чего может давать коллизии имен на высоконагруженных системах, а также подвергаться ошибкам состояния гонки. В реальной среде лучше прибегнуть к более надежному источнику случайности для генерируемого имени файла.
- ❺ Вывод можно было бы передать обратно в консоль, но безопаснее — в файл, поскольку при таком шифровании получаются необработанные байты. В этом случае имя файла генерируется динамически на основе системных часов.
- ❻ Остальные шаги по шифрованию повторяют рецепт 9.5.

Код из рассмотренного примера позволяет передавать данные из файла непосредственно в PHP, задействуя стандартный буфер ввода. Подобная операция выглядит примерно так: `cat plaintext-file.txt | php encrypt.php`.

Учитывая, что в результате операции шифрования получится файл, можно проделать обратную операцию с помощью аналогичного скрипта и похожим образом использовать `cat` для передачи необработанного двоичного файла назад в PHP, как показано в примере 11.4.

#### Пример 11.4. Расшифровка `stdin` с помощью Libsodium

```
if (empty($key = getenv('ENCRYPTION_KEY'))) {
    throw new Exception('Ключ шифрования не предоставлен!');
}
```

```
$key = hex2bin($key);
if (strlen($key) !== SODIUM_CRYPTO_STREAM_XCHACHA20_KEYBYTES) {
    throw new Exception('Предоставлен неверный ключ шифрования!');
}

$in = fopen('php://stdin', 'r');
$filename = sprintf('decrypted-%s.txt', uniqid());
$out = fopen($filename, 'w');

$header = fread($in, SODIUM_CRYPTO_SECRETSTREAM_XCHACHA20POLY1305_HEADERBYTES);
$state = sodium_crypto_secretstream_xchacha20poly1305_init_pull($header, $key);

try {
    while (!feof($in)) {
        $cipher = fread($in, 8192);

        [$plain, ] = sodium_crypto_secretstream_xchacha20poly1305_pull(
            $state,
            $cipher
        );

        if ($plain === false) {
            throw new Exception('Ошибка при расшифровке файла!');
        }

        fwrite($out, $plain);
    }
} finally {
    sodium_memzero($state);

    fclose($in);
    fclose($out);

    echo sprintf('Написано %s' . PHP_EOL, $filename);
}
```

Благодаря оберткам потоков ввода-вывода в PHP произвольными потоками ввода так же легко манипулировать, как и собственными файлами на локальном диске.

## Читайте также

Документация по оберткам потоков ввода-вывода PHP (<https://oreil.ly/wKjj9>).

## 11.3. Запись в поток вывода PHP

### Задача

Вы хотите вывести данные напрямую.

### Решение

Воспользуйтесь потоком `php://output`, чтобы направить данные непосредственно в стандартный поток вывода (`stdout`) (<https://oreil.ly/coZ8n>):

```
$stdout = fopen('php://stdout', 'w');
fputs($stdout, 'Hello, world!');
```

### Обсуждение

PHP предоставляет три стандартных потока ввода-вывода для пользовательского кода: `stdin`, `stdout` и `stderr`. По умолчанию все, что вы набираете в своем приложении, отправляется в стандартный поток вывода (`stdout`), что делает следующие две строки кода функционально эквивалентными:

```
fputs($stdout, 'Hello, world!');
echo 'Hello, world!';
```

Многие разработчики применяют операторы `echo` и `print` в качестве простых способов отладки приложения. Добавив индикатор в код, можно легко определить, где именно компилятор допустил ошибку, или вывести значение скрытой переменной. Однако это не единственный способ управления выводом. Поток `stdout` является общим для многих приложений, и запись в него напрямую (в отличие от неявного оператора `print`) — это возможность сконцентрировать внимание приложения на том, что оно должно делать.

Аналогично как только вы приступите к работе с `php://stdout` непосредственно для вывода данных клиенту, вы можете начать использовать поток `php://stderr` для вывода сообщений об ошибках. Эти два потока обрабатываются операционной системой по-разному, что позволяет сегментировать сообщения, которые вы получаете, на полезные уведомления и сигналы об ошибках.



В консольных приложениях допускается напрямую задействовать предопределенные константы `STDOUT` и `STDERR` (<https://oreil.ly/HArEs>). PHP нативно открывает эти потоки для вас, то есть вам не нужно создавать новые переменные ресурсов.

Код из примера 11.4 разрешает вам прочитать зашифрованные данные из `php://stdin`, расшифровать их, а затем сохранить расшифровку в файле. Более полезным примером было бы представление расшифрованных данных в `php://stdout` (а любых ошибок — в `php://stderr`), как показано в примере 11.5.

### Пример 11.5. Расшифровка stdin в stdout

```
if (empty($key = getenv('ENCRYPTION_KEY'))) {
    throw new Exception('Ключ шифрования не предоставлен!');
}

$key = hex2bin($key);
if (strlen($key) !== SODIUM_CRYPTO_STREAM_XCHACHA20_KEYBYTES) {
    throw new Exception('Предоставлен неверный ключ шифрования!');
}

$in = fopen('php://stdin', 'r');
$out = fopen('php://stdout', 'w'); ❶
$err = fopen('php://stderr', 'w'); ❷

$header = fread($in, SODIUM_CRYPTO_SECRETSTREAM_XCHACHA20POLY1305_HEADERBYTES);
$state = sodium_crypto_secretstream_xchacha20poly1305_init_pull($header, $key);

while (!feof($in)) {
    $cipher = fread($in, 8192);

    [$plain, ] = sodium_crypto_secretstream_xchacha20poly1305_pull(
        $state,
        $cipher
    );

    if ($plain === false) {
        fwrite($err, 'Ошибка при расшифровке файла!'); ❸
        exit(1);
    }
    fwrite($out, $plain);
}

sodium_memzero($state);

fclose($in);
fclose($out);
fclose($err);
```

❶ Вместо создания промежуточного файла вы можете писать напрямую в стандартный поток вывода.

- ❷ Вам также следует разобраться со стандартным потоком ошибок.
- ❸ Вместо того чтобы вызывать исключение, допускается писать непосредственно в поток ошибок.

## Читайте также

Документация по оберткам потоков ввода-вывода PHP (<https://oreil.ly/PmXdc>).

## 11.4. Чтение из одного потока и запись в другой

### Задача

Вы хотите соединить два потока, передавая байты из одного в другой.

### Решение

Воспользуйтесь функцией `stream_copy_to_stream()` для копирования данных из одного потока в другой:

```
$source = fopen('document1.txt', 'r');
$dest = fopen('destination.txt', 'w');

stream_copy_to_stream($source, $destination);
```

### Обсуждение

Механизмы потоковой передачи данных в PHP обеспечивают очень эффективные способы работы с довольно объемными данными. Зачастую в PHP-приложении могут быть задействованы файлы, размер которых превышает доступную память приложения. Большинство файлов вы можете сделать общедоступными и отправить пользователю напрямую через Apache или NGINX. В других случаях, например, от вас потребуется защитить загрузку крупных файлов (например, zip-файлов или видео) с помощью скриптов, написанных на PHP, для подтверждения личности пользователя.

Такой сценарий возможен в PHP, потому что системе не нужно хранить весь поток в памяти, но достаточно записывать байты в один поток по мере их считывания из другого потока. В примере 11.6 предполагается, что ваше PHP-приложение на-

прямую аутентифицирует пользователя и проверяет его права на определенный файл перед передачей его содержимого.

**Пример 11.6.** Копирование большого файла в stdout путем слияния потоков

```
if ($user->isAuthenticated()) {  
    $in = fopen('largeZipFile.zip', 'r'); ❶  
    $out = fopen('php://stdout', 'w');  
  
    stream_copy_to_stream($in, $out); ❷  
    exit; ❸  
}
```

❶ Открыв поток, вы просто получаете доступ к базовым данным. Ни один байт еще не был прочитан системой.

❷ Копирование одного потока в другой осуществляется напрямую без сохранения в памяти всего содержимого какого-либо из потоков. Помните, что потоки работают с фрагментами, подобно ведерной бригаде, поэтому в любой момент времени в памяти хранится только часть необходимых байтов.

❸ Важно всегда выходить из программы после копирования потока, в противном случае вы рискуете по ошибке добавить лишние байты.

Аналогично можно программно создавать большой поток и копировать его в другой при необходимости. Некоторые веб-приложения могут нуждаться в программном создании крупных объемов данных (например, очень большие одностраничные веб-приложения). Такие элементы данных стоит записывать во временную память PHP и затем копировать байты обратно, когда это необходимо. Пример 11.7 иллюстрирует, как именно это будет работать.

**Пример 11.7.** Копирование временного потока в stdout

```
$buffer = fopen('php://temp', 'w+'); ❶  
fwrite($buffer, '<html><head>');  
  
// ... Несколько сотен операций fwrite() спустя ...  
  
fwrite($buffer, '</body></html>');  
rewind($buffer); ❷  
  
$output = fopen('php://stdout', 'w');  
stream_copy_to_stream($buffer, $output); ❸  
exit; ❹
```

❶ Временный поток использует временный файл на диске. Вы ограничены не объемом памяти, доступной PHP, а свободным пространством, выделенным операционной системой под временные файлы.

- ❷ После записи всего HTML-документа во временный файл перемотайте поток обратно к началу, чтобы скопировать все эти байты в `stdout`.
- ❸ Механизм копирования одного потока в другой остается неизменным, даже если ни один из этих потоков не указывает на определенный файл на диске.
- ❹ Во избежание случайных ошибок всегда выходите из программы после того, как все байты будут скопированы на клиент.

## Читайте также

Документация по функции `stream_copy_to_stream()` ([https://oreil.ly/Us\\_Yj](https://oreil.ly/Us_Yj)).

## 11.5. Компоновка различных обработчиков потока

### Задача

Вы хотите объединить несколько концепций потоков, например обертку и фильтр в одном участке кода.

### Решение

При необходимости добавьте фильтры и используйте соответствующий протокол-обертку. В примере 11.8 применяется протокол `file://` для доступа к локальной файловой системе и два дополнительных фильтра для управления Base64-кодированием и распаковкой файлов.

#### Пример 11.8. Применение нескольких фильтров к потоку

```
$fp = fopen('compressed.txt', 'r'); ❶
stream_filter_append($fp, 'convert.base64-decode'); ❷
stream_filter_append($fp, 'zlib.inflate'); ❸

echo fread($fp, 1024) . PHP_EOL; ❹
```

- ❶ Предположим, что этот файл существует на диске и содержит следующий набор символов `80jNyscnXUS jPL8pJUQQA`.
- ❷ Первый фильтр потока, добавленный в стек, преобразует ASCII-текст в кодировке Base64 в необработанные байты.

❸ Второй фильтр использует сжатие Zlib для распаковки (или декомпрессии) необработанных байтов.

❹ Если вы начали с литерального содержимого в шаге 1, это, скорее всего, выведет `Hello, world!` на консоль.

## Обсуждение

Когда речь идет о потоках, полезно думать о слоях. Основой всегда является обработчик протокола, с помощью которого инстанцируется поток. В примере выше протокол не указан, поэтому PHP по умолчанию применит `file://`. Поверх основы располагается любое количество слоев фильтров потока.

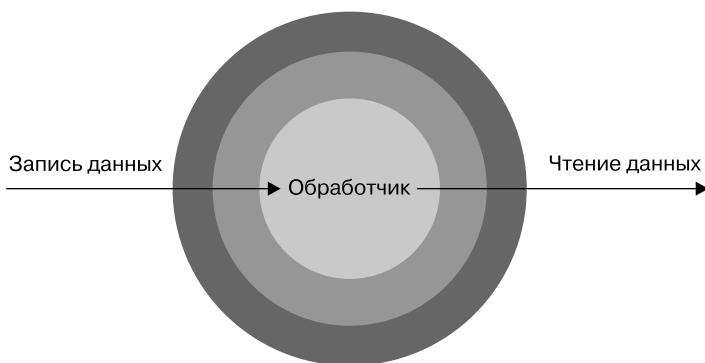
В приведенном рецепте используются сжатие Zlib и кодирование Base64 для сжатия текста и кодирования необработанных (сжатых) байтов соответственно. Чтобы создать такой сжатый/кодированный файл, выполните следующее:

```
$fp = fopen('compressed.txt', 'w');

stream_filter_append($fp, 'zlib.deflate');
stream_filter_append($fp, 'convert.base64-encode');

fwrite($fp, 'Спокойной ночи, луна!');
```

В блоке кода выше применяются те же протокольные обертки и фильтры, что и в примере из «Решения». Но обратите внимание, что порядок их добавления обратный. Это связано с тем, что фильтры потока работают подобно слоям твердой карамели. Обработчик находится в центре, а данные поступают из этого центра во внешний мир, проходя через каждый последующий уровень в определенном порядке (рис. 11.2).



**Рис. 11.2.** Данные, проходящие через потоковые фильтры PHP

В PHP уже встроено несколько фильтров, которые разрешается применить к потоку. Кроме того, у вас есть возможность определять собственный фильтр. Необработанные байты можно кодировать как в Base64, так и в шестнадцатеричном формате. Такой фильтр вы определяете сами, расширяя класс `php_user_filter` аналогично тому, как это было сделано в примере 11.1. Рассмотрим класс в примере 11.9.

**Пример 11.9.** Кодирование/декодирование шестнадцатеричного числа с помощью фильтра

```
class HexFilter extends php_user_filter
{
    private string $mode;

    public function filter($in, $out, &$consumed, bool $closing): int
    {
        while ($bucket = stream_bucket_make_writeable($in)) {
            switch ($this->mode) {
                case 'encode':
                    $bucket->data = bin2hex($bucket->data);
                    break;
                case 'decode':
                    $bucket->data = hex2bin($bucket->data);
                    break;
                default:
                    throw new Exception('Неверная модель декодирования!');
            }

            $consumed += $bucket->datalen;
            stream_bucket_append($out, $bucket);
        }

        return PSFS_PASS_ON;
    }
    public function onCreate(): bool
    {
        switch($this->filtername) {
            case 'hex.decode':
                $this->mode = 'декодирование';
                return true;
            case 'hex.encode':
                $this->mode = 'кодирование';
                return true;
            default:
                return false;
        }
    }
}
```

Класс, определенный в примере 11.9, допускается использовать для произвольного кодирования в шестнадцатеричную систему и декодирования из нее, применяя его в качестве фильтра к любому произвольному потоку. Просто зарегистрируйте его, как и любой другой фильтр, а затем примените его к какому-либо потоку, который необходимо преобразовать.

Кодировка Base64 из примера «Решение» может быть полностью заменена шестнадцатеричной, как показано в примере 11.10.

**Пример 11.10.** Комбинирование шестнадцатеричного фильтра потока со сжатием Zlib

```
stream_filter_register('hex.*', 'HexFilter'); ❶
```

```
// Запись данных
```

```
$fp = fopen('compressed.txt', 'w');
```

```
stream_filter_append($fp, 'zlib.deflate');
stream_filter_append($fp, 'hex.encode');
```

```
fwrite($fp, 'Hello, world!' . PHP_EOL);
fwrite($fp, 'Goodnight, moon!');
```

```
fclose($fp); ❷
```

```
$fp2 = fopen('compressed.txt', 'r');
stream_filter_append($fp2, 'hex.decode');
stream_filter_append($fp2, 'zlib.inflate');
```

```
echo fread($fp2, 1024); ❸
```

❶ После создания фильтра его необходимо зарегистрировать, чтобы PHP знал, как с ним обращаться. Использование символа \* при регистрации позволяет одновременно зарегистрировать и кодирование, и декодирование.

❷ Содержимое файла compressed.txt в этот момент будет иметь вид f348cdc9c9d75128cf2fcfa4951e472cfcf4fc9cb4ccf28d151c8cdcf530400.

❸ После декодирования и распаковки в консоль будет выведено сообщение Hello world! Goodnight, moon! (с разделителем новой строки между двумя утверждениями).

## Читайте также

Поддерживаемые протоколы и обертки (<https://oreil.ly/HxKpb>), а также список доступных фильтров (<https://oreil.ly/IE5UR>). Кроме того, в примере 11.1 приведен пользовательский фильтр потока.

## 11.6. Создание пользовательской обертки потока

### Задача

Вы хотите определить собственный протокол потока.

### Решение

Создайте собственный класс, повторяющий прототип `streamWrapper`, и зарегистрируйте его в PHP. Например, класс `VariableStream` может предоставлять интерфейс потока для чтения из определенной глобальной переменной или записи в нее, как показано ниже<sup>1</sup>:

```
class VariableStream
{
    private int $position;
    private string $name;
    public $context;

    function stream_open($path, $mode, $options, &$opened_path)
    {
        $url = parse_url($path);
        $this->name = $url['host'];
        $this->position = 0;

        return true;
    }

    function stream_write($data)
    {
        $left = substr($GLOBALS[$this->name], 0, $this->position);
        $right = substr($GLOBALS[$this->name], $this->position + strlen($data));
        $GLOBALS[$this->name] = $left . $data . $right;
        $this->position += strlen($data);
        return strlen($data);
    }
}
```

Вышеуказанный класс будет зарегистрирован и использован в PHP следующим образом:

```
if (!in_array('var', stream_get_wrappers())) {
    stream_wrapper_register('var', 'VariableStream');
}

$varContainer = '';
```

---

<sup>1</sup> В руководстве по PHP есть аналогичный класс (<https://oreil.ly/b0PLM>) с гораздо более широкой функциональностью, чем та, что продемонстрирована в этом примере.

```
$fp = fopen('var://varContainer', 'w');

fwrite($fp, 'Привет' . PHP_EOL);
fwrite($fp, 'Мир' . PHP_EOL);
fclose($fp);

echo $varContainer;
```

## Обсуждение

Конструкция `streamWrapper` в PHP является прототипом класса. К сожалению, это нерасширяемый класс и неконкретно реализуемый интерфейс. Вместо этого `streamWrapper` представляет собой документированный формат, которому должны следовать любые пользовательские протоколы потоков.

Хотя классы можно зарегистрировать в качестве обработчиков протоколов с помощью другого интерфейса, настоятельно рекомендуется, чтобы все потенциальные классы протоколов реализовывали каждый метод, определенный интерфейсом `streamWrapper` (скопированным из документации PHP как определение псевдоинтерфейса в примере 11.11), чтобы удовлетворить ожидаемое PHP поведение потока.

### Пример 11.11. Определение интерфейса `streamWrapper`

```
class streamWrapper {
    public $context;

    public __construct()
    public dir_closedir(): bool
    public dir_opendir(string $path, int $options): bool
    public dir_readdir(): string
    public dir_rewinddir(): bool
    public mkdir(string $path, int $mode, int $options): bool
    public rename(string $path_from, string $path_to): bool
    public rmdir(string $path, int $options): bool
    public stream_cast(int $cast_as): resource public stream_close(): void
    public stream_eof(): bool
    public stream_flush(): bool
    public stream_lock(int $operation): bool
    public stream_metadata(string $path, int $option, mixed $value): bool
```

```
public stream_open(
    string $path,
    string $mode,
    int $options,
    ?string &$opened_path
): bool

public stream_read(int $count): string|false

public stream_seek(int $offset, int $whence = SEEK_SET): bool

public stream_set_option(int $option, int $arg1, int $arg2): bool

public stream_stat(): array|false

public stream_tell(): int

public stream_truncate(int $new_size): bool

public stream_write(string $data): int

public unlink(string $path): bool

public url_stat(string $path, int $flags): array|false

public __destruct()
}
```

Некоторые специфические функции, например `mkdir`, `rename`, `rmdir` или `unlink`, вообще не должны реализовываться, если в протоколе нет для них особого применения. В противном случае система не выдаст вам (или разработчикам, взаимодействующим с вашей библиотекой) полезных сообщений об ошибках и будет вести себя неожиданно.

Хотя большинство протоколов, с которыми вы работаете ежедневно, встроены в PHP, можно написать новые обработчики протоколов или задействовать те, что были созданы другими людьми.

Часто встречаются ссылки на облачные хранилища, где используется сторонний протокол (например, `s3://` от Amazon Web Services), а не более распространенные `https://` или `file://`. Фактически AWS публикует общедоступный SDK (<https://oreil.ly/RVXlw>), который использует `stream_wrapper_register()` для предоставления протокола `s3://` другому коду приложений. Это позволяет вам работать с данными, размещенными в облаке, так же легко, как и с локальными файлами.

## Читайте также

Документация по `streamWrapper` (<https://oreil.ly/SyhD8>).

## ГЛАВА 12

---

# Обработка ошибок

Лучшие планы мышей и людей часто идут вкрай и вкось.

Роберт Бернс

Если вы работаете в сфере ИТ, то, вероятно, сталкивались с ошибками и процессом отладки. Возможно, вы даже тратите столько же времени, если не больше, на поиск ошибок, сколько и на написание кода. Такова природа программного обеспечения: независимо от того, насколько усердно команда работает над созданием качественного продукта, неизбежно возникнет сбой, который нужно выявить и исправить.

К счастью, поиск ошибок в PHP относительно прост. Неприхотливость этого языка часто превращает ошибку скорее в неприятность, чем в фатальную проблему.

Следующие рецепты предлагают самый быстрый и простой способ выявления и устранения ошибок в вашем коде. В них также подробно описано, как создавать и обрабатывать пользовательские исключения, генерируемые вашим кодом при получении недопустимых данных от стороннего API или в случае другого некорректного поведения системы.

### 12.1. Поиск и исправление ошибок синтаксиса

#### Задача

Компилятор PHP не смог проанализировать скрипт в вашем приложении; вы хотите быстро найти и устраниТЬ ошибку.

#### Решение

Откройте проблемный файл в текстовом редакторе и просмотрите строку, на которую указывает парсер. Если сразу не удается понять, в чем загвоздка, пройдитесь по коду вверх, проверяя каждую строку по очереди, пока не найдете ошибку и не исправите ее. Затем сохраните файл.

## Обсуждение

PHP — это язык, который даже неправильный или проблемный скрипт будет пытаться довести до конца. Однако парсер далеко не всегда способен правильно интерпретировать строку кода, чтобы определить, что должно быть сделано, и вместо этого возвращает ошибку.

В качестве наглядного примера перечислим западные штаты США:

```
$states = ['Вашингтон', 'Орегон', 'Калифорния'];
foreach $states as $state {
    print "{$state} находится на западном побережье." . PHP_EOL;
}
```

Интерпретатор PHP при запуске этого кода выдаст ошибку `Parse error` на второй строке:

```
PHP Parse error: syntax error, unexpected variable "$states", expecting "(" in php
shell code on line 2
```

По одному только этому сообщению можно локализовать проблему. Помните, что, хотя `foreach` — это языковая конструкция, она все равно записывается как вызов функции с круглыми скобками. Правильный способ итерации по массиву состояний будет выглядеть следующим образом:

```
$states = ['Вашингтон', 'Орегон', 'Калифорния'];
foreach ($states as $state) {
    print "{$state} находится на западном побережье." . PHP_EOL;
}
```

Эта конкретная ошибка — пропуск скобок при использовании языковых конструкций — распространена среди разработчиков, часто переходящих с одного языка на другой. Например, тот же механизм в Python выглядит почти так же, но синтаксически корректен, если опустить круглые скобки в вызове `foreach`. Например:

```
states = ['Вашингтон', 'Орегон', 'Калифорния'].
for state in states:
    print(f"{state} находится на западном побережье.")
```

Синтаксис этих двух языков сбивает с толку своим сходством. К счастью, парсер каждого из них достаточно чувствителен, чтобы заметить ошибку и предупредить вас.

Удобно, что такие IDE, как Visual Studio Code (<https://oreil.ly/CkzbA>), автоматически анализируют ваш скрипт и подсвечивают все синтаксические ошибки еще до запуска приложения (см. рис. 12.1).

The screenshot shows a code editor window for a file named 'states.php'. The code contains several syntax errors:

```
1 <?php
2 $states = ['Washington', 'Oregon', 'California'];
3 foreach $states as $state {
4     echo "$state is on the West coast.";
5 }
```

Visual Studio Code highlights the first character of the opening tag with a red squiggle, and the entire 'foreach' keyword and its opening brace with red squiggles. The code is otherwise displayed in a standard monospaced font.

Рис. 12.1. Visual Studio Code идентифицирует и выделяет синтаксические ошибки до запуска приложения

## Читайте также

Список меток парсера PHP ([https://oreil.ly/Zw\\_II](https://oreil.ly/Zw_II)).

## 12.2. Создание и обработка пользовательских исключений

### Задача

Вы хотите, чтобы ваше приложение в случае проблем генерировало (и перехватывало) пользовательское исключение.

### Решение

Расширьте базовый класс `Exception`, чтобы внедрить в него пользовательское поведение, а затем используйте блоки `try/catch` для захвата и обработки исключений.

### Обсуждение

PHP определяет базовый интерфейс `Throwable` (<https://oreil.ly/NkLuC>), реализуемый любым видом ошибки или исключения в языке. Внутренние проблемы представляются классом `Error` (<https://oreil.ly/eFMGz>) и его потомками, а проблемы в пользовательской среде — классом `Exception` и его потомками.

Как правило, вы будете расширять только класс `Exception` в рамках своего приложения, но у вас есть возможность перехватывать любую реализацию `Throwable` в стандартном блоке `try/catch`.

Предположим, вы реализуете функцию деления с очень тонкой, пользовательской функциональностью:

- деление на 0 не допускается;
- все десятичные значения округляются в меньшую сторону;
- целое число 42 не допускается в качестве числителя;
- числитель должен представлять собой целое число, но знаменатель может быть числом с плавающей точкой.

Подобная функция допускает использование встроенных ошибок, например `ArithmicError` или `DivisionByZeroError`. Однако в списке правил третье выделяется как требующее пользовательского исключения. Перед определением функции необходимо задать пользовательское исключение, как показано в примере 12.1.

**Пример 12.1.** Простое определение пользовательского исключения

```
class HitchhikerException extends Exception
{
    public function __construct(int $code = 0, Throwable $previous = null)
    {
        parent::__construct('42 является неделимым.', $code, $previous);
    }

    public function __toString()
    {
        return __CLASS__ . "::__construct(): {$this->message}\n";
    }
}
```

Когда пользовательское исключение создано, генерируйте его в своей пользовательской функции деления следующим образом:

```
function divide(int $numerator, float|int $denominator): int
{
    if ($denominator === 0) {
        throw new DivisionByZeroError;
    } elseif ($numerator === 42) {
        throw new HitchhikerException;
    }

    return floor($numerator / $denominator);
}
```

После того как вы определили свою пользовательскую функциональность, необходимо использовать этот код в приложении. Вы знаете, что функция может

выдать ошибку, поэтому важно обернуть любой ее вызов в оператор `try` и обработать ошибку соответствующим образом. В примере 12.2 выполняются итерация по четырем парам чисел, попытка деления на каждой из них и обработка всех последующих ошибок/исключений.

### Пример 12.2. Обработка ошибок в пользовательском делении

```
$pairs = [
    [10, 2],
    [2, 5],
    [10, 0],
    [42, 2]
];

foreach ($pairs as $pair) {
    try {
        echo divide($pair[0], $pair[1]) . PHP_EOL;
    } catch (HitchhikerException $he) { ❶
        echo 'Неверное деление на 42!' . PHP_EOL;
    } catch (Throwable $t) { ❷
        echo 'Смотрите, бешеный сурок!' . PHP_EOL;
    }
}
```

❶ Если в качестве числителя передается число 42, функция `divide()` сгенерирует исключение `HitchhikerException` и не сможет восстановиться. Перехват этого исключения, позволит вам дать обратную связь либо приложению, либо пользователю и двигаться дальше.

❷ Любая другая ошибка или исключение, сгенерированные функцией, будут перехвачены как реализация `Throwable`. В этом случае вы отклоняете ошибку и продолжаете выполнение.

## Читайте также

Документация по следующим вопросам:

- базовый класс `Exception` (<https://oreil.ly/2s4mn>);
- список предопределенных исключений (<https://oreil.ly/tdegn>);
- дополнительные исключения, определенные стандартной библиотекой PHP (SPL) (<https://oreil.ly/gsdeg>);
- создание пользовательских исключений с помощью расширений (<https://oreil.ly/-jrvt>);
- иерархия ошибок в PHP 7 (<https://oreil.ly/KF1Zd>).

## 12.3. Скрытие сообщений об ошибках от конечных пользователей

### Задача

Вы исправили все известные вам ошибки и готовы запустить приложение в эксплуатацию. Но вы также хотите предотвратить отображение новых ошибок у конечных пользователей.

### Решение

Чтобы полностью подавить ошибки в эксплуатационной среде, установите обе директивы `error_reporting` и `display_errors` в `php.ini` в положение `Off`:

```
; Disable error reporting
error_reporting = Off
display_errors = Off
```

### Обсуждение

Изменение конфигурации повлияет на все ваше приложение. Ошибки будут полностью подавлены и, даже если они возникнут, конечный пользователь никогда не увидит уведомления о них. Считается плохой практикой отображать ошибки или необработанные исключения непосредственно пользователям. Это также может привести к проблемам безопасности, если трассировка стека будет иметь публичный статус.

Однако если ваша программа поведет себя неправильно, то у команды разработчиков не появится логов для диагностики и устранения неполадок.

Если для рабочего экземпляра приложения оставить `display_errors` в значении `Off`, то ошибки от конечных пользователей будут по-прежнему скрыты, но возврат `error_reporting` к уровню по умолчанию позволит регистрировать все ошибки в журнале.

Возможно, существуют страницы с записями об уже известных ошибках (из-за устаревшего кода, плохо написанных зависимостей или известного «технического долга»), которые вы захотите пропустить. В таких ситуациях вы можете программно установить уровень отчетности об ошибках с помощью функции `error_reporting()`. Она принимает новый уровень сообщения об ошибках и возвращает тот уровень, который был установлен ранее (по умолчанию, если он прежде не был настроен).

Это позволяет использовать вызовы `error_reporting()`, чтобы обернуть проблемные блоки кода и предотвратить нагромождение несущественных ошибок в журналах. Например:

```
$error_level = error_reporting(E_ERROR); ❶
// ... Вызов другого кода вашего приложения.

error_reporting($error_level); ❷
```

**❶** Устанавливает уровень ошибок на абсолютный минимум, в том числе фатальные ошибки среди выполнения, которые останавливают выполнение сценария.

**❷** Возврат уровня ошибок к предыдущему состоянию.

По умолчанию используется уровень ошибок `E_ALL`, который включает в себя все ошибки, предупреждения и уведомления<sup>1</sup>. Допускается использовать целочисленные значения уровней, но PHP представляет несколько именованных констант, которые представляют каждую потенциальную настройку. Эти константы перечислены в табл. 12.1.



До PHP 8.0 уровень отчетности об ошибках по умолчанию начинался с `E_ALL`, а затем явно удалялись диагностические уведомления (`E_NOTICE`), строгие предупреждения о типах (`E_STRICT`) и уведомления об устаревании (`E_DEPRECATED`).

**Таблица 12.1.** Константы уровня сообщений об ошибках

Числовое значение	Константа	Описание
1	<code>E_ERROR</code>	Фатальные ошибки среди выполнения. Это неустранимые средствами самого скрипта ошибки. Выполнение скрипта в таком случае прекращается
2	<code>E_WARNING</code>	Предупреждения (некритические ошибки), которые не останавливают выполнение скрипта
4	<code>E_PARSE</code>	Ошибки на этапе компиляции. Должны генерироваться только парсером
8	<code>E_NOTICE</code>	Указывают на то, что во время выполнения скрипта произошло что-то, что может указывать на ошибку, хотя это может происходить и при обычном выполнении программы

Продолжение ↗

<sup>1</sup> Уровень ошибок по умолчанию можно задать непосредственно в файле `php.ini`, и во многих средах он может быть установлен не на `E_ALL`, а на что-то другое. Проверьте конфигурацию своего окружения.

**Таблица 12.1 (продолжение)**

Числовое значение	Константа	Описание
16	E_CORE_ERROR	Фатальные ошибки, возникающие во время запуска PHP. Схожи с E_ERROR, за исключением того, что они генерируются ядром PHP
32	E_CORE_WARNING	Предупреждения (некритические ошибки), возникающие во время запуска PHP. Такие предупреждения схожи с E_WARNING, только генерируются ядром PHP
64	E_COMPILE_ERROR	Фатальные ошибки на этапе компиляции. Схожи с E_ERROR, только генерируются скриптовым движком Zend
128	E_COMPILE_WARNING	Предупреждения на этапе компиляции (некритические ошибки). Схожи с E_WARNING, только генерируются скриптовым движком Zend
256	E_USER_ERROR	Сообщения об ошибках, генерируемые пользователем. Схожи с E_ERROR, только генерируются в коде скрипта средствами PHP-функции trigger_error() ( <a href="https://oreil.ly/eNgVf">https://oreil.ly/eNgVf</a> )
512	E_USER_WARNING	Предупреждения, сгенерированные пользователем. Схожи с E_WARNING, за исключением того, что они генерируются в коде скрипта средствами функции PHP trigger_error()
1024	E_USER_NOTICE	Уведомления, сгенерированные пользователем. Такие уведомления схожи с E_NOTICE, за исключением того, что они генерируются в коде скрипта, средствами функции PHP trigger_error()
2048	E_STRICT	Включаются для того, чтобы PHP предлагал изменения в коде, которые обеспечат лучшее взаимодействие и совместимость кода
4096	E_RECOVERABLE_ERROR	Фатальные ошибки с возможностью обработки. Указывают, что, вероятно, возникла опасная ситуация, но при этом скриптовый движок остается в стабильном состоянии. Если такая ошибка не обрабатывается функцией, определенной пользователем для обработки ошибок, выполнение приложения прерывается, как происходит при ошибках E_ERROR
8192	E_DEPRECATED	Уведомления среди выполнения об использовании устаревших конструкций. Включаются для того, чтобы получать предупреждения о коде, который не будет работать в следующих версиях PHP
16384	E_USER_DEPRECATED	Уведомления среди выполнения об использовании устаревших конструкций, сгенерированные пользователем. Такие уведомления схожи с E_DEPRECATED за исключением того, что они генерируются в коде скрипта с помощью функции PHP trigger_error()
32767	E_ALL	Все поддерживаемые ошибки, предупреждения и замечания

Обратите внимание, что PHP предоставляет возможность комбинировать уровни ошибок с помощью бинарных операций, создавая битовую маску. Простой уровень отчетности может включать только ошибки, предупреждения и ошибки парсера (без учета ядра, ошибок пользователя и уведомлений). Этот уровень задается следующим образом:

```
error_reporting(E_ERROR | E_WARNING | E_PARSE);
```

## Читайте также

Документация по функции `error_reporting()` (<https://oreil.ly/b4eIH>), директиве `error_reporting` (<https://oreil.ly/t5IW2>) и директиве `display_errors` (<https://oreil.ly/lxXNs>).

# 12.4. Использование пользовательского обработчика ошибок

## Задача

Вы хотите настроить собственный способ обработки и отображения ошибок.

## Решение

Определите свой обработчик как вызываемую функцию в PHP, а затем передайте эту функцию в `set_error_handler()` следующим образом:

```
function my_error_handler(int $num, string $str, string $file, int $line)
{
    echo "Возникла ошибка $num в $file в строке $line: $str" . PHP_EOL;
}

set_error_handler('my_error_handler');
```

## Обсуждение

PHP применит пользовательский обработчик в тех ситуациях, когда ошибка считается исправимой. Однако фатальные ошибки, ошибки ядра и проблемы во время компиляции (например, ошибки парсера) приводят к приостановке или полному прерыванию программы и не обрабатываются пользовательской функцией (к последнему относятся ошибки `E_ERROR`, `E_PARSE`, `E_CORE_ERROR`, `E_CORE_WARNING`,

`E_COMPILE_ERROR` и `E_COMPILE_WARNING`). Кроме того, большинство ошибок `E_STRICT` в файле, который вызывал функцию `set_error_handler()`, также не могут быть зафиксированы, так как они возникнут до регистрации пользовательского обработчика.

Если вы определите пользовательский обработчик ошибок, аналогичный тому, что представлен в «Решении», то при любых перехватываемых ошибках будут вызываться эта функция и выводиться данные на экран. Как показано в примере 12.3, попытка выдать `echo` неопределенной переменной приведет к ошибке `E_WARNING`.

### Пример 12.3. Перехват восстановимых ошибок среды выполнения

```
echo $foo;
```

Если определить и зарегистрировать `my_error_handler()` из примера выше, то ошибочный код в примере 12.3 выведет на экран следующий текст, ссылающийся на целочисленное значение типа ошибки `E_WARNING`:

```
Возникла ошибка 2 в коде php shell в строке 1: Неопределенная переменная $foo
```

Обрабатывая ошибку в своем коде, вы должны принять решение о дальнейших действиях. Если ошибка может нарушить работу приложения, стоит вызвать функцию `die()`, чтобы прервать выполнение программы. PHP не будет делать этого за вас вне обработчика и продолжит обработку приложения, как если бы ошибки не возникло.

Чтобы восстановить исходный обработчик ошибок (по умолчанию), воспользуйтесь функцией `restore_error_handler()`. Она отменяет предыдущую регистрацию обработчика и восстанавливает тот, который был зарегистрирован ранее.

Аналогично PHP позволяет регистрировать (и восстанавливать) пользовательские обработчики исключений. Они работают так же, как и обработчики ошибок, только фиксируют любое исключение, брошенное вне блока `try/catch`. В этом случае выполнение программы будет остановлено после вызова пользовательского обработчика исключений.

Для более подробной информации об исключениях ознакомьтесь с рецептом 12.2 и документацией для функций `set_exception_handler()` ([https://oreil.ly/\\_pf4H](https://oreil.ly/_pf4H)) и `restore_exception_handler()` (<https://oreil.ly/TOEuz>).

## Читайте также

Документация по `set_error_handler()` (<https://oreil.ly/IAh69>) и `restore_error_handler()` ([https://oreil.ly/SIT\\_d](https://oreil.ly/SIT_d)).

## 12.5. Регистрация ошибок во внешний поток

### Задача

Вы хотите записывать ошибки приложения в файл или внешний источник для последующей отладки.

### Решение

Используйте функцию `error_log()` для записи ошибок в стандартный файл журнала следующим образом:

```
$user_input = json_decode($raw_input);
if (json_last_error() != JSON_ERROR_NONE) {
    error_log('JSON Error #' . json_last_error() . ': ' . $raw_input);
}
```

### Обсуждение

По умолчанию функция `error_log()` записывает ошибки в то место, которое указано в директиве `error_log` (<https://oreil.ly/3lVPn>) файла `php.ini`. В системах на базе Unix этот файл обычно располагается в каталоге `/var/log`, однако это можно изменить по вашему усмотрению.

Необязательный второй параметр `error_log()` позволяет маршрутизировать сообщения об ошибках при необходимости. Если сервер настроен на отправку электронной почты, вы можете указать тип сообщения 1 и предоставить адрес электронной почты в дополнительном третьем параметре для отправки ошибок по электронной почте:

```
error_log('Какое-то сообщение об ошибке', 1, 'developer@somedomain.tld');
```



По сути, функция `error_log()` использует ту же технологию, что и `mail()` для отправки ошибок по электронной почте. Во многих случаях это может быть отключено из соображений безопасности. Убедитесь в работоспособности почтовых систем, прежде чем полагаться на эту функциональность, особенно в эксплуатационной среде.

Кроме того, вы можете указать файл, отличный от стандартного расположения журналов, и передать целое число 3 как тип сообщения. Вместо записи в стандартные журналы PHP добавит сообщение непосредственно в этот файл. Например:

```
error_log('Какое-то сообщение об ошибке', 3, 'error_log.txt');
```



При записи ошибок непосредственно в файл с помощью функции `error_log()` система не будет автоматически добавлять символ новой строки. Вам придется либо добавить `PHP_EOL` к любой строке, либо закодировать символы новой строки `\r\n`.

В главе 11 подробно рассматриваются файловый протокол, а также другие потоки, реализуемые PHP. Помните, что прямое обращение к файлу неявно использует протокол `file://`, так что в действительности вы регистрируете ошибки в файловом потоке с помощью предыдущего блока кода. С тем же успехом можно ссылаться на любой другой вид потока, если правильно указан его протокол. В следующем примере ошибки записываются непосредственно в стандартный поток ошибок консоли:

```
error_log('Какое-то сообщение об ошибке', 3, 'php://stderr');
```

## Читайте также

Документация по функции `error_log()` (<https://oreil.ly/QUQRH>). Рецепт 13.5, в котором рассказывается о Monolog — более полной библиотеке протоколирования для PHP-приложений.

## ГЛАВА 13

---

# Отладка и тестирование

Несмотря на все усилия разработчиков, идеальный код остается недостижимой мечтой. Вы неизбежно столкнетесь с ошибкой, которая повлияет на работу приложения или вызовет разочарование конечного пользователя, когда что-то пойдет не так, как задумывалось.

Правильная обработка ошибок имеет важное значение для обеспечения стабильности и надежности приложения<sup>1</sup>. Однако не все ошибки, возникающие в приложении, являются ожидаемыми или даже исправимыми. В таких случаях вы должны понимать, как правильно выполнять отладку приложения, то есть найти проблемную строку кода для ее дальнейшей корректировки.

Одним из наиболее распространенных методов, используемых PHP-разработчиками для отладки кода, является оператор `echo`. Без формального отладчика часто можно увидеть код, замусоренный заявлениеми `echo "Здесь!"`, которые предназначены для того, чтобы помочь команде отследить места возникновения проблем.

Фреймворк Laravel популяризировал функцию `dd()` (сокращение от `dump and die`) (<https://oreil.ly/N-bOz>). Она фактически предоставляется модулем Symfony — `var-dumper` (<https://oreil.ly/8pXGo>) и эффективно работает как в интерфейсе командной строки PHP, так и при использовании интерактивного отладчика. Сама функция определяется следующим образом:

```
function dd(...$vars): void
{
    if (!in_array(\PHP_SAPI, ['cli', 'phpdbg'], true) && !headers_sent()) {
        header('HTTP/1.1 500 Internal Server Error');
    }

    foreach ($vars as $v) {
        VarDumper::dump($v);
    }
    exit(1);
}
```

---

<sup>1</sup> Подробнее об обработке ошибок читайте в главе 12.

Вышеупомянутая функция, используемая в приложении Laravel, выведет на экран содержимое любой переменной, которую вы ей передадите, а затем немедленно остановит выполнение программы. Хотя использование этой функции не является самым элегантным способом отладки приложения (как и echo), она быстра, надежна и часто применяется разработчиками в условиях нехватки времени для отладки системы.

Один из лучших способов отладки кода — модульное тестирование. Разбив код на мельчайшие логические единицы, можно написать дополнительный код, который автоматически проверяет работу этих логических блоков. Затем эти тесты подключаются к конвейеру сборки и развертывания, позволяя убедиться в том, что в вашем приложении не осталось ошибок.

Проект PHPUnit (<https://phpunit.de/>) с открытым исходным кодом предоставляет простой и понятный инструмент для модульного тестирования всего вашего приложения и автоматического тестирования его функциональности. Все тесты пишутся на PHP. Они напрямую нагружают функции и классы приложения и явно документируют ожидаемое поведение приложения.



Альтернативой PHPUnit является библиотека Behat (<https://oreil.ly/mAWR5>) с открытым исходным кодом. В то время как PHPUnit ориентирован на методологию разработки через тестирование (TDD), Behat фокусируется на парадигме разработки, основанной на описании поведения (BDD). Оба этих подхода одинаково эффективны для тестирования кода, и ваша команда должна решить, какого из них придерживаться. Тем не менее PHPUnit — более авторитетный проект, и мы будем применять его на протяжении всей главы.

Безусловно, лучший способ отладки кода — это использование интерактивного отладчика. Xdebug (<https://xdebug.org/>) — это отладочное расширение для PHP, которое предоставляет мощные инструменты для отладки, трассировки, профилирования и анализа кода. Оно также интегрируется с такими проектами, как PHPUnit, для демонстрации покрытия кода приложения тестами. Более того, Xdebug обеспечивает интерактивную, пошаговую отладку вашего приложения.

Вооружившись Xdebug и совместимой IDE, вы сможете устанавливать в коде специальные маркеры, называемые точками останова (breakpoints). Когда приложение достигает этих точек, оно приостанавливает свое выполнение и позволяет вам в интерактивном режиме просмотреть состояние приложения. Это означает, что вы можете обозреть все переменные в области видимости, узнать, откуда они взялись, и продолжить выполнение программы по одной команде за раз в поисках ошибок. Это, без сомнения, самый мощный инструмент в арсенале PHP-разработчика!

Следующие рецепты посвящены основам отладки PHP-приложений. Вы узнаете, как настроить интерактивную отладку, обрабатывать ошибки, правильно тестиировать код для предотвращения регресса и быстро определять, когда и где произошло нежелательное изменение.

## 13.1. Использование расширения отладчика

### Задача

Вы хотите использовать внешний отладчик для проверки и управления приложением, чтобы выявлять, профилировать и устранять ошибки в бизнес-логике.

### Решение

Установите расширение Xdebug. Его можно установить непосредственно в операционных системах Linux с помощью менеджера пакетов по умолчанию. Ниже показано, как это сделать на Ubuntu:

```
$ sudo apt install php-xdebug
```

Или, чтобы установить актуальную версию проекта, воспользуйтесь менеджером расширений PECL:

```
$ pecl install xdebug
```

Как только Xdebug появится в вашей системе, он автоматически включит отображение ошибок, представив богатые трассировки стека и отладочную информацию, чтобы облегчить обнаружение ошибок, если что-то идет не так.

### Обсуждение

Xdebug — это мощное расширение для PHP, позволяющее полностью тестировать, профилировать и отлаживать приложения эффективными способами, которые изначально не поддерживаются языком. Одна из наиболее полезных функций, доступных по умолчанию без дополнительных настроек, — это значительное улучшение отчетов об ошибках.

По умолчанию Xdebug автоматически фиксирует все ошибки, возникающие в приложении, и предоставляет дополнительную информацию о них:

- стек вызовов (как показано на рис. 13.1), включая данные о времени и использовании памяти (это помогает определить момент сбоя в программе и место в коде, где происходили вызовы функций);
- переменные из локальной области видимости, чтобы не гадать, какие данные были в памяти в момент возникновения ошибки.

Расширенная интеграция с такими инструментами, как Webgrind (<https://oreil.ly/OXg9b>), также позволяет динамически оценивать производительность приложения. Xdebug (опционально) будет фиксировать время выполнения каждого вызова функции и записывать как это время, так и «стоимость» вызова функции на диск.

Затем приложение Webgrind предлагает удобное визуальное представление, которое поможет вам выявить узкие места в коде и оптимизировать программу при необходимости.

(!) Fatal error: Maximum execution time of 1 second exceeded in /home/httpd/html/test/xdebug/docs/stack.php on line 70				
Call Stack				
#	Time	Memory	Function	Location
1	0.0007	374160	{main}()	.../stack.php:0
2	0.0010	376762	foo( \$a = array (42 => FALSE, "foo" => 9121240, 43 => class stdClass { public \$bar = 100 }, 44 => class stdClass {} ), 45 => resource(3) of type (stream))	.../stack.php:84

Variables in local scope (#2)

\$a =	/home/httpd/html/test/xdebug/docs/stack.php:70: array (size=5) 42 => boolean false 'foo' => int 9121240 43 => object(stdClass)[1] public 'bar' => int 100 44 => object(stdClass)[2] 45 => resource(3, stream)
\$i =	/home/httpd/html/test/xdebug/docs/stack.php:70:int 3047104

**Рис. 13.1.** Xdebug обогащает и форматирует информацию, представленную при возникновении ошибок

Также допускается напрямую связать Xdebug со средой разработки для пошаговой отладки (<https://oreil.ly/FK9iz>). Сопрягая среду (например, Visual Studio Code (<https://oreil.ly/u4dZy>)) с конфигурацией Xdebug, можно размещать точки останова в коде и буквально приостанавливать выполнение, когда интерпретатор PHP попадает в эти точки.



Расширение PHP Debug (<https://oreil.ly/vVCVY>) значительно упрощает интеграцию Xdebug с Visual Studio Code. Оно добавляет в вашу IDE все необходимые интерфейсы, включая точки останова и интроспекцию среды. Кроме того, расширение поддерживается сообществом Xdebug, так что вы можете быть уверены в его актуальности.

При отладке в пошаговом режиме приложение приостанавливается на точке останова и предоставляет вам прямой доступ ко всем переменным в области действия программы. У вас появляется возможность проверять и изменять эти переменные для тестирования окружения. Кроме того, во время такой паузы вы получаете доступ к консоли приложения, позволяющей определить, что происходит. Стек

вызовов доступен напрямую, поэтому не возникает проблем с тем, чтобы узнать, какая именно функция или метод объекта привели к точке останова, и внести необходимые изменения.

Находясь в точке останова, вы можете либо выполнять программу по одной строке за раз, либо «продолжить» выполнение до следующей точки останова или до первой ошибки. Точки останова также допускается отключить, не удаляя их из IDE, так что разрешается продолжать выполнение по мере необходимости, но вернуться к определенным проблемным местам позже, если потребуется.



Xdebug — это очень мощный инструмент разработки для любой команды PHP-разработчиков. Однако, как известно, даже самое маленькое приложение может значительно замедлиться из-за него. Убедитесь, что вы используете это расширение только для локальной разработки или в защищенных средах с тестовыми развертываниями. Никогда не развертывайте ваше приложение в эксплуатации с установленным Xdebug!

## Читайте также

Документация и домашняя страница для Xdebug (<https://xdebug.org/>).

## 13.2. Написание модульного теста

### Задача

Вы хотите проверить поведение определенного фрагмента кода, чтобы убедиться, что будущий рефакторинг не повлияет на функциональность вашего приложения.

### Решение

Напишите класс, который расширяет `TestCase` из PHPUnit и явно тестирует поведение приложения. Например, если ваша функция предназначена для извлечения доменного имени из адреса электронной почты, вы должны определить ее следующим образом:

```
function extractDomain(string $email): string
{
    $parts = explode('@', $email);

    return $parts[1];
}
```

Затем создайте класс для тестирования и проверки функциональности этого кода:

```
use PHPUnit\Framework\TestCase;

final class FunctionTest extends TestCase
{
    public function testSimpleDomainExtraction()
    {
        $this->assertEquals('example.com', extractDomain('php@example.com'));
    }
}
```

## Обсуждение

Самый важный аспект PHPUnit — это организация вашего проекта. Прежде всего проект должен использовать Composer для автозагрузки как вашего прикладного кода, так и всех зависимостей (включая и PHPUnit)<sup>1</sup>. Обычно код приложения размещается в каталоге `src/` в корне проекта, а тестовый код — в каталоге `tests/` рядом с ним.

В примере выше вы поместите функцию `extractDomain()` в `src/functions.php`, а класс `FunctionTest` — в `tests/FunctionTest.php`. Предполагая, что автозагрузка корректно настроена через Composer, вы можете запустить тест с помощью инструмента командной строки PHPUnit:

```
$ ./vendor/bin/phpunit tests
```

По умолчанию эта команда автоматически определит и запустит каждый тестовый класс, определенный в каталоге `tests/`, через PHPUnit. Чтобы более полно контролировать работу PHPUnit, допускается использовать локальный конфигурационный файл для описания наборов тестов, списков разрешенных файлов и настройки любых специфических переменных окружения, необходимых во время тестирования.

Конфигурация на основе XML-файла редко применяется за исключением сложных или запутанных проектов, но в документации проекта (<https://oreil.ly/Gz86n>) подробно описывается, как произвести настройку. Простой файл `phpunit.xml`, который можно использовать в этом рецепте или других подобных сценариях, будет выглядеть как в примере 13.1.

### Пример 13.1. Базовая XML-конфигурация PHPUnit

```
<?xml version="1.0" encoding="UTF-8"?

<phpunit bootstrap="vendor/autoload.php"
    backupGlobals="false"
```

---

<sup>1</sup> Подробнее о Composer см. в рецепте 15.1.

```
    backupStaticAttributes="false"
    colors="true"
    convertErrorsToExceptions="true"
    convertNoticesToExceptions="true"
    convertWarningsToExceptions="true"
    processIsolation="false"
    stopOnFailure="false">

<coverage>
    <include>
        <directory suffix=".php">src</directory>
    </include>
</coverage>

<testsuites>
    <testsuite name="unit">.
        <directory>tests</directory>
    </testsuite>
</testsuites>

<php>
    <env name="APP_ENV" value="testing"/>
</php>

</phpunit>
```

Вооружившись предыдущим файлом `phpunit.xml` в своем проекте, вам достаточно вызвать сам PHPUnit для запуска тестов. Больше нет необходимости указывать каталог `tests/`, так как теперь это обеспечивается определением `testsuite` в конфигурации приложения.

Аналогично вы можете задать несколько наборов тестов для разных сценариев. Допустим, один набор тестов создается командой разработчиков в процессе написания кода (модульные тесты в приведенном выше примере). Другую группу тестов пишет QA-команда для воспроизведения ошибок, о которых сообщают пользователи (тесты на регрессию). Преимущество второго набора тестов в том, что допускается изменять структуру своего приложения до тех пор, пока все ошибки не будут исправлены (успешное прохождение тестов), при этом гарантируя, что общее поведение вашего приложения не было изменено.

Вы также можете быть уверены, что старые ошибки не появятся вновь!

Кроме того, у вас есть возможность выбирать, какой набор тестов запускать в то или иное время, передавая в PHPUnit дополнительный флаг `--testsuite`. Большинство тестов проводятся быстро и лишний раз не тратят время вашей команды. Такие тесты стоит запускать как можно чаще во время разработки, чтобы убедиться в корректной работе кода и отсутствии новых (или старых) ошибок в нем. Однако иногда требуется запускать и ресурсозатратные тесты. Их следует хранить в отдельном тестовом наборе, чтобы выполнять их по мере необходимости. Подобные

тесты могут использоваться перед развертыванием, не замедляя повседневную разработку при частом выполнении обычных тестов.

Тесты функций, как в примере из «Решения», довольно просты. Тесты объектов похожи тем, что вы инстанцируете объект внутри теста и применяете его методы. Однако самое сложное — смоделировать несколько возможных входов в определенную функцию или метод. PHPUnit решает эту проблему с помощью поставщиков (провайдеров) данных.

Для наглядности рассмотрим функцию `add()` в примере 13.2. Она явно использует нестрогую типизацию для сложения двух значений (независимо от их типов).

### Пример 13.2. Простая функция сложения

```
function add($a, $b): mixed
{
    return $a + $b;
}
```

Поскольку параметры здесь могут быть разных типов (`int/int`, `int/float`, `string/float` и т. д.), вам следует протестировать различные комбинации, чтобы убедиться, что ничего не сломается. Такая тестовая структура будет выглядеть как класс (см. пример 13.3).

### Пример 13.3. Простой тест операции сложения PHP

```
final class FunctionTest extends TestCase
{
    // ...

    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected): void
    {
        $this->assertSame($expected, add($a, $b));
    }

    public function additionProvider(): array
    {
        return [
            [2, 3, 5],
            [2, 3.0, 5.0],
            [2.0, '3', 5.0],
            ['2', 3, 5]
        ];
    }
}
```

Аннотация `@dataProvider` сообщает PHPUnit имя функции в классе теста, которая должна предоставлять данные для тестирования. Вместо того чтобы писать четыре

отдельных теста, вы дали PHPUnit возможность запускать один тест четыре раза с разными входными и ожидаемыми выходными данными. Конечный результат тот же — четыре отдельных теста вашей функции `add()`, — но без необходимости явно писать эти дополнительные тесты.

Учитывая структуру функции `add()`, определенной в примере 13.2, вы можете столкнуться с некоторыми ограничениями типов в PHP. Хотя в функцию можно передавать числовые строки (они приводятся к числовым значениям перед сложением), подобное действие с нечисловыми данными приведет к предупреждению PHP. В среде, где пользовательский ввод передается в эту функцию, такая проблема может и будет возникать. Лучше всего заранее защититься от нее, явно проверив входные значения с помощью функции `is_numeric()` и отбросив известное исключение, которое может быть зафиксировано в другом месте.

Для начала напишите новый тест, который будет ожидать исключения и проверять, что оно обрабатывается соответствующим образом. Такой тест будет выглядеть как в примере 13.4.

#### Пример 13.4. Проверка ожидаемого наличия исключений в коде

```
final class FunctionTest extends TestCase
{
    // ...

    /**
     * @dataProvider invalidAdditionProvider
     */
    public function testInvalidInput($a, $b, $expected): void
    {
        $this->expectException(InvalidArgumentException::class);
        add($a, $b);
    }

    public function invalidAdditionProvider(): array
    {
        return [
            [1, 'invalid', null],
            ['invalid', 1, null],
            ['invalid', 'invalid', null]
        ];
    }
}
```



Когда вы пишете тесты до изменения кода, это дает вам четкую цель, которую нужно достичь при рефакторинге. Однако этот новый тест не будет работать до тех пор, пока вы не преобразуете код приложения. Будьте внимательны и не добавляйте в систему контроля версий проекта не проходящие тесты, иначе вы поставите под угрозу способность вашей команды к непрерывной интеграции!

После добавления нового теста весь набор тестов перестает работать, поскольку функция теперь не соответствует документированному или ожидаемому поведению. Добавьте соответствующие проверки `is_numeric()` в функцию следующим образом:

```
function add($a, $b): mixed
{
    if (!is_numeric($a) || !is_numeric($b)) {
        throw new InvalidArgumentException('Ввод должен быть числовым!');
    }

    return $a + $b;
}
```

Модульные тесты — это эффективный способ документировать ожидаемое и соответствующее поведение вашего приложения, поскольку они представляют собой исполняемый код, который также подтверждает корректность работы программы. Вы можете проводить как позитивное, так и негативное тестирование и даже имитировать различные зависимости в вашем коде.

Проект PHPUnit также предоставляет возможность проактивно определять процент покрытия (<https://oreil.ly/PEdVd>) кода модульными тестами (<https://oreil.ly/PEdVd>). Более высокий процент не гарантирует отсутствия ошибок, но обеспечивает надежный способ быстрого обнаружения и исправления ошибок с минимальным вмешательством в работу конечных пользователей.

## Читайте также

Документация о том, как использовать PHPUnit (<https://oreil.ly/5oYv4>).

## 13.3. Автоматизация модульных тестов

### Задача

Вы хотите, чтобы модульные тесты вашего проекта запускались автоматически, без участия пользователя, перед любыми изменениями в коде, которые будут сохранены в системе контроля версий Git.

### Решение

Используйте Git-хук для фиксации коммита (commit), чтобы автоматически запускать модульные тесты перед локальным сохранением изменений. Возьмем хук `pre-commit` из примера 13.5. Он будет машинально запускать PHPUnit каждый раз, когда пользователь выполняет `git commit`, но до записи данных в репозиторий.

**Пример 13.5.** Простой крючок предварительной комиссии Git для PHPUnit

```
#!/usr/bin/env php
<?php

echo "Запуск тестов... ";
exec('vendor/bin/phpunit', $output, $returnCode);

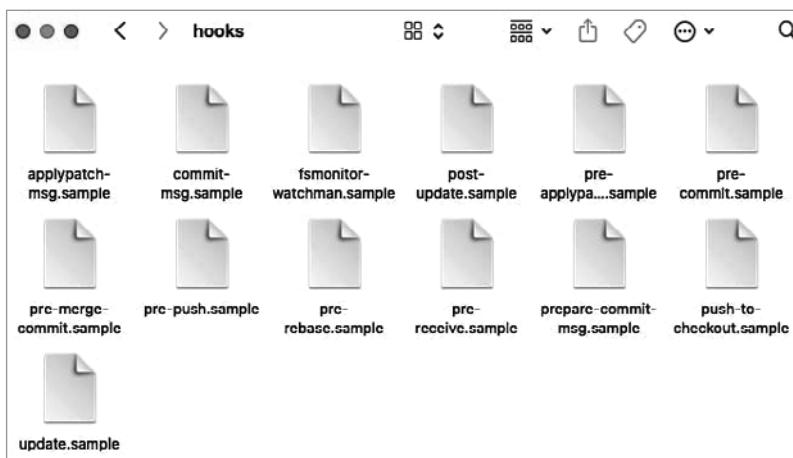
if ($returnCode !== 0) {
    echo PHP_EOL . implode($output, PHP_EOL) . PHP_EOL;
    echo "Прерывание коммита..." . PHP_EOL;
    exit(1);
}

echo array_pop($output) . PHP_EOL;
exit(0);
```

## Обсуждение

Git — самая популярная распределенная система контроля версий. Она имеет открытый исходный код и очень гибкая в плане размещения репозиториев и настройки рабочих процессов и структур проектов.

В частности, Git позволяет настраивать проект с помощью хуков. Git-хуки — это скрипты, которые автоматически выполняются в определенных точках рабочего процесса Git. Ваши хуки находятся в каталоге `.git/hooks` вашего проекта вместе с другой информацией, которую Git использует для отслеживания состояния проекта. По умолчанию даже пустой репозиторий Git включает в себя несколько образцов хуков (рис. 13.2).



**Рис. 13.2.** Git инициализирует даже пустой репозиторий с помощью хуков-примеров

Каждый из образцов хуков имеет расширение `.sample`, которое по умолчанию отключает его. Если вы хотите использовать один из этих хуков, просто удалите расширение, и хук активируется при соответствующем действии.

В случае автоматизированного тестирования вам явно нужен хук `pre-commit`, и вам следует создать файл с таким именем и содержимым, как показано в примере 13.5. С установленным хуком Git всегда будет запускать этот скрипт перед фиксацией кода.

Статус выхода 0 в конце сценария говорит Git, что все в порядке и он может продолжить сохранение. Если какой-либо из ваших модульных тестов не проходит, статус выхода 1 сигнализирует о проблеме, фиксация в этом случае прервется без изменения в репозитории.

Если вы абсолютно уверены в своих действиях и по каким-либо причинам вам нужно обойти хук, добавьте флаг `--no-verify` при отправке кода.



Хук `pre-commit` полностью выполняется на стороне клиента и не входит в ваш репозиторий кода. Каждый разработчик должен будет установить хук индивидуально. Кроме командных инструкций или политики компании нет эффективного способа гарантировать использование хука (или что кто-то не обойдет его с помощью `--no-verify`).

Если ваша команда работает с Git для контроля версий, есть большая вероятность, что вы также используете GitHub для хостинга вашего репозитория. Если это так, вы можете задействовать GitHub Actions для запуска тестов PHPUnit (<https://oreil.ly/BmGZC>) на сервере GitHub в рамках процесса интеграции и развертывания.

Локальный прогон тестов помогает защититься от случайной фиксации регрессий (кода, который повторно вносит известную ошибку) или другой ошибки в репозиторий. Выполнение тех же тестов в облаке обеспечивает еще большую функциональность, поскольку вы можете делать это для различных конфигураций. Обычно разработчики запускают локально только одну версию PHP, но в целом допускается выполнять код приложения и тесты в контейнерах на сервере, используя различные версии PHP или зависимостей.

Применение GitHub Actions для запуска тестов также дает следующие преимущества:

- если новый разработчик еще не настроил `pre-commit` в Git и попытается зафиксировать код с ошибкой, Action runner немедленно пометит коммит как нерабочий и предохранит разработчика от ошибки;

- использование детерминированной среды в облаке защитит вашу команду от таких проблем, как «у меня все работало, не знаю, что у вас не так», — когда код работает в одной среде, а затем перестает работать в другой, имеющей иную конфигурацию;
- ваш рабочий процесс интеграции и развертывания должен создавать новые артефакты развертывания после каждой фиксации. Подключение этого процесса сборки к вашим тестам гарантирует, что каждый артефакт не содержит известных дефектов и, по сути, может быть развернут.

## Читайте также

Документация по настройке Git с помощью хуков (<https://oreil.ly/TzVOA>).

## 13.4. Использование статического анализа кода

### Задача

Вы хотите применить внешний инструмент, чтобы еще до запуска кода убедиться в отсутствии ошибок.

### Решение

Используйте инструмент статического анализа кода, например PHPStan (<https://phpstan.org/>).

### Обсуждение

PHPStan — это инструмент статического анализа кода для PHP, предназначенный для минимизации ошибок в коде на стадии разработки. Он наиболее эффективен в паре со строгой типизацией, помогая вашей команде писать более надежные и понятные приложения<sup>1</sup>.

Как и многие другие инструменты разработки, PHPStan можно установить в ваш проект через Composer:

```
$ composer require --dev phpstan/phpstan
```

<sup>1</sup> Строгая типизация подробно рассматривается в рецепте 3.4.

Затем не составит труда запустить PHPStan в своем проекте, чтобы проанализировать как код приложения, так и непосредственно тесты:

```
$ ./vendor/bin/phpstan analyze src tests
```

По умолчанию PHPStan запускается на уровне 0, что является самым низким уровнем статического анализа. Для указания более высокого уровня сканирования, передайте флаг `--level` в командную строку с числом, большим 0. В табл. 13.1 перечислены доступные уровни. Для хорошо поддерживаемых, строго типизированных приложений анализ на уровне 9 — лучший способ обеспечить качество кода.

**Таблица 13.1.** Уровни правил PHPStan

Уровень	Описание
0	Базовая проверка на наличие неизвестных классов, функций или методов класса. Также проверяются количество аргументов в вызовах функций и любые переменные, которые никогда не были определены
1	Проверяет возможные неопределенные переменные, неизвестные магические методы и динамические свойства, получаемые через магические геттеры
2	Проверяет неизвестные методы во всех выражениях и функциональной документации (PHPDoc)
3	Проверяет типы возвращаемых значений и типы, присваиваемые свойствам
4	Проверяет мертвый код (например, условия, которые всегда <code>false</code> ) и недостижимые пути кода
5	Проверяет типы аргументов
6	Сообщает о недостающих типах
7	Сообщает о частично неправильных объединениях типов (примеры объединения типов см. в обсуждении примера 3.9)
8	Проверяет вызовы любых методов или доступ к свойствам для типов со значением <code>nullable</code>
9	Строгие проверки на использование смешанной типа <code>mixed</code>

---

После анализа вы можете заняться обновлением приложения для исправления основных недостатков и ошибок валидации. У вас также есть возможность автоматизировать статический анализ, как и тестирование (см. рецепт 13.3).

## Читайте также

Домашняя страница проекта PHPStan и документация (<https://phpstan.org/>).

## 13.5. Запись отладочной информации

### Задача

Вы хотите регистрировать информацию о проблемах в своей программе, чтобы впоследствии отладить возможные ошибки.

### Решение

Используйте проект с открытым исходным кодом Monolog (<https://oreil.ly/yDIM7>), чтобы реализовать полноценный интерфейс протоколирования. Сначала установите пакет с помощью Composer следующим образом:

```
$ composer require monolog/monolog
```

Затем подключите логгер к своему приложению, чтобы при необходимости оно могло выдавать предупреждения и ошибки. Например:

```
use Monolog\Level;
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$logPath = getenv('LOG_PATH') ?? '/var/log/php/error.log';
$logLevel = getenv('LOG_LEVEL') !== false
    ? Level::fromInt(getenv('LOG_LEVEL'))
    : Level::Warning;

$logger = new Logger('default');
$logger->pushHandler(new StreamHandler($logPath, $logLevel));

$log->warning('Привет!');
$log->error('Мир!');
```

### Обсуждение

Самый простой способ регистрации информации в PHP — использовать встроенную функцию `error_log()` (<https://oreil.ly/kXYJP>). Она позволяет записывать ошибки либо в журнал ошибок сервера, либо в плоский файл, как настроено в `php.ini`. Единственная проблема заключается в том, что эта функция явно регистрирует ошибки в приложении.

В результате любое содержимое, фиксируемое функцией `error_log()`, рассматривается как ошибка всеми системами, анализирующими файл журнала. Это усложняет разграничение фактических ошибок (например, сбоев при входе пользователя в систему) и отладочных сообщений. Их смешивание может затруднить

конфигурирование среды выполнения, особенно если требуется отключить определенный тип протоколирования в различных окружениях. Обходной путь — обернуть все вызовы `error_log()` проверкой текущего уровня ведения журнала, как показано в примере 13.6.

**Пример 13.6.** Выборочное протоколирование ошибок с помощью функции `error_log()`

```
enum LogLevel: int ①
{
    case Debug    = 100;
    case Info     = 200;
    case Warning  = 300;
    case Error    = 400;
}

$logLevel = getenv('LOG_LEVEL') !== false ②
    ? LogLevel::from(intval(getenv('LOG_LEVEL')))
    : LogLevel::Debug;

// Какой-то код приложения ...
if (user_session_expired()) {
    if ($logLevel >= LogLevel::Info) { ③
        error_log('Сессия пользователя истекла. Выход из системы ...');
    }

    logout();
    exit;
}
```

❶ Самый простой способ перечислить уровни ведения журнала — это использовать литеральный тип `enum` в PHP.

❷ Уровень протоколирования должен быть доступен из системного окружения. Если это не предусмотрено, то следует применить стандартное значение по умолчанию, которое жестко задается в коде программы.

❸ При каждом вызове `error_log()` необходимо явно проверять текущий уровень и принимать решение, действительно ли следует записывать данное сообщение об ошибке в журнал.

Проблема примера 13.6 заключается не в использовании `enum` и не в необходимости динамически загружать уровни протоколирования из среды, а в том, что перед каждым вызовом `error_log()` нужно явно проверять этот уровень, чтобы убедиться, что программа действительно должна выдать ошибку. Такая частая проверка приводит к появлению большого количества «спагетти-кода» и делает ваше приложение сложным в обслуживании.

Опытный разработчик догадается, что идеальным решением здесь было бы обернуть всю логику ведения журнала (включая проверку уровня) в функциональный интерфейс, чтобы сохранить чистоту приложения. Это абсолютно правильный подход, и именно поэтому существует пакет Monolog!



Хотя Monolog является популярным пакетом PHP для протоколирования, это не единственный доступный пакет. Monolog реализует стандартный интерфейс PHP Logger (<https://oreil.ly/76eAV>); любой пакет, реализующий тот же интерфейс, можно использовать вместо Monolog для обеспечения аналогичной функциональности.

Monolog — нечто большее, чем просто запись строк в журнал ошибок. Он также поддерживает каналы, различные обработчики, процессоры и уровни протоколирования.

При инстанцировании нового логгера сначала определяется канал для этого объекта. Это позволит вам создать несколько экземпляров логгеров, хранить их содержимое раздельно и даже направлять их на различные средства вывода. По умолчанию для работы логгера требуется не только канал, но и обработчик, который нужно поместить в стек вызовов.

Обработчик ([https://oreil.ly/\\_1wLC](https://oreil.ly/_1wLC)) определяет, что Monolog должен делать с любым сообщением, переданным в определенный канал. Он может направлять данные в файл, хранить сообщения в базе данных, отправлять ошибки по электронной почте, уведомлять команду или канал в Slack о возникшей проблеме или даже взаимодействовать с такими системами, как RabbitMQ или Telegram.



Monolog также поддерживает различные средства форматирования, которые могут быть прикреплены к различным обработчикам. Каждый из этих средств определяет, как сообщения будут обработаны и отправлены конкретному обработчику, например в виде одностroчной строки, JSON-блока или документа Elasticsearch. Если вы не используете обработчик, которому требуются данные в определенном формате, скорее всего, вам подойдет формат по умолчанию.

Процессор — это дополнительный optionalный элемент, который может добавлять данные в сообщение. Так, процессор IntrospectionProcessor (<https://oreil.ly/jp-US>) автоматически добавит в журнал строку, файл, класс и/или метод, из которого был сделан вызов журнала. Базовая настройка Monolog для регистрации в плоский файл с интроспекцией будет выглядеть, как показано в примере 13.7.

### Пример 13.7. Конфигурация Monolog с интроспекцией

```
use Monolog\Level;
use Monolog\Logger;
use Monolog\Handler\StreamHandler;
use Monolog\Processor\IntrospectionProcessor;

$logger = new Logger('default');
$logger->pushHandler(new StreamHandler('/var/log/app.log', Level::Debug));
$logger->pushProcessor(new IntrospectionProcessor());

// ...

$logger->debug('Что-то произошло...');
```

Последняя строка в примере 13.7 вызывает сконфигурированный вами логгер и отправляет литеральную строку через процессор в обработчик, который вы подключили. Кроме того, вы можете передать дополнительные данные о контексте выполнения или самой ошибке в массиве в качестве необязательного второго параметра.

Даже без дополнительного контекста, если весь этот блок кода будет находиться в файле с названием `/src/app.php`, то в журнале приложения он выдаст нечто, напоминающее следующее:

```
[2023-01-08T22:02:00.734710+00:00] default.DEBUG: Что-то произошло ...
[] {"file":"/src/app.php", "line":15, "class":null, "callType":null,
"function":null}
```

Достаточно было создать одну строку текста (`Что-то произошло ...`), и `Monolog` автоматически фиксировал временную метку события, уровень ошибки и подробности о стеке вызовов благодаря зарегистрированному процессору. Благодаря всей этой информации отладка и исправление потенциальных ошибок становятся намного проще для вас и вашей команды.

`Monolog` также избавляет вас от необходимости проверять уровень ошибок при каждом вызове. Достаточно определять его в двух местах:

- при регистрации обработчика для самого экземпляра логгера (обработчик будет перехватывать только ошибки этого уровня или выше);
- при отправке сообщения в канал логгера (например, `::debug()` отправляет сообщение с явным уровнем ошибки `Debug`).

`Monolog` поддерживает восемь уровней ошибок, перечисленных в табл. 13.2. Все они иллюстрируют протокол `syslog`, описанный в RFC 5424 (<https://oreil.ly/Jtm9k>).

**Таблица 13.2.** Уровни ошибок в Monolog

Уровень ошибки	Метод логгера	Описание
Level::Debug	::debug()	Уровень отладки, который используется для записи подробной информации, необходимой для диагностики и устранения проблем
Level::Info	::info()	Предназначен для записи информации о текущем состоянии системы или ходе выполнения операций
Level::Notice	::notice()	Используется для регистрации уведомлений о событиях, которые не являются критическими, но могут быть полезны для мониторинга состояния приложения
Level::Warning	::warning()	Этот уровень предназначен для регистрации предупреждений о возможных проблемах, которые еще не произошли. Предупреждения могут указывать на проблемы с конфигурацией или потенциальные проблемы в будущем
Level::Error	::error()	На этом уровне регистрируются ошибки, которые могут вызвать сбои в работе приложения
Level::Critical	::critical()	Указывает на наличие критической проблемы, которая может привести к серьезным последствиям, если ее не устраниТЬ
Level::Alert	::alert()	Указывает на серьезные ошибки, требующие мгновенного внимания и реагирования. В критически важных приложениях такой уровень ошибок подразумевает вызов дежурного инженера
Level::Emergency	::emergency()	Означает, что приложение непригодно для использования

С помощью Monolog вы можете грамотно обернуть сообщения об ошибках в соответствующий метод логгера и определить, когда эти ошибки отправляются обработчику в зависимости от уровня ошибок, использованного при создании самого логгера. Если вы создадите логгер только для сообщений уровня Error и выше, ни один вызов ::debug() не будет записан в журнал. Возможность раздельного контроля вывода логов при эксплуатации и разработке жизненно важна для создания стабильного и хорошо протоколируемого приложения.

## Читайте также

Инструкции по использованию пакета Monolog ([https://oreil.ly/\\_5wx6](https://oreil.ly/_5wx6)).

## 13.6. Выгрузка содержимого переменных в виде строк

### Задача

Вы хотите проверить содержимое сложной переменной.

### Решение

Используйте `var_dump()`, чтобы преобразовать переменную в понятный человеку формат и вывести ее в текущий поток вывода (например, в консоль командной строки). Например:

```
$info = new stdClass;
$info->name = 'Book Reader';
$info->profession = 'PHP Developer';
$info->favorites = ['PHP', 'MySQL', 'Linux'];

var_dump($info);
```

При запуске в CLI предыдущий код выведет на консоль следующее сообщение:

```
object(stdClass)#1 (3) {
    ["name"]=>
        string(11) "Book Reader"
    ["profession"]=>
        string(13) "PHP Developer"
    ["favorites"]=>
        array(3) {
            [0]=>
                string(3) "PHP"
            [1]=>
                string(5) "MySQL"
            [2]=>
                string(5) "Linux"
        }
}
```

### Обсуждение

Каждая форма данных в PHP имеет строковое представление. Объекты могут перечислять свои типы, поля и методы, а массивы — свои члены. Скалярные типы способны раскрывать как свои типы, так и значения. Разработчики получают до-

ступ к внутреннему содержимому любой переменной одним из трех отличающихся, но равнозначных способов.

Функция `var_dump()`, используемая выше, непосредственно выводит содержимое переменной на консоль. Такое строковое представление включает в себя подробную информацию о типах, именах полей и значениях внутренних членов. Этот метод полезен для быстрой проверки содержимого переменной, но не более того.



Не допускайте попадания функции `var_dump()` в эксплуатационную среду. Эта функция не экранирует данные и может вывести несанкционированный пользовательский ввод, что создаст серьезную уязвимость в системе безопасности<sup>1</sup>.

Более полезной является функция `var_export()`. По умолчанию она также выводит содержимое любой переданной переменной, только формат вывода — сам исполняемый PHP-код. Тот же объект `$info` из «Решения» будет иметь следующий вид:

```
(object) Array(
    'name' => 'Book Reader',
    'profession' => 'PHP Developer',
    'favorites' =>
Array (
    0 => 'PHP',
    1 => 'MySQL',
    2 => 'Linux',
),
)
```

В отличие от `var_dump()`, функция `var_export()` принимает дополнительный необязательный второй параметр, который указывает функции вернуть результат, а не выводить его на экран. В итоге возвращается строковый литерал, представляющий содержимое переменной, который можно сохранить в другом месте для последующего использования.

Третья и последняя альтернатива — использовать функцию `print_r()`. Как и предыдущие, она выдает удобочитаемое представление содержимого переменной. Как и в случае с `var_export()`, здесь допускается передать дополнительный параметр, чтобы функция вернула вывод, а не отображала его на экране.

Однако не вся информация о типе выводится непосредственно через `print_r()`. Например, тот же объект `$info` в примере из «Решения» будет выведен следующим образом:

```
stdClass Object
```

<sup>1</sup> Подробнее о санитизации данных см. в рецепте 9.1.

```
(  
    [name] => Book Reader  
    [profession] => PHP Developer  
    [favorites] => Array ()  
        (  
            [0] => PHP  
            [1] => MySQL  
            [2] => Linux  
        )  
)
```

Каждая функция отображает различное количество информации, относящейся к рассматриваемой переменной. Какой вариант вам больше подходит, зависит от ваших потребностей. В контексте отладки или протоколирования возможность `var_export()` и `print_r()` возвращать строковое представление, а не выводить его напрямую в консоль была бы полезной, особенно в паре с таким инструментом, как Monolog, как описано в рецепте 13.5.

Если вы хотите экспортировать содержимое переменных, чтобы легко импортировать их обратно в PHP, то лучше всего подойдет `var_export()`. Если вы отлаживаете содержимое переменной и требуется информация о типе и размере, то вывод `var_dump()` по умолчанию будет наиболее информативным, даже если его нельзя напрямую экспорттировать как строку.

Если все же нужно задействовать функцию `var_dump()` и экспорттировать ее вывод в виде строки, вы можете воспользоваться буфером вывода (<https://oreil.ly/2AUks>). В частности, создайте его перед вызовом `var_dump()`, а затем сохраните содержимое этого буфера в переменную для дальнейшего применения, как показано в примере 13.8.

#### Пример 13.8. Буферизация вывода для захвата содержимого переменных

```
ob_start(); ❶  
var_dump($info); ❷  
  
$contents = ob_get_clean(); ❸
```

❶ Создание буфера вывода. Теперь любой код, выведенный на консоль после этого вызова, будет захвачен буфером.

❷ Вывод содержимого заданной переменной на консоль или в буфер.

❸ Получение содержимого буфера и его последующее удаление.

Результатом предыдущего примера будет строковое представление выгруженного содержимого `$info`, сохраненное в `$contents` для дальнейшего использования. Если продолжить выгружать содержимое самой `$contents`, то получится следующее:

```
string(244) "object(stdClass)#1 (3) {  
    ["name"]=>  
        string(11) "Book Reader"  
    ["profession"]=>  
        string(13) "PHP Developer"  
    ["favorites"]=>  
        array(3) {  
            [0]=>  
                string(3) "PHP"  
            [1]=>  
                string(5) "MySQL"  
            [2]=>  
                string(5) "Linux"  
        }  
}  
"
```

## Читайте также

Документация по `var_dump()` (<https://oreil.ly/uYuoV>), `var_export()` ([https://oreil.ly/V\\_vZ-](https://oreil.ly/V_vZ-)) и `print_r()` (<https://oreil.ly/0D891>).

# 13.7. Использование встроенного веб-сервера для быстрого запуска приложения

## Задача

Вы хотите запустить веб-приложение локально, не настраивая настоящий веб-сервер, например Apache или NGINX.

## Решение

Используйте встроенный в PHP веб-сервер для быстрого запуска скрипта, чтобы он был доступен из браузера. Например, если ваше приложение находится в каталоге `public_html/`, запустите веб-сервер следующим образом:

```
$ cd ~/public_html  
$ php -S localhost:8000
```

Затем наберите в адресной строке браузера `http://localhost:8000`, чтобы просмотреть любой файл (статический HTML, изображения или даже исполняемый PHP) в этом каталоге.

## Обсуждение

PHP CLI предоставляет встроенный веб-сервер, который упрощает тестирование или демонстрацию приложений или скриптов в контролируемой локальной среде. CLI поддерживает как запуск PHP-скриптов, так и возврат статического содержимого из запрашиваемого пути.

Статический контент может включать в себя отрисованные HTML-файлы или что-либо из следующих стандартных типов/расширений MIME:

```
.3gp, .apk, .avi, .bmp, .css, .csv, .doc, .docx, .flac, .gif, .gz, .gzip, .htm, .html, .ics, .jpe, .jpeg, .jpg, .js, .kml, .kmz, .m4a, .mov, .mp3, .mp4, .mpeg, .mpg, .odp, .ods, .odt, .oga, .ogg, .ogv, .pdf, .png, .pps, .pptx, .qt, .svg, .swf, .tar, .text, .tif, .txt, .wav, .webm, .wmv, .xls, .xlsx, .xml, .xsl, .xsd и .zip.
```



Встроенный веб-сервер предназначен для разработки и отладки. Не используйте его в эксплуатации. Для рабочих задач настоятельно рекомендуется применять полноценный веб-сервер. Оптимальными вариантами считаются NGINX или Apache вместе с PHP-FPM.

Кроме того, вы можете передать веб-серверу определенный скрипт в качестве скрипта-маршрутизатора, в результате чего PHP будет направлять все запросы к этому скрипту. Преимущество такого подхода в том, что он имитирует популярные PHP-фреймворки, использующие маршрутизаторы. Недостаток — ручное управление маршрутизацией для статических ресурсов.

В среде Apache или NGINX запросы браузера к изображениям, документам или другому статическому контенту обслуживаются напрямую без обращения к PHP. При использовании веб-сервера CLI вам сначала нужно проверить наличие этих ресурсов и вернуть явное `false`, чтобы сервер обработал их должным образом.

Затем скрипт маршрутизатора фреймворка должен проверить, работает ли вы в режиме CLI, и, если да, направить содержимое соответствующим образом. Например:

```
if (php_sapi_name() === 'cli-server') {
    if (preg_match('/\.(?:png|jpg|jpeg|gif)$/', $_SERVER["REQUEST_URI"])) {
        return false;
    }
}

// Продолжение выполнения маршрутизатора
```

Предыдущий файл `router.php` допускается использовать для загрузки локального веб-сервера следующим образом:

```
$ php -S localhost:8000 router.php
```

Веб-сервер разработки можно сделать доступным для любого интерфейса (из локальной сети), передав при вызове `0.0.0.0` вместо `localhost`. Однако помните, что подобный сервер не предназначен для использования в эксплуатации и не структурирован для защиты вашего приложения от злоумышленников. Не используйте этот веб-сервер в публичной сети!

## Читайте также

Документация по встроенному в PHP веб-серверу (<https://oreil.ly/Hm9U7>).

# 13.8. Использование модульных тестов для обнаружения регрессий в проекте, управляемом системой контроля версий с помощью `git-bisect`

## Задача

Вы хотите быстро определить, фиксация какого коммита привела к появлению определенной ошибки.

## Решение

Воспользуйтесь `git bisect`, чтобы найти первый неудачный коммит в вашем дереве исходных данных, как показано ниже.

1. Создайте новую ветку в проекте.
2. Напишите негативный модульный тест (тест, который воспроизводит ошибку).
3. Зафиксируйте этот тест в новой ветке.
4. Используйте `git rebase`, чтобы переместить коммит, вводящий ваш новый тест, в более раннюю точку истории проекта.
5. Запустите `git bisect` из данной точки для автоматического запуска ваших модульных тестов на последующих коммитах, чтобы найти тот, на котором тест не сработал.

Когда вы перебазируете историю коммитов вашего проекта, их хеши изменятся. Отслеживайте новый хеш коммита для вашего теста, чтобы правильно настроить `git bisect`. Предположим, что после перемещения коммит имеет хеш `48cc8f0`. В таком случае (см. пример 13.9) вы определили бы данный коммит как «хороший», а `HEAD` (последний коммит) в проекте — как «плохой».

**Пример 13.9.** Пример навигации git bisect после пересохранения тестового примера

```
$ git bisect start  
$ git bisect good 48cc8f0 ❶  
$ git bisect bad HEAD ❷  
$ git bisect run vendor/bin/phpunit ❸
```

❶ Для Git требуется указать первый хороший коммит, на который он должен обратить внимание.

❷ Поскольку вы не знаете точно, где находится битый коммит, передайте константу HEAD, и Git проанализирует все коммиты после ранее указанного хорошего.

❸ Git может выполнять определенную команду для каждого подозрительного коммита. В этом случае запустите набор тестов. Git продолжит просматривать историю коммитов вашего проекта, пока не найдет тот, в котором набор тестов не сработал.

После того как Git определит проблемный коммит (например, 16c43d7), воспользуйтесь git diff, как показано в примере 13.10, чтобы посмотреть, что на самом деле изменилось в этом коммите.

**Пример 13.10.** Сравнение заведомо плохого Git-коммита

```
$ git diff 16c43d7 HEAD
```

Как только вы узнаете, что именно сломалось, запустите git bisect reset, чтобы восстановить нормальную работу репозитория. В этот момент вернитесь в основную ветку (и, возможно, удалите тестовую ветку), чтобы начать исправление обнаруженной ошибки.

## Обсуждение

Инструмент Git bisect эффективен в обнаружении «плохих» коммитов в вашем проекте. Он особенно полезен в крупных, активных проектах, где между заведомо хорошим и явно плохим состоянием может быть несколько коммитов. В больших проектах часто нецелесообразно тратить время разработчиков на итерацию каждого коммита для проверки его валидности.

Команда git bisect работает по принципу бинарного поиска. Сначала она находит коммит, который расположен в средней точке между заведомо исправным и явно ошибочным состоянием, и тестирует его. Затем она перемещается сторону одного из коммитов, основываясь на результатах этого теста.

По умолчанию git bisect ожидает, что вы вручную проверите каждый подозрительный коммит, пока не обнаружите проблему. Однако подкоманда git bisect run позволяет делегировать эту проверку автоматизированной системе вроде

PHPUnit. Если команда `test` возвращает стандартный статус 0 (то есть успех), коммит считается корректным. PHPUnit завершается с кодом ошибки 0, когда все тесты пройдены.

Если тесты не срабатывают, PHPUnit возвращает код ошибки 1, который `git bisect` интерпретирует как плохой коммит. Таким образом, вы можете быстро и легко автоматизировать обнаружение битых коммитов среди тысяч коммитов.

В примере из «Решения» вы сначала создали новую ветку. Это делается для сохранения проекта чистым, чтобы вы могли отбросить любые потенциальные тестовые коммиты, как только определите проблемный коммит. В этой ветке вы зафиксировали единственный тест для воспроизведения ошибки, обнаруженной в вашем проекте. Используя журнал `git log`, вы можете быстро просмотреть историю вашего проекта, включая этот тестовый коммит, как показано на рис. 13.3.

```
ericmann@pop-os:~/Projects/git-bisect-demo$ git log --oneline
d442759 (HEAD -> testing) Test addition with negatives
9bd24f4 (origin/main, origin/HEAD, main) Division
48bcaa3 Misc cleanup
2b4fb8e multiplication
3bd869a Test method visibility
b51f515 Add subtraction
916161c Initial project commit
8550717 Initial commit
ericmann@pop-os:~/Projects/git-bisect-demo$
```

Рис. 13.3. Журнал Git, демонстрирующий основную и тестовую ветки с одним коммитом

Этот журнал полезен тем, что в нем вы найдете краткий хеш как вашего тестового коммита, так и всех остальных. Если вам известен заранее хороший коммит, вы можете переделать проект так, чтобы тестовый коммит шел сразу после этого корректного коммита.

На рис. 13.3 хеш тестового коммита — `d442759`, а последнего известного «хорошего» коммита — `916161c`. Чтобы упорядочить проект, запустите `git rebase` в интерактивном режиме с начального коммита проекта (`8550717`) — так вы сможете перебазировать тестовый коммит в более раннюю часть проекта (см. пример 13.11).

### Пример 13.11. Использование git rebase для упорядочивания коммитов

```
$ git rebase -i 8550717
```

Git откроет текстовый редактор и представит одинаковые SHA-хеши для каждого возможного коммита. Чтобы сохранить историю коммитов, оставьте ключевые слова `pick` и расположите тестовый коммит сразу после заранее исправленного, как показано на рис. 13.4.

```

pick 916161c Initial project commit
pick d442759 Test addition with negatives
pick b51f515 Add subtraction
pick 3bd869a Test method visibility
pick 2b4fb8e Multiplication
pick 48bc当地3 Misc cleanup
pick 9bd24f4 Division

# Rebase 8550717..d442759 onto 8550717 (7 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                               commit's log message, unless -C is used, in which case
#                               keep only this commit's message; -c is same as -C but
#                               opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit>] [-c <commit>] <label> [<oneline>]
#   .      create a merge commit using the original merge commit's
#   .      message (or the oneline, if no original merge commit was
#   .      specified); use -c <commit> to reword the commit message
#
:wq

```

**Рис. 13.4.** Git позволяет изменять или переупорядочивать коммиты

Сохраните файл, и Git восстановит историю проекта на основе перемещенного коммита. При возникновении проблем сначала устраните их локально и зафиксируйте результаты. Затем с помощью `git rebase --continue` двигайтесь дальше. Когда вы закончите, ваш проект будет реструктурирован таким образом, что новый тестовый пример появится сразу после известного хорошего коммита.



Известный корректный коммит будет иметь тот же хеш, что и все коммиты, которые предшествовали ему. Однако к вашему перемещенному коммиту и всем последующим будут применены новые хеши. Убедитесь, что вы используете правильные хеши коммитов во всех последующих командах Git!

Как только перебазирование завершится, выполните команду `git log --oneline`, чтобы снова увидеть историю коммитов и сослаться на новый коммит, связанный с вашим модульным тестом. Затем вы можете запустить `git bisect` от этого коммита до HEAD вашего проекта, как вы делали в примере 13.9. Git будет запускать PHPUnit на каждом подозрительном коммите, пока не найдет первый битый коммит, выдавая результат, подобный тому, что показан на рис. 13.5.

Вооружившись информацией о первом неудачном коммите, вы сможете просмотреть различия на этом этапе и понять, где и как именно ошибка закралась в ваш проект. Затем вернитесь в основную ветку и начните готовить исправления.

```
There was 1 failure:

1) FunctionTest::testAddAllNegatives
Failed asserting that 8 is identical to 2.

/home/ericmann/Projects/git-bisect-demo/tests/FunctionTest.php:63

FAILURES!
Tests: 17, Assertions: 18, Failures: 1.
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[167ca3b17b6e4362d83e32d7fe7c84effb963b08] multiplication
running 'vendor/bin/phpunit'
PHPUnit 9.5.28 by Sebastian Bergmann and contributors.

..... 17 / 17 (100%)

Time: 00:00.005, Memory: 6.00 MB

[17 tests, 18 assertions]
16c43d7cbc165fcda635b9d8b6d05d0c31175221 is the first bad commit
commit 16c43d7cbc165fcda635b9d8b6d05d0c31175221
Author: Eric Mann <eric@eemann.com>
Date:   Sat Jan 14 14:14:51 2023 -0800

Misc cleanup

src/functions.php      | 6 +++++-
tests/FunctionTest.php | 2 ++
2 files changed, 6 insertions(+), 2 deletions(-)
bisect found first bad committericmann@pop-os:~/Projects/git-bisect-demo$
```

Рис. 13.5. Работа git bisect

Неплохо бы добавить сюда и ваш новый юнит-тест.



Хотя вы могли бы снова воспользоваться git rebase, чтобы переместить тестовый коммит обратно на его первоначальное место, операция rebase все равно оставит историю проекта измененной по сравнению с ее прежним состоянием. Вместо этого вернитесь в корень дерева и создайте новую ветку для фактического исправления ошибки на этом этапе. Включите туда ваш тестовый коммит (возможно, с помощью git cherry-pick (<https://oreil.ly/tTFx3>)) и внесите все необходимые изменения.

## Читайте также

Документация по git bisect (<https://oreil.ly/LXgBP>).

## ГЛАВА 14

---

# Настройка производительности

Динамически интерпретируемые языки, такие как PHP, славятся своей гибкостью и простотой использования, но не скоростью. Отчасти это объясняется тем, как устроена их система типов. Когда типы определяются во время выполнения программы, компилятор не может точно знать, как выполнить ту или иную операцию, пока ему не станут доступны данные.

Рассмотрим следующую функцию PHP с нестрогой типизацией для сложения двух элементов:

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

Поскольку функция не объявляет типы передаваемых переменных или тип возвращаемого значения, она может иметь несколько сигнатур. Все сигнатуры методов в примере 14.1 являются одинаково корректными способами вызова предыдущей функции.

**Пример 14.1.** Различные сигнатурды для одного и того же определения функции

```
add(int $a,      int $b):    int ①
add(float $a,   float $b):  float ②
add(int $a,      float $b):  float ③
add(float $a,   int $b):    float ④
add(string $a,  int $b):    int ⑤
add(string $a,  int $b):    float ⑥
```

- ① `add(1, 2)` возвращает `int(3)`
- ② `add(1., 2.)` возвращает `float(3)`
- ③ `add(1, 2.)` возвращает `float(3)`
- ④ `add(1., 2)` возвращает `float(3)`
- ⑤ `add("1", 2)` возвращает `int(3)`
- ⑥ `add("1.", 2)` возвращает `float(3)`

В предыдущем примере показано, как одну и ту же функцию можно вызвать разными способами. PHP не знает, какая версия функции вам нужна, пока не увидит данные, которые вы ему предоставите, и при необходимости приведет некоторые значения к другим типам. Однако во время выполнения фактическая функция компилируется в опкод (операционный код), который выполняется процессором через специальную виртуальную машину. PHP создает несколько версий опкода одной и той же функции для обработки различных типов входных и возвращаемых данных.



Система нестрогой типизации, реализованная в PHP, делает его простым для изучения. Однако она также является причиной серьезных программных ошибок. В книге мы постарались уделить особое внимание строгой типизации везде, где это возможно, чтобы избежать подводных камней. Ознакомьтесь с рецептом 3.4, чтобы получить дополнительную информацию об использовании строгой типизации в собственном коде.

В компилируемых языках проблема нестрогой типизации была бы тривиальной — просто скомпилируйте программу до нескольких развилок опкода и двигайтесь дальше. К сожалению, PHP — это скорее интерпретируемый язык: он перезагружает и перекомпилирует ваш скрипт по требованию в зависимости от способа загрузки вашего приложения. К счастью, падение производительности из-за множества путей кода компенсируется с помощью двух современных функций, встроенных в сам язык: ЛТ-компиляции (just-in-time) и кэширования опкода.

## ЛТ-компиляция

Начиная с версии 8.0, PHP поставляется с ЛТ-компилятором (<https://oreil.ly/XS9gg>), который позволяет ускорить выполнение программы и повысить производительность приложений. Для этого он отслеживает фактические инструкции, передаваемые виртуальной машине (VM), выполняющей скрипт. Если конкретная последовательность команд выполняется часто, PHP автоматически распознает важность операции и определяет, стоит ли компилировать код.

Последующие вызовы того же кода будут задействовать скомпилированный байт-код, а не динамический скрипт, что приведет к значительному увеличению производительности. Согласно статистике, опубликованной компанией Zend после выхода PHP 8.0 (<https://oreil.ly/LpZ3w>), включение ЛТ-компилятора увеличивает скорость работы эталонного пакета PHP в три раза!

Следует помнить, что ЛТ-компиляция в первую очередь полезна для низкоуровневых алгоритмов. К ним относятся обработка чисел и манипулирование данными. Если вы не выполняете операций, требующих большой вычислительной мощности (работа с графикой или интеграция с тяжелыми базами данных), эти

изменения не дадут вам очевидного преимущества. Однако, зная о существовании JIT-компилятора, вы можете воспользоваться его возможностями и начать взаимодействовать с PHP по-новому.

## Кэширование опкодов

Один из самых простых способов увеличить производительность — собственно, именно так и действует JIT-компилятор — это кэшировать ресурсозатратные операции и ссылаться на результат, а не выполнять их снова и снова. Начиная с версии 5.5, PHP поставляется с дополнительным расширением для кэширования прекомпилированного байт-кода в память под названием OPcache (<https://oreil.ly/wH2ue>)<sup>1</sup>.

Помните, что PHP — это прежде всего динамический интерпретатор скриптов, и при запуске программы он считывает ваши скрипты. Если вы постоянно перезапускаете приложение, PHP придется перекомпилировать ваш скрипт в читаемый компьютером байт-код, чтобы он выполнялся правильно. Такие многократные операции могут привести к ощутимому снижению производительности. Однако OPcache позволяет выборочно компилировать скрипты, предоставляя байт-код PHP перед запуском остальной части приложения, что избавляет PHP от необходимости каждый раз загружать и обрабатывать скрипты!



ЛТ-компилятор, начиная с PHP 8, активируется только в том случае, если на сервере также включен OPcache, поскольку он использует кеш в качестве общей памяти. Однако вам не нужно применять ЛТ-компилятор, чтобы за действовать сам OPcache.

И ЛТ-компиляция, и кэширование опкодов — это низкоуровневые улучшения производительности языка, которые можно легко использовать во время работы. Также важно понимать, как определять время выполнения пользовательских функций. Это позволяет относительно быстро выявлять узкие места в бизнес-логике. Комплексное тестирование приложения также помогает оценить изменения производительности при развертывании в других средах, на новых версиях языка или с обновленными зависимостями в будущем.

Следующие рецепты описывают, как измерять время выполнения и производительность кода приложения на уровне пользователя, а также как использовать кеш опкодов на уровне языка для оптимизации производительности вашего приложения и среды.

---

<sup>1</sup> Новый ЛТ-компилятор, выпущенный с PHP 8.0, использует OPcache, однако вы все равно можете прибегнуть ручному кэшированию для управления системой, даже если ЛТ-компиляция недоступна.

## 14.1. Измерение времени выполнения функций

### Задача

Вы хотите понять, сколько времени занимает выполнение той или иной функции, чтобы выявить потенциальные возможности для оптимизации.

### Решение

Используйте встроенную в PHP функцию `hftime()` как перед ее запуском, так и после завершения, чтобы определить, сколько времени заняло выполнение. Например:

```
$start = hftime(true);

doSomethingComputationallyExpensive();

$totalTime = (hftime(true) - $start) / 1e+9;

echo "Функция выполнялась {$totalTime} секунд." . PHP_EOL;
```

### Обсуждение

Функция `hftime()` возвращает системное время высокого разрешения, отсчитываемое от заданной произвольной точки времени. По умолчанию она возвращает массив из двух целых чисел — секунд и наносекунд соответственно. Если передать в функцию `true`, она вернет общее количество наносекунд. Чтобы преобразовать результат в секунды, нужно поделить его на `1e+9`.

Более продвинутым подходом является абстрагирование механизма синхронизации в объекте-декораторе. Как упоминалось в главе 8, декоратор — это шаблон программирования, который позволяет расширить функциональность отдельного вызова функции (или целого класса), обернув его в реализацию другого класса. В нашем случае для определения времени выполнения функции задействована функция `hftime()`, которая не меняет саму функцию (см. пример 14.2).

**Пример 14.2.** Объект декоратора с таймером для измерения производительности вызова функции

```
class TimerDecorator
{
    private int $calls = 0;
    private float $totalRuntime = 0..;
```

```

public function __construct(public $callback, private bool $verbose = false) {}
public function __invoke(...$args): mixed ❶
{
    if (! is_callable($this->callback)) {
        throw new ValueError('Класс не оборачивает вызываемую функцию!');
    }

    $this->calls += 1;
    $start = hrtime(true); ❷

    $value = call_user_func($this->callback, ...$args); ❸

    $totalTime = (hrtime(true) - $start) / 1e+9;
    $this->totalRuntime += $totalTime;

    if ($this->verbose) {
        echo "Функция выполнялась {$totalTime} секунд." . PHP_EOL; ❹
    }
}

return $value; ❺
}

public function getMetrics(): array ❻
{
    return [
        'calls'    => $this->calls,
        'runtime'  => $this->totalRuntime,
        'avg'      => $this->totalRuntime / $this->calls
    ];
}
}

```

**❶** Магический метод `__invoke()` делает экземпляры классов вызываемыми, как если бы они были функциями. Использование оператора `spread (...)` позволит захватить любые аргументы, переданные во время выполнения, чтобы в дальнейшем передать их обернутому методу.

**❷** Фактический механизм синхронизации, используемый декоратором, такой же, как и в примере из «Решения».

**❸** Если предположить, что обернутая функция является вызываемой, PHP вызовет ее и передаст все необходимые аргументы благодаря оператору `spread (...).`

**❹** Этую реализацию декоратора разрешается инстанцировать с флагом детализации, который также будет выводить время выполнения в консоль.

**❺** Так как обернутая функция способна возвращать данные, стоит убедиться, что декодер возвращает и этот вывод.

❶ Поскольку декорируемая функция сама является объектом, можно напрямую открывать дополнительные свойства и методы. В этом случае декоратор отслеживает агрегированные метрики, которые доступны для прямого извлечения.

Если предположить, что функция `doSomethingComputationallyExpensive()` из примера в «Решении» — та самая функция, которую вы хотите протестировать, то предыдущий декоратор позволяет обернуть функцию и выдать метрики, как показано в примере 14.3.

**Пример 14.3.** Использование декоратора для определения времени выполнения функции

```
$decorated = new TimerDecorator('doSomethingComputationallyExpensive');

$decorated(); ❶

var_dump($decorated->getMetrics()); ❷
```

❶ Поскольку класс-декоратор реализует магический метод `__invoke()`, вы можете использовать экземпляр класса так, как если бы он сам был функцией.

❷ Полученный массив метрик будет содержать количество вызовов, общее и среднее время выполнения (в секундах) для всех вызовов.

Аналогично допускается протестировать одну и ту же обернутую функцию несколько раз и получить суммарные метрики времени выполнения из всех вызовов следующим образом:

```
$decorated = new TimerDecorator('doSomethingComputationallyExpensive');

for ($i = 0; $i < 10; $i++) {
    $decorated();
}

var_dump($decorated->getMetrics());
```

Поскольку класс `TimerDecorator` способен обернуть любую вызываемую функцию, вы получаете возможность использовать его для декорирования методов класса так же легко, как и для встроенных функций. Класс в примере 14.4 определяет как статический метод, так и метод экземпляра, любой из которых может быть обернут декоратором.

**Пример 14.4.** Простое определение класса для тестирования декораторов

```
class DecoratorFriendly
{
    public static function doSomething()
```

```
{  
    // ...  
}  
  
public function doSomethingElse()  
{  
    // ...  
}  
}
```

Пример 14.5 показывает, как методы класса (как статические, так и связанные с экземпляром) могут быть названы вызываемыми во время выполнения в PHP. Все, что может быть выражено в виде вызываемого интерфейса, можно обернуть декоратором.

**Пример 14.5.** Любой вызываемый интерфейс разрешается оборачивать декоратором

```
$decoratedStatic = new TimerDecorator(['DecoratorFriendly', 'doSomething']); ❶  
$decoratedStatic(); ❷  
  
var_dump($decoratedStatic->getMetrics());  
  
$instance = new DecoratorFriendly();  
  
$decoratedMember = new TimerDecorator([$instance, 'doSomethingElse']); ❸  
$decoratedMember(); ❹  
  
var_dump($decoratedMember->getMetrics());
```

❶ Статический метод класса используется в качестве вызываемого, передавая массив из имен класса и его статического метода.

❷ Декорированный статический метод после создания допускается вызывать так же, как и любую другую функцию, и он будет выдавать метрики тем же способом.

❸ Метод экземпляра класса служит в качестве вызываемого, передавая массив из инстанцированного объекта и строкового имени метода.

❹ Декорированный метод экземпляра можно вызвать, как и любую другую функцию, для заполнения метрик внутри декоратора.

Знание времени выполнения функции, позволяет вам сосредоточиться на оптимизации ее работы. Например, вы можете сделать рефакторинг логики или применить альтернативный алгоритм.

Для использования `hftime()` в PHP изначально требовалось расширение HRTIME ([https://oreil.ly/P\\_4Fq](https://oreil.ly/P_4Fq)), однако теперь оно по умолчанию является неотъемлемой частью функции. Если вы используете версию PHP старше 7.3 или дистрибутив,

в котором расширение было явно опущено, то и сама функция может отсутствовать. В этом случае либо установите расширение самостоятельно через PECL, либо воспользуйтесь аналогичной функцией `microtime()`<sup>1</sup>.

Вместо того чтобы считать секунды от произвольной временной точки, функция `microtime()` возвращает количество микросекунд, прошедших с начала эпохи Unix. Этую функцию можно использовать вместо `hftime()` следующим образом:

```
$start = microtime(true);  
  
doSomethingComputationallyExpensive();  
  
$totalTime = microtime(true) - $start;  
  
echo "Функция выполнялась {$totalTime} секунд." . PHP_EOL;
```

Независимо от того, применяете ли вы `hftime()`, как в примере из «Решения», или `microtime()`, как в предыдущем фрагменте, убедитесь, что вы последовательно считываете полученные данные. Оба механизма возвращают представления о времени с разной степенью точности, что может привести к путанице, если вы захотите смешать их при форматировании выходных данных.

## Читайте также

Документация PHP по `hftime()` (<https://oreil.ly/AjZ4H>) и `microtime()` ([https://oreil.ly/r\\_U84](https://oreil.ly/r_U84)).

# 14.2. Оценка производительности приложения

## Задача

Вы хотите контролировать производительность вашего приложения, чтобы оценивать изменения (например, снижение производительности) по мере развития кодовой базы, зависимостей и версий используемого языка.

## Решение

Прибегните к помощи такого автоматизированного инструмента, как PHPBench, для анализа вашего кода и регулярного тестирования производительности.

---

<sup>1</sup> Подробнее о PECL и управлении расширениями см. в рецепте 15.4.

Например, следующий класс создан для тестирования производительности всех доступных алгоритмов хеширования для строк различного размера<sup>1</sup>.

```
/**  
 * @BeforeMethods("setUp")  
 */  
class HashingBench  
{  
    private $string = '';  
  
    public function setUp(array $params): void  
    {  
        $this->string = str_repeat('X', $params['size']);  
    }  
  
    /**  
     * @ParamProviders({  
     *      "provideAlgos",  
     *      "provideStringSize"  
     * })  
     */  
    public function benchAlgos($params): void  
    {  
        hash($params['algo'], $this->string);  
    }  
  
    public function provideAlgos()  
    {  
        foreach (array_slice(hash_algos(), 0, 20) as $algo) {  
            yield ['algo' => $algo];  
        }  
    }  
  
    public function provideStringSize() {  
        yield ['size' => 10];  
        yield ['size' => 100];  
        yield ['size' => 1000];  
    }  
}
```

Чтобы запустить данный код сначала клонируйте PHPBench, затем установите зависимости Composer и, наконец, выполните следующую команду:

```
$ ./bin/phpbench run --profile=examples --report=examples --filter=HashingBench
```

В итоге у вас должно получиться что-то вроде того, что изображено на рис. 14.1.

---

<sup>1</sup> Этот конкретный пример взят из стандартного пакета PHPBench (<https://oreil.ly/zZE4X>).

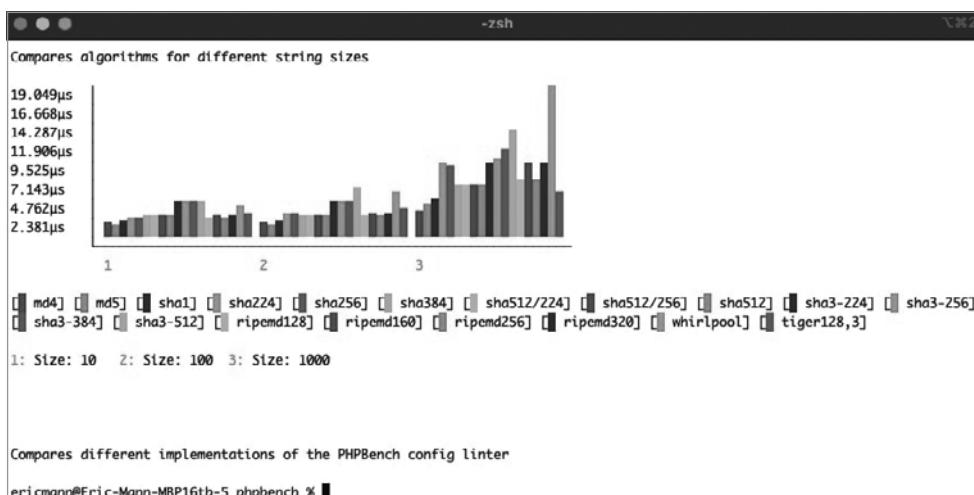


Рис. 14.1. Диаграмма выходных метрик PHPBench

## Обсуждение

PHPBench — это эффективный способ оценки производительности кода в различных сценариях. Он часто используется для измерения уровня производительности новых функций, а также включается непосредственно в среды непрерывной интеграции.

Собственная конфигурация PHPBench на GitHub Actions (<https://oreil.ly/D6PMf>) запускает полный набор тестов для самого приложения при каждом запросе и изменении. Это позволяет специалистам по сопровождению убедиться в том, что программа продолжает стабильно работать с каждым внесенным изменением.

Если проект предполагает применение автоматизированных тестов, в нем должен быть задействован Composer<sup>1</sup>. Вам нужно настроить автозагрузку Composer, чтобы PHPBench знал, где искать классы. После этого можно начинать сборку проекта.

Предположим, вы создаете проект, в котором используются объекты-значения и хеширование для защиты хранящихся в них конфиденциальных данных. Ваш исходный файл composer.json может выглядеть примерно так:

```
{
  "name": "phpcookbook/valueobjects",
  "require-dev": {
    "phpbench/phpbench": "^1.0"
  },
}
```

<sup>1</sup> Подробнее об инициализации проекта в Composer читайте в Рецепте 15.1.

```
"autoload": {
    "psr-4": {
        "Cookbook\\": "src/"
    }
},
"autoload-dev": {
    "psr-4": {
        "Cookbook\\Tests\\": "tests/"
    }
},
"minimum-stability": "dev",
"prefer-stable": true
}
```

Естественно, код вашего проекта будет находиться в папке `src/`, а любые тесты, в том числе и тесты производительности, — в отдельном каталоге `tests/`. Однако для последних вы захотите создать специальный каталог `tests/Benchmark/`, чтобы отслеживать пространства имен и фильтруемый код.

Первый класс, который стоит проверить, — это объект-значение, принимающий адрес электронной почты и которым легко манипулировать, как строкой. Но когда он передает свое содержимое в контекст отладки, скажем, в `var_dump()` или `print_r()`, он автоматически хеширует значение.



Электронная почта — достаточно распространенный формат, поэтому даже хеширование данных не сможет защитить их от профессионального взломщика. Примеры в этом рецепте призваны продемонстрировать, как можно замаскировать данные с помощью хеша. Не следует воспринимать это полноценным руководством по безопасности.

Создайте класс, определенный в примере 14.6, в новом каталоге `src/` и назовите его `ProtectedString.php`. Данный класс объединяет в себе прежде всего несколько реализованных магических методов, исключающих возможность случайной сериализации объекта и получения его внутреннего значения. Получить доступ к содержимому объекта `ProtectedString` можно только через метод `::getValue()`. Любой другой метод вернет содержимое в виде хеша SHA-256.

#### Пример 14.6. Определение класса-обертки защищенной строки

```
namespace Cookbook;

class ProtectedString implements \JsonSerializable
{
    protected bool $valid = true;

    public function __construct(protected ?string $value) {}
```

```
public function getValue(): ?string
{
    return $this->value;
}

public function equals(ProtectedString $other): bool
{
    return $this->value === $other->getValue();
}

protected function redacted(): string
{
    return hash('sha256', $this->value, false);
}

public function isValid(): bool
{
    return $this->valid;
}

public function __serialize(): array
{
    return [
        'value' => $this->redacted()
    ];
}

public function __unserialize(array $serialized): void
{
    $this->value = null;
    $this->valid = false;
}

public function jsonSerialize(): mixed
{
    return $this->redacted();
}

public function __toString()
{
    return $this->redacted();
}

public function __debugInfo()
{
    return [
        'valid' => $this->valid,
        'value' => $this->redacted()
    ];
}
```

Вы хотите оценить производительность выбранного алгоритма хеширования. SHA-256 — хороший вариант, но нужно проверить все возможные средства сериализации данных на производительность, чтобы гарантировать, что переход на другой алгоритм хеширования в будущем не повлияет на быстродействие системы.

Чтобы приступить к тестированию этого класса, создайте в корне вашего проекта файл `phpbench.json`:

```
{  
    "$schema": "./vendor/phpbench/phpbench/phpbench.schema.json",  
    "runner.bootstrap": "vendor/autoload.php"  
}
```

Наконец, протестируйте различные способы сериализации строки. Тест, определенный в примере 14.7, должен находиться в каталоге `tests/Benchmark/ProtectedStringBench.php`.

**Пример 14.7.** Тестирование производительности класса `ProtectedString`

```
namespace Cookbook\Tests\Benchmark;  
  
use Cookbook\ProtectedString;  
  
class ProtectedStringBench  
{  
    public function benchSerialize()  
    {  
        $data = new ProtectedString('testValue');  
        $serialized = serialize($data);  
    }  
  
    public function benchJsonSerialize()  
    {  
        $data = new ProtectedString('testValue');  
        $serialized = json_encode($data);  
    }  
  
    public function benchStringTypecast()  
    {  
        $data = new ProtectedString('testValue');  
        $serialized = '' . $data;  
    }  
  
    public function benchVarExport()  
    {  
        $data = new ProtectedString('testValue');
```

```

    ob_start();
    var_dump($data);
    $serialized = ob_end_clean();
}
}

```

Наконец, вы можете запустить свои тесты производительности с помощью следующей команды оболочки:

```
$ ./vendor/bin/phpbench run tests/Benchmark --report=default
```

Эта команда выдаст результат, подобный тому, что изображен на рис. 14.2, с подробным описанием использования памяти и времени выполнения для каждой из операций сериализации.

```

ericmann@Eric-Mann-MBP16tb-5 vobj % ./vendor/bin/phpbench run tests/Benchmark --report=default
PHPBench (1.2.7) running benchmarks... #standwithukraine
with configuration file: /Users/ericmann/Projects/tester/vobj/phpbench.json
with PHP version 8.2.0, xdebug ✘, opcache ✘

\Cookbook\Tests\Benchmark\ProtectedStringBench

benchSerialize.....IO - Mo319.000µs (±0.00%)
benchJsonSerialize.....IO - Mo349.000µs (±0.00%)
benchStringTypecast.....IO - Mo298.000µs (±0.00%)
benchVarExport.....IO - Mo309.000µs (±0.00%)

Subjects: 4, Assertions: 0, Failures: 0, Errors: 0
+-----+-----+-----+-----+-----+-----+-----+-----+
| iter | benchmark | subject | set | revs | mem_peak | time_avg | comp_z_value | comp_deviation |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0   | ProtectedStringBench | benchSerialize | 1 | 1 | 684,296b | 319.000µs | +0.00% | +0.00% |
| 0   | ProtectedStringBench | benchJsonSerialize | 1 | 1 | 684,312b | 349.000µs | +0.00% | +0.00% |
| 0   | ProtectedStringBench | benchStringTypecast | 1 | 1 | 684,312b | 298.000µs | +0.00% | +0.00% |
| 0   | ProtectedStringBench | benchVarExport | 1 | 1 | 684,296b | 309.000µs | +0.00% | +0.00% |
+-----+-----+-----+-----+-----+-----+-----+-----+
ericmann@Eric-Mann-MBP16tb-5 vobj %

```

**Рис. 14.2.** Вывод PHPBench для сериализации объекта-значения с хешированием

В каждый компонент вашей программы можно и нужно интегрировать тесты производительности. Это значительно упростит оценку производительности приложения в различных условиях — на новом серверном оборудовании или с новой версией PHP. Постарайтесь найти время и подключить эти тесты к системе непрерывной интеграции, чтобы они выполнялись и записывались как можно чаще.

## Читайте также

Официальная документация по проекту PHPBench (<https://oreil.ly/4HCMc>).

## 14.3. Ускорение работы приложения с помощью кэша опкодов

### Задача

Вы хотите использовать кэширование опкодов в своей среде, чтобы улучшить общую производительность приложения.

### Решение

Установите расширение OPcache и настройте его в `php.ini` для вашего окружения<sup>1</sup>. Поскольку это стандартное расширение, достаточно лишь обновить конфигурацию, чтобы включить кэширование. Следующие настройки обычно рекомендуются для обеспечения надежной работы, но их стоит протестировать для конкретного приложения и инфраструктуры:

```
opcache.memory_consumption=128
opcache.interned_strings_buffer=8
opcache.max_accelerated_files=4000
opcache.revalidate_freq=60
opcache.fast_shutdown=1
opcache.enable=1
opcache.enable_cli=1
```

### Обсуждение

Когда PHP запущен, интерпретатор считывает ваши скрипты и компилирует PHP-код. К сожалению, поскольку PHP формально не является компилируемым языком, ему приходится выполнять эту операцию при каждой загрузке скрипта. Для простого приложения это не является большой проблемой. Однако в более крупных проектах это может замедлить загрузку и вызвать задержки при повторных запросах.

Самый простой способ обойти эту проблему — кэшировать скомпилированный байт-код, чтобы повторно использовать его при последующих запросах.

Для локального тестирования кэширования опкодов можно задействовать флаг `-d` в командной строке при запуске скрипта. Флаг `-d` позволяет явно переопределить значение конфигурации, которое иначе устанавливается (или остается по умолчанию).

---

<sup>1</sup> OPcache — это стандартное расширение PHP, которое может быть отключено при компиляции с флагом `--disable-all`. В этом случае вам придется перекомпилировать PHP с флагом `--enable-opcache` или установить свежую версию PHP, скомпилиированную с данным флагом.

нию) в файле `php.ini`. В частности, флаги командной строки в примере 14.8 позволяют запустить локальный сервер разработки с полностью отключенным OPcache.

**Пример 14.8.** Запуск локального веб-сервера PHP без поддержки OPcache

```
$ php -S localhost:8080 -t public/ -dopcache.enable_cli=0 -dopcache.enable=0
```

Аналогично, вы можете выполнить почти ту же команду с явным включением кэша опкодов (см. пример 14.9), чтобы напрямую сравнить поведение и производительность вашего приложения.

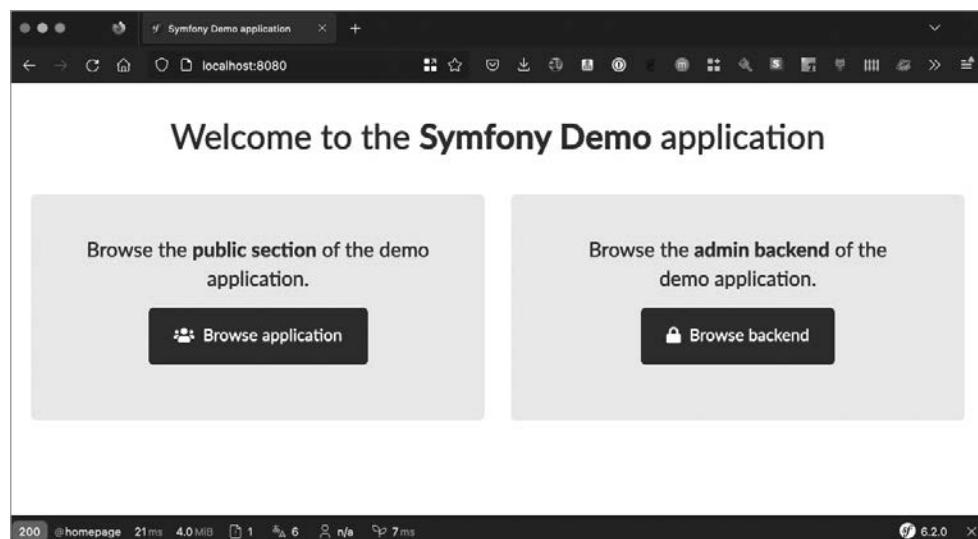
**Пример 14.9.** Запуск локального веб-сервера PHP с поддержкой OPcache

```
$ php -S localhost:8080 -t public/ -dopcache.enable_cli=1 -dopcache.enable=1
```

Чтобы увидеть, как это работает, потратьте время на установку демоприложения Symfony. Следующие две команды клонируют демонстрационное приложение в локальный каталог `/demosite/` и задействуют Composer для установки необходимых зависимостей:

```
$ composer create-project symfony/symfony-demo demosite  
$ cd demosite && composer install
```

Далее используйте встроенный веб-сервер PHP для запуска самого приложения. Выполните команду из примера 14.8 для запуска без поддержки opcache. Приложение станет доступно на порту 8080 и будет выглядеть примерно так, как показано на рис. 14.3.



**Рис. 14.3.** Загрузочная страница демоприложения Symfony

Приложение по умолчанию работает локально, используя легковесную базу данных SQLite, поэтому оно должно загружаться довольно быстро. Как показано в примере 14.10, вы можете проверить время загрузки с помощью команды cURL в терминале.

**Пример 14.10.** Простая команда cURL для измерения времени отклика веб-приложения

```
curl -s -w "\nLookup time:\t%{time_namelookup}\nConnect time:\t%{time_connect}\nPreXfer time:\t%{time_prettransfer}\nStartXfer time:\t%{time_starttransfer}\n\nTotal time:\t%{time_total}\n" -o /dev/null \
http://localhost:8080
```

Без включенного кэширования опкодов демоприложение Symfony загружается за ~0,3677 секунды. Это очень быстро, но, опять же, приложение выполняется исключительно в локальной среде. В эксплуатации с удаленной базой данных оно, скорее всего, будет медленнее, но это вполне приемлемый уровень.

Теперь остановите приложение и запустите его с включенным кэшированием опкодов с помощью команды, определенной в примере 14.9. Затем повторите тест производительности cURL из примера 14.10. С кэшированием опкодов приложение теперь загружается за ~0,0371 секунды.

Это стандартное простое приложение увеличивает производительность системы практически в десять раз. Чем быстрее загружается приложение, тем больше клиентов может обслужить ваша система за тот же период времени!

## Читайте также

Документация по PHP для расширения OPcache (<https://oreil.ly/Tb8rC>).

## ГЛАВА 15

---

# Пакеты и расширения

PHP — это язык высокого уровня, который использует динамическую типизацию и управление памятью, чтобы облегчить разработку программного обеспечения для конечных пользователей. К сожалению, компьютеры не очень хорошо справляются с высокоуровневыми концепциями, поэтому любая высокоуровневая система должна быть построена на блоках более низкого уровня. В случае с PHP вся система написана на языке С и построена на его основе.

Поскольку PHP — проект с открытым исходным кодом, вы можете загрузить весь код языка прямо с GitHub ([https://oreil.ly/Z1\\_IP](https://oreil.ly/Z1_IP)), осуществить сборку языка из исходников на своей системе, внести в него изменения или написать собственные расширения (на уровне С).

В Ubuntu Linux вам понадобятся следующие пакеты для сборки:

- `pkg-config` — пакет Linux, возвращающий информацию об установленных библиотеках;
- `build-essential` — метапакет, включающий в себя отладчик GNU, компилятор `g++` и другие инструменты для работы с проектами на C/C++;
- `autoconf` — пакет макросов для создания сценариев оболочки, конфигурирующих пакеты кода;
- `bison` — генератор синтаксических анализаторов общего назначения;
- `re2c` — компилятор регулярных выражений и лексер с открытым исходным кодом для C и C++;
- `libxml2-dev` — библиотека разработки на C, которая предоставляет интерфейсы для работы с XML документами;
- `libssqlite3-dev` — пакет разработки программного обеспечения, содержащий библиотеку SQLite3 и вспомогательные инструменты.

Вы можете установить их все с помощью команды apt:

```
$ sudo apt install -y pkg-config build-essential autoconf bison re2c \
    libxml2-dev libssqlite3-dev
```

Когда зависимости станут доступны, используйте сценарий `buildconf` для генерации скрипта конфигурации, а затем `configure` сам подготовит среду сборки. Несколько

опций (<https://oreil.ly/md2qt>) можно передать непосредственно в `configure`, чтобы управлять настройкой среды. В табл. 15.1 перечислены некоторые из наиболее полезных.

**Таблица 15.1.** Параметры конфигурации PHP

Опция	Описание
<code>--enable-debug</code>	Компиляция с отладочной информацией. Полезно для разработки изменений в ядре PHP или написания новых расширений
<code>--enable-libgcc</code>	Разрешает явно использовать <code>libgcc</code>
<code>--enable-php-streams</code>	Включает экспериментальную функциональность потоков PHP
<code>--enable-phpdbg</code>	Включает поддержку интерактивного отладчика <code>phpdbg</code>
<code>--enable-zts</code>	Включает безопасность потоков
<code>--disable-short-tags</code>	Отключает поддержку коротких тегов PHP (например, <code>&lt;?</code> )

Понимание того, как собрать PHP, не является обязательным условием для его использования. Большинство сред позволяют установить бинарный дистрибутив непосредственно из стандартного менеджера пакетов. Например, на Ubuntu вы можете установить PHP напрямую, выполнив следующие действия:

```
$ sudo apt install -y php
```

Однако не лишним будет знать, как собрать PHP из исходников, если в будущем вы захотите изменить поведение языка, включить не входящее в комплект расширение или написать собственный модуль.

## Стандартные модули

По умолчанию PHP использует собственную систему расширений для реализации большей части основных функций языка. Помимо основных модулей, с PHP напрямую связаны различные расширения<sup>1</sup>. К ним относятся:

- BCMath для математических операций с произвольной точностью (<https://oreil.ly/QwfUv>);
- FFI (Foreign Function Interface) для загрузки общих библиотек и вызова функций в них (<https://oreil.ly/sktWY>);
- PDO (PHP Data Objects) для определения простого и согласованного интерфейса для доступа к базам данных в PHP (<https://oreil.ly/BEsdu>);
- SQLite3 для прямого взаимодействия с SQLite (<https://oreil.ly/Zejtz>).

<sup>1</sup> Полный список встроенных и сторонних расширений можно найти в руководстве по PHP (<https://oreil.ly/SEWGK>).

Стандартные модули поставляются вместе с PHP и включаются в конфигурационном файле `php.ini`. Внешние расширения, такие как PDO для Microsoft SQL Server, также доступны, но их нужно устанавливать и активировать отдельно. Например, инструмент PECL, рассмотренный в рецепте 15.4, упрощает установку этих модулей.

## Библиотеки/Composer

Помимо встроенных расширений, у вас есть возможность использовать Composer (<https://getcomposer.org/>) — самый популярный менеджер зависимостей для PHP. Любой PHP-проект может (и, вероятно, должен) быть определен как модуль Composer через файл `composer.json`, который описывает проект и его структуру. Наличие такого файла дает два ключевых преимущества, даже если вы не собираетесь устанавливать Composer для внедрения стороннего кода в свой проект:

- вы (или другой разработчик) можете использовать свой проект в качестве зависимого от другого проекта, что повышает портируемость вашего кода и позволяет повторное применение определений функций и классов;
- если в проекте есть файл `composer.json`, вы получаете возможность задействовать функции автоматической загрузки Composer для динамического включения классов и функций в ваш проект без явного обращения к `require()` для их непосредственной загрузки.

Рецепты в этой главе содержат информацию о том, как настроить проект в качестве пакета Composer и как использовать Composer для поиска и подключения сторонних библиотек. Вы также узнаете, как с помощью PHP Extension Community Library (PECL) и PHP Extension and Application Repository (PEAR) добавить собственные расширения.

### 15.1. Определение проекта Composer

#### Задача

Вы хотите начать новый проект с Composer для динамической загрузки кода и зависимостей.

#### Решение

Используйте команду `init` в командной строке Composer, чтобы загрузить новый проект с файлом `composer.json`. Например:

```
$ composer init --name ericmann/cookbook --type project --license MIT
```

При выполнении команды выше Composer будет интерактивно запрашивать значения полей для заполнения (описание, автор пакета, стабильность пакета и т. д.). После этого у вас появится файл `composer.json`.

## Обсуждение

Composer работает, определяя информацию о вашем проекте в JSON-документе и используя ее для создания дополнительных скриптов-загрузчиков и интеграций. У только что инициализированного проекта мало деталей в этом документе. Содержимое файла `composer.json`, созданного командой `init` ранее, изначально будет выглядеть следующим образом:

```
{  
    "name": "ericmann/cookbook",  
    "type": "project",  
    "license": "MIT",  
    "require": {}  
}
```

Данный конфигурационный файл не определяет ни зависимости, ни дополнительные скрипты, ни автозагрузку. Чтобы он был полезен для чего-то, кроме определения проекта и лицензии, вам нужно его наполнять. В первую очередь необходимо определить автозагрузчик, который будет подтягивать код вашего проекта.

Используйте стандартное пространство имен `Cookbook` и поместите весь свой код в каталог `src/` внутри проекта. Затем обновите `composer.json`, чтобы сопоставить это пространство имен с данным каталогом следующим образом:

```
{  
    "name": "ericmann/cookbook",  
    "type": "project",  
    "license": "MIT",  
    "require": {},  
    "autoload": {  
        "psr-4": {  
            "Cookbook\\": "src/"  
        }  
    }  
}
```

После настройки Composer запустите в командной строке команду `composer dumpautoload`, чтобы заставить Composer перезагрузить конфигурацию и определить автоматическое сопоставление источников. Затем Composer создаст новый каталог `vendor/`, который содержит два важных компонента:

- скрипт `autoload.php`, который нужен для запуска `require()` при загрузке приложения;
- каталог `composer`, содержащий процедуры загрузки кода Composer для динамического добавления ваших скриптов.

Чтобы увидеть, как работает автозагрузка, создайте два новых файла. Сперва добавьте в каталог `src/` файл `Hello.php`, содержащий класс `Hello`, как в примере 15.1.

**Пример 15.1.** Простое определение класса для автозагрузки Composer

```
<?php
namespace Cookbook;

class Hello
{
    public function __construct(private string $greet) {}

    public function greet(): string
    {
        return "Hello, {$this->greet}!";
    }
}
```

Затем в корне проекта создайте файл `app.php` со следующим содержимым для запуска кода из предыдущего фрагмента:

```
<?php

require_once 'vendor/autoload.php';

$intro = new Cookbook\Hello('world');

echo $intro . PHP_EOL;
```

Наконец, вернитесь в командную строку. Поскольку вы добавили в проект новый класс, нужно еще раз выполнить команду `composer dumpautoload`, чтобы Composer определил класс. Теперь запустите `php app.php` для вызова приложения напрямую и получения следующего результата:

```
$ php app.php
Hello, world!
$
```

Любые определения классов, необходимые для вашего проекта или приложения, могут быть определены таким же образом. Базовое пространство имен `Cookbook` всегда будет корнем каталога `src/`. Если вы хотите определить вложенное пространство имен для объектов, например `Cookbook\Recipes`, то создайте папку с аналогичным именем (например, `Recipes/`) в каталоге `src/`, чтобы Composer знал, где найти определения классов.

Аналогично, вы можете использовать команду `require` в Composer для импорта сторонних зависимостей<sup>1</sup>. Они будут загружены в ваше приложение во время выполнения так же, как и ваши пользовательские классы.

---

<sup>1</sup> Подробнее об установке сторонних библиотек с помощью Composer см. в рецепте 15.3.

## Читайте также

Документация Composer по команде `init` (<https://oreil.ly/6J29w>) и по автозагрузке PSR-4 (<https://oreil.ly/Buns1>).

## 15.2. Поиск пакетов Composer

### Задача

Вы хотите найти готовую библиотеку для решения конкретной задачи, чтобы не тратить время на написание собственной реализации с нуля.

### Решение

Воспользуйтесь репозиторием пакетов PHP на Packagist (<https://packagist.org/>), чтобы найти подходящую библиотеку и с помощью Composer установить ее в ваше приложение.

### Обсуждение

Многие разработчики считают, что большую часть своего времени они тратят на перепроектирование логики или систем, созданных ими ранее. Несмотря на то что каждое приложение служит определенным целям, для их функционирования часто применяются однотипные базовые компоненты и принципы.

Это один из ключевых факторов, лежащих в основе таких парадигм, как объектно-ориентированное программирование. В его рамках вы инкапсулируете логическую структуру приложения в виде объектов, которыми можно независимо управлять, изменять их или даже повторно использовать. Вместо того чтобы многократно переписывать один и тот же код, вы определяете его как объект и повторно за-действуете в вашем приложении или вообще переносите в следующий проект<sup>1</sup>.

В PHP эти неоднократно применяемые компоненты кода часто оформляются в виде отдельных библиотек, доступных для импорта через Composer. Подобно тому как в рецепте 15.1 было показано, как создавать проект Composer и автоматически импортировать определения классов и функций, эту же схему можно использовать для добавления сторонней логики в вашу систему<sup>2</sup>.

---

<sup>1</sup> Более подробную информацию об объектно-ориентированном программировании и по-вторном использовании кода вы найдете в главе 8.

<sup>2</sup> Установка сторонних пакетов Composer рассмотрена в рецепте 15.3.

Сначала определите, для чего нужна конкретная операция или часть логики. Предположим, что вашему приложению необходима интеграция с системой одноразовых временных паролей (TOTP), например Google Authenticator. Для этого вам понадобится библиотека TOTP. Чтобы найти ее, откройте браузер и перейдите на packagist.org — репозиторий пакетов PHP. Главная страница будет выглядеть примерно как на рис. 15.1 с поисковой строкой в верхней части.

The screenshot shows the Packagist.org homepage. At the top, there's a navigation bar with links for 'Browse', 'Submit', 'Create account', and 'Sign in'. Below the navigation is a search bar with the placeholder 'Search packages...'. A small icon of a dog is visible next to the search bar. The main content area has two main sections: 'Getting Started' on the left and 'Publishing Packages' on the right. Under 'Getting Started', there's a section titled 'Define Your Dependencies' with a code snippet:

```
{
  "require": {
    "vendor/package": "1.3.2",
    "vendor/package2": "^2.0",
    "vendor/package3": "2.0.3"
  }
}
```

Below this, it says 'For more information about packages versions usage, see the composer documentation.' Under 'Install Composer In Your Project', there's a command line example:

```
curl -sS https://getcomposer.org/installer | php
```

And a note to download composer.phar. There are also sections for 'Install Dependencies', 'Autoload Dependencies', and a code example for 'require vendor/autoload.php;'. Under 'Publishing Packages', there's a section titled 'Define Your Package' with a code snippet:

```
{
  "name": "your-vendor-name/package-name",
  "description": "A short description of what your package does",
  "require": {
    "php": ">=7.4",
    "another-vendor/package": "1.*"
  }
}
```

It says 'This is the strictly minimal information you have to give.' Below this, there are sections for 'Validate The File', 'Commit The File', 'Publish It', and 'Sharing Private Code'.

At the bottom of the page, there's a footer with links for 'About Packagist', 'Statistics', 'API', 'Status', and social media icons for email, Twitter, and GitHub. It also mentions 'Packagist maintenance and hosting is provided by Private Packagist'.

**Рис. 15.1.** Packagist — это метод бесплатного распространения пакетов PHP, устанавливаемых через Composer

Затем найдите нужный вам инструмент (в данном случае TOTP). Вы получите список доступных проектов, упорядоченных по степени их популярности. Вы можете дополнительно указать тип пакета и различные теги, связанные с каждой библиотекой, для сужения результатов поиска до нескольких возможных вариантов.



Популярность пакета на Packagist определяется как количеством скачиваний, так и рейтингом на GitHub. Это хороший способ оценить, насколько часто библиотека используется на практике, но далеко не единственный, на который следует обращать внимание. Многие разработчики по-прежнему массово копируют и вставляют сторонний код в свои программы, поэтому количество «загрузок», не отраженных в статистике Packagist, может достигать миллионов. Кроме того, высокая популярность или широкое распространение пакета не означает, что он безопасен или подходит для ваших целей. Потратьте время на тщательный анализ библиотеки, чтобы удостовериться, что она не несет ненужного риска для вашего приложения.

Помимо прочего, если вам известен конкретный автор модуля, чью работу вы считаете надежной, вы можете искать его напрямую. Например, поиск по запросу Eric Mann totp выдаст конкретную реализацию TOTP (<https://oreil.ly/7touz>), созданную автором этой книги.

## Читайте также

Packagist.org: репозиторий пакетов PHP.

### 15.3. Установка и обновление пакетов Composer

#### Задача

Вы нашли на Packagist пакет, который хотите включить в свой проект.

#### Решение

Установите пакет через Composer (предполагается версия 1.0) следующим образом:

```
composer require "vendor/package:1.0"
```

#### Обсуждение

Composer работает с двумя файлами в вашей локальной файловой системе: `composer.json` и `composer.lock`. Первый вы создаете для описания вашего проекта, правил автозагрузки и используемых лицензий. Например, возьмем исходный файл `composer.json` из рецепта 15.1:

```
{
  "name": "ericmann/cookbook",
  "type": "project",
  "license": "MIT",
```

```
    "require": {},
    "autoload": {
        "psr-4": {
            "Cookbook\\": "src/"
        }
    }
}
```

После выполнения оператора `require` из примера «Решения» Composer обновит ваш файл `composer.json`, добавив в него указанную зависимость от поставщика. Теперь ваш файл будет выглядеть следующим образом:

```
{
    "name": "ericmann/cookbook",
    "type": "project",
    "license": "MIT",
    "require": {
        "vendor/package": "1.0"
    },
    "autoload": {
        "psr-4": {
            "Cookbook\\": "src/"
        }
    }
}
```

Когда вы подключаете пакет, Composer выполняет три действия.

1. Он проверяет, существует ли пакет, и выбирает либо последнюю версию, либо указанную вами. Затем он обновляет `composer.json`, чтобы сохранить пакет в ключе `require`.
2. По умолчанию Composer загружает и устанавливает пакет в каталог `vendor/` вашего проекта. Он также обновляет скрипт автозагрузки, так что пакет сразу же становится доступным для всех частей проекта.
3. Composer также поддерживает файл `composer.lock`, который явно определяет, какие версии пакетов установлены.

В примере из «Решения» была выбрана версия пакета 1.0. Если этого не сделать, Composer скачает последнюю доступную версию. Если же версия 1.0 и есть последняя, Composer поместит знак `^` перед `1.0` и в будущем установит все потенциальные основные версии (например, `1.0.1`). Файл `composer.lock` отслеживает точную версию каждого пакета, поэтому, даже если удалить весь каталог `vendor/` и заново установить пакеты через `composer install`, вы все равно получите те же версии, что и раньше.

Composer также будет стремиться найти версию, наиболее подходящую для вашего локального окружения. Для этого он сравнивает версию PHP, необходимую для вашего окружения (и используемый для запуска инструмент), с теми версиями,

которые поддерживаются запрашиваемыми пакетами. Composer также предпримет попытку согласовать все зависимости, как явно объявленные вашим проектом, так и неявно импортированные через транзитивную зависимость, объявленную в другом месте. Если системе не удастся найти совместимую версию, она сообщит об ошибке и вы сможете вручную сверить номера версий, указанные в файле `composer.json`.



В своих ограничениях по версиям Composer следует семантическому версионированию. Требование `^1.0` позволяет устанавливать только поддерживаемые версии (например, `1.0.1`, `1.0.2`). Ограничение вида `>=1.0` разрешает устанавливать любую стабильную версию, начиная с `1.0` и выше. Важно отслеживать, как вы задаете ограничения версий, чтобы случайно не импортировать изменения в пакете, внесенные в основные версии. Дополнительную информацию о том, как определять ограничения версий, можно найти в документации Composer (<https://oreil.ly/gvoGC>).

Общедоступные библиотеки, размещенные в пакетах, — не единственное, что можно включить в Composer. Вы можете указать Composer на публичные или частные проекты, размещенные, например, на GitHub.

Для этого добавьте ключ `repositories` в `composer.json`, чтобы система знала, где его искать. Затем обновите ключ `require`, чтобы он подтягивал нужный проект. Запустив `composer update`, Composer скачает пакет не из Packagist, а непосредственно из GitHub и включит его в проект, как и любую другую библиотеку.

Например, если вы хотите использовать определенную библиотеку ТОТР и обнаружили в ней незначительную ошибку, создайте ответвление исходного репозитория GitHub под своей учетной записью и ветку в GitHub для хранения своих изменений. Наконец, обновите `composer.json`, указав в нем на ответвление и ветку на GitHub, как показано в примере 15.2.

### Пример 15.2. Использование Composer для извлечения проектов из репозитория GitHub

```
{  
    "name": "ericmann/cookbook",  
    "type": "project",  
    "license": "MIT",  
    "repositories": [  
        {  
            "type": "vcs",  
            "url": "\https://github.com/phpcookbookreader/package" ❶  
        }  
    ],  
    "require": {  
        "vendor/package": "dev-bugfix". ❷  
    },
```

```
"minimum-stability": "dev", ❸
"autoload": {
    "psr-4": {
        "Cookbook\\": "src/"
    }
}
}
```

❶ Убедитесь, что пакет, который вы хотите включить, находится в репозитории, к которому у вас есть доступ. Он может быть как публичным, так и частным. Если он частный, то необходимо предоставить персональный токен доступа к GitHub в качестве переменной окружения, чтобы у Composer были соответствующие полномочия для извлечения кода.

❷ Когда репозиторий определен, добавьте спецификацию новой ветки в блок `require`. Чтобы Composer знал, какую ветку следует использовать, добавьте к имени ветки префикс `dev-`.

❸ Для включения ветвей разработки в проект, необходимо указать минимальный уровень стабильности (<https://oreil.ly/U9iWR>), требуемый проектом, чтобы избежать потенциальных проблем с подключением.

Попадет ли библиотека в ваш проект как общедоступный пакет, репозиторий или жестко закодированный ZIP-артефакт (<https://oreil.ly/xEpJh>), зависит от команды разработчиков. Однако любой многократно используемый пакет легко загружается через Composer и становится доступным в других частях вашего приложения.

## Читайте также

Документация по команде `require` (<https://oreil.ly/d32oK>) в Composer.

# 15.4. Установка нативных расширений PHP

## Задача

Вы хотите установить общедоступное расширение, которое написано специально для PHP, например APC User Cache (APCu) (<https://oreil.ly/Jppw->).

## Решение

Найдите расширение в репозитории PECL и установите его в систему с помощью PEAR:

```
$ pecl install apcu
```

## Обсуждение

Сообщество PHP применяет две технологии для распространения расширений самого языка: PEAR и PECL. Основное различие между ними в том, для каких пакетов они используются.

Сам PEAR может упаковать практически все, что угодно. Распространяемые им пакеты упаковываются в виде gzip-архивов TAR, состоящих из PHP-кода. Таким образом, PEAR, как и Composer, можно задействовать для управления, установки и обновления различных PHP-библиотек в вашем приложении<sup>1</sup>. Пакеты PEAR загружаются не так, как в Composer, поэтому будьте внимательны, если решите совместить эти два менеджера пакетов.

PECL — это библиотека встроенных расширений для PHP, написанная на языке C, являющимся основой для самого PHP. PECL использует PEAR для установки и управления расширениями, а доступ к новым функциям, представленным в расширении, реализуется теми же способами, что и к функциям самого языка.

Многие пакеты PHP, включенные в современные версии языка, изначально создавались как расширения PECL и могли быть установлены разработчиками для тестирования и первичной интеграции. Например, библиотека шифрования sodium (<https://oreil.ly/QdfyM>) впервые появилась как расширение PECL, а затем была добавлена в основной дистрибутив PHP, начиная с версии 7.2<sup>2</sup>.

Некоторые базы данных (например, MongoDB ([https://oreil.ly/Xoh5\\_](https://oreil.ly/Xoh5_))) распространяют свои основные драйверы для PHP в виде встроенных расширений PECL. Существуют также различные библиотеки для работы с сетями, мультимедиа и терминалом. Все они написаны на C и благодаря PECL и привязкам к PHP функционируют словно это часть самого языка.

В отличие от инструментов вроде Composer, PECL поставляет необработанный C-код прямо в ваше окружение. Команда `install` выполняет следующие действия:

- загружает исходный код расширения;
- компилирует исходный код под вашу систему, используя локальную среду, ее конфигурацию и архитектуру системы для обеспечения совместимости;
- создает скомпилированный файл с расширением .so в каталоге расширений (<https://oreil.ly/KFNg9>), определенным вашим окружением.

---

<sup>1</sup> Подробнее об установке пакетов через Composer см. в рецепте 15.3.

<sup>2</sup> Расширение подробно рассматривается в главе 9.



Несмотря на то что некоторые расширения, как оказалось, активируются самостоятельно, скорее всего, вам придется изменить файл php.ini вашей системы, чтобы явно включить расширение. После этого следует перезапустить веб-сервер (Apache, NGINX или аналогичный), чтобы убедиться, что PHP загрузил новое расширение.

В системах Linux предварительно скомпилированное нативное расширение можно установить с помощью менеджера пакетов. Установка APCu в системе Ubuntu Linux выглядит так:

```
$ sudo apt install php-apcu
```

Независимо от того, используете ли вы PECL для сборки расширения или предварительно скомпилированные двоичные файлы через менеджер пакетов, работа с расширениями PHP проста. Эти модули расширяют возможности языка и помогают вашим приложениям стать значительно полезнее.

## Читайте также

Документация по репозиторию PECL (<https://oreil.ly/28K08>) и системе упаковки расширений PEAR (<https://pear.php.net/>).

## ГЛАВА 16

---

# Базы данных

Современное программное обеспечение, особенно веб-приложения, используют для своей работы состояние. Состояние — это способ представления текущей ситуации с приложением для конкретного запроса: кто вошел в систему, на какой странице он находится, какие предпочтения задал.

Как правило, код пишется более или менее статичным. Он будет функционировать одинаково независимо от состояния пользовательской сессии (именно это делает поведение системы предсказуемым в рамках приложения для нескольких пользователей). Когда веб-приложение развертывается, оно опять же выполняется в режиме «без состояния».

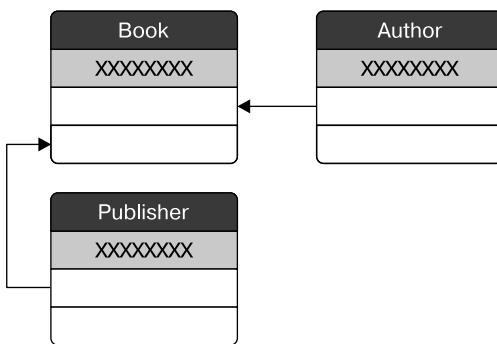
Но состояние жизненно важно для отслеживания активности пользователя и изменения поведения приложения по мере взаимодействия с пользователем. Для того чтобы код, не имеющий состояния, знал о состоянии, ему нужно получить это состояние из какого-то источника.

Как правило, для этого используется база данных. Базы данных — это эффективный способ хранения структурированной информации. В PHP существуют четыре типа БД: реляционные, графовые, документоориентированные и базы данных «ключ — значение».

## Реляционные базы данных

В реляционной базе данные разбиваются на объекты и связи между ними. Конкретная запись, например книга, представлена в виде строки в таблице, а столбцы этой таблицы содержат дополнительную информацию. Эти столбцы могут включать название, ISBN и тему. Главное, что нужно помнить о реляционных БД, — это то, что разные типы данных хранятся в различных таблицах.

Хотя один столбец таблицы `book` может содержать имя автора, скорее всего, у вас будет отдельная таблица `author` для этого. Обе таблицы могут иметь отдельные столбцы ID, а таблица `book` — столбец `author_id`, ссылающийся на таблицу `author`. На рис. 16.1 показаны отношения между подобными таблицами.



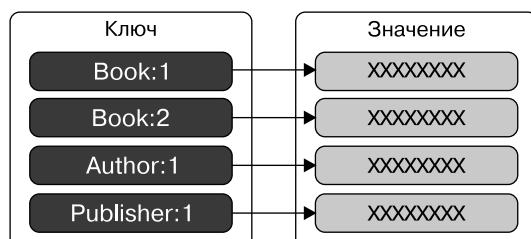
**Рис. 16.1.** Реляционные базы данных определяются таблицами и ссылками между элементами в каждой из них

Примерами реляционных БД являются MySQL (<https://www.mysql.com/>) и SQLite (<https://oreil.ly/5s4ps>).

## База данных «ключ — значение»

База данных «ключ — значение» гораздо проще реляционной. По сути, это одна таблица, в которой один идентификатор (ключ) сопоставляется с некоторым хранимым значением. Многие приложения используют подобные хранилища в качестве обычных утилит кэширования, отслеживая примитивные значения в эффективной памяти системы поиска.

Как и в реляционных БД, данные, хранящиеся в системе «ключ — значение», могут быть типизированы. Если вы работаете с числовыми данными, большинство таких систем предоставляют дополнительные функции для прямого управления этими данными. Например, вы можете увеличивать целочисленные значения без необходимости предварительно считывать основные данные. На рис. 16.2 показаны отношения «один к одному» между ключами и значениями.



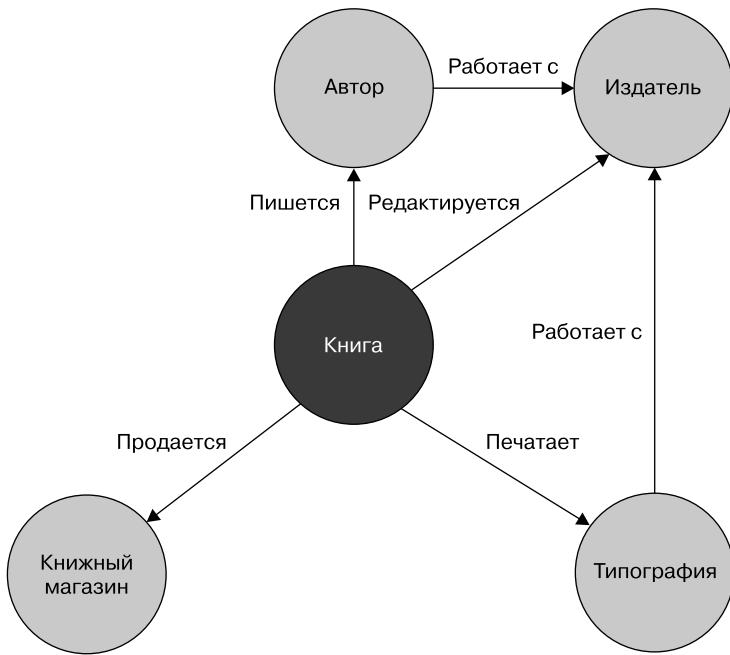
**Рис. 16.2.** Базы данных «ключ — значение» структурированы как поиск между отдельными идентификаторами, сопоставленными с опционально типизированными значениями

Примерами подобных хранилищ являются Redis (<https://redis.io/>) и Amazon DynamoDB (<https://oreil.ly/BYCIM>).

## Графовые базы данных

Вместо того чтобы сосредоточиваться на структурировании самих данных, графовые БД фокусируются на моделировании отношений (называемых ребрами) между данными. Элементы данных заключены в узлы, а ребра между узлами связывают их вместе и предоставляют семантический контекст данных в системе.

Из-за высокого приоритета, придаваемого связям между данными, графовые базы данных хорошо подходят для визуализаций, подобных рис. 16.3, иллюстрирующе-му ребра и узлы в такой структуре. Они также обеспечивают высокоэффективные запросы к отношениям между данными, что делает их надежным выбором для работы с данными с сильной связью.



**Рис. 16.3.** Графовые базы данных определяют приоритеты и показывают отношения (ребра) между данными (узлами)

Примерами графовых баз данных являются Neo4j (<https://neo4j.com/>) и Amazon Neptune (<https://oreil.ly/8Uezn>).

## Документоориентированные базы данных

Данные также допускается хранить в виде неструктурированных или полуструктурных документов. Документ может быть хорошо структурированным фрагментом данных (как литеральный XML-документ) или блоком байтов в свободной форме (например, PDF).

Ключевым отличием документоориентированных баз данных от других, рассмотренных ранее, является структура. Документоориентированные хранилища не структурированы и используют динамическую схему ссылок на данные. Они невероятно полезны в некоторых ситуациях, но их использование требует гораздо более тонкого подхода. Для глубокого погружения в методологию работы с документами читайте книгу Шеннона Брэдшоу «MongoDB. Полное руководство».

Следующие рецепты в основном посвящены реляционным БД и их применению в PHP. Вы узнаете, как подключаться к локальным и удаленным БД, как использовать фиксированные данные во время тестирования, а также более сложную библиотеку объектно-реляционного отображения (ORM) для работы с вашими данными.

## 16.1. Подключение к базе данных SQLite

### Задача

Вы хотите использовать локальную копию БД SQLite для хранения данных приложения. Вашему приложению необходимо корректно открывать и закрывать базу данных.

### Решение

Для этого воспользуйтесь классом `SQLite`. Для большей эффективности допускается расширить класс собственным конструктором и деструктором:

```
class Database extends SQLite3
{
    public function __construct(string $databasePath)
    {
        $this->open($databasePath);
    }

    public function __destruct()
    {
        $this->close();
    }
}
```

Затем с помощью своего нового класса откройте базу данных, чтобы выполнить несколько запросов и автоматически закрыть соединение по завершении работы. Например:

```
$db = new Database('example.sqlite');

$create_query = <<SQL
CREATE TABLE IF NOT EXISTS users (
    user_id INTEGER PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE
);
SQL;

$db->exec($create_query);

$insert_query = <<SQL
INSERT INTO users (first_name, last_name, email)
VALUES ('Eric', 'Mann', 'eric@phpcookbook.local')
ON CONFLICT(email) DO NOTHING;
SQL;

$db->exec($insert_query);

$results = $db->query('SELECT * from users;');
while ($row = $results->fetchArray()) {
    var_dump($row);
}
```

## Обсуждение

SQLite — это быстрая, полностью автономная БД, которая хранит все свои записи в одном файле на диске. PHP поставляется с расширением (включенным по умолчанию в большинстве дистрибутивов), которое напрямую взаимодействует с этой БД, предоставляя вам возможность создавать, записывать и читать данные из хранилищ по своему усмотрению.

Метод `open()` по умолчанию генерирует файл базы данных, если он еще не существует в указанном каталоге. Это поведение можно перенастроить, изменив флаги, заданные в качестве второго параметра при вызове метода. По умолчанию PHP будет передавать флаги `SQLITE3_OPEN_READWRITE | SQLITE3_OPEN_CREATE`,

которые откроют БД для чтения и записи, а также создадут ее, если она еще не существует.

Доступные флаги перечислены в табл. 16.1.

**Таблица 16.1.** Дополнительные флаги, доступные для открытия БД SQLite

Флаг	Описание
SQLITE3_OPEN_READONLY	Открыть базу данных только для чтения
SQLITE3_OPEN_READWRITE	Открыть базу данных как для чтения, так и для записи
SQLITE3_OPEN_CREATE	Создать базу данных, если она не существует

Пример из «Решения» включает класс, который открывает базу данных SQLite по указанному пути, создавая ее, если она не существует. Учитывая, что наш класс расширяет базовый класс SQLite, разрешается использовать его вместо стандартного экземпляра SQLite для генерации таблиц, вставки данных и осуществления прямых запросов к ним. Деструктор класса автоматически закрывает соединение с БД, как только экземпляр выходит из области видимости.



Обычно явного закрытия соединения SQLite не требуется, так как PHP автоматически закрывает соединение при выходе из программы. Однако если есть вероятность, что приложение (или поток) может продолжить работу, стоит закрыть соединение, чтобы освободить системные ресурсы. Для локального соединения с файловой базой данных это не столь критично, однако при работе с удаленными реляционными БД, такими как MySQL, это важно. Последовательность в управлении БД — полезная привычка.

База данных SQLite хранится в виде двоичного файла на диске. Если у вас есть среда разработки, например Visual Studio Code ([https://oreil.ly/k\\_LBI](https://oreil.ly/k_LBI)), можно использовать специальные расширения, такие как SQLite Viewer (<https://oreil.ly/QzF0J>), для подключения к локальной БД и ее визуализации. Наличие нескольких методов просмотра схемы и данных в БД — быстрый и эффективный способ убедиться в корректности работы вашего кода.

## Читайте также

Документация по PHP для расширения базы данных SQLite3 (<https://oreil.ly/kMU8Y>).

## 16.2. Использование PDO для подключения к внешнему провайдеру баз данных

### Задача

Вы хотите использовать PDO в качестве уровня абстракции для подключения к удаленной базе данных MySQL и выполнения запросов к ней.

### Решение

Сначала определите класс, расширяющий основное определение PDO, который будет обрабатывать создание и закрытие соединений:

```
class Database extends PDO
{
    public function __construct($config = 'database.ini')
    {
        $settings = parse_ini_file($config, true);

        if (!$settings) {
            throw new RuntimeException("Ошибка чтения конфигурации: `{$config}`.");
        } else if (!array_key_exists('database', $settings)) {
            throw new RuntimeException("Неверная конфигурация: `{$config}`.");
        }

        $db = $settings['database'];
        $port = $db['port'] ?? 3306;
        $driver = $db['driver'] ? 'mysql';
        $host = $db['host'] ?? '';
        $schema = $db['schema'] ?? '';
        $username = $db['username'] ?? null;
        $password = $db['password'] ?? null;

        $port = empty($port) ? '' : ";port={$port}";
        $dsn = "{$driver}:host={$host}{$port};dbname={$schema}";

        parent::__construct($dsn, $username, $password);
    }
}
```

Конфигурационный файл для предыдущего класса должен быть в формате INI. Например:

```
[database]
driver = mysql
```

```
host = 127.0.0.1
port = 3306
schema = cookbook
username = root
password = toor
```

После того как файл настроен, вы можете напрямую направлять запросы к базе данных с помощью абстракций, предоставленных PDO:

```
$db = new Database();

$create_query = <<SQL
CREATE TABLE IF NOT EXISTS users (
    user_id int NOT NULL AUTO_INCREMENT,
    first_name varchar(255) NOT NULL,
    last_name varchar(255) NOT NULL,
    email varchar(255) NOT NULL UNIQUE,
    PRIMARY KEY (user_id)
);
SQL;

$db->exec($create_query);

$insert_query = <<SQL
INSERT IGNORE INTO users (first_name, last_name, email)
VALUES ('Eric', 'Mann', 'eric@phpcookbook.local');
SQL;

$db->exec($insert_query);

foreach($db->query('SELECT * from users;') as $row) {
    var_dump($row);
}
```

## Обсуждение

В примере выше применяется та же структура таблиц и данных, что и в рецепте 16.1, лишь с разницей в используемом здесь движке базы данных MySQL. MySQL (<https://www.mysql.com/>) — это популярная бесплатная БД с открытым исходным кодом, поддерживаемая компанией Oracle. По словам разработчиков, с ней взаимодействуют многие популярные веб-приложения, включая Netflix и Uber (<https://oreil.ly/fluva>). Фактически MySQL настолько популярна, что те, кто отвечают за сопровождение системы, по умолчанию поставляют расширение MySQL вместе с PHP, что еще больше упрощает подключение к системе и избавляет вас от необходимости самостоятельно устанавливать новые драйверы.



В PHP нет метода для явного закрытия соединения при использовании PDO, как в примере из рецепта 16.1. Вместо этого установите значение обработчика базы данных (представленного в виде \$db выше) как null, чтобы вывести объект из области видимости и закрыть соединение.

В примере из «Решения» вы сначала определили класс для обертывания самого PDO и абстрагирования соединения к БД MySQL. Это не обязательно, но, как и в рецепте 16.1, это хороший способ приучить себя к поддержанию чистых соединений с данными. Как только соединение будет установлено, вы сможете создать таблицу, вставить данные и считать их.



В примере из «Решения» предполагается, что схема cookbook уже существует в БД, к которой вы подключаетесь. Если вы не создавали ее, такое неявное подключение завершится ошибкой PDOException, сообщающей о неизвестной базе данных. Крайне важно сначала создать схему в БД MySQL, прежде чем работать с ней.

В отличие от SQLite для MySQL требуется отдельное приложение. Зачастую оно запускается на другом сервере, а ваша программа подключается по TCP по определенному порту (обычно 3306). Для локальной разработки и тестирования достаточно запустить базу данных вместе с вашим приложением с помощью Docker (<https://www.docker.com/>). Следующая команда создаст контейнер Docker с локальной базой данных MySQL по порту 3306 и установит пользователю root пароль toor для входа:

```
$ docker run --name db -e MYSQL_ROOT_PASSWORD=toor -p 0.0.0.0:3306:3306 -d mysql
```



Независимо от того, в какой среде используется MySQL в Docker, в официальном образе контейнера (<https://oreil.ly/4btCa>) подробно описаны различные параметры конфигурации для настройки и обеспечения безопасности среды.

При первом запуске в контейнере не будет ни одной схемы, доступной для запроса (то есть остальная часть примера из «Решения» пока не может быть использована). Чтобы сформировать стандартную схему для cook book, необходимо подключиться к базе данных и создать схему. В примере 16.1 символ \$ обозначает команды оболочки, а приглашение mysql> — команду, выполняемую в самой базе данных.

### Пример 16.1. Использование MySQL CLI для создания схемы базы данных

```
$ mysql --host 127.0.0.1 --user root --password=toor ①
```

```
mysql> create database `cookbook`; ②
```

```
mysql> exit ③
```

❶ Контейнер Docker открывает MySQL по TCP для локального окружения, что требует указания локального хоста по IP-адресу. Если этого не сделать, MySQL попытается подключиться через сокет Unix, что в данном случае не удастся. Также необходимо указать имя пользователя и пароль.

❷ Подключившись к движку базы данных, вы можете создать в нем новую схему.

❸ Чтобы отключиться от MySQL, просто введите `exit` или `quit` и нажмите клавишу `Enter`.

Если у вас не установлена командная строка MySQL, воспользуйтесь интерфейсом командной строки Docker для подключения к запущенному контейнеру базы данных. В примере 16.2 показано, как с помощью контейнера Docker обернуть MySQL CLI при создании схемы базы данных.

**Пример 16.2.** Использование Docker-хостинга MySQL CLI для создания схемы базы данных

```
$ docker exec -it db bash ❶
$ mysql --user root --password=toor ❷
mysql> create database `cookbook`; ❸
mysql> exit
$ exit ❹
```

❶ Поскольку MySQL уже запущен локально в виде контейнера с именем `db`, вы можете выполнить команду внутри контейнера в интерактивном режиме, обратившись к тому же имени. Флаги `i` и `t` Docker указывают на то, что требуется выполнить команду в интерактивной сессии терминала. Команда `bash` явно указывает, что необходимо запустить интерактивную оболочку `bash` внутри контейнера.

❷ Подключиться к базе данных внутри контейнера так же просто, как использовать MySQL CLI. Вам не нужно ссылаться на имя хоста, поскольку внутри контейнера можно напрямую подключиться к открытому сокету Unix.

❸ Создание таблицы и выход из MySQL CLI происходят точно так же, как и в предыдущем примере.

❹ После выхода из CLI вам все равно нужно выйти из интерактивного режима `bash` внутри контейнера Docker, чтобы вернуться в основной терминал.

Есть два ключевых преимущества использования PDO вместо прямого функционального интерфейса с драйверами для соединения с БД:

- интерфейсы PDO одинаковы для всех технологий баз данных. Хотя вам может потребоваться рефакторинг определенных запросов под тот или иной движок БД (сравните синтаксис `CREATE TABLE` в этом примере и рецепте 16.1),

нет необходимости перерабатывать PHP-код для подключений, выполнения операторов или обработки запросов. PDO — это уровень абстракции доступа к данным, который предоставляет вам одинаковый метод доступа и управления вне зависимости от используемой в приложении БД;

- PDO поддерживает постоянные соединения, передавая значение `true` ключу `PDO::ATTR_PERSISTENT` при открытии соединения. Такое соединение остается открытым даже после выхода из области видимости экземпляра PDO и окончания выполнения скрипта. При попытке повторного открытия соединения система будет искать уже существующее соединение и повторно использовать его. Это помогает повысить производительность длительно работающих многопользовательских приложений, где открытие нескольких соединений может повредить саму базу данных. (Подробнее о постоянных соединениях с базами данных можно прочитать в документации PHP ([https://oreil.ly/\\_nHH-](https://oreil.ly/_nHH-))).

Помимо этих двух преимуществ, PDO также поддерживает концепцию подготовленных (или параметризованных) запросов, которые помогают снизить риск вредоносных SQL-инъекций. Подробнее об этом читайте в рецепте 16.3.

## Читайте также

Полная документация по расширению PDO ([https://oreil.ly/\\_6--V](https://oreil.ly/_6--V)).

## 16.3. Очистка пользовательского ввода для запроса к базе данных

### Задача

Вы хотите передать пользовательский ввод в запрос к базе данных, но не уверены, что он не является вредоносным.

### Решение

Используйте подготовленный запрос в PDO для автоматической очистки пользовательского ввода перед его передачей в запрос:

```
$db = new Database();

$insert_query = <<SQL
INSERT IGNORE INTO users (first_name, last_name, email)
VALUES (:first_name, :last_name, :email);
SQL;
```

```
$statement = $db->prepare($insert_query);

	statement->execute([
		'first_name' => $_POST['first'],
		'last_name'  => $_POST['last'],
		'email'      => $_POST['email']
	]);
foreach($db->query('SELECT * from users;') as $row) {
	var_dump($row);
}
```

## Обсуждение

Концепция санитизации пользовательского ввода уже обсуждалась в рамках рецепта 9.1. Хотя такой подход достаточно эффективен, разработчики могут забыть включить фильтр пользовательского ввода при выполнении обновлений. Гораздо безопаснее явно подготавливать запросы к выполнению, чтобы предотвратить вредоносные SQL-инъекции.

Рассмотрим запрос, с помощью которого осуществляется поиск данных пользователя с целью отображения информации о его профиле. Подобный запрос может использовать адреса электронной почты пользователей в качестве индексов, чтобы отличить одного пользователя от другого. Например:

```
SELECT * FROM users WHERE email = ?;
```

В PHP вам потребуется передать адрес электронной почты текущего пользователя, чтобы запрос работал эффективно. Один из подходов показан в примере 16.3.

### Пример 16.3. Простой запрос с интерполяцией строк

```
$db = new Database();

$statement = "SELECT * FROM users WHERE email = '{$_POST['email']}'";

$results = $db->query($statement);
var_dump($results);
```

Если человек вводит только свое имя пользователя (например, eric@phpcookbook.local), то этот запрос вернет соответствующие данные для этого пользователя. Однако нет никакой гарантии, что конечный пользователь не злоумышленник и что он не отправит вредоносный код в надежде внедрить произвольный запрос в движок вашей базы данных. Зная, как адрес электронной почты интерполируется в SQL-запрос, взломщик может отправить вместо него ' OR 1=1;--.

Эта строка завершит кавычки (WHERE email = ''), добавит составное булево выражение, соответствующее любому результату (OR 1=1), и явно закомментирует

все последующие дополнительные символы. В итоге ваш запрос вернет данные по всем пользователям, а не по тому, который сделал запрос.

Аналогично злоумышленники могут применить тот же подход для внедрения произвольных операторов `INSERT` (запись новых данных) там, где вы ожидаете только чтения информации. Они также способны незаметно обновлять существующие данные, удалять поля или иным образом нарушать надежность вашего хранилища данных.

Внедрение SQL-кода невероятно опасно. Кроме того, подобный тип атак настолько распространен в мире программного обеспечения, что он признан третьим наиболее часто встречающимся риском безопасности приложений по версии проекта Open Worldwide Application Security Project (OWASP) Top Ten (<https://oreil.ly/Cveyu>).

К счастью, в PHP инъекции также легко предотвратить!

В «Решении» представлен интерфейс параметризованных запросов PDO. Вместо интерполяции строки с данными, предоставленными пользователем, вы вставляете в запрос именованные占位符. Они должны иметь префикс с одним двоеточием и могут быть названы любым именем. При выполнении запроса к базе данных PDO заменит эти заполнители литеральными значениями, переданными во время выполнения.



Можно также задействовать символ вопросительного знака в качестве заполнителя и передавать значения в подготовленный запрос на основе их положения в простом массиве. Однако положение элементов легко перепутать при последующем рефакторинге. Чтобы избежать путаницы и обезопасить свой код на будущее, старайтесь всегда использовать именованные параметры при подготовке запросов.

Параметризованные запросы работают как с операторами манипулирования данными (вставка, обновление, удаление), так и с произвольными запросами. Простой запрос, который вы видели в примере 16.3, может быть переписан с применением подготовленных запросов, как показано в примере 16.4.

#### Пример 16.4. Простой параметризованный запрос

```
$db = new Database();

$query = "SELECT * FROM users WHERE email = :email;";
$statement = $db->prepare($query);

$statement->execute(['email' => $_POST['email']]);

$results = $statement->fetch();
var_dump($results);
```

Этот код использует PDO для автоматической обработки пользовательского ввода и передачи значения в виде литерала в механизм БД. Если пользователь действительно указал свой адрес электронной почты, запрос сработает как положено и вернет ожидаемый результат.

Если пользователь вместо этого вводит вредоносный код, например '`OR 1=1;--`', то параметризованный запрос автоматически экранирует специальные символы, такие как одинарные кавычки ('), чтобы они не интерпретировались как часть SQL запроса. В результате база данных не найдет адреса электронной почты, точно соответствующего вредоносному коду, и вернет пустой результат. При этом пользовательские данные останутся защищенными, а злоумышленник не сможет повредить вашу БД.

## Читайте также

Документация по методу `prepare()` (<https://oreil.ly/q3DCh>) в PDO.

# 16.4. Имитация данных для интеграционного тестирования

## Задача

Вы хотите использовать БД для хранения данных в производственной среде, но при этом изолировать интерфейс БД во время выполнения автоматизированных тестов приложения.

## Решение

Используйте шаблон проектирования «Репозиторий» как слой абстракции между бизнес-логикой и уровнем доступа к данным. Например, определите интерфейс хранилища, как показано в примере 16.5.

### Пример 16.5. Определение интерфейса хранилища данных

```
Interface BookRepository
{
    public function getById(int $bookId): Book;
    public function list(): array;
    public function add(Book $book): Book;
    public function delete(Book $book): void;
    public function save(Book $book): Book;
}
```

Затем используйте предыдущий интерфейс для определения конкретной реализации базы данных (применяя что-то вроде PDO). С помощью того же интерфейса определите имитационную реализацию, которая возвращает предсказуемые, статические данные, а не реальные данные из удаленной системы. См. пример 16.6.

**Пример 16.6.** Реализация интерфейса репозитория с имитацией данных

```
class MockRepository implements BookRepository
{
    private array $books;

    public function __construct()
    {
        $this->books = [
            new Book(id: 0),
            new Book(id: 1),
            new Book(id: 2)
        ];
    }

    public function getById(int $bookId): Book
    {
        return $this->books[$bookId];
    }

    public function list(): array
    {
        return $this->books;
    }

    public function add(Book $book): Book
    {
        $book->id = end(array_keys($this->books)) + 1;
        $this->books[] = $book;
        return $book;
    }

    public function delete(Book $book): void
    {
        unset($this->books[$book->id]);
    }

    public function save(Book $book): Book
    {
        $this->books[$book->id] = $book;
    }
}
```

## Обсуждение

В примере из «Решения» продемонстрирован простой способ отделить бизнес-логику от уровня данных через абстракции. Благодаря репозиторию данных, обрамляющему слой базы данных, вы можете поставлять несколько реализаций одного и того же интерфейса. В рабочем приложении ваше хранилище может выглядеть примерно так, как показано в примере 16.7.

### Пример 16.7. Конкретная реализация интерфейса репозитория для базы данных

```
class DatabaseRepository implements BookRepository
{
    private PDO $dbh;

    public function __construct($config = 'database.ini')
    {
        $settings = parse_ini_file($config, true);

        if (!$settings) {
            throw new RuntimeException("Ошибка чтения конфигурации: `{$config}`.");
        } else if (!array_key_exists('database', $settings)) {
            throw new RuntimeException("Неверная конфигурация: `{$config}`.");
        }

        $db = $settings['database'];
        $port = $db['port'] ?? 3306;
        $driver = $db['driver'] ? 'mysql';
        $host = $db['host'] ?? '';
        $schema = $db['schema'] ?? '';
        $username = $db['username'] ?? null;
        $password = $db['password'] ?? null;

        $port = empty($port) ? '' : ";port={$port}";
        $dsn = "{$driver}:host={$host}{$port};dbname={$schema}";

        $this->dbh = new PDO($dsn, $username, $password);
    }

    public function getById(int $bookId): Book
    {
        $query = 'Select * from books where id = :id';

        $statement = $this->dbh->prepare($query);
        $statement->execute(['id' => $bookId]);

        $record = $statement->fetch();
    }
}
```

```
if ($record) {
    return Book::fromRecord($record);
}

throw new Exception('Книга не найдена');
}

public function list(): array
{
    $books = [];

$records = $this->dbh->query('select * from books;');
foreach($record as $book) {
    $books[] = Book::fromRecord($book);
}

return $books;
}

public function add(Book $book): Book
{
    $query = 'insert into books (title, author) values (:title, :author);';

$this->dbh->beginTransaction();
$statement = $this->dbh->prepare($query);
$statement->execute([
    'title' => $book->title,
    'author' => $book->author,
]);
$this->dbh->commit();

$book->id = $this->dbh->lastInsertId();

return $book;
}

public function delete(Book $book): void
{
    $query = 'delete from books where id = :id';
$this->dbh->beginTransaction();
$statement = $this->dbh->prepare($query);
$statement->execute(['id' => $book->id]);
$this->dbh->commit();
}

public function save(Book $book): Book
{
    $query = 'update books set title = :title,
        author = :author where id = :id';
```

```
$this->dbh->beginTransaction();
$statement = $this->dbh->prepare($query);
$statement->execute([
    'title' => $book->title,
    'author' => $book->author,
    'id' => $book->id
]);
$this->dbh->commit();

return $book;
}
}
```

Пример 16.7 реализует тот же интерфейс, что и макет хранилища из «Решения», за исключением того, что он подключается к «живой» базе данных MySQL и управляет данными в этой отдельной системе. В реальности ваш рабочий код будет обрабатывать эту реализацию, а не имитацию. Однако при тестировании вы можете легко поменять `DatabaseRepository` на экземпляр `MockRepository`, если ваша бизнес-логика ожидает класс, реализующий `BookRepository`.

Предположим, что вы работаете с фреймворком Symfony (<https://symfony.com/>). Ваше приложение строится на контроллерах, использующих инъекцию зависимостей для взаимодействия с внешними интеграциями. Для API библиотеки, управляющей несколькими книгами, достаточно определить контроллер `BookController`, который будет выглядеть следующим образом:

```
class BookController extends AbstractController
{
    #[Route('/book/{id}', name: 'book_show')]
    public function show(int $id, BookRepository $repo): Response
    {
        $book = $repo->getById($id);

        // ...
    }
}
```

Преимущество предыдущего кода в том, что контроллеру все равно, передаете ли вы ему экземпляр `MockRepository` или `DataRepository`. Оба класса реализуют один и тот же интерфейс `BookRepository` и предоставляют метод `getByID()` с одинаковой сигнатурой. Для вашей бизнес-логики функциональность идентична за исключением того, что в первом случае приложение будет обращаться к удаленной БД для получения (и, возможно, изменения) данных, а во втором — к статичному, полностью детерминированному набору фиктивных данных.



Стандартный слой абстракции данных, поставляемый с Symfony, называется Doctrine ([https://oreil.ly/JvdG\\_](https://oreil.ly/JvdG_)) и по умолчанию использует паттерн «репозиторий». Doctrine предоставляет обширный уровень абстракции для различных вариантов SQL, включая MySQL, без необходимости вручную создавать запросы через PDO. Кроме того, в комплект поставки входит утилита командной строки, которая автоматически генерирует PHP-код как для хранимых объектов (называемых сущностями), так и для репозиториев!

Когда речь идет о написании тестов, детерминированные и фиктивные данные лучше реальных, потому что они всегда будут одинаковыми, а значит, ваши тесты будут очень надежными. Это также говорит о том, что вы не сможете случайно перезаписать данные в настоящей базе данных, если кто-то локально допустит ошибку в конфигурации.

Дополнительным преимуществом станет скорость выполнения тестов. Имитация интерфейсов данных избавляет от необходимости пересыпать данные между вашим приложением и независимой БД, значительно сокращая время ожидания любых вызовов функций, связанных с данными. Тем не менее вам, вероятно, все равно понадобится отдельный набор интеграционных тестов для проверки удаленных интеграций, и для того, чтобы его можно было использовать, вам потребуется реальная база данных.

## Читайте также

Ознакомьтесь с рецептом 8.7, чтобы узнать больше о классах, интерфейсах и наследовании. Подробнее о контроллерах (<https://oreil.ly/ucip3>) и инъекции зависимостей (<https://oreil.ly/WYpxe>) читайте в документации Symfony.

## 16.5. Запрос к базе данных SQL с помощью Eloquent ORM

### Задача

Вы хотите управлять схемой вашей БД и содержащимися в ней данными без ручного написания SQL.

### Решение

Используйте стандартный ORM Laravel Eloquent, чтобы динамически определять объекты данных и схему, как показано в примере 16.8.

**Пример 16.8.** Определение таблицы для использования в Laravel

```
Schema::create('books', function (Blueprint $table) {
    $table->id();
    $table->string('title');
    $table->string('author');
});
```

Этот код можно использовать для динамического создания таблиц, независимо от типа SQL в Eloquent. После создания таблицы данные в ней могут быть смоделированы Eloquent с помощью следующего класса (см. пример 16.9).

**Пример 16.9.** Определение модели Eloquent

```
use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    use HasFactory;

    public $timestamps = false;
}
```

## Обсуждение

Объектно-реляционный проектор (ORM) Doctrine, кратко упомянутый в рецепте 16.4, опирается на шаблон «репозиторий» для сопоставления объектов, хранящихся в базе данных, с их представлением в бизнес-логике. Он хорошо сочетается с фреймворком Symfony, но это лишь один из подходов к моделированию данных в реальном приложении.

Фреймворк с открытым исходным кодом Laravel, построенный на базе Symfony, для моделирования данных использует ORM Eloquent (<https://oreil.ly/x7lcI>). В отличие от Doctrine, Eloquent основан на шаблоне проектирования Active record (AR), в котором таблицы в базе данных напрямую связаны с соответствующими моделями, применяемыми для представления этих таблиц. Вместо создания/чтения/обновления/удаления моделей через отдельный репозиторий, моделируемые объекты предоставляют собственные способы управления.



Несмотря на популярность фреймворка Laravel, многие разработчики считают подход к моделированию данных с помощью AR антипаттерном, то есть подходом, которого следует избегать. Убедитесь, что ваша команда разработчиков разделяет ваши взгляды на абстракции, используемые в проекте, поскольку сочетание нескольких шаблонов одновременно может привести к путанице и серьезным проблемам с обслуживанием в дальнейшем.

Классы моделей, предоставляемые Eloquent, довольно просты, как видно в примере из «Решения». Однако они весьма динамичны — фактические свойства модели не нужно определять непосредственно в самом классе модели. Вместо этого Eloquent автоматически считывает и анализирует любые столбцы и типы данных из базовой таблицы, а также добавляет их в качестве свойств в класс модели при ее инстанцировании.

Например, в таблице из примера 16.8 определены три столбца:

- целочисленный идентификатор;
- строковое название;
- строковое имя автора.

Когда Eloquent читает эти данные напрямую, он эффективно создает объекты в PHP, которые выглядят примерно так:

```
class Book
{
    public int    $id;
    public string $title;
    public string $author;
}
```

Фактический класс будет включать различные дополнительные методы, такие как `save()`, но в остальном он содержит прямое представление данных в том виде, в котором они отображаются в вашей SQL-таблице. Чтобы создать новую запись в БД без прямого редактирования SQL, достаточно сгенерировать новый объект и сохранить его, как показано в примере 16.10.

#### **Пример 16.10.** Создание объекта базы данных с помощью Eloquent

```
$book = new Book;
$book->title = 'PHP Cookbook';
$book->author = 'Eric Mann';

$book->save();
```

Обновление данных происходит так же просто: с помощью Eloquent получите объект, который вы хотите отредактировать, внесите изменения в PHP, а затем вызовите метод `save()` объекта, чтобы сохранить результат. Пример 16.11 обновляет объекты в базе данных, заменяя одно значение в определенном поле другим.

#### **Пример 16.11.** Обновление элемента на месте с помощью Eloquent

```
Book::where('author', 'Eric Mann')
    ->update(['author', 'Eric A Mann']);
```

Ключевым преимуществом Eloquent является возможность работать с объектами данных как со встроенными объектами PHP, без необходимости писать, управлять и поддерживать SQL-запросы вручную. Еще одна особенность ORM заключается в том, что он самостоятельно обрабатывает экранирование пользовательского ввода, то есть те шаги, что мы рассмотрели в рецепте 16.3, можно опустить.

Хотя прямое использование SQL-соединений (с PDO или без него) — быстрый и эффективный способ взаимодействовать с БД, мощь полнофункционального ORM облегчит работу с вашим приложением. Это касается как начальной разработки, так и рефакторинга.

## Читайте также

Документация по Eloquent ORM (<https://oreil.ly/4J-Jz>).

## ГЛАВА 17

---

# Асинхронный PHP

Многие базовые PHP-скрипты выполняют операции синхронно, то есть скрипт запускает один монолитный процесс от начала до конца и выполняет только одно действие за раз. В современных реалиях, когда приложения усложняются, таких подходов становится недостаточно, и возникает потребность в более продвинутых методах работы. И здесь на сцену выходит асинхронное программирование.

При обсуждении асинхронного программирования часто встречаются два термина: конкурентность (*concurrent*) и параллельность (*parallel*). Когда большинство людей говорят о параллельном программировании, на самом деле они имеют в виду конкурентное программирование. В случае конкурентности ваше приложение выполняет две задачи, но не обязательно в одно и то же время. Представьте бариста, который обслуживает несколько клиентов одновременно: он работает в многозадачном режиме и готовит разные напитки, но на самом деле способен делать только один напиток за раз.

При параллельных операциях вы совершаете два действия одновременно. Представьте, что на стойке в кафе установлена капельная кофеварка. Одних посетителей по-прежнему обслуживает бариста, а другие параллельно получают кофе из отдельной машины. На рис. 17.1 показаны конкурентные и параллельные операции на примере бариста.



Существует также третья концепция — конкурентно-параллельные операции (*concurrent parallel operations*), когда два потока работают одновременно (параллельно), но при этом выполняют многозадачные индивидуальные потоки (конкурентно). Несмотря на полезность этой комбинированной концепции, в данной главе мы сосредоточимся лишь на двух основных типах.

Большинство распространенных PHP-приложений, будь то современные или старые версии, написаны как однопоточные. Такой код не является ни конкурентным, ни параллельным. Многие разработчики предпочитают избегать PHP, когда хотят реализовать концепции параллельного или конкурентного выполнения, и обращаются к таким языкам, как JavaScript или Go. Однако современный PHP поддерживает оба режима — как с помощью дополнительных библиотек, так и без них.



Конкурентный: две стойки обслуживаются одновременно одним бариста



Параллельный: две стойки обслуживают независимо друг от друга бариста и кофе-машина

Рис. 17.1. Конкурентный и параллельный режимы работы

## Библиотеки и среды выполнения

Поддержка параллельных и конкурентных операций появилась в PHP относительно недавно, и ее сложно применить на практике. Тем не менее несколько библиотек помогают построить по-настоящему асинхронные приложения.

### AMPHP

Проект AMPHP (<https://amphp.org/>) — это асинхронная библиотека PHP, основанная на обработке событий, который обеспечивает параллельное выполнение задач. AMPHP предлагает богатый набор функций и объектов, позволяющих полностью освоить асинхронность в PHP. В частности, AMPHP предоставляет полный цикл событий, а также эффективные абстракции для промисов, сопрограмм, асинхронных итераторов и потоков.

### ReactPHP

Подобно AMPHP, ReactPHP (<https://reactphp.org/>) — это устанавливаемая через Composer библиотека, предлагающая функциональность, основанную на событиях, и абстракции для PHP. Она предоставляет цикл событий, а также полнофункциональные асинхронные серверные компоненты, такие как клиент сокетов и DNS-резольвер.

## Open Swoole

Open Swoole (<https://openswoole.com/>) — это низкоуровневое расширение PHP, которое можно установить с помощью PECL. Как и AMPHP, и ReactPHP, Open Swoole предлагает асинхронный фреймворк и реализацию как промисов, так и сопрограмм. Поскольку Open Swoole — это скомпилированное расширение (а не библиотека PHP), его производительность значительно выше, чем у различных альтернатив. Оно также поддерживает истинный параллелизм в вашем коде, а не просто конкурентное выполнение задач.

## RoadRunner

RoadRunner (<https://roadrunner.dev/>) — это альтернативная среда выполнения PHP, реализованная на языке Go. Проект предоставляет тот же интерфейс PHP, к которому вы привыкли, но поставляется с собственным сервером приложений и асинхронным менеджером процессов. RoadRunner позволяет хранить все приложение в памяти и вызывать атомарные процессы параллельно выполнению приложения по мере необходимости.

## Octane

В 2021 году фреймворк для веб-приложений Laravel представил новый проект под названием Octane (<https://oreil.ly/bLnkA>), который использует Open Swoole или Roadrunner для «повышения производительности вашего приложения». В то время как такие инструменты на уровне фреймворка, как AMPHP или ReactPHP, позволяют вам намеренно писать асинхронный код, Octane использует асинхронные основы Open Swoole или RoadRunner для ускорения работы существующего приложения на базе Laravel<sup>1</sup>.

# Асинхронные операции

Чтобы полностью понять асинхронный PHP, вам нужно разобраться как минимум в двух специфических концепциях: промисах (promises) и сопрограммах (coroutines).

---

<sup>1</sup> Octane обещает повысить производительность большинства приложений без каких-либо преобразований в их коде. Тем не менее в производственной среде, скорее всего, будут возникать нестандартные ситуации, требующие изменений, поэтому тщательно тестируйте свой код, прежде чем полагаться на проект в качестве замены среды выполнения.

## Промисы

В программном обеспечении промис — это объект, возвращаемый функцией, которая работает асинхронно. Однако промис не представляет собой дискретное значение, а отражает общее состояние операции. Когда функция впервые возвращает промис, он не имеет собственного значения, поскольку сама операция еще не завершена. Вместо этого он будет находиться в состоянии ожидания, указывающем на то, что программе следует заняться чем-то другим, пока асинхронная операция завершается в фоновом режиме.

Когда операция завершится, промис будет либо выполнен, либо отклонен. Состояние «выполнено» означает, что все прошло успешно, при этом возвращается дискретное значение. Состояние «отклонено» возникает, если что-то пошло не так и вместо значения возвращается ошибка.

Проект AMPHP реализует промисы с помощью генераторов и связывает выполненное и отклоненное состояния в метод `onResolve()` на объекте промиса. Например:

```
function doSomethingAsync(): Обещание
{
    // ...
}

doSomethingAsync()->onResolve(function (Throwable $error = null, $value = null) {
    if ($error) {
        // ...
    } else {
        // ...
    }
});
```

В то же время проект ReactPHP реализует ту же спецификацию промисов, что и JavaScript (<https://oreil.ly/ZRwcW>). Это позволяет использовать конструкцию `then()`, которая может быть знакома программистам Node.js. Например:

```
function doSomethingAsync(): Promise
{
    // ...
}

doSomethingAsync()->then(function ($value) {
    // ...
}, function ($error) {
    // ...
});
```



Хотя API, представленные для промисов в AMPHP и ReactPHP, в некотором роде уникальны, они вполне совместимы. AMPHP явно не соответствует абстракциям промисов в стиле JavaScript, чтобы полностью задействовать генераторы PHP. Однако он принимает экземпляры PromiseInterface от ReactPHP везде, где работает с собственным экземпляром Promise.

Оба API невероятно мощные, и каждый из них предлагает эффективные асинхронные абстракции для PHP. Однако для удобства мы сосредоточимся на библиотеке AMPHP, так как она является более универсальным для PHP.

## Сопрограммы

Сопрограмма (coroutine) — это функция, выполнение которой можно приостановить для завершения другой операции. В PHP сопрограммы реализуются через генераторы с помощью ключевого слова `yield` для временной остановки работы<sup>1</sup>.

В то время как традиционный генератор задействует ключевое слово `yield` для возврата значения в итераторе, AMPHP использует то же самое ключевое слово в качестве прерывателя в сопрограмме. Значение по-прежнему возвращается, но выполнение самой сопрограммы прерывается, чтобы другие операции (например, другим сопрограммам) могли продолжить работу. Когда сопрограмма получает промис, она отслеживает его состояние и автоматически возобновляет выполнение, когда состояние меняется на «разрешено».

В качестве примера можно привести асинхронные запросы к серверу, выполняемые через сопрограммы напрямую в AMPHP. В следующем коде показано, как сопрограммы применяются для получения страницы и декодирования тела ее ответа, возвращая объект промиса, который можно использовать в других частях вашего кода:

```
$client = HttpClientBuilder::buildDefault();

$promise = Amp\call(function () use ($client) {
    try {
        $response = yield $client->request(new Request("https://eamann.com"));

        return $response->getBody()->buffer();
    } catch (HttpException $e) {
        // ...
    }
});

$promise->onResolve(function ($error = null, $value = null) {
    if ($error) {
```

---

<sup>1</sup> Подробнее о генераторах и ключевом слове `yield` читайте в рецепте 7.15.

```
// ...
} else {
    var_dump($value);
}
});
```

## Файбера

Новейшая функция параллелизма в PHP, начиная с версии 8.1, — это файбер (Fiber). Файбер представляет собой отдельный поток операций, который может контролироваться основным процессом вашего приложения. Файбер не работает параллельно с основным приложением, а создает отдельный стек выполнения с собственными переменными и состоянием.

С помощью файберов можно запускать совершенно независимые субприложения внутри основного приложения и явно контролировать, как обрабатывается каждая конкурентная операция.

Когда файбер запускается, он работает до тех пор, пока либо не завершит выполнение, либо не вызовет функцию `suspend()`, чтобы передать управление родительскому процессу (потоку) и вернуть ему значение. Затем он может быть перезапущен родительским процессом с помощью функции `resume()`. Приведенный ниже пример из официальной документации наглядно иллюстрирует эту концепцию:

```
$fiber = new Fiber(function (): void {
    $value = Fiber::suspend('fiber');
    echo "Значение, используемое для возобновления файбера: ", $value, "\n";
});

$value = $fiber->start();

echo "Значение от приостановки файбера: ", $value, "\n";

$fiber->resume('test');
```

Файбера не предназначены для использования непосредственно в коде, но служат низкоуровневым интерфейсом, полезным для таких фреймворков, как AMPHP и ReactPHP. Эти фреймворки способны задействовать файбера для полного абстрагирования среды выполнения сопрограмм, обеспечивая чистое состояние вашего приложения и эффективное управление его параллелизмом.

В следующих рецептах рассказывается о тонкостях работы с конкурентным и параллельным кодом в PHP. Вы узнаете, как управлять несколькими конкурентными запросами, структурировать асинхронные сопрограммы и использовать встроенную в PHP реализацию файбера.

## 17.1. Получение данных из удаленных API асинхронно

### Задача

Вы хотите получить данные с нескольких удаленных серверов одновременно и работать с результатом, когда все они вернут данные.

### Решение

Используйте модуль `http-client` из проекта AMPHP, чтобы выполнять несколько одновременных запросов в виде отдельных промисов, а затем действовать в зависимости от результата запроса. Например:

```
use Amp\Http\Client\HttpClientBuilder;
use Amp\Http\Client\Request;

use function Amp\Promise\all;
use function Amp\Promise\wait;

$client = HttpClientBuilder::buildDefault();
$promises = [];

$apiUrls = ['\https://github.com', '\https://gitlab.com', '\https://bitbucket.org'];

foreach($apiUrls as $url) {
    $promises[$url] = Amp\call(static function() use ($client, $url) {
        $request = new Request($url);
        $response = yield $client->request($request);
        $body = yield $response->getBody()->buffer();
        return $body;
    });
}

$responses = wait(all($promises));
```

### Обсуждение

В типичном синхронном PHP-приложении ваш HTTP-клиент делал бы один запрос за раз и ждал ответа сервера, прежде чем продолжить работу. Эта последовательная схема достаточно быстра для большинства реализаций, но становится обременительной при управлении большим количеством запросов одновременно.

Модуль `http-client` фреймворка AMPHP позволяет отправлять конкурентные запросы<sup>1</sup>. Все запросы выполняются в неблокируемом режиме с помощью промисов, которые передают состояние запроса и конечный результат. Прелесть такого подхода заключается не только в возможности конкурентного выполнения запросов клиентом AMPHP, но и в обертке `Amp\call()`, применяемой для объединения всех запросов вместе.

Обернув анонимную функцию с помощью `Amp\call()`, вы превращаете ее в сопрограмму<sup>2</sup>. Ключевое слово `yield` в теле сопрограммы указывает ей ждать ответа асинхронной функции. Общий результат работы сопрограммы возвращается в виде экземпляра `Promise`, а не скалярного значения. В примере выше сопрограмма генерирует новый экземпляр `Promise` для каждого запроса API и хранит их вместе в одном массиве.

Фреймворк AMPHP предоставляет две полезные функции, позволяющие дождаться разрешения всех ваших промисов: `all()` и `wait()`.

- `all()` принимает массив промисов и возвращает один новый промис, который будет разрешен только тогда, когда будут разрешены все промисы в исходном массиве. Значение, обернутое этим новым промисом, станет массивом значений его промисов.
- `wait()` фактически преобразует асинхронный код в синхронный, так что выполнение программы приостанавливается до тех пор, пока не завершится асинхронная операция, связанная с переданным промисом. Когда промис разрешен, функция `wait()` вернет значение, которое было разрешено промисом.

Таким образом, в примере из «Решения» выполняется несколько конкурентных асинхронных запросов к различным API, а затем их ответы объединяются в массив, пригодный для использования в остальной части вашего синхронного приложения.



Запросы могут быть выполнены в любом порядке, а не строго в том, в котором вы их отправили. Если вы увеличите количество запросов, результирующий массив может иметь порядок, отличный от ожидаемого. Чтобы в дальнейшем не удивляться, что ответы API изменили порядок следования, полезно отслеживать дискретный индекс (например, использовать ассоциативный массив).

## Читайте также

Документация по модулю `http-client` из проекта AMPHP (<https://oreil.ly/OUE0n>).

<sup>1</sup> Как и в случае с любым модулем и самим фреймворком AMPHP, вы можете установить пакет `http-client` с помощью Composer. Дополнительные сведения о пакетах Composer см. в рецепте 15.3.

<sup>2</sup> Подробнее об анонимных функциях, или лямбдах, читайте в рецепте 3.9.

## 17.2. Ожидание результатов нескольких асинхронных операций

### Задача

Вы хотите выполнить несколько параллельных операций, а уже потом действовать на основе общего результата всех этих операций.

### Решение

Воспользуйтесь модулем `parallel-functions` фреймворка AMPHP, чтобы выполнять операции действительно параллельно, а затем работать с конечным ответом всей коллекции операций, как показано в примере 17.1.

**Пример 17.1.** Пример параллельного отображения массива

```
use Amp\Promise;
use function Amp\ParallelFunctions\parallelMap;

$values = Promise\wait(parallelMap([3, 1, 5, 2, 6], function ($i) {
    echo "Сон в течение {$i} секунд." . PHP_EOL; ❶
    \sleep($i); ❷
    echo "Спал {$i} секунд." . PHP_EOL; ❸
    return $i ** 2; ❹
}));

print_r($values); ❺
```

❶ Первый оператор `echo` используется просто для демонстрации порядка выполнения операции параллельного отображения. Вы увидите сообщения в консоли в том же порядке, в котором массив изначально был передан в `parallelMap()`, а именно: [3, 1, 5, 2, 6].

❷ Функция `sleep()` является блокирующей, то есть она приостанавливает выполнение вашей программы до истечения заданного количества секунд. Этот вызов функции можно заменить любой другой блокирующей операцией с аналогичным эффектом. Цель данного примера — показать, что каждая операция действительно выполняется параллельно.

❸ После завершения ожидания, вызванного функцией `sleep()`, приложение снова выведет сообщение, демонстрирующее порядок завершения параллельных операций. Обратите внимание, что этот порядок будет отличаться от того, в котором

операции были вызваны изначально! В частности, числа отобразятся в порядке возрастания из-за времени, необходимого до завершения каждого вызова `sleep()`.

- ❸ Любое возвращаемое значение из вашей функции в итоге будет обернуто в объект `Promise` до тех пор, пока асинхронная операция не завершится.
- ❹ За пределами `Promise` и функции `wait()` все собранные вами промисы будут разрешены, а в конечной переменной окажется скалярное значение. В данном случае конечная переменная предстанет в виде массива квадратных значений в том же порядке, что и исходные входные данные.

## Обсуждение

Модуль `parallel-functions` — это фактически слой абстракции над модулем `parallel` от AMRPHP. Оба модуля можно установить через Composer. Ни один из них не требует специальных расширений для работы. Тем не менее они позволят вам выполнять настоящие параллельные операции в PHP.

Без каких-либо расширений `parallel` порождает дополнительные процессы PHP для обработки ваших асинхронных операций. Он обрабатывает создание и сбор дочерних процессов за вас, чтобы вы могли сосредоточиться на фактической реализации вашего кода. В системах с расширением `parallel` (<https://oreil.ly/kW0n5>) библиотека попытается задействовать более легкие потоки для размещения вашего приложения.

Но в любом случае ваш код будет выглядеть одинаково. Независимо от того, использует ли система процессы или потоки, AMRPHP скроет это от вас. Это позволит вам написать приложение, которое просто применяет абстракции уровня `Promise` и полагается на то, что все будет работать как надо.

В примере 17.1 вы определили функцию, содержащую несколько ресурсозатратных блокирующих вызовов ввода-вывода. Здесь специально использовалась функция `sleep()`, хотя это мог быть и удаленный вызов API, и дорогостоящая операция хеширования, и длительный запрос к БД. В любом случае, это тот тип функции, который заморозит ваше приложение до завершения, если потребуется, не один раз.

Вместо того чтобы прибегать к синхронному коду, когда вы передаете в функцию каждый элемент коллекции по очереди, вы можете использовать фреймворк AMRPHP для одновременной обработки нескольких вызовов.

Функция `parallelMap()` ведет себя аналогично встроенной в PHP функции `array_map()`, за исключением того, что она применяется к каждому элементу массива параллельно (и с аргументами в обратном порядке)<sup>1</sup> в отдельном процессе или

---

<sup>1</sup> Подробнее о функции `array_map()` читайте в рецепте 7.13.

потоке. Поскольку сама операция является асинхронной, `parallelMap()` возвращает `Promise`, чтобы обернуть конечный результат функции.

В итоге у вас остается массив промисов, представляющих отдельные параллельные операции, выполняемые в фоновом режиме. Чтобы вернуться к синхронному коду, воспользуйтесь функцией `wait()` в AMPHP, как в рецепте 17.1.

## Читайте также

Документация по модулям `parallel` (<https://oreil.ly/6Um1H>) и `parallel-functions` (<https://oreil.ly/8Qffs>) из фреймворка AMPHP.

### 17.3. Прерывание одной операции для выполнения другой

#### Задача

Вы хотите выполнить две независимые операции и перемещаться между ними на одном потоке.

#### Решение

Используйте сопрограммы фреймворка AMPHP, как показано в примере 17.2.

##### Пример 17.2. Параллельные циклы for с сопрограммами

```
use Amp\Delayed;
use Amp\Loop;
use function Amp\asyncCall;

asyncCall(function () {
    for ($i = 0; $i < 5; $i++) { ❶
        print "Цикл А - " . $i . PHP_EOL;
        yield new Delayed(1000); ❷
    }
});

asyncCall(function () {
    for ($i = 0; $i < 5; $i++) { ❸
        print "Цикл В - " . $i . PHP_EOL;
        yield new Delayed(400); ❹
    }
});

Loop::run(); ❺
```

- ❶ Первый цикл просто считает от 0 до 4, каждый раз увеличивая шаг на 1.
- ❷ Объект `Delayed()` фреймворка AMPHP — это промис, который разрешается сам по себе через заданное время (в данном случае через секунду).
- ❸ Второй цикл также считает от 0 до 4 с шагом 1.
- ❹ Промис второго цикла разрешается через 0,4 секунды.
- ❺ Оба вызова `asyncCall()` сработают немедленно и выведут на экран 0. Однако циклы не возобновятся до тех пор, пока цикл событий не будет запущен официально (чтобы промисы `Delayed` могли разрешиться).

## Обсуждение

Выше вводятся два понятия, важных в асинхронном программировании PHP: цикл событий и сопрограммы.

Цикл событий — это основа того, как AMPHP обрабатывает конкурентные операции. Без цикла событий PHP пришлось бы выполнять ваше приложение или скрипт строго сверху вниз. Однако цикл событий дает интерпретатору возможность вернуться и выполнить дополнительный код иным способом. В частности, функция `Loop::run()` продолжит выполняться, пока цикл событий не обработает все или пока само приложение не получит сигнал SIGINT (например, после нажатия Ctrl+C на клавиатуре).

В рамках фреймворка AMPHP существуют две функции, создающие сопрограммы: `call()` и `asyncCall()`. Обе они немедленно вызывают переданный им обратный вызов: `call()` возвращает экземпляр `Promise`, а `asyncCall()` — нет. Внутри обратного вызова любое использование ключевого слова `yield` создает сопрограмму — блок кода, который в нужный момент приостанавливает свое выполнение, чтобы запустился другой код. Сопрограмма возобновляет свою работу после разрешения объекта `Promise`.

В примере из «Решения» этим объектом является `Delayed`. Сопрограмма — это способ AMPHP заставить процедуру приостановить выполнение, подобно `sleep()`. Однако, в отличие от `sleep()`, объект `Delayed` считается неблокируемым. По сути, он «уснет» на определенное время, а затем возобновит работу при следующем проходе по циклу событий. Пока процедура отложена (или «спит»), PHP способен выполнять другие операции.

Запустив пример из «Решения» в консоли PHP, вы получите следующий результат:

```
% php concurrent.php
Цикл А – 0
Цикл В – 0
```

```
Цикл В – 1
Цикл В – 2
Цикл А – 1
Цикл В – 3
Цикл В – 4
Цикл А – 2
Цикл А – 3
Цикл А – 4
```

Предыдущий вывод демонстрирует, что PHP нет необходимости ждать завершения одного цикла перед запуском другого. Оба цикла выполняются одновременно.

Обратите внимание, что при синхронном выполнении этих двух циклов весь сценарий занял бы не менее 7 секунд (первый цикл ожидает 1 секунду для каждого из пяти проходов, а второй цикл занимает по 0,4 секунды на каждую итерацию). Одновременный запуск этих циклов длится всего 5 секунд. Чтобы полностью продемонстрировать это, сохраните значение `microtime(true)` в переменной при запуске процесса и сравните с системным временем после завершения цикла. Например:

```
use Amp\Delayed;
use Amp\Loop;
use function Amp\asyncCall;

$start = microtime(true);

// ...

Loop::run();

$time = microtime(true) - $start;
echo "Выполнен за {$time} секунд" . PHP_EOL;
```

Создание цикла событий влечет за собой незначительные издержки, но повторный запуск примера из «Решения» с предыдущими изменениями гарантированно даст результат около 5 секунд в сумме. Более того, можно увеличить счетчик цикла во втором вызове `asyncCall()` с 5 до 10. Данный цикл по-прежнему займет около 4 секунд. Опять же, при синхронном выполнении оба цикла потребовали бы 9 секунд, но благодаря сопрограммам сценарий все равно завершится примерно за 5 секунд. На рис. 17.2 показана разница между синхронным и конкурентным выполнением.

Обрабатывая два отдельных цикла как сопрограммы в цикле событий AMPHP, PHP может прервать работу одного из них, чтобы продолжить выполнение другого. Переключаясь между сопрограммами, PHP способен максимально эффективно нагрузить ваш процессор и позволить вашему приложению завершить свою работу быстрее, чем если бы оно выполняло свою логику синхронно.

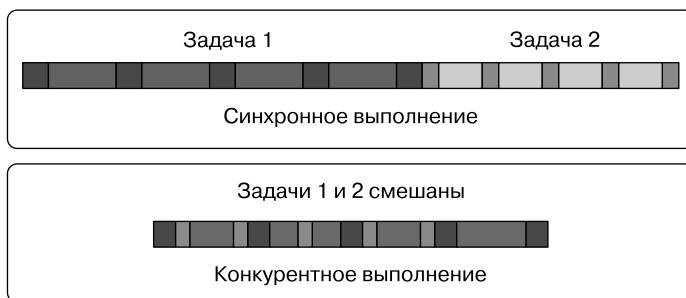


Рис. 17.2. Одновременное выполнение двух сопрограмм

Рассмотренный нами пример с использованием задержек или пауз — это выдумка, призванная продемонстрировать, что бывает, когда вы используете неблокирующий, но медленный процесс. Можно сделать сетевой запрос и применить сопрограмму, чтобы приложение продолжало обработку, пока ждет завершения запроса. Достаточно обратиться к базе данных или другому слою хранения данных и поместить неблокируемый вызов в сопрограмму. В некоторых системах разрешается запускать другие процессы (например, Sendmail или другой системный процесс) без блокирования этими вызовами общего выполнения приложения.

## Читайте также

Документация по функции `asyncCall()` (<https://oreil.ly/8QfFs>) фреймворка AMPHP и по сопрограммам в целом (<https://oreil.ly/oC2oW>).

# 17.4. Выполнение кода в отдельном потоке

## Задача

Вы хотите запустить одну или несколько тяжелых операций в отдельном потоке, чтобы основное приложение могло сообщать о ходе выполнения.

## Решение

Используйте пакет `parallel` проекта AMPHP, чтобы определить задачу `Task`, которую необходимо выполнить, и экземпляров `Worker` для ее запуска. Затем вызовите один или несколько так называемых рабочих (`workers`)<sup>1</sup> в виде отдельных потоков

<sup>1</sup> Worker в многопоточном программировании — это поток (процесс, который может выполняться параллельно с другими процессами), который обычно создается для выполнения определенной задачи или группы задач. — Примеч. ред.

или процессов. В примере 17.3 массив значений сокращается до одного выходного значения путем рекурсивного использования однонаправленного хеша. Для этого хеш-операция оборачивается в асинхронную задачу Task, предназначенную для выполнения в составе рабочего пула. Затем, как показано в примере 17.4, определяется пул рабочих, которые выполняют несколько операций Task в отдельных потоках, обернутых в сопрограмму.

**Пример 17.3.** Сокращения массива до одного значения при помощи рекурсивного хеша

```
class Reducer implements Task
{
    private $array;
    private $preHash;
    private $count;

    public function __construct(
        array $array,
        string $preHash = '',
        int $count = 1000)
    {
        $this->array = $array;
        $this->preHash = $preHash;
        $this->count = $count;
    }

    public function run(Environment $environment)
    {
        $reduction = $this->preHash;

        foreach($this->array as $item) {
            $reduction = hash_pbkdf2('sha256', $item, $reduction, $this->count);
        }

        return $reduction;
    }
}
```

**Пример 17.4.** Рабочий пул может выполнять несколько задач

```
use Amp\Loop;
use Amp\Parallel\Worker\DefaultPool;

$results = [];

$tasks = [
    new Reducer(range('a', 'z'), count: 100),
    new Reducer(range('a', 'z'), count: 1000),
    new Reducer(range('a', 'z'), count: 10000),
    new Reducer(range('a', 'z'), count: 100000),
    new Reducer(range('A', 'Z'), count: 100),
```

```
new Reducer(range('A', 'Z'), count: 1000),
new Reducer(range('A', 'Z'), count: 10000),
new Reducer(range('A', 'Z'), count: 100000),
];

Loop::run(function () use (&$results, $tasks) {
    require_once DIR . '/vendor/autoload.php';
    use PhpAmqpLib\Connection\AMQPStreamConnection;
    use PhpAmqpLib\Message\AMQPMessage;
    $timer = Loop::repeat(200, function () {
        printf('.');
    });
    Loop::unreference($timer);

    $pool = new DefaultPool;

    $coroutines = [];

    foreach ($tasks as $index => $task) {
        $coroutines[] = Amp\call(function () use ($pool, $index, $task) {
            $result = yield $pool->enqueue($task);

            return $result;
        });
    }
    $results = yield Amp\Promise\all($coroutines);

    return yield $pool->shutdown();
});

echo PHP_EOL . 'Результаты хеширования:' . PHP_EOL;
echo var_export($results, true) . PHP_EOL;
```

## Обсуждение

Преимущество параллельной обработки заключается в том, что вы больше не ограничены выполнением одной операции за раз. Современные многоядерные компьютеры способны выполнять более одной независимой операции одновременно. В PHP эта функциональность реализована через модуль `parallel` фреймворка AMPHP<sup>1</sup>.

В примере выше эта абстракция используется для параллельной обработки нескольких хеш-значений, позволяя родительскому приложению лишь сообщать о прогрессе и конечном результате. Первый компонент, класс `Reducer`, принимает

<sup>1</sup> Фреймворк AMPHP также предоставляет пакет `parallel-functions`, который раскрывает несколько полезных вспомогательных функций, обернутых в низкоуровневый пакет `parallel`. Подробнее об этих функциях и их использовании читайте в рецепте 17.2.

массив строк и создает итеративный хеш этих значений. Конкретно, он выполняет определенное количество операций хешей выведенных ключей на основе пароля для каждого значения в массиве, передавая результат вывода в операцию хеширования для следующего элемента массива.



Операции хеширования предназначены для быстрого преобразования известного значения в кажущееся случайным. Это односторонние операции, то есть вы можете легко перейти от начального значения к хешу, но обратное преобразование хеша для получения его начального значения практически невозможно. В некоторых системах обеспечения безопасности используется несколько этапов хеширования — во многих случаях десятки тысяч, — чтобы явно замедлить процесс и предотвратить атаки типа Guess-and-Check, основанные на попытке угадать начальное зерно.

Поскольку эти операции хеширования затратны (в плане времени), вы не захотите выполнять их синхронно. Учитывая, сколько времени они могут занимать, не стоит даже запускать их конкурентно. Единственный оптимальный вариант — работать с ними в полностью параллельном режиме, чтобы задействовать все доступные ядра процессора. Поместив операцию в объект, расширяющий Task, они могут выполняться одновременно при вызове в пуле потоков.

Пакет AMRHP `parallel` предоставляет пул потоков с конфигурацией по умолчанию, где вы легко сможете поместить в пул столько операций, сколько захотите, если они реализуют Task. Пул вернет экземпляр промиса, обрабатывающего задачу, а это значит, что вы можете добавлять свои задачи в сопрограммы и ожидать разрешения всех промисов, которые они представляют.

Поскольку все операции асинхронны, родительское приложение продолжит выполнять код, пока хеширование происходит параллельно. В примере из «Решения» используется это преимущество, путем настройки повторяющейся операции `printf()` для вывода на экран десятичной точки каждые 200 миллисекунд. Это в некотором роде напоминает индикатор прогресса или проверку активности, предоставляя подтверждение того, что параллельный процесс все еще выполняется.

После завершения всех параллельных задач по хешированию общая операция выводит результаты на экран.

На самом деле, вы можете добавить в очередь любой параллельный процесс таким образом, чтобы выполнять несколько задач одновременно. В AMRHP есть функция `enqueueCallable()`, которая позволяет превратить любой вызов обычной функции в параллельную операцию. Допустим, вам нужно получить сводки погоды из Национальной метеорологической службы США (NWS). Вместо того чтобы вызывать несколько заданий хеширования, как в примере из «Решения», можно воспользоваться способом, показанным в примере 17.5.

### Пример 17.5. Асинхронное получение метеорологических сводок

```
use Amp\Parallel\Worker;
use Amp\Promise;

$forecasts = [
    'Вашингтон' => 'https://api.weather.gov/gridpoints/LWX/97,71/forecast',
    'Нью-Йорк'   => 'https://api.weather.gov/gridpoints/OKX/33,37/forecast',
    'Туалатин'   => 'https://api.weather.gov/gridpoints/PQR/108,97/forecast',
];

$promises = [];
foreach ($forecasts as $city => $forecast) {
    $promises[$city] = Worker\enqueueCallable('file_get_contents', $forecast); ❶
}

$responses = Promise\wait(Promise\all($promises)); ❷

foreach($responses as $city => $forecast) {
    $forecast = json_decode($forecast); ❸
    $latest = $forecast->properties->periods[0];

    echo "Прогноз погоды для города {$city}:" . PHP_EOL;
    print_r($latest);
}
```

❶ Каждая конечная точка URL может быть получена независимо с помощью функции `file_get_contents()`. Функция AMPHP `enqueueCallable()` автоматически сделает это в рамках независимого процесса параллельно с основным приложением.

❷ Каждый параллельный запрос обернут в объект `Promise`. Чтобы вернуться к синхронному выполнению, вы должны дождаться, пока все эти промисы будут разрешены. Функция `all` собирает различные промисы в один объект `Promise`. Функция `wait()` блокирует выполнение до тех пор, пока этот промис не будет разрешен. Затем она извлекает содержащееся в нем значение для использования в вашем синхронном коде.

❸ API NWS возвращает объект JSON, представляющий прогноз по конкретной метеостанции. Прежде чем вы сможете использовать эти данные в своем приложении, необходимо обработать строку в кодировке JSON.



Погодный API NWS совершенно бесплатен, но для отправки запроса требует указать уникальный агент пользователя. По умолчанию PHP отправляет простую строку агента пользователя PHP при использовании `file_get_contents()`. Чтобы изменить это, настройте конфигурацию `user_agent` в вашем файле `php.ini`. Без этого данный API, скорее всего, отклонит ваш запрос с ошибкой 403 Forbidden. Для получения дополнительной информации обратитесь к разделу FAQ по API (<https://oreil.ly/4WVIO>).

Использует ли фреймворк AMPHP отдельные потоки или полностью автономные процессы, зависит от того, как изначально настроена ваша система. Ваш код остается неизменным и при отсутствии каких-либо расширений, поддерживающих многопоточный PHP, скорее всего, по умолчанию будет работать с созданными процессами PHP. В любом случае функция `enqueueCallable()` требует от вас либо родную функцию PHP, либо пользовательскую функцию, загружаемую через Composer. Это связано с тем, что запущенный дочерний процесс знает только о системных функциях, функциях, загруженных через Composer, и любых сериализованных данных, переданных родительским процессом.

Эта последняя деталь очень важна: данные, которые вы отправляете из родительского приложения в фоновый рабочий процесс, будут сериализованы. Некоторые пользовательские объекты могут сломаться, когда PHP попытается их сериализовать и десериализовать. Даже некоторые ключевые объекты (например, контексты потоков) нельзя передать в дочерний поток или процесс, так как они несовместимы с сериализацией.

Внимательно следите за тем, какие задачи вы запускаете в фоновом режиме. Данные, которые вы отправляете, должны быть совместимы с сериализацией и параллельными операциями.

## Читайте также

Документация по параллельному пакету (<https://oreil.ly/C41Rb>) из фреймворка AMPHP.

## 17.5. Пересылка сообщений между отдельными потоками

### Задача

Вы хотите взаимодействовать с несколькими запущенными потоками, чтобы синхронизировать состояние или управлять задачами, которые эти потоки выполняют.

### Решение

Используйте очередь сообщений или шину между вашим приложением и отдельными потоками, которые оно организует, чтобы обеспечить бесперебойную связь. Например, RabbitMQ может выступать в качестве посредника между вашим

основным приложением (см. пример 17.7) и независимыми рабочими потоками, как показано в примере 17.6.

**Пример 17.6.** Фоновая задача, используемая для отправки почты на основе очереди

```
use PhpAmqpLib\Connection\AMQPStreamConnection;

$connection = new AMQPStreamConnection('127.0.0.1', 5762, 'guest', 'guest'); ❶

$channel = $connection->channel();
$channel->queue_declare('default', false, false, false); ❷

echo '... Ожидание сообщений. Для выхода нажмите CTRL+C' . PHP_EOL;
$callback = function($msg) {
    $data = json_decode($msg->body, true); ❸
    $to = $data['to'];
    $from = $data['from'] ?? 'worker.local';
    $subject = $data['subject'];
    $message = wordwrap($data['message'], 70) . PHP_EOL;

    $headers = "From: {$from} PHP_EOL X-Mailer: PHP Worker";

    print_r([$to, $subject, $message, $headers]) . PHP_EOL; ❹

    mail($to, $subject, $message, $headers);

    $msg->ack(); ❺
};

$channel->basic_consume('default', '', false, false, false, false,
$callback); ❻
while(count($channel->callbacks)) {
    $channel->wait(); ❻
}
```

❶ Откройте соединение с локально запущенным сервером RabbitMQ с помощью стандартных учетных данных и порта. В эксплуатационной среде эти значения отличаются, поэтому должны быть загружены из самого окружения.

❷ Объявление очереди на сервере RabbitMQ просто открывает канал связи. Если очередь уже существует, эта операция ничего не дает.

❸ Когда данные поступают в рабочий поток из RabbitMQ, они оборачиваются в объект сообщения. Фактические данные, которые вам нужны, находятся в теле сообщения.

❹ Вывод данных в рабочем потоке — это полезный способ диагностики и проверки поступающих данных на наличие потенциальных ошибок.

- ❸ Когда рабочий поток завершает обработку сообщения, ему необходимо подтвердить получение сообщения на сервере RabbitMQ, в противном случае другой рабочий поток может «забрать» сообщение и повторить его позже.
- ❹ Получение сообщений — это синхронная операция. Когда сообщение поступает из RabbitMQ, система активирует обратный вызов, переданный этой функции, с самим сообщением в качестве аргумента.
- ❺ Пока есть обратные вызовы на сообщение, данный цикл будет выполняться вечно, а метод `wait()` — держать соединение с RabbitMQ открытым, чтобы рабочий поток мог принимать и обрабатывать любые сообщения в очереди.

### Пример 17.7. Основное приложение, отправляющее сообщения в очередь

```
use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;

$connection = new AMQPStreamConnection('127.0.0.1', 5672, 'guest', 'guest'); ❶

$channel = $connection->channel();
$channel->queue_declare('default', false, false, false); ❷

$message = [
    'subject' => 'Добро пожаловать в команду!',
    'from'      => 'admin@mail.local',
    'message'   => "Добро пожаловать в команду!\r\nМы рады видеть вас здесь!"
];

$teammates = [
    'adam@eden.local',
    'eve@eden.local',
    'cain@eden.local',
    'abel@eden.local',
];
}

foreach($teammates as $employee) {
    $email = $message;
    $email['to'] = $employee;
    $msg = new AMQPMessage(json_encode($email)); ❸
    $channel->basic_publish($msg, '', 'default'); ❹
}

$channel->close(); ❽
$connection->close();
```

- ❶ Как и в случае с рабочим потоком, вы открываете соединение с локальным сервером RabbitMQ, используя параметры по умолчанию.

- ❷ Затем объявляете очередь. Если очередь уже существует, вызов метода ничего не даст.
- ❸ Прежде чем отправить сообщение, его нужно закодировать. В нашем случае данные буду сериализованы в виде JSON-строки.
- ❹ Для каждого сообщения вы выбираете очередь, в которой оно будет опубликовано, и отправляете его в RabbitMQ.
- ❺ После того как вы отправили сообщения, лучше явно закрыть канал и соединение, прежде чем выполнять какие-либо иные действия. В текущем примере нет другой работы (и процесс завершится немедленно), но явная очистка ресурсов — полезная привычка для любого разработчика.

## Обсуждение

В примере из «Решения» используется несколько явных процессов PHP для обработки сложных операций. Скрипт, определенный в примере 17.6, может быть назван `worker.php` и инстанцирован несколько раз. Если сделать это в двух различных консолях, запустятся два независимых PHP-процесса, которые подключаются к RabbitMQ и прослушивают задания.

Выполнение кода из примера 17.7 в третьем окне запустит главный процесс и отправит задания в очередь по умолчанию, созданную RabbitMQ. Рабочие потоки будут самостоятельно забирать эти задания, обрабатывать их и ждать дальнейших указаний.

Полноценное взаимодействие между родительским процессом (пример 17.7) и двумя асинхронными рабочими процессами (пример 17.6), где RabbitMQ выступает в качестве брокера сообщений, представлено в виде трех независимых консольных окон на рис. 17.3.

Различные процессы не взаимодействуют напрямую. Для этого вам потребуется создать интерактивный API. Однако более простым средством связи служит промежуточный брокер сообщений — в нашем случае RabbitMQ (<https://oreil.ly/GtgI0>).

RabbitMQ — это инструмент с открытым исходным кодом, совместимый со многими языками программирования. Он позволяет создавать множество очередей, доступных для чтения одним или более выделенными рабочими потоками для обработки содержимого сообщения. В примере из «Решения» вы использовали рабочие потоки и встроенную функцию PHP `mail()` для отправки электронных писем. Более сложный рабочий поток способен обновлять записи в базе данных, взаимодействовать с удаленным API или даже выполнять такие ресурсоемкие операции, как хеширование, описанное в рецепте 17.4.



Поскольку RabbitMQ поддерживает несколько языков, ваша реализация не ограничивается только PHP. Если есть определенная библиотека, которую вы хотите применить на другом языке, вы можете написать рабочие потоки на этом языке, импортировать библиотеку и отправлять задания из вашего основного PHP-приложения.

```

php
ericmann@Eric-Mann-MBP16tb-5 ~ % php worker.php
... Waiting for messages. To exit press CTRL+C
Array
(
    [0] => mickey.mouse@disney.local
    [1] => Welcome to the team!
    [2] => Welcome to the team!
    We're excited to have you here!
    [3] => From: admin@mail.local
X-Mailer: PHP Worker
)
Array
(
    [0] => donald.duck@disney.local
    [1] => Welcome to the team!
    [2] => Welcome to the team!
    We're excited to have you here!
    [3] => From: admin@mail.local
X-Mailer: PHP Worker
)
[]

php
ericmann@Eric-Mann-MBP16tb-5 ~ % php worker.php
... Waiting for messages. To exit press CTRL+C
Array
(
    [0] => minnie.mouse@disney.local
    [1] => Welcome to the team!
    [2] => Welcome to the team!
    We're excited to have you here!
    [3] => From: admin@mail.local
X-Mailer: PHP Worker
)
Array
(
    [0] => daisey.duck@disney.local
    [1] => Welcome to the team!
    [2] => Welcome to the team!
    We're excited to have you here!
    [3] => From: admin@mail.local
X-Mailer: PHP Worker
)
[]

-zsh
ericmann@Eric-Mann-MBP16tb-5 ~ % php processor.php
ericmann@Eric-Mann-MBP16tb-5 ~ %

```

**Рис. 17.3.** Несколько процессов PHP, работающих через RabbitMQ

В эксплуатационной среде ваш сервер RabbitMQ будет использовать аутентификацию по логину/паролю или же давать доступ только определенным авторизованным серверам. Однако для разработки можно задействовать локальное окружение, стандартные учетные данные и такие инструменты, как Docker (<https://www.docker.com/>), чтобы запустить сервер RabbitMQ на своей машине. Чтобы напрямую запустить RabbitMQ с помощью порта по умолчанию и стандартной авторизации, выполните следующую команду Docker:

```
$ docker run -d -h localhost -p 127.0.0.1:5672:5672 --name rabbit rabbitmq:3
```

После загрузки сервера вы сможете зарегистрировать столько очередей, сколько необходимо для управления потоком данных в приложении.

## Читайте также

Официальная документация (<https://oreil.ly/einsN>) и руководства (<https://oreil.ly/lEqc9>) по настройке и взаимодействию с RabbitMQ.

# 17.6. Использование файбера для управления содержимым потока

## Задача

Вы хотите применить новейшую функцию параллелизма PHP, чтобы извлекать данные из потока по частям и работать с ним, а не буферизировать все его содержимое сразу.

## Решение

Используйте файбер, чтобы обернуть поток и считывать его содержимое по одному фрагменту за раз. В примере 17.8 веб-страница считывается в файл 50-байтовыми фрагментами с отслеживанием общего количества байтов, потребляемых по мере чтения содержимого.

**Пример 17.8.** Чтение удаленного потока через файбер по одному фрагменту за раз

```
$fiber = new Fiber(function($stream): void {
    while (!feof($stream)) {
        $contents = fread($stream, 50); ①
        Fiber::suspend($contents); ②
    }
});

$stream = fopen('https://www.eamann.com/', 'r');
stream_set_blocking($stream, false); ③

$output = fopen('cache.html', 'w'); ④

$contents = $fiber->start($stream); ⑤

$num_bytes = 0;
while (!$fiber->isTerminated()) {
    echo chr(27) . "[0G"; ⑥

    $num_bytes += strlen($contents);
    fwrite($output, $contents); ⑦
```

```
echo str_pad("Записано {$num_bytes} байт ...", 24, ' ', STR_PAD_RIGHT);
usleep(500); ❸

$contents = $fiber->resume(); ❹
}

echo chr(27) . "[0G";
echo "Запись {$num_bytes} байт в cache.html завершена!" . PHP_EOL;

fclose($stream); ❽
fclose($output);
```

- ❶ При запуске файбер сам принимает потоковый ресурс в качестве единственного параметра. Пока поток существует, файбер будет считывать по 50 байт.
- ❷ После считывания данных из потока файбер приостановит работу и передаст управление обратно в стек родительского приложения.
- ❸ В родительском стеке приложения поток открывается и настраивается так, чтобы не блокировать выполнение остальной части приложения. В неблокирующем режиме любые вызовы функции `fread()` будут возвращаться сразу, а не ждать данных в потоке.
- ❹ В родительском приложении разрешается открывать и другие ресурсы, например локальные файлы, в которые можно кэшировать содержимое удаленного ресурса.
- ❺ При запуске файбера вы передаете ресурс главного потока в качестве параметра, чтобы он был доступен в стеке вызовов самого файбера. Когда файбер приостановит выполнение, он также вернет вам 50 байт, которые прочитал.
- ❻ Чтобы записать данные поверх предыдущей строки консольного вывода, перейдите символ `ESC` (`chr(27)`) и управляющую последовательность `ANSI` для перемещения курсора в первый столбец в терминале (`[0G]`). Теперь любой последующий текст, выводимый на экран, будет перезаписывать все ранее отображенное.
- ❼ После получения данных из удаленного потока вы можете записать их непосредственно в локальный файл кэша.
- ❽ Вызов `sleep` не является обязательным здесь, но он демонстрирует, как в стеке родительского приложения происходят другие вычисления, пока файбер приостановлен.
- ❾ При возобновлении работы файбер получит следующие 50 байт из удаленного потокового ресурса, если, конечно, в нем что-то осталось. Если извлекать нечего, файбер завершится, а ваша программа выйдет из цикла `while`.
- ❿ После завершения выполнения и очистки файбера не забудьте закрыть все открытые потоки или другие ресурсы.

## Обсуждение

Файбера похожи на сопрограммы и генераторы тем, что их выполнение может быть приостановлено, чтобы приложение могло приступить к другой работе до возобновления предыдущей. В отличие от иных конструкций, файбера имеют собственные стеки вызовов, независимые от стека остальной части приложения. Это позволяет приостанавливать их выполнение даже внутри вложенных вызовов функций без изменения типа возвращаемого значения функции, по вине которой возникла пауза.

Если генератор временно прерывает работу с помощью команды `yield`, вы должны вернуть экземпляр `Generator`. В случае файбера, использующего метод `::suspend()`, разрешается вернуть любой тип данных.

Возобновить работу файбера после его приостановки можно из любого места в родительском приложении, чтобы перезапустить его отдельный стек вызовов. Это позволяет эффективно переключаться между несколькими контекстами выполнения, не слишком заботясь о контроле состояния приложения.

Вы также можете эффективно обмениваться данными с файбером. Когда файбер находится в режиме ожидания, он способен отправить данные обратно в родительское приложение — опять же, любого типа, который вам нужен. Когда вы возобновляете работу файбера, вы вправе передать любое значение или не передавать его вовсе. У вас также есть возможность отправить исключение в файбер с помощью метода `::throw()`, а затем обработать это исключение внутри файбера. В примере 17.9 продемонстрировано, как будет выглядеть подобная обработка исключения.

### Пример 17.9. Обработка исключения внутри файбера

```
$fiber = new Fiber(function(): void {
    try {
        Fiber::suspend(); ❶
    }
    catch (Exception $e) {
        echo $e->getMessage() . PHP_EOL; ❷
    }

    echo 'Завершено в пределах файбера' . PHP_EOL; ❸
});

$fiber->start(); ❹
$fiber->throw(new Exception('Error'));
```

❶ После запуска файбер немедленно приостановит выполнение и вернет управление в стек родительского приложения.

❷ Если файбер при возобновлении работы поймает исключением `Exception`, он выведет сообщение об ошибке.

- ❸ Когда файбер завершит выполнение, он выведет полезное сообщение, прежде чем закончит конкурентное выполнение и вернет управление главному приложению.
- ❹ При запуске файбера просто создается стек вызовов, и, поскольку файбер сразу же приостанавливается, выполнение продолжается в контексте родительского стека.
- ❺ При передаче исключения из родительской части в файбер срабатывает условие `catch` и на консоль выводится сообщение `Error`.

Файбера — это эффективный способ управления контекстами выполнения между стеками вызовов в PHP, однако они все еще считаются низкоуровневыми. Они легко интегрируются с простыми операциями, но более сложные вычисления могут усложнить управление. Понимание принципов работы файберов критично для их эффективного использования, равно как и выбор правильной абстракции для управления ими. Пакет `Async` (<https://oreil.ly/vmkZJ>) из ReactPHP предоставляет эффективные абстракции для асинхронных операций, в том числе файбера, упрощая создание сложных конкурентных приложений.

## Читайте также

Руководство PHP по файбераам (<https://oreil.ly/iU6JH>).

## ГЛАВА 18

---

# Командная строка PHP

Разработчиками на PHP становятся люди из самых разных областей и с разным уровнем опыта в программировании. Независимо от того, являетесь ли вы выпускником факультета информационных технологий, бывалым программистом или человеком из области, не связанной с ИТ, простота языка позволяет легко влиться в эту тему. Тем не менее самым серьезным камнем преткновения для начинающих может стать интерфейс командной строки (CLI) PHP.

Новичкам, скорее всего, будет удобнее пользоваться графическим интерфейсом и управлять процессом с помощью мыши и дисплея. Если такому пользователю дать терминал с командной строкой, интерфейс может вызвать у него панику.

Будучи языком бэкенда, PHP часто управляется из командной строки. Это отпугивает разработчиков, не привыкших к текстовым интерфейсам. К счастью, приложения командной строки на базе PHP относительно просты в создании и очень мощны в использовании.

Приложение может предоставлять палитру команд, аналогичную стандартному RESTful-интерфейсу, делая взаимодействие с терминалом похожим на работу через браузер или API. Другое приложение может скрыть свой административный инструментарий в CLI, защищая менее продвинутых пользователей от случайной поломки приложения.

Одним из самых популярных PHP-приложений сегодня является WordPress (<https://wordpress.org/>) — открытая веб-платформа для ведения блогов. Большинство пользователей взаимодействуют с платформой через ее графический веб-интерфейс, однако сообщество WordPress поддерживает и богатый интерфейс командной строки WP-CLI (<https://wp-cli.org/>). Этот инструмент позволяет пользователю управлять всеми функциями графического интерфейса, но с помощью текстового терминала. Кроме того, он предлагает команды для управления пользовательскими ролями, конфигурацией системы, состоянием базы данных и даже системным кэшем. Все эти функции недоступны в стандартном веб-интерфейсе!

Любой PHP-разработчик обязан разбираться в возможностях командной строки в плане как самого PHP, так и расширения функциональности приложения. По-настоящему многофункциональное веб-приложение на определенном этапе будет работать на сервере без графического интерфейса. Поэтому способность управлять приложением из командной строки — это не просто шаг вперед, это необходимость.

Следующие рецепты помогут разобраться в тонкостях работы с консолью.

## 18.1. Разбор аргументов программы

### Задача

Вы хотите, чтобы при вызове вашего скрипта пользователь передавал аргумент для его последующего разбора внутри приложения.

### Решение

Используйте целое число \$argc и массив \$argv для получения значения аргумента непосредственно в скрипте. Например:

```
<?php
if ($argc !== 2) {
    die('Недопустимое количество аргументов.');
}

$name = htmlspecialchars($argv[1]);

echo "Привет, {$name}!" . PHP_EOL;
```

### Обсуждение

Если предположить, что скрипт из примера выше вы назвали `script.php`, то его можно вызвать следующей командой в терминале:

```
% php script.php World
```

Внутри переменной \$argc содержится количество параметров, переданных в PHP при выполнении скрипта. В примере из «Решения» имеются ровно два параметра:

- название самого скрипта (`script.php`);
- любое строковое значение, которое вы передаете после имени скрипта.



И \$argc, и \$argv можно отключить во время выполнения скрипта, установив флаг register\_argc\_argv (<https://oreil.ly/ZKuH>) в значение false в файле php.ini. Если эти параметры включены, то они будут содержать либо аргументы, передаваемые скрипту, либо информацию о GET-запросе, перенаправленном от веб-сервера.

Первым аргументом всегда будет имя выполняемого скрипта или файла. Все остальные аргументы разделяются пробелами. Если вам нужно передать составной аргумент (например, строку с пробелами), заключите его в двойные кавычки. Например:

```
% php script.php "дорогой читатель"
```

Более сложные реализации могут задействовать функцию PHP getopt() вместо непосредственной работы с переменными аргументов. Данная функция разбирает как короткие, так и длинные опции и передает их содержимое в массивы, которые ваше приложение сможет использовать в дальнейшем.

Короткие опции — это отдельные символы, указываемые в командной строке через дефис, например -v. Каждая опция либо просто присутствует (как флаг), либо сопровождается данными (как параметр).

Длинные опции предваряются двойным дефисом, но в остальном действуют так же, как и их короткие аналоги. Вы можете использовать в своем приложении и те и другие.



Иногда в консольных приложениях для одного и того же параметра применяется как длинная, так и короткая опция. Например, -v и --verbose часто используются для управления уровнем вывода скрипта. С помощью getopt() вы можете легко задействовать оба варианта, однако PHP не будет связывать их вместе. Если вы поддерживаете два разных метода для передачи одного и того же значения или флага, вам придется согласовывать их в скрипте вручную.

Функция getopt() принимает три параметра и возвращает массив, представляющий опции, которые интерпретатор PHP разобрал:

- первый аргумент — это одна строка, где каждый символ является короткой опцией или флагом;
- второй аргумент — это массив строк, где каждая строка обозначает имя длинной опции;
- последний аргумент, который передается по ссылке, представляет собой целое число, обозначающее индекс в \$argv, на котором разбор останавливается, когда PHP встречает неопциональный параметр.

Как короткие, так и длинные опции также принимают модификаторы. Если вы передаете опцию саму по себе, PHP не будет принимать значение для нее, а интерпретирует ее как флаг. Если вы добавите двоеточие к опции, PHP потребует значение. Если вы добавите два двоеточия, PHP рассмотрит значение как необязательное.

Для наглядности в табл. 18.1 перечислены различные способы использования коротких и длинных опций с этими дополнительными модификаторами.

**Таблица 18.1.** Аргументы PHP getopt()

Аргумент	Тип аргумента	Описание
a	Краткая опция	Одиночный флаг без значения: -a
b:	Краткая опция	Одиночный флаг с требуемым значением: -b value
c::	Краткая опция	Одиночный флаг с необязательным значением: -c value или просто -c
ab:c	Краткая опция	Комбинация трех флагов, где a и c не имеют значения, а b требует значения: -a -b value -c
verbose	Длинная опция	Строка параметров без значения: --verbose
name:	Длинная опция	Строка параметров с требуемым значением: --name Alice
output::	Длинная опция	Строка параметров с необязательным значением: --output file.txt или просто --output

Чтобы увидеть, насколько полезен разбор опций, определите программу, как в примере 18.1, в которой представлены короткие и длинные опции, а также используется свободный ввод (без опций) после флагов. Следующий сценарий будет ожидать:

- флаг, определяющий, должен ли вывод быть заглавными буквами (-c);
- имя пользователя (--name);
- дополнительный произвольный текст после опций.

### Пример 18.1. Функция getopt() с несколькими опциями

```
<?php
$optionIndex = 0;
```

```
$options = getopt('c', ['name:'], $optionIndex); ❶

$firstLine = "Привет, {$options['name']}!" . PHP_EOL; ❷

$rest = implode(' ', array_slice($argv, $optionIndex)); ❸

if (array_key_exists('c', $options)) { ❹
    $firstLine = strtoupper($firstLine);
    $rest = strtoupper($rest);
}

echo $firstLine;
echo $rest . PHP_EOL;
```

❶ Используйте `getopt()` для определения коротких и длинных опций, которые ожидает ваш скрипт. Третий, необязательный параметр передается по ссылке и будет перезаписан индексом, на котором интерпретатор закончит разбор.

❷ Извлечение из результирующего ассоциативного массива опций со значениями.

❸ Индекс из `getopt()` можно использовать для быстрого получения дополнительных данных из команды путем извлечения неразобранных значений из массива `$argv`.

❹ Опции без значений все равно установят ключ в ассоциативном массиве, но его значением будет булево `false`. Проверьте, что ключ существует, но не полагайтесь на его значение из-за противоречивой природы результата.

Если сценарий, определенный в примере 18.1, назвать `getopt.php`, то можно ожидать следующего результата:

```
% php getopt.php -c --name Reader Это весело
ПРИВЕТ, ЧИТАТЕЛЬ!
ЭТО ВЕСЕЛО
%
```

## Читайте также

Документация по `$argc` (<https://oreil.ly/BXdSI>), `$argv` (<https://oreil.ly/ODRwK>) и функции `getopt()` (<https://oreil.ly/ZfqTP>).

## 18.2. Чтение интерактивного пользовательского ввода

### Задача

Вы хотите считать пользовательский ввод в переменную.

### Решение

Реализуйте считывание данных из стандартного потока ввода с помощью константы файлового дескриптора `STDIN`. Например:

```
echo 'Введите свое имя: ';  
  
$name = trim(fgets(STDIN, 1024));  
  
echo "Добро пожаловать, {$name}!" . PHP_EOL;
```

### Обсуждение

Стандартный поток ввода облегчает чтение любых данных, поступающих с запросом. Чтение данных непосредственно из потока в программе с помощью функции `fgets()` приостановит выполнение вашей программы до тех пор, пока конечный пользователь не предоставит вам эти данные.

В примере выше применяется сокращенная константа `STDIN` для обращения к входному потоку. С таким же успехом можно использовать полное имя потока (вместе с явным `fopen()`), как показано в примере 18.2.

#### Пример 18.2. Чтение пользовательского ввода из `stdin`

```
echo 'Введите свое имя: ';  
  
$name = trim(fgets(fopen('php://stdin', 'r'), 1024));  
  
echo "Добро пожаловать, {$name}!" . PHP_EOL;
```



Специальные сокращенные имена `STDIN` и `STDOUT` доступны только в приложении. При использовании интерактивного терминала `REPL`, как в рецепте 18.5, эти константы не определяются и будут недоступны.

Альтернативной является расширение GNU Readline (<https://oreil.ly/eRhJw>). Это расширение выполняет большую часть ручной работы по запросу, получению и обработке вводимых пользователем данных. Если переписать пример из «Решения» с учетом этого расширения, получится код, как в примере 18.3.

**Пример 18.3.** Чтение ввода из расширения GNU Readline

```
$name = readline('Введите ваше имя: ');

echo "Добро пожаловать, {$name}!" . PHP_EOL;
```

Дополнительные функции, предоставляемые Readline, такие как `readline_add_history()` (<https://oreil.ly/J5do3>), позволяют эффективно управлять историей командной строки системы. Если расширение доступно, это мощный способ взаимодействия с пользовательским вводом.



В некоторых дистрибутивах PHP, например для Windows, поддержка Readline включена по умолчанию. В других ситуациях может потребоваться скомпилировать PHP, чтобы включить эту поддержку. Подробнее о расширениях PHP читайте в рецепте 15.4.

## Читайте также

Дальнейшее обсуждение стандартного ввода в рецепте 11.2.

## 18.3. Подсветка текста в консоли

### Задача

Вы хотите отображать текст в консоли разными цветами.

### Решение

Для корректного использования цветовых кодов в консоли необходимо применять правильное экранирование. Например, чтобы вывести строку С Новым годом, оформленную синим цветом на красном фоне, введите следующее:

```
echo "\e[0;34;41mС Новым годом! \e[0m" . PHP_EOL;
```

### Обсуждение

Unix-подобные терминалы поддерживают управляющие последовательности ANSI, которые предоставляют программам тонкий контроль над такими вещами, как расположение курсора и стиль шрифта. В частности, с помощью этой последовательности можно задать цвет, используемый терминалом для всего последующего текста:

```
\e[{foreground};{background}m
```

Цвета переднего плана бывают двух вариантов: обычные и полужирные (задаются дополнительным булевым флагом в определении цвета). Цвета фона не имеют таких различий. Все цвета обозначаются кодами, приведенными в табл. 18.2.

**Таблица 18.2.** Цветовые коды ANSI

Цвет	Нормальный передний план	Яркий передний план	Фон
Черный	0; 30	1;30	40
Красный	0; 31	1;31	41
Зеленый	0; 32	1;32	42
Желтый	0; 33	1;33	43
Синий	0; 34	1;34	44
Пурпурный	0; 35	1;35	45
Голубой	0; 36	1;36	46
Белый	0;37 (в действительности светло-серый)	1;37	47

Чтобы вернуть цвета терминала к стандартным настройкам, используйте простой `\0` вместо любого определения цвета, а код `\e[0m` сбросит все цветовые атрибуты.

## Читайте также

В Википедии рассказывается об управляющих последовательностях ANSI (<https://oreil.ly/y02cf>).

## 18.4. Создание консольного приложения с помощью Symfony Console

### Задача

Вы хотите создать полноценное консольное приложение без ручного написания кода разбора и обработки аргументов.

### Решение

Используйте компонент Symfony Console для определения вашего приложения и его команд. Так, в примере 18.4 задана команда Symfony для приветствия пользователя по имени с помощью Hello world в консоли. Затем в примере 18.5 этот

объект команды используется для создания приложения, которое приветствует пользователя в терминале.

#### Пример 18.4. Базовая команда hello world

```
namespace App\Command;

use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

#[AsCommand(name: 'app:hello-world')]
class HelloWorldCommand extends Command
{
    protected static $defaultDescription = 'Приветствие пользователя';

    // ...
    protected function configure(): void
    {
        $this
            ->setHelp('Эта команда приветствует пользователя...')
            ->addArgument('name', InputArgument::REQUIRED, 'Имя пользователя');
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $output->writeln("Привет, {$input->getArgument('name')}");
        return Command::SUCCESS;
    }
}
```

#### Пример 18.5. Создание реального консольного приложения

```
#!/usr/bin/env php
<?php
// application.php

require __DIR__.'/vendor/autoload.php';

use Symfony\Component\Console\Application;

$application = new Application();

$application->add(new App\Command>HelloWorldCommand()));

$application->run();
```

Затем выполните следующую команду:

```
% ./application.php app:hello-world User
```

## Обсуждение

Проект с открытым исходным кодом Symfony (<https://symfony.com/>) предоставляет надежную коллекцию многократно используемых компонентов для PHP. Он выступает в качестве фреймворка, упрощающего и значительно повышающего скорость разработки веб-приложений. Он очень хорошо документирован, обладает большим функционалом и, что особенно приятно, является бесплатным.



Фреймворк Laravel (<https://laravel.com/>), модули данных которого были рассмотрены в рецепте 16.9, сам по себе является мета-пакетом отдельных компонентов Symfony. Его собственный консольный инструмент Artisan (<https://oreil.ly/uY4QL>) построен на базе Symfony Console. Он обеспечивает обширное управление командной строкой над проектами Laravel, их конфигурацией и даже средами выполнения.

Как и любое другое расширение PHP, компоненты Symfony устанавливаются через Composer<sup>1</sup>. Сам компонент Console можно установить следующим образом:

```
% composer require symfony/console
```

Команда `require` обновит файл `composer.json`, включив в него компонент Console, а также установит этот компонент (и его зависимости) в каталог `vendor/` вашего проекта.



Если в вашем проекте еще не используется Composer, установка любого пакета автоматически создаст новый файл `composer.json`. Вам следует найти время, чтобы обновить его и добавить в автозагрузку необходимые классы и файлы, чтобы все работало без проблем. Подробнее о Composer, расширениях и автозагрузке читайте в главе 15.

Вы сможете приступить к использованию библиотеки сразу после ее установки. Бизнес-логика для различных команд может находиться в другом месте вашего приложения (например, в RESTful API), но также может быть импортирована и открыта через интерфейс командной строки.

По умолчанию каждый класс, происходящий от `Command`, позволяет вам работать с аргументами, предоставляемыми пользователем, и выводить содержимое

---

<sup>1</sup> Больше о Composer см. в рецепте 15.3.

обратно в терминал. Опции и аргументы генерируются с помощью методов `addArgument()` и `addOption()` внутри класса и могут быть прямо изменены в его методе `configure()`.

Вывод очень гибок. Допускается выводить содержимое непосредственно на экран с помощью любого из методов класса `ConsoleOutputInterface`, перечисленных в табл. 18.3.

**Таблица 18.3.** Методы вывода консоли Symfony

Метод	Описание
<code>writeln()</code>	Записывает одну строку в консоль. Эквивалентно использованию <code>echo</code> для некоторого текста, за которым следует явный символ новой строки <code>PHP_EOL</code>
<code>write()</code>	Записывает текст в консоль без добавления символа новой строки
<code>section()</code>	Создает новую секцию вывода, которой можно управлять атомарно, как независимым буфером вывода
<code>overwrite()</code>	Действует только для секции — перезаписывает содержимое секции указанными данными
<code>clear()</code>	Действует только для секции — очищает все содержимое секции

В дополнение к методам, представленным в табл. 18.3, Symfony Console позволяет создавать динамические таблицы в терминале. Каждый экземпляр таблицы `Table` привязан к интерфейсу вывода и может иметь столько строк, столбцов и разделителей, сколько вам нужно. В примере 18.6 показано, как создать простую таблицу и заполнить ее данными из массива перед выводом на консоль.

### Пример 18.6. Отображение таблиц в консоли с помощью Symfony

```
// ...

#[AsCommand(name: 'app:book')]
class BookCommand extends Command
{
    public function execute(InputInterface $input, OutputInterface $output): int
    {
        $table = new Table($output);
        $table
            ->setHeaders(['ISBN', 'Title', 'Author'])
            ->setRows([
                [
                    'ISBN' => '978-5-05-01000-1',
                    'Title' => 'Symfony в действии',
                    'Author' => 'Илья Гончаров'
                ],
                [
                    'ISBN' => '978-5-05-01000-2',
                    'Title' => 'Symfony в действии. Учебник',
                    'Author' => 'Илья Гончаров'
                ],
                [
                    'ISBN' => '978-5-05-01000-3',
                    'Title' => 'Symfony в действии. Учебник. Улучшенная версия',
                    'Author' => 'Илья Гончаров'
                ]
            ]);
        $table->render($output);
    }
}
```

```
[  
    '978-1-940111-61-2',  
    'Security Principles for PHP Applications',  
    'Eric Mann'  
],  
['978-1-098-12132-7', 'PHP Cookbook', 'Eric Mann'],  
])  
;  
$table->render();  
  
return Command::SUCCESS;  
}  
}  
}
```

Symfony Console автоматически разбирает содержимое, переданное в объект `Table`, и отображает таблицу с линиями сетки. Приведенная выше команда выводит на консоль следующий результат:

ISBN	Title	Author
978-1-940111-61-2	Security Principles for PHP Applications	Eric Mann
978-1-098-12132-7	PHP Cookbook	Eric Mann

Другие модули компонента помогают управлять и отображать динамические индикаторы выполнения (<https://oreil.ly/TszPm>), интерактивные подсказки и вопросы пользователя (<https://oreil.ly/8i5Hx>).

Компонент `Console` даже разрешает напрямую раскрашивать вывод терминала (<https://oreil.ly/arrtr>). В отличие от сложных управляющих последовательностей ANSI, рассмотренных в рецепте 18.3, `Console` позволяет непосредственно использовать именованные теги и стили для управления содержимым.



На момент написания книги компонент `Console` по умолчанию отключает подсветку вывода в системах Windows. Однако для Windows доступны различные бесплатные приложения (например, `Cmder` (<https://oreil.ly/gs5e6>)) в качестве альтернативы стандартному терминалу.

Терминал — это невероятно мощный интерфейс для ваших пользователей. Symfony `Console` позволяет легко настраивать этот интерфейс в вашем приложении, не прибегая к ручному разбору аргументов или оформлению вывода.

## Читайте также

Полная документация по компоненту `Console` в Symfony (<https://oreil.ly/vm8Qx>).

## 18.5. Использование встроенного в PHP цикла REPL

### Задача

Вы хотите протестировать некоторую логику PHP, не создавая для нее полноценного приложения.

### Решение

Воспользуйтесь интерактивной оболочкой PHP следующим образом:

```
% php -a
```

### Обсуждение

Интерактивная оболочка PHP работает по принципу REPL — цикл «чтение-оценка-вывод», который эффективно тестирует отдельные утверждения в PHP и, по возможности, выводит их непосредственно на терминал. В оболочке вы можете определять функции и классы или даже напрямую выполнять императивный код, не создавая файл сценария на диске.

Эта оболочка — отличный способ протестировать определенную строку кода или часть логики вне контекста полноценного приложения.

Она обеспечивает автозаполнение для всех функций и переменных PHP, а также для любых функций и переменных, которые вы определили во время работы оболочки. Просто введите первые несколько символов длинного имени, нажмите Tab, и оболочка автоматически завершит имя. Если существует несколько возможных вариантов завершения, дважды нажмите клавишу Tab, чтобы увидеть список доступных вариантов.

В конфигурационном файле `php.ini` можно настроить два параметра оболочки: `cli.page` позволяет внешней программе обрабатывать вывод, а не выводить его непосредственно на консоль, а `cli.prompt` — управлять стандартным приглашением `php >`.

Например, вы можете заменить само приглашение, передав произвольную строку в `#cli.prompt` в рамках сеанса оболочки следующим образом:

```
% php -a ①  
php > #cli.prompt=repl ~> ②  
repl ~> ③
```

- ❶ При первом вызове PHP запускается интерактивная оболочка.
- ❷ Прямое задание конфигурации `cli.prompt` отменяет значение по умолчанию до тех пор, пока не будет закрыта эта сессия.
- ❸ После переопределения стандартного приглашения вы будете видеть вашу новую версию до тех пор, пока не выйдете.



Обратные знаки могут использоваться для выполнения произвольного PHP-кода внутри самого приглашения. В некоторых примерах в документации по PHP (<https://oreil.ly/o6NU6>) данный метод применяется для добавления текущего времени к приглашению. Однако в разных системах это не всегда работает стабильно при выполнении PHP-кода.

Вы можете даже раскрасить вывод с помощью управляющих последовательностей ANSI, показанных в табл. 18.2. Это делает интерфейс визуально более приятным и позволяет подсвечивать важную информацию. Само приглашение CLI вводит четыре дополнительные управляющие последовательности (см. табл. 18.4).

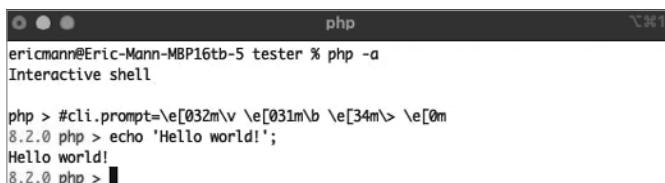
**Таблица 18.4.** Управляющие последовательности приглашения CLI

Последовательность	Описание
\e	Добавляет цвета к приглашению, используя ANSI-коды, представленные в рецепте 18.3
\v	Выводит версию PHP
\b	Указывает, в каком логическом блоке находится интерпретатор. По умолчанию это <code>php</code> , но может быть <code>/*</code> для представления многострочного комментария
\>	Представляет символ приглашения, который по умолчанию является <code>&gt;</code> . Когда интерпретатор находится внутри другого незавершенного блока или строки, этот символ меняется, чтобы указать, где находится оболочка. Возможные символы: <code>'</code> <code>"</code> <code>{</code> <code>(</code> <code>&gt;</code>

С помощью управляющих последовательностей ANSI для определения цветов и специальных последовательностей, заданных самим приглашением, у вас появляется возможность создать приглашение, которое отображает версию PHP и «координаты» интерпретатора, с приятным цветовым оформлением, как показано ниже:

```
php > #cli.prompt=\e[032m\v \e[031m\b \e[34m\> \e[0m
```

В результате предыдущей настройки на экране появится изображение, как на рис. 18.1.



```
ericmann@Eric-Mann-MBP16tb-5 tester % php -a
Interactive shell

php > #cli.prompt=\e[032m\v \e[031m\b \e[34m\> \e[0m
8.2.0 php > echo 'Hello world!';
Hello world!
8.2.0 php >
```

Рис. 18.1. Консоль PHP с обновленным цветовым оформлением



Не каждая консоль поддерживает раскрашивание через управляющие последовательности ANSI. Если вы намерены применить такой подход, тщательно протестируйте свои последовательности, прежде чем выпускать свою систему. Хотя грамотно оформленная консоль выглядит привлекательно и приятна в использовании, некорректно настроенные управляющие последовательности могут сделать работу с консолью практически невозможной.

## Читайте также

Документация по интерактивной командной оболочке PHP (<https://oreil.ly/HrCV->).

---

## **06 авторе**

Эрик А. Манн работает инженером-программистом почти два десятилетия. Он создавал масштабируемые проекты как для стартапов, так и для компаний из списка Fortune 500. Эрик часто выступает с докладами по архитектуре программного обеспечения, основам компьютерной безопасности и лучшим практикам разработки. Он регулярно публикуется в журнале php[architect] и больше всего любит помогать начинающим разработчикам не совершать тех же профессиональных ошибок, что и он когда-то.

---

## Иллюстрация на обложке

Животное на обложке — серебряная лофура, или серебряный фазан (*Lophura pustthemera*). Серебряные фазаны обитают в сосновых и бамбуковых лесах различных горных регионов Юго-Восточной Азии. Самцы фазанов имеют черно-белое оперение с небольшим хохолком из выющиеся черных перьев, в то время как самки в основном коричневые, с гораздо более коротким хвостом. Одной из наиболее заметных особенностей является ярко-красная маска на голове с бородками. Серебряные фазаны иногда гибридизируют с калийскими фазанами, где их ареалы обитания пересекаются.

Международным союзом охраны природы и природных ресурсов серебряным фазанам присвоен охранный статус LC, то есть это вид, вызывающий наименьшие опасения. Многие из животных, изображенных на обложках книг O'Reilly, находятся под угрозой исчезновения. Все они важны для нашего мира.

Иллюстрация на обложке выполнена Карен Монтгомери на основе старинной гравюры из журнала *Riverside Natural History*.

*Эрик А. Манн*

**Рецепты PHP.**

**Для профессиональных разработчиков**

*Перевел с английского А. Ларин*

Изготовлено в России. Изготовитель: ТОО «Спринт Бук».  
Место нахождения и фактический адрес: 010000, Казахстан, город Астана,  
район Алматы, Проспект Ракымжан Кошкарбаев, дом 10/1, н.п. 18

Дата изготовления: 09.2024.

Наименование: книжная продукция.

Срок годности: не ограничен.

Подписано в печать 19.07.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 33,540. Тираж 700. Заказ 0000.  
Отпечатано в ТОО «ФАРОС Графикс». 100004, РК, г. Караганда, ул. Молокова, 106/2.