

# JavaScript 闭包和模块化

# 函数的作用域和生命周期

```
var foo = 1;
function add() {
  foo++;
  console.log(foo);
}
add();
add();
```

```
function init() {
  var foo = 1;
  function add() {
    foo++;
    console.log(foo);
  }
  add();
}
init();
init();
```

```
function init() {
  var foo = 1;
  function add() {
    foo++;
    console.log(foo);
  }
  return add;
}
var add2 = init();
add2();
add2();
```

在计算机科学中，闭包（英语：Closure），又称词法闭包（Lexical Closure）或函数闭包（function closures），**是引用了自由变量的函数**。这个被引用的**自由变量**将和这个函数一同存在，**即使已经离开了创造它的环境也不例外**。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。

# 使用闭包模拟私有方法

```
var Counter = (function() {  
  var privateCounter = 0;  
  function changeBy(val) {  
    privateCounter += val;  
  }  
  return {  
    increment: function() {  
      changeBy(1);  
    },  
    decrement: function() {  
      changeBy(-1);  
    },  
    value: function() {  
      return privateCounter;  
    }  
  }  
})();
```

```
console.log(Counter.value()); /* logs 0 */  
Counter.increment();  
Counter.increment();  
console.log(Counter.value()); /* logs 2 */  
Counter.decrement();  
console.log(Counter.value()); /* logs 1 */
```

# 模块化

- 1) **可维护性**。 因为模块是独立的，一个设计良好的模块会让外面的代码对自己的依赖越少越好，这样自己就可以独立去更新和改进。
- 2) **命名空间**。 在 JavaScript 里面，如果一个变量在最顶级的函数之外声明，它就直接变成全局可用。因此，常常不小心出现命名冲突的情况。
- 3) **重用代码**。 我们有时候会喜欢从之前写过的项目中拷贝代码到新的项目，这没有问题，但是更好的方法是，通过模块引用的方式，来达到更好的效果。

# 常用的套路

- CommonJS
- AMD
- UMD

# CommonJS

- CommonJS 最开始是 Mozilla 的工程师于 2009 年开始的一个项目，它的目的是让浏览器之外的 JavaScript（比如服务器端或者桌面端）能够通过模块化的方式来开发和协作。

```
// myModule.js
```

```
function myModule() {  
  this.hello = function() {  
    return 'hello!';  
  }  
  
  this.goodbye = function() {  
    return 'goodbye!';  
  }  
}
```

```
module.exports = myModule;
```

```
// main.js
```

```
var myModule = require('myModule');
```

```
var myModuleInstance = new myModule();  
myModuleInstance.hello(); // 'hello!'  
myModuleInstance.goodbye(); // 'goodbye!'
```



- 避免全局命名空间污染，require 进来的模块可以被赋值到自己随意定义的局部变量中，所以即使是同一个模块的不同版本也可以完美兼容
- 让各个模块的依赖关系变得很清晰
- CommonJS 规范的主要适用场景是服务器端编程，所以采用同步加载模块的策略。如果我们依赖3个模块，代码会一个一个加载它们。



命令行:

```
$ browserify -r a -r b -r ./my-file.js:c > bundle.js
```

index.html:

```
<script src="bundle.js"></script>
<script>
  var through = require('a');
  var duplexer = require('b');
  var myModule = require('c');
  /* ... */
</script>
```

# AMD (Asynchronous Module Definition)

```
<script src="1.js"></script>  
<script src="2.js"></script>  
<script src="3.js"></script>  
<script src="4.js"></script>  
<script src="5.js"></script>  
<script src="6.js"></script>
```

# AMD规范诞生背景

- CommanJS规范

```
var math = require('math');  
math.add(2, 3); // 5
```

- 浏览器使用CommanJS会阻塞浏览器渲染
- AMD是"Asynchronous Module Definition"的缩写, 意思就是"异步模块定义"。

# AMD代码示例

```
define(['jquery'] , function ($) {  
    return function () {};  
});
```

# AMD模块定义

```
define(['myLib'], function(myLib){  
    function foo(){  
        myLib.doSomething();  
    }  
    return {  
        foo : foo  
    };  
});
```

- jQuery最初就支持AMD规范，可以作为一个模块被引入到项目中
- 自从jQuery 2.1， jQuery使用AMD来在内部管理依赖

# AMD规范

AMD规范 (Asynchronous Module Definition)

定义模块代码:

```
define(  
    module_id /*可选*/,  
    [dependencies] /*可选*/,  
    definition function /*用来初始化模块或对象的函数*/  
);
```



# 定义匿名模块方法1

```
define({  
    method1: function() {},  
    method2: function() {},  
});
```

# 定义匿名模块方法2

```
define(function () {  
    return {  
        method1: function() {},  
        method2: function() {},  
    };  
});
```

# 定义非独立模块

```
define(['module1', 'module2'], function(m1, m2) {  
    ...  
});
```

['module1', 'module2']表示我们定义的这个新模块依赖于module1模块和module2模块，只有先加载这两个模块，新模块才能正常运行。

当前面数组的所有成员加载成功后，它将被调用。它的参数与数组的成员一一对应。

# UMD

- 对于需要同时支持 AMD 和 CommonJS 的模块而言，可以使用 UMD (Universal Module Definition) 。
- 在执行UMD规范时，会优先判断是当前环境是否支持AMD环境，然后再检验是否支持CommonJS环境，否则认为当前环境为浏览器环境 (window) 。

```
(function (root, factory) {  
  if (typeof define === 'function' && define.amd) {  
    // AMD  
    define(['myModule', 'myOtherModule'], factory);  
  } else if (typeof exports === 'object') {  
    // CommonJS  
    module.exports = factory(require('myModule'), require('myOtherModule'));  
  } else {  
    // Browser globals (Note: root is window)  
    root.returnExports = factory(root.myModule, root.myOtherModule);  
  }  
})(this, function (myModule, myOtherModule) {  
  // Methods  
  function notHelloOrGoodbye(){}; // A private method  
  function hello(){}; // A public method because it's returned (see below)  
  function goodbye(){}; // A public method because it's returned (see below)  
  
  // Exposed public methods  
  return {  
    hello: hello,  
    goodbye: goodbye  
  }  
}));
```

# ES6 Module

```
// lib/counter.js  
export let counter = 1;
```

```
export function increment() {  
  counter++;  
}
```

```
export function decrement() {  
  counter--;  
}
```

```
// src/main.js  
import * as counter from '../..../counter';
```

```
console.log(counter.counter); // 1  
counter.increment();  
console.log(counter.counter); // 2
```