RESTFUL API

林新德

shinder.lin@gmail.com

參考專案 https://bitbucket.org/lsd0125/mfee43-next.git

0. 參考專案

```
# 本課程後端參考
https://bitbucket.org/lsd0125/mfee36-node.git

# 本課程前端參考
https://bitbucket.org/lsd0125/mfee36-next.git
```

```
# 網站使用 Google 帳號登入 (後端)
https://github.com/shinder/signin-google-try.git
```

1. Restful API

- REST為Representational State Transfer（表現層狀態轉換）的縮寫。
- 2000年由 Dr. Roy Thomas Fielding 在其博士論文中提出的 HTTP 資料交換風格。

1.1 什麼是 Restful API

- 要點：

  1. 以 URI 指定資源，使用 HTTP 或 HTTPS 為操作協定。
  2. 透過操作資源的表現形式來操作資料。
  3. 就是以 HTTP 的 GET, POST, PUT, DELETE 方法對應到操作資源的 CRUD。
  4. 資源的表現形式沒有限定，可以是 HTML, XML, **JSON** 或其它格式。
  5. REST 是設計風格並不是標準，所以沒有硬性的規定。

- 實作 REST 的 後端 API 一般稱作 Restful API

1.2 以商品資料為說明

- 取得列表：
- http://my-domain/products (GET)
- 取得單項商品：
- http://my-domain/products/17 (GET)
- 新增商品：
- http://my-domain/products (POST)
- 修改商品：
- http://my-domain/products/17 (PUT)
- 刪除商品：
- http://my-domain/products/17 (DELETE)

1.3 管理端 URI (需要呈現表單)

- 呈現新增商品的表單：
- http://my-domain/products/add (GET)
- 呈現修改商品的表單：
- http://my-domain/products/17/edit (GET)
- 呈現刪除商品的表單：
- http://my-domain/products/17/delete (GET)

1.4 API 實作

- 商品 API：
  - 1. 列表
  - 2. 搜尋
  - 3. 單項商品

- 購物車 API：
  - 1. 讀取
  - 2.加入
  - 3.移除
  - 4.修改
  - 5.清空

2 前端 AJAX

- XMLHttpRequest
- Fetch API
- Axios（套件）
- jQuery.ajax（jQuery 附屬功能）

2.1 XMLHttpRequest

- XMLHttpRequest 屬性及方法參考： https://developer.mozilla.org/zh-TW/docs/Web/API/XMLHttpRequest
- 事件：onreadystatechange、onload、onerror、onprogress 、onabort 。
- 範例：https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest
- 必須在呼叫 open() 方法開啟請求連線之前，就註冊好事件監聽器，否則事件監聽器將不會被觸發。
- 可選擇同步或非同步。
- 上傳進度，使用XMLHttpRequest.upload 物件的事件： load、error、progress 、abort 。

ajax-xhr-01.html

```
function doAjax() {
    var xhr = new XMLHttpRequest();

    xhr.onreadystatechange = function (event) {
        console.log(xhr.readyState, xhr.status);
        console.log(xhr.responseText);
        if(xhr.readyState===4 && xhr.status===200){
            info.innerHTML = xhr.responseText;
        }
    };
    // xhr.open('GET', 'data/sales01.json', true); // 非同步
    xhr.open('GET', 'data/sales01.json', false);   // 同步
    // XMLHttpRequest.open(method, url[, async[, user[, password]]])
    xhr.send();
}
```

2.2 Fetch API

- 參考 https://developer.mozilla.org/zh-TW/docs/Web/API/Fetch_API/Using_Fetch
- 使用 Promise 包裝的 API 。
- 只有網路錯誤或其他會中斷 request 的情況下，才會發生 reject 。
- 缺點：無法取得上傳或下載的進度過程。
- 在傳送少量資料時，是方便的選擇。
- 不會主動傳送 cookie，需設定 credentials 為 include。
- 放棄任務使用 AbortController 的 signal。

fetch: 使用 POST 方法

```
const url = 'https://example.com/profile';
const data = {username: 'example'};

fetch(url, {
    method: 'POST',
    body: JSON.stringify(data),
    headers: new Headers({
        'Content-Type': 'application/json'
    })
}).then(res => res.json())
    .catch(error => console.error('Error:', error))
    .then(response => console.log('Success:', response));
```

fetch: 上傳檔案及表單

```
const formData = new FormData(document.myForm);

fetch('https://example.com/profile/avatar', {
    method: 'PUT',
    body: formData
})
    .then(response => response.json())
    .catch(error => console.error('Error:', error))
    .then(response => console.log('Success:', response));
```

2.3 Axios

- Axios 工具: https://www.npmjs.com/package/axios

- Axios優點

  1. 方便使用，類似 jQuery 的 AJAX 方法
  2. Promise API 包裝
  3. 可以在後端 Node.js 中使用
  4. 體積輕量

axios: 上傳檔案及表單

```
axios.post('/try-upload3', formData, {
    headers: {
        'Content-Type': 'multipart/form-data'
    },
    onUploadProgress: function(progressEvent){
        const perc = Math.round(progressEvent.loaded/progressEvent.total*100);
        const t = new Date().toLocaleString();
        const str = `${perc} % : ${t}`;
        console.log(str);
        progress.innerHTML += str+'<br>';
    }

}).then(r=>{
    console.log(r.data);
    console.log(new Date());
});
```

3. 使用 JSON Web Token

- JSON Web Token 社群官網 https://jwt.io/
- 使用 https://www.npmjs.com/package/jsonwebtoken 套件。
- 先決條件：資料傳送過程必須在加密的環境中使用，如 HTTPS
- 優點：可在不同的用戶端環境使用，不局限於網站。
- 缺點：需存放在用戶端，由 JavaScript 發送，或其他前端技術發送。
- 過期時間 exp，也可以使用套件的設定 expiresIn。
- （使用 bcryptjs 套件加密用戶密碼）

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

3.1 JWT 編碼解碼

```
// 成功登入，將可以辨別用戶的資料加密為 JWT（編碼）
const token = jwt.sign({
    sid: r1[0].sid,
    account: r1[0].account,
}, process.env.JWT_SECRET);
```

```
const auth = req.get('Authorization');
res.locals.loginUser = null;
if(auth && auth.indexOf('Bearer ')===0){
    const token = auth.slice(7);
    // JWT 解密
    res.locals.loginUser = jwt.verify(token, process.env.JWT_SECRET);
}
```

3.2 登入功能

後端

```
app.post('/login', async (req, res)=>{
  const output = {
    success: false,
    code: 0,
    error: ''
  };
  if(! req.body.email || !req.body.password){
    output.error = '欄位資料不足'
    return res.json(output)
  }

  const sql = "SELECT * FROM members WHERE email=?";
  const [rows] = await db.query(sql, [req.body.email]);
  if(! rows.length) {
    // 帳號是錯的
    output.code = 402;
    output.error = '帳號或密碼錯誤'
    return res.json(output)
  }
```

```
 const verified = await bcrypt.compare(req.body.password, rows[0].password);
  if(!verified){
    // 密碼是錯的
    output.code = 406;
    output.error = '帳號或密碼錯誤'
    return res.json(output)
  }
  output.success = true;
  // 包 jwt 傳給前端
  const token = jwt.sign({
    id: rows[0].id,
    email: rows[0].email
  }, process.env.JWT_SECRET);
  output.data = {
    id: rows[0].id,
    email: rows[0].email,
    nickname: rows[0].nickname,
    token,
  }
  res.json(output)
});
```

登入功能前端

```
function sendData(e) {
  e.preventDefault();
  fetch("/login", {
    method: "POST",
    body: JSON.stringify({
      email: document.form1.email.value,
      password: document.form1.password.value,
    }),
    headers: {
      "Content-Type": "application/json",
    },
  }).then((r) => r.json()).then((data) => {
    if(data.success){
        const obj = {...data.data}
        localStorage.setItem('auth', JSON.stringify(obj))
        alert('登入成功')
    } else {
        alert(data.error || '帳密錯誤')
    }
  });
}
```

3.3 Token 以 headers 傳送

用戶端 Demo: 已經在 localStorage 存有 token 時

```
const jwtData = JSON.parse(localStorage.getItem('auth'));

function send(){
    fetch('/address-book/verify2',{
        headers: {
            Authorization: 'Bearer ' + jwtData.token
        }
    })
        .then(r=>r.json())
        .then(obj=>{
            console.log(obj);
        })
}
```

伺服端 ExpressJS Middleware Demo

```
app.use((req, res, next)=>{
    res.locals.sess = req.session;
    let auth = req.get('Authorization');
    if(auth && auth.indexOf('Bearer ')===0){
        auth = auth.slice(7);
        jwt.verify(auth, process.env.TOKEN_SECRET, function(error, payload) {
            if(!error){
                res.locals.jwtData = payload; // 解密後的資料掛在 res.locals 上
            }
            next();
        });
    } else {
        next();
    }
})
```

4. NextJS （ pages router ）設定

```
// next.config.js
/** @type {import('next').NextConfig} */
const nextConfig = {
  reactStrictMode: false,

  // 可將 server origin 設定為環境變數
  env: {
    API_SERVER: 'http://localhost:3002'
  }
}

module.exports = nextConfig
```

4.1 自訂 URL 設定

```
// src/config/ajax-path.js
export const SERVER = 'http://localhost:3600';

export const AB_GET_LIST = `${SERVER}/address-book/api`;
export const AB_GET_LIST_AUTH = `${SERVER}/address-book/api-auth`;

export const LOGIN_API = `${SERVER}/login-jwt`;
```

4.2 JSX 裡的類似 for 迴圈

```
{
  Array(11)
   .fill(1)
   .map(
      (v, i) => {}
   )
}
```

```
{
  [...Array(11)].map(
    (v, i) => {}
  )
}
```

4.3 CDN 只能設定在 _document.js

```
import { Html, Head, Main, NextScript } from "next/document";
export default function Document() {
  return (
    <Html lang="en">
      <Head>
        <link
          rel="stylesheet"
          href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
        />
      </Head>
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  );
}
```

4.4 Layout 設定

```
// @/components/Layout1
import Navbar from "@/components/Navbar";
import Head from "next/head";

export function Layout1({ children }) {
  return (
    <>
      <Head>
        <title>小新的網站</title>
      </Head>
      <div className="container">
        <Navbar />
      </div>
      <div className="container">{children}</div>
    </>
  );
}
```

使用 Layout

```
import { Layout1 } from "@/components/Layout1";
import Comp01 from "@/components/Comp01";
import { useState } from "react";
export default function Life01() {
  const [show, setShow] = useState(true);
  return <>
      <Layout1>
        <div>
          <button onClick={() => setShow(!show)}>toggle Comp01</button>
        </div>
        {show && (
          <>
            <Comp01 />
          </>
        )}
      </Layout1>
    </>;
}
```

4.5 NextJS 路由架構

- Pages router 以 pages/ 為根目錄。

檔案路徑	對應的 URL
pages/login.js	http://localhost:3001/login
pages/products/index.js	http://localhost:3001/products
pages/products/[pid].js	http://localhost:3001/products/17
pages/slug1/[slug2]/[slug3].js	http://localhost:3001/slug1/aaa/bbb

```jsx
// *** pages/slug1/[slug2]/[slug3].js ***
import { useRouter } from "next/router";

export default function Slug3() {
  const router = useRouter();

  return <pre>
    { JSON.stringify(router.query, null, 4) }
  </pre>;
}
```

http://localhost:3001/slug1/aaa/bbb

```
{
    "slug2": "aaa",
    "slug3": "bbb"
}
```

4.6 SSR

- 安裝 Chrome 擴充功能「Quick source viewer」提供者為 Tomi Mickelsson
- 注意頁面來源：
  - 什麼狀況頁面由 Server-Side Rendering (SSR) 來？
  - 什麼狀況頁面由 Client-Side Rendering (CSR) 來？
- SSR 時不要使用到前端的 DOM 相關物件或功能。例如，在元件內不要直接使用 window 物件，若要使用應該放在 useEffect() 的 callback 內使用，以確保是在前端執行。

4.7 useRouter

- 使用 useRouter() 所取得的 router 物件資料為非同步，在第一次 render 時，可能沒有資料。
- router.**query** 取得 query string parameters。
- router.query 同時可取得動態路由的 slug 資料。
- query string 和 slug 使用的名稱應避免相同。
- router.**push**(url, as, options) 前端不刷新頁面跳轉。
- https://nextjs.org/docs/pages/api-reference/functions/use-router

5. 以分頁按鈕取得分頁資料

```
// ****** pages/products/index.js ******
// 列表頁 1
export default function Products() {
  const router = useRouter();
  const [data, setData] = useState({
    redirect: "",
    totalRows: 0,
    perPage: 4,
    totalPages: 0,
    page: 1,
    rows: [],
  });
  const [keyword, setKeyword] = useState("");
```

```
// 列表頁 2
useEffect(() => {
  setKeyword(router.query.keyword || "");
  const usp = new URLSearchParams(router.query);

  fetch(`${process.env.API_SERVER}/products?${usp.toString()}`)
    .then((r) => r.json())
    .then((data) => {
      console.log(data);
      setData(data);
    });
}, [router.query]);
```

```jsx
{/* 列表頁 3 */}
<ul className="pagination">
  {Array(5)
    .fill(1)
    .map((v, i) => {
      const p = data.page - 2 + i; // 目前頁碼取前兩個
      const query = { ...router.query };
      if (p < 1 || p > data.totalPages) return;
      query.page = p;
      return (
        <li className={ `page-item ` + (p === data.page ? "active" : "") }
          key={p}
        >
          <Link
            className="page-link"
            href={"?" + new URLSearchParams(query).toString()}
          > {p} </Link>
        </li>
      );
    })}
</ul>
```

```
<table className="table table-bordered table-striped"> {/* 列表頁 4 */}
  <thead>
    <tr>
      <th>#</th>
      <th>書名</th>
      <th>作者</th>
      <th>價格</th>
      <th>書號</th>
    </tr>
  </thead>
  <tbody>
    {data.rows.map((i) => (
      <tr key={i.sid}>
        <td> {i.sid} </td>
        <td>
          <Link href={"/products/" + i.sid}> {i.bookname} </Link>
        </td>
        <td>{i.author}</td>
        <td>{i.price}</td>
        <td>{i.book_id}</td>
      </tr>
    ))}
  </tbody>
</table>
```

5.1 點擊項目進入單項商品頁

```
<td>
  <Link href={"/products/" + i.sid}> {i.bookname} </Link>
</td>
```

```
export default function ProductItem() {
  const router = useRouter();
  const [row, setRow] = useState(null);
  useEffect(() => {
    fetch(process.env.API_SERVER + "/products/" + router.query.pid)
      .then((r) => r.json())
      .then((result) => {
        if (result.success) {
          setRow(result.row);
        } else {
        }
      });
  }, [router.query]);
  return (
    <>
    <Navbar />
      <div>ProductItem</div>
      <p>{router.query.pid}</p>
      <pre>{row ? JSON.stringify(row, null, 4) : <p>沒有資料</p>}</pre>
    </>
  );
}
```

6. 兄弟元件間的資料共享

- ** 複習內容
- 單向資料流的限制。
- 只能透過上層元件，來共享資料或者操作資料的方法，如 setState()。
- CompA 為父元件，CampB 和 CampC 為其子元件。

```
// *** components/CompA.js
import {useState} from "react";
import CompB from '@/components/CompB'
import CompC from '@/components/CompC'

export default function CompA() {
  const [count, setCount] = useState(1);
  console.log('CompA render--')
  return (
    <div style={{border: '2px solid red'}}>
      <div>CompA</div>
      <CompB setCount={setCount} count={count}/>
      <CompC count={count}/>
    </div>
  );
}
```

```
// *** components/CompB.js
export default function CompB({count, setCount}) {
  console.log('CompB render--')
  return (
    <div style={{border: '2px solid green'}}>
      <div>CompB</div>
      <button
        onClick={() => {
          setCount((prevVal) => prevVal + 1);
        }}
      >
        {count}
      </button>
    </div>
  );
}
```

```
// *** components/CompC.js
export default function CompC({count}) {
  console.log("CompC render--");
  return (
    <div style={{border: '2px solid blue'}}>
      <div>CompC</div>
      <h2>count: {count}</h2>
    </div>
  );
}
```

7. 使用 Context (react.js)

7.1 網頁明暗模式 ThemeContext

```
// *** contexts/ThemeContext.js
import { createContext, useState } from "react";

export const themes = {
  dark: {
    name: 'dark',
    backgroundColor: 'gray',
    color: 'white',
  },
  light: {
    name: 'light',
    backgroundColor: 'yellow',
    color: 'black',
  }
};
const ThemeContext = createContext({});
export default ThemeContext;
```

7.2 使用 Context Provider

```
// *** contexts/ThemeContext.js

export const ThemeContextProvider = function ({children}){
  const [theme, setTheme] = useState(themes.light);
  return (
    <ThemeContext.Provider value={{theme, setTheme}}>
      {children}
    </ThemeContext.Provider>
  )
};
```

- ThemeContext 和 ThemeContextProvider 在同一個檔案 。

7.3 使用 useContext()

```
// *** components/Navbar.js
import { useContext } from "react";
import ToggleButton from "./ToggleButton";
import ThemeContext, { themes } from "../contexts/ThemeContext";

export default function Navbar() {
  // 對應到 value <ThemeContext.Provider value={{...theme, setTheme}}>
  const { theme, setTheme } = useContext(ThemeContext);
  // 其他內容
}
```

```
{/* 按鈕切換 theme */}
<li className="nav-item">
  <ToggleButton
    texts={["暗", "亮"]}
    statusIndex={theme.name === "dark" ? 0 : 1}
    handler={(i) => {
      setTheme(i === 0 ? themes.dark : themes.light);
    }}
  />
</li>
```

7.4 登入狀態 AuthContext

```
// *** contexts/AuthContext.js 1/3

import { createContext, useState } from "react";
import { useNavigate } from 'react-router-dom’;

const AuthContext = createContext({});

export default AuthContext;
```

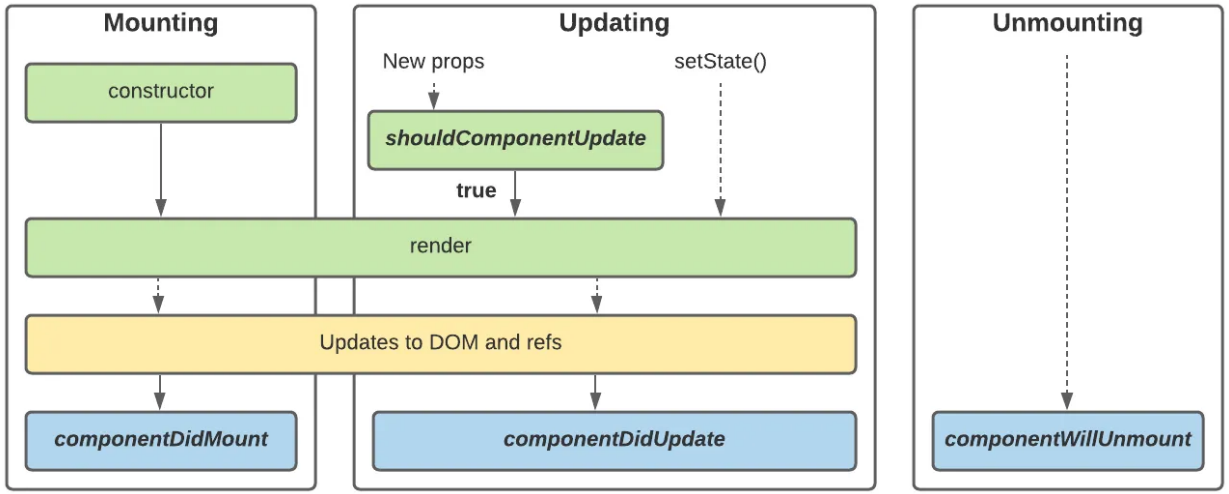
```
// *** contexts/AuthContext.js 2/3
export const AuthContextProvider = function ({children}){
  const navigate = useNavigate();
  const unAuth = {
    authorised: false, // 有沒有登入
    sid: 0,
    account: '',
    token: '',
  };
  let initAuth = {...unAuth};
  // 取得目前狀態, 看 localStorage 的資料是否表示已登入
  const str = localStorage.getItem('auth');
  if(str) {
    const localAuth = JSON.parse(str);
    if(localAuth && localAuth.token){
      initAuth = {...localAuth, authorised: true}
    }
  }
  const [myAuth, setMyAuth] = useState(initAuth);
```

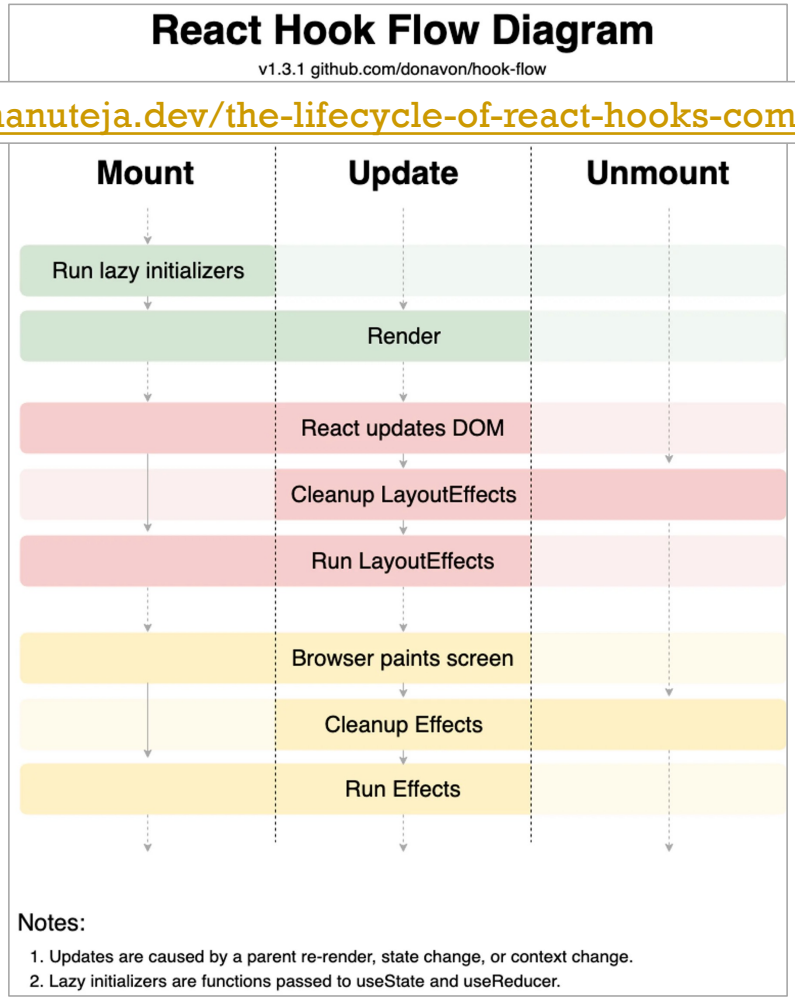
```
// *** contexts/AuthContext.js 3/3

  const logout = ()=>{
    localStorage.removeItem('auth');
    setMyAuth(unAuth);
    navigate('/login');
  };

  return (
    <AuthContext.Provider value={{myAuth, setMyAuth, logout}}>
      {children}
    </AuthContext.Provider>
  )
};
```

7.5 登入表單

```
// *** pages/Login.js
const mySubmit = async (e) =>{
  e.preventDefault();
  const { data } = await axios.post(LOGIN_API, formData);
  // console.log(data);
  if(data.success){
    localStorage.setItem('auth', JSON.stringify(data.auth) );
    alert('登入成功');
    // console.log(JSON.stringify(data))
    setMyAuth({...data.auth, authorised: true});
    navigate('/');
  } else {
    localStorage.removeItem('auth'); // 移除
    alert('登入失敗');
  }
};
```

8. 元件生命週期（參考資料）





https://bhanuteja.dev/the-lifecycle-of-react-hooks-component

```
// *** pages/life-cycle.js
import {useState} from "react";
import LifeA from "@/components/LifeA";
export default function LifeCyCle() {
  const [show, setShow] = useState(false);
  return (
    <>
      <div className="container">
        <button
          onClick={() => {
            setShow((old) => !old);
          }}
        >
          toggle
        </button>
        {show && <LifeA/>}
      </div>
    </>
  );
}
```

```jsx
// *** components/LifeA.js 1/2
import {useEffect, useState} from "react";

export default function LifeA() {
  const [val, setVal] = useState(10);
  console.log("render----1");
  useEffect(() => {
    console.log("掛載之後");
    let n = 1;
    const interval_id = setInterval(() => {
      console.log(n++);
    }, 1000)
    return () => {
      console.log("將要卸載");
      clearInterval(interval_id)
    }
  }, []);
```

```
// *** components/LifeA.js 2/2

  useEffect(() => {
    console.log("已更新");
    return () => {
      console.log("將要更新");
    }
  }, [val]);
  return (
    <div>
      LifeA
      {console.log("render----2")}
    </div>
  );
}
```

9. 使用 Canvas（參考資料）

- 若是載入圖檔再繪至 canvas 時，應該在 useEffect() 內操作。

https://bitbucket.org/lsd0125/mfee29-node/src/main/react-try/my-react01/src/pages/Canvas1.js

9.1 實作快取圖片檔

```
const [cart, setCart] = useState([]);
const [cache, setCache] = useState({}); // 快取 image 物件
```

```
const getImageFromPath = (path)=>{
    return new Promise((resolve, reject)=>{
        if(cache[path]){
            return resolve(cache[path]); // 回傳已存在的資料
        }
        const img = new Image();
        img.onload = () => {
            resolve(img);
            setCache({...cache, [path]:img});
        };
        img.src = path;
    });
}
```

9.2 重繪時避免閃爍

- 準備兩個 **canvas**，先繪在 **shadow-canvas**，都繪製完成後再繪到 **real-canvas**。

```
<canvas
    ref={shadowRef}
    width="800"
    height="600"
    hidden
></canvas>
<canvas
    ref={realRef}
    width="800"
    height="600"
    style={{ border: "1px dotted blue" }}
></canvas>
```

- 準備兩個 **canvas**，先繪在 **shadow-canvas**，都繪製完成後再繪到 **real-canvas**。

```
const renderCanvas = async ()=>{
    const realCtx = realRef.current.getContext("2d");
    const shadowCtx = shadowRef.current.getContext("2d");
    const bg = await getImageFromPath('/imgs/dish.jpeg');

    shadowCtx.clearRect(0, 0, shadowRef.current.width, shadowRef.current.height);
    shadowCtx.drawImage(bg, 0, 0);

    const tmpCart = cart.slice(0,5);
    for(let i=0; i<tmpCart.length; i++){
        const img = await getImageFromPath(`/imgs/${tmpCart[i].img}`);
        shadowCtx.drawImage(img, i*100, i*100);
    }
    realCtx.drawImage(shadowRef.current, 0, 0);
}
```

10. Google account 登入（參考資料）

- 先登入 https://console.cloud.google.com/
- 1. 到「API 程式庫」>「公開」>「社交」啟用 Google People API。
- 2. 到「API和服務 > 憑證」，點選「+建立憑證」>「OAuth 用戶 ID」。
- 3.「應用程式類型」選「網頁應用程式」
- 4.「已授權的 JavaScript 來源」加入正式環境及測試環境的 URI。
- 5.「已授權的重新導向 URI」加入接收 Query String 參數的頁面 URI，同樣包含測試及正式環境。
- 6. 建立完成後，「下載 OAuth 用戶端」的 JSON 檔。
- 7. 到「OAuth 同意畫面」，設定應用程式名稱，及測試帳號等。

10.1 使用OAuth2Client

```
const { OAuth2Client } = require('google-auth-library');

// 載入 GCP OAuth 2.0 用戶端 ID 憑證
const keys = require(__dirname + '/../client_secret.json');
const oAuth2c = new OAuth2Client(
  keys.web.client_id,
  keys.web.client_secret,
  keys.web.redirect_uris[0]  // 測試, http://localhost:3000/callback
);
```

10.2 建立連結 URL

```
router.get('/', async function(req, res, next) {
  // scopes 參考: https://developers.google.com/people/api/rest/v1/people/get
  const authorizeUrl = oAuth2c.generateAuthUrl({
    access_type: 'offline',
    // 欲取得 email, 要兩個 scopes
    scope: [
      'https://www.googleapis.com/auth/userinfo.profile',
      'https://www.googleapis.com/auth/userinfo.email',
    ]
  });

  res.render('index', { title: '點擊連結登入', authorizeUrl});
});
```

10.3 利用 tokens 取得資料

```
router.get('/callback', async function(req, res, next) {
  const qs = req.query;
  let myData = {};
  if(qs.code){
    // 內容參考 /references/from-code-to-tokens.json
    const r = await oAuth2c.getToken(qs.code);
    oAuth2c.setCredentials(r.tokens);
    const url =
'https://people.googleapis.com/v1/people/me?personFields=names,emailAddresses,photos';

    const response = await oAuth2c.request({url});
    // response 內容參考 /references/people-api-response.json
    myData = response.data;
  }
  res.render('callback', { title: 'Callback result', qs, myData });
});
```

附錄一 JSX

- JSX 為 JS 延伸的語法，用來轉換成 DOM 的元素，本質上依然是 JS 不是 HTML。
- JSX 為 React.createElement() 的語法糖。
- JSX 通常視為 JavaScript XML 的簡稱，顧名思義 JSX 必須符合 XML 的規定。
- 由於 JSX 語法貼近 HTML，方便用來描述 HTML。
- JSX 的目的是呈現 HTML 頁面的內容，和呈現無關的操作不應該放在 JSX 裡。
- 會自動做 HTML 跳脫（HTML escape）。

JSX 的優點

參考來源：React Quickly, Manning Publications

1. 改善開發人員的體驗，和 **React.createElement()** 相比較。
2. 較好的錯誤訊息。
3. 快速的執行速度，**JSX** 轉換成 **JS** 語法後，執行速度依然很快速。
4. 非專業的程式設定師或視覺設計師也容易參與編輯。
5. 較少的錯誤發生，程式碼越少表示錯誤越少。

```
const el = <Card>
  <Title>Hello</Title>
  <Link href="/article/1">Article 1</Link>
</Card>;
```

```
const el = React.createElement(
  Card,
  null,
  React.createElement(Title, null, 'Hello'),
  React.createElement(Link, {href: "/article/1"}, 'Article 1'),
);
```

JSX 常用規則

1. 只能有一個根節點。
2. 有起始標籤，也要有結束標籤，若為空元素時（沒有子節點），使用空元素的表示方式。
3. 可以使用在 JSX 檔（JS 檔）的任何位置，視為一個類型的物件。
4. 自訂標籤（元件），必須大寫字母開頭如：Card、MyCard。
5. 不可以使用到 JS 關鍵字，如：for、class，應以 htmlFor 和 className 取代。
6. 標籤的屬性必須使用 Camel 的方式表達（aria- 開頭者為例外）。
7. 標籤的屬性值的大括號表示綁定值或參照，可以是 JS 的任何類型物件。
8. 屬性沒有設定值時，表示綁定 true。
9. 行內 style 屬性必須綁定 Object 類型的物件。
10. 標籤之間的大括號表示要**輸出**成 jsx 內容的運算區塊。
11. 標籤之間若換行，則生成 html 時，兩標籤中間將不會有空白。

```
// 規則 1, 2, 3
const myForm = <form>
  <img src="/img/a.png" alt="img1" />
  <hr/>
  <input type="text" name="account" />
  <br/>
  <input type="submit" name="送出"/>
</form>;
```

```jsx
// 規則 5, 6, 8
const jsx = <form>
  <div className="mb-3">
    <label htmlFor="email" className="form-label">Email address</label>
    <input type="email"
           className="form-control"
           name="email"
           aria-describedby="emailHelp"
           required
    />
  </div>
  <button type="submit" className="btn btn-primary">Submit</button>
</form>
```

```
// 規則 7
const dataObj = {name: "David", age: 25};
const jsx2 = <form onSubmit={(e) => e.preventDefault()}>
  <div className="mb-3">
    <label htmlFor="email" className="form-label">Email address</label>
    <input type="email"
           className="form-control"
           name="email"
           value={myState}
           onChange={(e) => setMyState(e.target.value)}
    />
    <MyInput {...dataObj} myValue={dataObj}/>
  </div>
  <button type="submit" className="btn btn-primary">Submit</button>
</form>
```

```
// 規則 10
const dataObj = {name: "David", age: 25};
const jsx3 = <ul>
  {Object.keys(dataObj).map(el => {
    return <li key={el}>{el}: {dataObj[el]}</li>;
  })}
</ul>
```

JSX 流程控制的轉換用法：

1. if 敘述使用 &&。
2. if/else 敘述使用三元運算子。
3. 迴圈敘述使用陣列的 map() 方法。
4. switch/case敘述則無對應的方式，可以採用 Object 物件 key-value 對應的性質。

附錄二 Hooks 基本認識

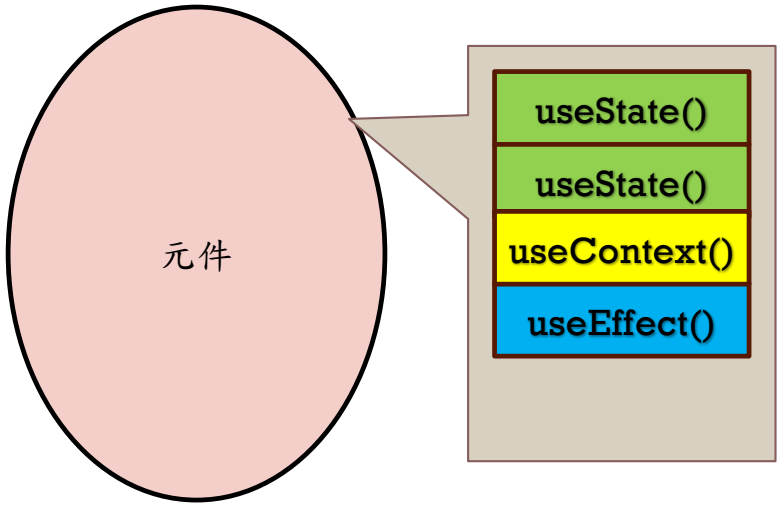
- Hooks（勾子）是因應 functional components 語法而生的。
- Components 使用 function 寫法時，基本上它就是一個 function，主要功能就是呈現（render）內容，沒有別的功能。
- 要像 class-based 元件有其它功能時，就必須借由 hooks 來賦予。
- 勾子的目的就是從 React 核心架構中勾一個功能到函式（元件）中來使用。
- Hooks 的名稱必須是以 **use** 為開頭，使用 Camel 的命名方式。
- Hooks 本身也是函式，可以組合基本的 hooks 來加以變化使用。
- Hooks 不可以放在 if/else 或者迴圈中使用。Hooks 使用時是跟著元件，並放在一個陣列中記錄，每次 render 時順序都必須一樣。
- Hooks 方便單元測試。

```jsx
// hooks 錯誤的使用方式
import {useState} from "react";

function MyButton() {
  if (Math.random() < .5) {
    return null;
  }
  const [val, setVal] = useState(0)
  return (
    <button>
      I'm a button {val}
    </button>
  );
}
```

Lint Error

React Hook "useState" is called conditionally. React Hooks must be called in the exact same order in every component render. Did you accidentally call a React Hook after an early return?

元件使用 hooks 示意圖



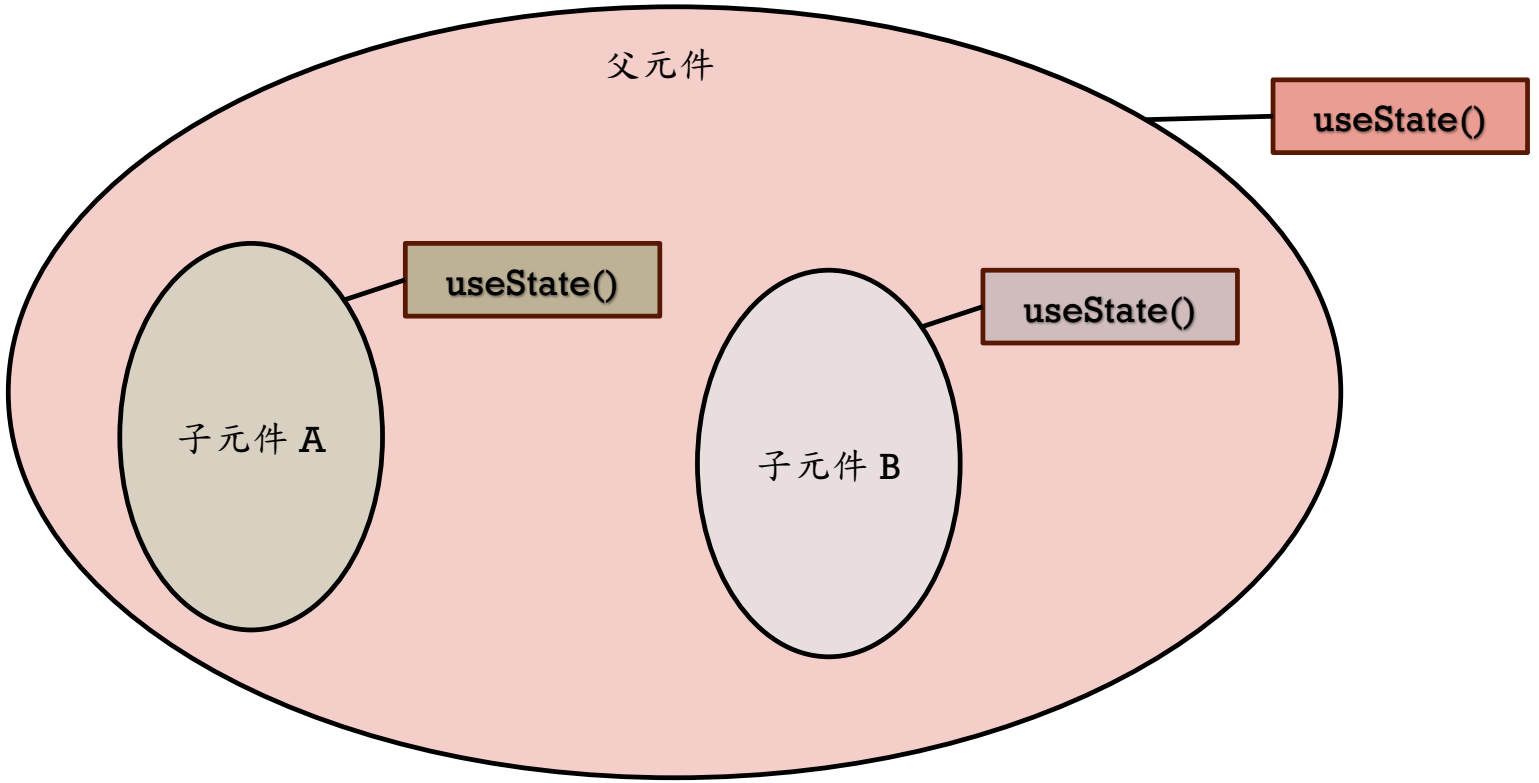
使用 hooks 的元件，相當於把使用的工具背在身上。
元件更新時（re-render），並不會影響 hooks 本身。

附錄三 State（狀態）概念

- useState() 將取得一個包含兩個參照的陣列，一個為**值**（getter），另一個為**變更函式**（setter）。
- **值**為狀態，視為「**唯讀**」，不可變更。（底層可能是使用 Proxy 實作的 getter）
- 變更狀態必須使用「**變更函式**」，而且其操作為**非同步**。意即呼叫**變更函式**後，值並沒有立即改變，而是必須等下次 render 時，**值**才能被觀察到變更。
- 狀態是跟在元件上的特殊屬性，當 state 變更時，表示元件需要更新（update），意即會觸發元件的 re-render 。
- 元件為函式時，re-render 相當於重新呼叫一次函式。函式內的所有區域變數將會被重新設定，包含存取狀態的**值**和**變更函式**。
- 狀態的**值**和**變更函式**，是由勾子處理，並不是在函式中處理。所以更新時，雖然**值**和**變更函式**被重新設定，但依然可以保有應有的狀態。

子元件 A 更新時的兩種情況

1. 子元件 A 自身的狀態改變
2. 父元件的狀態改變，以重新 render 子元件 A



模擬 useState - 1

```
<div id="app"></div>
<script>
  // *** 不能在嚴謹模式下測試此範例
  const app = document.querySelector('#app');
  const myMap = new Map(); // 儲存狀態用

  function MyApp() {
    const [val, setVal] = myUseState(5);
    // 模擬 render 行為
    app.innerHTML = `<div>
        <div>${val}</div>
        <div><button>click</button></div>
        </div>`;
    app.querySelector('button').onclick = () => {
      setVal(val + 1);
    }
  }
```

模擬 useState - 2

```
// 狀態類別
class MyState {
  constructor(init, caller) {
    this._value = init;
    this.belongTo = caller;
  }
  get value() {
    return this._value;
  }
  setValue = (v) => {
    this._value = v;
    this.belongTo(); // 重新 render
  }
}
```

模擬 useState - 3

```
 // 模擬 useState()
  function myUseState(init) {
    let item;
    if (myMap.has(myUseState.caller)) {
      item = myMap.get(myUseState.caller)
    } else {
      item = new MyState(init, myUseState.caller);
      myMap.set(myUseState.caller, item);
    }
    return [item.value, item.setValue];
  }

  MyApp();
</script>
```

** Function.prototype.caller 不建議使用，此範例純粹說明用。

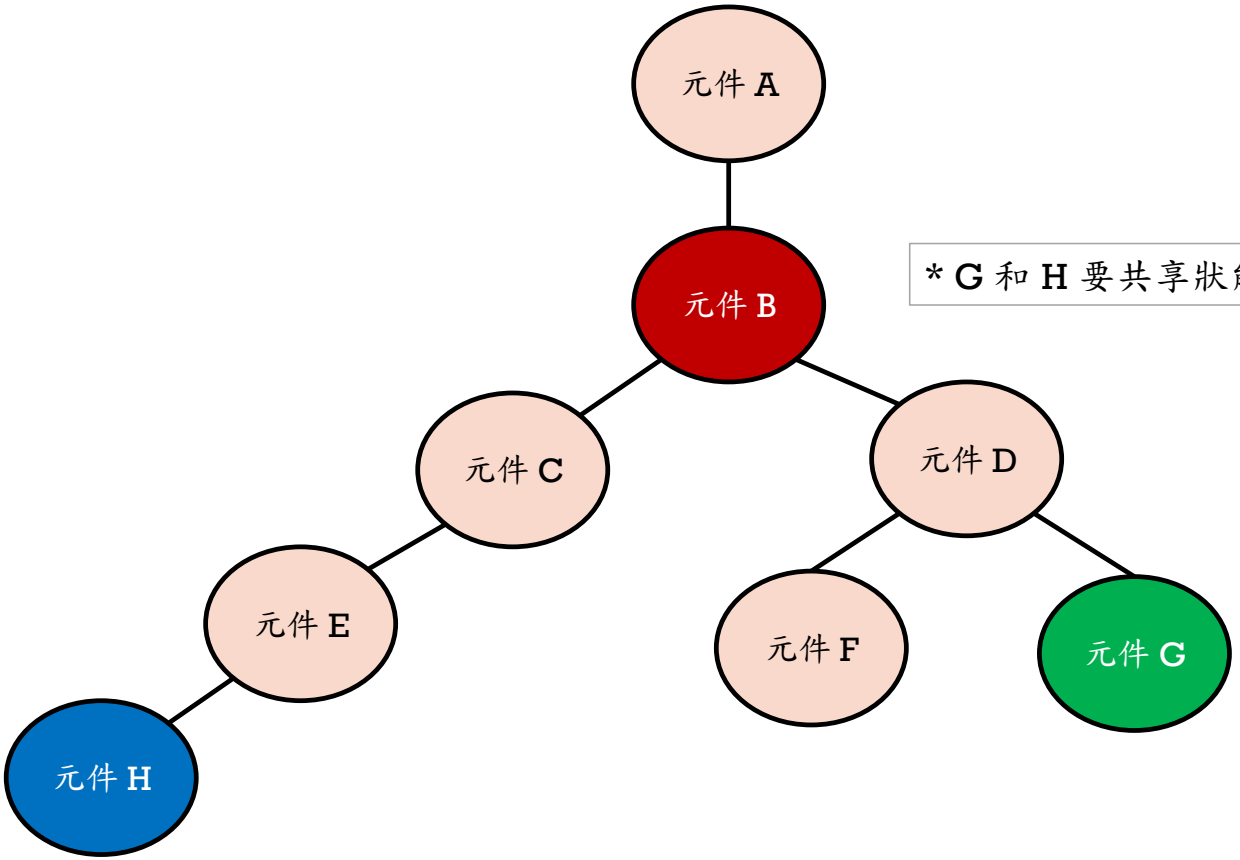
使用狀態時應注意的事項

- 父元件的狀態和子元件的狀態是各自獨立的。
- 各元件的狀態是各自獨立的。
- 單向資料流，只有父元件傳屬性給子元件，沒有子元件傳給父元件。
- 兩元件要溝通時，應將狀態設定在「共同祖先元件」上。
- 共同祖先元件的「狀態變更函式」往下一層一層傳給下游元件，讓該元件有能力直接變更共同祖先元件的狀態，以瀑布更新的方式，讓下游元件更新。
- 要在全域共享資料時，狀態應該設定在最頂層元件（App 或 _app）。
- 越頂層的狀態，若變更太頻繁可能造成效能不佳的情況。
- Context.Provider 中的狀態不應該有太頻繁的變更。

* 父元件狀態改變時會 **render** 子元件，子元件（函式）會被呼叫。

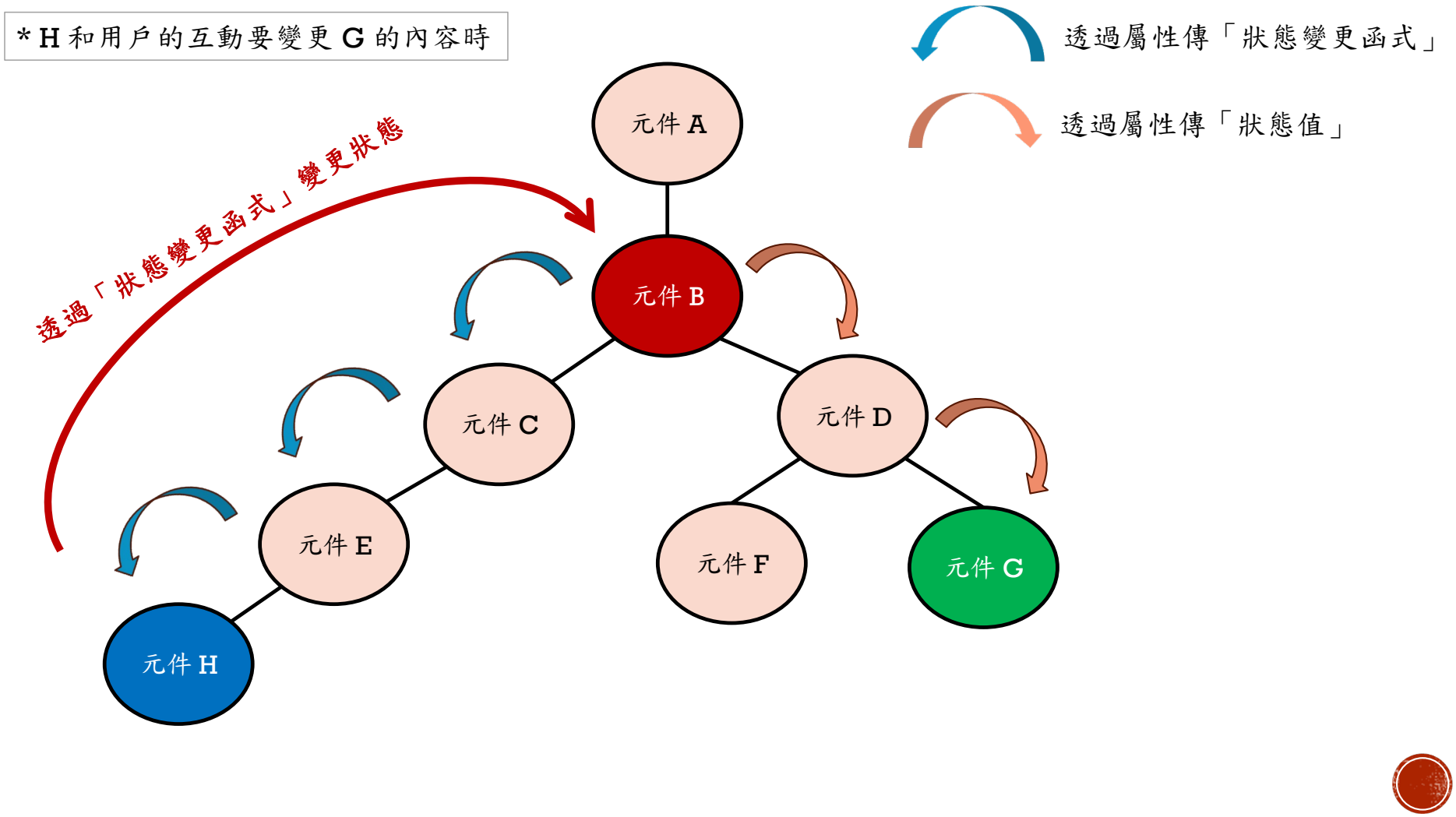
```
import {useState} from “react”;
let renderCount = 0;
export default function MyApp() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>click</button>
      <ChildA />
    </div>
  );
}
```

```
function ChildA() {
  renderCount ++;
  console.log({renderCount})
  return (
    <div>
      {renderCount}
    </div>
  );
}
```



* G 和 H 要共享狀能時應該設定在 B

*H 和用戶的互動要變更 G 的內容時



THANK YOU