# <https://shorturl.at/Ya2K3>

# What is IIFE（Immediately Invoked Function Expression）?

IIFE（立即調用函數表達式）是一種 JavaScript 設計模式，它允許您定義一個函數並立即執行它。簡單來說，它是一個匿名函數（anonymous function），在定義後立即被調用。

IIFE 的基本結構：

(function() {

// 函數體

})();

**匿名函數（Anonymous Function）：** function() { ... } 定義了一個沒有名稱的函數。

**圓括號（Parentheses）：** (...) 將匿名函數包起來，使其成為一個函數表達式（function expression）。

**立即調用（Immediately Invoked）：** () 在函數表達式後面立即調用該函數。

**為什麼使用 IIFE？**

* **創建獨立的作用域（Create Independent Scope）：**
  + IIFE 創建了一個新的函數作用域，這意味著在 IIFE 內部聲明的變數不會污染全局作用域（global scope）。
  + 這有助於避免命名衝突，並提高程式碼的模組化和可維護性。
* **資料隱藏（Data Hiding）：**
  + IIFE 可以用於創建私有變數和方法，這些變數和方法只能在 IIFE 內部存取。
  + 這有助於實現封裝（encapsulation），並保護資料的完整性。
* **避免變數污染（Avoid Variable Pollution）：**
  + 在舊版本的javascript，var變數會被提升到function scope的最頂端，在迴圈中，使用IIFE可以避免這種情況。
* **模組化（Modularization）：**

IIFE 可以將程式碼組織成獨立的模組，每個模組都有自己的作用域和功能。

# 

# **What is OOP 物件導向程式設計（Object-Oriented Programming, OOP）?**

物件導向程式設計（OOP）就像在玩積木：

想像一下，你有很多不同形狀和顏色的積木。在 OOP 中，這些積木就像「物件」（Objects）。

* 物件（Objects）：
  + 每個積木（物件）都有自己的「特性」（Properties），像是顏色、形狀、大小。
  + 每個積木（物件）也可以做一些「動作」（Methods），像是堆疊、移動、旋轉。
  + 例如，一個「汽車」物件，它的特性可能有顏色、品牌、型號，它的動作可能有發動、加速、煞車。
* 類別（Classes）：
  + 類別就像是積木的「模具」。它告訴我們某種積木（物件）應該有什麼特性和動作。
  + 例如，「汽車」類別會告訴我們，所有汽車物件都應該有顏色、品牌、型號，以及發動、加速、煞車的動作。
  + 我們可以根據同一個類別，製造出很多不同的物件。
* 封裝（Encapsulation）：
  + 封裝就像是把積木（物件）的內部構造藏起來，只讓我們看到和使用它提供的功能。
  + 例如，我們不需要知道汽車引擎的運作原理，只需要知道怎麼踩油門和煞車。
  + 這樣可以避免我們不小心弄壞積木（物件）的內部構造。
* 繼承（Inheritance）：
  + 繼承就像是從一個積木（類別）複製出另一個積木（類別），並且可以增加或修改一些特性和動作。
  + 例如，我們可以從「汽車」類別繼承出「跑車」類別，「跑車」類別除了有汽車的特性和動作，還可以有更快的速度和更酷的外觀。
  + 這樣可以減少重覆的程式碼，提高程式碼的再用性。
* 多型（Polymorphism）：
  + 多型就像是同一個動作，不同的物件可以有不同的反應。
  + 例如，我們對不同的動物物件發出「叫」的動作，「狗」物件會「汪汪」叫，「貓」物件會「喵喵」叫。
  + 這樣可以讓程式碼更靈活，更容易擴充。

簡單來說：

OOP 是一種程式設計的方法，它把程式碼組織成一個個的「物件」，這些物件有自己的特性和動作，並且可以互相互動。這樣可以讓程式碼更容易理解、修改和維護。

# 什麼是閉包？ (What is a Closure?)

閉包 (Closure) 就像一個函數 (function) 記得它「老家」的詞法作用域 (lexical scope) 的東西。即使它跑到別的地方執行，它還是能記得「老家」的東西。

**簡單來說 (In simple terms):**

一個函數 (function) 裡面又有一個函數 (nested function)，裡面的函數可以拿到外面函數的變數 (variables)，就算外面的函數已經執行完了，裡面的函數還是可以拿到。

**特性 (Characteristics):**

* **記住老家的東西 (Scope Preservation):** 裡面的函數可以拿到外面函數的變數 (outer function's variables)。
* **函數包函數 (Function within a Function):** 通常是一個函數 (function) 裡面又有一個函數 (nested function)。
* **變數留下來 (Variable Retention):** 裡面的函數把外面的變數 (variables)「包」起來 (closes over)，讓它們不會消失。

**簡單的例子：**

**function outerFunction(outerVariable) {**

**function innerFunction(innerVariable) {**

**// innerVariable will be undefined by default if not provided during the call, but outerVariable is accessible from the outer function's scope due to closure.**

**console.log(outerVariable + innerVariable);**

**}**

**return innerFunction;**

**}**

**// When outerFunction is called with "Hello, ", outerVariable is set to "Hello, " in the scope of innerFunction.**

**const myInnerFunction = outerFunction("Hello, ");**

**// Calling myInnerFunction with "World!" will output "Hello, World!", Assigning "World!" as innerVariable value**

**myInnerFunction("World!"); // 輸出 "Hello, World!"**

**在這個例子中，innerFunction 記住了 outerVariable，即使 outerFunction 已經執行完了。**

* **innerVariable：從函數調用時提供的參數（"World!"）獲得其值。**
* **outerVariable：保持不變，保留其原始值（"Hello, "），這是在調用 outerFunction 時設置的。**
* **簡而言之，innerVariable 是針對 innerFunction 的調用而特定的，而 outerVariable 則是由 outerFunction 創建的閉包的一部分。**

## A. 用途 (Uses):

* **保護資料（隱私） (Data Privacy/Encapsulation):**
  + 閉包就像是一個安全的保險箱，你可以把一些重要的資料（變數）放進去，只有擁有鑰匙（閉包函數）的人才能打開。
  + 在程式設計中，這表示你可以建立一些「私有」的變數或函數，它們只能在特定的範圍內存取，外部無法直接修改或查看。
  + 這有助於保護資料的完整性，避免意外的修改或錯誤。
    - [**例子 (Example):**](#_eoq4m9t2jgca)   
      假設你要做一個計數器 (counter)，你不想讓別人隨便改計數器的數字。你可以用閉包 (closure) 把計數器的數字藏起來。
    - **例子 (Example):**   
      假設你要設計一個線上遊戲的「角色」物件。每個角色都有一些屬性，例如生命值、經驗值、等級。  
      你可以使用閉包來封裝這些屬性，只允許透過角色的「方法」（例如「攻擊」、「升級」）來修改它們。  
      這樣，其他玩家或程式碼就無法直接修改角色的生命值，確保遊戲的公平性和安全性。
* **事件處理（Event Handlers） (Event Handlers):**
  + 當你點擊按鈕 (button) 時，電腦需要知道你點了哪個按鈕。閉包 (closure) 可以讓電腦記住你點的是哪個。
    - **例子 (Example):**   
      假設你設計一個網頁，上面有很多個產品圖片，每個圖片都有一個「加入購物車」的按鈕。  
      當使用者點擊任何一個按鈕時，你需要知道使用者點擊的是哪個產品，才能將正確的產品加入購物車。  
      你可以使用閉包來為每個按鈕建立一個事件處理器，這個處理器會「記住」對應產品的資訊。  
      這樣，無論使用者點擊哪個按鈕，你都能正確地知道他們想買哪個產品。
    - 其他例子
* **回呼函數（Callbacks） (Callbacks):**
  + 非同步操作的通知：
    - 在 JavaScript 中，有些操作需要花費時間才能完成，例如網路請求、檔案讀寫、計時器等。
    - 這些操作是非同步的，也就是說，程式碼不會等待這些操作完成，而是繼續執行後續的程式碼。
    - 回呼函數的作用，就是在這些非同步操作完成後，通知程式操作已經完成。
  + 取得操作結果：
    - 非同步操作完成後，通常會產生一些結果（例如從伺服器取得的資料、讀取的檔案內容）。
    - 回呼函數可以接收這些結果作為參數，讓你能夠在回呼函數內部處理這些資料。
  + 在外部處理資料：
    - 由於回呼函數是在非同步操作完成後被執行的，因此你可以在回呼函數內部，將操作結果傳遞給外部的變數、函數，或者執行其他操作。
    - 這使得你能夠在非同步操作完成後，在外部的程式碼中處理這些資料。
    - **例子 (Example):**   
      假設你要從伺服器下載一些資料，下載需要一些時間。  
      你可以編寫一個下載函數，並傳入一個回呼函數，當下載完成時，下載函數會呼叫回呼函數。  
      你可以使用閉包來讓回呼函數「記住」下載開始時的資料，這樣回呼函數就能正確地處理下載的結果。
* **模組化（Module Pattern） (Module Pattern):** 就像把程式碼 (code) 分成很多個小盒子 (modules)，每個盒子都有自己的功能，不會互相干擾。
  + **例子 (Example):** 你可以把一個計算器 (calculator) 做成一個模組 (module)，把顯示介面 (user interface) 做成另一個模組 (module)，然後用閉包 (closure) 讓它們互相溝通。

## 閉包的實際用途（Practical Use Cases of Closures）

閉包(Closures)在 JavaScript 開發中有很多實際用途。通常用來建立工廠函數（factory functions）、封裝資料（encapsulate data），以及在回調（callbacks）和事件處理器（event handlers）中管理狀態（manage state）。

### A1. 資料隱私／封裝（Data Privacy/Encapsulation）

閉包可以用來建立私有變數（private variables）或私有方法（private methods），這是物件導向程式設計（Object-Oriented Programming, OOP）的基本概念。使用閉包，你可以建立無法從函數外部直接存取的變數，提供一種封裝資料的方式。

**範例（Example）：**

function createBankAccount(initialBalance) {

let balance = initialBalance;

return {

getBalance: function() {

return balance;

},

deposit: function(amount) {

balance += amount;

return balance;

},

withdraw: function(amount) {

if (amount > balance) {

console.log('Insufficient funds');

return;

}

balance -= amount;

return balance;

},

};

}

const account = createBankAccount(100);

console.log(account.getBalance()); // 100

account.deposit(50);

console.log(account.getBalance()); // 150

account.withdraw(30);

console.log(account.getBalance()); // 120

在這個範例中，定義了一個 createBankAccount 函數（function），它接受一個初始餘額（initial balance）作為參數（parameter）。它初始化一個 balance 變數（variable），然後回傳一個物件（object），這個物件有三個方法（methods）（getBalance, deposit, withdraw），這些方法可以存取和修改 balance。

balance 無法從函數外部直接存取，這樣就確保了資料隱私（data privacy）。

* getBalance 讓你查看目前的餘額（current balance）。
* deposit 將指定的金額加到 balance。
* withdraw 將指定的金額從 balance 中扣除，如果餘額足夠的話。

當 createBankAccount 被呼叫（called）時，它會回傳一個物件（object），這個物件的方法（methods）可以存取 balance，即使 createBankAccount 函數（function）的執行環境（execution context）已經消失。這就是閉包（closure）的一個範例。

**簡單來說（Simply put）：**

就像你建立了一個銀行帳戶，帳戶的餘額（balance）是隱藏的，只有銀行的職員（這裡就是 getBalance, deposit, withdraw 這些方法）才能查看和修改。即使建立帳戶的人（createBankAccount 函數）離開了，職員仍然可以繼續操作帳戶。這就是閉包的作用。

### A1.1 維持狀態（Maintaining State）

閉包（Closures）是一種在函數呼叫之間維持狀態的好方法。當處理非同步程式碼（asynchronous code）、事件處理器（event handlers），或任何需要長時間保存特定狀態的情況時，這個特性特別有用。

例子：

function createCounter() {

let count = 0;

return function() { // state.count=2

count += 1;

console.log(count);

};

}

const counter = createCounter();

counter(); // 輸出：1

counter(); // 輸出：2

counter(); // 輸出：3

在這個範例中，counter 函數在不同的呼叫之間維持了 count 的狀態。

* createCounter 初始化一個 count 變數，並回傳一個匿名函數（anonymous function）。
* 每次呼叫回傳的函數時，它都會將 count 遞增並記錄該值。
* count 在回傳函數的呼叫之間保持其值，使其能夠作為一個持續的計數器。

**為什麼回傳的閉包函數能維持外部狀態，而不是每次都將 count 重置為 0？**

關鍵在於閉包的特性：

1. **詞法作用域（Lexical Scope）：**
   * 當 createCounter 函數執行時，它創建了一個新的作用域（scope）。
   * let count = 0; 在這個作用域中宣告了一個變數 count。
   * 回傳的匿名函數「記住」了這個作用域。
2. **閉包的形成：**
   * 回傳的匿名函數形成了一個閉包，它「封閉」了 createCounter 函數的作用域。
   * 這意味著，即使 createCounter 函數已經執行完畢，回傳的匿名函數仍然可以存取 count 變數。
3. **狀態的保存：**
   * 當你第一次呼叫 counter() 時，回傳的匿名函數會存取並修改 count 變數。
     1. let count = 0; 被執行，在 createCounter() 的作用域內創建了一個名為 count 的變數，並將其初始化為 0。
     2. return function() { ... }; 被執行，它返回一個匿名函數（也就是閉包）。這個匿名函數「記住」了 createCounter() 的作用域，包括 count 變數。
   * 由於閉包的特性，count 變數的值被保留下來。
   * 當你第二次、第三次呼叫 counter() 時，它會繼續存取並修改同一個 count 變數，而不是重新宣告一個新的 count 變數。
4. **避免重置：**
   * let count = 0; 只在 createCounter 函數第一次執行時執行。
   * 回傳的匿名函數每次被呼叫時，它都是存取已經存在的 count 變數，而不是重新執行 let count = 0;。

**簡單來說：**

就像你把一個盒子（count 變數）放在一個房間（createCounter 函數的作用域）裡，然後把一個知道這個房間在哪裡的人（回傳的匿名函數）帶走。這個人每次回到房間，都會找到同一個盒子，而不是重新放一個新的盒子。

因此，閉包函數能夠維持外部狀態，因為它保留了對外部作用域的存取權限，並且不會重新初始化外部作用域的變數。

### A1.3 工廠函數（Factory Functions）

閉包可以用來建立工廠函數，工廠函數會回傳一個物件（object），這個物件裡面有一些方法（methods）和屬性（properties），這些方法和屬性可以存取閉包內部的私有資料（private data）。

範例：

function personFactory(name, age) { // 工廠函數，建立 person 物件

return { // 回傳一個物件

getName: function() { // 取得 name 的方法

return name;

},

getAge: function() { // 取得 age 的方法

return age;

},

celebrateBirthday: function() { // 慶祝生日，增加 age 的方法

age += 1;

},

};

}

const john = personFactory('John', 30); // 建立一個 person 物件，名稱是 John，年齡是 30

console.log(john.getName()); // 輸出：John（Output: John）

console.log(john.getAge()); // 輸出：30（Output: 30）

john.celebrateBirthday(); // 慶祝生日

console.log(john.getAge()); // 輸出：31（Output: 31）

**解釋：**

* personFactory 函數接受 name 和 age 參數（parameters），並回傳一個物件（object），這個物件有三個方法（methods）：getName、getAge 和 celebrateBirthday。
* 每個方法都可以存取 name 和 age，這就是閉包行為（closure behavior）的展現。
* celebrateBirthday 方法可以修改 age，這說明了閉包如何更新封閉的變數（enclosed variables）。

**簡單來說：**

想像一下，personFactory 就像是一個「人偶工廠」。你給它一個名字和年齡，它就幫你做出一個「人偶」（物件）。這個人偶有自己的名字和年齡，而且它還會長大（慶祝生日）。

* **工廠函數（Factory Functions）：**
  + 就像一個「製造工廠」，專門生產特定類型的物件。
* **物件（Objects）：**
  + 就像工廠生產出來的「產品」，有自己的屬性和方法。
* **方法（Methods）：**
  + 就像產品的「功能」，可以執行特定的動作。
* **屬性（Properties）：**
  + 就像產品的「特性」，描述產品的狀態。
* **閉包行為（Closure Behavior）：**
  + 就像人偶可以記住自己的名字和年齡，即使工廠已經關閉（函數執行完畢）。
* **封閉的變數（Enclosed Variables）：**
  + 就像人偶內部的「秘密」，只有人偶自己和工廠（閉包）知道。

通過工廠函數和閉包，我們可以更靈活地創建物件，並實現資料隱藏和狀態管理。

### 

### A2 事件處理（Event Handling）

閉包（Closure）在事件處理（Event Handling）中被用來封裝狀態（state），並提供跨不同作用域（scopes）存取變數（variables）的能力。

**範例（Example）：**

function setupButtons() {

for (var i = 0; i < 3; i++) {

document.getElementById('button' + i).addEventListener('click', (function(i) {

return function() {

alert('Button ' + i + ' clicked');

};

})(i));

}

}

這段程式碼使用 IIFE 和閉包（closure），確保每個按鈕（button）的點擊事件處理器（click event handler）都能夠正確地存取其對應的迴圈變數（loop variable） i，從而正確顯示被點擊的按鈕編號（button number）。

好的，讓我們來看看在哪些情況下，每次呼叫回傳的閉包函數時，count 會被重置為 0，並探討 let、var 和 const 在此情境下的影響。

**重置 count 為 0 的情況：**

1. **每次呼叫 createCounter 函數：**
   * 如果你每次呼叫 createCounter 函數，它都會重新執行 let count = 0;，並回傳一個新的閉包函數。
   * 這樣，每次呼叫回傳的閉包函數時，它都會操作一個新的 count 變數，而不是保留之前的狀態。
   * function createCounter() {
   * let count = 0;
   * return function() {
   * count += 1;
   * console.log(count);
   * };
   * }
   * const counter1 = createCounter();
   * counter1(); // 輸出：1
   * counter1(); // 輸出：2
   * const counter2 = createCounter(); // 再次呼叫 createCounter
   * counter2(); // 輸出：1 (重置為 0)
2. **在閉包函數內部重新宣告 count：**
   * 如果你在回傳的閉包函數內部重新宣告 count，它會遮蔽（shadow）外部作用域的 count 變數。
   * 每次呼叫閉包函數時，它都會操作內部宣告的 count 變數，而不是外部的 count 變數。

function createCounter() {

let count = 0;

return function() {

let count = 0; // 在閉包內部重新宣告 count

count += 1;

console.log(count);

};

}

const counter = createCounter();

counter(); // 輸出：1

3counter(); // 輸出：1 (每次都重置為 0)

## B. 可能的陷阱 (Potential Pitfalls):

### **1. 記憶體洩漏 (Memory Leaks):** 如果閉包 (closure) 記住太多東西 (objects)，電腦可能會忘記釋放記憶體 (memory)，導致程式變慢。

* + **例子 (Example):** 如果閉包 (closure) 一直記住一個很大的圖片 (image)，就算你不再需要這張圖片，電腦還是會一直記住它。
  + [解決方案](#_3buexs5mgogs)

### **2. 變數值不對 (Unexpected Variable Values):** 如果閉包 (closure) 裡面的變數 (variables) 被改了，可能會出現意想不到的結果。

* + [**例子 (Example)**](#_j86fzsbqvkag)**:** 在迴圈 (loop) 中，如果閉包 (closure) 內部使用了迴圈的計數器 (loop counter)，可能會因為計數器在迴圈結束後的值，導致閉包的行為和你預期不同。

### **3. 速度變慢 (Performance Overhead):** 因為閉包 (closure) 要記住東西，所以可能會稍微慢一點。

## 可能的陷阱 (Potential Pitfalls) 補充資訊：

### B1.1 記憶體洩漏 (Memory Leaks) 解決方案 - 強制重置閉包狀態的方法：

如何強制重置閉包（Closure）的狀態，特別是當閉包記住大型物件（objects）如圖片（images）時，如何釋放記憶體（memory）。

**強制重置閉包狀態的方法：**

1. 解除引用（Dereferencing）：
   * 將閉包內部引用的物件（例如圖片）設定為 null，使其不再被閉包引用。
   * 這樣，垃圾回收機制（garbage collector）就能夠回收這些物件所佔用的記憶體。
   * function createImageProcessor(imageUrl) {
   * let image = new Image();
   * image.src = imageUrl;
   * return {
   * processImage: function() {
   * // 處理圖片的程式碼
   * console.log('Processing image...');
   * },
   * releaseImage: function() {
   * image.src = ''; // 清空圖片來源
   * image = null; // 解除對圖片的引用
   * console.log('Image released.');
   * },
   * };
   * }
   * const processor = createImageProcessor('large-image.jpg');
   * processor.processImage();
   * // 不再需要圖片時
   * processor.releaseImage();
2. 重新賦值（Reassignment）：

如果閉包是透過函數創建的，可以重新呼叫該函數，並將回傳的新閉包賦值給原來的變數。

這樣，原來的閉包就會被垃圾回收機制回收。

function createCounter() {

let count = 0;

return function() {

count += 1;

console.log(count);

};

}

let counter = createCounter();

counter(); // 輸出：1

// 重置計數器

counter = createCounter(); // 建立新的閉包

counter(); // 輸出：1 (重新開始)

#### 

### B2.1 變數值不對（Unexpected Variable Values）例子：

* + function createButtons() {
  + for (var i = 0; i < 5; i++) {
  + const button = document.createElement('button');
  + button.textContent = 'Button ' + i;
  + button.addEventListener('click', function() {
  + console.log('Button ' + i + ' clicked');
  + });
  + document.body.appendChild(button);
  + }
  + }
  + createButtons();

**問題說明：**

* + - 在這個範例中，我們創建了 5 個按鈕（buttons），並為每個按鈕添加了一個點擊事件處理器（click event handler）。
    - 事件處理器是一個閉包，它使用了迴圈的計數器 i。
    - 由於 var 宣告的變數具有函數作用域（function scope），因此所有閉包都共享同一個 i 變數。
    - 當迴圈結束時，i 的值變為 5。
    - 當點擊任何一個按鈕時，控制台（console）都會輸出 "Button 5 clicked"，而不是預期的 "Button 0 clicked"、"Button 1 clicked" 等。

**解決方法：**

1. 使用 let 或 const 宣告計數器，因為它們具有塊級作用域（block scope）。

function createButtons() {

for (let i = 0; i < 5; i++) {

const button = document.createElement('button');

button.textContent = 'Button ' + i;

button.addEventListener('click', function() {

console.log('Button ' + i + ' clicked');

});

document.body.appendChild(button);

}

}

createButtons();

1. 使用立即執行函式（Immediately Invoked Function Expressions, IIFE）來創建新的作用域。

function createButtons() {

for (var i = 0; i < 5; i++) {

(function(index) {

const button = document.createElement('button');

button.textContent = 'Button ' + index;

button.addEventListener('click', function() {

console.log('Button ' + index + ' clicked');

});

document.body.appendChild(button);

})(i);

}

}

createButtons();

## C. 優化方法 (Optimizations):

* **少記一點 (Minimize Variable Capture):** 只需要記住需要的變數 (variables) 就好。
* **忘記不需要的東西 (Release References):** 如果不再需要某個變數 (variable) 或物件 (object)，就讓電腦忘記它。
* **用 let 和 const (Use let and const):** 這樣可以更清楚地控制變數 (variables) 的範圍 (scope)。
* **不要亂用閉包 (Avoid Unnecessary Closures):** 如果不需要閉包 (closure)，就不要用它。

**檢查程式 (Profile Your Code):** 用工具 (profiling tools) 檢查程式，看看有沒有記憶體洩漏 (memory leaks) 或速度變慢 (performance bottlenecks) 的問題。

以下是使用工具（profiling tools）檢查程式碼，找出記憶體洩漏（memory leaks）或效能瓶頸（performance bottlenecks）的方法：

E.g. <https://google.com>

**1. 瀏覽器開發者工具（Browser Developer Tools）：**

* 現代瀏覽器（例如 Chrome、Firefox、Edge）都內建了強大的開發者工具，可用於剖析 JavaScript 程式碼。
* **記憶體分析（Memory Profiling）：**
  + 開啟開發者工具的「Memory」面板。
  + 使用「Take heap snapshot」功能，拍攝記憶體快照。
  + 重複執行程式碼，並拍攝多個記憶體快照。
  + 比較不同快照之間的差異，找出記憶體使用量不斷增加的物件，這些物件可能是記憶體洩漏的來源。
  + 查看物件的保留路徑（retaining paths），找出哪些物件仍然被引用，導致無法被回收。
* **效能分析（Performance Profiling）：**
  + 開啟開發者工具的「Performance」面板。
  + 點擊「Record」按鈕，開始錄製效能資料。
  + 執行需要分析的程式碼。
  + 點擊「Stop」按鈕，停止錄製。
  + 分析錄製的資料，找出執行時間較長的函數或程式碼區塊。
  + 查看函數的呼叫堆疊（call stack），找出效能瓶頸的根源。
  + 分析幀率(FPS)，查看頁面是否有掉幀（FPS 突然下降）的現象，查看原因，比如JavaScript 程式碼執行時間過長、 CSS 動畫或轉場效果複雜、圖片或影片資源過大、 硬體效能不足。

**使用 IIFE 解決迴圈中的閉包問題：**

for (var i = 0; i < 5; i++) {

(function(index) {

setTimeout(function() {

console.log(index);

}, index \* 1000);

})(i);

}

在這個範例中，IIFE 創建了一個新的作用域，並將迴圈的計數器 i 作為參數 index 傳遞給 IIFE。這樣，每個 setTimeout 回調函數都能夠保留其創建時的 index 值，從而避免了變數污染問題。

總結來說，IIFE 是一種強大的 JavaScript 設計模式，它能夠幫助您創建獨立的作用域、隱藏資料、避免變數污染，並提高程式碼的模組化和可維護性。

Exercise

1. 閉包(closure) 與計數器

function makeCounter() { // 被調用函數時，創建了新的函數作用域（function scope）

let count = 0; // 區域變數，只有第一次調用，才被賦予0的值

// 定義一個閉包函數，用於遞增計數器並回傳當前值

return function() { // 創建一個新的函數作用域（匿名函數的作用域），並且可以存取 makeCounter 的作用域

count += 2; // 變數修改：將 makeCounter 作用域內的 count 變數遞增 2

return count; // 回傳 makeCounter 作用域內的 count 變數的當前值

};

}

const counter = makeCounter();

console.log(counter()); // 2

console.log(counter()); // 4

console.log(counter()); // 6

2. 全域與局部作用域的混合

var x = 1; // 全域變數

function myFunction() { // 被調用函數時，創建了新的函數作用域（function scope）

var x = 2; // 新的函數作用域下的區域變數，只有第一次調用，才被賦予2的值

function innerFunction() {

x++;

return x;

}

return innerFunction();

}

console.log(myFunction()); // 調用 myFunction()，並將其返回值 3 輸出到控制台。

console.log(x); // 輸出全域變數 x 的值，也就是 1。

// 全域變數 x 和 myFunction() 作用域內的 x 是兩個不同的變數。

**作用域鏈（Scope Chain）：**

* 在 myFunction() 的作用域內，x 的值是 2。
* 當 innerFunction() 嘗試存取 x 時，它首先在自己的作用域內尋找。
* 如果在自己的作用域內找不到 x，它會沿著作用域鏈向上尋找，也就是在 myFunction() 的作用域內尋找。
* 它不會去全域作用域尋找，因為在 myFunction() 的作用域內已經找到了 x。

**為什麼 innerFunction() 中的 x 從 var x = 2; 取得？**

* **作用域優先級：**
  + JavaScript 在尋找變數時，會優先在當前作用域內尋找。
  + 如果在當前作用域內找到了變數，就不會繼續向上層作用域尋找。
* **閉包的特性：**
  + innerFunction() 是一個閉包，它可以存取 myFunction() 的作用域。
  + 因此，innerFunction() 可以存取 myFunction() 作用域內的 x 變數。