Eloquent JavaScript 2nd Edition 中文版

Baowen

Published with GitBook

Table of Contents

Introduction 0
First Chapter 1
第十章 模組 2

My Awesome Book

This file file serves as your book's preface, a great place to describe your book's content and ideas.

First Chapter

GitBook allows you to organize your book into chapters, each chapter is stored in a separate file like this one.

第十章 模組

一開始我們有一個很簡單的module。1.dayName函數是這個模組的interface，但是names變數不是。2.我們不希望names也佔用一個global variable的位置

```
var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday"];
function dayName(number) {
  return names[number];
}
console.log(dayName(1));
```

所以把data和function都包在一個anonymous函數裡，在js裡，函數裡的變數都是local variable

```
var dayName = function () {

    //模組裡面有一些 data
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thurs
    return function dayName(number) {
        return names[number];
    }
}();
console.log(dayName(1));
```

另一個把code藏起來的方法：這個模組會輸出資訊到console上，但不會提供其他資訊給其他模組使用。把模組包在函數裡是為了防止它所用的變數污染了global scope

```
(function() {
    function square(x) { return x * x };
    var hundred = 100;
    console.log(square(hundred));
})();
```

假設我們現在要再增加一個功能(function)，因為無法return兩個function，所以把他們定義成object的method，然後回傳這個object。

```
var weekDay = function () {
     var names = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thur
     return {
         name: function(number) {return names[number]; },
         number: function(name) {return names.indexOf(name); }
     };
}();
console.log(weekDay.name(weekDay.number("Sunday")));
```

對於大一點的模組而言，把這些要輸出的東西集合起來放入一個物件裡也會逐漸變得不太可行，因為東西太在是多了。所以另一個方法是宣告一個物件，通常叫做exports，然後把所有要輸出的值設成是exports的property或method。例如在下面的方法中，這個模組把interface object當作argument，這會允許外面的code可以直接使用它(不需要在模組裡面宣告)，並把它存在weekDay變數裡。在函數外面的時候，this指的是global scope object。

```
(function (exports) {
     var names = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thur
     exports.name = function(number) {
         return names[number];
     };
     exports.number = function(name) {
         return names.indexOf(name);
     };
})(this.weekDay = {});
console.log(weekDay.name(weekDay.number("Saturday")));
```

上述的方法還是會有問題，例如當多個模組同時用一樣的變數名稱(weekDay)，或是你想同時載入兩個不同版本的同個模組時，這些會讓變數產生衝突。用一點方法，我們可以在不經過global scope的情況下，讓一個模組可以直接access另一個模組的interface object。我們的目標是一個require函數，當被給定一個module

name的時候，可以載入那個module的file(從硬碟或網路上)，並回傳該模組的interface value。這麼做除了解決變數衝突的問題之外，還可以讓軟體所有的dependencies更加明確(因為你會明確的require其他模組)。require需要兩個功能：

1. readFile (以字串的方式回傳檔案內容)
2. 我們必須可以把這個字串當成javascript程式碼來執行

把data當程式碼執行

把data(程式碼的文字列）當成程式碼執行有很多方式，最明顯的就是eval，eval會在目前的scope下執行程式碼文字列。

```
function evalAndReturnX(code) {
    eval(code);
    return x;
}
console.log(evalAndReturnX("var x = 2"));
// -> 2
```

另外一個比較好的方式利用 Function 建構函數(constructor)，他會用到兩個arguments：arguments名稱以及函數的body

```
var plusOne = new Function("n", "return n + 1; ");
console.log(plusOne(4));
//-> 5
```

這就是我們的模組需要的，我們可以把模組的程式碼包在一個函數裡，而這個函數的scope就變成模組的scope。

Require

`Require` 最低限度需要下列的內容：

```
function require(name) {
   var code = new Function("exports", readFile(name));
   var exports = {};
   code(exports);
   return exports;
}
console.log(require("weekDay").name(1));
//-> Monday
```

因為 `new Function` 會把模組的程式碼包在一個函數裡面，在模組裡面我們就不用再另外包一個為了保護命名空間(namespace)的函數了。然後因為我們把exports設成是module函數的變數，所以在模組裡面我們也就不用宣告exports了。這為我們省了不少工作。

```
var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday"];

exports.name = function(number) {
  return names[number];
};
exports.number = function(name) {
  return names.indexOf(name);
};
```

當依循這樣的模式時，一個模組通常會從宣告變數，這些變數會載入幾個依靠的模組。

```
  var weekDay = require("weekDay");
  var today = require("today");
  console.log(weekDay.name(today(dayNumber())));
```

這個簡單的 `require` 實行方法有幾個問題：第一，每次一個模組被require的時候，這個模組就會被載入然後執行，所以如果有很多模組有相同的dependency，或是這個 `require` 呼叫被放在一個會被執行很多次的函數裡面，有很多時間和資源就會被浪費了。要解決這個問題，我們可以把已經被載入的模組存在一個物件裡，然後當下一次又被require的時候就直接回傳這個物件。

第二，模組只能輸出 `exports` 物件，而無法直接輸出一個值，例如一個function。例如一個模組可能想輸出一個他定義的物件的建構函數(constructor)，目前這無法被實現，因為 `require` 會以他宣告的 `exports` 物件來做為輸出的值。解決這件事有一個傳統的方法，那就是再提供模組另一個變數 `module` ， `module` 是一個物件，它擁有 `exports` 屬性。這個屬性一開始會指向 `require` 宣告的那個空的物件，但是可以被複寫(overwritten)，以輸出不同的東西。

```
function require(name) {
  if(name in require.cache)
    return require.cache[name];

  var code = new Function("exports, module", readFile(name));
  var exports = {}, module = {exports: exports};
  code (exports, module);

  require.cache[name] = module.exports;
  return module.exports;
}
require.cache = Object.create(null);
```

現在這個模組系統只用了一個global variable (require)來讓模組互相使用，且不用再經過global scope。這個style又叫做CommonJS modules，它內建於Node.js系統裡。更重要的是，這是一個更聰明的方法讓你從一個模組名稱(module name)讀到該模組實際的程式碼，允許「相對於目前檔案位置的relative path」和「安裝在local端的模組名稱」。

慢慢載入的模組

雖然我們可以用CommonJS module的style來寫瀏覽器的JavaScript，這其實是有點困難的，因為系統從網路上讀取一個檔案(模組)比從磁碟裡讀取還要慢得多，而當一個script在瀏覽器中執行時，瀏覽器就不能做其他事情了。意思是說如果每一個 `require` call都從遠端的伺服器上讀取檔案，那在載入檔案時，網頁將會持續凍結一段時間。

有一個方法是在你把程式碼上到網頁上之前，執行Browserify這個程式，這個程式會找到所有的 `require` call，解決所有的dependencies問題，然後把所有需要的程式碼收集到一個單獨的檔案中，所以這個網頁只要能夠載入這個檔案，就可以取得所有他需要的模組。

另一個方法是把模組的程式碼包在一個函數裡，然後module loader可以在背景裡先載入所有的dependencies，載完之後再呼叫這個函數、開始使用這個模組。這叫做Asynchronous Module Definition (AMD)。

```
define(["weekDay", "today"], function(weekDay, today) {
  console.log(weekDay.name(today.dayNumber()));
});
```

這個 `define` 函數是關鍵。第一個引數是module names的array，這是我們需要的modules(dependencies)，然後是一個函數，這個函數的引數就是前面module names。他會先開始在背景載入dependencies(如果還沒被載入的話)，在檔案還是被下載時，這會讓網頁可以繼續運作。當所有的dependencies都被載完時， `define` 就會呼叫第二個引數的那個函數。

以這樣的方式被載入的modules裡面必須包含一段呼叫 `define` 的程式碼。而被用來當作引數的值(modules names of dependencies)的東西，其實就是這個函數回傳的值。

```
define([], function() {
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thurs

    return {
      name: function(number) {return names[number];},
      number: function(name) {return names.indexOf[name];}
  };
});
```

為了可以呈現最低可能的(minimal implementation) `define` ，我們需要假裝我們有一個 `backgroundReadfile` 函數，有兩個引數：filename和function，一載完這個模組，他馬上會用載入的這個模組(名稱是filename)來呼叫這個function。

為了可以追蹤模組的載入進度， `define` 會需要一個可以記錄模組狀態的物件，告訴我們這個模組是否已經可用，當已經可以使用時，並提供這個介面給我們。

若我們給 `getModule` 函數一個filename，他就會回傳這個filename的模組並且確保下載動作被排程執行。他同時使用一個cache物件來避免載入已經載過的模組。

```
var defineCache = Object.create(null);
var currentMod = null;

function getModule(name){
  if (name in defineCache)
    return defineCache[name];

  var module = {exports: null,
                loaded: false,
                onLoad: []};
  defineCache[name] = module;
  backgroundReadFile(name, function(code) {
    currentMod = module;
    new Function("", code)();
  });
  return module;
}
```

我們假設被載入模組包含了一個(只有一個)對 `define` 的call。 `currentMod` 變數是用來告訴這個call有關這個正在被下載的module物件，所以當他結束下載時，currentMod才可以更新他的狀態。

`define` 函數本身用 `getModule` 來

```
function define(depNames, moduleFunction) {
  var myMod = currentMod;
  var deps = depNames.map(getModule);

  deps.forEach(function(mod) {
    if (!mod.loaded)
      mod.onLoad.push(whenDepsLoaded);
  });

  function whenDepsLoaded() {
    if(!deps.every(function(m) {return m.loaded; }))
      return;

    var args = deps.map(function(m) {return m.exports; });
    var exports = moduleFunction.apply(null, args);
    if (myMod) {
      myMod.exports = exports;
      myMod.loaded = true;
      myMod.onLoad.forEach(function(f) { f(); });
    }
   }
   whenDepsLoaded();
}
```