ECMAScript 6

bigsai95

Published with GitBook

Table of Contents

Ecmascript 6 讀書心得

參考資料

- https://developer.mozilla.org/en-US/docs/Web/JavaScript
- http://es6.ruanyifeng.com/
- http://es6-features.org/
- https://medium.com/ecmascript-2015
- http://www.codedata.com.tw/tag/ecmascript-6/
- http://www.ruanyifeng.com/blog/2012/10/javascript_module.html
- http://exploringjs.com/es6/ch_modules.html
- http://www.2ality.com/2014/09/es6-modules-final.html
- http://weizhifeng.net/node-js-exports-vs-module-exports.html
- http://www.openfoundry.org/tw/tech-column/8678-beginning-requirejs
- http://ilucas.me/2015/08/25/es6-unicode-regex/
- https://blog.othree.net/log/2015/04/05/loader/

Arrow(箭頭)

1. 讓程式 function 擁有更簡短的語法
2. 自動將 this 變數綁定到其定義時所在的物件
3. anonymous function

語法

```
s => n
```

等於下列

```
(function (s) {
  return n;
});
```

Example 1

```
//ES6
var a = [ "a", "b", "c" ];
var a1 = a.map( (s,n) => n + "=>" + s );

console.log(a1); //["0=>a", "1=>b", "2=>c"]
```

```
//舊
var a = [ "a", "b", "c" ];
var a2 = a.map(
     function(s, n) {
          return n + "=>" + s ;
     }
);

console.log(a2); //["0=>a", "1=>b", "2=>c"]
```

Example 2

```
//ES6
function Person(){
  this.age = 0;
  setInterval(() => {
    this.age++;
    console.log(this.age);
  }, 1000);
}
var p = new Person();
```

```
//舊
function Person() {
  var self = this;
  self.age = 0;
  setInterval(function () {
     self.age++;
     console.log(self.age);
  }, 1000);
}
```

Classes

1. 精簡的classes語法
2. 類別義定 class expressions 和 class declarations

語法

```
class name [extends] {
    //body
}
```

Class Declarations

```
class Person {
    constructor(name, age) {
      this.name = name;
      this.age = age;
    }
}
```

Class Expressions

```
// unnamed
var P1 = class {
    constructor(name, age) {
      this.name = name;
      this.age = age;
    }
};

// named
var P2 = class Person {
    constructor(name, age) {
      this.name = name;
      this.age = age;
    }
};
```

Example 1

```
//ES6
class Person {
    constructor(name, age) {
      this.name = name;
      this.age = age;
    }
    say () {
        return 'hello';
    }
    toString() {
        return '[' + this.name + ',' + this.age + ']';
    }
}

let p = new Person('Justin', 30);
console.log(p.toString()); //[Justin,30]
```

```
//舊
function toString() {
    return '[' + this.name + ',' + this.age + ']';
}

function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.say = function() {
    return 'hello';
};

Person.prototype.toString = function() {
    return '[' + this.name + ',' + this.age + ']';
};

var p = new Person('Justin', 30);
console.log(p.toString()); //[Justin,30]
```

Example 2 (classing with extends)

```
//ES6
class Student extends Person {
    toString() {
        return 'Student' + " " + this.name + " " + this.age;
    }
}

let p1 = new Student('super', 12);
console.log(p1.toString()); //Student super 12
```

```
//舊
function Student(name, age){
  Person.call(this, name, age);
}
Student.prototype = new Person();
Student.prototype.toString = function(){
    return 'Student' + " " + this.name + " " + this.age;
}

var p1 = new Student('super', 12);
console.log(p1.toString()); //Student super 12
```

Enhanced Object Literals

ES6允許直接寫入變量和函數，作為對象的屬性和方法。這樣的書寫更加簡潔。

語法

```
// Shorthand property names (ES6)
var a = "foo", b = 42, c = {};
var o = { a, b, c };

// Shorthand method names (ES6)
var o = {
  property([parameters]) {},
  get property() {},
  set property(value) {},
  * generator() {}
};

// Computed property names (ES6)
var prop = "foo";
var o = {
  [prop]: "hey",
  ["b" + "ar"]: "there",
};
```

Example 1 (Property definitions)

```
//ES6
var a = "foo", b = 42, c = {};
var o = { a, b, c };
```

```
//舊
var a = "foo", b = 42, c = {};
var o = { a: a, b: b, c: c };
```

Example 2 (Method definitions)

```
//ES6
var BigLoco = {
  locoName: 'Gordon',
  get name() { return this.locoName; },
  set name(n) { this.locoName = n }
};

console.log(BigLoco.name); // 'Gordon'
```

```
//舊
```

```
var BigLoco = Object.defineProperties({
  locoName: 'Gordon'
  },
  {
    name: {
        get: function get() {
            return this.locoName;
        },
        set: function set(n) {
            this.locoName = n;
        },
        configurable: true,
        enumerable: true
    }
});

console.log(BigLoco.name); // 'Gordon'
```

Example 3 (Computed property names)

```
//ES6
function type() { return 1; }

obj = {
    foo: "bar",
    [ "prop_" + foo() ]: 42
};

console.log(obj); //{foo: "bar", prop_3: 42}
```

```
//舊
function type() { return 1; }
obj = {
    foo: "bar"
};
obj[ "prop_" + foo() ] = 42;

console.log(obj); //{foo: "bar", prop_3: 42}
```

Template Strings

語法

```
`string text`

`string text line 1
 string text line 2`

`string text ${expression} string text`

tag `string text ${expression} string text`
```

Example 1 (Multi-line strings)

```
//ES6
console.log(`string text line 1
string text line 2`);
```

```
//舊
console.log("string text line 1 \nstring text line 2");
```

Example 2 (Expression interpolation)

```
//ES6
var a = 5;
var b = 10;
console.log(`Fifteen is ${a + b} and\nnot ${2 * a + b}.`);
```

```
//舊
var a = 5;
var b = 10;
console.log("Fifteen is " + (a + b) + " and\nnot " + (2 * a + b) + ".");
```

Example 3 (Tagged template strings)

```
//ES6
var a = 5;
var b = 10;

function tag(strings, ...values) {
  console.log(strings[0]); // "Hello "
  console.log(strings[1]); // " world "
  console.log(values[0]);  // 15
  console.log(values[1]);  // 50
  return "Bazinga!";
}

tag`Hello ${ a + b } world ${ a * b}`;
```

```
//舊
function _taggedTemplateLiteral(strings, raw) {
    return Object.freeze(Object.defineProperties(
        strings, {
                raw: { value: Object.freeze(raw) }
            }
        )
    );
}

var a = 5;
var b = 10;

function tag(strings) {
  console.log(strings[0]); // "Hello "
  console.log(strings[1]); // " world "
  console.log(arguments[1]); // 15
  console.log(arguments[2]); // 50

  return "Bazinga!";
}

tag(_taggedTemplateLiteral(["Hello ", " world ", ""], ["Hello ", " world ", ""]), a + b,
```

Example 4 (Raw strings)

```
//ES6
function tag(strings, ...values) {
  console.log(strings.raw[0]);
  // "string text line 1 \\n string text line 2"
}

tag`string text line 1 \n string text line 2`;
```

```
```

Destructuring

ES6允許按照一定模式，從數組和對像中提取值，對變量進行賦值，這被稱為解構（Destructuring）。

語法

```
[a, b] = [1, 2]
[a, b, ...rest] = [1, 2, 3, 4, 5]
{a, b} = {a:1, b:2}
{a, b, ...rest} = {a:1, b:2, c:3, d:4}  //ES7
```

Example 1 (Swap)

```
//ES6
var a = 1;
var b = 3;

[a, b] = [b, a];
```

```
//舊
var a = 1;
var b = 3;

var _ref = [b, a];
a = _ref[0];
b = _ref[1];

console.log(a, b); //3 1

//or
var a = 1;
var b = 3;

var tmp = a;
    a = b;
    b = tmp;

console.log(a, b); //3 1
```

Example 2 (Multiple-value returns)

```
//ES6
function f() {
  return [1, 2, 3];
}
var a, b;
[a, ,b] = f();
console.log("A is " + a + " B is " + b); //A is 1 B is 3
```

```
var a = f();
console.log("A is " + a); //A is 1,2,3
```

Example 3 (regular expression match)

```
//ES6
var url = "https://developer.mozilla.org/en-US/Web/JavaScript";

var parsedURL = /^(\w+)\:\/\/([^\/]+)\/(.*)$/.exec(url);
var [, protocol, fullhost, fullpath] = parsedURL;

console.log(protocol); // logs "https:"
```

Example 5 (Function argument defaults)

```
//ES6
function drawES6Chart({size = 'big', cords = { x: 0, y: 0 }, radius = 25} = {})
{
  console.log(size, cords, radius);
  // do some chart drawing
}

drawES6Chart({
  cords: { x: 18, y: 30 },
  radius: 30
});
```

```
//舊
function drawES5Chart(options) {
  options = options === undefined ? {} : options;
  var size = options.size === undefined ? 'big' : options.size;
  var cords = options.cords === undefined ? { x: 0, y: 0 } : options.cords;
  var radius = options.radius === undefined ? 25 : options.radius;
  console.log(size, cords, radius);
  // now finally do some chart drawing
}

drawES5Chart({
  cords: { x: 18, y: 30 },
  radius: 30
});
```

Example 6 (For of iteration and destructuring)

```
//ES6
var 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 (var {name: n, family: { father: f } } of people) {
  console.log("Name: " + n + ", Father: " + f);
}
//Name: Mike Smith, Father: Harry Smith
//VM303:32 Name: Tom Jones, Father: Richard Jones
```

Default + Rest + Spread

Example 1 (Default Parameter Values)

```
//ES6
function f (x, y = 10) {
    return x + y
}
console.log(f(1) === 11) //true
console.log(f(5) === 11) //false
```

```
//舊
function f (x, y) {
    if (y === undefined)
        y = 10;

    return x + y;
};

or

function f (x, y) {
    y = y || 10;

    return x + y;
};

console.log(f(1) === 11) //true
console.log(f(5) === 11) //false
```

Example 2 (Rest Parameter)

```
//ES6
function f (x, y, ...a) {
    return (x, y) * a.length;
}
console.log(f(1, 2, "hi", true, 7, 4)); // 12

function sum (...numbers) {
      var result = 0;
    numbers.forEach(function (number) {
         result += number;
     });

    return result;
}
console.log(sum(1, 2, 3)); // 6
```

```
//舊
function f (x, y) {
    var a = Array.prototype.slice.call(arguments, 2);
    return (x + y) * a.length;
}
console.log(f(1, 2, "hi", true, 7, 4)); // 12

function sum () {
      var result = 0;
    var numbers = Array.prototype.slice.call(arguments);
    numbers.forEach(function (number) {
         result += number;
     });

    return result;
}
console.log(sum(1, 2, 3)); // 6
```

Example 3 (Spread Operator)

```
//ES6 (未測試過)
function sum(a, b, c) {
  return a + b + c;
}
var args = [1, 2, 3];
console.log(sum(…args)); // 6
```

```
//舊
function sum(a, b, c) {
  return a + b + c;
}
var args = [1, 2, 3];
console.log(sum.apply(undefined, args)); // 6
```

Let + Const

1. ES6新增了let命令，用來聲明變數。它的用法類似於var，但是所聲明的變數，只在let命令所在的代碼區塊內有效。
2. let不允許在相同作用域內，重複聲明同一個變數。
3. const也用來聲明變數，但是聲明的是常數。一旦聲明，常數的值就不能改變。
4. const的作用域與let命令相同：只在聲明所在的區塊作用域內有效。
5. const聲明的常數，也與let一樣不可重複聲明。

語法

```
let var1 [= value1] [, var2 [= value2]] [, ..., varN [= valueN]];
```

```
const name1 = value1 [, name2 = value2 [, ... [, nameN = valueN]]];
```

Example 1 (let + const)

```
//ES6
let a = 1;
const b = 2;
```

```
//舊
var a = 1
var b = 2;
```

Example 2

```
//ES6
for(let i = 0; i < 5; i++){}

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

```
//舊
for(var i = 0; i < 5; i++){}

console.log(i); // 5
```

Example 3

```
//ES6 重複同一個變數
// 报错
```

```
function () {
  let a = 10;
  var a = 1;
}

// 报错
function () {
  let a = 10;
  let a = 1;
}
```

Example 4

```
const PI = 3.1415;
PI // 3.1415

PI = 3;
PI // 3.1415

const PI = 3.1;
PI // 3.1415
```

Example 5

```
const foo = {};
foo.prop = 123;

foo.prop
// 123

foo = {} // 不起作用

const a = [];
a.push("Hello"); // 可执行
a.length = 0;    // 可执行
a = ["Dave"];    // 报错
```

Iterators + For..Of

Iterator）就是這樣一種機制。它是一種接口，為各種不同的數據結構提供統一的訪問機制。任何數據結構只要部署Iterator接口，就可以完成foreach操作（即依次處理該數據結構的所有成員）。

1. 為各種數據結構，提供一個統一的、簡便的訪問接口
2. 使得數據結構的成員能夠按某種次序排列
3. ES6創造了一種新的forearch循環，Iterator為主要接口

```
//ES6
let fibonacci = {
    [Symbol.iterator]() {
        let pre = 0, cur = 1
        return {
           next () {
               [ pre, cur ] = [ cur, pre + cur ]
               return { done: false, value: cur }
           }
        }
    }
}

for (let n of fibonacci) {
    if (n > 1000)
        break
    console.log(n)
}
```

```
//舊
var fibonacci = {
    next: ((function () {
        var pre = 0, cur = 1;
        return function () {
            tmp = pre;
            pre = cur;
            cur += tmp;
            return cur;
        };
    })();
};

var n;
for (;;) {
    n = fibonacci.next();
    if (n > 1000)
        break;
    console.log(n);
}
```

Modules

JavaScript 一直以來都沒 Module，在ES6以前最主要有 CommonJS 和 AMD 二種。

CommonJS Modules： Node.js就是遵照 CommonJS 的規範(參考)

- 主要使用在 server
- 同步

Asynchronous Module Definition (AMD): 目前最常見的 AMD 實作就是 require.js (參考)

- 主要使用在 browsers
- 非同步

補充(module.exports vs exports)

ES6 Module System (瀏覽器目前還不支持ES6)

主要是由 export 和 import 組成。 export 用於模組對外接口， import 用載入模組功能。

語法

```
Example 1:
export name1, name2, ..., nameN;

Example 2:
export *;
```

```
import name from "module-name";
import { member } from "module-name";
import { member as alias } from "module-name";
import { member1 , member2 } from "module-name";
import { member1 , member2 as alias2 , [...] } from "module-name";
import name , { member [ , [...] ] } from "module-name";
import "module-name";
```

Example 1

```
//ES6
// foobar.js
var foo = 'foo', bar = 'bar';

export { foo, bar };

or

// foobar.js
export var foo = 'foo';
export var bar = 'bar';
```

```
//舊
var foo = 'foo';
exports.foo = foo;

var bar = 'bar';
exports.bar = bar;
```

Example 2

```
//ES6
//  lib/math.js
export function sum (x, y) { return x + y }
export var pi = 2

// app1.js
import * as math from "lib/math";
console.log(math.sum(2, 3));  // 5

// app2.js
import { sum, pi } from "lib/math";
console.log(sum(pi, pi)); // 4
```

```
//舊
//  lib/math.js
LibMath = {};
LibMath.sum = function (x, y) { return x + y };
LibMath.pi = 2;

//  app1.js
var sum = LibMath.sum;
console.log(sum(2, 3));

//  app2.js
var sum = LibMath.sum, pi = LibMath.pi;
console.log(sum(pi, pi)); // 4
```

Example 3 加載的變量名或函數名錯誤，則無法加載

```
//ES6
//test1.js
export var hello = 'world';

//app3.js
import { hello } from 'test1.js';
console.log(hello); // -> world

import { hello1 } from 'test1.js';
console.log(hello1); // -> undefined
```

Example 4 (export default)

使用import命令的時候，用戶需要知道所要加載的變量名或函數名，否則無法加載。為了給用戶提供方便，讓他們不用閱讀文檔就能加載模塊，就要用到export default命令，為模塊指定默認輸出。

- export default 命令用於指定模塊的默認輸出。
- export deault 命令只能使用一次，一個模塊只能有一個輸出。
- import 命令後面才不用加大括號，因為只可能對應一個方法。

匿名函數指定任意名字

```
//ES6
//------ myFunc.js ------
export default function () { console . log ( 'foo' ) ;  } // no semicolon!

//------ main1.js ------
import myFunc from 'myFunc';
myFunc();
```

非匿名函數，加載的時候，視同匿名函數加載

```
//ES6
//------ myFunc.js ------
function  foo ( )  {
  console . log ( 'foo' ) ;
}

export default foo ;

//------ main1.js ------
import myFunc from 'myFunc';
myFunc();
```

類別

```
//ES6
//------ MyClass.js ------
export default class { ··· } // no semicolon!

//------ main2.js ------
import MyClass from 'MyClass';
let inst = new MyClass();
```

Modules Loaders (模組加載)

Loader 是 ECMAScript 定義要來處理 module import/export 等等事情的底層介面。

註：Addy Osmani 有建立一個 Loader 的 polyfill 給 ES5 環境使用 Loader API，就叫做ES6 Module Loader Polyfill

註：使用 ES6 Module Loader Polyfill 測試網址

Module loaders support:

- Dynamic loading (動態加載)
- State isolation (狀態隔離)
- Global namespace isolation (全域命名空間隔離)
- Compilation hooks (編譯鉤子)
- Nested virtualization (嵌套虛擬化) - 註: 在模組內調用模組

預設的模組加載器是可配置的，也可以建構新的加載器，對在隔離和受限上下文中的代碼進行求值和加載。

```
// Dynamic loading – 'System' is default loader
System.import('lib/math').then(function(m) {
  alert("2π = " + m.sum(m.pi, m.pi));
});
```

```
// Create execution sandboxes – new Loaders
var loader = new Loader({
  global: fixup(window) // replace 'console.log'
});
loader.eval("console.log('hello world!');");
```

```
// Directly manipulate module cache
System.get('jquery');
System.set('jquery', Module({$: $})); // WARNING: not yet finalized
```

Module (module.exports vs exports)

Node.js 透過 CommonJS 的標準引入 Module 觀念，可以經由 module.exports 或 exports 將函數、變數導出，以 require() 的方式將函數載入使用。

1. exports 是 module.exports 的一個調用。
2. require() 返回的是 module.exports 而不是 exports。
3. module.exports 本身不具備任何屬性和方法，exports 收集到的屬性和方法，都會返回給 module.exports 調用。
4. module.exports 己具備一些屬性和方法，exports 收集來的屬性和方法將會備忽略。

```
console.log(module.exports === exports); //true (module.exports 和 exports 是相同的)
```

例：exports 建立模組(rocker.js)

```
exports.name = function() {
    console.log('My name is Lemmy Kilmister');
};
```

使用

```
var rocker = require('./rocker.js');
rocker.name(); // 'My name is Lemmy Kilmister'
```

例 1：

```
//rocker.js
module.exports = 'ROCK IT!';
exports.name = function() {
    console.log('My name is Lemmy Kilmister');
};
```

使用

```
var rocker = require('./rocker.js');
rocker.name(); // TypeError: Object ROCK IT! has no method 'name'
```

例 2：

```
//rocker.js
module.exports = 'LOL';
module.exports.age = 68;
```

```
exports.name = 'Lemmy Kilmister';
```

使用

```
var rocker = require('./rocker.js');
console.log(rocker); // LOL
```

例 3：

```
//rocker.js
module.exports.age = 68;
exports.name = 'Lemmy Kilmister';
```

使用

```
var rocker = require('./rocker.js');
console.log('%s is %s', rocker.name, rocker.age); // Lemmy Kilmister is 68
```

例：模組是一個Class (rocker.js)

```
module.exports = function(name, age) {
    this.name = name;
    this.age = age;
    this.about = function() {
        console.log(this.name +' is '+ this.age +' years old');
    };
};
```

使用

```
var Rocker = require('./rocker.js');
var r = new Rocker('Ozzy', 62);
r.about(); // Ozzy is 62 years old
```

例：模組是一個array (rocker.js)

```
module.exports = ['Lemmy', 'Ozzy', 'Ronnie', 'Steven', 'Mick'];
```

使用

```
var rocker = require('./rocker.js');
console.log('Rockin in heaven: ' + rocker[2]); //Rockin in heaven: Ronnie
```

Map + Set + WeakMap + WeakSet

對於常用算法來說高效的數據結構，而 WeakSet、WeakMap 提供了防止內存洩露的key對數據結構，相比來說更安全。

Set：

ES6提供的新數據結構， 裡面不存放重複的元素。

語法

```
new Set(iterable);
```

參數 (iterable)

- iteralbe是Array或其他可枚舉的對象，其每個元素是key、value的2元數組。

屬性和方法：

- Set.prototype.size
- Set.prototype.add(v)
- Set.prototype.delete(v)
- Set.prototype.has(v)
- Set.prototype.clear()
- Set.prototype.entries()
- Set.prototype.forEach(callback, thisArg)
- Set.prototype.keys()
- Set.prototype.values()

Example

```
//ES6
var mySet = new Set();
mySet.add(1).add(2).add(2);
// 注意2被加入了兩次

mySet.size // 2

mySet.has(1) // true
mySet.has(2) // true
mySet.has(3) // false

mySet.delete(2);
mySet.has(2) // false
```

Array.from方法可以將Set結構轉成陣列

```
var s1 = new Set();
s1.add(1);
s1.add(2);

console.log(s1); //Set {1, 2}
// toArray
var a1 = Array.from(s1);
console.log(a1); //[1, 2]
```

Set結構的實例有四個遍歷方法，可以用於遍歷成員。

```
var s1 = new Set();
s1.add(1);
s1.add(2);
s1.add(3);

// 輸出1, 2, 3
for (var i of s1) {
    console.log(i);
}

s1.delete(2);

for  (  let item of s1 . keys ( )  ) {
  console . log ( item ) ; //1 3
}

for  (  let item of s1 . values  ( )  ) {
  console . log ( item ) ; //1 3
}

for  (  let item of s1 . entries ( )  ) {
  console . log ( item ) ; //[1, 1] [3, 3]
}

s1.forEach(function(v) {
    console.log(v); // 1 3
});
```

WeakSet：

WeakSet 構造函數和普通的 Set 相同。

1. WeakSet的成員只能是對象，而不能是其他類型的值
2. 只有add/delete/clear/has三个方法，不能遍歷，没有size属性等

語法

```
new WeakSet(iterable);
```

屬性和方法：

- WeakSet.prototype.add(v)
- WeakSet.prototype.delete(v)
- WeakSet.prototype.has(v)

```
var ws =  new  WeakSet ();
ws.add ( 1 );
// TypeError: Invalid value used in weak set
```

WeakSet是一個構造函數，可以使用new命令，創建WeakSet數據結構。

```
var a =  [ [ 1 , 2 ] ,  [ 3 , 4 ] ];
var ws =  new  WeakSet ( a );
```

WeakSet可以接受一個數據或類似數據的對像作為參數。

```
var ws =  new  WeakSet () ;
var obj =  {};
var foo =  {};

ws.add( window );
ws.add( obj );

ws.has( window ); // true
ws.has( foo );    // false

ws.delete( window );
ws.has( window );    // false
```

WeakSet沒有size屬性，沒有辦法遍歷

```
ws.size // undefined
ws.forEach // undefined

ws.forEach ( function ( item ) { console . log ( 'WeakSet has '  + item ) } )
// TypeError: undefined is not a function
```

Map：

提供傳統意義上的Map。支持任意對像作為key。

語法

```
new Map(iterable);
```

屬性和方法：

- Map.prototype.size
- Map.prototype.get(k)
- Map.prototype.set(k,v)
- Map.prototype.has(k)
- Map.prototype.clear()
- Map.prototype.entries()
- Map.prototype.forEach(callback, thisArg)
- Map.prototype.keys()
- Map.prototype.values()

Example

```
//ES6
var m = new Map ();
var o = { p :  "Hello World" };

m.set( o ,  "content" );
m.get( o ); // "content"

m.has( o ); // true
m.delete( o ); // true
m.has( o ); // false
```

Map也可以接受一個數組作為參數。

```
var map = new Map( [  [ "name" ,  "張三" ] ,  [ "title" ,  "Author" ] ] );

map.size; // 2
map.has( "name" ); // true
map.get( "name" ); // "張三"
map.has( "title" ); // true
map.get( "title" ); // "Author"
```

set和get方法，表面是針對同一key，但實際上這是兩個值，記憶體位置是不一樣的，因此get方法無法讀取key，返回undefined。

```
var map =  new  Map ();
map.set ( [ 'a' ] ,  555 ) ;
map.get ( [ 'a' ] ); // undefined

var t = ['a'];
map.set ( t ,  555 ) ;
map.get ( t );  //555
```

Map原生提供三個方法

```
let map =  new  Map ( [
  [ 'F' ,  'no' ] ,
  [ 'T' ,   'yes' ] ,
] ) ;

for ( let key of map.keys() )  {
  console.log ( key ) ;
}
// "F"
// "T"

for ( let value of map.values() )  {
  console.log ( value ) ;
}
// "no"
// "yes"

for  ( let item of map.entries() )  {
  console.log ( item [ 0 ] , item [ 1 ] ) ;
}
// "F" "no"
// "T" "yes"

// 或者
for  ( let  [ key , value ] of map.entries() )  {
  console.log ( key , value ) ;
}

// 等同於使用map.entries()
for  ( let  [ key , value ] of map )  {
  console.log ( key , value ) ;
}

map.forEach(function(value, key, map) {
    console.log(key , value);
});
```

WeakMap：

WeakMap結構與Map結構基本類似，唯一的區別是它只接受對像作為key（null除外），不接受原始類型的值作為key，而且key所指向的對象，不計入垃圾回收機制。

語法

```
new WeakMap(iterable);
```

屬性和方法：

- WeakMap.prototype.clear()
- WeakMap.prototype.delete(k)
- WeakMap.prototype.get(k)
- WeakMap.prototype.has(k)

- WeakMap.prototype.set(k,v)

```
var wm =  new  WeakMap ( ) ;

wm.size;
// undefined

wm.forEach;
// undefined
```

```
var wm =  new  WeakMap ( ) ;
var element = {};

wm.set( element ,  "Original" ) ;
wm.get( element ); // "Original"

wm.set( 'el' ,  "Original" ) ;
wm.get( 'el' ); // Invalid value used as weak map key
```

WeakMap應用的典型場合就是DOM節點作為key。下面是一個例子

```
let myElement = document . getElementById ( 'logo' ) ;
let myWeakmap =  new  WeakMap ( ) ;

myWeakmap . set ( myElement ,  { timesClicked :  0 } ) ;

myElement . addEventListener ( 'click' ,  function ( )  {
  let logoData = myWeakmap . get ( myElement ) ;
  logoData . timesClicked ++ ;
  myWeakmap . set ( myElement , logoData ) ;
} ,  false ) ;
```

Generators

ES6 的Generator函數，需要用* modifier標注

Example

```
function * gen() {
  console.log('start');
  yield "called";
}

var g = gen();
//nothing happened

var a = g.next();
//顯示start
console.log(a.value);
//顯示called

console.log(a.done);
//顯示false

//Generator 只會執行一次yield
var b = g.next();
console.log(b.done);
//顯示true
```

Example - 接收外部傳來的data

```
function * gen() {
  console.log('start');
  var got = yield 'called';
  console.log(got);
}

var g = gen();
var a = g.next();
//顯示start
var b = g.next('hello generator');
//顯示hello generator

g.throw('got an error.');
//拋出一個例外，錯誤訊息是 'got an error.'
```

Unicode

ES6支援 Unicode 字串和正則表達式中的擴展。

- u 開啟各種 Unicode 相關特性

語法

```
regex.unicode
```

在正則式中使用u flag，將開啟 ES6 中的 Unicode 轉義模式 \u{...}.

```
//ES6
// Note: `a` is U+0061 LATIN SMALL LETTER A, a BMP symbol.
console.log(/\u{61}/u.test('a'));
// → true

// Note: `𝌆` is U+1D306 TETRAGRAM FOR CENTRE, an astral symbol.
console.log(/\u{1D306}/u.test('𝌆'));
// → true
```

Example

```
//ES6
// Note: `𝌆` is U+1D306 TETRAGRAM FOR CENTRE, an astral symbol.
var string = 'a𝌆b';

console.log(/a.b/.test(string));
// → false

console.log(/a.b/u.test(string));
// → true

var match = string.match(/a(.)b/u);
console.log(match[1]);
// → '𝌆'
```

```
//舊
var string = 'a𝌆b';

console.log(/a.b/.test(string));
// → false

console.log(/a(?:[\0-\t\x0B\f\x0E-\u2027\u202A-\uD7FF\uE000-\uFFFF]|[\uD800-\uDBFF][\uDC0
// → true

var match = string.match(/a((?:[\0-\t\x0B\f\x0E-\u2027\u202A-\uD7FF\uE000-\uFFFF]|[\uD800
console.log(match[1]);
// → '≣'
```

Example

```
//ES6
var regex = /^[bcd]$/;
console.log(
  regex.test('a'), // false
  regex.test('b'), // true
  regex.test('c'), // true
  regex.test('d'), // true
  regex.test('e')  // false
);
```

Proxies

ES6原生提供Proxy構造函數，用來生成Proxy實例。

語法

```
var p = new Proxy(target, handler);
```

```
var proxy =  new  Proxy ( { } ,  {
  get :  function ( target , property )  {
    return  35 ;
  }
} ) ;

proxy.time // 35
proxy.name // 35
proxy.title // 35
```

Symbols

ES5的對象屬性名都是字符串，這容易造成屬性名的衝突。比如，你使用了一個他人提供的對象，但又想為這個對象添加新的方法，新方法的名字就有可能與現有方法產生衝突。如果有一種機制，保證每個屬性的名字都是獨一無二的就好了，這樣就從根本上防止屬性名的衝突。這就是ES6引入Symbol的原因。

ES6引入了一種新的原始數據類型Symbol，表示獨一無二的值。Symbol值通過Symbol函數生成。屬於Symbol類型，就都是獨一無二的，可以保證不會與其他屬性名產生衝突。

注意，Symbol函數前不能使用new命令，否則會報錯。這是因為生成的Symbol是一個原始類型的值，不是對象。也就是說，由於Symbol值不是對象，所以不能添加屬性。基本上，它是一種類似於字符串的數據類型。

語法

```
let s =  Symbol ( ) ;

typeof s
// "symbol"
```

Subclassable Built-ins 可子類化的內建物件

在 ES6 中，內建物件，如Array、Date以及DOM元素可以被子類化。

針對名為Ctor的函數，其對應的物件的構造現在分為兩個階段（這兩個階段都使用虛分派）：

- 調用Ctor[@@create]為物件分配空間，並插入特殊的行為
- 在新實例上調用構造函數來進行初始化