Промисы

Базовый синтаксис для выглядит так:

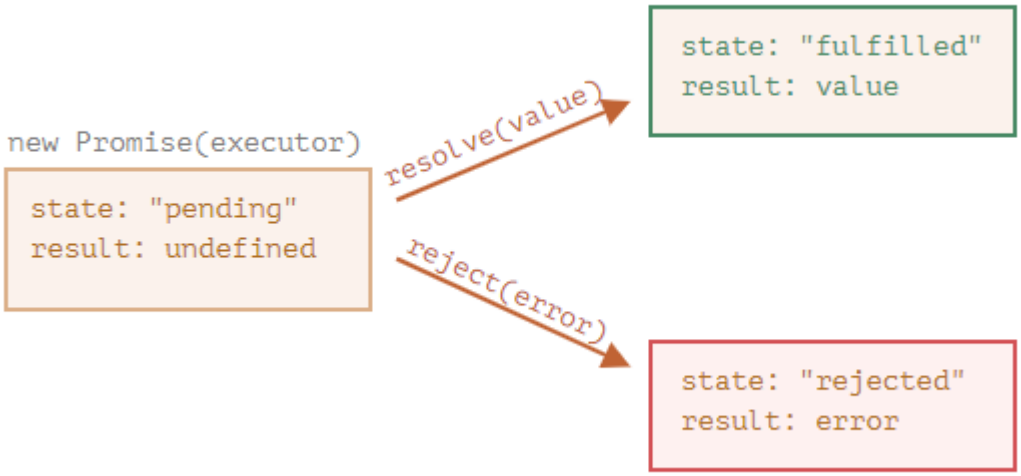
```
let promise = new Promise(function(resolve, reject) {
  // функция-исполнитель (executor)
});}
```

Когда **исполнитель** получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:

- resolve(value) — если работа завершилась успешно, с результатом value.
- reject(error) — если произошла ошибка, error – объект ошибки.
-

У объекта promise, возвращаемого конструктором new Promise, есть внутренние свойства:

- state («состояние») — вначале "**pending**" («ожидание»), потом меняется на "**fulfilled**" («выполнено успешно») при вызове resolve или на "**rejected**" («выполнено с ошибкой») при вызове reject.
- result («результат») — вначале **undefined**, далее изменяется на **value** при вызове resolve(value) или на **error** при вызове reject(error).



Исполнитель должен вызвать **что-то одно**: resolve или reject. Состояние промиса может быть изменено **только один раз**. Все последующие вызовы resolve и reject будут проигнорированы.

В случае, если что-то пошло не так, мы должны вызвать reject. Это можно сделать с аргументом любого типа (как и resolve), но рекомендуется использовать объект Error (или унаследованный от него).

Свойства state и result – это внутренние свойства объекта Promise и мы не имеем к ним прямого доступа. Для обработки результата следует использовать **методы .then/.catch/.finally**, про них речь пойдёт дальше.

then - наиболее важный и фундаментальный метод. Синтаксис:

```
promise.then(
  function(result) { /* обработает успешное выполнение */ },
  function(error) { /* обработает ошибку */ }
);
```

- Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «**выполнен успешно**», и получает результат.
- Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «**выполнен с ошибкой**», и получает ошибку.

Например, вот реакция на успешно выполненный промис:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});
// resolve запустит первую функцию, переданную в .then
promise.then(
  result => alert(result), // выведет "done!" через одну секунду
  error => alert(error) // не будет запущена
);
```

Если мы заинтересованы только в результате успешного выполнения задачи, то в then можно передать только одну функцию

catch

Если мы хотели бы только обработать ошибку, то можно использовать null в качестве первого аргумента: .then(null, errorHandlingFunction). Или можно воспользоваться методом .catch(errorHandlingFunction), который сделает то же самое:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Ошибка!")), 1000);
});
// .catch(f) это то же самое, что promise.then(null, f)
promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду
```

Вызов .catch(f) – это сокращённый, «укороченный» вариант **.then(null, f).**

finally

Вызов .finally(f) похож на **.then(f, f)**, в том смысле, что f выполнится в любом случае, когда промис завершится: успешно или с ошибкой. Например:

```
new Promise((resolve, reject) => {
  /* сделать что-то, что займёт время, и после вызвать resolve/reject */
})
  // выполнится, когда промис завершится, независимо от того, успешно или нет
  .finally(() => остановить индикатор загрузки)
  .then(result => показать результат, err => показать ошибку)
```

Но это не совсем псевдоним then(f,f), как можно было подумать. Существует несколько важных отличий:

- Обработчик, вызываемый из finally, **не имеет аргументов**. В finally мы не знаем, как был завершён промис. И это нормально, потому что обычно наша задача – выполнить «общие» завершающие процедуры.
- Обработчик finally **«пропускает» результат или ошибку дальше**, к последующим обработчикам.

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));

    document.head.append(script);
  });
}
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`${script.src} загружен!`),
  error => alert(`Ошибка: ${error.message}`)
);

promise.then(script => alert('Ещё один обработчик...'));
```

Цепочка промисов

```
promise.then(
    function(result) {
        return result + '!';
    }
).then(
    function(result) {
        console.log(result); // выведет 'string!'
    }
);
```

Мы можем к результату первого then применить еще один then, создав тем самым цепочку методов. При этом в результат следующего метода будет попадать то, что вернул через return предыдущий. Функции цепочки могут также возвращать промисы. В этом случае результат этого промиса попадет в следующий then

Обработка ошибок

Цепочки промисов отлично подходят для **перехвата ошибок**. Если промис завершается с ошибкой, то управление переходит в **ближайший обработчик ошибок** (then, в котором есть функция-обработчик ошибки, либо в первому catch, смотря что встретится раньше). Самый лёгкий путь перехватить все ошибки – это добавить .catch в конец цепочки.

Функция-обработчик имеет два варианта действий: если она справилась с исключительной ситуацией, то может вернуть результат через return и выполнение продолжится дальше по цепочке. Если же она не справилась с ошибкой, то может или ничего не возвращать, или выбросить исключение через **throw**. В этом случае выполнение перейдет к следующему перехватчику ошибки

```
promise.then(
    function(result) {
        return result + '1';
    }
).then(
    function(result) {
        if (всеХорошо) {
            return result + '2';
        } else {
            throw new Error('ошибка'); // переходим к ближайшему перехватчику
        }
    }
)
.then(
    function(result) {
        return result + '3';
    }
).catch(
    function(error) {
        // ближайший перехватчик
    }
);
```

Неявный try…catch

Вокруг функции промиса и обработчиков находится "невидимый try..catch". Если происходит исключение, то оно перехватывается, и промис считается отклонённым с этой ошибкой. Например, этот код:

```
new Promise((resolve, reject) => {
  throw new Error("Ошибка!");
}).catch(alert); // Error: Ошибка!
```

…Работает так же, как и этот:

```
new Promise((resolve, reject) => {
  reject(new Error("Ошибка!"));
}).c
```

Необработанные ошибки

Что произойдёт, если ошибка не будет обработана? В случае ошибки выполнение должно перейти к ближайшему обработчику ошибок. Что, если его нет? JavaScript-движок отслеживает такие ситуации и генерирует в этом случае глобальную ошибку. Можно увидеть ее в консоли. В браузере мы можем поймать такие ошибки, используя событие **unhandledrejection**:

```
window.addEventListener('unhandledrejection', function(event) {
  // объект события имеет два специальных свойства:
  alert(event.promise); // [object Promise] - промис, который сгенерировал ошибку
  alert(event.reason); // Error: Ошибка! - объект ошибки, которая не была обработана
});
new Promise(function() {
  throw new Error("Ошибка!");
}); // нет обработчика ошибок
```

Это событие является частью стандарта HTML.

Promise API

В классе Promise есть 5 статических методов.

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.

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

Например:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // выполнится спустя 3 секунды,  результат будет 1,2,3
// каждый промис даёт элемент массива
```

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.

Если любой из промисов завершится с ошибкой, то промис, возвращённый Promise.all, немедленно завершается с этой ошибкой. В случае ошибки, остальные результаты игнорируются

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».

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2,
  3
]).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 - метод Promise.allSettled в отлтчие от Promise.all **всегда** ждёт завершения всех промисов. В массиве результатов будет

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

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

Массив results в строке (*) будет таким:

```
[
  {status: 'fulfilled', value: ...объект ответа...},
  {status: 'fulfilled', value: ...объект ответа...},
  {status: 'rejected', reason: ...объект ошибки...}
]
```

Метод появился недавно. Для него есть полифил.

Promise.race - Метод очень похож на Promise.all, но ждёт только первый выполненный промис, из которого берёт результат (или ошибку). Синтаксис:

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

Например, тут результат будет 1:

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ошибка!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

Promise.resolve/reject

- **Promise.resolve(value)** – возвращает успешно выполнившийся промис с результатом value (То же самое, что: let promise = new Promise(resolve => resolve(value));).
- **Promise.reject(error)** – возвращает промис с ошибкой error.

Методы Promise.resolve и Promise.reject редко используются в современном коде, так как синтаксис async/await (делает его, в общем-то, не нужным. Используются для совместимости: когда ожидается, что функция возвратит именно промис.

Промисификация

Промисификация – это, когда мы берём функцию, которая принимает колбэк и меняем её, чтобы она вместо этого возвращала промис.

Например:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Ошибка загрузки скрипта ${src}`));
  document.head.append(script);
}

loadScript('path/script.js', (err, script) => {...})
```

Промисификация

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err)
      else resolve(script);
    });
  })
}

// использование:
// loadScriptPromise('path/script.js').then(...)
```

Помните, промис может иметь только один результат, но колбэк технически может вызываться сколько угодно раз. Поэтому промисификация используется для функций, которые вызывают колбэк только один раз. Последующие вызовы колбэка будут проигнорированы.

promisify(f) принимает функцию для промисификации f и возвращает функцию-обёртку.

```
// promisify(f, true), чтобы получить массив результатов
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // наш специальный колбэк для f
        if (err) {
          return reject(err);
        } else {
          // делаем resolve для всех results колбэка, если задано manyArgs
          resolve(manyArgs ? results : results[0]);
        }
```

```
    }

    args.push(callback);

    f.call(this, ...args);
  });
 };
};

// использование:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...)
```