30 days of JavaScript

Introdução: Dado um array de funções [f1, f2, f3, ..., fn], retorne uma nova função fn que é a composição funcional do array de funções.

A composição da função [f(x), g(x), h(x)] é fn(x) = f(g(h(x))).

A composição funcional de uma lista vazia de funções é a função identidade f(x) = x.

Você pode assumir que cada função no array aceita um número inteiro como entrada e retorna um número inteiro como saída.

Solução:

```
const compose = function(functions) {

    return function(x) {
        return functions.reduceRight((acc, fn) => fn(acc), x);
    }
};

const fn = compose([x => x + 1, x => 2 * x]);
fn(4); // 9
```

Explicação:

Eu criei uma função chamada compose, que é uma função de ordem superior. Isso significa que ela pode receber outras funções como argumento. A ideia principal dessa função é combinar várias funções em uma única função, onde a saída de uma função se torna a entrada da próxima. Para fazer isso, eu passei um array de funções como argumento para a função compose.

Dentro da função compose, eu retornei outra função que aceita um argumento x. Esta função interna usa o método reduceRight para aplicar as funções do array, começando da direita para a esquerda. Isso significa que a última função no array é a primeira a ser aplicada a x, e o resultado é passado para a próxima função, e assim por diante, até que todas as funções tenham sido aplicadas. O resultado final é retornado.

No exemplo que criei, eu usei a função compose para combinar duas funções: x => x + 1 e x => 2 * x. Quando eu chamei fn(4), a função compose aplicou essas duas funções em ordem, começando com x => 2 * x, que dobrou o valor de 4 para 8, e depois x => x + 1, que adicionou 1 a 8, resultando em 9.

Portanto, o valor retornado quando chamei fn(4) foi 9. Este código é útil para criar pipelines de transformações de dados, onde você deseja aplicar várias etapas de processamento de dados em sequência.