Проміси

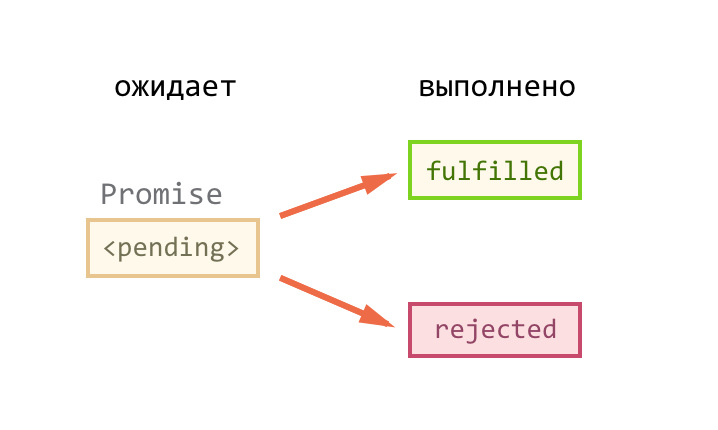
<https://javascript.info/async-await>

Promise (обычно их так и называют «промисы») – предоставляют удобный способ организации асинхронного кода.

В современном JavaScript промисы часто используются в том числе и неявно, при помощи генераторов, но об этом чуть позже.

## [Что такое Promise?](https://learn.javascript.ru/promise" \l "chto-takoe-promise)

Promise – это специальный объект, который содержит своё состояние. Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).



На promise можно навешивать коллбэки двух типов:

* onFulfilled – срабатывают, когда promise в состоянии «выполнен успешно».
* onRejected – срабатывают, когда promise в состоянии «выполнен с ошибкой».

Способ использования, в общих чертах, такой:

1. Код, которому надо сделать что-то асинхронно, создаёт объект promise и возвращает его.
2. Внешний код, получив promise, навешивает на него обработчики.
3. По завершении процесса асинхронный код переводит promise в состояние fulfilled (с результатом) или rejected (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде.

Синтаксис создания Promise:

var promise = new Promise(function(resolve, reject) {

// Эта функция будет вызвана автоматически

// В ней можно делать любые асинхронные операции,

// А когда они завершатся — нужно вызвать одно из:

// resolve(результат) при успешном выполнении

// reject(ошибка) при ошибке

})

Универсальный метод для навешивания обработчиков:

promise.then(onFulfilled, onRejected)

* onFulfilled – функция, которая будет вызвана с результатом при resolve.
* onRejected – функция, которая будет вызвана с ошибкой при reject.

С его помощью можно назначить как оба обработчика сразу, так и только один:

// onFulfilled сработает при успешном выполнении

promise.then(onFulfilled)

// onRejected сработает при ошибке

promise.then(null, onRejected)

**.catch**

Для того, чтобы поставить обработчик только на ошибку, вместо .then(null, onRejected)можно написать .catch(onRejected) – это то же самое.

**Синхронный throw – то же самое, что reject**

Если в функции промиса происходит синхронный throw (или иная ошибка), то вызывается reject:

'use strict';

let p = new Promise((resolve, reject) => {

// то же что reject(new Error("o\_O"))

throw new Error("o\_O");

})

p.catch(alert); // Error: o\_O

Посмотрим, как это выглядит вместе, на простом примере.

httpGet('/article/promise/userNoGithub.json')

.then(JSON.parse)

.then(user => httpGet(`https://api.github.com/users/${user.name}`))

.then(

JSON.parse,

function githubError(error) {

if (error.code == 404) {

return {name: "NoGithub", avatar\_url: '/article/promise/anon.png'};

} else {

throw error;

}

}

)

.then(function showAvatar(githubUser) {

let img = new Image();

img.src = githubUser.avatar\_url;

img.className = "promise-avatar-example";

document.body.appendChild(img);

setTimeout(() => img.remove(), 3000);

})

.catch(function genericError(error) {

alert(error); // Error: Not Found

});

## [Параллельное выполнение](https://learn.javascript.ru/promise" \l "parallelnoe-vypolnenie)

Что, если мы хотим осуществить несколько асинхронных процессов одновременно и обработать их результат?

В классе Promise есть следующие статические методы.

### [Promise.all(iterable)](https://learn.javascript.ru/promise" \l "promise-all-iterable)

Вызов Promise.all(iterable) получает массив (или другой итерируемый объект) промисов и возвращает промис, который ждёт, пока все переданные промисы завершатся, и переходит в состояние «выполнено» с массивом их результатов.

Например:

Promise.all([

httpGet('/article/promise/user.json'),

httpGet('/article/promise/guest.json')

]).then(results => {

alert(results);

});

Допустим, у нас есть массив с URL.

let urls = [

'/article/promise/user.json',

'/article/promise/guest.json'

];

Чтобы загрузить их параллельно, нужно:

1. Создать для каждого URL соответствующий промис.
2. Обернуть массив таких промисов в Promise.all.

Получится так:

'use strict';

let urls = [

'/article/promise/user.json',

'/article/promise/guest.json'

];

Promise.all( urls.map(httpGet) )

.then(results => {

alert(results);

});

Заметим, что если какой-то из промисов завершился с ошибкой, то результатом Promise.all будет эта ошибка. При этом остальные промисы игнорируются.

Например:

Promise.all([

httpGet('/article/promise/user.json'),

httpGet('/article/promise/guest.json'),

httpGet('/article/promise/no-such-page.json') // (нет такой страницы)

]).then(

result => alert("не сработает"),

error => alert("Ошибка: " + error.message) // Ошибка: Not Found

)

### [Promise.race(iterable)](https://learn.javascript.ru/promise" \l "promise-race-iterable)

Вызов Promise.race, как и Promise.all, получает итерируемый объект с промисами, которые нужно выполнить, и возвращает новый промис.

Но, в отличие от Promise.all, результатом будет только первый успешно выполнившийся промис из списка. Остальные игнорируются.

Например:

Promise.race([

httpGet('/article/promise/user.json'),

httpGet('/article/promise/guest.json')

]).then(firstResult => {

firstResult = JSON.parse(firstResult);

alert( firstResult.name ); // iliakan или guest, смотря что загрузится раньше

});

## [Promise.resolve(value)](https://learn.javascript.ru/promise" \l "promise-resolve-value)

Вызов Promise.resolve(value) создаёт успешно выполнившийся промис с результатом value.

Он аналогичен конструкции:

new Promise((resolve) => resolve(value))

Promise.resolve используют, когда хотят построить асинхронную цепочку, и начальный результат уже есть.

Например:

Promise.resolve(window.location) // начать с этого значения

.then(httpGet) // вызвать для него httpGet

.then(alert) // и вывести результат

## [Promise.reject(error)](https://learn.javascript.ru/promise" \l "promise-reject-error)

Аналогично Promise.reject(error) создаёт уже выполнившийся промис, но не с успешным результатом, а с ошибкой error.

Например:

Promise.reject(new Error("..."))

.catch(alert) // Error: ...

Метод Promise.reject используется очень редко, гораздо реже чем resolve, потому что ошибка возникает обычно не в начале цепочки, а в процессе её выполнения.

=================================================================

**async/await**

Объявление **async function** определяет асинхронную функцию, которая возвращает объект [AsyncFunction](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/AsyncFunction" \o "Конструктор AsyncFunction создает новый объект async function. В JavaScript любая асинхронная функция фактически является объектом AsyncFunction.).

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/async_function>

<https://habr.com/ru/company/ruvds/blog/326074/>

<https://javascript.info/async-await>

|  |  |
| --- | --- |
| async function *name*([*param*[, *param*[, ... *param*]]]) {  *statements*  }  **name**  Имя функции.  **param**  Имя аргумента, который будет передан в функцию.  **statements**  Выражение, содержащее тело функции.  После вызова функция async возвращает [Promise](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise" \o "Объект Promise (обещание) используется для отложенных и асинхронных вычислений. Promise может находиться в трёх состояниях:). Когда результат был получен, Promise завершается, возвращая полученное значение.  Когда функция async выбрасывает исключение, Promise ответит отказом с выброшенным (throws) значением. | function resolveAfter2Seconds(x) {  return new Promise(resolve => {  setTimeout(() => {  resolve(x);  }, 2000);  });  }  async function add1(x) {  const a = await resolveAfter2Seconds(20);  const b = await resolveAfter2Seconds(30);  return x + a + b;  }  add1(10).then(v => {  console.log(v); // prints 60 after 4 seconds.  });  async function add2(x) {  const a = resolveAfter2Seconds(20);  const b = resolveAfter2Seconds(30);  return x + await a + await b;  }  add2(10).then(v => {  console.log(v); // prints 60 after 2 seconds.  }); |
| Функция async может содержать выражение [await](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/await), которое приостанавливает выполнение функции async и ожидает ответа от переданного Promise, затем возобновляя выполнение функции async и возвращая полученное значение.  Ключевое слово await допустимо только в асинхронных функциях. В другом контексте вы получите ошибку SyntaxError. |

Приклад

|  |  |
| --- | --- |
| **З промісами** | **з async/await** |
| Лаконічний код | |
| const makeRequest = () =>  getJSON()     .then(data => {    console.log(data)       return "done"     })  makeRequest() | const makeRequest = async () => {   console.log(await getJSON())  return "done"  }  makeRequest() |
| Проста обробка помилок | |
| const makeRequest = () => {   try {     getJSON()       .then(result => {  *// парсинг JSON может вызвать ошибку*         const data = JSON.parse(result)         console.log(data)  })  .catch((err) => console.log(err)   } catch (err) {     console.log(err)   } | const makeRequest = async () => {   try {  *// парсинг JSON может вызвать ошибку*     const data = JSON.parse(await getJSON())     console.log(data)   } catch (err) {     console.log(err)   } } |
| Проста перевірка асинхронних дій | |
|  |  |
| Проста передача результатів промісів у наступний проміс | |
|  |  |