Разработка пользовательского веб-интерфейса

ES6, RAF, Обещания, fetch, async/await

Сергей Геннадьевич Синица

КубГУ, 2020

sin@kubsu.ru

Анимация с помощью RequestAnimationFrame.

Анимация, которую невозможно сделать с помощью CSS3, реализуется на JavaScript вычислением положения элементов для каждого кадра 60 раз в секунду.

Анимацию можно делать изменением DOM-свойств из JS по таймеру с помощью SetInterval раз в 17 мс.

Однако лучше делать это с помощью специального объекта RequestAnimationFrame, который оптимизирует вызовы функции отрисовки для получения 60 FPS, отключается на неактивных вкладках.

https://learn.javascript.ru/js-animation

https://developer.mozilla.org/ru/docs/DOM/window.requestAnimationFrame

Caniuse 97%

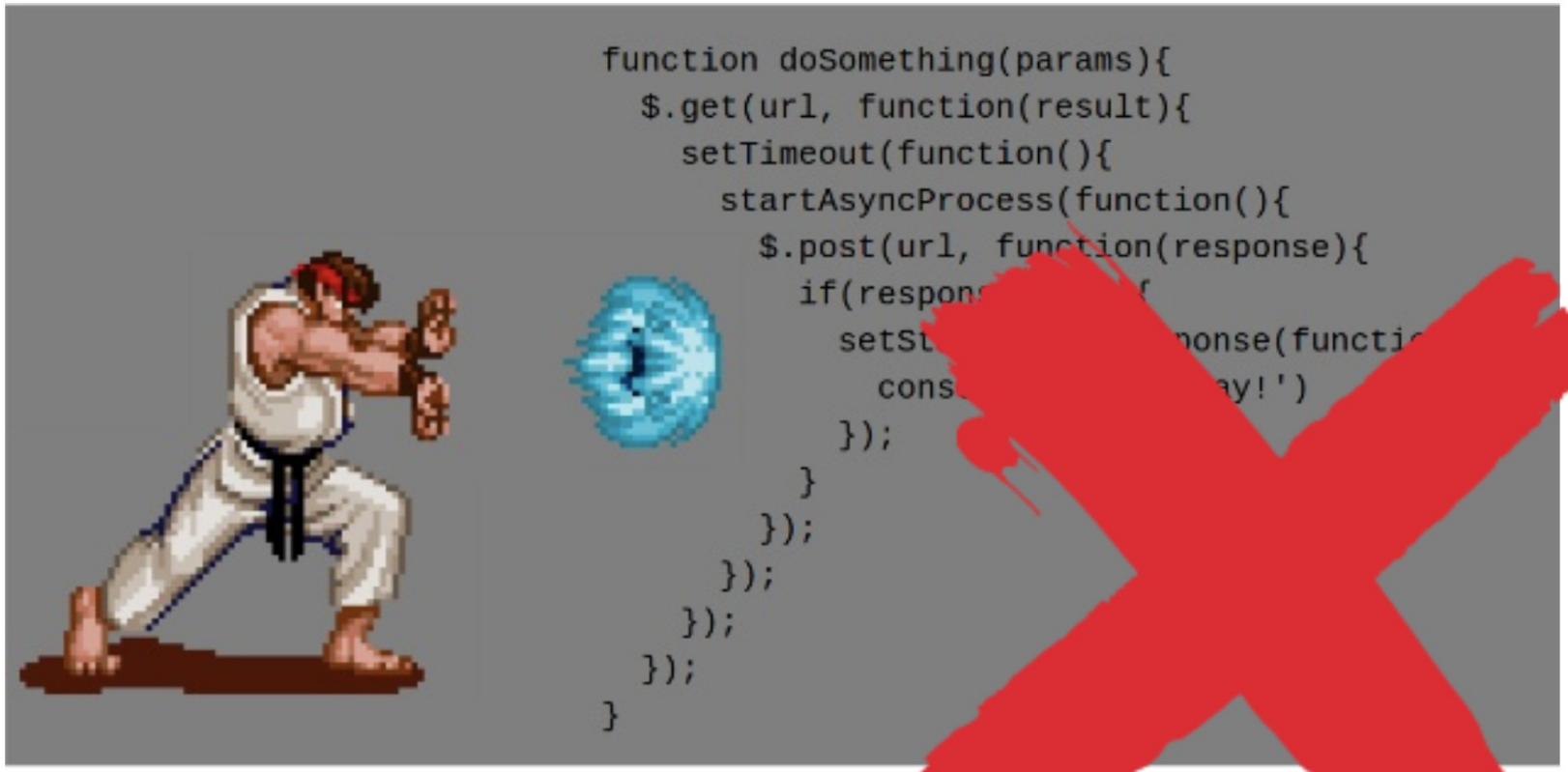
```
<div id="mybox" style="position:absolute; width: 100px;">Animation!</div>

<script>
// Переменная таймера начала анимации.
var start = null;
// Находим элемент для анимации.
var element = document.getElementById("mybox");

// Функция кадра горизонтальной анимации.
function step(timestamp) {
  // Вычисляем новую позицию.
  if (!start) start = timestamp;
  var progress = timestamp - start;
  // Сдвигаем элемент.
  element.style.left = Math.min(progress / 10, 200) + 'px';
  if (progress < 2000) {
    // Пока не прошло 2 секунды вызываем следующий кадр.
    window.requestAnimationFrame(step);
  }
}

// Ждем загрузки DOM.
window.addEventListener('DOMContentLoaded', function() {
  // Запускаем анимацию.
  window.requestAnimationFrame(step);
});
</script>
```

Анимация на JS

```
setTimeout(function() {
    window.requestAnimationFrame(step2(start, element));
    setTimeout(function() {
      window.requestAnimationFrame(step3(start, element));
    }, 1000);
  }, 1000);
```

```
runAnimation(0);
setTimeout(function() {
    runAnimation(1);
    setTimeout(function() {
        runAnimation(2);
        ...
    }, 1000);
}, 1000);
```

Ад обратных вызовов



```
function doSomething(params){
  $.get(url, function(result){
    setTimeout(function(){
      startAsyncProcess(function(){
        $.post(url, function(response){
          if(respon
            setSt
              cons
            });
          }
        });
      });
    });
  });
}
```

при работе с базой данных, при анимации,
при запросах на сервер, работе с воркерами

ES6 vs АД:
стрелки и обещания

ES6 стрелочные функции

```
// ES5
var add = function(a, b) {
  return a + b;
};

// ES6
var add = (a, b) => a + b;
```

ES6 стрелочные функции

```
// function expression: два способа в ES6
var f = function() { return 42; }
var f = () => 42;

// function declaration: один способ в ES6
function f() {
  return 42;
}
```

Rest … и arguments

```
// ES5

function printf(format) {
  var params = [].slice.call(arguments, 1);
  console.log('params: ', params);
  console.log('format: ', format);
}

printf('%s %d %.2f', 'adrian', 321, Math.PI);


// ES6

function printf(format, ...params) {
  console.log('params: ', params);
  console.log('format: ', format);
}

printf('%s %d %.2f', 'adrian', 321, Math.PI);
```

В JS нет полиморфных функций

```
// ES5
var sum = function() {
  return [].reduce.call(arguments, function(m, n) {
    return m + n;
  }, 0);
}

// ES6
var sum = (...args) => args.reduce((m, n) => m + n, 0);
```

Еще примеры

```
// В ES6 можно даже так:
var f = x => x * 2

// Но лучше писать скобки всегда:
var f = (x) => x * 2

// Без параметров:
var f = () => 42;
f(); // 42
```

Return работает

```
var g = (a, b) => {
  let m = Math.pow(2, parseInt(a));
  return b + m;
}
```

Анонимные стрелки, ФП в ES6!

```
const double = (arr) => arr.map( item => item * 2 );
double([1,2,3,4]); // [2, 4, 6, 8]

const filter = (arr) => arr.filter( item => !!item );
filter([false, undefined, 0, 1, 'str']); // [1, "str"]

const sum = (...args) => args.reduce((m, n) => m + n, 0);
sum(10, 15, 20); // 45
```

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/map
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

Упрощает работу с this!

```
// ES5
var obj = {
  btn: document.links[0],
  log: function (message) {
    console.log(message);
    return this;
  },
  init: function() {
    var self = this;

self.btn.addEventListener('click',
    function() {
      self.log('Button Clicked!');
    },
    false);
  }
};
```

```
// ES6
var obj = {
  btn: document.links[0],
  log: function(message) {
    console.log(message);
    return this;
  },
  init: function() {

this.btn.addEventListener('click',
() => this.log('Button Clicked!'),
false);
  }
};
```

Чтобы выполнить какой-либо метод из объекта до появления стрелочных функций, необходимо было воспользоваться одним из методов замены ключевого слова this: как в этом примере записать его в переменную self, или воспользоваться методом функций bind.

Что обещает Promise?

Объект Promise (промис, обещание) используется для отложенных и асинхронных вычислений.

Появился в ES6 (ECMAScript-2015).

Интерфейс Promise (промис) представляет собой обертку для значения, неизвестного на момент создания промиса.

Он позволяет обрабатывать результаты асинхронных операций так, как если бы они были синхронными: вместо конечного результата асинхронного метода возвращается обещание (промис) получить результат в некоторый момент в будущем.

https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects

A Promise is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation.

Any Promise object is in one of three mutually exclusive states: fulfilled, rejected, and pending:

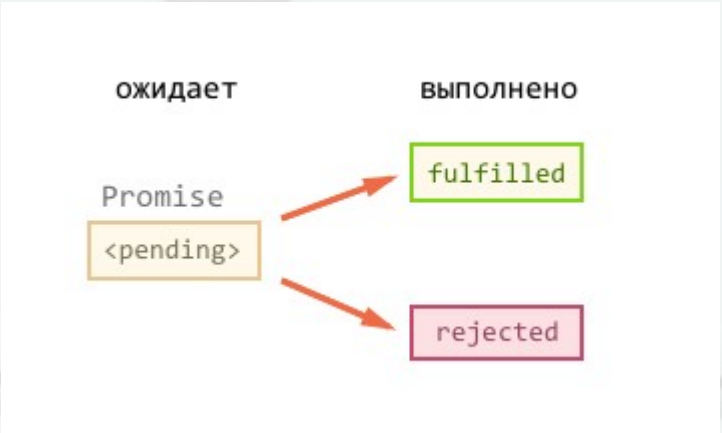
- A promise p is fulfilled if p.then(f, r) will immediately enqueue a Job to call the function f.

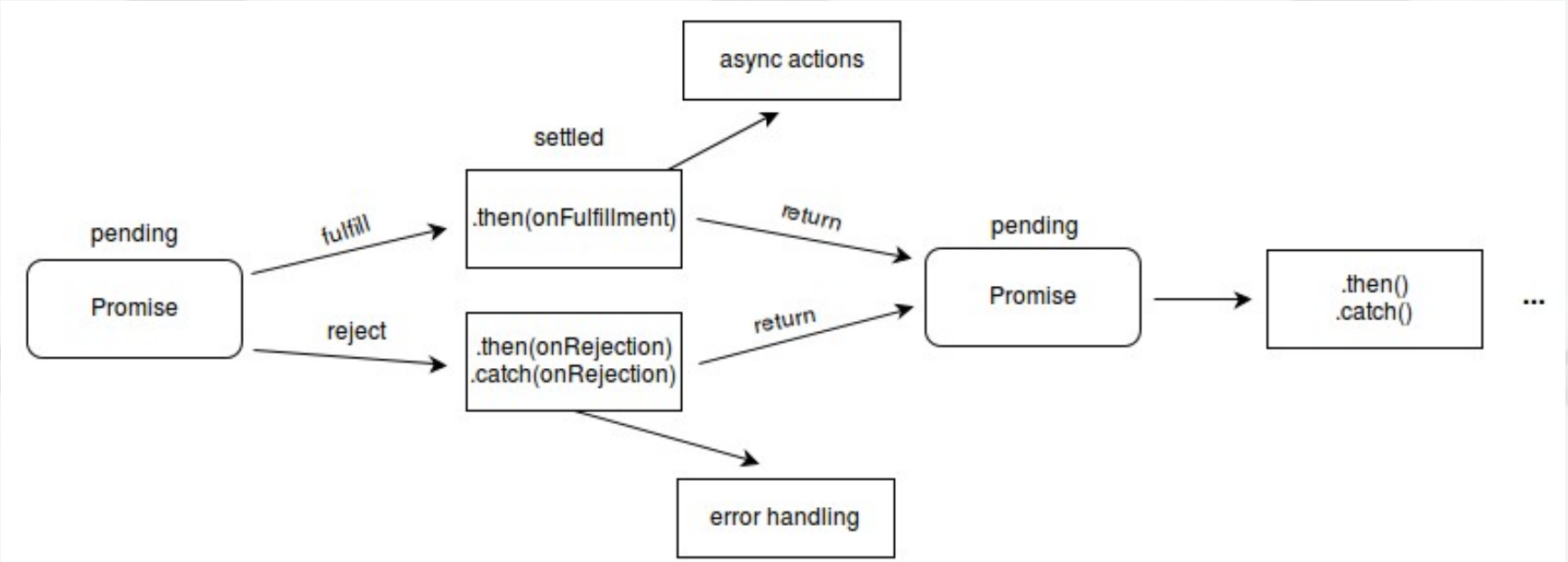
- A promise p is rejected if p.then(f, r) will immediately enqueue a Job to call the function r.

- A promise is pending if it is neither fulfilled nor rejected.

A promise is said to be settled if it is not pending, i.e. if it is either fulfilled or rejected.

A promise is resolved if it is settled or if it has been “locked in” to match the state of another promise. Attempting to resolve or reject a resolved promise has no effect. A promise is unresolved if it is not resolved. An unresolved promise is always in the pending state. A resolved promise may be pending, fulfilled or rejected.





https://learn.javascript.ru/promise
https://developer.mozilla.org/ru/docs/Web/JavaScript/reference/Global_Objects/Promise

Покажи уже примеры!

```
let myFirstPromise = new Promise((resolve, reject) => {
  // Мы вызываем resolve(...), когда асинхронная операция
  // завершилась успешно, и reject(...), когда она не удалась.
  // В этом примере мы используем setTimeout(...),
  // чтобы симулировать асинхронный код.
  // В реальности вы, скорее всего, будете использовать XHR,
  // HTML5 API или что-то подобное.
  setTimeout(function(){
    resolve("Success!"); // Ура! Всё прошло хорошо!
  }, 250);
});

myFirstPromise.then((successMessage) => {
  // successMessage - это что угодно,
  // что мы передали в функцию resolve(...) выше.
  // Это необязательно строка, но если это всего лишь сообщение
  // об успешном завершении, это наверняка будет она.
  console.log("Ура! " + successMessage);
});
```

```
function imgLoad(url) {
  // Create new promise with the Promise() constructor;
  // This has as its argument a function
  // with two parameters, resolve and reject
  return new Promise(function(resolve, reject) {
    // Standard XHR to load an image
    var request = new XMLHttpRequest();
    request.open('GET', url);
    request.responseType = 'blob';
    // When the request loads, check whether it was successful
    request.onload = function() {
      if (request.status === 200) {
        // If successful, resolve the promise by passing back the request response
        resolve(request.response);
      } else {
        // If it fails, reject the promise with a error message
        reject(Error('Image didn\'t load successfully; error code:' +
                request.statusText));
      }
    };
    request.onerror = function() {
        // Also deal with the case when the entire request fails to begin with
        // This is probably a network error,
        //so reject the promise with an appropriate message
        reject(Error('There was a network error.'));
    };
    // Send the request
    request.send();
  });
}
```

```
// Get a reference to the body element, and create a new image object
  var body = document.querySelector('body');
  var myImage = new Image();
  // Call the function with the URL we want to load, but then chain the
  // promise then() method on to the end of it. This contains two callbacks
  imgLoad('myLittleVader.jpg').then(function(response) {
    // The first runs when the promise resolves, with the request.response
    // specified within the resolve() method.
    var imageURL = window.URL.createObjectURL(response);
    myImage.src = imageURL;
    body.appendChild(myImage);
    // The second runs when the promise
    // is rejected, and logs the Error specified with the reject() method.
  }, function(Error) {
    console.log(Error);
  });
```

https://github.com/mdn/js-examples/blob/master/promises-test/index.html

Коллбеки VS промисы

1. Коллбэки являются функциями, обещания являются объектами

Коллбэки — это просто функции, которые выполняются в ответ на какое-либо событие, например, событие таймера или получение ответа от сервера. Любая функция может стать коллбэком, и любой коллбэк является функцией.

Обещания являются объектами, которые хранят информацию, произошли ли определенные события или нет, а если произошли — то и их результат.

Коллбеки VS промисы

2. Коллбэки передаются в качестве аргументов, обещания возвращаются

Коллбэки определяются независимо от вызывающей функции и передаются в качестве аргументов. Вызывающая функция сохраняет коллбэк и вызывает его, когда происходит определенное событие.

Обещания создаются внутри асинхронных функций и возвращаются. Когда происходит событие, асинхронная функция обновляет обещание, чтобы уведомить об этом внешний мир.

Коллбеки VS промисы

3. Коллбэки обрабатывают успешное или неуспешное завершение, обещания ничего не обрабатывают

Коллбэки, как правило, вызываются с информацией о том, успешно или неуспешно завершилась операция, и должны быть в состоянии обработать оба варианта.

Обещания ничего не обрабатывают по умолчанию, обработчики добавляются позже.

Коллбеки VS промисы

4. Коллбэки могут обрабатывать несколько событий, обещания связаны только с одним событием

Коллбэки можно вызывать несколько раз в функциях, в которые они переданы.

Обещания могут представлять только одно событие — они обратывают либо успешное его завершение, либо неуспешное только один раз.

Имея это в виду, давайте рассмотрим обещания более детально.

Выходим из ада коллбеков!

```
runAnimation(0);
setTimeout(function() {
    runAnimation(1);
    setTimeout(function() {
        runAnimation(2);
    }, 1000);
}, 1000);
```

1. new Promise(fn)

```
// Создание экземпляра обещания, который ничего не делает.
// Не волнуйтесь, мы рассмотрим этот момент подробнее.
promise = new Promise(function() {});

// Вызов new Promise немедленно вызовет функцию, переданную в качестве аргумента.
// Цель этой функции состоит в информировании объекта Promise, когда событие,
// с которым он связан, будет завершено.

// Для того, чтобы сделать это, функция, которую вы передаете в конструктор,
// может принимать два параметра, которые сами являются функциями — resolve и reject.

// Вызов resolve(value) пометит обещание как успешно завершенное
// и вызовет обработчик успешного завершения.

// Вызов reject(error) вызовет обработчик неуспешного завершения.

// Нельзя вызывать обе эти функции одновременно.

// Функции resolve и reject обе принимают один аргумент, который содержит в себе данные о событии.
```

1. new Promise(fn)

```
function delay(interval) {
    return new Promise(function(resolve) {
        setTimeout(resolve, interval);
    });
}

var oneSecondDelay = delay(1000);

/*
У нас есть обещание, которое резолвится через секунду.

Функция, которую мы передаем в new Promise в приведенном примере,
принимает только параметр resolve, мы опустили параметр reject. Это
потому, что setTimeout выполняется всегда и, таким образом, нет сценария,
где мы он мог бы завершить неуспешно.
*/
```

1. new Promise(fn)

```
function animationTimeout(step, interval) {
    new Promise(function(resolve, reject) {
        if (isAnimationSupported(step)) {
            setTimeout(resolve, interval);
        } else {
            reject('animation not supported');
        }
    });
}

var firstKeyframe = animationTimeout(1, 1000);
```

```
// Если в переданной функции возникнет исключение,
// обещание будет автоматически помечено как неуспешное.

var promise = new Promise(function(resolve, reject) {
    try {
        // your code
    }
    catch (e) {
        reject(e)
    }
});
```

2. promise.then(onResolve, onReject)

```
// Только обработчик успеха
promise.then(function(details) {
    // handle success
});

// Только обработчик отказа
promise.then(null, function(error) {
    // handle failure
});

// Обработчики успеха и отказа
promise.then(
    function(details) { /* handle success */ },
    function(error) { /* handle failure */ }
);
```

2. promise.then(onResolve, onReject)

```
// Это вызовет слезы и ужас
promise.then(
    function() {
        throw new Error('tears');
    },
    function(error) {
        // Не будет вызван
        console.log(error)
    }
);
```

2. promise.then(onResolve, onReject)

Обработчики, переданные в promise.then не просто обрабатывают результат предыдущего обещания — то, что они возвращают, передается в следующие обещание.

```
delay(1000)
    .then(function() {
        return 5;
    })
    .then(function(value) {
        console.log(value); // 5
    });
```

2. promise.then(onResolve, onReject)

Но что еще более важно, это работает с другими обещаниями — возвращение обещания из обработчика then передает это обещание в качестве возвращаемого значения then. Это позволяет реализовывать цепочки обещаний:

```
delay(1000)
    .then(function() {
        console.log('1 second elapsed');
        return delay(1000);
    })
    .then(function() {
        console.log('2 seconds elapsed');
    });
```

2. promise.then(onResolve, onReject)

```
runAnimation(0);
setTimeout(function() {
    runAnimation(1);
    setTimeout(function() {
        runAnimation(2);
        ...
    }, 1000);
}, 1000);
```

```
runAnimation(0);
delay(1000)
    .then(function() {
        runAnimation(1);
        return delay(1000);
    })
    .then(function() {
        runAnimation(2);
    })
    ...
```

Цепочка обещаний позволяет избежать пирамиды из коллбэков.
Независимо от того, как много уровней коллбэков,
эквивалент на обещаниях будет плоским.

2. promise.then(onResolve, onReject)

```
new Promise(function(resolve, reject) {
    reject(' :( ');
})
    .then(null, function() {
        // Handle the rejected promise
        return 'some description of :(';
    })
    .then(
        function(data) { console.log('resolved: '+data); },
        function(error) { console.error('rejected: '+error); }
    );


// В консоли будет resolved: some description of :(

// Обработчики отказа возвращают по умолчанию успешное обещание.
```

2. promise.then(onResolve, onReject)

```
delay(1000)
    .then(function() {
        throw new Error("oh no.");
    })
    .then(null, function(error) {
        console.error(error);
    });

// Промис съедает исключения!

// любое исключение, брошенное в обработчиках, переданных в
promise.then, будет возвращено в отклоненное обещание — вместо
того, чтобы отобразиться в консоли, как вы могли бы ожидать
```

3. promise.catch(onReject)

```
try {
    runAnimation(0);
}
catch (e) {
    runBackup(0);
}
delay(1000)
    .then(function() {
        runAnimation(1);
        return delay(1000);
    })
    .catch(function() {
        runBackup(1);
    })
    .then(function() {
        runAnimation(2);
    })
    .catch(function() {
        runBackup(2);
    });
```

```
// promise.catch(handler) — это
эквивалент promise.then(null, handler)

// Допустим, у нас есть три шага
анимации, с секундным отставанием
между ними. Каждый шаг может бросить
исключение, — например, из-за
отсутствия поддержки браузером — после
каждого then мы добавим блок catch, в
котором сделаем нужные изменения, но
без анимации.
```

4. Promise.all([promise1, promise2, …])

Она возвращает обещание, которое успешно, если все аргументы успешны, и отклонен, когда любой из его аргументов отклонен. В случае успеха результирующее обещание содержит массив результатов каждого обещания, а в случае неудачи — ошибку первого неуспешного обещания.

Допустим, у нас есть три функции, первые две parallelAnimation1() и parallelAnimation2() возвращают обещания, когда анимация завершится, а третья finalAnimation() должна вызваться, когда завершатся первые две. Реализовать такую логику мы можем следующим образом:

```
Promise.all([
    parallelAnimation1(),
    parallelAnimation2()
]).then(function() {
    finalAnimation();
});
```

4. Promise.all([promise1, promise2, …])

```
function finalRequestPromise(options) {
    return new Promise(function(resolve, reject) {
        finalRequest(options, function(error, data) {
            if (error) {
                reject(error);
            }
            else {
                resolve(data);
            }
        });
    });
}

Promise
    .all([initialRequestA(), initialRequestB()])
    .then(function(results) {
        var options =
getOptionsFromInitialData(results[0], results[1]);
        return finalRequestPromise(options);
    })
    .then(
        function(file) { alert(file); },
        function(error) { alert('ERROR: '+error); }
    );
```

Пример:

1. Скачать два файла с сервера
2. Извлечь из них некоторые данные
3. Использовать из, чтобы загрузить третий файл с сервера
4. Отобразить данные из третьего файла с помощью alert() или вызвать alert() с сообщением об ошибке.

Приоритет событий и промисов

```
const bar = () => {
  console.log('bar');
};

const baz = () => {
  console.log('baz');
};

const foo = () => {
  console.log('foo');
  setTimeout(bar, 0);
  new Promise((resolve, reject) => {
    resolve('Promise resolved');
  }).then(res => console.log(res))
    .catch(err => console.log(err));
  baz();
};

foo();
```

Результат выполнения:

```
foo
baz
Promised resolved
bar
```

Промисы выполняются до setTimeout, потому что их ответ хранится в отдельной очереди, у которого более высокий приоритет чем у очереди событий.

См. https://zen.yandex.ru/media/nuancesprog/asinhronnyi-javascript--cikl-obrabotki-sobytii-5e20e0de3d008800b13f7d56

Из собеседования на Senior Frontend Developer в Альфабанке

JavaScript

```
setTimeout(() => console.log('setTimeout 1'), 0);//2

Promise.resolve().then(() => console.log('Promise 1'));//4

Promise.resolve().then(() => setTimeout(() => console.log('setTimeout 2'), 0));//6

Promise.resolve().then(() => console.log('Promise 2'));//5

setTimeout(() => console.log('setTimeout 3'), 0);//3

console.log('final');//1
```

Fetch, почти как $.ajax

Возвращает Promise:

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => response.json())
  .then(commits => alert(commits[0].author.login));
```

fetch спецификация отличается от jQuery.ajax():

Promise возвращаемый вызовом fetch() не перейдет в состояние "отклонено" из-за ответа HTTP, который считается ошибкой, даже если ответ HTTP 404 или 500. Вместо этого, он будет выполнен нормально (с значением false в статусе ok) и будет отклонён только при сбое сети или если что-то помешало запросу выполниться.

По умолчанию, fetch не будет отправлять или получать cookie файлы с сервера, в результате чего запросы будут осуществляться без проверки подлинности, что приведёт к неаутентифицированным запросам, если сайт полагается на проверку пользовательской сессии (для отправки cookie файлов в аргументе init options должно быть задано значение свойства credentials отличное от значения по умолчанию omit).

https://developer.mozilla.org/ru/docs/Web/API/Fetch_API/Using_Fetch

Fetch POST

```
const myPost = {
        title: title,
        body: body
    };
fetch('https://jsonplaceholder.typicode.com/posts', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(myPost)
    })
        .then((res) => res.json())
        .then((data) => console.log(data))
```

await

```
// Оператор await используется для ожидания окончания Promise. Может быть использован только внутри async function.

const response = await fetch('http://example.com/movies.json')
const myJson = await response.json();
console.log(JSON.stringify(myJson));
```

Оператор await заставляет функцию, объявленную с использованием оператора async, ждать выполнения Promise и продолжать выполнение после возвращения Promise значения. Впоследствии возвращает полученное из Promise значение. Если типом значения, к которому был применен оператор await, является не Promise, то значение приводится к успешно выполненному Promise.

Если Promise отклоняется, то await генерирует исключение с отклонённым значением.

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/await

await

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  var x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}
f1();
```

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/await

await

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function add1(x) {
  const a = await resolveAfter2Seconds(20);
  const b = await resolveAfter2Seconds(30);
  return x + a + b;
}

add1(10).then(v => {
  console.log(v);  // prints 60 after 4 seconds.
});

async function add2(x) {
  const a = resolveAfter2Seconds(20);
  const b = resolveAfter2Seconds(30);
  return x + await a + await b;
}

add2(10).then(v => {
  console.log(v);  // prints 60 after 2 seconds.
});
```

Не путайте await и Promise.all

Функция add1 приостанавливается на 2 секунды для первого await и еще на 2 для второго. Второй таймер создается только после срабатывания первого. В функции add2 создаются оба и оба же переходят в состояние await. В результате функция add2 завершится скорее через две, чем через четыре секунды, поскольку таймеры работают одновременно. Однако запускаются они все же не паралелльно, а друг за другом - такая конструкция не означает автоматического использования Promise.all. Если два или более Promise должны разрешаться параллельно, следует использовать Promise.all.

Ссылки

https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects
https://github.com/mdn/js-examples/blob/master/promises-test/index.html
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/async_function
https://developer.mozilla.org/ru/docs/Web/JavaScript/reference/Global_Objects/Promise
https://learn.javascript.ru/promise
https://getinstance.info/articles/javascript/grokking-es6-promises-the-four-functions-you-need-to-avoid-callback-hel/
jsraccoon.ru/es6-arrow-functions