JS模块化编程（1）——模块化

Alex Sun
2014-11-17

1. 全局变量及其他

在规模不大的JS开发中，我们常常会使用如下写法：

```
var a = "hello world";

function aaa() {
    console.log("hahahha");
}

function b() {
    i = 1;
}
b();
```

其中变量a和函数aaa()都会挂载到window下，成为全局变量。当执行函数b()时，发现为定义的变量i，于是向上一级作用域里去找i的定义，一直找到window的时候发现i仍未定义，于是将i挂载到window下，此时i也成为了一个全局变量。

然而，随着项目规模的不断扩大，这样的写法就会产生问题。在大规模的JS开发中，会有多个js文件，假如a.js和b.js中都定义了函数aaa()，那么后加载的就会覆盖掉之前的函数定义，造成全局变量的污染。因此在开发中，我们需要进行模块化，就像Java中的类或包那样，从而避免全局变量污染等问题。

2. 简单的模块化

我们很容易想到，可以将变量的定义挂载到一个对象下面，之后再调用的时候，我们只需要通过该对象进行访问就可以了，例如：

```
var module = {
```

```
    name: "alex",

    sayHello: function() {
        return console.log("hello " + this.name);
    }
};

console.log(module.name);
module.sayHello();
```

但是这样也会存在问题，即私有变量的访问问题，看如下例子：

```
var module = {
    name: "alex",

    _password: "12345",

    showPassword: function() {
        console.log(this._password);
    }
};

module._password = "54321";
module.showPassword(); // 54321
```

这里_password是一个私有变量，在外部是不应该直接访问到的，但是事实上我们可以在模块外部随意的访问和修改。于是我们需要以一个闭包的形式来定义模块，例如：

```
var module = (function() {
    var _password = "12345";

    return {
        name: "alex",
        showPassword: function() {
            console.log(_password);
        },
        changePassword: function(newPwd) {
            _password = newPwd;
        }
    };
})();

console.log(module._password); // undefined
module.showPassword(); // 12345
module.changePassword("54321");
module.showPassword(); // 54321
```

3. 模块扩展

在复杂的项目中，有时候我们需要对一个模块进行扩展，此时可以通过一个立即执行的函数来实现（当然，直接通过module.newProp也是可以进行扩展的）：

```
var module1 = (function() {
    var _password = "12345";

    return {
        name: "alex",
        showPassword: function() {
            console.log(_password);
        },
        changePassword: function(newPwd) {
            _password = newPwd;
        }
    };
})();

var module2 = (function(module) {
    module.sayHello = function() {
        console.log("hello" + this.name);
    };

    return module;
})(module1);

module1.sayHello(); // hello alex
module2.sayHello(); // hello alex
```

如果扩展并不是依赖于module1，或者module1为定义，我们可以这样子实现：

```
var module1;

var module2 = (function(module) {
    module.sayHello = function() {
        console.log("hello world");
    };

    return module;
})(module1 || {});
```

```
module2.sayHello(); // hello alex
```

我们也可以为模块定义一个扩展方法来实现扩展，例如：

```
var module = (function() {
    return {
        name: "alex",
        extend: function(obj) {
            for (key in obj) {
                this[key] = obj[key];
            }
        }
    };
})();

module.extend({
    sayHello: function() {
        console.log("hello " + this.name);
    }
});

module.sayHello(); // hello alex
```

4. 模块导入

有些时候，一个模块的实现可能会依赖于另一个模块，例如某个模块内部可能会用到jQuery的一些方法，此时我们可以将依赖作为参数传入到自执行的函数中，例如：

```
var module = (function($) {
    return {
        querySelector: function(selector) {
            return $(selector);
        }
    };
})(jQuery);
```

这样写还有一个好处就是，假如在模块外部jQuery放弃了**操作符**，**但是在模块内依然可以使用**来代替jQuery，因为在模块内部$===jQuery。

一个JS库本质上就是一个单独的模块，事实上，很多JS库都是基于这样的一个方法来构建代码的，例如jQuery，其整体结构如下（jQuery-2.0.3）：

```
(function(window, undefined) {
    // ... ...
})(window);
```

参考：

JavaScript Module Pattern: In-Depth