


1. useState


- Dùng khi nào? Khi muốn dữ liệu thay đổi thì giao diện tự động được cập nhật (render lại)



```
1 import { useState } from 'react';
2
3 function Component() {
4     const [state, setState] = useState(initState)
5
6     ...
7 }
```

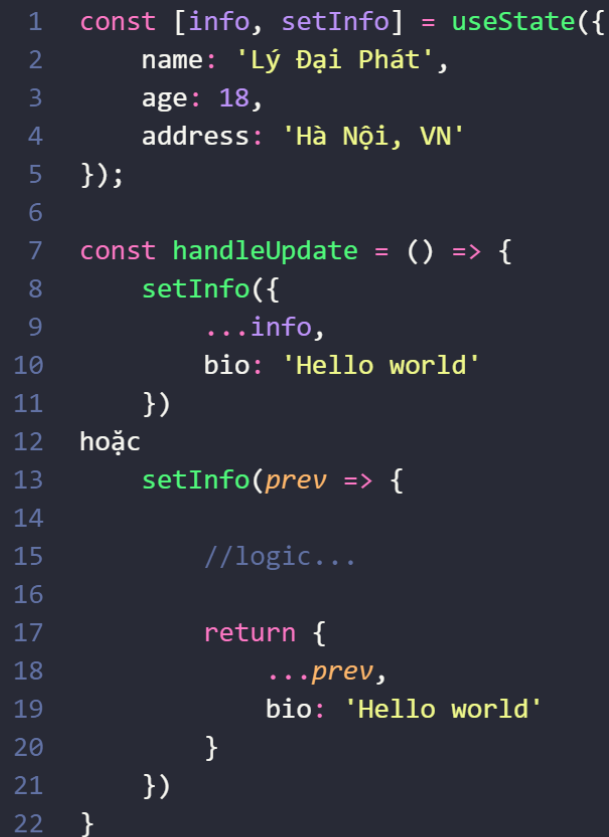
*Nếu initState là 1 function, thì sẽ lấy giá trị return của function đó làm giá trị initState

*Lưu ý: prevState trả về giá trị state trước đó, ví dụ ở đoạn code dưới đây setCounter sẽ thực hiện 3 lần



```
1 const [counter, setCounter] = useState(1)
2
3 const handleIncrease = () => {
4     setCounter(prevState => prevState + 1)
5     setCounter(prevState => prevState + 1)
6     setCounter(prevState => prevState + 1)
7 }
```

VD: cập nhật thêm thông tin



```
1  const [info, setInfo] = useState({
2    name: 'Lý Đại Phát',
3    age: 18,
4    address: 'Hà Nội, VN'
5  });
6
7  const handleUpdate = () => {
8    setInfo({
9      ...info,
10     bio: 'Hello world'
11   })
12  hoặc
13     setInfo(prev => {
14
15       //logic...
16
17       return {
18         ...prev,
19         bio: 'Hello world'
20       }
21     })
22  }
```

*One-way binding: Ràng buộc một chiều (ReactJS)

Khi dữ liệu state thay đổi, UI cũng thay đổi HOẶC ngược lại (chỉ 1 trong 2)

*Two-way binding: Ràng buộc hai chiều (ReactJS, VueJS)

Khi dữ liệu state thay đổi, UI cũng thay đổi VÀ ngược lại (cả 2)

*Ví dụ xử lý Form với two-way binding

```
1  *Ví dụ xử lý Form với two-way binding
2  function App() {
3      const [name, setName] = useState('');
4      const [email, setEmail] = useState('');
5
6      const handleSubmit = () => {
7          console.log({ name, email });
8      }
9
10     return (
11         <div className="App">
12             <input type="text" onChange={e => setName(e.target.value)} />
13             <input type="text" onChange={e => setEmail(e.target.value)} />
14             <button onClick={handleSubmit}>Register</button>
15         </div>
16     );
17 }
```

*Ví dụ xử lý Form Radio với two-way binding

```
1  * Ví dụ xử lý Form Radio với two - way binding
2
3  const courses = [
4    {
5      id: 1,
6      name: 'HTML, CSS'
7    },
8    {
9      id: 2,
10     name: 'JavaScript'
11   },
12   {
13     id: 3,
14     name: 'ReactJS'
15   },
16 ]
17
18
19 function App() {
20   const [checked, setChecked] = useState();
21
22   console.log(checked);
23   const handleSubmit = () => {
24     //Call API
25     console.log({ id: checked });
26   }
27
28   return (
29     <div className="App">
30       {courses.map(course => (
31         <div key={course.id}>
32           <input
33             type="radio"
34             checked={checked === course.id}
35             onChange={() => setChecked(course.id)}
36           />
37           {course.name}
38         </div>
39       ))}
40
41       <button onClick={handleSubmit}>Register</button>
42
43     </div>
44   );
45 }
```

*Ví dụ xử lý Form Checkbox với two-way binding

```
1  *Ví dụ xử lý Form Checkbox với two-way binding
2
3  const courses = [
4    {
5      id: 1,
6      name: 'HTML, CSS'
7    },
8    {
9      id: 2,
10     name: 'JavaScript'
11   },
12   {
13     id: 3,
14     name: 'ReactJS'
15   },
16 ]
17
18
19 function App() {
20   const [checked, setChecked] = useState([]);
21
22   const handleCheck = (id) => {
23     setChecked(prev => {
24       const isChecked = checked.includes(id);
25       if (isChecked) {
26         return checked.filter(item => item !== id);
27       } else {
28         return [...prev, id]
29       }
30     })
31   }
32
33   const handleSubmit = () => {
34     //Call API
35     console.log({ ids: checked });
36   }
37
38   return (
39     <div className="App">
40       {courses.map(course => (
41         <div key={course.id}>
42           <input
43             type="checkbox"
44             checked={checked.includes(course.id)}
45             onChange={() => handleCheck(course.id)}
46           />
47           {course.name}
48         </div>
49       ))}
50
51       <button onClick={handleSubmit}>Register</button>
52
53     </div>
54   );
55 }
```

*Lưu ý về performance, ví dụ todolist, nếu storageJobs để ở ngoài thì mỗi lần render chạy lại sẽ gọi storageJobs => lãng phí

Khắc phục: useState truyền vào 1 callback

```
1  function App() {
2
3      const [job, setJob] = useState('')
4      const [jobs, setJobs] = useState(() => {
5          //Đưa vào trong callback của useState => tăng performance do chỉ gọi 1 lần
6          const storageJobs = JSON.parse(localStorage.getItem('jobs'))
7          console.log(storageJobs);
8          //toán tử ??: nếu về đầu là null hoặc undefined thì lấy về sau
9          return storageJobs ?? []
10     })
11
12     const handleSubmit = () => {
13         setJobs(prev => {
14             const newJobs = [...prev, job]
15             const jsonJobs = JSON.stringify(newJobs)
16             localStorage.setItem('jobs', jsonJobs);
17
18             return newJobs
19         })
20
21         setJob('')
22     }
23
24     return (
25         <div className="App">
26             <input
27                 value={job}
28                 type="text"
29                 onChange={(e) => setJob(e.target.value)} />
30             <button onClick={handleSubmit}>Add</button>
31
32             <ul>
33                 {jobs.map((job, index) => (
34                     <li key={index}>{job}</li>
35                 ))}
36             </ul>
37         </div>
38     );
39 }
```

*MOUNTED & UNMOUNTED

mounted: thời điểm đưa component vào sử dụng

unmounted: thời điểm gỡ component, k sử dụng

2. useEffect hook

- Dùng update DOM, Call API, Listen DOM events, Cleanup

- Tại sao lại dùng useEffect? Để giải quyết trường hợp xử lý event sau khi render DOM xong mới xử lý, nếu chạy trước khi render DOM sẽ lỗi ngay.

* **Callback luôn được gọi lại sau khi component mounted** đối với cả 3 cách dùng.

* **Cleanup function luôn được gọi lại trước khi component unmounted**, đối với React@18 thì không cần thêm cleanup function cũng được

* **Cleanup function luôn được gọi lại trước khi callback được gọi** (trừ lần mounted)

1. `useEffect(callback)` - ứng dụng khi gõ tới đâu, hiện title lên trình duyệt tới đó

- Gọi callback mỗi khi component re-render
- Gọi callback sau khi component thêm element vào DOM
- Gọi callback nhiều lần

2. `useEffect(callback, [])` - ứng dụng khi call API, chỉ call 1 lần duy nhất

- Đối số thứ 2 là **mảng rỗng** => chỉ gọi callback 1 lần duy nhất

3. `useEffect(callback, [deps])` - ứng dụng khi có 3 option, click option nào thì render giao diện ấy ra

- Đối số thứ 2 là mảng chứa deps => Callback được gọi lại mỗi khi deps thay đổi

TH1: - ứng dụng khi gõ tới đâu, hiện title lên trình duyệt tới đó

```
1  TH1: - ứng dụng khi gõ tới đâu, hiện title lên trình duyệt tới đó
2  function App() {
3
4      const [title, setTitle] = useState('')
5
6      useEffect(() => {
7          document.title = title
8      })
9
10     return (
11         <div className="App">
12             <input
13                 type="text"
14                 value={title}
15                 onChange={e => setTitle(e.target.value)} />
16         </div>
17     );
18 }
```


TH2: - ứng dụng khi call API, chỉ call 1 lần duy nhất



```
1  TH2: - ứng dụng khi call API, chỉ call 1 lần duy nhất
2
3  function App() {
4
5      const [title, setTitle] = useState('')
6      const [posts, setPosts] = useState([])
7
8      useEffect(() => {
9          fetch('https://jsonplaceholder.typicode.com/posts')
10             .then(res => res.json())
11             .then(posts => {
12                 setPosts(posts)
13             })
14      }, [])
15
16      return (
17          <div className="App">
18              <input
19                  type="text"
20                  value={title}
21                  onChange={e => setTitle(e.target.value)} />
22              <ul>
23                  {posts.map(post => (
24                      <li key={post.id}>{post.title}</li>
25                  ))}
26              </ul>
27          </div>
28      );
29  }
```

TH3: - ứng dụng khi có 3 option, click option nào thì render giao diện ấy ra

```
1 TH3: - ứng dụng khi có 3 option, click option nào thì render giao diện ấy ra
2
3 const tabs = ['posts', 'todos', 'albums']
4
5 function App() {
6
7     const [title, setTitle] = useState('')
8     const [posts, setPosts] = useState([])
9     const [type, setType] = useState('posts')
10
11
12     useEffect(() => {
13         fetch(`https://jsonplaceholder.typicode.com/${type}`)
14             .then(res => res.json())
15             .then(posts => {
16                 setPosts(posts)
17             })
18     }, [type])
19
20     return (
21         <div className="App">
22             {tabs.map(tab => (
23                 <button
24                     key={tab}
25                     style={type === tab ?
26                         {
27                             color: '#fff',
28                             backgroundColor: '#333'
29                         } : {}}
30                     onClick={() => setType(tab)}>
31                     {tab}
32                 </button>
33             ))}
34
35             <input
36                 type="text"
37                 value={title}
38                 onChange={e => setTitle(e.target.value)} />
39             <ul>
40                 {posts.map(post => (
41                     <li key={post.id}>{post.title}</li>
42                 ))}
43             </ul>
44         </div>
45     );
46 }
47 }
```

*useEffect with DOM events

Show go to top button

```
1  function Content() {
2    ...
3    const [showGoToTop, setShowGoToTop] = useState(false)
4
5    ...
6    useEffect(() => {
7
8      const handleScroll = () => {
9        setShowGoToTop(window.scrollY >= 200)
10     }
11
12     window.addEventListener('scroll', handleScroll)
13
14     //Cleanup function (React@18 k cần cũng được)
15     return () => {
16       window.removeEventListener('scroll', handleScroll)
17     }
18
19   }, [])
20
21   return (
22     <div>
23       ...
24       {showGoToTop && (
25         <button
26           style={
27             {
28               position: 'fixed',
29               right: 20,
30               bottom: 20
31             }
32           }>
33         TOP
34       </button>
35     )}
36   </div>
37 )
38 }
```

Ta nên hủy bỏ việc lắng nghe một event khi mà một component bị unmouted, nếu component đã k cần dùng mà ta cứ lắng nghe, và sau đó component đó được thêm lại, ta tiếp tục lắng nghe cộng dồn thêm thì ta sẽ bị memory leak => Cần cleanup function để hủy bỏ việc lắng nghe trước khi component unmouted.

*Cleanup function: thêm return về 1 function trong useEffect là được.

*React@18 thì k cần thêm cleanup function cũng được.

Window resize width



```
1  *useEffect with DOM events
2  function Content() {
3      const [width, setWidth] = useState(window.innerWidth)
4
5      useEffect(() => {
6
7          const handleResize = () => {
8              setWidth(window.innerWidth)
9          }
10
11         window.addEventListener('resize', handleResize)
12
13         //Cleanup function
14         return () => {
15             window.removeEventListener('resize', handleResize)
16         }
17     }, [])
18
19     return (
20         <div>
21             <h1>{width}</h1>
22         </div>
23     )
24 }
```

Countdown Timer



```
1  function Content() {
2      const [countdown, setCountdown] = useState(180)
3
4      C1:  useEffect(() => {
5          const timerId = setInterval(() => {
6              setCountdown(prevState => prevState - 1)
7          }, 1000);
8
9          return () => clearInterval(timerId)
10     }, [])
11
12     C2:  useEffect(() => {
13         const timerId = setTimeout(() => {
14             setCountdown(countdown - 1)
15         }, 1000);
16
17         return () => clearInterval(timerId)
18     }, [countdown])
19
20     return (
21         <div>
22             <h1>{countdown}</h1>
23         </div>
24     )
25 }
```

Preview Avatar

```
1  function Content() {
2
3      const [avatar, setAvatar] = useState()
4
5      //mục đích để link xóa ảnh nếu ta chọn ảnh khác, tránh leak memory
6      useEffect(() => {
7          return () => {
8              //do lúc đầu avatar có state là rỗng, nếu xóa ngay sẽ có lỗi
9              //kiểm tra avatar có tồn tại thì mới xóa
10             avatar && URL.revokeObjectURL(avatar.preview)
11         }
12     }, [avatar])
13
14     const handlePreviewAvatar = (e) => {
15         const file = e.target.files[0];
16
17         //file là 1 object, nên có thể thêm key mới
18         file.preview = URL.createObjectURL(file)
19
20         setAvatar(file)
21     }
22
23     return (
24         <div>
25             <input
26                 type="file"
27                 onChange={handlePreviewAvatar} />
28             {avatar && (
29                 <img src={avatar.preview} alt="" width="30%" />
30             )}
31         </div>
32     )
33 }
```

fake Chat App

Mục đích: Tạo ứng dụng chat, comment, ... theo thời gian thực.

Ý tưởng: dựa trên việc subscribe và unsubscribe

- Giả sử bạn và những người dùng khác đang đứng ở bài 1, thì sẽ có 1 kênh truyền được tạo ra để theo dõi (subscribe) bài 1 này, nếu một người comment thì tất cả những người còn lại đang đứng ở bài 1 này sẽ nhìn thấy được comment.

- Nếu bạn chuyển sang bài khác, thì phải unsubscribe bài 1, subscribe bài khác, nếu không bạn sẽ subscribe cả 2 bài => memory leak



```
1 // Fake comments sử dụng custom event
2 function emitComment(id) {
3     setInterval(() => {
4         //chủ động phát 1 event
5         window.dispatchEvent(
6             new CustomEvent(`lesson-${id}`, {
7                 detail: `Nội dung comment của lesson ${id}`
8             })
9         )
10    }, 2000);
11 }
12
13 emitComment(1)
14 emitComment(2)
15 emitComment(3)
```



```
1  const lessons = [
2    {
3      id: 1,
4      name: 'ReactJS là gì? Tại sao nên học ReactJS?'
5    },
6    {
7      id: 2,
8      name: 'SPA/MPA là gì?'
9    },
10   {
11     id: 3,
12     name: 'Hooks là gì?'
13   }
14 ]
15
16 function Content() {
17   const [lessonId, setLessonId] = useState(1)
18
19   //destructuring 1 object có key là detail
20   const handleComment = ({ detail }) => {
21     console.log(detail);
22   }
23
24   useEffect(() => {
25     window.addEventListener(`lesson-${lessonId}`, handleComment)
26
27     return () => {
28       window.removeEventListener(`lesson-${lessonId}`, handleComment)
29     }
30   }, [lessonId])
31
32   return (
33     <div>
34       <ul>
35         {lessons.map((Lesson) => (
36           <li
37             key={Lesson.id}
38             style={{
39               color: lessonId === Lesson.id ?
40                 'red' : '#333'
41             }}
42             onClick={() => setLessonId(Lesson.id)}
43           >
44             {Lesson.name}
45         </li>
46         ))}
47       </ul>
48     </div>
49   )
50 }
51 }
```


3. useEffect

*Lưu ý: thực tế sử dụng useEffect, nếu gặp trường hợp hy hữu thì thử sử dụng useLayoutEffect.

Sự khác nhau giữa useEffect và useLayoutEffect:

useEffect:

1. Cập nhật lại state
2. Cập nhật lại DOM (mutated)
3. **Render lại UI**
4. Gọi cleanup func nếu deps thay đổi
5. Gọi useEffect callback

useLayoutEffect:

1. Cập nhật lại state
2. Cập nhật lại DOM (mutated)
3. Gọi cleanup func nếu deps thay đổi (sync)
4. Gọi useLayoutEffect callback (sync)
5. **Render lại UI**

Ta có 1 ví dụ nhỏ như sau, một ứng dụng đếm số từ 0 đến 3, nếu số lớn hơn 3 thì quay trở về 0.

Nếu ta dùng useEffect sẽ có một vấn đề phát sinh là UI sẽ hiện số 4 trước, sau đó mới set về 0. Khắc phục bằng cách sử dụng useLayoutEffect.

```
1  function Content() {
2      const [count, setCount] = useState(0)
3
4      const handleRun = () => {
5          setCount(count + 1)
6      }
7
8      useLayoutEffect(() => {
9          if (count > 3) setCount(0)
10     }, [count])
11
12     return (
13         <div>
14             <h1>{count}</h1>
15             <button onClick={handleRun}>Click</button>
16         </div>
17     )
18 }
```

mutated: đột biến, giả sử ta có 1 object, nó sẽ sửa 1 props trong object, trông bên ngoài object vẫn như vậy, nhưng bên trong ruột bị thay đổi 1 thành phần nhỏ.

Giải thích useEffect ở hình trên:

Khi count đang có giá trị là 3, khi ta click vào button thì render lại component, lúc này count đang là 4 (lưu ý chưa chạy vào useEffect vội), nó sẽ chạy xuống đoạn return để thực hiện mutated lại DOM nhưng chưa render ra UI mà nó sẽ quay lên gọi callback trong useEffect để setCount về 0. Khi count về 0 thì lặp lại quá trình cập nhật state > cập nhật DOM > không thỏa điều kiện trong useEffect > Render UI.

4. useRef hook

Ra đời nhằm mục đích lưu 1 giá trị bất kỳ nằm ngoài function, và có thể tham chiếu đến được.

```
1  function Content() {
2    const [count, setCount] = useState(60)
3
4    let timerID
5
6    const handleStart = () => {
7      timerID = setInterval(() => {
8        setCount(prevCount => prevCount - 1)
9      }, 1000);
10
11     console.log('start', timerID);
12   }
13
14   const handleStop = () => {
15     clearInterval(timerID)
16
17     console.log('stop', timerID);
18   }
19
20   return (
21     <div>
22       <h1>{count}</h1>
23       <button onClick={handleStart}>Start</button>
24       <button onClick={handleStop}>Stop</button>
25     </div>
26   )
27 }
```

Ở ví dụ này, cứ mỗi 1s thì `setCount` được gọi thì component `Content` này cũng được gọi lại, do đó các biến sẽ tạo ra một phạm vi mới, do đó `timerID` cũng tạo ra phạm vi mới và không giữ được giá trị ID cũ của nó (muốn xóa `setInterval` thì phải có id).

Có 2 cách để khắc phục:

- Cách 1: khai báo biến `timerID` ra ngoài function (giống như xem biến global)
- Cách 2: Dùng `useRef`

*`useRef` luôn trả về một object, có property key là `current` => truy cập value dùng `.current`

```
1  function Content() {
2    const [count, setCount] = useState(60)
3
4    let timerID = useRef()
5
6    const handleStart = () => {
7      timerID.current = setInterval(() => {
8        setCount(prevCount => prevCount - 1)
9      }, 1000);
10   }
11
12   const handleStop = () => {
13     clearInterval(timerID.current)
14   }
15
16   return (
17     <div>
18       <h1>{count}</h1>
19       <button onClick={handleStart}>Start</button>
20       <button onClick={handleStop}>Stop</button>
21     </div>
22   )
23 }
```

`useRef` cũng có thể dùng để lưu các tag với props có tên là `ref`



```
1  function Content() {  
2    const h1Ref = useRef()  
3  
4    useEffect(() => {  
5      const rect = h1Ref.current.getBoundingClientRect()  
6      console.log(rect);  
7    }, [])  
8    return (  
9      <div>  
10        <h1 ref={h1Ref}>Hi</h1>  
11  
12      </div>  
13    )  
14  }
```

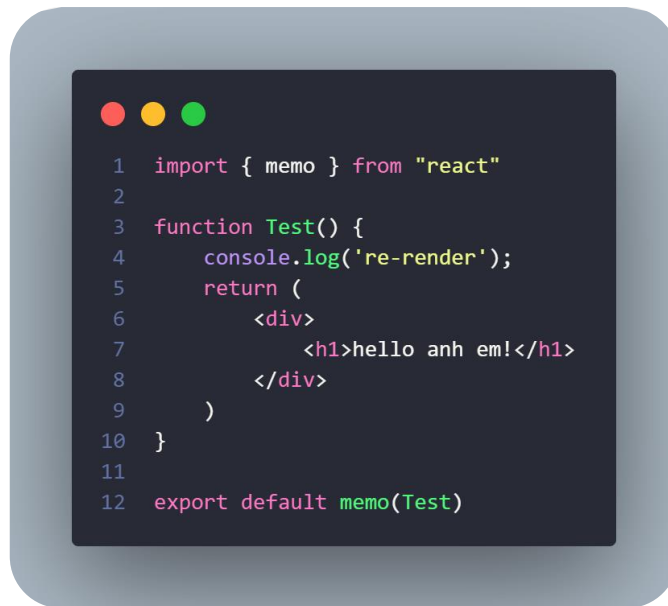
5. React.memo (k phải là hook)

*React.memo là HOC: Higher order component => tránh render 1 component không cần thiết!

Một component cha, có nhiều component con, mỗi khi component cha render lại thì sẽ có một vài component con không nhất thiết phải render lại mà vẫn bị render lại => cũng ảnh hưởng đến performance



```
1  function App() {  
2    const [count, setCount] = useState(0);  
3  
4    return (  
5      <div className="App">  
6        <Test />  
7        <h1>{count}</h1>  
8        <button onClick={() => setCount(count + 1)}>Click me!</button>  
9      </div>  
10    );  
11  }
```



hello anh em!

5

Click me!

Ta thấy, mỗi lần component App render lại, thì phần component Test nên giữ nguyên, k cần render lại (do props ko thay đổi) => thêm export default memo(Test)

6. useCallback (kết hợp với React.memo)

- Bản chất tương tự React.memo, nhưng dùng để tránh tạo ra function mới không cần thiết.

- Lần đầu ứng dụng chạy, sẽ tạo ra 1 tham chiếu hàm handleIncrease, sau đó truyền function này cho props. Khi ta click vào button đồng nghĩa ta sẽ set lại count dẫn tới việc render lại component App(). Lúc này 1 tham chiếu hàm handleIncrease mới sẽ được tạo ra (không liên quan gì tới hàm handleIncrease trước đó), sau đó truyền tham chiếu mới này cho props. Bên component Test(), React.memo() sẽ so sánh tham chiếu cũ và tham chiếu mới với toán tử ===, nhận được kết quả là false => render lại component Test().



```
1 function App() {
2   const [count, setCount] = useState(0);
3
4   const handleIncrease = () => {
5     setCount(prevCount => prevCount + 1);
6   }
7
8   return (
9     <div className="App">
10      <Test onIncrease={handleIncrease} />
11      <h1>{count}</h1>
12    </div>
13  );
14 }
```



```
1 function Test({ onIncrease }) {
2   console.log('re-render');
3   return (
4     <div>
5       <>
6         <h1>hello anh em!</h1>
7         <button onClick={onIncrease}>Click me!</button>
8       </>
9     </div>
10  )
11 }
12
13 export default memo(Test)
```

- Nói ngắn gọn, kết quả ví dụ dưới đây trả về false mặc dù nhìn bề ngoài thấy giống nhau, nhưng thực chất 2 tham chiếu vùng nhớ khác nhau.

```
const a = () => {};
```

```
const b = () => {};
```

```
a === b => false
```

- Cách khắc phục: Kết hợp sử dụng `useCallback`

- `useCallback` nhận 2 đối số, đối số thứ nhất là function, đối số thứ 2 là mảng chứa các deps (tương tự `useEffect`)

- Bản chất là function ở đối số thứ nhất sẽ được lưu ở ngoài component App(), giống như đưa ra ngoài làm biến global, để có thể gọi lại và truy cập mà không bị tạo mới.

```
1 function App() {
2   const [count, setCount] = useState(0);
3
4   const handleIncrease = useCallback(() => {
5     setCount(prevCount => prevCount + 1);
6   }, [])
7
8   return (
9     <div className="App">
10      <Test onIncrease={handleIncrease} />
11      <h1>{count}</h1>
12    </div>
13  );
14 }
```

```
1 function Test({ onIncrease }) {
2   console.log('re-render');
3   return (
4     <div>
5       <>
6         <h1>hello anh em!</h1>
7         <button onClick={onIncrease}>Click me!</button>
8       </>
9     </div>
10  )
11 }
12
13 export default memo(Test)
```

useCallback phải đi kèm với React.memo(). Ví dụ:

Component A: truyền props;

Component B: nhận props;

- React.memo() được sử dụng ở phía component B, nó sẽ thực hiện việc so sánh giữa hai lần props nhận được có bằng nhau hay không để quyết định xem có render lại component B hay không. Tuy nhiên việc so sánh ở trên chỉ là shallow comparison thôi, tức là với những kiểu dữ liệu primitive trong JS thì nó compare đúng, ngược lại thì không. Chính vì vậy mới cần sinh ra thêm thằng useCallback.

- useCallback được sử dụng ở component A, nó có nhiệm vụ wrap bên ngoài một function - function này thường là callback để truyền dưới dạng prop cho Component B. Mục đích là để khi component A bị render lại thì nó function được wrap bên trong useCallback đó sẽ không bị khởi tạo lại thành một tham chiếu mới, khi đó bên component B thực hiện compare hiểu được props nó nhận được là không thay đổi => Không render lại nữa.

Vì thế nên dùng usecallback mà không có React.memo() thì là vô nghĩa.

7. useMemo

*useMemo là 1 hook => tránh thực hiện 1 logic lặp lại không cần thiết!

```
1  function App() {
2    const [name, setName] = useState('');
3    const [price, setPrice] = useState('');
4    const [products, setProducts] = useState([]);
5
6
7    const handleSubmit = () => {
8      setProducts([...products, {
9        name,
10       price: +price
11     }])
12   }
13
14   const total = products.reduce((result, product) => result + product.price, 0)
15
16   return (
17     <div style={{ padding: '10px 32px' }}>
18       <input onChange={e => setName(e.target.value)} />
19       <br />
20       <input onChange={e => setPrice(e.target.value)} />
21       <br />
22       <button onClick={handleSubmit}>Add</button>
23       <br />
24       Total:{total}
25       <ul>
26         {products.map((product, index) => (
27           <li key={index}>{product.name} - {product.price}</li>
28         ))}
29       </ul>
30     </div>
31   );
32 }
```

Với đoạn code trên, khi đang nhập input (chưa submit) thì component App render lại ra UI, đồng nghĩa với việc biến total sẽ tính toán lại => không cần thiết, khi nào submit xong mới tính toán lại.

Cách khắc phục: useMemo

- useMemo nhận vào 2 đối số, đối số thứ nhất là function, đối số thứ 2 là mảng chứa các deps (tương tự useEffect).

```
1  function App() {
2    const [name, setName] = useState('');
3    const [price, setPrice] = useState('');
4    const [products, setProducts] = useState([]);
5
6    const nameRef = useRef();
7
8    const handleSubmit = () => {
9      setProducts([...products, {
10        name,
11        price: +price
12      }])
13      setName('')
14      setPrice('')
15      nameRef.current.focus()
16    }
17
18    const total = useMemo(() => {
19      const result = products.reduce((result, product) => {
20        return result + product.price
21      }, 0)
22      return result
23    }, [products])
24
25    return (
26      <div style={{ padding: '10px 32px' }}>
27        <input
28          ref={nameRef}
29          value={name}
30          onChange={e => setName(e.target.value)} />
31        <br />
32        <input
33          value={price}
34          onChange={e => setPrice(e.target.value)} />
35        <br />
36        <button
37          onClick={handleSubmit}>Add</button>
38        <br />
39        Total:{total}
40        <ul>
41          {products.map((product, index) => (
42            <li key={index}>{product.name} - {product.price}</li>
43          ))}
44        </ul>
45      </div>
46    );
47  }
```

8. useReducer

- Cung cấp việc sử dụng state cho component, bài toán có dữ liệu lồng nhau phức tạp hoặc có nhiều state thì nên dùng useReducer, kiểu dữ liệu đơn giản hoặc có ít state thì dùng useState.
- Các bài toán mà useState giải quyết được thì useReducer cũng giải quyết được và ngược lại.

VD1: Cho bài toán khi click vào button thì tăng hoặc giảm giá trị hiện tại đi 1 đơn vị.

*Phân tích bài toán với useState:

1. Init state: 0
2. Actions: Up (state +1) / Down (state - 1)

*Phân tích bài toán với useReducer:


1. Init state: 0
2. Actions: Up (state +1) / Down (state - 1)
3. Reducer (xử lý logic chính ở đây)
4. Dispatch (kích hoạt một actions)

- reducer nhận 2 tham số là state, action. Dựa vào hành động và state hiện tại thì quyết định trả ra state mới là gì.

- useReducer có thể nhận tới 3 đối số (thực tế thường dùng 2 đối số), đối số thứ nhất là hàm reducer, đối số thứ 2 là initState.

- hàm dispatch nhận vào 1 action.

- Khi component chạy lần đầu, useReducer sẽ chạy và gán giá trị initState cho biến count (hàm reducer tạm thời vẫn chưa chạy). Khi ta bấm vào button up hoặc down thì kích hoạt dispatch với action tương ứng, hàm reducer sẽ chạy với 2 đối số là initState và action vừa dispatch và trả ra kết quả.



```

1  const initState = 0
2
3  const UP_ACTION = 'up'
4  const DOWN_ACTION = 'down'
5
6  const reducer = (state, action) => {
7    switch (action) {
8      case UP_ACTION:
9        return state + 1
10     case DOWN_ACTION:
11       return state - 1
12     default:
13       throw new Error('Invalid action')
14   }
15 }
16
17 function App() {
18   const [count, dispatch] = useReducer(reducer, initState);
19
20   return (
21     <div
22       style={{ padding: '0 20px' }}>
23       <h1>{count}</h1>
24       <button
25         onClick={() => { dispatch(DOWN_ACTION) }}>
26         Down
27       </button>
28       <button
29         onClick={() => { dispatch(UP_ACTION) }}>
30         Up
31       </button>
32     </div>
33   );
34 }

```

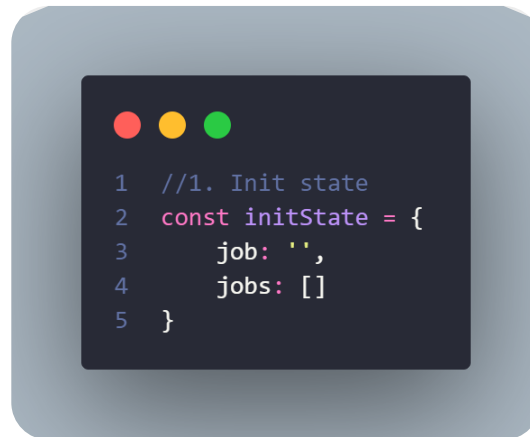
Todo App with useReducer

[Link code:](#)

*Phân tích bài toán:

1. Init state:

- Có ít nhất 2 state, state 1 thể hiện dữ liệu người dùng nhập vào ô input, state 2 thể hiện danh sách công việc.
- Mỗi khi gõ vào ô input thì update lại state 1, mỗi khi bấm add thì update lại state 2.



2. Actions:

- Xác định các hành động: set job, add job và xóa job



3. Reducer (viết dispatch trước, reducer viết sau):

4. Dispatch:

- state sẽ nhận initState là một object gồm job và jobs, do đó ta sẽ dùng destructuring để lấy ra job và jobs cho tiện, sau đó two way binding ra giao diện người dùng.
- Như vậy cơ bản là ta đã xong.

```

1 //4. Dispatch
2 function App() {
3     const [state, dispatch] = useReducer(reducer, initState)
4     const { job, jobs } = state;
5
6     return (
7         <div style={{ padding: '0 20px' }}>
8             <h3>Todo</h3>
9             <input
10                 placeholder="Enter todo..." value={job}
11             />
12             <button>Add</button>
13             <ul>
14                 {jobs.map((job, index) => (
15                     <li key={index}>job</li>
16                 ))}
17             </ul>
18         </div>
19     );
20 }

```

5. Tiếp tục phân tích:

- Khi nhập vào ô input, ta cần truyền dữ liệu đồng thời dispatch một actions, do đó ta phải viết thêm 1 hàm để trả về 1 object, object này bao gồm dữ liệu và action tương ứng.
- Mẫu setJob (2 hàm còn lại tương tự):

```

1 const setJob = payload => {
2     return {
3         type: SET_JOB,
4         payload
5     }
6 }

```

- Do đó, action sẽ nhận lại là 1 object gồm 2 key là type và payload. Ta sẽ xử lý switch case cho trường hợp SET_JOB như sau:

```

1 //3. Reducer
2 const reducer = (state, action) => {
3
4     switch (action.type) {
5         case SET_JOB:
6             return {
7                 ...state,
8                 job: action.payload
9             }
10        default:
11            throw new Error('Invalid action')
12    }
13 }

```

- Tiếp theo ta sẽ xử lý cho trường hợp click vào button, khi click ta sẽ dispatch action là addJob kèm theo payload là job.

```

1 <button onClick={handleSubmit}>Add</button>

```

```

1 const handleSubmit = () => {
2     dispatch(addJob(job))
3     dispatch(setJob('')) //reset input value
4     inputRef.current.focus();
5 }


```

```


1 case ADD_JOB:
2     return {
3         ...state,
4         jobs: [...state.jobs, action.payload]
5     }

```

- Cuối cùng là xử lý delete khi bấm vào button.



```
1 <ul>
2   {jobs.map((job, index) => (
3     <li key={index}>{job}
4       <span onClick={() => dispatch(deleteJob(index))}>
5         &times;
6       </span>
7     </li>
8   )})
9 </ul>
```



```
1 case DELETE_JOB:
2   const newJobs = [...state.jobs]
3   newJobs.splice(action.payload, 1)
4
5   return {
6     ...state,
7     jobs: newJobs
8   }
```