JavaScript - promise

Введение: колбэки
Промисы
Цепочка промисов
Промисы: обработка ошибок
Promise API

асинхронные функции

Эта функция загружает на страницу новый скрипт. Когда в тело документа добавится конструкция <script src="…">, браузер загрузит скрипт и выполнит его.

Такие функции называют «асинхронными», потому что действие (загрузка скрипта) будет завершено не сейчас, а потом.

Мы хотели бы использовать новый скрипт, как только он будет загружен. Скажем, он объявляет новую функцию, которую мы хотим выполнить.

Но если мы просто вызовем эту функцию после loadScript(…), у нас ничего не выйдет:

Действительно, ведь у браузера не было времени загрузить скрипт. Сейчас функция loadScript никак не позволяет отследить момент загрузки. Скрипт загружается, а потом выполняется. Но нам нужно точно знать, когда это произойдёт, чтобы использовать функции и переменные из этого скрипта.

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}

// загрузит и выполнит скрипт
loadScript('/my/script.js');

loadScript('/my/script.js');
// код, написанный после вызова функции loadScript,
// не будет дожидаться полной загрузки скрипта
// ...
```

callBack + асинхронные функции

Давайте передадим функцию callback вторым аргументом в loadScript, чтобы вызвать её, когда скрипт загрузится:

Теперь, если мы хотим вызвать функцию из скрипта, нужно делать это в колбэке:

Смысл такой: вторым аргументом передаётся функция (обычно анонимная), которая выполняется по завершении действия.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('/my/script.js', function() {
  // эта функция вызовется после того, как загрузится скрипт
  newFunction(); // теперь всё работает
  ...
});
```

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
  alert(`Здорово, скрипт ${script.src} загрузился`);
  alert( _ ); // функция, объявленная в загруженном скрипте
});
```

Колбэк в колбэке

Как нам загрузить два скрипта один за другим: сначала первый, а за ним второй?

Каждое новое действие мы вынуждены вызывать внутри колбэка. Этот вариант подойдёт, когда у нас одно-два действия, но для большего количества уже не удобно. Альтернативные подходы мы скоро разберём.

```
loadScript('/my/script.js', function(script) {

  alert(`Здорово, скрипт ${script.src} загрузился, загрузим ещё один`);

  loadScript('/my/script2.js', function(script) {
    alert(`Здорово, второй скрипт загрузился`);
  });

});

loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...и так далее, пока все скрипты не будут загружены
    });

  })

});
```

Перехват ошибок

Мы вызываем callback(null, script) в случае успешной загрузки и callback(error), если загрузить скрипт не удалось.

Опять же, подход, который мы использовали в loadScript, также распространён и называется «колбэк с первым аргументом-ошибкой» («error-first callback»).

Правила таковы:

- Первый аргумент функции callback зарезервирован для ошибки. В этом случае вызов выглядит вот так: callback(err).
- Второй и последующие аргументы — для результатов выполнения. В этом случае вызов выглядит вот так: callback(null, result1, result2…).

Одна и та же функция callback используется и для информирования об ошибке, и для передачи результатов.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Не удалось загрузить скрипт ${src}`));

  document.head.append(script);
}
```

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // обрабатываем ошибку
  } else {
    // скрипт успешно загружен
  }
});
```

Адская пирамида вызовов

- Мы загружаем 1.js. Продолжаем, если нет ошибок.
- Мы загружаем 2.js. Продолжаем, если нет ошибок.
- Мы загружаем 3.js. Продолжаем, если нет ошибок. И так далее (*).

Чем больше вложенных вызовов, тем наш код будет иметь всё большую вложенность, которую сложно поддерживать, особенно если вместо ... у нас код, содержащий другие цепочки вызовов, условия и т.д.

Иногда это называют «адом колбэков» или «адской пирамидой колбэков».

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...и так далее, пока все скрипты не будут загружены (*)
          }
        });

      }
    })
  }
});
```

Решение

Мы можем попытаться решить эту проблему, изолируя каждое действие в отдельную функцию, вот так:

Код абсолютно рабочий, но кажется разорванным на куски. Его трудно читать, вы наверняка заметили это. Приходится прыгать глазами между кусками кода, когда пытаешься его прочесть. Это неудобно, особенно, если читатель не знаком с кодом и не знает, что за чем следует.

Кроме того, все функции step* одноразовые, и созданы лишь только, чтобы избавиться от «адской пирамиды вызовов». Никто не будет их переиспользовать где-либо ещё. Таким образом, мы, кроме всего прочего, засоряем пространство имён.

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...и так далее, пока все скрипты не будут загружены (*)
  }
};
```

Промисы

Промисы (Promises) в JavaScript — это объекты, представляющие асинхронную операцию, которая может завершиться успешно или с ошибкой. Промисы позволяют обрабатывать результаты асинхронных операций более удобно и читаемо, чем с помощью колбэков.

История промисов в JavaScript

Ранняя версия: Асинхронное программирование в JavaScript изначально полагалось на колбэки. Это приводило к запутанному и трудночитаемому коду.

ES5 (2009): Появились сторонние библиотеки (например, jQuery Deferred) для работы с асинхронными операциями и управления колбэками.

ES6 (2015): В ECMAScript 2015 (ES6) был добавлен стандартный объект Promise, который определяет поведение промисов, включая методы .then(), .catch(), и .finally().

Современное использование: Промисы стали основой для других функциональностей, таких как async/await, которые были добавлены в ES2017 (ES8), обеспечивая еще более простой и интуитивно понятный способ работы с асинхронным кодом.

Зачем нужны промисы

Упрощение работы с асинхронным кодом: Промисы позволяют избежать "callback hell", обеспечивая более чистую и структурированную обработку результатов.

Цепочки обработки: Можно легко создавать цепочки асинхронных операций, где результат одной операции передается в следующую.

Управление ошибками: Промисы предоставляют единый способ обработки ошибок с помощью .catch(), что упрощает отладку.

Конкурентные операции: С помощью Promise.all() и Promise.race() можно управлять несколькими асинхронными операциями одновременно.

Синтаксис Pomise

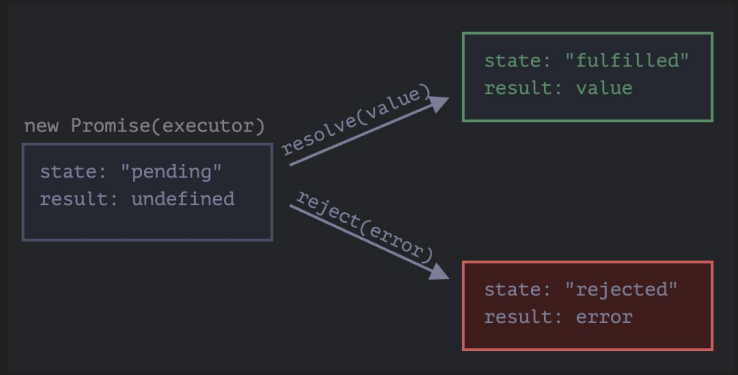
Функция, переданная в конструкцию new Promise, называется исполнитель (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат. В терминах нашей аналогии: исполнитель – это «певец».

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

Ее 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).

then

Метод then в промисах является ключевым компонентом для работы с результатами асинхронных операций. Он используется для определения, что должно произойти, когда промис будет выполнен (fulfilled) или отклонен (rejected).

promise.then(onFulfilled, onRejected);

- onFulfilled (необязательный): Функция, которая будет вызвана, когда промис выполнен. Она принимает один аргумент — значение, с которым был выполнен промис.
- onRejected (необязательный): Функция, которая будет вызвана, когда промис отклонен. Она принимает один аргумент — причину отклонения.

```
const myPromise = new Promise((resolve, reject) => {
    const success = true; // Пример условия

    if (success) {
        resolve('Успех!');
    } else {
        reject('Ошибка!');
    }
});

myPromise.then(
    result => {
        console.log(result); // 'Успех!'
    },
    error => {
        console.error(error); // 'Ошибка!'
    }
);
```

catch

Метод catch в промисах предназначен для обработки ошибок, возникающих в цепочке промисов. Он предоставляет удобный способ перехвата и обработки исключений или отклонений (rejections), происходящих в асинхронных операциях.

promise.catch(onRejected);

onRejected: Функция, которая будет вызвана, если промис будет отклонен. Она принимает один аргумент — причину отклонения (ошибку).

```
const myPromise = new Promise((resolve, reject) => {
    const success = false; // Пример условия

    if (success) {
        resolve('Успех!');
    } else {
        reject('Ошибка!');
    }
});

myPromise.catch(error => {
    console.error(error); // 'Ошибка!'
});
```

finally

Метод finally в промисах используется для выполнения кода после завершения промиса, независимо от того, был ли он выполнен успешно или отклонен. Это позволяет вам выполнять операции, которые должны происходить в любом случае, например, освобождение ресурсов или выполнение завершающих действий.

promise.finally(onFinally);

onFinally: Функция, которая будет вызвана, когда промис завершится (независимо от того, выполнен он или отклонен). Она не принимает аргументов.

```
const myPromise = new Promise((resolve, reject) => {
    const success = true; // Пример условия

    if (success) {
        resolve('Успех!');
    } else {
        reject('Ошибка!');
    }
});

myPromise
    .then(result => {
        console.log(result); // 'Успех!'
    })
    .catch(error => {
        console.error(error); // Этот код не выполнится, так как success = true
    })
    .finally(() => {
        console.log('Операция завершена.'); // 'Операция завершена.'
    });
```

Цепочка промисов

Цепочка промисов — это последовательность асинхронных операций, связанных между собой с помощью методов then, catch и finally. Каждый метод в цепочке возвращает новый промис, позволяя строить последовательные шаги обработки данных и управления ошибками.

Принцип работы цепочки промисов

- Промис выполняется или отклоняется: Когда промис завершается, вызывается соответствующий обработчик (onFulfilled или onRejected).
- Обработчики возвращают значения или промисы: Обработчики, переданные в then и catch, могут возвращать значения, которые становятся результатом следующего промиса в цепочке. Они также могут возвращать новые промисы, которые будут асинхронно обработаны.
- Новые промисы обрабатываются: Возвращенные значения или промисы обрабатываются в следующем then или catch.

```
const firstPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('Первый шаг завершен');
    }, 1000);
});

firstPromise
    .then(result => {
        console.log(result); // 'Первый шаг завершен'
        return 'Результат второго шага';
    })
    .then(result => {
        console.log(result); // 'Результат второго шага'
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve('Третий шаг завершен');
            }, 1000);
        });
    })
    .then(result => {
        console.log(result); // 'Третий шаг завершен'
    })
    .catch(error => {
        console.error('Ошибка:', error);
    })
    .finally(() => {
        console.log('Цепочка завершена');
    });
```

Promise API

Promise API — это набор методов и свойств, предоставляемых объектом Promise в JavaScript для работы с асинхронными операциями. Эти методы и свойства помогают создавать, управлять и координировать промисы. Вот основные компоненты Promise API:

- Promise.resolve(value)
- Promise.reject(reason)
- Promise.all(iterable)
- Promise.allSettled(iterable)
- Promise.race(iterable)
- Promise.any(iterable)

Promise.resolve(value)

Метод Promise.resolve(value) используется для создания промиса, который немедленно разрешается с указанным значением. Этот метод особенно полезен для преобразования различных значений в промисы и для создания быстроразрешаемых промисов в коде.

Promise.resolve(value);

value: Значение, с которым будет разрешен промис. Может быть любым, включая другие промисы или объекты, имеющие метод then.

Особенности

- Если value является промисом: Если value уже является промисом, Promise.resolve(value) вернет этот промис как есть.
- Если value имеет метод then (thenable): Если value — это объект или функция, у которых есть метод then, возвращаемый промис будет "следовать" за этим объектом или функцией, пытаясь разрешиться с их результатом.
- Если value не является промисом или thenable: Если value — это обычное значение (не промис и не thenable), возвращаемый промис немедленно разрешается с этим значением.

Promise.reject(reason)

Метод Promise.reject(reason) используется для создания промиса, который немедленно отклоняется с указанной причиной (ошибкой). Этот метод полезен для создания быстроотклоняемых промисов и для приведения различных ошибок к форме промисов.

Promise.reject(reason);

reason: Причина отклонения промиса. Может быть любым значением, но обычно это объект Error или строка, объясняющая причину отклонения.

Особенности

- Немедленное отклонение: Возвращаемый промис немедленно отклоняется с указанной причиной.
- Тип значения reason: reason может быть любым значением, которое лучше всего объясняет причину отклонения. Чаще всего используются объекты Error для более точного описания ошибок.

Promise.all(iterable)

Метод Promise.all(iterable) принимает итерируемый объект (например, массив) промисов и возвращает новый промис, который разрешается, когда все промисы в итерируемом объекте разрешаются, или отклоняется, если хотя бы один из промисов отклоняется.

Promise.all(iterable);

iterable: Итерируемый объект (например, массив), содержащий промисы или значения (которые будут автоматически преобразованы в разрешенные промисы).

Особенности

- Разрешается, если все промисы в итерируемом объекте разрешены. Возвращает массив значений всех разрешенных промисов, сохраняя порядок исходного итерируемого объекта.
- Отклоняется, если хотя бы один промис отклоняется. Возвращает причину отклонения первого отклоненного промиса.
- Если любой из значений в итерируемом объекте не является промисом, он автоматически преобразуется в разрешенный промис с этим значением.

Promise.allSettled(iterable)

Метод Promise.allSettled(iterable) принимает итерируемый объект (например, массив) промисов и возвращает новый промис, который разрешается, когда все промисы в итерируемом объекте завершены, независимо от того, были ли они разрешены или отклонены. Возвращаемый промис содержит массив объектов, каждый из которых описывает состояние одного из промисов.

Promise.allSettled(iterable);

iterable: Итерируемый объект (например, массив), содержащий промисы или значения.

Особенности

- Разрешается, когда все промисы из итерируемого объекта завершены (разрешены или отклонены).
- Возвращает массив объектов с состоянием каждого промиса.
- Каждый объект в возвращаемом массиве имеет два свойства:
- status: строка, указывающая состояние промиса ("fulfilled" или "rejected").
- value: значение, если промис был разрешен, или reason: причина отклонения, если промис был отклонен.

Promise.race(iterable)

Метод Promise.race(iterable) принимает итерируемый объект (например, массив) промисов и возвращает новый промис, который разрешается или отклоняется с результатом первого завершившегося промиса в итерируемом объекте. "Завершение" может означать либо разрешение, либо отклонение.

Promise.race(iterable);

iterable: Итерируемый объект (например, массив), содержащий промисы или значения.

Особенности

- Первый завершившийся промис: Возвращаемый промис будет разрешен или отклонен с результатом или причиной первого завершившегося промиса. Это может быть как выполненный, так и отклоненный промис.
- Если переданы не-промисы: Если в массиве есть значения, которые не являются промисами, они будут проигнорированы, если не завершатся первыми.

Promise.any(iterable)

Метод Promise.any(iterable) принимает итерируемый объект (например, массив) промисов и возвращает новый промис, который разрешается с результатом первого выполненного промиса. Если все промисы отклонены, возвращаемый промис отклоняется с массивом причин отклонения.

Promise.any(iterable);

iterable: Итерируемый объект (например, массив), содержащий промисы или значения.

Особенности

- Первый выполненный промис: Возвращаемый промис разрешается с результатом первого промиса, который успешно выполнен (разрешен).
- Отклонение при отсутствии выполненных промисов: Если все промисы отклонены, возвращается промис, отклоняющийся с массивом всех причин отклонения.
- Необязательные значения: Если переданные значения не являются промисами, они обрабатываются как выполненные и возвращаются немедленно.

простая реализация метода Promise.all на JavaScript

Проверка аргумента: Убедитесь, что переданный аргумент — это массив. Если нет, возвращаем отклоненный промис.

Инициализация: Создаем массив results для хранения результатов и переменную completed для отслеживания завершенных промисов.

Цикл по промисам: Для каждого промиса:

- Преобразуем его в промис с помощью Promise.resolve.
- Если промис разрешается, сохраняем результат в соответствующем индексе и увеличиваем счетчик.
- Если все промисы завершены, вызываем resolve с массивом результатов.
- Если любой промис отклоняется, отклоняем итоговый промис.

Эта реализация позволяет обрабатывать массив промисов, возвращая массив результатов после их разрешения.

```
function promiseAll(promises) {
    return new Promise((resolve, reject) => {
        // Проверяем, что аргумент является массивом
        if (!Array.isArray(promises)) {
            return reject(new TypeError('Аргумент должен быть массивом'));
        }

        const results = [];
        let completed = 0;

        // Обрабатываем каждый промис
        promises.forEach((promise, index) => {
            // Если элемент не является промисом, оборачиваем его в разрешенный промис
            Promise.resolve(promise)
                .then(value => {
                    results[index] = value; // Сохраняем результат по индексу
                    completed++; // Увеличиваем счетчик завершенных промисов

                    // Проверяем, завершены ли все промисы
                    if (completed === promises.length) {
                        resolve(results); // Разрешаем итоговый промис
                    }
                })
                .catch(reject); // Если любой промис отклонен, отклоняем итоговый промис
        });
    });
}
```

Своя реализация Promise

Объяснение кода

Конструктор: Принимает функцию executor, которая принимает два параметра: resolve и reject.

Состояния и значения:

- state может быть pending, fulfilled или rejected.
- value хранит результат выполнения, а reason — причину отклонения.

Метод then:

- Возвращает новый промис.
- Обрабатывает выполненные и отклоненные состояния, добавляя соответствующие колбэки.

Метод catch: Упрощает обработку ошибок, делая это через метод then.

```
const promise = new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve('Успех!');
    }, 1000);
});

promise
    .then(value => {
        console.log(value); // 'Успех!'
        return 'Дальнейшая обработка';
    })
    .then(value => {
        console.log(value); // 'Дальнейшая обработка'
    })
    .catch(error => {
        console.error(error); // Обработка ошибок
    });
```

```
class MyPromise {
    constructor(executor) {
        this.state = 'pending'; // начальное состояние
        this.value = undefined;  // значение, если промис выполнен
        this.reason = undefined; // причина отклонения, если промис отклонен
        this.onFulfilledCallbacks = []; // колбэки для выполнения
        this.onRejectedCallbacks = [];  // колбэки для отклонения

        const resolve = (value) => {
            if (this.state === 'pending') {
                this.state = 'fulfilled';
                this.value = value;
                this.onFulfilledCallbacks.forEach(callback => callback(value));
            }
        };

        const reject = (reason) => {
            if (this.state === 'pending') {
                this.state = 'rejected';
                this.reason = reason;
                this.onRejectedCallbacks.forEach(callback => callback(reason));
            }
        };

        try {
            executor(resolve, reject); // запускаем executor
        } catch (error) {
            reject(error); // ловим ошибки
        }
    }

    then(onFulfilled, onRejected) {
        return new MyPromise((resolve, reject) => {
            const handleFulfilled = () => {
                try {
                    const result = onFulfilled ? onFulfilled(this.value) : this.value;
                    resolve(result); // разрешаем с результатом
                } catch (error) {
                    reject(error); // отклоняем с ошибкой
                }
            };

            const handleRejected = () => {
                try {
                    const result = onRejected ? onRejected(this.reason) : this.reason;
                    resolve(result); // разрешаем с результатом
                } catch (error) {
                    reject(error); // отклоняем с ошибкой
                }
            };

            if (this.state === 'fulfilled') {
                handleFulfilled(); // если промис выполнен
            } else if (this.state === 'rejected') {
                handleRejected(); // если промис отклонен
            } else {
                // если состояние pending, добавляем колбэки
                this.onFulfilledCallbacks.push(handleFulfilled);
                this.onRejectedCallbacks.push(handleRejected);
            }
        });
    }

    catch(onRejected) {
        return this.then(null, onRejected); // делаем catch через then
    }
}
```

Ресурсы

Введение: колбэки - ТЫК

Промисы - ТЫК

Цепочка промисов - ТЫК

Промисы: обработка ошибок - ТЫК

Promise API - ТЫК

Реализация Promise на JS - ТЫК

Реализация Promise.all на JS - ТЫК