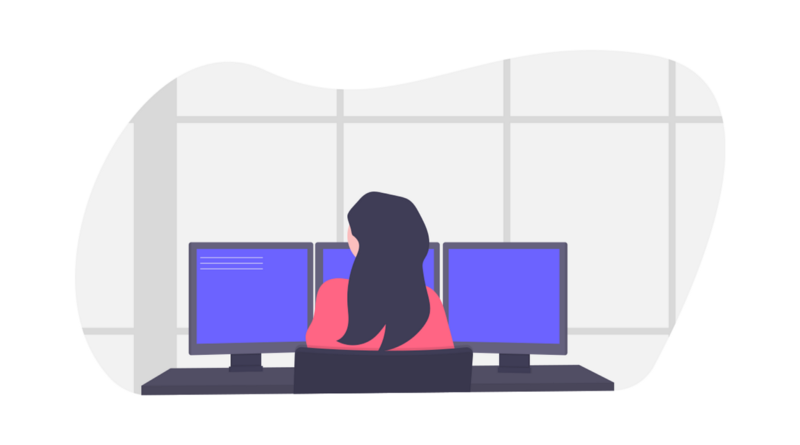
[Jintsuu](https://habr.com/ru/users/Jintsuu/" \o "Автор публикации)10 февраля 2019 в 19:40

Разбираемся с промисами в JavaScript

[Автор оригинала: Sukhjinder Arora](https://blog.bitsrc.io/understanding-promises-in-javascript-c5248de9ff8f)

* [JavaScript](https://habr.com/ru/hub/javascript/)
* Перевод

Доброго времени суток, Хабр! Представляю вашему вниманию перевод статьи [«Understanding Promises in JavaScript»](https://blog.bitsrc.io/understanding-promises-in-javascript-c5248de9ff8f) автора Sukhjinder Arora.  
  
  
  
*От автора перевода: Так же, как и сам автор, я надеюсь, что статья оказалась для вас полезной. Пожалуйста, если она и вправду помогла вам узнать для себя что-то новое, то не поленитесь зайти на оригинал статьи и поблагодарить автора! Буду рад вашему фидбеку!*[*Ссылка на перевод статьи по асинхронному JavaScript от этого же автора*](https://habr.com/ru/post/439620/)*.*  
  
JavaScript — это однопоточный язык программирования, это означает, что за раз может быть выполнено что-то одно. До ES6 мы использовали обратные вызовы, чтобы управлять асинхронными задачами, такими как сетевой запрос.  
  
Используя промисы, мы можем избегать “ад обратных вызовов” и сделать наш код чище, более читабельным и более простым для понимания.  
  
Предположим, что мы хотим асинхронно получить некоторые данные с сервера, используя обратные вызовы мы сделали бы что-то вроде этого:

getData(function(x){

console.log(x);

getMoreData(x, function(y){

console.log(y);

getSomeMoreData(y, function(z){

console.log(z);

});

});

Здесь я запрашиваю некоторые данные с сервера при помощи функции *getData()*, которая получает данные внутри функции обратного вызова. Внутри функции обратного вызова я запрашиваю дополнительные данные при помощи вызова функции *getMoreData()*, передавая предыдущие данные как аргумент и так далее.  
  
Это то, что мы называем “адом обратных вызовов”, где каждый обратный вызов вложен внутрь другого, и каждый внутренний обратный вызов зависит от его родителя.  
  
Мы можем переписать приведенный выше фрагмент используя промисы:

getData()

.then((x) => {

console.log(x);

return getMoreData(x);

})

.then((y) => {

console.log(y);

return getSomeMoreData(y);

})

.then((z) => {

console.log(z);

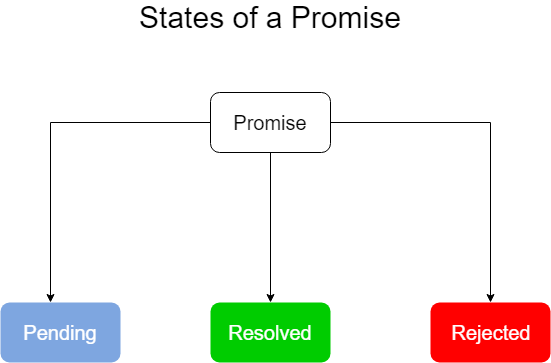
});

Вы можете видеть, что стало более читабельно, чем в случае первого примера с обратными вызовами.

Что такое Промисы?

Промис(Обещание) — это объект который содержит будущее значение асинхронной операции. Например, если вы запрашиваете некоторые данные с сервера, промис обещает нам получить эти данные, которые мы сможем использовать в будущем.  
  
Прежде чем погрузиться во все эти технические штуки, давайте разберемся с терминологией промисов.

Состояния промисов

Промис в JavaScript, как и обещание в реальной жизни, имеет 3 состояния. Это может быть 1) нерешенный(в ожидании), 2) решенный/resolved (выполненный) или 3) отклоненный/rejected.  
  
  
  
**Нерешенный или Ожидающий** — Промис ожидает, если результат не готов. То есть, ожидает завершение чего-либо(например, завершения асинхронной операции).  
**Решенный или Выполненный** — Промис решен, если результат доступен. То есть, что-то завершило свое выполнение(например, асинхронная операция) и все прошло хорошо.  
**Отклоненный** — Промиc отклонен, если произошла ошибка в процессе выполнения.  
  
Теперь мы знаем, что такое Промис и его терминологию, давайте вернемся назад к практической части промисов.

Создаем Промис

В большинстве случаев вы будете просто использовать промисы, а не создавать их, но все же важно знать как они создаются.  
  
Синтаксис:

const promise = new Promise((resolve, reject) => {

...

});

Мы создали новый промис, используя конструктор Промисов, он принимает один аргумент, обратный вызов, также известный как исполнительная функция, которая принимает 2 обратных вызова, *resolve* и *reject*.  
  
Исполнительная функция выполняется сразу же после создания промиса. Промис становится выполненным при помощи вызова *resolve()*, а отклоненным при помощи *reject()*. Например:

const promise = new Promise((resolve, reject) => {

if(allWentWell) {

resolve('Все прошло отлично!');

} else {

reject('Что-то пошло не так');

}

});

*resolve()* и *reject()* принимают один аргумент, который может быть строкой, числом, логическим выражением, массивом или объектом.  
  
Давайте взглянем на другой пример, чтобы полностью понять как создаются промисы.

const promise = new Promise((resolve, reject) => {

const randomNumber = Math.random();

setTimeout(() => {

if(randomNumber < .6) {

resolve('Все прошло отлично!');

} else {

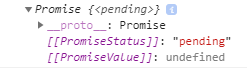
reject('Что-то пошло не так');

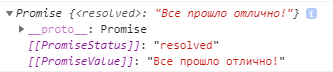
}

}, 2000);

});

Здесь я создал новый промис используя конструктор Промисов. Промис выполняется или отклоняется через 2 секунды после его создания. Промис выполняется, если *randomNumber* меньше, чем .6 и отклоняется в остальных случаях.  
  
Когда промис был создан, он будет в состоянии ожидания и его значение будет *undefined*.



После 2 секунд таймер заканчивается, промис случайным образом либо выполняется, либо отклоняется, и его значением будет то, которое передано в функцию *resolve* или *reject*. Ниже пример двух случаев:  
  
Успешное выполнение:  
  
  
  
Отклонение промиса:  
  
  
  
**Примечание:** Промис может быть выполнен или отклонен только один раз. Дальнейшие вызовы *resolve()* или *reject()* никак не повлияют на состояние промиса. Пример:

const promise = new Promise((resolve, reject) => {

resolve('Promise resolved'); *// Промис выполнен*

reject('Promise rejected'); *// Промис уже не может быть отклонен*

});

Так как *resolve()* была вызвана первой, то промис теперь получается статус “выполненный”. Последующий вызов *reject()* никак не повлияет на состояние промиса.

Использование Промиса

Теперь мы знаем как создавать промисы, давайте теперь разберемся как применять уже созданный промис. Мы используем промисы при помощи методов *then()* и *catch()*.  
  
Например, запрос данных из API при помощи *fetch*, которая возвращает промис.  
  
*.then()* **синтаксис:** *promise.then(successCallback, failureCallback)*  
  
*successCallback* вызывается, если промис был успешно выполнен. Принимает один аргумент, который является значением переданным в *resolve()*.  
  
*failureCallback* вызывается, если промис был отклонен. Принимает один аргумент, который является значением преданным в *reject()*.  
  
Пример:

const promise = new Promise((resolve, reject) => {

const randomNumber = Math.random();

if(randomNumber < .7) {

resolve('Все прошло отлично!');

} else {

reject(new Error('Что-то пошло не так'));

}

});

promise.then((data) => {

console.log(data); *// вывести 'Все прошло отлично!'*

},

(error) => {

console.log(error); *// вывести ошибку*

}

);

Если промис был выполнен, то вызывается *successCallback* со значением, переданным в *resolve()*. И если промис был отклонен, то вызывается *failureCallback* со значением, переданным в reject().  
  
*.catch()* **синтаксис:** *promise.catch(failureCallback)*  
  
Мы используем *catch()* для обработки ошибок. Это более читабельно, нежели обработка ошибок внутри *failureCallback* внутри обратного вызова метода *then()*.

const promise = new Promise((resolve, reject) => {

reject(new Error('Что-то пошло не так'));

});

promise

.then((data) => {

console.log(data);

})

.catch((error) => {

console.log(error); *// вывести ошибку*

});

Цепочка промисов

Методы *then()* и *catch()* также могут возвращать новый промис, который может быть обработан цепочкой других then() в конце предыдущего метода then().  
  
Мы используем цепочку промисов, когда хотим выполнить последовательность промисов.  
  
Например:

const promise1 = new Promise((resolve, reject) => {

resolve('Promise1 выполнен');

});

const promise2 = new Promise((resolve, reject) => {

resolve('Promise2 выполнен');

});

const promise3 = new Promise((resolve, reject) => {

reject('Promise3 отклонен');

});

promise1

.then((data) => {

console.log(data); *// Promise1 выполнен*

return promise2;

})

.then((data) => {

console.log(data); *// Promise2 выполнен*

return promise3;

})

.then((data) => {

console.log(data);

})

.catch((error) => {

console.log(error); *// Promise3 отклонен*

});

И так, что тут происходит?

Когда *promise1* выполнен, вызывается метод *then(),* который возвращает promise2.  
Далее, когда выполнен *promise2*, снова вызывается *then()* и возвращает *promise3*.  
  
Так как promise3 отклонен, вместо следующего *then()*, вызывается *catch()*, который и обрабатывает отклонение *promise3*.  
  
**Примечание:** Как правило достаточно одного метода *catch()* для обработки отклонения любого из промисов в цепочке, если этот метод находится в конце неё.

Распространенная ошибка

Достаточно много новичков делают ошибку, вкладывая одни промисы внутрь других. Например:

const promise1 = new Promise((resolve, reject) => {

resolve('Promise1 выполнен');

});

const promise2 = new Promise((resolve, reject) => {

resolve('Promise2 выполнен');

});

const promise3 = new Promise((resolve, reject) => {

reject('Promise3 отклонен');

});

promise1.then((data) => {

console.log(data); *// Promise1 выполнен*

promise2.then((data) => {

console.log(data); *// Promise2 выполнен*

promise3.then((data) => {

console.log(data);

}).catch((error) => {

console.log(error); *// Promise3 отклонен*

});

}).catch((error) => {

console.log(error);

})

}).catch((error) => {

console.log(error);

});

Несмотря на то, что это будет работать нормально, это считается плохим стилем и делает код менее читабельным. Если у вас есть последовательность промисов для выполнения, будет лучше ставить их один за другим, нежели вкладывать один внутрь другого.

Promise.all( )

Этот метод берет массив промисов и возвращает новый промис, который будет выполненным, когда все промисы внутри массива выполнены или отклонен, как только встречается промис, который отклоняется. Например:

const promise1 = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Promise1 выполнен');

}, 2000);

});

const promise2 = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Promise2 выполнен');

}, 1500);

});

Promise.all([promise1, promise2])

.then((data) => console.log(data[0], data[1]))

.catch((error) => console.log(error));

Здесь аргументом внутри *then()* выступает массив, который содержит значения промисов в том же порядке, в котором они передавались в *Promise.all()*.(Только в том случае, если все промисы выполняются)  
  
Промис отклоняется с причиной отклонения первого промиса в переданном массиве. Например:

const promise1 = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Promise1 выполнен');

}, 2000);

});

const promise2 = new Promise((resolve, reject) => {

setTimeout(() => {

reject('Promise2 отклонен');

}, 1500);

});

Promise.all([promise1, promise2])

.then((data) => console.log(data[0], data[1]))

.catch((error) => console.log(error)); *// Promise2 отклонен*

Здесь у нас есть два промиса, где один выполняется через 2 секунды, а другой отклоняется через 1.5 секунды. Как только второй промис отклоняется, возвращенный от *Promise.all()* промис отклоняется не дожидаясь выполнения первого.  
  
Этот метод может быть полезен, когда у вас есть более одного промиса и вы хотите знать, когда все промисы выполнены. Например, если вы запрашиваете данные из стороннего API и вы хотите что-то сделать с этими данными только тогда, когда все запросы проходят успешно.  
  
По итогу мы имеем *Promise.all()*, который ждет успешное выполнение всех промисов, либо завершает свое выполнение при обнаружении первой неудачи в массиве промисов.

Promise.race( )

Этот метод принимает массив промисов и возвращает один новый промис, который будет выполненным, как только встретится выполненный промис в массиве или же отклоняется, если отклоненный промис встречается раньше. Например:

const promise1 = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Promise1 выполнен');

}, 1000);

});

const promise2 = new Promise((resolve, reject) => {

setTimeout(() => {

reject('Promise2 отклонен');

}, 1500);

});

Promise.race([promise1, promise2])

.then((data) => console.log(data)) *// Promise1 выполнен*

.catch((error) => console.log(error));

Тут мы имеем два промиса, где один выполняется через 1 секунду, а другой отклоняется через 1.5 секунды. Как только первый промис выполнен, возвращенный из Promise.race() промис будет иметь статус выполненного не дожидаясь статуса второго промиса.  
  
Здесь *data*, которая передается в *then()* является значением первого, выполненного, промиса.  
  
По итогу, *Promise.race()* дожидается первого промиса и берет его статус как статус возвращаемого промиса.  
  
*Комментарий автора перевода: Отсюда собственно и название. Race — гонка*

Заключение

Мы узнали, что такое промисы и с чем их едят в JavaScript. Промисы состоят из двух частей 1) Создание промиса и 2) Использование промиса. Большую часть времени вы будете пользоваться промисами, нежели создавать их, но важно знать как они создаются.  
  
Вот и все, надеюсь эта статья оказалась для вас полезной!