**React Hooks 简介**

React Hooks 是对 React function 组件的一种扩展，通过一些特殊的函数，让无状态组件拥有状态组件才拥有的能力。

Hooks 是 React 函数组件中的一类特殊函数，通常以 use 开头，比如 useRef，useState，useReducer 等。通常在我们写 React 组件的时候，如果这个组件比较复杂，拥有自己的生命周期或者 state，就将其写成 class 组件；如果这个组件仅仅用来展示，就将其写成 function 组件。

React Hooks 使用 function 组件的写法，通过 useState 这样的 API 解决了 function 组件没有 state 的问题，通过 useEffect 来解决生命周期的问题，通过自定义 hooks 来复用业务逻辑。

**Hooks 解决哪些问题**

* 复用与状态有关的逻辑，之前引申出来 HOC 的概念，但是 HOC 会导致组件树的臃肿。
* 解决组件随着业务扩展变得难以维护的问题。
* 使用更容易理解并且对初学者更友好的 function 组件。

**用法**

Hooks 主要分三种：

* State hooks: 允许开发者在 function 组件中使用 state。
* Effect hooks: 允许开发者在 function 组件中使用生命周期和 side effect。
* Custom hooks: 自定义 hooks，可以在里面使用 State Hooks 和 Effect Hooks，达到组件之间逻辑复用。

**State Hooks**

看一下官方给出的 demo

import { useState } from 'react';

function Example() {

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

return (

<div>

<p>You clicked {count} times</p>

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

Click me

</button>

</div>

);

}

复制

这里的 useState 就是一个 hook，返回一个数组，第一个 count 表示一个 state，默认值为 0；第二个 setCount 相当于 class function 中的 setState，表示对 count 的更新操作。

这样写的好处是每个 state 独立管理，避免状态复杂的时候 state 臃肿。

基本用法描述如下：

const [state, setState] = useState(initialState);

setState(newState);

复制

useState 返回一个数组，第一个值是一个 stateful（有状态）的值，第二个值是更新这个状态值的函数。在初始渲染的时候，返回的 state 与 initialState 相同，在后续重新渲染时，useState 返回的第一个值将始终是应用更新后的最新 state(状态) 。

setState 函数用于更新 state(状态) ，它接受一个新的 state(状态) 值，并将组件排入重新渲染的队列。

由于 setState 使用函数式的更新方式，所以可以传递函数给 setState，该函数将接收先前的值，并返回更新的值。

function Counter({initialCount}) {

const [count, setCount] = useState(initialCount);

return (

<>

Count: {count}

<button onClick={() => setCount(0)}>Reset</button>

<button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>

<button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>

</>

);

}

复制

上述代码可以使用上次的 state 来计算新的 state。与 React 类组件中的 setState 不同，useState 不会自动合并更新对象。所以如果要更新的 state 依赖前一个 state 的时候，需要使用对象扩展的方式：

setState(prevState => {

// Object.assign 也是可行的

return {...prevState, ...updatedValues};

});

复制

initialState 参数既可以是一个值，也可以是一个函数，如果初始状态是高开销的计算结果，则可以改为提供函数，该函数仅在初始渲染时执行：

const [state, setState] = useState(() => {

const initialState = someExpensiveComputation(props);

return initialState;

});

复制

initialState 参数只有在初始渲染期间才会使用，在随后的渲染中，它会被忽略。

**Effect Hooks**

Effect Hooks 允许在组件中执行副作用（side effects），类似于类中的生命周期方法。通常我们需要在 componentDidMount 和 componentDidUpdate 写一些操作，可能是更新数据，也可能是更新 Dom。除此之外，我们还会在 componentWillUnmount 的时候解绑一些事件监听防止内存泄露。这些都导致了组件维护成本的增大。而在 function 组件中，又没有这些生命周期，因此 Hooks 使用 Effect Hooks 来取代这些生命周期，完成一部分能力。

看一下官方给出的动态更改 title 的 demo：

import { useState, useEffect } from 'react';

function Example() {

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

// Similar to componentDidMount and componentDidUpdate:

useEffect(() => {

// Update the document title using the browser API

document.title = `You clicked ${count} times`;

});

return (

<div>

<p>You clicked {count} times</p>

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

Click me

</button>

</div>

);

}

复制

在 useEffect 之前，我们需要在 componentDidMount 和 componentDidUpdate 中同时去调用更改 title 的方法，以完成组件初始化的状态和数据更新的状态。useEffect 传递一个函数给 React，React 在组件渲染完成后和更新后调用这个函数来完成上述功能。默认情况下，它在第一次渲染之后和每次更新之后都运行。

可以将 useEffect Hook 视为 componentDidMount，componentDidUpdate 和 componentWillUnmount 的组合。

那 useEffect 什么时候执行 componentWillUnmount 的操作呢？

如果 useEffect 中返回一个函数，在 React 卸载当前的组件的时候，会执行这个函数，用于清理 effect。

对比需要清理 effect 和不需要清理 effect 的两种写法：

function FriendStatusWithCounter(props) {

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

useEffect(() => {

document.title = `You clicked ${count} times`;

});

const [isOnline, setIsOnline] = useState(null);

useEffect(() => {

ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

return () => {

ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);

};

});

function handleStatusChange(status) {

setIsOnline(status.isOnline);

}

return (/\*...\*/);

}

复制

通过跳过 Effect 来优化性能。

通常，每次组件渲染或者更新都去执行某些逻辑会带来无谓的消耗，所以我们经常会写这样的代码：

componentDidUpdate(prevProps, prevState) {

if (prevState.count !== this.state.count) {

document.title = `You clicked ${this.state.count} times`;

}

}

复制

只有组件更新前后的 state.count 发生变化的时候，才去更新 title。

用 Hooks 可以更简单地处理这个问题

useEffect(() => {

document.title = `You clicked ${count} times`;

}, [count]); // Only re-run the effect if count changes

复制

给 useEffect 传入第二个参数，这个参数是一个数组。如果组件重新渲染，只有这个 count 发生变化的时候 React 才会执行函数 中的内容，否则会直接跳过这个 effect。如果数组中是多个参数，那么只要其中一个发生变化，React 都会执行函数中的内容。

这也适用于具有清理阶段的 effect ：

useEffect(() => {

ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

return () => {

ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);

};

}, [props.friend.id]); // Only re-subscribe if props.friend.id changes

复制

如果希望 effect 只在组件 componentDidMount 和 componentWillUnmount 的时候执行，则只需要给第二个参数传一个空数组即可。传入一个空数组 [] 输入告诉 React 你的 effect 不依赖于组件中的任何值，因此该 effect 仅在 mount 时运行，并且在 unmount 时执行清理，从不在更新时运行。

**Hooks 的规则**

React Hooks 其实不仅仅是功能层面的增强，也给 React 注入了新的软件思想。这就是最近几年开始流行的 “约定大于配置”，比如 Hooks 函数必须使用 use 开头，还有接下来要讲的规则。前面在我的文章 [webpack4 新特性](https://cloud.tencent.com/developer/tools/blog-entry?target=https%3A%2F%2Flz5z.com%2Fwebpack4-new%2F) 也提到了这个内容。

**只在顶层调用 Hooks**

Hooks 只能在顶层调用，不要在循环，条件或嵌套函数中调用 Hook。原因是 React 需要保证每次组件渲染的时候都以相同的顺序调用 Hooks。

假如一个组件中有多个 Hooks，React 如何知道哪个 state(状态) 对应于哪个 useState 调用呢？答案是 React 依赖于调用 Hooks 的顺序。本质上来说 Hooks 就是数组（[React hooks: not magic, just arrays](https://cloud.tencent.com/developer/tools/blog-entry?target=https%3A%2F%2Fmedium.com%2F%40ryardley%2Freact-hooks-not-magic-just-arrays-cd4f1857236e)）。每次执行 useState 都会改变下标，如果 useState 被包裹在 condition 中，那每次执行的下标就可能对不上，导致 useState 更新错数据。

**只能在 React Function 中调用 Hooks**

Hooks 只能在 React function 组件中调用，或者在自定义 Hooks 中调用。通过遵循此规则，可以确保组件中的所有 stateful （有状态）逻辑在其源代码中清晰可见。

**eslint**

[eslint-plugin-react-hooks](https://cloud.tencent.com/developer/tools/blog-entry?target=https%3A%2F%2Fwww.npmjs.com%2Fpackage%2Feslint-plugin-react-hooks) 可以保证强制执行上述两个规则。

$ npm install [email protected]

复制

// Your ESLint configuration

{

"plugins": [

// ...

"react-hooks"

],

"rules": {

// ...

"react-hooks/rules-of-hooks": "error"

}

}

复制

**自定义 Hooks**

自定义 Hooks 就是将组件之间需要共有的逻辑抽出来写成单独的函数。与一般的函数的区别是，自定义 Hooks 是一个以 use 开头的函数，内部可以调用其它的 Hooks。

import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {

const [isOnline, setIsOnline] = useState(null);

function handleStatusChange(status) {

setIsOnline(status.isOnline);

}

useEffect(() => {

ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);

return () => {

ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);

};

});

return isOnline;

}

export useFriendStatus;

复制

在另外一个组件中，将其引入后，就可以使用了

import {useFriendStatus} from 'hooks/xxx.js';

function FriendListItem(props) {

const isOnline = useFriendStatus(props.friend.id);

return (

<li style={{ color: isOnline ? 'green' : 'black' }}>

{props.friend.name}

</li>

);

}

复制

可以看出，自定义 Hooks 就是一个 JavaScript 函数而已，并没有什么特别。不过需要注意的是，自定义 Hooks 函数也必须以 use 开头（规约优先）。

**useContext**

const context = useContext(Context);

复制

接受一个 context（上下文）对象（从 React.createContext 返回的值）并返回当前 context 值，当提供程序更新时，此 Hook 将使用最新的 context 值触发重新渲染。

**useReducer**

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

复制

useReducer 可以理解为 Redux 的 Hooks，接受的第一个参数是 (state, action) => newState 的 reducer，并返回与 dispatch 方法配对的当前状态。

const initialState = {count: 0};

function reducer(state, action) {

switch (action.type) {

case 'reset':

return initialState;

case 'increment':

return {count: state.count + 1};

case 'decrement':

return {count: state.count - 1};

default:

// A reducer must always return a valid state.

// Alternatively you can throw an error if an invalid action is dispatched.

return state;

}

}

function Counter({initialCount}) {

const [state, dispatch] = useReducer(reducer, {count: initialCount});

return (

<>

Count: {state.count}

<button onClick={() => dispatch({type: 'reset'})}>

Reset

</button>

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

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

</>

);

}

复制

useReducer 接受可选的第三个参数 initialAction，表示在组件初始化期间执行的操作。比如利用 props 传递的值来初始化 state 的操作。

const initialState = {count: 0};

function reducer(state, action) {

switch (action.type) {

case 'reset':

return {count: action.payload};

case 'increment':

return {count: state.count + 1};

case 'decrement':

return {count: state.count - 1};

default:

// A reducer must always return a valid state.

// Alternatively you can throw an error if an invalid action is dispatched.

return state;

}

}

function Counter({initialCount}) {

const [state, dispatch] = useReducer(

reducer,

initialState,

{type: 'reset', payload: initialCount},

);

return (

<>

Count: {state.count}

<button

onClick={() => dispatch({type: 'reset', payload: initialCount})}>

Reset

</button>

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

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

</>

);

}

复制

**useRef**

const refContainer = useRef(initialValue);

复制

useRef 返回一个可变的 ref 对象，通过 .current 属性对其进行访问，返回的对象将存留在整个组件的生命周期中。

function TextInputWithFocusButton() {

const inputEl = useRef(null);

const onButtonClick = () => {

// `current` points to the mounted text input element

inputEl.current.focus();

};

return (

<>

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

<button onClick={onButtonClick}>Focus the input</button>

</>

);

}

复制

**useImperativeMethods**

useImperativeMethods(ref, createInstance, [inputs]);

复制

useImperativeMethods 与 forwardRef 共同使用，表示强制方法。通过 ref 将子组件的某个方法暴露给父组件。

子组件：

function FancyInput(props, ref) {

const inputRef = useRef();

useImperativeMethods(ref, () => ({

focus: () => {

inputRef.current.focus();

}

}));

return <input ref={inputRef} ... />;

}

FancyInput = forwardRef(FancyInput);

复制

父组件：

function FancyParent() {

const fancyInputRef = useRef(null);

useEffect(() => {

fancyInputRef.current.focus();

});

return (

<FancyInput ref={fancyInputRef} />

);

}

复制

**useLayoutEffect**

用法与 useEffect 相同，但在所有 DOM 变化后同步触发。使用它来从 DOM 读取布局并同步重新渲染。 在浏览器绘制之前 useLayoutEffect 将同步刷新。

useEffect 中的函数会在 layout(布局) 和 paint(绘制) 后触发。 这使得它适用于许多常见的 side effects ，例如设置订阅和事件处理程序，因为大多数类型的工作不应阻止浏览器更新屏幕。

但是如果 effect 不能够推迟，比如要 DOM 改变必须在下一次绘制之前同步触发，使用 useLayoutEffect 会更加合适。

**Hooks API**

参考 [Hooks API Reference](https://cloud.tencent.com/developer/tools/blog-entry?target=https%3A%2F%2Freactjs.org%2Fdocs%2Fhooks-reference.html)

**总结**

Hooks 通过设定某些特殊函数，在 React 组件内部“钩住”其生命周期和 state，帮助开发者解决一些逻辑复用的问题，通过自定义的 Hooks 对代码进行抽象，让我们写出更加符合函数式编程的规范，同时也减少了层层嵌套带来的问题。

**参考文档**

* [精读《React Hooks》](https://cloud.tencent.com/developer/tools/blog-entry?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5be8d3def265da611a476231)
* [React docs - Introducing HooksReact docs - Introducing HooksReact docs - Introducing HooksReact docs - Introducing Hooks](https://cloud.tencent.com/developer/tools/blog-entry?target=https%3A%2F%2Freactjs.org%2Fdocs%2Fhooks-intro.html)