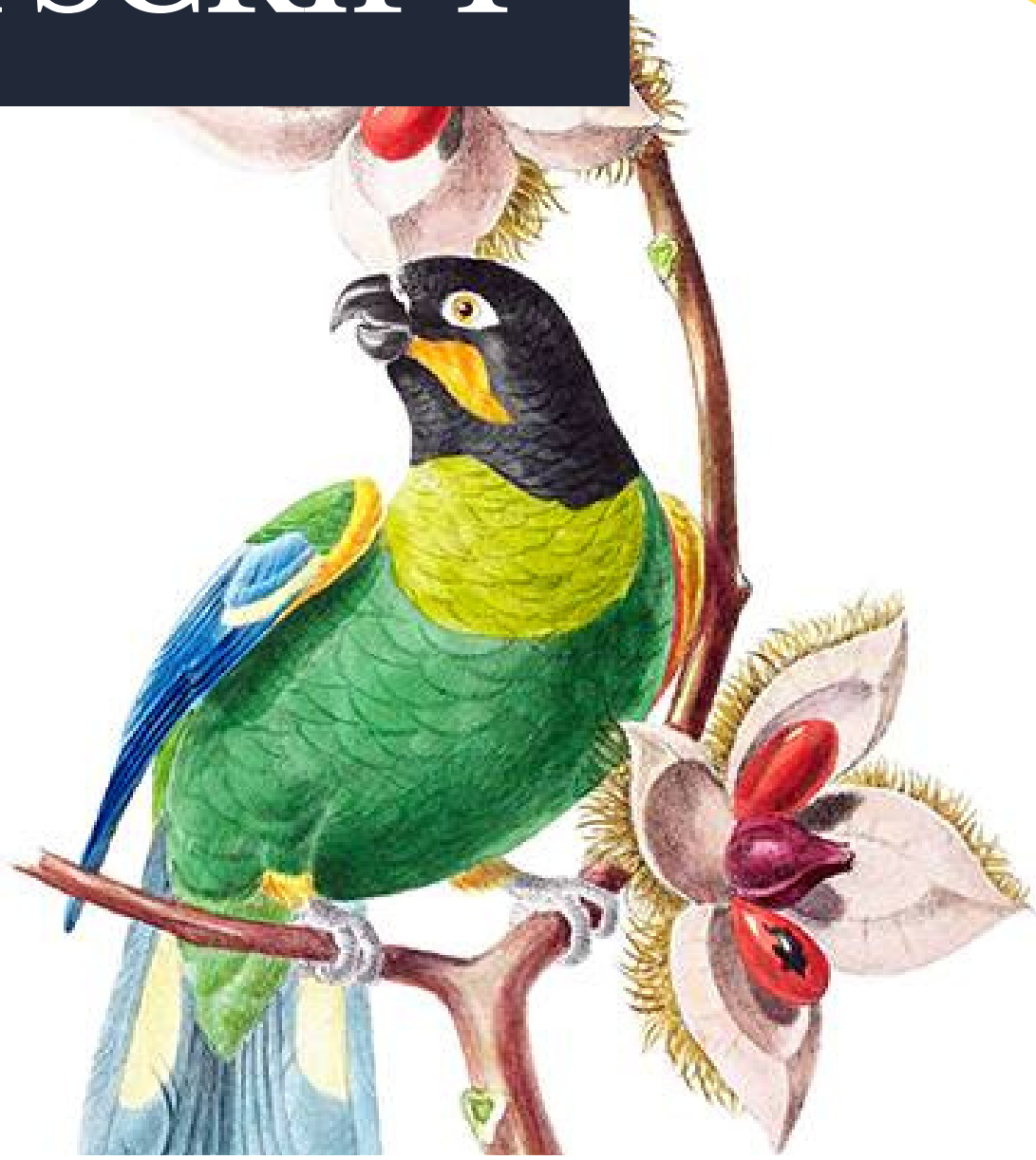
BRUNNA CROCHES

PARTE 1

JAVA SCRIPT



Artist Image :Descourtilz, Jean-Théodore | Dates:179?-1855

Guia Avançado: JavaScript

ABOUT ME

Brunna Croches

Developer Full Stack

Brunna Croches é Dev FullStack, advogada e empreendedora.

Apaixonada por tech, vem adquirido vasto conhecimento na área.

Desenvolveu projetos ricos em diversidade, buscando captar as próximas tendências e necessidades do mercado.

Neste e-book você aprenderá ou recapitulará de forma simplificada e otimizados conceitos de programação feito por ela.

let's share

SUMMARY

DOING

01

FUNÇÕES ESSENCIAIS

02

COLEÇÕES CHAVEADAS

03

CLASSES

04

1.0 - FUNÇÕES ESSENCIAIS

HTML Content

David Walsh

DAVID WALSH

Tem 31 artigospublicados com180267 visualizaçõesdesde 2011

7 funções essenciais em JavaScript

100 visualizações COMPARTILHE!

DAVID WALSH

31

É desenvolvedor web e engenheiro de software. Atualmente, é desenvolvedor web sênior e evangelista na Mozilla, palestrante de conferências e desenvolvedor do MooTools JavaScript Framework e de vários plugins MooTools.

LEIA MAIS

23 AGO, 2016
Prevenindo commits ruins com Husky

19 MAI, 2016
Crie um carregador básico com JavaScript Promises

22 ABR, 2016
POST de dados de formulário com cURL

Eu me lembro dos primeiros dias do JavaScript, quando era necessário usar uma função simples para quase tudo, porque os fabricantes de navegadores implementavam recursos de forma diferente, e não apenas recursos básicos, como addEventListenere

attachEvent. Os tempos mudaram, mas ainda existem algumas funções muito úteis que cada desenvolvedor deve ter em seu arsenal, para aumentar o desempenho do desenvolvimento e para fins de facilidades funcionais.

debounce

A função debouncepode ser um divisor de águas quando se trata de desempenho impulsionado por eventos. Se você não estiver usando uma função debouncecom os eventos scroll, resize e key*, então provavelmente você está fazendo algo errado. Aqui está uma função debouncepara manter seu código eficiente:

```
// Returns a function, that, as long as it continues to be invoked, will not
// be triggered. The function will be called after it stops being called for
// N milliseconds. If `immediate` is passed, trigger the function on the
// leading edge, instead of the trailing.
function debounce(func, wait, immediate) {
    var timeout;
    return function() {
        var context = this, args = arguments;
        var later = function() {
            timeout = null;
            if (!immediate) func.apply(context, args);
        };
        var callNow = immediate && !timeout;
        clearTimeout(timeout);
        timeout = setTimeout(later, wait);
        if (callNow) func.apply(context, args);
    };
};

// Usage
var myEfficientFn = debounce(function() {
    // All the taxing stuff you do
}, 250);
window.addEventListener('resize', myEfficientFn);
```

A função debouncenão permitirá que um callback seja usado mais de uma vez por um determinado período de tempo. Isso é especialmente importante quando a atribuição de uma função de callback para eventos é utilizada com uma frequência de disparo.

poll

Como mencionei em relação à função debounce, às vezes você não consegue se conectar a um evento para verificar seu estado – se o evento não existir, você precisa verificar o seu estado em intervalos:

```
function poll(fn, callback, errback, timeout, interval) {
    var endTime = Number(new Date()) + (timeout || 2000);
    interval = interval || 100;

    (function p() {
            // If the condition is met, we're done!
            if(fn()) {
                callback();
            }
            // If the condition isn't met but the timeout hasn't elapsed, go again
            else if (Number(new Date()) < endTime) {
```

```
                setTimeout(p, interval);
            }
            // Didn't match and too much time, reject!
            else {
                errback(new Error('timed out for ' + fn + ': ' + arguments));
            }
        })();
    }

    // Usage:  ensure element is visible
    poll(
        function() {
            return document.getElementById('lightbox').offsetWidth > 0;
        },
        function() {
            // Done, success callback
        },
        function() {
            // Error, failure callback
        }
    );
```

O polling tem sido útil na web e continuará sendo no futuro!

once

Há momentos em que você pode preferir que uma determinada funcionalidade só aconteça uma vez, semelhante à maneira como você usaria um evento onload. Este código fornece esse recurso rapidamente:

```
    function once(fn, context) {
        var result;

        return function() {
            if(fn) {
                result = fn.apply(context || this, arguments);
                fn = null;
            }

            return result;
        };
    }

    // Usage
    var canOnlyFireOnce = once(function() {
        console.log('Fired!');
    });

    canOnlyFireOnce(); // "Fired!"
    canOnlyFireOnce(); // nada
```

A função oncegarante que uma determinada função só pode ser chamada uma vez e, assim, evitar a inicialização duplicada!

getAbsoluteUrl

Obter uma URL absoluta de uma cadeia variável de dados não é tão fácil quanto você pensa. Há o construtor URLque pode ser usado para isso, mas ele não funciona se você não fornecer os argumentos necessários (que às vezes você simplesmente não tem). Aqui está um truque simples para obter uma URL absoluta de uma cadeia de entrada:

```
var getAbsoluteUrl = (function() {
    var a;

    return function(url) {
        if(!a) a = document.createElement('a');
        a.href = url;

        return a.href;
    };
})();

// Usage
getAbsoluteUrl('/something'); // http://davidwalsh.name/something
```

O elemento “burn” do hrefcria um objeto com a URL, fornecendo uma URL absoluta no retorno.

isNative

Saber se uma determinada função é nativa ou não pode sinalizar se você poderia substituí-la. Este código vem a calhar para lhe dar a resposta:

```
;(function() {

  // Used to resolve the internal `[[Class]]` of values
  var toString = Object.prototype.toString;

  // Used to resolve the decompiled source of functions
  var fnToString = Function.prototype.toString;

  // Used to detect host constructors (Safari > 4; really typed array specific)
  var reHostCtor = /^\[object .+?Constructor\]$/;

  // Compile a regexp using a common native method as a template.
  // We chose `Object#toString` because there's a good chance it is not being mucked with.
  var reNative = RegExp('^' +
    // Coerce `Object#toString` to a string
    String(toString)
    // Escape any special regexp characters
    .replace(/[.*+?^${}()|[\]\/\\]/g, '\\amp;')
    // Replace mentions of `toString` with `.*?` to keep the template generic.
    // Replace thing like `for ...` to support environments like Rhino which add extra info
    // such as method arity.
    .replace(/toString|(function).*?(?=\\\()| for .+?(?=\\\])/g, '$1.*?') + '#039;
  );

  function isNative(value) {
    var type = typeof value;
    return type == 'function'
      // Use `Function#toString` to bypass the value's own `toString` method
```

```
        // and avoid being faked out.
        ? reNative.test(fnToString.call(value))
        // Fallback to a host object check because some environments will represent
        // things like typed arrays as DOM methods which may not conform to the
        // normal native pattern.
        : (value && type == 'object' && reHostCtor.test(toString.call(value))) || false;
    }

    // export however you want
    module.exports = isNative;
}());

// Usage
isNative(alert); // true
isNative(myCustomFunction); // false
```

A função não é bonita, mas faz um trabalho bem feito!

insertRule

Nós todos sabemos que podemos pegar um NodeList de um seletor (via document.querySelectorAll) e dar a cada um deles um estilo próprio, mas o que é mais eficiente é definir um estilo a um seletor (assim como seria feito em uma folha de estilo):

```
var sheet = (function() {
    // Create the <style> tag
    var style = document.createElement('style');

    // Add a media (and/or media query) here if you'd like!
    // style.setAttribute('media', 'screen')
    // style.setAttribute('media', 'only screen and (max-width : 1024px)')

    // WebKit hack 😕
    style.appendChild(document.createTextNode(''));

    // Add the <style> element to the page
    document.head.appendChild(style);

    return style.sheet;
})();

// Usage
sheet.insertRule("header { float: left; opacity: 0.8; }", 1);
```

Isso é especialmente útil quando se trabalha em um site dinâmico, contendo AJAX pesado. Se você definir o estilo de um seletor, não precisará dar conta de denominar cada elemento que pode corresponder ao seletor (agora ou no futuro).

matchesSelector

Muitas vezes nós validamos a entrada antes de avançar, garantindo um valor verdadeiro, garantindo que os dados de formulários são válidos etc. Mas quantas vezes temos que garantir que um determinado elemento se qualifica para seguir em frente? Você pode usar uma função matchesSelectorpara validar se um elemento é de um determinado jogo de seletores:

```
function matchesSelector(el, selector) {
    var p = Element.prototype;
    var f = p.matches || p.webkitMatchesSelector || p.mozMatchesSelector || p.msMatchesSelector || f
        return [].indexOf.call(document.querySelectorAll(s), this) !== -1;
    };
    return f.call(el, selector);
}

// Usage
matchesSelector(document.getElementById('myDiv'), 'div.someSelector[some-attribute=true]')
```

Agora você tem sete funções JavaScript que todo desenvolvedor deve ter em sua caixa de ferramentas. Tem alguma função muito útil que eu perdi? Por favor, compartilhe!

***

David Walshfaz parte do time de colunistas internacionais do iMasters. A tradução do artigo é feita pela redação iMasters, com autorização do autor, e você pode acompanhar o artigo em inglês no link: http://davidwalsh.name/essential-javascript-functions

2.0 - COLEÇÕES CHAVEADAS (GET,SET)

Coleções chaveadas
Estes objetos representam coleções que usam chaves; estas contém elementos que são iteráveis na ordem de inserção.

- Map
- Set
- WeakMap
- WeakSet

Set e Getter

- ## set

O set permite que você armazene valores *únicos* de qualquer tipo, desde valores primitivos a referências a objetos. Objetos Set são coleções de valores nas quais é possível iterar os elementos em ordem de inserção. Um valor no Set **pode ocorrer apenas uma vez**; ele é único na coleção do Set.

- ## getter

A **get** sintaxe associa uma propriedade de objeto a uma função que será chamada quando essa propriedade for pesquisada.

3.0 - CLASSES

Classes

são simplificações da linguagem para as heranças baseadas nos protótipos. A sintaxe para classes não introduz um novo modelo de herança de orientação a objetos em JavaScript. Classes em JavaScript provêm uma maneira mais simples e clara de criar objetos e lidar com herança.

Expressões de Classes

Uma **Expressão de Classe** (class expression) é outra forma para definir classes. Expressões de Classes podem possuir nomes ou não (anônimas). O nome dado para uma expressão de classe é local ao corpo da classe

```
// sem nome
let Retangulo = class {
  constructor(altura, largura) {
    this.altura = altura;
    this.largura = largura;
  }
};

// nomeada
let Retangulo = class Retangulo {
  constructor(altura, largura) {
    this.altura = altura;
    this.largura = largura;
  }
};
```

HTML Content

Classes

Classes em JavaScript são introduzidas no ECMAScript 2015 e são simplificações da linguagem para as heranças baseadas nos protótipos. A sintaxe para classes **não** introduz um novo modelo de herança de orientação a objetos em JavaScript. Classes em JavaScript provêm uma maneira mais simples e clara de criar objetos e lidar com herança.

Definindo classes

As Classes são, de fato, "funções especiais", e, assim como você pode definir "function expressions"e "function declarations", a sintaxe de uma classe possui dois componentes: "class expressions" e "class declarations".

Declarando classes

Uma maneira de definir uma classe é usando uma declaração de classe. Para declarar uma classe, você deve usar a palavra-chave `class` seguida pelo nome da classe (aqui "Retangulo").

```
class Retangulo {
  constructor(altura, largura) {
    this.altura = altura;
    this.largura = largura;
  }
}
```

Uso antes da declaração (Hoisting - Tradução Literal: Lançamento)

Uma diferença importante entre **declarações de funções**das **declarações de classes**, é que declararações de funções são hoistede declarações de classes não são. Primeiramente deve declarar sua classe para só então acessá-la, pois do contrário o código a seguir irá lançar uma exceção: `ReferenceError`:

```
const p = new Retangulo(); // Erro de referência (ReferenceError)

class Retangulo {}
```

Expressões de Classes

Uma **Expressão de Classe**(class expression) é outra forma para definir classes. Expressões de Classes podem possuir nomes ou não (anônimas). O nome dado para uma expressão de classe é local ao corpo da classe.

```
// sem nome
let Retangulo = class {
  constructor(altura, largura) {
    this.altura = altura;
    this.largura = largura;
  }
};

// nomeada
let Retangulo = class Retangulo {
  constructor(altura, largura) {
    this.altura = altura;
    this.largura = largura;
  }
};
```

Nota: As **expressões de classe**também sofrem com o mesmo problema de hoistedmencionados em **declarações** de classe.

Corpo de uma classe e definições de métodos

O corpo de uma classe é a parte que está entre chaves `{}`. É aí onde você define os membros da classe, como os métodos, ou os construtores.

Modo Estrito (strict mode)

Os corpos das Declarações de Classes e das Expressões de Classes são executados em modo estrito.

Construtor

O método `constructor`é um tipo especial de método para criar e iniciar um objeto criado pela classe. Só pode existir um método especial com o nome "constructor" dentro da classe. Um erro de sintáxe `SyntaxError`será lançado se a classe possui mais do que uma ocorrência do método `constructor`.

Um construtor pode usar a palavra-chave `super` para chamar o construtor de uma classe pai.

Métodos Protótipos

Veja também definições de métodos (method definitions).

```
class Retangulo {
    constructor(altura, largura) {
      this.altura = altura; this.largura = largura;
    }
  //Getter
    get area() {
        return this.calculaArea()
    }

    calculaArea() {
        return this.altura * this.largura;
    }
}

const quadrado = new Retangulo(10, 10);

console.log(quadrado.area);
```

Métodos estáticos

A palavra-chave `static` define um método estático de uma classe. Métodos estáticos são chamados sem a instanciação da sua classe e não podem ser chamados quando a classe é instanciada. Métodos estáticos são geralmente usados para criar funções de utilidades por uma aplicação.

```
class Ponto {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static distancia(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;

        return Math.hypot(dx, dy);
    }
}

const p1 = new Ponto(5, 5);
const p2 = new Ponto(10, 10);

p1.distancia; //undefined
p2.distancia; //undefined

console.log(Ponto.distancia(p1, p2));
```

Empacotando com protótipos e métodos estáticos

Quando um método estático ou protótipo é chamado sem um objeto "this" configurado (ou com "this" como boolean, string, number, undefined ou null), então o valor "this" será **`undefined`** dentro da função chamada. Autoboxing não vai acontecer. O comportamento será o mesmo mesmo se escrevemos o código no modo não-estrito.

```
class Animal {
  falar() {
    return this;
  }
  static comer() {
    return this;
  }
}

let obj = new Animal();
obj.falar(); // Animal {}
let falar = obj.falar;
falar(); // undefined

Animal.comer(); // class Animal
let comer = Animal.comer;
comer(); // undefined
```

Se escrevemos o código acima usando classes baseadas em função tradicional, então o autoboxing acontecerá com base no valor de "this" para o qual a função foi chamada.

```
function Animal() { }

Animal.prototype.falar = function() {
  return this;
}

Animal.comer = function() {
  return this;
}

let obj = new Animal();
let falar = obj.falar;
falar(); // objeto global

let comer = Animal.comer;
comer(); // objeto global
```

Propriedades de instância

Propriedades de instâncias devem ser definidas dentro dos métodos da classe:

```
class Retangulo {
  constructor(altura, largura) {
    this.altura = altura;
    this.largura = largura;
  }
}
```

Propriedades de dados estáticos e propriedades de dados prototipados (prototype) devem ser definidos fora da declaração do corpo da classe.

```
Retangulo.larguraEstatico = 20;
Retangulo.prototype.larguraPrototipagem = 25;
```

Sub classes com o `extends`

A palavra-chave `extends` é usada em uma *declaração de classe*, ou em uma *expressão de classe* para criar uma classe como filha de uma outra classe.

```
class Animal {
  constructor(nome) {
    this.nome = nome;
  }

  falar() {
    console.log(this.nome + ' emite um barulho.');
  }
}

class Cachorro extends Animal {
  falar() {
    console.log(this.nome + ' latidos.');
  }
}

let cachorro = new Cachorro('Mat');
cachorro.falar();
```

Se existir um contrutor nas subclasses, é necessário primeiro chamar super() antes de usar a keyword "this".

Também é possivel ampliar (extends) "classes" baseadas em funções tradicionais.

```
function Animal (nome) {
  this.nome = nome;
}

Animal.prototype.falar = function() {
   console.log(this.nome + ' faça barulho.');
}

class Cachorro extends Animal {
  falar() {
    console.log(this.nome + ' lati.');
  }
}
```

```
let cachorro = new Cachorro('Mitzie');
cachorro.falar(); // Mitzie lati.
```

Note que classes não extendem objetos normais (não construíveis). Se você quer herdar de um objeto, é necessário utilizar `Object.setPrototypeOf()`:

```
let Animal = {
   falar() {
      console.log(this.nome + ' faça barulho.');
   }
};

class Cachorro {
   constructor(nome) {
      this.nome = nome;
   }
}

Object.setPrototypeOf(Cachorro.prototype, Animal);

let cachorro = new Cachorro('Mitzie');
cachorro.falar(); //Mitzie faça barulho.
```

Species

Você pode querer retornar um objeto `Array` na sua classe `MinhaArray` derivada de array. O padrão Species permite a sobrescrita do construtor padrão.

Por exemplo, quando utilizando um método como `map()` que retorna o construtor padrão, você pode querer que esse método retorne um objeto `Array` ao invés do objeto `MinhaArray`.
O `Symbol.species` te permite fazer isso:

```
class MinhaArray extends Array {
   // Sobrescreve species para o construtor da classe pai Array
   static get [Symbol.species]() { return Array; }
}

let a = new MinhaArray(1,2,3);
let mapped = a.map(x => x * x);

console.log(mapped instanceof MyArray); // false
console.log(mapped instanceof Array); // true
```

Chamada da classe pai com `super`

A palavra-chave (keyword) `super` é utilizada para chamar funções que pertencem ao pai do objeto.

```
class Gato {
   constructor(nome) {
      this.nome = nome;
```

```
    }

    falar() {
        console.log(this.nome + ' faça barulho.');
    }
}

class Leao extends Gato {
    falar() {
        super.falar();
        console.log(this.nome + ' roars.');
    }
}

let leao = new Leao('Fuzzy');
leao.falar();

// Fuzzy faça barulho.
// Fuzzy roars.
```

Mix-ins

Subclasses abstratas ou *mix-ins*são templates para classes. Uma classe do ECMAScript pode apenas ter uma classe pai, assim sendo, não é possível a classe ter herança múltipla.

Para se ter um comportamento similar ao de herança múltipla no ECMAScript usa-se mix-ins, uma forma de implementar mix-ins é usar um template de subclasse que é uma função que instancia uma classe base e retorna uma subclasse extendida desta classe base:

```
class Humano {
  constructor(nome) {
    this.nome = nome;
  }
  andar() {
    return this.nome+' andou um passo'
  }
}

const HumanoFalante = Base => class extends Base {
  falar() {
    return this.nome+' diz: olá mundo!'
  }
}

const HumanoFalanteMixado = Base => class extends Base {}

const HumanoFinal = HumanoFalanteMixado(HumanoFalante(Humano))

const humano = new HumanoFinal('Bill Gates')

console.log(humano.andar())
console.log(humano.falar())
```

4.0 - OPERADORES

Operadores Síncronos

- **this**

Operadores Assíncronos :

- **.then()** : encadear várias operações assíncronas usando várias .then() operações, passando o resultado de uma para próxima como uma entrada.
- **.catch()** : todos os erros são tratados por um único .cath() bloco no final do bloco, em vez de serem tratados individualmente em cada nível da pirâmide.

CONTACT

Brunna Croches

Developer Full Stack

brunnacroches.dev

linkedin.com/brunnacroches

github.com/brunnacroches

@brunnacroches.dev

discord.com/brunnacroches

brunnacroches@gmail.com

let's share