



Lesson 14: Class-based Components

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



Một cách khác để tạo các component

Với phiên bản React 16.8. ReactJS giới thiệu hook là một giải pháp tốt để tạo ra các component. Trước phiên bản đó, nếu như muốn sử dụng state và những tính năng khác, React đòi hỏi chúng ta cần phải sử dụng `class` để tạo ra một component.



Trong thực tế ngày nay, chúng ta sẽ ít nhìn thấy class component trong các ứng dụng. Tuy nhiên, có rất nhiều ứng dụng sử dụng các phiên bản cũ của React vẫn đang sử dụng cách viết này. Do đó, phần học này hoàn toàn là tùy chọn. Chúng ta vẫn nên sử dụng function component và các hooks cho các ứng dụng hiện tại.

1. Giới thiệu về `class`

Trong JS, Class đơn giản là một “bản thiết kế” để tạo ra các objects. Với Class, chúng ta có thể định nghĩa ra các trường dữ liệu và các phương thức mà một object được tạo ra từ class đó sẽ có.

Trong một class có 2 thành phần chính:

- Properties: các thuộc tính mà object đó sở hữu: `name`, `color`, `age`, `description`, ...
- Methods: các phương thức (function) mà object đó có thể gọi được: `sayHello`, `run`, `render`, ... Bên trong các phương thức, ta có thể truy cập vào các thuộc tính thông qua con trỏ `this`

```
class Person() { constructor(name) { this.name = name } sayHello() { console.log(`Hello, my name is ${this.name}`) } } const personA = new Person("Alice") personA.sayHello() // Hello, my name is Alice const personB = new Person("Bob") personB.sayHello() // Hello, my name is Bob
```

Một object được khởi tạo từ class thông qua toán tử `new`. Toán tử `new` được sử dụng với class để tạo ra một object mới, bao gồm các thuộc tính và phương tính được định sẵn bên trong class, thông qua một method đặc biệt là `constructor`. Constructor được sử dụng để nhận các tham số truyền từ bên ngoài vào ở thời điểm object được tạo ra.

Con trỏ `this` bên trong class đề cập tới thực thể riêng biệt của class đó. Trong ví dụ trên, `this` của `personA` sẽ riêng biệt với `this` của `personB`.

Kế thừa và con trỏ `super`

Một class có thể kế thừa một class khác. Kế thừa giúp cho class con có toàn bộ các thuộc tính và phương thức của class cha.

```
class Person { constructor(name) { this.name = name } sayHello() {
console.log(`Hello, my name is ${this.name}`) } } class Student extends Person
{ constructor(name, grade) { super(name) this.grade = grade } sayHello() {
super.sayHello(); console.log("I am grade " + this.grade) } } const studentA =
new Student("Alice", 10) studentA.sayHello()
```

Trong ví dụ này, Person là class cha, Student là class con

Con trỏ `super` được sử dụng để có truy cập vào các giá trị được kế thừa từ class cha.

Một cách khác để khai báo các properties & methods

Các properties và methods trong phiên bản JS mới hơn (ES7+) có thể được khai báo theo cú pháp `classProperties`


```
class Person { constructor(name)
{ this.name = name } sayHello() {
// your code here } }
```

Cú pháp thường

```
class Person { name = ""
constructor(name) { this.name =
name } sayHello = () => { // your
code here } }
```

classProperties

Với cách thông thường, chúng ta buộc phải khai báo các properties bên trong `constructor`, và các methods cần phải được viết dưới dạng function tiêu chuẩn. Với cú pháp mới, chúng ta có thể khởi tạo các trường dữ liệu mà không cần sử dụng `constructor`, và các methods có thể được viết dưới dạng arrow function. Về cơ bản, hai cách này không có quá nhiều khác biệt. Điểm khác biệt chủ yếu là con trỏ `this` bên trong các methods. Arrow function không có `this`, nên nó có thể tránh được các vấn đề liên quan xử lý các sự kiện trong JS.



Javascript không có đầy đủ các tính chất của lập trình hướng đối tượng. Nên sử dụng các ngôn ngữ lập trình khác như Java hoặc C++ cho mục đích thực hành lập trình hướng đối tượng

2. Class component

Một ví dụ đơn giản với function component và ví dụ tương đương với class component:

```
const NameCard = () => { return (
<div> <div>Name: Alice</div>
<div>Age: 20</div> </div> ) }
```

Function component

```
import {Component} from 'react'
class NameCard extends Component
{ render( return ( <div>
<div>Name: Alice</div> <div>Age:
20</div> </div> ) } }
```

Class component

Đầu tiên, chúng ta khởi tạo component dưới dạng class. Class này cần được kế thừa class `Component` bên trong thư viện React. Việc kế thừa này đảm bảo việc component `NameCard` sẽ có đầy đủ các properties và methods cần thiết của một React component.

Bên trong class `NameCard`, chúng ta bắt buộc phải có một method có tên là `render`. Đây chính là method chịu trách nhiệm trả về phần JSX của component. `render` buộc phải trả về một React element. Ta có thể quan sát thấy `render` khá giống function component thông thường.

Truy cập vào Props

Với function component, props là tham số được truyền vào. Đối với class component, props có thể được truy cập thông qua một property đặc biệt là `props`

```
const NameCard = (props) => {
  return (
    <div>
      <div>Name:
      {props.name}</div>
      <div>Age:
      {props.age}</div>
    </div>
  )
} ...
<NameCard name="Alice" age={20} />
```

```
import {Component} from 'react'
class NameCard extends Component
{
  render() {
    return (
      <div>
        <div>Name: {this.props.name}
        </div>
        <div>Age: {this.props.age}
        </div>
      </div>
    )
  }
}
<NameCard name="Alice" age={20} />
```

Việc kế thừa từ class Component cho phép ta có thêm một property là `props`. Props trong class component và function component hoàn toàn giống nhau về cách sử dụng.

Sử dụng State với class component

Với function component, state có thể được khai báo thông qua hook `useState` đã biết. Tuy nhiên, với class component, state được sử dụng thông qua property `state`

```
import {useState} from 'react'
const Counter = () => {
  const [value, setValue] = useState(0)
  return (
    <div>
      <span>{value}</span>
      <button>Increase</button>
    </div>
  )
}
```

Function component

```
import {Component} from 'react'
class Counter extends Component {
  state = {
    value: 0
  }
  render() {
    return (
      <div>
        <span>{this.state.value}</span>
        <button>Increase</button>
      </div>
    )
  }
}
```

Class component

Tương tự như `props`, `state` trong class component có giá trị dạng key-value. Khác với function component, chúng ta chỉ có một object state duy nhất trong class component. Việc lưu nhiều giá trị state đạt được bằng cách sử dụng những key khác nhau bên trong object `state`.

Về mặt ý nghĩa, State bên trong cả 2 ví dụ trên đều hoạt động giống nhau: khi giá trị state được cập nhật lại, React sẽ thực hiện việc tính toán lại kết quả của component và cập nhật lại giao diện.

Xử lý event và cập nhật state

Để xử lý sự kiện bên trong class component, chúng ta có thể sử dụng method làm event handler

```
class Counter extends Component {
  state = {
    value: 0,
  }
  handleClick = () => {
    this.setState({
      value: this.state.value + 1
    })
  }
  render() {
    return (
      <div>
        <span>{this.state.value}</span>
        <button onClick={this.handleClick}>Increase</button>
      </div>
    )
  }
}
```

Việc xử lý event trong class component cũng khá tương tự với function component. Lưu ý, chúng ta nên sử dụng arrow function đối với các event handler thay vì standard function. Lý do là con trỏ `this` trong standard function sẽ là giá trị của đối tượng thực hiện việc gọi function đó. Do đó, trong class component, nếu chúng ta muốn sử dụng standard function, chúng ta cần thêm một bước xử lý nữa để event handler sẽ nhận giá trị `this` của class. Công đoạn đó thường được gọi là **event handler binding** và thường được viết bên trong class `constructor`:

```
class Counter extends Component { state = { value: 0, } constructor() {
super() this.handleClick = this.handleClick.bind(this) } handleClick() {
console.log("Increase clicked") } render() { return ( <div> <span>
{this.state.value}</span> <button onClick={this.handleClick}>Increase</button>
</div> ) } }
```

Đây là vấn đề của JS, không phải vấn đề của React. Việc này có thể tránh bằng cách sử dụng arrow function thay thế. Arrow function không có con trỏ `this` , do đó, `this` bên trong luôn là `this` của context bao ngoài nó.

Để cập nhật giá trị của state bên trong class component, chúng ta sử dụng một function đặc biệt được kế thừa từ `Component` là `setState` . Cách hoạt động của `setState` khá tương tự với các function update state từ hook `useState` . Điểm khác biệt ở đây là thay vì thay thế giá trị state cũ bằng state mới, `setState` sẽ tiến hành gộp các giá trị state mới và ghi đè các giá trị state cũ.

```
class Counter extends Component { state = { value: 0, } constructor() {
super() this.handleClick = this.handleClick.bind(this) } handleClick() {
this.setState({}) // value will not change } render() { return ( <div> <span>
{this.state.value}</span> <button onClick={this.handleClick}>Increase</button>
</div> ) } }
```

3. Context

Để sử dụng context bên trong một function component, ta có 2 cách sau:

- `<Context.Consumer>` với children là một function (renderProps)
- `useContext` hook

Việc sử dụng `Consumer` là hoàn toàn tương tự với function component bên trong class component. Chúng ta có thể khai báo nhiều consumer của nhiều context provider.

Tuy nhiên, trong class component có một cách khác để xử lý context như sau:

```
import {createContext, Component} from 'react' const ThemeContext =
createContext({currentTheme: "light"}) const App = () => { return (
<ThemeContext value={{currentTheme: "light"}}> <MyComponent /> </ThemeContext>
) } class MyComponent extends Component { static contextType = ThemeContext
render() { return ( <div>Current theme is {this.context.currentTheme}</div> )
} }
```

Trong class component, chúng ta khai báo một static property là `contextType` , nhận vào giá trị là React Context. Sau đó, sử dụng `this.context` bên trong các methods của component. `this.context` sẽ chính là giá trị bên trong context với `contextType` được khai báo tương ứng.

Hạn chế của class component là nó không cho phép có 2 `contextType` trong một component. Do đó, trong các trường hợp cần sử dụng nhiều hơn 2 context, chúng ta cần sử dụng tới `Context.Consumer`

4. Component lifecycle methods

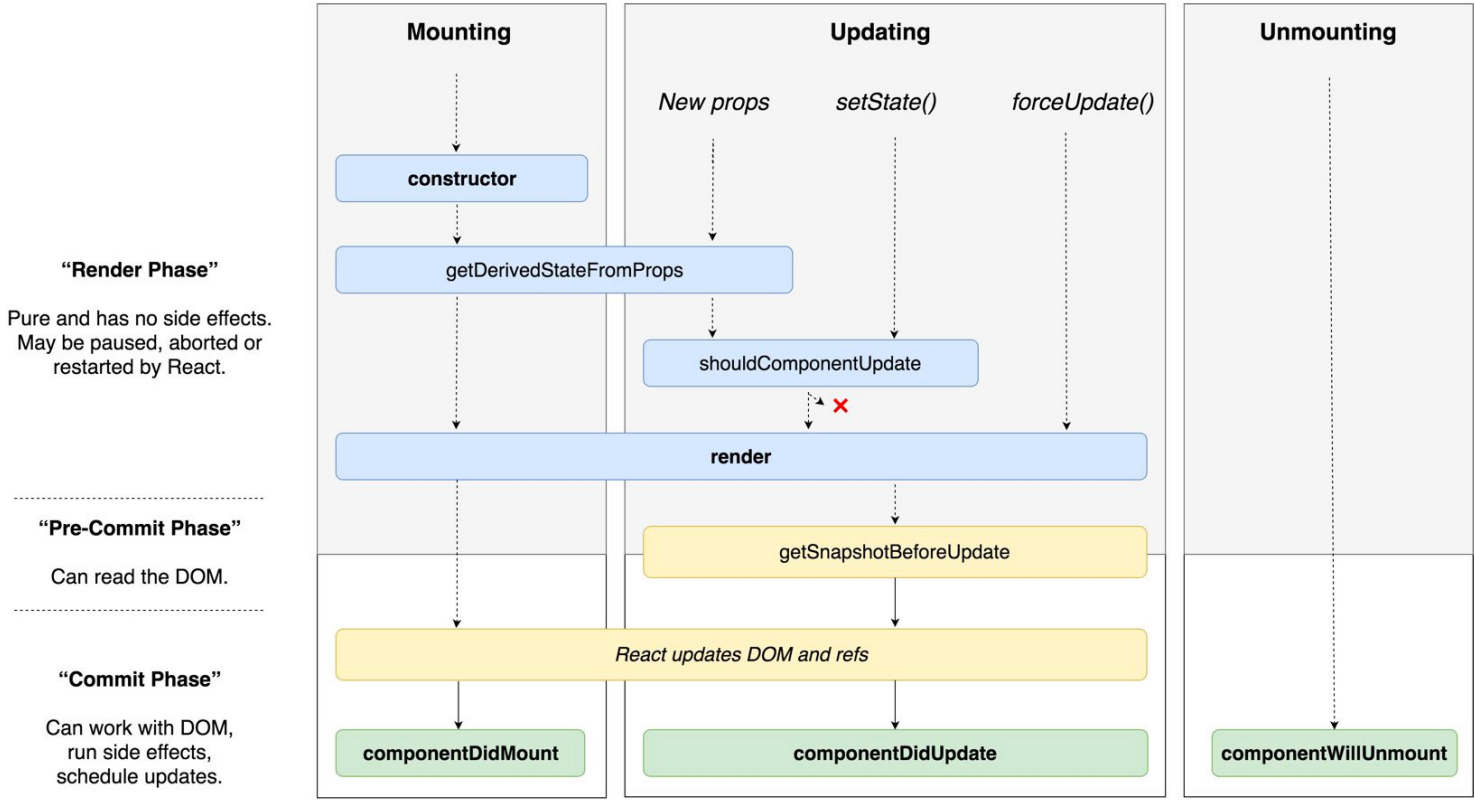
Một component trong ứng dụng React sẽ có 3 trạng thái:

- Mounting**: Thời điểm component bắt đầu được tạo ra và render trên ứng dụng

- **Updating:** Component được cập nhật theo thời gian, thay đổi các state, props, ...
- **Unmounting:** Component bị huỷ, xoá khỏi ứng dụng

Life cycle (vòng đời) methods là những methods bên trong class component cho phép chúng ta thực thi những logic nhất định ở các thời điểm trong vòng đời của component.

💡 Life cycle methods là khái niệm chỉ tồn tại bên trong class component. Đối với function component, chúng ta không có khái niệm này



Các lifecycle bên trong React được phân theo các giai đoạn bao gồm:

Mounting:

- `constructor()` : method đầu tiên được gọi khi component bắt đầu được khởi tạo
- `static getDerivedStateFromProps(props, state)` : method được gọi tiếp theo, sử dụng để tính toán các giá trị state dựa vào state và props hiện tại. Method này cần trả về kết quả là một object để cập nhật state, hoặc `null` nếu không muốn cập nhật state nào xảy ra.
- `render()` : method được gọi để thực hiện trả ra UI cho component
- `componentDidMount()` : method được gọi khi component được trình duyệt render thành công. Method này thường được sử dụng để thực hiện các “side effect” như gọi API, subscribe các dịch vụ, ...

Updating:

- `static getDerivedStateFromProps(props, state)`
- `shouldComponentUpdate(nextProps, nextState)` : method được gọi trước khi component thực hiện việc tính toán lại. Method này thường được sử dụng với mục đích là optimization. Method cần return `true` (component cần tính toán lại) hoặc `false` (không cần tính toán lại)
- `render()`
- `getSnapshotBeforeUpdate(prevProps, prevState)` : method ít dùng, được sử dụng để thực hiện các tính toán ở last-minute. Kết quả tính toán sẽ được gửi xuống cho life cycle tiếp theo.
- `componentDidUpdate(prevProps, prevState, snapshot)` : method được gọi ngay sau khi component được update. Được sử dụng để xử lý các “side effect” khi có sự thay đổi về state hoặc props

Unmounting:

- `componentWillUnmount()` : method được gọi trước khi component bị huỷ, thường được sử dụng cho các mục đích cleanup, teardown

Chi tiết về các lifecycle có thể tham khảo thêm ở đây: <https://reactjs.org/docs/react-component.html>

5. `componentDidCatch`

Trong ứng dụng React, có nhiều trường hợp mà ứng dụng có thể xảy ra lỗi. Đó không hẳn là do code. Đó có thể là những lỗi liên quan tới kết nối, đường truyền, ...

Error: Child crashed!

`Child.render`

`C:/esites/testing/src/components/Child.js:5`

```
2 |  
3 | export default class Child extends Component {  
4 |   render() {  
> 5 |     throw new Error('Child crashed!');  
6 |     return (  
7 |       <div>  
8 |         Child is here
```

Một màn hình lỗi điển hình trong ứng dụng React

Với JS thông thường, chúng ta có thể sử dụng `try-catch` để có thể bắt lỗi và xử lý lỗi nếu muốn. Tuy nhiên, `try-catch` chỉ có thể bắt lỗi bên trong một component. Để có thể bắt lỗi ở nhiều component hơn, chúng ta sử dụng một component đặc biệt với lifecycle đặc biệt là `getDerivedStateFromError` & `componentDidCatch`.

```
import {Component} from 'react' const App = () => { return ( <ErrorBoundary>  
<Counter /> </ErrorBoundary> ) } const Counter = () => { throw new  
Error("Something wrong") return <button>Counter</button> } class ErrorBoundary  
extends Component { state = { hasError: false } static  
getDerivedStateFromError(error) { return { hasError: true }; }  
componentDidCatch(err, errInfo) { console.log(err, errInfo) } render() { if  
(this.state.hasError) { return <div>Something went wrong</div> } return  
this.props.children } }
```