

Раздел

[Промисы, async/await](#)

Навигация по уроку


Потребители: then, catch, finally

Пример: loadScript

Задачи (3)

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠 → Язык программирования JavaScript](#)
[→ Промисы, async/await](#) 15-го июля 2020

Промисы

Представьте, что вы известный певец, которого фанаты постоянно донимают расспросами о предстоящем сингле.

Чтобы получить передышку, вы обещаете разослать им сингл, когда он будет выпущен. Вы даёте фанатам список, в который они могут записаться. Они могут оставить там свой e-mail, чтобы получить песню, как только она выйдет. И даже больше: если что-то пойдёт не так, например, в студии будет пожар и песню выпустить не выйдет, они также получат уведомление об этом.

Все счастливы! Вы счастливы, потому что вас больше не донимают фанаты, а фанаты могут больше не беспокоиться, что пропустят новый сингл.

Это аналогия из реальной жизни для ситуаций, с которыми мы часто сталкиваемся в программировании:

1. Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети. В нашей аналогии это – «певец».
2. Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции. Это – «фанаты».
3. Promise (по англ. promise, будем называть такой объект «промис») – это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе. В терминах нашей аналогии – это «список для подписки». «Создающий» код может выполняться сколько потребуется, чтобы получить результат, а промис делает результат доступным для кода, который подписан на него, когда результат готов.

Аналогия не совсем точна, потому что объект Promise в JavaScript гораздо сложнее простого списка подписок: он обладает дополнительными возможностями и ограничениями. Но для начала и такая аналогия хороша.

Синтаксис создания Promise:

```
1 let promise = new Promise(function(resolve, reject) {  
2   // функция-исполнитель (executor)  
3   // "певец"  
4 });
```

Функция, переданная в конструкцию new Promise, называется *исполнитель* (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат. В терминах нашей аналогии: *исполнитель* – это «певец».

Её аргументы resolve и reject – это колбэки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.

Когда он 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.

Итак, исполнитель запускается автоматически, он должен выполнить работу, а затем вызвать resolve или reject.

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

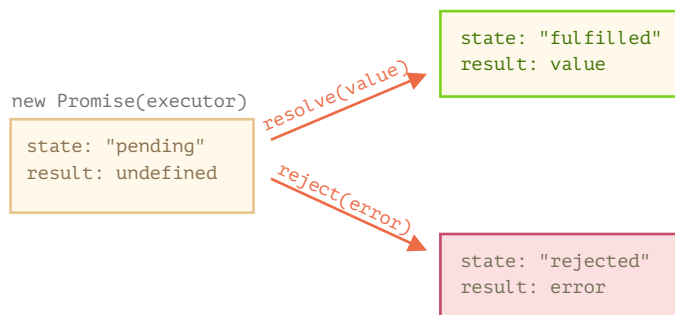
- state («состояние») – вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve



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) .

Так что исполнитель по итогу переводит promise в одно из двух состояний:



Позже мы рассмотрим, как «фанаты» узнают об этих изменениях.

Ниже пример конструктора Promise и простого исполнителя с кодом, дающим результат с задержкой (через setTimeout):

```
1 let promise = new Promise(function(resolve, reject) {
2   // эта функция выполнится автоматически, при вызове new Promise
3
4   // через 1 секунду сигнализировать, что задача выполнена
5   setTimeout(() => resolve("done"), 1000);
6 });
```

Мы можем наблюдать две вещи, запустив код выше:

1. Функция-исполнитель запускается сразу же при вызове new Promise .
2. Исполнитель получает два аргумента: resolve и reject – это функции, встроенные в JavaScript, поэтому нам не нужно их писать. Нам нужно лишь позаботиться, чтобы исполнитель вызвал одну из них по готовности.

Спустя одну секунду «обработки» исполнитель вызовет resolve("done") , чтобы передать результат:



Это был пример успешно выполненной задачи, в результате мы получили «успешно выполненный» промис.

А теперь пример, в котором исполнитель сообщит, что задача выполнена с ошибкой:

```
1 let promise = new Promise(function(resolve, reject) {
2   // спустя одну секунду будет сообщено, что задача выполнена с ошибкой
3   setTimeout(() => reject(new Error("Whoops!")), 1000);
4 });
```



Подведём промежуточные итоги: исполнитель выполняет задачу (что-то, что обычно требует времени), затем вызывает resolve или reject , чтобы изменить состояние соответствующего Promise .

Промис – и успешный, и отклонённый будем называть «завершённым», в отличие от изначального промиса «в ожидании».

Раздел

[Промисы, async/await](#)

Навигация по уроку

Потребители: `then`, `catch`, `finally`

Пример: `loadScript`

Задачи (3)

Комментарии

Поделиться



[Редактировать на GitHub](#)



❗ Может быть что-то одно: либо результат, либо ошибка

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:

```
1 let promise = new Promise(function(resolve, reject)
2   resolve("done");
3
4   reject(new Error("...")); // игнорируется
5   setTimeout(() => resolve("...")); // игнорируется
6 });
```

Идея в том, что задача, выполняемая исполнителем, может иметь только один итог: результат или ошибку.

Также заметим, что функция `resolve / reject` ожидает только один аргумент (или ни одного). Все дополнительные аргументы будут проигнорированы.

❗ Вызывайте `reject` с объектом `Error`

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). Почему так? Скоро нам станет понятно.

❗ Вызов `resolve / reject` сразу

Обычно исполнитель делает что-то асинхронное и после этого вызывает `resolve / reject`, то есть через какое-то время. Но это не обязательно, `resolve` или `reject` могут быть вызваны сразу:

```
1 let promise = new Promise(function(resolve, reject)
2   // задача, не требующая времени
3   resolve(123); // мгновенно выдаст результат: 123
4 });
```

Это может случиться, например, когда мы начали выполнять какую-то задачу, но тут же увидели, что ранее её уже выполняли, и результат закеширован.

Такая ситуация нормальна. Мы сразу получим успешно завершённый `Promise`.

❗ Свойства `state` и `result` – внутренние

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.

Потребители: `then`, `catch`, `finally`

Объект `Promise` служит связующим звеном между исполнителем («создающим» кодом или «певцом») и функциями-потребителями («фанатами»), которые получают либо результат, либо ошибку. Функции-

потребители могут быть зарегистрированы (подписаны) с помощью методов `.then`, `.catch` и `.finally`.

then

Наиболее важный и фундаментальный метод – `.then`.

Синтаксис:

```
1 promise.then(  
2   function(result) { /* обработает успешное выполнение */  
3   function(error) { /* обработает ошибку */ }  
4 );
```

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:

```
1 let promise = new Promise(function(resolve, reject) {  
2   setTimeout(() => resolve("done!"), 1000);  
3 });  
4  
5 // resolve запустит первую функцию, переданную в .then  
6 promise.then(  
7   result => alert(result), // выведет "done!" через одну  
8   error => alert(error) // не будет запущена  
9 );
```

Выполнилась первая функция.

А в случае ошибки в промисе – выполнится вторая:

```
1 let promise = new Promise(function(resolve, reject) {  
2   setTimeout(() => reject(new Error("Whoops!")), 1000);  
3 });  
4  
5 // reject запустит вторую функцию, переданную в .then  
6 promise.then(  
7   result => alert(result), // не будет запущена  
8   error => alert(error) // выведет "Error: Whoops!" спустя  
9 );
```

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

```
1 let promise = new Promise(resolve => {  
2   setTimeout(() => resolve("done!"), 1000);  
3 });  
4  
5 promise.then(alert); // выведет "done!" спустя одну секунду
```

catch

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

```
1 let promise = new Promise((resolve, reject) => {  
2   setTimeout(() => reject(new Error("Ошибка!")), 1000);  
3 });
```

Раздел

[Промисы, async/await](#)

Навигация по уроку

Потребители: `then`, `catch`, `finally`

Пример: `loadScript`

Задачи (3)

Комментарии

Поделиться



[Редактировать на GitHub](#)

Раздел

[Промисы, async/await](#)

Навигация по уроку

Потребители: `then`, `catch`,
`finally`

Пример: `loadScript`

Задачи (3)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
4
5 // .catch(f) это тоже самое, что promise.then(null, f)
6 promise.catch(alert); // выведет "Error: Ошибка!" спуст.
```

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

finally

По аналогии с блоком `finally` из обычного `try {...} catch {...}, y` промисов также есть метод `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.

`finally` хорошо подходит для очистки, например остановки индикатора загрузки, его ведь нужно остановить вне зависимости от результата.

Например:

```
1 new Promise((resolve, reject) => {
2   /* сделать что-то, что займёт время, и после вызвать
3 })
4   // выполнится, когда промис завершится, независимо от
5   .finally(() => остановить индикатор загрузки)
6   .then(result => показать результат, err => показать о
```

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

1. 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.
2. Обработчик `finally` «пропускает» результат или ошибку дальше, к последующим обработчикам.

Например, здесь результат проходит через `finally` к `then`:

```
1 new Promise((resolve, reject) => {
2   setTimeout(() => resolve("result"), 2000)
3 })
4   .finally(() => alert("Промис завершён"))
5   .then(result => alert(result)); // <-- .then обрабо
```

А здесь ошибка из промиса проходит через `finally` к `catch`:

```
1 new Promise((resolve, reject) => {
2   throw new Error("error");
3 })
4   .finally(() => alert("Промис завершён"))
5   .catch(err => alert(err)); // <-- .catch обрабоае
```

Это очень удобно, потому что `finally` не предназначен для обработки результата промиса. Так что он просто пропускает его через себя дальше.

Мы более подробно поговорим о создании цепочек промисов и передаче результатов между обработчиками в следующей главе.

3. Последнее, но не менее значимое: вызов `.finally(f)` удобнее, чем `.then(f, f)` – не надо дублировать функции `f`.

Раздел

[Промисы, async/await](#)

Навигация по уроку

Потребители: then, catch, finally

Пример: loadScript

Задачи (3)

Комментарии

Поделиться



[Редактировать на GitHub](#)



На завершённых промисах обработчики запускаются сразу

Если промис в состоянии ожидания, обработчики в `.then/catch/finally` будут ждать его. Однако, если промис уже завершён, то обработчики выполнятся сразу:

```
1 // при создании промиса он сразу переводится в состояние 'fulfilled'
2 let promise = new Promise(resolve => resolve('готово!'));
3
4 promise.then(alert); // готово! (выведется сразу)
```

Теперь рассмотрим несколько практических примеров того, как промисы могут облегчить нам написание асинхронного кода.

Пример: loadScript

У нас есть функция `loadScript` для загрузки скрипта из предыдущей главы.

Давайте вспомним, как выглядел вариант с колбэками:

```
1 function loadScript(src, callback) {
2   let script = document.createElement('script');
3   script.src = src;
4
5   script.onload = () => callback(null, script);
6   script.onerror = () => callback(new Error('Ошибка загрузки скрипта'));
7
8   document.head.append(script);
9 }
```



Теперь перепишем её, используя `Promise`.

Новой функции `loadScript` более не нужен аргумент `callback`. Вместо этого она будет создавать и возвращать объект `Promise`, который перейдет в состояние «успешно завершён», когда загрузка закончится. Внешний код может добавлять обработчики («подписчиков»), используя `.then`:



```
1 function loadScript(src) {
2   return new Promise(function(resolve, reject) {
3     let script = document.createElement('script');
4     script.src = src;
5
6     script.onload = () => resolve(script);
7     script.onerror = () => reject(new Error('Ошибка загрузки скрипта'));
8
9     document.head.append(script);
10  });
11 }
```

Применение:

```
1 let promise = loadScript('https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js');
2
3 promise.then(
4   script => alert(`${script.src} загружен!`),
5   error => alert(`Ошибка: ${error.message}`)
6 );
7
8 promise.then(script => alert('Ещё один обработчик...'))
```

Сразу заметно несколько преимуществ перед подходом с использованием колбэков:

Раздел

[Промисы, async/await](#)

Навигация по уроку

Потребители: then, catch, finally

Пример: loadScript

Задачи (3)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Промисы

Промисы позволяют делать вещи в естественном порядке. Сперва мы запускаем `loadScript(script)`, и затем (`.then`) мы пишем, что делать с результатом.

Мы можем вызывать `.then` у `Promise` столько раз, сколько захотим. Каждый раз мы добавляем нового «фаната», новую функцию-подписчика в «список подписок». Больше об этом в следующей главе: [Цепочка промисов](#).

Колбэки

У нас должна быть функция `callback` на момент вызова `loadScript(script, callback)`. Другими словами, нам нужно знать что делать с результатом *до того*, как вызовется `loadScript`.

Колбэк может быть только один.

Таким образом, промисы позволяют улучшить порядок кода и дают нам гибкость. Но это далеко не всё. Мы узнаем ещё много полезного в последующих главах.

✓ Задачи

Можно ли "перевыполнить" промис? [↗](#)

Что выведет код ниже?

```
1 let promise = new Promise(function(resolve, reject) {
2   resolve(1);
3
4   setTimeout(() => resolve(2), 1000);
5 });
6
7 promise.then(alert);
```

решение

Задержка на промисах [↗](#)

Встроенная функция `setTimeout` использует колбэк-функции. Создайте альтернативу, использующую промисы.

Функция `delay(ms)` должна возвращать промис, который перейдёт в состояние «выполнен» через `ms` миллисекунд, так чтобы мы могли добавить к нему `.then`:

```
1 function delay(ms) {
2   // ваш код
3 }
4
5 delay(3000).then(() => alert('выполнилось через 3 секунды'));
```

решение

Анимация круга с помощью промиса [↗](#)

Перепишите функцию `showCircle`, написанную в задании [Анимация круга с помощью колбэка](#) таким образом, чтобы она возвращала промис, вместо того чтобы принимать в аргументы функцию-callback.

Новое использование:

```
1 showCircle(150, 150, 100).then(div => {
2   div.classList.add('message-ball');
3   div.append("Hello, world!");
4 });
```

Раздел

[Промисы, async/await](#)

Навигация по уроку

Потребители: then, catch, finally

Пример: loadScript

Задачи (3)

Комментарии

Поделиться



Редактировать на GitHub



Возьмите решение из [Анимация круга с помощью колбэка](#) в качестве основы.

решение

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



Комментарии

перед тем как писать...

© 2007—2020 Илья Кантор | [о проекте](#) | [связаться с нами](#) | [пользовательское соглашение](#) | [политика конфи](#)

