Vue 3 學習筆記

1. 組合式 API(Composition API)
2. 單文件元件(*.vue)
3. Pinia 全域資料管理: Pinia
4. Vue 官方路由: Vue-Router
5. Vue UI 框架: Element Plus

黃懷慶

2024/10/14

內容

環境架設

1. 安裝 node.js
   https://nodejs.org/en/
2. 安裝 IDE 工具: VS CODE
   https://code.visualstudio.com/
3. VS CODE 安裝 開發套件
   Auto Import
   Vue - Official

＊官方網站
https://zh-hk.vuejs.org/

透過官方工具建立專案

```
C:\Users\ray03\Desktop\Vue 3 演練>npm create vue@latest
Need to install the following packages:
create-vue@3.11.0
Ok to proceed? (y) y

Vue.js - The Progressive JavaScript Framework

√ 請輸入專案名稱： ... vue3-example
√ 是否使用 TypeScript？ ... 否 / 是
√ 是否啟用 JSX 支援？ ... 否 / 是
√ 是否引入 Vue Router 進行單頁應用程式開發？ ... 否 / 是
√ 是否引入 Pinia 用於狀態管理？ ... 否 / 是
√ 是否引入 Vitest 用於單元測試 ... 否 / 是
√ 是否要引入一款端對端（End to End）測試工具？ » 不需要
√ 是否引入 ESLint 用於程式碼品質檢測？ ... 否 / 是
√ 是否引入 Vue DevTools 7 擴充元件以協助偵錯？（試驗性功能） ... 否 / 是

正在建置專案 C:\Users\ray03\Desktop\Vue 3 演練\vue3-example...

專案建置完成，可執行以下命令：

  cd vue3-example
  npm install
  npm run dev
```

1. 開啟 CMD 執行 `npm create vue@latest` 即可透過官方工具進行專案建置
2. 切換到 專案資料夾後 進行後續安裝
   A. 安裝 API 工具 axios
      `npm install axios`
   B. 安裝 UI 框架: Element Plus
      `npm install element-plus --save`
      `npm install -D unplugin-vue-components unplugin-auto-import`
   C. 安裝依賴
      `npm install`
   D. 啟動
      `npm run dev`

＊如果需要 重新安裝依賴包，可以先刪除 node_modules 資料夾，再 執行安裝依賴。
＊Element Plus 官網：https://element-plus.org/zh-CN/

調整專案架構

為了整體程式擺放的方便，這裡先一次性地將後面的設定建立完畢，詳細的介紹，後面再進行。

- Element Plus 設定 自動引入
  - A. vite.config.js
    1.
```
import AutoImport from 'unplugin-auto-import/vite'
import Components from 'unplugin-vue-components/vite'
import { ElementPlusResolver } from 'unplugin-vue-components/resolvers'
```
    2.
```
export default defineConfig({
    // ...
    plugins: [
        // ...
        AutoImport({
        resolvers: [ElementPlusResolver()],
        }),
        Components({
        resolvers: [ElementPlusResolver()],
        }),
    ],
})
```

vue-demo > JS vite.config.js > ...

```
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
// Element Plus
import AutoImport from 'unplugin-auto-import/vite'
import Components from 'unplugin-vue-components/vite'
import { ElementPlusResolver } from 'unplugin-vue-components/resolvers'

// https://vite.dev/config/
export default defineConfig({
  plugins: [
    vue(),
    // Element Plus
    AutoImport({
      resolvers: [ElementPlusResolver()],
    }),
    Components({
      resolvers: [ElementPlusResolver()],
    }),
  ],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    }
  }
})
```

- Element Plus 整體設定為 黑暗主題
  A. main.js
     >> import 'element-plus/theme-chalk/dark/css-vars.css'
  B. index.html
     >> <html class="dark">

vue-demo > src > JS main.js > ...

```
1   // import './assets/main.css'
2
3   import { createApp } from 'vue'
4   import { createPinia } from 'pinia'
5
6   import App from './App.vue'
7   // vue-router
8   import router from './router'
9
10  // Element Plus
11  import ElementPlus from 'element-plus'
12  import 'element-plus/dist/index.css'
13  import 'element-plus/theme-chalk/dark/css-vars.css'
14
15  const app = createApp(App)
16
17  app.use(createPinia())
18  app.use(router)        // vue-router
19  app.use(ElementPlus)   // Element Plus
20  app.mount('#app')
21
```

vue-demo > index.html > html.dark

```
1   <!DOCTYPE html>
2   <html lang="en" class="dark">
3     <head>
4       <meta charset="UTF-8">
5       <link rel="icon" href="/favicon.ico">
6       <meta name="viewport" content="width=device-width, initial-scale=1.0">
7       <title>Vite App</title>
8     </head>
9     <body>
10      <div id="app"></div>
11      <script type="module" src="/src/main.js"></script>
12    </body>
13  </html>
```

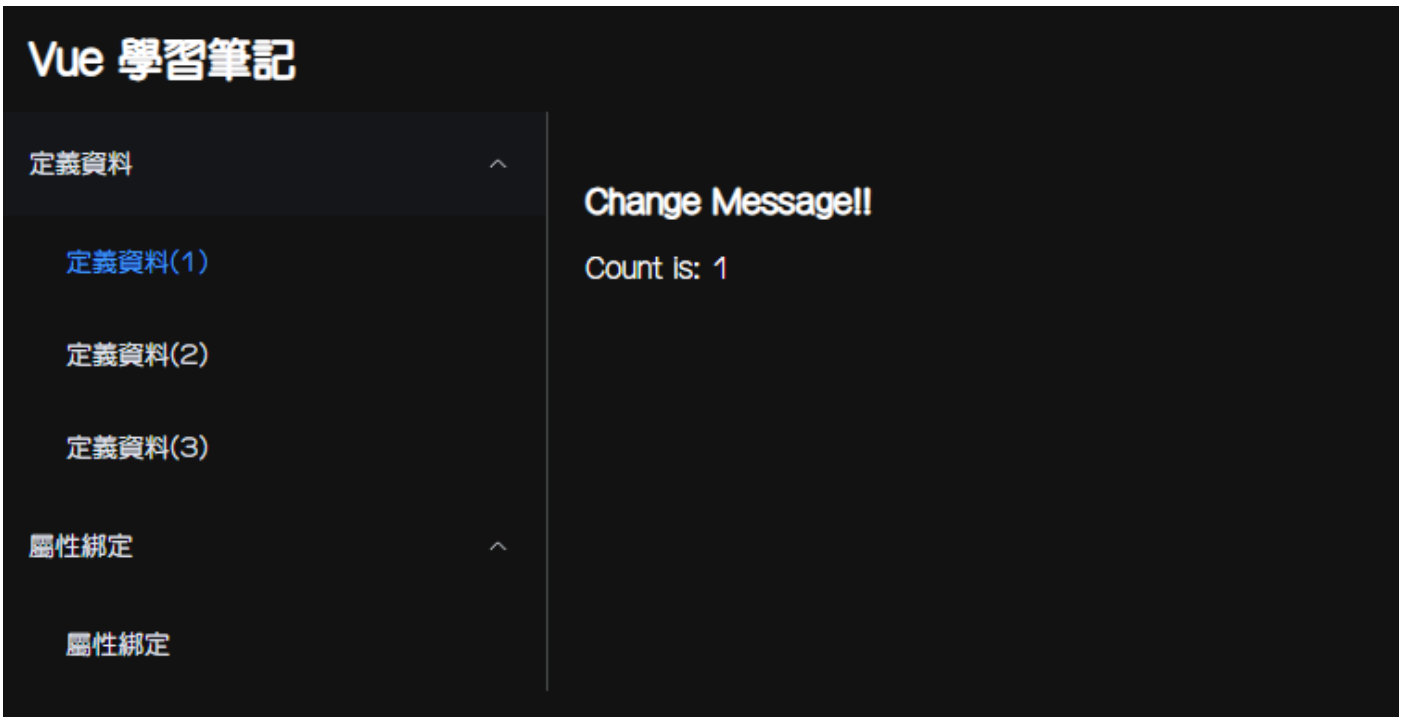
- 調整 App.vue
  A. 使用 Element Plus 的 layout 布局 + menu 組件，來建立 左側邊欄
  B. 使用 vue-router 來動態管理 連結設定，因此 vouter 文件也要有對應的設定

vue-demo > src > App.vue > ...

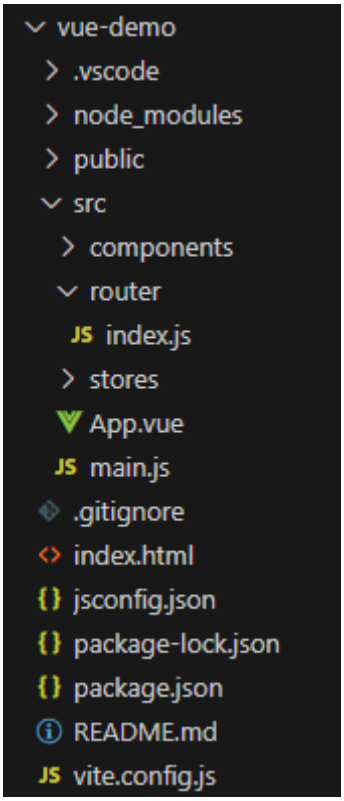
```vue
<script setup>
import { RouterView } from 'vue-router'
// vue-router
import router from './router'
const routesList = router.options.routes
</script>

<template>
  <div class="common-layout">
    <el-container>
      <!-- Hader 欄位 -->
      <el-header>
        <h1>Vue 學習筆記</h1>
      </el-header>

      <el-container>
        <!-- 左側邊欄 -->
        <el-aside>
          <!-- el-menu: 設定 router 代表要開啟 vue-router 功能 -->
          <el-menu router>
            <!-- el-sub-menu: 1. 透過 v-for 來 掃描 router 的全部設定資料                    -->
            <!--              2. :index 是 menu 的 key，要設定不同值，才不會同時全部打開/關閉 -->
            <el-sub-menu
              v-for="router in routesList"
              :index="router.path"
            >
              <!-- 這裡的 template 代表 第一層的 中文名稱 -->
              <template #title>{{ router.name }}</template>
              <!-- el-menu-item: 1. 透過 v-for 來 掃描 router.children 的全部 子層級資料 已建立 RouterLink -->
              <!--               2. 這裡的 :index 就是 實際上 Link 要使用的路徑 【父路徑/子路徑】 -->
              <!--               3. 最後在中間 嵌入 子路徑的名稱 -->
              <el-menu-item
                v-for="children in router.children"
                :index="router.path + '/' + children.path"
              >
                {{ children.name }}
              </el-menu-item>
            </el-sub-menu>
          </el-menu>
        </el-aside>

        <!-- 右內容欄: 直接使用 RouterView 來顯示頁面就好了 -->
        <el-main>
          <RouterView />
        </el-main>
      </el-container>

    </el-container>
  </div>
</template>
```

vue-demo > src > router > index.js > router > routes

```js
import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
        path: "/Ch1",      // 父網址：開頭需要有【/】
        name: "定義資料",  // 名稱
        children: [        // 子層級資料的擺放處
            {
                path: "Demo1",                // 子網址：不需要【/】
                name: "定義資料(1)",           // 名稱
                meta: {requiresAuto: true},    // 透過 meta 屬性，來設定檢核用參數：【requiresAuto: true】 代表需要權限檢核
                component: () => import('../components/Demo1.vue')  // 頁面檔案位置
            },
            {
                path: "Demo2",
                name: "定義資料(2)",
                meta: {requiresAuto: true},
                component: () => import('../components/Demo2.vue')
            },
            {
                path: "Demo3",
                name: "定義資料(3)",
                meta: {requiresAuto: true},
                component: () => import('../components/Demo3.vue')
            },
        ]
    },
    {
        path: "/Ch2",
        name: "屬性綁定",
        children: [
            {
                path: "Demo4",
                name: "屬性綁定",
                meta: {requiresAuto: true},
                component: () => import('../components/Demo4.vue')
            },
        ]
    },
  ]
})
```



- 刪除預設範例程式
  A. scr/assets 資料夾全部刪除
  B. src/router 資料夾全部刪除
  C. src/views 資料夾全部刪除
  D. src/components 清空內部檔案
  E. src/stores 清空內部檔案

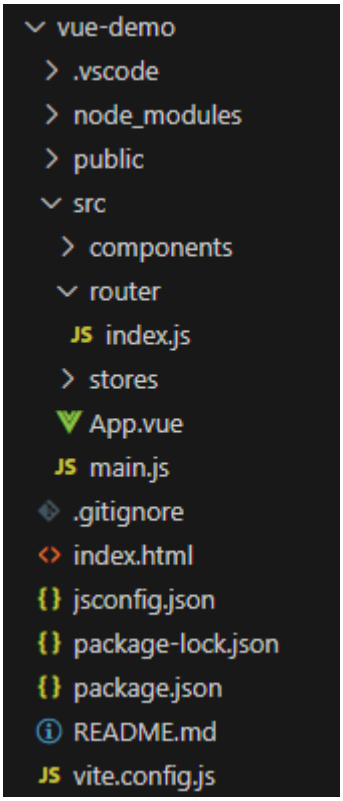


圖一：調整後的資料結構

資料架構

1. 資料夾結構

A. Vue 的 頁面檔 的 副檔名 為 `*.vue`



頁面啟動：
index.html -> main.js -> App.vue
(使用 vue-router 串連各頁面)

【*.vue】為 vue 的應用程式頁面檔案

「components」資料夾 為 擺放子元件的地方

「stores」資料夾 為 擺放 工具程式的地方

「config」資料夾 為 擺放設定檔的地方
1. router/index.js 為 vue-router 的設定檔

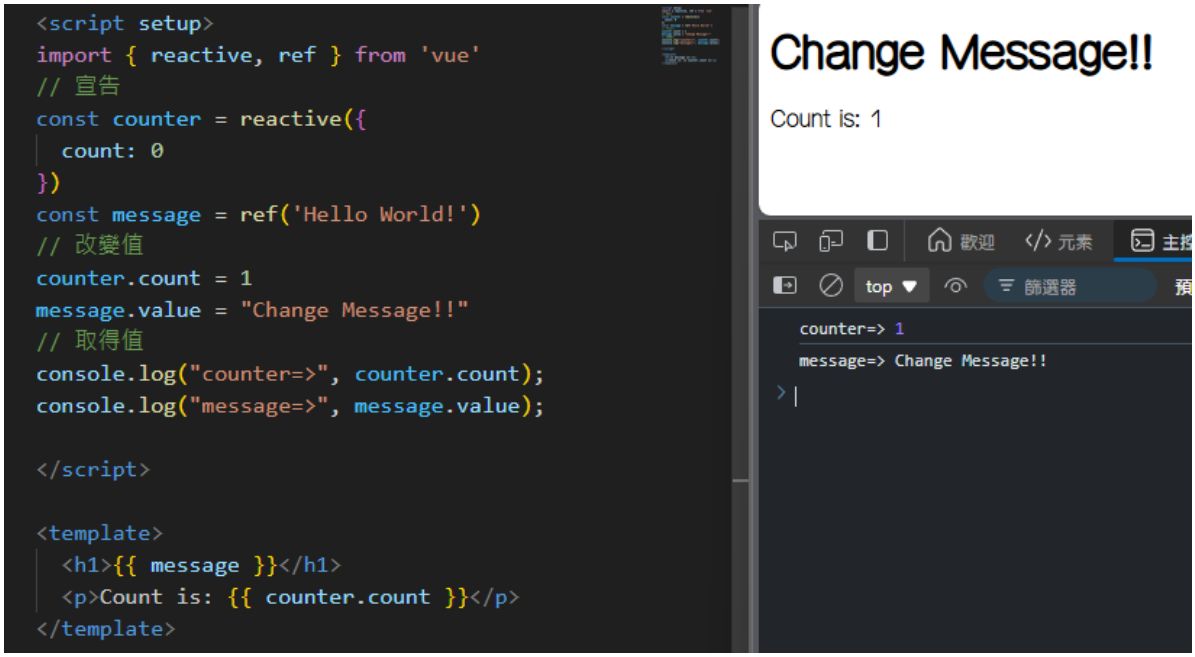
2. vue 的頁面架構

```
<script setup>
    /**這個區域用來處理程式邏輯
     * 「script setup」會讓 Vue 自動將 邏輯 暴露給 template 使用
     * 建議 code 的擺放順序為
     * 1. import
     * 2. 官方相關的變數宣告
     * 3. 變數宣告
     * 4. 函式
     * 5. 生命週期函式 (依照生命週期執行順序擺放)
     */
</script>

<template>
    <!-- 這個區域用來設定 頁面的內容 -->
    <!-- 如果要使用 script 裡面的變數 -->
    <!-- 要使用 雙大括號 包起來，如：「{{ 變數 }}」 -->
</template>

<style scoped>
    /* 這個區域用來設定 CSS 樣式 */
    /* 有設定 scoped 屬性，該樣式為 本地樣式，只會部屬在當前頁面上 */
    /* 不會渲染到 子組件中 */
</style>
```

定義資料：ref, reactive 和 computed

- 響應式狀態：
  資料數值改變時，畫面會同步渲染

1. ref
   >> 會包裝成一個物件，數值 保存在 value 欄位中。
   >> 數值 可以是 任何型態的資料。
   A. 宣告
      const 變數 = ref(數值)
   B. Setting 方法
      變數.value = 要改變的數值
   C. Getting 方法
      變數.value
   D. template 使用
      {{ 變數 }}
2. reactive
   >> 數值 只可以是 物件{} 或 陣列物件[]
   A. 宣告
      const 變數 = reactive(數值)
   B. Setting 方法
      變數.欄位 = 要改變的數值
   C. Getting 方法
      變數.欄位
   D. template 使用
      {{ 變數.欄位 }}
3. 兩者差異 與 使用建議
   A. 當資料 非物件 或 不需要監聽 時，使用 ref。
   B. 當資料為 物件 時，使用 reactive。
   C. 當 ref 擺放物件時，監聽器 watch 無法監聽 物件內的欄位，
      因此 當資料為物件時，不建議使用 ref。



- 計算屬性（computed）

用來描述 依賴響應式狀態 的複雜邏輯

當 響應式狀態 的內容改變時，計算屬性 的數值 也會同步改變，並且 重新渲染畫面

1. 一般使用

>> 只有 getting 方法

A. 宣告

```
const 變數 = computed(() => {
  處理邏輯
  return 經過邏輯判斷後的數值
})
```

B. Getting 方法

```
變數.value
```

C. template 使用

```
{{ 變數 }}
```

2. 進階使用

>> 透過物件的刑事，將可以使用 getting 和 setting 方法

A. 宣告

```
const 變數 = computed({
  get: () => {
      處理邏輯
      return 經過邏輯判斷後的數值
  },
  set: (val) => {
      改變 響應式狀態 的數值
      // val = 要改變的數值
  }
})
```

B. Getting 方法

```
變數.value
```

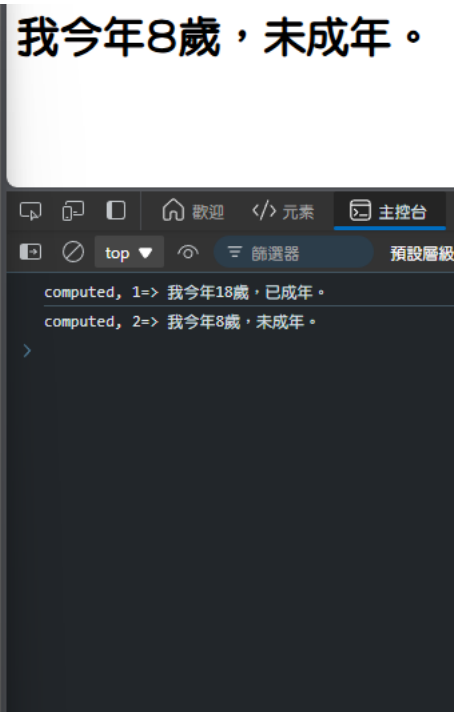
C. Setting 方法

```
變數.value = 要改變的數值
```

D. template 使用

```
{{ 變數 }}
```

```
<script setup>
    import { computed, reactive, ref } from 'vue';
    // 響應式狀態
    const age = ref(18)
    // 宣告
    const isAdult = computed({
        get: () => {
            const adultDesc = age.value >= 18 ? "已成年" : "未成年"
            return `我今年${age.value}歲，${adultDesc}。`
        },
        set: (val) => {
            age.value = val
        }
    })
    // Getting
    console.log("computed, 1=>",isAdult.value);
    // Setting
    isAdult.value = 8
    console.log("computed, 2=>",isAdult.value);
</script>

<template>
    <h1>{{ isAdult }}</h1>
</template>
```



屬性綁定 v-bind:

要在 template 的 元件 的 屬性，使用 script 的變數

就需要 將該屬性 設定為 屬性綁定

1. 一般寫法

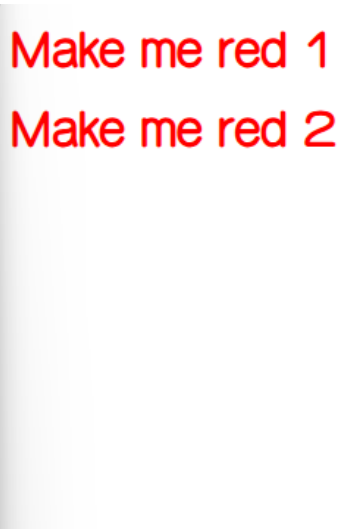
```
<template>
    <元件 v-bind:屬性 = "變數"></元件>
</template>
```

2. 簡化寫法

```
<template>
    <元件 :屬性 = "變數"></元件>
</template>
```

＊因為「v-bind:」很常使用，所以可以簡化為「:」

```
<script setup>
    import { ref } from 'vue'

    const titleClass = ref('title')
</script>

<template>
  <h1 v-bind:class="titleClass">Make me red 1</h1>
  <h1 :class="titleClass">Make me red 2</h1>
</template>

<style>
    .title {
    color: red;
    }
</style>
```



事件監聽 v-on:

當 template 的 組建 的 事件，要觸發 script 的函式
就需要 將該事件 設定為 監聽事件

1. 一般寫法

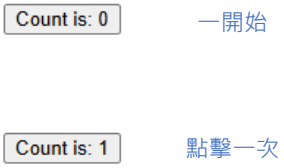
```
<template>
    <元件 v-on:事件 = "函式"></元件>
</template>
```

2. 簡化寫法

```
<template>
    <元件 @事件 = "函式"></元件>
</template>
```

＊因為「v-on:」很常使用，所以可以簡化為「@」

```
<script setup>
import { ref } from 'vue'

const count = ref(0)
const addCount = () => {
    count.value++
}
</script>

<template>
  <button @click="addCount">Count is: {{ count }}</button>
</template>
```



一開始

點擊一次

嵌套元件

在設計 Vue 的頁面時，通常會透過 嵌套 子元件 來創建。

使用方式：

1. 撰寫 子元件

   如：撰寫 XXX.vue

2. import 子元件

   如：import XXX from './components/XXX.vue'

3. 於 template 使用子元件

   如：

```
<template>
   <XXX />
</template>
```

元件資料傳遞(父傳子) props

當 需要將資料 傳遞給 子元件 使用，就必須使用 props。

- 父元件

1. 將 要傳遞的資料，透過 屬性 傳遞過去

```
<template>
  <元件 :屬性 A="變數"
        屬性 B="數值" ... />
</template>
```

- 子元件

1. 宣告

>> 建議每個屬性都要設定 型態 與 預設值

```
const props = defineProps({
  屬性 A: {
      type: 型態,
      default: 預設值
  }, ...
})
```

2. 取值

```
props.屬性 A
```

Demo7.vue

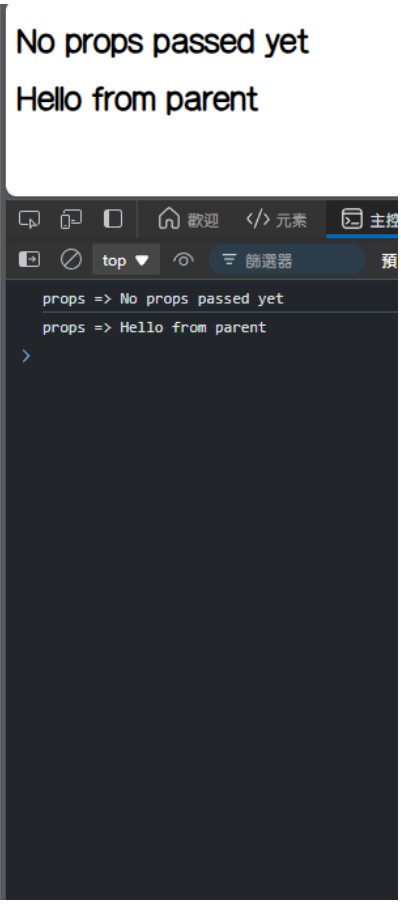
src > components > Demo7.vue > {} template > ChildComp

```
1  <script setup>
2  import { ref } from 'vue'
3  import ChildComp from './ChildComp2.vue'
4
5  const greeting = ref('Hello from parent')
6  </script>
7
8  <template>
9    <ChildComp />
10   <ChildComp :msg="greeting"/>
11 </template>
```

ChildComp2.vue

src > components > ChildComp2.vue > {} script setup > props > msg

```
1  <script setup>
2  const props = defineProps({
3    msg: {
4      type: String,
5      default: "No props passed yet"
6    }
7  })
8  console.log("props =>",props.msg);
9
10 </script>
11
12 <template>
13   <h2>{{ props.msg }}</h2>
14 </template>
```



元件資料傳遞(子傳父) emit

若 子元件 需要將資料 傳遞給 父元件 時，要將傳遞的資料 透過 emit 封裝

- 子元件

  透過 將 要傳遞的資料，放到 自定義事件 中，來進行傳遞

1. 宣告 自定義事件

```
const emit = defineEmits(["自定義事件"])
```

2. 設定傳遞資料

```
emit("自定義事件", 要傳遞資料)
```

- 父元件

  監聽 自定義事件，傳遞的資料 會以 input 的形式，送進 監聽事件 對應啟動的 函式 中

1. 語法

```
<script setup>
 const 對應函式 = (傳遞資料) => {
     處理邏輯
 }
</script>
<template>
 <XXX @自定義事件="對應函式" />
</template>
```

Demo8.vue

src > components > Demo8.vue > {} script setup > callBack

```vue
<script setup>
import { ref } from 'vue'
import ChildComp3 from './ChildComp3.vue'

const number = ref(0)
const callBack = (data) => {
  number.value = data
}

</script>

<template>
  <p>{{ number }}</p>
  <ChildComp3 @addNumber="callBack"/>
</template>
```

ChildComp3.vue

src > components > ChildComp3.vue > {} script setup > add

```vue
<script setup>
import { ref } from 'vue'
const amt = ref(0)
const emit = defineEmits(['addNumber'])
const add = (a, b) => {
    amt.value = a + b
    emit("addNumber", amt.value)
}

</script>

<template>
  <button @click="add(amt,1)">按我 + 1</button>
</template>
```

0 一開始

按我 + 1

1

按我 + 1 點擊一次

條件渲染 v-if 和 v-show

根據 條件 來決定 元件是否要被渲染。

也就是 透過 if-else-if 或 v-show 的條件判斷 來渲染元件。

＊簡單的情況 會使用 v-show，複雜的情況 才可能使用 v-if（比較耗資源）

＊可以用在元件的權限控管。

1. v-if

   當 v-if 為 true 時，渲染元件

   <元件 A v-if="布林變數" />

2. v-else-if

   搭配 v-if 使用， 當 v-if 為 false 才會進行判斷

   且 當 v-else-if 為 true 時，渲染元件

   <元件 B v-else-if="布林變數" />

3. v-else

   搭配 v-if 或 v-if、v-else-if 使用

   當 前面的條件都是 false 時，渲染此元件

   <元件 C v-else />

4. v-show

十分單純的條件渲染，true 會進行渲染；false 不會進行渲染

<元件 A v-show="布林變數" />

Demo10.vue

```vue
<script setup>
  import { ref } from 'vue'
  import ChildComp5_1 from './ChildComp5_1.vue'
  import ChildComp5_2 from './ChildComp5_2.vue'
  import ChildComp5_3 from './ChildComp5_3.vue'
  const isTeacher = ref(false)
  const isStudent = ref(false)
  const changeRenk = () => {
    // 工友 -> 老師 -> 學生
    if (isTeacher.value) {
      isTeacher.value = false
      isStudent.value = true
    } else if (isStudent.value) {
      isStudent.value = false
    } else {
      isTeacher.value = true
    }
  }
</script>
<template>
  <button @click="changeRenk" >身分切換</button>
  <ChildComp5_1 v-if="isTeacher" />
  <ChildComp5_2 v-else-if="isStudent" />
  <ChildComp5_3 v-else />
  <h1 v-show="isStudent" >今年即將畢業</h1>
</template>
```

ChildComp5_1.vue

```vue
<template>
    <h1>我是老師</h1>
</template>
```

ChildComp5_2.vue

```vue
<template>
    <h1>我是學生</h1>
</template>
```

ChildComp5_3.vue

```vue
<template>
    <h1>我是工友</h1>
</template>
```

身分切換 一開始

我是工友

身分切換 按一次

我是老師

身分切換 按二次

我是學生

今年即將畢業

身分切換 按三次

我是工友

列表渲染 v-for

透過 v-for 可以根據 陣列的資料 對 元件 進行 迭代渲染(foreach)

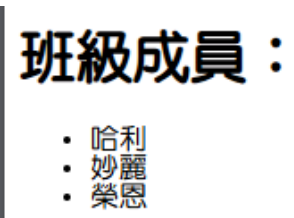
1. 語法

<元件 v-for="變數 in 陣列變數" :kye="變數.key" > {{ 變數.欄位 }} </元素>

Demo11.vue

src > components > Demo11.vue > {} script setup > name

```vue
<script setup>
import { reactive } from 'vue';
  let id = 0
  const userList = reactive([])
  userList.push({
    id: id++,
    name: "哈利"
  })
  userList.push({
    id: id++,
    name: "妙麗"
  })
  userList.push({
    id: id++,
    name: "榮恩"
    })
</script>

<template>
  <h1>班級成員：</h1>
  <ul>
    <li v-for="user in userList" :key="user.id">
      {{ user.name }}
    </li>
  </ul>
</template>
```



動態切換元件 component

當需要根據不同狀態，顯示不同子頁面，可以使用 component 來達成。

＊不推薦使用 v-if 來進行處理，因為會導致 template 中存在邏輯，造成程式碼混亂。

1. import 子頁面

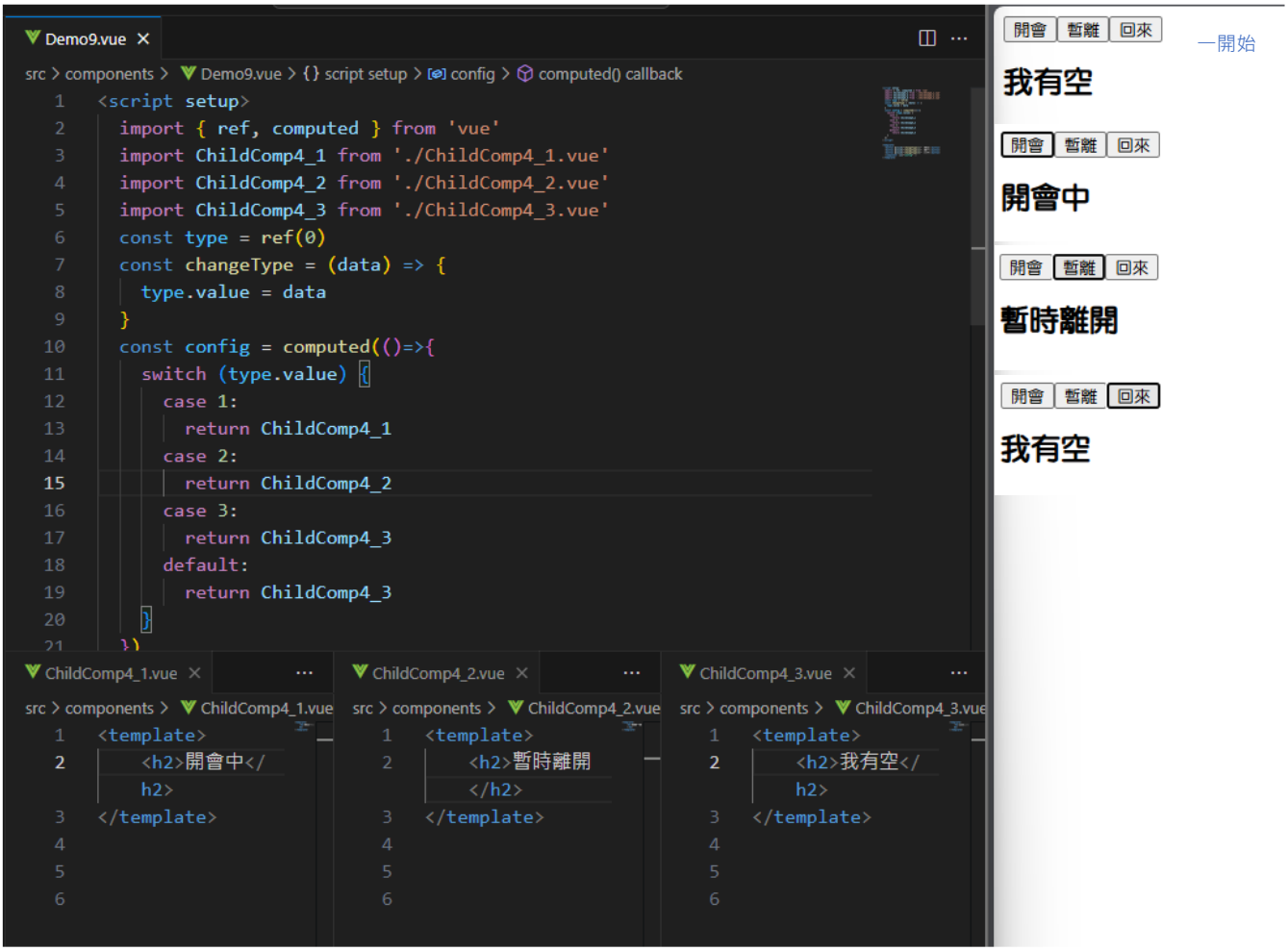
```
import 頁面 A from './components/頁面 A.vue'
import 頁面 B from './components/頁面 B.vue'
import 預設頁面 from './components/預設頁面.vue'
```

2. 設定 切換函式

```
const 切換函式 = computed(()=>{
    switch (判斷變數) {
        case 狀態 A:
            return 頁面 A;
        case 狀態 B:
            return 頁面 B;
        defalut:
            return 預設頁面;
    }
})
```

3. template 設定頁面

```
<template>
    <component :is="切換函式"
</template>
```

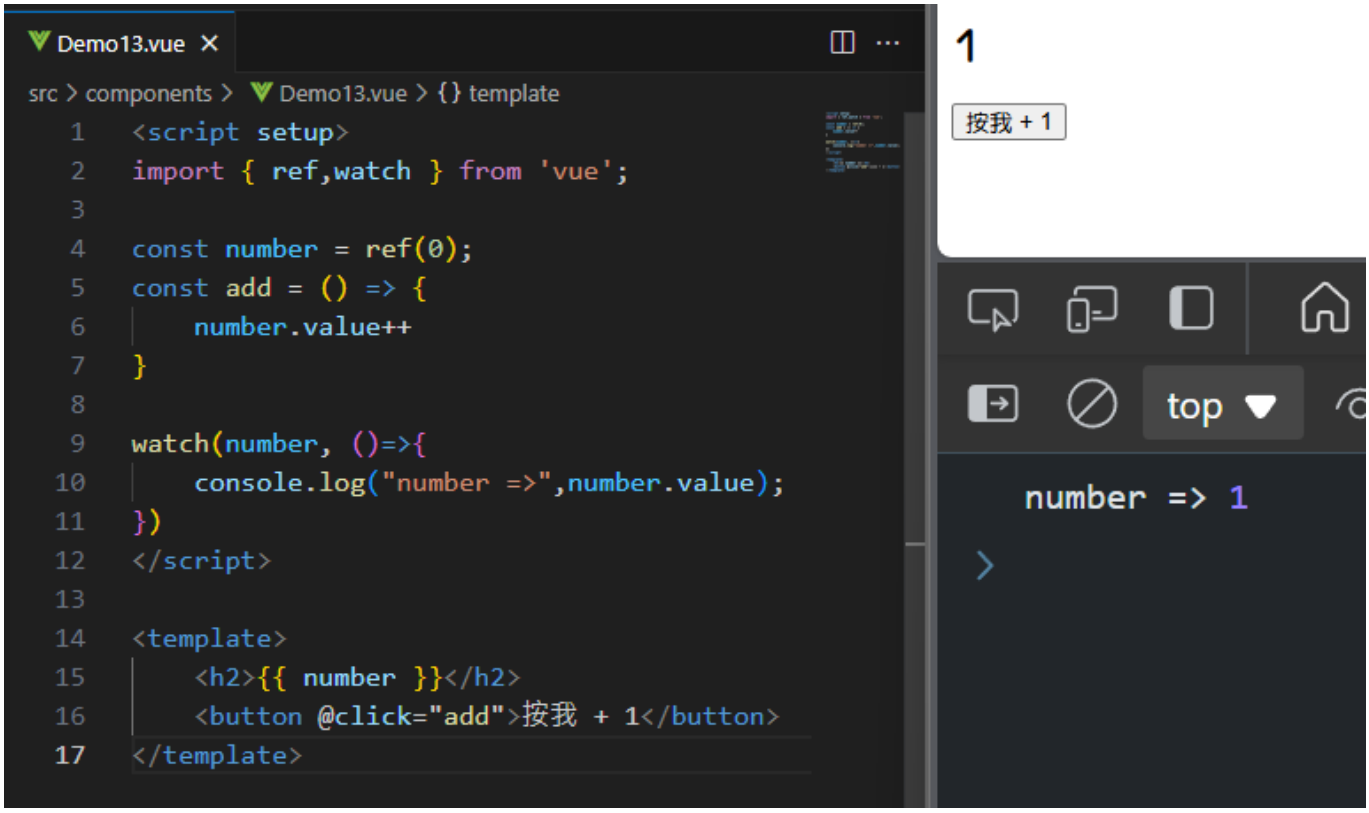


監聽器 watch

當 變數的資料 改變時，需要 觸發某種行為，可以透過 watch 來達成此需求。

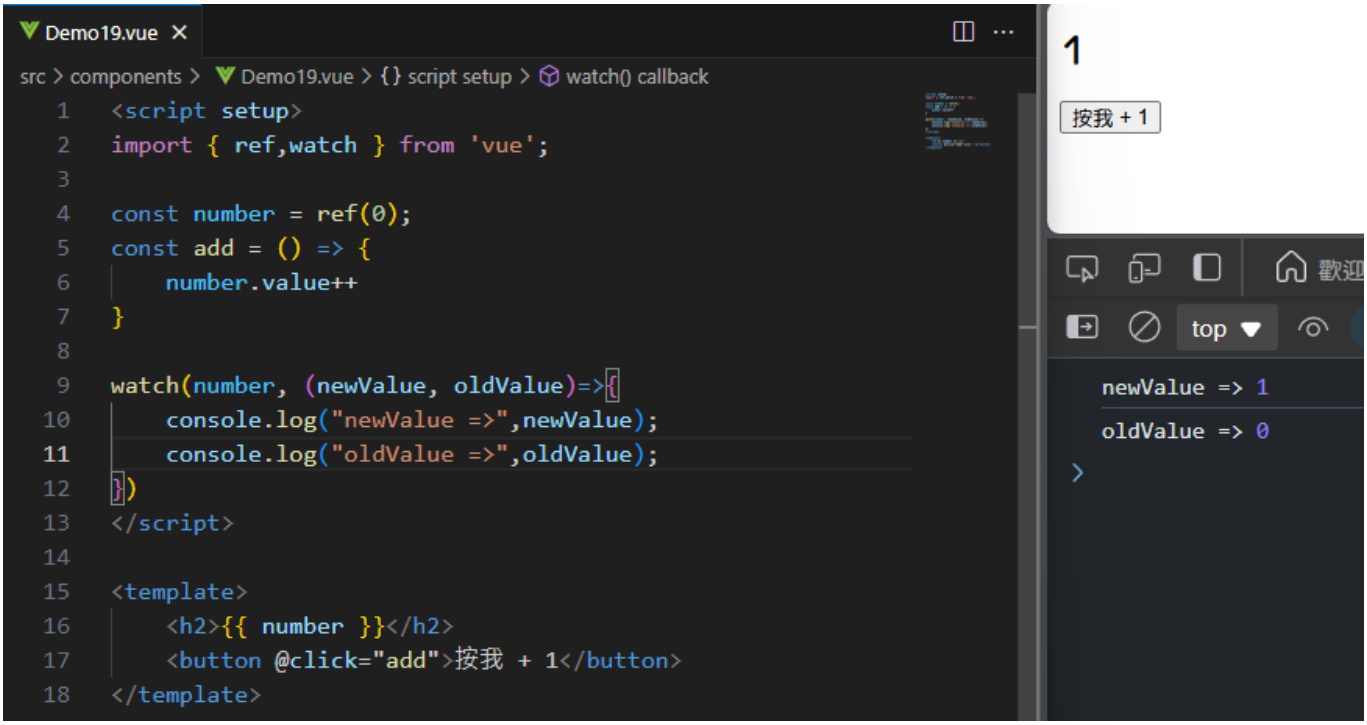
1. 一般用法

```
watch(監聽變數, ()=>{
    要進行的邏輯處理
})
```



2. 進階用法

```
watch(監聽變數, (新數值, 舊數值)=>{
    要進行的邏輯處理
})
```



資料雙向綁定 v-model

想要將 input 元件 的輸入值，綁定到 script 的變數中
並且 當 輸入值改變時，畫面會同步更新
可以投過 v-model 搭配 ref 來達成此要求。

1. 語法
   A. 設定一個響應式狀態變數
      ```
      const 狀態變數 = ref(初始值)
      ```
   B. 於 template 的 input 元件 增加 v-model 屬性
      ```
      <input type="text" v-model="狀態變數"/>
      ```
2. v-model 的 修飾符（修飾符可以同時使用好多個）
   A. lazy
      >> 綁上 change 事件，也就是 完全輸入完畢(焦點離開)，才會觸發資料同步。
      ```
      <input type="text" v-model.lazy="狀態變數"/>
      ```
   B. number
      >> 當 輸入值 為 數字時，會自動轉型為 數字型態
      >> (數字 + 文字 = 只留下數字)(文字 + 數字 = 文字型態)
      ```
      <input type="text" v-model.number="狀態變數"/>
      ```
   C. trim
      >> 資料同步時，會刪除 輸入值前後的空格
      ```
      <input type="text" v-model.trim="狀態變數"/>
      ```
3. v-model 可以綁定的元件
   A. input
   B. textarea
   C. selct
   D. 自定義元件
4. 自定義元件使用 v-model
   A. 子元件
      I. 設定 v-model 預留的 props 和 emit
      ```
      const props = defineProps(["modelValue"])
      const emit  = defineEmits(["update:modelValue"])
      ```
      II. template 的 input 元件 設定 props 和 emit
      ```
      <template>
         <input
            type="text"
            :value="props.modelValue"
            @input="emit('update:modelValue', $event.target.value)" />
      </template>
      ```
      *「$event.target.value」: 輸入框輸入完畢後的資料
   B. 父元件
      ```
      <template>
      ```

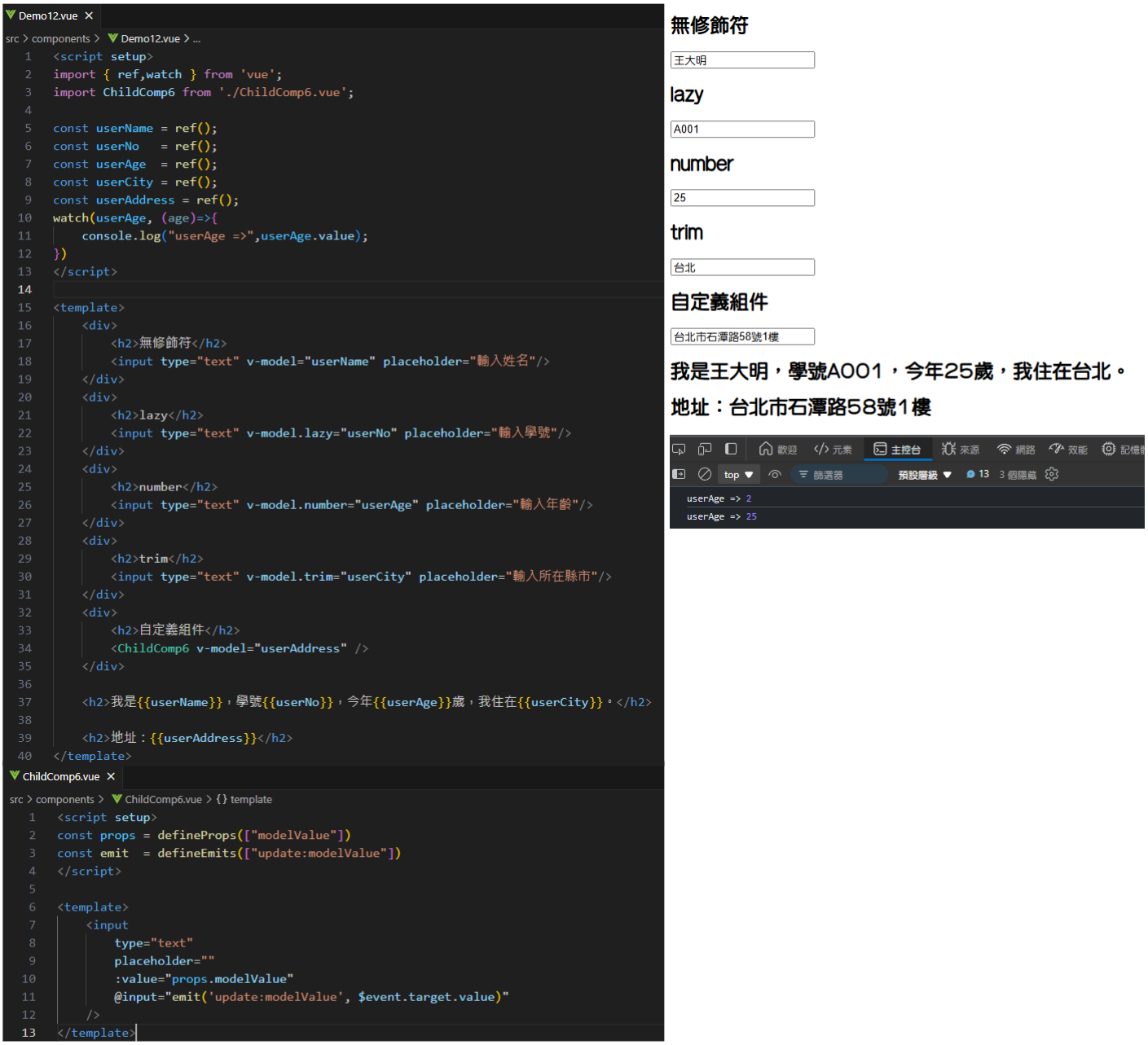
```
    <子元件 v-model="狀態變數" />
</template>
```

Demo12.vue

```vue
<script setup>
import { ref,watch } from 'vue';
import ChildComp6 from './ChildComp6.vue';

const userName = ref();
const userNo   = ref();
const userAge  = ref();
const userCity = ref();
const userAddress = ref();
watch(userAge, (age)=>{
    console.log("userAge =>",userAge.value);
})
</script>

<template>
    <div>
        <h2>無修飾符</h2>
        <input type="text" v-model="userName" placeholder="輸入姓名"/>
    </div>
    <div>
        <h2>lazy</h2>
        <input type="text" v-model.lazy="userNo" placeholder="輸入學號"/>
    </div>
    <div>
        <h2>number</h2>
        <input type="text" v-model.number="userAge" placeholder="輸入年齡"/>
    </div>
    <div>
        <h2>trim</h2>
        <input type="text" v-model.trim="userCity" placeholder="輸入所在縣市"/>
    </div>
    <div>
        <h2>自定義組件</h2>
        <ChildComp6 v-model="userAddress" />
    </div>

    <h2>我是{{userName}}，學號{{userNo}}，今年{{userAge}}歲，我住在{{userCity}}。</h2>

    <h2>地址：{{userAddress}}</h2>
</template>
```

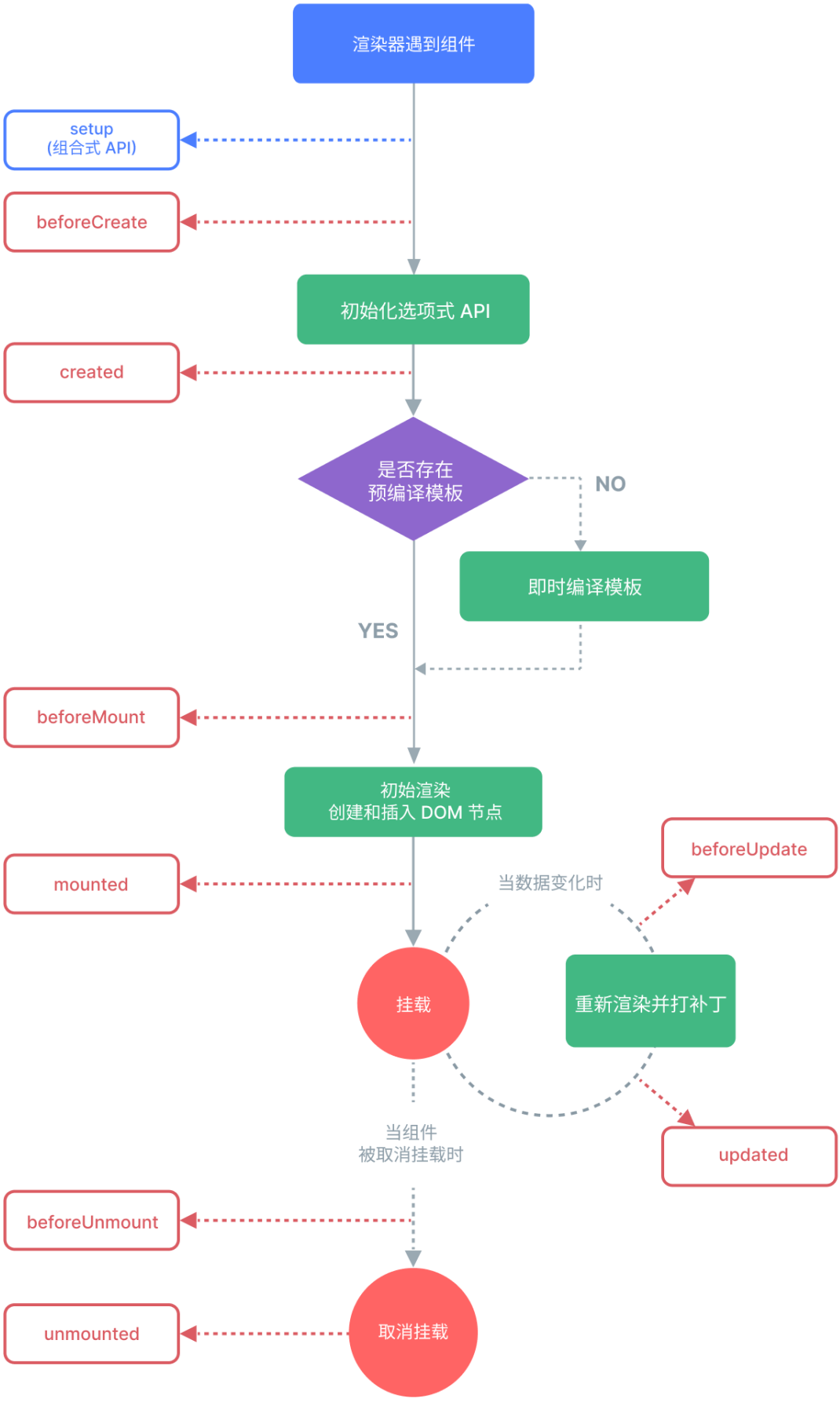
ChildComp6.vue

```vue
<script setup>
const props = defineProps(["modelValue"])
const emit  = defineEmits(["update:modelValue"])
</script>

<template>
    <input
        type="text"
        placeholder=""
        :value="props.modelValue"
        @input="emit('update:modelValue', $event.target.value)"
    />
</template>
```



生命週期函數

每個 Vue 元件 從 初始化創建 到 取消掛載 的各個階段，
都有設定 生命週期函數 來讓 開發人員 設定 需要進行的邏輯處理。



上圖中，紅框代表 此時生命週期的名稱。

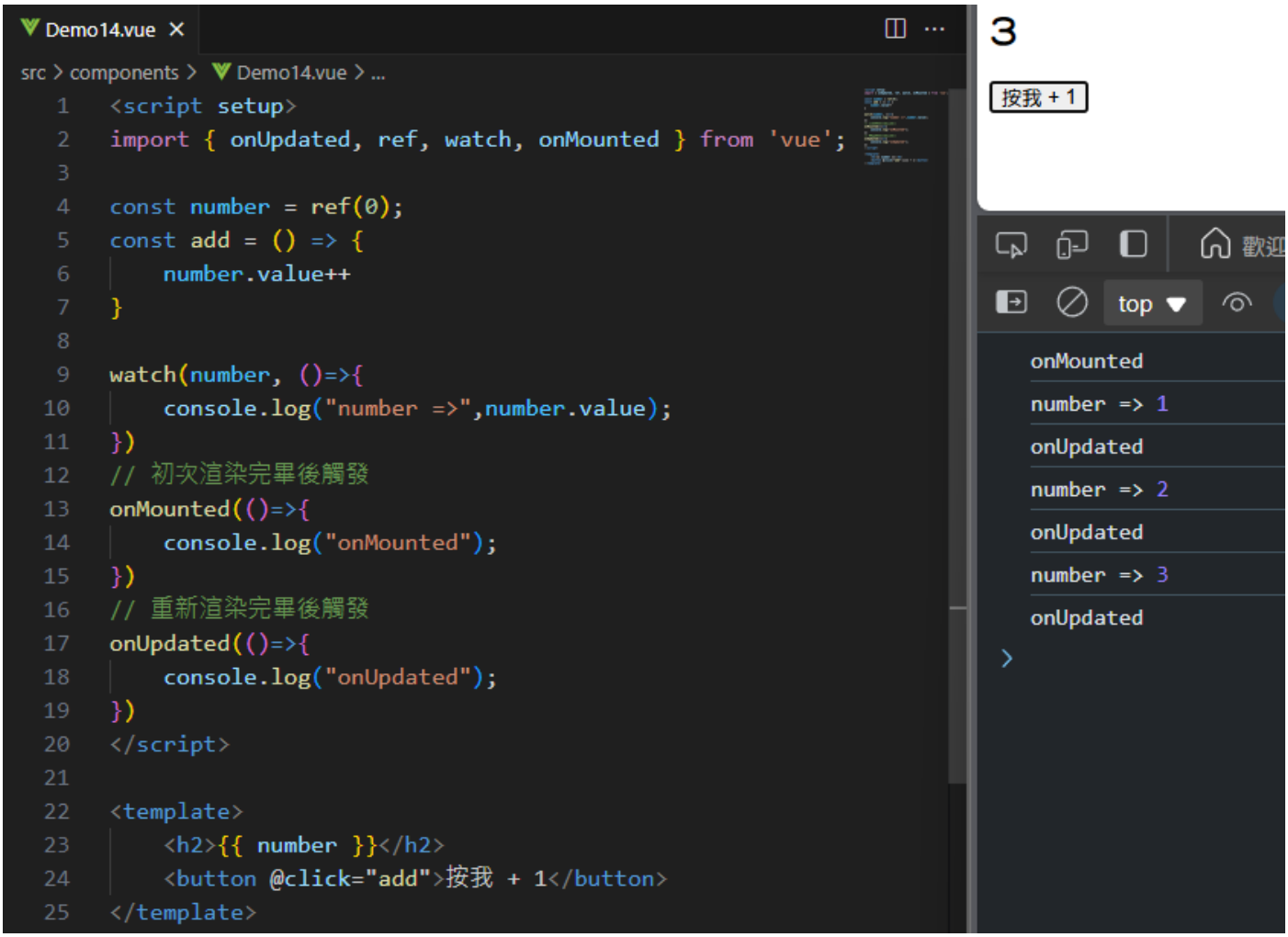
如：

初始渲染完成時，該階段為「mounted」，此時有對應的生命週期函數「onMounted」可以使用。

頁面重新渲染後，該階段為「updated」，此時有對應的生命週期函數「onUpdated」可以使用。

1. 語法範例

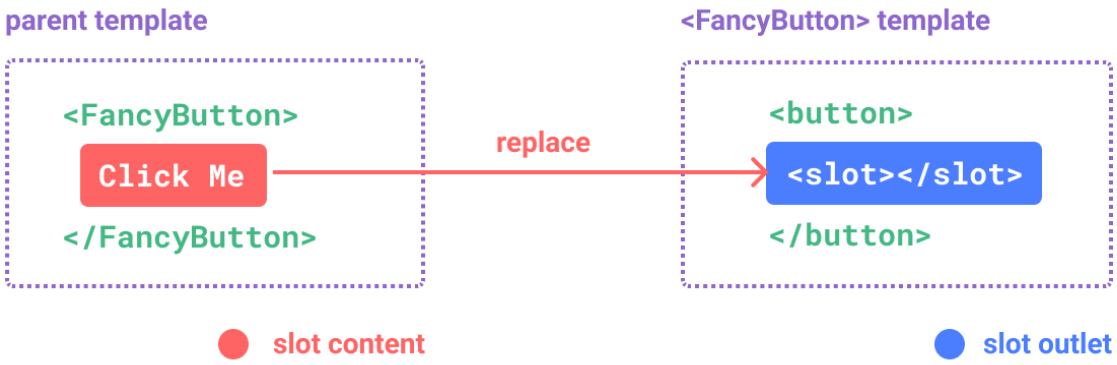
```
onMounted(()=>{
    初始化渲染完畢後，要進行的邏輯處理
})
onUpdated(()=>{
    重新渲染完畢後，要進行的邏輯處理
})
```



插槽 slot

要從 父元件 將資料傳給 子元件的頁面 使用，除了透過 props 外，
若 子元件 僅只有 template 中會使用到 該資料，
則 還可以透過 slot 的方法傳遞資料。

- 一般用法



1. 子元件

```
<template>
    <slot> 無傳遞資料時的預設值 </slot>
</template>
```

2. 父元件

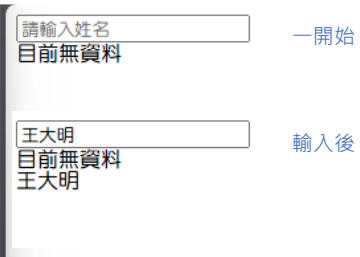
```
<template>
    <子元件> 要傳遞的資料 </子元件>
</template>
```

Demo15.vue

```
<script setup>
import { ref } from 'vue';
import ChildComp7 from './ChildComp7.vue';
const name = ref('')
</script>

<template>
    <input type="text" v-model="name" placeholder="請輸入姓名"/>
    <ChildComp7></ChildComp7>
    <ChildComp7>{{ name }}</ChildComp7>
</template>
```

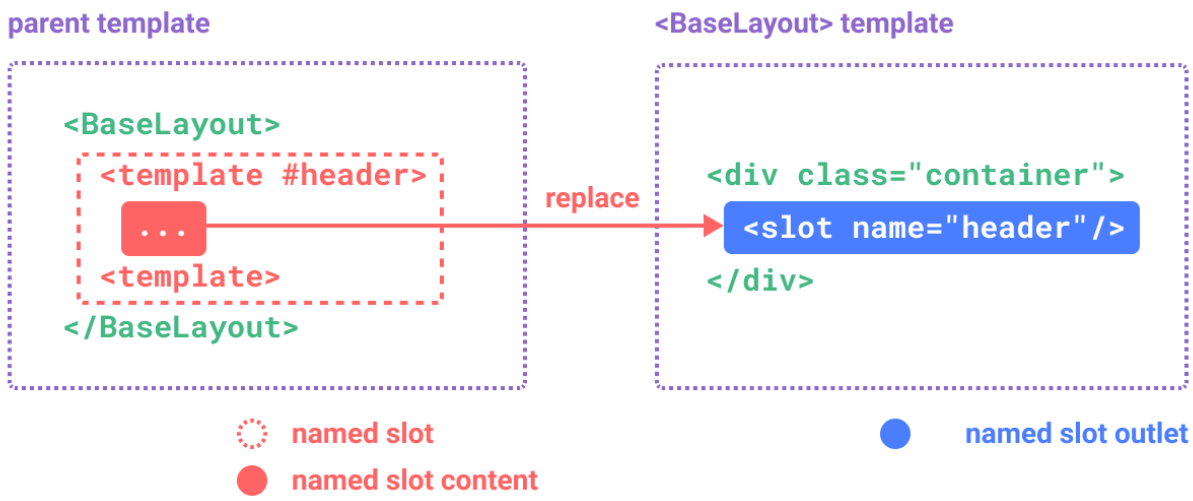
ChildComp7.vue

```
<template>
    <div>
        <slot>目前無資料</slot>
    </div>
</template>
```



- 進階用法

子頁面 在設定 slot 時，也可以設定 多個 slot，透過 slot ame 進行區分。
於 父頁面 使用時，只需要透過 <template v-slot:slotName>內容</template> 即可使用。
＊「v-slot:」可以簡寫為「#」



Demo20.vue

```vue
<script setup>
import ChildComp9 from './ChildComp9.vue';
</script>

<template>
    <ChildComp9>
        <template #header>
            <p>插入 header 文字</p>
        </template>
        <template #main>
            <p>插入 main 文字</p>
        </template>
        <template #footer>
            <p>插入 footer 文字</p>
        </template>
    </ChildComp9>
</template>
```

ChildComp9.vue

```vue

<template>
    <header>
        <slot name="header"></slot>
    </header>
    <main>
        <slot name="main"></slot>
    </main>
    <footer>
        <slot name="footer"></slot>
    </footer>
</template>
```

插入 header 文字

插入 main 文字

插入 footer 文字

共用邏輯的撰寫

在開發的過程中，可以將 共用邏輯 移到外部 進行單獨撰寫

並在元件中，透過 import 共用函式 來進行 函式呼叫

＊vue 官方建議：

1. 程式名稱 和 函式名稱 建議相同。
2. 名稱使用 use 開頭。

Demo16.vue

src > components > Demo16.vue > {} template

```vue
<script setup>
import { ref } from 'vue';
import { useAmt } from '../stores/useAmt.ts'
const xAmt = ref(0)
const yAmt = ref(0)
const {sumAmt, setSumAmt} = useAmt();
</script>

<template>
    <input type="text" v-model.number="xAmt" @change="setSumAmt(xAmt,yAmt)" />
    {{ ` + ` }}
    <input type="text" v-model.number="yAmt" @change="setSumAmt(xAmt,yAmt)"/>
    {{ ` = ${sumAmt}` }}
</template>
```

useAmt.ts

src > stores > useAmt.ts > useAmt > setSumAmt

```ts
import { ref } from 'vue';
export function useAmt() {
    const sumAmt = ref(0)

    const setSumAmt = (x: number, y:number) => {
        sumAmt.value = x + y
    }

    return {
        sumAmt, setSumAmt
    }
}
```

0 + 0 = 0

5 + 9 = 14

透過 共用邏輯 處理 API

此範例會透過 axios 來呼叫 API，故需要先安裝 axios

1. 在 專案資料夾根目錄，開啟 CMD 執行 `npm install axios`

```
PS C:\Users\ray03\Desktop\Vue 3 演練\vue-project> npm install axios

added 9 packages, and audited 134 packages in 984ms

13 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

此處 我們嘗試 透過 共用邏輯 處理 api，於 頁面載入時，透過 API 取得頁面資訊。

1. 建立 共用邏輯，並在其中設定
   A. 資料變數
   B. 錯誤訊息變數
   C. API 處理函式
2. import 共用邏輯，並進行變數宣告
3. 於 onMounted 執行 API 處理函式

Demo16.vue | Demo17.vue

src > components > Demo17.vue > {} script setup

```vue
<script setup>
import { onMounted, ref } from 'vue';
import { useApi } from '@/stores/useApi';

const {data, errorMsg, fetchApi} = useApi();

onMounted(()=>{
    fetchApi()
})
</script>
<template>
    <h1 v-if="data.length === 0">Loading.....</h1>
    <h1 v-else-if="errorMsg !== ''">{{ errorMsg }}</h1>
    <pre v-else>{{ data }}</pre>
</template>
```

useApi.ts

src > stores > useApi.ts > ...

```ts
import axios from "axios"
import { ref } from "vue"
export function useApi() {
    const data = ref([])
    const errorMsg = ref("")
    const fetchApi = async () => {
        try {
            const url = "https://64f622702b07270f705e3258.mockapi.io/GetUser"
            const res = await axios.get(url)
            data.value = res.data
        } catch {
            errorMsg.value = "API 發生錯誤!"
        }
    }
    return {
        data, errorMsg, fetchApi
    }
}
```

Loading.....

一開始

```
[
  {
    "UserName": "我是API來的使用者",
    "id": "1"
  },
  {
    "UserName": "UserName 2",
    "id": "2"
  }
]
```

讀取完畢

Pinia 全域資料管理

傳統 要進行 元件之間的資料交換，需要透過 props 和 emit 來逐層進行
但是 當專案架構龐大時，資料還是必須一層一層的進行傳遞，會導致 程式碼變得非常雜亂
為了解決這問題，vue 官方推薦使用 Pinia 來進行資料的管理。

Pinia 官網：https://pinia.vuejs.org/zh/

- 前置作業

1. 透過 `npm create vue@latest` 進行專案創建時，需要安裝 Pinia

```
C:\Users\ray03\Desktop\Vue 3 演練>npm create vue@latest

Vue.js - The Progressive JavaScript Framework

√ 請輸入專案名稱： ... vue-demo
√ 是否使用 TypeScript？ ... 否 / 是
√ 是否啟用 JSX 支援？ ... 否 / 是
√ 是否引入 Vue Router 進行單頁應用程式開發？ ... 否 / 是
√ 是否引入 Pinia 用於狀態管理？ ... 否 / 是
√ 是否引入 Vitest 用於單元測試 ... 否 / 是
√ 是否要引入一款端對端（End to End）測試工具？ » 不需要
√ 是否引入 ESLint 用於程式碼品質檢測？ ... 否 / 是
√ 是否引入 Vue DevTools 7 擴充元件以協助偵錯？（試驗性功能） ... 否 / 是

正在建置專案 C:\Users\ray03\Desktop\Vue 3 演練\vue-demo...

專案建置完成，可執行以下命令：

  cd vue-demo
  npm install
  npm run dev
```

- 撰寫 Pinia 函式

1. 語法

```
export const Pinia函式 = defineStore('PiniaId', ()=>{
    // 變數宣告與初始化
    ....
    // 計算屬性
    ....
    // 應用函式
    ....
    return {
        變數,
        計算屬性,
        應用函式
```

```
    }
})
```

TS useUserName.ts

src > stores > TS useUserName.ts > useUserNameStore

```ts
import { defineStore } from "pinia";
import { computed, ref, type Ref } from "vue";
// 第一個參數: 變數名稱
// 第二個參數: 函式，裡面寫 變數、設定計算屬性、設定變數相關應用函式
export const useUserNameStore = defineStore('userName', ()=>{
    // 變數宣告與初始化
    const userName: Ref<String[]> = ref([])
    // 計算屬性
    const showUserName = computed(()=>{
        const showUser: String[] = []
        userName.value.forEach((name)=>{
            const msg = "我是" + name
            showUser.push(msg)
        })
        return showUser
    })
    // 應用函式
    // 1. 新增名單
    const addUser = (addName: String) => {
        userName.value.push(addName)
    }
    // 2. 刪除名單
    const deleteUser = (deleteName: String) => {
        const data: String[] = []
        userName.value.filter((name) => name !== deleteName)
                .forEach((name)=>{
                    data.push(name)
                })
        userName.value = data
    }

    return {
        userName,
        showUserName,
        addUser,
        deleteUser
    }
})
```

- 元件端使用方式

1. import Pinia 函式
   import { Pinia 函式 } from '@/stores/Pinia 函式.ts'
2. 宣告 Pinia 函式
   const store = Pinia 函式()
3. 使用 Pinia 變數
   const 變數 = store.變數
4. 使用 Pinia 計算屬性
   const 變數 = store.計算屬性
5. 使用 Pinia 應用函式
   store.應用函式(參數)

Demo18.vue

src > components > Demo18.vue > {} script setup

```vue
<script setup>
import { useUserNameStore } from '@/stores/useUserName.ts';
import ChildComp8_1 from './ChildComp8_1.vue';
import ChildComp8_2 from './ChildComp8_2.vue';
import { ref } from 'vue';
// 宣告 Pinia函式
const store = useUserNameStore();
const addName = ref("")
const deleteName = ref("")

</script>
<template>
    <div>
        <input v-model="addName" />
        <!-- click 使用 Pinia應用函式 -->
        <button @click="store.addUser(addName)">增加名單</button>
    </div>
    <div>
        <input v-model="deleteName" />
        <!-- click 使用 Pinia應用函式 -->
        <button @click="store.deleteUser(deleteName)">減少名單</button>
    </div>
    <div>
        <ChildComp8_1 />
        <ChildComp8_2 />
    </div>
</template>
```

ChildComp8_1.vue

src > components > ChildComp8_1.vue > {} script setup

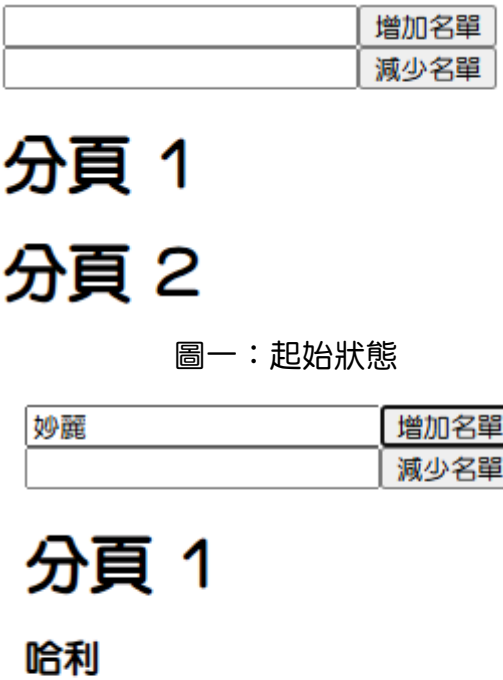
```vue
<script setup>
import { useUserNameStore } from '@/stores/useUserName';

// 宣告 Pinia函式
const store = useUserNameStore();
</script>
<template>
    <h1>分頁 1</h1>
    <!-- 對 Pinia變數 進行 foreach 展示資料 -->
    <h3 v-for="userName in store.userName">{{ userName }}</h3>
</template>
```

ChildComp8_2.vue

src > components > ChildComp8_2.vue > {} script setup

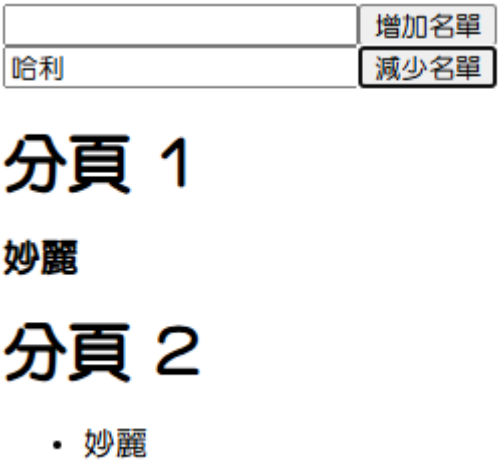
```vue
<script setup>
import { useUserNameStore } from '@/stores/useUserName';

// 宣告 Pinia函式
const store = useUserNameStore();
</script>
<template>
    <h1>分頁 2</h1>
    <ul>
        <!-- 對 Pinia變數 進行 foreach 展示資料 -->
        <li v-for="userName in store.userName">{{ userName }}</li>
    </ul>
</template>
```

從 執行結果可知，父元件 進行 增加/減少名單，子元件的資料會同步更新
各元件之間並沒有使用 `props` 和 `emit` 進行資料交換



圖一：起始狀態

圖二：增加 哈利 與 妙麗



圖三：減少 哈利，只看到妙麗

Vue-Router: vue 的官方路由

- 簡介
  Vue Router 是 Vue 官方的 網頁建構工具，透過 這工具可以實踐 動態切換網頁。
- 官網
  https://router.vuejs.org/zh/

- 說明
  練習的範例檔，是結合 Element Plus 的 layout 布局 和 menu 元件，
  讓整體顯示起來更好看，
  並透過 v-for 和 vue-router 來動態產生練習頁面的連結
  ＊詳細設定方式，詳見 調整專案架構

- router 的路徑設定
  A. router 的路徑 設定於 router 設定檔的 routers 變數中
  B. 如果 該路徑 的跳轉 需要權限控管，可以加上「meta」的屬性
     並透過 生命週期變數 beforeEach 來進行 權限檢核

src > router > JS index.js > ...

```
1  import { createRouter, createWebHistory } from 'vue-router'
2  import HomeView from '../views/HomeView.vue'
3
4  const router = createRouter({
5    history: createWebHistory(import.meta.env.BASE_URL),
6    routes: [
7      {
8        path: '/',
9        name: 'home',
10       component: HomeView      // 靜態跳轉頁面 (網站初始化時就掛載)
11     },
12     { // 父路由
13       path: '/about',        // 路徑 (父路由 開頭要有【/】)
14       name: 'about',         // 名稱
15       meta: {requiresAuto: true},   // 是否進行驗證: 【requiresAuto: true】代表要驗證
16       component: () => import('../views/AboutView.vue'),  // 動態載入跳轉頁面 (有用到再掛載)
17       children: [     // 子路由
18         {
19           paht: "aaa",     // 路徑 (子路由 不需要【/】)
20           meta: {requiresAuto: true},
21           component: () => import('../views/AboutView.vue')
22         }
23       ]
24     }
25   ]
26 })
```

圖一：router 文件，各屬性代表的意義

```
27  /** routre 的生命週期函數
28   *  參數 1: 跳轉過去的 router
29   *  參數 2: 目前所在的 router
30   */
31  router.beforeEach((to, from) => {
32    console.log("beforeEach: to",to.path);      // 顯示 前往的路徑
33    console.log("beforeEach: from",from.path);  // 顯示 目前路徑
34    if (to.meta.requiresAuto) {       // 檢查 前往的路徑是否需要 權限認證
35      console.log("需要進行授權驗證");
36      // return from.path                 // 無權限者 頁面不進行跳轉
37    }
38  })
39
40  export default router
```

圖二：可以透過 生命週期函數 beforeEach 來進行 權限控管

- 取得 router 設定檔 的 routes 資料
  A. 載入 router 設定檔
  B. routes 資料 保存在 router 設定檔.options.routes 中

```
import router from './router'
const routesList = router.options.routes  // 這裡可獲得 router 的資料
```

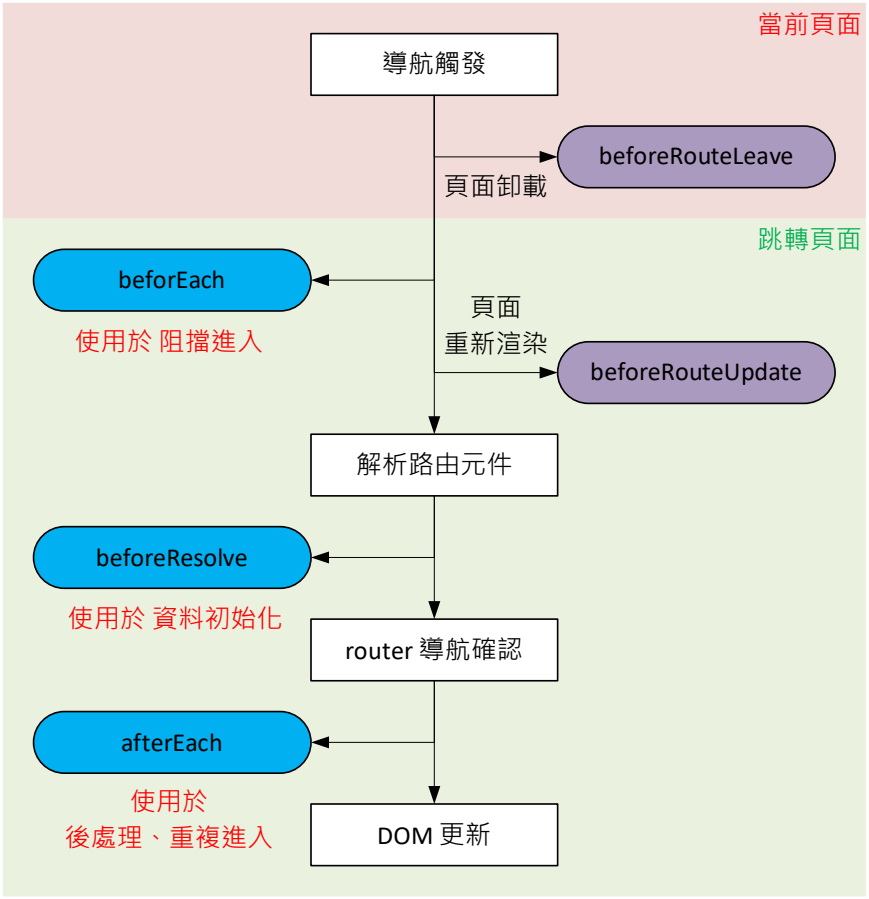
- 重要的組件
  A. RouterLink: 頁面切換按鈕
     >> <RouterLink to="路徑">按鈕的名稱</RouterLink>
     1. 根據 路徑 跳轉網頁 URL
     2. 若 URL 於 router 有找到相關設定，就會於 RoutterView 顯示該網頁
        如果沒有找到，就顯示空白頁
  B. RouterView: 畫面顯示元件
     負責將 RouterLink 找到的 router 設定網頁，顯示於畫面上
     >> <RouterView/>

```vue
<script setup>
import { RouterLink, RouterView } from 'vue-router'
import HelloWorld from './components/HelloWorld.vue'
</script>

<template>
  <header>
    <img alt="Vue logo" class="logo" src="@/assets/logo.svg" width="125" height="125" />

    <div class="wrapper">
      <HelloWorld msg="You did it!" />

      <nav>
        <RouterLink to="/">Home</RouterLink>
        <RouterLink to="/about">About</RouterLink>
      </nav>
    </div>
  </header>

  <RouterView />
</template>
```

圖一：官方範例，重點是 RouterLink 和 RouterView 的使用方式

- router 的 生命週期函式

A. 生命週期流程



B. 生命週期函數

```
router.beforEach((to, from)=>{
   // to   代表 前往的 router
   // from 代表 目前的 router
   // 網頁載入前你要做的事情
})
```

src > config > JS useCreateRouter.js > useCreateRouter

```js
import { createRouter, createWebHistory } from 'vue-router'
import { routerData } from "./router.js";

export const useCreateRouter = () => {

  // 初始化文件
  const router = createRouter({
    history: createWebHistory(import.meta.env.BASE_URL),
    routes: routerData
  })

  router.beforeEach(()=>{                    全局
    console.log("router => beforEach")
  })

  return router
}
```

src > components > Demo1.vue > ...

```vue
<script setup>
import { reactive, ref } from 'vue'
import { onBeforeRouteLeave, onBeforeRouteUpdate } from 'vue-router';
// 宣告
const counter = reactive({
  count: 0
})
const message = ref('Hello World!')
// 改變值
counter.count = 1
message.value = "Change Message!!"
// 取得值
console.log("counter=>", counter.count);
console.log("message=>", message.value);

onBeforeRouteLeave((to, from) => {                元件
  console.log("router => onBeforeRouteLeave")
})
onBeforeRouteUpdate((to, from) => {
  console.log("router => onBeforeRouteUpdate")
})
</script>

<template>
  <h1>{{ message }}</h1>
  <p>Count is: {{ counter.count }}</p>
</template>
```