ntander Coders 2024.1 | Front-End | #1178

unções de alta-ordem

Isaias Soares
Id: 1178008

Funções de alta-ordem

Vamos começar criando algumas funções matemáticas simples:

```
const somar = (a, b) => a + b;
const subtrair = (a, b) => a - b;
const multiplicar  = (a, b) => a * b;
const dividir = (a, b) => b > 0 ? a / b : NaN;
```

Aqui temos as 4 operações básicas, todos os returns foram omitidos porque usamos arrow functions. A única um pouco mais complexa é a divisão, onde usamos um ternário para evitar divisões por zero, retornando "not a number" nesse caso.

Agora criaremos uma função de alta ordem que recebe uma operação (uma das funções acima) e os dois números e faz a conta.

```
const calcular = (operacao, a, b) => calcular(a,b);
```

O parâmetro `operacao` de calcular é uma função, ela é invocada usando `a` e `b` como seus parâmetros. Para utilizar a função calcular é simples, basta passar por parâmetro a função desejada e dois números.

```
const resultado = calcular(somar, 1, 2);
console.log(resultado); // 3
```

Mas qual é a vantagem?

Isso simplificaria um programa em que a escolha da operação é feita pelo usuário, por exemplo. Imagine que podemos ter 3 variáveis, valor1, valor2 e operação. Quando o usuário clica no botão de um uma operação (+ - * /) já colocamos a função correta na variável operação e depois aplicamos com `calcular()` a operação escolhida, isso nos pouparia um monte de condicionais. O mais importante aqui é observar que somar foi passado como um função, mas não invocado (não há parênteses).

Se procurarmos fazer um software priorizando esse tipo de sintaxe, tentando ao máximo delegar a invocação das funções e a passagem de parâmetros para outras estruturas teremos um estilo de programação chamado pointfree. Podemos também criar outra funções em termos de `calcular()` por exemplo:

```
const media = (a,b) => calcular(dividir, calcular(somar, a, b), 2.0);
//ou
const desconto = (preco) => calcular(subtracao, preco, calcular(multiplicar, preco, 0.1));
```

Não vou dizer quais são as formas mais fáceis de criar essas funções, mas é possível expressar qualquer operação complexa usando `calcular()` sem usar os parênteses de precedência e sem usar nenhum operador, apenas funções.

Em alguns momentos, trocar operadores binários como “+” por uma função pode ser mais apropriado, especialmente quando trabalhamos com árvores. Se quiser, dê uma olhada em operadores infix, prefix e postfix, mas isso é tópico de estrutura de dados que não trataremos aqui.

Retornando funções

Podemos fazer funções de alta ordem para retornar outras funções, por exemplo, imagine que você tem a mesma função em dois softwares diferentes, ela teria que fazer a mesma coisa, no entanto seu retorno, em um dos softwares será um JSON e no outro um XML.

Podemos fazer uma função de alta ordem produzir duas versões diferentes de uma função para adequarem o retorno ao programa que as invoca, assim não precisamos criar duas funções diferentes, uma em cada programa, que façam a mesma coisa.

```
const createFormatContatoFor = (software) => {

    if(software === "software1")
      return contato => JSON.stringify(contato);
    if(software === "software2")
      return contato => `
        <contato>
            <nome>${contato.nome}</nome>
            <telefone>${contato.telefone}</telefone>
        </contato>
    `
}
```

No software 1:

```
const formatContato = createFormatContatoFor("software1");
const contato = { nome: "teste", telefone: "000000000" };
console.log(formatContato(contato));
/*Saída
{"nome":"teste","telefone":"000000000"}
*/
```

No software 2:

```
const formatContato = createFormatContatoFor("software2");
const contato = { nome: "teste", telefone: "000000000" };
console.log(formatContato(contato));
/*Saída
```

```
<contato>
    <nome>teste</nome>
    <telefone>000000000</telefone>
</contato>
*/
```

> Vale mencionar que o XML acima não está aderente às regras de formatação é apenas ilustrativo.

No exemplo, `createFormatContatoFor()` utiliza o parâmetro software para decidir qual das duas versões utilizar, a que formata em JSON ou a que formata em XML. A função `formatContato()` é o resultado dessa decisão. Nela não há nenhum vestígio que ela foi produzida pela lógica interna de outra função, em todos os sentidos é uma função normal como qualquer outra.

Vale mencionar que as funções podem ser declaradas dentro de outras funções no JavaScript, não apenas retornadas. Também é interessante apontar que as funções internas têm acesso aos parâmetros, variáveis e constantes da função maior. Isso se dá por um conceito chamado "closure".

No nosso exemplo, se as funções internas precisassem do valor do parâmetro `software` para alguma coisa, poderiam utilizá-lo sem problemas. Logo mais veremos o que é aplicação parcial, com ela podemos utilizar o conceito que acabamos de ver para produzir funções especializadas a partir de funções mais genéricas.

Referências e Materiais Complementares

- JavaScript: Introdução e funções de alta ordem
- Higher Order Functions

Próximo Tópico