Tipos de hooks

Não use Hooks dentro de loops, regras condicionais ou funções aninhadas (funções dentro de funções). Em vez disso, sempre use Hooks no nível superior de sua função React, antes de qualquer retorno antecipado. Seguindo essas regras, você garante que os Hooks serão chamados na mesma ordem a cada vez que o componente renderizar. Ou seja, não é possível usar hooks dentro de blocos.

**UseState** – fornece um estado ao componente. Já to cansado de saber e não preciso de exemplo.

**UseEffect** – O Effect Hook (Hook de Efeito) te permite executar efeitos colaterais em componentes funcionais. Ele age como um método de ciclo de vida, fazendo alguma coisa apenas quando um componente for montado ou desmontado e etc. É um ótimo hook para se fazer requisições http, devido a natureza assíncrona das requisições.

O useEffect é um Hook que serve para lidar com os efeitos. Podemos usá-los como os lifeCycles componentDidMount, componentDidUpdate e componentWillUnmount.

A beleza deste Hook é que podemos definir facilmente quando queremos que esta função seja executada novamente. Basta passar como segundo parâmetro ao useEffect() um Array com as variáveis que controlarão esse Hook.

E se quisermos que a função seja executada apenas uma vez, que é quando o componente é inserido na tela, basta passar um Array vazio.

Buscar dados, configurar uma subscription, e mudar o DOM manualmente dentro dos componentes React são exemplos de efeitos colaterais. Esteja você acostumado ou não a chamar essas operações de “efeitos colaterais” (ou somente “efeitos”), você provavelmente já usou eles em seus componentes antes.

Existem dois tipos comuns de efeitos colaterais nos componentes React: aqueles que não precisam de limpeza, e aqueles que precisam.

Usando esse Hook, você diz ao React que o componente precisa fazer algo apenas depois da renderização.

**import React, { useState, useEffect } from 'react';**

**function Exemplo() {**

**const [count, setCount] = useState(0);**

**// Similar ao componentDidMount e componentDidUpdate: useEffect(() => { // Atualiza o titulo do documento usando a API do browser document.title = `Você clicou ${count} vezes`; });**

**return (**

**<div>**

**<p>Você clicou {count} vezes</p>**

**<button onClick={() => setCount(count + 1)}>**

**Clique aqui**

**</button>**

**</div> ); }**

**UseCallback** – Essencialmente faz com que uma função só renderize uma vez. Exemplo: tenho uma função ligada a outra e toda vez que uma renderiza a outra também renderiza, mas não é isso que quero. Na função que não quero renderização, uso um call back. Ou seja, no fim das contas ele salva funções. Faz um cache de funções que não mudam.

UseCallback retornará uma versão memoizada do callback que só muda se uma das entradas tiver sido alterada. Isto é útil quando utilizamos callbacks a fim de otimizar componentes filhos, que dependem da igualdade de referência para evitar renderizações desnecessárias (como por exemplo shouldComponentUpdate).

Dentro Informática, memoização é uma otimização técnica usada principalmente para acelerar programas de computador armazenando os resultados de chamadas de função e retornar o resultado em cache quando as mesmas entradas ocorrerem novamente.

Exemplo no app.

import P from 'prop-types';

import './App.css';

import React, { useState, useEffect, useCallback } from 'react';

const Button = *React*.memo(function Button({ *incrementButton* }) {

  console.log('Filho, renderizou');

  return <button *onClick*={() => incrementButton(100)}>+</button>;

});

Button.propTypes = {

  incrementButton: P.func,

};

function App() {

  const [counter, setCounter] = useState(0);

  const incrementCounter = useCallback((*num*) => {

    setCounter((*c*) => *c* + *num*);

  }, []);

  console.log('Pai, renderizou');

  return (

    <div *className*="App">

      <p>Teste 3</p>

      <h1>C1: {counter}</h1>

      <*Button* *incrementButton*={incrementCounter} />

    </div>

  );

}

export default App;

**UseMemo** – Faz a mesma coisa do call back, mas ao invés de salvar funções salva valores.

**UseRef** – Basicamente pega uma referência no documento ou algum valor. Ele cria uma referencia de memoria e não renderiza este valor.

useRef retorna um objeto ref mutável, no qual a propriedade .current é inicializada para o argumento passado (initialValue). O objeto retornado persistirá durante todo o ciclo de vida do componente.

Um caso comum de uso é o acesso imperativamente a um componente filho:

function TextInputWithFocusButton() {

const inputEl = useRef(null);

const onButtonClick = () => {

// `current` aponta para o evento de `focus` gerado pelo campo de texto

inputEl.current.focus();

};

return (

<>

<input ref={inputEl} type="text" />

<button onClick={onButtonClick}>Focus no input</button>

</>

);

}

Essencialmente, useRef é como uma “caixa” que pode conter um valor mutável em sua propriedade .current.

**UseContext** - Sem utilizar Contexto, temos que passar props para todos os componentes na árvore de componentes para chegar onde é necessário, que é o componente que vai usar essas informações.

Portanto, Contexto fornece uma maneira de passar dados pela árvore de componentes sem precisar passar props manualmente em todos os níveis.

Substitui um pouco o redux, pois serve para o gerenciamento dos estados dos componentes.

import { useContext, createContext, useState } from 'react';

import './App.css';

const globalState = {

  title: 'O título que contexto',

  body: 'O body do contexto',

  counter: 0,

};

const GlobalContext = createContext();

// eslint-disable-next-line

const Div = ({ *children* }) => {

  return (

    <>

      <*H1* />

      <*P* />

    </>

  );

};

// eslint-disable-next-line

const H1 = () => {

  const theContext = useContext(GlobalContext);

  const {

*contextState*: { title, counter },

  } = theContext;

  return (

    <h1>

      {title} {counter}

    </h1>

  );

};

// eslint-disable-next-line

const P = () => {

  const theContext = useContext(GlobalContext);

  const {

*contextState*: { body, counter },

    setContextState,

  } = theContext;

  return (

    <p

*onClick*={() => setContextState((*s*) => ({ ...*s*, counter: *s*.counter + 1 }))}

    >

      {body} {counter}

    </p>

  );

};

function App() {

  const [contextState, setContextState] = useState(globalState);

  return (

    <*GlobalContext.Provider* *value*={{ contextState, setContextState }}>

      <*Div* />

    </*GlobalContext.Provider*>

  );

}

export default App;

UseReducer - Uma alternativa para [useState](https://pt-br.reactjs.org/docs/hooks-reference.html" \l "usestate)

useReducer é geralmente preferível em relação ao useState quando se tem uma lógica de estado complexa que envolve múltiplos sub-valores, ou quando o próximo estado depende do estado anterior. useReducer também possibilita a otimização da performance de componentes que disparam atualizações profundas porque [é possível passar o dispatch para baixo, ao invés de callbacks](https://pt-br.reactjs.org/docs/hooks-faq.html#how-to-avoid-passing-callbacks-down).

Aqui está o exemplo do contador na seção [useState](https://pt-br.reactjs.org/docs/hooks-reference.html" \l "usestate), reescrito para usar um reducer:

import { useReducer } from 'react';

import './App.css';

const initialState = {count: 0}

function reducer(*state*, *action*) {

  switch (*action*.type) {

    case 'increment':

      return {count: *state*.count + 1};

    case 'decrement':

      return {count: *state*.count - 1};

    default:

      throw **new** *Error*();

  }

}

function App () {

  const [state, dispatch] = useReducer(reducer, initialState);

  return (

    <div>

      Count: {state.count}

      <button *onClick*={() => dispatch({type: 'increment'})}>add</button>

      <button *onClick*={() => dispatch({type: 'decrement'})}>sub</button>

    </div>

  )

}

export default App;

**useLayoutEffect** – pouco usado e normalmente você só vai precisar dele quando seu código estiver errado. Ele possibilita uma manipulação direta do DOM.

bouderingErros – não é um hook mas está nessa seção. É um mecanismo do react ao qual se um componente quebrar ele dá erro apenas no componente e não na página inteira.