

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.....	2
透過官方工具建立專案.....	3
調整專案架構.....	4
資料架構.....	9
定義資料 : ref, reactive 和 computed	10
屬性綁定 v-bind:	13
事件監聽 v-on:.....	14
嵌套元件.....	15
元件資料傳遞(父傳子) props	16
元件資料傳遞(子傳父) emit	17
條件渲染 v-if 和 v-show.....	18
列表渲染 v-for.....	20
動態切換元件 component.....	21
監聽器 watch.....	22
資料雙向綁定 v-model	24
生命週期函數.....	26
插槽 slot.....	29
共用邏輯的撰寫.....	31
透過 共用邏輯 處理 API	32
Pinia 全域資料管理.....	34
Vue-Router: vue 的官方路由.....	40

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 > ...  
1  import { fileURLToPath, URL } from 'node:url'  
2  
3  import { defineConfig } from 'vite'  
4  import vue from '@vitejs/plugin-vue'  
5  // Element Plus  
6  import AutoImport from 'unplugin-auto-import/vite'  
7  import Components from 'unplugin-vue-components/vite'  
8  import { ElementPlusResolver } from 'unplugin-vue-components/resolvers'  
9  
10 // https://vite.dev/config/  
11 export default defineConfig({  
12   plugins: [  
13     vue(),  
14     // Element Plus  
15     AutoImport({  
16       resolvers: [ElementPlusResolver()],  
17     }),  
18     Components({  
19       resolvers: [ElementPlusResolver()],  
20     }),  
21   ],  
22   resolve: {  
23     alias: {  
24       '@': fileURLToPath(new URL('./src', import.meta.url))  
25     }  
26   }  
27 })
```

- 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>
14
```

● 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 > ...
1  <script setup>
2  import { RouterView } from 'vue-router'
3  // vue-router
4  import router from './router'
5  const routesList = router.options.routes
6  </script>
7
8  <template>
9    <div class="common-layout">
10      <el-container>
11          <!-- Hader 欄位 -->
12          <el-header>
13              <h1>Vue 學習筆記</h1>
14          </el-header>
15
16          <el-container>
17              <!-- 左側邊欄 -->
18              <el-aside>
19                  <!-- el-menu: 設定 router 代表要開啟 vue-router 功能 -->
20                  <el-menu router>
21                      <!-- el-sub-menu: 1. 透過 v-for 來掃描 router 的全部設定資料 -->
22                      <!-- 2. :index 是 menu 的 key，要設定不同值，才不會同時全部打開/關閉 -->
23                      <el-sub-menu
24                          v-for="router in routesList"
25                          :index="router.path"
26                      >
27                          <!-- 這裡的 template 代表 第一層的 中文名稱 -->
28                          <template #title>{{ router.name }}</template>
29                          <!-- 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 -->
30                          <!-- 2. 這裡的 :index 就是 實際上 Link 要使用的路徑【父路徑/子路徑】 -->
31                          <!-- 3. 最後在中間 嵌入 子路徑的名稱 -->
32                          <el-menu-item
33                              v-for="children in router.children"
34                              :index="router.path + '/' + children.path"
35                          >
36                              {{ children.name }}
37                          </el-menu-item>
38                      </el-sub-menu>
39                  </el-menu>
40              </el-aside>
41
42                  <!-- 右內容欄: 直接使用 RouterView 來顯示頁面就好了 -->
43                  <el-main>
44                      <RouterView />
45                  </el-main>
46              </el-container>
47
48          </el-container>
49      </div>
50  </template>
```

```

vue-demo > src > router > index.js > router > routes
1 import { createRouter, createWebHistory } from 'vue-router'
2
3 const router = createRouter({
4   history: createWebHistory(import.meta.env.BASE_URL),
5   routes: [
6     {
7       path: "/Ch1", // 父網址：開頭需要有【/】
8       name: "定義資料", // 名稱
9       children: [ // 子層級資料的擺放處
10         {
11           path: "Demo1", // 子網址：不需要【/】
12           name: "定義資料(1)", // 名稱
13           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
14           component: () => import('../components/Demo1.vue') // 頁面檔案位置
15         },
16         {
17           path: "Demo2",
18           name: "定義資料(2)",
19           meta: {requiresAuto: true},
20           component: () => import('../components/Demo2.vue')
21         },
22         {
23           path: "Demo3",
24           name: "定義資料(3)",
25           meta: {requiresAuto: true},
26           component: () => import('../components/Demo3.vue')
27         },
28       ],
29     },
30     {
31       path: "/Ch2",
32       name: "屬性綁定",
33       children: [
34         {
35           path: "Demo4",
36           name: "屬性綁定",
37           meta: {requiresAuto: true},
38           component: () => import('../components/Demo4.vue')
39         },
40       ],
41     },
42   ],
43 })

```

Vue 學習筆記

定義資料

- [定義資料\(1\)](#)
- [定義資料\(2\)](#)
- [定義資料\(3\)](#)

屬性綁定

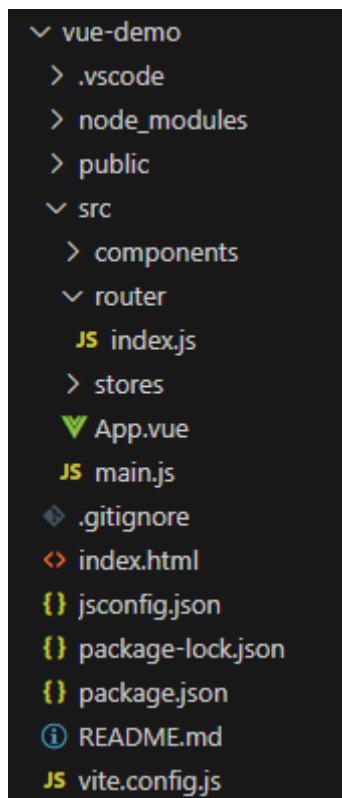
- [屬性綁定](#)

Change Message!!

Count is: 1

- 刪除預設範例程式

- A. scr/assets 資料夾全部刪除
- B. scr/router 資料夾全部刪除
- C. scr/views 資料夾全部刪除
- D. scr/components 清空內部檔案
- E. scr/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

<pre>└─ vue-demo ├─ .vscode ├─ node_modules └─ public └─ index.html └─ src ├─ components ├─ router │ └─ index.js └─ stores └─ App.vue └─ main.js └─ .gitignore └─ index.html └─ jsconfig.json └─ package-lock.json └─ package.json └─ README.md └─ vite.config.js</pre>	<p>頁面啟動： index.html -> main.js -> App.vue (使用 vue-router 串連各頁面)</p> <p>【*.vue】為 vue 的應用程式頁面檔案</p> <p>「components」資料夾 為 擺放子元件的地方</p> <p>「stores」資料夾 為 擺放 工具程式的地方</p> <p>「config」資料夾 為 擺放設定檔的地方</p> <p>1. router/index.js 為 vue-router 的設定檔</p>
---	--

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 --&gt;
    &lt;!-- 如果要使用 script 裡面的變數 --&gt;
    &lt;!-- 要使用 雙大括號 包起來，如：「{{ 變數 }}」 --&gt;
&lt;/template&gt;

&lt;style scoped&gt;
    /* 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 */
&lt;/style&gt;</pre>
```

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。

```

<script setup>
import { reactive, ref } from 'vue'
// 声明
const counter = reactive({
  count: 0
})
const message = ref('Hello World!')
// 改变值
counter.count = 1
message.value = "Change Message!!"
// 取得值
console.log("counter=>", counter.count);
console.log("message=>", message.value);

</script>

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

```

Change Message!!

Count is: 1

```

counter=> 1
message=> Change Message!!
>

```

● 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. 告白

```

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 }}`

```

<script setup>
import { computed, reactive, ref } from 'vue';
// 響應式狀態
const user = reactive({
  name: '王大明',
  age: 10
})
// 告白
const isAdult = computed(()=>{
  const adultDesc = user.age >= 18 ? "已成年" : "未成年"
  return `我是${user.name}，今年${user.age}歲，${adultDesc}。`
})
// Getting
console.log("computed=>", isAdult.value);
</script>

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

```

我是王大明，今年10歲，未成年。

```

computed=> 我是王大明，今年10歲，未成年。
>

```

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

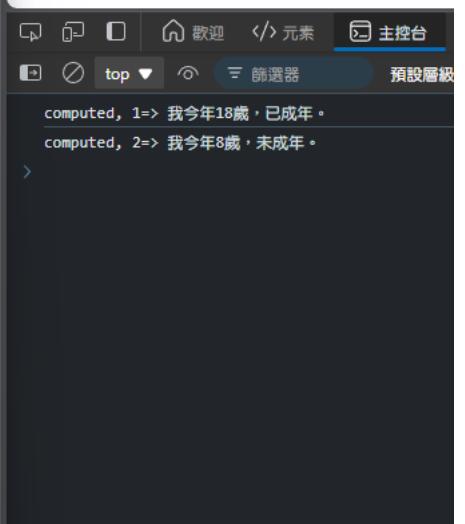
D. template 使用

{{ 變數 }}

```
<script setup>
  import { computed, reactive, ref } from 'vue';
  // 響應式狀態
  const age = ref(18)
  // 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>
```

我今年8歲，未成年。



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>
```



Make me red 1

Make me red 2

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

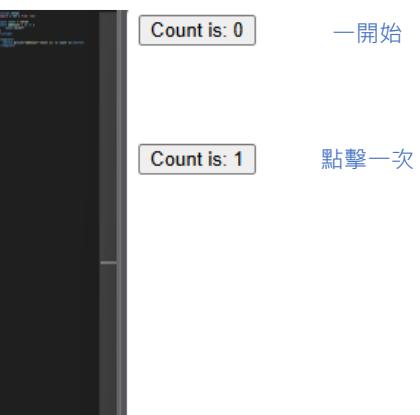
1. 一般寫法

```
<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「@」



The screenshot shows a Vue.js application interface. On the left, the code editor displays the following code:

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

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

</script>

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

On the right, there is a button labeled "Count is: 0". Below it, a blue link says "一開始". To the right of the button, the text "點擊一次" is displayed.

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>
```

The screenshot shows a code editor with two files open:

- Demo6.vue**:
A Child Component!
src > components > Demo6.vue > {} template
1 <script setup>
2 import ChildComp from './ChildComp.vue'
3 </script>
4
5 <template>
6 | <ChildComp />
7 </template>
- ChildComp.vue**:
A Child Component!
src > components > ChildComp.vue > ...
1 <template>
2 | <h2>A Child Component!</h2>
3 </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

The screenshot shows a Vue development environment with two files open:

- Demo7.vue**:
A script setup block imports 'ref' from 'vue' and 'ChildComp' from './ChildComp2.vue'. It defines a 'greeting' ref to 'Hello from parent' and uses a template with two ChildComp components. The second ChildComp component has a msg prop set to 'greeting'.
- ChildComp2.vue**:
A script setup block uses 'defineProps' to create a prop named 'msg'. It has a type of 'String' and a default value of 'No props passed yet'. It logs 'props =>' and 'props.msg' to the console and displays the msg value in an h2 element.

On the right, the browser preview shows the output: 'No props passed yet' for the first ChildComp and 'Hello from parent' for the second ChildComp.

```
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>

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>
```

No props passed yet
Hello from parent

props => No props passed yet
props => Hello from parent

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 ]
    1  <script setup>
    2  import { ref } from 'vue'
    3  import ChildComp3 from './ChildComp3.vue'
    4
    5  const number = ref(0)
    6  const callBack = (data) => {
    7    |  number.value = data
    8  }
    9
    10 </script>
    11
    12 <template>
    13  |  <p>{{ number }}</p>
    14  |  <ChildComp3 @addNumber="callBack"/>
    15 </template>

    ChildComp3.vue
    src > components > ChildComp3.vue > {} script setup > [ add ]
    1  <script setup>
    2  import { ref } from 'vue'
    3  const amt = ref(0)
    4  const emit = defineEmits(['addNumber'])
    5  const add = (a, b) => {
    6    |  amt.value = a + b
    7    |  emit("addNumber", amt.value)
    8  }
    9
    10 </script>
    11
    12 <template>
    13  |  <button @click="add(amt,1)">按我 + 1</button>
    14 </template>

```

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" />

```
src > components > Demo10.vue > {} template
1  <script setup>
2  import { ref } from 'vue'
3  import ChildComp5_1 from './ChildComp5_1.vue'
4  import ChildComp5_2 from './ChildComp5_2.vue'
5  import ChildComp5_3 from './ChildComp5_3.vue'
6  const isTeacher = ref(false)
7  const isStudent = ref(false)
8  const changeRenk = () => {
9    // 工友 -> 老師 -> 學生
10   if (isTeacher.value) {
11     isTeacher.value = false
12     isStudent.value = true
13   } else if (isStudent.value) {
14     isStudent.value = false
15   } else {
16     isTeacher.value = true
17   }
18 }
19 </script>
20 <template>
21   <button @click="changeRenk" >身分切換</button>
22   <ChildComp5_1 v-if="isTeacher" />
23   <ChildComp5_2 v-else-if="isStudent" />
24   <ChildComp5_3 v-else />
25   <h1 v-show="isStudent" >今年即將畢業</h1>
26 </template>
```

```
src > components > ChildComp5_1.vue > {} template
1  <template>
2  |   <h1>我是老師</h1>
3  </template>
```

```
src > components > ChildComp5_2.vue > {} template
1  <template>
2  |   <h1>我是學生</h1>
3  </template>
```

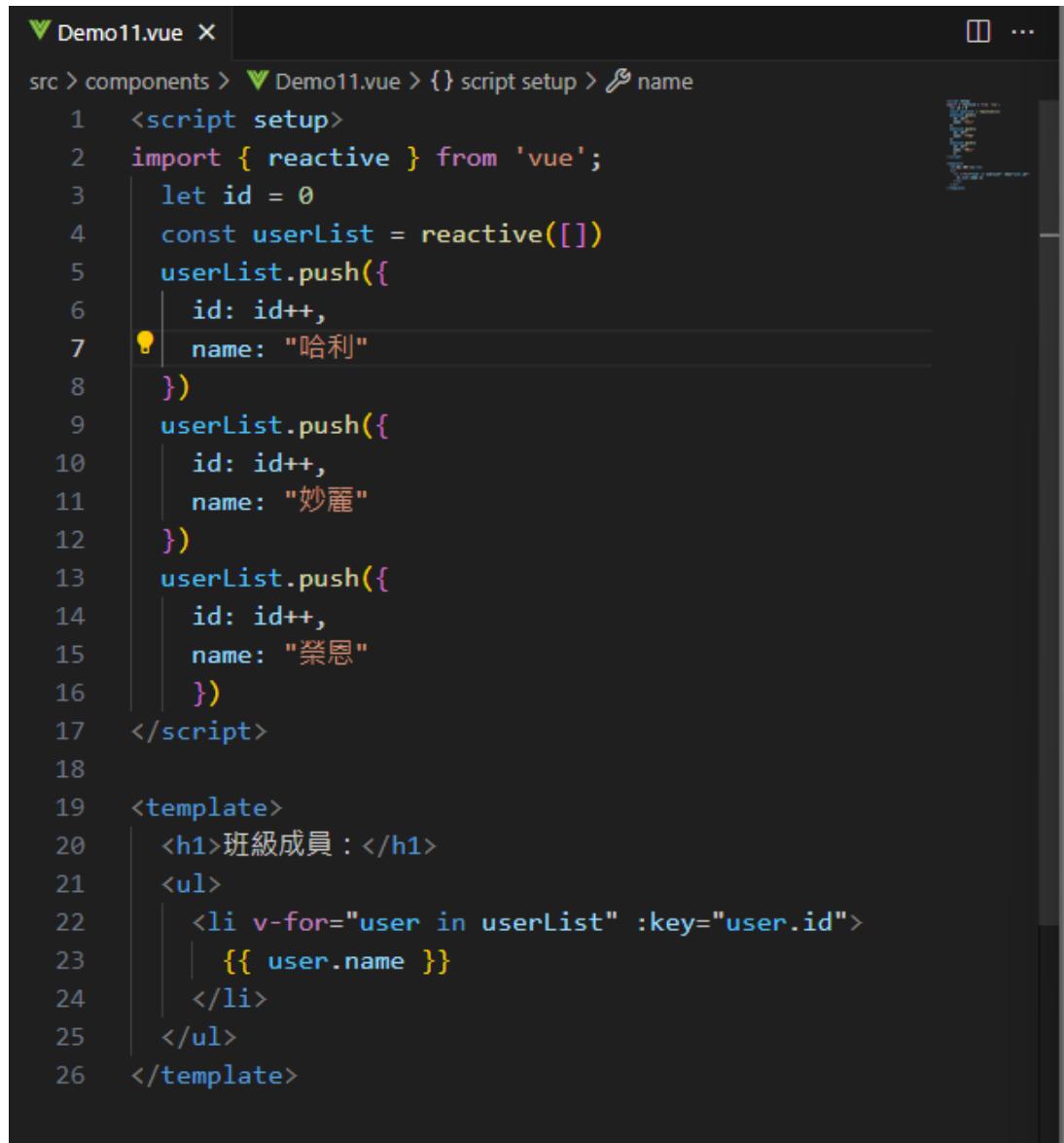
```
src > components > ChildComp5_3.vue > {} template
1  <template>
2  |   <h1>我是工友</h1>
3  </template>
```

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" :key="變數.key" > {{ 變數.欄位 }} </元素>



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

```
1 <script setup>
2 import { reactive } from 'vue';
3 let id = 0
4 const userList = reactive([])
5 userList.push({
6   id: id++,
7   name: "哈利"
8 })
9 userList.push({
10   id: id++,
11   name: "妙麗"
12 })
13 userList.push({
14   id: id++,
15   name: "榮恩"
16 })
17 </script>
18
19 <template>
20   <h1>班級成員：</h1>
21   <ul>
22     <li v-for="user in userList" :key="user.id">
23       {{ user.name }}
24     </li>
25   </ul>
26 </template>
```

班級成員：

- 哈利
- 妙麗
- 榮恩

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;  
        default:  
            return 預設頁面;  
    }  
})
```

3. template 設定頁面

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

The screenshot shows a code editor with three files:

- Demo9.vue** (Top Left):


```

1 <script setup>
2 import { ref, computed } from 'vue'
3 import ChildComp4_1 from './ChildComp4_1.vue'
4 import ChildComp4_2 from './ChildComp4_2.vue'
5 import ChildComp4_3 from './ChildComp4_3.vue'
6 const type = ref(0)
7 const changeType = (data) => {
8   type.value = data
9 }
10 const config = computed(()=>{
11   switch (type.value) [
12     case 1:
13       return ChildComp4_1
14     case 2:
15       return ChildComp4_2
16     case 3:
17       return ChildComp4_3
18     default:
19       return ChildComp4_3
20   ]
21 })
      
```
- ChildComp4_1.vue** (Bottom Left):


```

1 <template>
2   <h2>開會中</h2>
3 </template>
4
5
6
      
```
- ChildComp4_2.vue** (Middle):


```

1 <template>
2   <h2>暫時離開</h2>
3 </template>
4
5
6
      
```
- ChildComp4_3.vue** (Right):


```

1 <template>
2   <h2>我有空</h2>
3 </template>
4
5
6
      
```

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
})
      
```

The screenshot shows a Vue.js component named Demo13.vue. The code uses a simple setup script to manage a state variable 'number' and a template with a button to increment it. The browser's developer tools show the state 'number => 1' and a log entry 'number => 1' from the console.

```
src > components > Demo13.vue > {} template
1  <script setup>
2  import { ref,watch } from 'vue';
3
4  const number = ref(0);
5  const add = () => {
6    number.value++
7  }
8
9  watch(number, ()=>{
10    console.log("number =>",number.value);
11  })
12 </script>
13
14 <template>
15   <h2>{{ number }}</h2>
16   <button @click="add">按我 + 1</button>
17 </template>
```

2. 進階用法

watch(監聽變數, (新數值, 舊數值)=>{

要進行的邏輯處理

)

The screenshot shows a Vue.js component named Demo19.vue. It demonstrates the use of the watch() method with two arguments to log both the new and old values of the 'number' state. The browser's developer tools show the logs 'newValue => 1' and 'oldValue => 0'.

```
src > components > Demo19.vue > {} script setup > watch() callback
1  <script setup>
2  import { ref,watch } from 'vue';
3
4  const number = ref(0);
5  const add = () => {
6    number.value++
7  }
8
9  watch(number, (newValue, oldValue)=>{
10    console.log("newValue =>",newValue);
11    console.log("oldValue =>",oldValue);
12  })
13 </script>
14
15 <template>
16   <h2>{{ number }}</h2>
17   <button @click="add">按我 + 1</button>
18 </template>
```

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

可以透過 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

>> 繩上 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. select

- 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

```
src > components > Demo12.vue > ...
1  <script setup>
2  import { ref,watch } from 'vue';
3  import ChildComp6 from './ChildComp6.vue';
4
5  const userName = ref();
6  const userNo = ref();
7  const userAge = ref();
8  const userCity = ref();
9  const userAddress = ref();
10 watch(userAge, (age)=>{
11   console.log("userAge =>",userAge.value);
12 })
13 </script>
14
15 <template>
16   <div>
17     <h2>無修飾符</h2>
18     <input type="text" v-model="userName" placeholder="輸入姓名"/>
19   </div>
20   <div>
21     <h2>lazy</h2>
22     <input type="text" v-model.lazy="userNo" placeholder="輸入學號"/>
23   </div>
24   <div>
25     <h2>number</h2>
26     <input type="text" v-model.number="userAge" placeholder="輸入年齡"/>
27   </div>
28   <div>
29     <h2>trim</h2>
30     <input type="text" v-model.trim="userCity" placeholder="輸入所在縣市"/>
31   </div>
32   <div>
33     <h2>自定義組件</h2>
34     <ChildComp6 v-model="userAddress" />
35   </div>
36
37   <h2>我是{{userName}}，學號{{userNo}}，今年{{userAge}}歲，我住在{{userCity}}。</h2>
38
39   <h2>地址：{{userAddress}}</h2>
40 </template>
```

ChildComp6.vue

```
src > components > ChildComp6.vue > {} template
1  <script setup>
2  const props = defineProps(["modelValue"])
3  const emit = defineEmits(["update:modelValue"])
4  </script>
5
6  <template>
7    <input
8      type="text"
9      placeholder=""
10     :value="props.modelValue"
11     @input="emit('update:modelValue', $event.target.value)"
12   />
13 </template>
```

無修飾符

王大明

lazy

A001

number

25

trim

台北

自定義組件

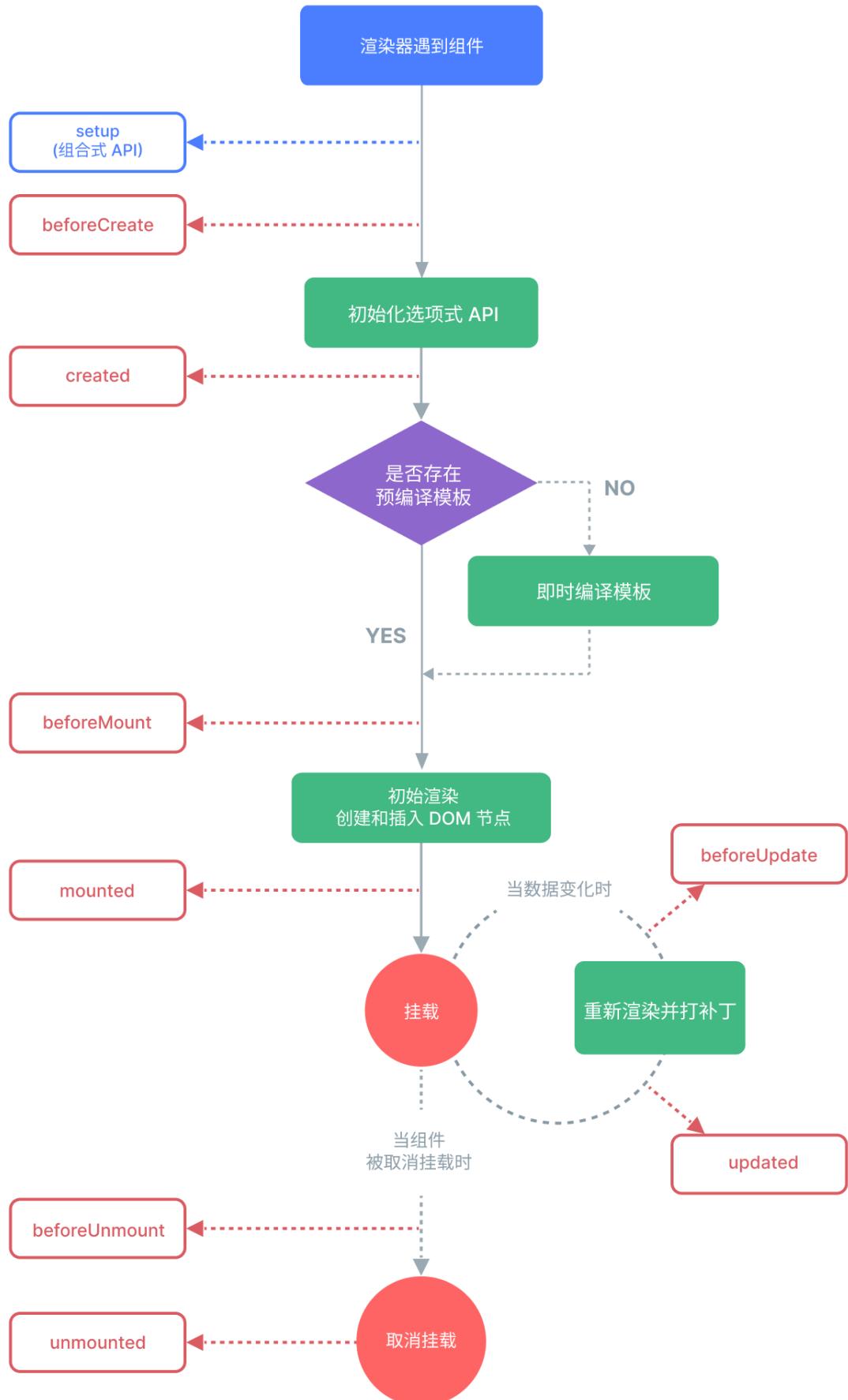
台北市石潭路58號1樓

我是王大明，學號A001，今年25歲，我住在台北。
地址：台北市石潭路58號1樓



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
})
```

The screenshot shows a Vue.js application running in a browser. On the left, the code for `Demo14.vue` is displayed:

```
<script setup>
import { onUpdated, ref, watch, onMounted } from 'vue';
const number = ref(0);
const add = () => {
    number.value++;
}
watch(number, ()=>{
    console.log("number =>", number.value);
})
// 初次渲染完畢後觸發
onMounted(()=>{
    console.log("onMounted");
})
// 重新渲染完畢後觸發
onUpdated(()=>{
    console.log("onUpdated");
})
</script>
<template>
    <h2>{{ number }}</h2>
    <button @click="add">按我 + 1</button>
</template>
```

On the right, the browser's developer tools are open, showing the value of `number` at different stages:

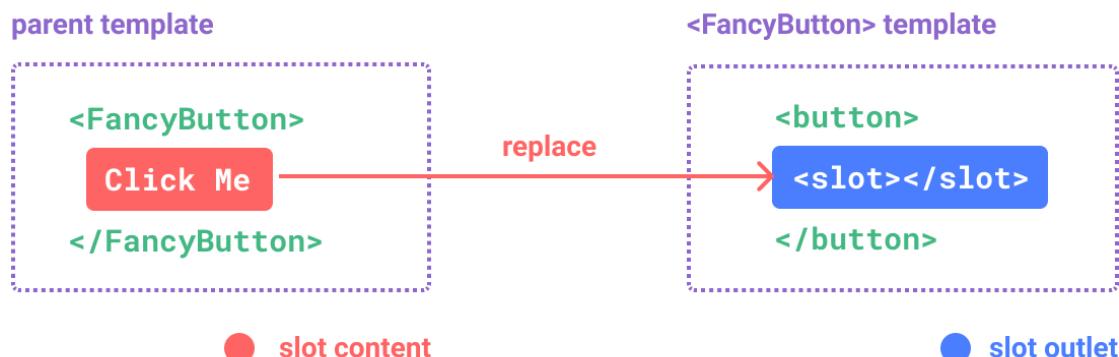
- Initial render: `number => 1`
- After first click (on `onMounted`): `number => 2`
- After second click: `number => 3`
- After third click: `number => 4`

A button labeled "按我 + 1" is visible in the browser window, which triggers the `add` function and the `onUpdated` event.

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>
```

The screenshot shows a Vue development environment with two files and a browser preview.

Demo15.vue:

```
src > components > Demo15.vue > {} script setup
1  <script setup>
2  import { ref } from 'vue';
3  import ChildComp7 from './ChildComp7.vue';
4  const name = ref('')
5  </script>
6
7  <template>
8    <input type="text" v-model="name" placeholder="請輸入姓名"/>
9    <ChildComp7></ChildComp7>
10   <ChildComp7>{{ name }}</ChildComp7>
11 </template>
```

ChildComp7.vue:

```
src > components > ChildComp7.vue > {} script setup
1
2  <template>
3    <div>
4      <slot>目前無資料</slot>
5    </div>
6  </template>
```

Browser Preview:

Initial state (before input):

- Input field placeholder: 請輸入姓名
- Output area: 目前無資料
- Console: 一開始

After input "王大明":

- Input field value: 王大明
- Output area: 目前無資料
- Console: 輸入後

Final state (after slot update):

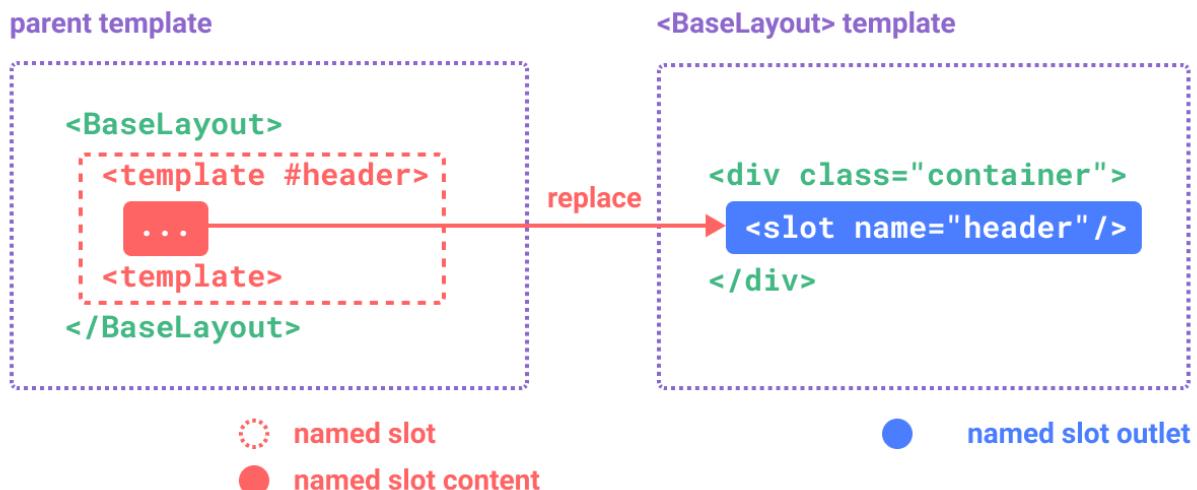
- Input field value: 王大明
- Output area: 王大明
- Console: 王大明

● 進階用法

子頁面 在設定 slot 時，也可以設定 多個 slot，透過 slot name 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

```

src > components > Demo20.vue > {} template > ChildComp9 > template > ...
1  <script setup>
2  import ChildComp9 from './ChildComp9.vue';
3  </script>
4
5  <template>
6    <ChildComp9>
7      <template #header>
8        <p>插入 header 文字</p>
9      </template>
10     <template #main>
11       <p>插入 main 文字</p>
12     </template>
13     <template #footer>
14       <p>插入 footer 文字</p>
15     </template>
16   </ChildComp9>
17 </template>
  
```

ChildComp9.vue

```

src > components > ChildComp9.vue > {} template
1
2  <template>
3    <header>
4      <slot name="header"></slot>
5    </header>
6    <main>
7      <slot name="main"></slot>
8    </main>
9    <footer>
10   <slot name="footer"></slot>
11 </footer>
12 </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
1  <script setup>
2  import { ref } from 'vue';
3  import { useAmt } from '../stores/useAmt.ts'
4  const xAmt = ref(0)
5  const yAmt = ref(0)
6  const {sumAmt, setSumAmt} = useAmt();
7  </script>
8
9  <template>
10    <input type="text" v-model.number="xAmt" @change="setSumAmt(xAmt,yAmt)" />
11    {{ ` + ` }}
12    <input type="text" v-model.number="yAmt" @change="setSumAmt(xAmt,yAmt)"/>
13    {{ ` = ${sumAmt}` }}
14 </template>
15
16
17
TS useAmt.ts ×
src > stores > TS useAmt.ts > ⚡ useAmt > [⊕] setSumAmt
1  import { ref } from 'vue';
2  export function useAmt() {
3    const sumAmt = ref(0)
4
5    const setSumAmt = (x: number, y:number) => {
6      sumAmt.value = x + y
7    }
8
9    return {
10      sumAmt, setSumAmt
11    }
12 }
```

$$0 + 0 = 0$$

$$5 + 9 = 14$$

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

- 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。

- 建立 共用邏輯，並在其中設定
 - 資料變數
 - 錯誤訊息變數
 - API 處理函式
- `import` 共用邏輯，並進行變數宣告
- 於 `onMounted` 執行 API 處理函式

```
src > components > Demo17.vue > {} script setup
1  <script setup>
2  import { onMounted, ref } from 'vue';
3  import { useApi } from '@/stores/useApi';
4
5  const { data, errorMsg, fetchApi } = useApi();
6
7  onMounted(()=>{
8    fetchApi()
9  })
10 </script>
11 <template>
12   <h1 v-if="data.length === 0">Loading.....</h1>
13   <h1 v-else-if="errorMsg !== ''">{{ errorMsg }}</h1>
14   <pre v-else>{{ data }}</pre>
15 </template>

ts useApi.ts > ...
src > stores > ts useApi.ts > ...
1  import axios from "axios"
2  import { ref } from "vue"
3  export function useApi() {
4    const data = ref([])
5    const errorMsg = ref("")
6    const fetchApi = async () => {
7      try {
8        const url = "https://64f622702b07270f705e3258.mockapi.io/
GetUser"
9        const res = await axios.get(url)
10       data.value = res.data
11     } catch {
12       errorMsg.value = "API 發生錯誤!"
13     }
14   }
15   return {
16     data, errorMsg, fetchApi
17   }
18 }
```

Loading.....

一開始

讀取完畢

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


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

- 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

- 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 > [❷] useUserNameStore
1  import { defineStore } from "pinia";
2  import { computed, ref, type Ref } from "vue";
3  // 第一個參數: 變數名稱
4  // 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
5  export const useUserNameStore = defineStore('userName', ()=>{
6      // 變數宣告與初始化
7      const userName: Ref<String[]> = ref([])
8      // 計算屬性
9      const showUserName = computed(()=>{
10         const showUser: String[] = []
11         userName.value.forEach((name)=>{
12             const msg = "我是" + name
13             showUser.push(msg)
14         })
15         return showUser
16     })
17     // 應用函式
18     // 1. 新增名單
19     const addUser = (addName: String) => {
20         userName.value.push(addName)
21     }
22     // 2. 刪除名單
23     const deleteUser = (deleteName: String) => {
24         const data: String[] = []
25         userName.value.filter((name) => name !== deleteName)
26             .forEach((name)=>{
27                 data.push(name)
28             })
29         userName.value = data
30     }
31
32     return {
33         userName,
34         showUserName,
35         addUser,
36         deleteUser
37     }
38 })
```


- 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 計算屬性 = 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  
1  <script setup>  
2  import { useUserNameStore } from '@/stores/useUserName.ts';  
3  import ChildComp8_1 from './ChildComp8_1.vue';  
4  import ChildComp8_2 from './ChildComp8_2.vue';  
5  import { ref } from 'vue';  
6  // 宣告 Pinia函式  
7  const store = useUserNameStore();  
8  const addName = ref("")  
9  const deleteName = ref("")  
10  
11 </script>  
12 <template>  
13   <div>  
14     <input v-model="addName" />  
15     <!-- click 使用 Pinia應用函式 -->  
16     <button @click="store.addUser(addName)">增加名單</button>  
17   </div>  
18   <div>  
19     <input v-model="deleteName" />  
20     <!-- click 使用 Pinia應用函式 -->  
21     <button @click="store.deleteUser(deleteName)">減少名單</button>  
22   </div>  
23   <div>  
24     <ChildComp8_1 />  
25     <ChildComp8_2 />  
26   </div>  
27 </template>
```

▼ ChildComp8_1.vue ×

```
src > components > ▼ ChildComp8_1.vue > {} script setup
1   <script setup>
2     import { useUserNameStore } from '@/stores/useUserName';
3
4     // 宣告 Pinia函式
5     const store = useUserNameStore();
6   </script>
7   <template>
8     <h1>分頁 1</h1>
9     <!-- 對 Pinia變數 進行 foreach 展示資料 -->
10    <h3 v-for="userName in store.userName">{{ userName }}</h3>
11  </template>
```

▼ ChildComp8_2.vue ×

```
src > components > ▼ ChildComp8_2.vue > {} script setup
1   <script setup>
2     import { useUserNameStore } from '@/stores/useUserName';
3
4     // 宣告 Pinia函式
5     const store = useUserNameStore();
6   </script>
7   <template>
8     <h1>分頁 2</h1>
9     <ul>
10      <!-- 對 Pinia變數 進行 foreach 展示資料 -->
11      <li v-for="userName in store.userName">{{ userName }}</li>
12    </ul>
13  </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

	增加名單
	減少名單

分頁 1

分頁 2

圖一：起始狀態

妙麗	增加名單
	減少名單

分頁 1

哈利

妙麗

分頁 2

- 哈利
- 妙麗

圖二：增加 哈利 與 妙麗

	增加名單
哈利	減少名單

分頁 1

妙麗

分頁 2

- 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          path: "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  /** route 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 RouterView 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/>

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

<template>
  <header>
    

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

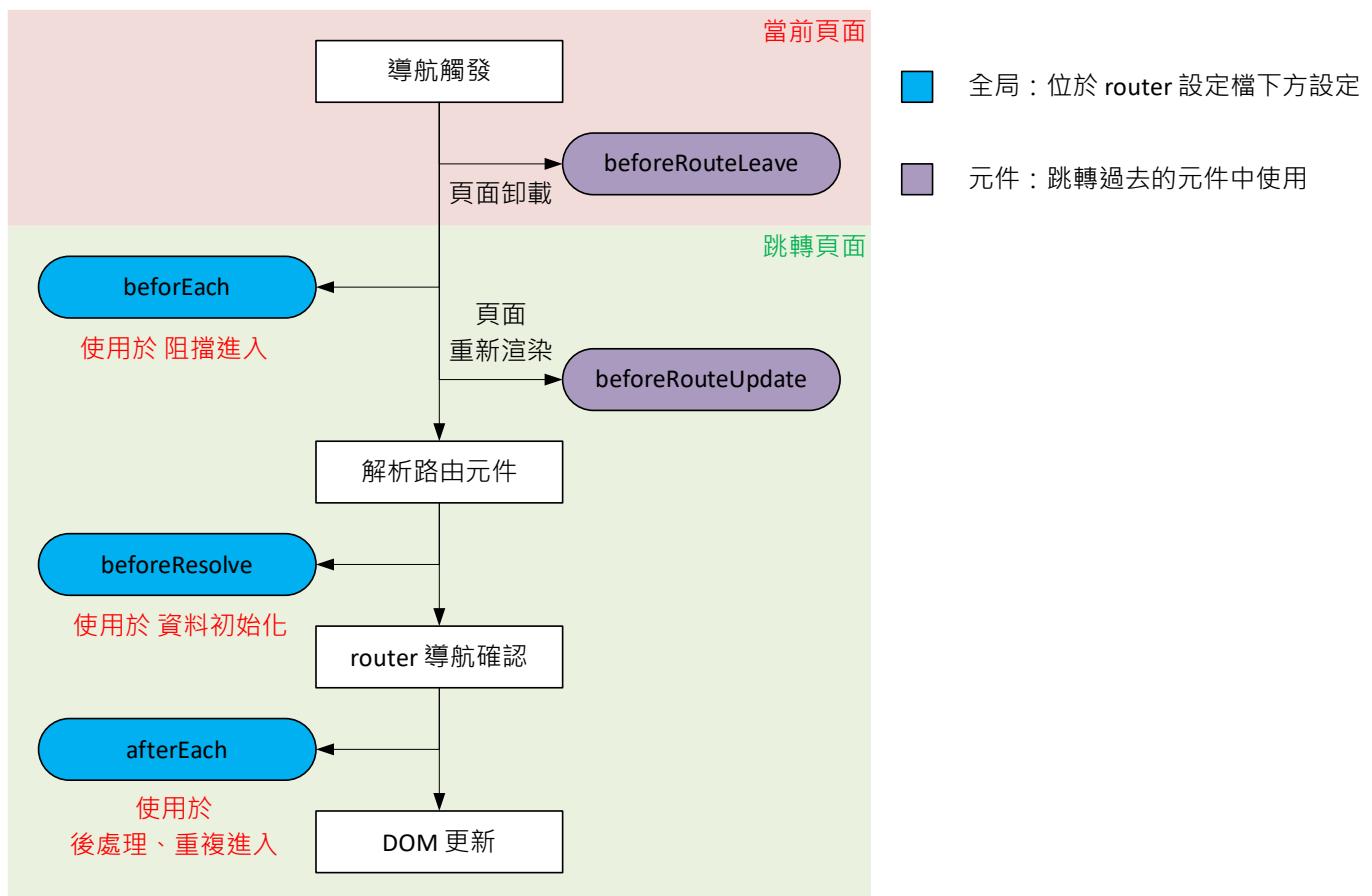
      <nav>
        <RouterLink to="/">Home</RouterLink>
        <RouterLink to="/about">About</RouterLink>
      </nav>
    </div>
  </header>

  <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.beforeEach((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
  1 import { createRouter, createWebHistory } from 'vue-router'
  2 import { routerData } from "./router.js";
  3
  4 export const useCreateRouter = () => {
  5
  6   // 初始化文件
  7   const router = createRouter({
  8     history: createWebHistory(import.meta.env.BASE_URL),
  9     routes: routerData
 10   })
 11
 12   router.beforeEach(()=>{          全局
 13     console.log("router => beforeEach")
 14   })
 15
 16   return router
 17 }
 18
```

```
src > components > Demo1.vue > ...
  1 <script setup>
  2 import { reactive, ref } from 'vue'
  3 import { onBeforeRouteLeave, onBeforeRouteUpdate } from 'vue-router';
  4 // 告示
  5 const counter = reactive({
  6   count: 0
  7 })
  8 const message = ref('Hello World!')
  9 // 改變值
 10 counter.count = 1
 11 message.value = "Change Message!!"
 12 // 取得值
 13 console.log("counter=>", counter.count);
 14 console.log("message=>", message.value);
 15
 16 onBeforeRouteLeave((to, from) => {          元件
 17   console.log("router => onBeforeRouteLeave")
 18 })
 19 onBeforeRouteUpdate((to, from) => {
 20   console.log("router => onBeforeRouteUpdate")
 21 })
 22 </script>
 23
 24 <template>
 25   <h1>{{ message }}</h1>
 26   <p>Count is: {{ counter.count }}</p>
 27 </template>
```

