從ES6開始的JavaScript學習生活

Start JavaScript learning life from ES6

GitBook

Eddy Chang

目錄

關於本書

關於本書

本書尚在撰寫中，有些章節尚未全部完成

這是一本JavaScript程式語言的中文入門書籍。市面上或網路上，有很多的JavaScript的入門書籍，或是教學文章，與它們所不同的比較點如下:

- 符合中文使用者學習的使用範例
- 以ECMAScript 6(ES6, ES2015)標準爲基礎寫成
- 加入了以中文拼音英文字詞的方式，強調程式語言中的英文
- 加入風格樣式指引(主要爲Airbnb所製作)的提示
- 儘可能涵蓋了大部份常使用到的JavaScript語法與用法

如果你對JavaScript程式語言有興趣，你可以參考這本書，本書的閱讀對象並沒有預設任何基礎。

本書不包含什麼

- jQuery或其他外部函式庫的語法
- Node.js的API或設計
- 屬於各別瀏覽器的自有或非標準設計
- 低於ES5支援的瀏覽器相容內容(IE8以下)

授權規定

本電子書的使用的授權規定如下，如果有其他問題請再連絡作者:

序言

序言

本書的撰寫構想是在很多年之前，或許JavaScript程式語言在當時並不是像今天這麼的流行，但問題仍然是差不多的。或許你有聽過，JavaScript是一個容易被誤解的程式語言，那麼，你可能會好奇到底是什麼地方被誤解了？這本書或許可以提供你一些解答。

其次是在市面上的JavaScript的書籍非常多，尤其是入門的書籍更多，網路上也有很多入門教學的文章，那與這本電子書的差異會在什麼地方？

我認爲第一個差異在於ES6標準，實際上大概二、三年前的舊書或舊教學可以不用再看了，未來這幾年將會是新式的JavaScript標準ES6(ES2015)的時代，現在學習新的技術才能搭上未來的這股潮流，對於已經熟悉JavaScript程式語言的開發者，也需要花點時間學習一下新式的ES6標準的用法，因爲改變的幅度超過你能想像的。市面上已經開始有不少ES6標準的中文書籍，你也可以參考一下。

第二個差異在於我理想中的入門書，是不需要涵蓋所有的內容的，只需要先教"基本"的知識就行了，ES6中有很多進階的新特性，但在這本書中並沒有包含這些。也就是說這本書只挑選在ES6中，對於剛入門學習者所需要的知識部份而已。不過，對於何爲"基本知識"的理解，每個作者是不同的。我所希望針對的讀者對象，是對程式語言完全不熟悉或沒有經驗的，可以將JavaScript作爲基礎的程式語言來學習。

第三個差異是在於英文。我們知道所有的程式語言，不論它的程式碼或是手冊，都是由英文作爲主要語言來撰寫。英文對使用中文的程式入門學習者來說，是一個門檻。在我教課的經驗中，大部份的學生英文能力都不好，英讀與英聽更是普遍都不好。程式開發使用的英文與一般我們在學校英文課堂上的英文有很大的不同，這在這本書中希望用另一種方式來學習。當然，這也免不了需要背誦一些單字，或解釋一些英文字句。我希望能透過英文字詞的理解，能更深入理解程式語言本身的設計內涵。

第四個差異在於習慣，也就是撰寫程式碼的風格樣式或習慣。過去這部份在程式開發的教學中經常會被忽略，但這幾年卻非常的盛行。JavaScript是一個鬆散的程式語言，同一件事可能有很多種不同的寫法，也很容易被誤用或用了不建議的寫法。其實很多流行的直譯式腳本語言，都有類似的問題，不好的撰寫習慣，除了會導致嚴重的可閱讀性低外，甚至會很容易造成程式潛在的問題、除錯上的困難等等。對於初學者來說，一開始就學習使用功能強大的開發工具，並沒有太大用處，反而學習好的開發習慣，才是必要的。

> 優良的撰寫程式碼習慣，比再棒的程式語言或開發工具還重要

對於讀者來說可能很重要的一點是，作者本身對於JavaScript程式語言到底是有多理解，才能來寫這本書，然後來教讀者們？我從10多年前在學校作的第一個網站時，就已經開始使用JavaScript程式語言。多年來，網站應用程式是我個人所熟悉的技術與工作，自然經常會接觸到。不過，這並不代表我對每種技術都很熟悉，我只比較熟悉我有使用過的部份，或是我有興趣研究的部份。

實際上，我認爲沒有任何一個人可以對任何一種程式語言的每個細節部份，都可以說他很熟悉。尤其在今天開放原始碼相當盛行的時代，所有的程式語言、軟體技術都不斷的進步當中，每個軟體其背後都有很多技術強，而且還投入相當多時間與心力的程式設計師，絕非一人所爲。所以沒有一本書或是某個人，可以通曉所有內部包含的技術，技術需要不斷的精進，對初學者是這樣，對已經熟悉的開發者也是如此。

實際動手作遠比學得再多重要

當然，學習一個程式語言並不是說買本書讀完就叫作學會，或是把整本書背起來，就像你今天買了本如何學騎腳踏車的書，看過後就馬上會騎腳踏車是不可能作得到的。學習新的知識就像吃東西一樣，都需要時間消化。不可能短時間一直吃同樣的東西，就算是美食當前也會覺得噁心。每個人的學習能力不同，需要消化(理解知識)的時間與方式也不同。千萬不需要心急於一時，就想要學會所有的東西，學習只是一點一滴持之以恆的累積成果。

寫作格式

寫作格式

本書使用了以下的風格作爲撰寫風格:

專有名詞

對於專有名詞，例如函式庫中的定義、函式、方法、物件等專用名詞，保留原字詞不進行"取代式"的翻譯，而使用括號(())加註其後作爲補充翻譯，例如:

```
state(狀態)
props(屬性)
```

符號

提供的符號，儘可能使用括號(())加註符號在字句中。例如:

單行註解使用雙斜線符號(//)。

中文字詞與符號

可以使用全形的逗號(，)、頓號(、)與句號(。)，但儘量不使用全形的括號(（）)、框號(「」)、分號(；)與其他符號等。而是使用半形符號。

適當的在中文字詞的前後加上空白字元，作爲明顯的區隔出這個字詞。例如:

使用 擴充套件->管理 進入這個管理的頁面。

不使用分號作爲每行程式碼的結尾

本書的所有程式碼，每行後面都"不"再使用分號(semicolon)(;)作爲程式碼段落之用，程式碼只需要記得有分行就行了。

有許多現代新式的程式語言，也是不需要用分號(;)來作每行程式碼的結尾，例如Swift、Python或Ruby。而實際上，Javascript現在也可以不需要用分號(;)來分行，有許多知名的函式庫例如npm、redux都已經採用這個撰寫風格。所以，我們鼓勵所有的Javascript程式設計師，使用這種方式，讓程式碼看起來會更加簡潔外，也理解爲何可以不使用分號(;)。

更多參考資訊:

- Semicolons in JavaScript are optional
- An Open Letter to JavaScript Leaders Regarding Semicolons

讀音(用中文拼音)

爲了方便初學者能很容易的唸出英文字詞的讀音，本書使用"以中文讀音拼出英文字詞"的作法。這是參考日本使用五十音來拼英文字詞的作法，方法好不好或對不對見仁見智。拼音只針對重要字詞，以前後斜線(/)符號含括中文拼音，例如:

`const/康死/` 是 `constant/康死撐/` 的縮寫，中譯是"常數"的意思。

英文字詞解說

針對重要的專業字詞，提供英文解說，並加入相關的字詞說明。如果可以的話，提供有趣的範例加深學習印象。

風格指引

這是參考目前網路上知名的數個Javascript程式碼撰寫風格指引，提供在每個段落相關的建議。不過有些風格指引是針對ES5的，僅作參考而已，還是會以章節內容爲主。目前參考的風格樣式指引有以下幾個:

- Airbnb JavaScript Style Guide
- Principles of Writing Consistent, Idiomatic JavaScript
- Google JavaScript Style Guide
- Code Conventions for the JavaScript Programming Language

其他參考

- 翻譯指引 Translation Guide

更新日誌

更新日誌

請參考github中的更新說明。

160606

- 開始上傳電子書。

問答與回饋

問答與回饋

本書的儲存庫使用github，如果對本書的內容有疑問、建議或有寫錯的地方，請至以下的連結中提交:

- 使用Github的ISSUE
- 使用GitBook的討論(Discussions)功能

在學習的過程中，如果你有任何的問題，可以用以下的方式發問或尋求協助:

- 每篇章節的最底下都有Disqus留言功能
- 使用GitBook的討論(Discussions)功能
- 使用GitBook的文章段落內嵌留言功能
- 作者的部落格&Email: eddychang.me

介紹

歷史

歷史

爲何要了解歷史？因爲有過去的歷史，才有現在的JavaScript程式語言。

而且，學習程式語言是件滿枯燥無聊的事情，累或煩的時候可以看看這段不用花太多腦袋的故事。

在1995年時，由當時的Netscape(網景)公司的Brendan Eich(布蘭登·艾克)，在網景領航員瀏覽器上首次設計實作而成(據說完成第一個版本，他只花了10天...)。因爲Netscape(網景)公司與Sun(昇陽)公司合作，網景公司管理階層希望它像Java這麼拉風，因此取名爲JavaScript，雖然它與Java程式語言並沒太大關係。JavaScript的名稱所有權是屬於Sun(昇陽)公司，後來變爲Oracle(甲骨文)公司。

Microsoft(微軟)公司在1996年發佈了類似的網頁的直譯式Script(腳本)語言VBScript與JScript，其中JScript即是模仿JavaScript程式語言的產品，在Internet Explorer 3開始提供這兩個語言的支援。

爲了統一JavaScript的標準，以及對抗Microsoft(微軟)公司的競爭，Netscape(網景)公司於1996年向ECMA International歐洲國際標準組織，提交了JavaScript，希望它能成爲業界的標準，在1997 年的時候產生了初版的ECMAScript草案，自此ECMAScript成爲JavaScript程式語言的標準規範。

隨著Microsoft(微軟)公司在早期的瀏覽器大戰中取得了大部份的市場佔有率，1998年，Netscape(網景)公司被美國線上(AOL)收購，以及Netscape(網景)公司開放了瀏覽器與相關產品的原始碼，成爲現在的Mozilla基金會的Firefox瀏覽器與其他產品的基礎。雖然在2003年Microsoft(微軟)公司因壟斷敗訟，賠償美國線上(AOL)公司數億美金，但被收購後的Netscape(網景)公司後繼無力，終於在2003年解散。此次戰役稱爲第一次瀏覽器大戰，Microsoft(微軟)公司的Internet Explorer瀏覽器獲得勝利，在2002達到90%以上的市場佔有率。而值得一提的，在這場戰役開始時，Netscape瀏覽器具有80%市場佔有率。

Microsoft(微軟)公司對於JavaScript程式語言，也是有很大的貢獻，它在2000年發佈的Internet Explorer 5中，發表了XMLHttpRequest (XHR)的第一個實作版本，誕生了後來的Ajax技術，這個技術在今天是JavaScript程式語言中非常重要的應用之一。但Microsoft(微軟)公司始終不是開放原始碼的擁護者，而且是敵對的一方，Microsoft(微軟)公司有許多技術都是不公開的專利技術與非開放標準。

第二次瀏覽器大戰約莫在2004年開始，首先由Mozilla基金會發表Firefox 1.0瀏覽器，另外一家歐洲的軟體公司的Opera瀏覽器，與Mozilla共同參與了推動瀏覽器相關規格的開放標準化。Mozilla基金會代表的是承襲自Netscape的技術血統，這次更加入了開放原始碼的力量。這場開放原始碼加上開放標準，對戰Microsoft(微軟)公司獨佔的市場大戰開始上演。此後，在2003年Apple(蘋果)公司加入戰局，發表分支於其他開放原始碼專案的WebKit渲染引擎，以及發佈Safari瀏覽器，除了支援Mac平台外，也跨足到Windows作業系統上的版本。Microsoft(微軟)公司則宣佈在2007年，停止在Mac平台上的IE支援。

2008年Google進軍瀏覽器市場，發佈Chrome瀏覽器，爲這場大戰投下了震撼彈。Chrome採用了WebKit渲染引擎與自行研發的更快速V8 JavaScript引擎，接著發佈了Mac、Windows與Linux上的多版本瀏覽器，以及名稱爲Chromium的開放原始碼專案。在這期間雖然Internet Explorer推出了改進的7、8版本，但眾多的瀏覽器品牌快速的侵蝕原本Internet Explorer的市場，使得在

2010年Internet Explorer首次下降到50%。Microsoft(微軟)公司的瀏覽器市場龍頭地位顯得岌岌可危。

在之後短短數年間的瀏覽器的大戰，戰場從桌上電腦延燒到了行動裝置之上，也就是智慧型手機與平板電腦上。Microsoft(微軟)公司在行動裝置上的發展受到很大的挫敗，反觀Google的Android作業系統獲得極大的成功，掌握大部份的行動將置作業系統市場佔有率，間接促進瀏覽器的佔有率。以全平台的市場佔有率來看，在2012年Internet Explorer與Chrome瀏覽器的市場佔有率趨勢呈現了交叉。自此Google Chrome瀏覽器獲得最多的市場佔有率，在2015年底時，約有50%左右的市場，而Internet Explorer、Firefox、Safari三者的市佔率相近，差不多是10%~15%之間。

Google Chrome瀏覽器中的V8 JavaScript 引擎，大幅度的提升了JavaScript程式碼在瀏覽器中執行的效能，由於它也是開放原始碼專案的產物，目前已被用於其他軟體的計劃之中，例如Node.js、 Couchbase、MongoDB等，促使了JavaScript在除了瀏覽器平台以外的應用領域，這是一大進步。

規格版本

ECMAScript是Javascript規格標準，以下是各版本的發行日期，但標準發行並不代表在各個瀏覽器品牌中，即可使用其中的功能規格，還需等待各瀏覽器品牌廠商進行實作。

- ECMAScript 6 (ES6) 發行於2015年中，爲目前最新的官方版本
- ECMAScript 5 (ES5) 發行於2009年底
- ECMAScript 4 (ES4) 棄用
- ECMAScript 3 (ES3) 發行於1999年底

發展

發展

ES6標準的未來

ES6標準在2015年確立後，雖然ES6中的新語法與API非常多，在今年(2016)已經有很多桌上電腦的瀏覽器品牌實作完幾乎所有的標準規格。相信在很快的將來，2017年幾乎是所有的平台上的瀏覽器，包含行動裝置、伺服器平台等等，都會實作完成。這代表你今天學習這些新式的語法與API，很快就可以使用在瀏覽器或伺服器平台上。

在ES6標準之前，有很多好用的方法或語法，都是要透過外部的函式庫，或是編譯工具，才能直接執行在瀏覽器上。當ES6標準正式上路後，你可以不需要再使用那些工具或函式庫了，更棒的是效能上因爲是瀏覽器的JavaScript引擎直接支援，效能會更好。

ES6也因爲它是個眞正的JavaScript標準，不管如何，很多大公司或開發週邊的廠商，是一定要支持的，畢竟這是他們其中很多人參與共同討論出來的結果。對於學習者來說，學習標準有很多好處，尤其像現在各家函式庫或框架，發展很快而且各自競爭不相容得很嚴重，有些函式庫或框架還會使用很多更進階的超集語言，例如TypeScript、CoffeeScript等等，學習門檻拉得更高。整體來說，我認爲學習ES6標準的一些好處如下:

- 不需要依靠外部函式庫(尤其是工具型的函式庫)，就可以作一樣的事
- 新式的語法有許多非常棒的功能，甚至是其他函式庫與框架沒辦法作或作得不夠的，例如模組系統、箭頭函式、類別等等
- 未來有很多新的機會，例如把舊的程式碼重新改寫爲ES6標準的

行動裝置

要談到JavaScript語言用於行動裝置開發的發展，就不得不先說目前的兩大生態圈，一是以Facebook爲首的ReactJS陣營，ReactJS只是一套視圖用的函式庫，但加入相當創新的想法，使用類似的語法可以使用React Native專案開發手機App，是可以跨iOS與Android平台，React Native開發出來的App是Native(原生)的，它目前是新式的一種開發手機App方式。

另一個是以Google爲主的AngularJS陣營，它是一個完整的應用程式框架，另有一個專案是ionic，也是用來開發跨平台手機App，它還基於另一個專案Apache Cordova，這個專案原本就是使用HTML、CSS與JS來開發跨平台手機App的工具專案，這種開發出來的App稱爲Hybrid(混合) Apps。

使用Hybrid(混合) Apps的開發工具或專案還有很多，例如jQuery Mobile、Framework 7、Sencha Touch等等，各自有各自使用的框架或函式庫。因爲React Native的出現，有許多函式庫開始往原生的路線發展，目前原生的App大多使用JavaScriptCore作爲執行的核心技術。

桌面應用程式

Electron是由Github公司主導的，桌面應用程式開發框架的開放原始碼專案，基於Chromium與Node.js。它原本是用來要開發Atom程式碼編輯工具所使用的專案，原名稱爲Atom Shell，後來成爲獨立的專案，歷經兩年左右的開發，現在已有很多廠商使用它來開發桌面軟體。

Electron除了也是使用JavaScript語言與HTML作爲基礎程式語言，其優點主要是開發出來的桌

面應用可以跨作業系統，能在Mac OS X、Windows、Linux上運行，開發的門檻會低很多。不過，缺點是應用程式的檔案大小會比原生的應用大很多，而且執行效能比原生的應用程式差，對電腦資源如CPU與記憶體的需求會較高，目前這些缺點仍然是需要改進的重點。

此外也有類似的如NW.js專案，也是基於Chromium與Node.js的另一個開發平台。

伺服器端

Node.js的核心是使用高速的V8 JavaScript引擎，可以運行於一般常見的各種作業系統平台之上。Node.js有效的利用JavaScript語言中非阻塞I/O的優勢，在相同的伺服器資源下，可以提供承載高並行的運算處理。從2009年發展至今，已經逐漸成熟，近年來也發展在多核或多執行緒執行環境下的解決方案。目前已有許多大型的網站採用這個執行環境，例如IBM、Microsoft、Yahoo!、Walmart、Groupon、SAP、LinkedIn、Rakuten、PayPal等等公司。

工具

工具

工具

本書使用的工具與其他週邊軟體/服務如下。

快速開始樣版文件

- webpack-es6-startkit

需先安裝Node.js，安裝目前最新的LTS版本即可。

Flow檢查工具

- Flow靜態資料類型的檢查工具，10分鐘快速入門

快速學習與使用時候

- 測試、除錯與觀看結果的瀏覽器是Google Chrome。
- 學習用的網路線上撰寫平台是JS Bin

JS Bin中的JSHint設定值

```
{
  "esnext": "true",
  "asi": "false"
}
```

ES6環境

ES6環境說明

瀏覽器支援

根據ECMAScript相容表，在這本書開始撰寫的同時，2016年6月發佈的Google Chrome v51(Mac平台，桌上型電腦)，在ES6功能的相容性已經高99%。也就是說幾乎完全不需要額外的函式庫或套件，可以直接在瀏覽器上使用與測試ES6功能。不過，普及還需要一些時間，因爲其他的瀏覽器品牌，伺服器端或是行動裝置使用的瀏覽器軟體，目前還有很大一部份的功能沒有實作完成，不過，這也是時間的問題而已。

babel

babel是一個專爲未來性JavaScript語法所設計的編譯工具，目前我們可以先在這段時間中，使用它來將ES6或更新的程式碼，編譯爲現行大部份瀏覽器品牌都可執行的語法。這種作法目前大量的在JavaScript界被使用，也就是說我們撰寫程式碼時，可以使用ES6+的新功能和語法，至於測試或發行時，再用babel來編譯即可。

由於babel只編譯(或變換)語法，對於一些新的API或方法是不足的(例如Promise)，所以在使用新的API時，還需要使用babel-polyfill進行補充，對這些新的API提供支援。在使用到這些API的程式碼最上方加入:

```
import "babel-polyfill"
```

在webpack中只需要把babel-polyfill加入webpack.config.js的entry(進入點)陣列中即可:

```
module.exports = {
   entry: ['babel-polyfill', './app/js']
};
```

輸出

輸出

JavaScript是搭配HTML使用的，最終輸出(呈現結果)的方式，可以用HTML的輸出格式作爲輸出。不過在一開始學習時，或是在開發時簡單測試時，我們可以使用以下的語法在瀏覽器的除錯主控台中輸出數值:

```
console.log('Hello')
console.log('變數abc的值', abc)
```

> 注意: console.log並非涵蓋所有瀏覽器版本與品牌的標準方法，有些比較舊版本的瀏覽器中(尤其是Microsoft Internet Explorer6、7)是不能使用的，會造成程式錯誤。

> 註: 更多資訊請參考Google Chrome中的console API

其他的輸出方式

除了上述的`console.log`方式，還有以下的幾種輸出方式，以下簡單說明。

alert

`alert`是一個存在已久的全域方法(屬於window物件)，它可以在很舊的瀏覽器品牌與版本中使用。用它可以跳一個極醜的警告視窗:

```
//alert屬於window物件
window.alert('Hello')
//直接用alert也可以
alert('Hello')
```

除非你眞的有需要在如Internet Explorer6、7上作相容性的測試或開發，不然這個`alert`可以略過不要用，它有很多缺點，建議不要再使用它。

> 注意: 在之前早期的年代，這個`alert`有很多濫用的情況，很容易造成瀏覽器當掉。請不要作一個偷懶的工程師，用這個來當跳出的提示或錯誤視窗。現在已經有很多取代的跳出視窗方案，不但美觀而且問題少。

document.write

`document.write`是屬於`document`物件的一個寫入方法，它可以在HTML網頁上寫出輸出的結果。

```
document.write('Hello')
```

它同樣不是一個好方法，也是一個存在已久的輸出方法。表面上雖然看起來就這樣用，但實際上並不是那麼簡單。它只建議用在測試時，眞正使用上千萬不要使用。

> `document`相當於`window.document`，它是在`window`物件下的內容物件。

innerHTML

innerHTML需要配合document.getElementById(id)方法使用，它是一個DOM節點的屬性值，可以動態進行指定，例如:

```
document.getElementById("demo").innerHTML = 'Hello'
```

其中的demo要求是HTML中已經存在的DOM節點，例如下面這個HTML碼:

```
<div id="demo"></div>
```

註: 這個方式是推薦的輸出方式，也是現今很多JavaScript函式庫用來輸出結果的方式。

createTextNode

createTextNode是一個document物件中的方法，看它的名稱就知道用途是"建立文字的節點"。不過你要建立節點需要與其他的方法搭配才行，不是獨立使用的方法。例如:

```
const newText = document.createTextNode('Hello')
const demo = document.getElementById("demo")
demo.appendChild(newText)
```

其中的demo要求是HTML中已經存在的DOM節點。不過，這個方法比innerHTML複雜些，而且效能更低，只會用在特別的情況，例如有需要作節點的處理時。

結語

不論是innerHTML、alert、createTextNode、document.write，因爲輸出到網頁上的HTML碼中，所以必定會轉變爲字串值。也就是不論這轉出的值原本是數字的2進位、8進位、16進位，就會變成10進位，而其他的類型的值也會自動轉成字串值。在開發時並不會用這樣的輸出方式，所以還是只有console.log這方式可用。

基礎

變數、常數與命名

變數、常數與命名

變數與常數

變數(Variable)在JavaScript語言中扮演了重要的資料(值)存放角色。在ES6標準之前，只有定義變數沒有定義常數的方式，現在這兩者有不同的定義關鍵字與方式。

你可以把變數中儲放的資料(值)，想成是暫時存放的，而在常數中的資料(值)則是要長期(永久)存放的，一旦給定值後就不會再改變的資料。範例如下:

```
//這個a是不可變的(常數)
const a = 10

//這行程式碼會發生錯誤: "a" is read-only(只能讀不能寫)
a = 11

//這個b是可變的(變數)
let b = 5

//b可以再改變其中的值
b = 6
```

> 註: 這本書中的程式碼，將不會使用分號(;)作爲程式碼的結尾，請參考"寫作格式"這一章的說明。

你可把常數和變數想像是不同種類的盒子:

- 常數const/康死/: 裝入東西(值)之後就上鎖的盒子，之後不可以再更動裡面的值
- 變數let/累/: 暫時存放值的盒子，盒子是打開的，可以更動裡面的值

尤其JavaScript程式語言的程式設計師，在之前常會犯有不宣告就直接使用的壞習慣。我建議你最好在宣告常數或變數時，就給定一個值。雖然變數並沒有要求一開始要給定值，只有常數才有這個要求。

不過，在JavaScript語言中，常數指是的"不能再次指定值"(not re-assigned)，而不是"完全無法改變值"(immutable)，這兩個是有差異的概念。在之後的內容中會說明，物件類型的常數內容屬性是可以改變的，而在這個情況下常數會是一個常數指標。基本上，JavaScript語言中並沒有immutable(不可改變的)的特性。

因此，通常在程式碼撰寫中，我們使用`const`的情況會遠比`let`頻率高很多。除非你很確定這個變數等會在程式碼中其他部份會需要被改變，或是它是一個在類別定義裡的變數，不然就用`const`就對了，如果發生定義上的問題，除錯主控台或檢測工具會提醒你。

另外還有一個常見的風格指引建議是"一行宣告一個變數或常數"，範例如下:

```
//不好的宣告方式
const items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';
```

```
//好的宣告方式
const items = getItems()
const goSportsTeam = true
const dragonball = 'z'
```

這大概是想要偷懶少打一些const關鍵字，不過明顯這樣作是個壞習慣，雖然JavaScript允許你可以這樣作。你可能會問這樣作有什麼差異，好的宣告方式除了清楚明白外，最大的優點是它在程式除錯時，可以很清楚的理解是哪一行作什麼事。

> 註: ES5以前都只會用"var"(肉<台>)作爲標記，它是Variable(買<台>里哦波)的縮寫。"let"與"const"是ES6的新加入特性，你應該開始使用它們，本書不使用var來宣告變數。關於這兩種宣告方式的差異請參考: 這裡的問答與範例與這篇文章的說明

英文解說

const/康死/ 是 constant/康死撐/ 的縮寫，中譯是"常數"的意思。

let/累/，中譯是"讓..."， 例如"let's go"大概是"大家一起走吧"或"please let me go"(讓我走吧...其實是男女朋友說"分手"的意思)

風格指引

- (crockford)(airbnb 13.2)建議一行一個變數(或常數)宣告與註解，然後最好是按英文字母排列。
- (crockford)在函式中的最前面的敘述宣告變數。
- (airbnb 13.3)把let宣告的放在一起，const宣告的放在一起。
- (google)總是宣告變數。

命名(Naming)

我們需要先爲這些數值的代表名稱命名，稱爲常數名稱(constant name)或變數名稱(variable name)。以下說明的命名規則，不只包含常數與變數，還包含了函式、類別名稱的命名。

命名規則

變數或常數名稱基本上要遵守以下的規則:

- 開頭字元需要是ASCII字元(英文大小寫字元)，或是下底線(_)、錢號($)。注意，數字不可用於開頭字元。
- 接下來的字元可以使用英文字元、數字或下底線(_)。
- 大小寫是有差異的。
- 名稱不可使用JavaScript語言保留字詞。
- 名稱被稱爲identifier/愛登得罰兒/ (識別符，簡稱ID)，它在程式碼上下文中具有唯一性。

> 注意: 以下底線(_)爲開頭的命名通常是有特別用途，它是用在類別中的私有變數、常數或函式(方法)。錢號($)通常也會用於特殊的情況。

好的變數(函式、類別)命名

變數、函式、類別命名

變數與函式，都用小駝峰式(camelCase)的命名。類別用巴斯卡(PascalCase)或大駝峰式命名法

(CamelCase)命名:

```
let numberOfStudents
const numberOfLegs
function setBackgroundColor()
class Student{}
```

常數命名

在許多舊式的風格樣式指引中，會建議使用全大寫英文命名，字詞間使用下底線(_)連接:

```
const NAMES_LIKE_THIS='Hello'
```

不過，因爲ES6中加入了`const`用於指示爲常數後，其實這個規則並沒有那麼需要，程式檢查工具或執行時都會用常數的方式來檢查。所以你可以用一般對變數的命名規則就可以了:

```
const helloString = 'Hello'
```

少用簡寫或自己發明縮寫

- 不好的命名: setBgColor / 好的命合: setBackgroundColor
- 不好的命名: userAdr / 好的命名: userAddress
- 不好的命名: fName, lName / 好的命名: firstName, lastName

語意不明或對象不明

- 不好的命名: insert() 這是要插入什麼？/ 好的命名: insertDiv()
- 不好的命名: name 這是什麼名稱？/ 好的命名: memberName
- 不好的命名: isOk() 什麼東西ok不ok？ / 好的命名: isConnected 連上線了嗎？

不要拼錯英文單字

- 錯誤的拼字: memuItem / 正確的拼字: menuItem
- 錯誤的拼字: pueryString / 正確的拼字: queryString

英文單複數

陣列之類的集合結構，有數量很多的意思，大部份都用"複數"型態的字詞，或者資料的類型來分別，例如:

```
studentArray
students
```

執行的動作（函式或方法），如果針對單一個變數的行爲，用單數:

```
addItem()
```

如果針對多個數的行爲，用複數:

```
addItems()
```

針對全體的行爲，會用「All」:

`removeAll()`

常見的英文計量字詞:

count numberOf amountOf price cost length width height speed

常見的布林值開頭字詞:

isEmpty hasBasket

常見的字串值開頭字詞:

string name description label text

常用的動作詞（函數用）開頭

- make take 作某…事
- move 移動
- add 加上、相加
- delete/remove 移除
- insert 單體 splice 複合體
- extend append 展開
- set 設定
- get 獲得
- print 印出
- list 列出
- reset 重置
- link 連至
- repeat 重覆
- replace 取代
- find search 尋找
- xxxxTo 到xxx

具時間意義或指示的字詞

- will 通常指即將發生但未發生
- did 已發生
- should 應該發生

使用長一點的命名是可以提供更佳的閱讀性，而且與效能一點關係都沒有，JavaScript的程式碼最後都會再經過醜化與壓縮，變數名稱會用很短的名稱來取代，這點與程式開發中使用的名稱無關。

風格指引

- (airbnb 22.1) 避免使用單個英文字元的命名，像q, a, b, x, y, z
- (airbnb 22.4) 避免在命名的前後使用下底線(_)，例如 `__firstName__`、`_firstName`、`firstName_`都是很糟的命名

註解(Comment)

註解(Comment/康門/)　的用意是要讓以後的自己或其他人看到程式碼時，能很快與正確的理解，這段程式碼是在作什麼用的。所以在重要的程式碼的地方附近(通常在上面)，加上註解是好的習慣。註解也常常用在暫時性的把某段程式碼先擋住讓它不執行。好的註解習慣的養成建議:

- 不管你的英文有多破，用英文來進行註解是比較好的
- 註解上面需要加一行空白行
- 用`// FIXME:`與`// TODO:`來標示目前問題的位置與未來需要要作的功能處
- 用註解來說明函式的參數與回傳值

> 提示: 就已經英文很破爲何還要逼你寫英文註解？主要的原因是爲了"加快與專注在程式碼上的撰寫"。因爲要輸入中文註解還需要切換到中文輸入法，一下子寫程式碼又要切回英文輸入法，除了按鍵不同，符號的輸入也不同。能專心寫程式就寫程式碼不是很好嗎？

單行註解用的是雙斜線(//)(double slash/打波 死內需/)符號。以下爲範例:

```
//我是一個單行註解
```

多行註解用的是`/*...*/`符號，這也稱爲註解區塊(comment block)。以下爲範例:

```
/*
我是一個多行註解區塊
這是第二行註解
這是第三行註解
*/
```

風格指引

- (airbnb 17.2)註解上面需要加一行空白行
- (airbnb 17.4)用`// FIXME:`來指示目前問題所在點
- (airbnb 17.4)用`// TODO:`來指示之後需要作的項目所在點

英文解說

Comment/康門/是很常見的網路用語，它除了用在程式開撰寫裡當"註解"外，也有"評論"、"留言"的意思。部落格通常下面有個留言的功能，它的英文就是這個字詞。

slash/死內需/有猛砍、鞭打的意思，大概是右撇子英文，因爲右手拿斧頭砍下去畫出來的動作線就像這樣(/)。

反斜線(\)的英文是backslash/背可死內需/，它在程式撰寫時常用來作跳脫字元的符號

符號

我們在程式碼中會用到很多符號，除了你剛上面已經看到的幾個，還有常見的以下幾個。這裡把它們的英文都列出來，尤其是有一些中文翻譯字義上，我覺得實在是太容易混淆了，建議你還是用英文來看會比較清楚:

等號(=)

你可能會誤解這個符號的英文應該是是equal/伊括/，在中文是相等的意思。平常是這樣是沒錯，但用在程式碼上中並不是，它還有幾個不同的符號樣式。

一個等號(=)是指定(Assignment/餓賽門/)運算符號，也稱之爲Basic/貝喜客/ Assignment，因爲還有其他的指定運算符號，例如(+=)、(*=)。Assignment/餓賽門/中文有"分配"、"指定"的意思。

二個等號(==)時稱爲相等(equal/伊括/)，它是比較運算符號，用於比較兩個變數or常數的其中的值是否相等。它的反義符號是(!=)。

三個等號(===)時稱爲(identity/愛登得體/)，它也是比較運算符號，用於比較兩個變數or常數是否完全相等(包含類型與值)。它的反義符號是(!==)。

類型與比較之後的章節會再說明。這裡只要記得這些英文的和大概的意思就行了。

括號

括號本身的英文字詞在世界各地的英文國家中，有一些不同的講法，以下以美國的說法爲主。

括號(())

英文爲Parentheses/波潤捨西斯/，英文有"插入語"、"插曲"的其他意思。中文也有稱爲圓括號、小括號。在JavaScript的用途主要是函式呼叫、包括流程敘述以及分隔運算的優先次序。

中括號([])

英文爲Brackets/布累克特斯/，或是方括號(square/史魁兒/ brackets)，英文就有"支架"、"括住"的意思。主要用於JavaScript中的陣列(Array)。

大括號({})

英文爲Braces/布累謝斯/，英文就有"吊帶"、"背帶"的其他意思。中譯常稱爲"花括號"。常見有另一英文講法爲Curly/顆粒/brackets/布累克特斯/，中譯爲"捲毛的括號"，就是"花括號"了。在JavaScript語言中用於程式敘述的區域分隔(函式、迴圈、流程條件)，另外也用於物件的字面文字(Object Literals)。

引號

引號(")

英文爲Quote/括特/，其他還有"引言"、"引用"、"報價"的意思。有時爲了區分單引號，中文會翻成"雙引號"，在JavaScript語言中用於字串的宣告。

單引號(')

英文爲Single/醒狗/ Quote/括特/。在JavaScript語言中用於字串的宣告。

算術運算符

標準算術運算符

在所有的程式語言都是使用相同的算術運算符:

- 加(+) addition/額弟遜/

- 減(-) subtraction /捨吹遜/
- 乘(*) multiplication /摸特佛K遜/
- 除(/) division /第V俊/

JavaScript提供的其他的算術運算符

- 餘(%) Remainder/瑞妹的兒/
- 遞增(++) Increment/盈虧門/
- 遞減(--) Decrement/第虧門/
- 正號(+) Unary plus
- 負號(-) Unary negation

Unary/油呢瑞/ 是"一元的"意思，代表運算單一個參與的值。上面的遞增與遞減也是一元的運算符，例如以下的範例:

```
const a = 3++
const b = 5
const c = -b
const d = +"3"
```

註: (+=)這個符號稱爲加法指定(Addition assignment)，它屬於指定運算符。a+=b相當於a=a+b，其他的像(-=)(*=)(/=)(%=)都是類似的方式。

字串運算

只有以下兩個運算符可以用在字串連接(concatenate)時使用，其他的並不能使用:

- 加(+) addition/額弟遜/
- 加法指定(+=) Addition assignment

邏輯運算符

- 邏輯與(&&) Logical AND
- 邏輯或(||) Logical OR
- 邏輯非(!) Logical NOT

比較運算符

除了上面的已說明的等號(=)相關的比較較運算外，還有以下幾個常用的:

- 大於(>) Greater/貴特兒/ than
- 小於(<) less/蕾絲/ than
- 大於等於(>=) Greater than or equal
- 小於等於(<=) Less than or equal

註: 如果你搞不清大於與小於符號哪個是哪個，舉起你的右手，打開虎口就像">"，所以它也是右撇子符號，其實這和人類社會書寫習慣有關，我們都是從左至右書寫程式碼。

其他常用符號

- 逗號(,) comma /康馬/
- 冒號(:) colon /叩龍/
- 分號(;) semicolon /些米叩龍/
- 句點(.) dot/搭/

- 驚嘆號(!) exclamation mark /A死客妹遜/ /馬克/
- 問號(?) question mark /虧遜/ /馬克/
- 下底線(_) underscore /昂得死過/
- 重音符號(`) back-tick /背踢克/
- 連接號(-) hyphen /嗨芬/
- 星號(*) asterisks /A死特瑞死/
- 錢號($) dollar signs /打惹/ /賽/

連接號(-)與負號(-)、破折號(-)這三個在通用的電腦鍵盤上長得是一樣的。實際上它們三個的英文單字不同，分別是hyphen、minus sign與dash，用途也各有不同。

英文解說

用中文來解說這段程式碼怎麼解說:

```
const abc = 10
```

先看一下英文會怎麼說(以下來自MDN的說法):

```
// define abc as a constant and give it the value 10
```

中文這樣說對嗎？

定義(宣告)abc識別符爲常數，它的值等於10

這個講法有一個嚴重的問題，就是等於這個中文說法是不正確的，因爲依照我們之前的解說，一個等號(=)時，中文是"指定、分配"的意思。所以下面的說法會比較正確:

定義(宣告)abc(識別符)爲常數，與指定它的值爲10

這相當於兩件事寫在同一個敘述中，一個是定義，一個是指定值。常數的話，規定在宣告時就一定要給定值，但變數沒這規定，所以變數的情況可以分成兩個敘述:

```
let xyz
xyz = 5
```

以中文來說的解說方式就會是兩個敘述:

定義(宣告)xyz(識別符)爲變數
指定xyz(識別符)的值爲5

保留字詞

保留字詞通常是JavaScript中具有特別用途的類別、函式、變數名稱，這些字詞不能使用到命名中。保留字可以區分以下各分類:

- JavaScript中的函式、標記、方法保留字
- JavaScript中的類別、屬性、資料類型的保留字
- JavaScript未來標準規格可能會使用的保留字
- HTML中Windows物件的保留字
- HTML事件處理的保留字
- 所使用的JavaScript函式庫或框架中的保留字

以下網頁中可參考所有的保留字:

- JavaScript Reserved Words(W3Schools)
- Keywords(MDN)

對於保留字你可以先不用那麼在意，因爲現在已經有很多檢查工具(linter)，都加入了保留字的檢查功能，如果你有誤用時，工具會出現錯誤或警告訊息。另外，保留字詞通常很短，而且是有意義的英文字詞。當你在寫自己的命名時，儘可能用組合的字詞(例如firstName, studentName)，然後長度長一些，誤用到保留字詞的機會大概不會有。

這裡說的保留字，在使用上是用"全部小寫"的英文單字，例如`let`不能寫成"lEt", "LET", "Let"。

結語

本章主要說明了兩個主要的宣告關鍵字`let`與`const`的用法，以及在JavaScript中的命名規則，這些命名規則通用於所有的類別、函式、變數與常數的識別符。本章也有簡單介紹程式碼註解的使用方式。

最後將所有常用到的運算符、符號，整理出來它的符號樣式，以及英文字詞，供初學者參考。如果方便的話，應該要作個單字卡來學習一下所有的英文單字。對初學者之後學習是相當有幫助的(例如聽懂英文教學影片在講什麼)。

家庭作業

作業一

學校給了你一個新的工作，希望把學生的資料的各項功能，寫成一個JavaScript的應用程式，經過調查和討論後，學生會有幾個基本的資料:

- 學號
- 學生的姓名
- 學生的出生年月日
- 學生的住址
- 學生的聯絡電話
- 學生的家長(其中一位聯絡用的)

現在要請你把這些資料的項目，寫成變數或常數的名稱，請你進行命名的工作。

作業二

最近公司接了一個家用遊戲代理商委託的案子，希望能作一個網站上的JavaScript應用程式，把遊戲公司目前代理的遊戲軟體，放到網路上提供給他們的客户了解。經過調查和討論後，遊戲大致會有以下幾個資料欄位:

- 遊戲的發行編號(例如: KK123)
- 遊戲名稱
- 遊戲的製作公司
- 遊戲的發行日期
- 遊戲支援的語言(例如: 日文、中文、英文...)
- 遊戲支援的平台(例如: PS4, Xbox...)
- 遊戲的種類(例如: 動作類、射擊類)
- 遊戲是否爲18禁
- 遊戲是否需要網路連線

現在要請你把這些資料的項目，寫成變數或常數的名稱，請你進行命名的工作。

作業三

打字練習對於提升撰寫程式碼的速度是很有幫助的。有人說很多學寫程式的人，最後只會兩個按鍵組合，一個是複製(Ctrl+C)，另一個是貼上(Ctrl+V)。對於我們有遠大理想的未來優秀程式設計師，這樣的按鍵組合是不足的，所以需要多練習打字。請依照以下的兩個區域，打出這些常用的關鍵字與符號了。如果你覺得太簡單，可以把行數對調再練習下一次，而且最好能練習到不看鍵盤就能打出來:

```
if else else if const let let const
import export return import export return
switch case default break case switch break
class extends super class extends super
for in do while continue while for in do
try throw function catch function try catch
if else function const let for return

+-*/% {!?:} (++--) [1][0][3]
(){_-/\} (_.){}() => >=
<= == !== && ||
```

資料類型(值)

資料類型(值)

資料是所有電腦運算的基礎，對於電腦來說，所有的資料都只是0與1的訊號。但對於人類來說，資料類型的區分就複雜得多了，需要因應各種不同的情況來使用。

JavaScript中有6種原始的原始資料類型(Primitive Data Type)，分別是數字(Number)、字串(String)與布林(Boolean)，以及空(null)與未定義(undefined)兩種特殊值，ES6後加入了符號(Symbol)。

另外還有另一個非常重要的第7種資料類型 - Object(物件)。

我們常常會把屬於物件的陣列(Array)、日期時間(Date)、函式(Function)、正規表述式(RegExp)等等獨立出來說明，因爲這些物件都有各自獨立的作用，算是特別的物件類型。尤其是函式(Function)，JavaScript在把函式視爲'function'的一種獨立類型(使用typeof的回傳值)。

總之，我們的分類是像下面這樣，本章節也是以這個分類開始來介紹，而陣列與物件將會在另一章節說明。至於新的資料類型symbol(符號)屬於進階的資料類型，本書中並不會介紹。

主要的原始資料類型:

- 數字(Number/難波/)
- 字串(String/斯翠/)
- 布林(Boolean/布林/)

特殊的原始資料類型:

- 空(null/怒偶/)
- 未定義(undefined/昂滴犯/)
- 符號(Symbol/辛波/)

複合型(composite)或參考型(reference)的資料類型:

- 陣列(Array/額瑞/)
- 物件(Object/阿捷特/)

註: 所謂的"原始資料類型"，指的是在程式語言中，最低階的一種資料類型，不屬於物件也沒有方法。它也具有不可改變的(immutable)特性。不過，JavaScript語言中也存在名稱爲String、Number、Boolean的物件，用來對應原始資料類型，它們是一種包裝物件(wrapper object)，提供了原始資料類型可以使用的一些延申的屬性與方法。

註: JavaScript的確是一個物件導向的程式語言。不過，JavaScript中的物件導向特性與其他目前所流行的物件導向語言例如Java、C++等有很大的不同。

鬆散資料類型

JavaScript是一個鬆散資料類型(loosely typed)的程式語言，或稱之爲"動態的程式語言(dynamic language)"。意思是說你不需要爲變數/常數在定義時，就規定要使用哪一種的資料類型。取而代之的是，只需要指定它的值，JavaScript會依照你所指定的值決定變數/常數的資料類型。

```
let foo = 42    // foo現在是Number資料類型
let foo = 'bar' // foo現在是String資料類型
let foo = true // foo現在是Boolean資料類型
```

判斷資料類型 - typeof

那麼要如何判斷目前變數/常數的資料類型？最簡單的方式就是使用這個運算符typeof，它是個類似正負號(+-)的運算符，是個一元的運算符，它是加在運算元的前面。typeof最後會回傳一個所屬資料類型的字串，範例如下:

```
console.log(typeof 37) //'number'
console.log(typeof NaN) //'number'
console.log(typeof '') //'string'
console.log(typeof (typeof 1)) //'string'
console.log(typeof true) //'boolean'
console.log(typeof null) //'object'
console.log(typeof function(){}) //'function'
```

> 註: 爲何null資料類型的的typeof結果是'object'(物件)，而不是'null'呢？依據ECMAScript的標準章節11.4.3條對typeof的規定就是回傳'object'。不過目前有一些反對的聲音，認爲null是原始資料類型，理應當回傳自己本身的資料類型。也有認爲現在修改這個太晚了，一經改動的話會造成很多舊程式被影響。截至ES6標準這一規定仍然沒有更動。

> 註: typeof的回傳值還有一個特例，就是函式的回傳值是'function'，因爲在JavaScript覺得它太特別了，所以另外給它獨有的回傳值。

數字(Number)

JavaScript的數字(Number/難波/)類型是64位元的浮點數，類似於其他程式語言(如Java)的double/打波/ 資料類型。但在JavaScript中數字類型沒有如其他程式語言中，有獨立的整數(int)或浮點數(float)類型。所以對JavaScript來說，1與1.0指的是相同的資料類型與值。另外，數字可以使用算術運算符(+-*/%)等來進行運算。以下是幾個宣告的範例:

```
const intValue = 123
const floatValue = 10.01
const negValue= -5.5
```

> 註: 浮點數(float, FP)指的是帶有小數的數值，使用64位元儲存一個浮點數的稱爲雙精度浮點數(double)

> 註: 雖然對JavaScript來說，浮點數與整數都屬於同一種資料類型。但瀏覽器內部的JavaScript引擎中是有分別的，它在執行時其實會把這兩種不同的數字類型區分出來，整數的運算理論上都會比浮點數快很多。

電腦中的程式語言在數學上的運算，總會有超出儲存的極限與無法處理的情況。JavaScript使用三個特殊的記號，來代表在數字上處理的極限值或非數字的情況，但它們都屬於數字資料類型:

- +Infinity/硬飛了踢/: 正無限值(相當於Infinity)
- -Infinity: 負無限值
- NaN: 代表不是數字

```
console.log(1/0) //Infinity
```

```
console.log(1/-0) //-Infinity，0有分+0與-0，這有點算陷阱
console.log(Infinity - Infinity) //NaN
```

在JavaScript中數字的最大與最小值，可以使用Number.MAX_VALUE與Number.MIN_VALUE獲得。

```
console.log(Number.MAX_VALUE)
console.log(Number.MIN_VALUE)
```

整數的進位

以整數來說，除了最常使用的10進位，還有2、8、16進位幾種。

2進位(Binary/敗了綠/)在ES6之後可以使用0b或0B開頭來定義:

```
const FLT_SIGNBIT  = 0b10000000000000000000000000000000 // 2147483648
const FLT_MANTISSA = 0B00000000011111111111111111111111 // 8388607
```

8進位(Octal/阿多/)在ES6之後可以使用0o或0O開頭來定義:

```
const n = 0O755 // 493
const m = 0o644 // 420
```

16進位(Hexadecimal/海斯達斯摸/)中直接可以用0x開頭來定義:

```
const x = 0xFF
const y = 0xAA33BC
```

在ES6之前，對於2或8進位並沒有內建的直接可定義方式，需要透過一個字串轉數字(整數)的方法parseInt(string, radix)，將一個2進位或8進位的數字字串轉換，這方式也可以轉換16進位的數字字串:

```
//8進位
const octalNumber = parseInt('071', 8)

//2進位
const binaryNumber = parseInt('0111', 2)

//16進位
const hexNumber = parseInt('0xFF', 16)
```

> 註: parseInt雖然主要是傳入字串，轉換爲整數。但也有傳入浮點數，轉換爲整數的功能。

如果是ES6之後的2、8進位定義格式的字串，需要用Number()方法才能轉爲10進位，parseInt是無法正確轉換爲整數的。範例如下:

```
const binaryNumber = Number('0b11') // 3
const octalNumber = Number('0o11') // 9
const hexNumber = Number('0x11')  // 17
```

> 註: 不論是2、8、16進位的數字，直接輸出到HTML碼中必定會自動轉成10進位的數字字串輸出。

另一個情況是要將10進位數字以不同的進位基數(radix)轉爲字串，通常是用在輸出時，這時要

使用數字的方法 - toString([radix])。不過輸出後的字串格式與上面所說的格式不同，請見以下的範例:

```
const decimalNumber = 125
console.log(decimalNumber.toString()) //'125'
console.log(decimalNumber.toString(2)) //'1111101'
console.log(decimalNumber.toString(8)) //'175'
console.log(decimalNumber.toString(16)) //'7d'
```

浮點數的轉換

字串轉浮點數

對照上面的字串轉數字(整數)的parseInt方法，字串中還有另一個parseFloat(string)可以轉換數字字串爲浮點數，不過就像最上面所說明的，數字1.0相當於1，小數點後面如果是0都會自動消失不見。以下是範例:

```
const aNumber = parseFloat("10")  //10
const bNumber = parseFloat("10.00") //10
const cNumber = parseFloat("10.33") //10.33
const dNumber = parseFloat("34 45 66") //34
const eNumber = parseFloat("40 years") //40
const fNumber = parseFloat("He was 40")  //NaN
```

浮點數轉整數

至於浮點數要轉換爲整數，需要使用Math物件中的幾個方法來轉換，因爲浮點數要轉換有幾種不同的情況，看是要轉換爲直接進位，還是四捨五入，還是直接捨去小數點，就看你的需要，以下爲範例:

```
const floatValue = 10.55
const intValueOne = Math.floor( floatValue ) //地板值 10
const intValueTwo = Math.ceil( floatValue ) //天花板值 11
const intValueThree = Math.round( floatValue ) //四捨五入值 11
```

> 註: Math是專門用於數學運算使用的JavaScript語言內建工具物件，裡面有很多好用的數學方法，請參考Math(MDN)與JavaScript Math Object

整數轉浮點數

再強調一下，對於JavaScript來說，數字就是數字類型，沒有什麼浮點數或整數的類型。3.00與3的數字在JavaScript中都是一樣的，直接轉出數字3.00依然會用3來顯示。

唯一可能作的事情是調整輸出的格式，在輸出的時候都會變成字串類型。這時可以用數字物件中的toFixed([digits])方法來達成，以下爲範例:

```
const myString = (3).toFixed(2) //string, 3.00

const numObj = 12345.6789
const numObjString = numObj.toFixed() //string, 12346
```

> 註: toFixed方法如果不加傳入參數，會視爲0位小數點，而且會作四捨五入。

其他類型轉換爲整數

雙位元反相運算(Double Bitwise NOT)(~~)

波浪符號(Tilde)(~)，在JavaScript語言中的運算符名稱爲 "位元反相運算(Bitwise NOT)"，這是長得像波浪或毛毛蟲樣子的符號。根據它的功能文件說明，它會把"數字 x 轉換爲 -(x + 1)"。也就是說像下面這樣的例子:

```
const a = ~10 //a is -11
```

那如果是兩條毛毛蟲(Double Bitwise NOT)(~~)呢？上面的例子會變成如何:

```
const b = ~~10 //b is 10
```

兩條毛毛蟲看起來沒什麼用，只是回復原本的數字值而已。不過它的眞正功用是轉換其他類型爲整數，而且它有相當於`parseInt`的功能，但並不是百分之百相等。在"正"數值情況下，由浮點數轉爲整數時，也相當於`Math.floor()`。但重點在於它的效能在某些瀏覽器非常快，所以有很多程式設計師會使用這樣的寫法。以下爲範例:

```
console.log(~~'-1')  // -1
console.log(~~'0')   // 0
console.log(~~'1')   // 1
console.log(~~true)  // 1
console.log(~~false) // 0
console.log(~~null) // 0
console.log(~~undefined) // 0
```

> 註: `parseInt`雖然主要是傳入字串轉換爲整數。但它也有傳入浮點數，轉換爲整數的功能。

> 註: 位元操作符(Bitwise operators)是用於數字類型的位元運算使用的，本書並沒有特別提及，請參考Bitwise operators(MDN)。

正號(+)

正號(Unary Plus)(+)也是一個一元的運算符，它也有轉換其他類型爲數字的能力，不過它的功能接近`parseFloat`，但並非完全相等。負號(-)也有同樣的功用，但很少被使用。以下爲範例:

```
console.log(+'2.3') //2.3
console.log(+'0') //0
console.log(+'foo') //NaN
console.log(+true) //1
console.log(+false) //0
console.log(+null) //0
```

> 註: 這份網路上的Stackoverflow問答中有一張表，列出所有轉換的情況。

數字的精確問題

在電腦中的數字並非是永無極限的，有最大的數值上限也有最小的下限。另一種情況是，當在進行運算時，有時候會出現不如你想像的結果，尤其是在浮點數的運算上，這些大部份與浮點數的精確度(precision)有關，以下是幾個典型的範例，這些已經算是陷阱題目:

範例一：

注意: 第二行的值失去精確，它會變成另一個值

```
console.log(999999999999999)
console.log(9999999999999999)
```

範例二

注意: x, y都不是你想的結果

```
const x = 0.2 + 0.1
const y = 0.3 - 0.1
```

註: 範例來自Number, Math

因此，在處理浮點數時，要格外小心。如果你遇到了不可預期的結果，有可能是精確度出了問題。如果你需要處理浮點數的運算，可以尋找合適的函式庫，單靠JavaScript語言本身可能會有所不足。例如BigDecimal.js或decimal.js

字串(String)

字串類型用於描述一般的字串值，是使用相當廣泛的值。JavaScript常與HTML搭配使用，而從HTML取得或輸入的值，以及輸出到HTML的值，都是字串類型的值。在JavaScript中，字串是使用Unicode作爲編碼(UCS-2/UTF-16)。以下爲簡單的宣告範例：

```
const aString = '你好'
const bString = 'Hello'
```

對於JavaScript語言來說，使用雙引號標記("")與單引號標記(')來定義字串值，結果都是完全一樣的。在本書中，會推薦優先使用單引號標記(')。主要原因是JavaScript經常需要與HTML碼搭配使用，而HTML碼中也會使用引號來作爲標記屬性值的定義，所以讓HTML屬性使用雙引號標記("")，而Javascript中的字串使用單引號標記(')，這變成一個約定俗成的撰寫習慣。

註: 有些程式語言(例如PHP)，對於使用雙引號標記("")與單引號標記(')定義兩種字串值，其特性並不完全相同。不過在JavaScript語言中是完全相同。

註: 優先使用單引號標記(')只是推薦的撰寫習慣，但不見得每個JavaScript函式庫或框架都這樣作，像Google, Airbnb, Node, npm都是這麼作。而推薦優先使用雙引號標記("")，經常被提到的是jQuery函式庫。

字串與字元

存取一個字串中的單一個字元，可以用類似陣列的存取方式，或是用`charAt()`方法。不過，JavaScript中並沒有所謂的字元(Character)類型，所以它依然是個字串(String)資料類型，只是只有一個字元而已。JavaScript中隱喻的將字串設計作爲字元陣列，所以你可能會在字串類型中看到很多與陣列同名的方法，以及類似的使用情況，但這並不代表字串是一種眞正的陣列物件。

```
const a = 'cat'.charAt(1)   //  'a'
const b = 'cat'[1]   // 'a'

console.log(typeof a) //string
```

注意: 使用陣列的提取字串中的字元這種方式爲不安全(unsafe)的方式，尤其在某些舊的瀏覽器品牌中不支援。

跳脫字元(Escape characters)

當需要在雙引號記號("")中使用雙引號(")，或在單引號記號(')中使用單引號(')時，或在字串中使用一些特殊用途的字元。使用反斜線符號blackslash(\)來進行跳脫，這稱爲跳脫字元或跳脫記號。例如：

```
const aString = 'It\'s ok'
const bString = 'This is a blackslash \\'
```

註: 在JavaScript中，跳脫字元在雙引號記號("")與單引號記號(')中均可使用，這一點可能與其他程式語言有些不同。

樣版字串(Template strings)

ES6中新的字串寫法，稱爲樣版字串(Template strings)，使用的是重音符號backtick(``)來敘述字串。樣版字串可以多行、加入字串變數，還有其他延伸的用法，這在一些新式的函式庫或框架中(例如Angular)被大量使用。範例如下：

```
const aString = `hello world`
const aString = `hello!

 world!`
```

在樣版字串中可以嵌入變數/常數，也可以作運算，嵌入的符號是使用錢號與花括號的組合(${}):

```
const firstName = 'Eddy'
console.log(`Hello ${firstName}!
Do you want some
rabbits tonight?`)


const x = 5
console.log(`5 + 3= ${x + 3}`)
```

字串運算

字串中有很多可以進行處理字串使用的函式與符號，以下以不同的處理情況來說明。不過，因爲字串的搜尋、取代等等，需要配合正規表述式的樣式，在這裡的內容就先不討論。

字串串接

使用加號(+)或加法指定(+=)是最直覺的方式，效能也比concat()好，這兩種方式都是同樣的結果，範例如下：

```
//使用concat()串接
const aString = 'JavaScript'
const bString = aString.concat(' is a', ' script language')
console.log(bString)

//使用(+=)串接
```

```
let cString = 'JavaScript'
cString += ' is a'
cString += ' script language'
console.log(cString)
```

與其他類型作加號(+)運算

不過，當你看到加號(+)時，可能會有個疑問，不是在數字的運算中也有加號(+)，像下面這個程式碼，結果應該是什麼，到底會是101還是11？

```
const a = '10' + 1
console.log(a)

const b = 10 + '1'
console.log(b)
```

依據ECMAScript的標準章節11.6.1條對加號(+)運算的規定，簡單說明如下:

> 當運算式的左邊運算元或右邊運算元，是字串類型時，則使用字串連接(concatenating)的運算，非字串類型的運算元則使用ToString(轉爲字串)的轉換。

所以上面的兩個加號(+)運算的結果將會是101。那更複雜的運算像下面的程式碼又會是什麼結果呢？你可以試看看:

```
console.log( 3 + 4 + '5' )
console.log( 4 + 3 + '5' + 3 )
```

所以，當你想要轉換某個類型爲字串時，直接與一個空白字串作加號(+)運算，就會變成字串類型了。這也是當用來把其他類型轉爲字串的最簡便的語法。當然這是因爲每個原始的資料類型，都有內建的toString方法，用加號(+)也是類似的效果，不過這對複合型的資料類型是沒有用的，例如陣列或物件。

```
const a = 6 + ''
const b = true + ''
```

> 註: 除了加號(+)運算外，其他的數字運算符號(-/*)都會將運算元轉換爲數字。

字串長度

length是字串的屬性值，可以讀取出字串目前的長度(字元個數)，中文字每個字算1個長度:

```
const aString = 'Hello World!'
const bString = '你好'
const aStringLength = aString.length //12
const bStringLength = bString.length //2
```

大寫與小寫

toUpperCase與toLowerCase兩個方法，用於將英文字串轉變爲大寫或小寫:

```
const aString = 'Hello World!'
const bString = aString.toUpperCase()
const cString = aString.toLowerCase()
```

清除字串左右空白字元

trim方法，可以清除字串左右(前後)空白字元:

```
const aString = ' Hello World!    '
const bString = aString.trim()
```

子字串搜尋(索引值)

indexOf可以回傳目前在字串中尋找的子字串的索引值，索引值的順序是從左邊由0開始計算，一直到字串的總長度減1。如果尋找不到該子字串，則會回傳-1，它還有第二個傳入參數是可以指定開始尋找的索引值。以下爲範例:

```
const aString = 'Apple Mongo Banana'

console.log( aString.indexOf('Apple') ) // 0
console.log( aString.indexOf('Mongo') ) // 6
console.log( aString.indexOf('Banana') ) // 12
console.log( aString.indexOf('Honey') ) // -1
```

取出子字串

JavaScript中對於子字串的取出，有三種方法可以使用substr、substring、slice，其傳入參數並不相同。substr與substring這兩個英文字詞，在字義上是一模一樣，差異只在於簡寫。而slice的字義是"切割"的意思。另外，在陣列(Array)類型中，也有一個同名方法slice，這代表在JavaScript中對於子字串提取，與陣列(Array)中的slice方法類似。從上面的內容看來，字串資料類型，本身結構也類似由多個單字串組成的陣列。實際上slice方法，與substring十分類似，僅有一些行爲上的差異。

substring

substring是ECMAScript 5.1標準定義的方法，使用兩個參數，一個是要取出的子字串的開頭索引值，另一個是要取出的子字串的結束索引值，子字串將不會包含結束索引值的那個字元。如同之前indexOf中說明的，字串的索引值從字串的開頭是以0開始計算。

語法: str.substring(start[, end])

範例:

```
const aString = '0123456789'
console.log(aString.substring(0, 3)) //012
console.log(aString.substring(5, 8)) //567

//以下爲特殊情況
console.log(aString.substring(4, 4)) //''
console.log(aString.substring(5)) //56789
console.log(aString.substring(5, 2)) //234
console.log(aString.substring(5, 20)) //56789
console.log(aString.substring(-5, 2)) //01
console.log(aString.substring(2, -5)) //01
console.log(aString.substring(-5, -5)) //''
```

substring方法有很多特殊的情況，例如超出索引值、開頭索引值大於結束索引值，以及索

引值出現負數情況等等。其中最特別的是，當"開頭索引值大於結束索引值"時，它會自動對調兩者的參數位置，也就是開頭索引值會變結束索引值，這個作法經常會出現出乎意料的結果。

另外，從上面的例子來看，substring會直接把負數的參數當作0看待，而超過索引值大小就當作最大索引值。

> 註: 由於substring的自動對調索引值、負數參數自動變爲0之類的古怪特性，它不建議被使用。而是要改用slice方法。

slice

slice扮演的是substring的替代救援方法，它的大部份的行爲和substring類似，傳入參數值也是一樣的。看以下的範例和substring的範例比較一下，大概就知道在很多特殊情況下，slice是不會有值(只有空字串)回傳的。整體來說，它的回傳結果比substring容易預測，而且也較爲合理。

> 語法: str.slice(start[, end])

```
const aString = '0123456789'
console.log(aString.slice(0, 3)) //012
console.log(aString.slice(5, 8)) //567

//以下爲特殊情況
console.log(aString.slice(4, 4)) //''
console.log(aString.slice(5)) //56789
console.log(aString.slice(5, 2)) //''
console.log(aString.slice(5, 20)) //56789
console.log(aString.slice(-5, 2)) //''
console.log(aString.slice(2, -5)) //234
console.log(aString.slice(-5, -5)) //''
```

如果眞的認眞比較slice與substring在特殊的情況下的不同:

- 當開頭索引值大於結束索引值時，它不會互換兩者的位置
- 當索引值有負值的情況下，它是從最後面倒過來計算位置的(最後一個索引值相當於-1)
- 只要遵守"開頭索引值"比"結束索引值"的位置更靠左，就有回傳值。其他都是空字串。

substr

substr基本上並不屬於在ECMAScript標準中所定義的方法，它是歸類在附錄章節中瀏覽器相容方法。也就是說，它是因爲原本有某些瀏覽器品牌自己製作出來的一個方法，後來把它放在附錄中，作爲其他瀏覽器品牌的參考之用。因此它有可能在不同的瀏覽器品牌中有不同的結果。例如在IE8以前的版本，它在特殊情況(開頭索引值爲負數時)，結果就和IE9或其他瀏覽器不同。所以建議你儘量不要用這個方法，尤其是用在負數索引值的情況。除此之外，它的方法名稱實在和substring也太像了，很容易搞混兩個的傳入參數差異。

> 語法: str.substr(start[, length])

```
const aString = '0123456789'
console.log(aString.substr(2,4)) //2345
console.log(aString.substr(0,8)) //01234567
```

> 註: 口訣"截長補短"，所以縮短字詞的方法"substr"用的是長度(length)。

總之，在字串提取子字串的功能，優先使用`slice`方法。

> 注意: 再強調一次，IE瀏覽器不能支援負數的開頭參數。我建議你不要用`substr`方法，因爲它不是一個ECMAScript標準的方法。會放上這段的內容，主要是要讓你了解有這個方法，而且能看懂其他的教學文章或其他設計師所寫的程式碼。

風格指引

- (Airbnb 6.1) 雖然使用雙引號(")與單引號(')都是一樣的宣告方式，但建議使用單引號(')
- (Airbnb 6.2/6.3) 字串中的長度超過100字元時，需分成多個字串，然後使用字串的串接符號(+)
- (Airbnb 6.4) 當需要在程式中建立字串時，優先採用樣版字串的方式，可以提高可閱讀性與精確。
- (Airbnb 6.6) 避免在字串中使用不必要的跳脫字元
- (Google) 單引號(')優先使用於雙引號(")。這有助於建立包含HTML的字串。
- (Google) 定義多行字串時，分成一行行的字串，再使用串接符號(+)進行串接。

布林(Boolean)

布林(Boolean/布林/)或簡寫爲 Bool/布爾/，它是由發明的科學家George Boole命名。是一種使用絕對兩分法的值(黑白/陰陽/眞假)，在JavaScript中以關鍵字`true`與`false`來作爲布林的兩種可用的值。布林值通常用於判斷式中，與比較運算符有關，常用於流程控制的結構中。 例如以下的範例:

```
const a = true
const b = false

console.log(typeof a) //boolean
console.log(1=='1') //true
console.log(typeof (1=='1')) //boolean
console.log(b!=a) //true
```

> 注意: 所有的JavaScript關鍵字(保留字)都是小寫的英文，像`true`的話，如果寫成True、TRUE、TrUe，都是不對的寫法。

布林(Boolean)直接在HTML格式中輸出時，`true`會輸出'true'字串，`false`輸出'false'字串。不過因爲HTML輸入時(或是在HTML上抓取數值時)，有可能需要將'true'字串或'false'字串轉成布林值的情況，這時候要用像下面的轉換方式:

```
const aString = 'true'
const bString = 'false'
const aBool = (aString === 'true')
const bBool = (bString === 'true')

console.log(aBool, typeof aBool) //true "boolean"
console.log(bBool, typeof bBool) //false "boolean"
```

驚嘆號(!)之前有說過是個邏輯運算符，名稱爲"Logic NOT"，用在布林值上具有反轉(Inverted)的功能，雙驚嘆號(!!)就等於"反轉再反轉"，等於轉回原本的布林值:

```
const aBool = true
const bBool = !aBool //false
const cBool = !!aBool //true
```

雙驚嘆號(!!)並不單純是在這樣用的，它是爲了要轉換一些可以形成布林值的情況值，列出如下:

- `false`: 0, -0, null, false, NaN, undefined, 空白字串('')
- `true`: 不是false的其他情況

```
const aBool = !!0 //false
const bBool = !!'false' //true
const cBool = !!NaN //false
```

註: 你也可以用Boolean物件來作轉換其他資料類型爲布林的這件事。

falsy與短路求值

Douglas Crockford是在JavaScript界相當知名的大師級人物，他主張使用"**truthy**"與"**falsy**"來描述資料類型的值。也就是說，像上面說講的那些會轉換爲布林值的false值的資料類型的值，通稱爲"falsy"(字典裡是沒這個字詞，意思是"false的")，你可以把它當成是"false家族成員"。

"falsy"包含了0, -0, null, NaN, undefined, 空白字串('')，當然也一定包含了false值

"falsy"的概念在JavaScript的邏輯運算，以及布林值中都是很重要的概念。之前已經看到一個邏輯運算符 - 邏輯反相(Logic NOT)(!)的運用，還有兩個邏輯運算符，也很常用到:

- 邏輯與(Logical AND)(&&)
- 邏輯或(Logical OR)(||)

邏輯與(&&)與邏輯或(||)的運算，正常的使用情況下，並沒有什麼太大的問題，範例如下:

```
console.log(true && false) //false
console.log(true || false) //true
```

只不過，因爲JavaScript採用了與其他程式語言不同的邏輯運算的回傳值設計，我們把這兩個運算符稱爲"短路求值**(Short-circuit)**"的運算符。實際上JavaScript中，在經過邏輯與(&&)與邏輯或(||)的運算後，它的回傳值是 - 最後的值**(Last value)**，並不是像在常見的Java、C++、PHP程式語言中的是布林值。

因此，短路求值運算變成JavaScript中一種常被使用的特性，尤其是邏輯或(||)。那這與之前所說的"falsy"特性有何關係？關於邏輯或(||)運算在JavaScript語言中的可以這樣說明:

邏輯或(Logical OR)(||)運算符在運算時，如果當第1個運算子爲"falsy"時，則回傳第2個運算子。否則，將會回傳第1個運算子。

也就是說像下面這樣的程式碼，基本上它的回傳值都不是布林值，而是其他的資料類型的值:

```
console.log('foo' || 'bar') // 'foo'
console.log(false || 'bar') // 'bar'
```

"falsy"擴大了這種運算的範圍，像下面的程式碼的回傳情況:

```
console.log( 0 || '' || 5 || 'bar') //5
console.log(false || null || '' || 0 || NaN || 'Hello' || undefined)
```

而出現了一種常見的在程式碼中簡短寫法，這稱爲"指定預設值"的寫法，就是使用邏輯或(Logical OR)(||)運算符，範例如下:

```
let a = value || 5 //5是預設值
```

不過，這樣的指定預設值的寫法，並不能完全精確判斷value屬於哪一種資料類型與值的方式，在某些情況下會出現意外，例如你可能認爲`value=0`也是合法的值，但`value=0`時，a會被短路求值成5。所以，只要當value是"falsy"時，變數a就會被指定爲預設值5。所以在使用時還是需要特別注意，不然你會得到出乎意料的結果。

註: 每種程式語言都有短路求值的一些設計方式，請參考Short-circuit evaluation

風格指引

- (Airbnb 23.3) 如果屬性(變數/常數)或方法(函式)是一個布林值，使用像isVal()或hasVal()的命名。

空(null)與未定義(undefined)

比較其它程式語言的設計，一般都只有一種類似Null的空值原始資料類型。而JavaScript多了一種未定義(undefined)類型，會這樣設計是有它的歷史背景與原因。

空(null)就是空值，代表的是沒有值的意思。

未定義(undefined)即是尚未被定義類型，或也有可能是根本不存在，完全不知道是什麼。

以下是最簡單的一種解釋說法:

```
let name //當name從來未被定義完成，不知道其類型
```

你問JavaScript: name是什麼?

JavaScript回答: name? 從來沒聽過? 我不知道你在說什麼。

```
let name = null //name沒有值
```

你問JavaScript: name是什麼?

JavaScript回答: 我不知道

以上的回答來自: Why is null an object and what's the difference between null and undefined?

比較運算

null與undefined作比較運算時，在值的比較(==)是相等的，而在值與類型比較(===)時，是不相等的，這是能夠比較出兩個差異的地方。

```
null  == undefined // true
null === undefined // false
```

而在`typeof`運算符回傳的類型時，目前null的類型仍然是`object`，而不是`null`。關於這一點要特別注意，有人或許它是個bug，但目前仍然沒有更動。

```
typeof null        // object (bug in ECMAScript, should be null)
typeof undefined   // undefined
```

未定義(undefined)與"眞正的根本完全沒有定義"而會造成錯誤中斷的程式碼有區別，見以下的範例:

```
let a
console.log(a) //undefined
console.log(typeof a) //undefined
console.log(typeof b) //undefined

//注意，此行會因錯誤中斷之後的程式碼
console.log(b) //Error: b is not defined
```

用途

經常的用途是，null會用來當作一種運算後的特殊情況回傳值。而未定義(undefined)則常使用的是加上`typeof`運算符的判斷方式。這裡只是讓你能大概理解這兩個原始資料類型的涵意。

雖然JavaScript並沒有明確規範這兩個資料類型的用途，但對JavaScript程式設計師而言，應儘量使用空值(null)來進行運算後判斷，而未定義(undefined)就留給JavaScript系統用。

註: 要更深入理解請參考Exploring the Eternal Abyss of Null and Undefined)

家庭作業

作業一

Math.random()方法是在JavaScript中產生亂數的一個方法。不論你要作野球拳遊戲，還是線上博杯拿大獎的應用程式，都可以用它來產生亂數。`Math.random`會隨時產生一個0到1的浮點數，它的基本用法如下:

```
Math.floor(Math.random() * 6) + 1
```

- 上面的語句代表要產生1到6的整數亂數
- 1代表開始數字
- 6代表可能情況有6種

所以如果是以最大值max，最小值min來寫這個語句，這也是產生整數的亂數，會像下面這樣:

```
Math.floor(Math.random() * (max - min + 1)) + min
```

現在要利用這個產生亂數的方法，作一個線上抽獎的活動，客户說有下面幾個獎品:

1. 50吋液晶電視 1台
2. PS4遊戲機 3台
3. 手機行動充電器 10台
4. 7-11的100元購物券 100張

預計參與抽獎的人數有1萬人，要如何來寫這個程式的抽獎過程的應用程式？

作業二

有一家日本的房地產公司來台灣設立新的分公司，因爲日本的房地產用的計算單位與台灣的不同，所以要委託你寫一個程式，這個程式是希望能轉換平方公尺爲坪數。要如何來寫這個轉換的公式？

資料類型(值)

控制流程

控制流程

由於程式碼的執行順序，是由最上方的程式碼開始，往下逐行執行。有些時候，我們需要在其中特定的位置，判斷在當時執行期間的情況值，來決定之後程式執行的走向，這稱之爲控制流程(Control Flow/康錯 佛肉/)的語法結構。

舉例來說，電腦中的文字冒險類遊戲(例如: 美少女遊戲Galgame)中，主人翁在關鍵時刻必須對攻略對象，在故事進行的對話選項間作出選擇，依照一連串不同的選擇，有可以導向最後是好的結局(GoodEnding)或是壞的結局(BadEnding)。這種依選項(判斷情況)不同，而導致不同的執行結果的程式碼語法結構，就是控制流程的結構。

Expression(表達式)

Expression/依斯伯累遜/ (表達式)是以字面文字(literal/立的羅/)、變數/常數名稱、值(運算元)、運算符，或其他Expression(表達式)的組合體，最終能運算而計算求出(evaluates)一個值。

> Expression(表達式)代表任何合法的可計算產出值的程式碼單位

簡單的表達式範例如下，這看起有點像某行程式碼的一部份，或只是單純的字面文字(固定值的意思，例如 數字 或 字串):

```
'Hello'
3.1415
x = 7
3 + 5
```

Side Effect(副作用)

你如果有去醫院看過醫生拿過藥，在藥包上都會註明吃了這個藥物後會產生的副作用，Side Effect/塞的 依費特/(副作用)就是這個意思。Side Effect(副作用)這個詞，經常會出現在很多進階使用的JavaScript框架或函式庫之中，它算是一種概念，在表達式中有使用這個概念來進行分類。Expression(表達式)就上面所說的，是用來求出值的，但常見的一些Expression(表達式)的作用在指定值，而當在指定原本的變數/常數的Expression(表達式)值時，如果會變動原本的變數/常數內的值，就稱爲是"具有**Side Effect(**副作用**)**"的Expression(表達式)，例如像下面這些具有副作用的表達式範例:

```
counter++
x += 3
y = "Hello " + name
```

沒有副作用的表達式的範例，它們大概都只是字面文字(literal)或變數/常數名稱，或是產生一個新的運算結果值:

```
3 + 5
true
1.9
x
x > y
```

```
'Hello World'
```

Side Effect(副作用)並不是指好或不好的意思，而是它有可能會影響到其他環境的使用情況，在使用時要特別注意這樣。它除了在表達式中有這個概念，我們在函式中也會再見到它。這裡就先大概了解表達式是如何區分有無副作用的。

Statements(語句)

Statements/斯疊門/(語句、陳述)是在程式語言中，一小段功能性的程式碼，語句中包含了關鍵字與合法的語法(Syntax/新泰斯/)。在JavaScript語言中，傳統上是以半形分號(;)作爲代表結束與分隔其他的語句，但也可以不使用半形分號(;)，而改以斷行來區分。撰寫一支程式，就如同在寫一篇文章時，其中會包含了各種描述語句。

Statements(語句)可視爲在JavaScript的最小獨立執行程式碼組合

```
//註解也是一種語句

//指定值也是一種語句
const x = 10

//block statement(區塊語句)
{
  statement_1
  statement_2
  ...
  statement_n
}

//function statement(函式語句)
function name() {
   [statements]
}
```

在JavaScript語言中，Expression(表達式)主要用來產生"值"，因爲它的功用很特殊，通常會獨立出來說明稱之爲Expression Statements(表達式語句)。

一般的Statements(語句)主要功能是執行動作或定義某種行爲，例如之前說過的註解(Comment)就是一種語句。

Statements(語句)還可以依不同情況的使用進行分類，以下列出:

- 控制流程(Control flow)
- 定義(Declarations)
- 函式與類別(Functions and classes)
- 迭代/迴圈(Iterations/依特瑞遜/)
- 其他(Others)

註: 此分類參考MDN的分類方式，ECMAScript標準分類並不是這樣。

註: Iterations/依特瑞遜/這個字詞，中文一般是翻譯爲"迭代"這個字，這個中文字詞太過專業，對初學者來說是有看沒有懂。比較好的理解是它有"重覆作某件事"的意思，也就是和"迴圈"、"重覆"的意思相近。

控制流程

控制流程(flow control)語句是一種特殊的語句，它與程式的執行流程有關，會因爲其中的判斷結果不同，導致不同的執行結果。

區塊(block)語句

區塊(block)語句可以組合多行語句，或用於分隔不同語句，區塊(block)語句經常使用於控制流程的語句之中。使用花括號(curly brackets)標記({})，將多行語句包含在其中:

```
{
  statement_1
  statement_2
  ...
  statement_n
}
```

if...else語句

if/伊芙/...else/埃額斯/是常見的控制流程的語句，在中文的意思就是"如果...要不然"。所以整體的結構就像是中文的意思"如果aaa命題爲眞情況下，就xxx，要不然就ooo"。這個aaa就是一個判斷情況(condition)，判斷情況使用的都是比較運算符(Comparison operators)與邏輯運算符(Logical Operators)，而判斷情況結果則是會用布林的 true/觸/ 與 false/佛斯/ 值來判斷，但要特別注意之前說明的"falsy/佛西/"情況，這些在判斷情況中會自動轉成布林的false。

一個簡單的if...else語句的範例如下:

```
const x = 10

if (x > 100) {
    console.log('x > 100')
} else {
    console.log('x < 100')
}
```

> 註: if...else並不是寫了if就一定要有else，也可以單獨只用if語句，但else不能單獨使用。

if...else語句有幾個常見的延申結構，例如多個判斷情況時可以再其中加入else if。不過當你在寫這些判斷情況時，最好是讓它有眞的會出現的情況，以免寫出根本不會有的判斷情況，成爲程式碼的多餘部份:

```
const x = 10

if (x > 100) {
    console.log('x > 100')
} else if ( x < 50){
    console.log('x < 100 but x > 50')
} else {
    console.log('x < 50')
}
```

另一種是巢狀的語法結構，也就是進入某個判斷情況下，可以在裡面再寫出另一個if...else語句，例如:

```
const x = 10

if (x > 100) {
    if (x > 500){
        console.log('x > 500')
    }else{
        console.log('x > 100 but x < 500')
    }
} else if ( x < 50){
    console.log('x < 100 but x > 50')
} else {
    console.log('x < 50')
}
```

在判斷後的執行語句，如果是只有一行的情況下，就可以不需要使用區塊語句({})。也可以用空一格然後把判斷情況與結果語句寫在同一行，不過要注意這裡不要寫過長的語句，以免造成閱讀上的困難。例如:

```
const x = 10

//去除block statement
if (x > 100)
    console.log('x > 100')
else
    console.log('x < 50')

//寫成一行
if (x === 10) console.log('x is 10')
```

而在使用`if`加上`else`時，爲了簡化語句，使用三元運算符(Ternary Operator)(?:)，來讓程式碼更簡潔，這也只能用於簡單的判斷情況與執行語句時:

```
const x = 10

(x > 100) ? console.log('x > 100') : console.log('x < 50')
```

> 注意: 建議只使用三元運算符在單一行的語句上，雖然它可以使用在多行與巢狀結構。

由於三元運算符的語法很簡單，它也常被配合用在指定值的語句中，例如:

```
const foo = value1 > value2 ? 'baz' : null
```

這個像我們之前說過的，用邏輯或運算符(||)來指定變數/常數預設值的語句。不過如果是單純的判斷某個值是否存在，然後設定它爲預設值，用邏輯或運算符(||)是比較好的作法，例如:

```
//相當於 const foo = a ? a : b
const foo = a || b
```

> 心得提示: 因爲人生還很長，而我們需要寫的程式碼還有很多，所以你會本書上看到有很多感覺上像是偷懶或密技的簡短語法，或許它並不是出現在標準中，也不是課堂上會教的"正規"寫法，但的確是廣泛被使用的一些語法。

比較運算符

比較運算符我們在前面的內容中已經有介紹一部份，還有一些相等比較的概念。以下將它大略分成2個分類來說明:

- 相等比較運算: 相等(Equality)(==)，完全一致(Identity)(===)。不相等(Inequality)(!=)，不完全一致(Non-identity)(!==)。
- 關係比較運算: 大於(>)，小於(<)，大於等於(>=)，小於等於(<=)

相等比較運算時，需要遵守標準中所定義的"抽象相等比較演算法"((==)與(!=)符號的比較)與"嚴格相等比較演算法"((===)與(!==)符號的比較)的規定。抽象相等比較演算法((==)與(!=)符號的比較)的規則如下(假設x是左邊的運算子，而y是右邊的運算子):

- Type(x)與Type(y)的類型相同時，則進行嚴格相等比較
- 數字與字串比較時，字串會自動轉成數字再進行比較
- 其中有布林值時，`true`轉爲1，`false`轉爲+0
- 當物件與數字或字串比較時，物件會先轉爲預設值以及轉成原始資料類型的值，再作比較

至於嚴格相等比較演算法(也就是(===)與(!==)符號的比較)，由於一定要比較原始資料類型的類型，只要類型不同就一定是回傳爲`false`。除了類型相同，值也要相等，才會有回傳`true`的情況，什麼值轉換成什麼值再比較就根本不需要。

註: 物件的情況會比較特別，我們會在說明物件類型的章節再詳細說明。

你如果之前已經有"falsy"與"truthy"的概念，就會理解到很多比較運算的最終結果，都會以這個爲基礎轉變爲布林的`false`或`true`值。

在`if`中的判斷情況表達式中可以直接運算出布林值，這就是依照"falsy"與"truthy"的概念。或是再搭配 邏輯反相Logical NOT符號(!)來回傳反轉的布林值，例如:

```
if (undefined) console.log('true') //false

if (null) console.log('true') //false

if (+0) console.log('true') //false

if (!'') console.log('!\'\' true') //true

if (123) console.log('123 true') //true

//下面的還沒教到，是空物件與空陣列
const a = {}
const b = []

if (a) console.log(a, 'true')
if (b) console.log(b, 'true')
```

邏輯運算符

邏輯運算符實際上在前面的課程都已經介紹過了，只有三個而已:

- 邏輯與Logical AND(&&)
- 邏輯或Logical OR(||)
- 邏輯反相Logical NOT(!)

邏輯與Logical AND(&&) 與 邏輯或Logical OR(||) 兩個符號可以組合多個不同的比較運算，然後以邏輯運算的"與"與"或"來作最後的布林值的運算。不過要注意的是，它們的運算回傳值，在JavaScript中並非布林值，而是最終的值，轉換爲布林值是判斷情況中的作用。

註: 邏輯與Logical AND(&&)的結果只有同時爲眞時才能爲眞。也就是"眞&&眞 爲 眞"，其他都爲"假"。 註: 邏輯或Logical OR(||)的結果只要其一爲眞，結果就爲眞。

這兩個運算符通常會搭配 群組運算符(())，也就是括號標記(())，這在數學運算上是用來作爲優先運算的區分，或是用來區隔不同的比較運算。以下爲範例:

```
const x = 50
const y = 60
const z = 100

if ((x > 10) && (x < 100)) console.log(true)
if (( ((x + y) > 10) || (x === 50) ) && (z == 100)) console.log(true
```

註: (&)符號英文爲And或Amphersand。(|)符號的英文在電腦上通常稱爲Pipe，管道的意思。(!)在英文中稱爲Exclamation mark，驚嘆號的意思。

風格指引

- (Airbnb 15.1) 優先使用(===)與(!==)而非(==)與(!=)
- (Airbnb 15.2) 像if的條件語句內會使用ToBoolean的抽象方法強制轉爲布林類型，遵循以下規範：
  - 物件 轉換爲 true
  - Undefined 轉換爲 false
  - Null 轉換爲 false
  - 布林 轉換爲 該布林值
  - 數字 如果是 +0, -0, 或 NaN 則轉換爲 false，其他的皆爲 true
  - 字串 如果是空字串 '' 則轉換爲 false，其他的皆爲 true
- (Airbnb 15.3) 使用簡短寫法（註: 範例中這兩種判斷情況的簡短寫法很常見)

```
// 壞寫法
if (name !== '') {
  // ...stuff...
}

// 好的寫法
if (name) {
  // ...stuff...
}

// 壞寫法
if (collection.length > 0) {
  // ...stuff...
}

// 好的寫法
if (collection.length) {
  // ...stuff...
}
```

- (Airbnb 15.6) 不應該使用巢狀的三元運算語句，一般都只會用單行的表達式。

- (Airbnb 15.7) 避免不必要的三元運算語句。

```
// 壞寫法
var isYes = answer === 1 ? true : false

// 好寫法
var isYes = answer === 1

// 壞寫法
var isNo = answer === 1 ? false : true

// 好寫法
var isNo = answer !== 1

// 壞寫法
var foo = bar ? bar : 1

// 好寫法
var foo = bar || 1
```

- (Airbnb 16.1) 具有多行語句的區塊，需要使用花括號(braces)({})框起來

```
// 壞寫法
if (test)
  return false

// 好寫法
if (test) return false

// 好寫法
if (test) {
  return false
}

// 壞寫法
function foo() { return false }

// 好寫法
function bar() {
  return false
}
```

- (Airbnb 16.2) 如果你在多行區塊時使用if與else，將else放在if區塊的結尾花括號(})後的同一行。

```
// 壞寫法
if (test) {
  thing1()
  thing2()
}
else {
  thing3()
}
```

```
// 好寫法
if (test) {
  thing1()
  thing2()
} else {
  thing3()
}
```

switch語句

switch/斯餵取/語句有一種講法與用法，是相當於多個if...else的組合語句，也就是用於判斷情況有很多種不同的回傳值的組合情況，不過這個理解是有些問題。實際上，它最常被使用的是在完全一致相等值(===)比較的情況下，很少用在相關比較運算的情況下。基本上它的語法結構如下:

```
switch (expression) {
  case value1:
    //符合運算得到value1的執行語句
    break
  case value2:
    //符合運算得到value2的執行語句
    break
  ...
  case valueN:
    //符合運算得到valueN的執行語句
    break
  default:
    //符合運算得到其他值的執行語句
    break
}
```

舉一個簡單的範例來說，像下面這樣的if...else語句:

```
const x = 10

if (x > 100){
  console.log('x > 100')
} else if (x < 100 && x >50) {
  console.log('x < 100 && x >50')
} else {
  console.log('x < 50')
}
```

轉換爲switch語句會變成像下面這樣。switch(true)代表當case/K斯/中的比較運算需要爲布林值的true時，才能滿足而執行其中包含的語句:

```
const x = 10

switch (true) {
    case (x >100):
        console.log('x > 100')
        break
    case (x < 100 && x >50):
        console.log('x < 100 && x >50')
```

```
        break
    default:
        console.log('x < 50')
        break
}
```

這相當於下面這個if...else的語句:

```
const x = 10

if ((x > 100) === true) {
    console.log('x > 100')
}else if ((x < 100 && x >50) === true) {
    console.log('x < 100 && x >50')
} else {
    console.log('x < 50')
}
```

而switch語句最常被使用的情況，是用於判斷相等值的情況下，在這時候case語句裡的比較結果的相等於完全一致(Identity)(===)的比較結果，在滿足的情況下才會執行包含在case其中的執行語句，例如:

```
const x = 10

switch (x) {
    case 100 :
        console.log('x is 100')
        break
    case 50:
        console.log('x is 50')
        break
    case 10:
        console.log('x is 10')
        break
    default:
        console.log(x)
        break
}
```

> 注意: 因爲case語句中的值會以一致相等運算(===)來比較，所以資料類型也要完全相等才行，上面的範例如果把case 10:改爲case '10':，就會輸出10，而不是x is 10

break/博累克/關鍵字雖然在語法上是可選的，但case語句沒有搭配break會一直執行到出現break或是switch整個語句的結尾，這會出現不正確的結果，像下面這個範例:

```
const x = 50

switch (x) {
    case 100 :
        console.log('x is 100')
        break
    case 50:
        console.log('x is 50')
        //這邊少一個break
```

```
    case 10:
        console.log('x is 10')
        break
    default:
        console.log(x)
        break
}
```

最後的結果會是像下面這樣，通常這個結果並不是我們想要的:

```
x is 50
x is 10
```

註: 這算是一個有陷阱的設計，在eslint的no-fallthrough規則頁面上可以看到更多的說明

註: 所以不管如何，只要case語句中有包含其它需要執行的語句，一定需要以break作爲結尾。但最後一個case或default語句可以不需要break。

判斷時有多個case情況而執行同一個語句時，會使用像下面這個範例的語法，這個語法結構也是很常見的用法，例如:

```
const fruit = '芒果'

switch (fruit)
{
   case '芭樂':
   case '香蕉':
       console.log(fruit, '是四季都出產的水果')
       break
   case '西瓜':
   case '荔枝':
   case '芒果':
   default:
       console.log(fruit, '是只有夏季出產的水果')
       break
}
```

風格指引

- (Airbnb 15.5)當case與default區塊中包含了字面宣告時（例如：let、const、function 及class）時使用花括號({})來建立區塊語句。(註: 這是爲了要區隔出每個case中的各自字面宣告區塊，以免造成重覆宣告的錯誤)

```
// 壞寫法
  switch (foo) {
    case 1:
      let x = 1
      break
    case 2:
      const y = 2
      break
    case 3:
      function f() {}
      break
```

```
    default:
      class C {}
}

// 好寫法
switch (foo) {
  case 1: {
    let x = 1
    break
  }
  case 2: {
    const y = 2
    break
  }
  case 3: {
    function f() {}
    break
  }
  case 4:
    bar()
    break
  default: {
    class C {}
  }
}
```

- (Airbnb 18.3) 在控制語句(if、while等等)的開頭花括號符號({)前面多空一格空白，函式的呼叫或定義則不需要。
- `defalt`語句習慣固定是放在`switch`語句中的最後一段的位置，雖然它也不是一定要放在那裡。
- `switch`語句的`case/default`語句，假使是位於最後一段(通常是`default`語句)，它的`break`在功能上並非必要，習慣加上只是爲了讓程式碼更具一致性。

英文解說

Expression名詞，有"表情"、"表達"的意思，表情通常指的是人臉部的表情，這個字詞常常用在專業技術性的呈現或表達。至於像常會用的"表情符號"並不是這個字詞，英文字詞是使用emoticon或emoji。emoticon是emotion加上icon的新字詞，emotion則有"情感"、"情緒"的意思，近似於feeling(感受)的意思。Expression Emoticon這個組合字詞，通常是指"和人臉部表情有關的表情符號"，哭哭笑笑一類。不過表情符號的範圍比較廣，而且有很多是和動作、物品或標識有關。

Statement名詞，有"聲明"、"語句"、"陳述"等意思，它也可以當作財務或商業上的"報表"、"結算表"來使用，看起來這個字詞是用在正經八百的文件內容或訊息上。另外，它的字根是state，這個字的動詞有"宣佈"、"聲明"、"規定"的意思，名詞還有"州"或"國家"的意思，在電腦中最常拿來作"狀態"的名詞使用。

Literal名詞，有"文字"、"字面"的意思，也就是"按照字面上的意思就是這樣"。在電腦專業領域通常用這個字詞來作爲"固定值"(fixed value)的記號，例如String literal稱爲字串字面量，Numeric literal稱爲數字字面量，其他還有陣列、物件、函式的字面量。而相對於Literal(字面)的就是 變數/常數 記號。

結語

本章一開始說明了兩個重要的名詞定義，表達式(Expression)與語句(Statement)。前面所學的變數/常數，以及資料類型的部份，將組合成爲完整的程式碼語句的一部份。本章的重點是在於控制流程語句`if...else`，以及`switch`語句的應用，以及判斷情況的概念。

另外，在ES6後還有另一個可以用在控制流程的結構 - Promise(承諾)，由於它需要更多基礎的知識，我們將會在"特性"單元中，用獨立的一個章節來說明它。

家庭作業

作業一

公司交給你一個案子，要撰寫一支對網站上線上填寫表單進行檢查欄位的程式，以下是這個表單的欄位與要檢查的規則:

- 姓名(fullname): 最多4個字元，必填
- 手機號碼(mobile): 手機號碼必需是10個數字
- 出生年月日(birthday): 年1970-2000，月份1-12，日期1-31
- 住址(address): 最少8個字元，最多50個字元，必填
- Email(email): 最少10個字元，最多50個字元，必填

請問要如何寫出每個欄位的判斷檢查的程式碼。

迴圈

迴圈

迴圈(loop)是"重覆作某個事"的語句結構，用比較詳細的解釋，則是"在一個滿足的條件下，重覆作某件事幾次"這樣的語句。

迴圈語句經常會搭配陣列與物件之類的集合性資料結構使用，重覆地或循環地取出某些滿足條件的值，或是重覆性的更動其中某些值。

for語句

for語句需要三個表達式作爲參數，它們各自有其功用，分別以分號(;)作爲區分。for迴圈的基本語法如下:

```
for ([initialExpression]; [condition]; [incrementExpression])
  statement
```

- initialExpression: 初始(開始時)情況的表達式
- condition: 判斷情況或測試條件，代表可執行for中包含執行動作的滿足條件
- incrementExpression: 更新表達式，每次for迴圈中的執行動作語句完成即會執行一次這個表達式

一個簡單的範例如下:

```
for (let count = 0 ; count < 10 ; count++){
    console.log(count)
}
```

整個程式碼的解說是:

- 在迴圈一開始時，將count變數指定爲數字0
- 定義當count < 10時，才能滿足執行for中區塊語句的條件。也就是只有在count < 10時才能執行for中區塊語句。
- 每次當執行for中區塊的語句執行後，即會執行count++

可自訂的表達式們

for語句中共有三個表達式，分別代表迴圈進行時的設定與功用，這三個表達式的運用方式就可以視不同的應用情況而調整。所有的三個表達式都是可自訂的，也是可有可無。下面這個看起來怪異的for語句是一個無窮迴圈，執行這個語句會讓你的瀏覽器當掉:

```
//錯誤示範
for (;;){
    console.log('bahbahbah')
}
```

註: 用for(;;)這樣的語法，基本上失去for語句原本的意義，完全不建議使用。

注意: 迴圈是一個破壞性相當高的語句，如果不小心很容易造成整個程式錯誤或當掉。

第一個表達式是用作初始值的定義。它是可以設定多個定義值的，每個定義值之間使用逗號(,)作爲分隔，定義值可使用的範圍只在迴圈內部的語句中:

```
for (let count = 0, total = 10 ; count < 10 ; count++){
    console.log(count, total)
}
```

註: 初始值同樣也可以定義在迴圈之外，但它的作用範圍會擴大到迴圈之外。

第二個表達式是用於判斷情況(condition)，也就是每次執行迴圈中區塊的語句前，都會來測試(檢查)一下，是不是滿足其中的條件，如果滿足的話(布林`true`值)，就執行迴圈中區塊的語句，不滿足的話(布林`false`值)，就跳出迴圈語句或略過它。這個表達式就是會造成無止盡的或重覆執行次數不正確的原兇，所以它是這三個表達式中要特別注意的一個。

由於它是一個標準的判斷情況(condition)表達式，如果有多個判斷情況的時候，要使用邏輯與(&&)和邏輯或(||)來連接，邏輯或(||)因爲結果會是`true`的情況較爲容易，所以要更加小心:

```
for (let count = 0, total = 10 ; count < 10 && total < 20 ; count++){
    console.log(count, total)
}
```

第三個表達式是用於更新的，常被稱爲遞增表達式(incrementExpression)。不過它也不只用在遞增(increment)，應該說用在"更動(update)"或許會比較恰當，每次一執行完迴圈內部語句就會執行一次的表達式。它的作用相當於在迴圈內的區塊中的最後面加上這個語句，例如下面這個for迴圈和最上面一開始的範例的結果是一樣的:

```
for (let count = 0 ; count < 10 ;){
    console.log(count)
    count++
}
```

多個表達式也是可以的，同樣也是用逗點(,)分隔不同的表達式:

```
for (let count = 0, total = 30 ; count < 10 && total > 20 ; count++,
    console.log(count, total)
}
```

這裡有個小小問題，是有關於遞增運算符(++)與遞減運算符(--)，它們運算元的位置差異，放在運算元的前面和後面差異是什麼？例如以下的範例:

```
let x = 1
let y = 1

console.log(x++) //1
console.log(x) //2

console.log(++y) //2
console.log(y) //2
```

所以放在運算元(值)前面(先)的遞增(++)或遞減(--)符號，就會先把值作加1或減1，也就會直接變動值。放後面的就是值的變動要在下一次再看得到。

口訣: 遞增減運算符(++/--) "錢仙"=(放)前(面)先(變動)

小結

實際上這三個表達式並沒有限定只能用哪些表達式，只要是合法的表達式都可以，只不過第二個表達式會作判斷情況，也就是轉換爲布林值，這一點要注意的。有些程式設計師喜歡用這個特性寫出感覺很高超的語法，只是讓別人更難看懂在寫什麼而已，我完全不認同這種寫法，如何提供高閱讀性的程式碼才是最好的寫法。

至於每個表達式執行的情況，第一個表達式(初始化值)只會在讀取到for語句時執行一次，之後不會再執行。第二個表達式(判斷情況)會在讀取到for語句時執行一次，之後每次重覆開始時都會執行一次。第三個表達式會在有滿足條件下，執行到for語句區塊內部的語句的最後才執行。

多重迴圈

所謂的多重迴圈是巢狀的迴圈語句結構，也就是在迴圈的區塊語句中，再加上另一個內部的迴圈語句。典型的範例就是九九乘法表:

```
for (let i = 1 ; i < 10 ; i++){
  for (let j = 1 ; j < 10 ; j++){
    console.log(i + ' x ' + j + ' = ', i * j )
  }
}
```

> 註: 下面的while語句與do...while語句也可以寫爲多重迴圈(巢狀迴圈)，類似上面的程式碼，就不再多重覆說明了。

while語句

相較於需要很確定重覆次數(重覆次數爲可被計算的)的for迴圈，while迴圈語句可以說它是for迴圈語句的簡單版本，你可以根據不同的使用情況選擇要使用哪一種。它只需要一個判斷情況(condition)的表達式即可，它的基本語法結構如下:

```
while (condition)
  statement
```

一個簡單的範例如下，這個範例與之前的for迴圈的範例功能是相等的:

```
let count = 0

while (count < 10) {
    console.log(count)
    count++
}
```

while語句沒什麼特別要說明的，它把更新用表達式的部份交給程式設計師自行處理，在while語句中的區塊語句中再加上。使用時仍要避免出現無窮迴圈或重覆次數不正確的情況。

do...while語句

do...while語句是while語句的另一種變形的版本，差異只在於"它會先執行一次區塊中的語句，再進行判斷情況"，也就是會"保証至少執行一次其中的語句"，它的基本語法結構如下:

```
do
  statement
while (condition)
```

正常的使用情況與while語句沒什麼差異，只是位置上下顛倒再加個do而已:

```
let count = 0

do {
  console.log(count)
  count++
}
while (count < 10)
```

實際上while語句如果要相等於do...while的語句，應該是如下面這樣，也就是while語句前要先執行一次do...while中的語句才能算相等:

```
do_statment
while (condition)
  do_statment
```

因爲do...while語句具有先執行語句再檢查，也就是保証會執行一次的特性，要視應用情況來使用，以簡單的實際例子來說明while與do...while語句的應用情況:

- while: 當有一群人要進入電影院，需要對每個入場者收取門票與檢查，然後才能入場
- do...while: 當有幾組參賽者參加歌唱比賽，需要先表演後再對每個參賽者評分

for...in語句

for...in語句主要是搭配物件類型使用的，不要使用在陣列資料類型。它是用於迭代物件的可列舉的屬性值(enumerable properties)，而且是任意順序的。因爲陣列資料在進行迭代時，順序是很重要的，所以陣列資料類型不會用這個語句，而且陣列資料類型本身就有太多好用的迭代方法。

另外，for迴圈語句完全可以取代for...in語句，而且for迴圈語句可以保証順序，所以這個語句算是不一定要使用的，在這裡說明只是單純的比較其他的語句而已。

> 註: for...in語句是任意順序的迴圈語句，使用時通常是用來檢測用的語句

> 註: 對於複雜的物件資料類型，單純使用JavaScript中的語言特性是常常有所不足，建議是使用其他的函式庫中的API，例如jQuery、Lodash或underscore.js。

for...of語句

for...of語句是新的ES6語句，可以用於可迭代物件上，取出其中的值，可迭代物件包含陣列、字串、Map物件、Set物件等等。簡單的範例如下:

```
//用於陣列
let aArray = [1, 2, 3]

for (let value of aArray) {
  console.log(value)
}

//用於字串
let aString = "abcd";
```

```
for (let value of aString) {
  console.log(value);
}
```

break與continue

break(中斷)與continue(繼續)是用於迴圈區塊中語句的控制，在switch語句中也有看過break這個關鍵字的用法，是一個跳出switch語句的語法。

break是"中斷整個迴圈語句"的意思，可以中斷整個迴圈，也就是"跳出"迴圈區塊的語句，如果是在巢狀迴圈時是跳出最近的一層。break通常會用在迴圈語句中的if判斷語句裡，例如下面這個例子:

```
let count = 0

while (count < 10) {
    console.log(count)
    //count的值爲6時將會跳出迴圈
    if(count === 6) break
    count++
}
```

continue是"繼續下一次迴圈語句"的意思，它會忽略在continue之下的語句，直接跳到下一次的重覆執行語句的最開頭。因爲continue有"隱含的goto到迴圈的最開頭程式碼"，它算是一個在JavaScript語言中的壞特性，它會讓你的判斷情況(condition)整個無效化，也可以破壞整體的迴圈語句執行結構，容易被濫用出現不被預期的結果，結論是能不用就不要使用。一個簡單的範例如下:

```
let count = 0
let a = 0

while (count < 10) {
    console.log('count = ', count)
    console.log('a = ', a)

    count++
    //count的值大於爲6時將會不再遞增a變數的值
    if(count > 6) continue
    a++
}
```

註:　雖然JavaScript語言中並沒有goto語法，但迴圈語句中的continue搭配labeled語句，相當於其他程式語言中的goto語法。goto語法幾乎在所有的程式語言中，都是惡名昭彰的一種語法結構。

風格指引

- (Airbnb 18.3) 在控制語句的圓括號開頭(()前放一個空格(if, while等等)。在函式呼叫與定義的函式名稱與傳入參數之間就不需要空格。eslint: keyword-spacing jscs: requireSpaceAfterKeywords
- (Airbnb 7.3) 絕對不要在非函式區塊中宣告一個函式(if, while等等)。改用指定一個函式給一個變數。瀏覽器可以允許你可以這樣作，但是壞消息是，不同瀏覽器可能會轉譯爲不同的結果。eslint: no-loop-func

- (Google) for-in迴圈經常會錯誤地被使用在陣列上。當使用陣列時，總是使用一般的for迴圈。
- 在迴圈中區塊語句裡，定義變數/常數是個壞習慣，應該是要定義在迴圈外面或for迴圈的第一個表達式中。

```
//壞的寫法
for (let i=0; i<10; i++){
  let j = 0
  console.log(j)
  j++
}

//好的寫法
for (let i=0, let j = 0; i<10; i++){
  console.log(j)
  j++
}
```

- 用於判斷情況的運算，應該要儘可能避免，先作完運算在迴圈外部或或for迴圈的第一個表達式中，效率可以比較好些。

```
//較差的寫法
for (let i=0 ; i < aArray.length ; i++) {
  //statment
}

//較好的寫法
for (let i=0, len = aArray.length ; i < len; i++) {
  //statment
}

//另一種寫法，這種寫法腦袋要很清楚才看得懂
for (let i = aArray.length; i--;){
  //statment
}

//較差的寫法
let i = 0
while (i < aArray.length) {
  //statment
  i++
}

//較好的寫法
let i = 0
let len = aArray.length

while ( i < len) {
  //statment
  i++
}
```

註: 上面有一說法是在陣列資料類型的迴圈中的第三表達式(更新用表達式)中，使用i--會比i++效率更佳。實際上在現在的瀏覽器上這兩種的執行速度是根本一樣，沒什麼差異。

迴圈

結語

家庭作業

函式與作用域

函式與作用域

函式(function)

函式(function/訪遜/)是JavaScript的非常重要的特性。函式用於程式碼的重覆使用、資訊的隱藏與複合(composition)。我們經常會把一整組的功能程式碼，寫成一個函式，之後可以重覆再使用，JavaScript在執行時的呼叫堆疊也是以函式作爲單位。

> 注意: 依據ECMAScript標準的定義，函式的`typeof`回傳值是'function'，而不是'object'。由此可見在標準中定義的'object'類型，只是針對"單純"的物件定義而言，但具有函式呼叫(call)實作的物件，將會歸類爲'function'，因爲它們的內部都有建構函式的特性，例如Date、String、Object這些內建物件，它們的`typeof`回傳值都是'function'。`typeof`的回傳值只能作爲參考用，有很多複雜的應用下並沒有辦法辨別得出是什麼樣的物件。

> 註: 那麼要如何精確又有效的檢查一個變數/常數是否爲函式類型？請參考這篇How can I check if a javascript variable is function type?的問答。

函式定義

函式的基本語法結構如下:

```
//匿名函式
function() {}

//有名稱的函式
function foo() {}
```

函式的名稱也是一個識別符，命名方式如同變數/常數的命名規則。

而匿名函式並沒有函式的名稱，通常用來當作一個指定值，指定給一個變數/常數，被指定後這個變數/常數名稱，就成了這個函式的名稱。實際上，匿名函式也有其他的用法，例如拿來當作其他函式的傳入參數值，或是進行一次性執行。

函式使用`return`作爲最後的回傳值輸出，函式通常會有回傳值，但並非每種函式都需要回傳值，也有可能利用輸出的方式來輸出結果。以下兩種方式對於函式都是可以使用的宣告(定義)方式，使用帶有名稱的函式稱爲"函式定義"的方式，而另一種用變數/常數指定匿名函式的稱爲"函式表達式"的方式:

```
//函式定義 - 使用有名稱的函式
function sum(a, b){
    return a+b
}

//函式表達式 - 常數指定爲匿名函式
const sum = function(a, b) {
    return a+b
}
```

函式的呼叫是使用函式名稱加上括號(())，以及在括號中傳入對應的參數值，即可呼叫這個函式執行。例如:

```
const newValue = sum(100,0)
console.log(sum(99, 1))
```

ES6中有一種新式的函式語法，稱爲"箭頭函式(Arrow Function)"，使用肥箭頭符號(Flat Arrow)(=>)，它是一種匿名函式的縮短寫法，下面這個寫法相當於上面的sum函式定義:

```
const sum = (a, b) =>  a + b
```

箭頭函式(Arrow Function)因爲語法簡單，而且可以綁定this變數，所以算得上最受歡迎的ES6新功能，現在在很多程式碼中被大量使用。我們在特性篇中裡會專門有一章的內容來說明箭頭函式(Arrow Function)。

> 註: this變數與物件有關，在物件的章節會說明。

> 註: 箭頭函式在"特性篇"有一篇專門的教學文章

傳入參數

函式的傳入參數是需要討論的，它是函式與外部環境溝通的管道，也就是輸入資料的部份。

不過，你可能會在函式的"傳入參數"常會看到兩個不同的英文字詞，一個是parameter(或簡寫爲param)，另一個是argument，這常常會造成混淆，它們的差異在於:

- parameters: 指的是在函式的那些傳入參數名稱的定義。我們在文章中會以"傳入參數定義名稱"來說明。
- arguments: 指的是當函式被呼叫時，傳入到函式中眞正的那些值。我們在文章中會以"實際傳入參數值"來說明。

傳入參數預設值

關於函式的傳入參數預設值，在未指定實際的傳入值時，一定是undefined ，這與變數宣告後但沒有指定值很相似。而且在函式定義時，我們並沒有辦法直接限定傳入參數的資料類型，所以在函式內的語句部份，一開始都會進行實際傳入參數值的資料類型檢查。

有幾種方式可以用來在函式內的語句中，進行預設值的設定，例如用typeof的回傳值來判斷是否爲undefined，或是用邏輯或(||)運算符的預設值設定方式。用來指定預設值的範例:

```
//用邏輯或(||)
const link = function (point, url) {
    let point = point || 10
    let url = url || 'http://google.com'
    ...
}

//另一種設定的方式，typeof是回傳類型的字串值
const link = function (point, url) {
    let point = typeof point !== 'undefined' ? point : 10
    let url = typeof url !== 'undefined' ? url : 'http://google.com'
    ...
}
```

> 注意: 邏輯或(||)運算符設定預設值，雖然語法簡單，但有可能不精準。它會在實際傳入參數值只要是"falsy"，就直接指定爲預設值。

在ES6中加入了函式傳入參數的預設值指定語法，現在可以直接在傳入參數時就定義這些參數的預設值，這個作法是建議的用法:

```
const link = function (point = 10, url = 'http://google.com') {
    ...
}
```

> 註: 只有undefined的情況下才會觸發預設值的指定值。

> 註: 有預設值的參數在習慣上都是擺在傳入參數列表的"後面"，雖然這並不是個強制的作法只是個習慣。

以函式作爲傳入參數

前面有說明函式可以作爲變數/常數的指定值。不僅如此，在JavaScript中函式也可以當作實際傳入參數的值，將一個函式傳入到另一個函式中作爲參數值，而且在函式的最後也可以回傳函式。這種函式的結構稱之爲"高階函式(Higher-order function)"，是一種JavaScript程式語言的獨特特性，高階函式可以讓在函式的定義與使用上能有更多的彈性，它也延申出很多不同的應用結構。你可能常聽到JavaScript的callback(回呼、回調)結構，它就是高階函式的應用。

習慣上，因爲函式在定義時，它的傳入參數並沒辦法限定資料類型，所以當要定義傳入參數將會是個函式時，通常會用fn或func作爲傳入參數名稱，以此作爲辨別。不過，當你在撰寫一個函式時，最好是要加上傳入值的，以及回傳值的註解說明。以下爲一個簡單的範例:

```
const addOne = function(value){
    return value + 1
}

const addOneAndTwo = function(value, fn){
    return fn(value) + 2
}

console.log(addOneAndTwo(10, addOne)) //13
```

無名的傳入參數**(unnamed arguments)**

這是函式的一種隱藏機制，實際上對於傳入的參數值是有一個隱藏在背後的物件，名稱爲arguments，它會對傳入參數實際值進行保存，可以直接在函式內的語句中直接取用。arguments雖是一個物件資料類型，但它有"陣列"的一些基本特性，不過缺少大部份陣列中的方法，所以被稱作"pseudo-array"(僞陣列)。以下爲一個簡單的範例:

```
function sum() {
  return arguments[0]+arguments[1]
}

console.log(sum(1, 100))
```

不過，如果你在函式的傳入參數定義名稱中，使用了arguments這個參數名稱，或是在函式中的語句裡，定義了一個名稱爲arguments的自訂變數/常數名稱，這個隱藏的物件就會被覆蓋掉。總之它的行爲相當怪異，有一說是說它一開始設計時就錯了，隱藏的arguments物件

對開發者來說，並不像隱藏版的密技，而是比較像是隱藏版的陷阱。

註: 關於arguments的詳細介紹，可以參考The JavaScript arguments object…and beyond

不固定傳入參數**(Variadic)**與其餘**(rest)**參數

像下面這個範例中，原先sum函式中，定義了要有三個傳入參數，但如果眞正在呼叫函式時傳入的參數值(arguments)並沒有的情況下，或是多出來的時候，會發生什麼情況？

前面有說到，沒有預設值的時候會視爲undefined值，而多出來的情況，是會被直接略過。有的時候需要一種能夠"不固定傳入參數"的機制，在各種函式應用時，才能比較方便。

```
function sum(x, y, z) {
  return x+y+z
}

console.log(sum(1, 2, 3))  //6
console.log(sum(1, 2))    //NaN
console.log(sum(1, 2, 3, 4))  //6
console.log(sum('1', '2', '3'))  //123
console.log(sum('1', '2')) //12undefined
console.log(sum('1', '2', '3', '4'))  //123
```

雖然上一節有說過的，有個隱藏的arguments物件，它可以獲取到所有傳入的參數值，然後用類似陣列的方式來使用，但它的設計相當怪異(有一說是設計錯誤)，使用時要注意很多例外的情況，加上使用前根本也不需要定義，很容易造成程式碼閱讀上的困難。另外，在一些測試報告中，使用arguments物件本身比有直接使用具有名稱傳入參數慢了數倍。所以結論是，arguments物件的是不建議使用它的。

那麼，有沒有其他的機制可以讓程式設計師能處理不固定的傳入參數？

在ES6中加入了其餘參數(rest parameters)的新作法，它使用省略符號(ellipsis)(...)加在傳入參數名稱前面，其餘參數的傳入值是一個標準的陣列值，以下是一個範例:

```
function sum(...value) {
    let total = 0
    for (let i = 0 ; i< value.length; i++){
        total += value[i]
    }
    return total
}

console.log(sum(1, 2, 3))  //6
console.log(sum(1, 2))    //3
console.log(sum(1, 2, 3, 4))  //10
console.log(sum('1', '2', '3'))  //123
console.log(sum('1', '2')) //12
console.log(sum('1', '2', '3', '4'))  //1234
```

如果要寫得更漂亮、簡潔的的語法，直接使用Array(陣列)本身的好用方法，像下面這樣把原本的範例重寫一下:

```
function sum(...value) {
```

```
    return value.reduce((prev, curr) => prev + curr )
}
```

註: reduce(歸納)是陣列的方法之一，它可以用來作"累加"

其餘參數只是扮演好參數的角色，代表不確定的其他參數名稱，所以如果一個函式中的參數值有其他的確定傳入參數名稱，其餘參數名稱應該要寫在最後一個位子，而且一個函式只能有一個其餘參數名稱:

```
function(a, b, ...theArgs) {
  // ...
}
```

其餘參數與arguments物件的幾個簡單比較:

- 其餘參數只是代表其餘的傳入參數值，而arguments物件是代表所有傳入的參數值
- 其餘參數的傳入值是一個標準陣列，可以使用所有的陣列方法。而arguments物件是"僞"陣列的物件類型，不能使用陣列的大部份內建方法
- 其餘參數需要定義才能使用，arguments物件不需要定義即可使用，它是隱藏機制

內部(巢狀)函式

函式中的語句中，還可以包含其他的函式，這稱爲內部函式(inner)，或是巢狀函式(nested)的結構。以下爲一個簡單的範例:

```
function addOuter(a, b) {

    function addInner() {
        return a + b
    }

    return addInner()
}

addOuter(1, 2) //3
```

這樣的程式碼有點類似下面的寫法，不過你可以仔細的比較一下這兩個程式碼之間，傳入參數值的差異:

```
function addOuter(a, b) {
    return addInner(a, b)
}

function addInner(a, b) {
    return a + b
}

addOuter(1, 2) //3
```

內部函式可以獲取到外部函式所包含的環境值，例如外部函式的傳入參數、宣告變數等等。而內部函式又可以成爲外部函式的回傳值，所以當內部函式接收到外部函式的環境值，又被回傳出去，內部函式間接變成一種可以讓函式對外曝露包含在函式內部環境值的溝通管道，這種結構稱之爲"閉包(closure)"。

內部函式在JavaScript中被廣泛的使用，因爲它可以形成所謂"閉包"(closure)的結構。

註: "閉包"(closure)在特性篇中有一個獨立的章節來說明。

作用範圍(scope)

"作用範圍(scope)"或稱之爲"作用域"，指的是"變數或常數的定義與語句的可見(被存取得到)的範圍"，作用範圍可簡單區分爲本地端的(local)與全域的(global)。

JavaScript程式語言的作用範圍，基本上是使用"函式作用範圍(function scope)"的，或稱之以函式爲基礎(function-based)的作用範圍。也就是說只有使用函式才能劃出一個本地端的作用範圍，其他的區塊像if、for、switch、while等等，雖然有使用區塊語句({...})，但卻是無法界定出作用範圍。

因此，當你在所有函式的外面宣告變數或常數，這個變數或常數就會變成"全域的"作用範圍的一員，稱爲"全域變數或常數"，也就是說在程式碼裡的任何地方都可以被存取得到。

註: 在JavaScript語言中，你應該要把"函式"也當作一種"值"來看待，它可以被指定到一個變數/常數，也可以作爲函式回傳值。而在作用範圍中的行爲也和"值"類似。

在ES6後的新作法，加入"區塊作用範圍(block scope)"概念，也就是使用具有區塊的語句，例如上述的if、for、switch、while等等，都可以劃分出作用範圍，這是一個很棒的改進。那要怎麼作呢？就是以`let`與`const`取代原本`var`的宣告變數的方式，`var`可以完全捨棄不使用。

如果是使用`var`來定義變數，程式碼中的變數x並不是在函式中定義的，所以會變爲"全域變數"：

```
if (true) {
  var x = 5
}

console.log(x) //5
```

對比使用`let`來宣告變數，程式碼中的y位於區塊中，無法在外部環境獲取得到：

```
if (true) {
  let y = 5
}

console.log(y) //y is not defined
```

其他函式特性

回調(callback)

回調(callback)是一種特別的函式結構，也因爲JavaScript具有"高階函式(Higher-order function)"的特性，意思是說在函式中可以用另一個函式當作傳入參數值，最後也可以回傳函式。

一般而言，函式使用回傳值(return)作爲最後的執行語句。但回調並不是，回調結構首先會定義一個函式類型的傳入參數，在此函式的最後執行語句，即是呼叫這個函式傳入參數，這個函式傳入參數，通常我們稱它爲回調(callback)函式。回調函式經常使用匿名函式的語法，直接寫在函式的傳入參數中。

```
function showMessage(greeting, name, callback) {
    console.log('you call showMessage')
    callback(greeting, name)
}

showMessage('Hello!', 'Eddy', function(param1, param2) {
    console.log(param1 + ' ' + param2)
})
```

由於回調函式是一個函式類型，通常會在使用它的函式中，作一個基本檢查，以免造成程式錯誤，本章節的最前面有說明過了，函式的`typeof`回傳值是'function'，以下爲改寫過的程式碼，改寫過後不論是不是有傳入回調函式，都可以正常運作，也就是回調函式變成是一個選項:

```
function showMessage(greeting, name, callback) {
     console.log('you call showMessage')

    if (callback && typeof(callback) === 'function') {
        callback(greeting, name)
    }
}
```

回調(callback)提供了使用此函式的開發者一種彈性的機制，讓程式開發者可以自行定義在此函式的最後完成時，要如何進行下一步，這通常是在具有執行流程的數個函式的組合情況。實際上，回調函式實現了JavaScript中非同步(asynchronously)的執行流程，這使得原本只能從頭到底(top-to-bottom)的程式碼，可以在同時間執行多次與多種不同的程式碼。在實際應用情況時，回調結構在JavaScript程式中大量的被使用，它也變成一種很明顯的特色，例如以下的應用中很常見:

- HTML中的DOM事件
- AJAX
- 動畫
- Node.js

提升**(Hoisting)**

簡單的來說，提升是JavaScript語言中的一種執行階段時的特性，也是一種隱性機制。不過，沒先定義與指定值就使用，這絕對是個壞習慣是吧？變數/常數沒指定好就使用，結果一定是不是你要的。

`var`、`let`和`const`會被提升其定義，但指定的值不會一併提升上去，像下面這樣的程式碼:

```
console.log(x) //undefined
var x = 5

console.log(y) //undefined
let y = 5
```

最後的結果出乎意料，竟然只是沒指定值的`undefined`，而不是程式錯誤。實際上這程式碼裡的變數被提升(Hoisting)了，相當於:

```
var x
console.log(x)
x = 5
```

```
let y
console.log(y)
y = 5
```

函式定義也會被提升，而且它變成可以先呼叫再定義，也就是整個函式定義內容都會被提升到程式碼最前面。不過這對程式設計師來說是合理的，在很多程式語言中都可以這樣作:

```
foo() //可執行

function foo(){
  console.log('Hello')
}
```

不過使用匿名函式的指定值方式(函式表達式, FE)，就不會有整個函式定義都被提升的情況，只有變數名稱被提升，這與上面的變數宣告方式的結果一致:

```
foo() //錯誤: foo is not a function

let foo = function(){
  console.log('Hello')
}
```

結論如下:

- 所有的定義(var, let, const, function, function*, class)都會被提升
- 使用函式定義時，在函式區塊中的這些定義也會被提升到該區塊的最前面
- 當函式與變數/常數同名稱而提升時，函式的優先程度高於變數/常數。
- 遵守好的風格習慣可以避免掉變數提升的問題

全域作用範圍污染

全域作用範圍污染(global scope pollution)，整體來說，也是壞的程式特性+壞的程式寫作習慣，所造成的不良後果。例如沒有經過`var`宣告的變數，會自動變爲全域作用範圍，或是是在把變數宣告在全域作用範圍中。過多的變數常常會造成記憶體無法回收，或是全域變數與函式中的變數常常互相衝突。

全域作用範圍污染在JavaScript中，一直是一個長久以來經常會發生的問題，尤其是在程式愈來愈龐大，整體的組織與結構沒有一開始就預作規劃時。ES6中針對作用範圍作了很多的標準上的改進，但不論程式特性如何進步，維護一個良好的寫作習慣，可能比程式本身的功能還重要幾百倍。

所以，好的程式設計師，在撰寫程式時應遵守一些建議的風格習慣，好好地理解作用範圍的概念，這樣就可以避免全域作用範圍污染情況發生。

匿名函式與**IIFE**

匿名函式還有另一個會被使用的情況，就是實現只執行一次的函式，也就是IIFE結構。IIFE是Immediately-invoked function expressions的縮寫，中文稱之爲"立即呼叫的函式表達式"，IIFE可以說是JavaScript中獨特的一種設計模式，它是被某些聰明的程式設計師研究出來的一種結構，它與表達式的強制執行有關，有兩種語法長得很像但功能一樣，這兩種都有人在使用:

```
(function () { … })()
(function () { … }())
```

IIFE在執行環境一讀取到定義時，就會立即執行，而不像一般的函式需要呼叫才會執行，這也是它的名稱的由來 - 立即呼叫，唯一的例外當然是它如果在一個函式的內部中，那只有呼叫到那個函式才會執行。

```
(function(){
    console.log('IIFE test1')
}())

function test2(){
    (function(){
        console.log('IIFE test2')
    }())
}

test2()
```

IIFE的主要用途，例如分隔作用範圍，避免全域作用範圍的污染，避免在區塊中的變數提升(Hoisting)。IIFE也可以再進而形成一種Module Pattern(模組模式)，用來封裝物件的公用與私有成員。許多知名的函式庫例如jQuery、Underscore、Backbone一開始發展時，都使用模組模式來作爲擴充結構。

不過，模組模式的結構存在於JavaScript已有很久一段時間，算是前一代主要的設計模式與程式碼組織方式，現在網路上看到的教學文，大概都是很有歷史的了。而現今的JavaScript已經都改爲另一種更具彈性的、更全面化的稱爲Module System(模組系統)的作法，例如AMD、CommonJS與Harmony(ES6標準的代號)，這在特性篇會有一個獨立的章節再作介紹。

純粹函式與副作用

在表達式的章節中，我們有講到在表達式中副作用(Side Effect)的分別，函式也有這種區分方式，不過對於函式來說，具有副作用代表著可能會更動到外部環境，或是更動到傳入的參數值。函式的區分是以 純粹(pure)函式 與 不純粹(impure)函式 兩者來區分，這不光只有無副作用的差異，還有其他的條件。

純粹函式(pure function)即滿足以下定義的函式:

- 給定相同的輸入(傳入值)，一定會回傳相同輸出值結果(回傳值)
- 不會產生副作用
- 不依賴任何外部的狀態

一個典型的純粹函式的例子如下:

```
const sum = function(value1, value2) {
  return value1 + value2;
}
```

套用上面說的定義，你可以用下面這樣理解它是不是一個純粹函式:

- 只要每次給定相同的輸入值(例如1與2)，就一定會得到相同的輸出值(例如3)
- 不會改變原始輸入參數，或是外部的環境，所以沒有副作用
- 不依賴其他外部的狀態(變數之類的)

那什麼又是一個不純粹的函式？看以下的範例就是，它需要依賴外部的狀態值(變數值):

```
let count = 1;
```

```
let increaseAge = function(value) {
  return count += value;
}
```

在JavaScript中不純粹函式很常見，像我們一直用來作爲輸出的console.log函式，或是你可能會在很多範例程式看到的alert函式，都是不純粹函式，這類函式因爲沒有回傳值，都是用來作某件事而已。像console.log會更動瀏覽器的主控台(外部環境)的輸出，也算是一種副作用。

純粹函式具有以下的優點:

- 程式碼可以簡單化，閱讀性提高
- 較爲封閉與固定，可重覆使用性高
- 易於單元測試(Unit Test)、除錯

不過有許多內建的或常用的函式都是免不了有副作用的，例如這些應用:

- 會改變傳入參數變數(物件、陣列)的函式
- 時間性質的函式
- I/O相關
- 資料庫相關
- AJAX

例如而每次輸出值都不同的不純粹函式一類，最典型的就是Math.random，這是產生隨機值的內建函式，既然是隨機值當然每次執行的回傳值都不一樣。

總之，並不是說有副作用的函式就不要使用，而且要理解這個概念，然後儘可能在你自己的寫的函式上使用純粹函式，以及讓必要有副作用的函式得到管理與控制。現在已經有一些新式的函式庫或框架，會特別要求在某些地方只能使用純粹函式，而具有副作用的不純粹函式只能在特定的情況下才能使用，這就需要先有這樣的概念與特別注意了。

風格指引

- (Airbnb 7.3、Google) 永遠不要在非函式區塊(if, while等等)裡面，定義一個函式。而是把函式指定給一個變數(註: 函式表達式)

- (Airbnb 18.3) 在控制語句(if, while等等)的圓括號(()開頭加上一個空白字元。函式在呼叫或定義時，函式名稱與傳入參數列則不需要空白。

- (idiomatic.js 7.B) 提前回傳可以增加程式碼可閱讀性，對於效率沒有明顯差異。

```
// 不好的寫法:
function returnLate( foo ) {
  var ret;

  if ( foo ) {
    ret = "foo";
  } else {
    ret = "quux";
  }
  return ret;
}

// 好的寫法:
```

```
function returnEarly( foo ) {

  if ( foo ) {
    return "foo";
  }
  return "quux";
}
```

常見問答

函式要用那種定義方式比較好？

由上面的內容來說，有三種定義函式的方式，一種是傳統的函式定義(FD)，另一種是用常數或變數指定匿名函式的方式(函式表達式, FE)，最後一種是新式的箭頭函式，這三種的範例如下:

```
//函式定義
function sum(a, b){
  return a+b
}

//函式表達式
const sum = function(a, b) {
  return a+b
}

//箭頭函式
const sum = (a, b) => a+b
```

那麼，到底那一種是比較建議的方式？

首先，由於第二種方式(函式表達式)完全可以被箭頭函式取代，箭頭函式又有另外的好處(綁定this)，所以它幾乎可以不用了。

而第一種方式(函式定義)有一些特點，所以它會被用以下的情況:

- 全域作用範圍
- 模組作用範圍
- Object.prototype的屬性值

函式定義的優點:

- 函式定義名稱可以加入到執行期間的呼叫堆疊(call stack)中，除錯方便
- 函式定義可以被提升，也就是可以在定義前呼叫(請參考上面的說明內容)

除此之外，都使用第三種方式，也就是箭頭函式。

> 請參考When should I use Arrow functions in ECMAScript 6?

arguments物件還要用嗎？

當然是不要用。這東西是有設計缺陷的，除非你對它很了解，不然用了也可能會出問題，不過話說回來，如果你對它很了解的話，就不會想用它了。改用"其餘參數"就可以了。

IIFE語法結構還要用嗎？

看情況而定。如果是有一些函式庫例如jQuery中的擴充方式會用到，這當然避免不了。如果是其他的已經採用新式的模組系統，可能是根本不需要。

英文解說

Scope/史溝波/　中文有"視野"、"導彈範圍"的意思。也就是相當於程式語言中，看不看得到(能不能存取得到)的意思。一般中文會翻譯成"作用域"或"作用範圍"，有點難懂的文言文。與作用範圍(scope)相關的還有一個Namespace(命名空間)，這是一種語法結構或組織方法，讓程式設計師可以把不同的識別符與程式碼敘述，放到不同的"空間"之中，以免造成衝突或混亂。不過，JavaScript語言中並沒有內建的命名空間(Namespace)的特性。

Context/康鐵斯/　，中文有"上下文"、"環境"的意思。在程式語言中指的是程式碼的執行上下文內容，它與作用範圍相關，但不相等於作用範圍。這個名詞有時會和Scope一起拿出來比較，在JavaScript語言中它也是很重要的一個概念。我們在物件中將會再說明它的意思。以下是基本的比較:

- Scope是屬於以函式爲基礎的(function-based)。而Context則是以物件爲基礎的(object-based)。
- Scope指的是在程式碼函式中的變數的可使用範圍。而Context指的是`this`，也就是指向擁有(或執行)目前執行的程式碼的物件。

家庭作業

陣列

陣列

陣列是一種有順序的複合式的資料結構，用於定義、存放複數的資料類型，在JavaScript的陣列中，並沒有規定它能放什麼資料進去，可以是原始的資料類型、其他陣列、函式等等。

陣列是用於存放大量資料的結構，要如何有效地處理資料，需要更加注意。它的搭配方法與語法很多，也有很多作相同事情的不同方式，並不是每樣都要學，有些只需要用到再查詢相關用法即可。

> 註:　雖然陣列資料類型是屬於物件，但Array這個包裝物件的`typeof　Array`也是回傳'function'

陣列定義

陣列定義有兩種方式，一種是使用陣列字面文字，以下說明定義的方式。

陣列的索引值(index)是從0開始的順序整數值，陣列可以用方括號([])來取得成員的指定值，用這個方式也可以改變成員包含的值:

```
const aArray = []
const bArray = [1, 2, 3]

console.log(aArray.length) //0

aArray[0] = 1
aArray[1] = 2
aArray[2] = 3
aArray[2] = 5

console.log(typeof aArray) // object
console.log(aArray) // [1,2,5]
console.log(aArray.length) //3
console.log(aArray[3]) //undefined
```

> 註:　陣列爲參照的(reference)資料類型，其中包含的值是可以再更改的，這與const的常數宣告無關。

另一種是使用Array包裝物件的預先分配空間的方式，但這種方式並不建議使用，這種定義語法除了容易搞混之外，經測試過它對效能並沒有太大幫助。而且這語法在分配後，一定會把長度值(成員個數)固定住，除非你百分之百確定陣列裡面的成員個數，不然千萬不要用。以下爲範例:

```
//警告: 不要使用這種定義方式

//預先分配的定義
const aArray = new Array(10)
//混淆語法，這是定義三個陣列值
const bArray = new Array(1, 2, 3)
```

```
console.log(aArray.length) //10
```

註: JavaScript的內建物件都不建議new作初始定義的。不過有一些特例是一定要的，例如Date、Error等等。

陣列定義注意事項

一開始就搞混的語法

注意初學者很容易犯的一個錯誤，就是用下面這種陣列定義語法，這語法並不會導致執行錯誤，這是很怪異的合法定義語法，特別在這裡指出來就是希望你不要搞混了:

```
//這是錯誤示範
const aArray = [10]
```

實際上這相當於:

```
const aArray = []
aArray[0] = 10
```

多維陣列

對JavaScript中的陣列結構來說，多維陣列指的是"陣列中的陣列"結構，只是在陣列中保存其他的陣列值，例如像下面的範例:

```
const magicMatrix = [
      [2, 9, 4],
      [7, 5, 3],
      [6, 1, 8]
]

magicMatrix[2][1]
```

維數過多時在處理上會愈複雜，一般常見的只有二維。通常需要另外撰寫專屬的處理函式，或是搭配額外的函式庫在容易上會較爲方便。

關聯陣列

JavaScript中並沒有關聯陣列(Associative Array)這種資料結構，關聯陣列指的是"鍵-值"的資料陣列，也常被稱爲字典(dictionary)的資料陣列，有很多程式語言有這樣的資料結構。

基本上在JavaScript中有複合類型只有物件和陣列，物件屬性的確是"鍵-值"對應的，但它並不是陣列，也沒有像陣列有這麼多方法可使用。雖然在ES6標準加入幾個特殊的物件結構，例如Set與Map，但目前的使用還不廣泛，其中支援的方法也少。在處理大量複合資料時，陣列的處理效率明顯高出物件許多，陣列的用途相當廣泛。

儲存多種資料類型

雖然並沒有規定說，你只能在同一個陣列中使用單一種資料類型。但是，在陣列中儲存多種不同的資料類型，絕對是個壞主意。在包含有大量資料的陣列中會嚴重影響處理效能，例如像下面這樣的例子。如果是多種資料類型，還不如先直接都用字串類型，需要取值時再作轉換。

```
var arr = [1, '1', undefined, true, 'true']
```

另外你需要注意的是，雖然數字類型在JavaScript中並沒有分浮點數或整數，但實際上在瀏覽器的JavaScript引擎(例如Google Chrome的V8引擎)中，整數的陣列的處理效率高於浮點數的陣列，可見其實引擎可以分辨各種不同的資料類型，然後會作最有效的儲存與運算，比你想像中聰明得很。

在ES6後加入了一種新式的進階資料結構，稱爲型別陣列(Typed Arrays)，它是類似陣列的物件，但並非一般的陣列，也沒有大部份的陣列方法。這種資料結構是儲存特定的資料時使用的，主要是爲了更有效率的處理二進位的資料(raw binary data)，例如檔案、圖片、聲音與影像等等。(註: Typed Arrays標準)

不過，就像上面一段說明的，聰明的JavaScript引擎在執行時會認得在一般陣列中儲存的資料類型，然後作最有效率的運算處理，在某些情況型別陣列(Typed Arrays)在運算上仍然不見得會比一般陣列還有效率。

多洞的陣列**(Holey Arrays)**或稀疏陣列**(Sparse Arrays)**

多洞的陣列代表你在定義陣列時，它的陣列值和索引(index)並沒有塞滿，或是從一個完整的陣列刪除其中一個(留了個空位)。像下面這樣的陣列定義:

```
const aArray = []
aArray[0] = 1
aArray[6] = 3

const bArray = [1, , 3]
```

另一種情況是，陣列大部份的值都是根本不使用的，例如用`new Array(100)`先定義出一個很大的陣列，但實際上裡面的值很少，這叫作稀疏陣列(Sparse Arrays)，其實多洞的陣列(Holey Arrays)或稀疏陣列(Sparse Arrays)都差不多指的是這一類的陣列，稀疏陣列可以再重新設計讓它在程式上的效率更高。這種陣列在處理效能上都會有很大的影響，在大的陣列中要避免使用到這樣的情況。

拷貝**(copy)**陣列

把原有的陣列指定給另一個變數(或常數)並不會讓它成爲一個全新的陣列，這是因爲當指定值是陣列時，是指定到同一個參照，也就是同一個陣列，看下面的範例:

```
const aArray = [1, 2, 3]
const bArray = aArray

aArray[0] = 100

console.log(bArray) //[100, 2, 3]
```

由範例可以看到，`bArray`與`aArray`是共享同一陣列中的值，就像是不同名字的連體嬰，不論你在`aArray`或`bArray`中修改其中的值，增加或減少其中的值，都是對同一值作這件事。這種不叫拷貝陣列，只是指向同一個陣列而已。

拷貝陣列並不是像這樣作的，而且它可能是件複雜的事情，複雜的原因是陣列中包含的值，可以是各種不同的值，包含數字、字串、布林這些基本原始資料，也可以是其他陣列、物件、函式、其他特殊物件。而且物件類型的資料，都是使用參照(reference)指向的，裡面的資料也是複合式的，所以有可能也是複雜的。題外話是當你在進行拷貝物件資料時，也是會遇到同樣的複雜的情況。

拷貝陣列的情況，大致可以區分爲淺拷貝(shallow copy)與深拷貝(deep copy)兩種。淺拷貝只能完全複製原陣列中，包含像數字、字串之類的基本原始資料值，而且當然是只有一維的平坦陣列，如果其中包含巢狀(多維)陣列、物件、函式、其他物件，只會複製到參照，意思是還是只能指向原來同一個值。

深拷貝(deep copy)反而容易理解，它是眞正複製出另一個完全獨立的陣列。不過，深拷貝是一種高花費的執行程序，所以對於效率與精確，甚至能包含的特殊物件範圍都需要考慮。如果要進行深拷貝，一般就不直接用JavaScript內建的語法來達成，而是要用外部的函式庫例如jQuery、understore或lodash來作，而這些函式庫中的深拷貝不是只有支援陣列結構而已，同樣也支援一般物件或特殊物件的資料結構。不過，如果你的陣列結構很簡單，也可以用for或while迴圈自己作深拷貝這件事。

以下的方式主要是針對"淺拷貝"部份，一樣也有很多種方式可以作同樣這件事，以下列出四種:

展開**(spread)**運算符

推薦使用

ES6後的新運算符，長得像之前在函式章節講到的其餘參數，使用的也是三個點的省略符號(ellipsis)(...)，語法相當簡單，現在很常被使用:

```
const aArray = [1, 2, 3]
const copyArray = [...aArray]
```

它也可以用來組合陣列

```
const aArray = [1, 2, 3]
const bArray = [5, 6, ...aArray, 8, 9]
```

註: 展開(spread)運算符目前用babel轉換爲ES5相容語法時，是使用`concat`方法

slice

slice(分割)原本是用在分割陣列爲子陣列用的，當用0當參數或不加參數，相當於淺拷貝，這個方式是目前是效率較好的方式，語法也很簡單:

```
const newArray = oldArray.slice(0)
const newArray = oldArray.slice()
```

concat

concat(串聯)是用於合併多個陣列用的，把一個空的陣列和原先陣列合併，相當於拷貝的概念。在這裡寫出來是爲了比較一下展開運算符:

```
const newArray = [].concat(oldArray)
```

for/while迴圈語句

迴圈語句也可以作爲淺拷貝，語句寫起來不難也很直覺，只是相較於其他方式要多打很多字，通常不會單純只用來作淺拷貝。以下爲範例程式:

```
const newArray = []
```

```
for (let i = 0, len = oldArray.length ; i < len ; i++){
    newArray[i] = oldArray[i]
}

const newArray = []
let i = oldArray.length

while (i--){
    newArray[i] = oldArray[i]
}
```

判別是否爲陣列

最常見的情況是，如果有個函式要求它的傳入參數之一爲陣列，而且確定不能是物件或其他類型。你要如何判斷傳進來的值是眞的一個陣列？

直接使用`typeof`來作判斷是沒辦法作這件事的，它對陣列資料類型只會直接回傳`'object'`。

在JavaScript中，有很多種方式可以作同一件事，這件事也不意外，不過每種方法都有一些些不同的細節或問題。以下的`variable`代表要被判斷的變數(或常數)值。

isArray

推薦使用

最簡單的判斷語法應該是這個，用的是內建Array物件中的`isArray`，它是個ES5標準方法:

```
Array.isArray(variable)
```

constructor

下面這個是在Chrome瀏覽器中效能最佳的判斷方法，它是直接用物件的建構式來判斷:

```
variable.constructor === Array
```

如果你是要判斷物件中的其中屬性是否爲陣列，你可以先判斷這個屬性是否存在，像下面這樣(prop指的是物件屬性):

```
variable.prop && variable.prop.constructor === Array
```

失效情況: 當使用在一個繼承自陣列的陣列會失效

instanceof

這也是用物件的相關判別方法來判斷，`instanceof`是用於判斷是否爲某個物件的實例，優點爲語法簡潔清楚:

```
variable instanceof Array
```

失效情況: 處理不同window或iframe時的變數會失效

toString.call

推薦使用

這也是用物件中的`toString`方法來判斷，這是所有情況都可以正確判斷的一種。它也是萬用方式，可以判斷陣列以外的其他特別物件，缺點是效率最差:

```
Object.prototype.toString.call(variable) === '[object Array]'
```

註: jQuery、underscore函式庫中的判斷陣列的API是用這種判斷方法

註: 在JavaScript: The Definitive Guide, 6th Edition書中有提到，`Array.isArray`其實就是用這個方式的實作方法。

方式結論

這幾個方式的選擇，我的建議是只要學最後一種就行了(不考慮舊瀏覽器就用第一種)，它可以正確判斷並應用在各種情況，有時候正確比再快的效能更重要，更何況它其實是萬用的，除了陣列之外也可以用於其它的判斷情況。雖然它的語法對初學者來說，可能無法在此時完全理解，不過就先知道要這樣用就行了。

參考資料: How do you check if a variable is an array in JavaScript?

陣列屬性與方法

陣列屬性與方法的細節多如牛毛，以下只列出常用到的方法與屬性。

屬性

length長度(成員個數)

`length`用來回傳陣列的長度(成員個數)，這個屬性有時候是不可信的，就如同上面用`new Array(10)`定義時，會被固定住爲10，不論現在的裡面的值有多少個。多洞的陣列中也是與目前有值成員的個數不同:

```
const aArray = [1, , undefined, '123'] //有洞的陣列
console.log(aArray.length) //4
```

多維陣列的情況上面有說明過了，只是陣列中的陣列而已，它的`length`只會回傳最上層陣列的個數:

```
const magicMatrix = [
      [2, 9, 4],
      [7, 5, 3],
      [6, 1, 8]
]

console.log(magicMatrix.length) //3
```

`length`的整數值竟然是可以更動的，它並不是只能讀不能寫的屬性:

```
const bArray = [1, 2, 3]
console.log(bArray.length)

bArray.length = 4
```

```
console.log(bArray.length)
console.log(bArray)

bArray.length = 2
console.log(bArray.length)
console.log(bArray)
```

更動length經測試過，事實上是從陣列最後面"截短(truncate)"的語法，它的效率是所有類似功能語法中最好的:

```
const sArray = ['apple', 'banana', 'orange', 'mongo']

sArray.length = 2
console.log(sArray)
```

另外，length指定爲0也可以用於清空陣列，清空陣列一樣也是有好幾種方式，以下爲各種清空陣列的範例程式碼，注意第一種的原陣列不能使用const宣告，就意義上它不是眞的把原來的陣列清空。一般情況下第一種效率最好，第四種最差:

```
let aArray = ['apple', 'banana', 'orange', 'mongo']

//第一種
aArray = []

//第二種
aArray.length = 0

//第三種
while(aArray.length > 0) {
    aArray.pop();
}

//第四種
aArray.splice(0, aArray.length)
```

方法

indexOf

indexOf是簡便的搜尋索引值用的方法，它可以給定一個要在陣列中搜尋的成員(值)，如果找到的話就會回傳成員的索引值，沒找到就會回傳"-1"數字。多個成員符合的話，它只會回傳最先找到的那個(一律是從左至右)，它的比對是使用完全符合的比較運算符(===)，可加入第二個參數，這是可選擇的，它是"開始搜尋的索引值"，如果是負整數則從最後面倒過來計算位置的(最後一個索引值相當於-1)。

```
const aArray = ['a', 'b', 'c', 'a', 'c', 'c']

console.log(aArray.indexOf('a')) //0
console.log(aArray.indexOf('c')) //2
console.log(aArray.indexOf('c', 3)) //4
console.log(aArray.indexOf('a', -3)) //3，-3代表要從索引值3開始搜尋
```

pop與push、shift與unshift

副作用方法

陣列的傳統處理方法，pop是"砰出"最後面一個值，然後把這個值從陣列移除掉。push是"塞入"一個值到陣列最後面。shift與pop類似，不過它是砰出最前面的值。unshift則與push類似，它是塞到陣列列前面。

pop的例子如下，它會回傳被砰出的值:

```
const aArray = [1, 2, 3]
const popValue = aArray.pop()

console.log(aArray) //[1,2]
console.log(popValue) //3
```

push的例子如下，它則是回傳新的長度值:

```
const aArray = [1, 2, 3]
aArray.push(4)

console.log(aArray) //[1,2,3,4]

const pushValue = aArray.push(5)

console.log(aArray) //[1,2,3,4,5]
console.log(pushValue) //5
```

口訣記法:　有"p"的pop與push是針對陣列的"屁股"(最後面)。pop-corn是爆米花，所以pop用來爆出值的。有u的push與unshift是同一掛的。

concat

concat(串聯)是用於合併其他陣列或值，到一個陣列上的方法。它最後會回傳一個新的陣列。

語法: array.concat(value1[, value2[, ...[, valueN]]])

前面已經有看到它可以作陣列的淺拷貝，它與展開運算符可以互爲替代，以下爲一個簡單的範例:

```
const strArray = ['a', 'b', 'c']
const numArray = [1, 2, 3]
const aString = 'Hello'

const newArray = strArray.concat(numArray)
console.log(newArray)

//連鎖(chain)運算
const newArray1 = strArray.concat(numArray).concat(aString)
console.log(newArray1)

//展開運算符 相等的作法
const newArray2 = [...strArray, ...numArray]
console.log(newArray2)
```

注意:　陣列沒有運算符這種東西。但是字串類型可以用合併運算符(+)，或是用同名方法concat作合併。

slice

slice(分割)是用於分割出子陣列的方法，它會用淺拷貝(shallow copy)的方式，回傳一個新的陣列。這個方法與字串中的分割子字串所使用的的同名稱方法`slice`，類似的作法。

語法: array.slice(start[, end])

`slice`使用陣列的索引值作爲前後參數，大部份的重點都是這兩個索引值的正負值情況。以下是對於特殊情況的大致規則:

- 當開頭索引值爲undefined時(空白沒寫)，它會以0計算，也就是陣列最開頭。
- 當索引值有負值的情況下，它是從最後面倒過來計算位置的(最後一個索引值相當於`-1`)
- 只要遵守"開頭索引值"比"結束索引值"的位置更靠左，就有回傳值。
- `slice(0)`相當於陣列淺拷貝

```
const aArray = [1, 2, 3, 4, 5, 6]

const bArray = aArray.slice(1, 3)
const cArray = aArray.slice(0)
const dArray = aArray.slice(-1, -3)
const eArray = aArray.slice(-3, -1)
const fArray = aArray.slice(1, -3)
```

splice

副作用方法

splice(粘接)這個字詞與上面的slice(分割)長得很像，但用途不相同，這個方法是用於刪除或增加陣列中的成員，以此來改變原先的陣列的。

爲何會有這種用途？主要是要在陣列的中間"插入"幾個新的成員(值)，或是"刪掉"其中的幾個成員(值)用的。

語法: array.splice(start, deleteCount[, item1[, item2[, ...]]])

這個方法的參數值會比較多用起來會複雜些，先說明它的參數如下:

- start 是開始要作這件事的索引值，左邊從0開始，如果是負數則從右邊(最後一個)開始計算
- deleteCount 是要刪除幾個成員(值)，最少爲0代表不刪除成員(值)
- item1...　這是要加進來的成員(值)，不給這些成員(值)的話，就只會作刪除的工作，不會新增成員(值)。注意如果是陣列值，會變成巢狀(子)陣列成員(值)。

實際上有幾個基本的用法範例，splice通常會用在一些特定的情況，也可能會搭配搜尋語法或迴圈語句。以下爲常用的幾個範例:

插入一個新成員(值)在某個值之後

插入某個值之後需要先找出這個某個值的索引值，所以會用`indexOf`來找，不過如果是多個值的情況，就要用迴圈語句了，這是只有單個"某個值"的情況。下面的兩個範例的結果是相同的，所以你要在"某個值"的後面插入新成員(值)，"後面"代表新成員(值)的索引值還需要加1才

行。

```
const dictionary = ['a', 'b', 'd', 'e', 'f']

//先找到b的位置，等會要在b後面插入c
const bIndex = dictionary.indexOf('b')

//bIndex大於-1代表有存在，插入c，不刪除
if (bIndex > -1) {
    dictionary.splice(bIndex+1, 0, 'c')
}
```

用新成員(值)取代某個值

單個值的情況:

```
const dictionary = ['a', 'b', 'x', 'd', 'e']

//先找到x的位置，等會要用c來取代x
const xIndex = dictionary.indexOf('x')

//bIndex大於-1代表有存在，插入c，刪除x
if (bIndex > -1) {
    dictionary.splice(xIndex, 1, 'c')
}
```

多個值的情況，用迴圈語句:

```
const dictionary = ['x', 'b', 'x', 'x', 'b']

for (let i = 0, len = dictionary.length; i < len; i++){

    if (dictionary[i] === 'x'){
        dictionary.splice(i, 1, 'c')
    }
}
```

用於刪除成員(值)

註: 其實完全與取代範例幾乎一樣

單個值的情況:

```
const dictionary = ['a', 'b', 'x', 'd', 'e']

//先找到x的位置，等會要刪除
const xIndex = dictionary.indexOf('x')

//xIndex大於-1代表有存在，刪除x
if (xIndex > -1) {
    dictionary.splice(xIndex, 1)
}
```

多個值的情況，用迴圈語句:

```
const dictionary = ['x', 'b', 'x', 'x', 'b']

for (let i = 0, len = dictionary.length; i < len; i++){

    if (dictionary[i] === 'x'){
        dictionary.splice(i, 1)
    }
}
```

陣列與字串 **join**與**split**

join(結合)與split(分離)是相對的兩個方法，join(結合)是陣列的方法，用途是把陣列中的成員(值)組合爲一個字串，組合的時候可以加入組合時用的分隔符號(或空白字元)。

```
const aArray = ['Hello', 'Hi', 'Hey']
const aString = aArray.join()   //Hello,Hi,Hey
const bString = aArray.join(', ') //Hello, Hi, Hey
const cString = aArray.join('') //HelloHiHey
```

split(分離)是倒過來，它是字串中的方法，把字串拆解成陣列的成員，拆解時需要給定一個拆解基準的符號(或空白)，通常是用逗號(,)分隔，以下爲範例:

```
const monthString = 'Jan,Feb,Mar,Apr,May'
const monthArray1 = monthString.split(',') //["Jan", "Feb", "Mar", "

//以下爲錯誤示範
const monthArray2 = monthString.split()  //["Jan,Feb,Mar,Apr,May"]
const monthArray3 = monthString.split('') //["J", "a", "n", ",", "F",
```

迭代

forEach爲副作用方法

陣列的迭代可以單純地使用迴圈語句都可以作得到，但在ES6後加入幾個新的方法，例如forEach、map與reduce提供更多彈性的運用。

forEach

forEach類似於for迴圈，但它執行語句的是放在回調函式(callback)的語句中，下面這兩個範例是相等功能的:

```
const aArray = [1, 2, 3, 4, 5]
for (let i = 0, len = aArray.length; i < len; i++) {
    // 對陣列元素作某些事
    console.log(i, aArray[i])
}

const aArray = [1, 2, 3, 4, 5]
aArray.forEach(function(value, index, array){
// 對陣列元素作某些事
console.log(index, value)
})
```

forEach的回調函式(callback)共可使用三個參數:

- value 目前成員的值
- index 目前成員的索引
- array 要進行尋遍的陣列

forEach雖然可以與for迴圈語句相互替代，但他們兩個設計原理不同，也有一些明顯的差異，在選擇時你需要考慮這些。不同之處在於以下幾點:

- forEach無法提早結束或中斷
- forEach被呼叫時，this會傳遞到回調函式(callback)裡
- forEach方法具有副作用

map(映射)

map(映射)反而是比較常被使用的迭代方法，由於它並不會改變輸入陣列(呼叫map的陣列)的成員值，所以並不會產生副作用，現在有很多程式設計師改用它來作爲陣列迭代的首選使用方法。

map(映射)一樣會使用回調函式(callback)與三個參數，而且行爲與forEach幾乎一模一樣，不同的地方是它會回傳一個新的陣列，也因爲它可以回傳新陣列，所以map(映射)可以用於串接(chain)語法結構。

```
var aArray = [1, 2, 3, 4];
var newArray = aArray.map(function (value, index, array) {
    return value + 100
})

//原陣列不會被修改
console.log(aArray) // [1, 2, 3, 4]
console.log(newArray) // [101, 102, 103, 104]
```

reduce(歸納)

reduce(歸納)這個方法是一種應用於特殊情況的迭代方法，它可以藉由一個回調(callback)函式，來作前後值兩相運算，然後不斷縮減陣列中的成員數量，最終回傳一個值。reduce(歸納)並不會更動作爲傳入的陣列(呼叫reduce的陣列)，所以它也沒有副作用。

reduce(歸納)比map又更複雜了一些，它多了一個參數值，代表前一個值，主要就是它至少要兩個值才能進行前後值兩相運算。你也可以給定它一個初始值，這時候reduce(歸納)會從索引值的0接著取第二個值進行迭代，如果你沒給初始值，reduce(歸納)會從索引值1開始開始迭代。可以用下面這個範例來觀察它是如何運作的:

```
const aArray = [0, 1, 2, 3, 4]

const total = aArray.reduce(function(pValue, value, index, array){
    console.log('pValue = ', pValue, ' value = ', value, ' index = '
    return pValue + value
})

console.log(aArray) // [0, 1, 2, 3, 4]
console.log(total) // 10

//下面給定初始值10，注意它放的地方是在回調函式之後的參數中
const total2 = aArray.reduce(function(pValue, value, index, array){
    console.log('pValue = ', pValue, ' value = ', value, ' index = '
```

```
    return pValue + value
}, 10)

console.log(total2) // 20
```

按照這個邏輯，reduce(歸納)具有分散運算的特點，可以用於下面幾個應用之中:

- 兩相比較最後取出特定的值(最大或最小值)
- 計算所有成員(值)，總合或相乘
- 其它需要兩兩處理的情況(組合巢狀陣列等等)

排序與反轉 **sort與reverse**

sort

副作用方法

sort是一個簡單的排序方法，以Unicode字串碼順序來排序，對於英文與數字有用，但中文可能就不是你要的。以下爲簡單的範例:

```
const fruitArray = ['apple', 'mongo', 'cherry', 'banana' ]
fruitArray.sort()
console.log(fruitArray) //["apple", "banana", "cherry", "mongo"]

const fruitArray = ['蘋果', '芒果', '櫻桃', '香蕉' ]
fruitArray.sort()
console.log(fruitArray) //["櫻桃", "芒果", "蘋果", "香蕉"]
```

中文的排序一般來說只會有兩種情況，一種是要依big5編碼來排序，另一種是要依筆劃來排序，這時候需要在sort方法傳入參數中，另外加入比較的回調(callback)函式，這個回調函式中將使用localeCompare這個可以比較本地字串的方法，以下爲範例程式，其中本地(locale)的參數請參考locales argument:

```
const fruitArray = ['蘋果', '芒果', '櫻桃', '香蕉', '大香蕉', '小香蕉' ]

//使用原本的排序
fruitArray.sort()

console.log(fruitArray)
//["大香蕉", "小香蕉", "櫻桃", "芒果", "蘋果", "香蕉"]


fruitArray.sort(function(a, b){
  //zh-Hans-TW、zh-Hans-TW-u-co-big5han、pinyin等等參數同樣結果
  return a.localeCompare(b, 'zh-Hans-TW')
})

console.log(fruitArray)
//["大香蕉", "芒果", "蘋果", "香蕉", "小香蕉", "櫻桃"]

fruitArray.sort(function(a, b){
  //按筆劃從小到大排序
  return a.localeCompare(b, 'zh-Hans-TW-u-co-stroke')
})
```

```
console.log(fruitArray)
//["大香蕉", "小香蕉", "芒果", "香蕉", "蘋果", "櫻桃"]
```

註: 這個字串比較方法應該可以再最佳化其效率，有需要可以進一步參考其文件的選項設定

reverse

副作用方法

reverse(反轉)這語法用於把整個陣列中的成員順序整個反轉，就這麼簡單。以下爲範例:

```
const fruitArray = ['蘋果', '芒果', '櫻桃', '香蕉' ]
fruitArray.reverse()

console.log(fruitArray)
//["香蕉", "櫻桃", "芒果", "蘋果"]
```

另一個情況是字串中的字元，如果要進行反轉的話，並沒有字串中的reverse方法，要用這個陣列的reverse方法加上字串與陣列的互相轉換的split與join方法，可以使用以下的函式:

```
function reverseString(str) {
    return str.split('').reverse().join('');
}
```

過濾與搜尋 **filter**與**find/findIndex**

filter(過濾)是使用一個回調(callback)函式作爲傳入參數，將的陣列成員(值)進行過濾，最後回傳符合條件(通過測試函式)的陣列成員(值)的新陣列。它的傳入參數與用法與上面說的迭代方法類似，實際上也是另一種特殊用途的迭代方法:

```
const aArray = [1, 3, 5, 7, 10, 22]

const bArray = aArray.filter(function (value, index, array){
        return value > 6
    })

console.log(aArray) //[1, 3, 5, 7, 10, 22]
console.log(bArray) //[7, 10, 22]
```

find與findIndex方法都是在搜尋陣列成員(值)用的，一個在有尋找到時會回傳值，一個則是回傳索引值，當沒找到值會回傳undefined。它們一樣是使用一個回調(callback)函式作爲傳入參數，來進行尋找的工作，回調函式的參數與上面說的迭代方法類似。

findIndex與最上面說的indexOf不同的地方也是在於，findIndex因爲使用了回調(callback)函式，可以提供更多的在尋找時的彈性應用。以下爲範例:

```
const aArray = [1, 3, 5, 7, 10, 22]
const bValue = aArray.find(function (value, index, array){
        return value > 6
    })
const cIndex = aArray.findIndex(function (value, index, array){
        return value > 6
```

```
    })

console.log(aArray) //[1, 3, 5, 7, 10, 22]
console.log(bValue) //7
console.log(cIndex) //3
```

陣列處理純粹函式

改寫原有的處理方法(或函式)爲純粹函式並不困難，相當於要拷貝一個新的陣列出來，進行處理後回傳它。ES6後的展開運算子(...)可以讓語法更簡潔。

> 註: 以下範例並沒有作傳入參數的是否爲陣列的檢查判斷語句，使用時請再自行加上。

> 註: 下列範例來自Pure javascript immutable arrays，更多其他的純粹函式可以參考這裡的範例。

push

```
//注意它並非回傳長度，而是回傳最終的陣列結果
function purePush(aArray, newEntry){
  return [ ...aArray, newEntry ]
}

const purePush = (aArray, newEntry) => [ ...aArray, newEntry ]
```

pop

```
//注意它並非回傳pop的成員(值)，而是回傳最終的陣列結果
function purePop(aArray){
  return aArray.slice(0, -1)
}

const purePush = aArray => aArray.slice(0, -1)
```

shift

```
//注意它並非回傳shift的成員(值)，而是回傳最終的陣列結果
function pureShift(aArray){
  return aArray.slice(1)
}

const pureShift = aArray => aArray.slice(1)
```

unshift

```
//注意它並非回傳長度，而是回傳最終的陣列結果
function pureUnshift(aArray, newEntry){
  return [ newEntry, ...aArray ]
}

const pureUnshift = (aArray, newEntry) => [ newEntry, ...aArray ]
```

splice

這方法完全要使用slice與展開運算符(...)來取代，是所有的純粹函式最難的一個。

```
function pureSplice(aArray, start, deleteCount, ...items) {
  return [ ...aArray.slice(0, start), ...items, ...aArray.slice(star
}

const pureSplice = (aArray, start, deleteCount, ...items) =>
[ ...aArray.slice(0, start), ...items, ...aArray.slice(start + delet
```

sort

```
//無替代語法，只能拷貝出新陣列作sort
function pureSort(aArray, compareFunction) {
  return [ ...aArray ].sort(compareFunction)
}

const pureSort = (aArray, compareFunction) => [ ...aArray ].sort(com
```

reverse

```
//無替代語法，只能拷貝出新陣列作reverse
function pureReverse(aArray) {
  return [ ...aArray ].reverse()
}

const pureReverse = aArray => [ ...aArray ].reverse()
```

delete

刪除(delete)其中一個成員，再組合所有子字串:

```
function pureDelete (aArray, index) {
   return aArray.slice(0,index).concat(aArray.slice(index+1))
}

const pureDelete = (aArray, index) => aArray.slice(0,index).concat(a
```

英文解說

常見問題

爲什麼要那麼強調副作用？

副作用的概念早已經存在於程式語言中很久了，但在最近幾年才受到很大的重視。在過去，我們在撰寫這種腳本直譯式程式語言，最重視的其實是程式效率與相容性，因爲同樣的功能，不同的寫法有時候效率會相差很多，也有可能這是不同瀏覽器品牌與版本造成的差異。

但現在的電腦硬體已經進步太多，所謂的執行資源的限制早就與10年前不同。效率在今天的程式開發早就已經不是唯一的重點，更多其他的因素都需要加入來一併考量，以前的應用程式也可能只是小小的特效或某個小功能，現在的應用程式將會是很龐大而且結構複雜的。所以，程式碼的閱讀性與語法簡潔、易於測試、維護與除錯、易於規模化等等，都會變成要考量的其他重點。純粹函式的確是未來的主流想法，當一個應用程式慢慢變大、變複雜，純粹函式可以提供的好處會變成非常明顯，所以一開始學習這個概念是必要的。

參考

物件

物件

物件(Object)類型是電腦程式的一種資料類型，用抽象化概念來比喻爲人類現實世界中的物體。

在JavaScript中，除了原始的資料類型例如數字、字串、布林等等之外，所有的資料類型都是物件。不過，JavaScript的物件與其他目前流行的物件導向程式語言的設計有明顯的不同，它一開始是使用原型基礎(prototype-based)的設計，而其他的物件導向程式語言，大部份都是使用類別基礎(class-based)的設計。

在ES6之後加入了類別爲基礎的語法(是原型基礎的語法糖)，JavaScript仍然是原型基礎，但可以用類別語法建立物件與繼承之用，雖然目前來說，仍然是很基本的類別語法，但讓開發者多了另一種選擇。

物件在JavaScript語言中可分爲兩種應用層面來看:

- 主要用於資料的描述，它扮演類似關連陣列的資料結構，儲存"鍵-值"的成對資料。很常見到用陣列中包含物件資料來代表複數的資料集合。
- 主要用於物件導向的程式設計，可以設計出各種的物件，其中包含各種方法，就像已經介紹過的各種包裝物件，例如字串、陣列等等的包裝物件。

物件類型使用以屬性與方法爲主要組成部份，這兩種合稱爲物件成員(member):

- 屬性:　物件的基本可被描述的量化資料。例如水果這個物件，有顏色、產地、大小、重量、甜度等等屬性。
- 方法:　物件的可被反應的動作或行爲。例如車子這個物件，它的行爲有加速、煞車、轉彎、打方向燈等等的行爲或可作的動作。

物件定義方式

物件字面**(Object Literals)**

用於資料描述的物件定義，使用花括號(curly　　braces)({})作爲區塊宣告，其中加入無關順序的"鍵-值"成對值，屬性的值可以是任何合法的值，可以包含陣列、函式或其他物件。

而在物件定義中的"鍵-值"，如果是一般的值的情況，稱爲"屬性(property,　prop)"，如果是一個函式，稱之爲"方法(method)"。屬性與方法我們通常合稱爲物件中的成員(member)。

　註：屬性名稱(鍵)中也不要使用保留字，請使用合法的變數名稱

```
const emptyObject = {}

const player = {
    fullName: 'Inori',
    age: 16,
    gender: 'girl',
    hairColor: 'pink'
}
```

以如果你已經對陣列有一些理解的基礎下，物件的情況相當類似，首先在定義與獲取值上:

```
const aArray = []
const aObject = {}

const bArray = ['foo', 'bar']
const bObject = {
    firstKey: 'foo',
    secondKey: 'bar'
}

bArray[2] = 'yes'
bObject.thirdKey = 'yes'

console.log(bArray[2]) //yes
console.log(bObject.thirdKey) //yes
```

不過，對於陣列的有順序索引值，而且只有索引值的情況，我們會更加關心物件中"鍵"的存在，物件中的成員(屬性與方法)，都是使用物件加上點(.)符號來存取。上面的程式碼雖然在`thirdKey`不存在時，會自動進行擴充，但通常物件的定義是在使用前就會定義好的，總是要處於可預測情況下是比較好的作法。物件的擴充是經常使用在對現有的JavaScript語言內建物件，或是函式庫的擴充之用。

> 心得口訣: 對於初學者要記憶是用花括號({})來定義物件，而方括號([])來定義陣列，可以用口訣來快速記憶: 物(霧)裡看花。方陣快炮。
>
> 註: 存取物件中的成員(屬性或方法)，使用的是句點(.)符號，這已經在書中的很多內建方法的使用時都有用到，相信你應該不陌生。
>
> 註: 相較於陣列中不建議使用的`new Array()`語法，也有`new Object()`的語法，也是不需要使用它。
>
> 註: 物件內的成員(方法與屬性)的存取，的確也可以使用像`obj[prop]`的語法，有一些情況下會這樣使用，例如在成員(方法與屬性)還未確定的函式裡面使用，一般情況下爲避免與陣列的成員存取語法混淆，所以很少用。以下範例來自這裡:

```
const luke = {
  jedi: true,
  age: 28,
}

function getProp(prop) {
  return luke[prop]
}

const isJedi = getProp('jedi');
```

上面這種定義物件的字面文字方式，這是一種單例(singleton)的物件，也就是在程式碼中只能有唯一一個物件實體，就是你定義的這個物件。當你需要產生同樣內容的多個物件時，那又該怎麼作？那就是要用另一種定義方式了。

物件字面定義方式，通常單純只用於物件類型的資料描述，也就是只用於定義單純的"鍵-值"對應的資料，在裡面不會定義函式(方法)。而基於物件字面定義，發展出JSON(JavaScript Object Notation)的資料定義格式，這是現今在網路上作爲資料交換使用的一種常見的格式，在

特性篇會再對JSON格式作更多的說明。

類別(Class)

類別(Class)是先裡面定義好物件的整體結構藍圖(blue print)，然後再用這個類別定義，來產生相同結構的多個的物件實例，類別在定義時並不會直接產生出物件，要經過實體化的過程(new運算符)，才會產生眞正的物件實體。另外，目前因爲類別定義方式還是個很新的語法，在實作時除了比較新的函式庫或框架，才會開始用它來撰寫。以下的爲一個簡單範例:

> 註: 在ES6標準時，現在的JavaScript中的物件導向特性，並不是眞的是以類別爲基礎(class-based)的，這是骨子裡還是以原型爲基礎(prototype-based)的物件導向特性語法糖。

```
class Player {
    constructor(fullName, age, gender, hairColor) {
        this.fullName = fullName
        this.age = age
        this.gender = gender
        this.hairColor = hairColor
    }

    toString() {
        return 'Name: '+this.fullName+', Age:'+this.age
    }
}

const inori = new Player('Inori', 16, 'girl', 'pink')
console.log(inori.toString())
console.log(inori.fullName)

const tsugumi = new Player('Tsugumi', 14, 'girl', 'purple')
console.log(tsugumi.toString())
```

> 註: 注意類別名稱命名時要使用大駝峰(ClassName)的寫法

下面分別說明一些這個例子中用到的語法與關鍵字的重要概念，以及類別延伸的一些語法。

this

在這個物件的類別定義中，我們第一次眞正見到this關鍵字的用法，this簡單的說來，是物件實體專屬的指向變數，this指向的就是"這個物件實體"，以上面的例子來說，也就是當物件眞正實體化時，this變數會指向這個物件實體。this是怎麼知道要指到哪一個物件實體？是因爲new運算符造成的結果。

this變數是JavaScript的一個特性，它是隱藏的內部變數之一，當函式呼叫或物件實體化時，都會以這個this變數的指向對象，作爲執行期間的依據。

還記得我們在函式的章節中，使用作用範圍(Scope)來說明以函式爲基礎的檢視角度，在函式區塊中可見的變數與函式的領域的概念。而JavaScript中，另外也有一種上下文環境(Context)的概念，就是對於this的在執行期間所依據的影響，即是以物件爲基礎的的檢視角度。

this也就是執行上下文可以簡單用三個情況來區分:

1. 函式呼叫: 在一般情況下的函式呼叫，this通常都指向global物件。這也是預設情況。

2. 建構式(constructor)呼叫: 透過new運算符建立物件實體，等於呼叫類型的建構式，this會指向新建立的物件實例
3. 物件中的方法呼叫: this指向呼叫這個方法的物件實體

所以當建構式呼叫時，也就是使用new運算符建立物件時，this會指向新建立的物件，也就是下面這段程式碼:

```
const inori = new Player('Inori', 16, 'girl', 'pink')
```

因此在建構式中的指定值的語句，裡面的this值就會指向是這個新建立的物件，也就是inori:

```
constructor(fullName, age, gender, hairColor) {
        this.fullName = fullName
        this.age = age
        this.gender = gender
        this.hairColor = hairColor
    }
```

也就是說在建立物件後，經建構式的執行語句，這個inori物件中的屬性值就會被指定完成，所以可以用像下面的語法來存取屬性:

```
inori.fullName
inori.age
inori.gender
inori.hairColor
```

第3種情況是呼叫物件中的方法，也就是像下面的程式碼中，this會指向這個呼叫toString方法的物件，也就是inori:

```
inori.toString()
```

對於this的說明大致上就是這樣而已，這裡都是很直覺的說明。this還有一部份的細節與應用情況，在特性篇中有獨立的一個章節來說明this的一些特性與應用情況，this的概念在JavaScript中十分重要，初學者真的需要多花點時間才能真正搞懂。

建構式(constructor)

建構式是特別的物件方法，它必會在物件建立時被呼叫一次，通常用於建構新物件中的屬性，以及呼叫上層父母類別(如果有繼承的話)之用。用類別(class)的定義時，物件的屬性都只能在建構式中定義，這與用物件字面的定義方式不同，這一點是要特別注意的。如果物件在初始化時不需要任何語句，那麼就不要寫出這個建構式，實際上類別有預設的建構式，它會自動作建構的工作。

關於建構式或物件方法的多形(polymorphism)或覆蓋(Overriding)，在JavaScript中沒有這種特性。建構式是會被限制只能有一個，而在物件中的方法(函式)也沒這個特性，定義同名稱的方法(函式)只會有一個定義被使用。所以如果你需要定義不同的建構式在物件中，因應不同的物件實體的情況，只能用函式的不定傳入參數方式，或是加上傳入參數的預設值來想辦法改寫，請參考函式內容中的說明。以下爲一個範例:

```
class Option {
    constructor(key, value, autoLoad = false) {
        if (typeof key != 'undefined') {
            this[key] = value
```

```
        }
        this.autoLoad = autoLoad
    }
}

const op1 = new Option('color', 'red')
const op2 = new Option('color', 'blue', true)
```

私有成員

JavaScript截至ES6標準爲止，在類別中並沒有像其他程式語言中的私有的(private)、保護的(protected)、公開的(public)這種成員存取控制的修飾關鍵字詞，基本上所有的類別中的成員都是公開的。雖然也有其他"模擬"出私有成員的方式，不過它們都是複雜的語法，這裡就不說明了。

目前比較簡單常見的區分方式，就是在私有成員(或方法)的名稱前面，加上下底線符號(_)前綴字，用於區分這是私有的(private)成員，這只是由程式開發者撰寫上的區分差別，與語言本身特性無關，對JavaScript語言來說，成員名稱前有沒有有下底線符號(_)的，都是視爲一樣的變數。以下爲簡單範例:

```
class Student {
    constructor(id, firstName, lastName) {
        this._id = id
        this._firstName = firstName
        this._lastName = lastName
    }

    toString() {
        return 'id is '+this._id+' his/her name is '+this.firstName+
    }
}
```

註:　如果是私有成員，就不能直接在外部存取，要用getter與setter來實作取得與修改值的方法。私有方法也不能在外部呼叫，只能在類別內部使用。

getter與setter

在類別定義中可以使用get與set關鍵字，作爲類別方法的修飾字，可以代表getter(取得方法)與setter(設定方法)。一般的公開的原始資料類型的屬性值(字串、數字等等)，不需要這兩種方法，原本就可以直接取得或設定。只有私有屬性或特殊值，才需要用這兩種方法來作取得或設定。getter(取得方法)與setter(設定方法)的呼叫語法，長得像一般的存取物件成員的語法，都是用句號(.)呼叫，而且setter(設定方法)是用指定值的語法，不是傳入參數的那種語法。以下爲範例:

```
class Option {
    constructor(key, value, autoLoad = false) {
        if (typeof key != 'undefined') {
            this['_' + key] = value;
        }
        this.autoLoad = autoLoad;
    }

    get color() {
```

```
      if (this._color !== undefined) {
        return this._color
      } else {
        return 'no color prop'
      }
    }

    set color(value) {
      this._color = value
    }
}

const op1 = new Option('color', 'red')
op1.color = 'yellow'

const op2 = new Option('action', 'run')
op2.color = 'yellow'
```

註: 所以getter不會有傳入參數，setter只會有一個傳入參數。

靜態成員

靜態(Static)成員指的是屬於類別的屬性或方法，也就是不論是哪一個被實體化的物件，都共享這個方法或屬性。而且，實際上靜態(Static)成員根本不需要實體化的物件來呼叫或存取，直接用類別就可以呼叫或存取。JavaScript中只有靜態方法，沒有靜態屬性，使用的是static作爲方法的修飾字詞。以下爲一個範例:

```
class Student {
    constructor(id, firstName, lastName) {
        this.id = id
        this.firstName = firstName
        this.lastName = lastName

        //這裡呼叫靜態方法，每次建構出一個學生實體就執行一次
        Student._countStudent()
    }

    //靜態方法的定義
    static _countStudent(){
      if(this._numOfStudents === undefined) {
          this._numOfStudents = 1
      } else {
          this._numOfStudents++
      }
    }

    //用getter與靜態方法取出目前的學生數量
    static get numOfStudents(){
      return this._numOfStudents
    }

}
```

物件

```
const aStudent = new Student(11, 'Eddy', 'Chang')
console.log(Student.numOfStudents)

const bStudent = new Student(22, 'Ed', 'Lu')
console.log(Student.numOfStudents)

const cStudent = new Student(33, 'Horward', 'Liu')
console.log(Student.numOfStudents)
```

靜態屬性目前來說有兩種解決方案，一種是使用ES7的Class Properties標準，可以使用`static`關鍵字來定義靜態屬性，另一種是定義到類別原本的定義外面:

```
// ES7語法方式
class Video extends React.Component {
  static defaultProps = {
    autoPlay: false,
    maxLoops: 10,
  }
  render() { ... }
}

// ES6語法方式
class Video extends React.Component {
  constructor(props) { ... }
  render() { ... }
}

Video.defaultProps = { ... }
```

註: ES7的靜態(或類別)屬性的轉換，要使用bebal的stage-0 perset。

繼承

用extends關鍵字可以作類別的繼承，而在建構式中會多呼叫一個`super()`方法，用於執行上層父母類別的建構式之用。`super`也可以用於指向上層父母類別，呼叫其中的方法或存取屬性。

繼承時還有有幾個注意的事項:

- 繼承的子類別中的建構式，`super()`需要放在第一行，這是標準的呼叫方式。
- 繼承的子類別中的屬性與方法，都會覆蓋掉原有的在父母類別中的同名稱屬性或方法，要區爲不同的屬性或方法要用`super`關鍵字來存取父母類別中的屬性或方法

```
class Point {
    constructor(x, y) {
        this.x = x
        this.y = y
    }
    toString() {
        return '(' + this.x + ', ' + this.y + ')';
    }
}

class ColorPoint extends Point {
```

```
    constructor(x, y, color) {
        super(x, y)
        this.color = color
    }
    toString() {
        return super.toString() + ' in ' + this.color
    }
}
```

物件相關方法

instanceof運算符

instanceof運算符用於測試某個物件是否由給定的建構式所建立，聽起來可能會覺得有點怪異，這個字詞從字義上看起來應該是"測試某個物件是否由給定的類別所建立"，但要記得JavaScript中本身就沒有類別這東西，物件的實體化是由建構函式，組成原型鏈而形成的。以下爲一個簡單的範例:

```
const eddy = new Student(11, 'Eddy', 'Chang')
console.log(eddy instanceof Student) //true
```

> 註: instanceof運算符並不是100%精確的，它有一個例外情況是在處理來自HTML的frame或iframe資料時會失效。

> 註: 關於原型鏈的說明請見特性篇的"原型物件導向"章節

物件的拷貝

在陣列的章節中，有談到淺拷貝(shallow copy)與深拷貝(deep copy)的概念，同樣在物件資料結構中，在拷貝時也同樣會有這個問題。陣列基本上也是一種特殊的物件資料結構，其實這個概念應該是由物件爲主的發展出來的。詳細的內容就不多說，以下只針對淺拷貝的部份說明:

Object.assign()

> 推薦方式

Object.assign()是ES6中的新方法，在ES6前都是用迴圈語句，或其他的方式來進行物件拷貝的工作。Object.assign()的用法很直覺，它除了拷貝之外，也可以作物件的合併，合併時成員有重覆名稱以愈後面的物件中的成員爲主進行覆蓋:

```
//物件拷貝
const aObj = { a: 1, b: 'Test' }
const copy = Object.assign({}, aObj)
console.log(copy) // {a: 1, b: "Test"}

//物件合併
const bObj = { a: 2, c: 'boo' }
const newObj = Object.assign(aObj, bObj)
console.log(newObj) //{a: 2, b: "Test", c: "boo"}
```

> 註: null或undefined在拷貝過程中會被無視。

> 註: 如果需要額外的擴充(Polyfill)可以參考Object.assign(MDN)，或是ES2015

Object.assign() ponyfill

JSON.parse加上JSON.stringify

不建議的方式

JSON是使用物件字面文字的定義方式，延伸用來專門定義資料格式的一種語法。它經常用來搭配AJAX技術，作爲資料交換使用，也有很多NoSQL的資料庫更進一步用它改良後，當作資料庫裡的資料定義格式。

這個方式是把物件定義的字面文字字串化，然後又分析回去的一種語法，它對於物件中的方法(函式)直接無視，所以只能用於只有數字、字串、陣列與一般物件的物件定義字面:

```
const aObj = { a: 1, b: 'b', c: { p : 1 }, d: function() {console.log
```

```
const aCopyObj = JSON.parse(JSON.stringify(aObj))
console.log(aCopyObj)
```

```
const bCopyObj = Object.assign({}, aObj)
console.log(bCopyObj)
```

這方式其實不推薦使用，爲什麼會寫出來的原因，是你可能會看到有人在使用這個語法，會使用這個語法的主要原因以前沒有像`Object.assign`這麼簡單的語法。除此之外，你可能還可以找到各種物件拷貝的各種函式或教學文件。

物件的拷貝的使用原則與陣列拷貝的說明類似，要不就使用`Object.assign`，要不然就使用外部函式庫例如jQuery、underscore或lodash中拷貝的API。

物件屬性的"鍵"或"值"判斷

undefined判斷方式

直接存取物件中不存在的屬性，會直接回傳`undefined`時，這是最直接的判斷物件屬性是否存在的方式，也是最快的方式。不過它有一個缺點，就是當這個屬性本身就是`undefined`時，這個判斷方法就失效了，如果你本來要的值本來就絕對不是`undefined`，所以可以這樣用。

```
//判斷鍵是否存在
typeof obj.key !== 'undefined'
```

```
//判斷值是否存在
obj.key !== undefined
obj['key'] !== undefined
```

註: 這個語法也可以判斷某個方法是否存在於物件中。

in運算符 與 **hasOwnProperty**方法

推薦使用 hasOwnProperty方法

這兩個語法在正常情況下，都是可以正確回傳物件屬性的"鍵"是否存在的判斷:

```
obj.hasOwnProperty('key')
```

```
'key' in obj
```

它們還是有明顯的差異，hasOwnProperty方法不會檢查物件的原型鏈(prototype chain，或稱之爲原型繼承)，也就是說hasOwnProperty方法只會檢查這個物件中有的屬性鍵，用類別定義時的方法是沒辦法檢測到，由原型繼承的方法也沒辦法檢測到，以下爲範例:

```
const obj ={}
obj.prop = 'exists'

console.log(obj.hasOwnProperty('prop') )
console.log(obj.hasOwnProperty('toString'))        // false
console.log(obj.hasOwnProperty('hasOwnProperty'))   // false

console.log('prop' in obj)
console.log('toString' in obj)
console.log('hasOwnProperty' in obj)
```

搭配物件類別定義使用時，hasOwnProperty的行爲是無法檢測出在類別中定義的方法，只能檢測該物件擁有的屬性，以及在建構式(constructor)中定義的物件擁有方法(算是一種具有函式值的屬性)。

```
class Base {
  constructor(a){
    this.a = a
    this.fnBase = function(){
      console.log('fnBase')
    }
  }

  baseMethod(){
    console.log('base')
  }

}

class Child extends Base{
  constructor(a, b){
    super(a)
    this.b = b
    this.fnChild = function(){
      console.log('fnChild')
    }
  }

  childMethod(){
    console.log('child')
  }

}

const aObj = new Child(1, 2)

console.log(aObj.hasOwnProperty('a'))
```

```
console.log(aObj.hasOwnProperty('b'))
console.log(aObj.hasOwnProperty('fnBase'))
console.log(aObj.hasOwnProperty('fnChild'))
console.log(aObj.hasOwnProperty('baseMethod')) //false
console.log(aObj.hasOwnProperty('childMethod')) //false

console.log('a' in aObj)
console.log('b' in aObj)
console.log('fnBase' in aObj)
console.log('fnChild' in aObj)
console.log('baseMethod' in aObj)
console.log('childMethod' in aObj)
```

hasOwnProperty由於只有判斷物件本身屬性的限制，它會比較常被使用，in運算符反而很少被用到。但這兩種判斷的效率都比直接用undefined判斷屬性值慢得多，所以要不就用undefined判斷就好，雖然這並不完全精準，要不然就用hasOwnProperty。

物件的遍歷(traverse)

在JavaScript中的定義，一般物件不是內建爲可迭代的(Iterable)，只有像陣列、字串與TypedArray、Map、Set這三種特殊物件，才是可迭代的。所以這種一般稱爲對物件屬性遍歷(traverse，整個走過一遍)或列舉(enumerate)的語句，而且一般物件的遍歷的效率與陣列的迭代相比非常的差。

> 註: for..of只能用在可迭代的(Iterable)的物件上。

for...in語句

for...in語句是用來在物件中以鍵(key)值進行迭代，因爲是無序的，所以有可能每次運算的結果會不同。它通常會用來配合hasOwnProperty作判斷，主要原因是in運算符和前面在判斷時一樣，它會對所有原型鏈(prototype chain)都整個掃過一遍，hasOwnProperty可以限定在物件本身的屬性。

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

> 註: for...in語句不要用在陣列上，它不適合用於陣列迭代。

Object.keys轉爲陣列，然後加上使用forEach方法

Object.keys方法會把給定物件中可列舉(enumerable)的鍵，組合成一個陣列回傳，它的結果情況和for...in語句類似，差異就是在對原型鏈並不會整個掃過，只會對物件擁有的屬性的鍵。

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

風格指引

- (Airbnb 22.3) 在命名類別或建構式時，使用大駝峰(PascalCase)命名方式。
- (Airbnb 9.4) 撰寫自訂的toString()方法是很好的，但要確定它是可以運作，而且不會有副作用的。
- (Airbnb 23.3) 如果 屬性/方法 是布林值，使用像isVal()或hasVal()的命名。

常見問題

物件導向程式設計在**JavaScript**語言中很重要嗎？

由物件導向程式設計以及週邊發展出的相關設計模式、API，到更進一步用於整體結構的程式框架，在現在流行的其他程式語言中，都是非常重要的程式語言特性。不過，在JavaScript中就很少有這類的發展情況，大致的原因有幾個:

- 以原型爲基礎的與類別爲基礎的物件導向的設計相當不同，許多程式開發的樣式是由函式發展而來，而非物件導向。
- JavaScript長久以上的執行環境是瀏覽器，首要任務是處理HTML的DOM結構，對於程式的執行效率與相容性爲首要重點，JavaScript對於函式的設計反而較具彈性與效率，而物件導向的程式是高消費的應用程式，使得程式設計師較專注於函式的設計部份，也就是功能性(函式)導向(functional oriented)程式設計，而非物件導向的程式設計。
- 長久以來，許多設計模式都是由函式發展出來的而非物件導向，目前較爲流行的許多工具函式庫，也都是以功能性(函式)導向程式設計，例如jQuery。另一方面，提倡以物件導向設計爲主的函式庫或框架，除了效率一定不會太好之外，學習門檻反而很高，造成會使用的程式設計師很少。

以下是幾個物件導向使用的現況:

- 物件字面定義的資料描述，用於單純的物件資料類型，常稱之爲只有資料的物件(data-only objects)
- 物件字面定義延伸出的JSON資料格式，成爲JavaScript用來作資料交換的格式
- 物件導向的相關特性與語法只會拿來應用現成的DOM或內建物件，或是用來擴充原本內建的物件之用
- 對於程式碼的組織方式與命名空間的解決方案，主要會使用模組樣式(即IIFE)或模組系統爲主要方式，而非物件導向的相關語法或模式
- 以合成(或擴充)代替繼承(composition over inheritance)

JavaScript中的物件可以多重繼承嗎？或是有像介面的設計？

沒有多重繼承，可以用合併多個物件產生新物件(合成 composition)、Mixins(混合)樣式，或是用ES6標準的Proxy物件，來達到類似多重繼承的需求。

介面或抽象類別也沒有，因爲這些就是類別爲基礎的物件導向才會有的概念。不過，有模擬類似需求的語法或樣式。

錯誤與例外處理

錯誤與例外處理

例外處理的思維是一種"求敗"的程式設計哲學，Exception(例外)明確來說有"異常"、"非預期"的意思。程式設計師應該考量許多異常或非預期的情況，在這些情況發生時，能夠加以控制或管理，這種處理稱之爲例外處理。

在程式設計學術領域中，實際上只有"例外處理(Exception Handling)"的說法，而沒有"錯誤處理(Error handling)"，因爲錯誤在定義上是無法處理的，例如在Java程式語言中，就有明確區分這兩種不同的錯誤與例外物件。

不過，在JavaScript語言中，Error(錯誤)與Exception(例外)基本上是同一種意思。這是因爲JavaScript一直以來是在一個資源受限的環境下執行(瀏覽器)，對它來說錯誤是在一個有限的範圍發生，所以在設計上也只有一個專門處理例外用的Error物件，在ECMAScript標準中並沒有很明確的區別。

但是，如果是在伺服器端的JavaScript(Node.js)程式，它的執行環境就不是使用有限的環境，在錯誤處理上必須考量更多情況，在整體的設計也會與客戶端程式有所分別，這部份會比較複雜。

註: 例外處理(Exception Handling) 維基百科

錯誤何時會發生

錯誤何時會發生？在JavaScript語言中有很多種不同的分類法，基本上我們可以把錯誤可分爲兩大類:

- 程式開發者的錯誤(Programmer errors): 其實就是程式臭蟲(bugs)，錯誤是程式開發者自己在撰寫程式碼中造成的，有很多狀況是開發者應避免而未避免，這種錯誤難以處理，需要開發者進一步修正。例如像以下的錯誤:
    - 嘗試讀取某個"undefined"的屬性
    - 用"string"類型傳入某個應該爲"object"的參數
- 運算的錯誤(Operational errors): 在執行階段發生的錯誤，在程式本身並沒有臭蟲(bugs)的情況，錯誤的發生源是來自系統本身，大部份是發生於程式在執行時與外部環境互動產生的異常情況，例如使用者輸入不合法、網路連線失敗、檔案讀取失敗或是記憶體問題等等。例如像以下的錯誤:
    - 不合法的使用者輸入
    - 伺服器無法連線
    - 記憶體超出限制

由於我們只能處理"運算的錯誤"而無法處理"程式開發者的錯誤"，大致的處理方式有幾個重點:

- 直接處理錯誤: 有些錯誤是可以直接處理的，例如第一次開啓檔案時，有可能檔案並不存在，這時候你在偵測到這個錯誤時，可以建立一個新的檔案。
- 傳播錯誤給你的客戶端: 如果你不知道如何處理這個錯誤情況，可以直接把這個錯誤傳播給你的客戶端。
- 嘗試重新操作: 有些錯誤是因爲網路連線造成的錯誤，通常可以在一段時間後，使用嘗

試重新操作來進行。
- 記錄錯誤，不作任何處理:　某些錯誤並沒有辦法立即處理，或是根本也不是能處理的情況，所以用這個方式先記錄來追蹤。

註: 以上說明來自Error Handling in Node.js的說法

Error物件

Error物件是JavaScript中內建的用於處理錯誤的物件，其中共分成6個種類，也就延伸出來6種不同的物件，每個不同的種類名稱即是這些物件的name屬性，可以用於不同的的錯誤情況，以下爲這些種類的說明:

- EvalError:　配合eval()方法使用所產生的錯誤，但因eval是一個不建議使用的全域方法，幾乎不會用到。
- SyntaxError:　這個也是配合eval()方法使用所產生的錯誤，幾乎不會用到。其他的語法錯誤將由瀏覽器直接回報。
- RangeError: 超出範圍所產生的的錯誤
- ReferenceError: 參照錯誤
- TypeError: 資料型態錯誤
- URIError: 配合encodeURI()或decodeURI()方法使用的錯誤

自訂的Error物件可以使用Error物件的建構式實體化，它只需要一個訊息字串作爲傳入參數，Error物件也是一個必定要使用new運算符進行實體化的內建物件，通常會配合throw語句使用，例如以下的範例:

```
try {
  throw new Error('Whoops!');
} catch (e) {
  console.log(e.name + ': ' + e.message);
}
```

使用Error物件有一些優點，有些瀏覽器品牌會針對Error物件作額外的擴充屬性或功能，讓它在除錯上更佳方便，例如stack(堆疊)這個屬性，它可以對目前Error物件發生的程式碼，列出呼叫的堆疊追蹤。在複雜的應用程式中，可以很快找出是位於某個程式碼中的呼叫所造成的錯誤。

不過，這些都算是Error物件中非標準的屬性，在每種瀏覽器品牌的實作情況的不一定，除了在開發階段使用，不建議使用非標準的屬性在正式的應用程式中。如果要使用相容各瀏覽器版本品牌的stack屬性，可以考慮使用外部的函式庫，例如TraceKit或stacktrace.js。

捕捉例外的try...catch語句

try...catch語句也是一種控制流程的語句。意思是如果所有位於try區塊中的語句都沒問題的話，就執行try其中的語句，當發生例外時會將控制權轉到catch區塊，執行catch區塊中的語句。catch可以在例外發生時，自動捕捉所有的例外情況，這在程式撰寫時相當方便。簡單的範例如下:

```
try{
    document.getElementById('test').innerHTML  = 'test'
} catch(e) {
    console.log(e) //TypeError: Cannot set property 'innerHTML' of nu
}
```

可以用if...else語句來類似的測試語句，不過這個時候不會自動捕捉錯誤，程式設計師要自己決定:

```
if(document.getElementById('test')){
    document.getElementById('test').innerHTML  = 'test'
} else {
    console.log(new TypeError('Cannot set property \'innerHTML\''))
}
```

try...catch語句最後還可以額外加上finally區塊，它是不論如何都會執行的語句區塊，例如你在try語句中開啓了一個檔案要進行處理，不論有沒有發生例外，最後需要把這個檔案進行關閉，這就是寫在finally區塊中。

try...catch語句聽起來似乎不錯，但在使用上的確需要再三考慮，尤其是在伺服器端的Node.js上，只能在必要的時候才會使用。在瀏覽器上的應用程式則是不需要考量那麼多。有幾個明顯的原因:

- 它是高消費的語句: 在有重大效能考量或迴圈的語句，不建議使用try...catch語句。
- 它是同步的語句: 如果是重度使用callback(回調)或promise樣式的異步程式，不需要使用它。此外，Promise語句可以完全取代它，而且是異步的語句。
- 不需要: 如果可以使用if...else的簡單程式碼中，將不會看到它的存在。另外，在一些對外取得資源的功能例如Ajax，我們一般都會使用額外函式庫來協助處理，這些函式庫都有考慮到比你想得到還完整的各種例外情況，所以也不需要由你親自來作例外處理。

那麼try...catch語句會使用在什麼情況下？通常會搭配會在例外發生時，直接丟出例外的方法上，這些方法有可能是JavaScript內建的，也可能是程式設計師自己設計的。最常見的是JSON.parse這個內建的用於解析JSON格式字串爲對應物件的方法，範例如下:

```
function validateData(jsonData){
    let data
    try{
      data = JSON.parse(jsonData);
    } catch(e) {
      console.log('Error data', e)
      data = null
    }
    return data
}

console.log(validateData('[1, 5, "false"]'))
console.log(validateData('1, 11'))
```

另外用於使用者輸入檢查的情況，這也是會用到try...catch語句的常見情況，這種通常稱之爲錯誤中心(Error Center)的方式，用於集中很多不同的例外爲一個語句中。以下爲範例程式:

```
function checkUserEntry()
{
    const userEntry = document.getElementById('email').value

    if (userEntry.length === 0) {
        throw new Error('請輸入字串')
    } else if (userEntry.indexOf('@') === -1) {
```

```
        throw new Error('請提供合法的Email住址')
    }

    document.getElementById('console').innerHTML =
    '<h3 style="color:yellow">你的Email是: </h3>' + userEntry
}

function validateEntry()
{
    try {
        checkUserEntry()
    } catch(e) {
        document.getElementById('console').innerHTML =
        '<h3 style="color:red"> 警告: </h3>' + e.message
    }
}

document.getElementById('clickme').addEventListener('click', validat
```

這是這個範例的html檔案內容:

```
<input type="text" id="email" />
<button id="clickme">點我</button>
<div id="console" style="height:200px; width:300px; background-color
```

丟出例外的throw

throw語句會"丟出"程式設計師自訂的例外，throw語句會中斷執行，所以在下面的語句並不會再被執行，然後把控制權轉交給呼叫堆疊(call stack)中第一個的catch區塊，如果沒有catch區塊則會立即停止應用程式。

throw語句後面雖然是可以使用任何的表達式，但一般使用上都會直接使用Error物件。以下爲簡單的範例:

```
if( x === 0){
  throw new Error('x equals zero')
}
```

有些內建的JavaScript方法，在發生錯誤時就會自動"丟出"對應的例外，例如常見的JSON.parse方法，你也可以在自己撰寫的函式裡這樣作，例如上一節中的範例，這也是很常見的一種作法。

window.onerror

瀏覽器中全域的事件處理器(event handler)，它會在錯誤發生時被觸發，這是另一種利用事件作爲錯誤處理的機制。以下有兩種會觸發錯誤事件的情況:

- JavaScript執行階段錯誤(runtime error): 各種錯誤，包含語法錯誤均會觸發ErrorEvent介面，然後呼叫window.onerror方法。
- 當資源無法載入時: 例如<img>或<script>無法正確載入時，會觸發該元素的Event介面，以及呼叫該元素的onerror()方法。

window.onerror是針對JavaScript執行階段錯誤所設計的錯誤事件呼叫的方法，它的語法如下:

```
window.onerror = function(message, filename, lineNo, columnNo, error(
```

註: errorObj參數指的就是Error物件

而針對每個元素所設計的onerror方法，傳入的參數並不同，以此作爲區別。它的語法如下:

```
element.onerror = function(event) { ... }
```

我們的重點會放在`window.onerror`方法上，它算是所有執行階段錯誤的集中處，也是瀏覽器最後可以捕捉到錯誤的方法，通常會用於記錄錯誤之用。但這個方法並不是很完美的方法，它有一些明顯的缺陷。

首先，它在各種瀏覽器品牌與版本上，一直沒有完整的標準，例如以最後傳入的Error物件參數爲例，目前只有在Firefox、Chrome、IE 11上才有支援，在最新的Microsoft Edge或是iOS、Safari瀏覽器上，都沒有這個傳入參數，也可以靠`try/catch`補強的方法來讓所有瀏覽器都可以有這個傳入值。

另外，有一個問題會發生在如果使用的JavaScript程式檔案是來自於CDN時，在Firefox與Chrome瀏覽器會因爲安全機制，導致`window.onerror`方法會完全失效，目前已有解決的方式。

由此看來，這個`window.onerror`方法在基礎標準上並未完善，如果你要使用的話，需要參考一些補強或是相關的文件。

註: 瀏覽器相容的資料來自這篇文章Capture and report JavaScript errors with window.onerror

註: CDN失效的解決方式請參考這篇文章How to catch JavaScript Errors with window.onerror (even on Chrome and Firefox)

結語

錯誤處理基本上只是一個概念與設計哲學，上面所說明的各種語句例如`try/catch`、`throw`都是一些工具，方便讓程式設計師對錯誤處理的程式碼使用而已。

錯誤處理在新式的JavaScript中，可以使用Promise語法，取代原有的`try/catch`語句，詳細的內容請參考Promise的說明。

日期與時間

日期與時間

Date在JavaScript中是一個內建的特殊物件，Date(日期)的資料型態應該是指Datetime(日期時間)而言，所以不只包含所謂的日期 - 年月日而已，它也包含時間中的時、分、秒與微秒。

Date物件複雜的地方不只是它的進位並非十進位而已，年月日的進位還涉及閏年、星期幾的問題，以及全球各種不同語言、全球時區使用的本地化格式的問題等等各種問題。所以Date物件有時是個很難處理的東西，在複雜的應用情況時，以及需要較好的瀏覽器相容性時，我們會用其他的工具函式庫來協助，例如一套很常使用的函式庫 - Moment.js，不過它並不在本章的範圍之中，有需要可以再參考使用。

Date物件中包含了許多對日期時間處理方法，大致上可以分成以下幾類:

- getter: 獲取某種時間格式，例如getFullYear是回傳西元年
- setter: 設定某種時間格式，例如setFullYear設定西元年
- 格式化的getter: 通常是要獲取不同地區的日期(或時間)格式，以及轉換成各種資料格式

註: 微秒(Millisecond, ms)是千分之一秒

Date物件的建構式

Date物件一定只能使用new運算符來建立，這是JavaScript中的硬規則，有一些特殊的內建物件一定要使用建構式進行物件的實體化。

Date物件有四種可在建構式傳入參數的類型，如以下的語法說明，其中只有第三種是可以傳入字串資料類型，傳入參數dateString指的是遵守通用的日期時間字串、國際標準RFC2822或ISO 8601這幾種的字串格式，後面會再說明其中的內容。

```
const date = new Date()
const date = new Date(milliseconds)
const date = new Date(dateString) //只有這種方式是傳入字串類型，其他都不是
const date = new Date(year, month, day, hours, minutes, seconds, mil
```

以下就每種建立Date物件的應用情況分別說明，從範例中來理解它的用法。

取得目前日期時間

new Date()不加任何參數，就會使用瀏覽器環境來取得的目前日期時間作爲傳入參數，說白一點也就是獲得當下的日期時間，輸出時會使用國際日期標準的格式輸出，相等於呼叫toString()方法，注意它仍然是個物件類型，只是無法直接輸出裡面資料結構而已:

```
const datetime = new Date()
console.log(datetime) //Tue Jul 12 2016 11:35:48 GMT+0800 (CST)
console.log(datetime.toString()) //Tue Jul 12 2016 11:35:48 GMT+0800
console.log(datetime.toUTCString()) //Tue, 12 Jul 2016 03:35:48 GMT
console.log(datetime.toTimeString()) //11:35:48 GMT+0800 (CST)
```

另一種要用於計算時間用的格式，則是把Date物件轉成只用微秒數值的格式，它是一個由1

January 1970 00:00:00 UTC開始計算，到目前日期時間的微秒數字值。這個時間類似稱之爲UNIX時間，原本是UNIX系統用來表示時間的方式，不過UNIX時間只計算到秒，而這個數字值是計算到微秒，現在很常用於在程式中計算時間之用。有兩種方式可以獲得這個數字值:

```
//很常用，正號(+)是強制轉換爲數字的語法
const timeInMs1 = +new Date()

//因爲是靜態方法，不常見
const timeInMs2 = Date.now()
```

> 註:　使用Date物件來評估或測量JavaScript應用程式(或語句)的執行時間並不夠可靠與精確，有這個需求的話，你應該使用performance.now或benchmark.js函式庫。

取得日期時間其中某個值

在得到Date物件實體後，可以對Date物件取得包含在其中的某個值，例如年、月、日、星期、時分秒等等。

我們先來看年月日與星期怎麼取得的範例:

```
const dateObject = new Date() //Fri Jul 15 2016 16:23:49 GMT+0800 (CS

const date = dateObject.getDate() //15
const day = dateObject.getDay()  //5
const month = dateObject.getMonth()  //6
const year = dateObject.getFullYear()  //2016
```

Date物件的取值方法，說實在如果直接從名稱上理解會有些不清楚，先看比較沒問題的部份:

- getDate: 取得日期的值，範圍爲1-31
- getFullYear: 取得年(西元年)，爲4位數數字

那麼不清楚或一眼就看得出來的部份是下面兩個:

- getDay: 取得星期幾的值，範例爲0-6
- getgetMonth: 取得月份的值，範例爲0-11

這兩個方法的回傳值，實際上是陣列索引值，也就是說如果你要轉成眞正的月份與星期的值，需要有一個對照的陣列，然後用這個回傳值當作陣列索引值去轉換出來。仔細想想，會這樣設計也是合理的，因爲這兩個值在不同世界地區或語言中，都有不同的代表字詞(例如台灣的"星期"，又可以稱爲"週"或"禮拜"，台語也可簡稱"拜")，所以乾脆保留給設計師自行處理。以下是月份的範例:

```
const monthNamesEn = [
  'January', 'February', 'March', 'April', 'May', 'June',
  'July', 'August', 'September', 'October', 'November', 'December'
]

const monthNamesZh = [
  '一月', '二月', '三月', '四月', '五月', '六月',
  '七月', '八月', '九月', '十月', '十一月', '十二月'
]
```

```
console.log(monthNamesEn[month]) //July
console.log(monthNamesZh[month]) //七月
```

以下則是星期的範例，注意陣列的第一個(索引值爲0)是指"星期日"，有些人眞的不知道"星期日"才是一週的第一天:

```
const dayNamesEn = [ 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thu
const dayNamesZh = [ '星期日', '星期一', '星期二', '星期三', '星期四', '星
```

```
console.log(dayNamesEn[day]) //Friday
console.log(dayNamesZh[day]) //星期五
```

時、分、秒與微秒的取得就很標準一致，以下爲範例，後面註解說明它的回傳數字的範圍:

```
const hour = dateObject.getHours() //0-24
const minute = dateObject.getMinutes() //0-59
const second = dateObject.getSeconds() //0-59
const ms = dateObject.getMilliseconds() //0-999
```

> 註: 仔細比對一下上面範例，取得年月日星期的方法爲單數英文字詞，取得時分秒微秒的是複數英文字詞。

另外，Date物件中還有數個名稱中帶有UTC的取得方法，這些都是在獲取UTC(標準時間)的值。至於取得時區的方法是getTimezoneOffset，它是用分鐘計算的值，而且是以目前程式執行的時區爲基準，所以例如以下的範例會回傳-480，實際上是UTC+8的時區:

```
const timezoneByMins = dateObject.getTimezoneOffset() //-480

//除以60後正負相轉才是時區
const timezone = -(dateObject.getTimezoneOffset()/60)
```

> 註: 時區處理也有可能很複雜，如果有需要可以用專門處理時區的工具函式庫來協助，例如Moment Timezone。

設定日期時間 與 日期時間字串

在Date物件實體化時，給定傳入參數值可以直接設定這個Date物件內的內容，除了不給定內容的建構式會自動抓取瀏覽器目前的日期時間。其它兩種建構式就不多作說明了，本節的重點在於日期時間字串，

JavaScript中的日期格式字串是個頭痛的問題，目前雖然已經是標準的一員。但這標準可不是一開始就有的，而是在ES5標準後才訂定的，所以有些舊版的瀏覽器中並不一定可以這樣用，也有可能在這個瀏覽器可以這樣定義，到別的瀏覽器就不能用了。這裡有一份參考的時間日期字串格式相容表。

通用的格式

建議使用

這並不算什麼標準，只是用這樣定義的話，幾乎在每種瀏覽器品牌或版本都可以相容使用。除非你有非不得已要用其他格式的理由，通常都建議只用這種定義格式就好了，而且最好是使用固定位數的數字，實作上會方便很多:

```
2009/07/12
```

```
2009/7/12
2009/07/12 12:34
2009/07/12 12:34:56
```

把年放在最後面也是可以，不過這方式是英文語言使用者常用的方式，請不要搞混月份與日期的位置，第一個位置是"月份":

```
07/02/2012
7/2/2012
7/2/2012 12:34
```

如果你想要直接在字串中加上微秒的定義，會出現嚴重的不相容性，大概只有Chrome瀏覽器認得。所以解決方式就拆開所有的值，改用使用第四種的`Date`物件建構式分別傳入每個值，或是再額外用設定值的方法`setMilliseconds`直接設定才行。

> 註: 月、日的表示如果要用兩位數字就都用兩位(例如01/01與11/04)，如果要用1到2位的數字就統一這樣用(例如1/1或11/4)。如果你要混用的話，不見得一定可以用，瀏覽器的相容性可能會有問題。

> 註: 在時間上的字串值，時與分至少都要有，而且都用兩位數字，例如01:00或12:09。

ISO 8601格式

> 注意: 舊版瀏覽器不相容

EMCAScript標準中定義的格式，實際上它就是ISO 8601標準，格式如下:

```
YYYY-MM-DDTHH:mm:ss.sssZ
```

YMD代表年月日，Hms代表是時分秒，這兩部份比較沒什麼太大的問題。

那麼`T`代表什麼？`T`只是分隔年月日與時分秒而已。

那麼`Z`代表什麼？`Z`是時區的意思，單純用`Z`代表是UTC標準時間，如果是其他時區的時間可以用`+`或`-`，再加上`HH:mm`作爲時區的表達式，所以像下面的幾個日期時間字串是合於這個規定的:

```
2016-01-01 //年月日可以只有年、年月
2016-07-15T09:00  //時間的部份至少要有時與分
2016-07-15T09:26:23.216Z
2016-07-15T09:20:37.788+08:00
```

RFC 2822格式

另一種格式則是RFC2822格式，它並不是一種專門用於定義日期時間格式的標準，這個標準所定義的標題名稱是"Internet Message Format"(網際網路訊息格式)，主要是用於定義Email(或RSS)的相關格式的標準，而其中有一章是關於日期時間格式的章節。所以，這種日期時間格式，主要的使用群是英文使用者，原因在它使用了英文中的月份與星期縮寫在字串，以下爲格式範例:

```
ddd, DD MMM YYYY HH:mm:ss ZZ
MMM DD, YYYY
DD MMM, YYYY
```

以下是幾個範例:

```
Mon, 15 Aug 2005 15:52:01 +0000
Thu, 18 Feb 2016 15:33:10 +0200
Jan 1, 2015
1 Jan, 2015
```

由於月份與星期都已經字串化(不是數字)，所以你可能會看到有很多種位置不同的寫法，基本上瀏覽器處理這種字串並不會太困難。RFC2822標準並沒有明確指出格式的限制，程式語言都可以很容易實作與解析這種日期字串。例如在MSDN中的說明，把它歸類爲其他格式之中，與其他的日期格式使用相同的規則來作解析處理。

> 注意: 這段的內容只是說明RFC2822與ISO 8601格式定義日期時間的用法，一般情況下你只需要使用第一種的定義方式即可。
>
> 註: 你可能不太明白爲何在設定日期時間時，還要加上星期幾的意義何在。實際上這個格式通常是用來轉換到其他的格式之用。
>
> 注意: JavaScript有一個內建的Date.parse()方法，它是對照使用日期時間字串的Date物件實體化的靜態方法，但它的不相容性相當高，建議不要使用它。

設定方法

設定方法對照取得某個值的方法，除了把get變爲set外，也少了星期的部份。要注意的是，除了上面說的日期時間字串(dateString)，其他的傳入參數一律只能用數字類型。

所有的set方法，都可以使用正負整數或0作爲傳入值，例如以setMonth爲例，雖然它的設定範例是0-11，例超過這個範圍後，會開始進行相加或相減，有可能會影響到年份增加或減少。傳入參數值的規則如下:

- 0: 對setDate方法是回傳上個月的最後一日。setMonth方法則是回傳一月(索引值爲0)。
- 範圍內的整數: 直接設定爲該數字
- 其他正整數: 日期相加
- 其他負整數: 日期相減

先看年月日的部份，這部份除了月的部份也是要用陣列的索引值外，還有一個setDate要特別注意:

```
const dateObject = new Date()

dateObject.setFullYear(2015) //年，4位數字
dateObject.setMonth(0) //月，陣列索引值 0-11
dateObject.setDate(24) //日，正負整數
```

時分秒與微秒的設定方法部份，都有類似setDate的規則，只要超過可設定的範例，就會開始進行相加或相減的情況，以下爲簡單的範例:

```
dateObject.setHours(2)
dateObject.setMinutes(20)
dateObject.setSeconds(60)
dateObject.setMilliseconds(150)
```

最後是setTime方法，它是對照傳入值爲微秒數值的Date物件實體化的建構式，傳入值是上

面所說的由1 January 1970 00:00:00 UTC開始計算，到目前日期時間的微秒數字值。

此外，設定方法中也有多個帶有UTC字串的方法，這些都是專門用於設定UTC時間用的，用法與上述類似，就不再多說。

> 注意: 由於setYear與getYear兩個方法存在了Y2K(千禧蟲危機)的問題而被棄用了，目前已改用 setFullYear與getFullYear。

本地化與格式化

Date物件中已有幾個可以輸出本地化格式字串的方法，主要有三個分別爲toLocaleString、toLocaleDateString、toLocaleTimeString，它們會依照JavaScript程式所執行的環境(瀏覽器)進行本地化，一個簡單的範例如下:

```
const date = new Date()

console.log(date.toLocaleString())      //2016/9/14 下午4:33:41
console.log(date.toLocaleDateString())  //2016/9/14
console.log(date.toLocaleTimeString()) //下午4:33:41
```

格式化的需求通常會用於各種不同的日期或時間的顯示，配合上面所說的獲取日期或時間中某個值的說明，就可以進行各種日期或時間的格式化。Date物件並沒有提供可以進行格式化(format)的標準方法，所以這部份需要程式設計師自己撰寫或使用外部函式庫來輔助，一個簡單的格式化範例如下:

```
const now = new Date()
const months = ['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep
const formattedDate = now.getDate() + '-' + months[now.getMonth()] +

console.log(formattedDate)
```

> 註: toLocaleFormat()方法是一個非標準的方法，大部份的瀏覽器品牌都不能使用。

日期比較

Date物件可以直接比較大小，這種比較是比較日期時間的早晚順序，日期時間愈晚的Date物件會愈大，一個簡單的範例如下:

```
const myDate = new Date()
const today = new Date()

myDate.setFullYear(2015,8,9)

if ( myDate > today ){
  console.log('今天比2015.8.9日早')
} else {
  console.log('今天比2015.8.9日晚')
}
```

Date物件也可以轉爲自微秒數值的格式，它是一個由1 January 1970 00:00:00 UTC開始計算，到目前日期時間的微秒數字值。要取得這個數值是使用getTime方法，一個簡單的範例如下:

```
const myDate = new Date()
```

```
const today = new Date()

myDate.setFullYear(2015,8,9)

console.log(myDate.getTime())
console.log(today.getTime())
```

實例

時鐘

一個簡單的電子時鐘範例如下:

```
function startTime(){
  const today = new Date()
  const hour  = today.getHours()
  const min   = today.getMinutes()
  const sec   = today.getSeconds()

  // 轉成字串，如果低於10，前面加上'0'
  const hourString = ( hour < 10)? ('0'+ hour) : ('' + hour)
  const minString = ( min < 10)? ('0'+ min) : ('' + min)
  const secString = ( sec < 10)? ('0'+ sec) : ('' + sec)

  document.getElementById('content').innerHTML = hourString + ':' + r
}

//重覆每秒執行
setInterval(startTime ,1000)
```

特性

原型基礎物件導向

原型基礎物件導向

> If you don’t understand prototypes, you don’t understand JavaScript.
>
> 如果你沒搞懂原型，你不算眞的懂JavaScript

JavaScript本身就是原型爲基礎的物件導向設計，至ES6標準制定後仍沒變動過。在物件的章節中所介紹的類別定義方式，只是原型物件導向語法的語法糖，骨子裡還是原型，並不是眞正的以類型爲基礎的物件導向設計。理解JavaScript的原型是很重要的，只是混亂得讓初學者難以理解。

> 註: 語法糖(Syntactic sugar)指的是在程式語言中添加的某些語法，這些語法對語言本身的功能並沒有影響，但是能更方便使用，可以讓程式碼更加簡潔，有更高可讀性。另外類似的術語還有"語法糖精"與"語法鹽"。

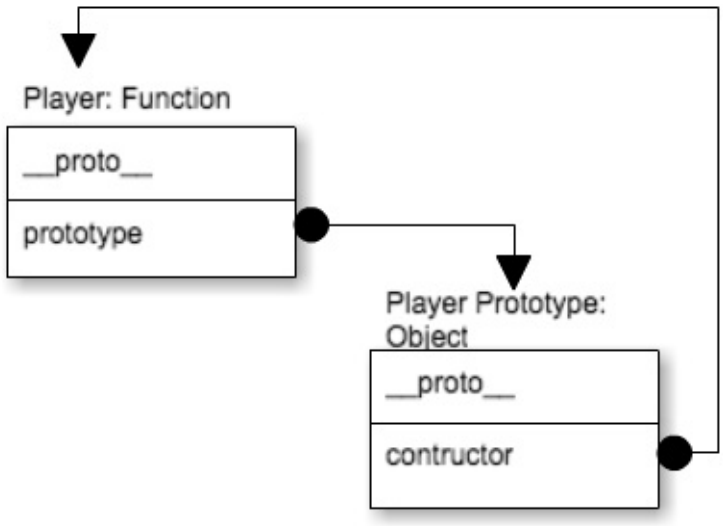
函式 - 原型的起手式

所有JavaScript中的函式都有一個內建的`prototype`屬性，指向一個特殊的prototype物件，prototype物件中也有一個`contructor`屬性，指向原來的函式，互相指來指去會讓你覺得有點怪異，但設計就是如此。

以下的程式碼可以看出這個關係:

```
function Player() { }

console.log(Player)
console.log(Player.prototype)
console.log(Player.prototype.constructor)
console.log(Player.prototype.constructor === Player) //true
```

爲了更容易理解，以下是一個簡單的關係圖:



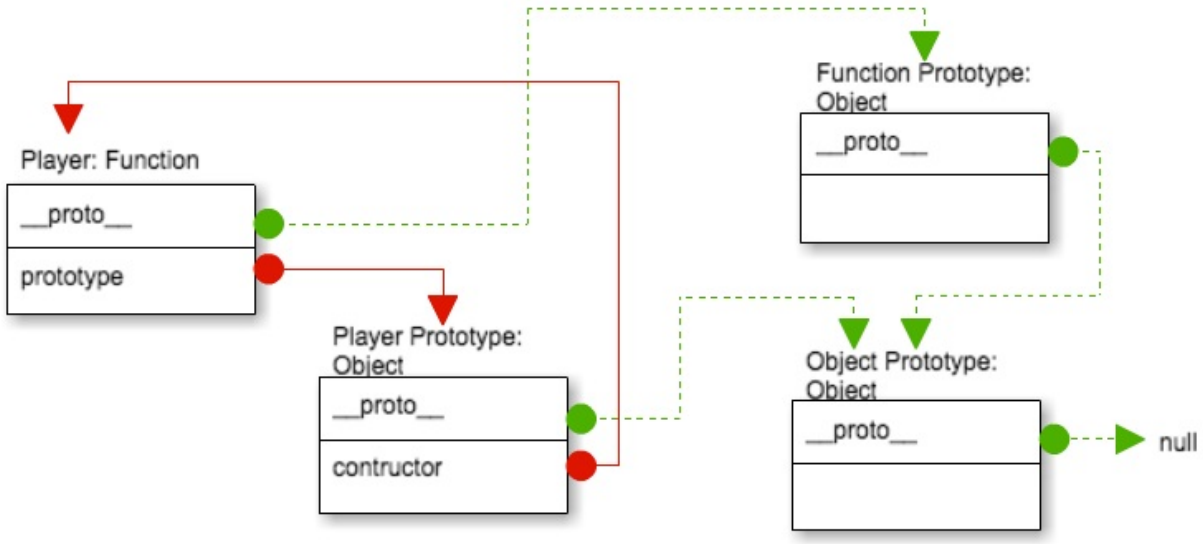
再來是__proto__這個內部屬性，它是一個存取器(accessor)屬性，意思是用getter和setter函式合成出來的屬性，我們可以用它來更加深入理解整個原型的樣貌。__proto__是每一個JavaScript中物件都有的內部屬性，代表該物件繼承而來的源頭，也就是指向該物件的原型(prototype)，它會用來連接出原型鏈，或可以理解爲原型的繼承結構。

對於一個函式而言，它本身也是一個物件，它的原型就是最上層的Function.prototype，你可以說這是所有函式的發源地。所以Player函式本身的__proto__指向Function Prototype，這應該可以很容易理解。

那麼，Player.prototype的__proto__指向哪裡？Player.prototype本身也是個物件，它指向的就是所有JavaScript中最上層的物件起源，也就是Object.prototype。由此也可推知，Function.prototype也同樣指向Object.prototype。以下面的程式就可以看到這個結果:

```
function Player() { }

console.log(Player.__proto__)
console.log(Player.prototype.__proto__)

//最上層的Object.prototype的__proto__是null值，它是一個特例
console.log(Object.prototype.__proto__)  //null

console.log(Player.__proto__ === Function.prototype)  //true
console.log(Player.prototype.__proto__ === Object.prototype) //true
console.log(Function.prototype.__proto__ === Object.prototype) //tru
```

爲了更容易理解，以下是一個簡單的關係圖，在圖片中綠色的虛線即是__proto__的指向，原本的prototype爲紅色的實線:



註: __proto__注意是前後各有兩條下底線(_)，不是只有一條而已。

註: __proto__在一些舊的瀏覽器品牌(例如IE)中不能使用。雖然在ES6中已經正式納入標準之中，它是個危險的內部屬性，也不要用在眞正的應用程式上。

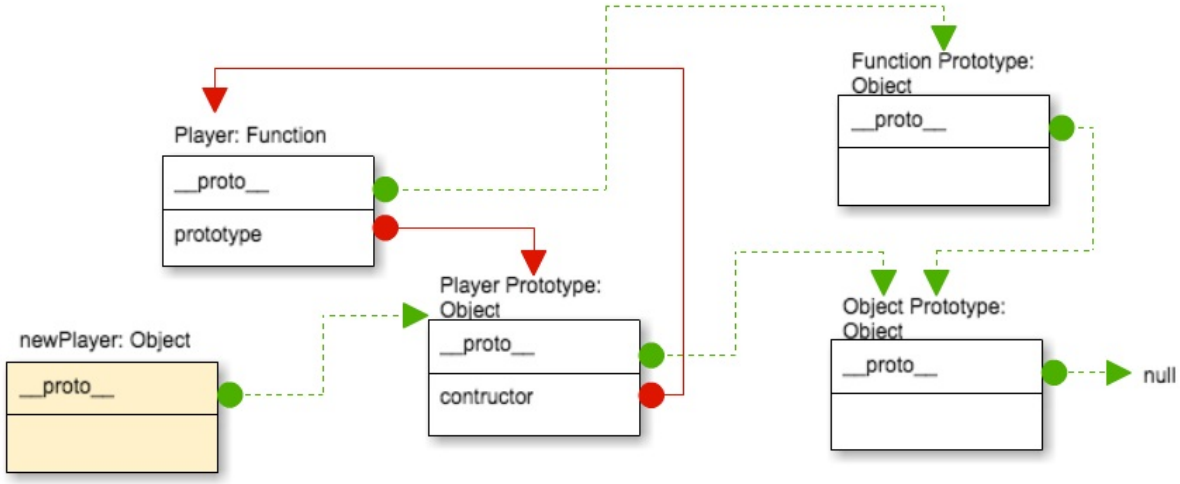
當進一步使用Player函式作爲建構函式，產生物件實體時，也就是使用new運算符的語句。像下面這樣簡單的例子，在建構函式中會用this.name的方式來指定傳入參數，this按照之前在物件篇的內容所說明，指向的是new運算符中指定的物件實體。

```
function Player(name) {
  this.name = name
}

const newPlayer = new Player('Inori')
```

此時在newPlayer物件中的prototype與__proto__又是如何？由於newPlayer是一個物件，並不是函式，它不會有prototype這個屬性。newPlayer的__proto__則是指向Player.prototype，把物件的原型鏈整個串接起來。

```
//不是函式，不會有prototype
console.log(newPlayer.prototype)  // undefined

console.log(newPlayer.__proto__)
console.log(newPlayer.__proto__ === Player.prototype) //true
```

以下是一個簡單的關係圖，黃色的代表剛剛實體化的newPlayer物件，在圖片中綠色的虛線即是__proto__的指向:



最後總結以下的幾個摘要，讓這章節的內容更加清楚:

- 每個函式中都會有`prototype`屬性，指向一個`prototype`物件。例如MyFunc函式的`prototype`屬性，會指向對應的MyFunc.prototype物件。
- 每個函式的`prototype`物件，會有一個`constructor`屬性，指回到這個函式。例如MyFunc.prototype物件的`constructor`屬性，會指向MyFunc函式。
- 每個物件都有一個`__proto__`內部屬性，指向它的繼承而來的原型`prototype`物件
- 由`__proto__`指向連接起來的結構，稱之爲原型鏈(prototype chain)，也就是原型繼承的整個連接結構

原型眞相 - 由問答理解

原型到底是什麼，從上面看到原型鏈、new運算符、建構式等等的概念，你可能會有一些疑惑或誤解。以下是幾個重點，我們用問答的方式來說明。

原型是什麼？

一種讓別的物件繼承其中的屬性的物件。

任何物件都有其原型？

是的。唯一的例外是Object.prototype(Object的原型)沒有原型，它是所有物件的最上層的源頭。

任何物件可以拿來作爲原型？

是的。

那爲什麼上面的例子中，`newPlayer`物件沒有`prototype`屬性

用`Object.getPrototypeOf`方法或`__proto__`屬性，可以找到這個物件眞正的原型是什麼，而這個prototype屬性是給建構函式用的。事實上，每個物件都一定有constructor(建構式)屬性，constructor(建構式)屬性會指向建立這個物件的函式，建構函式的屬性其中就有prototype屬性。

```
const newPlayer = new Player('Inori')
```

```
console.log(Object.getPrototypeOf(newPlayer))
console.log(newPlayer.__proto__)

//只能用於不是使用Object.create建立的物件
console.log(newPlayer.constructor.prototype)
```

爲什麼函式中一定會有**prototype**屬性，那建構函式與函式的區分又是什麼？

設計上的原則而已，JavaScript並沒有對你所建立的函式，區分建構函式與一般的函式，所以只要是函式，就一定會有prototype屬性(除了語言內建的函式不會有這個屬性)。而且，函式以外的類型，不會有這個屬性。

擴充

原型可以很容易的擴充屬性與方法，而且是動態的，可以在物件實體後繼續擴充其中的成員，這也是JavaScript中常用來擴充內建物件的方式。當使用`Player.prototype`來進行擴充時，這些擴充出來的屬性與方法，是所有物件共享的，見以下的範例。

```
function Player(name) {
  this.name = name
}

const newPlayer = new Player('Inori')
console.log(newPlayer.age)

Player.prototype.age = 11
console.log(newPlayer.age)

Player.prototype.toString = function() {
    return 'Name: ' + this.name
}

console.log(newPlayer.toString())
console.log(newPlayer) //觀察物件的內容
```

> 註: 如果以類別爲基礎的物件導向，在物件實體化後，物件或類別都是無法擴充其中的成員的。這一點是原型物件導向的彈性之處。

繼承

繼承是什麼？我們回歸本質上來思考"繼承的目的是什麼"，在程式開發時的目的通常是爲了擴充原有的物件定義不足之處。繼承基本上可以依照不同程式語言的物件導向特性區分爲:

- 類別的繼承(Classical inheritance)
- 原型爲基礎的繼承(Prototype-based inheritance)

原型繼承是什麼，其實就是原型的擴充語法，上一節有說明過了。至於類別的繼承方式，在JavaScript可以用`Object.create`方法模擬出來，不過複雜多了。

```
//superclass
function Player(name) {
    this.name = name
```

```
}

// 1. 呼叫上層的建構式
function VipPlayer(name, level) {
    Player.call(this, name)
    this.level = level
}

// 2. 使用Object.create建立prototype物件。
//    VipPlayer建構式的prototype.constructor爲自己。
VipPlayer.prototype = Object.create(Player.prototype)
VipPlayer.prototype.constructor = VipPlayer


var inori = new VipPlayer('inori', 5)

console.log(inori instanceof Player) //true
console.log(inori instanceof VipPlayer) //true
```

相當於ES6中的類別定義方法，使用extends關鍵字來作類別的繼承:

```
class Player{
  constructor (name){
    this.name = name
  }
}

class VipPlayer extends Player {
  constructor (name, level){
    super(name)
    this.level = level
  }
}
```

總而言之，在很多眞實的應用情況下，"以合成(或擴充)代替繼承(composition over inheritance)"才是正解，在JavaScript中合成比繼承容易得多了，彈性高應用也很廣，也比較符合語言本身的特性。思考的重點不同，才能撰寫出符合應用情況的程式碼。

私有/公開成員

在十多年前Douglas Crockford的這篇Private Members in JavaScript就有提出關於私有成員的樣式，使用的是建構式樣式來模擬私有、公有與特權方法。網路上大部份的文章都是使用這個樣式來說明，或是進一步改良。

不過，私有或公開成員完全不是JavaScript語言中原本就有的設計，用這種方法會限制住只能使用"建構式樣式"的語法來作物件實體化，而且也與原型物件導向概念相去甚遠。原型物件導向對於封裝的概念，基本上是根本沒有。

目前比較簡單常見的區分方式，就是在私有成員(或方法)的名稱前面，加上下底線符號(_)前綴字，用於區分這是私有的(private)成員，這只是由程式開發者撰寫上的區分差別，與語言本身特性無關，對JavaScript語言來說，成員名稱前有沒有有下底線符號(_)的，都是視爲一樣的變數，至於要如何保護這些私有成員，就靠程式設計者自己了。

靜態屬性/靜態方法

JavaScript語言中也沒這概念。不過，原型物件另外定義的屬性與方法，是所有以此原型物件實體化的物件共享的就是了:

```
function Employee() {
}
Employee.prototype.count = 0;

var e = new Employee();
console.log(e.hasOwnProperty('count')); // logs "false"
e.count += 1;
console.log(e.hasOwnProperty('count')); // logs "true"

console.log(e.count);                   // logs "1"
console.log(Employee.prototype.count);  // logs "0"
var e = new Employee();

console.log(e.count);                   // logs "0"
++Employee.prototype.count;
console.log(e.count);                   // logs "1"

var e = new Employee();
console.log(e.count);    // Logs "0", because it's using `Employee.pr
++e.count;               // Now `e` has its own `count` property
console.log(e.count);    // Logs "1", `e`'s own `count`
delete e.count;          // Now `e` doesn't have a `count` property a
console.log(e.count);    // Logs "0", we're back to using `Employee.p
```

另一種作法是用IIFE模擬出靜態變數的結構:

```
var Employee = (function() {
    var sharedVariable = 0;

    function Employee() {
        ++sharedVariable;
        console.log("sharedVariable = " + sharedVariable);
    }

    return Employee;
})();

new Employee(); //1
new Employee(); //2
new Employee();  //3
new Employee();  //4
```

原型鏈(prototype chain)

原型鏈是JavaScript中以原型爲基礎的物件導向特性，指的是使用原型來作物件實體化，會產生"原型鏈"的結構。原型鏈的觀念如此重要，在於有很多物件的行爲都是與它相關，在物件篇已經有介紹過物件中的一些方法，都是會遍歷整個物件的原型鏈，而不僅是物件本身而已。這與原型的物件實體化設計有關，因爲物件的實體化過程，就是原型的繼承過程，也就是物

件的實體化是由繼承其他物件而來的。

先看一下物件屬性的存取這件事，下面的例子中，我們並沒有在player物件中定義toString方法，但它的確是存在的，這個toString方法是來自原型鏈上層的物件中，也就是繼承得來的。

```
const player = {}
console.log(player.toString())
```

instanceof

instanceof是一個運算符，主要是用來判斷一個物件在原型鏈中是否有存在某個建構式，如果存在回傳true，要不然就是false值。它的前面的運算子是物件，後面則是建構式，語法是像下面這樣:

```
object instanceof constructor
```

instanceof常會拿來和存取物件實體的constructor屬性的方式作比較，由於constructor只會指向最接近的建構式，而instanceof會找遍整個原型鏈，當然結果會有所有不同，以下爲範例:

```
function Animal(){}

function Cat(){}
Cat.prototype = new Animal()
Cat.prototype.constructor = Cat

const kitty = new Cat()


console.log(kitty instanceof Cat) // true
console.log(kitty instanceof Animal) // true
console.log(kitty.constructor === Cat) // true
console.log(kitty.constructor === Animal) // false
```

instanceof有一些例外情況，它對於原始資料類型(數字、字串、布林、null、undefined)是無法判斷的，必定是回傳false，這和用constructor屬性判斷是不同的結果，例如以下的範例:

```
console.log(3 instanceof Number) // false
console.log(true instanceof Boolean) // false
console.log('abc' instanceof Boolean) // false
console.log((3).constructor === Number) // true
console.log(true.constructor === Boolean) // true
console.log('abc'.constructor === String) // true
```

另外對在iframe、frame或著另開視窗中所產生的物件進行判斷時，它也會失效。

在不使用建構式的物件建立的情況，它也無法判斷，而且會產生錯誤，所以它並不能使用於像Object.create方法，或直接回傳物件的工廠樣式，只能用於建構式樣式。

in與for...in

in運算符是用來判斷某個屬性是否存在於某個物件中，存在的話會回傳true，否則會回傳

false。in一樣會尋遍整個原型鏈，對比hasOwnProperty則是只會在這物件當中，並不會往原型鏈尋找，in通常會搭配for使用for...in語句。以下這個語法是最常見到的:

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

鴨子類型(Duck typing)

那麼我們要如何正確的判斷一個物件，就是我們需要的物件？首先，typeof運算符所能判斷的情況過少，只能用於判斷資料類型，在資料類型那個章節就有提及它的內容。對於物件、陣列、null來說，它都會回傳'object'。

instaceof只能用於以new實體化的物件，也就是有建構式的情況，而且它有一些失效的情況，instaceof有時會直接產生錯誤中斷執行，而不是回傳false。instaceof並不是不能使用，常見的使用情況是用於判斷語言內建的幾個物件，例如以下幾個:

```
[1, 2, 3] instanceof Array // true
/abc/ instanceof RegExp // true
({}) instanceof Object // true
(function(){}) instanceof Function // true
```

那如果是在一個函式的傳入參數，要如何判斷這個傳入物件實體是我們要的？

答案就是使用"鴨子類型(Duck typing)"，也就是使用該物件的屬性或行爲複合地來判斷它，類似像下面的說明:

> 當看到一隻鳥走起來像鴨子、游泳起來像鴨子、叫起來也像鴨子，那麼這隻鳥就可以被稱爲鴨子。

所以當我們要判斷一個自訂的物件，可以用其中應該包含的屬性與方法來判斷，例如以下的判斷函式:

```
function isPlayer(object) {
  return object != null && typeof object === 'object' &&
    'name' in object && 'age' in object && 'toString' in object
}
```

new有害說與物件實體化

JavaScript長期以來就有反對使用new運算符用於實體化物件的言論，建議大家不要使用它來作物件的實體化，主要的理由是語言本身設計上的缺陷:

> 忘了在物件實體化時加上new運算符，函式並無明確區分建立物件用的建構式與一般的函式，如果你是要用來實體化物件，而忘了加上new關鍵字，雖然不會產生任何錯誤，但事情會很大條。

以下用幾個方向來討論如何正確的其他作法，以及如何避免其中可能的問題。

防止錯誤的語法樣式

這個樣式可以防止程式設計師少加了new運算子。一般的JavaScript函式庫並不鼓勵使用它的

程式設計師使用new，反而會希望用它的程式設計師都使用函式的方式來建立物件實體，原因除了防止漏寫的錯誤外，在函式也有可能會隱藏對於物件實體的複雜的生成過程。以下爲範例:

```
function Player(name, age)
{
   if (this instanceof Player){
      this.name = name
      this.age = age
   }else{
      return new Player(name, age)
   }
}

const aPlayer = Player('Inori', 16)
const bPlayer = new Player('Gi', 16)
```

Object.create

Object.create方法可以使用的是物件的原型來建立物件。這是一個ES5後加入的新方法，最早在10年前在這篇文章Prototypal Inheritance in JavaScript提出的想法與實作。文章中提及new本身就是一個爲了要讓JavaScript中的物件實體化，用起來像是類別爲基礎的程式語言，才會設計的一個語法，但因此模糊了原型繼承的眞正作法。

Object.create並不只是new運算符的取代方法這麼簡單，它提供了更多的彈性，把物件導向的語法結構變成原本的原型導向，它的基本語法如下:

Object.create(proto[, propertiesObject])

必要的傳入參數是物件的原型，在下面的範例中可以看到。不過要先說明的是爲何它把整個語法結構都轉變了。按照new運算符來作實體物件的工作，原本的步驟是像下面這樣的:

1. 先撰寫建構函式，定義好裡面的屬性與方法
2. 然後用new來建立物件實體

new會作的工作已經說明過很多次了，指向this到新建立的物件實體、執行建構函式，最後回傳物件實體。如果你比較一下ES6中的類別語法，這個流程與以類別爲基礎的物件導向幾乎無異，差異只是在撰寫類別定義與撰寫建構函式定義上，類別定義現在只是個語法糖，還記得嗎？所以當然是一樣的。

那麼原型物件導向原來的流程應該是如何的？原型物件導向有幾個核心的概念:

- 物件繼承自其他物件: 並不是先有物件的藍圖(類別)，然後實體化它。而是先有要作爲原型的物件，然後用由這個物件產生新的物件。
- 以合成(或擴充)代替繼承: 合成(或擴充)的方式才是重點，由原型物件開始實體一個新物件，可以很容易的合成與擴充。

Object.create方法相較於new運算符，在物件實體化過程中有一個非常明顯的差異:

Object.create不會執行建構函式

Object.create的使用流程會是這樣的，這是簡化過的版本，實際上可能不只這樣:

1. 定義一個物件當作原型物件

2. 從這個原型物件建立另一個新的物件(過程可以加入其他的屬性)

以一個最簡單的範例來說，你可以看到Object.create完全不使用建構函式來設定新物件實體的屬性值，只是從原型物件產生一個新物件而已。

```
const PlayerPrototype = {
  name: '',
  toString() {
    return 'Name: '+ this.name
  }
}

const inori = Object.create(PlayerPrototype)
inori.name = 'iNori'
console.log(inori.toString())
```

那麼我們如果要對物件實體進行初始化要怎麼作，其實就是呼叫原型物件裡一個自訂的初始化用的方法就行了，通常會用init來當這個方法的名稱:

```
const PlayerPrototype = {
    init(name, age) {
        this.name   = name
        this.age  = age
    },
    toString() {
        return 'Name: '+ this.name +' Age:'+ this.age
    }
}

const inori = Object.create(PlayerPrototype)
inori.init('iNori', 16)
console.log(inori.toString())
```

註: 這個語法樣式，有個專有名稱叫作OLOO(objects linked to other objects)樣式

註: 因爲沒有了建構式，所以有一些屬性(例如constructor)與方法(例如instanceof)不能使用，會有錯誤的情況。可以用isPrototypeOf方法來判斷原型的關係。

Object.create方法的第二個傳入參數，可以使用一種特殊的屬性物件(properties Object)，提供更多在物件建立時的彈性運用，例如以下的範例:

```
const inori = Object.create(PlayerPrototype, {
                hairColor: {
                    value: 'pink',
                    writable: true
                }
            })
inori.init('iNori', 16)

console.log(inori)
```

註: 屬性物件是一種用來定義屬性的特殊物件，請參考Object.defineProperties()

建立物件的語法比較

我們目前爲止已經看到物件的建立有以下這三種方式:

```
//物件字面定義，相等於new Object()
const newObject = {}

//使用Object.create方法
const newObject = Object.create( proto )

//ES6類別定義，或是建構函式。通常稱爲建構式樣式。
const newObject = new ConstructorFunc()
const newObject = new ClassName()
```

在進入建立物件的主題前，我想先說明一下，在JavaScript中關於建立物件這件事，是有需要那麼常用到的嗎？或是眞的會在單一個應用程式中，建立大量的物件的情況？

你應該了解，JavaScript的應用程式都是執行在瀏覽器的環境中，這是一個有受到限制的執行環境。而且是當一個使用者連到網頁時，JavaScript的應用程式才會透過網站傳遞到使用者電腦中的瀏覽器，然後才執行，這與一般的桌面或手機上的應用程式，先安裝後才執行完全不同。

一般常見程式設計師會在JavaScript應用程式裡，建立不重覆的自訂物件資料的應用情況有可能是以下幾種:

- 網頁上的UI小元件、行事曆、對話盒等等: 一個網頁上頂多是10-20個物件。
- 用來描述資料模型的物件:　用來作爲最終的資料交換使用，描述資料的物件，頂多5~10個物件。
- 用於應用程式的物件:　例如一個遊戲中，對於怪物、玩家角色、NPC的這些物件，大概就是20-50個。

以效能來說，物件的建立這件事，不太可能像在網路上測試報告的情況，一次建立幾十萬個或百萬個物件。物件的建立與各種運算，本身就是高消費的，所以在物件的建立，反而效率並不是太重要的課題，而是它在撰寫時的高閱讀性、易於維護與擴充、使用的彈性等等。

以上面的三種物件建立的方式來說，效率最佳的是物件字面定義(花括號({})定義物件)，其次爲建構函式加上new運算符這種，通常稱之爲"建構式樣式(Constructor　　Pattern)"，最差的則是Object.create。

但物件字面定義語法有一些問題，它只會有一個物件實體。它沒辦法直接複製出其他的物件實體，所以如果是要指"可建立多個物件"的語法，這個並不是可以這樣使用的，它需要寫成一個像下面這樣的函式，才能達到需求，這稱之爲工廠樣式(Factory Pattern)的語法:

```
function PlayerFactory(name, age) {

  return {
    name : name,
    age: age,
    toString : function() {
        return 'Name: '+ this.name +' Age:'+ this.age
    }
  }
}

const inori = PlayerFactory('Inori', 16)
```

```
console.log(inori.toString())

const ayase = PlayerFactory('ayase', 17)
console.log(ayase.toString())
```

單純使用物件字面定義的工廠樣式在效率上是吊車尾的，而且由於所有的物件實體都類似於物件字面所定義出來的，它們的原型都是Object.prototype。工廠模式也可以用的Object.create方法來建立物件實體，搭配物件字面定義出來的物件，建立新的物件實體，上面已經有範例，要用哪一種，都是要視應用的情況而定的。

必要的情況(內建物件)

有些JavaScript語言中的內建物件，在使用時一定要用new運算符進行實體化，例如以下幾個:

```
new Date()
new XMLHttpRequest()
new Error
```

除了這些之外，JavaScript語言中內建的包裝物件幾乎都不使用new作物件實體化，也不建議使用。

建構式樣式 vs 工廠樣式

new在眞實的應用情況也很少會用到，不過我認爲主因應該是語法樣式，而非單純只有new本身的問題，重點應該放在，對於"建構式樣式"與"工廠樣式"的比較。至少到目前爲止的所看到的，"建構式樣式"教得人多，但用得人很少，"工廠樣式"的使用頻率是遠遠勝過"建構式樣式"。

"建構式樣式"是以建構函式式爲主的語法樣式，使用new作爲物件實體化的唯一方式，最後回傳物件實體。而只能把物件的定義內容寫在建構式之中，只會回傳物件實體，限制住很多能使用的情況。建構函式原本就是一個JavaScript中十分怪異的設計，除了初學者一定很容易和一般的函式搞混，它有一些隱含的機制也很奇特。

"工廠樣式"的語法提供了更多的彈性，相較於"建構式樣式"只能回傳物件，"工廠樣式"最後直接回傳物件實體，但"工廠樣式"也可以多了很多彈性，可以視情況提供各種物件實體的應對程式碼，最後可以回傳以物件字面定義的物件、使用new或Object.create方法。此外，工廠樣式可以對物件資料進行更好的封裝(encapsulation)與資料隱藏(data hiding)，這一點在建構式樣式中完全是個無法比得上的。

更多的樣式

原型鏈共享的工廠樣式

工廠樣式提供了更多的彈性，因爲Object.create直接由一個單純的物件來建立物件，失去了原型鏈的擴充彈性，你可以用下面的樣式來調整:

```
function Player(){}

Player.prototype.toString = function(){
  return this.name
}

function PlayerFactory(name){
```

```
  const obj = Object.create(Player.prototype)
  obj.name = name

  return obj
}
```

實際上Player函式與PlayerFactory函式兩者可以合併，像下面這樣:

```
function PlayerFactory(name){
  const obj = Object.create(PlayerFactory.prototype)
  obj.name = name

  return obj
}

PlayerFactory.prototype.toString = function(){
  return this.name
}

const inori = PlayerFactory('Inori')
console.log(inori.toString())
console.log(inori)
```

多重繼承(複合)樣式

這個樣式可以建立繼承自多個物件的新物件，用的是Object.assign加上Object.create方法的語法，這方式可以一次增加多個新物件的屬性與方法，此外Object.assign並沒有限定第二個參數之後只能加一個物件進來合併，所以可以加很多物件來合併成爲一個新的物件，類似多重繼承的結果。以下爲範例:

```
let player = {
  name: 'player',
  toString() {
   return this.name
  }
}

function PlayerFactory(name, age) {
  return Object.assign(Object.create(player), {
    name: name,
    age: age,
    toString(){
      return 'Name: '+ this.name + ' Age:'+ this.age
    }
  })
}

const inori = PlayerFactory('inori', 16)
console.log(inori.toString())
console.log(inori)
```

extend(擴充)樣式

某些時候對於簡單的物件，像是設定值之類的物件，如果都要用到物件實體化或各種語法樣式，實在太過沉重。這個時候用extend(擴充)的方式是快速簡便的，並不一定要額外進行物件實體化，extend(擴充)樣式是一種Mixins(混合)樣式，它並不是繼承或物件實體化的樣式。簡單的extend函式就只是個迴圈語句而已，範例如下(出自SweetAlert):

```
var extend = function extend(a, b) {
  for (var key in b) {
    if (b.hasOwnProperty(key)) {
      a[key] = b[key];
    }
  }
  return a;
};
```

> 註: 許多JavaScript函式庫例如jQuery、underscore、lodash都有提供extend(擴充)的API，其他也有像clone(複製)、merge(合併)、assign(指定)之類的用於物件的API

結語

JavaScript中原型物件導向設計其實並不難理解，難的是它裡面混雜的太多奇奇怪怪的設計，而且又常要與類別爲基礎的物件導向設計相比較。本章除了提供原型鏈的基礎知識說明外，也加入了很多你可能會在實際使用時遇到的樣式，工廠樣式與建構式樣式，這兩個基本的樣式你應該要先熟悉。

this

this

在其他以類別爲基礎的程式語言中，this指的是目前使用類別進行實體化的物件。而JavaScript語言中因爲在設計上並不是以類別爲基礎的物件導向，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，因爲不同的物件呼叫，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，也有看到用函式作爲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，這是因爲babel預設會開啓strict mode(嚴格模式)，爲了安全性的理由，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爲什麼可以這樣作。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。以下爲一個範例:

```
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。爲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)
```

不過因爲用了物件，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，產生一個稱爲部份套用(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，直接設定爲call方法裡面的thisArg(this參數值)。
2. 當thisArg(this參數值)是null或undefined時，會綁定爲全域(global)物件。
3. 當thisArg(this參數值)的類型不是物件類型時，會綁定爲轉爲物件類型的值。
4. 都不是以上的情況時，綁定爲thisArg(this參數值)。

第1點就明確的說明了，爲什麼使用strict mode(嚴格模式)後，在全域的函式呼叫執行，this值一定都是`undefined`，因爲在call中根本沒傳入thisArg值。除非關閉strict mode(嚴格模式)才會變爲第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，爲什麼要這樣作？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通常被稱爲Variable Scope(變數作用域)，意思是"作用域代表變數存取的範圍"。

Context通常被稱爲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爲主的開發者常常會搞混，其實像這麼像的名詞，以英文爲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，所以我把它稱之爲一種結構。

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)來區分爲三個類型，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，因爲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，因爲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"僞"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，如果你有認眞看過"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。這也是爲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，爲什麼？因爲執行上下文是以函式呼叫作爲區分，所以this值在不同的函式呼叫時，預設上就會不同。這稱之爲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。它只是爲了暫時保存在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，不過因爲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(因爲這裡不適合使用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)除了作爲以函式表達式(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，以下爲改寫過的範例，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，因爲這是使用了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，因爲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。這是因爲陣列的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爲基礎，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，就是爲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`的概念因爲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)

Callback 回調

Callback(回調)

Callback(回調)是什麼？

中文翻譯字詞會用"回呼""、"回叫"、"回調"，都是聽起來很怪異的講法，Callback在英文是"call back"兩個單字的合體，你應該有聽過"Call me back"的英文，實際情況大概是有客户打來電話給你，可是你正在電話中，客户會留話說請你等會有空時再"回電"給它，這裡的說法是指在電信公司裡的callback的涵意。而在程式開發上上，callback它的使用情境其實也是有類似的地方。

CPS風格與直接風格

延續傳遞風格(Continuation-passing style, CPS)，它的對比是"直接風格(Direct style)"，這兩種是程式開發時所使用的風格，CPS早在1970年代就已經被提出來了。CPS用的是明確地移轉控制權到下一個函式中，也就是使用"延續函式"的方式，一般稱它爲"回調函式"或"回調(Callback)"。回調是一個可以作爲傳入參數的函式，用於在目前的函式呼叫執行最後移交控制權，而不使用函式回傳值的方式。直接風格的控制權移交是不明確的，它是用回傳值的方式，然後進行到下一行程式碼或呼叫接下來其他函式。下面以範例來說明會比較容易。

直接風格的範例如下，其實就是一般函式呼叫的方式，或是用回傳的方式:

```
//直接風格
function func(x) {
  return x
}
```

CPS風格就不是這樣，它會用另一個函式作爲函式中的傳入參數的樣式來撰寫程式，然後將本來應該要回傳的值(不限定只有一個)，傳給下一個延續函式，繼續下個函式的執行:

```
//CPS風格
function func(x, cb) {
  cb(x)
}
```

以明確的程式流程的例子來說，假設現在要從資料庫獲取某個會員的資料，然後把裡面的大頭照片輸出。

用直接風格的寫法:

```
function getAvatar(user){
   //...一些程式碼
   return avatar
}

function display(avatar){
  console.log(avatar)
}
```

```
const avatar = getAvatar('eddy')
display(avatar)
```

用CPS風格的寫法，像下面這樣:

```
function getAvatar(user, cb){
  //...一些程式碼
   cb(avatar)
}

function display(avatar){
  console.log(avatar)
}

getAvatar('eddy', display)
```

長久以來在程式語言開發界，直接風格的程式碼是最常被使用的，因爲它容易被學習與理解，一個步驟接著一個步驟，學校的程式語言課程大部份也是用這種風格來教學，不論在個人電腦上的、在伺服器端的程式語言設計通常也是這樣。主要是因爲個人電腦端或是伺服器端，通常使用多執行緒或改進底層運作的執行方式，來解決多工或並行的問題。CPS風格反而很少被使用，並沒有明顯的誘因讓程式設計師一定要用CPS風格，而且也不是所有的程式語言都能使用CPS風格，在過去CPS風格並不算是主流的程式開發寫作風格。

CPS風格相較於直接風格還有一些明顯的缺點:

- 在愈複雜的應用情況時，程式碼愈不易撰寫與組織，維護性與閱讀性也很低
- 在錯誤處理上較爲困難

JavaScript中會大量使用CPS風格，除了它本身可以使用這種風格外，其實是有原因:

- 只有單執行緒，在瀏覽器端只有一個使用者，但事件或網路要求(AJAX)要求不能阻塞其他程式的進行，但這也僅限在這些特殊的情況。不過在伺服器端的執行情況都很嚴峻，要能同時讓多人連線使用，必需要達到不能阻塞I/O，才能與以多執行緒執行的伺服器一樣的執行效益。
- 一開始就是以CPS風格來設計事件異步處理的模型，用於配合異步回調函式的執行使用。

基本上一個程式語言要具有高階函式(High Order Function)的特性才能使用CPS風格，也就是可以把某個函式當作另一函式的傳入參數，也可以回傳函式。除了JavaScript語言外，具有高階函式特性的程式語言常見的有Python、Java、Ruby、Swift等等。

異步回調函式

並非所有的使用callbacks(回調)函式的API都是異步執行的，但CPS的確是一種可以確保異步回調執行流程的風格。在JavaScript中，除了DOM事件處理中的回調函式9成9都是異步執行的，語言內建API中使用的回調函式不一定是異步執行的，也有同步執行的例如`Array.forEach`，要讓開發者自訂的callbacks(回調)的執行轉變爲異步，有以下幾種方式:

- 使用計時器(timer)函式: `setTimeout`, `setInterval`
- 特殊的函式: `nextTick`, `setImmediate`
- 執行I/O: 監聽網路、資料庫查詢或讀寫外部資源
- 訂閱事件

針對callbacks(回調)函式來說，異步與同步的執行到底是差在那裡？你可能會產生疑惑。下面

用個簡單的例子來說明。

```
function aFunc(value, callback){
  callback(value)
}

function bFunc(value, callback){
  setTimeout(callback, 0, value)
}

function cb1(value){ console.log(value) }
function cb2(value){ console.log(value) }
function cb3(value){ console.log(value) }
function cb4(value){ console.log(value) }

aFunc(1, cb1)
bFunc(2, cb2)
aFunc(3, cb3)
bFunc(4, cb4)
```

aFunc是一個簡單的回調結構，callback回調函式被傳入後最後以value作爲傳入參數執行。

bFunc函式則是包裹了一個setTimeout內建方法，它可以在一定時間內(第二個參數)執行第一個參數，也就是setTimeout會執行的回調函式，第三個參數是要加入到回調函式的傳入參數值。

aFunc中使用了一般的回調函式，只是傳入到函式中當作參數，然後最後執行而已，這種是同步執行的回調函式，只是用了CPS風格的寫法。

bFunc中使用了計時器APIsetTimeout會把傳入的回調函式進行異步執行，也就是先移到工作佇列中，等執行主執行緒的呼叫堆疊空了，在某個時間回到主執行緒再執行。所以即使它的時間設定爲0秒，裡面的回調函式並不是立即執行，而是會暫緩(延時)執行的一種回調函式，一般稱爲異步回調函式。

最後的執行結果是1 -> 3 -> 2 -> 4，也就是說，所有的同步回調函式都執行完成了，才會開始依順序執行異步的回調函式。如果你在瀏覽器上測試這個程式，應該會明顯感受到，2與4的輸出時，會有點延遲的現象，這並不是你的瀏覽器或電腦的問題，這是因爲不論你設定的setTimeout爲0，它要回到主執行緒上執行，仍然需要按照內部事件迴圈所設定的時間差，在某個時間點才會回來執行。

這個程式執行的流程，可以看這個在loupe網站的流程模擬，輸出一樣在瀏覽器的主控台中可以看到。

由這個範例中，可以看到異步回調函式執行比同步回調函式更慢，異步回調函式還有另一個名稱是延時回調(defer callback)，是用延時執行特性來命名。這只是一種因應特別情況所採用的函式執行方式，例如需要與外部資源存取(I/O)、DOM事件處理或是計時器的情況。等待的時間則是在Web API中，等有外部資源有回應了(或超時)才會加到佇列中，佇列裡並不會執行函式中的程式碼，只是個準備排隊進入主執行緒的機制，函式一律在主執行緒中執行。

> 關於函式的異步執行與事件迴圈一些原理的說明，請再參考[異步執行與事件迴圈]的章節裡的內容。

回調函式的複雜性

callback(回調)運用在瀏覽器端似乎並沒有想像中複雜，一個事件的處理範例大概會像下面這樣:

```
const el = document.getElementById('myButton')

el.addEventListener( 'click', function(){
     console.log('hello!')
}, false)
```

你也可以把callback寫成另一個函式定義，看起來會更清楚:

```
function callback(){
     console.log('hello!')
}

const el = document.getElementById('myButton')

el.addEventListener('click', callback, false)
```

AJAX是另一個常使用的情況，內建的XMLHttpRequest物件的行爲類似於事件處理，而且都打包好好的。實際上onreadystatechange這個屬性，就是XMLHttpRequest物件在處理事件用的callback(回調)函式。以下爲一個簡單的範例:

```
var xhr = new XMLHttpRequest();

xhr.onreadystatechange = function() {
    if (xhr.readyState == XMLHttpRequest.DONE ) {
       if (xhr.status == 200) {
           document.getElementById('myDiv').innerHTML = xhr.response
       }
       else if (xhr.status == 400) {
          console.log('There was an error 400')
       }
       else {
           console.log('something else other than 200 was returned')
       }
    }
}

xhr.open('GET', 'ajax_info.txt', true)
xhr.send()
```

"匿名函式"、"函式定義"與"函式呼叫"的混合

我會認爲回調函式會複雜的原因是主要是來自"匿名函式"、"函式定義"與"函式呼叫"的混合寫法。所以當在看程式碼時，你的腦袋很容易打結。

```
function func(x, cb){
  cb(x)
}

func(123456, function(value){
  console.log(value)
})
```

這例子很簡單，要分作幾個部份來看:

這是一個完整的"函式呼叫"，也就是說它是一個被執行的語句結構:

```
func(123456, function(value){
  console.log(value)
})
```

但其中的這一段是什麼，這個是一個"函式定義"，而且還是個"匿名函式"定義，它是一個callback(回調)函式的定義，它代表了func函式執行完後要作的下一件事，這個定義是在func函式中的程式碼的最後一句被呼叫執行。:

```
function(value){
  console.log(value)
}
```

所以整個語法是代表"在函式呼叫時，要寫出下一個要執行的函式定義"，這就是常見回調函式的語法樣式。當然，你可以另外用一個函式來寫得更清楚:

```
function func(x, cb){
  cb(x)
}

function callback(value){
  console.log(value)
}

func(123456, callback)
```

不過，你可以發現幾件事情:

- callback(回調)的函式名稱，可以用匿名函式取代。(實際上callback的名稱在除錯時很有用，可以在錯誤的堆疊上指示出來)
- callback(回調)因爲是函式的定義，所以傳入參數value的名稱叫什麼其實都可以。
- callback(回調)其實有Closure(閉包)結構的特性，可以獲取到func中的傳入參數，以及裡面的定義的值。(實際上JavaScript中只要函式建立就會有閉包產生)

那麼要說到callback(回調)的最大優點，就是它給了程式開發者很大的彈性，允許開發者可以自訂下一個要執行函式的內容，等於說它可以提高函式的擴充性與重覆使用性。

回調地獄(Callback Hell)

複雜的情況是在於CPS風格使用callback(回調)來移往下一個函式執行，當你開始撰寫一個接著一個執行的流程，也就是一個特定工作的函式呼叫後要接下一個特定工作的函式時，就會看到所謂的"回調地獄"的結構，像下面這樣的例子:

```
step1(x, function(value1){
  //do something...
  step2(y, function(value2){
    //do something...
    step3(z, function(value3){
        //do something...
    })
  })
```

```
})
```

它的執行順序應該是step1 -> step2 -> step3沒錯，這三個都可能是已經寫好要作某件特定工作的函式。所以眞正是這樣的流程嗎？你可能忘了匿名函式(callback)也是一個函式，所以執行的步驟是像下面這樣才對:

1. step1執行後，"value1"已經有值，移往function(value1)執行
2. function(value1)執行到step2，step2執行到最後，"value2"已經有值，移往function(value2)執行
3. function(value2)執行到step3，step3執行到最後，"value3"已經有值，移往function(value3)執行
4. function(value3)執行完成

寫成流程大概是像下面這樣的順序，一共有6個函式要執行的流程，其中的這三個匿名回調函式的主要工作，是負責準備接續下一個要執行特定工作的函式:

```
step1 -> function(value1) -> step2 -> function(value2) -> step3 -> fu
```

那爲何爲不使用直接風格？而一定要用這麼不易理解的程式流程結構。上面已經有講爲什麼JavaScript中會大量的使用CPS的原因:

> 因爲有些I/O或事件類的函式，用直接風格會造成阻塞，所以要寫成異步的回調函式，也就是一定要用CPS

你可能會認爲阻塞有這麼容易發生嗎？是的，在JavaScript中要"阻塞"太容易了，它是單執行緒執行的設計，一個比較長時間的程序執行就會造成阻塞，下面的for迴圈就會讓你的按鈕按下去沒反應，而且幾個訊息都要一段時間執行完才會顯示出來:

```
const el = document.getElementById('myButton')

el.addEventListener( 'click', function(){
     alert('hello!')
}, false)

const aArray = []
for(let i=0; i< 100000000;i++){
  aArray[i] = i+10
}

console.log('aArray done!')

const bArray = []
for(let i=0; i< 100000000;i++){
  bArray[i] = i*10
}

console.log('bArray done!')
```

或許你會認爲在瀏覽器上讓使用者等個幾秒鐘不會怎麼樣，但如果在要求能讓多個使用者同時使用的伺服器上，每個使用者都來阻塞主執行緒幾秒，這個伺服器程式就可以廢了。不過，以異步執行的異步回調函式並不代表就不會阻塞，也有可能從佇列回到主緒行緒後，因爲需要CPU密集型的運算，仍然會阻塞到緒行緒的進行。異步回調函式，只是暫時先移到佇列中放著，讓它先不干擾目前的主執行緒的執行而已。這是JavaScript爲了在只有單執行緒的情況，用來達成並行(concurrency)模型的設計方式。

如果要配合JavaScript的異步處理流程，也就是非阻塞的I/O處理，只有CPS可以這樣作。

在伺服端的Node.js一開始就使用了CPS作爲主要的I/O處理方式，老實說是一個不得已的選擇，當時沒有太多的選擇，而且這原本就是JavaScript中對異步回調函式的設計。Node.js使用error-first(以錯誤爲主)的CPS風格，因爲考慮到callback(回調)要處理錯誤不容易，所以要優先處理錯誤，它的主要原則如下:

- callback的第一個參數保留給Error(錯誤)，當錯誤發生時，它將會以第一個參數回傳。
- callback的第2個參數保留給成功回應的資料。當沒有錯誤發生時，error(即第一個參數)會設定爲null，然後將成功回應的資料傳入第二個參數。

一個典型的Node.js的回調語法範例如下:

```
var fs = require('fs');

fs.readFile('foo.txt', 'utf8', function(err, data) {
   if(err) {
    console.log('Unknown Error');
    return;
  }
  console.log(data);
});
```

Node.js使用CPS風格在複雜的流程時，很容易出現回調地獄的問題，這是因爲在伺服器端的各種I/O處理會相當頻繁而且複雜。像下面這個資料庫連接與查詢的範例，出自Node.js MongoDB Driver API:

```
// A simple query using the find method on the collection.

var MongoClient = require('mongodb').MongoClient,
  test = require('assert');
MongoClient.connect('mongodb://localhost:27017/test', function(err, 

  // Create a collection we want to drop later
  var collection = db.collection('simple_query');

  // Insert a bunch of documents for the testing
  collection.insertMany([{a:1}, {a:2}, {a:3}], {w:1}, function(err, 
    test.equal(null, err);

    // Peform a simple find and return all the documents
    collection.find().toArray(function(err, docs) {
      test.equal(null, err);
      test.equal(3, docs.length);

      db.close();
    });
  });
});
```

上面這個範例還算簡單，但裡面的回調函式有3個，函式呼叫了11個，再複雜的話就會更不好維護，我會認爲這是一種舊時代的程式碼組織方式的弊病，或是當時不得已的解決方案，當可以採用更好的語法結構來取代它時，這種語法未來大概只會出現在教科書中。現在這種寫法都是一般都已經不建議使用。

現在已經有很多協助處理的方式，回調地獄可以用例如Promise、generator、async/await之類的語法結構，或是Async、co外部函式庫等，來改善或重構原本的程式碼結構，在往後維護程式碼上會比較容易，這些才是你現在應該就要學習的方式。

此外，隨著技術不斷的進步，現在的JavaScript也已經有可以讓它使用其他執行緒的技術，有Web Worker，或是專門給Node.js使用的child_process模組與cluster(叢集)模組。

反樣式(anti-pattern)

回傳callback

callback(回調)有個很常見的反樣式，它會出現在如果回調除了要進行下一步之外，還要負責處理函式在執行中途的錯誤情況，例如:

```
//這是錯誤的寫法，最後的callback()依然會執行
function foo(err, callback) {
    if (err) {
        callback(err)
    }
    callback()
}
```

爲了讓最後的callback()不執行，可以正確的作錯誤處理，有可能會寫成這樣:

```
//相當不好的寫法
function foo(err, callback) {
    if (err) {
        return callback(err)
    }
    callback()
}
```

但是`return`用在callback上是個不對的樣式，正確的寫法應該是要用下面的寫法:

```
function foo(err, callback) {
    if (err) {
        callback(err)
        return
    }
    callback()
}
```

或是用`if...else`寫清楚整個情況，但這個樣式也不是太理想，CPS風格寫法的最後一行應該就是個回調函式:

```
function foo(err, callback) {
    if (err) {
        callback(err);
    } else {
        callback();
    }
}
```

參考資源

- By example: Continuation-passing style in JavaScript
- Asynchronous programming and continuation-passing style in JavaScript
- Enforce Return After Callback

Closure 閉包

Closure 閉包

閉包定義

> Closure這個字詞是由Close與字尾-ure所構成，-ure有"動作"、"進行"或"結果"的意思，如果close是關閉的意思，clousure就是關閉的結果或動作，它可以作爲名詞或動詞使用，中文有"封閉"、"終止"或"結束"的意思。

由於JavaScript語言中的函式是頭等函式(first-class function)的設計，代表函式在語言中的應用上享有與一般原始資料類型的值同等地位，函式可以傳入其他函式作爲傳入參數，可以當作另一個函式的回傳值，也可以指定爲一個變數的值，或是儲存在資料結構中(例如陣列或物件)，在語言中甚至是有自己獨有的資料類型(`typeof`一個函式回傳值是'function')。

閉包是一種資料結構，包含函式以及記住函式被建立時當下環境。

> 由於"閉包"這個字詞有多層意義，你可以說它是一種技術，或是一種資料結構，或是這種有記憶環境值的函式。

在JavaScript中每當函式被建立時，一個閉包就會被產生，閉包是一個函式建立時的就有的自然特性。雖然我們經常使用函式中的函式，也就是巢狀(nested)函式(或內部函式)的語法結構作爲範例來說明閉包，它也是一種最常見的、可被重覆利用閉包的語法樣式，但並不是只有巢狀函式才能產生閉包。

```
function aFunc(x){
  function bFunc(){
    console.log( x++ )
  }
  return bFunc
}

const newFunc = aFunc(1)
newFunc()
newFunc()
```

bFunc可以不需要名稱，直接用return匿名函式的語法更簡潔:

```
function aFunc(x){
  return function(){
    console.log( x++ )
  }
}
```

用箭頭函式更是簡潔，已經快要可以寫成一行了:

```
function aFunc(x){
  return () => console.log( x++ )
}
```

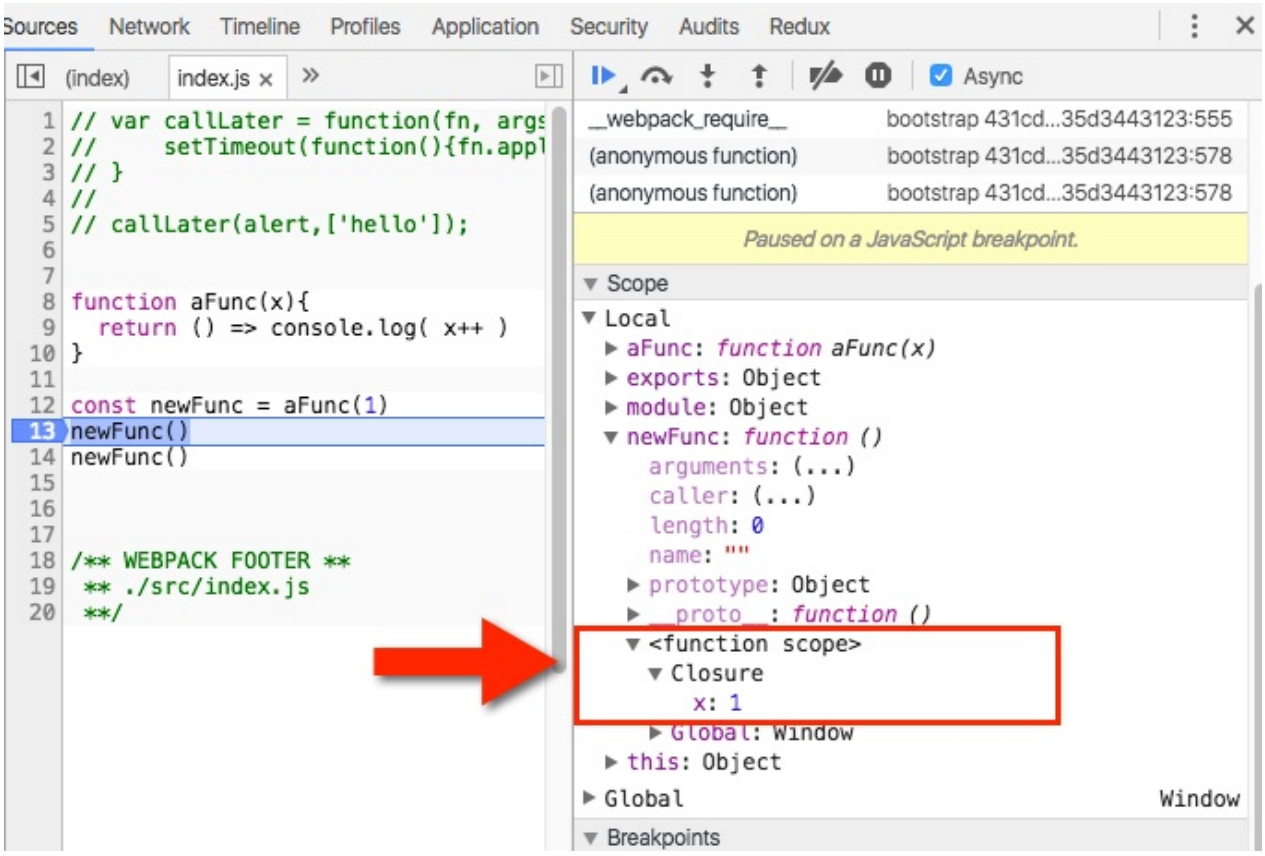
執行這個範例後，你會發現x值會在aFunc呼叫後陰魂不散的還留在新的newFunc函式裡，每

當執行一次newFunc函式，x值就會再+1。

不過，用這個這個閉包範例可能會產生誤解的地方，在於以下幾點:

- "函式呼叫"與"函式建立"實際上是兩件事，在aFunc"函式呼叫"過後，newFunc才是"函式建立"，這時候newFunc中的閉包結構才會產生。
- 匿名函式的使用。閉包並不是只會在匿名函式才會產生，純粹爲了簡化語法使用匿名函式。
- 閉包不是只會產生在巢狀(內部)函式的回傳時。所有函式在建立時都會產生閉包。

要觀察閉包中所記錄的環境變數值，可以從瀏覽器的除錯器中看到，像上面的範例如果用瀏覽器除錯器在第一個newFunc函式加入中斷點時，執行後應該可以看到像下面的圖:



圖中可以看到作用域(Scope)會出現一種名稱爲"Closoure(閉包)"的變數值，這是可以觀察閉包中的環境變數值的方式。

> 註: 在閉包中所記憶的變數與值，通常稱爲自由(free)變數或獨立(independent)變數，這些變數是在函式中使用，但被封入作用域之中。

典型範例圖解

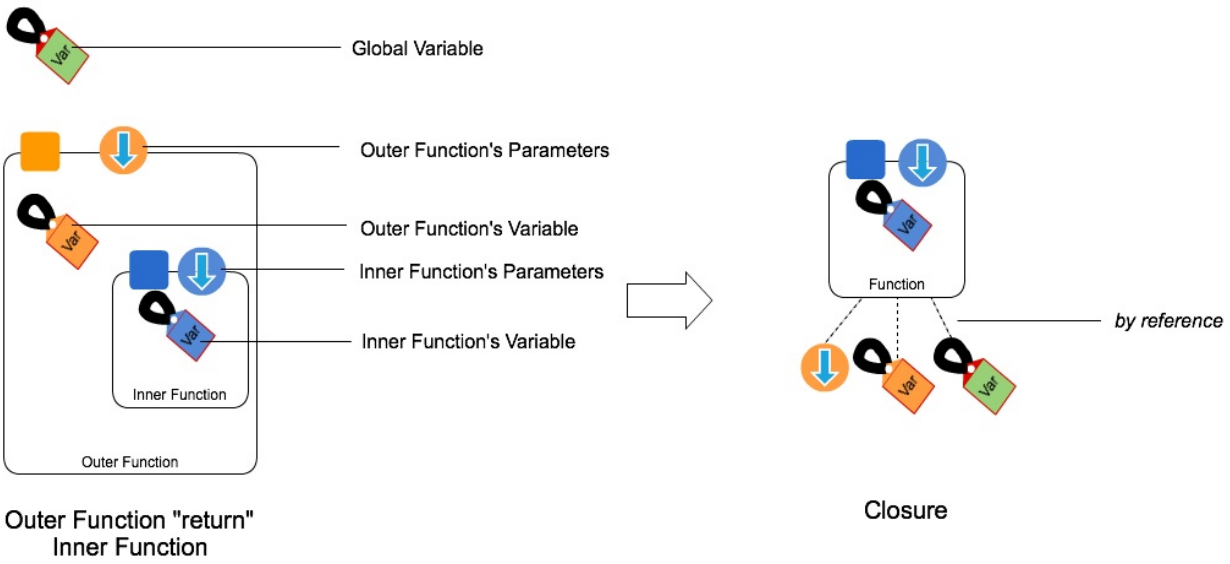
一個典型的Closure(閉包)會長這個樣子：

```
const varGlobal = 'x'

function outer(paramOuter){
  const varOuter ='y'

  function inner(paramInner){
```

```
    const varInner ='z'

    //print
    console.log(varGlobal)
    console.log(varOuter)
    console.log(varInner)
    console.log(paramOuter)
    console.log(paramInner)
  }

  return inner
}

const func = outer('a')
func('b')
```

圖解

用簡單的圖來表示上面的例子，內部(Inner)函式被回傳後，除了自己本身的程式碼外，也會捕抓到了環境的變數值，記住了執行當時的環境：



為什麼可以這麼作？

要深究爲什麼可以這麼作的原因，其實可以從下面兩個方面來看，當然複雜的底層實作就先不討論了，基本上這是一個語言的特性就是:

- 函式在JavaScript中的設計：函式可以像一般的數值使用，可以在變數、物件或陣列中儲存，也可以傳入到另外的函式裡當參數，也可以當回傳值回傳。
- 函式作用域連鎖規則：內部函式可以看到(或存取得到)外部函式，而形成一個Scope Chain(作用域連鎖)，內部函式可以有三個作用域：
    - 自己本身的
    - 外部函式的
    - 全域的

閉包的記憶環境

閉包的最大特點(賣點)就是它會記憶函式建立時的環境，也就是內部函式所能存取得到的作用域連鎖中的所有變數當下的值。

那麼一個問題會由然而生，閉包是完全複製(copy)了這些值？還是參照(refer)指向這些值而已？答案是"參照**(refer)**而非複製"。

最常用來解說的這個概念的是下面這個典型範例，因爲它用了異步的回調函式，所以你可能要對異步回調有點概念才看得懂:

```
//錯誤示範
function counter() {
    let i = 0
    for (i = 0; i< 5; i++) {
        setTimeout(function() {
            console.log('counter is ' + i)
        }, 1000)
    }
}

counter()
```

這個counter是我們希望能利用閉包結構，記憶其中的變數i，最後希望的結果是0,1,2,3,4這樣，不過上面這個範例是個錯誤的示範，並不會產生這個結果，最後的結果是是5,5,5,5,5。

會造成這個結果的原因在這一節的最前面就說完了，因爲"閉包結構中所記憶的環境值是用參照指向的"，setTimeout中的異步回調會先移到工作佇列中準備延時執行，等它回來主執行緒執行時，i老早就跳出迴圈執行，而且還變成了5，所以接下來執行的這些異步回調函式，能獲取到的i值全部都是5。這個問題可以用好幾種解法，其實都是要考驗你的概念與基礎知識。

第一種是解除原先counter中閉包的結構，也就是說你已經知道這樣會造成閉包，而且會記憶(參照)到環境值，乾脆讓這個情況解除，像下面的寫法:

```
function counter(x) {
  setTimeout(function() {
      console.log('counter is ' + x)
  }, 1000)
}

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

新的寫法中，counter函式中確保不會再帶有函式裡面的變數值，而是用一個傳入參數值讓setTimeout中的回調函式去作對應的輸出，因爲每次傳入counter函式的x值都不同，也就能控制回調函式所能存取得到的環境值，所以能確保異步回調的輸出每次都是不同的。

第二種是想辦法鎖住setTimeout中回調函式的環境值，這個解法的有點像武俠小說中的慕容家族的"以彼之道，還之彼身"的感覺，用閉包來解決閉包的問題:

```
function print(i){
```

```
  return function(){
    console.log('counter is ' + i)
  }
}

function counter() {
  let i = 0
  for (i = 0; i< 5; i++) {
      setTimeout(print(i), 1000)
  }
}

counter()
```

setTimeout中的print(i)一但被執行，會回傳一個帶有閉包的函式，也就是會鎖住當下傳入的值，作爲setTimeout的回調函式。

第三種解決是要用IIFE(立即呼叫函式表達式)的樣式，這個樣式已經是很特殊的用法了，IIFE也有儲存閉包的環境狀態的功用:

```
for (let i = 0; i < 5; i++) {
  (function(value) {
    setTimeout(function(){
      console.log('counter is ' + value);
    }, 1000)
  })(i)
}
```

註: 在for迴圈中的語法稱爲"Immediately-Invoked Function Expression"（立即被呼叫的函式語樣）或簡稱爲"IIFE"

閉包的記憶環境例外變數

閉包只有以下兩個外部函式的環境變數是不會自動記憶的，相信你如果很仔細的看過上面的範例，就已經有發現了，這其實是內部函式的特性，這兩個本來就不算是作用域鏈的成員之一。除非你先用另外的變數指定這兩個值，不然內部函式是獲取不到的，實際上內部函式自己也是個函式，它也有自己的this與arguments值。這兩個例外值是:

- this: 執行外部函式時的this值
- arguments: 函式執行時的一個隱藏僞陣列物件

相關樣式

利用閉包這種特性，可以發展出很多適合各種情況應用的樣式，有許多常見的樣式或許你已經有看過，或是一直有使用到的。

柯里化(Currying)與部份應用(Partial application)

柯里化是一種源自數學中求值的技術，它與部份應用(Partial application)經常被一起討論，這些都是在程式設計上稱爲"部份求值"或"惰性(延時)求值"的一種技巧。JavaScript語言中可以使用閉包結構很容易地實現這個技術。這個技術可以應用到不同的複雜情況，這裡只是篇簡介而已。

要理解這個技術先理解其中幾個專用術語的不同定義:

- 應用(Application): 代表傳入一個函式所需的傳入值，然後最後得到回傳結果。
- 部份應用: 代表一個函式其中有部份的傳入值(一個或多個)被傳入，然後回傳一個已經有部份傳入值的函式。
- 柯里:　一個具有多個傳入參數的函式，轉變爲一個一次只傳入一個參數，只會回傳一個只有一個傳入參數的函式。也就是說把原本多個傳入參數的函式，轉變爲一次只傳入一個參數的函式，可以連續呼叫使用。

柯里化與部份應用雖然都是套件部份的傳入參數值，但它們不太相同，也有下面幾個很明顯的差異:

- 部份應用: 一次套用一個或多個傳入參數，回傳函式有可能與原來函式結構的不同。
- 柯里:　一次只套用一個傳入參數，回傳另一個函式，回傳的函式與原來的結構相同，直到所有傳入參數都被套用才會回傳值。

部份應用或柯里，常會用在固定某些已確定的參數值使用，在許多工具性函式庫或框架中經常被使用，以此提高函式的重覆使用性。基本上部份應用或柯里都是從左至右傳入參數值的。如果你要從右至左傳入參數值，類似像外部函式庫lodash中有從右至左的curryRight與partialRight函式，可能要再改寫或用這類的工具函式庫。另外也有一套知名的工具函式庫Ramda，它是完全以部份應用與柯里的特性爲核心的函式庫。

> 註:　Partial在專業術語中，中文也常會翻譯爲"局部"、"偏"，例如"partial　differential equation, PDE"是偏微分方程式的意思。
>
> 註: Curry的英文字詞是"咖哩"的意思，不過這裡是指這個技術以"Haskell　Curry(柯里)"數學家的名字來命名。

部份應用**(Partial application)**

部份應用按照定義並沒有那麼嚴格，就只要能套用部份的傳入參數值就行了，有很多種寫法也不一定要用閉包結構。以下是要改寫的原本函式:

```
//原本的函式
function add(x, y, z){
  return x+y+z
}
```

第一種寫法是用另一個函式來套用部份的值即可:

```
function addXY(z){
  return add(1, 2, z)
}

addXY(3)
```

第二種寫法是用函式物件中的bind方法，它可以回傳一個套用部份參數值的新函式(第一個參數值是context，也就是this值，這裡不需要):

```
const addXY = add.bind(null, 1, 2)

addXY(3)
```

第三種寫法是用閉包結構，不過要把原來的函式改寫才行:

```
//改寫
function add(x, y, z){
  return function(z){
    return x+y+z
  }
}

const addXY = add(1, 2)
addXY(3)
```

柯里化

柯里化會比較麻煩些，只能使用閉包結構來改寫原本的函式，例如下面的原本函式與柯里化後的樣子比較:

```
//原本的函式
add(x, y, z)

//柯里化後
add(x)(y)(z)
```

因爲JavaScript中有閉包的特性，所以要改寫是容易的，需要改寫一下原先的函式:

```
//原本的函式
function add(x, y, z){
  return x+y+z
}

//柯里化
function add(x, y, z){
  return function(y){
    return function(z){
        return x + y + z
    }
  }
}

add(1)(2)(3)
```

第一個傳入值x會變爲閉包中的變數被記憶，然後是第二個傳入值y，最後的加總是由閉包結構中的x與y與傳入參數z一起加總。這個範例中用了三個傳入參數，如果你沒辦法一下子看清楚，可以用二個傳入參數的情況來練習看看。

IIFE(立即呼叫函式表達式)

IIFE本身就是一種運用閉包與匿名函式立即執行的樣式，它的常見基本語法(實際有很多種寫法)有下面這兩種:

```
(function(){ /* code */ }())

(function(){ /* code */ })()
```

IIFE是一種會在建立時就會立即執行的匿名函式，經常用於封閉住一個作用域，避免與全域作用域污染。一個簡單的IIFE範例如下，counter是一個會鎖住函式裡面的變數值的閉包，這

個樣式通常會用來模擬靜態變數。

```
const counter = (function() {
    let i = 1

    return function() {
        console.log(i++)
    }
}())

counter() //1
counter() //2
```

物件封裝/物件工廠

使用閉包來產生物件，而不是用Prototype與new運算符。程式碼來自Why use "closure"?：

```
// 宣告一個工廠
function newPerson(name, age) {

  // 在閉包中儲存訊息
  const message = name + ', who is ' + age + ' years old, says hi!'

  return {

    // 定義同步執行的函式
    greet: function greet() {
      console.log(message)
    },

    // 定義異步執行的函式
    slowGreet: function slowGreet() {
      setTimeout(function () {
        console.log(message)
      }, 1000)
    }

  }
}

const tim = newPerson('Tim', 28)
tim.greet()
```

模組(Module)樣式

模組樣式是用來模擬物件中的私有成員上(私有屬性與方法)與公開成員，JavaScript語言中的物件導向本身並沒有這種設計，這個樣式使用了IIFE。以下程式碼來自JavaScript Closures and the Module Pattern

```
var Module = (function() {
    // 以下的方法是私有的，但是可以被公開的函式存取
    function privateFunc() { ... }
```

```
    // 回傳要指定給模組的物件
    return {
        publicFunc: function() {
            privateFunc() // publicFunc 可以直接存取 privateFunc
        }
    }
}())
```

模組樣式有另一種變型，稱之爲"暴露的模組樣式(Revealing Module Pattern)"，語法會比原來的模組樣式容易理解得多。

```
var Module = (function() {
    // 所有函式現在可以互相存取
    var privateFunc = function() {
        publicFunc1()
    }

    var publicFunc1 = function() {
        publicFunc2()
    }

    var publicFunc2 = function() {
        privateFunc()
    }

    // 回傳要指定給模組的物件
    return {
        publicFunc1: publicFunc1,
        publicFunc2: publicFunc2
    }
}())
```

閉包使用時注意事項

閉包結構的各種運用是一種高消費的語法，主因在於它需要尋遍整個作用域連鎖與記憶環境。這些都是需要時間來完成的動作。閉包結構也會鎖住額外的記憶體，所以要小心運用在記憶體高使用的語法上，例如迴圈或定時執行的函式。一般情況下，我們不需要擔心記憶體回收的問題，JavaScript引擎有很好的GC機制來回收不需要使用的記體體。

小結

總結一下所有本章節的內容，讓你對閉包的理解有比較清楚的思維。

- 所有函式在建立時都會產生閉包。
- 閉包不是只會產生在巢狀(內部)函式的回傳時，巢狀(內部)函式是一種最常利用閉包結構的樣式，因爲它可以重覆使用閉包中的記憶環境。
- 閉包所記憶的環境，其原理是來自作用域連鎖的設計，內部函式可以看(獲取)到外部函式的變數值與傳入參數值。
- 函式的`this`值與`arguments`值並不屬於作用域連鎖，所以不包含在閉包記憶的環境中。
- 閉包的運用是一種高消費的語法。

參考資料

- How do JavaScript closures work?
- Master the JavaScript Interview: What is a Closure?
- Understanding JavaScript Closures
- Really Understanding Javascript Closures

異步程式設計與事件迴圈

異步程式設計與事件迴圈

JavaScript程式語言在設計時，需考量異步、單執行緒與非阻塞I/O等等的問題

JavaScript程式執行的確都在單一個執行緒(Single Thread)中的。聽起來有點不可思議，現在的電腦硬體不都是多核心(多執行緒)而且資源豐富嗎？這個設計對於程式執行不會有問題嗎？

我們用另外的角度來思考，爲何JavaScript會這樣設計，仍然可以符合程式執行的需求的幾個原因:

- JavaScript從一開始就是這樣設計，它執行的主要環境是在瀏覽器上，只有一個使用者，而且是在資源受限的環境中執行，這是一個很合理的設計。
- JavaScript程式執行雖然是單執行緒，但伺服器或瀏覽器執行環境並不是: 表面上看起來是只有一個執行緒在執行JavaScript程式，但實際上在背後有數個的其他在執行環境中的執行緒，在輔助程式碼的執行。
- 外部資料的執行時間，大部份都是等待時間: 像連接資料庫、執行資料庫查詢等等的執行語句，眞正執行是在資料庫裡，程式只是傳送對應的查詢語句而已，並不是在JavaScript程式中執行查詢，這種情況對JavaScript程式而言，大部份的執行時間中都是在等待查詢結果而已。讀寫檔案、網路連線要求與回應、傳送資料等等，都有類似的情況。大量而且複雜的運算，的確是在語言的程式中執行，不過JavaScript原本就不是設計用來作複雜運算用的，或許要在特殊的執行環境中才有辦法作這件事。

要理解JavaScript是如何執行程式，首先要先理解同步(Synchronous, sync)與異步(Asynchronous, async)程式執行的差異。

> 由於JavaScript中的執行區分是以執行上下文(EC)作爲一個單位，也就是以函式區分，所以一般要討論異步或同步執行，也通常會用異步執行的函式或同步執行的函式來區分。程式碼中的每個語句一定是同步執行的。而異步執行函式必定會使用CPS風格的異步回調函式，這是JavaScript中的設計，關於異步回調函式，可以參考特性篇的"Callback 回調"章節的內容。

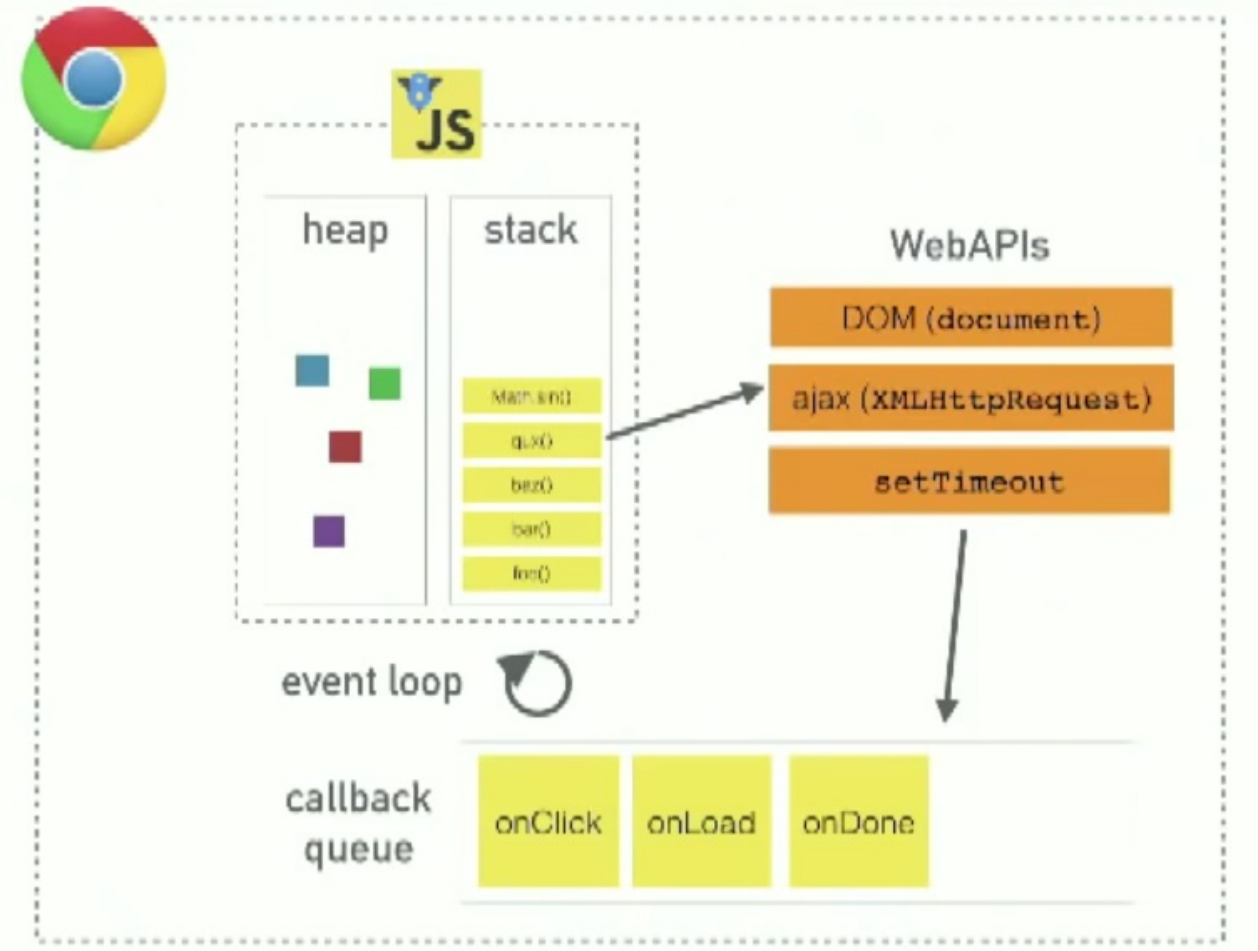
同步程序執行是指程式碼的執行順序，都是由上往下依順序執行，一個執行程序完成後才會再接著下一個，一般的程式語言都是按照這樣的流程來執行，JavaScript語言也不例外。例如像連接資料庫存取資料的程式，應該會遵守下列的步驟進行:

1. 連接資料庫(給定帳號、密碼、主機、資料庫名)
2. 執行資料庫查詢語句
3. 取得資料，格式化資料

這對於"從資料庫查詢資料"的這種程式本身並沒有太特別的地方，一般都是這樣執行沒錯。但對於JavaScript這種只有單執行緒的程式語言，這樣作會造成阻塞(blocking)，也就是說當這個資料庫查詢的執行程序，需要很長的一段時間才能結束時，在這期間其他的操作都會停擺，像是滑鼠要點按按鈕之類的功能，就完全沒有作用。因此，我們需要用另一種不同的方式，來進行這類會阻塞其他程式的執行，也就是異步程式執行的方式。

異步程式執行的作法，是使用異步callback(回調)函式的語法，讓會造成阻塞的程式組成一個異步回調函式，先丟往一個任務佇列(task queue)先丟，在之後的某個時間再回傳它的值與狀態回來。這些函式裡的程式碼多半都是與外部資源存取的I/O有關，如果需要等待的話，是在實作的API裡(外部模組)，並不是在佇列或語言執行緒中，超時或有回應後再加入到佇列中，事

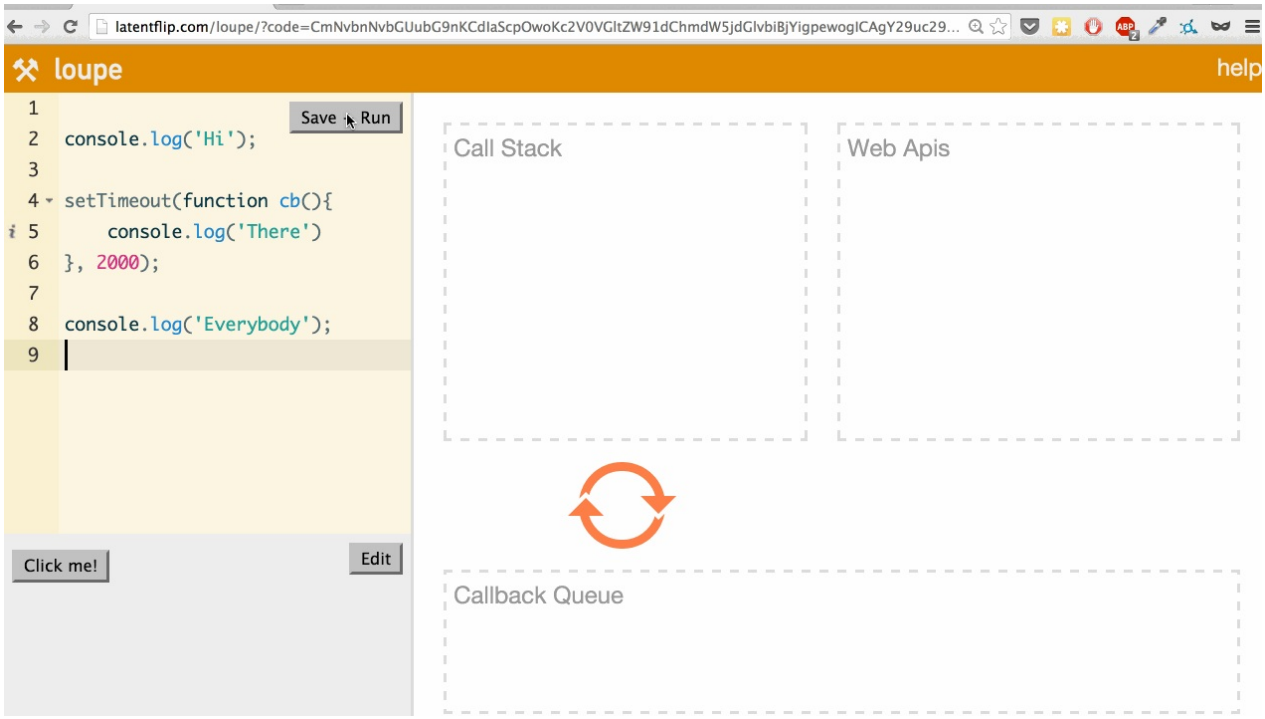
件迴圈在主執行緒完全沒有其他的EC時，再加回到主執行緒中執行。下面這張圖是出自影片Philip Roberts: What the heck is the event loop anyway?|JSConf EU 2014。



最常使用的例子是用`setTimeout`這個內建的方法，它會在某個設定的時間的執行其中的callback(回調)函式傳入值一次:

```
console.log('a')

setTimeout(
    function cb(){
        console.log('b')
    }, 1000)

console.log('c')
```

按照同步程式的執行順序，應該是`a -> b -> c`這個結果，但眞正的結果是`a -> c -> b`，也就是`cb`這個在`setTimeout`中的callback(回調)函式，在程式執行到這一行時，先被移出主執行緒外面，先到任務佇列去了，最後等主執行緒空了，在某個時間才會再回到主執行緒中，執行其中的輸出值的動作。下面是類似的程式的執行模擬圖解，來自JavaScript's Call Stack, Callback Queue, and Event Loop。如果你覺得圖解不夠，你可以直接到這個loupe網站來執行看看。



註: 請不要誤解了，並不是所有的callback(回調)函式都是會丟到任務佇列(task queue)之中執行。只有經過特殊設計過的異步callback(回調)才會這樣作。

另一個最常見的例子是AJAX技術的實作，AJAX的全名是"Asynchronous JavaScript and XML"，它在名稱上就有異步的字詞，是運用`XMLHttpRequest`物件與網站伺服器溝通的一種技術，我們在特性篇有一篇專文介紹它。一個簡單範例如下:

```
const xhr = new XMLHttpRequest()

xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      console.log(xhr.responseText)
    }
}

xhr.open('GET', 'test.txt', true)
xhr.send()
```

AJAX技術可以不需要刷新瀏覽器的頁面，它是一種在瀏覽器背後模擬與伺服器要求與回應溝通的機制，因爲是與外部環境作溝通，有可能會因爲網路連線或伺服器的狀況造成等待時間，所以一開始就被設計爲異步的API，也就是說當AJAX執行時，`onreadystatechange`屬性中的這個回調函式，也會先往一個任務佇列(task queue)丟去，之後等主執行緒清空後，在某個時間點再回來回傳回應的值。

註: 任務佇列(task queue)也有其他名稱的講法，例如消息佇列(message queue)、事件佇列(event queue)、回調佇列(callback queue)

異步程式設計中，JavaScript使用Event Loop(事件迴圈)的設計來協助達成異步函式的執行，它是一種並行(Concurrency)的模型，事件迴圈可以想成是一個內部迴圈功能，它會不斷地每一段時間就檢查佇列與執行程序，然後決定是否要把佇列中的任務程序，移回目前JavaScript程式的主要執行緒中(呼叫堆疊)執行，原則其實簡單，"當只有在呼叫堆疊空空如也時，才會把佇列中任務移回呼叫堆疊"。

你或許會認爲JavaScript因爲只有單一條執行緒的並行(Concurrency)模型，根本就不是"同時執行"的，同一時間還是只能作一件事的確是個事實，不過單執行緒那也只能這樣設計。單只有一條執行緒的並行模型，並非完全沒有任何的優點，相較於多執行緒的設計複雜，它會比較簡單，而且以作同樣的事情(例如服務同樣多的使用者連線)時，消耗的資源會比較少。

Event Loop(事件迴圈)直接由名稱上理解，主要是爲了事件(Events)所設計的，存放有被進行分派的事件函式程序到任務佇列中，內部的迴圈功能會不斷的重覆檢查，目前現在瀏覽器上的各種HTML元素是不是有被觸發事件，當有事件被觸發時，就立即把函式，先移到JavaScript的呼叫堆疊中來執行，當然它也是一種異步的回調函式，所以也要先移往任務佇列中，等其他在呼叫堆疊中的程式都執行完了，才會回到主執行緒來執行。

那麼，什麼樣的執行程序(函式)會被移到任務佇列之中？它的順序又是如何的？

依照W3C的定義，Event Loop(事件迴圈)中可能會有一個以上的任務佇列(task queue)，一般認爲就是用以下幾種分出不同的佇列，但實作部份要視瀏覽器實作決定，其中的順序是依照FIFO(先進先出)，以下是幾種會包含的任務:

- Events(事件): EventTarget物件異步分派到對應的Events物件
- Parsing(解析): HTML parser
- Callbacks(回調): 呼叫異步回調函式
- 使用外部資源: 資料庫、檔案、Web I/O
- DOM處理的反應: 回應DOM處理時的元素對應事件

註: 對照Call Stack(呼叫堆疊)是FILO(先進後出)或LIFO(後進先出)的順序。

在瀏覽器端的JavaScript程式語言中，除了一般的事件分派外，還有少數幾個內建的API與相關物件有類似的異步機制，有一些簡單的樣式可以利用它們模擬出異步的執行程式:

- setTimeout
- setInterval
- XMLHttpRequest
- requestAnimationFrame
- WebSocket
- Worker
- 某些HTML5 API，例如File API、Web Database API
- 有使用onload的API

而在伺服器端(Node.js)的JavaScript程式，大部份的API都會考量到異步的問題，尤其是與I/O相關的，是半點都不能夠有阻塞的情況發生，這稱爲非阻塞I/O(Non-blocking I/O)的設計方式，都有對應的異步呼叫方式。

註: 有個說法是說"JavaScript是有非阻塞I/O特性的程式語言"，比較好的理解應該是"JavaScript是個沒辦法阻塞住I/O的程式語言"，畢竟只有一條執行緒，一但塞住了就會無法正常運作。因此"非阻塞I/O(Non-blocking I/O)"才會成爲它的一種特性。

不過，對於異步程序(函式)也有一些問題要考量:

- 異步程序(函式)間沒辦法保証執行的時間順序: 在複雜的多個的異步程序(函式)，可能有很多存在於任務佇列的等待被執行的程序，也可能有一個以上的任務佇列，它們的執行時間與順序的沒有辦法保証。
- Run-to-completion(一執行就要執行到完成): 每個任務程序(函式)中的程式語句都會只要一執行就會到完成，所以基本上任務程序(函式)中都是同步執行的程序，一個完成才會接著下一個任務。這個特性有可能會導致過長時間的任務，阻塞到Event Loop(事件迴圈)的進行，建議是把任務切割成更小的任務。

因此，異步執行程式並非只有單純的幾個函式呼叫這麼簡單，有很多情況是需要整個程式的執行流程一併考慮的，例如上面的資料庫查詢的例子，如果後面還有一些要把查詢到的資料進行其他運算的程式碼，就會需要進行執行流程上的分離或合併(例如異步合併同步、同步中的異步、異步中的同步等等)。

異步的程式流程的組織方式，現在也有好幾種作法:

- Promise語法結構(ES6)
- Generators (ES6)
- 使用工具函式庫，例如Async
- Async函式(ES7)

常見問答

異步函式執行比同步函式執行快？

沒有。一定比較慢。

事件迴圈在呼叫堆疊與任務佇列切換要加入的異步函式，是需要一定時間的。你可以把異步執行的函式，視爲一種"暫緩執行"的函式，既然是暫緩執行，一定不會比直接在呼叫堆疊中的執行的函式快。

JavaScript沒有辦法使用多執行緒之類的作法嗎？

現在有一些新的技術，可以使用到其他的執行緒，例如:

- Web Workers
- 伺服器端(Node.js)用的child_processes模組與cluster模組

異步執行函式裡面如果還有異步執行的其他函式，這樣的執行有順序規則可言嗎？

有。不過執行順序還是要視情況決定。

主執行緒的執行規則是同步規則，所以是一行接一行，先放到呼叫堆疊中。呼叫堆疊在執行時，是先進後出(FILO)的規則。

如果呼叫堆疊中看到異步的函式，會先移到任務佇列中，任務佇列中等事件迴圈看到呼叫堆疊沒有其他函式EC(執行上下文後)後才會把佇列中的EC移回主執行緒中。這個移回去的順序是依照先進先出(FILO)的規則。

異步中的異步，上面的流程會再重新作一遍，不斷循環直到所有程式碼都執行完畢。所以以幾個情況來說，如果假設都是相同延時(例如都是1000ms延時執行)的異步函式。

1. 相同的異步執行函式，看誰在全域EC中(也就是程式碼中)先被執行，誰就先完成執行
2. 異步只要差一層，在全域EC中(也就是程式碼中)的執行順序就會無關，愈多層的一定比較慢完成

下面的範例的輸出結果必定是`a->b`，因爲`setTimeout`中的時間代表加入到任務佇列的時間，只要加入佇列的時間一樣，就會依程式碼從上到下執行的順序爲順序。

```
function aFunc(value, cb){
  setTimeout(cb, 1000, value)
}
```

```
function bFunc(value, cb){
  setTimeout(cb, 1000, value)
}

aFunc('a', function cbA(value){console.log(value)})
bFunc('b', function cbB(value){console.log(value)})
```

但上面都是同樣的異步執行函式的規則，如果是有一點點的時間差，結果會變爲b->a，像下面的範例:

```
function aFunc(value, cb){
  setTimeout(cb, 1000, value)
}

function bFunc(value, cb){
  setTimeout(cb, 900, value)
}

function cbA(value){
  console.log(value)
}

function cbB(value){
  console.log(value)
}

aFunc('a', cbA)
bFunc('b', cbB)
```

時間差不只對同樣層的異步執行函式有用，對多層的情況也會影響，因爲時間差代表的是異步回調函式加到佇列的時間，如果這個時間晚於前一個多層異步函式加入又移回執行，然後又加入移回執行，那就只能比之前的慢。下面的範例說明了這點。

```
function aFunc(value, cb){
  setTimeout(cb, 1000, value)
}

function bFunc(value, cb){
  setTimeout(cb, 0, value)
}

function inCbB(value){
  console.log(value)
}

function cbB(value){
  setTimeout(inCbB, 0, value)
}

function cbA(value){
  console.log(value)
}

aFunc('a', cbA)
```

```
bFunc('b', cbB)
```

雖然aFunc中的異步執行函式cbA只有1秒的時間差才加到任務佇列中，但bFunc中的cbB會搶先加到任務佇列，然後回到呼叫堆疊中執行，inCbA再搶先加到任務佇列中，先執行完成，對我們來說短短一秒，實際上在電腦世界裡是可能有幾個小時的感覺差異。這個結果也是b->a

要確保異步回調函式的執行順序，這已經涉及到異步程序的執行流程的問題，只能使用新式的Promise、Generators、asyc/await或外部相關函式庫的組織方式。單純這樣用是沒辦法的。

小結

以下列出常見的對JavaScript語言"誤解"的講法:

- JavaScript中的callback(回調)函式都是異步執行的。
- JavaScript執行是多執行緒執行的。
- 異步執行的函式通常比同步的還快。
- JavaScript中的程式碼每段都是異步執行的。
- JavaScript會把異步的函式先移到佇列中去執行，執行後再把結果返回到呼叫堆疊中。
- 大部份的事件處理函式是直接執行的函式，是同步執行的而不是異步執行的。

上面全部都是"錯誤"的。如果你有一點點懷疑，請再看一遍這篇的內容吧。

參考資料

- JavaScript's Call Stack, Callback Queue, and Event Loop
- loupe
- Philip Roberts: What the heck is the event loop anyway? | JSConf EU 2014
- Concurrency model and Event Loop(MDN)
- Evolution of Asynchronous JavaScript
- Node Hero - Understanding Async Programming in Node.js
- The JavaScript Event Loop: Explained

事件處理

事件處理

JavaScript是一個以事件驅動(Event-driven)的程式語言。事件驅動程式設計的主要流程，是由圖形化使用者操作介面(UI)的互動事件爲主要核心，藉由事件的觸發動作(滑鼠點按、鍵盤輸入等等)或是感應器的訊息，來啓動整體的程式流程。

依據"異步程式設計與事件迴圈"一章的內容中的說明，在JavaScript中有一個不斷偵測事件發生的事件迴圈(Event Loop)，所有在網頁上的DOM元素註冊的事件，都會進入一個佇列(queue)中，等待被觸發事件，然後作出相對的回應送還給使用者。負責實作事件迴圈的是瀏覽器環境，佇列有可能不只一個，一般認爲至少會依照W3C所定義的各種不同會發生佇列情況，也就是有5種佇列，包含事件、回調、使用外部資源等等。

我們在這裡所稱的事件(Event)，通常我們把它稱爲瀏覽器事件(browser events)，這些是網頁中的DOM(Document Object Model)元素的事件，標準制定者是W3C組織，然後由ECMAScript負責支援的角色。也有一些特定的事件是由瀏覽器品牌自行制定，這不在我們討論的範圍之中。

近年來實現了伺服器端的JavaScript語言執行環境的Node.js，它並沒有這一系列直接可以使用的事件，內建的事件模型是Node.js自行設計的，不過它採用了相當類似的樣式，使用一個`EventEmitter`類別的物件實體來作事件處理，在底層也使用事件迴圈(Event Loop)的設計來達成。不過，你如果要進入伺服器端的事件處理領域，建議先從熟悉瀏覽器端的事件先著手，你會發覺它們有很多相似的設計與樣式。

DOM事件

DOM(Document Object Model, 文件物件模型)是由W3C組織所制定的跨平台與跨程式語言的應用程式介面，它是把HTML、XHTML、XML文件的視爲樹狀的結構，每個節點都是代表文件的一個物件。DOM的標準大戰可以從第一次瀏覽器大戰，由Netscape與微軟IE開始算起，到2015年已經是第4級(Level 4)的標準。

DOM事件則是可以註冊各種事件處理器/監聽器(event handlers/listeners)在DOM的節點元素上，在DOM標準的第2級(Level 2)時，W3C標準化了DOM中的事件模型，統一不同瀏覽器中的事件模型差異。這個標準也就是我們現在在瀏覽器上使用的事件模型基準。

JavaScript程式語言一直以來對瀏覽器中的物件模型(object model)與其事件模型，有非常高的支援性，雖然DOM標準從來就不是只專門制定給JavaScript單一種語言使用的，不過因爲JavaScript的使用群太高了，現在也差不多就是制定給它用的。

那麼，什麼樣的DOM元素中會對應什麼樣的DOM事件，或是什麼樣的操作方式會觸發怎麼樣的事件？

這都由DOM中的標準來制定，例如像這個DOM事件列表非常的長，從滑鼠、鍵盤輸入到HTML表單...等等，近年來由於觸碰式的行動裝置很流行，W3C也開始制定觸碰事件(Touch Events)的標準。不過，有幾個瀏覽器品牌自己實作了不少非標準的事件，有些事件是只用於該瀏覽器的特殊用途。

註: DOM events (維基百科)

Event物件(介面)

JavaScript中定義了Event物件，其中包含了事件通用的屬性與方法，例如事件的對象元素等等，事件的來源是來自DOM，所以這個物件通常稱之爲Event介面。W3C制定了標準的Event介面規格。其中有幾個常用的屬性與方法分列如下。

Event物件屬性(以下屬性都是只能讀不能寫):

- currentTarget: 目前的事件對象
- target: 分派事件的原始對象
- type: 事件的類型，共有數十種
- bubbles: 冒泡狀態，布林值，true代表會在DOM中往上冒泡
- cancelable: 事件是否爲可取消的，布林值

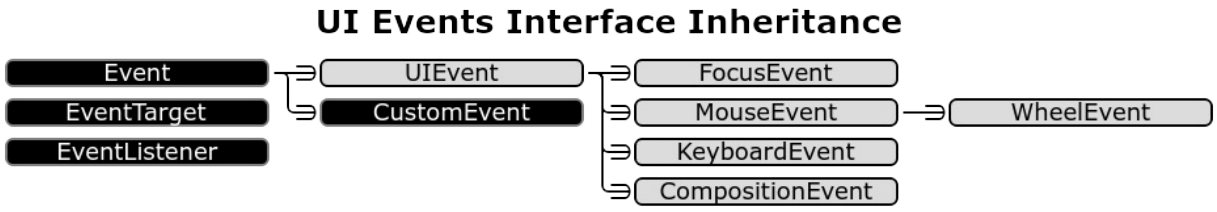
註: 目前的事件對象(currentTarget)與分派事件的原始對象(target)，在冒泡與捕捉階段，其中的值會不同。

Event物件方法:

- preventDefault(): 取消事件的行爲(需爲可取消的事件)，但無法阻止事件的傳播(propagation)
- stopPropagation(): 停止事件的傳播行爲

註:事件的傳播行爲在下面的章節有再加說明。

Event介面只是個基礎物件，從它擴充了使用於特定情況的事件，包含對特定事件的資訊，詳見以下的階層圖(出自這個網站):



依照事件階層圖中，UIEvent與CustomEvent(自訂事件)繼承自Event物件，從UIEvent中又擴充出各種不同的對應事件，例如針對滑鼠與鍵盤的，FocusEvent是設計給Focus(鎖定、聚焦)事件使用的，CompositionEvent是針對輸入文字事件使用的，這是因爲輸入文字並不一定單純使用鍵盤(用輸入法輸入或用語音輸入等等)，它與鍵盤事件也可以相互輔助。而WheelEvent是有滾輪的設備使用的。

註: 許多外部函式庫例如jQuery，對於Event物件會以W3C的標準進行擴充。

EventTarget物件(介面)

EventTarget物件則是JavaScript所設計的一種當作介面的物件，它可以接收事件，以及讓監聽者註冊到上面。DOM元素、document、window物件，是最常見的EventTarget物件，另外也有其他的物件可作爲EventTarget。EventTarget物件中有三個方法:

- `addEventListener`: 在事件對象上加入事件監聽者
- `removeEventListener`: 從事件對象移除事件監聽者
- `dispatchEvent`: 送出事件給所有有訂閱的監聽者

另外需要提到的一點，W3C標準中對於EventListener也有定義它是一個介面，作爲事件監聽者之用，不過JavaScript語言在所有的函式中都有實作這個介面，所以事件監聽者在呼叫handleEvent(處理事件)方法時，相當於呼叫函式。EventListener(事件監聽者)或稱爲事件處理函式，可以自動得到事件傳入參數值，以此可以存取得到事件的屬性與方法，例如以下的範例:

```
const me = document.getElementById('me')

me.addEventListener('click',
    function(e){
      console.log(e.currentTarget)
      console.log(e.target)
      console.log(e.type)
      console.log(e.bubbles)
      console.log(e.cancelable)
      e.stopPropagation()
    },
    false)
```

事件處理模型

現行的事件處理模型通常會使用"監聽"(listen)的方式，作爲事件處理的標準樣式，因爲DOM標準制定歷史版本不同，實際上有好幾種不同的方法可以作事件處理。以下分述這幾級的差異，其中第一種與第二種是舊的方式，不建議使用。

內聯模型

這種方式是最簡單的，也稱爲內聯模型(Inline model)。它直接在HTML裡DOM元素中標記中使用，每個元素都會實作對應的可使用事件，名稱都會是像"onxxxxx"這樣的全小寫字詞，例如按鈕會實作onclick的事件屬性(attribute)，就在這裡面寫上事件處理的程式碼:

```
<button onclick="console.log('hello!');"> Say Hello! </button>
```

JavaScript引擎中會產生一個對應的匿名函式，包含在onclick中的語句。這個方式也是最不建議的方式，它完全不像個用JavaScript語言寫的應用程式，也因爲需要直接寫在HTML中，完全沒有彈性可言。

傳統模型

傳統模型(Traditional model)方式，提供了分離HTML與JavaScript程式碼的語法，它比之前內聯模型的方式好得多了。不過它依然有個大問題，就是它只能在一個元素上使用一個事件，所以也不建議使用。

```
document.getElementById('myButton').onclick = function(){
    console.log('hello!')
}
```

DOM Level 2

這個方式又被稱爲W3C方式(W3C Way)模型，這個方式才能說它是用事件監聽(Event Listen)的方式，使用callback(回調)函式，作爲事件的監聽者(或稱之爲事件處理函式)。

```
const el = document.getElementById('myButton')
```

```
el.addEventListener( 'click', function(){
     console.log('hello!')
}, false)
```

事件監聽的方式可以對一個元件附加多個事件處理函式，而且以標準來說基本有定義三種方法可使用:

- `addEventListener`: 在事件對象上加入事件監聽者
- `removeEventListener`: 從事件對象移除事件監聽者
- `dispatchEvent`: 送出事件給所有有訂閱的監聽者

> 註: 微軟IE瀏覽器在舊版本中使用自己定義的事件處理方法，所以要與舊版本相容時需要特別注意。IE9之後就能使用上述的事件監聽方式。

事件觸發的順序

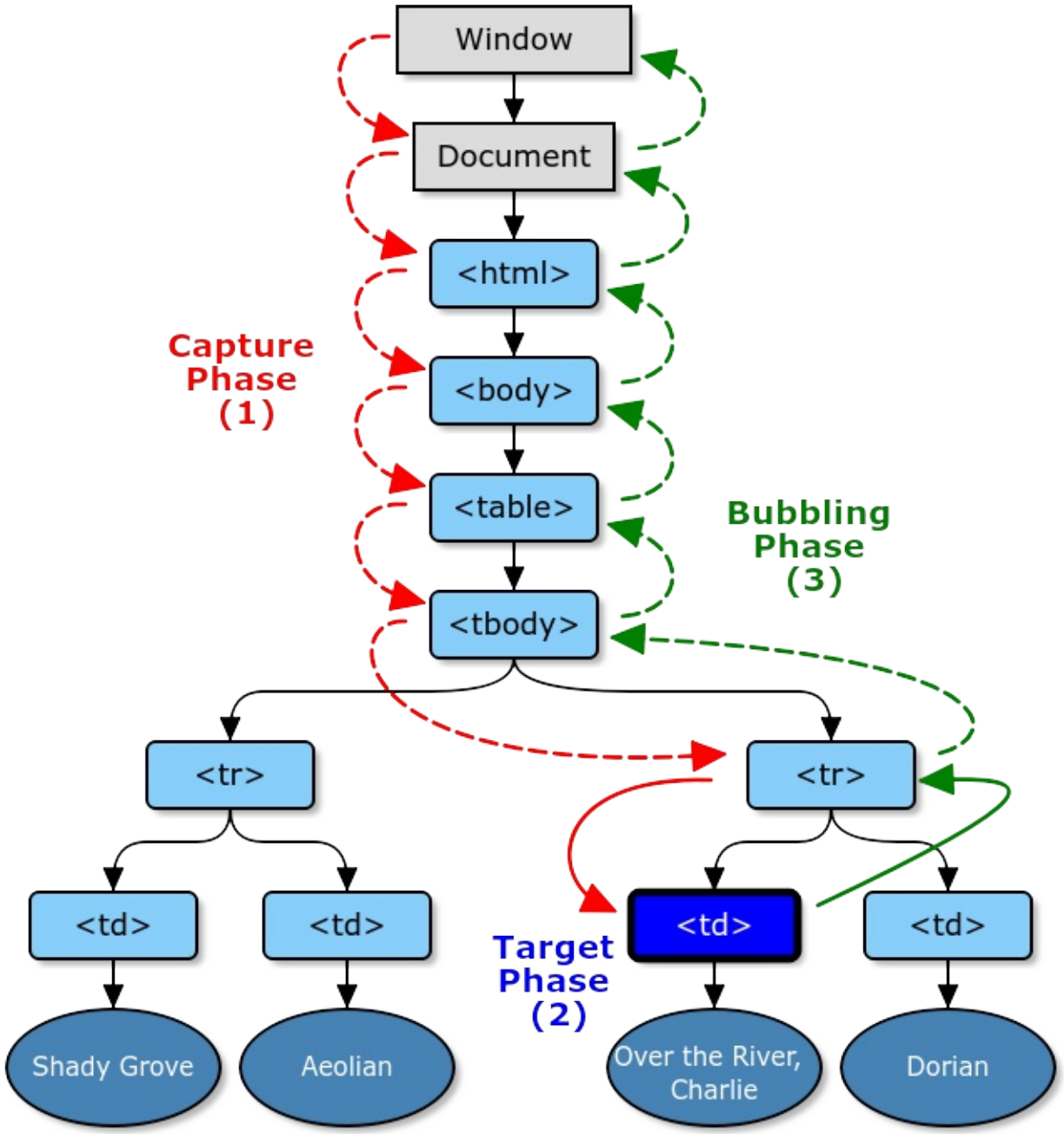
在同一個元素上註冊多個不同的事件監聽者(事件處理函式)，它的在事件觸發時的執行順序是怎麼樣子的？是按照程式碼所寫的順序嗎？

由於W3C的DOM第2代標準中並沒有明確的說明順序這件事，所以當如果有多個事件處理函式在同一元素中註冊時，它們的執行順序將會由瀏覽器來決定，有些舊的瀏覽器並不是按照程式碼中的順序執行。

在W3C的DOM第3代標準也已經明定順序由註冊到事件目標的監聽者順序決定，而且現在瀏覽器各種不同品牌與新版本，都會依照程式碼中的順序爲執行的順序。

事件的冒泡、捕捉

事件的冒泡(往上冒泡)與捕捉(往下捕捉)是兩種事件在DOM中的傳播(propagation)的方式。這是由於DOM元素的樹狀結構，它是有父母-子女關係(parent-child)，這兩種傳播會在事件監聽時，形成一種特別的事件傳播模型，影響事件監聽者在不同父母-子女關係(parent-child)時的執行順序。例如下面的圖解(出自這個網站):



那爲什麼會出現兩種相反的事件傳播方式？這是因爲在第一次瀏覽器大戰時，Netscape採用了事件捕捉(capturing)，而微軟採用了事件冒泡(bubbling)，現行的W3C標準則一併使用了兩者，目前的瀏覽器品牌中都有支援兩者，而微軟從IE9之後也支援兩者。事件捕捉(capturing)與事件冒泡(bubbling)並沒有說哪一種就比較好，這純粹是看程式設計師的需求而定，不過要特別注意的是，有些幾個事件並沒有支援事件的傳播，例如onfocus或onblur。

那麼要如何控制使用哪一種？一般都是使用事件監聽的方法，以傳入參數值作控制，也就是addEventListener方法的最後一個參數`phase`(階段)來決定。`addEventListener`的語法如下:

```
addEventListener( type, handler, phase )
```

`phase`(階段)是一個布林值，如果是`false`就用事件冒泡(bubbling)，如果是`true`就使用事件捕捉(capturing)。預設沒寫的話，就是`false`，也就是預設使用事件冒泡(bubbling)機制。

> 註: 記法有很多種，自行想像發揮力。例如 fb(false = bubbling)。true與capture都有"t"。抓兔子(捕捉=true)，廢砲(false=冒泡)，浴室才會有泡泡(預設使用泡泡)。

事件冒泡(bubbling)的情況時，當最內部的元素被觸發事件時，會先執行自己本身的事件處理函式，然後才會執行上層父母元素的事件處理函式。以下爲範例:

```
const parent = document.getElementById('parent')
const child = document.getElementById('child')

parent.addEventListener('click',
     function(){ console.log('parent clicked') }, false)

child.addEventListener('click',
     function(){ console.log('child clicked') }, false)
```

HTML中的程式碼如下:

```
<div id="parent">
      click me(parent)
    <div id="child">
      click me(child)
    </div>
</div>
```

事件捕捉(capturing)則是倒過來的情況，當最內部的元素被觸發事件時，會先從最外圍的事件處理函式執行，依序到最後才是執行自己本身的處理函式。以下爲範例:

```
const parent = document.getElementById('parent')
const child = document.getElementById('child')

parent.addEventListener('click',
     function(){ console.log('parent clicked') },true)

child.addEventListener('click',
     function(){ console.log('child clicked') }, true)
```

HTML中的程式碼如下:

```
<div id="parent">
      click me(parent)
    <div id="child">
      click me(child)
    </div>
</div>
```

不論事件的傳播方式爲何種，Event物件提供了stopPropagation()方法，可以阻止事件的傳播，這可以在某些應用情況中使用。不過這個stopPropagation方法在事件冒泡(bubbling)與事件捕捉(capturing)兩種不同情況下使用時要特別小心，以免連元素自己的事件處理函式都被擋住。例如像下面的例子:

```
const taiwan =  document.getElementById('taiwan')
const taipei = document.getElementById('taipei')
const me = document.getElementById('me')

taiwan.addEventListener('click',
     function(){ console.log('taiwan clicked') },false)
```

```
taipei.addEventListener('click',
     function(){ console.log('taipei clicked') },false)

me.addEventListener('click',
     function(){ console.log('me clicked') }, false)
```

index.html上的HTML程式碼如下，爲了明顯標出每個區域，加了顏色與大小的樣式:

```
<div id="taiwan" style="border:1px solid green; height:300px; width:
  Click Taiwan
  <div id="taipei" style="border:1px solid blue; height:200px; width
    Click Taipei
    <div id="me" style="border:1px solid red; height:100px; width:10
  </div>
</div>
```

taiwan是最外圍的DOM元素，然後裡面含有taipei層，最裡面是me這一層。以下是點按me層元素的結果:

```
//事件冒泡(bubbling)，全部false參數值的結果是
me -> taipei -> taiwan

//事件捕捉(capturing)，全部爲true參數值的結果是
taiwan -> taipei -> me
```

假使在第2層中，也就是taipei層加上stopPropagation方法，阻止事件傳播，如以下的程式碼:

```
taipei.addEventListener('click',
     function(e){
      console.log('taipei clicked')
      e.stopPropagation()
    },false)
```

你可以再比對一次點按me層元素的結果如下:

```
//事件冒泡(bubbling)，全部false參數值的結果是
me -> taipei (stop!)

//事件捕捉(capturing)，全部爲true參數值的結果是
taiwan -> taipei (stop!)
```

也就是說只要經過有阻止傳播的層，就會不再有事件處理函式被觸發，不論這個事件處理函式是不是已經被註冊爲該層的事件處理函式，這種特性需要特別小心使用。

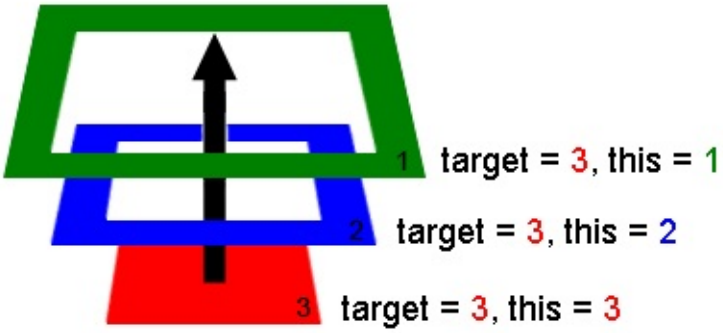
另外，混用事件冒泡(bubbling)與事件捕捉(capturing)兩種在程式碼中，絕對是個不智之舉，在簡單的DOM結構時，可能可以推測出來事件處理函式的順序，但在眞實情況，DOM的結構的複雜程度是超過你所能想像的，千萬不要這麼作。如果你不確定該使用哪一種方式，使用預設的事件冒泡(bubbling)方式，也就是使用false值就行了。

> 註: 事件捕捉(capturing)也有人稱它爲事件滴流(trickling)，滴流是向下流動的意思。總之就是向上冒泡的反義詞。

this與event.target、event.currentTarget

在W3C標準的定義中，this會相等於event.currentTarget，也就是說this永遠會指向目前的事件目標對象，就像地震的傳播一樣，地震中心點先震完了，開始傳播到別的地區，event.currentTarget會跟著改變。事件監聽者(事件處理函式)是一個callback(回調)函式，但這個行爲與一般的callback(回調)函式不同，一般的callback(回調)函式的this值會總是指向全域的window物件。

但不管是事件冒泡(bubbling)或事件捕捉(capturing)，event.target永遠指向觸發事件的那個元素，也就是地震的發生源。以下有一張事件冒泡(bubbling)的解說圖片，你可以看一下event.target與this(也就是event.currentTarget)的比較。(來自這個網站):



自訂事件

自訂事件是由開發者自己建立所需要的事件，再來是附加到某個DOM元素上監聽，最後也是要由開發者自行觸發。會這樣作的原因是，DOM元素上的事件早就被定義固定了，按鈕有按扭可用的事件，表單中的元素有自己的事件。那如果在開發者自己作的應用程式中，想要依照程式中的某個執行功能，定義自訂的事件要怎麼作？例如會員登入時想要用個會員登入事件，聊天室程式常見的好友上線時的好友上線事件，這就是自訂事件的功用了。

首先我們會先使用CustomEvent這個介面(物件)來定義自訂的事件物件，它只需要設定一些屬性，這些屬性與Event中無異，唯一的差異是在於它可以使用detail屬性，來額外新加入自訂事件的屬性值，下面是範例:

```
const myEvent = new CustomEvent(
  'userLogin',
  {
    detail: {
              message: 'Hello World!',
              time: new Date(),
          },
          bubbles: true,
          cancelable: true
  }
)
```

'userLogin'是這個自訂事件的名稱，detail物件中的屬性是額外的屬性。bubbles與cancelable是可以定義也可以不需要定義的屬性，沒定義會使用預設值false，CustomEvent介面(物件)的屬性是繼承自Event(介面)物件而來，所以會有這兩個屬性。bubbles值代表可否使用冒泡機制，cancelable則是代表可否使用stopPropagation()方法。

接下來在你想要監聽這個事件的元素上，加入這個事件與對應的事件處理函式:

```
const myButton = document.getElementById('myButton')

myButton.addEventListener('userLogin', function(e) {
    console.log('Event is: ', e)
    console.log('Custom data is: ', e.detail)
})
```

要觸發這個自訂事件，你只能手動地在程式碼中觸發，使用下面的程式碼:

```
myButton.dispatchEvent(myEvent)
```

自訂事件在IE系統的瀏覽器沒辦法直接使用，有相容性的問題，在其他的瀏覽器上則不會有。IE9之後的瀏覽器可以使用這裡的Polyfill(填充)程式碼或是外部函式庫(如jQuery)來擴充這個功能。

參考資料

- UI Events(W3C)
- Document Object Model (DOM) Level 2 Events Specification(W3C)
- HTML5 - 6.1.5 Events(W3C)
- Event developer guide(MDN)
- Advanced event registration models
- Event order
- Bubbling and capturing
- A crash course in how DOM events work
- The Order of Multiple Event Listeners
- Unable to understand useCapture attribute in addEventListener
- How to Create Custom Events in JavaScript

箭頭函式

箭頭函式

箭頭函式(Arrow　Functions)是ES6標準中，最受歡迎的一種新語法。它會受歡迎的原因是好處多多，而且沒有什麼副作用或壞處，只要注意在某些情況下不要使用過頭就行了。有什麼好處呢？大致上有以下幾點:

- 語法簡單。少打很多字元。
- 可以讓程式碼的可閱讀性提高。
- 某些情況下可以綁定this值。

語法

箭頭函式的語法如下，出自箭頭函數(MDN)：

```
([param] [, param]) => {
   statements
}

param => expression
```

簡單的說明如下:

- 符號是肥箭頭(=>) (註: "->"是瘦箭頭)
- 基本特性是"函式表達式(FE)的簡短寫法"

一個簡單的範例是：

```
const func = (x) => x + 1
```

相當於

```
const func = function (x) { return x + 1 }
```

所以你可以少打很多英文字元與一些標點符號之類的，所有的函式會變成匿名的函式。基本上的符號使用如下說明：

- 花括號({})是有意義的，如果函式沒回傳東西就要花括號。例如 ( )=>{}
- 只有單一個傳入參數時，括號(())可以不用，例如 x=>x*x

最容易搞混的是下面這個例子，有花括號({})與沒有是兩碼子事:

```
const funcA = x => x + 1
const funcB = x => { x + 1 }

funcA(1) //2
funcB(1) //undefined
```

當沒用花括號({})時，代表會自動加return，也只能在一行的語句的時候使用。使用花括號({})則是可以加入多行的語句，不過return不會自動加，有需要你要自己加上，沒加這個函

式最後等於`return undefined`(註: 這是JavaScript語言中函式的設計)。

綁定`this`值

箭頭函式可以取代原有使用`var self = this`或`.bind(this)`的情況，它可以在詞彙上綁定`this`變數。這在特性篇"this"章的內容中有提到。但有時候情況會比較特殊，要視情況而定，而不是每種情況都一定可以用箭頭函式來取代。

原本的範例:

```
const obj = {a:1}

function func(){
  const that = this

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

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

可以改用箭頭函式:

```
const obj = {a:1}

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

func.call(obj)
```

用`bind`方法的來回傳一個部份函式的語法，也可以用箭頭函式來取代，範例出自Arrow functions vs. bind():

```
function add(x, y) {
       return x + y
   }

const plus1 = add.bind(undefined, 1)
```

箭頭函式的寫法如下:

```
const plus1 = y => add(1, y)
```

不可使用箭頭函式的情況

以下這幾個範例都是與`this`值有關，所以如果你的箭頭函式裡有用到`this`值要特別小心。以下的範例都只能用一般的函式定義方式，"不可"使用箭頭函式。

使用字面文字定義物件時，其中的方法

箭頭函式會以定義當下的this值爲this值，也就是window物件(或是在嚴格模式的undefined)，所以是存取不到物件中的this.array值的。

```
const calculate = {
  array: [1, 2, 3],
  sum: () => {
    return this.array.reduce((result, item) => result + item)
  }
}

//TypeError: Cannot read property 'array' of undefined
calculate.sum()
```

物件的**prototype**屬性中定義方法時

這種情況也是像上面的類似，箭頭函式的this值，也就是window物件(或是在嚴格模式的undefined)。

```
function MyCat(name) {
  this.catName = name
}

MyCat.prototype.sayCatName = () => {
  return this.catName
}

cat = new MyCat('Mew')
// ReferenceError: cat is not defined
cat.sayCatName()
```

DOM事件處理的監聽者**(**事件處理函式**)**

箭頭函式的this值，也就是window物件(或是在嚴格模式的undefined)。這裡的this值如果用一般函式定義的寫法，應該就是DOM元素本身，才是正確的值。

```
const button = document.getElementById('myButton')

button.addEventListener('click', () => {
  this.innerHTML = 'Clicked button'
})
//TypeError: Cannot set property 'innerHTML' of undefined
```

建構函式

這會直接在用new運算符時拋出例外，根本不能用。

```
const Message = (text) => {
  this.text = text;
}
// Throws "TypeError: Message is not a constructor"
const helloMessage = new Message('Hello World!');
```

其他限制或陷阱

- 函式物件中的`call`、`apply`、`bind`三個方法，無法"覆蓋"箭頭函式中的`this`值。
- 箭頭函式無法用於建構式(constructor)，使用`new`會產生錯誤。(上面有範例)
- 箭頭函式沒有一般函式有的隱藏arguments物件。
- 箭頭函式不能當作generators使用，使用`yield`會產生錯誤。
- 箭頭函式用於解決一般的`this`問題是可以，但並不適用於全部的情況，尤其是在像jQuery、underscore之類有callback(回調)之類的API時，有可能不是如預期般的結果。

參考資料

- Arrow Functions
- TypeScript Deep Dive
- ES6 Arrow Functions: The New Fat & Concise Syntax in JavaScript
- When 'not' to use arrow functions

Promise 承諾

Promise 承諾

本章節已獨立爲一本小電子書，請前往從Promise開始的JavaScript異步生活觀看其內容。

展開運算符與其餘運算符

展開運算符與其餘運算符

展開運算符(Spread Operator)與其餘運算符(Rest Operator)是ES6中的其中兩種新特性，雖然這兩種特性的符號是一模一樣的，都是(...)三個點，但使用的情況與意義不同。我們常常在文字敘述或聊天時，這個(...)常用來代表了"無言"、"無窮的想像"或"還有其他更多的"的意思。

簡單摘要一下這個語法的內容：

- 符號都是三個點(...)
- 都與陣列有關
- 一個是展開陣列中的值，一個是集合其餘的值成爲陣列

註: 三個點符號(...)，比較正式的英文字詞是Ellipsis，翻成中文是"省略"符號的意思，不過它有各種形式(全形或半形)，也有超過三個點的情況，所以一般要說得明白只有三個點的情況，會用"three dots"與"dot-dot-dot"反而更爲明確。

註: 目前看到的名詞上並沒有統一的但都是指這種語法。在ES6標準上的用語是SpreadElement與rest parameter，並沒有集合成爲一個章節，而是內容散落在各章節中。在MDN上用Spread operator與Spread syntax的名詞(標題是syntax，網址列上是operator)，以及Rest operator與Rest parameters(標題是parameters，但連結是operator)。

註: 用於物件上的類似語法並不是ES6中的特性，它是正在制定中的新語法標準，不過在React與Redux中很常見，能透過babel編譯，這稱爲Object Rest/Spread Properties

展開運算符(Spread Operator)

展開運算符是把一個陣列展開成個別的值的速寫語法，它只會在陣列字面定義與函式呼叫時使用

展開運算符(Spread Operator)是把一個陣列展開(expand)成個別值，這個運算符後面必定接著一個陣列。最常見的是用來組合(連接)陣列，對應的陣列方法是`concat`，以下是一個簡單的範例：

```
const params = [ "hello", true, 7 ]
const other = [ 1, 2, ...params ] // [ 1, 2, "hello", true, 7 ]
```

展開運算符可以作陣列的淺拷貝，當然陣列的淺拷貝有很多種方式，這是新的一種語法，也是目前語法上最簡單的一種:

```
const arr = [1,2,3]
const arr2 = [...arr]

arr2.push(4) //不會影響到arr
```

註: 淺拷貝(shallow-copy)對於陣列中的陣列值(多維陣列)，或是有複雜的物件值情況時，是只會拷貝參照值而已。

註: 上述的展開運算符在陣列字面中使用時，並沒有限制位置，或是在個數。像`const arr = [...a, 1, ...b]`這樣的語法都是可以的。

你也可以用來把某個陣列展開，然後傳入函式作爲傳入參數值，例如下面這個一個加總函式的範例：

```
function sum(a, b, c) {
  return a + b + c
}
const args = [1, 2, 3]
sum(…args) // 6
```

對照ES5中的相容語法，則是用apply函式，它的第二個參數也是使用陣列，以下是用ES5語法與上面相同結果的範例程式：

```
function sum(a, b, c) {
  return a + b + c;
}

var args = [1, 2, 3];
sum.apply(undefined, args) ;// 6
```

展開運算符還有一個特別的功能，就是把可迭代(iterable)或與陣列相似(Array-like)的物件轉變爲陣列，在JavaScript語言中內建的可迭代(iterable)物件有String、Array、TypedArray、Map與Set物件，而與陣列相似(Array-like)的物件指的是函式中的隱藏物件"arguments"。下面的範例是轉變字串爲字元陣列:

```
const aString = "foo"
const chars = [ ...aString ] // [ "f", "o", "o" ]
```

下面的範例是把函式中的隱藏僞物件"arguments"轉成眞正的陣列物件:

```
function aFunc(x){
  console.log(arguments)
  console.log(Array.isArray(arguments))

  //轉爲眞正的陣列
  const arr = [...arguments]
  console.log(arr)
  console.log(Array.isArray(arr))
}

aFunc(1)
```

其餘運算符(Rest Operator)

其餘運算符是收集其餘的(剩餘的)這些值，轉變成一個陣列。它會用在函式定義時的傳入參數識別名定義(其餘參數, Rest parameters)，以及解構賦值時

會用其餘運算符(Rest Operator)的主要意義是因爲它會用在兩個地方，一個是比較常提及的在函式定義中的傳入參數定義中，稱之爲其餘參數(Rest parameters)。另一種情況是用在解構賦值時。

其餘參數(Rest parameters)

就像在電影葉問中的台詞："我要打十個"，其餘參數可以讓你一次打剩下的全部，不過葉問會變成一個陣列。

既然是一個參數的語法，當然就是用在函式的傳入參數定義。其餘參數代表是將"不確定的傳入參數值們"在函式中轉變成爲一個陣列來進行運算。例如下面這個加總的範例：

```
function sum(…numbers) {
  const result = 0

  numbers.forEach(function (number) {
    result += number
  })

  return result
}

sum(1) // 1
sum(1, 2, 3, 4, 5) // 15
```

特別注意: 其餘參數在傳入參數定義中，必定是位於最後一位，並且在參數中只能有一個其餘參數。

其餘參數的值在沒有傳入實際值時，會變爲一個空陣列，而不是undefined，以下的範例可以看到這個結果:

```
function aFunc(x, ...y){
  console.log('x =', x,  ', y = ' , y)
}

aFunc(1,2,3) //x = 1, y = [2, 3]
aFunc() //x = undefined, y = []
```

其餘參數的設計有一個很明確的用途，就是要取代函式中那個隱藏"僞陣列"物件arguments，arguments雖然會包含了所有的函式傳入參數，但它是個類似陣列的物件卻沒有大部份陣列方法，它不太像是個隱藏的密技，比較像是隱藏的陷阱，很容易造成誤解或混亂，完全不建議你使用arguments這個東西。

其餘參數的值是一個眞正的陣列，而且它需要在傳入參數宣告才能使用，至少在程式碼閱讀性上勝出太多了。

解構賦值(destructuring)時

解構賦值在另一獨立章節會講得更詳細，這裡只是要說明其餘運算符的另一個使用情況。解構賦值也是一個ES6中的新特性。

解構賦值是用在"陣列指定陣列"或是"物件指定物件"的情況下，這個時候會根據陣列原本的結構，以類似"鏡子"對映樣式(pattern)來進行賦值。聽起來很玄但用起來很簡單，這是一種爲了讓陣列與物件指定值時更方便所設計的一種語法。例如以下的範例:

```
const [x, y, z] = [1, 2, 3]

console.log(x) //1
```

像這個例子就是最簡單的陣列解構賦值的範例，x當然會被指定爲1，y與z你應該用腳底板也想得到是被指定了什麼值。

當使用其餘運算符之後，就可以用像其餘參數的類似概念來進行解構賦值，例如以下的範例:

```
const [x, ...y] = [1, 2, 3]

console.log(x) //1
console.log(y) //[2,3]
```

當右邊的值與左邊數量不相等時，"鏡子對映的樣式"就會有些沒對到，用了其餘運算符的那個識別名稱，就會變成空陣列。就會像下面這個例子一樣:

```
const [x, y, ...z] = [1]

console.log(x) //1
console.log(y) //undefined
console.log(z) //[]
```

在函式傳入參數中作解構賦值，這個例子會的確也是一種解構賦值的語法，而且加了上一節的函式中的其餘參數的用法。例子出自MDN的這裡:

```
function f(...[a, b, c]) {
  return a + b + c;
}

f(1)          // NaN (b and c are undefined)
f(1, 2, 3)    // 6
f(1, 2, 3, 4) // 6 (the fourth parameter is not destructured)
```

你可以回頭再看一下"其餘參數"的使用情況，是不是與解構賦值時很相似。

特別注意: 在使用解構賦值時一樣只能用一個其餘運算符，位置也只能放在最後一個。

ES7+標準的其餘屬性(Rest Properties)與展開屬性(Spread Properties)

上面都只有談到與陣列搭配使用，但你可能會看到在物件上也會使用類似語法與符號(...)，尤其是在React與Redux中。這些都是還在制定中的ES7之後草案標準，稱爲其餘屬性(Rest Properties)與展開屬性(Spread Properties)。例如下面這樣的範例，來自這裡：

```
// Rest Properties
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 }
console.log(x) // 1
console.log(y) // 2
console.log(z) // { a: 3, b: 4 }

// Spread Properties
let n = { x, y, ...z }
console.log(n) // { x: 1, y: 2, a: 3, b: 4 }
```

有些新式的框架或函式庫中已經開始使用了，babel轉換工具可以支援轉換這些語法，但使用時要注意要額外加裝`babel-plugin-transform-object-rest-spread`外掛。

撰寫風格建議

- 不要使用函式中的arguments，總是使用其餘參數語法來取代它。(Airbnb 7.6, Google 5.5.5.2, eslint: prefer-rest-params).

- 不要在展開運算符與其餘運算符後面有空格，也就是與後面的識別名稱(傳入參數名稱、陣列名稱)之間要緊接著。(Google 5.2.5/5.5.5.2, eslint: rest-spread-spacing)
- 用展開運算符的語法來作拷貝陣列。(Airbnb 4.3)
- 用展開運算符的語法來取代函式中的`apply`的語法，作不定個數傳入參數的函式呼叫。(Airbnb 7.14, eslint: prefer-spread)
- 用展開運算符的語法來取代`slice`與`concat`方法的語法。(Google 5.2.5)
- 優先使用物件展開運算符(object spread operator)的語法取代Object.assign，來作物件的淺拷貝。(Airbnb 3.8) (ES7+標準)

結論

展開運算符比較容易理解，它是把已有(看得到)的陣列值"展開"爲一個一個單獨的值。其餘運算符都是用在函式定義或指定值時，它是要收集其餘的(剩餘的)值，形成一個陣列再來進行運算，你可能還對展開運算符與其餘運算符的分別還有點混亂，因爲符號都是相同的三個點符號(...)，只是它們在不同的使用情況下功用是不太相同的，所以要區分它們要從使用情況來區分:

- 展開運算符: 用在陣列的字面文字定義裡面(例如`[1, ...b]`)，或是函式呼叫時(例如`func(...args)`)
- 其餘運算符: 用在函式的定義，裡面的傳入參數名稱定義時(例如`function func(x, ...y)`)。或是在解構賦值時(例如`const [x, ...y] = [1,2,3]`)

參考資源

- Spread syntax(MDN)
- Rest parameters
- Exploring ES6 - 10.7.2 Rest operator
- Exploring ES6 - 11.8 The spread operator
- ES6—default + rest + spread
- ES6 Spread and Butter in Depth

AJAX與Fetch API

AJAX與Fetch API

AJAX與XMLHttpRequest

AJAX這個技術名詞的出現是在十年前(2005)，其中內容包含XML、JavaScript中的XMLHttpRequest物件、HTML與CSS等等技術的整合應用方式，這個名詞並非專指某項特定技術或是軟體，Google在時所推出的Gmail服務與地圖服務，獲得很大的成功，當時這個技術名詞以此作爲主要的案例說明。實際上這個技術的實現是在更早之前(2000年之前)，一開始是微軟公司實作了一個Outlook與郵件伺服器溝通的介面，後來把它整合到IE5瀏覽器上。在2006年XMLHttpRequest正式被列入W3C標準中，現在已被所有的瀏覽器品牌與新版本所支援。

所謂的AJAX技術在JavaScript中，即是以XMLHttpRequest物件(簡稱爲XHR)爲主要核心的實作。正如它的名稱，它是用於客户端對伺服器端送出httpRequest(要求)的物件，使用的資料格式是XML格式(但後來JSON格式才是最爲流行的資料格式)。流程即是建立一個XMLHttpRequest(XHR)物件，打開網址然後送出要求，成功時最後由回調函式處理伺服器傳回的Response(回應)。整體的流程是很簡單的，但經過這麼長久的使用時間(11年)，它在使用上產生不少令人頭痛的問題，例如:

- API設計得過於高階(簡單)，所有的輸出與輸入、狀態，都只能與這個XHR物件溝通取得，進程狀態是用事件來追蹤。
- XHR是使用以事件爲基礎(event-based)的模組來進行異步程式設計。
- 跨網站的HTTP要求(cross-site HTTP request)與CORS(Cross-Origin Resource Sharing)不易實作。
- 對非文字類型的資料處理上不易實作。
- 除錯不易。

XHR在使用上都是像下面的範例程式碼這樣，其實你可以把它視作一種事件處理的結構，大小事都是依靠XHR物件來作，語法中並沒有把每件事情分隔得很清楚，而比較像是擠在一團:

```
function reqListener() {
  const data = JSON.parse(this.responseText);
  console.log(data)
}

function reqError(err) {
  console.log('Fetch Error :-S', err)
}

const oReq = new XMLHttpRequest();
oReq.onload = reqListener
oReq.onerror = reqError
oReq.open('get', './sample.json', true)
oReq.send()
```

在今天瀏覽器功能相當強大，以及網站應用功能複雜的時代，XHR早就已經不敷使用，它在架構上明顯的有太多的問題，尤其在很多功能的應用情況，程式碼會顯得複雜且不易維護。除非你是有一定要使用原生JavaScript的強迫症，要不然現在要作AJAX功能時，程式設計師並不會使用原生XHR物件來撰寫，大部份時候會使用外部函式庫。因爲一個AJAX的程式，並不

是單純到只有對XHR的要求與回應這麼簡單，例如你可能會對伺服器要求一份資料，當成功得到資料後，後面還有需要進一步的資料處理流程，這樣就會涉及到異步程式的執行結構，原生XHR並沒有提供可用的方式，它只是單純的作與伺服器互動那件事而已。

XHR Level 2(第2級)

XHR並不是沒有在努力進步，在約5年前已經有制定XHR的第2級新標準，但它仍然與原有XHR向下相容，所以整體的模型架構並沒有重大的改變，只是針對問題加以補強或是擴充。目前XHR第2級在9成以上的瀏覽器品牌新版本都已經支援全部的功能，除了IE系列要版本10之後才支援，以及Opera Mini瀏覽器完全不支援，還有一小部份功能在不同瀏覽器上實作細節會有所不同。XHR第2級(5年前)相較於原有的XHR(11年前)多加了以下的功能，這也是現在我們已經可以使用到的XHR的新特性:

- 指定回應格式
- 上傳文件與blob格式檔案
- 使用FormData傳送表單
- 跨來源資源共享(CORS)
- 監視傳輸的進程

不過，XHR第2級的新標準並沒有太引人注目的新功能，它比較像是解決長期以來的一些嚴重問題的補強版本。而且，XHR在原本上的設計就是這樣，常被批評的是它的語法結構不論在使用與設定都相當的零亂。補強或擴充都還是跳脫不了基本的結構，現今是HTM5、CSS3與ES6的時代，有許多新的技術正在蓬勃發展，說句實在話，就是XHR技術已經舊掉了，當時的設計不符合現在時代需求了，這也無關對或錯。

jQuery

外部函式庫例如jQuery很早就看到XHR物件中在使用的問題，使用像jQuery的函式庫來撰寫AJAX相關功能，不光是在解決不同瀏覽器中的不相容問題，或是提供簡化語法這麼簡單而已。jQuery它擴充了原有的XHR物件爲jqXHR物件，並加入類似於Promise的介面與Deferred Object(延遲物件)的設計。

爲何要加入類似Promise的介面？可以看看它的說明中，是爲了什麼而加入的？

> 這些方法可以使用一個以上的函式傳入參數，當`$.ajax()`的要求結束時呼叫它們。這可以讓你在單一個(request)要求中指定多個callbacks(回調)，甚至可以在要求完成後指定多個callbacks(回調)。 ~譯自jQuery官網The jqXHR Object

原生的XHR根本就沒有這種結構，Promise的結構基本上除了是一種異步程式設計的架構，它也可以包含錯誤處理的流程。簡單地來說，jQuery的目標並不是只是簡化語法或瀏覽器相容性而已，它的目標是要"取代以原生**XHR**物件的**AJAX**語法結構"，雖然本質上它仍然是以XHR物件爲基礎。

jQuery作得相當成功，十分受到程式設計師們的歡迎，它的語法結構相當清楚，可閱讀性與設定彈性相當高，常讓人忘了原來的XHR是有多不好使用。在Promise還沒那麼流行的前些年，裡面就已經有類似概念的設計。加上現在的新版本(3.0)已經支援正式的Promise標準，說實在沒什麼理由不去使用它。以下是jQuery中ajax方法的範例:

```
// 使用 $.ajax() 方法
$.ajax({

    // 進行要求的網址(URL)
    url: './sample.json',
```

```
    // 要送出的資料 (會被自動轉成查詢字串)
    data: {
        id: 'a001'
    },

    // 要使用的要求method(方法)，POST 或 GET
    type: 'GET',

    // 資料的類型
    dataType : 'json',
})
  // 要求成功時要執行的程式碼
  // 回應會被傳遞到回調函式的參數
  .done(function( json ) {
     $( '<h1>' ).text( json.title ).appendTo( 'body' );
     $( '<div class=\'content\'>').html( json.html ).appendTo( 'body
  })
  // 要求失敗時要執行的程式碼
  // 狀態碼會被傳遞到回調函式的參數
  .fail(function( xhr, status, errorThrown ) {
    console.log( '出現錯誤，無法完成!' )
    console.log( 'Error: ' + errorThrown )
    console.log( 'Status: ' + status )
    console.dir( xhr )
  })
  // 不論成功或失敗都會執行的回調函式
  .always(function( xhr, status ) {
    console.log( '要求已完成!' )
  })
```

把原生的XHR用Promise包裹住，的確是一個好作法，有很多其他的函式庫也是使用類似的作法，例如axios與SuperAgent，相較於jQuery的多功能，這些是專門只使用於AJAX的函式庫，另外這些函式庫也可以用在伺服器端，它們也是有一定的使用族群。

Fetch

Fetch是近年來號稱要取代XHR的新技術標準，它是一個HTML5的API，並非來自ECMAScript標準。在瀏覽器支援性的部份，首先由Mozilla與Google公司在2015年3月發佈Fetch實作消息，目前也只有Firefox與Chrome、Opera瀏覽器在新版本中原生支援，微軟的新瀏覽器Edge也在最近宣佈支援(新聞連結)(應該是Edge 14)，其他瀏覽器目前可以使用polyfill來作填充，提供暫時解決相容性的方案。另外，Fetch同樣要使用ES6 Promise的新特性，這代表如果瀏覽器沒有Promise特性，一樣也需要使用es6-promise來作填充。

Fetch並不是一個單純的XHR擴充加強版或改進版本，它是一個用不同角度思考的設計，雖然是可以作類似的事情。此外，Fetch還是基於Promise語法結構的，而且它的設計足夠低階，這表示它可以依照實際需求進行更多彈性設定。相對於XHR的功能來說，Fetch已經有足夠的相對功能來取代它，但Fetch並不僅於此，它還提供更多有效率與更多擴充性的作法。

> 註: 英文中 fetch/費曲/ 有"獲取"、"取回"的意思。它與get、bring單詞是近義詞。

Fetch基本語法

`fetch()`方法是一個位於全域window物件的方法，它會被用來執行送出Request(要求)的工

作，如果成功得到回應的話，它會回傳一個帶有Response(回應)物件的已實現Promise物件。fetch()的語法結構完全是Promise的語法，十分清楚容易閱讀，也很類似於jQuery的語法:

```
fetch('http://abc.com/', {method: 'get'})
.then(function(response) {
    //處理 response
}).catch(function(err) {
    // Error :(
})
```

但要注意的是fetch在只要在伺服器有回應的情況下，都會回傳已實現的Promise物件狀態(只要不是網路連線問題，或是伺服器失連等等)，在這其中也會包含狀態碼爲錯誤碼(404, 500...)的情況，所以在使用時你還需要加一下檢查:

```
fetch(request).then(response => {
  //ok 代表狀態碼在範圍 200-299
  if (!response.ok) throw new Error(response.statusText)
  return response.json()
}).catch(function(err) {
    // Error :(
})
```

或是先用另一個處理狀態碼的函式，使用Promise.resolve與Promise.reject將回應的情況包裝爲回傳不同狀態的Promise物件，然後再下個then方法再處理:

```
function processStatus(response) {
    // 狀態 "0" 是處理本地檔案 (例如Cordova/Phonegap等等)
    if (response.status === 200 || response.status === 0) {
        return Promise.resolve(response)
    } else {
        return Promise.reject(new Error(response.statusText))
    }
}

fetch(request)
    .then(processStatus)
    .then()
    .catch()
```

Fetch相關介面說明

fetch的核心由GlobalFetch、Request、Response與Headers四個介面(物件)與一個Body(Mixin混合)。概略的內容說明如下:

- GlobalFetch: 提供全域的fetch方法
- Request: 要求，其中包含method、url、headers、context、body等等屬性與clone方法
- Response: 回應，其中包含headers、ok、status、statusText、type、body等等屬性與clone方法
- Headers: 執行Request與Response中所包含的headers的各種動作，例如取回、增加、移除、檢查等等。設計這個介面的原因有一部份是爲了安全性。
- Body: 同時在Request與Response中均有實作，裡面有包含主體內容的資料，是一種

ReadableStream(可讀取串流)的物件

註: Mixin(混合)樣式是一種將多個物件(或類別)中會共同使用(分享)的方法或屬性另外用一個物件或介面整合包裝起來，然後讓其他的物件(或類別)來使用其中的方法或屬性的設計樣式。Mixins(混合)的主要目的是要讓程式碼功能可以達到重覆使用，但並不是透過類別繼承的方式。

與XHR有很大的明顯不同，每個XHR物件都是一個獨立的物件，麻煩的是每次作不同的Request(要求)或要處理不同的Response(回應)時，就得再重新實體化一個新的XHR物件，然後再設定一次。而fetch中則是可以明確地設定不同的Request(要求)或Response(回應)物件，提供了更多細部設定的彈性，而且這些設定過的物件都可以重覆再使用。Request(要求)物件可以直接作爲fetch方法的傳入參數，例如下面的這個範例:

```
const req = new Request(URL, {method: 'GET', cache: 'reload'})

fetch(req).then(function(response) {
  //處理 response
}).catch(function(err) {
    // Error :(
})
```

另一個很棒的功能是你可以用原有的Request(要求)物件，當作其他要新增的Request(要求)物件的基本樣版，像下面範例中的新的`postReq`即是把原有的`req`物件的method改爲'POST'而已，這可以很容易重覆使用原先設定好的Request(要求)物件。

```
const postReq = new Request(req, {method: 'POST'})
```

以下摘要Request(要求)物件中可以包含的屬性值，可以看到設定值相當多，可以依使用情況設定到很細:

- method: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`。
- url: 要求的網址。
- headers: 與要求相關的Headers物件。
- referrer - `no-referrer`, `client`或一個網址。預設爲`client`。
- mode - `cors`, `no-cors`, `same-origin`, `navigate`。預設爲`cors`。Chrome(v47~)目前的預設值是`same-origin`。
- credentials - `omit`, `same-origin`, `include`。預設爲`omit`。Chrome(v47~)目前的預設值是`include`。
- redirect - `follow`, `error`, `manual`。Chrome(v47~)目前的預設值是。`manual`。
- integrity - Subresource Integrity(子資源完整性, SRI)的值
- cache - `default`, `no-store`, `reload`, `no-cache`, 或 `force-cache`
- body: 要加到要求中的內容。注意，method爲`GET`或`HEAD`時不使用這個值。

註: 由於不能在GET時使用body屬性，如果你需要在GET時用到query字串，解決方案請參考以下的相關問答集: 問答或問答

Request(要求)物件中可以包含`headers`屬性，它是一個以`Headers()`建構式進行實體化的物件，實體化後可以再使用其中的方法進行設定。例如以下的範例:

```
const httpHeaders = { 'Content-Type' : 'image/jpeg', 'Accept-Charset
const myHeaders = new Headers(httpHeaders)

const req = new Request(URL, {headers: myHeaders})
```

```
const httpHeaders = new Headers()
httpHeaders.append('Accept', 'application/json')

const req = new Request(URL, {headers: httpHeaders})
```

註: Headers()建構式的傳入參數可以是其他的Headers物件，或是內含符合HTTP headers的位元組字串的物件。

註: Headers物件中還有一個很特殊的屬性guard，它與安全性有關，請參考Basic_concepts#Guard

當然fetch方法也可以不需要一定得要傳入Request(要求)實體物件，它可以直接使用相同結構的物件字面當作傳入參數，例如以下的範例:

```
fetch('./sample.json', {
    method: 'GET',
    mode: 'cors',
    redirect: 'follow',
    headers: new Headers({
        'Content-Type': 'text/json'
    })
}).then(function(response) {
  //處理 response
})
```

fetch的語法連鎖下一個.then方法，如果成功的話，會得到一個帶有Response(回應)物件值的已實現狀態的Promise物件。雖然在fetch API中也允許你自己建立一個Response(回應)物件實體，不過，Response(回應)物件通常都是從外部資源要求所得到，自訂Response(回應)物件算是會在特殊的情況下才會作的事情。

Response(回應)物件中包含的屬性摘要如下:

- type: basic, cors
- url: 回應網址
- useFinalURL: 布林值，代表這個網址是否爲最後的網址(也可能是重新導向的網址)
- status: 狀態碼 (例如: 200, 404, 500...)
- ok: 代表成功的狀態碼 (狀態碼介於200-299)
- statusText: 狀態碼的文字 (例如: OK)
- headers: 與回應相關的Headers物件

由於Response(回應)實作了Body介面(物件)，可以由Body的方法來取得回應回來的內容，但因爲Body屬性值本身是個ReadableStream的物件，需要再依照不同的內容資料類型使用對應的方法，才能眞正取到資料物件，其中最常使用的是json與text方法:

- arrayBuffer()
- blob()
- formData()
- json()
- text()

這幾個方法在使用過後，會產生帶有相關已解析資料值的已實現Promise物件，通常的作法還需要再下一個then方法中才取得到其中的已解析資料值(物件)。另一種作法是使用巢狀的Promise語法來取得資料，不過巢狀的Promise語法容易造成語法複雜，你可以獨立出來解析JSON資料物件的程式碼到另一個函式中會比較清楚。

註: arrayBuffer請參考ArrayBuffer

註: blob請參考Blob

不過要特別注意的是，Body實體的在Request(要求)與Response(回應)中的設計是"只要讀取過就不能再使用"，Request(要求)或Response(回應)物件其中都有一個`bodyUsed`只能讀不能寫的屬性，它在被讀取過會變成`true`，代表不能再被重覆使用。所以如果要重覆使用Body物件，必須在被讀取前(即`bodyUsed`被設定爲`true`之前)，先呼叫Request(要求)或Response(回應)物件中的`clone`方法，另外拷貝出一個新的實體。

以下分別由幾種不同的資料類型來撰寫的樣式。

純文字/**HTML**格式文字

```
fetch('/next/page')
  .then(function(response) {
    return response.text()
  }).then(function(text) {
      console.log(text)
  }).catch(function(err) {
      // Error :(
  })
```

json格式

json方法會回傳一個帶有包含JSON資料的物件值的Promise已實現物件。

```
fetch('https://davidwalsh.name/demo/arsenal.json').then(function(res
    // 直接轉成JSON格式
    return response.json()
}).then(function(j) {
    // `j`會是一個JavaScript物件
    console.log(j)
}).catch(function(err) {
  // Error :(
})
```

blob(原始資料**raw data)**

```
fetch('https://davidwalsh.name/flowers.jpg')
    .then(function(response) {
      return response.blob();
    })
    .then(function(imageBlob) {
      document.querySelector('img').src = URL.createObjectURL(imageB
    })
```

註: URL也是一個的Web API，在新式的瀏覽器上都有支援。請參考URL.createObjectURL

FormData

FormData是在要求時傳送表單資料時使用。以下爲範例:

```
fetch('https://davidwalsh.name/submit', {
    method: 'post',
    body: new FormData(document.getElementById('comment-form'))
})
```

也可以使用JSON格式的物件資料來作要求:

```
fetch('https://davidwalsh.name/submit-json', {
    method: 'post',
    body: JSON.stringify({
        email: document.getElementById('email').value,
        answer: document.getElementById('answer').value
    })
})
```

註: FormData介面包含在XMLHttpRequest的新標準之中，目前只有Chrome與Firefox支援，請參考FormData

相較於**jQuery.ajax**

jQuery的ajax及相關方法的設計，已經很與fetch的語法結構很類似，不過它的回傳值仍然只是XHR物件的擴充jqXHR物件，需要經過轉換才能成爲ES6的Promise物件。除此之外，有兩個重要的不同之處需要注意，來自window.fetch polyfill:

- `fetch`方法回傳的Promise物件不會在有收到Response(回應)，但是是在HTTP錯誤狀態碼(例如404、500)的時候變成已拒絕(rejected)狀態。也就是說，它只會在網路出現問題或是被阻止進行Request(要求)時，才會變成已拒絕(rejected)狀態，其他都是已實現(fulfilled)。
- `fetch`方法預設是不會傳送任何的認証証書(credentials)例如cookie到伺服器上的，這有可能會造成有管理使用者連線階段(session)的伺服器視爲未經認証的Request(要求)。要加上傳送cookie可以用`fetch(url, {credentials: 'include'})`的語法來設置。

問題點

要求中斷或是設定**timeout**

Fetch目前沒有辦法像XHR可以中斷要求、或是設定timeout屬性，請參考Add timeout option #20的討論。這篇部落格JavaScript Fetch API in action有提供一個用Promise物件包裝的暫時解決方式，部份程式碼如下:

```
var MAX_WAITING_TIME = 5000;// in ms

var timeoutId = setTimeout(function () {
    wrappedFetch.reject(new Error('Load timeout for resource: ' + pa
}, MAX_WAITING_TIME);

return wrappedFetch.promise// getting clear promise from wrapped
    .then(function (response) {
        clearTimeout(timeoutId);
        return response;
    });
```

進程事件(Progress events)

Fetch目前沒辦法觀察進程事件(或傳輸狀態)。在Fetch標準中有提供一個簡單的範例，但並不是太好的作法，程式碼如下:

```
function consume(reader) {
  var total = 0
  return pump()
  function pump() {
    return reader.read().then(({done, value}) => {
      if (done) {
        return
      }
      total += value.byteLength
      log(`received ${value.byteLength} bytes (${total} bytes in tota
      return pump()
    })
  }
}

fetch("/music/pk/altes-kamuffel.flac")
  .then(res => consume(res.body.getReader()))
  .then(() => log("consumed the entire body without keeping the whole
  .catch(e => log("something went wrong: " + e))
```

結論

Fetch在瀏覽器的實作與XHR不同，裡面的功能內容與API也相差很多，它有很多設計是爲了新式的HTML5相關應用所設計的。現在已經有許多大公司的網站開始大量的使用Fetch API來取代XHR的作法，相信這個技術在這二、三年會愈來愈普及，畢竟AJAX技術對網站應用實在太重要，而這是一種扮演關鍵角色的技術。而且值得一提的是，現在在Chrome瀏覽器中在Service Worker技術實作中也提供了Fetch方法，但只限制在Service Worker中使用，這是一個非常新的應用技術，稱爲Progressive Web App(漸進式的網路應用, PWA)。

補充: AJAX與Fetch函式庫比較表

本表主要參考自AJAX/HTTP Library Comparison。

名稱	Github星	瀏覽器支援	Node支援	Promise	原生	最後發佈日
XMLHttpRequest	-	是	-	-	是	-
Node HTTP	-	-	是	-	是	-
fetch	-	部份	-	是	是*	-
window.fetch polyfill	9269	全部	-	是	-	2016/5
node-fetch	894	-	是	是	-	2016/5
isomorphic-fetch	2587	是	是	是	-	2015/11
axios	2035	是	是	是	-	2016/7
SuperAgent	8175	是	是	是	-	2016/7
jQuery	40718	是	-	是*	-	2016/7

註記:

1. 以上統計數據爲2016/7月
2. 瀏覽器是以目前各瀏覽器品牌的最新發佈穩定版本而言。
3. isomorphic-fetch是混合window.fetch polyfill(whatwg-fetch模組)與node-fetch的專案。
4. Promise爲ES6新特性，瀏覽器支援一覽表。需要另外填充時使用es6-promise。
5. jQuery並非專門用於AJAX的函式庫，3.0版本後支援Promise目前標準。

參考資料

Fetch標準

- Fetch Standard @Github
- Fetch Standard
- Fetch API

教學

- Introduction to the Fetch API
- 深入浅出Fetch API
- Basic Fetch Request
- That's so fetch!
- fetch API
- JavaScript Fetch API in action
- Handling Failed HTTP Responses With fetch()

XHR&相關協定

- What is an Internet Protocol?
- XMLHttpRequest

解構賦值

解構賦值

解構賦值(Destructuring Assignment)是一個在ES6的新特性，用於提取(extract)陣列或物件中的資料，這是一種對原本語法在使用上的改進，過去要作這件事可能需要使用迴圈或迭代的語句才行，新語法可以讓程式碼在撰寫時更爲簡短與提高閱讀性。

解構賦值的解說只有一小段英文：

> The destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects using a syntax that mirrors the construction of array and object literals.

這句後面的`mirrors the construction of array and object literals`，代表這個語法的使用方式 - 如同"鏡子"一般，對映出陣列或物件字面的結構。也就是一種樣式(pattern)對映的語法。

解構賦值如果你能抓得住它的基本概念，就可以很容易理解與使用。不過與ES6其他的特性配合時，會顯得複雜比較難理解。

在使用時有以下幾種常見的情況:

- 從陣列解構賦值
- 從物件解構賦值(或是從混用物件或陣列)
- 非物件或非陣列解構賦值
- 解構賦值時給定預設值
- 搭配函式的傳入參數使用

destructuring: 變性、破壞性。使用"解構"是對照`de-`字頭有"脫離"、"去除"的意思。

assignment: 賦值、指派。賦值通常指的是程式中使用等號(=)運算符的語句。

從陣列解構賦值(Array destructuring)

從陣列解構賦值沒太多學問，唯一比較特別的是可以用其餘運算符(Rest Operator)的語法，既然是其餘運算符，最後就會把其餘的對應值集合成一個陣列之中。下面是幾個幾個範例：

```
//基本用法
const [a, b] = [1, 2] //a=1, b=2

//先宣告後指定值，要用let才行
let a, b
[a, b] = [1, 2]

// 略過某些值
const [a, , b] = [1, 2, 3] // a=1, b=3

// 其餘運算
const [a, ...b] = [1, 2, 3] //a=1, b=[2,3]
```

```
// 失敗保護
const [, , , a, b] = [1, 2, 3] // a=undefined, b=undefined

// 交換值
const a = 1, b = 2;
[b, a] = [a, b] //a=2, b=1

// 多維複雜陣列
const [a, [b, [c, d]]] = [1, [2, [[[3, 4], 5], 6]]]

// 字串
const str = "hello";
const [a, b, c, d, e] = str
```

用法就是這麼簡單，用來賦值的等號符號(=)左邊按照你寫的變數或常數樣式，然後在右邊寫上要對映數值，就像之前說的"鏡子"般的對應。當沒有對應的值時，就會得到undefined。

從物件解構賦值(Object destructuring)

物件除了有使用特別的字面符號，也就是花括號({})來定義，其中也會包含屬性。按照基本的原則，也是用像"鏡子"般的樣式對應，一樣看範例就很容易理解：

```
// 基本用法
const { user: x } = { user: 5 } // x=5

// 失敗保護(Fail-safe)
const { user: x } = { user2: 5 } //x=undefined

// 賦予新的變數名稱
const { prop: x, prop2: y } = { prop: 5, prop2: 10 } // x=5, y=10

// 屬性賦值語法
const { prop: prop, prop2: prop2 } = { prop: 5, prop2: 10 } //prop =

// 相當於上一行的簡短語法(Short-hand syntax)
const { prop, prop2 } = { prop: 5, prop2: 10 } //prop = 5, prop2=10

// ES7+的物件屬性其餘運算符
const {a, b, ...rest} = {a:1, b:2, c:3, d:4} //a=1, b=2, rest={c:3, (
```

下面的語法是個有陷阱的語法，這是錯誤的示範：

```
// 錯誤的示範:
let a, b
{ a, b } = {a: 1, b: 2}
```

> 註: 這個語法如果使用const來宣告常數是根本不能使用，只能用let來宣告變數。而且eslint檢查工具一定會回報語法錯誤。

因爲在Javascript語言中，雖然使用花括號符號({})是物件的宣告符號，但這個符號用在程式敘述中，也就是前面沒有let、const、var這些宣告字詞時，則是代表程式碼的區塊(block)。在外面再加上括號符號(())就可以改正，括號符號(())有表達式運算的功能，正確的寫法如下：

```
let a, b
({ a, b } = {a: 1, b: 2}) //a=1, b=2
```

註: 在大部份情況，你應該是在定義常數或變數時，就進行解構賦值。

複雜的物件或混合陣列到物件，如果你能記住之前說的鏡子樣式對映基本原則，其實也很容易就能理解：

```
// 混用物件與陣列
const {prop: x, prop2: [, y]} = {prop: 5, prop2: [10, 100]}

console.log(x, y) // => 5 100

// 複雜多層次的物件
const {
  prop: x,
  prop2: {
    prop2: {
      nested: [ , , b]
    }
  }
} = { prop: "Hello", prop2: { prop2: { nested: ["a", "b", "c"]}}}

console.log(x, b) // => Hello c
```

從非陣列或非物件解構賦值

從其他的資料類型進行陣列或物件解構，一開始是null或undefined這兩種時，你會得到錯誤：

```
const [a] = undefined
const {b} = null
//TypeError: Invalid attempt to destructure non-iterable instance
```

如果是從其他的原始資料類型布林、數字、字串等作物件解構，則會得到undefined值。

```
const {a} = false
const {b} = 10
const {c} = 'hello'

console.log(a, b, c) // undefined undefined undefined
```

從其他的原始資料類型布林、數字、字串等作陣列解構的話，只有字串類型可以解構出字元，在上面的例子有看到這種情況，其他也是得到undefined值：

```
const [a] = false
const [b] = 10
const [c] = 'hello' //c="h"

console.log( a, b, c)
```

註: 字串資料類型的值只能用在陣列的解構賦值，無法用在物件的解構賦值。

以上會有出現這樣的結果，是當一個值要被進行解構時，它會先被轉成物件(或陣列)，因爲

null或undefined無法轉成物件(或陣列)，所以必定產生錯誤，這是第一階段。下一個階段如果這個值轉換的物件(或陣列)，沒有附帶對應的迭代器(Iterator)就無法被成功解構賦值，所以最後回傳undefined。

解構賦值時的預設值

在等號左邊的樣式(pattern)中是可以給定預設值的，作爲如果沒有賦到值時(對應的值不存在)的預設數值。

```
const [missing = true] = []
console.log(missing)
// true

const { message: msg = 'Something went wrong' } = {}
console.log(msg)
// Something went wrong

const { x = 3 } = {}
console.log(x)
// 3
```

要作一個簡單的陷阱題滿簡單的，你可以試看看下面這個範例中到底是賦到了什麼值：

```
const { a ='hello' } = 'hello'
const [ b ='hello' ] = 'hello'

console.log( a, b)
```

在函式傳入參數定義中使用

在函式傳入參數定義中也可以使用解構賦值，因爲函式的傳入參數本身也有自己的預設值設定語法，這也是ES6的一個特性，所以使用上會容易與解構賦值自己的預設值設定搞混。這地方會產生不少陷阱。

一個簡單的解構賦值用在函式的參數裡，這是正常情況的語法：

```
function func({a, b}) {
  return a + b
}

func({a: 1, b: 2}) // 3
```

當你用上了預設值的機制，而且前面的a有預設值，後面的b就沒有，這時候因爲沒有賦到值時，都會是undefined值，任何數字加上undefined都會變成NaN，也就是非數字的意思：

```
function func({a = 3, b}) {
  return a + b
}

func({a: 1, b: 2}) // 3
func({b: 2}) // 5
func({a: 1}) // NaN
func({}) // NaN
```

```
func() // Cannot read property 'a' of undefined
```

當a與b兩個都有預設值時，NaN的情況不存在：

```
function func({a = 3, b = 5}) {
  return a + b
}

func({a: 1, b: 2}) // 3
func({a: 1}) // 6
func({b: 2}) // 5
func({}) // 8
func() // Cannot read property 'a' of undefined
```

實際上函式傳入參數它自己也可以加預設值，但這情況會讓最後一種func()呼叫時與func({})相同結果：

```
function func({a = 3, b = 5} = {}) {
  return a + b
}

func({a: 1, b: 2}) // 3
func({a: 1}) // 6
func({b: 2}) // 5
func({}) // 8
func() // 8
```

另一種情況是在函式傳入參數的預設值中給了另一套預設值，這只會在func()時發揮它的作用：

```
function func({a = 3, b = 5} = {a: 7, b: 11}) {
  return a + b
}

func({a: 1, b: 2}) // 3
func({a: 1}) // 6
func({b: 2}) // 5
func({}) // 8
func() // 18
```

你可以觀察一下，當對某個變數賦值時你給他null或void 0，到底是用預設值還是沒有值，這個範例的g()函式是個對照組：

```
function func({a = 1, b = 2} = {a: 1, b: 2}) {
  return a + b
}

func({a: 3, b: 5}) // 8
func({a: 3}) // 5
func({b: 5}) // 6
func({a: null}) // 2
func({b: null}) // 1
func({a: void 0}) // 3
func({b: void 0}) // 3
```

```
func({}) // 3
func() // 3

function g(a = 1, b = 2) {
  return a + b
}

g(3, 5) // 8
g(3) // 5
g(5) // 7
g(void 0, 5) // 6
g(null, 5) // 5
g() // 3
```

註：所以在函式傳入參數中作解構賦值時，給定null值時會導致預設值無用，請記住這一點。當數字運算時，null相當於0。

實例應用

迭代物件中的屬性值

這個範例用了for...of語法。出自Destructuring assignment：

```
const people = [
  {
    name: 'Mike Smith',
    family: {
      mother: 'Jane Smith',
      father: 'Harry Smith',
      sister: 'Samantha Smith'
    },
    age: 35
  },
  {
    name: 'Tom Jones',
    family: {
      mother: 'Norah Jones',
      father: 'Richard Jones',
      brother: 'Howard Jones'
    },
    age: 25
  }
];

for (let {name: n, family: { father: f } } of people) {
  console.log('Name: ' + n + ', Father: ' + f)
}

// "Name: Mike Smith, Father: Harry Smith"
// "Name: Tom Jones, Father: Richard Jones"
```

結合預設值與其餘參數

這個範例混用了一些ES6的語法，出自Several demos and usages for ES6 destructuring.：

```
// 結合其他ES6特性
const ajax = function ({ url = 'localhost', port: p = 80}, ...data) {
  console.log('Url:', url, 'Port:', p, 'Rest:', data)
}

ajax({ url: 'someHost' }, 'additional', 'data', 'hello')
// => Url: someHost Port: 80 Rest: [ 'additional', 'data', 'hello' ]

ajax({ }, 'additional', 'data', 'hello')
// => Url: localhost Port: 80 Rest: [ 'additional', 'data', 'hello' ]
```

_.pluck

這個例子相當於Underscore.js函式庫中的`_.pluck`，把深層的屬性值往上拉出來。

```
var users = [
  { user: "Name1" },
  { user: "Name2" },
  { user: "Name2" },
  { user: "Name3" }
]

var names = users.map( ({ user }) => user )

console.log(names)
// => [ 'Name1', 'Name2', 'Name2', 'Name3' ]
```

React Native的解構賦值

這個是React Native的一個教學，裡面有用了解構賦值的語法。出自React Native Tutorial: Building Apps with JavaScript：

```
var React = require('react-native')

var {
  StyleSheet,
  Text,
  TextInput,
  View,
  TouchableHighlight,
  ActivityIndicatorIOS,
  Image,
  Component
} = React
```

參考資料

- Several demos and usages for ES6 destructuring.
- Destructuring Assignment in ECMAScript 6
- Destructuring assignment MDN
- Destructuring assignmentのご利用は計画的に

模組系統

模組系統

注意: 本文尚在撰寫中，有一些部份還未完成。

JavaScript語言長期以來並未內建支援模組系統，社群上發展了兩套知名的模組系統，但它們並不相容:

- CommonJS Modules
- Asynchronous Module Definition (AMD)

ES6中加入了模組系統的支援，它採用了CommonJS與AMD的優點，成了未來JavaScript語言中重要的特性。

模組系統是什麼？

當程式碼愈寫愈多，應用程式的規模愈來愈大時，我們需要一個用於組織與管理程式碼的方式，這個需求相當明確，或許不只是應用程式發展到一定程度才會考慮這些，而是應該在開發程式之前的規劃就需要考量進來。

JavaScript語言是一個沒有命名空間設計的程式語言，也沒有支援類似的組織與程式碼分離的設計。有些人認爲使用物件定義的字面文字，可以定義出物件的方法與屬性，但如果你看過"物件"、"this"與"原型物件導向"的章節內容，就知道物件中並沒有區分私有、公開成員或方法的特性，這個組織法頂多只是把方法或屬性整理集中而已。

而在很早之前(2003)在社群上發展出一個稱之爲模組樣式(module pattern)，以及之後的變型如暴露模組樣式(Revealing Module Pattern)，就是第一期的程式碼組織方式。模組樣式實作相當簡單，有許多早期開始發展的函式庫或框架採用這個樣式，甚至到今天也可以看到它的使用身影。一個簡單的範例如下(以下範例來自jQuery):

```
// The module pattern
var feature = (function() {

    // Private variables and functions
    var privateThing = "secret";
    var publicThing = "not secret";

    var changePrivateThing = function() {
        privateThing = "super secret";
    };

    var sayPrivateThing = function() {
        console.log( privateThing );
        changePrivateThing();
    };

    // Public API
    return {
        publicThing: publicThing,
```

```
        sayPrivateThing: sayPrivateThing
    };
})();

feature.publicThing; // "not secret"

// Logs "secret" and changes the value of privateThing
feature.sayPrivateThing();
```

它使用了IIFE函式的特性，區分出作用域，不過它並沒有辦法徹底解決問題，它在小型的應用程式可以用得很好，但在複雜的程式中仍然有很大的問題，例如以下的問題:

- 沒辦法在程式中作模組載入
- 模組之間的相依性不易管理
- 異步載入模組
- 除錯與測試都不容易
- 在大型專案中不易管理

模組樣式似乎是一個暫時性的解決方案，但不得不說它的確是上一代很重要的程式碼組織方式，第二代的模組系統，是在2009年之後的CommonJS與AMD計劃，它們實作出眞正完整的模組系統，CommonJS是專門設計給伺服器端的Node.js使用的，而AMD的目標對象則是瀏覽器端。當然它們兩者的設計有所不同，也不相容，使用時也可能需要搭配載入工具來一併使用，不過這個階段的模組系統已經是較前一代完善許多，在相依性與模組輸出與輸入，都有相對的解決方式，程式碼的管理與組織方便了許多。

CommonJS與AMD並不會在這裡討論，我們的重點是是ES6中的模組系統，ES6中加入了模組系統的支援，它採用了CommonJS與AMD的優點，是一個語言內建的模組系統，而且它可以使用於瀏覽器與伺服器端，這是一個重大的新特性，可以讓你的開發日子更輕鬆許多。

模組如何開始使用

ES6的模組系統使用上相當簡單，大致上只有三個重點:

- ES6的模組程式碼會自動變成strict-mode(嚴格模式)，不論你有沒有使用"use strict"在程式碼中。
- ES6的模組是一個檔案一個模組
- ES6模組使用export(輸出)與import(輸入)語句來進行模組輸出與輸入。輸出通常位於檔案最後，輸入位於最前面。

模組輸出與輸入

有寫成模組的程式碼檔案，才能讓其他程式碼檔案進入輸入。模組輸出可以使用export關鍵字，在想要輸出(也就是變爲公開部份)加在前面，物件、類別、函式與原始資料(變數與常數)都可以輸出，例如以下的範例:

多個輸出名稱

```
export const aString = 'test'

export function aFunction(){
  console.log('function test')
}
```

```
export const aObject = {a: 1}

export class aClass {
  constructor(name, age){
    this.name = name
    this.age = age
  }
}
```

上面稱之爲多個輸出名稱的情況，有兩種方式可以進行輸入，一種是每個要輸入的名稱都需要定在在花括號({})之中，例如以下的範例:

```
import {aString, aObject, aFunction, aClass} from './lib.js'

console.log(aString)
console.log(aObject)
```

另一種是使用萬用字元(*)，代表要輸入所有的輸出定義的值，不過你需要加上一個模組名稱，例如下面程式碼中的myModule，這是爲了防止命名空間的衝突之用的，之後的程式碼中都需要用這個模組名稱來存取輸出模組中的值，這個作法不常使用:

```
import  * as myModule from './lib.js'

console.log(myModule.aString)
console.log(myModule.aObject)

myModule.aFunction()
const newObj = new myModule.aClass('Inori', 16)
console.log(newObj)
```

單一輸出名稱

這個要輸出成爲模組的程式碼檔案中，只會有一個輸出的變數/常數、函式、類別或物件，通常會加上default關鍵詞。如果要使用有回傳值的函式，通常也是用單一輸出的方式。例如以下的範例:

```
function aFunction(param){
  return param * param
}

export default aFunction
```

對單一輸出的模組就不需要用花括號，這代表只輸入以default值定義的輸出語句:

```
import aFunction from './lib2.js'

console.log(aFunction(5))
```

這是最特別的，可以在輸入時改變輸入值的名稱，這樣可以讓作輸入檔案中，確保不會有名稱衝突的情況:

```
import square from './lib2.js'

console.log(square(5))
```

合法的輸出語法

```
export var x = 42;                        // export a named variable
export function foo() {};                 // export a named function

export default 42;                        // export the default export
export default function foo() {};         // export the default export

export { encrypt };                       // export an existing variabl
export { decrypt as dec };                // export a variable as a new
export { encrypt as en } from 'crypto'; // export an export from anot
export * from 'crypto';                   // export all exports from an
```

合法的輸入語法

```
import 'jquery';                          // import a module without an
import $ from 'jquery';                   // import the default export
import { $ } from 'jquery';               // import a named export of a
import { $ as jQuery } from 'jquery';     // import a named export to a
import * as crypto from 'crypto';    // import an entire module insta
```

參考資料

- http://stackoverflow.com/questions/36795819/react-native-es-6-when-should-i-use-curly-braces-for-import/36796281#36796281
- http://exploringjs.com/es6/ch_modules.html
- http://www.2ality.com/2014/09/es6-modules-final.html
- https://addyosmani.com/resources/essentialjsdesignpatterns/book/#modularjavascript
- https://www.nczonline.net/blog/2016/04/es6-module-loading-more-complicated-than-you-think/

附錄