招 聘 技 术 部

JavaScript 设计模式之单例模式

贾磊



DEEP THINKING 深入思考

**目录**

1. 定义
2. 实现单例模式
3. 透明的单例模式
4. 用代理实现单例模式
5. JavaScript中的单例模式
6. 惰性单例
7. 通用的惰性单例
8. 小结
9. 最后



DEEP THINKING 深入思考

**一、定义**

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。



DEEP THINKING 深入思考

**二、实现单例模式**



DEEP THINKING 深入思考

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：

// 单例模式

{

const Singleton = function ({ name } = {}) {

this.name = name;

this.instance = null;

}

Singleton.prototype.getName = function () {

return this.name;

}

Singleton.getInstance = function (params) {

if (!this.instance) {

this.instance = new Singleton(params)

}

return this.instance

}

const a = Singleton.getInstance({ name: "a" })

const b = Singleton.getInstance({ name: "b" })

console.log(a === b) // true

}



DEEP THINKING 深入思考

**二、实现单例模式**



DEEP THINKING 深入思考

或者：

// 单例模式

{

const Singleton = function ({ name } = {}) {

this.name = name;

}

Singleton.prototype.getName = function () {

return this.name;

}

Singleton.getInstance = (function () {

let instance = null;

return function (params) {

if (!instance) {

instance = new Singleton(params)

}

return instance

}

})()

const a = Singleton.getInstance({ name: "a" })

const b = Singleton.getInstance({ name: "b" })

console.log(a === b) // true

}

**二、实现单例模式**



DEEP THINKING 深入思考

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。

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。下面我们一步步去优化。

**三、透明的单例模式**



DEEP THINKING 深入思考

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。

// 透明的单例模式

{

const Singleton = (function () {

let instance = null;

const Singleton = function ({ name } = {}) {

if (instance) {

return instance

}

this.name = name

this.init()

instance = this

}

Singleton.prototype.init = function () {

console.log('init')

}

return Singleton

})()

const a = new Singleton({ name: "a" })

const b = new Singleton({ name: "b" })

console.log(a === b) // true

}

**三、透明的单例模式**



DEEP THINKING 深入思考

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。

在这段代码中，Singleton的构造函数实际上负责了两件事情。

第一是创建对象。

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Singleton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。

**四、用代理实现单例模式**



DEEP THINKING 深入思考

现在我们通过引入代理类的方式，来解决上面提到的问题。

// 用代理实现单例模式

{

const Singleton = function ({ name } = {}) {

this.name = name

this.init()

}

Singleton.prototype.init = function () {

console.log('init')

}

const proxySingleton = (function () {

let instance = null;

return function (params) {

if (!instance) {

instance = new Singleton(params)

}

return instance

}

})()

const a = new proxySingleton({ name: "a" })

const b = new proxySingleton({ name: "b" })

console.log(a === b) // true

}

**四、用代理实现单例模式**



DEEP THINKING 深入思考

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proxySingleton中。这样一来，Singleton就变成了一个普通的类，它跟proxySingleton组合起来可以达到单例模式的效果。

这也是缓存代理的应用之一。

**五、JavaScript中的单例模式**



DEEP THINKING 深入思考

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。

**五、JavaScript中的单例模式**



DEEP THINKING 深入思考

例如：

{

const Singleton = {}

}

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。

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。以下两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**
2. **使用闭包封装私有变量**

**五、JavaScript中的单例模式-使用命名空间**



DEEP THINKING 深入思考

**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。

{

// 对象字面量

const namespace1 = {

a: () => {

console.log('a')

},

b: () => {

console.log('b')

}

}

}

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。

**五、JavaScript中的单例模式-使用命名空间**



DEEP THINKING 深入思考

另外我们还可以**动态地创建命名空间**：

{

// 动态创建命名空间

const App = {}

App.namespace = function ({ name, value } = {}, content = App) {

const keys = name.split('.')

for (let key in keys) {

if (!content[keys[key]]) content[keys[key]] = {}

if (JSON.parse(key) === keys.length - 1) content[keys[key]] = value

content = content[keys[key]]

}

}

App.namespace({ name: 'namespace1' })

App.namespace({ name: 'namespace2.a', value: function () { console.log('a') } })

console.log(App)

/\* {

namespace: [Function],

namespace1: undefined,

namespace2: { a: [Function: value] }

}

\*/

}

**五、JavaScript中的单例模式-使用闭包封装私有变量**



DEEP THINKING 深入思考

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：

{

// 使用闭包封装私有变量

const App = (function () {

const \_namespace1 = undefined

const \_namespace2 = {

a: function () { console.log('a') }

}

return {

namespace1: \_namespace1,

namespace2: \_namespace2

}

})()

console.log(App) // { namespace1: undefined, namespace2: { a: [Function: a] } }

}

我们用下划线来约定私有变量\_namespace1和\_namespace2，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。

**六、惰性单例**



DEEP THINKING 深入思考

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，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：

{

// 惰性单例

const Singleton = function ({ name } = {}) {

this.name = name

}

Singleton.getInstance = (function () {

let instance = null

return function (params) {

if (!instance) {

instance = new Singleton(params)

}

return instance

}

})()

const a = Singleton.getInstance({ name: 'a' })

const b = Singleton.getInstance({ name: 'b' })

console.log(a === b) // true

}

**六、惰性单例**



DEEP THINKING 深入思考

不过这是基于“类”的单例模式，基于“类”的单例模式在JavaScript中并不适用。

JavaScript中，更热衷于**与全局变量结合**实现惰性的单例。接下来以**登录弹窗**为例，介绍与全局变量结合实现惰性的单例。

假设我们现在有一个网站，某些功能需要用户登录后才能使用。所以我们会有一个登录弹窗，很明显这个弹窗在页面里总是唯一的，不可能出现同时存在两个登录弹窗的情况。

**六、惰性单例**



DEEP THINKING 深入思考

第一种解决方案是在页面加载完成的时候便创建好这个弹窗，这个弹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：

{

const popupEl = (function () {

//todo 创建弹窗

const div = document.createElement('div')

div.innerText = '返回弹窗实例'

div.style.display = 'none'

document.body.appendChild(div)

return div

})()

document.body.onclick = function () {

popupEl.style.display = 'block'

}

}

这种方式有一个问题，也许我们进入网站只是随便看看，根本不需要进行登录操作，因为登录弹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。

**六、惰性单例**



DEEP THINKING 深入思考

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。

{

const popupEl = function () {

//todo 创建弹窗

const div = document.createElement('div')

div.innerText = '返回弹窗实例'

div.style.display = 'none'

document.body.appendChild(div)

return div

}

document.body.onclick = function () {

const instance = popupEl()

instance.style.display = 'block'

}

}

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。

**六、惰性单例**



DEEP THINKING 深入思考

我们可以用一个变量来判断是否已经创建过登录浮窗。

{

const popupEl = (function () {

let instance = null

return function () {

if (!instance) {

//todo 创建弹窗

const div = document.createElement('div')

div.innerText = '返回弹窗实例'

div.style.display = 'none'

document.body.appendChild(div)

instance = div

}

return instance

}

})()

document.body.onclick = function () {

const instance = popupEl()

instance.style.display = 'block'

}

}

**七、通用的惰性单例**



DEEP THINKING 深入思考

前面实现了一个惰性单例，但是代码违反单一职责原则的，创建对象和管理单例的逻辑都放在popupEl对象内部。

我们需要把不变的部分隔离出来，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：

{

const obj = null;

if (!obj) {

obj = 'todo create instance'

}

}

**七、通用的惰性单例**



DEEP THINKING 深入思考

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getInstance函数内部，创建对象的方法fn被当成参数动态传入getInstance函数：

{

const getInstance = function (fn) {

let result = null;

return function () {

return result || fn.apply(this, arguments)

}

}

}

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。

**八、小结**



DEEP THINKING 深入思考

在getInstance函数中，实际上也提到了闭包和高阶函数的概念。

单例模式是一种简单但非常实用的模式，特别是惰性单例技术，在合适的时候才创建对象，并且只创建唯一的一个。

更奇妙的是，创建对象和管理单例的职责被分布在两个不同的方法中，这两个方法组合起来才具有单例模式的威力。

**九、最后**



DEEP THINKING 深入思考

最后看个单例模式的应用——vant 组件的 Toast。

默认是单例模式。

let queue: ComponentInstance[] = [];

let allowMultiple = false;

function getInstance() {

if (!queue.length || allowMultiple) {

const instance = createInstance();

queue.push(instance);

}

return queue[queue.length - 1];

}

通过调用Toast.allowMultiple() 可以开启多实例。

Toast.allowMultiple = (value = true) => {

allowMultiple = value;

};