



Lesson 13: Creating your own hooks

🕒 Created March 22, 2022 9:55 AM

🏷️ Tags Empty

📌 Related to Untitle... Empty



Sáng tạo các tính năng thú vị với việc tự tạo ra các hooks và tái sử dụng chúng

Từ phiên bản 16.8, React giới thiệu thêm tính năng hooks. Các hooks cho phép việc tái sử dụng các logic khác nhau ở nhiều component. Ngoài các hooks đã có sẵn trong React như `useState`, `useEffect`, `useMemo`, `useCallback`, `useRef`, `useContext`, ... React cho phép người dùng có thể tự định nghĩa ra các *custom hooks* để phục vụ một logic nhất định trong ứng dụng.

1. Quy tắc với hooks:

Khi làm việc với các hooks trong React, chúng ta cần ghi nhớ các quy tắc sau với chúng:

- **Hooks phải được gọi ở level ngoài cùng trong component.** Không sử dụng hooks bên trong các function, vòng lặp, câu điều kiện của component.

```
// Correct
const App = () => {
  const [state, setState] = useState();
  return <div></div>
}
// Wrong
const App = () => {
  const handleClick = () => {
    const [state, setState] = useState();
  }
  return <div></div>
}
```

- **Chỉ sử dụng hooks với các React component hoặc bên trong một hooks khác.** Không sử dụng hooks bên trong các function thông thường
- Các hooks cần được đặt tên bắt đầu với từ khoá `use`

2. Custom hooks:

`useInput`

Thông thường, để lấy điều khiển giá trị bên trong các `input` với React, ta có thể làm như sau:

```
import { useState } from "react";
const App = () => {
  const [input1, setInput1] = useState("");
  const [input2, setInput2] = useState("");
  const onChangeInput1 = (event) => {
    setInput1(event.target.value);
  };
  const onChangeInput2 = (event) => {
    setInput2(event.target.value);
  };
  return (
    <div>
      <input type="text" value={input1} onChange={onChangeInput1} />
      <input type="text" value={input2} onChange={onChangeInput2} />
    </div>
  );
};
```

Tuy nhiên, trong nhiều trường hợp, chúng ta sẽ có thể cần nhiều hơn các thẻ `input`. Điều này nhanh chóng làm code bị trùng lặp nhiều phần không cần thiết.


Giải pháp ở đây là chúng ta sẽ đóng góp logic liên quan tới các thẻ `input` thành một hooks riêng.

```
const useInput = () => { const [value, setValue] = useState(""); const onChange = (event) => { setValue(event.target.value); }; return { value: value, onChange: onChange }; };
```

Với việc logic được đóng gói bên trong một hook, chúng ta có thể dễ dàng tái sử dụng logic trên với nhiều thẻ `input` khác nhau. `useInput` trả về dữ liệu là một object. Trong đó, `value` là một state và `onChange` là một function xử lý event thay đổi của input.

```
const App = () => { const input1 = useInput(); const input2 = useInput(); return ( <div> <input type="text" value={input1.value} onChange={input1.onChange} /> <input type="text" value={input2.value} onChange={input2.onChange} /> </div> ); };
```

Code đã ngắn gọn hơn rất nhiều với việc sử dụng `useInput` hook

 Hãy thử tạo ra các hooks để xử lý các trường hợp input là checkbox, select, radio button, ...

useHover

Để theo dõi một thẻ HTML có đang được hover hay không, thông thường với React, ta làm như sau:

```
const App = () => { const [hover, setHover] = useState(false); const onMouseEnter = () => { setHover(true); }; const onMouseLeave = () => { setHover(false); }; return ( <div> <div style={{ width: 100, height: 100, border: "1px solid black" }} onMouseEnter={onMouseEnter} onMouseLeave={onMouseLeave} > {hover ? "Hovering" : "Not hovering"} </div> </div> ); };
```

Để tái sử dụng logic trên, chúng ta có thể tạo ra một custom hook để xử lý hành vi trên như sau:

```
import {useState, useRef} from 'react' const useHover = () => { const [isHover, setHover] = useState(false); const elemRef = useRef(); useEffect(() => { const elem = elemRef.current; const handleMouseenter = () => { setHover(true); }; const handleMouseleave = () => { setHover(false); }; elem.addEventListener("mouseenter", handleMouseenter); elem.addEventListener("mouseleave", handleMouseleave); return () => { elem.removeEventListener("mouseenter", handleMouseenter); elem.removeEventListener("mouseleave", handleMouseleave); }; }, []); return [elemRef, isHover]; }; const App = () => { const [elemRef, isHover] = useHover(); return ( <div> <div ref={elemRef} style={{ width: 100, height: 100, border: "1px solid black" }} > {isHover ? "Hovering" : "Not hovering"} </div> </div> ); };
```

useLocalStorage

Một cách đơn giản để chúng ta có thể đồng bộ giá trị state với giá trị bên trong LocalStorage

```
const App = () => { const [count, setCount] = useState(0); useEffect(() => {  
// load in the beginning setCount(Number(localStorage.getItem("count"))); },  
[]); useEffect(() => { // Save value after every changes  
localStorage.setItem("count", count); }, [count]); return ( <div> {count}  
<button onClick={() => { setCount(count + 1); }} > Increase </button> </div>  
</div>);
```

Để đồng bộ giữa LocalStorage và state, ta thường cần phải sử dụng các effect như trên để thực hiện load dữ liệu và lưu dữ liệu

Với custom hooks, mọi thứ trở nên đơn giản hơn khá nhiều:

```
import { useState, useEffect } from "react"; const App = () => { const [count,  
setCount] = useLocalStorage("count"); return ( <div> {count} <button onClick=  
{() => { setCount(count + 1); }} > Increase </button> </div> ); }; const  
useLocalStorage = (name) => { const [value, setValue] = useState(0);  
useEffect(() => { setValue(Number(localStorage.getItem(name))); }, [name]);  
useEffect(() => { localStorage.setItem(name, value); }, [value, name]); return  
[value, setValue]; };
```

Vì custom hooks là một function , chúng ta có thể dễ dàng truyền vào cho nó các tham số để chỉnh sửa các hành vi trong custom hooks

3. Một vài hooks hữu dụng khác:

Đây là danh sách một vài hooks có thể được sử dụng phổ biến:

- `useDebounce` : Sử dụng để debounce một value theo một khoảng thời gian
- `usePrevious` : Sử dụng để lưu lại giá trị cũ của một giá trị (state, props, ...) trong component

Các thư viện tổng hợp nhiều hooks:

- <https://usehooks.com/>
- [React Hooks Lib](#)
- [react-hanger](#)
- [React hookedUp](#)
- [react-use](#)
- [React Recipes](#)