JavaScript 程式設計新手村

單元16 - JavaScript 物件 Object 基礎（下）

@kdchang

Outline

1. 自己建立 JavaScript 物件
2. JavaScript 物件導向基礎
3. this 的主要用法和使用情境
4. 命名空間

自己建立 JavaScript 物件

自建 JavaScript 物件的四種主要方式

1. literal notation
2. new Object()
3. 使用建構函數建立物件
4. class（ES6+ 後有的特性）

使用 literal notation 建立物件

```
const nameCard = {
  name: 'Tony',
  age: 20,
  email: 'tonycc@gmail.com',
  printCard: function(){ //名稱：值(匿名函數)，物件方法
    console.log(this.name);
    console.log(this.email);
  }
} //this關鍵字參考物件本身
nameCard.name; // 亦可用 nameCard['name']; 存取
nameCard.printCard();
```

使用 Object 物件建立

Object 為所有 JavaScript 物件的祖先，我們也可以使用 Object 當做建構函數來建立物件，建立的是空屬性，繼承 Obeject 的物件

```
const nameCard = new Object(); // 等同 var nameCard = {};
nameCard.name = 'Tony'; // 新增屬性
//新增方法#1
nameCard.printCard = print;
function print() {
    console.log(this.name);
}
//新增方法#2(使用匿名函數的方式)
nameCard.printCard = function(){
    console.log(this.name);
}

nameCard.printCard();
```

使用建構函數建立物件

- Constructor (建構函數)是一個函數，能夠定義物件的屬性和方法（此種方式可以運用 prototype 繼承的特性）
- JavaScript 的內建函數 (EX. String 等) 就是建構函數，一般建構函數的首字大寫，this 關鍵字指物件本身
- 建立物件步驟如下：
    i. 使用建構函數宣告物件
    ii. 使用 `new` 運算子建立物件

使用建構函數建立物件

```
function NameCard(name, phone, email) {
        this.name = name;
        this.phone = phone;
        this.email = email;
        this.print = printCard;
}

function printCard(){
        console.log('printCard');
        console.log(this.phone);
}

NameCard.prototype.sayHi = function() {
        console.log('hi');
        console.log(this.email);
}
// prototype 用法，實例會共用方法（節省記憶體），在類別定義會每次實
var myCard = new NameCard('CD', '091234567', 'cd.cc@gmail.com')
myCard.print();
myCard.sayHi();
console.log(myCard.name);
```

使用 ES6 class 建立物件

```
class Cat {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(this.name + '喵喵');
  }
}

const c = new Cat('Mitzie');
c.speak();
```

所有物件都共用的方法

- `hasOwnProperty()`
  可以檢查物件是否直接擁有參數的屬性，回傳布林值，如為原型繼承則非直接擁有會回傳 false
- `isPrototypeOf()`
  可以檢查參數物件是否存在於其它原型物件的繼承鏈中，回傳布林值
- `toString()`
  輸出物件的內容字串
- `valueOf()`
  可以傳回物件數值

物件迭代取值

```
const obj = {
  name: 'MOMO',
  tel: '0982134512'
}
arr = ['coding', 'design'];

arr.forEach(function(value) {
  console.log(value);
});

Object.keys(obj).forEach(function(value) {
  console.log(obj[value]);
});

for (prop in obj) {
  if (obj.hasOwnProperty(prop)) {
    console.log(obj[prop]);
  }
}
```

範例程式

靜態屬性、方法

靜態屬性和方法為其他物件導向語言中類別(class)的屬性和方法。在 JS 中則是直接對建構函數新增靜態屬性和方法，不需要實例化才能存取，且靜態類別不能實現化 (Instantiated)

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static sum(a, b) {
      return a + b;
    }
}

console.log(Point.sum(1, 2));
```

JavaScript 物件導向基礎

物件導向（OOP）特性：

1. 封裝 (Encapsulation)
   封裝是將資料和函數建立成物件，亦即物件是由屬性和 method 方法（函數）所組成的黑盒子
2. 繼承 (Inheritance)
   重複利用類別的屬性和方法，JavaScript 使用 prototype
3. 多型 (Polymorphism)
   繼承父類別並使用同名方法但依照需求修改內容程式碼

原型物件、建構函數和繼承

- JavaScript 是 `prototype-based` (原型基礎)的物件導向，不同於 C++/Java/C# 等的 `class-based` （類別基礎）的物件導向
- 類別視為一個 `模子` ，開模後製出的物件實體都具備相同的功能
- 原型視為 `基底` 或是 `賽車底盤` ，可以動態加蓋其他功能或屬性

建構函數繼承方式

```
function Parent(name) {
    this.name = name;
}
// 共用，省記憶體
Parent.prototype.speak = function () {
    console.log(this.name);
};
// 各複製一份
function Child() {
    this.act = "cry";
    this.speak2 = function () {
        console.log(this.act);
    };
}

Child.prototype = new Parent("James");
Child.prototype.constructor = Child;
var c = new Child();
c.speak2();
c.speak();
console.log(c.name);
```

範例程式

ES6 class 繼承方式（語法糖）

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(this.name + '喵喵');
  }
}

class Dog extends Animal {
  speak() {
    console.log(this.name + '旺旺');
  }
  parentFunc() {
    super.speak();
  }
}
const d = new Dog('Mitzie');
d.speak();
d.parentFunc();
```

範例程式

this 的主要用法和使用情境

this 的四種主要用法

1. 隱性綁定 Implicit Binding（`{}`）
2. 顯性綁定 Explicit Binding（`call`、`apply`、`bind`）
3. new Binding（`new Object`）
4. 全域 window Binding
5. callback

延伸閱讀：#Javascript：this用法整理

隱性綁定 Implicit Binding（`{}`）

大原則：誰呼叫函式的就指到誰

```
var me = {
        name: 'Tyler',
        age: 25,
        sayName: function() {
                console.log(this.name);
        }
};

me.sayName();
```

顯性綁定 Explicit Binding（`call`、`apply`、`bind`）

為了解決隱性綁定 Implicit 問題，顯性綁定可以指派 this 的值，就是利用 `call` 與 `apply`，差別在 `apply` 參數使用陣列。`bind` 類似 `call` 只是可以傳給參數延遲呼叫

```
(A物件.)函式.call(B物件,參數1,參數2,參數3, ......);
//函式的 this 會指向B物件(若B物件為null，則指向全域物件)

(A物件.)函式.apply(B物件,[參數1,參數2,參數3, ......]);
//函式的 this 會指向B物件(若B物件為null，則指向全域物件)
```

顯性綁定 Explicit Binding（`call`、`apply`、`bind`）

```
var sayName = function(lang1, lang2, lang3) {
        console.log('My name is ' + this.name + lang1 + lang2 +
};

var stacey = {
        name: 'Stacey',
        age: 34
};

var languages = ['JS', 'Python', 'Swift'];

sayName.call(stacey, languages[0], languages[1], languages[2]);

sayName.apply(stacey, languages);

var newFn = sayName.bind(stacey, languages[0], languages[1], la
newFn();
```

new Binding `new Object`

若將函式當作建構式（constructor）來用，則內部的 `this` 則指向於new所產生之新物件

```
var Animal = function(color, name, type) {
        this.color = color;
        this.name = name;
        this.type = type;
}

var zebra = new Animal('black & white', 'Zorro', 'Zebra');
console.log(zebra.color);
```

全域 window Binding

```
var age = 12

var sayAge = function() {
        'use strict';
        console.log(this.age);
};

var me = {
        age: 25
};

sayAge();
```

全域 window Binding 誤解

```
var x = 10;
var obj = {
        x: 20,
        f: function(){
                console.log(this.x);
                var foo = function(){ console.log(this.x); }
                foo(); // (2)
        }
};

obj.f();  // (1)
```

全域 window Binding 解法

```
var x = 10;
var obj = {
        x: 20,
        f: function(){
                console.log(this.x);
                var that = this; //使用that保留在這個函式內的this
                var foo = function(){ console.log(that.x); } //
                foo();
        }
};

obj.f();
```

callback

這是 jQuery 中點擊按鈕的事件處理，當中的 this 會指到點擊的按鈕元素，好神奇！

```
$('#button').click(function(){
        this.html("Clicked");
})
```

實際底層實作在函數中傳入 callback 函數 innerf，讓 this 指向於調用放入該 callback 函式的 this 所指向之物件（上面例子就是按鈕）

```
var click = function(innerf){
        //前面的處理
        innerf.call(this, arg1, arg2, arg3, ......);
        //或是innerf.apply(this, [arg1, arg2, arg3, ......])
        //後面的處理
}
```

命名空間

命名空間

用於避免變數或函數因名稱相同而產生問題。在 JavaScript 中最常使用 Function Wrapper 來實作命名空間，在各大 Library 函式庫都可以看到使用。其中回傳的物件為使用介面的相關方法，透過前面提到的自調用函數可以建立獨立的命名空間，避免衝突

```
const App = App || {};
(function(a){
  var num = 1;
  a.add = function(){ return num++ };
})(App);
```

延伸閱讀：ECMAScript 6 Feature

總結

在這個章節中我們學會了：

1. 自己建立 JavaScript 物件
2. this 的主要用法和使用情境
3. 命名空間