> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад, присоединившись к русскоязычному сообществу MDN Web Docs.

Использование промисов

`Promise` (промис) - это объект, представляющий результат успешного или неудачного завершения асинхронной операции. Так как большинство людей пользуются уже созданными промисами, это руководство начнём с объяснения использования вернувшихся промисов до объяснения принципов создания.

В сущности, промис - это возвращаемый объект, в который вы записываете два колбэка вместо того, чтобы передать их функции.

Например, вместо старомодной функции, которая принимает два колбэка и вызывает один из них в зависимости от успешного или неудачного завершения операции:

JS

```
function doSomethingOldStyle(successCallback, failureCallback) {
  console.log("Готово.");
  // Успех в половине случаев.
  if (Math.random() > 0.5) {
    successCallback("Успех");
  } else {
    failureCallback("Ошибка");
  }
}

function successCallback(result) {
  console.log("Успешно завершено с результатом " + result);
}
```

```
function failureCallback(error) {
  console.log("Завершено с ошибкой " + error);
}

doSomethingOldStyle(successCallback, failureCallback);
```

...современные функции возвращают промис, в который вы записываете ваши колбэки:

JS

```
function doSomething() {
  return new Promise((resolve, reject) => {
    console.log("Готово.");
    // Успех в половине случаев.
    if (Math.random() > 0.5) {
      resolve("Успех");
    } else {
      reject("Ошибка");
    }
  });
}

const promise = doSomething();
promise.then(successCallback, failureCallback);
```

...или просто:

JS

```
doSomething().then(successCallback, failureCallback);
```

Мы называем это *асинхронным вызовом функции*. У этого соглашения есть несколько преимуществ. Давайте рассмотрим их.

Гарантии

В отличие от старомодных переданных колбэков промис даёт некоторые гарантии:

- Колбэки никогда не будут вызваны до завершения обработки текущего события в событийном цикле JavaScript.

- Колбэки, добавленные через `.then` даже *после* успешного или неудачного завершения асинхронной операции, будут также вызваны.

Цепочка вызовов

Но наиболее непосредственная польза от промисов - цепочка вызовов (*chaining*).

Общая нужда - выполнять две или более асинхронных операции одна за другой, причём каждая следующая начинается при успешном завершении предыдущей и использует результат её выполнения. Мы реализуем это, создавая цепочку вызовов промисов (*promise chain*).

Вот в чём магия: функция `then` возвращает новый промис, отличающийся от первоначального:

```
JS
let promise = doSomething();
let promise2 = promise.then(successCallback, failureCallback);
```

или

```
JS
let promise2 = doSomething().then(successCallback, failureCallback);
```

Второй промис представляет завершение не только `doSomething()`, но и функций `successCallback` или `failureCallback`, переданных вами, а они тоже могут быть асинхронными функциями, возвращающими промис. В этом случае все колбэки, добавленные к `promise2` будут поставлены в очередь за промисом, возвращаемым `successCallback` или `failureCallback`.

По сути, каждый вызванный промис означает успешное завершение предыдущих шагов в цепочке.

Раньше выполнение нескольких асинхронных операций друг за другом приводило к классической "Вавилонской башне" колбэков:

JS

```
doSomething(function (result) {
  doSomethingElse(
    result,
    function (newResult) {
      doThirdThing(
        newResult,
        function (finalResult) {
          console.log("Итоговый результат: " + finalResult);
        },
        failureCallback,
      );
    },
    failureCallback,
  );
}, failureCallback);
```

В современных функциях мы записываем колбэки в возвращаемые промисы - формируем цепочку промисов:

JS

```
doSomething()
  .then(function (result) {
    return doSomethingElse(result);
  })
  .then(function (newResult) {
    return doThirdThing(newResult);
  })
  .then(function (finalResult) {
    console.log("Итоговый результат: " + finalResult);
  })
  .catch(failureCallback);
```

Аргументы `then` необязательны, а `catch(failureCallback)` - это сокращение для `then(null, failureCallback)`. Вот как это выражено с помощью стрелочных функций:

JS

```
doSomething()
  .then((result) => doSomethingElse(result))
  .then((newResult) => doThirdThing(newResult))
  .then((finalResult) => {
    console.log(`Итоговый результат: ${finalResult}`);
  })
  .catch(failureCallback);
```

Важно: Всегда возвращайте промисы в return, иначе колбэки не будут сцеплены и ошибки могут быть не пойманы (стрелочные функции неявно возвращают результат, если скобки {} вокруг тела функции опущены).

Цепочка вызовов после catch

Можно продолжить цепочку вызовов *после* ошибки, т. е. после `catch`, что полезно для выполнения новых действий даже после того, как действие вернёт ошибку в цепочке вызовов. Ниже приведён пример:

```
new Promise((resolve, reject) => {
    console.log('Начало');

    resolve();
})
.then(() => {
    throw new Error('Где-то произошла ошибка');

    console.log('Выведи это');
})
.catch(() => {
    console.log('Выведи то');
})
.then(() => {
    console.log('Выведи это, несмотря ни на что');
});
```

В результате выведется данный текст:

```
Начало
Выведи то
Выведи это, несмотря ни на что
```

Заметьте, что текст "Выведи это" не вывелся, потому что "Где-то произошла ошибка" привела к отказу

Распространение ошибки

Вы могли ранее заметить, что `failureCallback` повторяется три раза в `"pyramid of doom"`, а в цепочке промисов всего лишь один раз:

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => console.log(`Итоговый результат: ${finalResult}`))
.catch(failureCallback);
```

В основном, цепочка промисов останавливает выполнение кода, если где-либо произошла ошибка, и вместо этого ищет далее по цепочке обработчики ошибок. Это очень похоже на то, как работает синхронный код:

```
try {
  let result = syncDoSomething();
  let newResult = syncDoSomethingElse(result);
  let finalResult = syncDoThirdThing(newResult);
  console.log(`Итоговый результат: ${finalResult}`);
} catch(error) {
  failureCallback(error);
}
```

Эта симметрия с синхронным кодом лучше всего показывает себя в синтаксическом сахаре `async` / `await` в ECMAScript 2017:

```
async function foo() {
  try {
    let result = await doSomething();
    let newResult = await doSomethingElse(result);
    let finalResult = await doThirdThing(newResult);
    console.log(`Итоговый результат: ${finalResult}`);
  } catch(error) {
    failureCallback(error);
  }
}
```

Работа данного кода основана на промисах. Для примера здесь используется функция `doSomething()`, которая встречалась ранее. Вы можете прочитать больше о синтаксисе здесь

Промисы решают основную проблему пирамид, обработку всех ошибок, даже вызовов исключений и программных ошибок. Это основа для функционального построения асинхронных операций.

Создание промиса вокруг старого колбэка

`Promise` может быть создан с помощью конструктора. Это может понадобится только для старых API.

В идеале, все асинхронные функции уже должны возвращать промис. Но увы, некоторые APIs до сих пор ожидают успешного или неудачного колбэка переданных по старинке. Типичный пример: `setTimeout()` функция:

```
setTimeout(() => saySomething("10 seconds passed"), 10000);
```

Смешивание старого колбэк-стиля и промисов проблематично. В случае неудачного завершения `saySomething` или программной ошибки, нельзя обработать ошибку.

К счастью мы можем обернуть функцию в промис. Хороший тон оборачивать проблематичные функции на самом низком возможном уровне, и больше никогда их не вызывать напрямую:

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

wait(10000).then(() => saySomething("10 seconds")).catch(failureCallback);
```

В сущности, конструктор промиса становится исполнителем функции, который позволяет нам резолвить или режектить промис вручную. Так как `setTimeout` всегда успешен, мы опустили reject в этом случае.

Композиция

`Promise.resolve()` и `Promise.reject()` короткий способ создать уже успешные или отклонённые промисы соответственно. Это иногда бывает полезно.

`Promise.all()` и `Promise.race()` - два метода запустить асинхронные операции параллельно.

Последовательное выполнение композиции возможно при помощи хитрости JavaScript:

```
[func1, func2].reduce((p, f) => p.then(f), Promise.resolve());
```

Фактически, мы превращаем массив асинхронных функций в цепочку промисов равносильно: `Promise.resolve().then(func1).then(func2);`

Это также можно сделать, объединив композицию в функцию, в функциональном стиле программирования:

```
const applyAsync = (acc,val) => acc.then(val);
const composeAsync = (...funcs) => x => funcs.reduce(applyAsync,
Promise.resolve(x));
```

`composeAsync` функция примет любое количество функций в качестве аргументов и вернёт новую функцию которая примет в параметрах начальное значение, переданное по цепочке. Это удобно, потому что некоторые или все функции могут быть либо асинхронными, либо синхронными, и они гарантированно выполнятся в правильной последовательности:

```
const transformData = composeAsync(func1, asyncFunc1, asyncFunc2, func2);
transformData(data);
```

В ECMAScript 2017, последовательные композиции могут быть выполнены более простым способом с помощью async/await:

```
for (const f of [func1, func2]) {
  await f();
}
```

Порядок выполнения

Чтобы избежать сюрпризов, функции, переданные в `then` никогда не будут вызваны синхронно, даже с уже разрешённым промисом:

```
Promise.resolve().then(() => console.log(2));
console.log(1); // 1, 2
```

Вместо немедленного выполнения, переданная функция встанет в очередь микрозадач, а значит выполнится, когда очередь будет пустой в конце текущего вызова JavaScript цикла событий (event loop), т.е. очень скоро:

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

wait().then(() => console.log(4));
Promise.resolve().then(() => console.log(2)).then(() => console.log(3));
console.log(1); // 1, 2, 3, 4
```

Вложенность

Простые цепочки promise лучше оставлять без вложений, так как вложенность может быть результатом небрежной структуры. Смотрите распространённые ошибки.

Вложенность - это управляющая структура, ограничивающая область действия операторов catch. В частности, вложенный catch только перехватывает сбои в своей области и ниже, а не ошибки выше в цепочке за пределами вложенной области. При правильном использовании это даёт большую точность в извлечение ошибок:

```
doSomethingCritical()
.then(result => doSomethingOptional()
  .then(optionalResult => doSomethingExtraNice(optionalResult))
  .catch(e => {})) // Игнорируется если необязательные параметр не выкинул
исключение
.then(() => moreCriticalStuff())
.catch(e => console.log("Критическая ошибка: " + e.message));
```

Обратите внимание, что необязательный шаги здесь выделены отступом.

Внутренний оператор catch нейтрализует и перехватывает ошибки только от doSomethingOptional() и doSomethingExtraNice(), после чего код возобновляется с помощью moreCriticalStuff(). Важно, что в случае сбоя doSomethingCritical() его ошибка перехватывается только последним (внешним) catch.

Частые ошибки

В этом разделе собраны частые ошибки, возникающие при создании цепочек промисов. Несколько таких ошибок можно увидеть в следующем примере:

```
// Плохой пример! Три ошибки!

doSomething().then(function(result) {
  doSomethingElse(result) // Забыл вернуть промис из внутренней цепочки + неуместное влаживание
  .then(newResult => doThirdThing(newResult));
}).then(() => doFourthThing());
// Забыл закончить цепочку методом catch
```

Первая ошибка это неправильно сцепить вещи между собой. Такое происходит когда мы создаём промис но забываем вернуть его. Как следствие, цепочка сломана, но правильнее было бы сказать что теперь у нас есть две независимые цепочки, соревнующиеся за право разрешится первой. Это означает, что `doFourthThing()` не будет ждать `doSomethingElse()` или `doThirdThing()` пока тот закончится, и будет исполнятся параллельно с ними, это, вероятно, не то что хотел разработчик. Отдельные цепочки также имеют отдельную обработку ошибок, что приводит к необработанным ошибкам.

Вторая ошибка это излишняя вложенность, включая первую ошибку. Вложенность также ограничивает область видимости внутренних обработчиков ошибок, если это не то чего хотел разработчик, это может привести к необработанным ошибкам. Примером этого является пример как не нужно создавать промисы , который комбинирует вложенность с чрезмерным использованием конструктора промисов для оборачивания кода который уже использует промисы.

Третья ошибка это забыть закончить цепочку ключевым словом `catch`. Незаконченные цепочки приводят к необработанным отторжениям промисов в большинстве браузеров.

Хорошим примером является всегда либо возвращать либо заканчивать цепочки промисов, и как только вы получаете новый промис, возвращайте его сразу же, чтобы не усложнять код излишней вложенностью:

```
doSomething()
.then(function(result) {
  return doSomethingElse(result);
})
.then(newResult => doThirdThing(newResult))
.then(() => doFourthThing())
.catch(error => console.log(error));
```

Обратите внимание что `() => x` это сокращённая форма `() => { return x; }`.

Теперь у нас имеется единственная определённая цепочка с правильной обработкой ошибок.

Использование `async` / `await` предотвращает большинство, если не все вышеуказанные ошибки, но взамен появляется другая частая ошибка — забыть ключевое слово `await`.

Смотрите также

- `Promise.then()`
- Спецификация Promises/A+ (EN)
- Нолан Лоусон (Nolan Lawson): У нас проблемы с промисами - распространённые ошибки (EN)

This page was last modified on 7 авг. 2023 г. by MDN contributors.