ИЗУЧАЕМ

АСИНХРОННЫЙ PHP

С REACTPHP

ПОЛНОЕ РУКОВОДСТВО ПО СОЗДАНИЮ АСИНХРОННЫХ ПРИЛОЖЕНИЙ НА PHP

ЖУК СЕРГЕЙ

Изучаем Асинхронный PHP с ReactPHP

Sergey Zhuk

Эта книга предназначена для продажи на
http://leanpub.com/event-driven-php-ru

Эта версия была опубликована на 2020-10-02

Это книга с Leanpub book. Leanpub позволяет авторам и издателям участвовать в так называемом Lean Publishing - процессе, при котором электронная книга становится доступна читателям ещё до её завершения. Это помогает собрать отзывы и пожелания для скорейшего улучшения книги. Мы призываем авторов публиковать свои работы как можно раньше и чаще, постепенно улучшая качество и объём материала. Тем более, что с нашими удобными инструментами этот процесс превращается в удовольствие.

Оглавление

Что такое ReactPHP

Проблема

Думаю, что сейчас в PHP-сообществе у нас нет какого-то общего мнения об асинхронном коде. Одна часть сообщества рассматривает асинхронный PHP - это что-то новое и интересно, то что поможет решить некоторые проблемы, связанные с производительностью. Однако другая часть сообщества категорически против асинхронного кода и вместо PHP предлагает использовать “более подходящие” для этого языки вроде Go или NodeJs. Давайте попробуем разобраться и понять, почему так происходит. Почему многие считают, что PHP не подходит для решения задач, требующих высокой производительности. Мне, например, нравится PHP и я хочу его использовать для решения разных задач. Я так же не хочу ограничивать себя только циклом запрос-ответ. Да и не хочется расширять свой стэк, добавляя в него ещё один язык со своей экосистемой, только лишь потому что кто-то в Твиттере сказал, что PHP мне не подходит.

И самая частая причина - это то, что PHP не создавался как язык для решения задач, связанных с производительностью. Что PHP это только про вэб и отрисовку HTML страниц. С одной стороны да, это так. Тяжело поспорить с тем, что PHP не создавался как язык для решения асинхронных задач. По правде говоря, он и не создавался как язык для создания больших и сложных приложений. В те времена и JavaScript’а ещё не было, и никто даже не думал об асинхронности. Все синхронно отрисовывали HTML и были этим вполне довольны.

Но с тех пор почти всё изменилось. Сейчас у нас есть Composer с огромным сообществом вокруг. И сам язык сильно изменился. У нас появились мощные фреймворки, которые помогают создавать настоящие энтерпрайз приложения на PHP. И шаг за шагом PHP уже больше не ограничен одним лишь циклом запрос-ответ. У нас появилось разделение на фронтенд и бэкенд, мы разрабатываем различные API, делаем интеграции с другими инфраструктурными

системами и конечно же консольные команды на PHP уже стали обыденностью. Более того и требования к нашим программам тоже сильно изменились. Теперь все считают миллисекунды и хотят чтобы приложение выполнялось максимально быстро.

Асинхронный код

Каким же образом асинхронный код поможет решить проблемы с производительностью? Чтобы разобраться в этом, проведём небольшую классификацию программ по типу выполнения.

Синхронный код. У программы один поток выполнения. Код выполняется последовательно строчка за строчкой. Пока одна строчка кода выполняется, вся остальная программа ждёт. Таким образом работает традиционный код на PHP.

Асинхронный код. У программы по-прежнему одни поток выполнения, но добавляется асинхронность. Программа всё время чем-то занята, постоянно переключаясь между разными задачами. Таким образом работает NodeJs и ReactPHP.

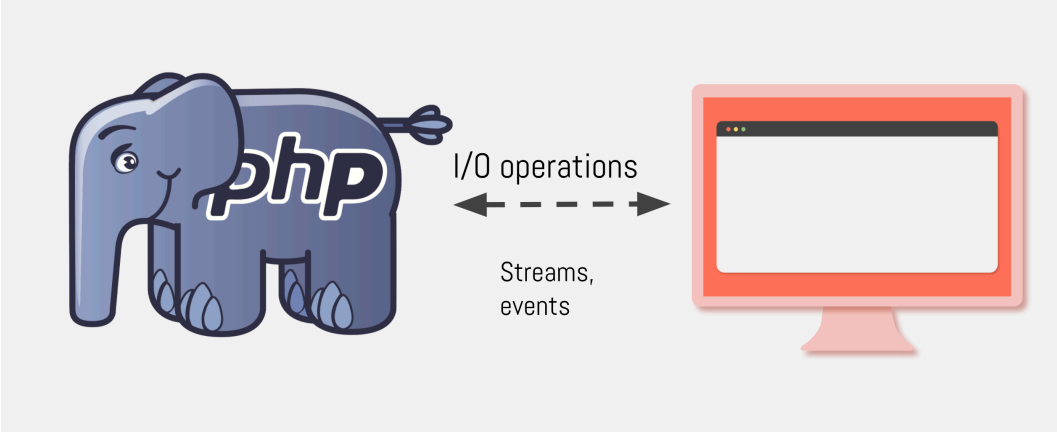
Параллельный код. У программы несколько потоков выполнения, каждый из которых может выполнять свою собственную задачу независимо от другого. Например, Python может работать таким образом.

Мы имеем дело с PHP и это означает, что мы в этой книге будем рассматривать асинхронную модель выполнения кода. Где у нас есть один поток выполнения, который постоянно чем-то занят. Если сравнивать такой подход с многопоточностью, то в теории используя асинхронность мы сможем выполнить больше задач за один и тот же промежуток времени. Потому что поток выполнения программы будет постоянно занят. Таким образом мы максимально задействуем доступные программе вычислительные ресурсы. Параллельное выполнение с несколькими потоками может оказаться довольно сложным с точки зрения написания кода: нужно будет синхронизировать потоки, управлять памятью и многое другое. В теории параллельное выполнение выглядит красиво, но на практике скорее всего код будет очень запутанным.

Как добавить PHP асинхронности? Чтобы ответить на этот вопрос, нам сначала нужно обсудить операции ввода-вывода, и концепцию неблокирующего I/O (когда операции чтения/записи не блокируют поток выполнения программы). Дело в том, что операции ввода/вывода довольно медленные. Как правило, они выполняются в разы медленнее чем вычисления. Например, когда мы говорим о вычислениях CPU, то речь идёт о наносекундах. Но какие-нибудь сетевые запросы уже могут измеряться в миллисекундах или даже в секундах. Разница очень существенна. Ввод/вывод в приложениях у нас везде: вызовы API, взаимодействие с файловой системой, запросы к БД. Таким образом программы, написанные в традиционном блокирующем подходе, проводят большую часть времени в ожидании ответа от диска или пока сетевой запрос не будет выполнен. Но если производительность для нас критична и нужно повысить скорость выполнения программы, то нам нужно каким-то образом запускать такие операции *в фоне*, выполняя их асинхронно.

Как это сделать в PHP? Как можно запускать задачи, и при этом не дожидаться завершения их выполнения? Можно попросить операционную систему выполнять все задачи, которые связаны с вводом-выводом. Например, нужно прочитать данные из файла. Мы не хотим, чтобы PHP-скрипт ждал ответа от файловой системы, поэтому не общаемся с ней напрямую. Вместо этого мы просим операционную систему открыть поток на чтение в файловой системе. И на этом всё. Дальше PHP-скрипт может выполнять любую другую задачу.

Как только операционная система выполнит чтение, скрипт получит событие об этом. Событие будет содержать в себе прочитанные данные, так что скрипт сможет вернуться назад и обработать их.



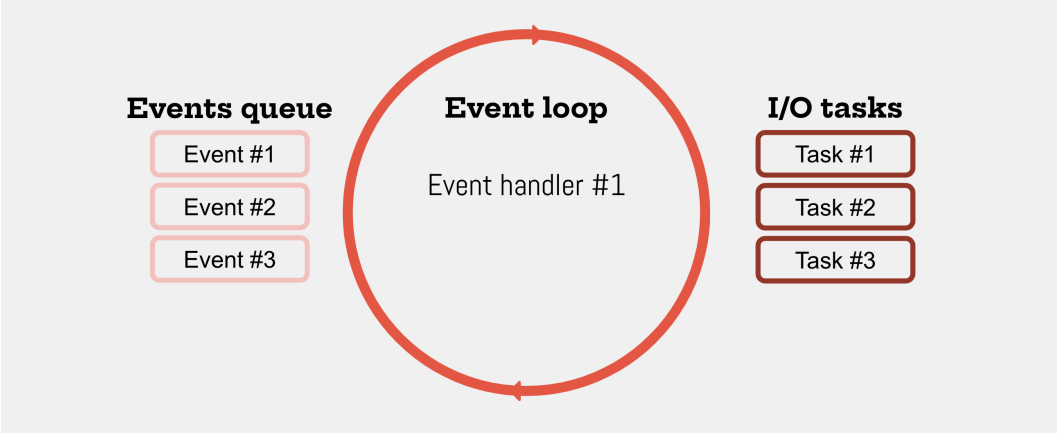
Событийно-ориентированная архитектура

Чтобы иметь возможность слушать события, возвращаться назад и обрабатывать их, нужна совершенно другая архитектура приложения, отличная от традиционного цикла запрос-ответ. Событийно-ориентированная архитектура, которая используется например в NodeJs. Здесь у PHP и NodeJs есть одна общая черта: оба выполняются в одном потоке. И на самом деле это не ограничивает нас. Всё, что может выполняться медленно, операции ввода-вывода будут выполнены не PHP-скриптом, а самой операционной системой. PHP будет заниматься только обработкой событий, полученных от системы.

Давайте более детально разберём такой подход. Вся работа, всё взаимодействие между различными частями кода в такой архитектуре построено на событиях. У нас есть один выполняющийся процесс – наш PHP-скрипт с запущенным циклом событий. Внутри цикл событий представляет собой простой “бесконечный” цикл, который слушает определенные события и вызывает в ответ на них обработчики. Например, у нас есть какая-то задача, связанная с вводом-выводом – чтение файла. Мы начинаем её и просим операционную систему прочитать данные из файла. На этом всё. Далее поток

выполнения программы может начать делать какую-то другую задачу. Мы запустили чтение файла, и не дожидаемся пока оно закончится. Как только операционная система прочитает очередную порцию данных из файла, она отправит нам событие вместе с прочитанными данными. Запись об этом событии попадает в очередь событий. Поток выполнения берёт первое событие из очереди и вызывает соответствующего обработчика для этого события. Допустим, мы делаем какие-то вычисления на основе данных, полученных из файла. Обработчик события сам может также начать другие операции ввода-вывода. Например, мы хотим отправить эти данные по HTTP. Эти новые задачи также сгенерируют свои собственные события. Как только обработчик события выполнится, поток выполнения программы возвращается к очереди событий и забирает из неё новые события, которые произошли пока поток был занят. Цикл событий выполняется до тех пор, пока ему есть чем заняться: или в фоне ещё выполняются задачи, или в очереди событий ещё остаются необработанные события. Как только все задачи будут выполнены, а очередь событий станет пустой, цикл событий остановится.

Имея подобного рода архитектуру, можно запускать несколько задач ввода-вывода, и при этом нам не нужно дожидаться их выполнения. Вместо этого, мы узнаем об их выполнении через события и сможем вернуться назад и обработать эти события. Программа больше не тратит время впустую. Все эти задачи ввода-вывода будут выполняться в фоне столько времени, сколько им нужно.



Hello world

Что же в PHP? Конечно, в самом языке у нас пока нет встроенной поддержки для написания асинхронного кода. Да, есть некоторые низко-уровневые возможности, но у нас нет высокоуровневых абстракций, таких как цикл событий, стримы и промисы. Но это ещё не означает, что мы не можем использовать PHP для решения асинхронных задач. С появлением Composer'а проблема с отсутствием инструментов была решена PHP-сообществом и ReactPHP это один из инструментов, который позволяет писать высокоуровневый асинхронный код на PHP.

Чтобы продемонстрировать то, как это выглядит, давайте быстро взглянем на простой пример кода на JavaScript.

```
setTimeout(function () {
  console.log("world");
}, 0);

console.log("Hello ");
```

Функция `setTimeout()` откладывает выполнение некоторого кода. В нашем случае она откладывает вывод на консоль слова `world`. Этот код будет выполнен через указанное количество миллисекунд. Мы указали `0` миллисекунд и здравый смысл подсказывает, что задержка в `0` миллисекунд - это то же самое, что выполнить прямо сейчас. И глядя на этот код, мы ожидаем, что слово "world" будет выведено первым. Однако если мы запустим этот код, то будет выведено "hello world". Так происходит, потому что функция `setTimeout()` асинхронная. Она сама ничего не выполняет. Внутри она берёт переданный ей колбэк и откладывает его в очередь на выполнение. После этого выполняется строчка кода, которая выводит "hello". Затем цикл событий просматривает очередь с таймерами, видит что задержка в `0` секунд уже выполнилась и выполняет соответствующий колбэк. И вот мы нарушили традиционный последовательный поток выполнения. Второй вызов `console.log()` будет выполнен до вызова внутри `setTimeout()`. А теперь точно такой же код на ReactPHP:

```
$loop = Factory::create();
$loop->addTimer(
  0,
  function () {
    echo 'world';
  }
);

echo 'Hello ';
$loop->run();
```

Что здесь происходит? Рассмотрим этот пример сверху вниз. Сперва мы создаём цикл событий. В PHP у нас нет такой роскоши, как выполняющийся в фоне цикл событий. Поэтому нам приходится явно вручную его создавать. Далее мы устанавливаем таймер, указывая что через `0` секунд нужно будет вывести слово “world”. Затем печатаем слово “Hello” и вручную запускаем цикл событий. К тому моменту как будет запущен цикл событий, слово “Hello” уже будет напечатано. Цикл событий запускается и видит, что таймер уже сработал. Генерируется соответствующее событие и в ответ на это событие вызывается обработчик – колбэк внутри `addTimer()`. Выводится слово “world”. На этом простом примере можно увидеть базовую структуру любого приложения на ReactPHP:

1. Один раз в начале программы создаётся цикл событий.
2. Код, который использует цикл событий. В этой части описывается поведение нашего приложения: как оно будет реагировать на разные события.
3. В конце программы один раз запускается цикл событий.

Если удалить создание и запуск цикла событий, то оказывается, что код, написанный на ReactPHP выглядит почти точно также как его аналог на JavaScript. Единственное отличие в том, что в PHP нам нужно явно создавать и запускать цикл событий.

Цикл событий

Компонент: EventLoop
Версия: 1.0
Установка: `composer require react/event-loop`

Основы

Event loop[1] является основным компонентом ReactPHP, это самый низкоуровневый компонент. Все другие компоненты используют его. Цикл событий выполняется в одном потоке и отвечает за планирование асинхронных операций. Он последовательно обрабатывает события в очереди, выполняя связанные с ними колбэки-обработчики. Мы регистрируем эти обработчики для определённых типов событий до запуска цикла событий. Колбэки должны быть короткими функциями, которые **не блокируют CPU** на длительное время, иначе весь цикл событий будет тоже заблокирован. Как только выполнение колбэка будет завершено цикл событий достаёт следующее событие из очереди и выполняет соответствующий колбэк.

Никакой другой код не выполняется параллельно. Цикл событий — это единственная синхронная вещь в приложении. Другими словами:

> *Всё кроме вашего кода выполняется параллельно.*

Цикл событий реализует шаблон Реактор. Мы регистрируем событие, подписываемся на него и начинаем слушать. Как только событие будет вызвано, мы получим уведомление и сможем *отреагировать* на это событие с помощью обработчика, выполнив некоторый код. Каждая итерация цикла событий

[1] http://reactphp.org/event-loop/

называется *тиком*. Как только в цикле больше не остаётся слушателей, он завершается.

С точки зрения использования цикла вам не придётся иметь дело с настолько низкоуровневыми деталями, но важно понимать как всё это внутри работает. Обычно вы будете создавать цикл событий с помощью фабрики, а затем просто передавать его в другие компоненты. После чего вы просто запускаете цикл, чтобы запустить приложение.

1. В начале программы один раз создаём цикл событий.
2. Настраиваем компоненты.
3. В конце программы запускаем цикл.

```
$loop = React\EventLoop\Factory::create();

// код, который использует цикл событий

$loop->run();
```

После вызова метода `run()` у объекта цикла событий, он будет выполняться до тех пор пока в цикле остаются обработчики событий. Затем программа останавливается. Чтобы принудительно остановить цикл событий, можно явно вызвать метод `stop()`.

Реализации

ReactPHP предоставляет несколько реализаций цикла событий в зависимости от того, какие расширения доступны у вас в окружении. Наиболее удобным и рекомендуемым способом создания объекта цикла является использование фабрики:

```
$loop = React\EventLoop\Factory::create();
```

Под капотом фабрика просто проверяет доступные расширения и выбирает подходящую реализацию цикла. Все реализации `React\EventLoop\LoopInterface` поддерживают:

- Опрос файловых дескрипторов.
- Таймеры.
- Отложенное выполнение колбэков.

Хотя внутри реализации циклов отличаются, сама программа не должна зависеть от какой-либо конкретной реализации. Могут быть небольшие различия в расчётах времени или порядке, в котором будут обрабатываться определённые типы событий. Но эти различия не должны повлиять на поведение программы.

Цикл событий и несколько CPU

Поскольку цикл событий является однопоточным, это означает, что он будет использовать только один CPU?

Да, PHP выполняется в одном потоке и использует только одно ядро процессора. Проблема с запуском некоторого кода одновременно на нескольких ядрах процессора заключается в том, что для этого требуется некоторая координация между этими потоками исполнения. Чтобы разбить задачу между несколькими ядрами процессора все потоки должны сообщать друг другу текущее состояние всей программы. Но основная идея ReactPHP состоит в том, чтобы полностью использовать процессорное время, а не в том, чтобы загрузить все его 16 ядер. Если вы хотите использовать ваши 16 ядер, просто запустите 16 серверов на разных портах и поставьте перед ними балансировщика.

Таймеры

Таймеры могут выполнять некоторый код в определённое время, через указанное число секунд в будущем (точно так же как делают функции `setTimeout()` и `setInterval` в JavaScript). Это не то же самое, что функция `sleep()`, наоборот таймеры представляют собой *события* в будущем. Таймеры будут запускаться по истечении указанного времени.

Асинхронность и параллельность

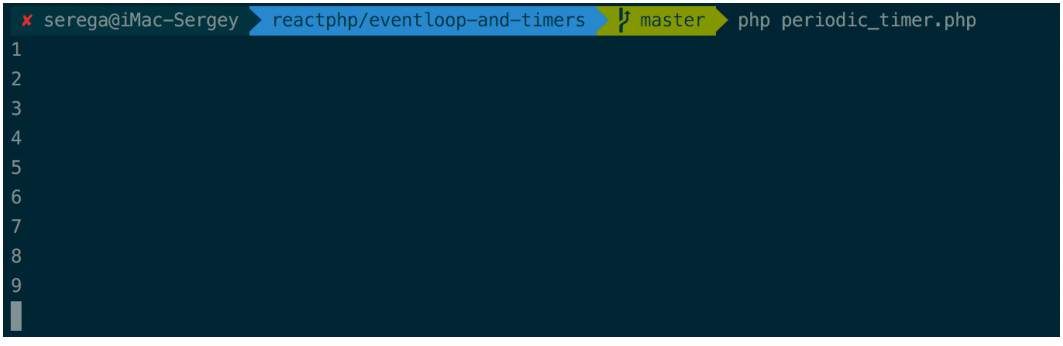
Асинхронность это не то же самое что *параллельность*. Считайте *асинхронностью* непоследовательное выполнение кода. Тогда как параллельность — это возможность
выполнять один и тот же код. Цикл событий работает асинхронно, но не параллельно. Это означает, что таймеры не являются на 100% точными, они могут немного запаздывать. Кроме того, если у вас есть несколько таймеров, которые планируется выполнить одновременно, порядок их выполнения не гарантируется. Любой таймер будет выполнен **не ранее указанного времени**. Код выполняется в одном потоке и не может быть прерван. Это значит что все таймеры выполняются в том же потоке, что и цикл событий. Если один таймер выполняется слишком долго, то все остальные таймеры будут ждать, пока этот таймер не будет выполнен. Кроме того, возможно, что некоторые таймеры вообще **никогда** не будут выполнены.

Периодический таймер

Данный таймер будет постоянно вызывать свой колбэк каждые n секунд. Периодический таймер может быть добавлен к циклу событий с помощью метода `addPeriodicTimer($interval, callable $callback)`. Метод принимает интервал в секундах и колбэк, который будет вызван по окончанию указанного интервала:

```php
$loop = React\EventLoop\Factory::create();
$counter = 0;

$loop->addPeriodicTimer(
  2,
  function () use (&$counter) {
    $counter++;
    echo "$counter\n";
  }
);

$loop->run();
```

Мы регистрируем периодический таймер в цикле событий. Затем мы стартуем цикл, вызывая `$loop->run()`. Когда будет запущен таймер, поток выполнения покинет цикл событий и перейдёт в *таймер*, код которого будет выполнен. Каждые две секунды таймер будет отображать увеличивающееся число. Цикл событий будет работать бесконечно.



Колбэк может принимать объект таймера, в котором он выполняется:

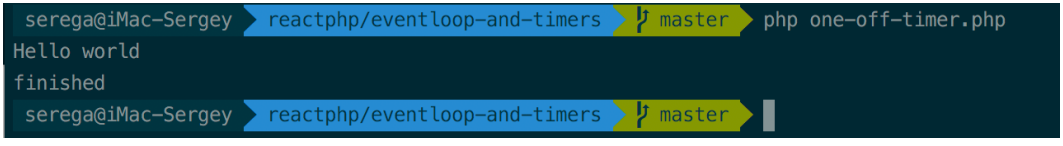
```
use React\EventLoop\TimerInterface;

$loop->addPeriodicTimer(
  2,
  function (TimerInterface $timer) {
    // ...
  }
);
```

Одноразовый таймер

Единственное отличие с периодическим таймером заключается в том, что данный таймер будет выполнен только один раз, и затем будет удалён из хранилища таймеров.

```
$loop = React\EventLoop\Factory::create();

$loop->addTimer(2, fn() => print "Hello world\n");

$loop->run();
echo "finished\n";
```

Через 2 секунды скрипт выведет строку `Hello world`, после чего таймер будет удалён, а цикл событий остановится.



Управление таймерами

У цикла событий есть ещё два метода для управления таймерами:

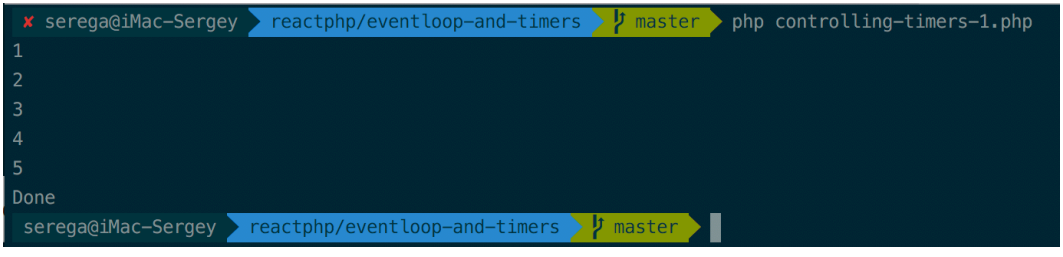
- `cancelTimer(TimerInterface $timer)` отменяет указанный таймер.

- `isTimerActive(TimerInterface $timer)` проверяет подключён ли указанный таймер к циклу событий.

Можно использовать переданный внутрь колбэка объект таймера, чтобы отменить его:

```
use React\EventLoop\TimerInterface;;

$loop = React\EventLoop\Factory::create();
$counter = 0;

$loop->addPeriodicTimer(
  2,
  function (TimerInterface $timer) use (&$counter, $loop) {
    $counter++;
    echo "$counter\n";

    if ($counter === 5) {
      $loop->cancelTimer($timer);
    }
  }
);

$loop->run();
echo "Done\n";
```

После пятого выполнения этот таймер будет удалён. Когда цикл события становится пустым, он останавливается.



Таймеры могут взаимодействовать друг с другом. Оба метода `addTimer()` и `addPeriodicTimer()` возвращают объект созданного таймера. Затем мы можем

использовать этот объект и передать его внутрь колбэка другого таймера. Таким образом можно указать тайм-аут для определённого события:

```
$loop = React\EventLoop\Factory::create();
$counter = 0;

$periodicTimer = $loop->addPeriodicTimer(
  2,
  function () use (&$counter) {
    $counter++;
    echo "$counter\n";
  }
);

$loop->addTimer(
  5,
  fn() => $loop->cancelTimer($periodicTimer)
);

$loop->run();
```

В примере выше периодический таймер будет выполняться только первые 5 секунд, после этого он будет удалён из цикла событий.

Следует избегать блокирующих операций

Так как все таймеры выполняются в одном потоке, то следует избегать блокирующих вызовов внутри их колбэков. Один заблокированный таймер может остановить весь цикл событий:

```
$loop = React\EventLoop\Factory::create();

$i = 0;

$loop->addPeriodicTimer(
  1,
  function () use (&$i) {
    echo ++$i, "\n";
  }
);

$loop->addTimer(2, fn() => sleep(10));

$loop->run();
```

Первый периодический таймер ждёт 10 секунд пока второй одноразовый таймер не будет завершён.

Заключение

Таймеры могут использоваться для выполнения некоторого кода в будущем. Этот код *может быть выполнен после* указанного интервала. Каждый таймер выполняется в том же потоке, что и весь цикл, поэтому любой таймер может повлиять на всё приложение. Таймеры могут быть полезными для неблокирующих операций, таких как ввод-вывод, но выполнение долгоживущего кода в них может привести к неожиданным результатам.

Вы можете найти примеры из этой главы на GitHub[2].

[2] https://github.com/seregazhuk/reactphp-book/tree/master/1-timers