DAY23 设计模式

2023年5月19日 18:42

单例模式

单例模式的定义是：保证一个类仅有一个实例(实例对象)，并提供一个访问它的全局访问点。

面向对象的核心：通过类创建多个具有相同属性和方法的实例对象

矛盾：一个实例对象　多个对象

比如：页面设计二十个弹框

1.设计二十个框(20个div)，每个框对应不同的内容，彼此没有关系。

2.设计一种框(1个div)，里面设置不同的内容(单例的一种应用)

```
class Single {
  constructor(text) {
    let cDiv = document.createElement('div');
    cDiv.innerHTML = text;
    document.body.appendChild(cDiv);
  }
}
```

只要new了两次，不可能产生一个实例，必须有两个

设置标记，接收实例化的对象，如果实例化对象不存在，实例化一次，如果存在，直接返回

```
let flag = null;
function dialog(text) {
  if (!flag) {
    flag = new Single(text);
  }
  return flag;
}
let d1 = dialog('zhangsan');
let d2 = dialog('lisi');
console.log(d1 === d2);
```

上面的写法可以保证实例化一次，产生问题

1.无法设置不同的内容，因为参数给了类名，第一次执行，第二次不执行，直接返回

解决方式：利用类里面的方法

2.全局变量

解决方式：闭包

3.一个函数解决

```
const Dialog = (function () {//自执行了
```

```
  class Dialog {
    constructor() {
      this.cDiv = document.createElement('div');
      document.body.appendChild(this.cDiv);
    }
    init(txt) {
      this.cDiv.innerHTML = txt;
    }
  }
  let flag = null;
  return function (txt) {
    if (!flag) flag = new Dialog();
    flag.init(txt);
    return flag;
  }
})();


// new：自动隐式返回对象，如果函数里面的return返回的是引用类型，new没有意义
let d1 = new Dialog('zhangsan');
let d2 = new Dialog('lisi');
let d3 = new Dialog('wangwu');
let d4 = new Dialog('zhaoliu')
console.log(d1 === d4);//true
```

注意：

new调用函数,自动隐式返回对象

new调用函数，如何函数内部返回的是对象类型，最终new没有意义

new调用函数，如何函数内部返回的是基本类型，基本类型无效的，最终返回的还是对象

```
function fn() {
  return 100;
}
console.log(fn());//100
console.log(new fn());//fn {}    new调用函数，自动隐式返回对象
```

策略模式

策略模式的定义是：定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。

案例.使用策略模式计算奖金

A:3倍 B:4倍 C:5倍.....

1. 最初的代码实现

```
const calcSalary = function (level, salary) {//level：等级     salary:奖金
  if (level === 'A') {
    return salary * 3
  }
```

```
  if (level === 'B') {
    return salary * 4
  }
  if (level === 'C') {
    return salary * 5
  }
};
console.log(calcSalary('B', 3000));//12000
console.log(calcSalary('C', 3000));//15000
```

开放-封闭原则

开放封闭原则的思想：当需要改变一个程序的功能或者给这个程序增加新功能的时候，可以使用增加代码的方式，但是不允许改动程序的源代码。

开放：针对的是程序扩展或者程序配置项

封闭：封闭的是函数(封装的源代码)

2. 使用组合函数重构代码

```
const levelA = function (salary) {
  return salary * 3;
};
const levelB = function (salary) {
  return salary * 4;
};
const levelC = function (salary) {
  return salary * 5;
};
const levelD = function (salary) {
  return salary * 6;
};
const calcSalary = function (level, salary) {
  if (level === 'A') {
    return levelA(salary)
  }
  if (level === 'B') {
    return levelB(salary)
  }
  if (level === 'C') {
    return levelC(salary)
  }
  if (level === 'D') {
    return levelD(salary)
  }
};
console.log(calcSalary('A', 3000));
console.log(calcSalary('B', 3000));
console.log(calcSalary('D', 3000));
```

3. 使用策略模式重构代码 - 符合开放-封闭原则

```
const levelObj = {
  A(salary) {
    return salary * 3;
  },
  B(salary) {
    return salary * 4;
  },
  C(salary) {
    return salary * 5;
  },
  D(salary) {
    return salary * 6;
  }
}
const calcSalary = function (level, salary) {
  return levelObj[level](salary);
}
console.log(calcSalary('A', 3000));
console.log(calcSalary('B', 3000));
console.log(calcSalary('D', 3000));
```

发布订阅模式

发布 — 订阅模式又叫观察者模式，

它定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知。

在 JavaScript开发中，我们一般用事件模型来替代传统的发布 — 订阅模式

事件绑定

```
document.addEventListener('click', function () {
  console.log('绑定的第一个事件');
});
document.addEventListener('click', function () {
  console.log('绑定的第二个事件');
});
document.addEventListener('click', function () {
  console.log('绑定的第三个事件');
});

function fn1() {
  console.log('绑定的第一个事件');
}
function fn2() {
```

```
  console.log('绑定的第二个事件');
}
function fn3() {
  console.log('绑定的第三个事件');
}

document.addEventListener('click', fn1);
document.addEventListener('click', fn2);
document.addEventListener('click', fn3);
```

普通事件模拟事件绑定

```
let arr = [fn1, fn2, fn3, fn4, fn5];//函数名等于函数指针，就是函数体
document.onclick = function () {
  arr.forEach(item => item())
};
```

封装自定义事件(事件类型，事件处理函数)

事件类型：买菜，买水果，买房子...

事件处理函数：自定义函数

增加事件类型以及事件处理函数

正在进行时(触发事件，执行事件处理函数)

删除事件

```
class Observer {
  constructor() {
    this.message = {};//记事本
    /*
    格式
    this.message = {
      '买菜':[function(){买白菜},function(){买青菜}],
      '买水果':[function(){买苹果},function(){买橙子},function(){买香蕉}]
    }
    */
  }
  // 添加事件
  add(type, fn) {//type事件类型，fn事件处理函数
    // 判断当前的事件类型是否存在，如果不存在，创建事件类型，如果存在追加事件
    if (!this.message[type]) {
      this.message[type] = [fn];
    } else {
      this.message[type].push(fn);
```

```
    }
  }
  // 执行事件
  emit(type) {//type事件的类型
    if (!this.message[type]) return;
    this.message[type].forEach(item => item())
  }
  // 删除事件
  remove(type, fn) {
    this.message[type] = this.message[type].filter(item => item != fn);
  }
}
let observer = new Observer();
function fn1() {
  console.log('买白菜...');
}
function fn2() {
  console.log('买青菜...');
}
observer.add('buycai', fn1);
observer.add('buycai', fn2);
observer.emit('buycai');//执行，进行中
console.log('----------------------------------------');
observer.remove('buycai', fn1);//完成买菜的第一个事件
```

浏览器垃圾回收机制

浏览器内存泄漏

是指程序中已动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

内存泄漏简单解读：变量不再被使用，然后也无法回收，形成内存泄漏

1.意外的全局变量引起的内存泄露

```
// var num = 1;//全局变量，一直存在，不会消失
```

2.滥用闭包引起的内存泄露

```
// function fn() {
//   var i = 1;
//   return function () {
//     return i;
//   }
// }
// var f1 = fn();
// f1();//1
```

```
// f1 = null;//防止内存泄漏
```

3.被遗忘的定时器或者回调 - 性能问题

4.对象循环引用

```
// function fn() {
//   var i = 1;
//   var obj = { num: i }
//   var obj1 = obj;
//   return obj1
// }
```

5.隐式的全局变量

```
// function fn1() {
//   num = 10;//没有声明的关键字，默认为全局变量，全局属性
//   this.num = 100;//this->window
// }
// fn1();
// console.log(num);
// console.log(window.num);
```