JavaScript this

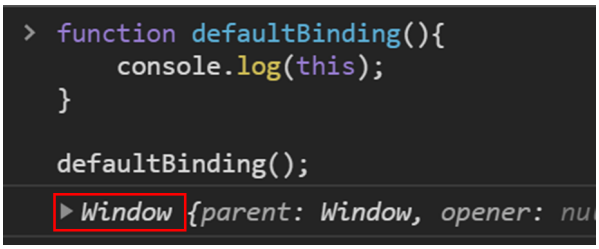
主要來說可以大致分為4種情況：

1. 預設綁定
2. 隱含綁定
3. 明確綁定
4. new關鍵字綁定

還有一個不合群的，就是在箭頭函式中的this。

預設綁定(Default Binding)

如果有寫過其他程式語言，就會知道在物件導向語言中，this代表的是那個 instance 本身。在JavaScript當中this的定義雖然跟其他語言不太一樣，但一旦脫離了物件，this也沒有太大的意義，就像下面的情況：




```
// 預設綁定
function defaultBinding(){
    console.log(this);
}
defaultBinding(); // window
```

所以在沒有意義的情況下，this有一個預設值，在非嚴格模式的情況下就是`window`，嚴格模式的情況下就是 `undefined`。

也就是：

1. 嚴格模式底下就都是`undefined`
2. 非嚴格模式，瀏覽器底下是`window`
3. 非嚴格模式，node.js 底下是`global`

而this在很多情況下是可以被動態更改的，以下會介紹幾種變更this指向的情況。

在繼續往下前，請先記住一點：

> this跟scope chain（範圍鏈）不同，跟本身的lexical context（詞彙環境）無關，只跟「如何呼叫」有關

若有興趣可參考以下兩篇：

範圍鏈 Scope Chain

範圍鏈 v.s this

隱含綁定 (Implicit Binding)

還記得上面所說的，this的值跟他在哪一行沒有關係，而是跟誰呼叫有關，簡單的說在有「物件」的情況下，簡單來說有一個規則，就是：

> 誰（物件）呼叫我，我(this)就指向誰


```
// 隱含綁定
var getDogName = function(){
    console.log(this.name);
}
var dog1 = {
    name: '黃金',
    getName: getDogName
};
dog1.getName(); // 黃金
```

這裡呼叫getDogName 的是物件dog1，所以getDogName 中的this也會指向dog1。

隱含綁定遺失

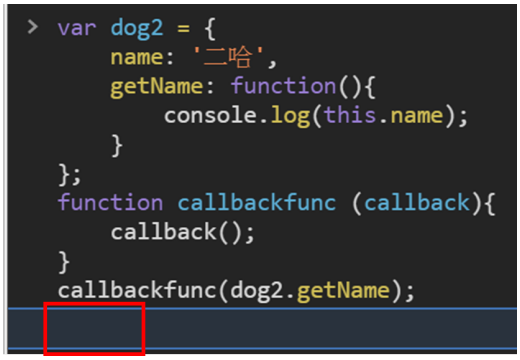
再來看看這一個例子：


```
// this隱含綁定遺失
var dog2 = {
    name: '二哈',
    getName: function(){
        console.log(this.name);
    }
};
dog2.getName(); // 二哈
var husky = dog2.getName;
husky(); // undefined
```

為什麼同一個函式，在第一次呼叫實name是二哈，第二次呼叫卻是undefined?

因為在某些情況下將造成隱含綁定的遺失，而上面這個情況屬於**傳遞參照(pass reference)**，在傳遞的過程中會遺失this的指向。

另一種會造成this指向遺失的則是**回呼函式(callback function)**：




```
var dog2 = {
    name: '二哈',
    getName: function(){
        console.log(this.name);
    }
};
function callbackfunc (callback){
    callback();
}
callbackfunc(dog2.getName); // undefined
```

總結一下，在有物件的情況下都是遵照「誰（物件）呼叫我，我(this)就指向誰」，但有兩種情況將造成隱含綁定遺失：

1. **傳遞參照(pass reference)**
2. **回呼函式(callback function)**

明確綁定

上面一開始有提到過某些情況下this式的指向是可以被動態更改的，現在就要來講講哪些方法可以變更this的指向。

這裡有三種方法可以更改：

1. `call`
2. `apply`
3. `bind`

其中`call`跟`apply`有點類似，先舉個例子：

```
'use strict
function saySomething(word1, word2){
    console.log('我是' + this.name + '，我要' + word1 + '跟' + word2);
}

saySomething('罐罐', '小魚乾'); // 我是undefined，我要罐罐跟小魚乾
saySomething.call(undefined, '罐罐', '小魚乾'); // 我是undefined，我要罐罐跟小魚乾
saySomething.apply(undefined, ['罐罐', '小魚乾']); // 我是undefined，我要罐罐跟小魚乾
```

有一個saySomething的函式，會印出this的值以及兩個參數word1跟word2。

在呼叫`saySomething('罐罐', '小魚乾')`時，由於在嚴格模式下執行，所以this的值是`undefined`，而word1跟word2則分別是罐罐、小魚乾。

下面兩個使用`call`跟`apply`的方法，忽略第一個參數，印出的結果跟`saySomething('罐罐', '小魚乾')`是一樣的。而`call`跟`apply`的差別只在於傳入參數的方式不同，`apply`除了第一個參數外，其餘的必須以**陣列**的形式傳入。

這樣看來，第一個傳入的參數代表什麼呢?

答案是**this要指向的對象**，也就是this的值。

第一個參數傳入什麼，不管原本this是不是已經有指定的對象，經過`call`跟`apply`後，this的值都會被覆蓋（更改）。

```
this.name = '人類';
function saySomething(word1, word2){
    console.log('我是' + this.name + '，我要' + word1 + '跟' + word2);
}

var lulu = {
```

```
    name: 'lulu'
}

saySomething.call(lulu, '罐罐', '小魚乾'); // 我是lulu，我要罐罐跟小魚乾
saySomething.apply(lulu, ['罐罐', '小魚乾']); // 我是lulu，我要罐罐跟小魚乾
```

最後一種可以改變this值的方法是`bind`。

```
this.name = 123;
function saySomething(word1, word2){
    console.log('我是' + this.name + '，我要' + word1 + '跟' + word2);
}

var lulu = {
    name: 'lulu'
}

var cat = saySomething.bind(lulu, '罐罐', '小魚乾');
cat(); // 我是lulu，我要罐罐跟小魚乾
```

那`call`、`apply`跟`bind`到底有什麼區別呢?

```
function saySomething(word1, word2){
    console.log('我是' + this.name + '，我要' + word1 + '跟' + word2);
}

var lulu = {
    name: 'lulu'
}

saySomething.call(lulu, '罐罐', '小魚乾'); // 我是lulu，我要罐罐跟小魚乾
saySomething.apply(lulu, ['罐罐', '小魚乾']); // 我是lulu，我要罐罐跟小魚乾
var cat = saySomething.bind(lulu, '罐罐', '小魚乾');
```

執行上面的程式碼，會發現`call`跟`apply`都有印出東西，而用了`bind`的functionfunction沒有。

那是因為:

> 使用`call`及`apply`後，會立即執行，但`bind`只會回傳一個新函式

要印出bind之後的結果，必須呼叫`cat();`才能 印出函式的內容。

如果把`call`、`apply`跟`bind`混在一起用，會有什麼結果呢?

```
function saySomething(word1, word2){
    console.log('我是' + this.name + '，我要' + word1 + '跟' + word2);
}

var lulu = {
    name: 'lulu'
}

var threefeet = {
    name: '三腳'
}

var cat = saySomething.bind(lulu);
cat.call(threefeet, '枕頭', '睡覺'); // 我是lulu，我要枕頭跟睡覺
```

會發現一旦`bind`之後，無論使用`call`或是`apply`都沒有辦法再改變this的值了。

總結一下:

1. `call`：將第一個參數作為this綁定對象，並立即執行函式
2. `apply`：同`call`，但後面參數以陣列傳入
3. `bind`：將第一個參數作為this綁定對向，並回傳新函式（不執行）

所以明確綁定可以理解為這三個方法可以**明確地告訴你要把this綁定到誰身上**，這樣在使用上也比較好判斷到底是哪一種綁定。

new關鍵字綁定

顧名思義，就是出現`new`這個關鍵字之後，this指向的變化，舉個例子：


```
function Dog (name, bark){
    this.name = name;
    this.bark = bark;
}

var dog3 = new Dog('杜賓', '嗷嗚');
console.log(dog3); // Dog {name: "杜賓", bark: "嗷嗚"}
```

當我使用`new`關鍵字產生新物件dog3，他會依照上面的Dog函式建構式，建立相同格式（屬性、方法）的物件，並將this轉為指向這個新物件。

所以當我們印出`dog3`，會看到`dog3`內部的屬性是傳入的參數`杜賓`及`嗷嗚`，因為在使用new關鍵字時，它已經幫我們把函式內部的this的指派給了新物件，並將傳入的第一個參數賦值給這個新物件的`name`屬性、第二個參數賦值給`bark`屬性。

但是如果出現下面這種情況，this會是什麼呢?

```
// new + return
function Cat(name, bark){
    this.name = name;
    this.bark = bark;
    return '阿瑪是隻貓';
}
var cat1 = new Cat('阿瑪', 'meow!!!');
console.log(cat1);
```

————————————————————————————--我是防雷線
————————————————————————————

————————————————————————————--我是防雷線
————————————————————————————

————————————————————————————--我是防雷線
————————————————————————————

```
> // new + return
  function Cat(name, bark){
      this.name = name;
      this.bark = bark;
      return '阿瑪是隻貓';
  }
  var cat1 = new Cat('阿瑪', 'meow!!!');
  console.log(cat1);
  ▶ Cat {name: "阿瑪", bark: "meow!!!"}
```

答案是物件`Cat {name: "阿瑪", bark: "meow!!!"}`而不是return的字串`"阿瑪是隻貓"`。

依據我的理解，一旦new關鍵字出現，會做以下幾件事：

1. 產生空物件 {} 並賦值給obj
2. 將原本指向window的this改為指向新物件obj
3. 如果原本的建構式有設定prototype，則將obj的_ _ proto_ _屬性指向建構式的prototype

   JS Prototype

4. 立即執行建構式函式function
5. 將建構式內的屬性及方法給obj
6. 執行完後立即回傳物件

所以依照上面的邏輯來看，回傳的會是以物件為主，當return的值不是物件時，就會回傳被new出的物件。

那麼再來看下面這個例子：

```
function Cat(name, bark){
    this.name = name;
    this.bark = bark;
    return {
        name: '柚子',
        bark: 'meow~'
    };
}
var cat1 = new Cat('阿瑪', 'meow!!!');
console.log(cat1);
```

————————————————————————————--我是防雷線
————————————————————————————

————————————————————————————--我是防雷線
————————————————————————————

————————————————————————————--我是防雷線
————————————————————————————

答案是`{name: "柚子", bark: "meow~"}`。還記得剛剛說的，因為new關鍵字是會先建立一個空物件，而當return回傳的也是物件時，便會以return的物件為主。

總結一下，當出現new關鍵字時回傳物件的優先順：

> 建構式return為物件 > 建構式沒有return > 建構式return非物件

了解new關鍵字的運作後，對於this指向哪一個物件就不會有混淆的情況。

那麼你可能會想說，如果回傳的是陣列呢?

```
function Cat(name, bark){
    this.name = name;
    this.bark = bark;
    return ['柚子', 'meow~'];
}

var cat2 = new Cat('浣腸', 'ZZZ......');
console.log(cat2);
```

很明顯cat2物件會是一個陣列`['柚子', 'meow~']`，因為typeof [ ]會是object，所以最終得到的會是陣列。

```
> console.log(typeof []);
  object
```

```
> function Cat(name, bark){
      this.name = name;
      this.bark = bark;
      return ['柚子', 'meow~'];
  }

  var cat2 = new Cat('浣腸', 'ZZZ......');
  console.log(cat2);
  console.log(typeof []);
  ▼(2) ["柚子", "meow~"] i
     0: "柚子"
     1: "meow~"
     length: 2
    ▶__proto__: Array(0)
  object
```

當然到目前為止，已經可以應付大部分的this（不包括箭頭函式的this），然而不同的情況是用不同的規則，這裡有一個小撇步能快速地幫你知道this的值是什麼：

> 💡 可以把所有的function call都轉成利用`call`的形式來看

規則是，是什麼再呼叫function，就把它放到call的括號內。

以這個例子來說：

```
'use strict'
var cat = {
    name: '招弟',
    saySomething: function(){
        console.log('我是' + this.name);
    }
}

cat.saySomething();
var cat2 = cat.saySomething;
cat2();
```

轉成`call`的型式來看會變成下面這樣：

```
'use strict'
var cat = {
```

```
    name: '招弟',
    saySomething: function(){
        console.log('我是' + this.name);
    }
}

// 原始
cat.saySomething();
// 轉成call
cat.saySomething().call(cat); // 印出結果：我是招弟

// 原始
var cat2 = cat.saySomething;
cat2();
// 轉成call
cat2.call(undefined);
```

因為cat2是在全域環境下被呼叫的，而全域環境在嚴格模式下，this的值會是`undefined`
所以cat2( )轉成`call`型式後會變成由`undefined`呼叫。

到這邊為止判斷大部分JS中的this的值應該都沒問題囉!

參考資料:

- 淺談 JavaScript 頭號難題 this：絕對不完整，但保證好懂
- JavaScript繼承機制的設計思想
- 彥誠ㄉㄉ的講義