第 4 章　单例模式

单例模式的定义是：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

单例模式是一种常用的模式，有一些对象我们往往只需要一个，比如线程池、全局缓存、浏览器中的window 对象等。在JavaScript开发中，单例模式的用途同样非常广泛。试想一下，当我们单击登录按钮的时候，页面中会出现一个登录浮窗，而这个登录浮窗是唯一的，无论单击多少次登录按钮，这个浮窗都只会被创建一次，那么这个登录浮窗就适合用单例模式来创建。

4.1 实现单例模式

要实现一个标准的单例模式并不复杂，无非是用一个变量来标志当前是否已经为某个类创建过对象，如果是，则在下一次获取该类的实例时，直接返回之前创建的对象。代码如下：

```
var Singleton = function( name ){
    this.name = name;
    this.instance = null;
};

Singleton.prototype.getName = function(){
    alert ( this.name );
};

Singleton.getInstance = function( name ){
    if ( !this.instance ){
        this.instance = new Singleton( name );
    }
    return this.instance;
};

var a = Singleton.getInstance( 'sven1' );
var b = Singleton.getInstance( 'sven2' );

alert ( a === b );    // true
```

或者：

```
var Singleton = function( name ){
    this.name = name;
};

Singleton.prototype.getName = function(){
    alert ( this.name );
};

Singleton.getInstance = (function(){
    var instance = null;
    return function( name ){
```

```
        if ( !instance ){
            instance = new Singleton( name );
        }
        return instance;
    }
})();
```

我们通过Singleton.getInstance 来获取Singleton 类的唯一对象，这种方式相对简单，但有一个问题，就是增加了这个类的“不透明性”，Singleton 类的使用者必须知道这是一个单例类，跟以往通过new XXX 的方式来获取对象不同，这里偏要使用Singleton.getInstance 来获取对象。

接下来顺便进行一些小测试，来证明这个单例类是可以信赖的：

```
var a = Singleton.getInstance( 'sven1' );
var b = Singleton.getInstance( 'sven2' );

alert ( a === b );     // true
```

虽然现在已经完成了一个单例模式的编写，但这段单例模式代码的意义并不大。从下一节开始，我们将一步步编写出更好的单例模式。

4.2 透明的单例模式

我们现在的目标是实现一个“透明”的单例类，用户从这个类中创建对象的时候，可以像使用其他任何普通类一样。在下面的例子中，我们将使用CreateDiv 单例类，它的作用是负责在页面中创建唯一的div 节点，代码如下：

```
var CreateDiv = (function(){

    var instance;

    var CreateDiv = function( html ){
        if ( instance ){
            return instance;
        }
        this.html = html;
        this.init();
        return instance = this;
    };

    CreateDiv.prototype.init = function(){
        var div = document.createElement( 'div' );
        div.innerHTML = this.html;
        document.body.appendChild( div );
    };

    return CreateDiv;

})();

var a = new CreateDiv( 'sven1' );
var b = new CreateDiv( 'sven2' );

alert ( a === b );     // true
```

虽然现在完成了一个透明的单例类的编写，但它同样有一些缺点。

为了把instance 封装起来，我们使用了自执行的匿名函数和闭包，并且让这个匿名函数返回真正的Singleton 构造方法，这增加了一些程序的

复杂度，阅读起来也不是很舒服。

观察现在的Singleton 构造函数：

```
var CreateDiv = function( html ){
    if ( instance ){
        return instance;
    }
    this.html = html;
    this.init();
    return instance = this;
};
```

在这段代码中，CreateDiv 的构造函数实际上负责了两件事情。第一是创建对象和执行初始化init 方法，第二是保证只有一个对象。虽然我们目前还没有接触过“单一职责原则”的概念，但可以明确的是，这是一种不好的做法，至少这个构造函数看起来很奇怪。

假设我们某天需要利用这个类，在页面中创建千千万万的div ，即要让这个类从单例类变成一个普通的可产生多个实例的类，那我们必须得改写CreateDiv 构造函数，把控制创建唯一对象的那一段去掉，这种修改会给我们带来不必要的烦恼。

4.3 用代理实现单例模式

现在我们通过引入代理类的方式，来解决上面提到的问题。

我们依然使用4.2节中的代码，首先在CreateDiv 构造函数中，把负责管理单例的代码移除出去，使它成为一个普通的创建div 的类：

```
var CreateDiv = function( html ){
    this.html = html;
    this.init();
};

CreateDiv.prototype.init = function(){
    var div = document.createElement( 'div' );
    div.innerHTML = this.html;
    document.body.appendChild( div );
};
```

接下来引入代理类proxySingletonCreateDiv：

```
var ProxySingletonCreateDiv = (function(){

    var instance;
    return function( html ){
        if ( !instance ){
            instance = new CreateDiv( html );
        }

        return instance;
    }

})();

var a = new ProxySingletonCreateDiv( 'sven1' );
var b = new ProxySingletonCreateDiv( 'sven2' );

alert ( a === b );
```

通过引入代理类的方式，我们同样完成了一个单例模式的编写，跟之前不同的是，现在我们把负责管理单例的逻辑移到了代理类proxySingletonCreateDiv 中。这样一来，CreateDiv 就变成了一个普通的类，它跟proxySingletonCreateDiv 组合起来可以达到单例模式的效果。

本例是缓存代理的应用之一，在第6章中，我们将继续了解代理带来的好处。

4.4 JavaScript中的单例模式

前面提到的几种单例模式的实现，更多的是接近传统面向对象语言中的实现，单例对象从“类”中创建而来。在以类为中心的语言中，这是很自然的做法。比如在Java中，如果需要某个对象，就必须先定义一个类，对象总是从类中创建而来的。

但JavaScript其实是一门无类（class-free）语言，也正因为如此，生搬单例模式的概念并无意义。在JavaScript中创建对象的方法非常简单，既然我们只需要一个“唯一”的对象，为什么要为它先创建一个“类”呢？这无异于穿棉衣洗澡，传统的单例模式实现在JavaScript中并不适用。

单例模式的核心是确保只有一个实例，并提供全局访问。

全局变量不是单例模式，但在JavaScript开发中，我们经常会把全局变量当成单例来使用。例如：

```
var a = {};
```

当用这种方式创建对象a 时，对象a 确实是独一无二的。如果a 变量被声明在全局作用域下，则我们可以在代码中的任何位置使用这个变量，全局变量提供给全局访问是理所当然的。这样就满足了单例模式的两个条件。

但是全局变量存在很多问题，它很容易造成命名空间污染。在大中型项目中，如果不加以限制和管理，程序中可能存在很多这样的变量。JavaScript中的变量也很容易被不小心覆盖，相信每个JavaScript程序员都曾经历过变量冲突的痛苦，就像上面的对象`var a = {};`，随时有可能被别人覆盖。

Douglas Crockford多次把全局变量称为JavaScript中最糟糕的特性。在对JavaScript的创造者Brendan Eich的访谈中， Brendan Eich本人也承认全局变量是设计上的失误，是在没有足够的时间思考一些东西的情况下导致的结果。

作为普通的开发者，我们有必要尽量减少全局变量的使用，即使需要，也要把它的污染降到最低。以下几种方式可以相对降低全局变量带来的命名污染。

1. 使用命名空间

适当地使用命名空间，并不会杜绝全局变量，但可以减少全局变量的数量。

最简单的方法依然是用对象字面量的方式：

```
var namespace1 = {
    a: function(){
        alert (1);
    },
    b: function(){
        alert (2);
    }
};
```

把a 和b 都定义为namespace1 的属性，这样可以减少变量和全局作用域打交道的机会。另外我们还可以动态地创建命名空间，代码如下（引自 ***Object-Oriented JavaScrtipt*** 一书）：

```
var MyApp = {};

MyApp.namespace = function( name ){
    var parts = name.split( '.' );
    var current = MyApp;
    for ( var i in parts ){
        if ( !current[ parts[ i ] ] ){
            current[ parts[ i ] ] = {};
        }
        current = current[ parts[ i ] ];
    }
};

MyApp.namespace( 'event' );
MyApp.namespace( 'dom.style' );
```

```
console.dir( MyApp );

// 上述代码等价于:

 var MyApp = {
     event: {},
     dom: {
         style: {}
     }
 };
```

2. 使用闭包封装私有变量

这种方法把一些变量封装在闭包的内部，只暴露一些接口跟外界通信:

```
var user = (function(){
    var __name = 'sven',
        __age = 29;

    return {
        getUserInfo: function(){
            return __name + '-' + __age;
        }
    }

})();
```

我们用下划线来约定私有变量__name 和__age ，它们被封装在闭包产生的作用域中，外部是访问不到这两个变量的，这就避免了对全局的命令污染。

4.5 惰性单例

前面我们了解了单例模式的一些实现办法，本节我们来了解惰性单例。

惰性单例指的是在需要的时候才创建对象实例。惰性单例是单例模式的重点，这种技术在实际开发中非常有用，有用的程度可能超出了我们的想象，实际上在本章开头就使用过这种技术，instance 实例对象总是在我们调用Singleton.getInstance 的时候才被创建，而不是在页面加载好的时候就创建，代码如下：

```
Singleton.getInstance = (function(){
    var instance = null;
    return function( name ){
        if ( !instance ){
            instance = new Singleton( name );
        }
        return instance;
    }
})();
```

不过这是基于“类”的单例模式，前面说过，基于“类”的单例模式在JavaScript中并不适用，下面我们将以WebQQ的登录浮窗为例，介绍与全局变量结合实现惰性的单例。

假设我们是WebQQ的开发人员（网址是web.qq.com），当点击左边导航里QQ头像时，会弹出一个登录浮窗（如图4-1所示），很明显这个浮窗在页面里总是唯一的，不可能出现同时存在两个登录窗口的情况。

图 4-1

第一种解决方案是在页面加载完成的时候便创建好这个div 浮窗，这个浮窗一开始肯定是隐藏状态的，当用户点击登录按钮的时候，它才开始显示：

```
<html>
    <body>
        <button id="loginBtn">登录</button>
    </body>

<script>
    var loginLayer = (function(){
        var div = document.createElement( 'div' );
        div.innerHTML = '我是登录浮窗';
        div.style.display = 'none';
        document.body.appendChild( div );
        return div;
    })();

    document.getElementById( 'loginBtn' ).onclick = function(){
        loginLayer.style.display = 'block';
    };
</script>
</html>
```

这种方式有一个问题，也许我们进入WebQQ只是玩玩游戏或者看看天气，根本不需要进行登录操作，因为登录浮窗总是一开始就被创建好，那么很有可能将白白浪费一些DOM节点。

现在改写一下代码，使用户点击登录按钮的时候才开始创建该浮窗：

```
<html>
    <body>
        <button id="loginBtn">登录</button>
    </body>

<script>
    var createLoginLayer = function(){
        var div = document.createElement( 'div' );
        div.innerHTML = '我是登录浮窗';
        div.style.display = 'none';
        document.body.appendChild( div );
        return div;
    };

    document.getElementById( 'loginBtn' ).onclick = function(){
        var loginLayer = createLoginLayer();
        loginLayer.style.display = 'block';
    };
</script>
</html>
```

虽然现在达到了惰性的目的，但失去了单例的效果。当我们每次点击登录按钮的时候，都会创建一个新的登录浮窗div 。虽然我们可以在点击浮窗上的关闭按钮时（此处未实现）把这个浮窗从页面中删除掉，但这样频繁地创建和删除节点明显是不合理的，也是不必要的。

也许读者已经想到了，我们可以用一个变量来判断是否已经创建过登录浮窗，这也是本节第一段代码中的做法：

```
var createLoginLayer = (function(){
```

```
    var div;
    return function(){
        if ( !div ){
            div = document.createElement( 'div' );
            div.innerHTML = '我是登录浮窗';
            div.style.display = 'none';
            document.body.appendChild( div );
        }

        return div;
    }
})();

document.getElementById( 'loginBtn' ).onclick = function(){
    var loginLayer = createLoginLayer();
    loginLayer.style.display = 'block';
};
```

4.6 通用的惰性单例

上一节我们完成了一个可用的惰性单例，但是我们发现它还有如下一些问题。

- 这段代码仍然是违反单一职责原则的，创建对象和管理单例的逻辑都放在createLoginLayer 对象内部。

- 如果我们下次需要创建页面中唯一的iframe ，或者script 标签，用来跨域请求数据，就必须得如法炮制，把createLoginLayer 函数几乎照抄一遍：

```
var createIframe= (function(){
    var iframe;
    return function(){
        if ( !iframe){
            iframe= document.createElement( 'iframe' );
            iframe.style.display = 'none';
           document.body.appendChild( iframe);
        }
        return iframe;
    }
})();
```

我们需要把不变的部分隔离出来，先不考虑创建一个div 和创建一个iframe 有多少差异，管理单例的逻辑其实是完全可以抽象出来的，这个逻辑始终是一样的：用一个变量来标志是否创建过对象，如果是，则在下次直接返回这个已经创建好的对象：

```
var obj;
if ( !obj ){
   obj = xxx;
}
```

现在我们就把如何管理单例的逻辑从原来的代码中抽离出来，这些逻辑被封装在getSingle 函数内部，创建对象的方法fn 被当成参数动态传入getSingle 函数：

```
var getSingle = function( fn ){
    var result;
    return function(){
        return result || ( result = fn .apply(this, arguments ) );
    }
};
```

接下来将用于创建登录浮窗的方法用参数fn 的形式传入getSingle ，我们不仅可以传入createLoginLayer ，还能传入createScript 、createIframe 、createXhr 等。之后再让getSingle 返回一个新的函数，并且用一个变量result 来保存fn 的计算结果。result 变量因为身在闭包中，它永远不会被销毁。在将来的请求中，如果result 已经被赋值，那么它将返回这个值。代码如下：

```
var createLoginLayer = function(){
    var div = document.createElement( 'div' );
    div.innerHTML = '我是登录浮窗';
    div.style.display = 'none';
    document.body.appendChild( div );
    return div;
};

var createSingleLoginLayer = getSingle( createLoginLayer );

document.getElementById( 'loginBtn' ).onclick = function(){
    var loginLayer = createSingleLoginLayer();
    loginLayer.style.display = 'block';
};
```

下面我们再试试创建唯一的iframe 用于动态加载第三方页面：

```
var createSingleIframe = getSingle( function(){
    var iframe = document.createElement ( 'iframe' );
    document.body.appendChild( iframe );
    return iframe;
});

document.getElementById( 'loginBtn' ).onclick = function(){
    var loginLayer = createSingleIframe();
    loginLayer.src = 'http://baidu.com';
};
```

在这个例子中，我们把创建实例对象的职责和管理单例的职责分别放置在两个方法里，这两个方法可以独立变化而互不影响，当它们连接在一起的时候，就完成了创建唯一实例对象的功能，看起来是一件挺奇妙的事情。

这种单例模式的用途远不止创建对象，比如我们通常渲染完页面中的一个列表之后，接下来要给这个列表绑定click 事件，如果是通过ajax 动态往列表里追加数据，在使用事件代理的前提下，click 事件实际上只需要在第一次渲染列表的时候被绑定一次，但是我们不想去判断当前是否是第一次渲染列表，如果借助于jQuery，我们通常选择给节点绑定one 事件：

```
var bindEvent = function(){
    $( 'div' ).one( 'click', function(){
        alert ( 'click' );
    });
};

var render = function(){
    console.log( '开始渲染列表' );
    bindEvent();
};

render();
render();
render();
```

如果利用getSingle 函数，也能达到一样的效果。代码如下：

```
var bindEvent = getSingle(function(){
    document.getElementById( 'div1' ).onclick = function(){
        alert ( 'click' );
    }
    return true;
});

var render = function(){
    console.log( '开始渲染列表' );
    bindEvent();
};

render();
render();
render();
```

可以看到，render 函数和bindEvent 函数都分别执行了3次，但div 实际上只被绑定了一个事件。