Artigo

React hoje e amanhã. O que muda com os Hooks?

Em outubro de 2018 Dan Abramov, engenheiro da equipe de desenvolvimento do React, anunciou os Hooks, desde então essa nova feature tem sido a funcionalidade mais aguardada na comunidade React.

Desde a introdução de classes à sintaxe da linguagem JavaScript, o React incorporou os class components como sendo a sua principal forma de criar componentes com

possui algumas desvantagens, por exemplo, o funcionamento do this em classes JavaScript é pouco intuitivo; a restrição de acesso aos ciclos de vida de um componente exclusivamente por métodos especiais, como componentDidMount. A **Listagem 1** mostra um exemplo de componente de classe.

```
import React, { Component } from 'react';

export default class Posts extends Component {
    constructor(props) {
        super(props);

        this.state = {
            titulo: '',
        };

        this.setTitulo = this.setTitulo.bind(this);
    }

    setTitulo(titulo) {
        this.setState({ titulo });
    }

    componentDidMount() {
        this.setTitulo('Home');
    }

    render() {
        return (
            <div>
                <input
                    type="text"
                    placeholder="titulo"
                    value={this.state.titulo}
                    onChange={event => this.setTitulo(event.target.val
```

```
32              </div>
33          );
34      }
35  }
```

Listagem 1. Componente de classe

Em outubro de 2018 o porta-voz da equipe de **desenvolvimento do React**, Dan Abramov, anunciou uma alternativa às classes para criar componentes que possuem estado interno e possuem lógica em seu ciclo de vida (componente é criado, atualizado, etc.): **os Hooks**. Esse novo recurso fornece a componentes funcionais um estado interno e acesso ao próprio ciclo de vida através de uma forma mais direta, limpa e autocontida.

Estado

Antes dos Hooks

O estado de um componente fica armazenado dentro de um objeto literal que é alocado no atributo state da classe do componente, que sofre alterações de acordo com o comportamento do componente. O valor de cada atributo desse objeto é refletido no template do componente de alguma maneira. Na **Listagem 2** vemos um componente de classe com estado.

```
1  import React, { Component } from 'react';
2
```

```
6
7           this.state = {
8               titulo: '',
9           };
10
11          this.setTitulo = this.setTitulo.bind(this);
12      }
13
14      setTitulo(titulo) {
15          this.setState({ titulo });
16      }
17
18      render() {
19          return (
20              <div>
21                  <input
22                      type="text"
23                      placeholder="titulo"
24                      value={this.state.titulo}
25                      onChange={event => this.setTitulo(event.target.val
26                  />
27                  <h3>{this.state.titulo}</h3>
28              </div>
29          );
30      }
31  }
```

Listagem 2. Componente com estado

No exemplo da **Listagem 2**, o atributo titulo de state é vinculado a um input e a um h3. Além disso, o método setTitulo altera o valor do atributo sempre que houver uma mudança no valor do input. Note, na linha 11, que é necessário executar o método `bind()` de `setTitulo` vinculando-o ao this da classe e não ao de seu próprio objeto

confuso e menos sustentável na medida em que o componente ganha mais complexidade em seu comportamento.

Utilizando Hooks

A **biblioteca React** fornece a função `useState` para componentes funcionais, que retornará um array com dois elementos, onde o primeiro é a constante que armazena aquele estado e o segundo é uma função para substituir o valor daquele estado. A **Listagem 3** mostra a **chamada de um Hook**.

```
1 import React, { useState } from 'react';
2
3 export default function Posts() {
4     const [titulo, setTitulo] = useState('');
5 }
```

Listagem 3. chamando o Hook useState

Podemos ver, na linha 4, um exemplo de useState sendo executada: a função recebe em seu parâmetro o valor inicial daquele estado e retorna o array mencionado. Por convenção utilizamos atribuição por desestruturação dos itens desse array em constantes com os nomes no padrão [estado, setEstado] para manter a clareza do que está sendo retornado.

A constante titulo inicialmente possui como valor uma string vazia e qualquer valor que for passado a setTitulo será atribuído à constante titulo.

O código da **Listagem 4** possui o mesmo comportamento do exemplo da **Listagem 2**

```jsx
import React, { useState } from 'react';

export default function Posts() {
    const [titulo, setTitulo] = useState('');

    return (
        <div>
            <input
                type="text"
                placeholder="titulo"
                value={titulo}
                onChange={event => setTitulo(event.target.value)}
            />
            <h3>{titulo}</h3>
        </div>
    );
}
```

Listagem 4. Componente funcional com Hook de estado

O código da **Listagem 4** possui o mesmo comportamento do exemplo da **Listagem 2** com um *class component* mas notavelmente com muito menos código.

Lifecycle sem Hooks

Da forma antiga, utilizando class components o acesso ao ciclo de vida (momento em que o componente é carregado, por exemplo) é fornecido através dos métodos chamados lifecycle methods, cuja execução ocorre dado um determinado ciclo de vida do componente, como ilustra a **Listagem 5**.

```
import { loginUrl, logoutUrl } from '../helpers/urls';
import Main from './main';

export default class App extends Component {
    componentDidMount() {
        if (!localStorage.getItem('NOME_CHAVE_TOKEN')) {
            fetch(loginUrl, { method: 'POST' })
                .then(response => response.json())
                .then(token => localStorage.setItem('NOME_CHAVE_TOKEN'
        }
    }

    componentDidUpdate() {
        if (!localStorage.getItem('NOME_CHAVE_TOKEN')) {
            fetch(loginUrl, { method: 'POST' })
                .then(response => response.json())
                .then(token => localStorage.setItem('NOME_CHAVE_TOKEN'
        }
    }

    componentWillUnmount() {
        fetch(logoutUrl, { method: 'POST' })
            .then(response => localStorage.removeItem('NOME_CHAVE_TOKE
    }

    render() {
        return <Main />;
    }
}

```

Listagem 5. Componente de classe com métodos `lifecycle`.

O componente acima é o primeiro a ser carregado pela aplicação e nele definimos efeitos colaterais em três ciclos de vida: quando ele é carregado, quando ele é

- **componentDidMount**: Executa quando o componente é carregado. Neste momento verificamos se o localstorage possui o atributo NOME_CHAVE_TOKEN, se não possuir, enviará uma requisição à URL de login para conseguir um token e irá armazená-lo no localstorage.
- **componentDidUpdate**: Executa quando o componente é atualizado. Repetimos o comportamento do lifecycle componentDidMount sempre que a página for atualizada.
- **componentWillUnmount**: Executa quando o componente começar a ser descarregado. Neste momento enviamos uma requisição de logout e a remoção do token do localstorage.

Lifecycle com Hooks

Em um componente funcional podemos, utilizando a função `useEffect()` **fornecida pelo React**, que recebe como parâmetro uma função callback e nela serão realizados os efeitos colaterais necessários pelo componente durante seu ciclo de vida. Observe o exemplo da **Listagem 6**.

```
1  import React, { useEffect } from 'react';
2
3  import { loginUrl, logoutUrl } from '../helpers/urls';
4  import Main from './main';
5
6  export default function AppFunc() {
7      useEffect(() => {
8          if (!localStorage.getItem('NOME_CHAVE_TOKEN')) {
9              fetch(loginUrl, { method: 'POST' })
10                 .then(response => response.json())
```

```
14          return () => {
15              fetch(logoutUrl, { method: 'POST' })
16                  .then(response => localStorage.removeItem('NOME_CHAVE_
17          };
18      });
19
20      return <Main />;
21  }
```

Listagem 6. Componente funcional com Hook useEffect

Opcionalmente, a função `useEffect` pode receber como parâmetro um array. Se este for vazio, a função de callback passada como parâmetro será executada quando o componente for montado (`componentDidMount`), mas não quando o mesmo for atualizado (`componentDidUpdate`). A função `useEffect` pode receber como parâmetro opcional um array: se estiver vazio a callback passada como parâmetro e `useEffect` será executada somente quando o componente for montado, mas não quando atualizado:

```
1  useEffect(() => {
2      if (!localStorage.getItem('NOME_CHAVE_TOKEN')) {
3          fetch(loginUrl, { method: 'POST' })
4              .then(response => response.json())
5              .then(token => localStorage.setItem('NOME_CHAVE_TOKEN', to
6      }
7  }, []);
```

O array passado como segundo parâmetro de useEffect também pode receber qualquer número de propriedades de componente (`props`) ou atributos de estado (state). Dessa forma, `useEffect` executará a callback passada em seu parâmetro sempre que esse(s) atributo(s) tiver(em) alguma alteração. A **Listagem 7** mostra esse uso dos Hooks.

```
1  const [titulo, setTitulo] = useState('');
2  useEffect(() => {
3      document.title = titulo;
4  }, [titulo]);
```

Listagem 7. Hook useEffect utilizando o estado

Hooks Customizados

Um dos recursos introduzidos são os chamados **Hooks customizados**, eles permitem o reaproveitamento do código de comportamento entre componentes. Toda chamada de Hook pode ser feita em uma função separada, contanto que esta mantenha o objeto React no escopo, no caso, a **função React**. Essa nova função se torna um Hook customizado e, por convenção, deve ter seu nome iniciado com use.

Para ilustrar a criação de um Hook customizado utilizaremos a **Listagem 8**.

```
1  import React, { useEffect } from 'react';
2
3  import { loginUrl, logoutUrl } from '../helpers/urls';
```

```
export default function AppFunc() {
    useEffect(() => {
        if (!localStorage.getItem('NOME_CHAVE_TOKEN')) {
            fetch(loginUrl, { method: 'POST' })
                .then(response => response.json())
                .then(token => localStorage.setItem('NOME_CHAVE_TOKEN'
        }

        return () => {
            fetch(logoutUrl, { method: 'POST' })
                .then(response => localStorage.removeItem('NOME_CHAVE_
        };
    });

    return <Main />;
}
```

Listagem 8. Hook useEffect

Nesse caso, podemos extrair um Hook customizado desse componente criando uma função que chama o Hook useEffect. Observe a **Listagem 9**.

```
import React, { useEffect } from 'react';

import { loginUrl, logoutUrl } from '../helpers/urls';
import Main from './main';

export default function AppFunc() {
    useTokenLocalStorage();

    return <Main />;
}
```

```
13          if (!localStorage.getItem('NOME_CHAVE_TOKEN')) {
14              fetch(loginUrl, { method: 'POST' })
15                  .then(response => response.json())
16                  .then(token => localStorage.setItem('NOME_CHAVE_TOKEN'
17          }
18
19          return () => {
20              fetch(logoutUrl, { method: 'POST' })
21                  .then(response => localStorage.removeItem('NOME_CHAVE_
22          };
23      });
24  }
25
```

Listagem 9. Hook

Regras dos Hooks

Para **utilizar os Hooks em componentes React** é preciso seguir duas regras específicas, caso contrário, não há como garantir a integridade do recurso na aplicação:

- **Chame os Hooks apenas no nível de cima:** nunca chame Hooks dentro de loops, estruturas condicionais ou funções `callback` aninhadas. Os Hooks devem ser chamados sempre no primeiro nível do componente ou função React em que eles serão utilizados. Isso se deve porque o React identifica os hooks de um componente pela ordem em que eles foram chamados. Seguindo essa regra, garantimos que os Hooks sejam chamados na mesma ordem sempre que o componente é renderizado, permitindo ao React preservar o estado dos Hooks apesar de múltiplas chamadas de useState e useEffect.

customizado, geralmente extraído de um componente funcional React.

Conclusão

Os Hooks estão entre os recursos mais aguardados do React desde a introdução dos componentes de classe. A expectativa é que, com a **utilização dos Hooks**, possamos simplificar componentes complexos e tornar o código mais entendível e sustentável. Apesar disso, o próprio Dan Abramov, co-criador do Redux, do Create-Ract-App e porta voz da equipe de desenvolvimento do React no Facebook, não recomenda que componentes em produção sejam reescritos, já que componentes que utilizam Hooks possuem 100% de retrocompatibilidade com componentes de classe. Dessa forma, o ideal é que apenas novos componentes sejam escritos utilizando este recurso.

Tecnologias:

JavaScript React

Por **Aylan**
Em 2019

Suporte ao aluno - Tire a sua dúvida.