7.2. Цепочки промисов

~ 10 минут

Мы разобрались, как создавать промисы. Всё, что для этого требуется, — создать экземпляр объекта `Promise`, передать ему функцию-исполнитель, а в ней описать асинхронную операцию. Следующим шагом разберёмся, как использовать объект промис и получить результат выполнения асинхронной операции. Вернёмся к примеру с приготовлением супа и рассмотрим вот этот кусочек кода:

```
const TIMEOUT = 1000;

const buyProducts = () => {
  console.log(`> Иду за продуктами...`);
  return new Promise((resolve) => {
    setTimeout(() => {
      const products = [`Картофель`, `Капуста`, `Мясо`];
      resolve(products);
    }, TIMEOUT);
  });
};
```

Про остальные функции пока не будем думать, чтобы не загружать голову лишними деталями. `buyProducts` вполне самостоятельна, и нам её достаточно для экспериментов. Задача функции `buyProducts` — приобрести продукты для приготовления супа.

Поскольку операция асинхронная, мы воспользовались промисами. Отчётливо видно, что результатом выполнения функции `buyProducts` станет новый экземпляр объекта `Promise`. Если посмотреть на функцию-исполнитель (напомним, мы передаём её при создании объекта `Promise`), то видно, что список продуктов будет передан в качестве параметра при вызове колбэка `resolve`. Запомните это.

Обратите также внимание, что при описании функции-исполнителя мы не описали параметр `reject`. Это не является ошибкой. Просто для задачи «купить продукты»

не предусмотрен негативный сценарий.

Теперь попробуем получить результат выполнения асинхронной операции. Для этого мы воспользуемся методом `then` у экземпляра объекта `Promise`. Метод позволяет определить функцию, которая выполняется при переходе промиса в состояние `fulfilled` (выполнен успешно). Именно с этой функцией мы работаем внутри функции-исполнителя через параметр `resolve`. Давайте посмотрим, как это выглядит в коде:

```
// Функция `buyProducts` вернёт новый Promise
const buyProductPromise = buyProducts();

// Передаём функцию для `resolve`
buyProductPromise.then((products) => console.log(`Купили продукты:
${products.join(`, `)}`));
```

Первым делом мы получаем результат выполнения `buyProducts`. Она вернёт новый промис. Мы сохраним его в переменную `buyProductPromise`. На самом деле, отдельную переменную создавать необязательно. Можно сразу обращаться к методу `then`: `buyProducts().then(...)`. Мы делаем это для повышения наглядности, чтобы зафиксировать, что результатом функции будет экземпляр объекта `Promise`.

Затем применяем метод `then`. Это своего рода мостик к функции-исполнителю, которую мы определили при создании экземпляра объекта `Promise`. Метод `then` в первом параметре принимает функцию, которая должна быть выполнена при переходе промиса в состояние `fulfilled`. То есть эта функция станет значением колбэка `resolve`. Говоря другими словами, в первый параметр `then` мы передаём функцию для параметра `resolve`. Остановитесь на этом месте и прочтите абзац ещё раз.

Никакой магии: мы определяем колбэк, который будет вызван после завершения асинхронной операции. При определении колбэка мы предусматриваем параметр `products`. Через него мы получим список приобретённых продуктов. Как они в него попадут? Ответ кроется в месте вызова колбэка `resolve`. Обратите внимание, при вызове `resolve` мы передаём в качестве параметра данные о товарах.

Обработка ошибок

Интерфейс функции-исполнителя предусматривает два параметра: `resolve` и `reject`. Давайте посмотрим, как можно обработать ситуацию, когда в асинхронной операции возникла какая-то ошибка и промис перешёл в состояние `rejected`, то есть был вызван колбэк `reject`.

Решить эту задачу можно двумя способами. Первый вам уже знаком: воспользова
методом `then`. В первом параметре он принимает колбэк, который будет вызван

в случае перехода промиса в `fulfilled`. А во втором, колбэк, который будет вызван при переходе в `reject`. Немного модифицируем функцию `buyProducts` — добавим поддержку негативного сценария.

```
const TIMEOUT = 1000;

const buyProducts = () => {
  console.log(`> Иду за продуктами...`);
  return new Promise((resolve, reject) => {
    setTimeout(() => {

      const products = [`Картофель`, `Капуста`, `Мясо`];
      return Math.random() > 0.5
        ? resolve(products)
        : reject(`> Упс! В магазине нет нужных товаров`);
    }, TIMEOUT);
  });
};

buyProducts().then(
    (products) => console.log(products),
    (error) => console.log(error)
);
```

Внутри функции-исполнителя мы добавили дополнительное условие: в зависимости от случайного значения будем вызывать либо `resolve()`, либо `reject()`. Обратите внимание на описание метода `then`. В первом параметре передаём функцию для `resolve`, а во втором для `reject`. Принцип определения такой функции точно такой же. Попробуй несколько раз выполнить этот код. В одних случаях результатом будет вызов `resolve`, а в другом `reject`.

catch

Второй способ обработки перехода промиса в состояние `rejected` заключается в использовании метода `catch` у промиса. В качестве параметра ему необходимо передать колбэк для `reject`. Рассмотрим на примере:

```
// Код функции `buyProducts` остался без изменений
buyProducts()
    .then((products) => console.log(products))
    .catch((error) => console.log(error));
```

В этом примере использования промиса, определение функции для `reject` переехало из второго параметра `then` в `catch`. Остальной код остался прежним.

А какой способ выбрать? Чаще всего удобней использовать `catch`, чтобы не загромождать метод `then`. Вторая причина выбора в пользу `catch`: обработка

ошибок в цепочках промисов.

Цепочки промисов

Мы рассмотрели общий принцип работы с промисами и теперь готовы познакомиться с важной функцией: построение цепочек. Начнём с основ. В прошлом разделе мы воспользовались методом `then` и `catch`, но не обсудили одну деталь: а что является результатом вызова этих методов?

Методы `then` и `catch` всегда возвращают новый промис. За счёт этого мы можем строить цепочки вызовов промисов. Собственно немного с этим мы уже познакомились, когда применяли метод `catch`. Это стало возможным благодаря тому, что `then` вернул новый промис.

А зачем нам может пригодиться эта возможность? Для последовательного выполнения операций. Вспомните пример с приготовлением супа. Он включает несколько действий (купить продукты, подготовить их и так далее). Каждая следующая операция должна происходить после выполнения предыдущей. Вот тут нам и пригождаются цепочки:

```
buyProducts()
   // Мы можем описывать функции прямо на месте
   // или указать ссылку на существующую функцию.
   // Помните, это обычный колбэк
  .then(prepareProducts)
  .then(makeSoup)
  // Для последнего действия готовой функции нет,
  // поэтому описываем её прям здесь.
  .then((result) => console.log(result))
  .catch((error) => console.log(error));
```

Давайте проговорим, как выполняется приведённый фрагмент кода. Метод `then` вернёт новый промис. Мы вызываем `then` и определяем колбэк для `resolve`. Затем повторяем этот алгоритм для всех остальных действий. Для последней задачи (результат приготовления супа) у нас нет отдельной функции, поэтому мы описываем её на месте в виде анонимной функции.

При построении цепочек из промисов мы получаем ещё одну крутую возможность — передача результата выполнения промиса вниз по цепочке. Это означает, что в каждом последующем `then` мы можем получить результат выполнения колбэка предыдущего `then`. Посмотрим на примере:

```
new Promise((resolve) => resolve(1))
    .then((result) => result + 1) // 2
    .then((result) => result + 1) // 3
    .then((result) => console.log(result)); // 3
```

Каждый колбэк, который мы описываем для `then` возвращает результат увеличения значения, полученного в виде параметра на единицу. Этот результат передаётся вниз по цепочке. Таким образом, в конце мы получаем `3`. Эту возможность можно сравнить с конвейерной лентой. На каждом шаге появляется возможность улучшить предыдущий результат.

Наверняка вас интересует вопрос применения цепочек из промисов на практике. Подход применяется довольно часто. Возьмём задачу получения данных от сервера. Сначала мы должны их получить (например, в виде JSON строки), затем их десериализовать (распарсить), потом, возможно, нужно что-то в них добавить или преобразовать к другому формату и так далее. Это прекрасная задачка для применения цепочек.

Обратите внимание на применение метода `catch`. Мы описываем его один раз в конце цепочки промисов. Как в этом случае будет происходить обработка ошибки? Если любой из вышестоящих промисов в цепочке будет отклонён (`rejected`), то мы попадём в `catch` и сможем обработать эту ситуацию. Стоит отметить, что выполнение цепочки на этом прекратится. Если ошибка произошла в первом `then`, то последующие выполнятся не будут, инициативу перехватит `catch`.

Никто не запрещает нам описывать отдельный колбэк для обработки `rejected`. Мы знаем, что это можно сделать во втором параметре `then` или расставив дополнительные `catch`, но такая необходимость возникает не так часто. Возьмём пример с получением данных от сервера. Если мы не смогли получить данные, то смысла в выполнении операций, которые их используют — попросту нет.

Резюме

Мы познакомились с промисами и узнали, как их применять на практике. Всё, что для этого требуется — создать экземпляр объекта `Promise`. При использовании промисов стоит запомнить несколько важных вещей. Функция-исполнитель выполняется сразу. Промис может находится в одном из трёх состояний. Нет возможности откатиться к предыдущему состоянию. Методы `then` и `catch` всегда возвращают новый промис. За счёт этого появляется возможность строить цепочки вызовов промисов.

Прочитали главу?

Нажмите кнопку «Готово», чтобы сохранить прогресс.

Готово