

Раздел

[Промисы, async/await](#)

Навигация по уроку

Возвращаем промисы

Пример: loadScript


Более сложный пример:
fetch

Итого

Задачи (1)

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠 → Язык программирования JavaScript](#)
[→ Промисы, async/await](#) 30-го ноября 2019

Цепочка промисов

Давайте вернёмся к ситуации из главы [Введение: колбэки](#): у нас есть последовательность асинхронных задач, которые должны быть выполнены одна за другой. Например, речь может идти о загрузке скриптов. Как же грамотно реализовать это в коде?

Промисы предоставляют несколько способов решения подобной задачи.

В этой главе мы разберём цепочку промисов.

Она выглядит вот так:

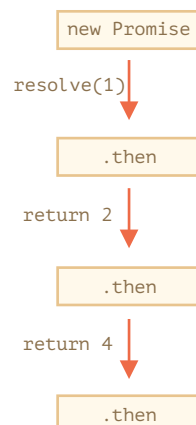
```
1 new Promise(function(resolve, reject) {
2
3   setTimeout(() => resolve(1), 1000); // (*)
4
5 }).then(function(result) { // (**)
6
7   alert(result); // 1
8   return result * 2;
9
10 }).then(function(result) { // (***)
11
12   alert(result); // 2
13   return result * 2;
14
15 }).then(function(result) {
16
17   alert(result); // 4
18   return result * 2;
19
20 });
```

Идея состоит в том, что результат первого промиса передаётся по цепочке обработчиков `.then`.

Поток выполнения такой:

1. Начальный промис успешно выполняется через 1 секунду (`*`),
2. Затем вызывается обработчик в `.then` (`**`).
3. Возвращаемое им значение передаётся дальше в следующий обработчик `.then` (`***`)
4. ...и так далее.

В итоге результат передаётся по цепочке обработчиков, и мы видим несколько `alert` подряд, которые выводят: `1 → 2 → 4`.



Раздел

[Промисы, async/await](#)

Навигация по уроку

Возвращаем промисы

Пример: loadScript

Более сложный пример:
fetch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Всё это работает, потому что вызов `promise.then` тоже возвращает промис, так что мы можем вызвать на нём следующий `.then`.

Когда обработчик возвращает какое-то значение, то оно становится результатом выполнения соответствующего промиса и передаётся в следующий `.then`.

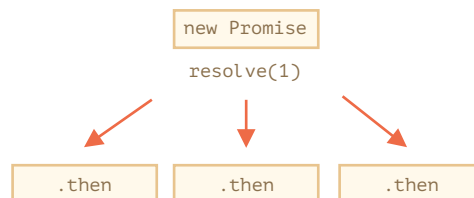
Классическая ошибка новичков: технически возможно добавить много обработчиков `.then` к единственному промису. Но это не цепочка.

Например:

```
1 let promise = new Promise(function(resolve, reject) {
2   setTimeout(() => resolve(1), 1000);
3 });
4
5 promise.then(function(result) {
6   alert(result); // 1
7   return result * 2;
8 });
9
10 promise.then(function(result) {
11   alert(result); // 1
12   return result * 2;
13 });
14
15 promise.then(function(result) {
16   alert(result); // 1
17   return result * 2;
18 });
```

Мы добавили несколько обработчиков к одному промису. Они не передают друг другу результаты своего выполнения, а действуют независимо.

Вот картина происходящего (сравните это с изображением цепочки промисов выше):



Все обработчики `.then` на одном и том же промисе получают одно и то же значение – результат выполнения того же самого промиса. Таким образом, в коде выше все `alert` показывают одно и то же: `1`.

На практике весьма редко требуется назначать несколько обработчиков одному промису. А вот цепочка промисов используется куда чаще.

Возвращаем промисы

Обработчик `handler`, переданный в `.then(handler)`, может вернуть промис.

В этом случае дальнейшие обработчики ожидают, пока он выполнится, и затем получают его результат.

Например:

```
1 new Promise(function(resolve, reject) {
2
3   setTimeout(() => resolve(1), 1000);
4
5 }).then(function(result) {
6
7   alert(result); // 1
8
9
10
```

Раздел

[Промисы, async/await](#)

Навигация по уроку

Возвращаем промисы

Пример: loadScript

Более сложный пример:
fetch

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
11   return new Promise((resolve, reject) => { // (*)
12     setTimeout(() => resolve(result * 2), 1000);
13   });
14
15 }).then(function(result) { // (**)
16
17   alert(result); // 2
18
19   return new Promise((resolve, reject) => {
20     setTimeout(() => resolve(result * 2), 1000);
21   });
22
23 }).then(function(result) {
24
25   alert(result); // 4
26
27   });
```

Здесь первый `.then` показывает 1 и возвращает новый промис `new Promise(...)` в строке `(*)`. Через одну секунду этот промис успешно выполняется, и его результат (аргумент в `resolve`, то есть `result * 2`) передаётся обработчику в следующем `.then`. Он находится в строке `(**)`, показывает 2 и делает то же самое.

Таким образом, как и в предыдущем примере, выводятся $1 \rightarrow 2 \rightarrow 4$, но сейчас между вызовами `alert` существует пауза в 1 секунду.

Возвращая промисы, мы можем строить цепочки из асинхронных действий.

Пример: loadScript

Давайте используем эту возможность вместе с промисифицированной функцией `loadScript`, созданной нами в [предыдущей главе](#), чтобы загружать скрипты по очереди, последовательно:



```
1 loadScript("/article/promise-chaining/one.js")
2   .then(function(script) {
3     return loadScript("/article/promise-chaining/two.js")
4   })
5   .then(function(script) {
6     return loadScript("/article/promise-chaining/three.js")
7   })
8   .then(function(script) {
9     // вызовем функции, объявленные в загружаемых скриптах
10    // чтобы показать, что они действительно загрузились
11    one();
12    two();
13    three();
14  });
```

Этот же код можно переписать немного компактнее, используя стрелочные функции:

```
1 loadScript("/article/promise-chaining/one.js")
2   .then(script => loadScript("/article/promise-chaining/two.js"))
3   .then(script => loadScript("/article/promise-chaining/three.js"))
4   .then(script => {
5     // скрипты загружены, мы можем использовать объявленные в них функции
6     one();
7     two();
8     three();
9   });
```

Здесь каждый вызов `loadScript` возвращает промис, и следующий обработчик в `.then` срабатывает, только когда этот промис завершается. Затем инициируется загрузка следующего скрипта и так далее. Таким образом, скрипты загружаются один за другим.

Раздел

[Промисы, async/await](#)

Навигация по уроку

Возвращаем промисы

Пример: `loadScript`

Более сложный пример:
`fetch`

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Мы можем добавить и другие асинхронные действия в цепочку. Обратите внимание, что наш код всё ещё «плоский», он «растёт» вниз, а не вправо. Нет никаких признаков «адской пирамиды вызовов».

Технически мы бы могли добавлять `.then` напрямую к каждому вызову `loadScript`, вот так:

```
1 loadScript("/article/promise-chaining/one.js").then(scr
2   loadScript("/article/promise-chaining/two.js").then(s
3   loadScript("/article/promise-chaining/three.js").th
4   // эта функция имеет доступ к переменным script1,
5   one();
6   two();
7   three();
8   });
9 });
10 });
```

Этот код делает то же самое: последовательно загружает 3 скрипта. Но он «растёт вправо», так что возникает такая же проблема, как и с колбэками.

Разработчики, которые не так давно начали использовать промисы, иногда не знают про цепочки и пишут код именно так, как показано выше. В целом, использование цепочек промисов предпочтительнее.

Иногда всё же приемлемо добавлять `.then` напрямую, чтобы вложенная в него функция имела доступ к внешней области видимости. В примере выше самая глубоко вложенная функция обратного вызова имеет доступ ко всем переменным `script1`, `script2`, `script3`. Но это скорее исключение, чем правило.



Раздел

[Промисы, async/await](#)

Навигация по уроку

Возвращаем промисы

Пример: loadScript

Более сложный пример:
fetch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Thenable

Если быть более точными, обработчик может возвращать не именно промис, а любой объект, содержащий метод `.then`, такие объекты называют «thenable», и этот объект будет обработан как промис.

Смысл в том, что сторонние библиотеки могут создавать свои собственные совместимые с промисами объекты. Они могут иметь свои наборы методов и при этом быть совместимыми со встроенными промисами, так как реализуют метод `.then`.

Вот пример такого объекта:

```
1 class Thenable {
2   constructor(num) {
3     this.num = num;
4   }
5   then(resolve, reject) {
6     alert(resolve); // function() { native code }
7     // будет успешно выполнено с аргументом this.r
8     setTimeout(() => resolve(this.num * 2), 1000);
9   }
10 }
11
12 new Promise(resolve => resolve(1))
13   .then(result => {
14     return new Thenable(result); // (*)
15   })
16   .then(alert); // показывает 2 через 1000мс
```

JavaScript проверяет объект, возвращаемый из обработчика `.then` в строке `(*)`: если у него имеется метод `then`, который можно вызвать, то этот метод вызывается, и в него передаются как аргументы встроенные функции `resolve` и `reject`, вызов одной из которых потом ожидается. В примере выше происходит вызов `resolve(2)` через 1 секунду `(**)`. Затем результат передаётся дальше по цепочке.

Это позволяет добавлять в цепочки промисов пользовательские объекты, не заставляя их наследовать от `Promise`.

Более сложный пример: fetch

Во фронтенд-разработке промисы часто используются, чтобы делать запросы по сети. Давайте рассмотрим один такой пример.

Мы будем использовать метод `fetch`, чтобы подгрузить информацию о пользователях с удалённого сервера. Этот метод имеет много опциональных параметров, разобранных в [соответствующих разделах](#), но базовый синтаксис весьма прост:

```
1 let promise = fetch(url);
```

Этот код запрашивает по сети `url` и возвращает промис. Промис успешно выполняется и в свою очередь возвращает объект `response` после того, как удалённый сервер присылает заголовки ответа, но до того, как весь ответ сервера полностью загружен.

Чтобы прочитать полный ответ, надо вызвать метод `response.text()`: он тоже возвращает промис, который выполняется, когда данные полностью загружены с удалённого сервера, и возвращает эти данные.

Код ниже запрашивает файл `user.json` и загружает его содержимое с сервера:

```
1 fetch('/article/promise-chaining/user.json')
2   // .then в коде ниже выполняется, когда удалённый сер
```

Раздел

[Промисы, async/await](#)

Навигация по уроку

Возвращаем промисы

Пример: loadScript

Более сложный пример:
fetch

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
3 .then(function(response) {
4   // response.text() возвращает новый промис,
5   // который выполняется и возвращает полный ответ се
6   // когда он загрузится
7   return response.text();
8 })
9 .then(function(text) {
10  // ...и здесь содержимое полученного файла
11  alert(text); // {"name": "iliakan", isAdmin: true}
12 });
```

Есть также метод `response.json()`, который читает данные в формате JSON. Он больше подходит для нашего примера, так что давайте использовать его.

Мы также применим стрелочные функции для более компактной записи кода:

```
1 // то же самое, что и раньше, только теперь response.js
2 fetch('/article/promise-chaining/user.json')
3   .then(response => response.json())
4   .then(user => alert(user.name)); // iliakan, получили
```

Теперь давайте что-нибудь сделаем с полученными данными о пользователе.

Например, мы можем послать запрос на GitHub, чтобы загрузить данные из профиля пользователя и показать его аватар:

```
1 // Запрашиваем user.json
2 fetch('/article/promise-chaining/user.json')
3   // Загружаем данные в формате json
4   .then(response => response.json())
5   // Делаем запрос к GitHub
6   .then(user => fetch(`https://api.github.com/users/${u
7   // Загружаем ответ в формате json
8   .then(response => response.json())
9   // Показываем аватар (githubUser.avatar_url) в течени
10  .then(githubUser => {
11    let img = document.createElement('img');
12    img.src = githubUser.avatar_url;
13    img.className = "promise-avatar-example";
14    document.body.append(img);
15
16    setTimeout(() => img.remove(), 3000); // (*)
17  });
```

Код работает, детали реализации отражены в комментариях. Однако в нём есть одна потенциальная проблема, с которой часто сталкиваются новички.

Посмотрите на строку `(*)`: как мы можем предпринять какие-то действия после того, как аватар был показан и удалён? Например, мы бы хотели показывать форму редактирования пользователя или что-то ещё. Сейчас это невозможно.

Чтобы сделать наш код расширяемым, нам нужно возвращать ещё один промис, который выполняется после того, как завершается показ аватара.

Примерно так:

```
1 fetch('/article/promise-chaining/user.json')
2   .then(response => response.json())
3   .then(user => fetch(`https://api.github.com/users/${u
4   .then(response => response.json())
5   .then(githubUser => new Promise(function(resolve, rej
6     let img = document.createElement('img');
7     img.src = githubUser.avatar_url;
8     img.className = "promise-avatar-example";
```

Раздел

[Промисы, async/await](#)

Навигация по уроку

Возвращаем промисы

Пример: loadScript

Более сложный пример:
fetch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



```
9     document.body.append(img);
10
11     setTimeout(() => {
12         img.remove();
13         resolve(githubUser); // (*)
14     }, 3000);
15 })
16 // срабатывает через 3 секунды
17 .then(githubUser => alert(`Закончили показ ${githubUser}`));
```

То есть, обработчик `.then` в строке (*) будет возвращать `new Promise`, который перейдёт в состояние «выполнен» только после того, как в `setTimeout` (**) будет вызвана `resolve(githubUser)`.

Соответственно, следующий по цепочке `.then` будет ждать этого.

Как правило, все асинхронные действия должны возвращать промис.

Это позволяет планировать после него какие-то дополнительные действия. Даже если эта возможность не нужна прямо сейчас, она может понадобиться в будущем.

И, наконец, давайте разобьём написанный код на отдельные функции, пригодные для повторного использования:

```
1 function loadJson(url) {
2     return fetch(url)
3         .then(response => response.json());
4 }
5
6 function loadGithubUser(name) {
7     return fetch(`https://api.github.com/users/${name}`)
8         .then(response => response.json());
9 }
10
11 function showAvatar(githubUser) {
12     return new Promise(function(resolve, reject) {
13         let img = document.createElement('img');
14         img.src = githubUser.avatar_url;
15         img.className = "promise-avatar-example";
16         document.body.append(img);
17
18         setTimeout(() => {
19             img.remove();
20             resolve(githubUser);
21         }, 3000);
22     });
23 }
24
25 // Используем их:
26 loadJson('/article/promise-chaining/user.json')
27     .then(user => loadGithubUser(user.name))
28     .then(showAvatar)
29     .then(githubUser => alert(`Показ аватара ${githubUser}`))
30     // ...
```

Итого

Если обработчик в `.then` (или в `catch/finally`, без разницы) возвращает промис, последующие элементы цепочки ждут, пока этот промис выполнится. Когда это происходит, результат его выполнения (или ошибка) передаётся дальше.

Вот полная картина происходящего:

вызов `.then(handler)` всегда возвращает промис:

```
state: "pending"
result: undefined
```

если `handler` заканчивается...

Раздел

[Промисы, async/await](#)

Навигация по уроку

Возвращаем промисы

Пример: loadScript

Более сложный пример:
fetch

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



возвратом значения

ошибкой

возвратом промиса

state: "fulfilled"
result: value

этот промис завершается с:

state: "rejected"
result: error



...с результатом
нового промиса...

✓ Задачи

Промисы: сравните then и catch [↗](#)

Являются ли фрагменты кода ниже эквивалентными? Другими словами, ведут ли они себя одинаково во всех обстоятельствах, для всех переданных им обработчиков?

```
1 promise.then(f1).catch(f2);
```

Против:

```
1 promise.then(f1, f2);
```

решение

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



💬 Комментарии

перед тем как писать...

