Lesson 12: Other React hooks

Created	March 22, 2022 9:55 AM
Tags	Empty
Related to Untitle...	Empty

💡 Giới thiệu thêm các hook cần biết trong React.

1. `useRef`

Trong nhiều trường hợp, chúng ta cần truy cập tới các phần tử HTML của component. Xét ví dụ sau:

```
const App = () => { const handleFocus = () => {
document.getElementById("myTxt").focus() } return ( <div> <input id="myTxt"
type="text" /> <button onClick={handleFocus}>Focus</button> </div> ) }
```

Khi click vào button “Focus”, ta muốn con trỏ chuột sẽ được “focus” vào phần tử input để người dùng có thể nhập liệu. Với cách làm thông thường, ta sử dụng một giá trị id cho `input` và tiến hành sử dụng các function mặc định để truy vấn tới thẻ `input` đó.

Tuy nhiên, giả sử chúng ta có nhiều hơn các component `App` cần được render, thì các `input` sẽ có trùng một id. Điều này dẫn tới button “Focus” không hoạt động theo mong muốn.

React cung cấp một giải pháp để có thể truy cập đến các phần tử “HTML” trong component với `useRef`. Ứng dụng trên có thể viết lại như sau:

```
import {useRef} from 'react' const App = () => { const inputRef = useRef(null)
const handleFocus = () => { inputRef.current.focus() } return ( <div> <input
ref={inputRef} type="text" /> <button onClick={handleFocus}>Focus</button>
</div> ) }
```

`useRef` là một hook trong React. Ta khởi tạo một object React Reference là `inputRef` trong component `App`. Tất cả các thẻ HTML trong JSX đều có một thuộc tính đặc biệt là `ref` . Thuộc tính này sẽ nhận giá trị là một React Reference. Khi này, ta có thể truy cập tới thẻ HTML thông qua `ref.current`

`inputRef` trong ví dụ trên là độc lập giữa các instance của component `App`. Giả sử trong trường hợp chúng ta có nhiều hơn một component `App` xuất hiện trên màn hình, mỗi một component đó sẽ có một giá trị `inputRef` riêng để tham chiếu tới thẻ `input` của nó.

📖 Chỉ có các thẻ HTML có sẵn của JSX là có thể sử dụng thuộc tính `ref` này. Các component do user định nghĩa sẽ coi `ref` giống như các giá trị props thông thường khác. Để tham chiếu tới một thẻ HTML bên trong của một component tự định nghĩa, ta cần sử dụng một kĩ thuật là "chuyển tiếp ref". Xem thêm ở đây: https://reactjs.org/docs/forwarding-refs.html

Ngoài khả năng tham chiếu tới các thẻ HTML, `useRef` còn có thể được sử dụng để lưu trữ **các giá trị qua những lần render** của component. Xét ví dụ sau:

```
import {useRef, useState} from 'react' const App = () => { let count = 0;
const countRef = useRef(0); const [countState, setCountState] = useState(0);
console.log("count: ", count); console.log("countRef: ", countRef.current);
console.log("countState", countState); const increase = () => { count ++ }
const increaseRef = () => { countRef.current++ } const increaseState = () => {
setCountState(countState + 1) } return ( <div> <div> {count} <button onClick=
{increase}>Increase Count<button> </div> <div> {countRef.current} <button
onClick={increaseRef}>Increase CountRef<button> </div> <div> {countState}
<button onClick={increaseState}>Increase CountState<button> </div> </div> ) }
```

Trong ví dụ trên, ta thấy:

- Khi cập nhật `countRef` và `count`, giao diện không được cập nhật
- Khi thay đổi giá trị `countState`, giao diện được cập nhật. Giá trị của `count` sẽ trở về 0, giá trị của `countRef` vẫn sẽ được giữ nguyên.

Bản chất của việc render lại của React chính là chạy lại function `App` một lần nữa. Với việc sử dụng `useRef`, React hiểu rằng nó cần phải giữ lại giá trị của `ref` đó sau mỗi lần re-render. Tuy nhiên, khác với state, việc thay đổi giá trị của `ref` không khiến cho React thực hiện render lại component. Vì vậy, `useRef` có thể được sử dụng để lưu trữ các giá trị cần được giữ nguyên qua nhiều lần render, nhưng không ảnh hưởng tới UI của ứng dụng React. Một vài ví dụ có thể dùng `useRef` là timer, các object từ thư viện bên thứ 3.

👷 Hãy thử viết một ứng dụng đồng hồ bấm giờ với 3 thành phần đơn giản: Thời gian đã chạy, button Start và button Stop. Render nhiều component để có nhiều đồng hồ bấm giờ riêng biệt nhau.

---

2. Tối ưu hoá component với `React.memo`

PureComponent với `memo`

Mặc định, component trong React luôn luôn re-render lại khi giá trị props của nó bị thay đổi. Ngay cả khi giá trị cũ và giá trị mới của chúng trông "có vẻ giống nhau". Xét ví dụ sau:

```
import { useState, memo } from "react"; export default function App() { const [value, setValue] = useState({ name: "MindX", age: 20 }); const updateValue = () => { setValue({ name: "MindX", age: 20 }); }; const updateName = () => { setValue({ ...value, name: value.name + "x" }); }; const increaseAge = () => { setValue({ ...value, age: value.age + 1 }); }; return ( <div> <div>Check your console</div> <button onClick={updateValue}>Change value</button> <button onClick={updateName}>Change name</button> <button onClick={increaseAge}>Increase age</button> <Normal name={value.name} age={value.age} /> <Memoized name={value.name} age={value.age} /> </div> ); } const Normal = (props) => { console.log("Normal, Re-render"); return <div>Normal: {props.name + " " + props.age}</div>; }; const Pure = (props) => { console.log("Pure, Re-render"); return <div>Pure: {props.name + " " + props.age}</div>; }; const Memoized = memo(Pure);
```

Khi ta click vào button “Change value”, ta cập nhật lại giá trị của `value` bằng với giá trị của một object khác chứa toàn bộ giá trị của `value`. Về mặt dữ liệu, giá trị sau khi cập nhật state không thay đổi.

Tuy nhiên vẫn có thể thấy rằng component `Normal` được tính toán lại, trên màn hình console vẫn sẽ nhìn thấy dòng log. Đây là một component thông thường của React. Mặc dù giá trị `value` trước và sau giống nhau, React đơn giản nhận thấy giá trị tham chiếu của nó đã thay đổi. Và nó tiến hành việc tính toán lại component. Việc tính toán đó là thừa thãi và không cần thiết.

PureComponent thực hiện so sánh giá trị bên trong object props thay vì đơn thuần so sánh tham chiếu như component thông thường (với ví dụ trên là so sánh `name` và `age` giữa 2 lần render). Nếu như nó không phát hiện ra sự thay đổi trong props, React không tiến hành tính toán lại Component. Thay vào đó, nó sử dụng lại giá trị cũ của lần render trước.

React mặc định cung cấp “shallow compare” với `memo`. Tuy nhiên chúng ta cũng có thể định nghĩa sự so sánh. Cú pháp như sau:

```
const propsAreEquals = (prevProps, nextProps) => { return prevProps.name === nextProps.name } const Memoized = memo(Pure, propsAreEquals);
```

Khi này, React vẫn sẽ hiểu là giá trị props cũ và mới vẫn bằng nhau khi function `propsAreEquals` vẫn trả ra kết quả là `true`.

> 📌 Hãy cẩn thận với việc so sánh props ở đây. Nó có thể tạo ra các bugs dẫn đến ứng dụng hoạt động sai. Ví dụ như trong đoạn code trên, việc click vào button “Increase age” sẽ không làm cho component `Memoized` được render lại, dẫn đến kết quả hiển thị trên màn hình không thay đổi.

---

3. Ghi nhớ giá trị trong component với `useMemo` & `useCallback`

Các giá trị thông thường trong Component của React sẽ được tính toán lại sau mỗi lần thay đổi. Trong nhiều trường hợp với các giá trị phức tạp dựa vào một số state và props nhất định, việc tính toán lại chúng khi các giá trị state và props không liên quan khác thay đổi sẽ làm giảm hiệu năng của ứng dụng. Xét ví dụ sau:

```
import { useState, useMemo, useEffect } from "react"; export default function App() { const [items, setItems] = useState([1, 2, 3]); const [visible, setVisible] = useState(true); const doubleItems = items.map((item) => item * 2); const memoizedDoubleItems = useMemo(() => { return items.map((item) => item * 2); }, [items]); const changeVisible = () => { setVisible(!visible); }; const addValue = () => { setItems([...items, 1]); }; useEffect(() => { console.log("doubleItems changed"); }, [doubleItems]); useEffect(() => { console.log("memoizedDoubleItems changed"); }, [memoizedDoubleItems]); return ( <div> <button onClick={changeVisible}>Change visible</button> <button onClick={addValue}>Add value</button> {visible && ( <div> <p>{items}</p> <p>{doubleItems}</p> <p>{memoizedDoubleItems}</p> </div> )} </div> ); }
```

Ta thấy, giá trị của `doubleItems` và `memoizedDoubleItems` đều tính toán dựa vào giá trị của `items` state. Khi giá trị state này thay đổi, cả 2 giá trị kia đều cần phải được tính toán lại. Tuy nhiên, khi giá trị của state `visible` thay đổi, việc tính toán lại 2 giá trị double kia là không cần thiết. Nhưng với đặc thù của React, component của chúng ta vẫn sẽ tính toán lại tất cả các giá trị bên trong nó sau mỗi lần re-render. Điều này khiến cho `doubleItems` nhận giá trị là một mảng mới, dẫn đến các effect phụ thuộc vào nó sẽ được chạy lại.

Với việc dùng `useMemo`, React hiểu rằng nó chỉ cần tính toán lại giá trị này nếu như một trong số các giá trị phụ thuộc của nó bị thay đổi. Cú pháp của `useMemo` như sau:

```
const memoizedValue = useMemo(() => { // heavy calculator. need to return a value return }, []) // deps array.
```

`useMemo` gồm 2 thành phần: function thực hiện tính toán cần trả ra một kết quả, và một array các giá trị mà việc tính toán này phụ thuộc vào. Ý nghĩa của nó cũng tương tự với `useEffect`.

Tương tự với `useMemo`, `useCallback` cũng được sử dụng để ghi nhớ một function. React chỉ thực hiện việc khởi tạo lại function khi một trong số các giá trị phụ thuộc của nó thay đổi. `useCallback` cực kỳ hữu dụng trong việc đóng gói các logic được sử dụng cả bên trong lẫn bên ngoài `useEffect` cũng như các hooks khác. Xem ví dụ sau:

```
import { useState, useCallback, useEffect } from "react"; export default function App() { const [user, setUser] = useState(null); const fetchData = useCallback(() => { fetch("https://randomuser.me/api/") .then((res) => res.json()) .then((resJson) => { setUser(resJson.results[0]); }); }, []); useEffect(() => { fetchData(); }, [fetchData]); return ( <div> <button onClick={fetchData}>Refresh</button> {user ? ( <div> <img src={user.picture.medium} alt="" /> <p> {user.name.first} {user.name.last} </p> </div> ) : null} </div> ); }
```

📌 Không nên lạm dụng `useCallback` và `useMemo` trong tất cả các trường hợp vì nó sẽ làm code trở nên phức tạp hơn cũng như việc so sánh để nhận biết sự khác biệt cũng tốn một khoảng thời gian nhất định để thực hiện

---

4. Các hooks khác nên tìm hiểu.

Ngoài những hook như đã biết. Chúng ta có thể tìm hiểu thêm nhiều các hooks khác của React với các tính năng thú vị.

- `useReducer` : một cách khác để cập nhật state trong React component. Thích hợp với các state phức tạp. (https://reactjs.org/docs/hooks-reference.html#usereducer)

- `useLocalStorage` : đồng bộ dữ liệu trong ứng dụng với LocalStorage (https://usehooks.com/useLocalStorage/)
- `useAsync` : xử lý các tác vụ bất đồng bộ với các meta data (https://usehooks.com/useAsync/)
- `useHistory` : một cách đơn giản để tạo ra các ứng dụng yêu cầu undo, redo, ... (https://usehooks.com/useHistory/)