Array.reduce — O canivete suíço da programação funcional

Gabriel Colombo Follow

Jun 4, 2018 · 5 min read

Imagem por Paul Felberbauer

Existem três operações fundamentais para transformar listas que todo desenvolvedor deveria conhecer— *Map*, *filter* e *reduce*.

Map é utilizado para transformar os itens de uma lista. *Filter* tem como função filtrar os itens de uma lista que satisfazem uma determinada

condição.

Já *reduce* difere da norma.

Nesse artigo veremos sua filosofia, funcionamento e algumas aplicações cotidianas interessantes.

Prontos? Vamos lá!

Introdução

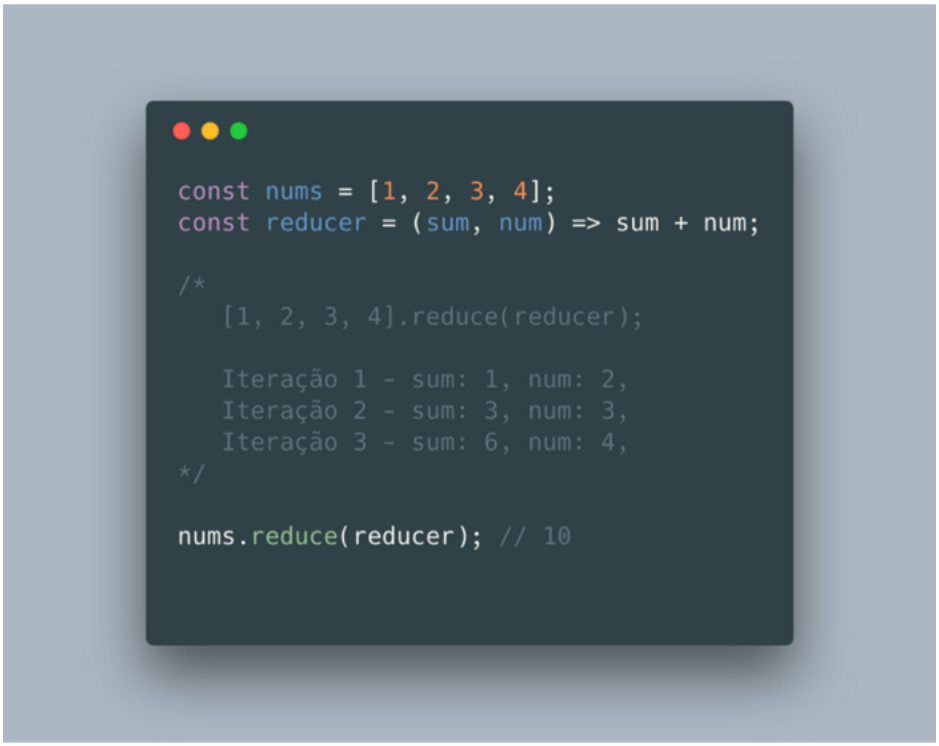
Ao contrário dos métodos *map* e *filter, o* r*educe*—também conhecido com *fold*—é um método de combinação.

Reduce opera em uma lista de valores como os outros métodos, porém, ao invés de retornar uma nova lista, seu propósito é combinar todos os valores em um único valor.

O primeiro parâmetro esperado pelo método é um callback—o *reducer* —com dois parâmetros:

- **Acumulador:** Valor único que será retornado ao final da operação.
- **Item atual:** Item do array que está sendo iterado.

Para cada item da lista, o *reducer* é executado, recebendo como parâmetro o valor acumulado da operação anterior e o item que está sendo iterado.



A iteração inicial recebe como soma o primeiro item da lista

O segundo parâmetro do *reduce* representa o valor inicial do acumulador. Caso não seja informado, é atribuído como valor inicial o primeiro item da lista.

O resultado final da redução pode variar de acordo com o valor inicial do acumulador

No exemplo acima temos como objetivo criar uma sigla utilizando a primeira letra de cada palavra de uma lista.

Na primeira redução não é informado um valor inicial para o acumulador, assumindo assim o primeiro item da lista e ocasionando um resultado inesperado.

Já a segunda redução recebeu como valor inicial uma string vazia, levando ao resultado correto.

Tendo em mente a identidade do método apresentada, vamos ver algumas de suas aplicações.

Aplicações

A versatilidade do método *reduce* vêm da sua capacidade de combinar valores além dos tipos primitivos. Essa característica nos fornece várias possibilidades interessantes de utilização.

Segue abaixo alguns casos de uso que encontrei com o passar dos anos.

Singularização de valores

Em sua forma mais básica, reduce pode ser utilizado para singularizar uma lista de valores em um único valor, como encontrar o valor total de uma compra a partir de uma lista de produtos.

Partindo de uma lista, o valor de cada item é acrescentado ao acumulador— nesse caso, iniciado com zero—que é retornado ao final da iteração, representando a soma final dos itens.

```
const products = [
  { label: 'Product #1', value: 4.25 },
  { label: 'Product #2', value: 2.50 },
  { label: 'Product #3', value: 1.40 },
];
```

Redução de arrays multi-dimensionais

Ao trabalhar com grandes volumes de dados é comum encontrar estruturas multi-dimensionais, como *arrays* dentro de *arrays*.

Vamos tomar como exemplo uma lista de usuários. Cada usuário é representado por um objeto contendo seu nome e uma lista de seus filmes favoritos.

Tendo como objetivo criar uma lista dos filmes favoritos de todos os usuários, podemos iniciar utilizando um *map* para transformar essa lista de objetos em uma lista de arrays—cada array representando a lista de filmes favoritos de um usuário.

Em seguida, utilizamos um *reduce* para remover os itens duplicados e reduzir essa lista para uma simples lista de objetos, representando os filmes favoritos de todos os usuários.

```
const users = [
  {
    name: 'User #1',
    bookmarks: [
      { title: 'Movie #1', id: 1 },
      { title: 'Movie #6', id: 6 },
      { title: 'Movie #3', id: 3 },
    ]
  },
  {
    name: 'User #2',
    bookmarks: [
      { title: 'Movie #1', id: 1 },
      { title: 'Movie #4', id: 4 },
      { title: 'Movie #6', id: 6 },
      { title: 'Movie #2', id: 2 },
    ]
  }
];

const bookmarks = users
  .map(user => user.bookmarks)              /* from: [
  .reduce((bookmarks, movies) => {          /* to:   [
    return bookmarks
      .concat(movies.filter((movie) => {
        return bookmarks
          .map(movie => movie.id)
          .indexOf(movie.id) === -1;
```

Encadeamento de promises

Uma outra utilização interessante do *reduce* é a execução de promises encadeadas.

No exemplo abaixo temos um objeto indicando a localização dos templates de um componente de acordo com seu estado.

Nossa tarefa é carregar e compilar cada template, retornando um objeto com a mesma estrutura, porém substituindo os caminhos pelos templates compilados *à la Handlebars*.

O método *compileTemplates* é responsável por gerenciar a execução das promises em sequência—retornando uma promise com o resultado dos encadeamentos.

Em seguida, a lista com o resultado das promises é repassada para o método *parsePromiseHash* que, através de outro *reduce*, retorna um único objeto no mesmo formato do objeto inicial, substituindo o caminho dos templates pelo template já compilado.

Com isso, é possível realizar cache dos templates, evitando recompilações desnecessárias.

```
const states = {
  default: '_partials/default',
  error: '_partials/error',
}

function compileTemplates(templates) {
  return Object
    .keys(templates)
    .map(key => ({
      key,
      value() {
        return new Promise((resolve) => {
          setTimeout(() => resolve({
            key,
            value: (data) => `Template "${templates[ke
          }), 1000);
        });
      },
    }))
    .reduce((promise, { key, value }) => {
      return promise.then((result) => {
        return value().then(Array.prototype.concat.bin
      });
    }, Promise.resolve([]));
}

function parsePromiseHash(hash) {
  return new Promise((resolve) => {
    const obj = hash
      .reduce((templates, { key, value }) => {
```

Composição de funções

Composição é um dos conceitos chave da programação funcional—permitindo combinar duas ou mais funções em uma única função.

A ordem dos parâmetros da composição é reverso—o primeiro método passado como parâmetro será o último a ser executado.

Partindo de uma lista de funções, podemos utilizar o método reduce para combiná-las e criar uma função composta.

No exemplo abaixo, o método *compose* recebe múltiplas funções como parâmetro e, a cada iteração, retorna uma nova função.

Essa nova função é executada e seu resultado é repassado como parâmetro para a próxima função a ser iterada, e assim sucessivamente.

O resultado final é uma função que recebe os argumentos desejados, dá início ao processo e retorna o resultado dos processamentos.

```
const fetchUsers = () => {
  return [
    { name: 'User #1' },
    { name: 'User #2' },
  ];
}

const getUsernames = (users) => {
  return users.map(user => user.name);
}

const parseToHtml = (users) => {
  return users.map(user => `<li>${user}</li>`).join('
}

const compose = (...fns) => {
  return fns
    .reverse()
    .reduce((composition, fn) => {
```

Mixins

Mixins são módulos compartilhados entre várias estruturas que têm como propósito evitar replicação de código com a mesma finalidade em arquivos diferentes.

Por exemplo, partindo de uma classe base, podemos criar vários *mixins* responsáveis por tarefas genéricas, como validação e formatação de dados que podem ser compostos utilizando *reduce* para criar uma nova classe sem a necessidade de replicar os métodos.

A grande vantagem dessa abordagem é a possibilidade de compor múltiplos *mixins* em uma única classe, algo que não pode ser alcançado com simples herança.

No exemplo abaixo, a classe *Button* extende uma classe composta pelas classes *Component* e *UIStateMixin*.

O método *mix* recebe uma classe base que é utilizada como o valor inicial do acumulador.

Em seguida, é realizada a iteração sobre a lista de *mixins*. A cada iteração é retornada uma nova classe que extende a partir do acumulador—representando a classe final, composta de todos os valores.

```
const mix = (baseClass) => ({
  with(...mixins) {
    return mixins.reduce((newClass, mixin) => mixin(ne
  }
});

const UIStateMixin = (baseClass) => {
  return class extends baseClass {
    constructor() {
      super(...arguments);

      this.disabled = false;
      this.loading = false;
    }

    disable() {
      this.disabled = true;

      return this;
    }

    showLoadingIndicator() {
      this.loading = true;

      return this;
    }
  }
}

class Component {
```

Conclusão

Nesse artigo pudemos contemplar a versatilidade do método *reduce* bem como algumas de suas utilizações.

Dominar as operações fundamentais oferece um grande leque de possibilidades ao desenvolver suas aplicações, além de proporcionar um melhor entendimento sobre o funcionamento de frameworks e bibliotecas como Redux e RxJS.

That's all folks! Espero que tenham gostado!

Fique à vontade para entrar em contato comigo no twitter —@gcolombo_.

Obrigado!

. . .

Gostou do artigo? Compartilhe sua opinião conosco!

Não esqueça de seguir nossa página para receber todas as novidades :)