|  |
| --- |
|  |
| React 學習筆記 |
| 搭配 Ant Design Pro 腳手架 |

|  |
| --- |
| 懷慶 黃  113/10/27 |

目錄

[環境搭建 與 參考網站 2](#_Toc180946149)

[資料結構的重點內容 3](#_Toc180946150)

[router 的路由結構 4](#_Toc180946151)

[頁面檔的基本結構 5](#_Toc180946152)

[JS 與 TypeScript 簡單介紹 6](#_Toc180946153)

[ 簡介 與 React的使用規則 6](#_Toc180946154)

[ 變數的宣告與型別設定 8](#_Toc180946155)

[ 複雜型別(type) 10](#_Toc180946156)

[ interface(介面) 12](#_Toc180946157)

[ 陣列 14](#_Toc180946158)

[ 函式 16](#_Toc180946159)

[ 多型別變數(聯集與交集) 18](#_Toc180946160)

[ JS 語句 20](#_Toc180946161)

[ if 20](#_Toc180946162)

[ switch 20](#_Toc180946163)

[ 三元運算子 20](#_Toc180946164)

[ for 21](#_Toc180946165)

[ foreach 21](#_Toc180946166)

[ while 21](#_Toc180946167)

[ 樣板字串 24](#_Toc180946168)

[state 狀態機 25](#_Toc180946169)

[props 組件的信息傳遞 (父傳子) 26](#_Toc180946170)

[reducer 管理複雜邏輯的狀態機 29](#_Toc180946171)

[context 遠端的信息傳遞 33](#_Toc180946172)

[context 的進階使用: 設定全域狀態變數 34](#_Toc180946173)

[useEffect 效果鉤子 38](#_Toc180946174)

[useRef 從輸入框獲取值 40](#_Toc180946175)

[Ant Design Pro 的全域變數工具: useModel 42](#_Toc180946176)

[React 的效能優化 44](#_Toc180946177)

[memo 暫存組件 44](#_Toc180946178)

[useCallback暫存函式本身 44](#_Toc180946179)

[useMemo 暫存函式運算結果 44](#_Toc180946180)

# 環境搭建 與 參考網站

* 簡介

LMS 所使用的前端框架 為 React，並結合 Ant Design Pro 作為 UI框架，  
以及 UmiJs 為 路由管理工具。

* 環境搭建

1. 安裝 Node.js (npm)  
   <https://nodejs.org/en/>
2. 安裝 yarn   
   於 CMD 執行 npm install --global yarn
3. 安裝 IDE 工具  
   VS Code  
   <https://code.visualstudio.com/>   
   建議安裝套件  
   ESlint  
   JS JSX Snippets  
   Chinese (Traditional) Language Pack for Visual Studio Code

* 參考網站

1. React 官網  
   <https://zh-hans.react.dev/learn>
2. Ant Design Pro  
   <https://pro.ant.design/zh-CN/docs/getting-started>
3. Ant Design 基本組件  
   <https://ant.design/components/overview-cn>
4. UmiJs  
   <https://umijs.org/docs/guides/getting-started>
5. ANTD 圖標庫  
   <https://ant.design/components/icon-cn>

* 基本指令 (於 專案資料夾根目錄 透過 CMD 執行)

1. 下載套件  
   yarn install
2. 啟動  
   yarn start

# 資料結構的重點內容

├── config                      # umi 配置，包含路由，构建等配置

│   └── routes.ts               # UmiJs 的路由設定檔

├── public                      # 放打包前html檔的地方

├── node\_modules                # 套件的擺放資料夾

├── src

│   ├── locales                 # 國際化文件資料夾

│       ├── zh-TW               # 繁體中文內容資料夾

│           └── zh-TW.ts        # 繁體中文 國際化文件 的 檔案 (要引入 zh-TW 的檔案 程式才找的到)

│   ├── pages

│       └── myExample           # 我自己建立的練習項目的資料夾

│           ├── components      # 擺放子頁面的資料夾

│               └── demo1.tsx   # 子元件的頁面檔

│           ├── store           # 擺放共用工具的資料夾

│               └── useUtel.ts  # 函式檔，通常用來設定 公用函式

│           └── index.tsx       # 頁面檔，其中 index.tsx 代表該項目的首頁

│   ├── app.tsx                 # 透過 UmiJs 封裝過的首頁 (使用 router 控管路由)

├── README.md

└── package.json                # 和設定打包有關的東西

* React 使用的檔案格式

1. \*.ts  
   撰寫 獨立函式、變數 所採用的檔案類型。
2. \*.tsx  
   React 組件使用的頁面檔 檔案類型。
3. index.tsx  
   每一個模組 的 首頁 頁面檔 檔名。

# router 的路由結構

  {

    path: '/myExample',    // 根目錄 路由，

    name: '練習頁面',       // 根目錄 名稱

    routes: [              // 子頁面的設定，有設定這個，就代表 不是節點 (按下去會展開)

      {

        path: 'demo0',            // 子路由，【\*/根路由/子路由】會自動導向該頁面

        name: '練習0',       // 子名稱

        component: './myExample',  // 導向的頁面路徑，以 src/pages 為出發點

      },

      {

        path: 'demo1',

        name: '練習1',

        routes: [                  // 需要展開第二層，可以再包一個 routes

          {

            path: 'demo1-1',

            name: '練習1-1',

            component: './myExample/components/demo1'

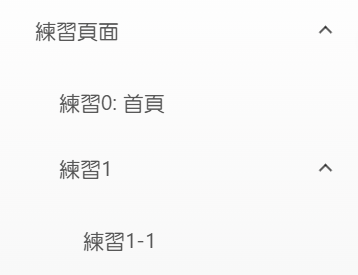
          }

        ]

      }

    ]

  }



# 頁面檔的基本結構

import React from 'react';

// 開頭 進行 元件的載入

// 頁面檔 其實就是一個函式，return 出去的就是要渲染的 html 元件

const myExample: React.FC = (輸入參數) => {

    // 這裡進行 邏輯處理

    return (    // 這裡進行 頁面刻畫，重點是要【包成一個】丟出去

        <div>

            練習首頁

        </div>

    )

}

export default myExample;



# JS 與 TypeScript 簡單介紹

## 簡介 與 React的使用規則

1. TypeScript = JavaScript + 嚴格的型別設定規則
2. JavaScript 可以在網頁中直接使用，  
   但是 TypeScript 必須經過編譯，才能將其轉譯為 JavaScript 於網頁中使用。
3. React 在使用時，需要遵守以下規定：
4. 使用 JS表達式，要使用『{ }』包起來
5. CSS 相關撰寫規定
   1. 套用 CSS 樣式，要使用 className 屬性  
      className = "CSS樣式名稱"
   2. 標籤的行內樣式，要使用『{{key:"value"}}』的形式撰寫  
      style={{key:"value", key:"value",...}}
6. 標籤的規則
   1. 開頭小寫，對應 html 標籤
   2. 開頭大寫，對應 React 組件

【index.tsx】

import './store/index.css'    // 引入 css 樣式模組

import Demo from './components/demo1'

const VDOM: React.FC = () => {

    const idvName = "divId"     // 設定變數

    const data = "Hello JSX"

    return (

        <div>

            {/\* 子組件要使用 大寫開頭 \*/}

            <Demo/>

            {/\* JS表達式 要用 {} 包住 \*/}

            <div id={idvName}>

                {/\* 使用 css 樣式模組:，要用 className 屬性 \*/}

                <h1 className="titleCss">

                    <span>此單元為 </span>

                    {/\* css 行內樣式 使用範例 \*/}

                    <span style={{color:"orange", fontSize:"20px"}}>

                        {data}

                    </span>

                </h1>

            </div>

        </div>

    )

}

export default VDOM

【index.css】

.titleCss {

    background-color: red;

    font-size: 30px;

  }

【demo1.tsx】

const demo1: React.FC = () => {

    return (

        <div>

            <h1>css使用範例</h1>

        </div>

    )

}

export default demo1



## 變數的宣告與型別設定

1. 變數的宣告方式

通常會使用 const 來進行宣告 並搭配 React Hook 使用

1. let  
   區域變數，僅在宣告的該函式內可以使用，數值可修改  
   >> let 變數名稱 = 初始化值
2. const  
   區域變數，僅在宣告的該函式內可以使用，數值不可直接修改  
   >> const 變數名稱 = 初始化值
3. 變數的基本型別：
4. boolean (布林)  
   只能是 true / false  
   >> let f\_boolean: boolean = true
5. number (數字)  
   只能是 整數 / 小數  
   >> let f\_number\_1: number = 10  
   >> const f\_number\_2: number = 3.14
6. string (字串)  
   >> const f\_string: string = "ABC"
7. void (無回傳值)  
   於 函式 使用
8. null (空值)
9. undefined (沒有賦值)
10. any (任意值)
11. 沒有宣告型別的處理方式
12. 變數有給值：  
    自動根據數值判定型別
13. 變數沒給值：  
    預設型別為 any

【index.tsx】

const VDOM: React.FC = () => {

    // 變數宣告 與 型態設定

    let f\_boolean: boolean = true

    const f\_number\_1: number = 10

    const f\_number\_2: number = 3.14

    const f\_string: string = "ABC"

    // let 可以 直接改變數值

    f\_boolean = false

    // 透過 console 顯示資訊

    console.log("boolean =>", f\_boolean)

    console.log("number(整數) =>", f\_number\_1)

    console.log("number(小數) =>", f\_number\_2)

    console.log("string =>", f\_string)

    return (

        <div>

        </div>

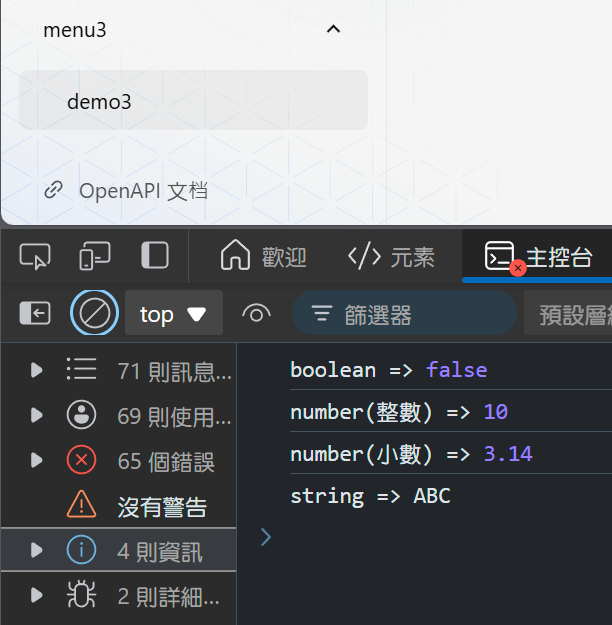
    )

}

export default VDOM

Boolean 一開始為 true

但因為後來數值改變了，所以最後顯示為 false

.

## 複雜型別(type)

1. type 用來定義 複雜的型別，也就是 裡面有多個屬性的型別。
2. 語法：

type 型別名稱 = {

必填屬性A: 型別;

選填屬性B?: 型別; // 選填屬性 用「?」表示

readonly 屬性C?: 型別; // 唯獨屬性 只有初始創建時能夠賦值

}

const VDOM: React.FC = () => {

    /\* 抽象型別 \*/

    // 書本 有 名稱、頁數

    type Book = {

        name: string;

        pages: number;

        amt?: number;

        readonly id: number;

    };

    // 產品A: 無選填屬性 amt

    let f\_data1: Book = {

        name: "輕鬆學習 TypeScript",

        pages: 150,

        id: 1

    }

    console.log("產品A", f\_data1);

    // 產品B: 有選填屬性 amt

    let f\_data2: Book = {

        name: "React 真簡單",

        pages: 280,

        amt: 1500,

        id: 2

    }

    console.log("產品B", f\_data2);

    return (

        <></>

    )

}

export default VDOM



## interface(介面)

1. 基本上 跟 type 相同，但多了 聲明合併的特性
2. 語法範例：

interface 型別名稱A = {

必填屬性A: 型別;

}

interface 型別名稱A = {

必填屬性B: 型別;

選填屬性C?: 型別; // 選填屬性 用「?」表示

readonly 屬性D?: 型別; // 唯獨屬性 只有初始創建時能夠賦值

}

此時 型別名稱A 的 屬性為

{

必填屬性A: 型別;

必填屬性B: 型別;

選填屬性C?: 型別; // 選填屬性 用「?」表示

readonly 屬性D?: 型別; // 唯獨屬性 只有初始創建時能夠賦值

}

const VDOM: React.FC = () => {

        /\* 介面 interface \*/

        interface Person {

            readonly id: number;

        }

        interface Person {

            name: string;

            age: number;

            sex?: string;

        }

        // 案例

        let ray: Person = {

            id: 2,

            name: "ray",

            age: 20,

            sex: "男",

        }

        console.log("ray", ray);

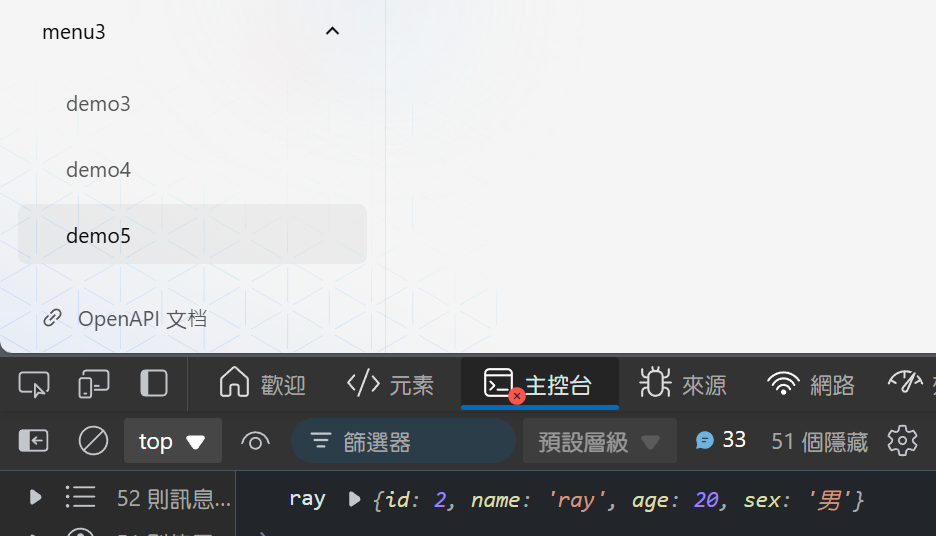
        return (

            <></>

    )

}

export default VDOM



## 陣列

1. 語法：
2. 一般

let 陣列名稱: 型別[] = [數值A, 數值B]

1. 泛型

let 陣列名稱: Array<型別> = [數值A, 數值B]

1. 常用方法：
2. 新增  
   陣列名稱.push(數值)
3. 長度  
   陣列名稱.length
4. 排序 (小到大)  
   陣列名稱.sort()
5. 排序 (大到小)  
   陣列名稱.sort((x,y) => y-x)

const VDOM: React.FC = () => {

    /\* 一般陣列 \*/

    let f\_numbers: number[] = [1,5,3,2,4];

    /\* 泛型陣列 \*/

    console.log("宣告", f\_numbers);

    let f\_currency: Array<string> = []

    // 新增資料

    f\_numbers.push(6);

    console.log("新增資料", f\_numbers);

    // 陣列長度

    let f\_length = f\_numbers.length;

    console.log("陣列長度", f\_length);

    // 排序 (小到大)

    let f\_sortA = f\_numbers.sort();

    console.log("排序 (小到大)", f\_sortA);

    // 排序 (大到小)

    let f\_sortB = f\_numbers.sort((x,y) => y-x);

    console.log("排序 (大到小)", f\_sortB);

    // 泛型範例

    f\_currency.push("TWD")

    f\_currency.push("USD")

    console.log("泛型", f\_currency);

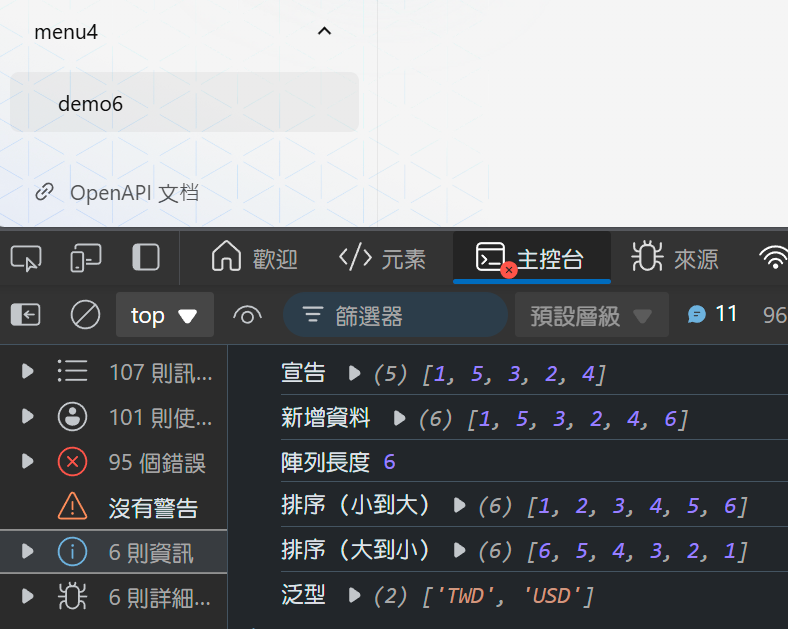
    return (

        <></>

    )

}

export default VDOM



## 函式

1. 函式：
2. 箭頭函式 (arrow function)

const 函式名稱 = (參數: 型別, ……): 回傳型別 => {

處理邏輯

return 回傳值

}

1. 函式宣告 (Function Declaration)

function 函式名稱(參數: 型別, ……): 回傳型別 {

處理邏輯

return 回傳值

}

1. 函式表示式 (Function Expression)

const 函式名稱 = function (參數: 型別, ……): 回傳型別 {

處理邏輯

return 回傳值

}

1. 參數：
   1. 必填參數

>> 參數: 型別

* 1. 選填參數

>> 參數?: 型別

* 1. 剩餘參數

1. 必須是 函數的 最後一個 參數
2. 必須是一個 陣列
3. 用途為 接收函數中剩下的 Input

>> ...參數: 型別[]

const VDOM: React.FC = () => {

    // 函式宣告: 使用 剩餘函數

    function sum(...numbers: number[]): number {

        let f\_sum = 0;

        numbers.forEach((number) => {

            f\_sum += number;

        });

        return f\_sum;

    }

    let f\_value\_1 = sum(1,2,3,4,5);

    // 箭頭函式

    const diff = (x: number, y:number): number => {

        return x-y

    }

    let f\_value\_2 = diff(5,2)

    return (

        <>

            <h2>1+2+3+4+5= {f\_value\_1}</h2>

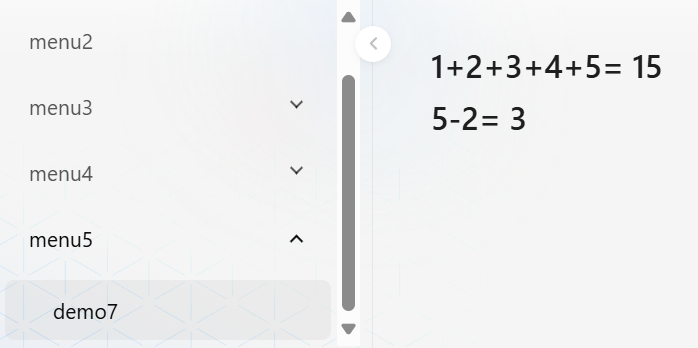
            <h2>5-2= {f\_value\_2}</h2>

        </>

    )

}

export default VDOM



## 多型別變數(聯集與交集)

1. 聯集 (或)
2. 只要符合 『型別A』 或 『型別B』 其中一種 即可
3. 語法

let 變數名稱: (型別A | 型別B) = 數值

const VDOM: React.FC = () => {

    /\* 聯集 (或) \*/

    // 類型A: 書本 有 名稱、頁數

    type Book = {

        name: string;

        pages: number;

    };

    // 類型B: 電腦 有 RAM、CPU

    type PC  = {

        RAM: string;

        CPU: string;

    };

    // 產品 可以是 書 或 電腦

    type Product = Book | PC;

    // 第一種產品，內容 = 書

    let f\_data1: Product = {

        name: "輕鬆學習 TypeScript",

        pages: 150,

    }

    console.log("第一種產品", f\_data1);

    // 第二種產品

    let f\_data2: Product = {

        RAM: "64GB",

        CPU: "i9",

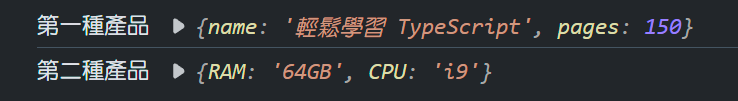
    }

    console.log("第二種產品", f\_data2);

    return (<></>)

}

export default VDOM



1. 交集 (且)
2. 必須同時符合 『型別A』 和 『型別B』
3. 語法  
   let 變數名稱: (型別A & 型別B) = 數值

const VDOM: React.FC = () => {

    /\* 交集 (且) \*/

    // 類型A: 軟體 有 作業系統、版本

    type Software  = {

        system: string;

        version: string;

    };

    // 類型B: 硬體 有 RAM、CPU

    type Hardware   = {

        RAM: string;

        CPU: string;

    };

    // 手機 同時要有 硬體 和 軟體

    type Phone = Software & Hardware;

    // 產品: iPhone 15

    let f\_data: Phone = {

        system: "ios",

        version: "17.0",

        RAM: "16GB",

        CPU:"A16"

    }

    console.log("iPhone 15", f\_data);

    return (<></>)

}

export default VDOM



## JS 語句

### if

1. 語法

if (布林條件式) {

內容;

}

else if (布林條件式) {

內容;

}

……以此類推

else {

內容;

}

### switch

1. 語法

switch (變數) {

case 值A:

內容;

break; ← 沒有break將會繼續往下執行

case 值B:

內容;

break;

……以此類推

case 值N:

內容;

break;

default: ← 其他情況：變數 不符合 (值A ~ 值N) 會執行這裡

內容;

break;

}

### 三元運算子

1. 語法  
   布林條件式 ? True的結果 : False的結果
2. 範例  
   const data = (age>=18 ? '成年' : '未成年')  
   >> 如果 age >= 18 會是 成年；否則是 未成年

### for

1. 語法

for (變數 = 初始值; 布林條件式; 迴圈計算式) {

內容;

}

1. 範例

const data2 = [];

for (let i = 1 ; i < 10 ; i++) {

    data2.push(<li>{i}</li>)

}

### foreach

1. 無回傳值

陣列變數.forEach((陣列元素變數) => {

    內容

})

1. 有回傳值 (會自動組成陣列)

const 變數 = 陣列變數.map((陣列元素變數) => {

  內容

})

### while

1. 語法

while (布林條件式) {

內容;

}

const VDOM: React.FC = () => {

    // 範例: if

    const sex = '1'

    let data1 = ""

    if (sex === '1') {

        data1 = "男"

    } else {

        data1 = "女"

    }

    // 範例: 三元表達式

    const age = 18;

    const data2 = (age>=18 ? '成年' : '未成年')

    // 範例: for

    const data3 = [];

    for (let i = 1 ; i < 5 ; i++) {

        data3.push(<li>{i}</li>)

    }

    // 範例: foreach

    const data4 = ['a','b','c','d']

    data4.forEach((data) => {

        data3.push(<li>{data}</li>)

    })

    const data5 = data4.map((data) => {

        return <li key={data}>{data}</li>

    })

    return (

        <>

            <h3>年齡: {age}，是否成年: {data2}，性別: {data1}</h3>

            <ul>

                第一區塊: {data3}

                第二區塊: {data5}

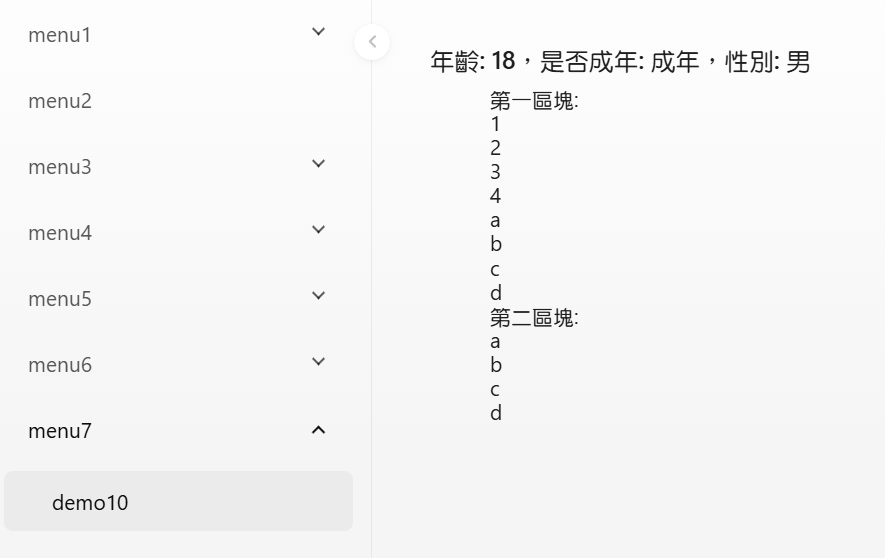
            </ul>

        </>

    )

}

export default VDOM



## 樣板字串

這是一個能夠讓你方便將 文字、變數、空格 進行串接的東西。

傳統JS要進行文字串接的話，需要將 文字 和 變數 分開寫，再用『+』串起來

整體而言非常麻煩

const msg = "我是 " + user.name + "，今年 " + user.age + " 歲，住在 " + user.city + " 。"

在 ES6 中，推出了 樣板字串 來讓這類需求可以更容易撰寫。

1. 前後使用『` `』夾起來
2. 變數 使用 『${ }』包起來
3. 如果 文字有特殊字元，則在前面使用 『\』就可以將其印出

const msg2 = `我是 ${user.name}，今年 ${user.age} 歲，住在 ${user.city} 。

* 範例

import React from 'react';

const MyComponent: React.FC = () => {

  const user = {

    name: '小明',

    age: 18,

    city: '台北'

  }

  const msg1 = "我是 " + user.name + "，今年 " + user.age + " 歲，住在 " + user.city + " 。"

  const msg2 = `我是 ${user.name}，今年 ${user.age} 歲，住在 ${user.city} 。`

  return (

    <>

      <h1>{msg2}</h1>

    </>

  );

};

export default MyComponent;



# state 狀態機

如果當變數內容改變時，畫面要重新渲染，就需要將這個變數設定為state(狀態機)。

1. 宣告

const [狀態變數(state), 更新方法(setState)] = useState(初始值)

>> const [isHot, setIsHot] = useState(true)

1. 數值更新
2. 直接給值 (後蓋前)  
   setState(新狀態)
3. 函式更新  
   setState((變數) => {更新邏輯})

＊setState 一般使用下，連續多次執行更新時，  
　結果會 後蓋前，只保留 最後一次的結果  
　如果 每次的結果都要保留到下一次使用，就必須使用 函式進行更新

1. useState 位於 react核心組件 中，使用時要記得引入  
   import { useState } from "react"

import { Button, Space } from "antd";

import { useState } from "react";

const VDOM: React.FC = () => {

    // 宣告

    const [isHot, setIsHot] = useState(true)

    // 天氣的顯示設定

    const isHotDesc = isHot ? '炎熱' : '涼爽'

    // 透過函式改變天氣值

    // 因為 天氣值 isHot 是狀態機，所以改變時會啟動畫面重新渲染

    // 因此 顯示的 天氣中文內容 就會同步改變

    function chgWeather () {

        setIsHot(!isHot)

    }

    return (

        <Space>

            <Button type="primary" onClick={chgWeather}>改變天氣</Button>

            <h3>今天天氣很{isHotDesc}</h3>

        </Space>

    )

}

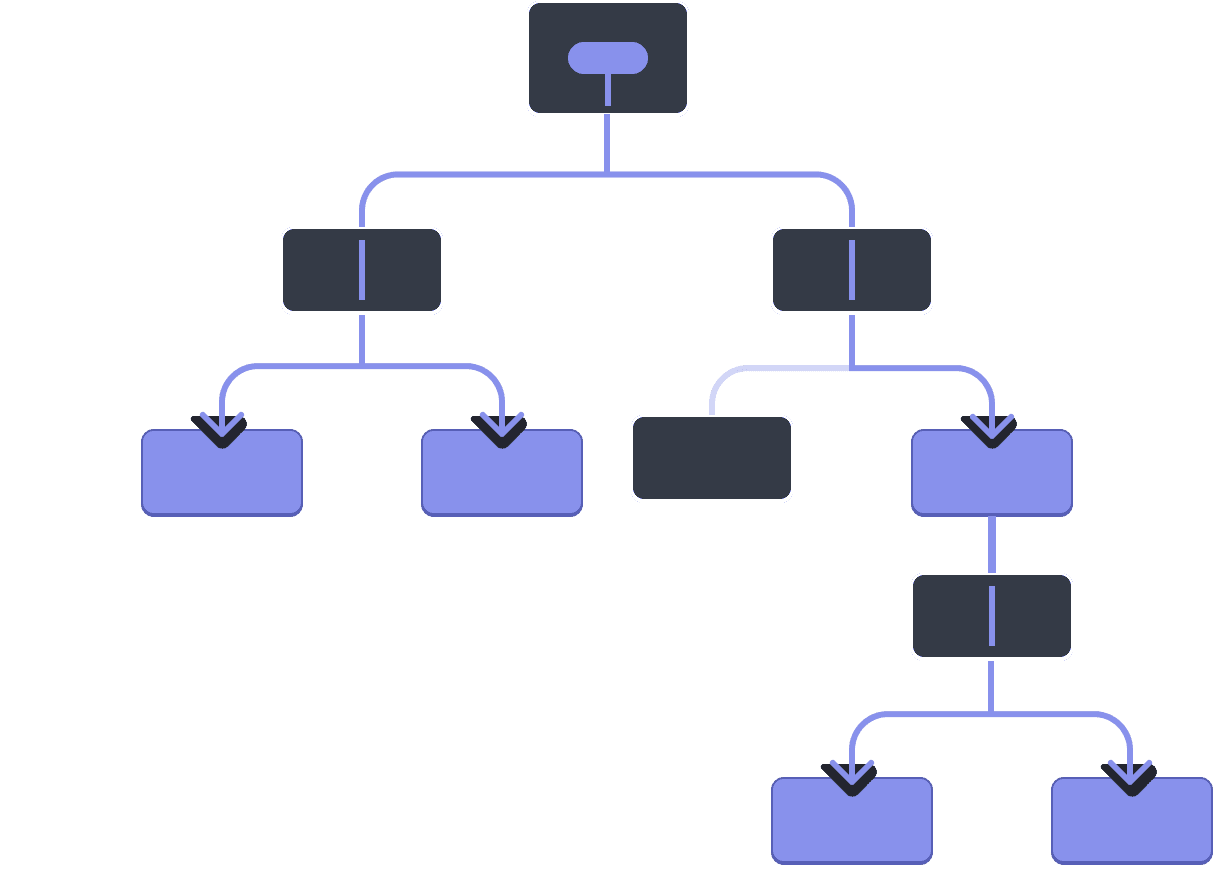
export default VDOM



# props 組件的信息傳遞 (父傳子)

React 的組件 使用 props 來將 父組件 的訊息 傳遞給 子組件。

他可以傳遞 任何 JavaScript值，包括 對象、數組、函數。



圖一：props 必須 逐級傳遞資訊

1. 【父組件】  
   透過 設定 子組件的 html標籤，將 想要傳遞的 信息 傳遞給 子組件使用  
   數據傳遞有兩種方法
2. 單獨傳遞

將每個信息，透過 屬性標籤 單獨傳遞過去  
<Component

name={userData.name}

age={userData.age}

sex={userData.sex}/>

1. 同時傳遞

將所有 信息 包裝起來，就可以一次傳遞過去

<Component {...userData}/>

1. 【子組件】

在 頁面函數 的 參數區域 中，設定要接受的變數名稱

const Component = (props: any) => {

return (

<ul>

<li>姓名：{props.name}</li>

<li>年齡：{props.age}</li>

<li>性別：{props.sex}</li>

</ul>

)

}

export default Component

【index.tsx】

import Component from "./components/demo12";

const VDOM: React.FC = () => {

        // 要傳遞的數據

        const userData = {

            name: '王大明',

            age: 25,

            sex: '男'

        }

        return (

            <>

                {/\* 這種寫法 是將每個要傳遞的屬性 單獨寫出來 \*/}

                <div>

                    <h1>單獨傳遞</h1>

                    <Component name={userData.name} age={userData.age} sex={userData.sex}/>

                </div>

                {/\* 下面的寫法，代表傳遞 userData 中全部的屬性 \*/}

                <div>

                    <h1>同時傳遞</h1>

                    <Component {...userData}/>

                </div>

            </>

    )

}

export default VDOM

【demo12.tsx】

const Component = (prpos: any) => {

    return (

        <ul>

            <li>姓名：{prpos.name}</li>

            <li>年齡：{prpos.age}</li>

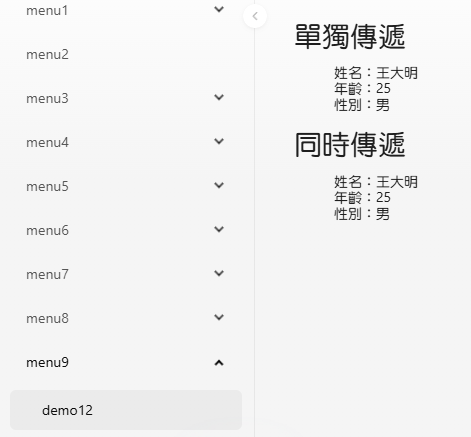
            <li>性別：{prpos.sex}</li>

        </ul>

    )

}

export default Component



# reducer 管理複雜邏輯的狀態機

useState 和 useReducer 都是狀態機，數值改變時，都會觸發畫面的重新渲染，

1. useState:  
   用於 變數 的狀態更新機制單純，每次都是 整體完全更新
2. useReducer:  
   可以視為 複雜的 useState，需要 自行設定 狀態改變函式  
   能夠根據不同類型，執行不同的 狀態改變函式

＊useState 就是 簡化版的 useReducer，  
　直接幫你設定好 狀態改變函式 (僅直接取代原有狀態值)

使用方式：

1. 宣告useReducer

const [狀態變數, 狀態改變方法] = useReducer(狀態改變對應函式, 初始值)

1. 設定 狀態改變對應函式
   * 參數 state： 代表 狀態變數 目前的數值
   * 參數 action： 代表 執行的操作物件，通常至少會有兩個參數：  
      type:   操作類型  
      value:  操作值

const 狀態改變對應函式 = (state: any, action: any) => {

// 透過 switch 來設定 不同類型 要執行的操作

switch (action.type) {

case "類型A":

類型A 要做的邏輯處理

return 類型A處理後的 狀態變數值

case "類型B":

類型B 要做的邏輯處理

return 類型B處理後的 狀態變數值

......

default:

其他情況 要做的邏輯處理

return 其他情況處理後的 狀態變數值，通常會直接丟出 state

}

1. 使用 狀態改變方法

onClick={() => 狀態改變方法({type: "操作類型", value: 操作值})}

＊於事件中 呼叫 狀態改變方法(執行的操作物件)  
　該方法 會自動 呼叫 狀態改變對應函式(目前狀態變數值, 執行的操作物件)

import { Button, Row, Space } from 'antd';

import React, { useReducer } from 'react';

const MyComponent: React.FC = () => {

  /\*\*

   \* reducer函式: 自定義的狀態改變函式

   \* @param state   目前的狀態機變數

   \* @param action  執行的操作物件，通常至少會有兩個參數：

   \*                type:   操作類型

   \*                value:  操作值

   \* @returns

   \*/

  const cityReducer = (state: any, action: any) => {

    // 根據 操作類型(action.type) 來判斷要執行的改變公式

    switch (action.type) {

      // 類型台北: 會進行數量的加減

      case "taipei":

        return {

          ...state,

          taipei: state.taipei + action.value

        }

      // 類型台中: 會進行數量的加減

      case "taichung":

        return {

          ...state,

          taichung: state.taichung + action.value

        }

      // 類型高雄: 會進行數量的加減

      case "kaohsiung":

        return {

          ...state,

          kaohsiung: state.kaohsiung + action.value

        }

        // 其他類型: 不進行改變

      default:

        return state

    }

  }

  /\*\*

   \* useReducer的宣告

   \* const [變數, 狀態改變方法] = useReducer(狀態改變方法會啟動的函式, 初始值)

   \*/

  const [cityState, cityDispatch] = useReducer(cityReducer, {

    taipei: 0,

    taichung: 0,

    kaohsiung: 0,

  })

  // 要執行 useReducer 的狀態改變，需要呼叫 狀態改變方法

  return (

    <>

      <Row>

        <Space>

          <h3>台北報名人數: {cityState.taipei}</h3>

          <Button onClick={() => cityDispatch({type: "taipei", value: 1})}>+ 1</Button>

          <Button onClick={() => cityDispatch({type: "taipei", value: -1})}>- 1</Button>

        </Space>

      </Row>

      <Row>

        <Space>

          <h3>台中報名人數: {cityState.taichung}</h3>

          <Button onClick={() => cityDispatch({type: "taichung", value: 1})}>+ 1</Button>

          <Button onClick={() => cityDispatch({type: "taichung", value: -1})}>- 1</Button>

        </Space>

      </Row>

      <Row>

        <Space>

          <h3>高雄報名人數: {cityState.kaohsiung}</h3>

          <Button onClick={() => cityDispatch({type: "kaohsiung", value: 1})}>+ 1</Button>

          <Button onClick={() => cityDispatch({type: "kaohsiung", value: -1})}>- 1</Button>

        </Space>

      </Row>

    </>

  );

};

export default MyComponent;



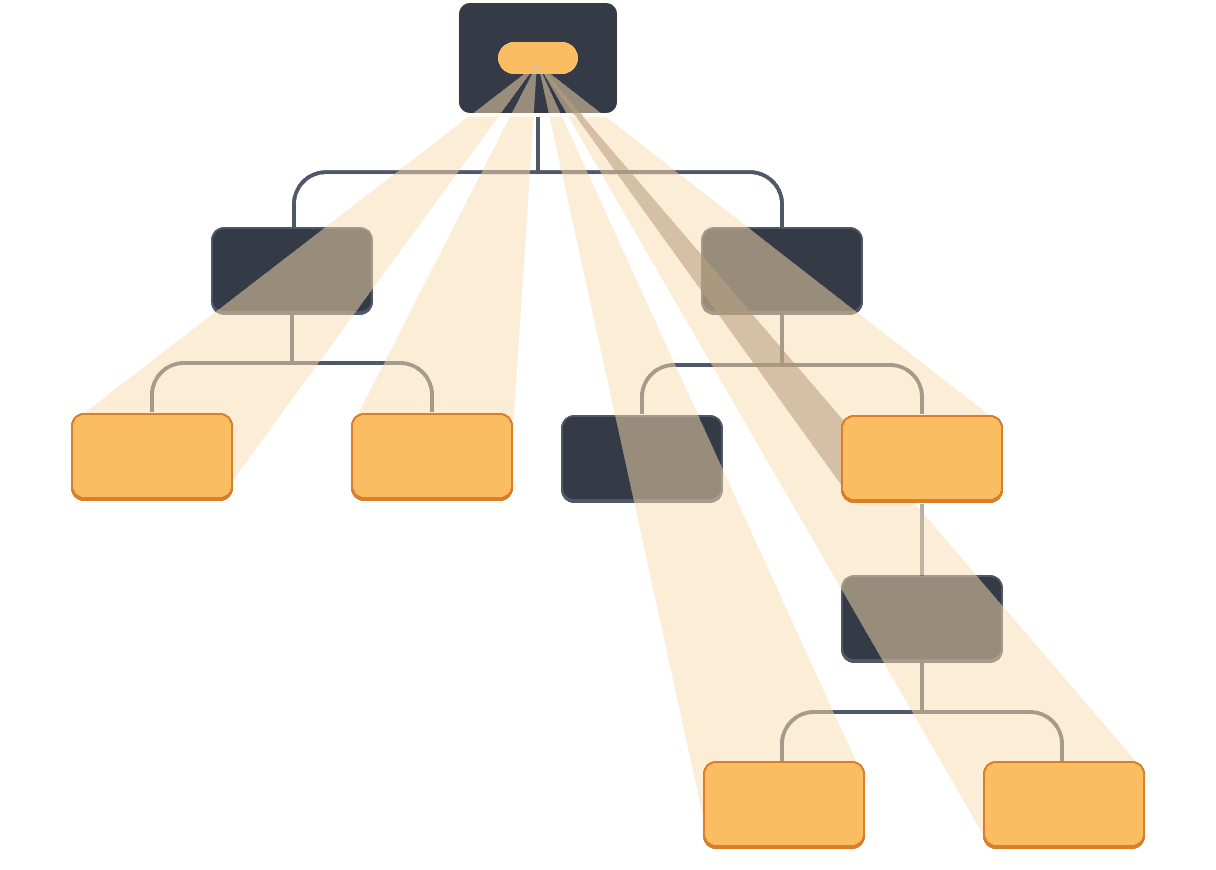
圖一：useReduer的範例，台北、台中、高雄的報名人數的可以獨立變更

# context 遠端的信息傳遞

一般在進行 信息傳遞時，會使用 props，但它有一個致命的缺點，就是必須逐級傳遞。

如果當 需要這個信息的子組件 位於 遠端時，信息傳遞就變得很麻煩。

為了滿足這類的需求，React 使用 context 進行 這類信息的傳遞。



圖一：context 可以 跨域傳遞信息

1. 傳統方法
2. 【父組件】

import { createContext } from "react"

// 宣告 context 變數

// 要下『export』因為 後層組件需要調用

export const Context變數 = createContext()

const VDOM: React.FC = () => {

return (

        // 使用 Provider 將 後層組件 包起來，並且 數值要放在 value標籤 中

        <Context變數.Provider value={信息}>

            <子組件/>

        </Context變數.Provider>

    )

}

export default VDOM

1. 【子組件】

// 載入 父組件所在的組件，並設定取得 Context變數

import { Context變數 } from 'context所在的組件'

import { useContext } from "react"

 const VDOM\_C = () => {

    // 使用 useContext 將 Context變數 的數值取出來

    const 存放數值的變數 = useContext(Context變數)

    return (

        <>

            // 這裡設定要顯示的內容

        </>

    )

}

export default VDOM\_C

# context 的進階使用: 設定全域狀態變數

* 全域狀態變數設定

【store/useUserContext.tsx】

import React, { createContext, useContext, useState, ReactNode } from 'react';

// 定義用戶資料的型別

interface User {

  name: string;

  age: number;

  gender: string;

}

// 定義全域狀態的型別

interface GlobalState {

  user: User;                         // 數值

  updateUser: (user: User) => void;   // 更新方法

}

// 創建 Context

const GlobalContext = createContext<GlobalState | undefined>(undefined);

// 創建提供者

export const GlobalProvider: React.FC<{ children: ReactNode }> = ({ children }) => {

  const [user, setUser] = useState<User>({

    name: '',

    age: 0,

    gender: '',

  });

  const updateUser = (newUser: User) => {

    setUser(newUser);

  };

  return (

    <GlobalContext.Provider value={{ user, updateUser }}>

      {children}

    </GlobalContext.Provider>

  );

};

// 自訂 hook 方便使用 Context

// 因為 GlobalContext 有 user 和 updateUser 屬性，所以會丟出 這兩個信息

export const useUserGlobalContext = () => {

  const context = useContext(GlobalContext);

  if (!context) {

    throw new Error('useGlobalContext must be used within a GlobalProvider');

  }

  return context;

};

* 父組件: 使用全域狀態變數

import React from 'react';

import { GlobalProvider } from './store/useUserContext';

import Profile1 from './components/demo13-1';

import Profile2 from './components/demo13-2';

const App: React.FC = () => {

  return (

    <GlobalProvider>

      <Profile1 />

      <Profile2 />

    </GlobalProvider>

  );

};

export default App;

* 子組件: 更新全域狀態變數

import React, { useState } from 'react';

import { useUserGlobalContext } from '../store/useUserContext';

import { Button, Input, Space } from 'antd';

const Profile: React.FC = () => {

  const { user, updateUser } = useUserGlobalContext();

  const [newUser, setNewUser] = useState(user);

  const handleUpdate = () => {

    updateUser(newUser);

  };

  return (

    <Space>

      <Input

        type="text"

        placeholder="姓名"

        value={newUser.name}

        onChange={(e) => setNewUser({ ...newUser, name: e.target.value })}

      />

      <Input

        type="number"

        placeholder="年齡"

        value={newUser.age}

        onChange={(e) => setNewUser({ ...newUser, age: Number(e.target.value) })}

      />

      <Input

        type="text"

        placeholder="性別"

        value={newUser.gender}

        onChange={(e) => setNewUser({ ...newUser, gender: e.target.value })}

      />

      <Button onClick={handleUpdate}>Update</Button>

    </Space>

  );

};

export default Profile;

* 子組件: 顯示全域狀態變數資料

import { useUserGlobalContext } from '../store/useUserContext';

const Component = () => {

    const { user } = useUserGlobalContext();

    return (

        <>

            <h2>姓名：{user.name}</h2>

            <h2>年齡：{user.age}歲</h2>

            <h2>性別：{user.gender}</h2>

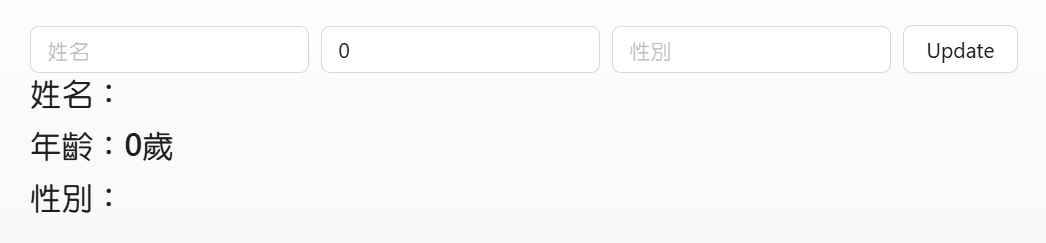
        </>

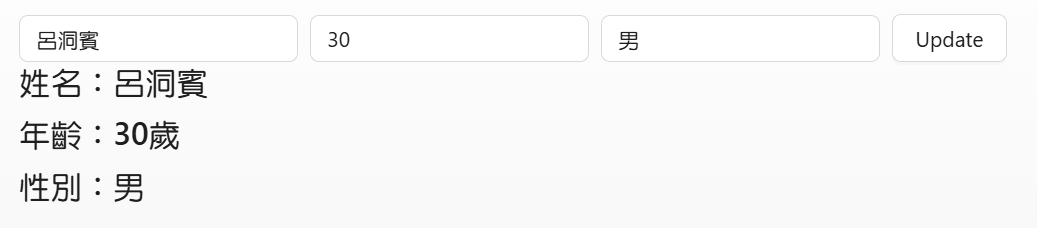
    )

}

export default Component

* 效果展示





# useEffect 效果鉤子

1. useEffect 可以讓你於 頁面渲染時，執行指定的操作
2. 每次渲染 執行

useEffect(()=>{

    // useEffect 『啟動』時要執行的操作

    return () => {

        // useEffect 『卸載』時要執行的操作

  }

})

1. 首次渲染 執行

useEffect(()=>{

    // useEffect 『啟動』時要執行的操作

    return () => {

        // useEffect 『卸載』時要執行的操作

    }

},[])

1. 特定對象觸發渲染 執行

useEffect(()=>{

    // Effect 『啟動』時要執行的操作

    return () => {

        // Effect 『卸載』時要執行的操作

    }

},[監測對象, 監測對象, ……])

import { useEffect, useState } from "react"

// 頁面顯示計時器，顯示頁面停留了幾秒

const VDOM: React.FC = () => {

  const [timer, setTimer] = useState(0)

  // Effect 僅在頁面首次渲染時執行

    useEffect(() => {

        // 設定計時器，每 1 秒 執行一次

        // 每次 時間 + 1秒

        setInterval(()=>{

            setTimer(timer => timer + 1)

        },1000)

    },[])

    return (

        <div>

            <h3>頁面停留 {timer} 秒</h3>

        </div>

    )

}

export default VDOM



# useRef 從輸入框獲取值

1. useRef 是 React 用來從 輸入框 中 獲取 輸入值 的 Hook。
2. 宣告：  
   const ref變數 = useRef<元件的Ref型態>(初始值)  
   ＊元件的Ref型態需要根據 使用的輸入框 來進行設定  
    若使用 Antd 的 Input 則要使用 InputRef  
   ＊初始值可以進行預設值的設定，建議使用時 將 初始值 設定為 null
3. 元素使用：  
   <Input type='text' ref={ref變數} onChange={改變事件} />  
   ＊透過 onChange 來觸發 狀態變數 的 改變事件，已達成 綁定的目的。  
    也就是 輸入框 內容改變時，會同時改變 後台的某個狀態變數
4. 取得數值：  
   const 變數 = inputRef.current?.input?.value

import React, { useRef, useState } from 'react';

import { Input, InputRef, Space } from 'antd';

const MyComponent: React.FC = () => {

  // useRef 的型態 根據 綁定元件 來使用，這裡因為是使用 Inupt，所以要用 InputRef 作為型態

  const inputRef = useRef<InputRef>(null);

  // 設定 姓名變數

  const [userName, setUserName] = useState<string>("")

  // 設定 姓名改變事件

  const chgUserName = () => {

    const refValue = inputRef.current?.input?.value

    const name = refValue !== undefined ? refValue : ""

    setUserName(name)

  }

  return (

    <div>

      {/\* 使用 ref 指向 Input 的 Ref \*/}

      <Space>

          <Input type='text' ref={inputRef} onChange={chgUserName} />

          <h1>我是{userName}</h1>

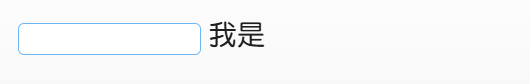
      </Space>

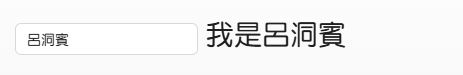
    </div>

  );

};

export default MyComponent;





# Ant Design Pro 的全域變數工具: useModel

在某些情況下，如：登入用戶的資訊管理，需要在多個頁面中共享這些信息。

為了這個目的 Ant Design Pro 框架 透過 @umijs/plugin-model 插件，

實現了一個 輕量級 的 全域資訊共享解決方案「useModel」。

1. 建立 model
2. 於 src/models 資料夾內，建立你所需要的 全域變數檔案  
   該檔案的格式 官方建議為 js, ts 或 tsx。
3. 你所建立的檔案名稱 就是 之後使用時，所要輸入的名稱。
4. 語法範例：

位置：src/models  
檔案名稱：檔名.tx

內容：

export default () => {

// 宣告 全域變數 建議設為 state(狀態機)

// 任何你所需要的函式

return {全域變數, 函式, ...}

}

1. 使用 model
2. import { useModel } from '@/.umi/plugin-model'
3. const 變數 = useModel('檔名')  
   ＊此時 變數 裡面會有 model 所 return 的東西

【src/models/counter.ts】

import { useState, useCallback } from 'react';

export default () => {

  const [counter, setCounter] = useState(0);

  const increment = useCallback(() => setCounter((c) => c + 1), []);

  const decrement = useCallback(() => setCounter((c) => c - 1), []);

  return { counter, increment, decrement };

};

【index.tsx】

import React from 'react';

import { Button, Space } from 'antd';

import { useModel } from '@/.umi/plugin-model';

const MyComponent: React.FC = () => {

  // 引入 model 全域變數 conuter.ts

  // 此時 counterModel 裡面有 conuter.ts 所設定的三個輸出

  // counter, increment, decrement

  const counterModel = useModel('counter')

  return (

    <div>

      <Space>

        <h2>目前數值{counterModel.counter}</h2>

<Button type='primary' onClick={counterModel.increment}>+1</Button>

        <Button type='primary' onClick={counterModel.decrement}>-1</Button>

      </Space>

    </div>

  );

};

export default MyComponent;





# React 的效能優化

想要優化 React 的執行效率，

可以透過 meno, useCallback, useMemo 來將部分功能停止渲染。

但因為這需要花費記憶體來進行資料暫存，因此建議 有需要再使用。

## memo 暫存組件

當 子元件 的 props 沒有改變時，子元件不會被重新渲染。

import React, { memo } from 'react';

const myExample: React.FC = memo((輸入參數) => {

    // 這裡進行 邏輯處理

  return (

    <>

要渲染的內容

    </>

  );

});

export default myExample;

## useCallback暫存函式本身

透過 監測某個狀態機 來判斷 函式要不要被重新渲染

當 監測的狀態數值改變 時，函式才會啟動重新渲染

＊因為 重新渲染函式 不會消耗多少資源，所以此方法很少使用

const 函式名稱 = useCallback((參數) => {

    函式邏輯

    return 函式回傳值

},[監測對象, 監測對象, ……])

## useMemo 暫存函式運算結果

暫存函式執行的結果，只有當 監測對象數值改變時，頁面重新渲染時 才會重新執行此函式

＊通常用於 每次運算需要花費大量時間，以避免造成畫面的卡頓

const 函式名稱 = useMemo((參數) => 要執行的函式運算, [監測對象, 監測對象, ……])