4 单例模式

定义：一个类仅有一个实例，并提供一个访问它的全局方法

例如线程池，全局缓存，浏览器中的window对象，还有点击Btn后弹出的登录浮窗，无论点击多少次这个浮窗应该至多被创建一次，是唯一的，这种时候就需要借助单例模式

实现单例模式

实现并不复杂，无非是用一个变量标志一下类是否创建过了对象，一个典型的实现如下：

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

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

Singleton.getInstance = (function(){
    // 闭包保存instance变量
    var instance = null;
    return function(name){
        if(!instance){
            instance = new Singleton(name)
        }
        return instance;
    }
})()

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

上面方式有一个问题，就是增加了这个类的“不透明性”，使用者必须知道它是一个单例类，有一个getInstance来调用才行

透明的单例模式

现在我们的需求是让用户使用这个类时候感知不到差异，实现一个“透明”单例类，这里我们可以引入代理类来专注于管理单例的逻辑，假设这里有一个创建div的工具类：

```
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)
}
```

我们可以引入代理类给他赋予单例的能力：

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

现在我们将管理单例的逻辑迁移了出去，通过组合实例单例效果

本例子是缓存代理的应用之一，在第六章将继续了解代理的好处

Javascript中的单例模式

前面几种实现其实是以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。

但JS其实是一门无类语言，既然我们只是需要一个“唯一”对象，为什么要为它先创建一个“类”呢，单例模式的核心是确保有有一个实例可以全局访问就行。

其实我们可能会习惯将全局变量当作单例来使用：

```
var a = {}
```

但是全局变量其实存在很多问题：

- 命名空间污染
- 变量容易被覆盖
- 全局变量甚至有时候被称为Javascript中最糟糕的特性

1. 使用命名空间进行优化

最简单的就是使用对象字面量的形式：

```
const namespace1 = {
    a: function(){},
    b: function(){}
}
```

2. 使用闭包封装私有变量

把变量封装在内部，只暴露接口跟外界通信

```
var user = (function(){
    var _name = 'sven',
        _age = 29

    return {
        getUserInfo:function(){
            return _name + '-' + _age;
        }
    }
})();
```

我们通过下划线来约定私有变量，外部无法访问，避免了对全局的污染

惰性单例

惰性单例指的是在实际需要的时候才创建对象实例，这种技术其实在实际开发中非常有用，也是单例模式的重点。

实际上本章开头演示的instance实例对象就是在调用instance.getInstance时候才会创建，而非页面创建好就创建。不过这种是基于“类”实现的单例，下面我们以WebQQ的登录浮窗为例，介绍结合全局变量实现惰性单例的方式

假设我们是WebQQ开发人员，进入网站后点击头像需要弹出登录浮窗，很明显浮窗在页面中应该总是唯一的，不可能出现同时存在两个登录窗口的情况

- 第一种方案就是在页面加载完成时候创建好这个浮窗，但是是隐藏状态，用户点击按钮之后改为显示

```
<html>
    <body>
        <button id={'login'}>登录</button>
    </body>
    <script>
        var loginLayer = (function(){
            var div = document.createElement('div')
            div.innerHTML = '我是登录窗口'
            div.style.display = 'none'
            document.body.appendChild(div)
            return div
        })()
        document.getElementById('login').onclick=function(){
            loginLayer.style.display='block'
        }
    </script>
</html>
```

但是主要问题就是如果不登陆的话，就白白浪费了DOM节点。

- 这里我们可以尝试利用一个变量来判断是否已经创建过登录浮窗

```
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)
        }
    }
})()

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

```

通用的惰性单例

上面的实现还是有一些问题：

- 代码违反单一职责原则，createLoginLayer既负责单例的管理，也负责业务逻辑
- 如果下次我们需要创建单例iframe或者script标签，那么必须如法炮制，copy这个函数在里面再修改

首先我们需要将管理单例的逻辑抽离，让其可以复用：

```
/**
 * 更加通用的惰性单例 - 抽离独立可复用的管理单例逻辑
 */
var getSingle = function(fn){
    // result身在闭包中，不会被销毁
    var result;
    return function(){
        return result || (result = fn.apply(this,arguments))
    }
}
```

接着将业务逻辑连接单例模式：

```
// 业务逻辑函数
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
}

// 连接
var createSingleLoginLayer = getSingle(createLoginLayer);

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

这种单例模式的用途远不止创建对象，在进行一些事件绑定很只希望重复一次的行为时候都可以用得上

小结

这一章我们学习到了因为语言差异性的单例模式不同实现，还提到了代理模式以及单一职责原则。

同时在getSingle函数中也提到了闭包和高阶函数的概念。特别是惰性单例还是很有用的。以及创建对象和管理单例的职责解耦和组合