Hooks

Hooks 是什麼

Hooks(勾子) 是 React 16.8(2019/02)加入的新功能。截至 v16.8.6 版本，共有基本的 Hooks 三種，分別是:

- useState - 可在函式型元件中使用 state(狀態)
- useEffect - 可在函式型元件中使用生命周期方法
- useContext - 可在函式型元件中使用 Context API

以及額外的 7 種(應用於特定情況、或是進階使用的)

Hooks 可以透過上述的內建 Hooks，再自訂出針對特定需定的 Hooks，但 Hooks 都是以 `use` 開頭的函式，例如 useFetch、usePromise、useArray 等等，在這個網站 - Collection of React Hooks上有維護一個目前可使用的自訂 Hooks 清單。

為什麼要使用 Hooks

元件中的 狀態的邏輯(stateful logic) 很難被重覆利用

目前的情況都是使用例如 HOC(高階元件)或是 render props 之類的樣式來重覆利用，但大量的 HOC、providers、render props 會造成層層包裝的結構。

結論: Hooks 可以在不更動元件階層的情況下重覆利用狀態的邏輯。

當功能愈來愈複雜時，元件的程式碼難以維護和拆分

元件的生命周期有可能會有相當複雜的應用情況，有各種不同的副作用與狀態邏輯，很容易造成 bugs 或不連續性的問題。

結論: Hooks 可以拆分一個元件功能到小的幾個函式中。

Classes(類別)難以被理解，不論是開發者還是電腦

React 中的類別容易造成學習者一開始的學習障礙，主要原因是 JavaScript 中的 this 與其它語言的特性不太一樣。此外類別也不容易進行最佳化，它有很多語法對電腦(編譯工具、最佳化工具…)來說可能也有理解上的問題。

結論: Hooks 可以讓開發者使用大部份的 React 功能，但不需要使用類別。

> 註: 極端一點來說，學習者不需要再煩惱 this 的學習和使用的問題

Hooks 兩個使用基本準則

只能在最上層呼叫 Hooks

你不能在迴圈(loops)、控制流程(conditions)、或是巢狀函式裡呼叫 Hooks。

> 註: 如果你需要控制流程的話，應該在 Hooks 函式的程式碼裡面使用它，而不是在外面。

只能在 React 函式型元件中呼叫 Hooks

你不能在一般的 JS 函式中呼叫 Hooks，你只能在以下情況使用它:

- React 函式型元件中呼叫 Hooks (以大駝峰命名的函式-PascalCase function)
- 在其它自訂的 Hooks 中呼叫 Hooks (以 use 開頭命名的函式)

ESLint Plugin

安裝:

```
# npm
npm install eslint-plugin-react-hooks --save-dev

# yarn
yarn add eslint-plugin-react-hooks --dev
```

設定:

```
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error",
    "react-hooks/exhaustive-deps": "warn"
  }
}
```

useState

基本使用範例

```
import React, { useState } from 'react'

function Example() {
  // 宣告一個新的 state(狀態) 變數，名稱為 "count"
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

相當於以下的 ES6 類別元件的例子:

```
import React from 'react'

class Example extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      count: 0,
    }
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1
})}>
          Click me
        </button>
      </div>
    )
  }
}
```

語法說明

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

這段語法是 陣列"解構賦值"(Destructuring Assignment)的語法。相當於下面的語法(經 babel 編譯後):

```
// const [count, setCount] = useState(0);
var _useState = useState(0),
  count = _useState[0],
  setCount = _useState[1]
```

所以 useState 基本的語法樣式如下:

```
import { useState } from 'react'
//...
const [value, setValue] = useState('')
```

至於原本在 ES6 類別型元件中讀取 state 值的動作，會由直接存取 useState 中的變數名稱來取代，也就是:

```
<p>You clicked {this.state.count} times</p>
```

改為

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

更新的部份則是原本用`this.setState()`, 改為使用`setCount`函式

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Click me
</button>
```

改為

```
<button onClick={() => setCount(count + 1)}>Click me</button>
```

各種使用情況範例

多個狀態值(變數)

特別注意：一個狀態宣告一行。React 會認為每行的執行有順序之分。

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42)
  const [fruit, setFruit] = useState('banana')
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }])
  // ...
}
```

表單元件

```
import React, { useState } from 'react'
import TextField from '@material-ui/core/TextField'

const TodoForm = ({ saveTodo }) => {
  const [value, setValue] = useState('')
  return (
    <form
      onSubmit={event => {
        event.preventDefault()
```

```
        saveTodo(value)
      }}
    >
      <TextField
        variant="outlined"
        placeholder="Add todo"
        margin="normal"
        onChange={event => {
          setValue(event.target.value)
        }}
        value={value}
      />
    </form>
  )
}
export default TodoForm
```

state 為陣列

```
import React, { useState } from 'react'
//...
const App = () => {
  const [todos, setTodos] = useState([])

  // return ...
}
```

useEffect

useEffect 是給函式型元件使用的泛用生命周期套用 Hooks。可以因應各種使用情況，組合 useState 來開發自訂的 Hooks。

基本範例

原本的 ES6 類別元件的範例(來源網站):

```
class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.state = { date: new Date() }
  }

  componentDidMount() {
    this.timerID = setInterval(() => this.tick(), 1000)
  }

  componentWillUnmount() {
    clearInterval(this.timerID)
```

```
  }

  tick() {
    this.setState({
      date: new Date(),
    })
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    )
  }
}

ReactDOM.render(<Clock />, document.getElementById('root'))
```

使用 useEffect 後的函式型元件的範例:

```
function Clock(props) {
  const [date, setDate] = useState(new Date())

  //取代 componentDidMount 與 componentWillUnmount
  useEffect(() => {
    var timerID = setInterval(() => tick(), 1000)

    return function cleanup() {
      clearInterval(timerID)
    }
  })

  function tick() {
    setDate(new Date())
  }

  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {date.toLocaleTimeString()}.</h2>
    </div>
  )
}
```

上述的例子可以額外寫出一個自訂的 Hooks:

```
function Clock(props) {
  //Here we reference our custom hook
  const timer = useNewTimer(new Date())

  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {timer.toLocaleTimeString()}.</h2>
    </div>
  )
}

////////////////////////////////////////////
//以下撰寫出一個自訂的、可重覆使用的hook
////////////////////////////////////////////

import { useState, useEffect } from 'react'

function useNewTimer(currentDate) {
  const [date, setDate] = useState(currentDate)

  useEffect(() => {
    var timerID = setInterval(() => tick(), 1000)
    return function cleanup() {
      clearInterval(timerID)
    }
  })

  function tick() {
    setDate(new Date())
  }

  return date
}

export { useNewTimer }
```

語法說明

useEffect 與下面三種生命周期方法有關，也就是會在某個時間點要執行的程式碼，其中第二個 componentDidUpdate 是在元件更新的階段使用:

- componentDidMount
- componentDidUpdate
- componentWillUnmount

useEffect 預設的行為模式，是在完整的 渲染(render) 之後被觸發執行。但這個行為可以透過第二個傳入參數調整，例如以下的程式碼範例:

```
useEffect(() => {
  const subscription = props.source.subscribe()
  return () => {
    subscription.unsubscribe()
  }
}, [props.source])
```

`[props.source]`會指定 useEffect 只有在`props.source`有更動時，才會觸發執行其中的程式碼。

如果你在這(第二個傳入參數)指定一個空陣列`[]`，就會讓 useEffect 因為不依賴任一個 props 值，所以只會執行一次(在 mount 與 unmount 時)，這會讓的 useEffect 行為很接近 componentDidMount 與 componentWillUnmount。

useEffect 的第一個傳入參數是一個 callback 函式，而這個 callback 函式的回傳，通常也是一個函式，這個函式稱為 clean-up(清掃、收尾)函式，這個函式對應的通常就是 componentWillUnmount 這個生命周期方法。

clean-up(清掃、收尾)函式不一定是必要的，如果你只是要每次更新(state 或傳入的 props 有更動)就要執行一次的話，並不需要回傳另一個函式，例如以下的範例:

```
function Example() {
  const [count, setCount] = useState(0)
  useEffect(() => {
    document.title = `You clicked ${count} times`
  })

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

對照原本的 ES6 類別中的生命周期方法，有可能你要在某個生命周期方法中作多件事情，然後會讓程式碼擠成一團。

useEffect 與 useState (或其它的 Hooks)都是採用分離的概念來撰寫，作一件事就寫一段這樣，以下面的例子來說明。

原本的 ES6 類別元件，你可以看到在同一個生命周期方法中會作多件事情，例如 componentDidMount 裡面除了要設定`document.title`外，也要作 ChatAPI 的訂閱動作:

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
```

```
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

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

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }
  // ...
```

useEffect 的作法會拆開成一件事情用一個 useEffect 來作，React 會依照它們的順序依次來執行:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0)
  useEffect(() => {
    document.title = `You clicked ${count} times`
  })

  const [isOnline, setIsOnline] = useState(null)
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline)
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange)
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange)
    }
  })
  // ...
}
```

各種使用情況範例

useFetch

```
// hooks.js
import { useState, useEffect } from 'react'

function useFetch(url) {
  const [data, setData] = useState([])
  const [loading, setLoading] = useState(true)
  async function fetchUrl() {
    const response = await fetch(url)
    const json = await response.json()
    setData(json)
    setLoading(false)
  }
  useEffect(() => {
    fetchUrl()
  }, [])
  return [data, loading]
}

export { useFetch }
```

参考資料

- https://reactjs.org/docs/hooks-intro.html
- https://overreacted.io/a-complete-guide-to-useeffect/
- https://medium.com/@dan_abramov/making-sense-of-react-hooks-fdbde8803889
- https://www.robinwieruch.de/react-hooks-fetch-data/
- https://itnext.io/how-to-create-react-custom-hooks-for-data-fetching-with-useeffect-74c5dc47000a
- https://medium.freecodecamp.org/how-to-build-a-todo-list-with-react-hooks-ebaa4e3db3b
- https://usehooks.com/
- https://nikgraf.github.io/react-hooks/
- https://medium.com/the-guild/under-the-hood-of-reacts-hooks-system-eb59638c9dba
- https://www.netlify.com/blog/2019/03/11/deep-dive-how-do-react-hooks-really-work/