JSDG 🦏 | Asynchronous JavaScript | Асинхронный JavaScript | chapter 13

`JavaScript: The Definitive Guide 7th EDITION • Master the World's Most-Used Programming Language • David Flanagan • 2021`

___

Объекты Promise, появившиеся в ES6, представляют пока не доступный результат асинхронной операции.

Ключевые слова async и await были введены в ES2017 и предлагают новый синтаксис, который упрощает асинхронное программирование, позволяя структурировать основанный на Promise код, как если бы он был синхронным.

Наконец, асинхронные итераторы и цикл f o r / a w a i t появились в ES2018 и дают нам возможность работать с потоками асинхронных событий, используя простые циклы, которые выглядят синхронными.

___

Асинхронное программирование с использованием обратных вызовов

На самом фундаментальном уровне асинхронное программирование JavaScript производится с помощью *обратных вызовов*. Обратный вызов . ЭТо функция, которую вы пишете и затем передаете какой-то другой функции Затем другая функция вызывает вашу функцию (“делает обратный вызов*) когда удовлетворяется определенное условие или происходит некоторое (асинхронное) событие.

Обратные вызовы и события в Node

const fs = require ("fs"); // Модуль fs содержит API-интерфейсы, связанные с файловой системой.
let options = {} // Объект для хранения параметров для нашей программы. Здесь задаются параметры по умолчанию.
Прочитать конфигурационный файл, затем вызвать функцию обратного вызова

```
fs.readFile("config.json", "utf-8", (err, text) => {
      if (err) ‹
            //Если возникла ошибка, тогда отобразить
предупреждение, но продолжить
            console.warn ("Could not read config file:", err);
            // Не удалось прочитать конфигурационный файл
      } else {
            // В противном случае произвести разбор содержимого
файла
            // и присвоить объекту параметров.
            Object.assign options, JSON.parse (text) );
            // В любом случае теперь мы можем начать выполнение
программы.
            startProgram (options);
});
```

—

Promise

```
getJSON ("/api/user/profile") .then (displayUserProfile,
handleProfileError);

getJSON ("/api/user/profile").then
(displayUserProfile).catch(handleProfileError);
```

Терминология, связанная с **объектами Promise**

При обсуждении обьектов Promise в JavaScript эквивалентными терминами будут "удовлетворено" и "отклонено".

Представье, что вы вызвали метод then() объекта Promise и передали ему две функции обратного вызова.

Мы говорим, что объект Promise (обещание) был удовлетворен (fulfilled), если и когда вызван первый обратный вызов.

И мы говорим, что объект Promise (обещание) был отклонен (rejected), если и когда вызван второй обратный вызов.

Если объект Promise ни удовлетворен, ни отклонен, тогда он считается ожидающим решения (pending).

А после того, как объект Promise удовлетворен или отклонен, мы говорим, что он является урегулированным (settled).

Имейте в виду, что объект Promise не может быть одновременно удовлетворенным и отклоненным.

Как только объект Promise становится урегулированным, он никогда не изменится с удовлетворенного на отклоненный и наоборот.

Причина, по которой я хочу уточнить терминологию, касающуюся Promise, связана с тем, что объекты Promise также могут быть разрешенными (resolved).

Разрешенное состояние легко спутать с удовлетворенным или урегулированным состоянием, но они не в точности одинаковы. Понимание разрешенного состояния - ключевой аспект глубокого понимания объектов Promise, и я вернусь к нему во время обсуждения цепочек Promise далее.

___

В ES2018 объекты Promise также определяют метод .finally(), цель которого аналогична цели конструкции finally в операторе try/catch/finally. Если вы добавите к своей цепочке Promise вызов . finally (), тогда переданный методу .finally () обратный вызов будет вызван, когда Promise урегулируется.

Ваш обратный вызов будет вызван в случае удовлетворения или отклонения Promise, но поскольку аргументы не передаются, то вы не сможете узнать, был Promise удовлетворен или отклонен.

Но если вам в любом случае нужно выполнить код очистки (скажем, закрывающий открытые файлы или сетевые подключения), тогда обратный вызов . finally () - идеальный способ сделать это.

Подобно .then () и . catch (), метод . finally () возвращает новый

обьект Promise.

Возвращаемое значение обратного вызова .finally() обычно игнорируется, а объект Promise, возвращенный .finally(), как правило, будет разрешен или отклонен с тем же значением, что и объект Promise, с которым вызывался метод .finally() при разрешении или отклонении.

Однако если в обратном вызове .finally() генерируется исключение, то возвращенный методом . finally() объект Promise будет отклонен с таким значением.

___

```
startAsyncOperation ()
. then doStageTwo)
.catch(recoverFronStageTwoError)
.then (doStageThree)
.then (doStageFour)
.catch(logStageThreeAndFourErrors);
```

Давайте более конкретно: если в предыдущем примере кода либо startAsyncOperation (), либо doStageTwo () генерирует ошибку, тогда будет вызвана функция recoverFromStageTwoError (). В случае нормального возврата из функции recoverFromStageTwoError () ее возвращаемое значение будет передано
doStageThree () и асинхронная операция благополучно продолжится. С другой стороны, если функции recoverFromStageTwoError () не удалось выполнить восстановление, то она сама сгенерирует ошибку (или повторно сгенерирует переданную ей ошибку). В такой ситуации ни doStageThree () , ни doStageFour ()
не вызывается, а ошибка, сгенерированная recoverFromStageTwoError (), будет передана logStageThreeAndFourErrors ().

___

Параллельное выполнение нескольких асинхронных операций с помощью Promise

```
// Мы начинаем с массива URL.
const urls = [ /* ноль или большее количество URL * / ];
// И преобразуем его в массив объектов Promise.
promises = urls.map (url => fetch (url) .then (r => r.text () )) ;
//Теперь получаем объект Promise для запуска всех объектов
Promise параллельно
Promise.all (promises)
      .then (bodies => { /* делать что-нибудь с массивом строк * / ])
      .catch (e => console.error (e));
```

Объект **Promise,** возвращенный **Promise.all ()**, отклоняется при отклонении любого из входных объектов **Promise.** Это происходит сразу же после первого отклонения и может случиться так, что остальные входные объекты все еще будут ожидать решения.

Функция **Promise.allSettledO** в **ES2020** принимает массив входных объектов **Promise** и подобно **Promise.all** возвращает объект **Promise.** Но **Promise.allSettled()** никогда не отклоняет возвращенный объект **Promise** и не удовлетворяет его до тех пор, пока не будут урегулированы все входные объекты **Promise.** Объект **Promise** разрешает ся в массив объектов, по одному на каждый входной объект **Promise.**

Каждый возвращенный объект имеет свойство **s t a t u s** (состояние), установленное **в " fu lfille d ”** (удовлетворен) или **"rejected"** (отклонен). Если состоянием является **"fulfilled",** тогда объект будет также иметь свойство **value** (зна чение), которое дает значение удовлетворения. А если состоянием оказывается
"rejected", то объект будет также иметь свойство **reason** (причина),

```
Promise.allSettled([ Promise.resolve(1), Promise.reject(2),
3]).then(results =>
      results [0] // ( status: "fulfilled", value: 1 }
      results [1] // { status: "rejected", reason: 2 }
      results [2] // { status: "fulfilled", value: 3 )
});
```

Изредка у вас может возникать необходимость запустить сразу

несколько объектов Promise, но заботиться только о значении первого удовлетворенно го объекта. В таком случае вы можете использовать Promise. race () вместо promise. al1 (). Функция Promise. race () возвращает объект Promise, который удовлетворен или отклонен, когда был удовлетворен или отклонен первый из объектов Promise во входном массиве.

```
function wait (duration) {
    // Создать и возвратить новый объект Promise.
    return new Promise ( (resolve, reject) => { // Это контролирует
        // объект Promise.
        // Если значение аргумента недопустимо, тогда
отклонить объект Promise
        if (duration < 0) I
            reject (new Error ("Time travel not yet implemented"));
            // Путешествия во времени пока не осуществимы
        }
        // В противном случае асинхронно ожидать и затем
разрешить
        // объект Promise.
        // setTimeout будет вызывать resolve () без аргументов, а
это значит,
        // что объект Promise будет удовлетворен в значение
undefined.
        setTimeout (resolve, duration);
    }):
}
```

——

async и await

```
let [valuel, value2] = await Promise.all ( [getJSON (ur11), getJSON
(ur12) ]);

async function f(x) { /* тело * / }

function f(x) {
    return new Promise (function (resolve, reject) I
    try {
        resolve((function (x) 1 /* body * / 1) (x) );
```

```
    catch (e) {
        reject (e);
    }
});
```

___

Асинхронная итерация

```
const fs = require ("fs");

async function parseFile (filename) {
    let stream = fs.createReadStream (filename, { encoding:
"utf-8" });

    for await (let chunk of stream) (
        parseChunk (chunk); // функция parseChunk ( ) определена
    }
}
```

Грубо говоря, асинхронный итератор производит объект Promise и цикл for/await ожидает, когда этот объект Promise будет удовлетворен, присваива ет значение удовлетворения переменной цикла и выполняет тело цикла. Затем он начинает заново, получая еще один объект Promise от итератора и ожидая, пока новый объект Promise будет удовлетворен.

```
for (const promise of promises) I
    response = await promise;
    handle (response) ;
}

for await (const response of promises) {
    handle (response);
}
```

В этом случае for/await всего лишь встраивает вызов await в цикл и делает код немного компактнее, но два примера решают одну и ту же задачу. Важно
отметить, что оба примера кода работают, только если находятся внутри функций, объявленных как async; таким образом, цикл

for / await ничем не отличается от обыкновенного выражения await.

Тем не менее, важно понимать, что в приведенном примере мы используем for/await с обычным итератором. Все становится более интересным, когда
речь идет о полностью асинхронных итераторах.

___

Асинхронные итераторы

Давайте вспомним ряд терминов из главы 12. Итерируемый объект - это такой объект, который можно задействовать в цикле for/of. В нем определен метод с символьным именем Symbol.iterator, возвращающий объект итератора. Объект итератора имеет метод next (), который можно многократно вызывать для получения значений итерируемого объекта. Метод next () объекта итератора возвращает объекты результатов итераций. Объект результата итерации имеет свойство value и/или свойство done.

Асинхронные итераторы похожи на обыкновенные итераторы, но обладают двумя важными отличиями.

Во-первых, в асинхронно итерируемом обьекте реализован метод с символьным именем Symbol.asyncIterator, а не Symbol.iterator.

(Как упоминалось ранее, цикл for/await совместим с обычными итерируемыми объектами, но предпочитает асинхронно итерируемые объекты и опробует сначала метод Symbol.asyncIterator и только затем метод Symbol.iterator.)

Во-вторых, метод next ( ) асинхронного итератора возвращает не непосредственно объект результата итерации, а объект Promise, который разрешается в объект результата итерации

Отличие тонкое: в случае асинхрон ных итераторов выбор момента окончания итерации может делаться асинхронно.

___

Асинхронные генераторы

То же самое справедливо в отношении асинхронных итераторов, которые мы можем реализовать с помощью генераторных функций, объявляемых как async.

Асинхронный генератор обладает возможностями асинхронных функций и генераторов: вы можете применять await, как делали бы в обычной асинхронной функции, а также использовать yield, как поступали бы в обыкновенном генераторе.

Но выдаваемые посредством yield значения автоматически помещаются в оболочки объектов Promise.

Даже синтаксис для асинхронных генераторов является комбинацией:

async function и function * обьединяются в async function *.

пример, который демонстрирует, каким образом можно было бы применить асинхронный генератор и цикл for /await для повторяющегося выполнения кода через фиксированные интервалы, используя синтаксис цикла вместо функции обратного вызова setInterval ():

```
// Оболочка на основе Promise вокруг setTimeout (),
// с которой можно использовать await.
// Возвращает объект Promise, который удовлетворяется
// с указанны количеством миллисекунд.
function elapsedTime (ms) {
    return new Promise (resolve => setTimeout (resolve, ms) );
}
    // Асинхронная генераторная функция, которая
инкрементирует счетчик
    // и выдает его указанное (или бесконечное) количество раз
    // через указанные интервалы.

async function* clock (interval, max=Infinity) I
    for (let count = 1; count <= max; count++) {// Обыкновенный
цикл for
        await elapsedTime (interval); // Ожидать в течение
```

```
указанного времени.
            yield count; // Выдать счетчик.
        }
}

// Тестовая функция, которая использует асинхронный генератор
// с циклом for/await.

async function test () { / / Асинхронная, поэтому можно применять
for/await.
        for await (let tick of    clock (300, 100)) { // Цикл 100 раз каждые
300 миллисекунд.
        console.log (tick);
)
```

Реализация асинхронных итераторов

```
function clock (interval, max=Infinity) {
        // Версия setTimeout с Promise, с которой можно
использовать await.
        // Обратите внимание, что функция принимает абсолютное
время,
        // а не интервал.

        function until (time) {
            return new Promise (resolve=>setTimeout (resolve, time-
Date. now () ));
        }

        // Возвратить асинхронно итерируемый объект.
        return {
            startTime: Date.now (), // Запомнить, когда мы начали.
            count: 1, // Запомнить, на какой мы итерации.

            async next () { // Метод next () делает это итератором.
                if (this.count > max) { // Мы закончили?
                    return { done: true } ; / / Результат итерации готов.
                }

                // Выяснить, когда должна начаться следующая
```

```
итерация,
                let targetTime = this.startTime + this.count * interval;
                // ожидать в течение этого времени
                await until (targetTime);
                // и возвратить значение счетчика в объекте
результата итерации
                return { value: this.count++ };
            },
            // Этот метод означает, что данный объект итератора
тоже итерируемый.
            [Symbol.asyncIterator] () { return this; }
        }
}


/**
* Класс асинхронно итерируемой очереди. Добавляйте значения
* с помощью enqueue () и удаляйте их с помощью dequeue () .
* Метод dequeue () возвращает объект Promise, поэтому значения
* можно извлекать из очереди до того, как они будут в
непомещены.
* Класс реализует методы [Symbol.asyncIterator] u next (), так что
* его можно использовать с щиклом for/avait (который не
завершится
* до тех пор, пока не будет вызван метод close () .)
* /
class AsyncQueue {
        constructor () {
            // Здесь хранятся значения, которые были помещены в
очередь, но еще не извлечены.
            this.values = [];

            // Когда объекты Promise извлекаются до того, как
            // соответствующие им значения помещаются в очередь,
здесь
            // сохраняются методы разрешения для таких объектов
Promise.
            this.resolvers = [];

            // После закрытия дополнительные значения помещать в
очередь нельзя,
            //и неудовлетворенные объекты Promise больше не
```

```
возвращаются.
        this.closed = false;
    }

    enqueue (value) {
        if (this.closed)
            throw new Error ("AsyncQueue closed");
            // Очередь AsyncQueue закрыта
        }

        if (this.resolvers.length > 0) I
            // Если это значение уже снабжалось объектом
Promise,
            // тогда разрешить данный объект Promise.
            const resolve = this.resolvers.shift ();
            resolve (value);
        } else {
            // В противном случае поместить значение в
очередь.
            this.values.push (value);
        }
    }

    dequeue () {
        if (this.values. length > 0) {
            // Если есть значение, помещенное в очередь, тогда
            // возвратить для него разрешенный объект Promise.
            const value = this.values.shift ();
            return Promise.resolve (value);
        } else if (this.closed) {
            // Если значения в очереди отсутствуют и очередь
закрыта, тогда
            // возвратить разрешенный объект Promise для
маркера конца потока
            return Promise.resolve(AsyncQueue.EOS);
        } else {
            // В противном случае возвратить неразрешенный
объект Promise,
            // поместив в очередь функцию разрешения для
будущего использования
            return new Promise ((resolve) => this.resolvers.push
(resolve); });
```

```
        }
    }

    close {
        // После закрытия дополнительные значения помещаться
в очередь не будут.
        // Таким образом, разрешить любые ожидающие
решения объекты Promise
        // в маркер конца потока.
        while (this.resolvers.length > 0) {
            this.resolvers.shift () (AsyncQueue.EOS) ;
        }
        this.closed = true;
    }

    // Определение метода, который делает этот класс
    // асинхронно итерируемым.
    [Symbol.asyncIterator] () { return this; }

    // Определение метода, который делает это асинхронным
итератором.
    // Объект Promise, возвращенный методом dequeue () ,
разрешается
    // в значение или сигнал EOS, если очередь закрыта.
    // Здесь нам необходимо возвратить объект Promise,
    // который разрешается в объект результата итерации.
    next () {
        return this.dequeue() .then (value => (value ===
AsyncQueue.EOS)
            ? { value: undefined, done: true }
            : { value: value, done: false });
    }
}

// Сигнальное значение, возвращаемое методом dequeue ()
// для пометки конца потока, когда очередь закрыта.
AsyncQueue.EOS = Symbol ("end-of-stream");



// Помещает события указанного типа, относящиеся к указанному
//  элементу документа, в объект AsyncQueue и возвращает
```

```
очередь
// И для использования в качестве потока событий.
function eventStream (elt, type) {
      const q = new AsyncQueue (); // Создать очередь.
      elt.addEventListener (type, e=>q. enqueue (e) ) ; // Поместить
события в очередь.
      return a;
}

async function handleKeys () {
      // Получить поток событий нажатия клавиш и пройти по ним в
цикле.
      for await (const event of eventSt ream (document, "keypress")) {
            console. log (event. key);
      }
}
```

Объекты, которые являются асинхронно итерируемыми, могут использоваться с циклом for/await. Асинхронно итерируемые объекты создаются за счет реализации метода [Symbol.asyncIterator] () или путем вызова генераторной функции async function *. Асинхронные итераторы предлагают альтернативу событиям "данных" в потоках среды Node и могут применяться для представления потока входных событий от пользователя в коде JavaScript стороны клиента.