06

Mais composição

Transcrição

Se quisermos realizar composição com auxílio da nossa função `pipe` precisaremos adequar essas funções para que recebam um parâmetro apenas.

Vamos recordar a implementação das funções:

```
// app/utils/operators.js

// código anterior omitido
export const takeUntil = (times, fn) => () => times-- > 0 && fn();

export const debounceTime = (milliseconds, fn) => {
    let timer = 0;
    return () => {
        clearTimeout(timer)
        timer = setTimeout(fn, milliseconds);
    };
};
// código posterior omitido
```

Ambas retornam uma função configurada que será utilizada pela nossa aplicação. Vejamos um exemplo de uso isolado para refrescarmos nossa memória:

```
// exemplo apenas, não entra em nosso programa
const doTake = takeUntil(2, () => console.log('Chamou callback!'));
doTake(); // exibe log
doTake(); // exibe log
doTake(); // não exibe log

const doDebounce = debounceTime(500, () => console.log('Chamou callback'));
doDebounce(); // não exibe log
doDebounce(); // não exibe log
doDebounce(); // exibe log
```

Através da função `partialize` podemos criar uma nova função que por debaixo dos panos assumirá o primeiro parâmetro. Dessa maneira, a nova função receberá apenas o callback como parâmetro.

```
//
const partialTake = partialize(takeUntil, 2);
// só recebe agora um parâmetro, o callback com a lógica que será
// executada quando a função for chamada
const configuredPartialTake = partialTake(() => console.log('oi'));

// só recebe agora um parâmetro, o callback com a lógica que será
// executada quando a função for chamada
const partialDebounce = partialize(debounceTime, 500);

// partialDebounce recebe o take já configurado
```

```
// partialDebounce recebe o take já configurado
const configuredPartialDebounce = partialDebounce(configuredPartialTake);

configuredPartialDebounce();
configuredPartialDebounce();
configuredPartialDebounce(); // exibe oi uma vez apenas
```

Agora que já recapitulamos, vamos tentar compor as operacões:

```
// app/app.js
import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime, partialize, pipe } from './utils/operators.js';

const operations = pipe(
    partialize(debounceTime, 500),
    partialize(takeUntil, 3)
);

const action = operations(() =>
    service
    .sumItems('2143')
    .then(console.log)
    .catch(console.log)
);

document
.querySelector('#myButton')
.onclick = action;
```

A tentativa foi boa, mas infelzimente um teste demonstra rapidamente que a operação não foi executada como esperado. Qual o motivo?

Um olhar atento perceberá que a operação `takeUntil` foi aplicada antes de `debounceTime`. Se isso é realmente verdade, se invertemos a ordem das funções passadas para `pipe` resolverá o problema? Vamos tentar:

```
// app/app.js
// código anterior omitido

// inverteu a ordem
const operations = pipe(
    partialize(takeUntil, 3),
    partialize(debounceTime, 500)
);
// código posterior omitido
```

Realmente, a aplicação volta a funcionar como esperado. Mas por que tivemos que inverter a ordem? Vamos escrutinar mais uma vez a função `pipe`:

```
// app/utils/operators.js
// código anterior omitido
export const pipe = (...fns) => value =>
```

```
        fns.reduce((previousValue, fn) =>
            fn(previousValue), value)
    // código posterior omitido
```

A função `pipe` retorna uma função que ao ser chamada invocará cada função recebida como parâmetro da direta para esquerda, passando o resultado da função anterior para a próxima e assim sucessivamente retornando no final o último resultado. No caso, na primeira iteração teremos o retorno de `debounceTime` . Seu retorno será o callback de `takeUntil` e a função configurada de `takeUntil` será o retorno da função. Não é isso que queremos, queremos exatamente o contrário, por isso foi necessário inverter a ordem.

Esse detalhe é necessário porque por debaixo dos panos estamos utilizando a função `pipe` para encadear chamadas de callback. É uma forçada de barra, mas nem por isso deixa de ser elegante.