ch2_Vue_Develop

Vue3開發方式

在 Vue 3 中，**局部開發**（Partial Rendering）和 **單頁應用（SPA, Single Page Application）** 是兩種常見的開發模式，兩者有不同的使用場景和工作原理。

局部開發（Partial Rendering） vs 單頁應用（SPA）

特徵	局部開發 (Partial Rendering)	單頁應用 (SPA)
定義	只在需要更新的地方進行渲染，並不重載整個頁面。	整個應用程式只有一個 HTML 頁面，所有內容動態加載和渲染。
頁面結構	多個獨立的頁面，每次請求只更新一部分頁面內容。	單一頁面，所有內容都會動態更新，無需重新加載頁面。
導航	當用戶導航到不同頁面時，會重新加載頁面。	使用路由控制，無需重新加載頁面，所有的內容會在同一頁面內切換。
頁面重載	需要重載頁面或局部內容。	不需要頁面重載，只有頁面內部的內容更新。
性能	由於每次都會有頁面重載，可能會有較高的延遲。	單頁應用在首次加載後性能較好，但可能會有較長的首次加載時間。
SEO 支援	每個頁面都是一個完整的 HTML 頁面，SEO 較好。	SEO 較差，因為所有的內容是動態載入，且通常需要額外的 SEO 配置。
路由管理	頁面通常是由伺服器控制路由，並將不同頁面映射到不同 URL。	使用客戶端路由（如 Vue Router）來管理頁面的不同狀態。
更新方式	當頁面需要更新時，會重新加載或更新部分 HTML。	只更新頁面的一部分，不會重新加載整個頁面。
適用場景	適用於傳統多頁應用，或部分需要動態更新的頁面。	適用於需要動態內容展示、豐富交互的大型 Web 應用

在 Vue 3 中，**SSR（Server-Side Rendering）** 是一種將 Vue 應用程式的渲染過程從客戶端移至伺服器端的技術。這意味著頁面在伺服器上生成 HTML，並將其發送到瀏覽器，而不是讓瀏覽器自行處理頁面的渲染。SSR 的優勢包括提升 SEO、加速首屏渲染以及改善首次加載性能。

項目	SPA (單頁應用)	SSR (伺服器端渲染)
定義	單頁應用，所有內容和交互都在同一頁面上動態加載。	伺服器生成整頁HTML並發送到客戶端，通常用於提升SEO效果。
表格生成與排序功能	表格數據和排序功能通常由前端處理，點擊表頭觸發排序並更新表格。	初始數據和表格由伺服器渲染，排序後可能需要向伺服器發送請求重新獲取排序數據。
渲染方式	客戶端渲染，初次頁面加載後，所有後續交互和數據更新都在客戶端進行。	伺服器端渲染，頁面初次加載時已經渲染好HTML，減少客戶端的渲染負擔。
SEO	SEO較差，因為大部分內容由JavaScript動態加載，搜索引擎可能無法索引。	SEO優勢，伺服器端渲染的頁面對搜索引擎友好，能夠更快被索引和抓取。
負擔	前端負擔較大，所有渲染和排序處理都在客戶端，數據量大時可能影響性能。	伺服器負擔較大，每次用戶交互可能需要向伺服器發送請求來重新渲染數據。
開發難度	前端邏輯較為複雜，開發者需要處理動態數據、異步請求和UI更新。	開發者需要處理伺服器端和前端的協同，並管理數據的同步，開發過程較為複雜。

總結：

- **SPA**：適合需要流暢交互和動態更新的應用，但對SEO支持較差，需要額外的SEO優化。
- **SSR**：適合需要良好SEO支持和初次加載較快的應用，但可能會增加伺服器的負擔，且開發難度較高。

Option_API & Composition_API

在 Vue.js 中，**Options API** 和 **Composition API** 是兩種不同的開發方式，用來編寫 Vue 組件。它們的區別主要體現在代碼結構、狀態管理、邏輯重用等方面。

Options API	Composition API
Options API 是 Vue.js 2.x 中的主要開發方式，也是 Vue.js 2 和 Vue.js 3 中的預設開發方式。在這種方式下，組件的選項（例如 data, methods, computed, watch）是根據選項（options）來組織和定義的。	Composition API 是 Vue 3 引入的新的開發方式，它使得組件內部的邏輯可以更靈活地組織。與 Options API 不同，Composition API 是基於 setup() 函數的。這種方式通過函數來封裝邏輯，使代碼更加模塊化、可重用。
1.組件的邏輯被分散在不同的選項中（data, methods, computed, watch 等）。 2.沒有明確的功能組織方式，容易造成大型組件中的邏輯難以維護。 3.更容易上手，適合 Vue 的初學者。	1.代碼更清晰，邏輯組織更加靈活，特別是當一個組件擁有多種功能或狀態時。 2.利用 reactive, ref, computed 等 API 管理狀態。 3.增強了邏輯重用，適合處理複雜的組件或代碼組織。 4.提供了更好的 TypeScript 支援。

const App = {
data(){},
methods:{},
};
Vue.createApp(App).mount('#app')
	const { ref } = Vue
	const App = {
	setup(){
	const message = ref('Hello')
	return {message,}
	},

	}, Vue.createApp(App).mount('#app')

特徵	Options API	Composition API
開發方式	使用選項（`data`, `methods`, `computed` 等）來定義組件邏輯	使用 `setup()` 函數來組織組件邏輯
代碼結構	邏輯基於選項分佈，較為分散	邏輯集中在 `setup()` 中，可以更靈活地組織和重用代碼
邏輯重用	重用較困難，可能需要 mixin 或者繼承	更容易重用邏輯，使用 `composables`（組件邏輯模組）
易於學習	相對簡單，適合初學者	需要理解更多的 API，學習曲線稍微陡峭
TypeScript 支援	支援有限，類型推斷較弱	更好支援 TypeScript，提供更強的類型推斷

- **Options API**：適用於簡單、直觀的應用，或者對 Vue.js 仍然不熟悉的開發者，快速構建小型應用或組件。
- **Composition API**：適用於複雜的應用，或需要高度重用邏輯的場景，尤其是當組件的邏輯較為複雜時，Composition API 提供更好的組織方式，並且它更適合和 TypeScript 配合使用。

Vue instance

Options API

```
data(){        //變數放data function
      return {message: 'Hello', }
},
methods:{ // 函數大部分放在methods裡面。}
computed:{ // 函數小部分放在computed物件裡面，放在裡面的函數不傳參數，一定要有傳回值。}
watch:{ // 偵測到 data 和computed的變化}
```

Composition API

```
setup(){
     //變數
     const message = Vue.ref("Hello")
     const loading    =Vue.ref(true)
     let score = 0
     // 函數
      let  active = ()=>{}
      // 計算屬性
      const calculate = Vue.computed(()=>{})
      return {  message, loading, active, calculate,     //message: message}
},
```

- 寫在return 裡面的值，HTML DOM template才會去偵測它。
- ()⇒{}
  - 你寫大括號，就一定要加return 。JS會將其視為code block而不是返回值。
  - 不寫大括號，就不可以加rerturn。
  - 記得不要寫大括號，可能會有東西顯示不出來。

生命週期

在 Vue 中，生命週期（Lifecycle）指的是一個 Vue 實例從創建到銷毀的過程，這個過程中會經歷一系列的鉤子（hook），讓開發者可以在特定的時間點執行一些操作。Vue 的生命週期有幾個重要的階段，這些階段的鉤子函數可以讓你在適當的時候插入代碼。

創建階段（Creation）

- beforeCreate: Vue 實例被創建，但資料、事件等還沒被初始化。
- created: Vue 實例創建完成，資料、事件等已經初始化，但還沒掛載到 DOM 上。

掛載階段（Mounting）

- beforeMount: 在 Vue 實例掛載到 DOM 前呼叫，這時候模板還沒有渲染。
- mounted: Vue 實例掛載到 DOM 上後呼叫，模板渲染完成並插入到 DOM 中。

更新階段（Updating）

- beforeUpdate: 當資料發生變化，並且 DOM 還未重新渲染時呼叫。可以在這裡訪問更新前的資料和 DOM。
- updated: 當資料更新，並且 DOM 完成重新渲染後呼叫。

銷毀階段（Destruction）

- beforeDestroy: Vue 實例即將銷毀前呼叫。可以在這裡進行一些清理操作，例如移除監聽器。
- destroyed: Vue 實例銷毀後呼叫。此時，Vue 實例的監聽器、事件和子組件已經被銷毀。

Vue instance – Life Circle Hooks

Vue 2	觸發時機	Vue3	觸發時機	範例
beforeCreate	在組件實例創建之前調用	x	組件實例創建前，無法訪問 data 和 methods	beforeCreate() { console.log("beforeCreate"); }

created	組件實例創建後調用，數據已設置，但未掛載到 DOM	x	組件實例創建後，data 和 methods 可用，但未渲染	created() { console.log("created"); }
beforeMount	組件掛載前調用，尚未渲染至 DOM	onBeforeMount()	在模板掛載前，尚未渲染到 DOM	onBeforeMount() {console.log("beforeMount"); }
mounted	組件掛載後調用，模板渲染並插入到 DOM 中	onMounted()	當組件的模板渲染並被插入到頁面中	onMounted() { console.log("mounted"); }
beforeUpdate	數據改變且 DOM 尚未更新時觸發	onBeforeUpdate()	數據變動時，尚未更新 DOM	onBeforeUpdate() { console.log("beforeUpdate"); }
updated	數據改變且 DOM 更新後觸發	onUpdated()	數據更新並且 DOM 渲染完成後觸發	onUpdated() { console.log("updated"); }
beforeDestroy	組件銷毀前調用，進行清理	onBeforeUnmount()	組件即將銷毀之前，適合清理資源	onBeforeUnmount() { console.log("beforeDestroy"); }
destroyed	組件銷毀後調用	onUnmounted()	組件銷毀並且從 DOM 中移除後觸發	onUnmounted() { console.log("destroyed"); }

```javascript
import { ref, onMounted, onUpdated, onUnmounted } from 'vue';

export default {
  setup() {
    const count = ref(0);

    // 組件掛載時觸發
    onMounted(() => {
      console.log('組件已經掛載到 DOM 上');
    });

    // 組件更新時觸發
    onUpdated(() => {
      console.log('組件更新了');
    });

    // 組件銷毀時觸發
    onUnmounted(() => {
      console.log('組件已經銷毀');
    });

    return {
      count
    };
  }
};
```

- 真實的標籤放在Template
  - template:`<h1>Hello</h1>`
- 定義組件放在 components
  - components: {}

Vue3 的ref reactive

在 Vue 3 中，ref 和 reactive 是兩個用來創建響應式資料的 API。它們用來處理組件中的資料，使得資料變更後，視圖會自動更新。這兩者雖然有相似之處，但也有一些重要的區別。

1. ref：用於基本資料類型、簡單的陣列或單一值

ref 是 Vue 3 中用來創建基本資料類型（例如：string、number、boolean）或其他單一值的響應式引用。

當你使用 ref 創建一個響應式資料時，這個資料會被包裹成一個物件，這個物件有一個名為 .value 的屬性，你可以通過這個屬性來讀取或修改資料。

用於創建基本資料類型的響應式引用。

需要使用 .value 屬性來讀取或寫入資料

2. reactive：用於複雜的物件或數組

reactive 用來創建物件或數組的響應式資料。它會將整個物件或數組轉換為一個響應式物件，並且不需要使用 .value 屬性來讀取或修改資料。

小結：

- 用於創建物件或數組的響應式資料。
- 不需要使用 .value 屬性來讀取或寫入資料。

特性	`ref`	`reactive`
用途	創建基本資料類型或單一值的響應式資料	創建物件或數組的響應式資料
使用方式	需要使用 `.value` 來讀取或寫入資料	不需要 `.value`，可以直接讀寫屬性
適用範圍	基本資料類型（`string`、`number`、`boolean`）	物件、數組

3. 舉例

是的，**陣列**可以使用 ref，並且這是一種常見的做法。當你希望在 Vue 中使用響應式陣列時，可以使用 ref 來包裝這個陣列。這樣，Vue 的響應式系統會自動追蹤陣列的變更，並在陣列發生改變時更新 DOM

當使用 ref 創建響應式陣列時，必須使用 .value 來讀取或修改陣列的內容。例如，items.value.push() 用來向陣列添加項目。

使用 reactive 與 ref 的區別：

如果你只是需要一個基本的陣列來進行響應式管理，那麼使用 ref 是完全可以的。但在一些情況下，當你處理複雜的對象或陣列時，也可以使用 reactive，它使得數據更加直觀，不需要每次都使用 .value

判斷使用 ref 還是 reactive

- **使用 ref 的情況**：
  - 當數據結構相對簡單，或是需要單一的變數（如數字、字串、陣列或物件）時，使用 ref 是一個簡單且直接的選擇。
  - 當數據是需要直接與 DOM 或其他狀態進行交互的時候，ref 是理想的選擇。
  - 對於 **基本數據類型**（如數字、字串、布林值等），最好使用 ref。
- **使用 reactive 的情況**：
  - 當數據是 **複雜的物件** 或 **嵌套的物件**（例如多層級的物件）時，reactive 更適合。
  - 如果需要處理的數據結構涉及多層嵌套的物件，使用 reactive 可以避免頻繁地使用 .value。

- reactive 會自動代理物件的屬性，使其所有屬性都變成響應式。

Setup()

在 Vue 3 中，setup() 是一個全新的組件選項，屬於 **Composition API** 的一部分。它讓你能夠以更靈活、可組合的方式來撰寫 Vue 組件的邏輯。setup() 函數會在 Vue 組件的 **創建階段**被調用，並且是組件的 **響應式邏輯** 的主要入口點。

1.setup() 的基本概念

- setup() 函數在組件實例創建之前被呼叫，並且只會被呼叫一次。
- 它接收兩個參數：
  1. **props**：組件的 props 屬性，這些屬性是從父組件傳遞過來的。
  2. **context**：一個包含 attrs、slots 和 emit 的物件，通常你不需要直接操作它，除非你需要操作插槽（slots）或發射事件（emit）。
- 在 setup() 中返回的所有資料、方法、計算屬性等，會暴露給模板（template）或其他組件的 setup() 函數，並且自動變為響應式。

2.setup() 的特性與工作原理

1. **組件的邏輯集中管理**：在 setup() 中，你可以集中寫入組件的狀態、方法、計算屬性等邏輯。這比傳統的 data、methods、computed 等選項更具彈性。
2. **響應式資料**：setup() 中返回的變數和方法都是響應式的。像 ref() 和 reactive() 這些函數創建的資料會在視圖中自動更新。
3. **無 this 指向**：setup() 中並不需要使用 this。這樣的設計使得組件邏輯更加直觀，且能夠更輕鬆地拆分、重用。

3.setup() 的優勢

1. **清晰的邏輯結構**：你可以把組件的狀態、計算屬性、方法等邏輯集中在 setup() 中，讓代碼結構更加清晰。
2. **更好的重用性**：可以將 setup() 函數中的邏輯提取為自定義的組件組件或組件函數（例如：useMyLogic()），增強代碼的可重用性。
3. **更靈活的組合方式**：不需要像選項式 API 一樣，必須將所有邏輯集中在特定的區塊（如 data、methods 等），setup() 允許你自由組織和組合邏輯。

4.setup() 與main() 的比較

相似之處：

1. **初始化功能**：就像 Java 中的 main 方法用來初始化應用程式一樣，setup 也用來初始化 Vue 組件的狀態和邏輯。它們都在應用或組件創建時被執行。
2. **設定資料/邏輯**：在 Java 的 main 方法中，你可能會創建和初始化物件、服務等；在 Vue 的 setup 中，你可以初始化響應式資料、方法、計算屬性等。

不同之處：

1. **作用範圍不同**：
   - **Java 的 main**：main 方法是應用程式的入口點，通常只執行一次，並且作用於整個應用程式。
   - **Vue 的 setup**：setup 是在每個組件內執行的，每個 Vue 組件都有自己的 setup。每個組件的 setup 是獨立的，且在組件實例化時執行。
2. **生命週期不同**：
   - **Java 的 main**：main 方法是應用程式啟動時執行的，並且僅執行一次，負責啟動整個應用。
   - **Vue 的 setup**：setup 在每個組件實例化時執行，並且每個組件的 setup 都是獨立的，會隨著組件的生命週期多次執行。
3. **功能不同**：
   - **Java 的 main**：main 方法主要是用來啟動應用程式，通常涉及命令列參數、應用初始化等。
   - **Vue 的 setup**：setup 是組件的核心初始化部分，用來設定組件的響應式資料、計算屬性、方法等，直接關係到組件的 UI 和功能。

小結：

- **Java 的 main 方法**：是程式的入口點，負責啟動應用程式的執行。
- **Vue 的 setup 方法**：是 Vue 3 中每個組件的初始化部分，負責設定組件的狀態、方法和其他邏輯。

雖然兩者都用於初始化功能，但它們的作用範圍和使用場景不同。main 方法是整個應用程式的入口點，而 setup 是每個 Vue 組件的初始化函式。

Watch

watch：監聽資料變化

watch 是 Vue 中用來觀察和響應資料變化的 API。當某個響應式資料發生變化時，watch 可以觸發一個回調函數。這對於一些需要對資料變動作出反應的情況（例如：發送 API 請求、處理副作用等）非常有用。

watch 進階用法：

- **立即執行回調**：可以設置 immediate: true 讓 watch 在一開始就執行一次回調。

- **深度監聽**：可以設置 deep: true 來監聽物件的嵌套屬性變化。

```javascript
import { watch, ref } from 'vue';

export default {
  setup() {
    const count = ref(0);

    watch(count, (newVal, oldVal) => {
      console.log(`count 改變了：從 ${oldVal} 到 ${newVal}`);
    });

    return {
      count
    };
  }
};
```

Fetch & Promise

new Promise(resolve, reject) 是 JavaScript 中創建 Promise 物件的一種語法，這是處理異步操作的核心概念之一。Promise 是一個表示異步操作最終完成（或失敗）及其結果值的物件，並允許你指定操作完成後該做什麼（使用 .then() 或 .catch()）。

new Promise(resolve, reject) 解析：

new Promise() 需要傳遞一個 **執行器函數**（executor function），這個函數接收兩個參數：

- **resolve**：一個函數，當異步操作成功時被調用，並且它的參數將成為 Promise 最終的結果值。
- **reject**：一個函數，當異步操作失敗時被調用，並且它的參數將成為錯誤原因。

resolve 和 reject 的工作：

- **resolve(value)**：當異步操作成功時，調用 resolve()，並且傳遞一個值（value）。這個值會成為 Promise 的最終結果。
- **reject(reason)**：當異步操作失敗時，調用 reject()，並且傳遞錯誤信息或原因（reason）。這個錯誤將成為 Promise 被拒絕的原因。

Promise 的狀態：

Promise 有三種狀態：

1. **Pending（待定）**：初始狀態，操作還在進行中。
2. **Fulfilled（已完成）**：操作成功完成，resolve() 被調用，並返回結果。
3. **Rejected（已拒絕）**：操作失敗，reject() 被調用，並返回錯誤。

```javascript
const myPromise = new Promise((resolve, reject) => {
  let success = true;  // 假設這是異步操作的結果

  // 模擬一個異步操作，比如等待 2 秒
  setTimeout(() => {
    if (success) {
      resolve('操作成功！');  // 成功時調用 resolve
    } else {
      reject('操作失敗');  // 失敗時調用 reject
    }
  }, 2000);
});

// 使用 then() 處理成功的結果
myPromise.then(result => {
  console.log(result);  // 輸出：操作成功！
}).catch(error => {
  console.log(error);  // 若操作失敗，則輸出錯誤信息
});
```

Vue 的非同步比較

非同步有很多種作法，統稱AJAX。擇一撰寫即可。建議使用Axios, Fetch, Async, Await

特性	`XMLHttpRequest` (XHR)	`jQuery.ajax()`	`Fetch API`	`Async/Await` (ES8)
語法	複雜，需要回調函數	簡單，但需要依賴 jQuery	基於 `Promise`，現代簡潔	基於 `Promise`，語法糖，簡潔
回調函數支持	支援回調函數	支援回調函數	支援 `Promise` 回傳結果	支援 `Promise`，改進可讀性
錯誤處理	必須手動處理錯誤 (`onerror`)	支援錯誤處理回調	`catch` 方法處理錯誤	使用 `try/catch`，簡潔直觀
依賴性	無依賴	需要 jQuery 庫	無依賴	無依賴
性能	性能較差，較為繁瑣	性能略低，依賴 jQuery 的處理	高效，現代瀏覽器的原生實現	性能與 `Fetch` 相同
瀏覽器支持	廣泛支持，老舊瀏覽器也支持	廣泛支持，但需要載入 jQuery 庫	現代瀏覽器支持良好，部分舊瀏覽器不支持	同 `Fetch`，現代瀏覽器支持
易用性	難以使用，且代碼可讀性差	使用簡便，但需要 jQuery	比 `XHR` 更簡單，但需要處理 `Promise`	最簡潔，可讀性最好
多重請求處理	較為麻煩	jQuery 提供的多重請求支持	支援多重 `Promise`，需處理串行或並行	同 `Promise`，簡化錯誤處理
優點	廣泛支持，靈活性高	方便，提供更多選項	原生 API，簡潔，支持 `Promise`	簡潔，基於 `Promise`，易於維護

2. jQuery.ajax():jQuery 提供了簡化的 $.ajax() 方法來處理 AJAX 請求，它隱藏了 XMLHttpRequest 的底層實現，並提供了許多方便的選項。

```javascript
$.ajax({
  url: "https://example.com/api",
  type: "GET",
  success: function(response) {
    console.log(response);
  },
  error: function(error) {
    console.log(error);
  }
});
```

3. Fetch API (ES6):Fetch 是一個現代化的 Web API，用來處理 HTTP 請求，基於 Promise 進行非同步操作，替代了 XMLHttpRequest。

```javascript
fetch("https://example.com/api")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

4. Async/Await (ES8):Async/Await 是基於 Promise 的語法糖，讓異步代碼看起來像同步代碼。這是目前最常見和推薦的處理非同步操作的方法。

```javascript
async function fetchData() {
  try {
    const response = await fetch("https://example.com/api");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.log(error);
  }
}
fetchData();
```

Vue 的Async / Await非同步 看起來變成同步的

async 和 await 是 JavaScript 中用來處理異步操作的一種語法糖，它們基於 Promise，使異步代碼更簡潔、易於理解。當你需要串接多個 API 請求時，使用 async/await 可以避免回調地獄（callback hell），並讓代碼看起來像是同步執行。

基本概念：

- **async**：用來標記一個函數為異步函數，異步函數會自動返回一個 Promise。
- **await**：用來等待一個 Promise 的解決（即返回結果或錯誤），並讓代碼繼續執行。await 只能在 async 函數內使用。

使用 async/await 串接 API：

假設你有兩個 API 需要依次調用，其中第二個 API 的請求取決於第一個 API 的回應。

```javascript
// 使用 async/await 串接 API 請求
async function fetchData() {
  try {
    // 第一個 API 請求
    const response1 = await fetch('https://api.example.com/data1');
    if (!response1.ok) {
      throw new Error('第一個 API 請求失敗');
    }
    const data1 = await response1.json();  // 將回應轉換為 JSON
    console.log('第一個 API 資料:', data1);

    // 根據第一個 API 的結果，發送第二個 API 請求
    const response2 = await fetch(`https://api.example.com/data2?id=${data1.id}`);
    if (!response2.ok) {
      throw new Error('第二個 API 請求失敗');
    }
    const data2 = await response2.json();  // 解析第二個 API 回應的 JSON 資料
    console.log('第二個 API 資料:', data2);

  } catch (error) {
    // 如果發生錯誤，捕獲並處理
    console.error('錯誤:', error);
  }
}

fetchData();
```

async：函數 fetchData 被標記為 async，這使得它返回一個 Promise。在這個函數內，你可以使用 await 來等待每個 Promise 的結果。

await：每次發送請求時，我們使用 await 等待 fetch() 返回的 Promise 被解決。這樣會讓代碼執行看起來像是同步的，但實際上它仍然是異步的。

fetch：用來發送 HTTP 請求。await fetch(url) 會等到 API 請求完成，然後返回 Response 物件。

- response1.json() 會把 API 回應的 JSON 內容解析成 JavaScript 對象。
- 根據第一個 API 的結果 (data1)，我們發送第二個 API 請求。
