今天无论是在浏览器中还是在浏览器外，JavaScript世界正在经历翻天覆地地变化。如果我们谈论脚本加载、客户端的MVC框架、压缩器、AMD、Common.js还有Coffeescript……只会让你的脑子发昏。对于那些已经早就熟知这些技术的人而言，或许很难想象到现在为止还有很多JS开发者还不熟悉这些工具，甚至事实上，他们很可能现在还不想去尝试这些工具。

这篇文章将会介绍一些很基础的JS知识，以及当开发者想要尝试Backbone.js和Ember.js之类的工具之前需要知道一些内容。当你理解了这篇文章中的大部分内容的时候，你会更有信心去学习其他高级JavaScript知识的时候。这篇文章是假设你曾经使用过JavaScript的，所以如果你从没有接触过它，也许你需要先了解下更基础的知识。现在我们开始吧！

模块

有多少人在一个文件中写的JS像下面的代码块一样？（注意：我可没有说内嵌在HTML文件中哦）：

```
var someSharedValue = 10;
var myFunction = function(){ //do something }
var anotherImportantFunction = function() { //do more stuff }
```

如果你做到了这一点，那么很有可能你正在写这样的代码。我不是在给你下定义，因为在相当长的一段时间里我也曾这么写程序。事实上这段代码有很多毛病，不过我们会专注在讨论全局命名空间的污染问题上。这样的代码代码会把方法和变量都暴露在了全局中，我们需要将让这些数据与全局命名空间独立开来，我们将会采用模块模式（Module Pattern）来实现这个目的。模块中可以有很多不同的形式达到我们的目标，我会从最简单的方法开始说：立即调用的函数表达式（Immediately Invoked Function Expression，简写为：IIFE）。

名字听起来很高大上，不过它的实现其实很简单：

```
(function(){
    //do some work
})();
```

如果在此之前你从未接触过这种函数，可能现在你会觉得它很怪 — 怎么会有这么多括号！它是会立即执行的函数，你可以这么理解：一个函数被创建了后又立刻被调用。它应该是一个表达而不是一个语句：一个函数语句是一定要有一个名字的，但是大家也看到了，立即执行函数是没有名字的。在函数定义的外部还有一组括号，这一点也能很好地帮助我们在代码中轻易找到匿名函数的身影。

现在我们知道要怎么写一个立即执行函数了，那就来聊聊为什么要使用它吧。在JS中我们都是在和各种作用域之中的函数打交道，所以如果我们想要创建一个作用域，就可以使用函数。匿名函数中的变量和方法的作用域仅仅在匿名函数中，就不会污染全局的命名空间，那么现在还需要考虑的一个问题是，我们要如何从外部取得那些在匿名函数作用域中的变量和方法呢？答案就是全局命名空间：将变量放入全局命名空间中，或者至少将作用变量与全局命名空间关联起来

想要在立即执行函数外部调用方法，我们可以将window对象传入立即执行函数，再将函数或变量值赋值到这个对象上。为了保证这个window对象的引入不会造成什么混乱，我们可以将widow对象作为一个变量传入我们的立即执行函数中。当做函数传入参数的方法同样适用于第三方库，甚至*undefined*这样的值。现在我们的立即执行函数看起来是这样的：

```
(function(window, $, undefined){
```

```
    //do some work
})(window, jQuery);
```

正如你所看到的，我们将window和jQuery传入函数中（’ $'符号表示的就是’jQuery’ ，把它用在这的原因是防止其他库也定义了’ $'），但是这个函数其实是接收了3个参数。如果我们没有传入第三个参数，形参*undefined* 的值会保持这个状态，而不会由于其他代码把全局的undefined改变了它也跟着改变。其实在函数内我们也是可以直接使用这些值，能这么做的原理是，JS的闭包会覆盖他们所处的上下文。对于这个话题，我曾写过一篇关于C#的文章以解释这个概念，这两者是互通的。

现在我们有了一个会立即执行的方法，还有一个相对安全的执行上下文，其中还包含有*window*、*$*和*undefined*变量（这几个变量还是有可能在这个脚本被执行前就被重新赋值，不过现在的可能性要小的多了）。现在我们已经做得很好了：把我们的代码从全局环境下的一团混乱的局面中拯救了出来；降低了与其他在同一应用中使用的脚本的冲突可能性。

任何我们想要从模块中获取的东西都可以通过window对象拿到。但是通常我不会直接将模块中的内容直接复制到window对象上，而是会用更有组织性地将模块中的内容。在大部分语言中，我们将这些容器称为“命名空间” ，在JS中我们可以用“对象”的方式来模拟。

命名空间

如果我们想要声明一个命名空间，将一个函数放进这个空间中，代码可以写成这样：

```
window.myApp = window.myApp || {};
window.myApp.someFunction = function(){
    //so some work
};
```

我们是在全局环境中创建了一个对象，它是用于查看另外的某个对象是否已经存在，如果已经存在了，那么我们就可以直接使用；不然就需要用’ {}’ 来创建一个新的对象。接着，我们可以开始添加这个命名空间的内容，将各种函数放入这个空间中，就像上面的代码片段所做的那样，但是我们又不希望这些函数就随便的放在那里，而是希望将模块和命名空间联系在一起，就像下面这样：

```
(function(myApp, $, undefined){
    //do some work
}(window.myApp = window.myApp || {}, jQuery));
```

还可以这么写：

```
window.myApp = (function(myApp, $, undefined){
    //do some work
    return myApp;
})(window.myApp || {}, jQuery);
```

现在，我们不再是将window传入我们的模块中，我们将一个和window对象联系在一起的命名空间传入模块中。之所以使用’ ||’ 的原因是我们可以重复使用同一个命名空间，而不是每次需要使用命名空间的时候我们又要重新创建一个。许多包含有命名空间方法的库会帮你创建好空间的，或者你可以使用一些想namespace.js这样的工具来构建嵌套的命名空间。由于在JS中，每一个在命名空间中的项你都不得不指定它的命名空间，所以通常我都尽量不会去创建深度嵌套的命名空间。如果你在*MyApp.MyModule.MySubModule*中创建了一个*doSomething*方法，你需要这么引用它：

```
MyApp.MyModule.MySubModule.doSomething();
```

每次你要调用它，或者你可以在你的模块中给这个命名空间一个别名：

```
var MySubModule = MyApp.MyModule.MySubModule;
```

这样定义以后，如果你想用*doSomething*这个方法可以用*MySubModule.doSomething()*来调用。不过这个方式其实是不必要的，除非你有非常非常多的代码，不然这么做只会将问题复杂化。

揭秘模块模式

在创建模块时你也常会看到另一种设计模式：揭秘模块模式（Revealing Module Pattern）。它和模块模式有一些不同：所有定义在模块中的内容都是私有的，然后你可以把所有要暴露到模块外部的内容放在一个对象中，再返回这个对象。你可以这么做：

```
var myModule = (function($, undefined){
     var myVar1 = '',
     myVar2 = '';

     var someFunction = function(){
         return myVar1 + " " + myVar2;
     };

     return {
         getMyVar1: function() { return myVar1; }, //myVar1 pul
         setMyVar1: function(val) { myVar1 = val; }, //myVar1 
         someFunction: someFunction //some function made publi
     }
})(jQuery);
```

一次就建立一个模块，然后返回一个包含有需要公有化的模块片段的对象，同时模块中需要保持私有的变量也不会被暴露。*myModule*变量会包含有两个共有的项，不过其中*Somefunction*中的*myVar2*是从外部获取不到的。

创建构造器（类）

在JS中没有“类”这个概念，但是我们可以通过创建构造器来创建“对象”。假设现在我们要创建一系列*Person*对象，还需要传入姓、名和年龄，我们可以将构造器定义成下面这样（这部分代码应该放在模块之中）：

```
var Person = function(firstName, lastName, age){
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
}

Person.prototype.fullName = function(){
    return this.firstName + " " + this.lastName;
};
```

现在先看第一个函数，你会看到我们创建了一个*Person*构造器。我们用它来构造新的*person*对象。这个构造器需要3个传入参数，然后将这3个参数赋值到执行上下文中。我们也是通过这种方式获取到公有实例变量。这里也可以创建私有变量：将传入参数赋值到这个构造器中的局部变量。但是这么做以后，公有的方法就没法获取这些私有的变量了，所以你最好还是把它们都变成公有的。也可以把方法放在构造器中同时

还能从外部获取到它，这样方法就能拿到构造器里的私有变量了，不过这么做的话又会出现一系列新的问题。

第二个方法中我们使用了*Person*构造器的”原型”（prototype）。一个函数的原型就是一个对象，当你需要在某个实例上解析它所调用到的字段或者函数时你需要遍历这个函数上所有的实例。所以这几行代码所做的就是创建一个*fullName*方法的实例，然后所有的*Person*的实例都能直接调用到这方法，而不是对每个*Person*实例都添加一个*fullName*方法，造成方法的泛滥。我们也可以在构造器中用

```
this.fullName = function() { …
```

的方式定义*fullName*，但这样每一个*Person*实例都会有*fullName*方法的副本，这不是我们希望的。

如果我们想要创建一个*Person*实例，我们可以这么做：

```
var person = new Person("Justin", "Etheredge");
alert(person.fullName());
```

我们也可以创建一个继承自*Person*的构造器：*Spy*构造器，我们会创建*Spy*的一个实例，不过只会声明一个方法：

```
var Spy = function(firstName, lastName, age){
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
};
Spy.prototype = new Person();

Spy.prototype.spy = function(){
    alert(this.fullName() + " is spying.");
}

var mySpy = new Spy("Mr.", "Spy", 50);
mySpy.spy();
```

正如你所看到的，我们创建了一个和*Person*很相似的构造器，但是它的原型是*Person*的一个实例。现在我们又添加上一些方法，使得*Spy*的实例又可以调用到*Person*的方法，同时还能直接取得*Spy*中的变量。这个方法比较复杂，不过一旦你明白怎么使用了，你的代码就会变得很优雅。

结语

看到这里，希望你已经学到了一些东西。不过这篇文章里并没有介绍多少关于“现代”JS的开发。这篇文章中涉及的还是旧知识，在过去几年里它们的使用面相当广。希望你看完这篇文章以后，找到了学习JS的正确的方向。现在可能你把代码放到了不同的模块不同的文件中（你应该做到这一点！），那么下一步你要开始着手研究如何将JS结合和压缩。如果你是使用Rails 3的开发者，可以在asset pipeline上免费获取这些信息或者工具。如果你是.NET开发者，你可以看看SquishIt框架，我就是从这里开始的。如果你是ASP.NET MVC 4的开发者，也有相关的资源。

希望这篇文章对你有帮助。以后我也会介绍关于现代JS的开发，期待到时候能看到你的ID。