# this

在其他以類別為基礎的程式語言中， this 指的是目前使用類別進行實體化的物件。而JavaScript語言中因為在設計上並不是以類別為基礎的物件導向，設計上不一樣，所以 this 的指向的是目前呼叫函式或方法的擁有者(owner)物件，也就是說它與函式如何被呼叫或調用有關，雖然是同一函式的呼叫，因為不同的物件呼叫，也有可能是不同的 this 值。

## 函式的呼叫

我們在"函式與作用域"、"物件"與"原型物件導向"的章節中，都有看到函式的一些說明內容，也有看到用函式作為建構式來作物件實體化的工作，這時候會看到以 this 的一些說明，那麼在不是用來當作建構式的函式中，就是我們所認知的一般函式，裡面也有 this 嗎？有的，不論是在物件中的方法，或是一般的函式，每個函式中都有 this 值。以下是一個很簡單的範例，一個是我們所認知的普通函式，一個是在物件中的方法:

```
function func(param1, param2){
  console.log(this)
}

const objA = {
  test(){
    console.log(this)
  }
}

func() //undefined
objA.test() //Object(objA)
```

func 函式在呼叫時的 this 值是 undefined ，原本它應該會回傳全域物件，在瀏覽器中就是window物件，這是因為babel預設會開啟strict mode(嚴格模式)，為了安全性的理由，原本的全域物件變成了 undefined ，以下的內容也是用這樣的作法。

objA.test 方法在呼叫時的 this 值就是 objA 物件本身，這一點不難理解。用以下的範例可以檢查 this 和 objA 是不是相同。

```
const objA = {
  test(){
    console.log(this)
    console.log(this === objA) //true
    console.log(objA)
  }
}

objA.test()
```

不過這裡有個小地方會讓你覺得很不可思議的是， ObjA 物件中的方法竟然可以讀取到 ObjA 物件？

是的，實際上它不止物件中可以讀取到物件本身，物件中的方法還可以執行自己本身的方法，下面的程式碼還可以讓你的瀏覽器無止盡的執行，然後當掉:

```
//注意: 瀏覽器有可能會當掉
const objA = {
  test(){
    objA.test()
  }
}
```

在函式中也可以這樣作，也是會讓你的瀏覽器最後當掉，這是都是錯誤的示範:

```
//注意: 瀏覽器有可能會當掉
function func(param1, param2){
  func()
}

func()
```

以上算是題外話，不過這部份在後面可以簡單的說明為什麼可以這樣作。我們先關心幾個重要的議題。

# 深入 函式 中

> 所有的函式在呼叫時，其實都有一個擁有者物件來進行呼叫。所以你可以說，其實所有函式都是物件中的"方法"。所有的函式執行都是以Object.method()方式呼叫。

關於這一點，下面的範例就可以說明一切了，有個全域物件(在瀏覽器中是window物件)是所有函式在呼叫時預設物件，下面三種函式呼叫都是同樣的作用:

```
function func(param){
  console.log(this)
}

window.func() //只能在不是strict mode下執行
this.func()  //只能在不是strict mode下執行
func()
```

> 對this值來說，它根本不關心函式是在哪裡定義或是怎麼定義的，它只關心是誰呼叫了它。

在JavaScript中函式是一個很奇妙的東西，它的確是一個物件類型，又不太像是一般的物件，以 `typeof` 的回傳值來說，它回傳的是 `function` ，代表擁有獨立的回傳類型值。在函式物件的API定義中，它比一般物件多了幾個特別的屬性與方法，其中最特別的是以下這三個，我把它們的定義寫出來:

- call(呼叫): 以個別提供的 `this` 值與傳入參數值來呼叫函式。
- bind(綁定): 建立一個新的函式，這個新函式在呼叫時，會以提供的this值與一連串的傳入參數值來進行呼叫。
- apply(應用): 與call方法功能一樣，只是除了this值傳入外，另一個傳入參數值使用陣列。

那麼，這個 `call` 方法與直接使用一般的程式呼叫方式來執行函式，例如 `func()` 有何不同？

基本上完全一樣，除了它在參數裡可以傳入一個物件，讓你可以轉換函式原本的上下文(context)到新的物件之前。(註: Context的說明在下面)

`call` 方法可以把函式的定義與呼叫拆成兩件事來作，定義是定義，呼叫是呼叫。以下為一個範例:

```
function func(param1){
  console.log('func', this)
}

const objA = {
  methodA(){
    console.log('objA methodA', this)
  }
}

const objB = { a:1, b:2 }

func.call(objB) //func Object {a: 1, b: 2}
objA.methodA.call(objB) //objA methodA Object {a: 1, b: 2}
```

這種現實讓你對函式的印象崩壞，不論是一般的 `func` 呼叫，或是位於物件 `objA` 中的方法 `methodA` ，使用了call方法後，竟然 `this` 值就會變成 `call` 中的第一個傳入參數值，也就是物件 `objB` 。~~有種辛辛苦苦養大的小孩，竟然被認賊作父的心情。~~

`bind` 方法更是厲害，它會從原有的函式或方法定義，產生一個新的方法。為了展示它的厲害之處，函式加了兩個傳入參數，下面是函式部份:

```
function funcA(param1, param2){
  console.log(this, param1, param2)
}

const objB = { a: 1, b: 2 }

funcA() //undefined undefined undefined
```

```
const funcB = funcA.bind(objB, objB.a)

funcB() //Object {a: 1, b: 2} 1 undefined
funcB(objB.b) //Object {a: 1, b: 2} 1 2
```

這是物件中的方法定義的範例，這和上面沒什麼兩樣，只是函式定義在物件objA之中而已:

```
const objA = {
  methodA(param1, param2){
    console.log('objA methodA', this, param1, param2)
  }
}

const objB = { a: 1, b: 2 }

objA.methodA()

const methodB = objA.methodA.bind(objB, objB.a)
methodB()
methodB(objB.b)
```

不過因為用了物件，應該要讓方法可以直接使用物件中的屬性才是妥善利用，另一種範例如下:

```
const objA = {
  a: 8,
  b: 7,
  methodA(){
    console.log(this, this.a, this.b)
  }
}

const objB = { a: 1, b: 2 }

objA.methodA() //Object {a: 8, b: 7} 8 7

const methodB = objA.methodA.bind(objB, objB.a)

methodB() //Object {a: 1, b: 2} 1 2
methodB(objB.b) //Object {a: 1, b: 2} 1 2
```

從上面的例子中，可以看到這個 `bind` 方法可以用原有的函式，產生一個稱為部份套用(Partially applied)的新函式，也就是對原有的函式的傳入參數值固定住部份傳入參數的值(從左邊開始算)。這是一種很特別的特性，有一些應用情況會用到它。

最後用下面這個例子來總結，什麼叫作"函式定義是定義，呼叫是呼叫"，實際上在物件定義的所謂方法，你可以把它當作，只是讓程式設計師方便集中管理的函式定義而已。

```
const objA = {a:1}

const objB = {
  a: 10,
  methodB(){
    console.log(this)
  }

}

const funcA = objB.methodB

objB.methodB() //objB
funcA() //undefined，也就是全域物件window
objB.methodB.call(objA) //objA
```

> 註: function call與function invoke(invocation)是同意義字詞

## this值是何時產生的？

函式呼叫執行時產生。

當函式被呼叫(call/invoke)時，有個新物件會被建立，裡面會包含一些資訊，例如傳入的參數值是什麼、函式是如何被呼叫的、函式是被誰呼叫的等等。這個物件裡面有個主要的屬性this參照值，指向呼叫這個函式的物件。不同的函式被呼叫時，this值就會不同。

### this值的產生規則是什麼？

this值會遵守ECMAScript標準中所定義的一些基本規則，大概摘要如下，函式中的 `this` 值按順序一一檢視，只會符合其一種結果(if...else語句):

1. 當使用strict code(嚴格模式程式碼)時，直接設定為call方法裡面的thisArg(this參數值)。
2. 當thisArg(this參數值)是null或undefined時，會綁定為全域(global)物件。
3. 當thisArg(this參數值)的類型不是物件類型時，會綁定為轉為物件類型的值。
4. 都不是以上的情況時，綁定為thisArg(this參數值)。

第1點就明確的說明了，為什麼使用strict mode(嚴格模式)後，在全域的函式呼叫執行，this值一定都是 `undefined` ，因為在call中根本沒傳入thisArg值。除非關閉strict mode(嚴格模式)才會變為第2點的全域window物件。

### Context是什麼？

`Context` 這個字詞是不易理解的，在英文裡有上下文、環境的意思，什麼叫作"上下文"？這中文翻譯也是有看沒有懂。還記得在國高中英文課的時候，英文老師有說過，有些英文字詞的意思需要用"上下文"來推敲才知道它的意思，為什麼要這樣作？老師一定沒有把原因說得很清楚，第一個原因是英文單字你學得不夠多，很多時候考試試題中的英文單字通常你都沒讀到，所以只好猜猜看(~~這個原因只是個笑話而已~~)。第二個原因是，英文字詞很多時候同一個字詞有很多種意思，有時候用於動詞與名詞是兩碼子事，舉個例子來說，"book"這個英文單字，你用腳底板不需經過大腦，第一時間就會說它是"書"的意思，幼稚園就學過了，但是你忘了那是用於當作名詞的情況，用於動詞是"預訂"的意思。

在程式語言中的 `Context` 指的是物件的環境之中，也就是處於物件所能提供的資料組合中，這個 `Context` 是由 `this` 值來提供。

再用白話一點的講法，來看函式與 `this` 的關係， `this` 是魔法師的角色，而函式是要施展的魔法，魔法的強度或破壞力，會依施法者的資質與能力有所不同，魔法在施展中會運用到施法者的本身的資質(智慧、MP、熟練度...等等)的這整體的素質特性，這就是所謂的 `Context` 了。

> 註: `Context` 與常會看到的另一個名詞 `Execution Context` (執行上下文)的意義是不同的，以下會有說明。

## 四種函式呼叫樣式(invocation pattern)

函式的呼叫樣式共有四種，在本書中已經看到過這四種了，這裡只是集中整理而已:

- 一般的函式呼叫(Function Invocation Pattern)
- 物件中的方法呼叫(Method Invocation Pattern)
- 建構函式呼叫(Constructor Invocation Pattern)
- 使用apply, call, bind方法呼叫(Apply invocation pattern或Indirect Invocation Pattern)

其中的建構函式呼叫，就是使用new運算符來進行物件實體化的一種函式呼叫樣式，請參考"物件"與"原型物件導向"的章節內容。

## Scope vs Context

Scope(作用域, 作用範圍)指的是在函式中變數(常數)的可使用範圍，JavaScript使用的是靜態或詞法的(lexical)作用域，意思是說作用域在函式定義時就已經決定了。JavaScript中只有兩種的Scope(作用域)，全域(global)與函式(function)。

Context(上下文)指的是函式在被呼叫執行時，所處的物件環境。上面已經有很詳細的解說了。這兩個東西雖然都與函式有關，但是是不一樣概念的東西。

> Scope通常被稱為Variable Scope(變數作用域)，意思是"作用域代表變數存取的範圍"。

> Context通常被稱為this Context，意思是"由this值所代表的上下文"。

## 執行上下文(Execution Context, EC)

執行上下文(Execution Context)看起來與上下文(Context)很像，但是它們是不同的概念。這不單只是我們以中文為主的開發者常常會搞混，其實像這麼像的名詞，以英文為主的開發者也很難理解的清楚。執行上下文(EC)的概念已經涉及JavaScript語言的執行底層設計，有很多艱

澀的專有名詞會在這裡一一出現，以下的說明都是用比較簡單的方式來解說，專業的內容可以參考網路上的其他文章。

JavaScript語言中使用執行上下文(EC)的抽象概念，來說明程式是如何被執行的，你可以把執行上下文當成是用來區分可執行程式碼用的，在標準中並沒有明確規定它應該是一個長什麼樣的結構，所以我把它稱之為一種結構。

> 所有的JavaScript程式碼都是在某個執行上下文中被執行。

程式碼會以執行上下文(EC)來區分為三個類型，也就是全域、函式呼叫，以及eval。

- 全域程式碼: 在全域環境下的程式碼，也就是要直接執行的程式碼，會在"全域執行上下文(EC)"中被執行。
- 函式程式碼: 每個函式的呼叫執行，都會有關聯這個函式的執行上下文(EC)。
- eval程式碼: 使用內建eval方法中傳入的程式碼，因為eval是JavaScript中設計很糟糕的一個方法，根本不會被使用，所以就不多加討論。

一個執行上下文(EC)的結構中會包含三個東西，但這只是概念上的內容:

- Variable object(變數物件，簡稱VO): 集合執行上下文會用到的變數資料與函式定義。
- Scope chain(作用域鏈，或作用域連鎖): 上層VO與自己的VO形成的作用域連鎖。
- this值: 上面有說過了

不過，當函式呼叫時的執行上下文，因為還需要包含傳入的參數值，以及那個設計相當有問題的隱藏"偽"陣列物件 - arguments物件，所以又多了一個新名詞叫Activation object(啓動物件，簡稱AO)，AO除了上面說的VO定義外，又會多包含了剛說的參數值與arguments物件。所以在函式呼叫的執行上下文，AO會用來扮演VO的角色。

Scope chain(作用域鏈)的設計概念與Prototype chain(原型鏈)非常相似，如果你有認真看過"原型物件導向"那個章節的內容，大概心中就有個底了。以函式執行上下文來說，AO裡面會有一個屬性，用來指向上一層(父母層)的AO，這個鏈結會一直串到全域的VO上。函式執行時，尋找變數時會用作用域鏈尋找。

Scope chain(作用域鏈)的概念，在實際使用上，會出現在函式中的函式(內部函式，子母函式)結構的情況，也就是JavaScript語言中強大但也是不易理解的其中一個特性 - 閉包(Closure) 的結構之中。這也是為何內部函式可以存取得到外部函式的作用域的原理。

> 註: 仔細回想，這種JavaScript語言中"強大但也是不易理解"的特性實在有夠多。

最後一點，在"事件迴圈"章節中所說的呼叫堆疊(call stack)，實際上就是由執行上下文集合而成的結構，你也可把它叫作執行上下文堆疊(Execution Context Stack)或執行堆疊(Excution Stack)。在呼叫堆疊的最下層一定是全域執行上下文。

# this的分界

當函式被呼叫執行時，this值隨之產生，那如果是函式中的函式呢？像下面這樣的巢狀或內部函式的結構:

```
const obj = {a:1}

function outter() {

  function inner(){
    console.log(this)
  }

  inner()
}

outter.call(obj) //undefined
```

結果是 `undefined` ，內部的 `inner` 函式不知道 `this` 值是什麼呢，為什麼？因為執行上下文是以函式呼叫作為區分，所以 `this` 值在不同的函式呼叫時，預設上就會不同。這稱之為 `this` 或 `Context` 的分界。

解決方式是要利用作用域鏈(Scope Chain)的設計，也就是說，雖然inner函式與外面的outter分屬不同函式，但inner函式具有存取得到outter函式的作用域的能力，所以可以用這樣的解決方法:

```
const obj = {a:1}

function outter() {
  //暫存outter的this值
  const that = this
```

```
  function inner(){
    console.log(that) //用作用域鏈讀取outter中的that值
  }

  inner()
}

outter.call(obj) //Object {a: 1}
```

that 是一個隨你高興的變數(常數)名稱，這並不是什麼特殊的關鍵字或保留字，也有人喜歡取 self 或 _that 。它只是為了暫時保存在outter函式被呼叫時的 this 值用的，讓 this 可以傳遞到inner函式之中。

第二種寫法其實也是同樣的概念，只不過用了call來呼叫，outter函式在呼叫時，它裡面是有 this 值的，因此可以當作call的傳入參數值，這範例與上面相同，也是有同樣作用:

```
const obj = {a:1}

function outter() {

  function inner(){
    console.log(this)
  }

  inner.call(this) //用outter中的this值來呼叫內部函式的inner
}

outter.call(obj) //Object {a: 1}
```

第三種寫法是用 bind 方法，不過因為 bind 方法會回傳新的函式，函式宣告要變成用函式表達式(FE)的方法才行:

```
const obj = {a:1}

function outter() {

   const inner = function(){
    console.log(this)
  }.bind(this)

  inner()
}

outter.call(obj) //Object {a: 1}
```

那麼在callback(回調)的情況下又是如何？this能順利傳到callback(回調)函式之中嗎？像下面的範例這樣，結果當然是不行:

```
const obj = {a:1}

function funcCb(x, cb){
  cb(x)
}

const callback = function(x){ console.log(this) }

funcCb.call(obj, 1, callback) //undefined
```

實際上傳入參數值這個東西，如果是函式的話，都是位於全域物件之下的，這callback(回調)在呼叫時的 this 值就是全域物件。用call或bind方法就可以解決這個問題:

```
const obj = {a:1}

function funcCb(x, cb){
  cb.call(this, x)
}

const callback = function(){ console.log(this) }
```

```
funcCb.call(obj, 1, callback) //Object {a: 1}
```

更進階的一種情況，使用例如像 `setTimeout` 方法，裡面帶有callback(回調)函式的傳入參數，像下面這樣的程式碼:

```
const obj = {a:1}

function func(){
  setTimeout(
    function(){
      console.log(this)
    }, 2000)
}

func.call(obj) //window物件
```

這也是運用上面類似的幾種作法，其一，用一個函式內的變數(常數)來傳遞 `this` 值:

```
const obj = {a:1}

function func(){
  const that = this

  setTimeout(
    function(){
      console.log(that)
    }, 2000)
}

func.call(obj) //Object {a: 1}
```

其二，直接用bind方法(因為這裡不適合使用call方法)

```
const obj = {a:1}

function func(){

  setTimeout(
    function(){
      console.log(this)
    }.bind(this), 2000)
}

func.call(obj)
```

或是寫得更清楚點，把其中的callback函式獨立出來:

```
const obj = {a:1}

function func(){

  function cb(){
    console.log(this)
  }

  setTimeout(cb.bind(this), 2000)
}

func.call(obj)
```

下面愈寫愈長，其實沒有那麼必要，只是說明也可以用另一個已經bind好的函式，傳入當作新的callback(回調)函式:

```
function func(){

  function cb(){
    console.log(this)
```

```
  }

  const cb2 = cb.bind(this)

  setTimeout(cb2, 2000)
}

func.call(obj)
```

## 箭頭函式綁定this

箭頭函式(Arrow Function)除了作為以函式表達式(FE)的方式來宣告函式之外，它還有一個特別的作用，即是可以作綁定(bind)this值的功能。這特性在一些應用情況下非常有用，可以讓程式碼有更高的可閱讀性，例如以上一節的例子中，有使用到bind方法的，都可以用箭頭函式來取代，以下為改寫過的範例，你可以比對一下:

```
const obj = {a:1}

function func(){
  setTimeout( () => { console.log(this) }, 2000)
}

func.call(obj)
```

## 在物件與陣列中函式的 `this`

按照上面章節的說明，在一般情況，也就是在全作用域中定義中的函式，它的this值預設是在瀏覽器中是window物件，在Node.js中是global物件，也就是全域(全局)，但要注意這只能適用在非嚴格模式(non-strict mode)的情況。

那如果是一般使用建構函式(或類別定義)所建立出來的物件實體，因為這是使用了 `new` 關鍵字進行實體化，符合了建構函式呼叫樣式(Constructor Invocation Pattern)，其中的成員的 `this` 值將自動指向新建立的物件實體。這在類別或物件的章節中都可以看到其中的內容，就不再多作描述。

但陣列中的函式，實際上它也是像這樣的物件實體，因為設計上陣列實際是個物件類型。所以像下面這樣的例子:

```
const a = []
const func = function(){ console.log(this) }

a.push(func)
a[0]() //指向a
```

你會發現如果對在陣列中的成員函式呼叫，它的 `this` 值將指回陣列本身，也就是 `a` 。這是因為陣列的 `const a = []` 的宣告，相當於 `const a = new Array()` ，陣列是個物件類型，正確來說是陣列是以原始的物件類型為基礎，再發展延伸出來的特殊物件類型。

這對應了不論是使用物件字面方式來定義的物件方式，或是使用 `new Object()` 來實體化物件的定義方式，兩種獲得的結果都是相同的。如下面的例子:

```
const e = { func: function(){ console.log(this)} }
e.func() //指向e

const f = new Object()
f.func = function(){ console.log(this) }
f.func() //指向f
```

有一個特別的實例是函式中的隱藏物件 - `arguments` 物件，它在實際上也隱藏了實體化的過程，由JavaScript自動實體化這個物件。因此，對應上面的陣列與物件的設計，預設裡面的成員如果是個函式是也是自動指向自己本身。如下面的例子:

```
function func(fn){

  arguments[0]() //指向arguments本身

  fn() //指向window(全域物件)
```

```
}

func( function(){console.log(this)} )
```

不過用 `arguments` 物件容易產生誤解，這也是其中一個常見的陷阱，在許多實際使用的情況都容易造成誤用，例如上面的例子中，直接呼叫傳入的函式與使用arguments物件來呼叫，結果是不同的， `this` 值是不同的。我建議你不要再使用 `arguments` 物件，它是一個相當有問題的設計。

> 特別注意: 在JavaScript中，不要使用 `new Array()` 或 `new Object()` 來建立陣列與物件的實體(實例)，原因是不需要，而且它們容易造成誤用。取而代之的是應該使用 `[]` 或 `{}` 的定義方式。

# 小小問題的解答

我們要回答最上面一開始發現的一個小問題，就是為什麼函式中呼叫自己本身是可以的，有個主要原因:

> 每個函式呼叫都是一個獨立的執行上下文

也就是像下面這樣的範例中:

```
function outter(){
  function inner(){
    console.log('inner')
  }
  inner()
}

outter()
```

它在call stack(呼叫堆疊)的結構是分成兩個獨立的執行上下文，類似像下面這樣:

```
ECStack = [
  <inner> functionContext
  <outter> functionContext
  globalContext
];
```

依照堆疊的執行順序，會從上面先開始執行，然後再來往下執行，也就是後進先出(LIFO, Last in First out)。所有如果是自己呼叫自己的迴圈，call stack(呼叫堆疊)會變成像下面這樣，無窮的執行上下文遞迴下去，最後造成瀏覽器當掉:

```
ECStack = [
  <func> functionContext - 遞迴
  <func> functionContext
  globalContext
];
```

# 結語

本章說明了很多JavaScript底層實作的技術，不過只是簡單的介紹而已。 `this` 的概念因為涉及很多底層的設計，所以造成很多初學者非常容易搞混與誤解，相信在讀過這章的內容後，你可以對JavaScript語言中， `this` 所扮演的角色，有更清楚或更深入的理解。在往後的開發日子中，可以更正確而且靈活的操控 `this` 的各種用法。在參考資料中有更多深入的內容等待你進一步去研究。

# 參考資料


- What is the Execution Context & Stack in JavaScript?
- ECMA-262-3 in detail. Chapter 1. Execution Contexts
- How to access the correct `this` / context inside a callback?
- How does the “this” keyword work?
- JavaScript function invocation and this (with examples)