Isaias Soares
Id: 1178008

Composição e encadeamento (pipe) de funções

Um professor uma vez disse: Imagine que você quer explicar o que é uma função para uma tribo de pescadores em uma ilha onde você chegou como náufrago. Como você faria? Sem usar linguagem matemática é bem difícil explicar o que é uma função.

Se você der exemplos fica pior:

- Você identifica o pescador e ela retorna quantos peixes ele pegou.
- Você identifica um local de pesca e ela retorna os peixes.
- Você identifica um coco e ela retorna a quantidade de água dentro do coco.

Dê exemplos o suficiente e a tribo vai acabar cultuando a função como um novo deus, afinal ela pode fazer absolutamente qualquer coisa. Vale mais a pena procurar explicar por propriedades que a coisa tem ou pela sua relação com as outras coisas.

Funções, segundo esse professor, seriam explicadas por composição. Para ele, a propriedade mais fundamental das funções é poder compor com outras funções produzindo novas funções.

Como fazer composição de funções

Composição só pode ser feita por funções de apenas um parâmetro, e onde o retorno da função anterior tenha o mesmo tipo que o parâmetro da próxima. No entanto, a primeira função pode ter mais parâmetros, pois sua entrada não depende do retorno de outra função. Os parâmetros da primeira função são os parâmetros da função composta.

A composição é mais prática escrita em "pointfree", então vamos usar esses estilos. Outro ponto importante é que métodos não podem fazer composição como as funções. Métodos são funções pertencentes aos objetos, por exemplo, tudo que vimos de array e strings eram métodos e não funções. Por sorte é bem simples reescrever métodos como funções.

Composição de funções no JavaScript

Javascript não tem uma forma pronta de fazer composição de funções. Precisamos escrever nossa própria função de composição. Então, uma função de composição poderia ser:

```
const compose = (fn1, fn2) => function() {
  return fn2(fn1(...arguments))
};
```

A composição produz uma nova função em vez de invocar ambas as funções imediatamente. Essa função composta fará tudo o que as demais funções faziam. Usamos `...arguments` para indicar que os parâmetros passados para a função composta serão passados para a primeira função.

Observe que a função composta teve que ser `function` e não `arrow` porque `arrows`, assim como não tem acesso ao `this`, também não tem acesso ao `arguments`. Vamos fazer um exemplo, receber um valor como string, fazer `parseFloat`, e depois `Math.round` usando composição.

```
const compose = (fn1, fn2) => function() {
  return fn2(fn1(...arguments))
};

const parseAndRound = compose(parseFloat, Math.round);

console.log(parseAndRound("10.5")); // 11
```

`Math.round()` pode ser composto diretamente, pois é um método estático que pode ser tratado como função. Vamos ver como adaptar alguns métodos não estáticos para composição.

Reescrevendo métodos como funções para composição

Se o método não tiver parâmetros é moleza transformá-lo em função. Basta receber por parâmetro o tipo desejado e invocar seu método:

```
const toUpperCase = str => str.toUpperCase();

//Agora invés de:
console.log("teste".toUpperCase());

//podemos usar:
console.log(toUpperCase("teste"));
```

Vamos compor alguns métodos:

```
const compose = (fn1, fn2) => function(){ return fn2(fn1(...arguments)) };
const toUpperCase = str => str.toUpperCase();
const trim = str => str.trim();

const trimAndUpper = compose(trim, toUpperCase);

console.log(trimAndUpper("  teste  ")); // "TESTE"
```

No exemplo acima, `trim()` vai tirar os espaços em branco no começo e no final da string enquanto `toUpperCase()` vai deixar todas as letras maiúsculas.

Composição de várias funções

Usando a função de composição que fizemos é meio chato fazer múltiplas composições de uma vez, por exemplo, se quisermos fazer `trim()`, depois `parseFloat()`, depois `Math.Round()` ficaria assim:

```
const compose = (fn1, fn2) => function(){ return fn2(fn1(...arguments)) };
const trim = str => str.trim();

const trimParseAndRound = compose(compose(trim, parseFloat), Math.round);

console.log(trimParseAndRound("  10.4 ")); //10
```

Veja que pela associabilidade isso é equivalente:

```
const compose = (fn1, fn2) => function(){ return fn2(fn1(...arguments)) };
const trim = str => str.trim();

const trimParseAndRound = compose(trim, compose(parseFloat, Math.round));

console.log(trimParseAndRound("  10.4 ")); //10
```

Não importa quem é composto primeiro, contanto que as funções estejam na ordem certa a composição pode ser feita em qualquer ordem. Para simplificar nossa vida, podemos utilizar o `reduce()` para criar uma função que componha qualquer número de funções:

```
const compose = function(){
    const functions = [...arguments];

    return function(){
        return functions.reduce((res, fn) => fn(res), ...arguments);
    }
```

```
}

// agora podemos fazer

const trim = str => str.trim();
const trimParseAndRound = compose(trim, parseFloat, Math.round);

console.log(trimParseAndRound("  10.4 ")); //10
```

Muitas bibliotecas como a Ramda já têm funções de composição, mas tome cuidado, a maioria delas utiliza composição *right to left* o que significa que suas funções devem ser passadas em ordem inversa. A primeira será o último parâmetro. O exemplo acima ficaria `compose(Math.round, parseFloat, trim)`.

psiu! O professor que eu citei -> Bartosz Milewski.

Referências e Materiais Complementares

- Encadeamento opcional
- Encadeamento de Métodos em JavaScript

Próximo Tópico