O'REILLY®

«Кайл распутывает клубок асинхронных возможностей JavaScript и показывает, как пользоваться ими при помощи обещаний и генераторов».

— Марк Грабански, исполнительный директор и UI-разработчик, Frontend Masters

КАЙЛ СИМПСОН

АСИНХРОННАЯ ОБРАБОТКА & ОПТИМИЗАЦИЯ

ВЫ НЕ ЗНАЕТЕ JS

ББК 32.988.02-018
УДК 004.738.5
С37

Симпсон К.

С37 {Вы не знаете JS} Асинхронная обработка и оптимизация. — СПб.: Питер, 2019. — 352 с. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1313-2

Каким бы опытом программирования на JavaScript вы ни обладали, скорее всего, вы не понимаете язык в полной мере. Это лаконичное, но при этом глубоко продуманное руководство посвящено новым асинхронным возможностям и средствам повышения производительности, которые позволяют создавать сложные одностраничные веб-приложения и избежать при этом «кошмара обратных вызовов».

Как и в других книгах серии «Вы не знаете JS», вы познакомитесь с нетривиальными особенностями языка, которых так боятся программисты. Только вооружившись знаниями, можно достичь истинного мастерства.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1491904220 англ.

Authorized Russian translation of the English edition of You Don't Know JS: Async & Performance (ISBN 9781491904220)

ISBN 978-5-4461-1313-2

Оглавление

1 Асинхронность: сейчас и потом

Одна из важнейших, но при этом часто недопонимаемых частей программирования в таких языках, как JavaScript, — это средства выражения и управления поведением программы во времени. Конечно, я не имею в виду то, что происходит от начала цикла `for` до конца цикла `for`, что, безусловно, занимает некоторое время (микросекунды или миллисекунды). Речь о том, какая часть программы выполняется сейчас, а какая часть программы будет выполняться позднее, а между «*сейчас*» и «*потом*» существует промежуток, в котором ваша программа не выполняется активно.

Практически всем нетривиальным программам, когда-либо написанным (особенно на JS), в том или ином отношении приходится управлять этим промежутком, будь то ожидание пользовательского ввода, запрос информации из базы данных или файловой системы, передача данных по сети и ожидание ответа или периодическое выполнение задачи с фиксированным интервалом времени (скажем, анимация). Во всех этих ситуациях ваша программа должна управлять состоянием на протяжении промежутка времени.

Собственно, связь между частями программы, выполняющимися *сейчас* и *потом*, занимает центральное место в асинхронном программировании.

Безусловно, асинхронное программирование было доступно с первых дней JS. Но многие разработчики JS никогда не задумывались над тем, как именно и почему оно возникает в их программах, и не использовали другие способы его организации. Всегда существовал *достаточно хороший* вариант — обычная функция обратного вызова (callback). И сегодня многие настаивают на том, что обратных вызовов более чем достаточно.

Но с ростом масштаба и сложности JS, которые необходимы для удовлетворения вечно расширяющихся требований полноценного языка программирования, работающего в браузерах и на серверах, а также на всех промежуточных мыслимых устройствах, проблемы с управлением асинхронностью становятся непосильными. Возникает острая необходимость в решениях, которые были бы одновременно более действенными и разумными.

Хотя все эти рассуждения могут показаться довольно абстрактными, уверяю вас, что тема будет более полно и конкретно рассмотрена в книге. В нескольких ближайших главах представлены перспективные методы асинхронного программирования JavaScript.

Но сначала необходимо гораздо глубже разобраться, что такое асинхронность и как она работает в JS.

Блочное строение программы

Программу можно написать в одном файле .js, но ваша программа почти наверняка состоит из нескольких блоков, только один из которых будет выполняться в данный момент, а остальные будут выполняться *позднее*. Самой распространенной структурной единицей *блоков* является функция.

Проблема, с которой сталкивается большинство начинающих разработчиков JS, заключается в том, что «*потом*» не наступает четко и немедленно после «*сейчас*». Иначе говоря, задачи, которые

не могут быть завершены прямо сейчас, по определению будут завершаться асинхронно, а следовательно, вы не будете наблюдать блокирующее поведение, как вы могли бы интуитивно предположить или желать.

Пример:

```
// ajax(..) - произвольная функция Ajax из библиотеки
var data = ajax( "http://some.url.1" );

console.log( data );
// Ой! В общем случае `data` не содержит результатов Ajax
```

Вероятно, вы знаете, что стандартные запросы Ajax не завершаются синхронно. Это означает, что у функции `ajax(..)` еще нет значения, которое можно было бы вернуть для присваивания переменной `data`. Если бы функция `ajax(..)` могла блокироваться до возвращения ответа, то присваивание `data = ..` работало бы нормально.

Но Ajax-взаимодействия происходят не так. Вы выдаете асинхронный запрос Ajax *сейчас*, а результаты получаете только *через какое-то время*. Простейший (но определенно не единственный и даже не всегда лучший!) способ «ожидания» основан на использовании функции, обычно называемой *функцией обратного вызова* (callback function):

```
// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", function myCallbackFunction(data){

    console.log( data ); // Отлично, данные `data` получены!

} );
```

Возможно, вы слышали о синхронных Ajax-запросах. Хотя такая техническая возможность существует, вы никогда, ни при каких условиях не должны пользоваться ею, потому что она полностью блокирует пользовательский интерфейс браузера (кнопки, меню, прокрутку и т. д.) и перекрывает любые взаимодействия с пользователем. Это очень плохая мысль, избегайте ее в любых случаях.

Прежде чем вы начнете протестовать — нет, ваше желание избежать путаницы с обратными вызовами не оправдывает блокирующие синхронные операции Ajax. Возьмем для примера следующий код:

```
function now() {
    return 21;
}

function later() {
    answer = answer * 2;
    console.log( "Meaning of life:", answer );
}

var answer = now();

setTimeout( later, 1000 ); // Meaning of life: 42
```

Программа состоит из двух блоков: того, что будет выполняться *сейчас*, и того, что будет выполняться *потом*. Наверное, вы и сами понимаете, что это за блоки, но чтобы не оставалось ни малейших сомнений:

Сейчас:

```
function now() {
    return 21;
}
function later() { .. }

var answer = now();

setTimeout( later, 1000 );
```

Потом:

```
answer = answer * 2;
console.log( "Meaning of life:", answer );
```

Блок «*сейчас*» выполняется немедленно, как только вы запускаете свою программу. Но `setTimeout(..)` также настраивает событие (тайм-аут), которое должно произойти в будущем, так что содер-

жимое функции `later()` будет выполнено позднее (через 1000 миллисекунд).

Каждый раз, когда вы упаковываете фрагмент кода в функцию и указываете, что она должна быть выполнена по некоторому событию (таймер, щелчок мышью, ответ Ajax и т. д.), вы создаете в своем коде блок «*потом*», а следовательно, вводите асинхронность в свою программу.

Асинхронный вывод в консоль

Не существует никаких спецификаций или наборов требований, определяющих, как работают методы `console.*`, — официально они не являются частью JavaScript, а добавляются в JS управляющей средой.

А значит, разные браузеры и среды JS действуют так, как считают нужным, что иногда приводит к неожиданному поведению.

В частности, для некоторых браузеров и некоторых условий `console.log(..)` не выводит полученные данные немедленно. Прежде всего, это может произойти из-за того, что ввод/вывод — очень медленная и блокирующая часть многих программ (не только JS). Для браузера может быть более производительно (с точки зрения страниц/пользовательского интерфейса) выполнять консольный ввод/вывод в фоновом режиме, и вы, возможно, даже не будете подозревать о нем.

Не слишком распространенная, но возможная ситуация, в которой можно понаблюдать за этим явлением (не из самого кода, а снаружи):

```
var a = {
    index: 1
};

// потом
console.log( a ); // ??

// еще позднее
a.index++;
```

Обычно мы ожидаем, что объект `a` будет зафиксирован в точный момент выполнения команды `console.log(..)` и будет выведен результат `{ index: 1 }`, чтобы в следующей команде при выполнении `a.index++` изменялось нечто иное, чем выводимое значение `a`.

В большинстве случаев приведенный выше код, скорее всего, выдаст в консоль инструментария разработчика представление объекта, чего вы и ожидаете. Но может оказаться, что тот же код будет выполняться в ситуации, в которой браузер сочтет нужным перевести консольный ввод/вывод в фоновый режим. И тогда может оказаться, что к тому времени, когда представление объекта будет выводиться в консоль браузера, увеличение `a.index++` уже произошло, и будет выведен результат `{ index: 2 }`.

На вопрос о том, при каких именно условиях консольный ввод/вывод будет отложен и будет ли вообще наблюдаться данный эффект, невозможно дать четкий ответ. Просто учитывайте возможную асинхронность при вводе/выводе в том случае, если у вас когда-нибудь возникнут проблемы с отладкой, когда объекты были изменены уже *после* команды `console.log(..)`, но эти изменения совершенно неожиданно проявляются при выводе.

Если вы столкнетесь с этой нечастой ситуацией, лучше всего использовать точки прерывания в отладчике JS, вместо того чтобы полагаться на консольный вывод. Следующий вариант — принудительно «зафиксировать» представление интересующего вас объекта, преобразовав его в строку (например, `JSON.stringify(..)`).

Цикл событий

Начну с утверждения (возможно, даже удивительного): несмотря на то что вы очевидным образом имели возможность писать асинхронный код JS (как в примере с тайм-аутом, рассмотренным выше), до последнего времени (ES6) в самом языке JavaScript никогда не было никакого встроенного понятия асинхронности.

Что?! Но это же полный бред? На самом деле это чистая правда. Сам движок JS никогда не делало ничего, кроме выполнения одного блока программы в любой конкретный момент времени, когда ему это приказывали. «Ему приказывали»... Кто приказывал? Это очень важный момент!

Движок JS не работает в изоляции. Он работает внутри управляющей среды, которой для большинства разработчиков становится обычный веб-браузер. За последние несколько лет (впрочем, не только за этот период) язык JS вышел за границы браузеров в другие среды — например, на серверы — благодаря таким технологиям, как Node.js. Более того, JavaScript сейчас встраивается в самые разные устройства, от роботов до лампочек.

Но у всех этих сред есть одна характерная особенность: у них существует механизм, который обеспечивает выполнение нескольких фрагментов вашей программы, обращаясь с вызовами к движку JS *в разные моменты времени*. Этот механизм называется *циклом событий*.

Иначе говоря, движок JS сам по себе не обладает внутренним чувством времени, но он становится средой исполнения для любого произвольного фрагмента JS. *Планирование* «событий» (выполнений фрагментов кода JS) всегда осуществляется окружающей средой.

Итак, например, когда ваша программа JS выдает запрос Ajax для получения данных с сервера, вы определяете код реакции в функции (обычно называемой функцией обратного вызова, или просто обратным вызовом), а движок JS говорит управляющей среде: «Так, я собираюсь ненадолго приостановить выполнение, но когда ты завершишь обработку этого сетевого запроса и получишь данные, пожалуйста, вызови *вот эту* функцию».

Браузер настраивается для прослушивания ответа от сети, и когда у него появятся данные, чтобы передать их программе, планирует выполнение функции обратного вызова, вставляя ее в цикл событий. Так что же такое «цикл событий»?

Следующий фиктивный код поможет представить суть цикла событий на концептуальном уровне:

```
// `eventLoop` - массив, работающий по принципу очереди
// (первым пришел, первым вышел)
var eventLoop = [ ];
var event;

// продолжать "бесконечно"
while (true) {
    // отработать "квант"
    if (eventLoop.length > 0) {
        // получить следующее событие в очереди
        event = eventLoop.shift();

        // выполнить следующее событие
        try {
            event();
        }
        catch (err) {
            reportError(err);
        }
    }
}
```

Конечно, этот сильно упрощенный псевдокод только демонстрирует основные концепции. Тем не менее и его должно быть достаточно для понимания сути.

Как видите, имеется непрерывно работающий цикл, представленный циклом `while`; каждая итерация цикла называется *тиком*. В каждом тике, если в очереди ожидает событие, оно, событие, извлекается и выполняется. Этими событиями становятся ваши функции обратного вызова.

Важно заметить, что функция `setTimeout(..)` не ставит обратный вызов в очередь цикла событий. Вместо этого она запускает таймер; по истечении таймера среда помещает ваш обратный вызов в цикл событий, чтобы некий тик в будущем подобрал его для выполнения.

А если в очереди в данный момент уже находятся 20 элементов? Вашему обратному вызову придется подождать. Он ставится в очередь после всех остальных — обычно не существует способа выйти вперед и обогнать других в очереди. Это объясняет, почему таймеры `setTimeout(..)` не гарантируют идеальной точности. Функция гарантирует (упрощенно говоря), что обратный вызов не сработает *до* истечения заданного вами интервала, но это может произойти в заданный момент или после него в зависимости от состояния очереди событий.

Другими словами, ваша программа обычно разбивается на множество мелких блоков, которые выполняются друг за другом в очереди цикла событий. И с технической точки зрения в эту очередь также могут попасть другие события, не имеющие прямого отношения к вашей программе.

Мы упомянули «до недавнего времени», говоря о том, как спецификация ES6 изменила природу управления очередью цикла событий. В основном это формальная техническая деталь, но ES6 теперь точно указывает, как работает цикл событий; это означает, что формально он попадает в сферу действия движка JS, а не только управляющей среды. Одна из главных причин для такого изменения — появление в ES6 обещаний (см. главу 3), для которых необходим прямой, точный контроль за операциями планирования очереди цикла событий (вызов `setTimeout(..0)` рассматривается в разделе «Кооперация»).

Параллельные потоки

Термины «асинхронный» и «параллельный» очень часто используются как синонимы, но в действительности они имеют разный смысл. Напомню, что суть асинхронности — управление промежутком между «сейчас» и «потом». Параллелизм обозначает возможность одновременного выполнения операций.

Самые распространенные средства организации параллельных вычислений — *процессы* и *потоки* (threads). Процессы и потоки

выполняются независимо и могут выполняться одновременно на разных процессорах и даже на разных компьютерах, но несколько потоков могут совместно использовать общую память одного процесса.

С другой стороны, цикл событий разбивает свою работу на задачи и выполняет их последовательно, что делает невозможным параллельный доступ и изменения в общей памяти. Параллелизм и последовательность не могут совместно существовать в форме взаимодействующих циклов событий в разных потоках.

Чередование параллельных потоков выполнения и чередование асинхронных событий происходит на совершенно разных уровнях детализации.

Пример:

```
function later() {
    answer = answer * 2;
    console.log( "Meaning of life:", answer );
}
```

Хотя все содержимое `later()` будет рассматриваться как один элемент очереди цикла событий, в потоке, в котором будет выполняться этот код, произойдет более десятка низкоуровневых операций. Например, для команды `answer = answer * 2` нужно будет сначала загрузить текущее значение `answer`, потом поместить куда-то `2`, затем выполнить умножение, затем взять результат и снова сохранить его в `answer`.

В однопоточной среде неважно, что элементы очереди потока являются низкоуровневыми операциями, потому что ничто не может прервать поток. Но в параллельной системе, в которой в одной программе могут выполняться два разных потока, легко могут возникнуть непредсказуемые последствия.

Пример:

```
var a = 20;

function foo() {
```

```
    a = a + 1;
}

function bar() {
    a = a * 2;
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

При однопоточном поведении в JavaScript, если `foo()` выполняется перед `bar()`, в результате `a` будет содержать `42`, а если `bar()` выполняется перед `foo()`, то результат будет равен `41`.

Но если события JS, совместно использующие одни данные, выполняются параллельно, проблемы становятся намного более коварными. Возьмем два списка задач из псевдокода как потоки, которые могут выполнять код `foo()` и `bar()` соответственно, и посмотрим, что произойдет, если они выполняются ровно в одно и то же время.

Поток 1 (X и Y — временные области памяти):

```
foo():
  a. загрузить значение `a` в `X`
  b. сохранить `1` в `Y`
  c. сложить `X` и `Y`, сохранить результат в `X`
  d. сохранить значение `X` в `a`
```

Поток 2 (X и Y — временные области памяти):

```
bar():
  a. загрузить значение `a` в `X`
  b. сохранить `2` в `Y`
  c. перемножить `X` и `Y`, сохранить результат в `X`
  d. сохранить значение `X` в `a`
```

Теперь предположим, что два потока выполняются действительно параллельно. Вы уже заметили проблему, верно? Они ис-

пользуют области общей памяти X и Y для своих промежуточных операций.

Какой результат будет сохранен в a, если операции будут выполняться в следующем порядке?

```
1a  (загрузить значение `a` в `X`   ==> `20`)
2a  (загрузить значение `a` в `X`   ==> `20`)
1b  (сохранить `1` в `Y`   ==> `1`)
2b  (сохранить `2` в `Y`   ==> `2`)
1c  (сложить `X` и `Y`, сохранить результат в `X`   ==> `22`)
1d  (сохранить значение `X` в `a`   ==> `22`)
2c  (перемножить `X` и `Y`, сохранить результат в `X`   ==> `44`)
2d  (сохранить значение `X` в `a`   ==> `44`)
```

Значение a будет равно 44. А как насчет такого порядка?

```
1a  (загрузить значение `a` в `X`   ==> `20`)
2a  (загрузить значение `a` в `X`   ==> `20`)
2b  (сохранить `2` в `Y`   ==> `2`)
1b  (сохранить `1` в `Y`   ==> `1`)
2c  (перемножить `X` и `Y`, сохранить результат в `X`   ==> `20`)
1c  (сложить `X` и `Y`, сохранить результат в `X`   ==> `21`)
1d  (сохранить значение `X` в `a`   ==> `21`)
2d  (сохранить значение `X` в `a`   ==> `21`)
```

На этот раз значение a будет равно 21.

Итак, многопоточное программирование может быть очень сложным, потому что если вы не предпримете специальных мер для предотвращения подобных прерываний/чередований, возможно очень странное недетерминированное поведение, которое часто создает проблемы.

В JavaScript совместное использование данных разными потоками невозможно, что означает, что этот уровень недетерминизма не создаст проблем. Но это не означает, что поведение JS всегда детерминировано. Вспомните предыдущий пример, в котором относительный порядок foo() и bar() мог приводить к двум разным результатам (41 или 42).

Возможно, это еще неочевидно, но недетерминизм не всегда плох. Иногда он обходится без последствий, иногда вводится намеренно. В этой и следующих главах вам встретятся другие примеры.

Выполнение до завершения

Из-за однопоточного характера JavaScript код внутри функций `foo()` (и `bar()`) выполняется атомарно; это означает, что после того, как функция `foo()` начнет выполняться, весь ее код будет завершен до того, как будет выполнен какой-либо код `bar()`, и наоборот. Это поведение называется *выполнением до завершения* (run-to-completion). Собственно, семантика выполнения до завершения становится более очевидной, когда `foo()` и `bar()` содержат больше кода:

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Поскольку `foo()` не может прерываться `bar()`, а `bar()` не может прерываться `foo()`, у этой программы есть только два возможных результата в зависимости от того, какая из функций запустится первой. Если присутствует многопоточное выполнение, а отдель-