РАЗРАБОТКА ВЕБ-САЙТОВ*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

Как работает JS: цикл событий, асинхронность и пять способов улучшения кода с помощью async / await

ПЕРЕВОД

ru_vds 19 октября 2017 в 15:02 39,2k

Оригинал: Alexander Zlatkov

Часть 1: Как работает JS: обзор движка, механизмов времени выполнения, стека вызовов
Часть 2: Как работает JS: о внутреннем устройстве V8 и оптимизации кода
Часть 3: Как работает JS: управление памятью, четыре вида утечек памяти и борьба с ними

Перед вами четвёртая часть серии материалов, посвящённых внутренним особенностям работы JavaScript. Эти материалы, с одной стороны, направлены на изучение базовых элементов языка и экосистемы JS, с другой, содержат рекомендации, основанные на практике разработки ПО в компании SessionStack. Конкурентоспособное JS-приложение должно быть быстрым и надёжным. Создание таких приложений — та цель, к которой, в конечном счёте, стремится любой, интересующийся механизмами JavaScript.

Эту статью можно считать развитием первого материала. Здесь будут рассмотрены ограничения однопоточной модели выполнения, и пойдёт речь о том, как эти ограничения преодолеть. Например — для разработки качественных пользовательских интерфейсов на JS. Как обычно, в конце материала будут приведены практические рекомендации по использованию рассмотренных технологий. Так как основная тема этой статьи — асинхронная разработка, это будут пять советов по улучшению кода с использованием `async / await`.

Ограничения однопоточной модели выполнения кода

В первом материале мы размышляли над следующим вопросом: «Что происходит, когда в стеке вызовов имеется функция, на выполнение которой нужно очень много времени?». Продолжим эти размышления.

Представьте себе сложный алгоритм обработки изображений, реализация которого запускается в браузере. Когда в стеке

вызовов есть работающая функция, браузер не может делать больше ничего. Он заблокирован. Это означает, что браузер не может выводить что-либо на экран, не может выполнять другой код. Самое заметное последствие такой ситуации — «тормоза» пользовательского интерфейса. Веб-страницу попросту «заклинивает».

В некоторых случаях это может оказаться не такой уж и серьёзной проблемой. Однако, это — не самое страшное. Как только браузер начинает выполнять слишком много задач, он может достаточно продолжительное время не реагировать на воздействия пользователя. Обычно в подобной ситуации браузеры предпринимают защитные меры и выдают сообщение об ошибке, спрашивая у пользователя, следует ли закрыть проблемную страницу. Лучше бы пользователю не видеть таких сообщений. Они полностью разрушают все усилия разработчиков по созданию красивых и удобных интерфейсов.

Как же быть, если хочется, чтобы веб-приложение и выглядело хорошо, и могло выполнять сложные вычисления? Начнём поиск ответа на этот вопрос с анализа строительных блоков JS-приложений.

Строительные блоки программ на JavaScript

Код JavaScript-приложения можно разместить в одном .js-файле, но оно, почти наверняка, будет состоять из нескольких блоков. При этом лишь один из этих блоков будет выполняться в некий определённый момент времени, скажем — прямо сейчас. Остальные будут выполняться позже. Наиболее распространённые блоки кода в JavaScript — это функции.

Судя по всему, большинство новых JS-разработчиков не вполне осознают тот факт, что «позже» не обязательно немедленно

следует за «сейчас». Другими словами, задача, которую невозможно выполнить «сейчас», должна быть выполнена асинхронно. Это означает, что мы не столкнёмся с блокирующим поведением, которого вы, возможно неосознанно, могли ожидать.

Взглянем на следующий пример:

```
// ajax(..) - некая библиотечная Ajax-функция
var response = ajax('https://example.com/api');

console.log(response);
// в переменной response не будет ответа от api
```

Возможно, вы знаете о том, что стандартные Ajax-запросы не выполняются синхронно. Это означает, что функция `ajax(..)`, сразу после её вызова, не может возвратить некое значение, которое могло бы быть присвоено переменной `response`.

Простой механизм организации «ожидания» результата, возвращаемого асинхронной функцией, заключается в использовании так называемых функций обратного вызова, или коллбэков:

```
ajax('https://example.com/api', function(response) {
    console.log(response); // теперь переменная response содержит
ответ api
});
```

Тут хотелось бы отметить, что выполнить Ajax-запрос можно и синхронно. Однако, так делать не следует. Если выполнить синхронный Ajax-запрос, пользовательский интерфейс JS-

приложения окажется заблокированным. Пользователь не сможет щёлкнуть по кнопке, ввести данные в поле, он не сможет даже прокрутить страницу. Синхронное выполнение Ajax-запросов не даст пользователю взаимодействовать с приложением. Такой подход хотя и возможен, приводит к катастрофическим последствиям.

Вот как этот ужас выглядит, однако, пожалуйста, никогда не пишите ничего подобного, не превращайте веб в место, где невозможно нормально работать:

```
// Исходим из предположения, что вы пользуетесь jQuery
jQuery.ajax({
    url: 'https://api.example.com/endpoint',
    success: function(response) {
        // Это - коллбэк.
    },
    async: false // А вот это - то, чего делать настоятельно не
рекомендуется
});
```

Здесь мы приводили примеры на Ajax-запросах, однако, любой фрагмент кода можно запустить в асинхронном режиме.

Например, сделать это можно с помощью функции `setTimeout(callback, milliseconds)`. Она позволяет планировать выполнение событий (задавая тайм-аут), которые должны произойти позже момента обращения к этой функции. Рассмотрим пример:

```
function first() {
    console.log('first');
```

```
}
function second() {
    console.log('second');
}
function third() {
    console.log('third');
}
first();
setTimeout(second, 1000); // вызвать функцию second через 1000
миллисекунд
third();
```

Вот что этот код выведет в консоль:

```
first
third
second
```

Изучение цикла событий

Это может показаться странным, но до ES6 JavaScript, несмотря на то, что он позволял выполнять асинхронные вызовы (вроде вышеописанного `setTimeout`), не содержал встроенных механизмов асинхронного программирования. JS-движки занимались только однопоточным выполнением неких фрагментов кода, по одному за раз.

Для того, чтобы узнать подробности о том, как работают JavaScript-движки (и, в частности, V8), взгляните на этот материал.

Итак, кто сообщает JS-движку о том, что он должен исполнить некий фрагмент программы? В реальности движок не работает в изоляции — его собственный код выполняется внутри некоего

окружения, которым, для большинства разработчиков, является либо браузер, либо Node.js. На самом деле, в наши дни существуют JS-движки для самых разных видов устройств — от роботов до умных лампочек. Каждое подобное устройство представляет собственный вариант окружения для JS-движка.

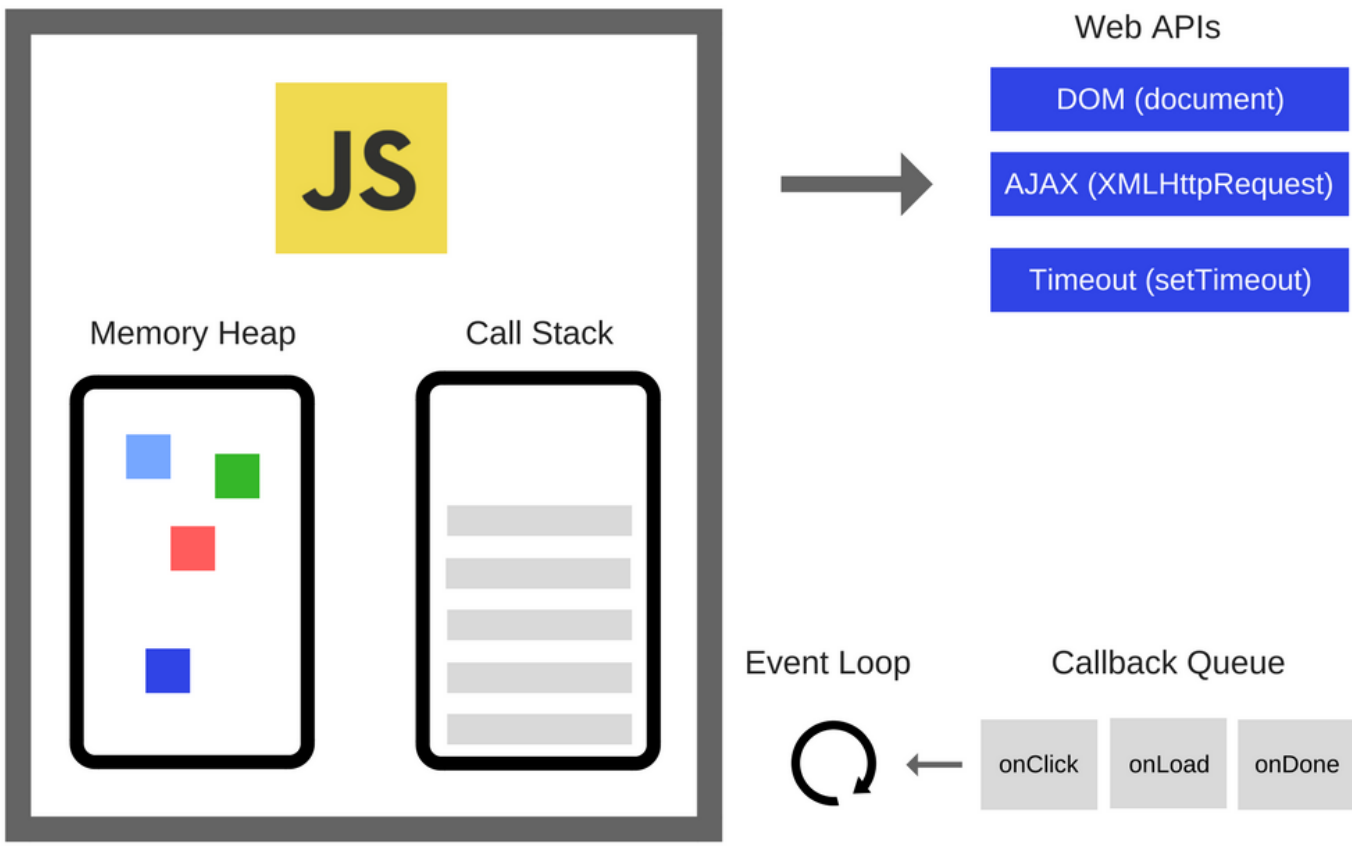
Общей характеристикой всех подобных сред является встроенный механизм, который называется циклом событий (event loop). Он поддерживает выполнение фрагментов программы, вызывая для этого JS-движок.

Это означает, что движок можно считать средой выполнения любого JS-кода, вызываемой по требованию. А планированием событий (то есть — сеансов выполнения JS-кода) занимаются механизмы окружения, внешние по отношению к движку.

Итак, например, когда ваша программа выполняет Ajax-запрос для загрузки каких-то данных с сервера, вы пишете команду для записи этих данных в переменную `response` внутри коллбэка, и JS-движок сообщает окружению: «Послушай, я собираюсь приостановить выполнение программы, но когда ты закончишь выполнять этот сетевой запрос и получишь какие-то данные, пожалуйста, вызови этот коллбэк».

Затем браузер устанавливает прослушиватель, ожидающий ответ от сетевой службы, и когда у него есть что-то, что можно возвратить в программу, выполнившую запрос, он планирует вызов коллбэка, добавляя его в цикл событий.

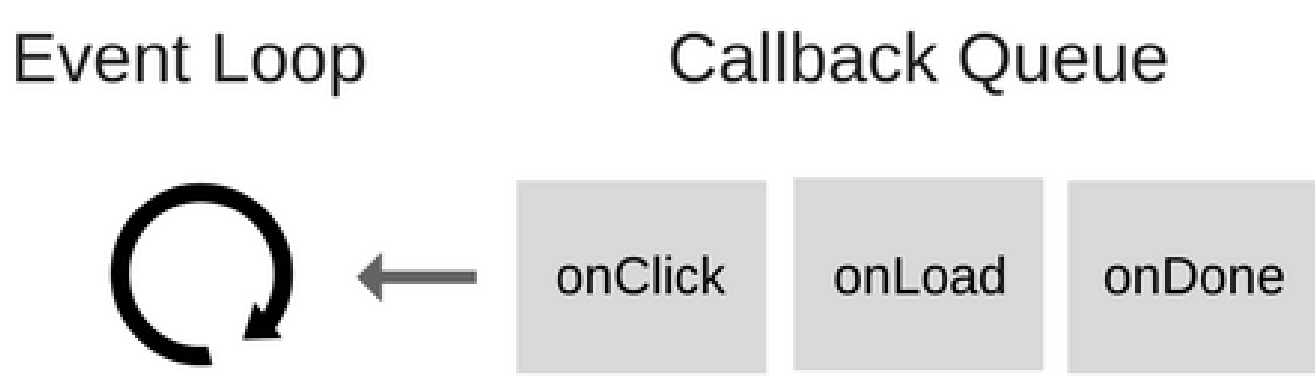
Взгляните на следующую схему:



Подробности о куче (memory heap) и стеке вызовов (call stack) можно найти здесь. А что представляют собой Web API? В целом, это — потоки, к которым у нас нет прямого доступа, мы можем лишь выполнять обращения к ним. Они встроены в браузер, где и выполняются асинхронные действия.

Если вы разрабатываете под Node.js, то подобные API реализованы средствами C++.

Итак, что же такое цикл событий?



Цикл событий решает одну основную задачу: наблюдает за стеком вызовов и очередью коллбэков (callback queue). Если стек вызовов пуст, цикл берёт первое событие из очереди и помещает его в стек, что приводит к запуску этого события на выполнение.

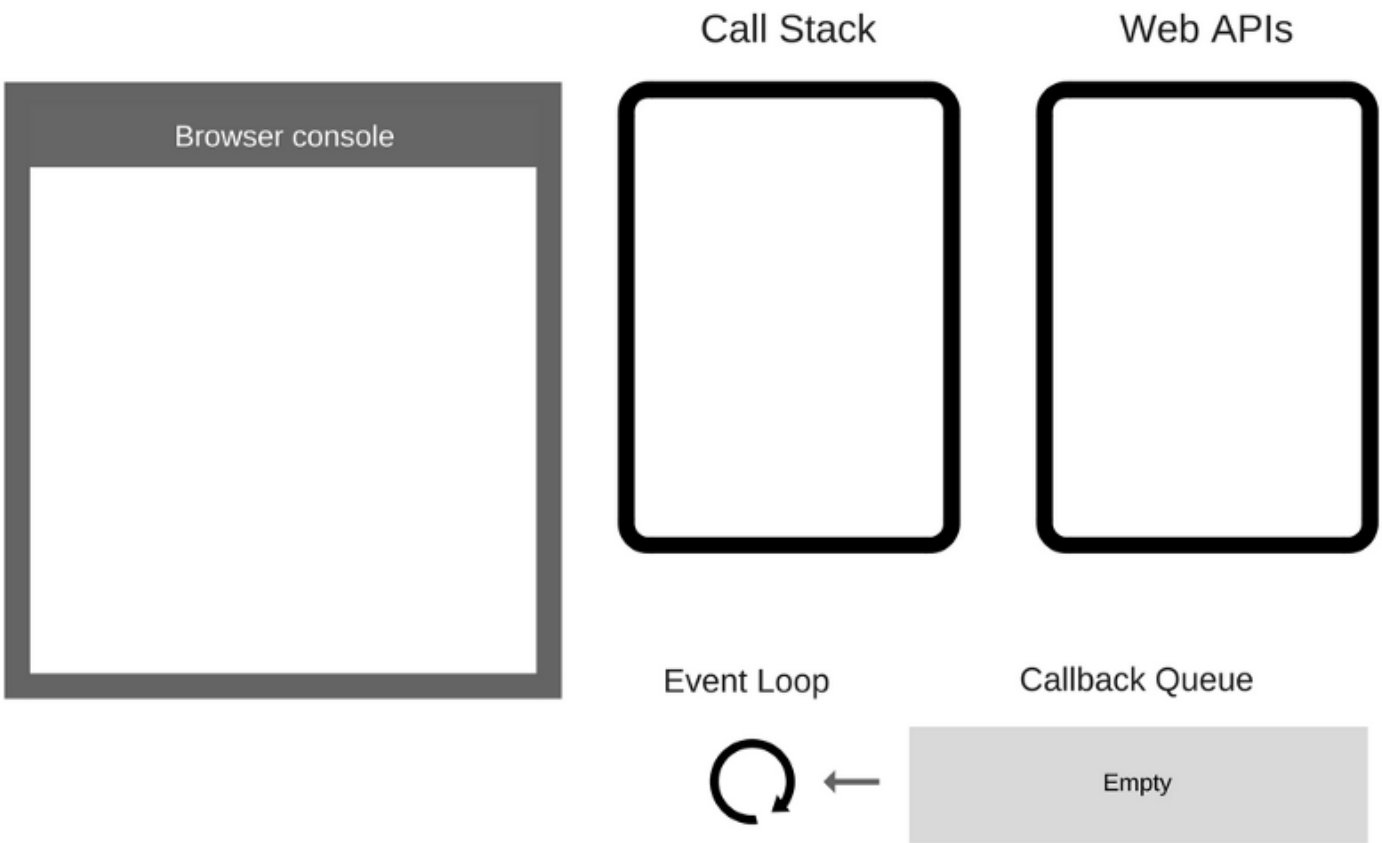
Подобная итерация называется тиком (tick) цикла событий. Каждое событие — это просто коллбэк.

Рассмотрим следующий пример:

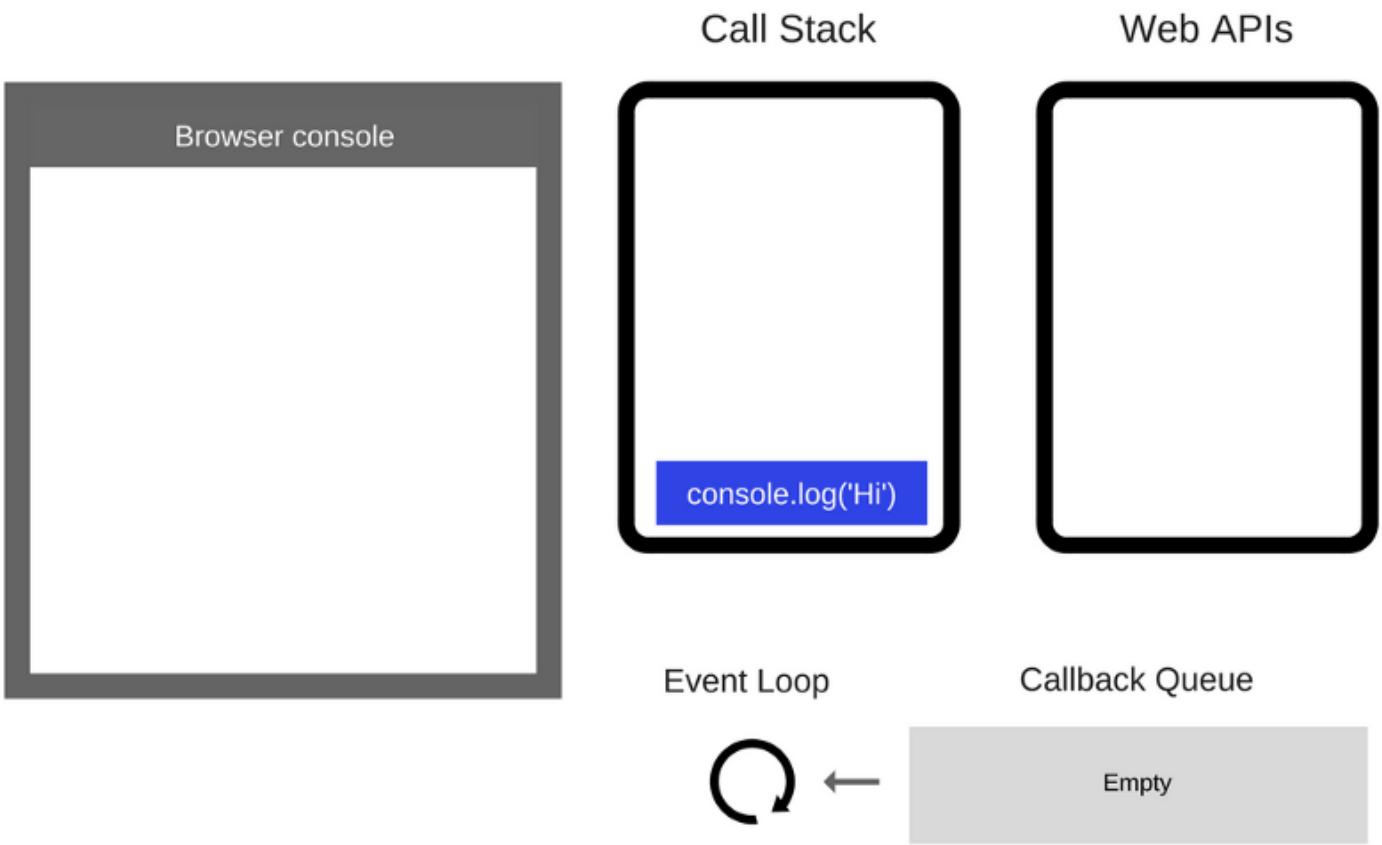
```
console.log('Hi');
setTimeout(function cb1() {
    console.log('cb1');
}, 5000);
console.log('Bye');
```

Займёмся пошаговым «выполнением» этого кода и посмотрим, что при этом происходит в системе.

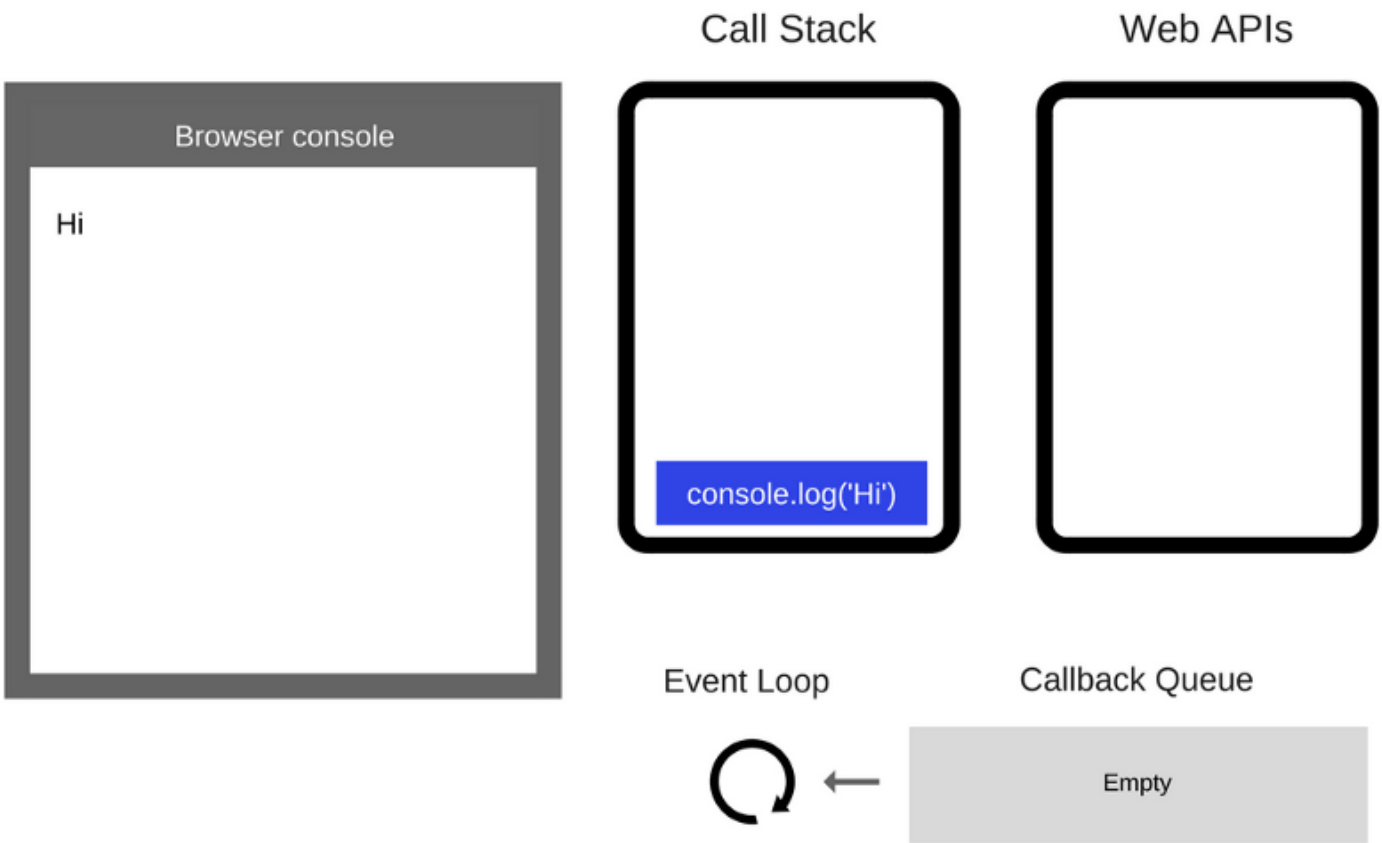
1. Пока ничего не происходит. Консоль браузера чиста, стек вызовов пуст.



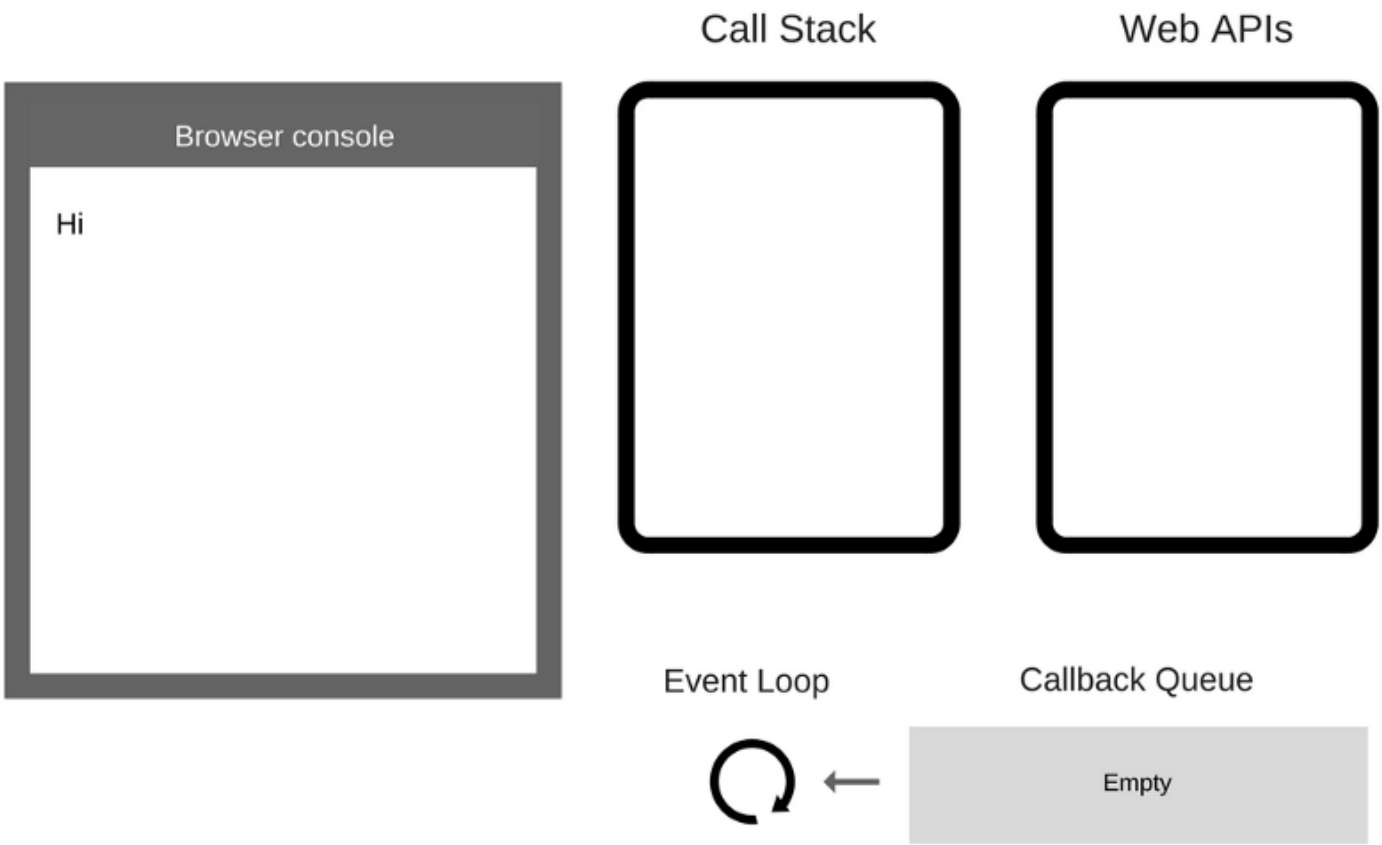
2. Команда `console.log('Hi')` добавляется в стек вызовов.



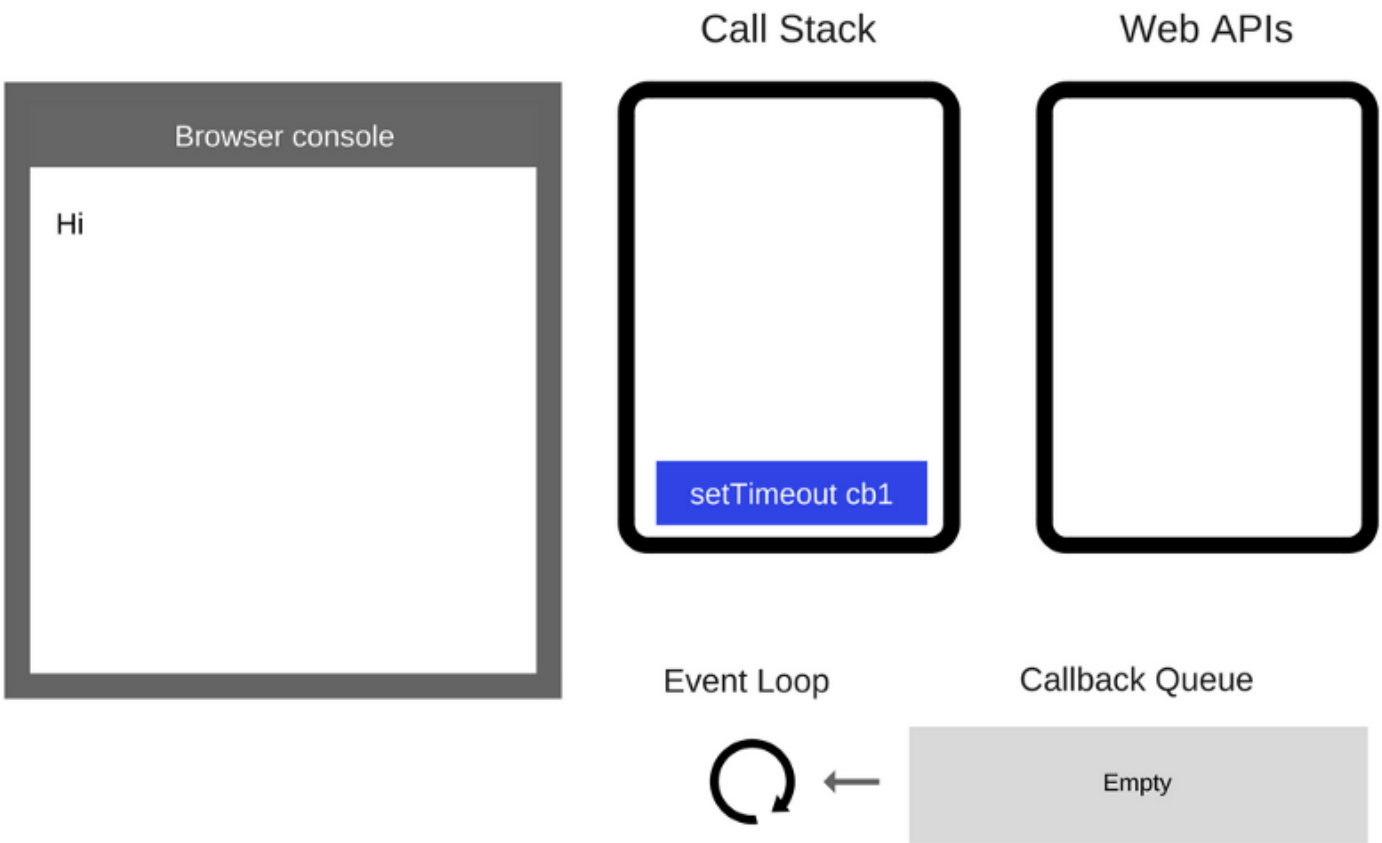
3. Команда `console.log('Hi')` выполняется.



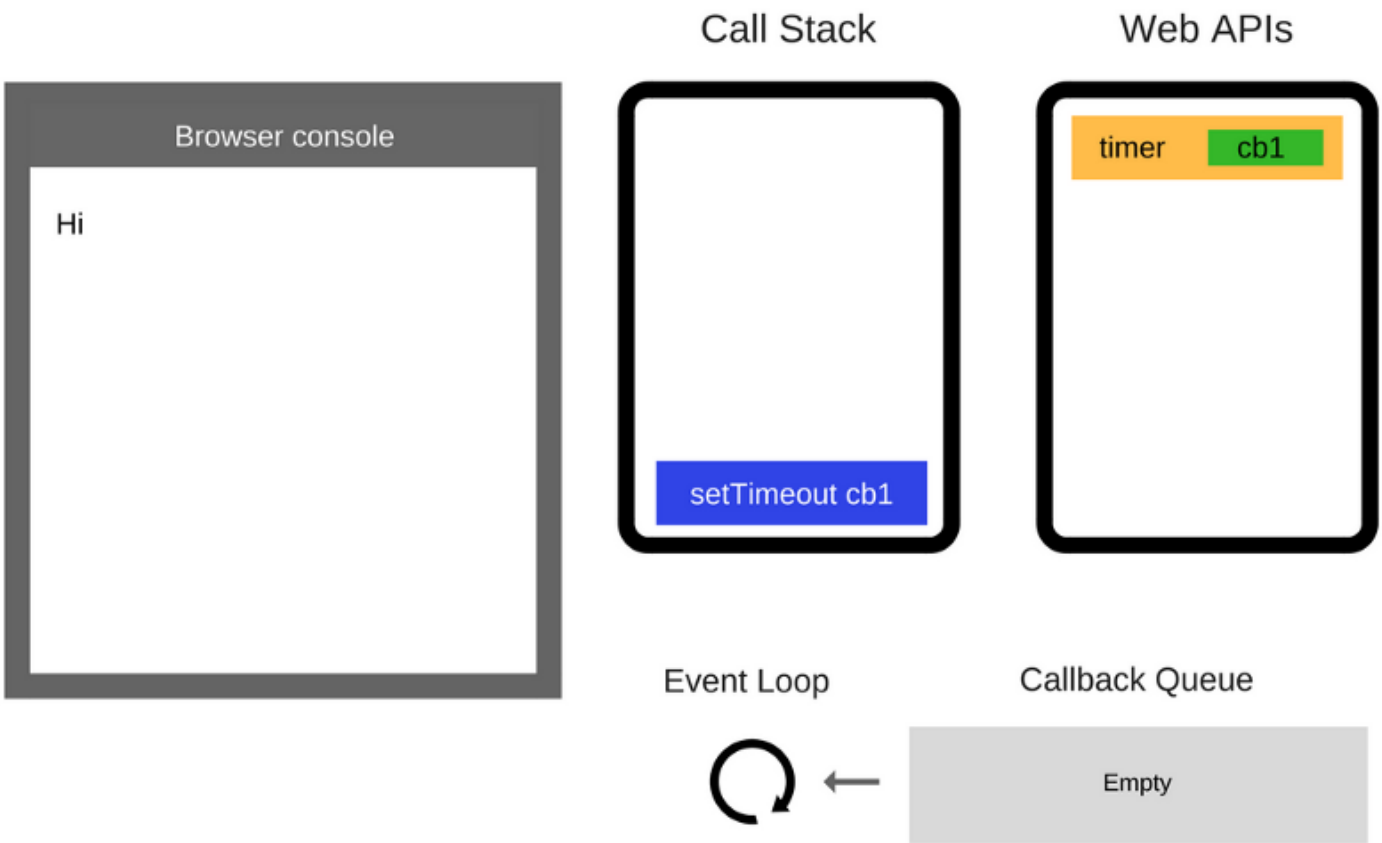
4. Команда `console.log('Hi')` удаляется из стека вызовов.



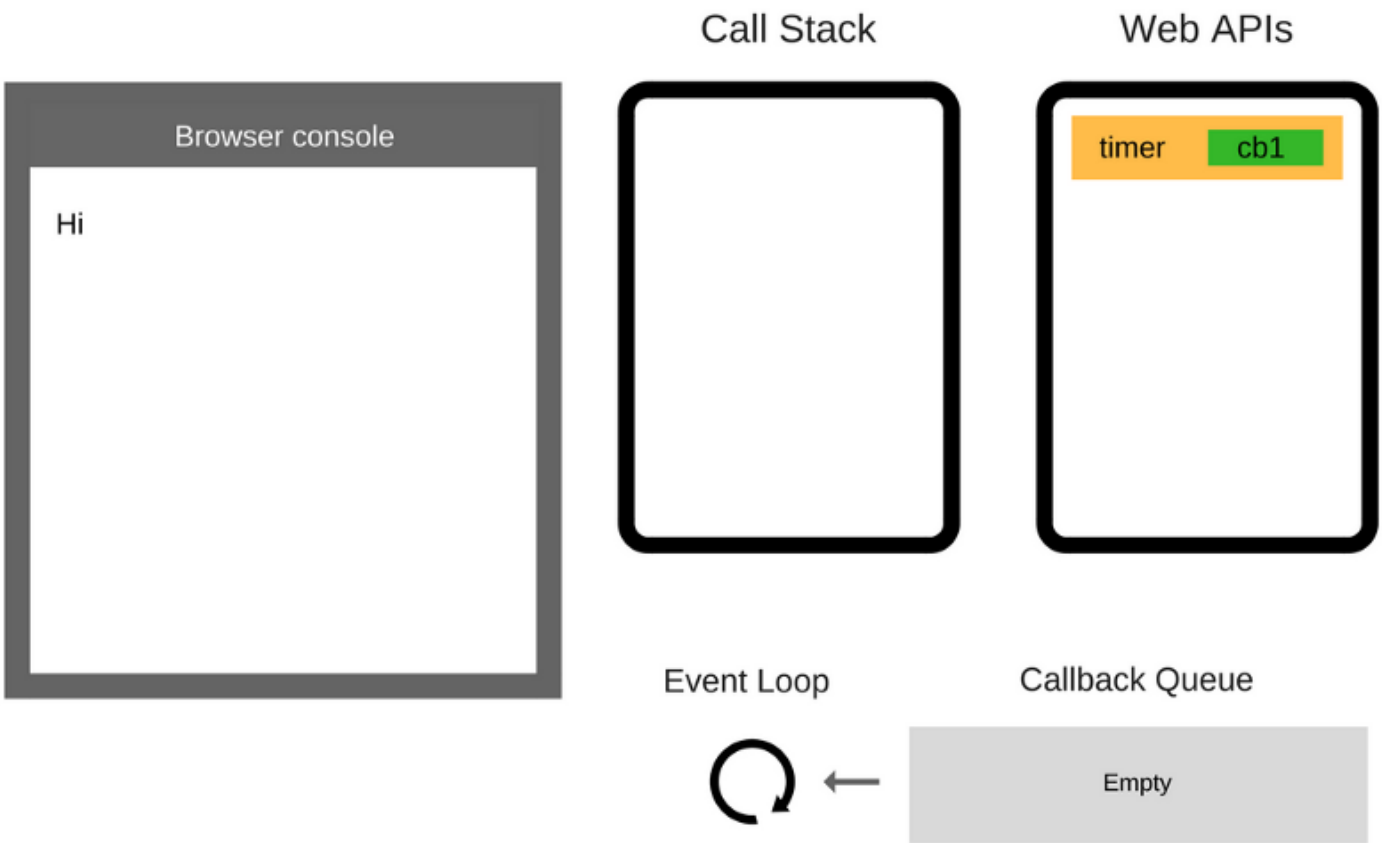
5. Команда `setTimeout(function cb1() { ... })` добавляется в стек вызовов.



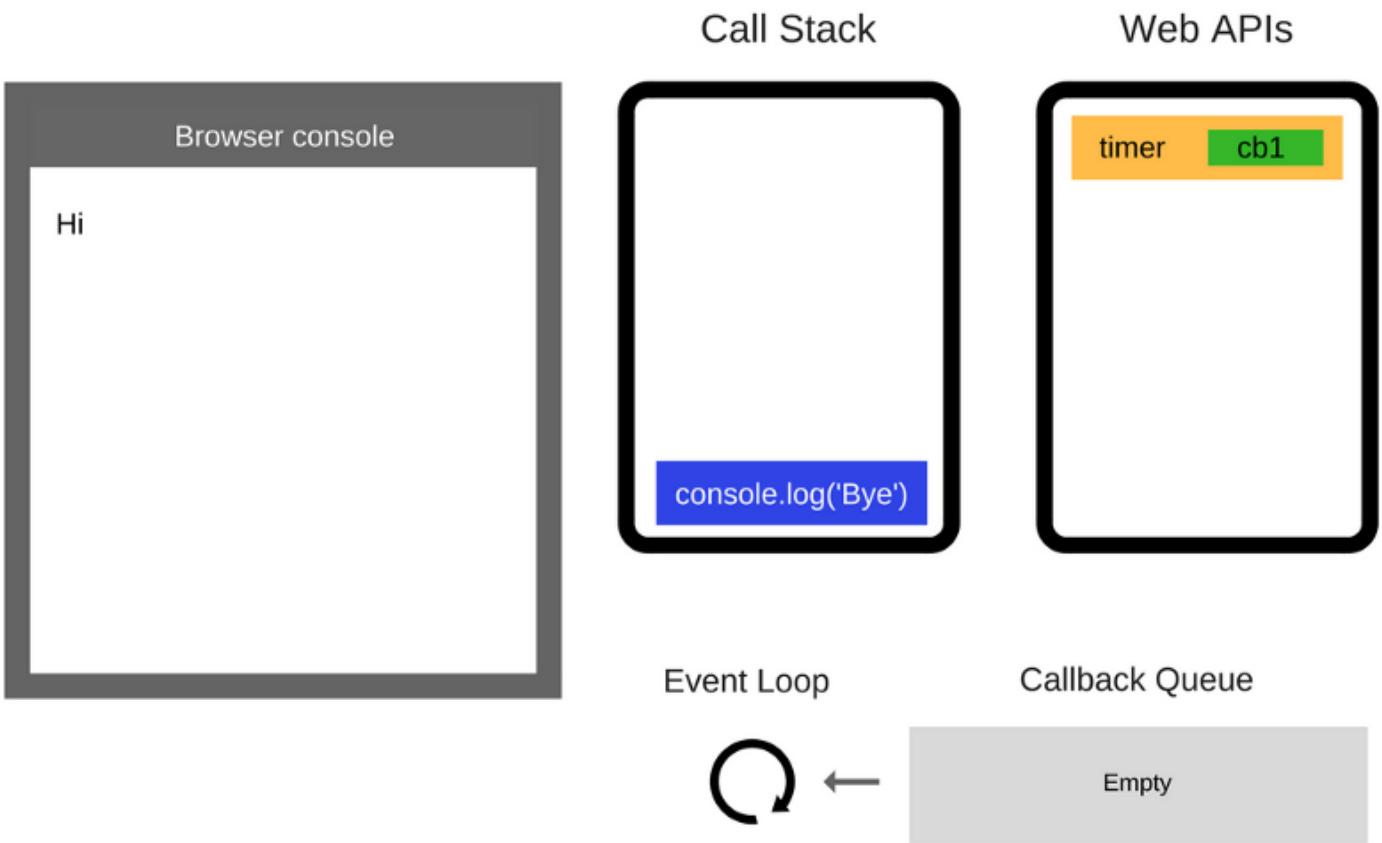
6. Команда `setTimeout(function cb1() { ... })` выполняется. Браузер создаёт таймер, являющийся частью Web API. Он будет выполнять обратный отсчёт времени.



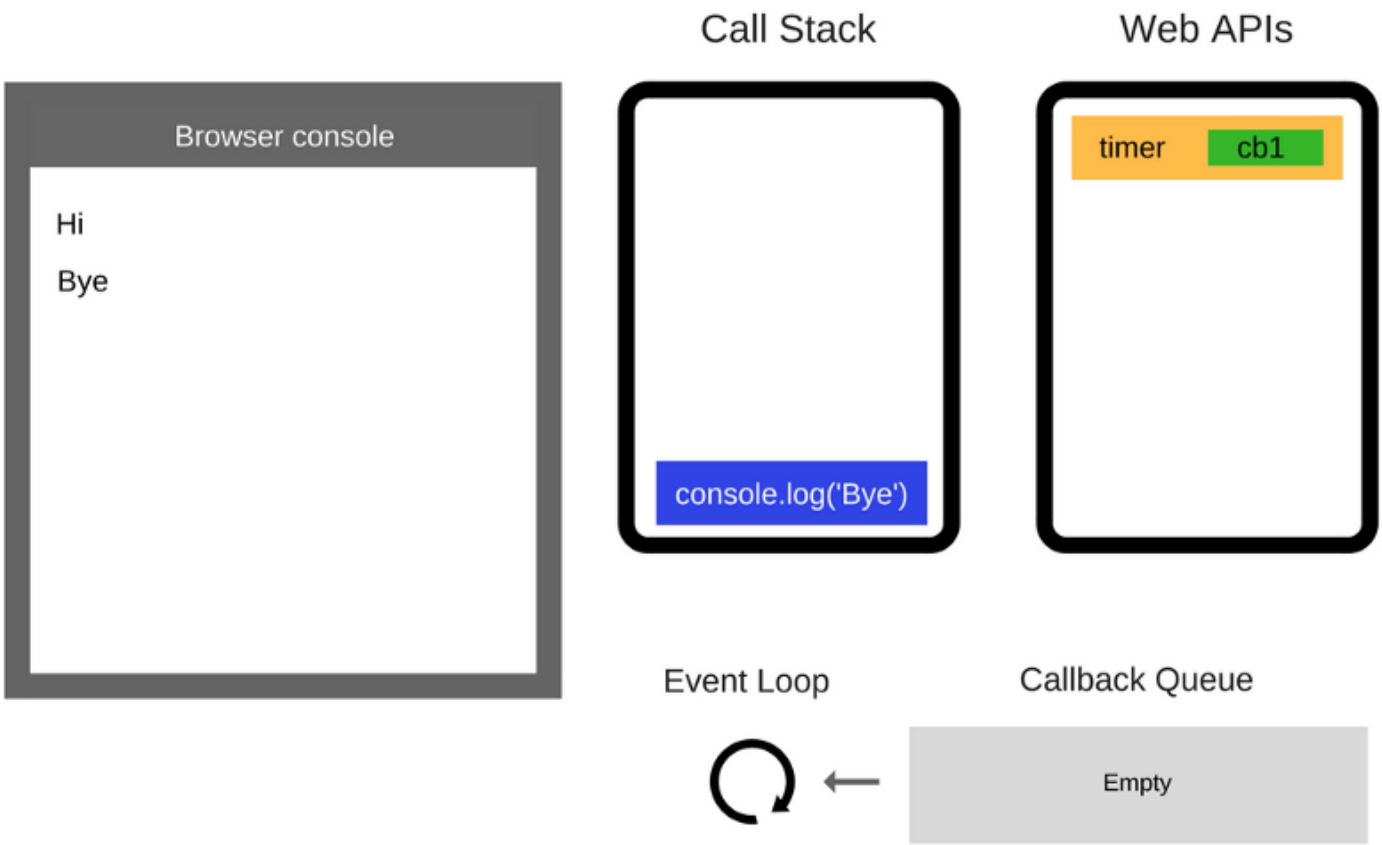
7. Команда `setTimeout(function cb1() { ... })` завершила работу и удаляется из стека вызовов.



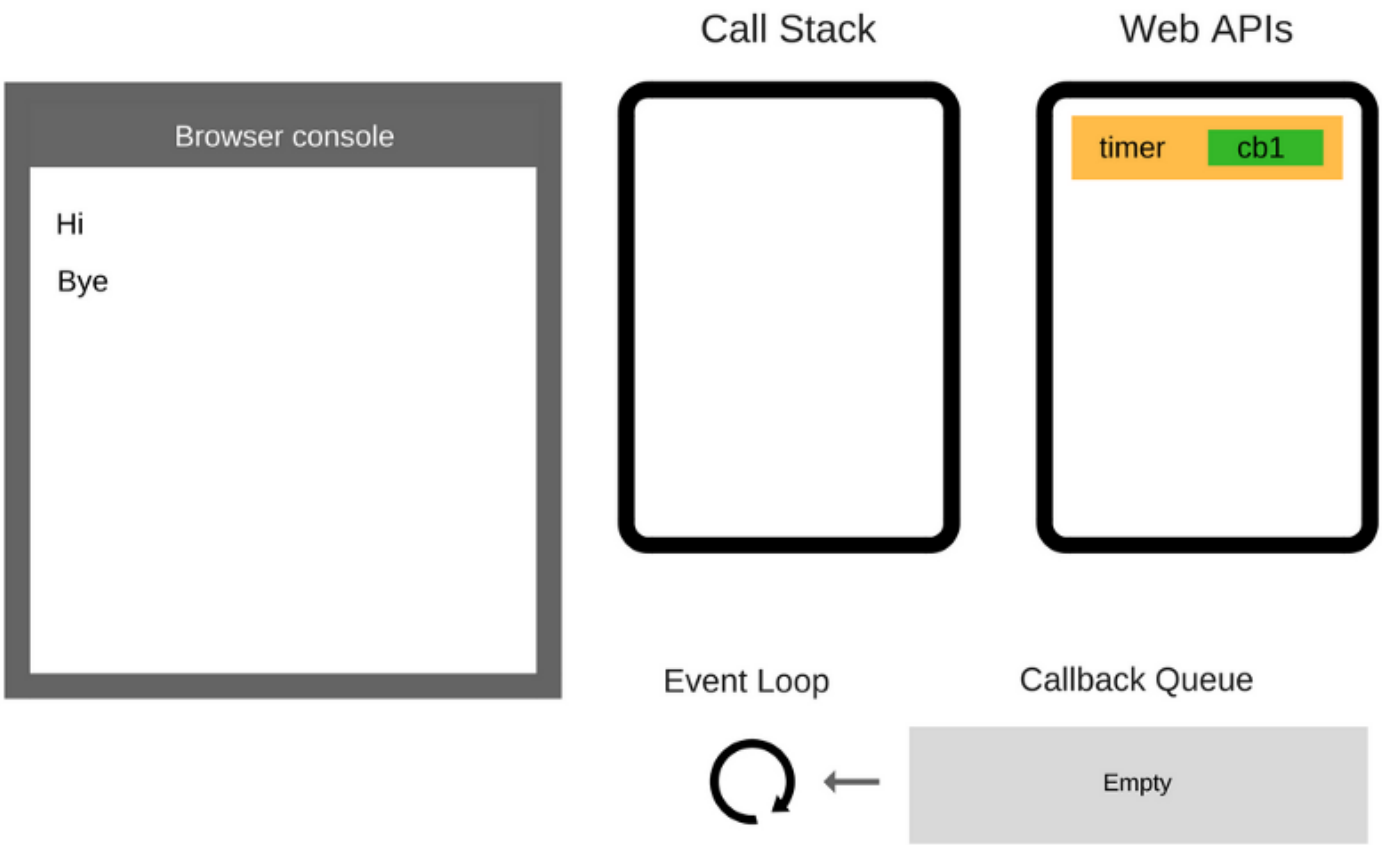
8. Команда `console.log('Bye')` добавляется в стек вызовов.



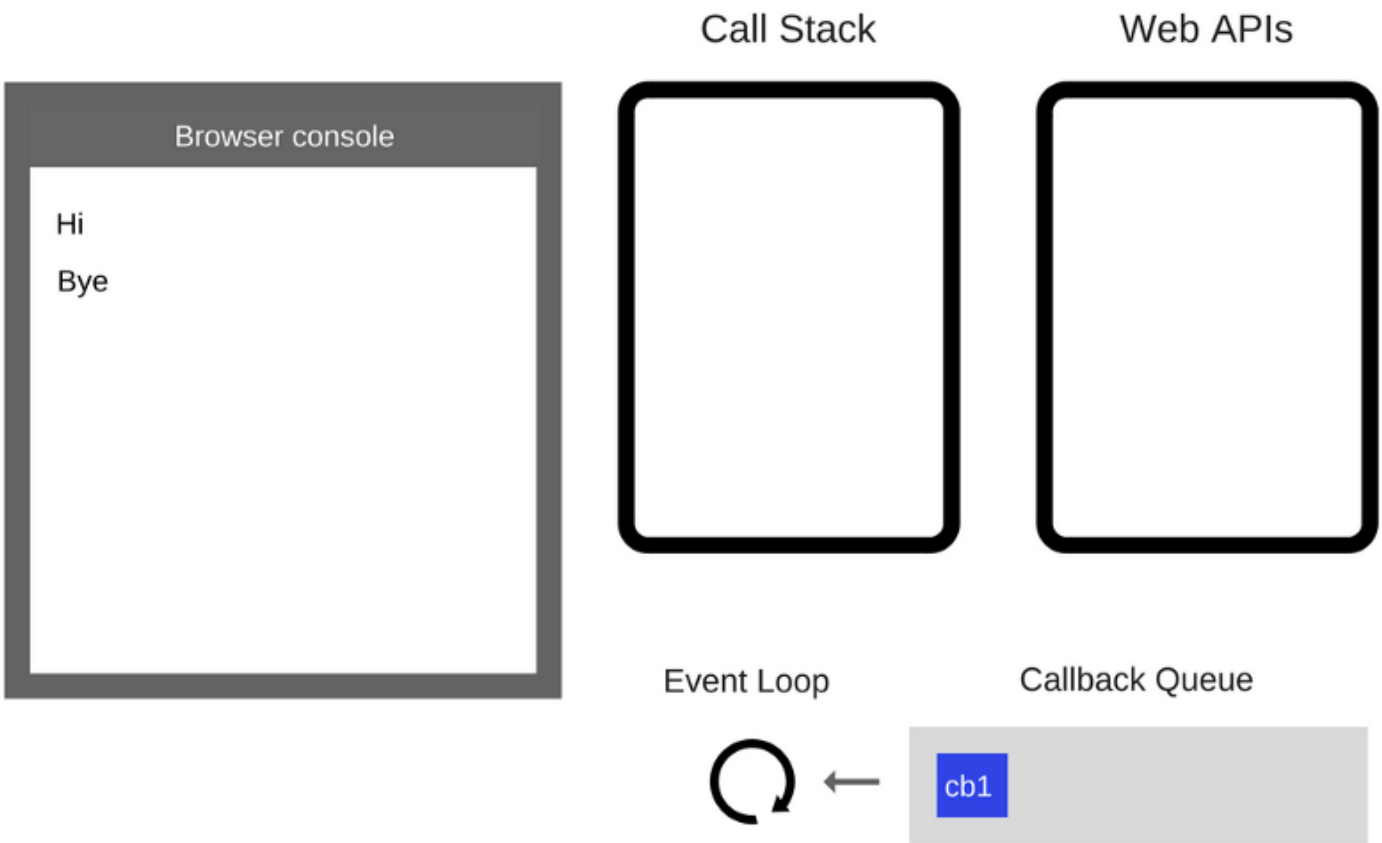
9. Команда `console.log('Bye')` выполняется.



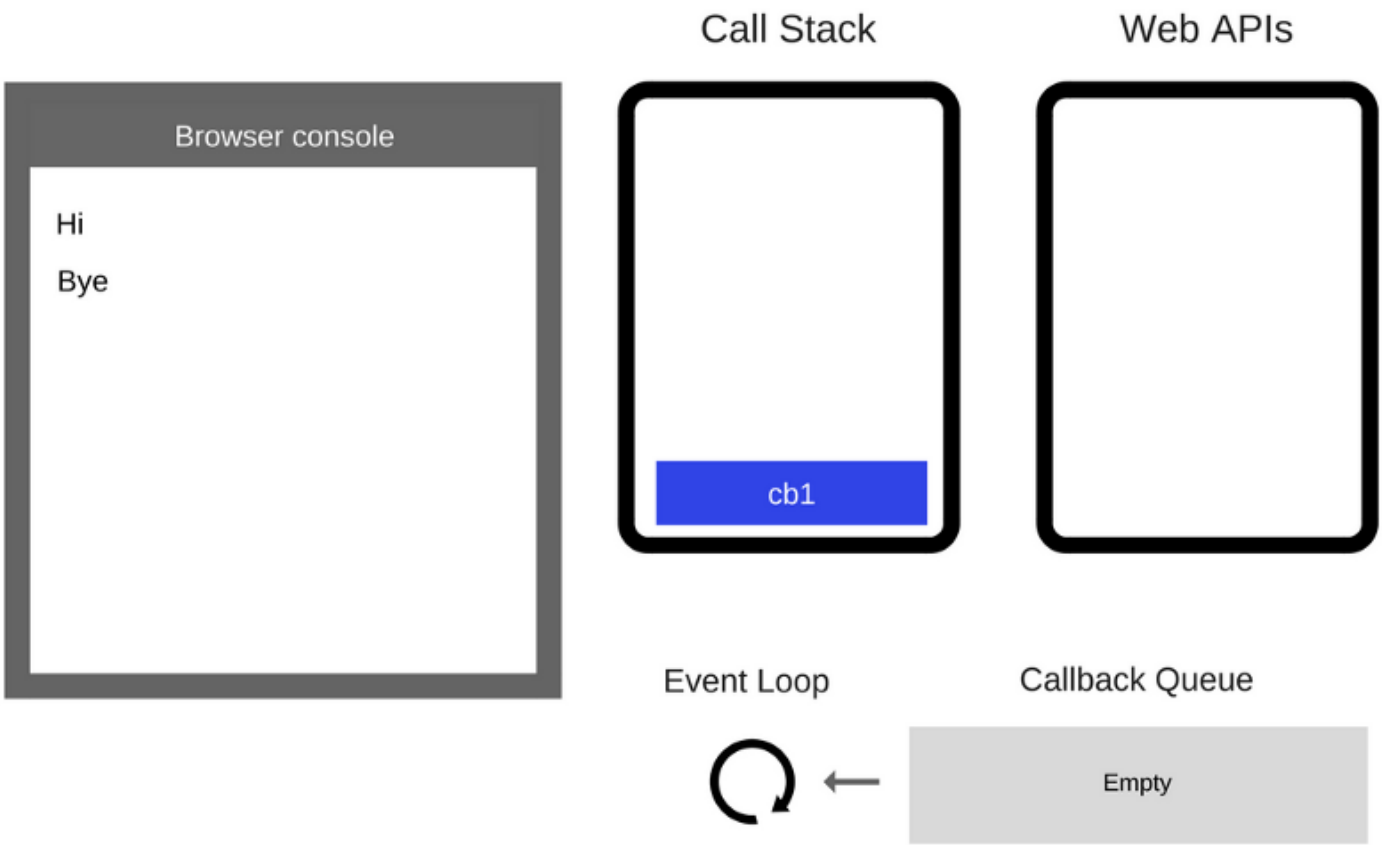
10. Команда `console.log('Bye')` удаляется из стека вызовов.



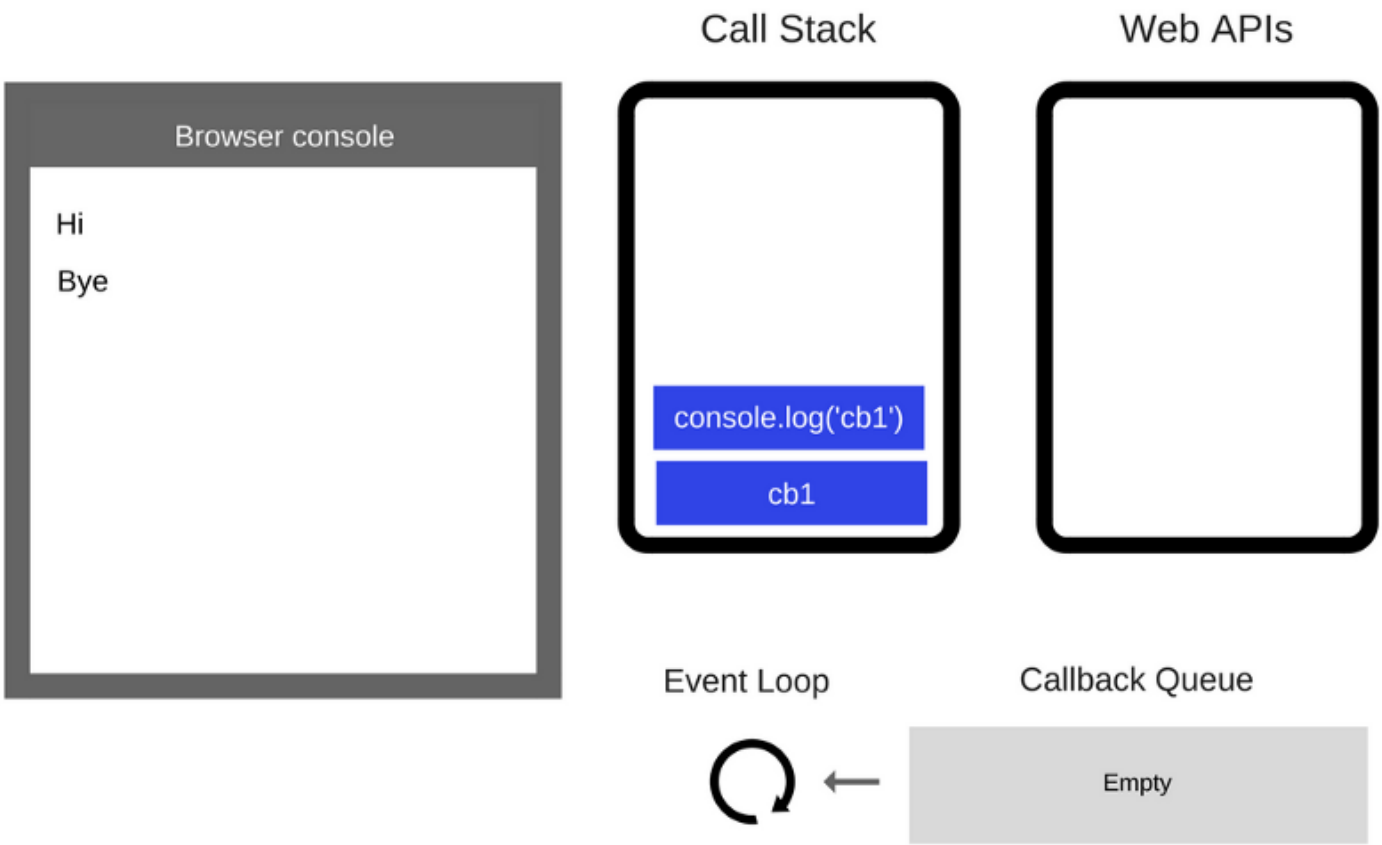
11. После того, как пройдут, как минимум, 5000 мс., таймер завершает работу и помещает коллбэк `cb1` в очередь коллбэков.



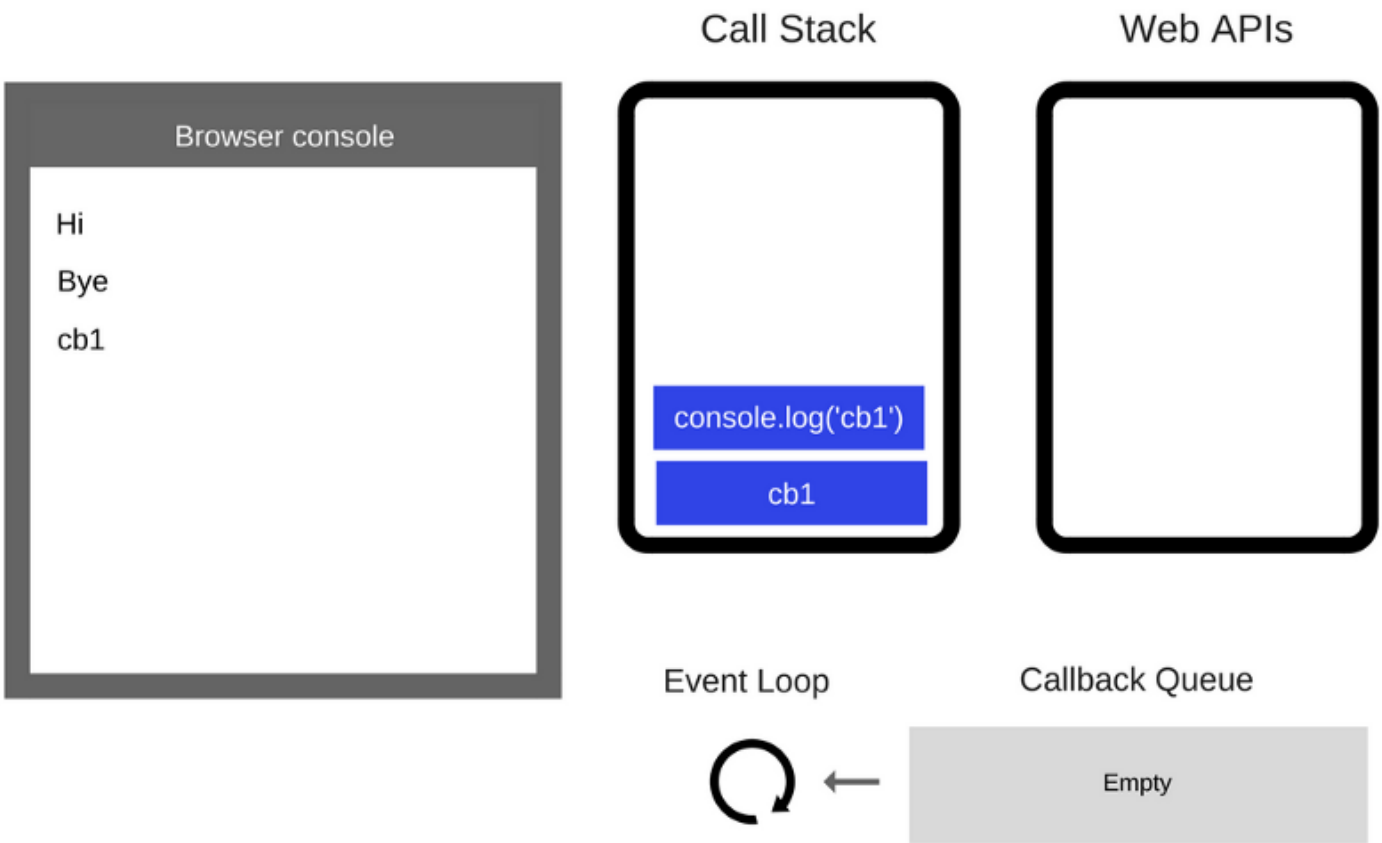
12. Цикл событий берёт с функцию `cb1` из очереди коллбэков и помещает её в стек вызовов.



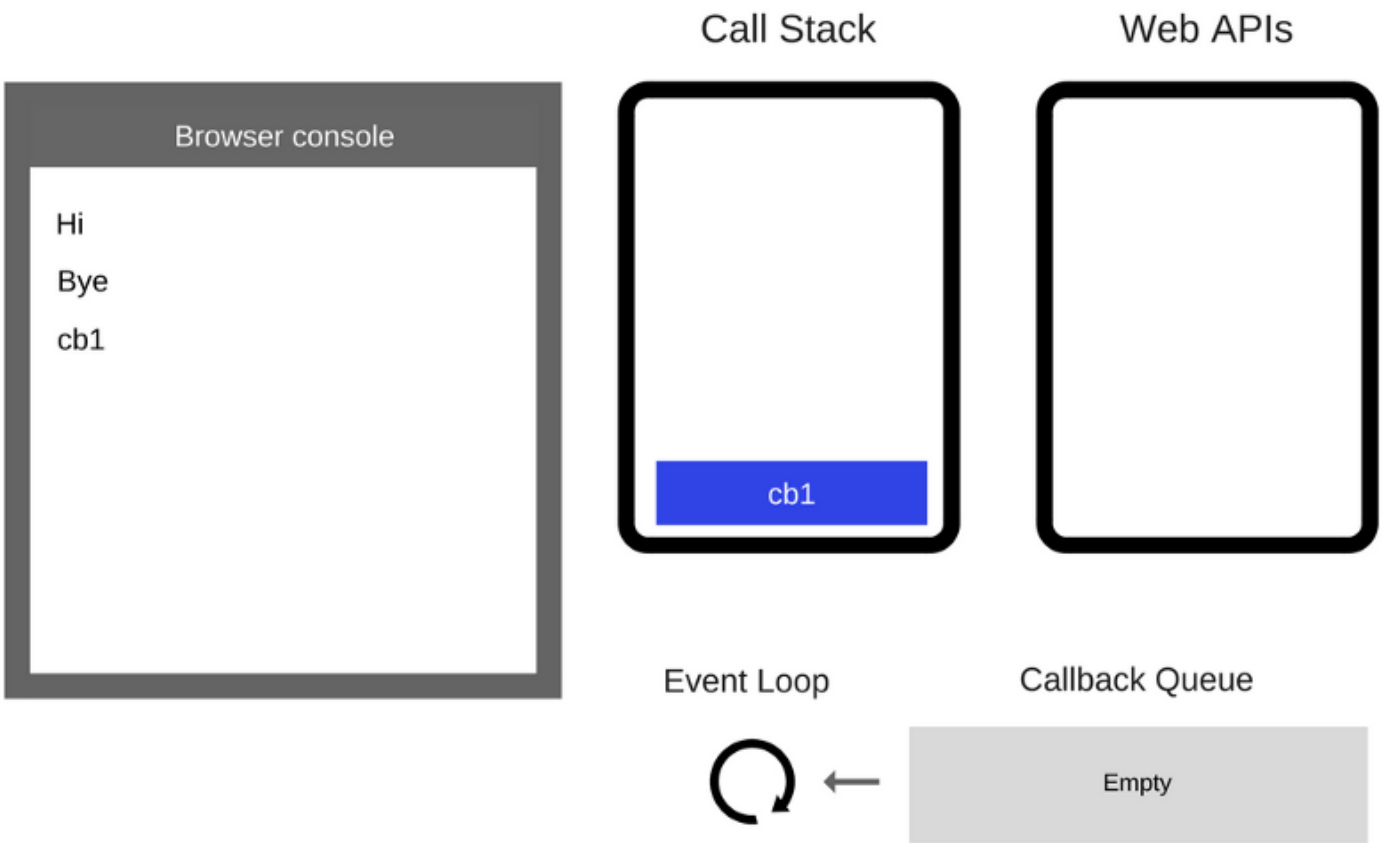
13. Функция `cb1` выполняется и добавляет `console.log('cb1')` в стек вызовов.



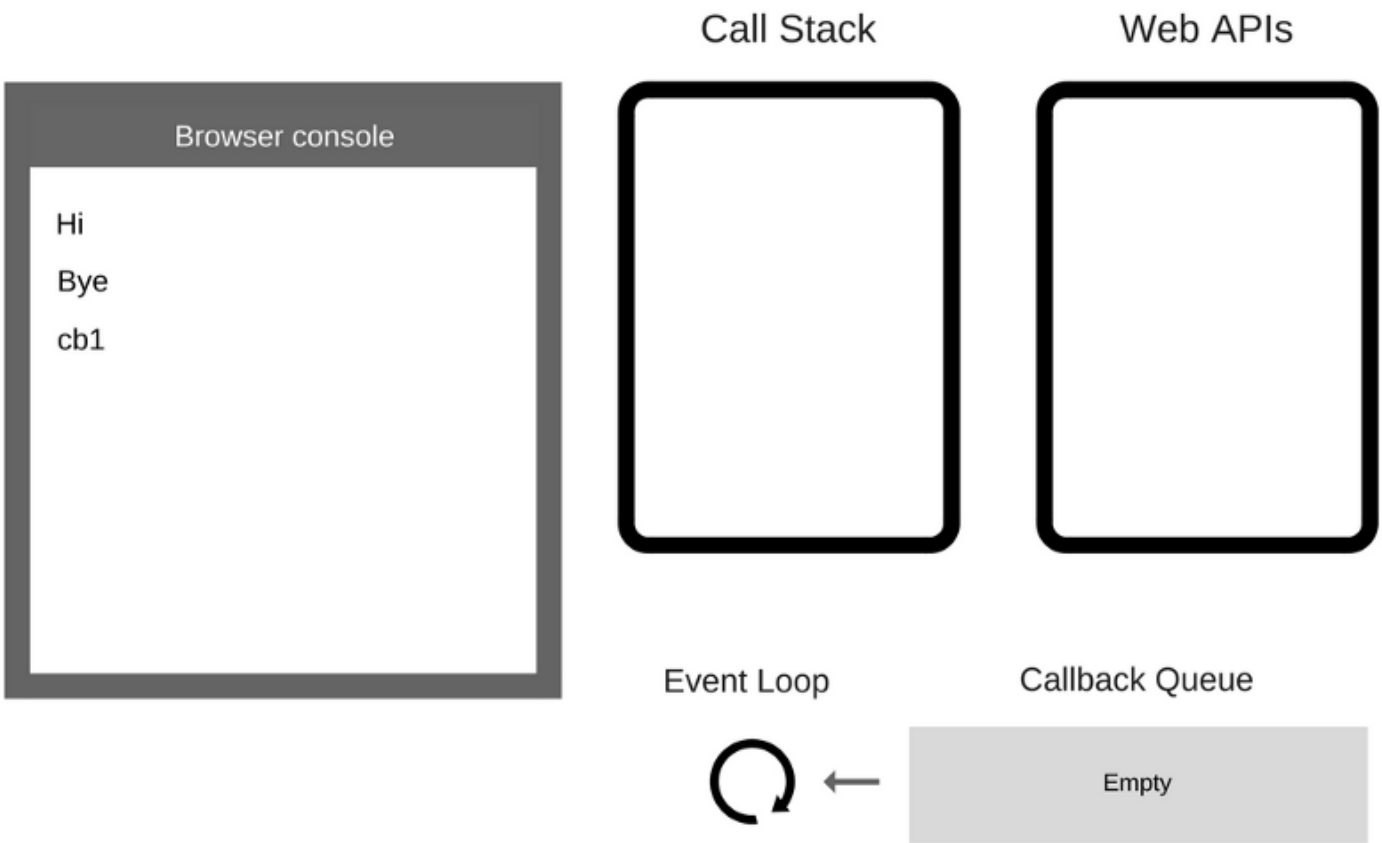
14. Команда `console.log('cb1')` выполняется.



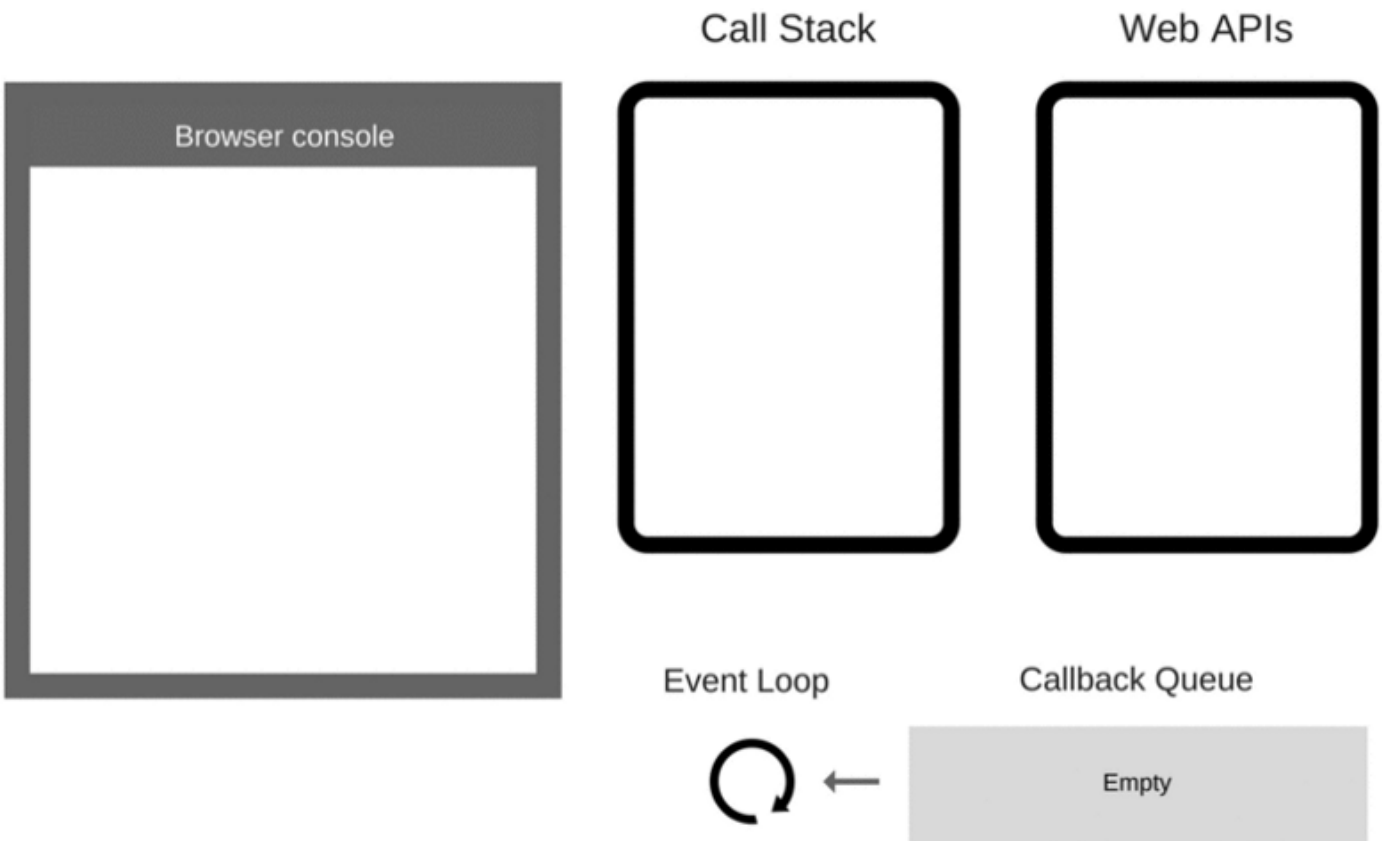
15. Команда `console.log('cb1')` удаляется из стека вызовов.



16. Функция `cb1` удаляется из стека вызовов.



Вот, для закрепления, то же самое в анимированном виде.



Интересно заметить, что спецификация ES6 определяет то, как должен работать цикл событий, а именно, указывает на то, что технически он находится в пределах ответственности JS-движка, который начинает играть более важную роль в экосистеме JS. Основная причина подобного заключается в том, что в ES6 появились промисы и им требуется надёжный механизм планирования операций в очереди цикла событий.

Как работает setTimeout(…)

Вызов `setTimeout(…)` не приводит к автоматическому размещению коллбэка в очереди цикла событий. Эта команда запускает таймер. Когда таймер срабатывает, окружение помещает коллбэк в цикл событий, в результате, в ходе какого-то из будущих

тиков, этот коллбэк будет взят в работу и выполнен. Взгляните на этот фрагмент кода:

```
setTimeout(myCallback, 1000);
```

Выполнение этой команды не означает, что `myCallback` будет выполнен через 1000 мс., правильнее будет сказать, что через 1000 мс. `myCallback` будет добавлен в очередь. В очереди, однако, могут быть другие события, добавленные туда ранее, в результате нашему коллбэку придётся подождать.

Существует довольно много статей, предназначенных для тех, кто только начинает заниматься асинхронным программированием на JavaScript. В них можно найти рекомендацию по использованию команды `setTimeout(callback, 0)`. Теперь вы знаете, как работает цикл событий и что происходит при вызове `setTimeout`. Учитывая это, вполне очевидно то, что вызов `setTimeout` со вторым аргументом, равным 0, просто откладывает вызов коллбэка до момента очищения стека вызовов.

Взгляните на следующий пример:

```
console.log('Hi');
setTimeout(function() {
    console.log('callback');
}, 0);
console.log('Bye');
```

Хотя время, на которое установлен таймер, составляет 0 мс., в

консоль будет выведено следующее:

```
Hi
Bye
callback
```

Задания ES6

В ES6 появилась новая концепция, которая называется очередью заданий (Job Queue). Эту конструкцию можно считать слоем, расположенным поверх очереди цикла событий. Вполне возможно, вы с ней сталкивались, когда вам приходилось разбираться с особенностями асинхронного поведения промисов.

Сейчас мы опишем это в двух словах, в результате, когда будем говорить об асинхронной разработке с использованием промисов, вы будете понимать, как асинхронные действия планируются и обрабатываются.

Представьте себе это так: очередь заданий — это очередь, которая присоединена к концу каждого тика в очереди цикла событий. Некие асинхронные действия, которые могут произойти в течение тика цикла событий не приведут к тому, что новое событие будет добавлено в очередь цикла событий, но вместо этого элемент (то есть — задание) будет добавлен в конец очереди заданий текущего тика. Это означает, что добавляя в очередь команды, которые должны быть выполнены в будущем, вы можете быть уверены в том, в каком порядке они будут выполняться.

Выполнение задания может приводить к добавлению

дополнительных заданий в конец той же очереди. В теории возможно, чтобы «циклическое» задание (задание которое занимается добавлением других заданий) работало бесконечно, истощая ресурсы программы, необходимые для перехода к следующему тику цикла событий. Концептуально это было бы похожим на создание бесконечного цикла вроде `while(true)`.

Задания — это что-то вроде «хака» s`etTimeout(callback, 0)`, но реализованные так, что они дают возможность соблюдения последовательности операций, которые выполняются позже, но так скоро, насколько это возможно.

Коллбэки

Как вы уже знаете, коллбэки являются самым распространённым средством выражения и выполнения асинхронных действий в программах на JavaScript. Более того, коллбэк является наиболее фундаментальным асинхронным шаблоном языка. Бесчисленное множество JS-приложений, даже весьма хитроумных и сложных, основано исключительно на коллбэках.

Всё это хорошо, но и коллбэки не идеальны. Поэтому многие разработчики пытаются найти более удачные шаблоны асинхронной разработки. Невозможно, однако, эффективно использовать любую абстракцию, не понимая того, как всё работает, что называется, под капотом.

Ниже мы подробно рассмотрим пару таких абстракций для того, чтобы показать, почему более продвинутые асинхронные

шаблоны, о которых мы ещё поговорим, необходимы и даже рекомендованы к использованию.

Вложенные коллбэки

Взгляните на следующий код:

```
listen('click', function (e){
    setTimeout(function(){
        ajax('https://api.example.com/endpoint', function (text){
            if (text == "hello") {

        doSomething();

    }

    else if (text == "world") {

        doSomethingElse();
            }
        });
    }, 500);
});
```

Здесь имеется цепочка из трёх вложенных в друг друга функций, каждая из них представляет собой шаг в последовательности действий, выполняемых асинхронно.

Такой код часто называют адом коллбэков. Но «ад» кроется не в том, что функции вложены, и не в том, что блоки кода приходится выравнивать относительно друг друга. Это — гораздо более глубокая проблема.

Разберём этот код. Сначала мы ожидаем события `click`, затем ждём срабатывания таймера, и, наконец, ожидаем прихода Ajax-

ответа, после прихода которого всё это может повториться снова.

На первый взгляд может показаться, что этот код выражает свою асинхронную природу вполне естественно, в виде последовательных шагов. Вот первый шаг:

```
listen('click', function (e) {

// ..
});
```

Вот второй:

```
setTimeout(function(){
    // ..
}, 500);
```

Вот третий:

```
ajax('https://api.example.com/endpoint', function (text){
    // ..
});
```

И наконец, происходит вот это:

```
if (text == "hello") {
    doSomething();
}
else if (text == "world") {
    doSomethingElse();
}
```

Итак, подобный подход к написанию асинхронного кода выглядит гораздо более естественным, правда? Должен быть способ писать его именно так.

Промисы

Взгляните на следующий фрагмент кода:

```
var x = 1;
var y = 2;
console.log(x + y);
```

Тут всё очень просто: значения переменных `x` и `y` складываются и выводятся в консоль. Но что если значение `x` или `y` недоступно и его ещё предстоит задать? Скажем, нам нужно получить с сервера то, что будет записано в `x` и `y`, а потом уже использовать эти данные в выражении. Представим, что у нас имеются функции `loadX` и `loadY`, которые, соответственно, загружают значения `x` и `y` с сервера. Затем представьте, что имеется функция `sum`, которая складывает значения `x` и `y` как только они будут загружены.

Это всё может выглядеть так (страшновато получилось, правда?):

```
function sum(getX, getY, callback) {
    var x, y;
    getX(function(result) {
        x = result;
        if (y !== undefined) {
            callback(x + y);
        }
    });
```

```
    getY(function(result) {
        y = result;
        if (x !== undefined) {
            callback(x + y);
        }
    });
}
// Синхронная или асинхронная функция, которая загружает значение`x`
function fetchX() {
    // ..
}


// Синхронная или асинхронная функция, которая загружает значение
`y`
function fetchY() {
    // ..
}
sum(fetchX, fetchY, function(result) {
    console.log(result);
});
```

Тут есть одна очень важная вещь. А именно, в этом коде мы относимся к `x` и `y` как к значениям, которые будут получены в будущем, и мы описываем операцию `sum(...)` (при её вызове, не вдаваясь в детали реализации) так, будто для её выполнения не имеет значения, доступны или нет значения `x` и `y` на момент её вызова.

Конечно, представленный здесь грубый подход, основанный на коллбэках, оставляет желать лучшего. Это — лишь первый маленький шаг к пониманию преимуществ, которые даёт возможность оперировать «будущими значениями», не беспокоясь о конкретном времени их появления.

Значение промиса

Сначала взглянем на то, как можно выразить операцию `x + y` с использованием промисов:

```
function sum(xPromise, yPromise) {

// `Promise.all([ .. ])` принимает массив промисов,

// и возвращает новый промис, который ожидает

// разрешения всех промисов, которые были в массиве

return Promise.all([xPromise, yPromise])


// когда этот промис будет разрешён, возьмём

// полученные значения `X` и `Y` и суммируем их.

.then(function(values){


// `values` - это массив сообщений из ранее


// разрешённых промисов


return values[0] + values[1];

} );
}

// `fetchX()` и `fetchY()` возвращают промисы для
// соответствующих значений. Эти значения могут быть
// доступны *сейчас* или *позже*.
sum(fetchX(), fetchY())

// получаем промис для сложения этих
// двух чисел.
// теперь мы используем вызов `then(...)` для организации ожидания
// разрешения этого возвращённого промиса.
.then(function(sum){
    console.log(sum);
});
```

В этом примере есть два слоя промисов.

Вызовы `fetchX()` и `fetchY()` выполняются напрямую, и значения, которые они возвращают (промисы!) передаются в `sum(...)`. Те значения, которые представляют эти промисы, могут быть готовы к дальнейшему использованию сейчас или позже, но каждый промис ведёт себя так, что сам по себе момент доступности значений неважен. В результате о значениях `x` и `y` мы рассуждаем без привязки ко времени. Это — будущие значения.

Второй слой промисов — это промис, который создаёт и возвращает (с помощью `Promise.all([ ... ])`) вызов `sum(…)`. Мы ожидаем значение, которое возвратит этот промис, вызывая `then(…)`. Когда операция `sum(…)` завершается, наше будущее значение суммы готово и мы можем вывести его на экран. Мы скрываем логику ожидания будущих значений `x` и `y` внутри `sum(…)`.

Обратите внимание на то, что внутри `sum(…)` вызов `Promise.all([ … ])` создаёт промис (который ожидает разрешения `promiseX` и `promiseY`). Вызов `.then(…)` создаёт ещё один промис, команда которого `values[0] + values[1]` немедленно выполняется (промис разрешается, давая результат сложения). Таким образом, вызов `then(…)`, который мы, в конце примера, присоединили к вызову `sum(…)`, на самом деле работает со вторым возвращённым промисом, а не с тем, который создан командой `Promise.all([ ... ])`. Кроме того, хотя мы не

присоединяем других вызовов ко второму `then(…)`, он тоже создаёт ещё один промис, с которым, если надо, вполне можно продолжить работу. Как результат, мы получаем возможность объединения вызовов `.then(…)` в цепочки.

При использовании промисов вызов `then(…)` может, на самом деле, принимать две функции. Первая функция — для случая, когда промис успешно разрешается. Вторая — для ситуации, когда промис оказывается отклонённым:

```
sum(fetchX(), fetchY())
.then(
    // обработчик успешного разрешения промиса
    function(sum) {
        console.log( sum );
    },
    // обработчик отклонения промиса
    function(err) {

console.error( err ); // что-то пошло не так
}
);
```

Если при загрузке значений `x` или `y` что-то пошло не так, или произошёл сбой в ходе сложения этих значений, промис, возвращённый функцией `sum(…)`, будет отклонён. После этого будет вызван второй коллбэк, переданный `then(…)`, представляющий собой обработчик ошибки. Он получит сообщение об ошибке от промиса.

Так как промисы инкапсулируют состояния, зависящие от времени, то есть — ожидание успешного или неуспешного получения

некоего значения, сами по себе, для окружающего кода, промисы являются конструкциями, от времени не зависящими. Таким образом, их можно комбинировать, добиваясь предсказуемых результатов, не зависящих от временных характеристик выполняемых внутри них операций.

Более того, как только промис разрешается, своё состояние он уже никогда не меняет. В этот момент он становится неизменяемым (иммутабельным) объектом, при этом к нему можно обращаться столько раз, сколько необходимо.

Очень удобно и то, что промисы можно объединять в цепочки:

```
function delay(time) {
    return new Promise(function(resolve, reject){
        setTimeout(resolve, time);
    });
}

delay(1000)
.then(function(){
    console.log("after 1000ms");
    return delay(2000);
})
.then(function(){
    console.log("after another 2000ms");
})
.then(function(){
    console.log("step 4 (next Job)");
    return delay(5000);
})
// ...
```

Вызов `delay(2000)` создаёт промис, который будет разрешён через 2000 мс., затем мы возвращаем этот промис из коллбэка успешного завершения операции в первом `then(…)`, что приводит

к тому, что промис второго `then(…)` будет ожидать эти 2000 мс.

Обратите внимание на то, что так как промисы, после разрешения, не подлежат модификации внешними средствами, теперь безопасно передавать эти значения любым программным механизмам, не опасаясь того, что они будут случайно или злонамеренно изменены. Это особенно актуально в случае, если имеется несколько конструкций, наблюдающих за промисом и ожидающих его разрешения. Любая из этих конструкций не может помешать другим ожидать разрешения промиса. Разговоры об иммутабельности, могут показаться слишком теоретизированными, но это, на самом деле, один из важных базовых аспектов устройства промисов, поэтому неправильно будет обойти его вниманием.

Промис или нет?

Важная деталь, касающаяся работы с промисами, заключается в точном знании того, является ли некое значение объектом `Promise` или нет. Другими словами, речь идёт о том, чтобы знать, будет ли некое значение вести себя как промис.

Известно, что промисы можно создавать с использованием конструкции `new Promise(…)`, в результате, можно подумать, что проверки вида `p instanceof Promise` будет достаточно. Однако, это не совсем так.

Дело в том, что получить промис можно из другого браузерного окна (то есть, из фрейма), в котором промисы создаются с

помощью его собственного конструктора `Promise`, отличного от того, который используется в текущем окне или фрейме. Как результат, вышеописанная проверка не сможет получить ответ на вопрос о том, является ли некое значение промисом.

Более того, в некоей библиотеке или каком-нибудь фреймворке могут использоваться их собственные промисы, а не те, что реализованы в ES6. На самом деле, можно даже использовать промисы, созданные с помощью библиотек, в старых версиях браузеров, которые и не подозревают о существовании стандартных промисов.

Проглатывание исключений

Если в любом месте при создании промиса или при ожидании его разрешения возникнет исключение, вроде `TypeError` или `ReferenceError`, это исключение будет перехвачено и приведёт к отклонению промиса.

Например:

```
var p = new Promise(function(resolve, reject){
    foo.bar();
  // `foo` не определено, поэтому возникает ошибка!
    resolve(374);  // сюда мы не попадём никогда :(
});

p.then(
    function fulfilled(){
        // сюда мы тоже не попадаем :(
    },
    function rejected(err){
        // `err` будет объектом исключения `TypeError`
        // из строки `foo.bar()`.
```

```
    }
);
```

Но что произойдёт, если промис разрешён, но на стадии получения его результата возникнет JS-исключение (в коллбэке, зарегистрированном `then(…)`)? Даже хотя такая ошибка и не будет потеряна, можно обнаружить, что обрабатывается эта ситуация, на первый взгляд, несколько неожиданно:

```
var p = new Promise( function(resolve,reject){

resolve(374);
});

p.then(function fulfilled(message){
    foo.bar();
    console.log(message);   // этого фрагмента достичь не удаётся
},
    function rejected(err){
        // этого тоже
    }
);
```

Кажется, будто исключение, вызванное строкой `foo.bar()` и в самом деле было «проглочено». Однако, это не так. На более глубоком уровне оказывается, что возникает нечто, чего мы не ожидаем. А именно, сам вызов `p.then(…)` возвращает другой промис, и именно этот промис будет отклонён с исключением `TypeError`.

Обработка неперехваченных исключений

Есть и другие подходы к работе с исключениями, которые многие

сочли бы более удачными.
Часто можно встретить предложения, заключающиеся в том, что при работе с промисами следует использовать блок `done(…)`, который сообщает о том, что обработка цепочки промисов «завершена». Блок `done(…)` не создаёт и не возвращает промисы, поэтому коллбэк, переданный `done(…)`, очевидно, не связан с сообщениями об ошибках, передаваемых присоединённым промисам, которых не существует.

Он обрабатывается, так, как и можно обычно ожидать от ситуации, в которой возникают неперехваченные ошибки: любое исключение внутри обработчика отклонения промиса в `done(…)` будет выброшено как глобальная необработанная ошибка (обычно сообщение об этом попадает в консоль разработчика):

```
var p = Promise.resolve(374);

p.then(function fulfilled(msg){
    // у числа нет строковых функций,
    // поэтому тут будет выброшена ошибка
    console.log(msg.toLowerCase());
})
.done(null, function() {
    // если здесь произойдёт исключение, оно попадёт в глобальную
область видимости
});
```

Новшество ES8: async / await

В JavaScript ES8 появилась конструкция `async / await`, которая упрощает работу с промисами. Сейчас мы кратко рассмотрим возможности, которые предлагает `async / await` и поговорим о

том, как использовать эти возможности для написания асинхронного кода.

Асинхронные функции объявляют, пользуясь ключевым словом `async`. Такая функция возвращает объект AsyncFunction. Этот объект представляет собой асинхронную функцию, которая выполняет код, находящийся внутри неё.

Когда асинхронная функция вызывается, она возвращает объект `Promise`. Если такая функция возвращает значение, которое не является объектом `Promise`, этот объект будет автоматически создан и разрешён с использованием значения, возвращённого функцией. Если функция, объявленная с ключевым словом `async`, выдаст исключение, промис будет отклонён с этим исключением.

Функция, объявленная с ключевым словом `async`, может содержать выражение с ключевым словом `await`, которое приостанавливает выполнение функции и ожидает разрешения промиса, фиругирующего в данном выражении. После этого выполнение async-функции продолжается, и, например, осуществляется возврат полученного после разрешения промиса значения.

Объекты Promise в JavaScript можно рассматривать как эквиваленты Future из Java или задач из C#.

Цель `async / await` заключается в том, чтобы упростить использование промисов.

Взглянем на следующий пример:

```
// Это - самая обыкновенная JS-функция
function getNumber1() {
    return Promise.resolve('374');
}
// Эта функция делает то же самое, что и getNumber1
async function getNumber2() {
    return 374;
}
```

Функции, которые выдают исключения, аналогичны функциям, которые возвращают отклонённые промисы:

```
function f1() {
    return Promise.reject('Some error');
}
async function f2() {
    throw 'Some error';
}
```

Ключевое слово `await` можно использовать только в функциях, объявленных с ключевым словом `async`. Оно позволяет организовать ожидание разрешения промиса. Если мы используем промисы за пределами async-функций, нам всё ещё нужно использовать коллбэки блока `then`:

```
async function loadData() {
    // `rp`- это функция request-promise.
    var promise1 = rp('https://api.example.com/endpoint1');
    var promise2 = rp('https://api.example.com/endpoint2');

    // В данный момент выполняются оба запроса
    // и нам нужно подождать их завершения.
    var response1 = await promise1;
    var response2 = await promise2;
```

```
    return response1 + ' ' + response2;
}
// Так как мы больше не в функции, объявленной с ключевым словом
`async`
// нам нужно использовать `then` с возвращённым объектом Promise
loadData().then(() => console.log('Done'));
```

Определять асинхронные функции можно и используя «асинхронное функциональное выражение», оно очень похоже на обычное определение с использованием инструкции `function` и имеет почти такой же синтаксис. Основное различие между функциональным выражением и обычным объявлением функции заключается в имени функции, которое, в функциональном выражении, может быть опущено, что приведёт к созданию анонимной функции. Асинхронное функциональное выражение может быть использовано как IIFE (Immediately Invoked Function Expression, немедленно вызываемое функциональное выражение), которое выполняется сразу после его определения.

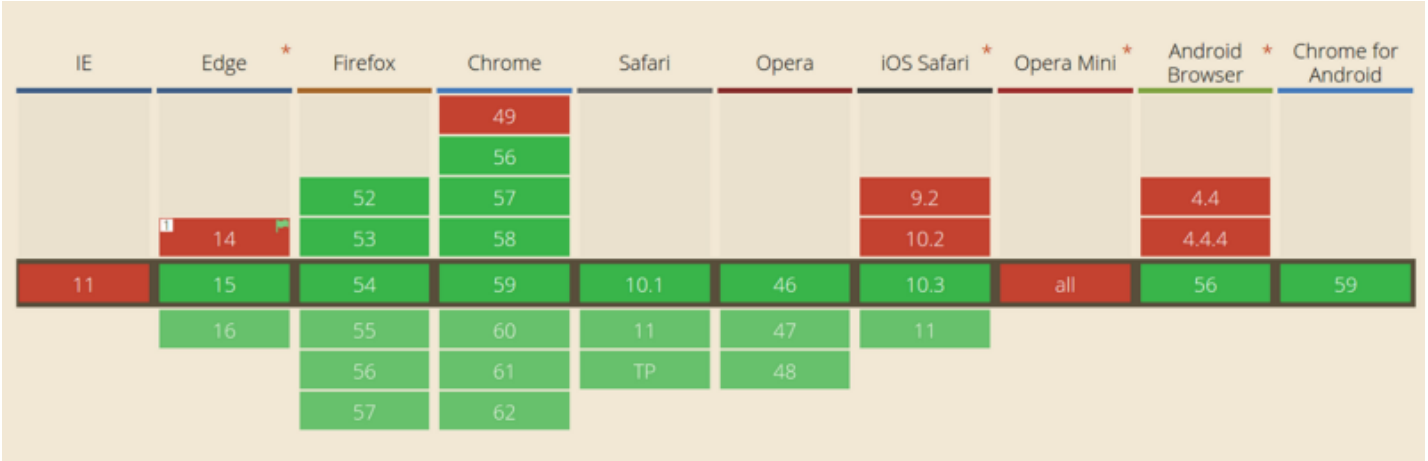
Выглядит это так:

```
var loadData = async function() {
    // `rp`- это функция request-promise.
    var promise1 = rp('https://api.example.com/endpoint1');
    var promise2 = rp('https://api.example.com/endpoint2');

    // В данный момент выполняются оба запроса
    // и нам нужно подождать их завершения.
    var response1 = await promise1;
    var response2 = await promise2;
    return response1 + ' ' + response2;
}
```

Стоит отметить, что конструкция `async / await` поддерживается

во всех основных браузерах.



Если нужный вам браузер не поддерживает эту технологию, есть и обходные пути, например — Babel и TypeScript.

Сейчас мы сравним промисы и конструкцию `async / await` и поговорим о том, как улучшить качество кода, разобрав пять примеров.

5 советов по написанию надёжного асинхронного кода, который легко поддерживать

Чистота кода

Использование конструкции `async / await` позволяет вам писать гораздо меньше кода. Каждый раз, используя `async / await`, вы избавляетесь от нескольких ненужных шагов. Среди них — использование блока `.then()`, создание анонимной функции для обработки ответа, задание в этой функции-коллбэке имени для переменной, содержащей результаты ответа, и так далее.

Вот фрагмент кода, в котором используются промисы:

```
/ `rp`- это функция request-promise.
rp('https://api.example.com/endpoint1').then(function(data) {
 // …
});
```

Вот то же самое, написанное с использованием `async / await`:

```
// `rp`- это функция request-promise.
var response = await rp('https://api.example.com/endpoint1');
```

Обработка ошибок

Конструкция `async / await` позволяет обрабатывать синхронные и асинхронные ошибки с использованием одних и тех же механизмов. А именно, речь идёт о широко известном выражении `try / catch`.

Взглянем на то, как обработка ошибок выполняется при использовании промисов. В частности, здесь нам приходится использовать блок `.catch()` для обработки асинхронных ошибок, и блок `try / catch` для обработки синхронных ошибок:

```
function loadData() {
    try { // Перехват синхронных ошибок.
        getJSON().then(function(response) {
            var parsed = JSON.parse(response);
            console.log(parsed);
        }).catch(function(e) { // Перехват асинхронных ошибок
            console.log(e);
        });
```

```
    } catch(e) {
        console.log(e);
    }
}
```

Вот как обрабатывать ошибки при использовании `async / await`:

```
async function loadData() {
    try {
        var data = JSON.parse(await getJSON());
        console.log(data);
    } catch(e) {
        console.log(e);
    }
}
```

Обработка условий

Применение `async / await` упрощает написание кода, использующего условия. Вот — код, основанный на промисах:

```
function loadData() {
  return getJSON()
    .then(function(response) {
      if (response.needsAnotherRequest) {
        return makeAnotherRequest(response)
          .then(function(anotherResponse) {
            console.log(anotherResponse)
            return anotherResponse
          })
      } else {
        console.log(response)
        return response
      }
    })
}
```

Вот — пример с `async / await`:

```
async function loadData() {
  var response = await getJSON();
  if (response.needsAnotherRequest) {
    var anotherResponse = await makeAnotherRequest(response);
    console.log(anotherResponse)
    return anotherResponse
  } else {
    console.log(response);
    return response;
  }
}
```

Трассировка стека

В отличие от `async / await`, стек ошибки, возвращённый из цепочки промисов, не содержит сведений о точном месте, в котором произошла ошибка. Вот как это выглядит при использовании промисов:

```
function loadData() {
  return callAPromise()
    .then(callback1)
    .then(callback2)
    .then(callback3)
    .then(() => {
      throw new Error("boom");
    })
}
loadData()
  .catch(function(e) {
    console.log(err);
// Error: boom at callAPromise.then.then.then.then (index.js:8:13)
});
```

Вот — то же самое, но с использованием `async / await`:

```
async function loadData() {
  await callAPromise1()
  await callAPromise2()
  await callAPromise3()
  await callAPromise4()
  await callAPromise5()
  throw new Error("boom");
}
loadData()
  .catch(function(e) {
    console.log(err);
    // вывод
    // Error: boom at loadData (index.js:7:9)
});
```

Отладка

Если вы пользовались промисами, то вы знаете, что отладка подобных конструкций — это кошмар. Например, если установить точку останова внутри блока `.then` и использовать команды отладки вроде «step-over», отладчик не перейдёт к следующему `.then`, так как он умеет «перешагивать» лишь через синхронный код. С использованием `async / await` можно переходить по вызовам, в которых используется ключевое слово `await` так, будто это — обычные синхронные операции.

Итоги

Написание асинхронного кода важно не только для обычных приложений, но и для библиотек. Например, библиотека

SessionStack записывает всё, происходящее в веб-приложении. А именно, речь идёт обо всех изменениях DOM, о взаимодействии с пользователями, об исключениях JavaScript, о трассировке стека, о сетевых запросах, давших сбой и об отладочных сообщениях.

Всё это должно происходить в продакшн-окружении и при этом не мешать работе пользовательского интерфейса. Для того, чтобы этого достичь, команде необходимо хорошо оптимизировать код, сделав его асинхронным настолько, насколько это возможно. Это позволяет не перегружать цикл событий. Кроме того, задача воспроизведения всего того, что происходило со страницей, также требует использования асинхронного кода.

В итоге можно сказать, что не стоит необдуманно стремиться к написанию асинхронного кода исключительно с использованием неких самых новых технологий. Важно понимать внутренние особенности реализации асинхронных операций в JavaScript, знать то, как устроены выбранные методы. У каждого подхода есть свои преимущества и недостатки, как, впрочем, и у всего остального в программировании.

Уважаемые читатели! Мы уже поднимали тему асинхронной JS-разработки. В частности, здесь можно почитать о паттернах и анти-паттернах при работе с промисами и поучаствовать в опросе. Здесь — узнать о переходе на `async / await`. Полагаем, тем, кого волнуют насущные вопросы асинхронности, интересно будет взглянуть и в комментарии к этим материалам. Сегодня же мы просим вас поделиться примерами улучшения (или ухудшения) некоей ситуации при переходе с промисов на `async / await`.

Проголосовать:

+28

Поделиться:

Сохранить:

Комментарии (28)

Похожие публикации

JavaScript ES8 и переход на async / await

107

ПЕРЕВОД

ru_vds • 10 октября 2017 в 14:58

Пишем симпатичные Node.js-API с использованием async/await и базы данных Firebase

6

ПЕРЕВОД

ru_vds • 10 июля 2017 в 14:19

Async/await: 6 причин забыть о промисах

182

ПЕРЕВОД

ru_vds • 10 апреля 2017 в 14:03

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю

0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни

33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018

70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ

40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству?

8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32