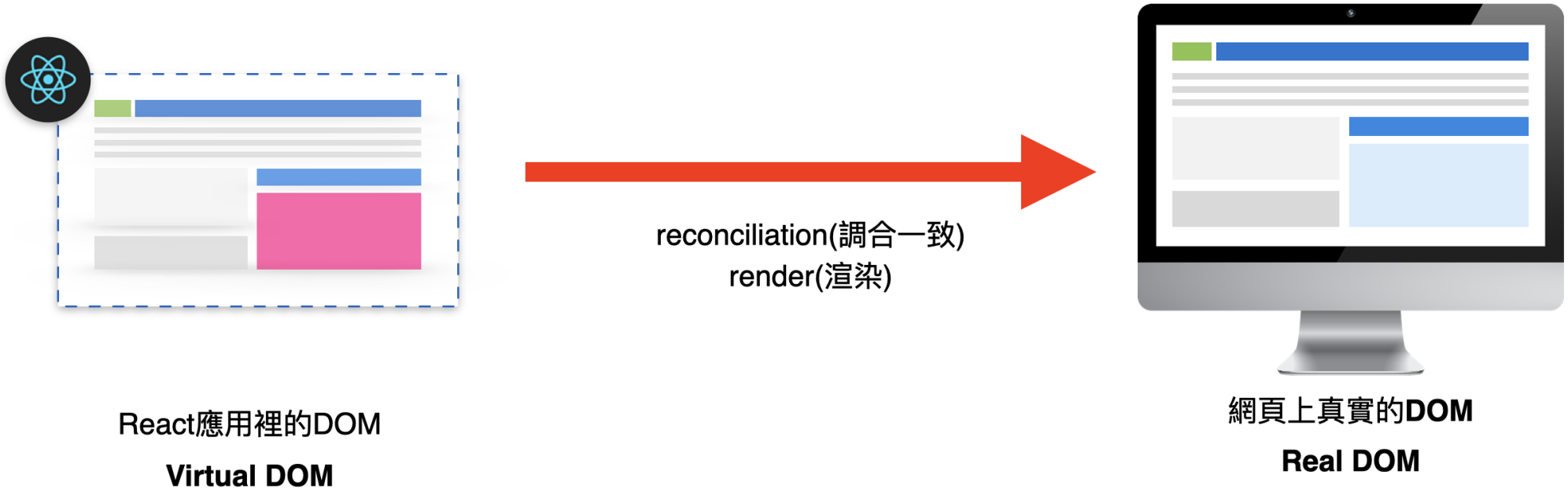
State 狀態

Eddy Chang

✉️ hello@eddychang.me

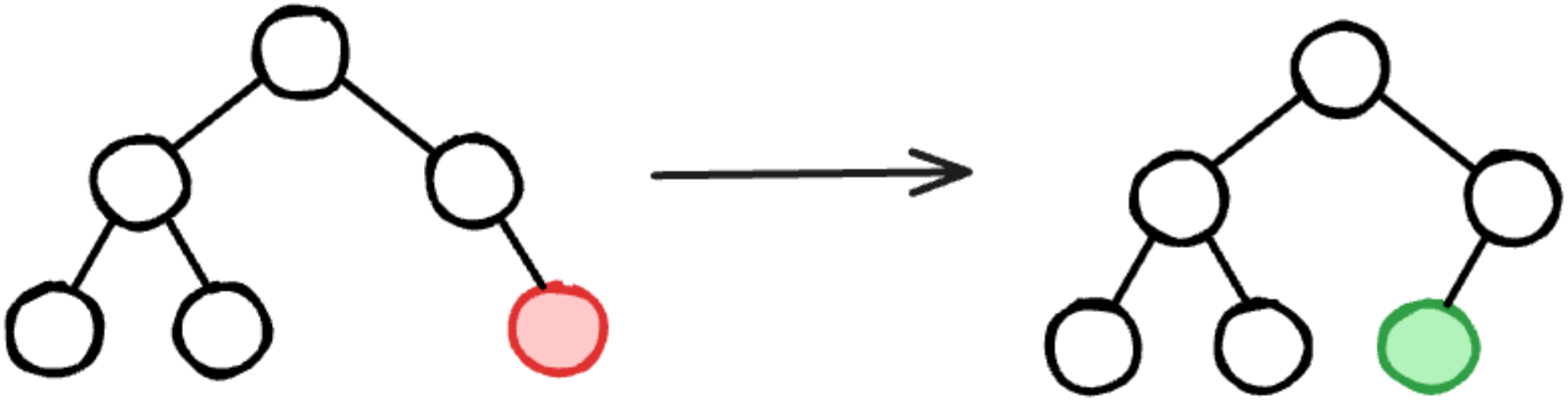
Virtual DOM(虛擬 DOM)

React 元件中自行管理的 DOM 結構，用於差異比較後再渲染到真實 DOM。



Virtual DOM(虛擬 DOM)

> ❓ 問題 1: 為何要讓 React 管理網頁 DOM 結構，而不是由開發者來控制管理？
> ❓ 問題 2: React 是怎麼改變網頁 DOM 結構的？

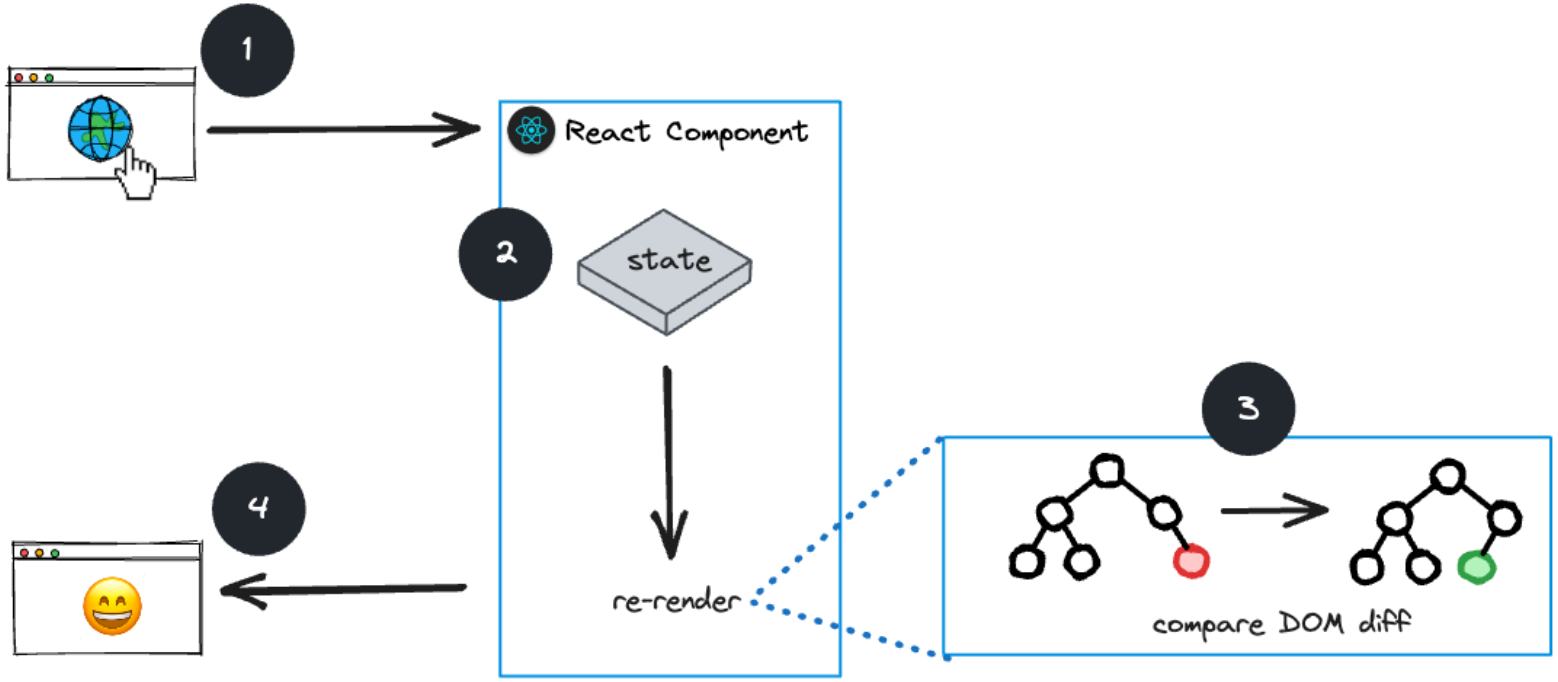


> 💡 React 不會比直接 DOM 處理更快，它只是協助開發者建立可維護的應用程式，而且"足夠快速"的進行 DOM 處理

State 狀態

問題 2: React 是怎麼改變網頁上的 DOM 結構的？

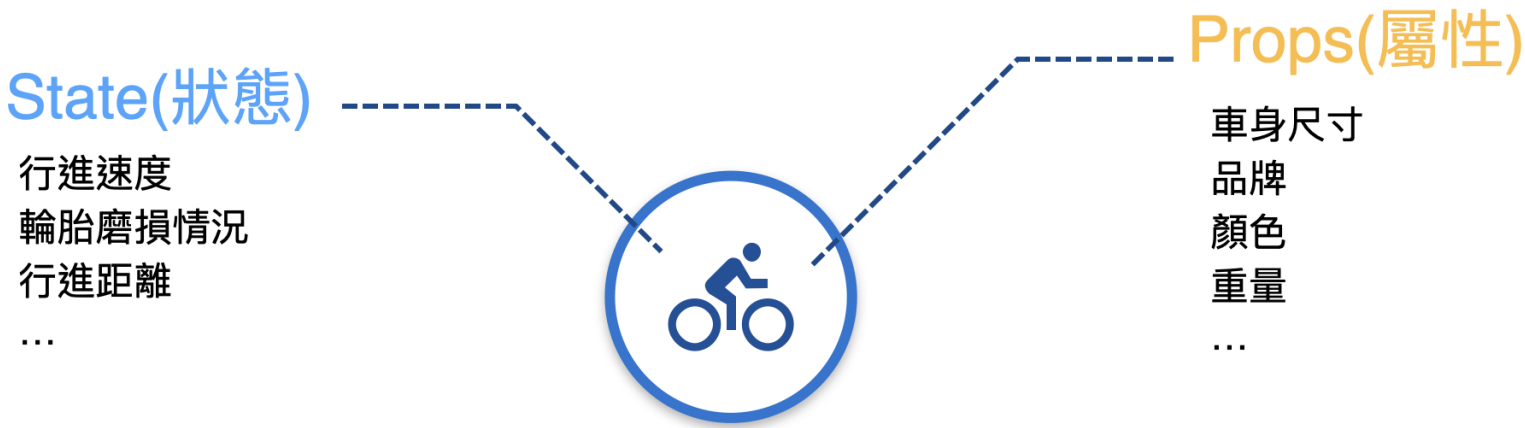
答案: **依靠 State 狀態改變**。狀態是元件內部的私有資料(變數)，只有透過改變狀態(setState)，才能改變真實網頁上的 DOM



CSR SSG SSR

State 狀態

> 是會因為使用者操作後，而不斷改變的資料值(變數值)



CSR SSG SSR

Render 渲染

> 註: Google 翻譯為"算繪"，微軟翻譯為"轉譯"。電腦繪圖術語常翻為"彩現、繪製"

瀏覽器中的渲染引擎(Rendering Engine)

將 HTML 原始碼轉譯為使用者所看到的網頁樣貌的過程，此為瀏覽器核心功能。例如把 `<img src="sample.jpg"/>` 呈現為可見到的圖片的過程。

React 中的渲染(Render)

更新虛擬 DOM，以及把虛擬 DOM 轉為真實 DOM 的過程。一整個 render 過程，應分為專司更新虛擬 DOM 的 render 階段，以及與瀏覽器交互進行更動真實 DOM 的 commit 階段。Render 和 Commit

Immutability 不可改變性

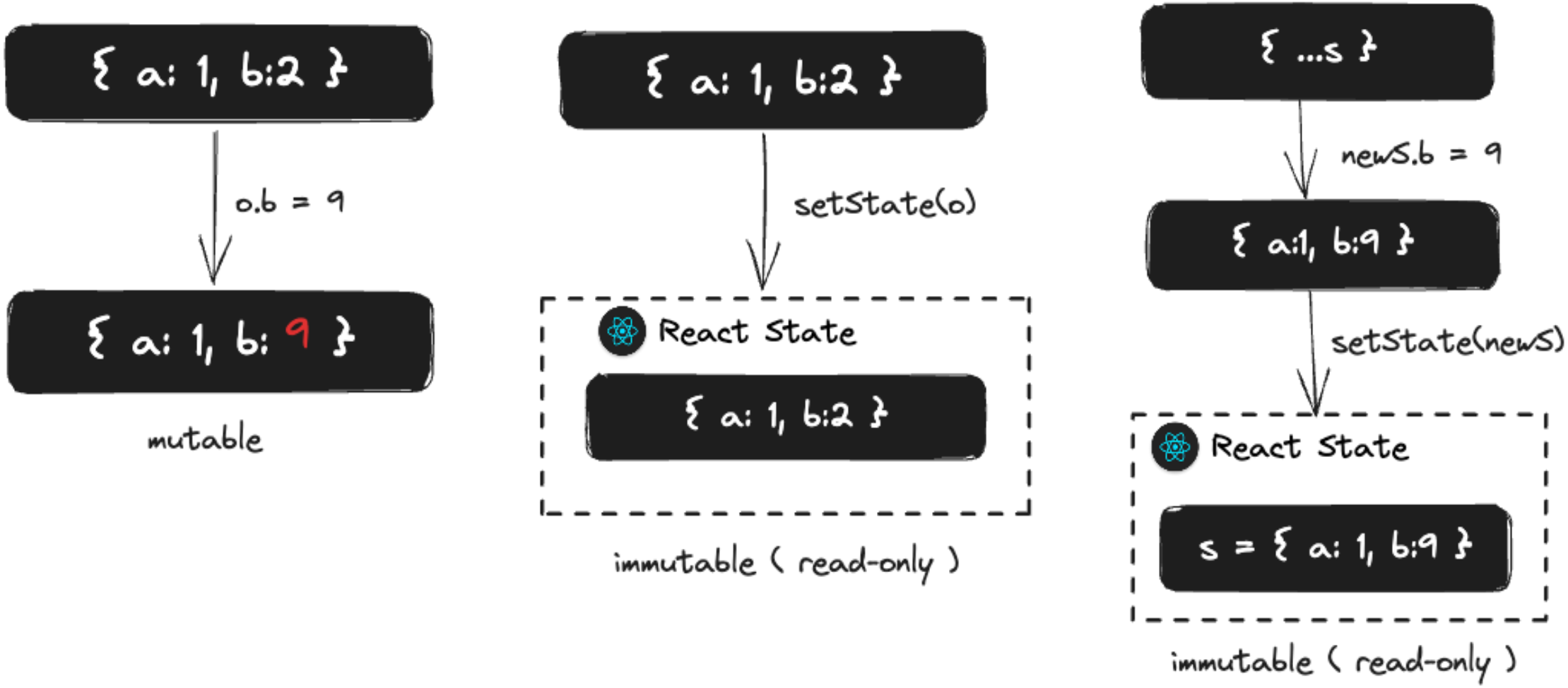
所有 React 中的狀態(state)都是不可改變的(immutable)

一旦某個值成為 state(狀態)後，它的值將無法直接改變。即使是物件的屬性值也要視為被凍結(freeze)或唯讀(read-only)不可改變。`state` 的任何改變都要透過`setState` 傳入一個全新的值/物件。

> 註: `immutable` 與 `unchangeable` 同義，都是**不可改變的**的意思

> 註: 實際上對元件而言，它接收到的 props(屬性)也是不可變的，不過較精確的說法是唯讀(read-only) 的純函式(pure function)特性

Immutability 不可改變性 - 圖解



為何 React 中的狀態(state)設計成"不可改變的"?

1. **易於除錯**: 開發者可以很清楚的比對 state 的前後改變
2. **最佳化策略**: React 的最佳化策略是用比對 state 的前後修改部份，決定是否要進行渲染或略過工作
3. **新特性(快照)**: React 中的 state 行為更像一份快照(snapshot)，而非一般的 JS 變數。每次元件函式回傳的 JSX 就像是一張 UI 的即時快照。它其中的一切都是利用當下進行渲染時的 state 計算出來的
4. **更動的需求**: 類似 `重作/復原` 的功能會更容易實作
5. **實作更簡單**

> 來源: react.dev

純函式(pure function)

一個函式如果符合以下的兩個原則，則稱為純函式(pure function)

1. **函式的輸出(返回值)只依賴輸入(傳入參數值)**。(不會變動或影響到任何本地端靜態變數、非本地端變數、可改變的參數或輸入變量流)
2. **函式沒有副作用(side effects)** (不會變動或影響到任何本地端靜態變數、非本地端變數、可改變的參數或輸出入變量流)

> 來源: 維基百科

- 只專注在它自己該作的事(It minds its own business)
- 給定相同輸入，必定得到相同輸出(Same inputs, same output)

> 來源: react.dev

純函式(pure function) 範例

> "純(pure)函式"範例:

```
function sum(a, b) {
  return a + b
}
```

> "不純(impure)函式"範例:

```
let guest = 0

function addGuest(a) {
  guest = guest + a
  return guest
}
```

純函式(pure function) 優點

- **可閱讀性(Readable)**: 程式碼容易閱讀
- **可重覆使用性(Reusable)**: 函式封閉與固定強健性高，可重覆使用性高，可組合、可移植性高
- **可預測性(Predictable)/可測試性(Testable)**: 輸出只相依輸入，可預測高，相對容易進行單元測試與除錯
- **記憶樣式(Memoization)**: 使用快取(緩存)或記憶樣式，在高花費的應用中，可優化運算效能

React 中關於純函式(pure function)

- React 中的元件就是"純函式"
- 在 React 元件中要處理有副作用的程序，要在事件處理函式中或使用 `useEffect` 勾子，特別獨立出來處理

> 元件必須是純函式 (A component must be pure) ~react.dev

> 所有 React 元件必須像純函式一樣的顧慮它們的屬性(props)值 (All React components must act like pure functions with respect to their props.) ~reactjs.org

副作用的程度差異

> 什麼是副作用(Side Effect)

- 隱性/不預期的: 值相等(==)運算、JS 內部轉型機制...etc
- 輕度: console.log/alert/Date/random
- 中度(通常): Ajax/FetchAPI/Timer/Promise/Generator
- 重度:它多個同步與異步混合的控制流程

`setState`

> 指的是使用用 `useState` 定義出的 `setXXX` 方法

1. **唯一**能更動 `state` (狀態) 的方法
2. ⚠️**不可**直接用 JS 的指定值語法來更動(ex. `s = 1`, `s++`, `s += 1` 等都不行)
3. 更動網頁上的 UI -> `state` (狀態) 需要被改變 -> 要呼叫 `setState` 方法
4. `setState` 方法的執行，會有異步(非同步)程序執行特性，使用時要注意程式執行順序與邏輯

useState 勾子(hooks)

```
// total - getter 獲得值的變數 (state)
// setTotal - setter 設定值的方法 (setState)
// useState中的傳入參數 - 狀態的初始值
const [total, setTotal] = useState(0)
```

⚠️ 使用注意

1. `state` 與 `setState` 務必按命名規則來命名(小駝峰，設定方法要有 `set` 開頭)
2. 一定要設定初始值給 `useState` 方法
3. 在應用執行期間都保持 `state` (狀態)的資料類型一致
4. 陣列初始值通常是 `[]` 。但物件初始值通常不會用 `null` 或 `{}` 它們只用於特殊情況

useState 範例

```
import { useState } from 'react'

export default function Counter() {
  // 宣告state(狀態)
  const [count, setCount] = useState(0)

  // 事件處理函式
  const updateCount = () => {
    setCount(count + 1)
  }

  // 相當於render方法
  return <button onClick={updateCount}>Count is: {count}</button>
}
```

更動物件/陣列狀態

1. 將 React 中的所有狀態視為不可改變的(immutable)
2. 你不能直接對物件/陣列中的值作更動，取而代之的是建立一個物件/陣列的新版本，之後設定給狀態
3. 可使用展開運算子進行物件狀態的拷貝，例如 `{...obj, someKey: 'newValue'}`，但它是淺層的(shallow)，意即只能拷貝一層深度
4. 當要更新巢狀物件/陣列時，你需要建立所有更新處以上的所有值，它們的拷貝複本(客製化+深拷貝)
5. 為了減少繁瑣的拷貝程式碼或更精確的目的，建議使用 immer

更動陣列(Array)狀態

-	🚫 避免(直接修改陣列)	✅ 推薦(回傳新陣列)
新增(add)	`push`, `unshift`	`concat`, `[...arr]`
移除(remove)	`pop`, `shift`, `splice`	`filter`, `slice`
更動(replace)	`splice`, `arr[i] = ...`	`map`
排序(sort)	`reverse`, `sort`	先拷貝整個陣列(深拷貝)再處理

> 資料來源: react.dev

更動陣列(Array)狀態 速查表-1

```
// 宣告與初始化 ex. todos =[{id, title},...]
const [todos, setTodos] = useState([])

// 新增
const addTodo = (todo) => {
  setTodos([...todos, todo])
}

// 移除
const deteleTodo = (id) => {
  const newTodos = todos.filter((todo) => {
    return todo.id !== id
  })

  setTodos(newTodos)
}
```

更動陣列(Array)狀態 速查表-2

```
// 更新其中的物件屬性
const updateTodoTitle = (id, newTitle) => {
  const newTodos = todos.map((todo) => {
    if (todo.id === id) {
      return { ...todo, title: newTitle }
    }

    return todo
  })

  setTodos(newTodos)
}
```

更動物件(Object)狀態 速查表-1

```
// 宣告與初始化 ex. user = {id, name, age}
const [user, setUser] = useState({ id: 0, name: '', age: 0 })

// 加入新屬性
const addPhone = (phone) => {
  setUser({ ...user, phone })
}

// 更新
const updateName = (newName) => {
  setUser({
    ...user,
    name: newName,
  })
}
```

更動物件(Object)狀態 速查表-2

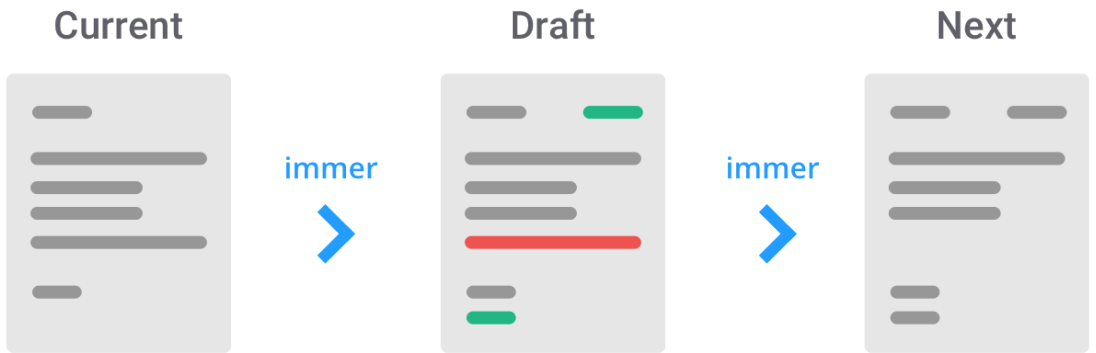
```
// 移除某屬性
const removeAge = () => {
  // ES6"其餘運算子"的語法
  const { age, ...rest } = user
  setUser(rest)
}

// 移除某屬性-2
const removeAge = () => {
  const newUser = { ...user }
  delete newUser[age]
  setUser(user)
}
```

immer

> 可以讓開發者用更簡便的方式，來處理不可改變的狀態(immutable state)

1. Redux Toolkit 內建，所以要學/用 Redux 時是必學的
2. 各種語法均為直接處理狀態的。實際上是在 Proxy(代理)狀態(或稱為 draft 物件)上處理(如圖)，不要搞混亂了
3. 如果狀態結構很簡單，直接用前述的處理語法即可，不一定要用 immer



immer 使用方式

1. 安裝:

```
npm install immer
```

1. 導入 `produce` 在程式碼中:

```
import { produce } from 'immer'
```

3. 語法: API 文件

```
produce(baseState, recipe: (draftState) => void): nextState
```

> 註: 也可用 `useImmer` 來取代 `useState` ，但需要另外安裝use-immer

immer 更動物件(Object)狀態 - 速查表

```
const todosObj = { id1: { done: false, body: '倒垃圾' } }

// 新增 add
const addedTodosObj = produce(todosObj, (draft) => {
  draft['id3'] = { done: false, body: '買 bananas' }
})
// 刪除 delete
const deletedTodosObj = produce(todosObj, (draft) => {
  delete draft['id1']
})
// 更動 update
const updatedTodosObj = produce(todosObj, (draft) => {
  draft['id1'].done = true
})
```

immer 更動陣列(Array)狀態 - 速查表

```
const todos = [{ id: 'id1', done: false, body: '倒垃圾' }]
// 新增(最後面) add (加入在最前面用 `unshift`)
const addedTodos = produce(todos, (draft) => {
  draft.push({ id: 'id3', done: false, body: '買香蕉' })
})
// 刪除某索引 delete by index
const deletedTodos = produce(todos, (draft) => {
  draft.splice(3 /* 索引 */, 1)
})
// 更新某索引 update by index
const updatedTodos = produce(todos, (draft) => {
  draft[3].done = true
})
// 插入某索引後 insert at index
const updatedTodos = produce(todos, (draft) => {
  draft.splice(3, 0, { id: 'id3', done: false, body: '買香蕉' })
})
```

```
// 移除最後一個項目 remove last item (要移除最前一個項目用 `shift`)
const updatedTodos = produce(todos, (draft) => {
  draft.pop()
})
// 移除某id項目 delete by id
const deletedTodos = produce(todos, (draft) => {
  const index = draft.findIndex((todo) => todo.id === 'id1')
  if (index !== -1) draft.splice(index, 1)
})
// 更動某id項目 update by id
const updatedTodos = produce(todos, (draft) => {
  const index = draft.findIndex((todo) => todo.id === 'id1')
  if (index !== -1) draft[index].done = true
})
// 過濾項目 filtering items (配合filter使用)
const updatedTodos = produce(todos, (draft) => {
  return draft.filter((todo) => todo.done)
})
```

setState 異步執行

> 狀態在進行改變時，React 會進行合併與最佳化等內部工作，必會有異步執行特性

```
<h1
  onClick={() => {
    setCount(count + 1) // 改變count狀態(!!異步!!)
    console.log(count) // 這裡得不到改變後的count
  }}
>
  {count}
</h1>
```

setState 異步執行的 - 對應策略 1 ✏️推薦！必學！

> **先用變數計算出最後的值**。意即完全不依賴 setState 執行後改變的 state 值。

```
<h1
  onClick={() => {
    //先行計算出值的最後會變為什麼
    const newCount = count + 1
    setCount(newCount) // 改變count狀態(!!異步!!)
    console.log(newCount) // 這裡用自己計算的值
  }}
>
  {count}
</h1>
```

CSR SSG SSR

setState 異步執行的 - 對應策略 2 ✏️必學！

> 使用 `useEffect` 勾子，它有類似於 state 的 change 事件監聽機制，可在每次 state 更動完成後觸發執行

```
useEffect(() => {
  console.log(count) // 這裡用可以得到count變動後的值
},[count])
// ^^^^^ 需要將相依的狀態變數放在此處，會在有變動(change)後(after)觸發執行包含在其中的程式碼

<h1
  onClick={() => {
    setCount(count + 1) // 改變count狀態(!!異步!!)
  }}
>
  {count}
</h1>
```