

Раздел

[Промисы, async/await](#)


Навигация по уроку

[Promise.all](#)[Promise.allSettled](#)[Promise.race](#)[Promise.resolve/reject](#)

Итого

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠 → Язык программирования JavaScript](#)
[→ Промисы, async/await](#) 18-го июня 2020

Promise API

В классе `Promise` есть 5 статических методов. Давайте познакомимся с ними.

Promise.all

Допустим, нам нужно запустить множество промисов параллельно и дождаться, пока все они выполнятся.

Например, параллельно загрузить несколько файлов и обработать результат, когда он готов.

Для этого как раз и пригодится `Promise.all`.

Синтаксис:

```
1 let promise = Promise.all([...промисы...]);
```

Метод `Promise.all` принимает массив промисов (может принимать любой перебираемый объект, но обычно используется массив) и возвращает новый промис.

Новый промис завершится, когда завершится весь переданный список промисов, и его результатом будет массив их результатов.

Например, `Promise.all`, представленный ниже, выполнится спустя 3 секунды, его результатом будет массив `[1, 2, 3]`:

```
1 Promise.all([
2   new Promise(resolve => setTimeout(() => resolve(1), 3000)),
3   new Promise(resolve => setTimeout(() => resolve(2), 2000)),
4   new Promise(resolve => setTimeout(() => resolve(3), 1000))
5 ]).then(alert); // когда все промисы выполнятся, результат
6 // каждый промис даёт элемент массива
```

Обратите внимание, что порядок элементов массива в точности соответствует порядку исходных промисов. Даже если первый промис будет выполняться дольше всех, его результат всё равно будет первым в массиве.

Часто применяемый трюк – пропустить массив данных через `map`-функцию, которая для каждого элемента создаст задачу-промис, и затем обернёт получившийся массив в `Promise.all`.

Например, если у нас есть массив ссылок, то мы можем загрузить их вот так:

```
1 let urls = [
2   'https://api.github.com/users/iliakan',
3   'https://api.github.com/users/remy',
4   'https://api.github.com/users/jeresig'
5 ];
6
7 // Преобразуем каждый URL в промис, возвращённый fetch
8 let requests = urls.map(url => fetch(url));
9
10 // Promise.all будет ожидать выполнения всех промисов
11 Promise.all(requests)
12   .then(responses => responses.forEach(
13     response => alert(`${response.url}: ${response.stat
14   }));
```

А вот пример побольше, с получением информации о пользователях GitHub по их логинам из массива (мы могли бы получать массив товаров по их идентификаторам, логика та же):

```
1 let names = ['iliakan', 'remy', 'jeresig'];
2
3 let requests = names.map(name => fetch(`https://api.git
4
5 Promise.all(requests)
6   .then(responses => {
7     // все промисы успешно завершены
8     for(let response of responses) {
9       alert(`${response.url}: ${response.status}`); //
10    }
11
12    return responses;
13  })
14  // преобразовать массив ответов response в response.j
15  // чтобы прочитать содержимое каждого
16  .then(responses => Promise.all(responses.map(r => r.j
17  // все JSON-ответы обработаны, users - массив с резул
18  .then(users => users.forEach(user => alert(user.name)
```

Если любой из промисов завершится с ошибкой, то промис, возвращённый `Promise.all`, немедленно завершается с этой ошибкой.

Например:

```
1 Promise.all([
2   new Promise((resolve, reject) => setTimeout(() => res
3   new Promise((resolve, reject) => setTimeout(() => rej
4   new Promise((resolve, reject) => setTimeout(() => res
5 ]).catch(alert); // Error: Ошибка!
```

Здесь второй промис завершится с ошибкой через 2 секунды. Это приведёт к немедленной ошибке в `Promise.all`, так что выполнится `.catch`: ошибка этого промиса становится ошибкой всего `Promise.all`.

⚠ В случае ошибки, остальные результаты игнорируются

Если один промис завершается с ошибкой, то весь `Promise.all` завершается с ней, полностью забывая про остальные промисы в списке. Их результаты игнорируются.

Например, если сделано несколько вызовов `fetch`, как в примере выше, и один не прошёл, то остальные будут всё ещё выполняться, но `Promise.all` за ними уже не смотрит. Скорее всего, они так или иначе завершатся, но их результаты будут проигнорированы.

`Promise.all` ничего не делает для их отмены, так как в промисах вообще нет концепции «отмены». В главе [Fetch: прерывание запроса](#) мы рассмотрим `AbortController`, который помогает с этим, но он не является частью Promise API.

Раздел

[Промисы, async/await](#)

Навигация по уроку

[Promise.all](#)

[Promise.allSettled](#)

[Promise.race](#)

[Promise.resolve/reject](#)

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)

Раздел

[Промисы, async/await](#)

Навигация по уроку

Promise.all

Promise.allSettled

Promise.race

Promise.resolve/reject

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



Promise.all(iterable) разрешает передавать не-промисы в iterable (перебираемом объекте)

Обычно, Promise.all(...) принимает перебираемый объект промисов (чаще всего массив). Но если любой из этих объектов не является промисом, он передаётся в итоговый массив «как есть».

Например, здесь результат: [1, 2, 3]

```
1 Promise.all([
2   new Promise((resolve, reject) => {
3     setTimeout(() => resolve(1), 1000)
4   }),
5   2,
6   3
7 ]).then(alert); // 1, 2, 3
```



Таким образом, мы можем передавать уже готовые значения, которые не являются промисами, в Promise.all, иногда это бывает удобно.

Promise.allSettled

Новая возможность

Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

Promise.all завершается с ошибкой, если она возникает в любом из переданных промисов. Это подходит для ситуаций «всё или ничего», когда нам нужны *все* результаты для продолжения:

```
1 Promise.all([
2   fetch('/template.html'),
3   fetch('/style.css'),
4   fetch('/data.json')
5 ]).then(render); // методу render нужны результаты всех
```

Метод Promise.allSettled всегда ждёт завершения всех промисов. В массиве результатов будет

- {status:"fulfilled", value:результат} для успешных завершений,
- {status:"rejected", reason:ошибка} для ошибок.

Например, мы хотели бы загрузить информацию о множестве пользователей. Даже если в каком-то запросе ошибка, нас всё равно интересуют остальные.

Используем для этого Promise.allSettled:

```
1 let urls = [
2   'https://api.github.com/users/iliakan',
3   'https://api.github.com/users/remy',
4   'https://no-such-url'
5 ];
6
7 Promise.allSettled(urls.map(url => fetch(url)))
8   .then(results => { // (*)
9     results.forEach((result, num) => {
10      if (result.status == "fulfilled") {
11        alert(`${urls[num]}: ${result.value.status}`);
12      }
13      if (result.status == "rejected") {
```



Раздел

[Промисы, async/await](#)

Навигация по уроку

[Promise.all](#)

[Promise.allSettled](#)

[Promise.race](#)

[Promise.resolve/reject](#)

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
14         alert(`${urls[num]}: ${result.reason}`);
15     }
16     });
17 }
```

Массив `results` в строке (*) будет таким:

```
1  [
2    {status: 'fulfilled', value: ...объект ответа...},
3    {status: 'fulfilled', value: ...объект ответа...},
4    {status: 'rejected', reason: ...объект ошибки...}
5  ]
```

То есть, для каждого промиса у нас есть его статус и значение/ошибка.

Полифил

Если браузер не поддерживает `Promise.allSettled`, для него легко сделать полифил:

```
1  if(!Promise.allSettled) {
2    Promise.allSettled = function(promises) {
3      return Promise.all(promises.map(p => Promise.resolve(
4        {status: 'fulfilled',
5          value: value
6        }, error => ({
7          status: 'rejected',
8            reason: error
9        })))));
10   };
11 }
```



В этом коде `promises.map` берёт аргументы, превращает их в промисы (на всякий случай) и добавляет каждому обработчик `.then`.

Этот обработчик превращает успешный результат `value` в `{state: 'fulfilled', value: value}`, а ошибку `error` в `{state: 'rejected', reason: error}`. Это как раз и есть формат результатов `Promise.allSettled`.

Затем мы можем использовать `Promise.allSettled`, чтобы получить результаты всех промисов, даже если при выполнении какого-то возникнет ошибка.

Promise.race

Метод очень похож на `Promise.all`, но ждёт только первый промис, из которого берёт результат (или ошибку).

Синтаксис:

```
1  let promise = Promise.race(iterable);
```

Например, тут результат будет 1:

```
1  Promise.race([
2    new Promise((resolve, reject) => setTimeout(() => res, 1000)),
3    new Promise((resolve, reject) => setTimeout(() => rej, 500)),
4    new Promise((resolve, reject) => setTimeout(() => res, 1000))
5  ]).then(alert); // 1
```

Быстрее всех выполнился первый промис, он и дал результат. После этого остальные промисы игнорируются.

Раздел

[Промисы, async/await](#)

Навигация по уроку

[Promise.all](#)

[Promise.allSettled](#)

[Promise.race](#)

[Promise.resolve/reject](#)

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



Promise.resolve/reject

Методы `Promise.resolve` и `Promise.reject` редко используются в современном коде, так как синтаксис `async/await` (мы рассмотрим его [чуть позже](#)) делает его, в общем-то, не нужным.

Мы рассмотрим их здесь для полноты картины, а также для тех, кто по каким-то причинам не может использовать `async/await`.

- `Promise.resolve(value)` создаёт успешно выполненный промис с результатом `value`.

То же самое, что:

```
1 let promise = new Promise(resolve => resolve(value));
```

Этот метод используют для совместимости: когда ожидается, что функция возвратит именно промис.

Например, функция `loadCached` ниже загружает URL и запоминает (кеширует) его содержимое. При будущих вызовах с тем же URL он тут же читает предыдущее содержимое из кеша, но использует

`Promise.resolve`, чтобы сделать из него промис, для того, чтобы возвращаемое значение всегда было промисом:

```
1 let cache = new Map();
2
3 function loadCached(url) {
4   if (cache.has(url)) {
5     return Promise.resolve(cache.get(url)); // (*)
6   }
7
8   return fetch(url)
9     .then(response => response.text())
10    .then(text => {
11      cache.set(url, text);
12      return text;
13    });
14 }
```

Мы можем писать `loadCached(url).then(...)`, потому что функция `loadCached` всегда возвращает промис. Мы всегда можем использовать `.then` после `loadCached`. Это и есть цель использования `Promise.resolve` в строке `(*)`.

Promise.reject

- `Promise.reject(error)` создаёт промис, завершённый с ошибкой `error`.

То же самое, что:

```
1 let promise = new Promise((resolve, reject) => reject(e
```

На практике этот метод почти никогда не используется.

Итого

Мы ознакомились с пятью статическими методами класса `Promise`:

1. `Promise.all(promises)` – ожидает выполнения всех промисов и возвращает массив с результатами. Если любой из указанных промисов вернёт ошибку, то результатом работы `Promise.all` будет эта ошибка, результаты остальных промисов будут игнорироваться.
2. `Promise.allSettled(promises)` (добавлен недавно) – ждёт, пока все промисы завершатся и возвращает их результаты в виде массива с объектами, у каждого объекта два свойства:

Раздел

[Промисы, async/await](#)

Навигация по уроку

[Promise.all](#)

[Promise.allSettled](#)

[Promise.race](#)

[Promise.resolve/reject](#)

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



- `state: "fulfilled"` , если выполнен успешно или `"rejected"` , если ошибка,
- `value` – результат, если успешно или `reason` – ошибка, если нет.

3. `Promise.race(promises)` – ожидает первый выполненный промис, который становится его результатом, остальные игнорируются.
4. `Promise.resolve(value)` – возвращает успешно выполнившийся промис с результатом `value` .
5. `Promise.reject(error)` – возвращает промис с ошибкой `error` .

Из всех перечисленных методов, самый часто используемый – это, пожалуй, `Promise.all` .

Проводим [курсы по JavaScript и фреймворкам](#).

Комментарии

перед тем как писать...

