Javascript常见设计模式

> 设计模式总的来说是一个抽象的概念，是软件开发人员在开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。

1、工厂模式

工厂模式是用来创建对象的一种最常用的设计模式。我们不暴露创建对象的具体逻辑，而是将将逻辑封装在一个函数中，那么这个函数就可以被视为一个工厂。

```
// 简单工厂模式：只需要一个正确的参数，就可以获取到你所需要的对象，用于生成实例
function Animal(opts){
    var obj = new Object();
    obj.color = opts.color;
    obj.name= opts.name;
    obj.getInfo = function(){
        return '名称：'+ obj.name+'， 颜色：'+ obj.color;
    }
    return obj;
}
var cat = Animal({name: '波斯猫', color: '白色'});

// 工厂方法模式：本意是将实际创建对象的工作推迟到子类中，这样核心类就变成了抽象类，
// 工厂方法只是一个实例化对象的工厂，只做实例化对象这一件事情，也是用于生成实例
class User {
  constructor(name = '', viewPage = []) {
    if(new.target === User) {
      throw new Error('抽象类不能实例化!');
    }
    this.name = name;
    this.viewPage = viewPage;
  }
}

class UserFactory extends User {
  constructor(name, viewPage) {
    super(name, viewPage) // 调用父类的constructor(name, viewPage)，继承父类的this对象
  }
  create(role) {
    switch (role) {
      case 'superAdmin':
        return new UserFactory( '超级管理员', ['首页', '通讯录', '发现页', '应用数据', '权限管理'] );
        break;
      case 'admin':
        return new UserFactory( '普通管理员', ['首页', '通讯录', '发现页'] );
```

```
        break;
      case 'user':
        return new UserFactory( '普通用户', ['首页', '通讯录', '发现页'] );
        break;
      default:
        throw new Error('参数错误, 可选参数:superAdmin、admin、user')
    }
  }
}

let userFactory = new UserFactory();
let superAdmin = userFactory.create('superAdmin');
let admin = userFactory.create('admin');
let user = userFactory.create('user');


//抽象工厂模式：并不直接生成实例，而是用于对产品类簇的创建
function getAbstractUserFactory(type) {
  switch (type) {
    case 'wechat':
      return UserOfWechat;
      break;
    case 'qq':
      return UserOfQq;
      break;
    case 'weibo':
      return UserOfWeibo;
      break;
    default:
      throw new Error('参数错误, 可选参数:wechat、qq、weibo')
  }
}

let WechatUserClass = getAbstractUserFactory('wechat');
let QqUserClass = getAbstractUserFactory('qq');
let WeiboUserClass = getAbstractUserFactory('weibo');

let wechatUser = new WechatUserClass('微信小李');
let qqUser = new QqUserClass('QQ小李');
let weiboUser = new WeiboUserClass('微博小李');
```

2、单例模式

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。

```
class Singleton {
  constructor() {}
}

Singleton.getInstance = (function() {
```

```
  let instance
  return function() {
    if (!instance) {
      instance = new Singleton()
    }
    return instance
  }
})()

let s1 = Singleton.getInstance()
let s2 = Singleton.getInstance()
console.log(s1 === s2) // true
```

在vuex源码中，通过一个外部变量来控制全局只有一个Vue实例：

```
let Vue // bind on install
export function install (_Vue) {
  if (Vue && _Vue === Vue) {
    // 如果发现 Vue 有值，就不重新创建实例了
    return
  }
  Vue = _Vue
  applyMixin(Vue)
}
```

3、适配器模式

适配器用来解决两个接口不兼容的情况，不需要改变已有的接口，通过包装一层的方式实现两个接口的正常协作。

```
// 已有的地图接口
var googleMap = {
    show: function(){
        console.log( '开始渲染谷歌地图' );
    }
};
var baiduMap = {
    display: function(){
        console.log( '开始渲染百度地图' );
    }
};
// 已有的渲染接口
var renderMap = function( map ){
    if ( map.show instanceof Function ){
        map.show();
    }
};
```

```
// 目的
renderMap( googleMap ); // 开始渲染谷歌地图
renderMap( baiduMap ); // 无效


// 适配器
var baiduMapAdapter = {
    show: function(){
        return baiduMap.display();
    }
};
renderMap( googleMap ); // 开始渲染谷歌地图
renderMap( baiduMapAdapter ); // 开始渲染百度地图
```

4、装饰模式

装饰模式不需要改变已有的接口，给对象添加额外的功能。比如使用ES7 中的装饰器语法：

```
function readonly(target, key, descriptor) {
  descriptor.writable = false
  return descriptor
}

class Test {
  @readonly
  name = 'yck'
}

let t = new Test()
t.yck = '111' // 不可修改
```

5、代理模式

在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在两个对象之间起到中介的作用。

```
class Car {
    drive() {
        return "driving";
    };
}
```

```
class Driver {
    constructor(age) {
        this.age = age;
    }
}

class CarProxy {
    constructor(driver) {
        this.driver = driver;
    }
    drive() {
        return  ( this.driver.age < 18) ? "too young to drive" : new Car().drive();
    };
}
const driver = new Driver(18)
const carProxy = new CarProxy(driver).drive() // driving
```

此外，还有es6 proxy，也就是在目标对象之前架设一层拦截，或者叫代理：

```
var obj = {}
var proxy = new Proxy(obj, {
  get: function (target, key, receiver) {
    console.log(`getting ${key}!`);
    return Reflect.get(target, key, receiver);
  },
  set: function (target, key, value, receiver) {
    console.log(`setting ${key}!`);
    return Reflect.set(target, key, value, receiver);
  }
});
proxy.count = 1
//  setting count!
++proxy.count
//  getting count!
//  setting count!
//  2
console.log(obj.count) // 2
```

6、观察者模式

定义了对象间一对多的依赖关系，当目标对象的状态发生改变时，所有依赖它的对象都会得到通知。

```
// 目标者类
class Subject {
  constructor() {
```

```
    this.observers = [];  // 观察者列表
  }
  // 添加
  add(observer) {
    this.observers.push(observer);
  }
  // 删除
  remove(observer) {
    let idx = this.observers.findIndex(item => item === observer);
    idx > -1 && this.observers.splice(idx, 1);
  }
  // 通知
  notify() {
    for (let observer of this.observers) {
      observer.update();
    }
  }
}

// 观察者类
class Observer {
  constructor(name) {
    this.name = name;
  }
  // 目标对象更新时触发的回调
  update() {
    console.log(`目标者通知我更新了，我是：${this.name}`);
  }
}

// 实例化目标者
let subject = new Subject();

// 实例化两个观察者
let obs1 = new Observer('前端开发者');
let obs2 = new Observer('后端开发者');

// 向目标者添加观察者
subject.add(obs1);
subject.add(obs2);

// 目标者通知更新
subject.notify();
// 输出：
// 目标者通知我更新了，我是前端开发者
// 目标者通知我更新了，我是后端开发者
```

vue源码中的依赖管理和通知更新就是基于观察者模式。

7、发布订阅模式

实现了对象间多对多的依赖关系，通过事件中心管理多个事件。目标对象并不直接通知观察者，而是通过事件中心来派发通知。

```
// 事件中心
let pubSub = {
  list: {}, // {onwork:[fn1,fn2],offwork:[fn1,fn2],launch:[fn1,fn2]}
  subscribe: function (key, fn) {   // 订阅
    if (!this.list[key]) {
      this.list[key] = [];
    }
    this.list[key].push(fn);
  },
  publish: function(key, ...arg) {  // 发布
    for(let fn of this.list[key]) {
      fn.call(this, ...arg);
    }
  },
  unSubscribe: function (key, fn) {     // 取消订阅
    let fnList = this.list[key];
    if (!fnList) return false;

    if (!fn) {
      // 不传入指定取消的订阅方法，则清空所有key下的订阅
      fnList && (fnList.length = 0);
    } else {
      fnList.forEach((item, index) => {
        if (item === fn) {
          fnList.splice(index, 1);
        }
      })
    }
  }
}

// 订阅
pubSub.subscribe('onwork', time => {
  console.log(`上班了：${time}`);
})
pubSub.subscribe('offwork', time => {
  console.log(`下班了：${time}`);
})
pubSub.subscribe('launch', time => {
  console.log(`吃饭了：${time}`);
})

// 发布
pubSub.publish('offwork', '18:00:00');
pubSub.publish('launch', '12:00:00');

// 取消订阅
pubSub.unSubscribe('onwork');
```

参考链接：

https://juejin.cn/post/6844903653774458888#heading-3

https://refactoringguru.cn/design-patterns/catalog

http://interview.poetries.top/docs/excellent.html#_34-设计模式

参考书籍：《JavaScript设计模式与开发实践》