Código Limpo Javascript

Obra Original:
GitHub - ryanmcdermott/clean-code-javascript

Tradução:
GitHub - felipe-augusto/clean-code-javascript: Conceitos de Código Limpo adaptados em JavaScript (Tradução PT-BR)

Adaptação Ebook:
https://github.com/luisrguerra/codigo-limpo-javascript-ebook-ptbr

Introdução

Princípios da Engenharia de Software, do livro de Robert C. Martin Código Limpo, adaptados para JavaScript. Isto não é um guia de estilos. É um guia para se produzir código legível, reutilizável e refatorável em JavaScript.

Nem todo princípio demonstrado deve ser seguido rigorosamente, e ainda menos são os que possuem consenso universal. São orientações e nada mais, entretanto, foram usadas em código durante muitos anos de experiência coletiva pelos autores de Código limpo.

Nosso ofício de engenharia de software tem pouco mais de 50 anos e ainda estamos aprendendo muito. Quando a arquitetura de software for tão velha quanto a própria arquitetura, talvez então tenhamos regras mais rígidas para seguir. Por enquanto, deixe que estas orientações sirvam como critério para se

avaliar a qualidade de código JavaScript que tanto você e o seu time produzirem.

Mais uma coisa: aprender isto não irá lhe transformar imediatamente em um desenvolvedor de software melhor e trabalhar com eles por muitos anos não quer dizer que você não cometerá erros. Toda porção de código começa com um rascunho, como argila molhada sendo moldada em sua forma final. Finalmente, talhamos as imperfeições quando revisamos com nossos colegas. Não se bata pelos primeiros rascunhos que ainda precisam de melhorias. Ao invés, bata em seu código.

Variáveis

Use nomes de variáveis que tenham significado e sejam pronunciáveis

Ruim:

```
const yyyymmdstr = moment().format('YYYY/MM/DD');
```

Bom:

```
const currentDate = moment().format('YYYY/MM/DD');
```

Use o mesmo vocabulário para o mesmo tipo de variável

Ruim:

```
getUserInfo();
getClientData();
getCustomerRecord();
```

Bom:

```
getUser();
```

Use nomes pesquisáveis

Nós iremos ler mais código que escrever. É importante que o código que escrevemos seja legível e pesquisável. Não dando nomes em variáveis que sejam significativos para entender nosso programa, machucamos

nossos leitores. Torne seus nomes pesquisáveis. Ferramentas como buddy.js e ESLint podem ajudar a identificar constantes sem nome.

Ruim:

```
// Para que diabos serve 86400000?
setTimeout(blastOff, 86400000);
```

Bom:

```
// Declare-as como `const` global em letras maiúsculas.
const MILLISECONDS_IN_A_DAY = 86400000;

setTimeout(blastOff, MILLISECONDS_IN_A_DAY);
```

Use variáveis explicativas

Ruim:

```
const address = 'One Infinite Loop, Cupertino 95014';
const cityZipCodeRegex = /^[^,\\]+[,\\\s]+(.+?)\s*(\d{5})?$/;
saveCityZipCode(address.match(cityZipCodeRegex)[1],
address.match(cityZipCodeRegex)[2]);
```

Bom:

```
const address = 'One Infinite Loop, Cupertino 95014';
const cityZipCodeRegex = /^[^,\\]+[,\\\s]+(.+?)\s*(\d{5})?$/;
const [, city, zipCode] = address.match(cityZipCodeRegex) || [];
saveCityZipCode(city, zipCode);
```

Evite Mapeamento Mental

Explícito é melhor que implícito.

Ruim:

```
const locations = ['Austin', 'New York', 'San Francisco'];
locations.forEach((l) => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  // Espera, para que serve o `l` mesmo?
  dispatch(l);
});
```

Bom:

```
const locations = ['Austin', 'New York', 'San Francisco'];
locations.forEach((location) => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  dispatch(location);
});
```

Não adicione contextos desnecessários

Se o nome de sua classe/objeto já lhe diz alguma coisa, não as repita nos nomes de suas variáveis.

Ruim:

```
const Car = {
  carMake: 'Honda',
  carModel: 'Accord',
  carColor: 'Blue'
};

function paintCar(car) {
  car.carColor = 'Red';
}
```

Bom:

```
const Car = {
  make: 'Honda',
  model: 'Accord',
  color: 'Blue'
};

function paintCar(car) {
  car.color = 'Red';
}
```

Use argumentos padrões ao invés de curto circuitar ou usar condicionais

Argumentos padrões são geralmente mais limpos do que curto circuitos. Esteja ciente que se você usá-los, sua função apenas irá fornecer valores padrões para argumentos `undefined`. Outros valores "falsos" como `''`, `""`, `false`, `null`, `0`, e `NaN`, não serão substituidos por valores padrões.

Ruim:

```
function createMicrobrewery(name) {
  const breweryName = name || 'Hipster Brew Co.';
  // ...
}
```

Bom:

```
function createMicrobrewery(breweryName = 'Hipster Brew Co.') {
  // ...
}
```

Funções

Argumentos de funções (idealmente 2 ou menos)

Limitar a quantidade de parâmetros de uma função é incrivelmente importante porque torna mais fácil testá-la. Ter mais que três leva a uma explosão combinatória onde você tem que testar muitos casos diferentes com cada argumento separadamente.

Um ou dois argumentos é o caso ideal, e três devem ser evitados se possível. Qualquer coisa a mais que isso deve ser consolidada. Geralmente, se você tem mais que dois argumentos então sua função está tentando fazer muitas coisas. Nos casos em que não está, na maioria das vezes um objeto é suficiente como argumento.

Já que JavaScript lhe permite criar objetos instantaneamente, sem ter que escrever muita coisa, você pode usar um objeto se você se pegar precisando usar muitos argumentos.

Para tornar mais óbvio quais as propriedades que as funções esperam, você pode usar a sintaxe de desestruturação (destructuring) do ES2015/ES6. Ela possui algumas vantagens:

1. Quando alguém olha para a assinatura de uma função, fica imediatamente claro quais propriedades são usadas.

2. Desestruturação também clona os valores primitivos específicos do objeto passado como argumento para a função. Isso pode ajudar a evitar efeitos colaterais. Nota: objetos e vetores que são desestruturados a partir do objeto passado por argumento NÃO são clonados.

3. Linters podem te alertar sobre propriedades não utilizadas, o que seria impossível sem usar desestruturação.

Ruim:

```
function createMenu(title, body, buttonText, cancellable) {
  // ...
}
```

Bom:

```
function createMenu({ title, body, buttonText, cancellable }) {
  // ...
}

createMenu({
  title: 'Foo',
  body: 'Bar',
  buttonText: 'Baz',
  cancellable: true
});
```

Funções devem fazer uma coisa

Essa é de longe a regra mais importante em engenharia de software. Quando funções fazem mais que uma coisa, elas se tornam difíceis de serem compostas, testadas e raciocinadas. Quando você pode isolar uma função para realizar apenas uma ação, elas podem ser refatoradas facilmente e seu código ficará muito mais limpo. Se você não levar mais nada desse guia além disso, você já estará na frente de muitos desenvolvedores.

Ruim:

```
function parseBetterJSAlternative(code) {
  const REGEXES = [
    // ...
  ];

  const statements = code.split(' ');
  const tokens = [];
  REGEXES.forEach((REGEX) => {
    statements.forEach((statement) => {
      // ...
    });
  });

  const ast = [];
  tokens.forEach((token) => {
    // lex...
  });

  ast.forEach((node) => {
    // parse...
  });
}
```

Bom:

```
function tokenize(code) {
  const REGEXES = [
    // ...
  ];

  const statements = code.split(' ');
  const tokens = [];
  REGEXES.forEach((REGEX) => {
    statements.forEach((statement) => {
      tokens.push( /* ... */ );
    });
  });

  return tokens;
}
```

```
function lexer(tokens) {
  const ast = [];
  tokens.forEach((token) => {
    ast.push( /* ... */ );
  });

  return ast;
}

function parseBetterJSAlternative(code) {
  const tokens = tokenize(code);
  const ast = lexer(tokens);
  ast.forEach((node) => {
    // parse...
  });
}
```