JS 網頁開發者 (試讀版)

劉君羿

2012/12/24

Contents

Part I

Getting Started with JavaScript

Chapter 1

JavaScript the Language

這個章節我將介紹 JS 這個語言的一些基本性質。

1.1 Primitive Value & Reference Value

JS 中有兩種變數：primitive value 和 reference value。Primitive value 包含 undefined, null, boolean, number, string；reference value 則是 object。這兩種 value 的差異在於，primitive value 是儲存在 memory 的 stack 中，而 reference value 則是儲存在 heap 中 (但 reference 本身存在 stack)。舉例來說，宣告變數：

```
1 // primitive value
2 var i = 0;
3 var b = false;
4 var s = "hello, world";
5 // reference value
6 var o = {};
7 var a = [];
```

則 stack 中會有 `i`, b, s 的值，以及 `o`, a 的 reference。存在 stack 中表示這些值大小是固定的。`o`，`a` 的 reference 指到 heap 中的某塊記憶體位址。由於 object 的大小常常會改變，改變的時候只要在 heap 中 allocate 一塊新的記憶體空間，然後讓 stack 中的 reference 只到它就好了。

你可能會想問為什麼 string 的大小也是固定的？

那是因為在 JS 中，string 在 assign 過後它的值就不會變。考慮這段程式

```
var s = "hello";
s += ", world";
```

由於 string 在 stack 中的大小已經固定，所以無法擴充它的長度以補上", world"，而是需在 stack 中則 allocate 一塊新的空間給合併後的結果"hello, world"、把 s assign 到這個新的 value，而原來 stack 中的"hello" 則不再使用。

你可能又會問，如果 string 是 primitive value 了話，那為什麼可以呼叫 string 的 slice(), split() 等 methods 和取得 length 大小？不是只有 object 才會有 methods 和 properties 嗎？

簡單的說，這是因為在 JS 中，number, string, boolean 這三個 primitive values 都有對應到的 **primitive wrapper type** (Number, String, Boolean)，這些 primitive values 在讀取之前會先被動態得 wrap 起來，wrap 起來後就是一個 object，因此才會有 methods 和 properties。之後介紹到 primitive wrapper types 時，會有更詳細的說明。

在使用上，primitive value 是 access by value，而 reference value 則是 access by reference。下方範例展示之間的區別：

```
var i = 0;
var j = i;
i++;
console.log(j); // 0

var userA = {
  name: "Tom";
};

var userB = userA;
userA.name = "Jerry";
console.log(userB.name);  // Jerry
```

把 i assign 給 j 時，i 的值會被複製一遍後 assign 給 j，因此修改 i 並不影響 j。把 userA assign 給 userB 時，userA 也會被複製一遍，但是複製的是 reference 而不是 object 本身，因此這時候 stack 中會有兩個 reference 指向同一塊 heap 位址，也就是真正儲存 userA 的地方，因此修改 userA 也會影響到 userB。

再看一個例子：

```
var i = 0;
var f = function (i) {
  i++;
};
f(i);
console.log(i); // 0

var user = {
  name: "Tom";
};
var f = function (user) {
  user.name = "Jerry";
};
f(user);
console.log(user.name); // Jerry
```

把 i 當作 f() 的參數時，i 會被複製一遍，因此在 f() 中修改 i 並不影響原來的 i；把 user 傳給 f() 時，user 也會被複製一遍，但是複製的是 reference 而不是 value，因此這時候 stack 中會有兩個 reference 指向同一塊 heap 位址，也就是真儲存 user value 的地方，因此在 f() 中操作 user 也會影響到原來的 user。

1.2 Scope

在詳細介紹 reference types 之前，我打算先介紹 scope 的概念。

JS 在還沒執行到任何 function 時，是在 **global execution context** 中運行；執行到一個 function 時便會 push 一個新的 execution context 到 execution stack 中；function 結束時會把新的 execution context pop 出去。Execute context 定義了當時的 scope，也就程式執行到那一行時可以看到那些 variables[1]。一個 execute context 的 scope 除了包括自身定義的 variables 外，也包括該 execution context 所屬的 execution context 的 scope 中的 variables(這就是 **scope chain** 的概念)。考慮下方的程式：

```
// 在 browser 中執行
var i = 0;

var f = function () {
```

[1]在 browser 中，global execution context 就是 window 這個 object，所以我們常使用的 global function encodeURI()、global reference types Object, Array、global instance document 等都定義在 window 的 scope 中。

```
5   var j = 0;
6   console.log(i);
7 };
8 f();  // console.log(i) 會輸出 0
9
10 console.log(j); // undefined
```

由於 `f()` 是在 global execution context 中定義的，因此在 `f()` 的 scope 中也可以看到 global execution context scope 中的 variable `i`。然而執行完 `f()` 之後，由於 `f()` 的 execution context 已經 pop 出 execution context stack 了，因此 `j` 也跟著消失。

必須注意的是，在宣告變數時，如果前面加了 `var` 則該變數會被加到最近的 scope 中，但是**如果沒加 `var` 了話，則宣告的變數會被加到 global scope 中**。請看下方範例：

```
1 var f = function () {
2   var i = 0;
3   j = 0;
4 };
5 f();
6
7 console.log(i); // undefined
8 console.log(j); // 0
```

在 `f()` 中宣告 `i` 時，由於有加 `var`，所以 `i` 會被加到離它最近的 `f()` 的 scope；反之，`j` 則會被加到 global scope 中。所以當 `f` 執行完後，`i` 跟著 `f()` 的 execution context 消失，但 `j` 仍然留在 global context 中。

因此，我的建議是，不管宣告任何變數，前面一律加上 `var`，避免覆蓋掉之前的變數。

還有一點必須注意的是，**JS 並沒有 block scope**。在某些語言 (C, JAVA 等) 中，凡是出現 {}(block) 就表示一個 scope，在 scope 外面看不到 scope 裡面的變數。因此下方的 C 程式碼並沒辦法成功 compile，因為 `i` 被一個 scope 包住，在那個 `scope` 外無法看到 `i`。

```
1 // C language
2 int main(void) {
3   {
4     int i = 0;
5   }
6   // gcc: 錯誤:「i」undeclared (first use in this function)
7   printf("%d\n", i);
8 }
```

但在 JS 中除了 `function () {}` 會產生 scope 外，其他 block 則不會。所以：

```
if (true) {
  var i = 0;
}
console.log(i); // 0
```

`if` 並不會產生新的 scope，所以在 `if` block 中宣告的變數會存在於外面的 scope 中。

同樣的，在 `for` 迴圈的 () 中宣告的變數也不會隨著 `for` 迴圈消失：

```
for (var i = 0; i < 10; i++) {}
console.log(i); // 10
```

1.2.1 Closure

有了 scope 的概念後，就可以介紹一個 JS function 的特殊用法：closure。

由於 function 會產生新的 scope，所以 closure 便運用這個概念，將變數保留在 closure 之中。Closure 的基本型態就是在 function 中回傳另一個 function，回傳後，只剩下被回傳的 function 看的到原來 function 的 scope。考慮下方程式：

```
var getCounter = function (startFrom) {
  var counter = function () {
    return startFrom++;
  };
  return counter;
};

var counter = getCounter(5);
counter();  // return 5
counter();  // return 6
```

`getCounter()` 會回傳另一個 function `counter()`，這個 `counter()` 本身並未定義 `startFrom`，不過由於它存在於 `getCounter()` 的 scope 中，因此也看的到 `getCounter()` 的 scope 中的變數 `startFrom`。當 `getCounter()` 執行完後，它的 scope 並不會馬上消失，因為它的 scope 已經「紀錄」在 `counter()` 的 scope chain 中了。所以，雖然在 global scope 中看不到 `startFrom`，但透過呼叫 `counter()`，仍可以將它的值加一。

舉一個現實中會運用到 closure 的例子：

```
// 這是結果不如預期的範例

// 我個人的習慣是，避免在 for () 中宣告變數，
// 以免 override 外面的變數
var i;
for (i = 0; i < i < 10; i++) {
  document.getElementById('button-' + i).onclick = function () {
    console.log(i);
  };
}
```

這個範例中，我希望當我點擊第 `i` 個 button 時，就會輸出 `i`。但實際的結果卻不如預期：不管點哪一個 button，都只會輸出 10。這是因為當我點擊任一 button 時，for loop 早就執行結束了，因此 `onclick()` 看到的 `i` 是 10。為了捕捉每個 loop 當下的 `i`，我們就必須用到 closure 的概念：

```
var i;
for (i = 0; i < 10; i++) {
  document.getElementById('button-' + i).onclick = (function (
      output) {
    return function () {
      console.log(output);
    };
  })(i);
}
```

這裡我們並非直接 assign 一個 function 給 `onclick`，而是呼叫一個 closure，把 `i` 當成 closure 的參數 `ouput`，然後 closure 會回傳一個會輸出 `output` 的 function。如同上一節所提的，把 primitive value 當成 function argument 傳遞是時其值會被複製一遍，因此 closure 中的 `output` 是當時 `i` 的複製，不會隨著 `i++` 而改變。Closure 回傳後，它的 execution context 就不再被外面的 context 所知 (也就是說，它已經 close 了)，只剩回傳的 function 還能讀取到 `output` 的值。如此，透過 closure 的紀錄功能，便能得到我們希望的結果：當我們 click 一個 button 時，`onclick()` 輸出當時記錄下來的 `output` 而非 `i`。

Closure 在 JS 中的運用很多，想要靈活的運用它就得清楚 scope 的概念。更多的範例將在之後的章節一一介紹。

1.3 Built-in Reference Types

在 JS 中，有幾個內建的 reference types，包括：

- `Object`
- `Array`
- `Function`
- `Date`
- `RegExp`
- Primitive wrapper types: `Boolean`, `Number`, `String`

每個 type 都有自己的 methods 和 properties，但我不打算一一介紹，而是針對否些 types 的特殊性質加以描述。

1.3.1 Function

在 JS 中，function 跟其他的 variables 沒什麼差別。在介紹 scope 的時候我們就看過把 function 當 return value 的例子了，其實 function 不僅可以被回傳，還可以當作其他 functions 的參數。比方：

```
1 var writingHomework = function () {
2   console.log("I'm writing HW.");
3 };
4 var student = {};
5 student.do = function (something) {
6   something();
7 };
8
9 student.do(writingHomewrok);  // I'm writing HW.
```

注意`student.do(writingHomewrok)`這行的 `writingHomework` 後面並沒有加上 `()`，因為我並沒有要直接呼叫它，而是把它交給 `student.do()` 呼叫。再看一個更實際的例子，我想要 sort 一個 array of objects：

```
1 var sortBy = function (array, comparator) {
2   // bubble sort (increment)
3   var i, j, tmp;
4   for (i = 0; i < array.length; i++) {
5     for (j = i + 1; j < array.length; j++) {
6       if (comparator(array[i], array[j]) > 0) {
7         // jth object is smaller, need swap
```

```
8           tmp = array[j];
9           array[j] = array[i];
10          array[i] = tmp;
11        }
12      }
13    }
14  };
15
16  var students = [
17    { name: "Tom", grade: 100 }
18    , { name: "Jerry", grade: 99 }
19  ];
20
21  var gradeComparator = function (s1, s2) {
22    if (s1.grade < s2.grade) {
23      return -1;
24    } else if (s1.grade > s2.grade) {
25      return 1;
26    } else {
27      return 0;
28    }
29  };
30
31  sortBy(students, gradeComparator);
32  console.log(students);  // the order is Jerry, Tom
```

上方範例首先定義了 `sortBy()`，第一個參數是要被排序的 array，第二個是 comparator；comparator 是一個 function，把任兩個 value 給它以後，會比較它們的大小。接下來我們想要用 `sortBy()` 將幾個 objects 排好，以這個例子來說，我希望按照成績大小排序兩個學生；但是 `sortBy()` 並不了解如何排序學生，因此我們必須定義好按照成績比大小的 comparator(`gradeComparator()`)，把它交給 `sortBy()` 使用。

這種方法的好處是提供 `sortBy()` 更大的可利用性；`sortBy()` 不僅可以用來排序學生，只要給它適當的 comparator 就能排序其他 objects；比方如果要用來按照價格排序書籍了話，只要給 `sortBy()` 一個 `priceComparator()` 就能使用了。

Functions Declarations vs. Function Expression

你或許有注意到我宣告 function 的習慣：我宣告 function 時會用`var f = function (){};`而非`function f (){}`(前者是宣告變數的寫法，所以我習慣上

會加上分號)。除了~~在美感上~~這種宣告方式比較能凸顯在 JS 中 function 跟其他 variables 沒有兩樣外，這兩者之間還是有本質上的不同。請看下方範例

```
f();  // 1
function f () {
  return 1;
};

g();  // TypeError: undefined is not a function
var g = function () {
  return 1;
};
```

前者稱為 function declaration，後者則稱為 function expression。前者之所以不會出問題是因為 JS interpreter 在處理 JS 程式碼時可以分成幾個步驟 (在此不加以描述)，function declaration 在程式執行就會被先處理，而 function expression 是在執行的時候才處理。因此呼叫 `g()` 時，`g()` 還沒被定義，所以會出現 `TypeError`。

或者可以換另外一種說法，使用 function declaration 時，function declaration 會被「提升[2]」到最近的 scope 的開頭。比方

```
function f () {
  g();
  if (false) {
    function g () {
      console.log(0);
    }
  }
};
f();  // 0
```

可以看成：

```
// 模擬「提升」function declaration
function f () {
  function g () {
    console.log(0);
  }
  g();
  if (false) {
  }
```

[2]原文 hoist，出自網路文章 JavaScript Scoping and Hoisting (`http://www.adequatelygood.com/2010/2/JavaScript-Scoping-and-Hoisting`)

```
};
f();  // 0
```

在這個模擬中，雖然`if (false){}`不會不執行到，但是 function declaration 是在執行前就被處理的，所以不管執不執行的到都一樣 (你可以試試看把`if (false){}`改成 `return` 看看，結果還是一樣)。而且如同我們先前在1.2中所提的，`if (false){}`並不是一個 scope，所以 `f()` 就成了距離 `g()` 最近的 scope，因此會被提升到 `f()` 的最上方[3]。

因此，為了避免提升造成讓人困惑的程式碼，我通常不會使用 function declaration。

Function Inner Objects

在 `Function` 內部有兩個特殊的 objects：`this` 和 `arguments`。`this` 就是 function 執行時所處的 scope。舉例來說：

```
// 假設在 browser 中執行 var f = function () console.log(this);;f(); //
   windowvar o = g: f;o.g(); // o
```

我們是在 `window` 的 scope 下呼叫 `f()` 的，所以 `f()` 中的 `this` 就是 `window`；而`o.g()`是把 `f()` 當作 `o` 的 method 呼叫，所以執行時的 scope 是 `o`。

另一個 inner object 是 `arguments`。請看下方用法：

```
var sum = function () {
  var i;
  var sum = 0;
  for (i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
};
console.log(sum(1, 2, 3));  // 6
```

[3]其實用 `var` 宣告變數也會有提升的效果，只是被提升後，會先預設成 `undefined`。所以：

```
console.log(a);  // ReferenceError: a is not defined

console.log(b);  // undefined
var b;
```

使用上，`arguments` 是一個 array of arguments[4]。有時候我們並不知道使用者到底給了一個 function 多少 arguments，因此就可以用 `arguments.length` 來判斷。

Function Methods

`Function` 有兩個特殊的 methods：`call()` 和 `apply()`。這兩個 methods 讓我們可以改變 function 執行的的 scope，也就是改變它的 `this`。延續我們在介紹 `this` 時所舉的例子，我們可以不需要定義 `o.g()` 就能在 `o` 的 scope 下呼叫 `f()`：

```
1  var f = function () {
2    console.log(this);
3  };
4
5  var o = {};
6  f.call(o);  // o
```

透過 `call()` 我們把 `f()` 執行的 scope 設定成 `o`，所以`console.log(o);`就會輸出 `o`。

使用 `call()`，第一個參數是 scope，之後的參數則是給 function 的參數 (`f.call(scope, arg1, arg2, ...)`)；`apply()` 跟 `call()` 的差別就只是 `apply()` 後面參數是包在一個 array 中 (`f.apply(scope, [arg1, arg2, ...])`)。舉個 `call()` 的例子：

```
1  var students = [
2    { name: "Tom" }
3    , { name: "Jerry" }
4  ];
5
6  var sayName = function (number) {
7    console.log("Student " + number + ": " + this.name);
8  }
9
10 var i;
11 for (i = 0; i < students.length; i++) {
12   sayName.call(students[i], i);
13 }
14 // output:
15 // Student 0: Tom
```

[4] `arguemtns` 並不是 `Array`(用`arguemtns instanceof Array`便能判斷出來)，只是使用上差不多而已。

```
16 // Student 1: Jerry
```

`call()` 和 `apply()` 的運用很多，較複雜的例子會在後面看到。

1.3.2 Primitive Wrapper Types

在1.1我曾經答應過要說明為什麼 string 是 primitive value 卻又有 methods 和 properties。這是因為在 JS 中，number, string, boolean 這三個 primitive value 都有對應到的 primitive wrapper type：`Number`, `String`, `Boolean`。**當讀取這三種 primitive values 時，JS interpreter 會先將它們用對應的 primitive wrapper types 包起來**，因此：

```
1 var length = "hello, Wolrd!".length;
```

可以被解讀成

```
1 var length = String("hello, Wolrd!").length;
```

不過該 wrapper 只有一行的效用，執行下一行之前就會被 unwrap，所以：

```
1 var book = "Notes for JS Web Developer";
2 book.author = "Trantor";
3 console.log(book.author); // undefined
```

在讀取 book 時，`book` 會先被 `String()` 包起來。竟然是 object，就可以動態的加上 property `author`。只不過執行到下一行時，`book` 已經先被 unwrap 後又 wrap 起來，而新的 wrapper 並沒有 `author` 這個 property，所以是 `undefined`。

1.4 Defining Reference Types

1.4.1 Constructor

在 JS 中並沒有 class，所有的 reference value (object) 都是某個 reference type 的 instance，更明確的說，**所有的 instance 都是 `new` 這個 key word 後面加上一個 reference type 產生的**。比方：

```
1 // 通常會用var o = {};
2 // 這兩種寫法是相同的
```

```
3 var o = new Object();
4
5 // 通常我們也不會用這種寫法
6 // 而是用更簡潔的var a = [1, 2, 3];
7 var a = new Array(1, 2, 3);
```

JS 透過 constructor 實作 reference type，constructor 說穿了其實就是 function，唯一的不同是使用的方式，**當 function 前面加了一個 new 時，就變成一個 constructor 了**。請看下方的程式：

```
1 var o = new (function () {});
2 console.log(o instanceof Object); // true
3
4 var Student = function (name) {
5   this.name = name;
6 };
7 var s = new Student("Tom");
8 console.log(s.name);  // Tom
```

當我們把 function 當 constructor 來用時，可以把它拆解成下方幾個步驟：

1. 產生一個新的 object
2. 把新的 object 當成 function 的 scope 呼叫 (也就是讓 function 中的 `this` 等於新的 object
3. function 回傳後，回傳新的 object

以上方的 Student 來說明，`var s = new Student()`這行程式會先產生一個新的 object，然後讓 Student 的 this 等於新的 object 後，呼叫 `Student()`。在 `Student()` 中，令新的 object 的 `name` 等於`"Tom"`，最後回傳新的 object。我們也可以把這個範例改成下方效果相等的程式：

```
1 var Student = function (name) {
2   this.name = name;
3   return this;
4 };
5 var s = {};
6 s = Student.call(s, "Tom");
7 console.log(s.name);  // Tom
```

這裡，我們先產生新的 object，然後用之前介紹過的 `call()` 把新的 object 當作 `Student()` 的 scope 呼叫，最後所得的 `s` 的 `name` 的值仍然一樣。注意這裡的 `Student()` 跟原來的不同，這裡多了一行`return this`，原來之所

以不用是因為 `new` 會自動幫我們回傳新的 object，因此 function 中不需要額外 `return`。

不過其實這兩種方式仍然有所不同：

```
var Student = function (name) {
  this.name = name;
  // 加這行並不影響 constructor 的用法，但對於第二種方法是必須的
  return this;
};

var s1 = new Student("Tom");
console.log(s1.constructor === Student);  // true
console.log(s1.constructor === Object);   // false
console.log(s1 instanceof Student);     // true
console.log(s1 instanceof Object);      // true

var s2 = {};
s2 = Student.call(s2, "Tom");
console.log(s2.constructor === Student);  // false
console.log(s2.constructor === Object);   // true
console.log(s2 instanceof Student);     // false
console.log(s2 instanceof Object);      // true
```

這是因為每個 object 都有一個叫 `constructor` 的 property，這個 property 指向 object 初始化時的 reference type，所以 `s1` 的 `constructor` 指向 `Student`，而 `s2` 的 `constructor` 則指向 `Object`。由於 `Student` 是由 `Function` 產生，而 `Function` 又是由 `Object` 產生，因此 `s1` 是 `Student` 的 instance，也是 `Object` 的 instance；反之，`s2` 不是由 `Student` 產生，所以只能算是 `Object` 的 instance[5]。

1.4.2 Prototype

雖然 constructor 可以用來定義一個 reference type 的 properties，但卻不適合用來定義 methods。請看下方範例：

```
var Student = function (name) {
  this.name = name;
  this.sayHi = function () {
    console.log("Hi, my name is", this.name);
  };
```

[5] *instanceof* 實際如和運作要等到介紹 *prototype chain* 後才會說明，這裡僅以簡單的繼承概念描述。

```
};

var s1 = new Student("Tom");
var s2 = new Student("Jerry");

console.log(s1.sayHi === s2.sayHi); // false
```

由於每次 `new Student()` 時，`Student` constructor 會產生一個新的 `sayHi()`，所以 `s1` 和 `s2` 的 `sayHi()` 並非同一個 function。這並不是我們想要的，我們只需要一個像 C++ 中的 static method 一樣、可以共用的 method 就好了。要解決這個問題，就必須搭配使用 prototype：

```
var Student = function (name) {
  this.name = name;
};
Student.prototype.sayHi = function () {
  console.log("Hi, my name is", this.name);
};

var s1 = new Student("Tom");
var s2 = new Student("Jerry");

console.log(s1.sayHi === s2.sayHi); // true
```

透過修改 student constructor 的 prototype，我們便可以幫它加上共用的 methods 或 properties。`s1` 和 `s2` 本身並沒有 `sayHi()`，之所以能夠呼 `sayHi()` 叫是因為呼叫時，JS interpreter 發現它們沒有 `sayHi()`，就會去它們的 prototype 裡面找，如果 prototype 中也沒有，那就去 prototype 的 prototype 裡面找，以此類推。這就是 **prototype chain** 的概念，有就是在 JS 中實作繼承的方法[6]。

實際上 prototype 是如何運作的呢？當一個 function 產生的時候，它的 prototype 也跟著產生，prototype 中有一個 property 叫做 constructor，constructor 指回 function 自己[7]。每當 constructor 產生新的 instance 時 (也就是前面加上 `new` 的時候)，新產生的 instance 會有一個 property 指回 constructor 的 prototye(而非直接指回 constructor)[8]。

所以當執行`s1.sayName()`時，是透過 `s1.__proto__`，找到 `Student.prototype`，然後執行 `Student.prototype` 的 `sayName()`。

[6]我並沒有打算在這本書中對繼承的概念加以著墨，不過相信有了 constructor 和 prototype 的概念後，只要上網找找 JS 繼承的相關文章便能了解

[7]所以`Student.prototype.constructor === Student`是 true

[8]所以`s1.__proto__ === Student.prototype`是 true

另外，直接修改 instance 上的 primitive values 並不會影響到 prototype：

```
var Student = function () {
};

// 透過 prototype 預設學生的學校
Student.prototype.school = "NTU";

var s1 = new Student();
var s2 = new Student();

s1.school = "CGU";

console.log(s1.school); // CGU
console.log(s2.school); // NTU
```

`s1.school = "CGU"`這一行不會影響到 `Student.prototype.school`，也就是說，`s1.school` 是`"CGU"`，但 `s1.__proto__.school` 仍然是`"NTU"`。之後當 access `s1.school` 時，由於 interpreter 在 `s1` 本身就找的到 `school` 這個 property 了，因此不需要往 `s1.__proto__` 裏頭找。我把這種現象稱作 `mask`，因為在 instance 的 property「遮蔽」掉了 prototype 上的 property。反之，當 access `s2.school` 時，由於在 `s2` 本身就找不到 `school`，因此往 `s1.__proto__` 裏頭找，得到的就是`"NTU"`。

但如果修改 instance 上的 reference values 就可能出問題：

```
var Student = function (name) {
  this.basicInfo.name = name;
};
Student.prototype.basicInfo = {
  name: ""
};

s = new Student("Tom");
console.log(Student.prototype.basicInfo.name);  // Tom
```

這是因為這個範例中並沒有 mask 的現象發生，修改 `s.basicInfo.name` 就是修改 `Student.prototype.basicInfo.name`。因此如果不想共用一個 reference property，就必須定義在 constructor 中。緊接著我們就會看到兩者合再一起的最終解決方案。

有了 prototype chain 的概念後，就可以說明 `instanceof` 如何運作：一個 instance 是 `instanceof` a constructor 如果能夠在 instance 的 prototype chain 中找到 constructor 的 prototype。所以由於 `s.__proto__ === Student.prototype`，所以 `s instanceof Student` 回傳 `true`。

1.4.3 The Hybrid Solution

讓我們回到如何定義 reference types 上吧。綜合 constructor 和 prototype 的好處，歸納出在 JS 上定義 reference types 的最佳解決方案：

- 用 constructor 定義非共享 (non-static) 的 properties
- 用 prototype 定義共享 (static) 的 properties 和 methods

因此就有了以下的範例：

```
var Student = function (name, school) {
  this.name = name;
  this.school = school;
  this.friends = [];
};
Student.prototype.sayHi = function () {
  console.log("Hi, my name is", this.name);
};
Student.prototype.makeFriendWith = function (student) {
  this.friends.push(student.name);
};

var s1 = new Student("Tom", "NTU");
var s2 = new Student("Jerry", "CGU");

s1.makeFriendWith(s2);

console.log(s1.friends);  // ["Jerry"]
```

不過如果你覺得定義每個 method 都要打一大串字很麻煩，你也可以用這種方法：

```
var Student = function (name, school) {
  this.name = name;
  this.school = school;
  this.friends = [];
};
Student.prototype = {
  constructor: Student
  , sayHi: function () {
    console.log("Hi, my name is", this.name);
  }
  , makeFriendWith: function (student) {
    this.friends.push(student.name);
  }
};
```

注意到由於我們直接覆蓋掉原來的 prototype，所以需要手動讓 `prototype.constructor` 指向 constructor。雖然就算不手動設定 constructor 也不會影響 `instanceof` 的結果，因為如同我們之前介紹的，`instanceof` 是透過 prototype 判斷，所以比方當我們透過 `s = new Student()` 產生新的 instance `s` 時，s.__proto__ 仍然會指向 `Student.prototype`，因此 `instanceof` 還是會回傳 `true`。只不過為了避免不必要的錯誤 (比方跟你合作的人拿 constructor 做某些判斷)，我們還是按照預設行為設定 prototype。

Chapter 2

Design Patterns

這章將會介紹一些常用到的 JS design patterns。雖然我們已經在1.4介紹過 constructor pattern 和 prototype pattern 了，但考慮它們對初學者的難度，我在這裡會再簡單的整理一遍。

2.1 Constructor

用 constructor 定義 reference type：

```
1  var Student = function (name, friends) {
2    // initializing properties
3    this.name = name;
4    this.friends = firends || [];
5
6    // defining methods
7    this.sayHi = function () {
8      console.log("Hi, my name is", this.name);
9    };
10 };
11
12 // using the constructor
13 s = new Student("Tom", ["Jerry"]);
```

這種方法的好處是可以在新增 instance 時，初始化一些 properties；缺點則是，用 constructor 定義的 methods 被非共享的，每 `new` 一個 instance 時都會重新產生新的 methods 給該 instance。

2.2 Prototype

用 prototype 定義 reference type：

```
var Student = function () {};

Student.prototype = {
  constructor: Student
  , name: null
  , friends: []
  , sayHi: function () {
    console.log("Hi, my name is", this.name);
  }
};

// using the prototype
s = new Student();
s.name = "Tom";
s.friends.push("Jerry");
```

這個方法可以解決 constructor 無法共享 methods 的問題，而且由於定義在 instance 上的 properties 只會 mask 掉 prototype 上的 properties，而非修改之，因此之後產生的 instances 不會受影響。不過這個方法並沒有辦法像 constructor 一樣可以 initialize properties，而且也不適合用來定義 reference properties，以這個例子來說，`s.friends.push("Jerry")`這行執行結束後，`Student.prototype.friends` 會變成 `["Jerry"]`，之後產生的 instances 初始的 `friends` 都會變成 `["Jerry"]`。

2.3 Constructor/Prototype Combined Pattern

在 JS 中定義 reference types 的最終解決方案：

```
var Student = function (name, friends) {
  // initializing properties
  this.name = name;
  this.friends = firends || [];
};

Student.prototype = {
  constructor: Student
  , sayHi: function () {
    console.log("Hi, my name is", this.name);
```

```
  }
};

// using the constructor
s = new Student("Tom", ["Jerry"]);
```

為了各取 constructor 和 prototype 所長，我們便將 properties 定義在 constructor 中；而 methods 則定義在 prototype 中。注意在定義 prototype 時要把 `constructor` 設定成原來的 constructor。

2.4 Module

有時候我們只是需要一個 instance 而非定義一個 reference type。比方我們想要一個 `validator`，它有幾個 method 可以驗證一個 variable 是不是我們預期的 type，是了話就回傳 `true`，反之回傳 `false`。或許我們一開始會這麼做：

```
var checkType = function (variable, type) {
  return (typeof variable === type);
};

var isArray = function (variable) {
  return checkType(variable, 'array');
};
var isObject = function (variable) {
  return checkType(variable, 'object');
};
// isNumber, isString...
```

首先定義 `checkType()`，`checkType()` 可以用來判斷一個 variable 是不是某個 type。之後為了方便使用，我們定義幾個更直接的 functions：`isArray()`，`isObject()`。

但如果什麼 functions 都這樣定義，gloable scope 很容易就被「汙染」了。因此我們就用一個 object 將它們包起來：

```
var validator = {
  checkType: unction (variable, type) {
    return (typeof variable === type);
  }
  , isArray: function (variable) {
    return this.checkType(variable, 'array');
  }
```

```
  , isObject: function (variable) {
    return this.checkType(variable, 'object');
  }
  // isNumber, isString...
};
```

確實，這樣是解決了問題；但是，這種方法能力有限，因為他無法定義 private properties/methods；而且要用到自己的 function 時一定要加上 `this`，稍微有點麻煩。

這時候 module pattern 就能派上用場了。Module pattern 透過 closure 的概念將 private properties/methods 包在 closure 中，再透過 `return` 一個 object，將 public properties/methods 傳遞到 closure 外。請看 module pattern：

```
var validator = (function () {
  var checkType = function (variable, type) {
    return (typeof variable === type);
  };

  var isArray = function (variable) {
    return checkType(variable, 'array');
  };
  var isObject = function (variable) {
    return checkType(variable, 'object');
  };
  // isNumber, isString...

  return {
    isArray: isArray
    , isObject: isObject
  };
})();
```

現在我只打算讓使用者使用 `isArray()`, `isObject()`，而 `checkType()` 則變成 module 的 private method。因此回傳的 object 並沒有 `checkType()`。透過 module pattern，我們既可定義 private properties/methods，同時也可以把這些 value 都包在 closure 內，不為外界所知。(如果你覺得這個範例難以理解了話，建議可以回頭看看1.2！)

2.5 PubSub

PubSub 是 publish/subscribe 的縮寫。PubSub 可以說是一個用來傳遞 events 的 pattern。最簡單的 pubsub object 提供兩個功能：publish 和 subscribe。也就是說透過這個 object，某些人可以關注某些 events，而某些人則可以發布 events。讓我們用最簡單的 pubsub pattern 實作一個聊天室：

```
var  chatRoom = (function () {
  var handlers = {};
  var pub = function (event) {
    // no callback, ignore the event
    if (!handlers[event]) return;

    var args = Array.prototype.slice.call(arguments, 1);

    var i;
    for (i = 0; i < handlers[event].length; i++) {
      handlers[event][i].apply(this, args);
    }
    return this;
  };
  var sub = function (event, callback) {
    if (!handlers[event]) handlers[event] = [];

    handlers[event].push(callback);
    return this;
  };

  return {
    pub: pub
    , sub: sub
  };
})();

// using username as the event name for registration
chatRoom.sub("Tom", function (msg) {
  console.log("Msg to Tom:", msg);
});

// talk to Tom
chatRoom.pub("Tom", "Hi, Tom. This is Jerry speaking.");
```

在講解這個範例如何運用 pubsub 之前，我先說明幾個你可能看不太懂的地方。

首先是`var args = Array.prototype.slice.call(arguments, 1)`這行，我們想把參數 `event` 從 `arguments` 移除，然後把剩下的參數 (`args`) 傳遞給 callbacks。不過如同我在1.3.1所提的，`arguments` 並不是 array，所以沒有 `slice()` 可以呼叫，因此必須借用 `Array` 的 `slice()`。

另外一點，你可能不知道為什麼在 `pub()` 和 `sub()` 中要回傳 `this`。這是為了提供一個方便的 chainable API；也就是說，使用者可以把一連串的 methods 串起來，像是`chatRoom.sub().pub().pub().sub()`，如此一來就能少打許多 `chatRoom` 了。

回到 pubsub 上，這個範例中，一個 user 可以把 username 作為 event 拿去 subscribe，註冊時給一個 callback 表示等這個 event 發生的時候就執行這個 callback。Callback 會被 push 到對應的 event 的 callback array 中。等之後 publish 某個 event 時，就會把該 event 的所有 callbacks 都拿出來執行。

PubSub pattern 在 JS 中的應用很多，其實 DOM 也運用了 pubsub pattern，DOM element 有一個 method 叫做 `addEventListener()`[1]，用法如下：`el.addEventListener("click", function (){});` jQuery 的 `bind()` 就是用 `addEventListener()` 實作的，用法也差不多：`$el.bind('click', function (){})`[2]。

此外，在 Node.js 中的 `events.EventEmitter` 也是 pubsub pattern，概念跟上面的範例差不多，只是多了幾個 methods，比方可以取消註冊，也可以註冊只執行一次就失效的 callback 等。雖然我們還沒教到 Node.js，不過還是節錄 Node.js 文件中的一段話讓你參考：

> Many objects in Node emit events: a `net.Server` emits an event each time a peer connects to it, a `fs.readStream` emits an event when the file is opened. All objects which emit events are instances of `events.EventEmitter`.

此外，我們之後會介紹到的 redis 以及 socket.io 也實作了 pubsub pattern，因此這可以說是一個非常重要的 pattern。

---

[1]可參考 MDN 的文件`https://developer.mozilla.org/en-US/docs/DOM/element.addEventListener`

[2]我習慣用 `el` 表示 DOM element；`$el` 表示 jQuery element

Chapter 3

Async JavaScript

JS 程式碼本身永遠是 sync 的，所謂 async 是指註冊一個 callback 到 JS engine 的 event loop 中，等 event 產生後才會執行。在 browser 或是 Node.js 中的 JS 都是在 single thread 上執行，這個 thread 分成兩個階段，第一階段是執行你的 JS，執行結束後，便進入 **event loop** 的階段。在 event loop 中，JS engine 會檢查 **event queue** 中是不是有新的 event 發生；有之，則把對應的 callbacks 都找出來執行。

請看下方範例：

```
1  var start = Date.now();
2  setTimeout(function () {
3    var end = Date.now();
4    console.log("run the callback after " + (end - start) + "ms");
5  }, 0);
6
7  var i;
8  for (i = 0; i < 1000000; i++) {}
9  console.log("for loop is done");
10
11 // output:
12 // for loop is done
13 // run the callback after 2451ms
```

首先我先註冊一個 callback 到 timeout event 的 queue 中，雖然我設定這個 timeout 是 0，也就是我希望 callback 馬上被執行[1]，不過實際執行的結果則

[1]在 Node.js 中，你可以用`process.nextTick(callback)`取代`setTimeout(callback, 0)`，因為它效能比較好，請看 benchmark：`https://gist.github.com/1257394`

是，callback 會在 2451ms 後才被執行；這是因為雖然 timeout event 當下就發生了，不過在沒執行完我們的程式之前，不會進入 event loop 的階段，所以 event 不會被處理；等 for loop 執行結束、進入 event loop 後，JS engine 發現在 event queue 中有一個 timeout event，就會把對應的 callback 拿出來執行，此時已經是在 event 發生的 2451ms 後了。也因此，`setTimeout()` 恐怕不是想像中的那麼精確，理由就是 callback 至少得等到我們的程式 `return` 後才會被執行。

JS 語言本身提供的 async API 就只有 `setTimeout()` 和 `setInterval()` 而已，剩下的 API 則來自於執行的環境；不同的環境下提供不同的 async API，比方在 browser 中，DOM 提供了 `addEventListener()`，以及 AJAX API 等，而 Node.js 中則是大部分 I/O 都提供了 sync 和 async 兩種 API。接下章節我打算針對上述幾類 async API 分別介紹。

不過在此之前，有一點值得一提的是，這本書中不斷用到的 `console.log()` 可能是 sync 也可能是 async：

```
1 var o = {};
2 console.log(o);
3 o.v = 0;
4
5 // output in Chrome, Firefox:
6 // { v: 0 }
7 // output in Node.js
8 // {}
```

這是因為在 Chrome 和 Firefox 中的 `console.log()` 是非同步的，而 Node.js 中的則是同步的。如此一來，在 browser 中用 `console.log()` debug 可能會產生誤解，因為輸出的 object 可能已經不是當時的樣子了；要解決這個問題，你可以先複製一份 object 後再交給 `console.log()`，若是用 Underscore.js 了話，可以這麼做：`console.log(_.clone(o))`[2]，不然，你也可以選擇印出 primitive value，比方`console.log(o.v)`就不會有這個問題。

[2]Underscore.js 是個很好用的 utility library，我們之後會教到的 Backbone.js 便把 Underscore.js 加入自己 dependencies；如果你還沒用過 Underscore.js 了話，建議你可以先去看看他們的文件。

3.1 DOM Event Object

使用 JS 在 DOM 上面設定一個 event listener 有兩種方式 (為了方便，以下我將用 e 作為 event 的簡寫)：

```
el.addEventListener('click', function (e) {
  console.log(e);
});

el.onclick(function (e) {
  console.log(e);
});
```

當某個 event 發生時，一個 DOM Event object 便產生了，這個 event object 會從 event target (也就是 event 發生的 element) 開始往上傳遞給該 target 的所有祖先；途中若遇到某個 element 有這個 event 的 event listener，該 listener 便會被觸發；觸發時會將這個 event object 當成 event listener 的第一個參數，也就是上方程式碼中的 e。考慮下方的 HTML：

```
<!DOCTYPE html>
<html>
<body>

<div>
  <button>Click Me</button>
</div>

</body>
</html>
```

當我點擊 <button> 時，event 會從 <button> 開始往上傳遞，途中經過 、<body>，最後則是 <html>。

DOM Event object 有幾個 properties 和 methods 很常用到；e.target 是產生該 event 的 element，e.currentTarget 則是擁有目前執行到的 event listener 的 element；請看下方範例：

```
// 此範例使用上方的 HTML

// 以下的輸出是在點擊 <button> 後產生
document.getElementsByTagName('div')[0].onclick = function (e) {
  console.log(e.target);       // <button>
  console.log(e.currentTarget); // <div>
});

```

```
9  document.getElementsByTagName('button')[0].onclick = function (e
       ) {
10   console.log(e.target);       // <button>
11   console.log(e.currentTarget); // <button>
12 });
```

這個範例中，e 從 <button> 開始往上傳遞，由於 e 是因為點擊了 <button> 而產生的，所以不管傳遞到哪，e.target 都是 <button>；e.currentTarget 則會在傳遞的過程中改變，因此在 的 event listener 中，e.currentTarget 就是 。

若是我們不希望發生在 <button> 上的 event 往上傳遞，則可以使用 stopPropagation() 這個 method：

```
1  // 此範例使用上方的 HTML
2
3  document.getElementsByTagName('div')[0].onclick = function (e) {
4  });
5
6  document.getElementsByTagName('button')[0].onclick = function (e
       ) {
7    e.stopPropagation();
8  });
```

如此一來，event 在執行到 <button> 的 event listener 後就被擋下來了，因此 的 event listener 不會被觸發。

最後，如果你不希望某一個 event 在 browser 中預設的行為發生，則可以使用 preventDefault()；比方有些 form validator library 在 form 的 submit button 的 click event listener 上判斷 form values 是不是有誤 (例如使用者帳號長度超過 20 個字元等)，如果有誤了話，就呼叫 e.preventDefault() 以避免 form 被 submit。

有一點必須清楚的是，event 不見得都是 async；如果你手動觸發一個 event 了話，那麼所有的事情都是 sync；請看下面的實驗：

```
1  // 此範例使用上方的 HTML
2
3  document.getElementsByTagName('button')[0].onclick = function (e
       ) {
4    console.log("Button was clicked.");
5  };
6
7  // simulate a click event
8  clickEvent = document.createEvent("MouseEvents");
```

```
9  clickEvent.initMouseEvent("click");
10 document.getElementsByTagName('button')[0].dispatchEvent(
       clickEvent);
11
12 console.log("After triggering the callback.");
13
14 // output:
15 // Button was clicked.
16 // After triggering the callback.
```

這裡我們手動模擬一個 click event，從輸出結果可以看出從 `dispatchEvent()` 到觸發 callback 都是同步的[3]；也就是說，**event 本身可能是同步或是非同步的，但是 event 產生後所發生的事情都是同步的** (當然，前提是 callback 本身如果也是同步了話)。

3.2 Promises in jQuery

Borwser 中的另一個 async API 就是 AJAX 了。

在不同的 browser 中產生 ajax 有不同的，在 IE5, IE6 中須透過 `ActiveXObject()` 產生；IE7+, Firefox, Chrome, Safari, Opera 則是使用 `XMLHttpRequest()`；為了省事，接下來我只打算用介紹 jQuery 中的 ajax 使用方法，jQuery 的 ajax API 已經幫我們做好 browser 的判斷了。

在 jQuery 1.4 的時候，ajax API 使用方法如下：

```
1 $.get('/some-api', {
2   success: function () {}
3   , error: function () {}
4   , complete: function () {}
5 });
```

我們可以分別針對 ajax success, error, complete 等 events 設定 callbacks。但是這樣的 API 有點不太方便，因為有時候我們並不打算在送出 ajax request 的時候就設定 callbacks，而是想晚一點再給 callbacks；比方在 Backbone 中 (雖然我們還沒介紹 Backbone，不過請先是想一下這個情境)，一個 `Model` 可以可以透過 `model.fetch()` 利用 ajax 從 server 取得 model 的資料，而一個 `View` 則是負責管理 HTML；所以如果我想先用 `model.fetch()` 取得 model 的最新資料，然後反應在 HTML 上，那麼我就需要在 model 中發送

[3]其實這點有待商榷，因為如同我們先前介紹的，`console.log()` 在某些 browser 中是同步的，某些則是非同步；這裡我們假設 `console.log()` 是同步的。

ajax、之後才在 view 中設定用來改變 HTML 的 callbacks。這個需求無法在 jQuery 1.5 以前達到，但 1.5 之後就可以；請看以下範例：

```
var promise = $.get('/some-api');
promise.success(function () {});
promise.error(function () {});
promise.complete(function () {});
```

也就是說，產生 ajax 之後，它會給你一個「承諾」，承諾你當 ajax 完成後就會執行你的 callbacks；你可以在任何時候在 `Promise` 上設定 callbacks，就算是在 ajax 已經完成之後也行 (此時設定的 callbacks 會馬上被執行)。因此考慮上方提到的 Backbone 情境，`model.fetch()` 可以將 ajax 產生的 promise 回傳給 view，view 就可以設定適當的 callbacks 給不同的結果 (success 或 error)。

另外，`Promise` 提供的是 chainable API，所以你也可以這樣寫：

```
var promise = $.get('/some-api');
promise
.success(function () {})
.error(function () {})
.complete(function () {})
```

除了 ajax 外，jQuery animation 也會產生 promise，透過`$el.promise()`便能取得一個 animation promise：

```
$el.fadeOut();
var promise = $el.promise();
promise.done(function () {
  console.log("Animation was done!");
});
```

3.2.1 Making Promises

你也可以自己產生一個 `Promise`，方法就是先產生一個 `Deferred`：

```
var deferred = new $.Deferred();
deferred
.done(function () {})
.fail(function () {})
.always(function () {})
```

`Deferred` 跟 `Promise` 的差別就在於你可以親自「解決」一個 `Defferred`：

```
var deferred = new $.Deferred();
deferred.resolve(); // trigger the "done" and "always" callback
deferred.reject();  // trigger the "fail" and "always" callback
```

如果你是做出承諾的一方，你就可以透過 `Deferred` 產生一個 `Promise` 給向你要求一個承諾的人；之後等你完成任務後，就可以透過 `resolve()` 或是 `reject()` 完成這個承諾：

```
var getClickPromise = (function () {
  var deferred = new $.Deferred();
  $('#some-button').click(function () {
    deferred.resolve();
  });

  return function () {
    deferred.promise();
  };
})();

var promise = getClickPromise();
promise.done(function () {
  console.log("Button was clicked!");
});
```

透過 `promise()` 我們可以取得一個 deferred 的 promise，一個 deferred 只會有一個 promise，所以不管呼叫幾次得到的都是同一個 promise。如此一來，點擊 `#some-button` 後就會輸出`"Button was clicked!"`，不過只會輸出一次，因為一個 promise 只會被 resolve 或是 reject 一次。

3.2.2 Combining Promises

你可以用 `$.when()` 合併 promises 和 deferreds，它會回傳一個合併後的 promise 給你：

```
$.when($.get('some-api'), $.get('another-api'))
.done(function () {
  // do something
});
```

如果你直接傳一個 deferred 給 `$.when()`，那它會回傳該 deferred 的 promise 給你：

```
var deferred = new $.Deferred();
var p1 = deferred.promise();
```

```
var p2 = $.when(deferred);
console.log(p1 === p2); // true
```

3.3 I/O in Node.js

在 Node.js 中，大部分的 I/O 都有提供 async 和 sync 兩種 API，前者使用 callback pattern，每一個 I/O function 的最後一個 parmater 是一個 callback，當 I/O 執行結束後 callback 就會被觸發。以 `fs` 這個 module 為例：

```
fs = require('fs'); // load the module

// sync API
var data = fs.readFileSync('/etc/passwd');
console.log(data);

// async API
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

在 Node.js 中，callback 的第一個 parameter 通常是 `err`；以這個例子來說，如果讀檔案的過程中發生錯誤 (比方權限不足或是檔案不存在等)，`err` 就是一個 `Error` instance；如果成功賭取檔案了話，`err` 就是 `undefined`。

3.4 Exceptions in Callbacks

看完了 async functions 的用法與範例後，我們來看看如何處理在 async functions 中產生的 exceptions。

在 sync 的環境下，我們可以用 try catch pattern 捕捉 exceptions，不過在 async 的世界中，這是難以辦到的；請看下方範例：

```
try {
  setTimeout(function () {
    throw "Error occurred!";
  });
} catch (e) {
  console.log(e);
```

```
}
```

在這個例子中，`catch` block 永遠不會被執行到，因為 exception 是在 event loop 階段才產生，在此之前，我們的程式已經正常結束了，所以永遠 `catch` 不到 exception。所以，**在 async function 外面包一層 try catch 是行不通的**，這也就是為什麼 Node.js 中，async function 的 callback 的第一個參數都是 `err`，不然了話我們 `catch` 不到它。

以上一節的 `fs.readFile()` 為例，讀取檔案時若是發生錯誤，我們可以透過 `err` 判斷錯誤為何，並決定下一個動作；我們也可以像範例中那樣，`throw err`，此時 Node.js process 就會 abort。

順帶一提，為了避免 process 因為 uncaught expection 結束，我們可以設定 `uncaughtException` 的 callback：

```
process.on('uncaughtException', function (err) {
  console.log('Caught exception: ' + err);
});
```

如此一來，任何的 uncaught exception 都會交給這個 callback 處理。

不過，除此之外，async function 還會造成 debug 上的困擾；在 sync function 中產生的 exception，透過 stack trace 可以找到 exception 發生前的經過，包括執行哪一行哪一個 function 等；但是 async function 中產生的 exception 只能 trace 到 exception 發生後的經過，因為之前的 stack 在進入 event loop 階段之前就結束了。請看下方 Node.js 中的範例：

```
process.on('uncaughtException', function (err) {
  console.log(err.stack);
});

var a = function () {
  throw new Error('Error occurred');
};
var b = function () {
  a();
};
var c = function () {
  b();
};

c();

// output:
// Error: Error occurred
```

```
19 //     at a (/sync-exception.js:6:8)
20 //     at b (/sync-exception.js:9:2)
21 //     at c (/sync-exception.js:12:2)
22 //     at Object.<anonymous> (/sync-exception.js:15:1)
23 //     ...
```

Listing 3.1: sync-exception.js

```
1  process.on('uncaughtException', function (err) {
2    console.log(err.stack);
3  });
4
5  var a = function () {
6    throw new Error('Error occurred');
7  };
8  var b = function () {
9    setTimeout(function () {
10     a();
11   });
12 };
13 var c = function () {
14   b();
15 };
16
17 c();
18
19 // output:
20 // Error: Error occurred
21 //     at a (/async-exception.js:6:8)
22 //     at Object.b [as _onTimeout] (/async-exception.js:10:3)
23 //     at Timer.list.ontimeout (timers.js:101:19)
```

Listing 3.2: async-exception.js

上方 async 的範例中，由於 `a()` 是在 timeout event 發生後才執行的，所以 stack trace 才 trace 的到。

3.5 Async Flow Control

Callback pattern 讓我們可以設定一個 callback 給一個 async function，但是當愈來愈多 async functions 串在一起的時候，就很容易寫成金字塔狀的程式碼；比方，我們要先判斷一個檔案存不存在，存在了話就讀取該檔，最後則是寫入修改後的內容：

```
fs = require('fs');

fs.stat('some-file.txt', function (err, stats) {
  if (err) throw err;
  // file exists
  fs.readFile('some-file.txt', function (err, data) {
    if (err) throw err;

    // do something on the data
    var newData = data += 'Hi, there!\n';

    fs.writeFile('some-file.txt', newData, function (err) {
      if (err) throw err;
    });
  });
});
```

因此，我們就需要用到 async flow control libraries；我們之前已經看過 promise 和 `$.when()` 等 flow control 的技巧了，雖然類似簡單的 flow control 在前台的許多情況下已經很夠用，不過在後台我們常需要更複雜的 flow control API；Node.js 本身並沒有提供 flow control 的 module，因此我們需要用到第三方的 libraries (隨然這些 libraries 有些也可以在 browser 中使用，不過以下我舉的範例都是在 Node.js 中執行)。

先看一個最 lightweight 的 **Step.js**：

```
fs = require('fs');
step = require('step');

step(
  function () {
    fs.stat('some-file.txt', this);
  }
  , function (err) {
    if (err) throw err;
    // file exists
    fs.readFile('some-file.txt', this);
  }
  , function (err, data) {
    if (err) throw err;
    // do something on the data
    var newData = data += 'Hi, there!\n';
    fs.writeFile('some-file.txt', newData, this);
  }
  , function (err) {
```

```
    if (err) throw err;
  }
);
```

我們把這些 tasks 交給 `Step` 去做，只要把每個 task 的 callback 都設成 `this`，`Step` 就會幫我們依序執行 (為了簡化程式碼，我一慮把 `err` throw 出去)。

稍微複雜一點，我們想先「平行」地讀取兩個檔案，讀完之後再寫到另一個檔案中：

```
fs = require('fs');
step = require('step');

step(
  function () {
    fs.stat('some-file.txt', this.parallel());
    fs.stat('another-file.txt', this.parallel());
  }
  , function (err) {
    if (err) throw err;
    // files exist
    fs.readFile('some-file.txt', this.parallel());
    fs.readFile('another-file.txt', this.parallel());
  }
  , function (err, someData, anotherData) {
    if (err) throw err;
    // do something on the data
    fs.writeFile('combined.txt', someData + anotherData, this);
  }
  , function (err) {
    if (err) throw err;
  }
);
```

如果對以上範例感到疑惑了話，不妨去 Step.js 的 GitHub repository 看看他們的文件和 source code；Step.js 的 source code 才 150 行左右，除了可以讓你了解 async control 是怎麼辦到的以外，也能讓你對 scope 的運用更加熟悉，蠻值得一看的！

如果嫌 Step.js 的 API 過於陽春，那 **Async.js** 就是最好的選擇。Async.js 提供的 API 包括：

- Collection:
  `forEach`, `map`, `filter`, `reject`, `reduce`, `detect`, `sortBy`, `some`,

`every`, `concat`

- Control Flow:
  `series`, `parallel`, `whilst`, `until`, `waterfall`, `queue`, `auto`, `iterator`, `apply`, `nextTick`
- Utils:
  `memoize`, `unmemoize`, `log`, `dir`, `noConflict`

怎麼樣，滿意了吧？說真的，我真正用過的也沒幾個；我並不打算針對每個 API 加以介紹，常用的 `series()`, `waterfall()`, `parallel()` 跟 Step.js 的功能差不多；就用 `waterfall()` 和 `map()` 舉個例子吧，我想要讀取一個資料夾裡所有的檔案，並印出檔名和大小：

```
fs = require('fs');
async = require('async');

async.waterfall([
  function (callback) {
    fs.readdir('.', callback);
  }
  , function (files, callback) {
    async.map(files, fs.stat, function (err, results) {
      callback(err, files, results);
    });
  }
]
, function (err, files, results) {
  var i;
  for (i = 0; i < results.length; i++) {
    console.log(files[i], '-', results[i].size, 'bytes');
  }
});

// sample output:
// another-file.txt - 11 bytes
// combined.txt - 22 bytes
// some-file.txt - 11 bytes
```

`waterfall()` 的定義是 `waterfall(tasks, [callback])`，第一個參數是 task array，第二個則是 optional callback；與 Step.js 不同的是，當 error 發生時，error 並不會被傳遞到下一個 task，而是直接跳到最後的 callback，並把 error 當成它的第一個參數。