02

Lidando com dados nulos

Transcrição

Vamos criar gradativamente a solução para o problema. Dentro de de `app/utils` , vamos criar o módulo `maybe.js` .

A palavra "maybe" significa "talvez" em português. Pelo menos o nome do nosso módulo faz sentido, porque talvez tenhamos um valor, talvez não.

Vamos declarar a classe `Maybe` que vai encapsular um valor que potencialmente pode ser `null` ou `undefined` . A ideia é que ela nos blinde de acessar esses valores:

```
export class Maybe {

    constructor(value) {

        this._value = value;
    }
}
```

Em `app/app.js` vamos realizar alguns testes:

```
app/app.js
// importações omitidas
import { Maybe } from './utils/maybe.js';

const maybe1 = new Maybe(10);
const maybe2 = new Maybe(null);
```

Criamos duas instâncias de `Maybe` . Todavia, ficar utilizando o operador `new` não é muito interessante, pois o programador pode esquecer de utilizá-lo o que resultará em um erro em tempo de execução, além de deixar um pouquinho verboso nosso código. Vamos utilizar outra abordagem adicionando um método estático à classe chamado `of` . É este método que receberá o valor que será encapsulado e também será o responsável em invocar o operador `new` , isto é, será o responsável em criar a instância de `Maybe` :

```
export class Maybe {

    constructor(value) {

        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }
}
```

Utilizando o novo método em `app/app.js`. Vamos deixar apenas o primeiro `Maybe`, pois um apenas é suficiente para avançarmos com nosso entendimento:

```
app/app.js
// importações omitidas
import { Maybe } from './utils/maybe.js';

const maybe = Maybe.of(10);
```

Nosso `Maybe` tem como objetivo nos blindar de acessar um valor `null` ou `undefined`. Para isso, ele precisa ter algum método que nos indique se há valor ou não. Vamos criar o método `isNothing`:

```
export class Maybe {

    constructor(value) {

        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }

    isNothing() {
        return this._value === null || this._value === undefined;
    }
}
```

Usando o novo método:

```
app/app.js
// importações omitidas
import { Maybe } from './utils/maybe.js';

const maybe = Maybe.of(10);
if(!maybe.isNothing()) {
    // acessa o valor, mas como se ele esta encapsulado?
}
```

A abordagem anterior não esta muito legal porque estamos, em primeiro lugar, fazendo um `if` e cairemos no problema anterior de termos que ficar adicionando várias condições `if` ao acessar o valor em todos os lugares que ele for acessado. Em segundo lugar, se quisermos acessar o dado do `Maybe` precisaremos acessar a propriedade `_value`. Todavia, sabemos que propriedades e métodos prefixados com *underline* não devem ser acessados fora da definição da classe.

```
app/app.js
// importações omitidas
import { Maybe } from './utils/maybe.js';

const maybe = Maybe.of(10);
if(!maybe.isNothing()) {
    // não deveria fazer isso!
```

```
    alert(maybe._value);
}
```

E agora? Bem, se nosso `Maybe` é um *wrapper* que embrulha/encapsula um valor, podemos realizar uma operação `map`. O que é uma operação `map`? É pegar um dado encapsulado ou armazenado e aplicar uma lógica de transformação nesse dado. Lembrando do `Array.map`? É a mesma lógica, o `Array` encapsula uma lista de dados e aplica um `map` sobre esses dados realizando uma transformação. Em nosso caso, nosso `Maybe` encapsula um valor apenas e através de um `map` podemos realizar transformações neste valor:

```
export class Maybe {

    constructor(value) {

        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }

    isNothing() {
        return this._value === null || this._value === undefined;
    }

    map(fn) {
        if(this.isNothing()) return Maybe.of(null);
        const value = fn(this._value);
        return Maybe.of(value);
    }
}
```

Ou podemos simplificar o método `map` da seguinte maneira:

```
map(fn) {
    if(this.isNothing()) return Maybe.of(null);
    return Maybe.of(fn(this._value));
}
```

Como vai funcionar? Vejamos em `app/app.js`:

```
app/app.js
// importações omitidas
import { Maybe } from './utils/maybe.js';

const maybe = Maybe
    .of(10)
    .map(value => value + 10)
    .map(value => value + 30)
```

A função `map` nos dá acesso ao valor encapsulado pelo `Maybe`. Pegamos esse valor e somamos com `10`. O retorno será um novo `Maybe` que encapsula agora o valor `20`. Como o retorno é um novo `Maybe`, podemos encadear uma chamada à função

`map` mais uma vez. No final, teremos como resultado um `Maybe` que encapsula o valor transformado do `Maybe` original. Que será `50`.

E se passarmos `null`? Não teremos erros nas operações `map` pois elas serão ignoradas e no final teremos um `Maybe` com o valor `null` encapsulado:

app/app.js
```
// importações omitidas
import { Maybe } from './utils/maybe.js';

const maybe = Maybe
    .of(null)
    // Não realizará os maps abaixo, simplesmente retornará um novo
    // novo Maybe que ecapsula null
    .map(value => value + 10)
    .map(value => value + 30)
```

Mas como teremos acesso ao valor encapsulado pelo `Maybe`? Ele continua encapsulado. Vamos criar o método `get` que retornará o valor encapsulado:

```
export class Maybe {

    constructor(value) {

        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }

    isNothing() {
        return this._value === null || this._value === undefined;
    }

    map(fn) {
        if(this.isNothing()) return Maybe.of(null);
        const value = fn(this._value);
        return Maybe.of(value);

    }

    get() {
        return this._value;
    }
}
```

app/app.js
```
// importações omitidas
import { Maybe } from './utils/maybe.js';

const resultado = Maybe
    .of(10)
    .map(value => value + 10)
```

```
    .map(value => value + 30)
    .get(); // retorna 50!
```

Excelente, mas pode ficar ainda melhor. Se o valor no final for `null` ? Será que podemos fornecer um valor padrão para ser utilizado no lugar de `null` ?

```
export class Maybe {

    constructor(value) {

        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }

    isNothing() {
        return this._value === null || this._value === undefined;
    }

    map(fn) {
        if(this.isNothing()) return Maybe.of(null);
        const value = fn(this._value);
        return Maybe.of(value);
    }

    getOrElse(value) {
        if(this.isNothing()) return value;
        return this._value;
    }
}
```

Testando:

```
app/app.js
// importações omitidas
import { Maybe } from './utils/maybe.js';

const resultado = Maybe
    .of(null)
    .map(value => value + 10)
    .map(value => value + 30)
    .getOrElse(0); // retorna 0
```

Outro teste:

```
app/app.js
// importações omitidas
import { Maybe } from './utils/maybe.js';

const resultado = Maybe
    .of(10)
```

```
        .map(value => value + 10)
        .map(value => value + 30)
        .getOrElse(0); // retorna 50
```

Ótimo!