第九课 工厂模式

学习目录

- **使用场景**
- **使用方式**
- **优缺点**

一. 使用场景

工厂模式之前我们有简单提到过，实际上它的作用就是构建出属于何种类型的对象。说道构造函数，我们可以通过 new 运算符来实例化构造函数，但是有些时候，我们的对象之间耦合性大、业务逻辑逐渐复杂，这时候仅仅依靠 new 运算符实例化构造函数已经满足不了我们的需求了，这时候可能需要用到工厂模式来解决对象耦合较多带来的问题。工厂模式的适合用来增加代码的健壮性，优化代码结构，在需要对代码重构的时候可以考虑工厂模式来调整优化代码中对象构建过程。

又比如说我们有几个构造函数，有时候我们可以直接通过 new 运算符来实例化构造函数，但是当我们不知道该实例化哪个构造函数的时候，那我们可以把实例化的过程推迟到运行期间在来决定，因此我们可以用来工厂模式帮助我们实例化构造函数得到对应类型的实例对象。

工厂模式的核心思想是把构造函数实例化放在运行期间来进行，同时根据运行需要得到对应类型实例对象。

二．使用方式

工厂模式也可以分成简单工厂模式和复杂工厂模式，简单工厂模式是使用一个对象来构建另一个对象的模式，复杂工厂模式是把实例化的过程推迟到子类来构建对象的模式。

简单工厂模式

```
function Fn1(num){
    this.num = num;
}
Fn1.prototype.getNum = function(){
    return this.num + 'Fn1';
}

function Fn2(num){
    this.num = num;
}
Fn2.prototype.getNum = function(){
    return this.num + 'Fn2';
}

function Fn3(num){
    this.num = num;
}
Fn3.prototype.getNum = function(){
```

```
    return this.num + 'Fn3';
}

var factoryFn = {
    createObj:function(type){
        var obj = null;
        switch (type) {
            case 'Fn1':
                obj = new Fn1(100);
                break;
            case 'Fn2':
                obj = new Fn2(300);
                break;
            case 'Fn3':
                obj = new Fn3(600);
                break;
            default:
                break;
        }
        return obj;
    }
}
```

```
function NumCenter(){}
NumCenter.prototype.showNum = function(type){
    var numObj = factoryFn.createObj(type);
    return numObj;
}

var center = new NumCenter();
var number1 = center.showNum('Fn1');
var number2 = center.showNum('Fn2');
var number3 = center.showNum('Fn3');
console.log(number1.getNum());
console.log(number2.getNum());
console.log(number3.getNum());
```

复杂工厂模式

```
/*A 类*/
function FnA1(num){
    this.num = num;
}
FnA1.prototype.getNum = function(){
    return this.num + 'FnA1';
```

```
}


function FnA2(num){
    this.num = num;
}
FnA2.prototype.getNum = function(){
    return this.num + 'FnA2';
}


function FnA3(num){
    this.num = num;
}
FnA3.prototype.getNum = function(){
    return this.num + 'FnA3';
}


/*B 类*/
function FnB1(num){
    this.num = num;
}
FnB1.prototype.getNum = function(){
    return this.num + 'FnB1';
```

```
}


function FnB2(num){
    this.num = num;
}
FnB2.prototype.getNum = function(){
    return this.num + 'FnB2';
}


function FnB3(num){
    this.num = num;
}
FnB3.prototype.getNum = function(){
    return this.num + 'FnB3';
}


function NumCenter(){}
NumCenter.prototype.showNum = function(type){
    var numObj = this.createNum(type);
    return numObj;
}
NumCenter.prototype.createNum = function(){
```

```
    throw new Error('子类决定如何显示数字');
}

function NumSubA(){
    NumCenter.apply(this);
}
NumSubA.prototype.createNum = function(type,num){
    var numObj = null;
    switch (type) {
        case 'FnA1':
            numObj = new FnA1(num);
            break;
        case 'FnA2':
            numObj = new FnA2(num);
            break;
        case 'FnA3':
            numObj = new FnA3(num);
            break;
        default:
            break;
    }
    return numObj;
```

```
}
var nA1 = new NumSubA();
var numberA1 = nA1.createNum('FnA1',100);
var numberA2 = nA1.createNum('FnA2',200);
var numberA3 = nA1.createNum('FnA3',300);
console.log(numberA1.getNum());//100FnA1
console.log(numberA2.getNum());//200FnA2
console.log(numberA3.getNum());//300FnA3

function NumSubB(){
    NumCenter.apply(this);
}
NumSubB.prototype.createNum = function(type,num){
    var numObj = null;
    switch (type) {
        case 'FnB1':
            numObj = new FnB1(num);
            break;
        case 'FnB2':
            numObj = new FnB2(num);
            break;
        case 'FnB3':
```

```
                numObj = new FnB3(num);
                break;
            default:
                break;
        }
        return numObj;
    }
    var nB1 = new NumSubB();
    var numberB1 = nB1.createNum('FnB1',1000);
    var numberB2 = nB1.createNum('FnB2',2000);
    var numberB3 = nB1.createNum('FnB3',3000);
    console.log(numberB1.getNum());//1000FnB1
    console.log(numberB2.getNum());//2000FnB2
    console.log(numberB3.getNum());//3000FnB3
```

三．优缺点

1.工厂模式适合用在需要大量构建重复对象的场景下，不同的构造函数需要多次实例化，这时候可以考虑使用工厂模式。

2.工厂模式能简化对象的构建过程，抽离对象的构建过程，属于比较底层的设计模式，适合运用在基础组件的构建过程中。

3.工厂模式一般在优化代码结构增加代码健壮性的时候可以考虑使用，并不适合面向过

程的常规开发过程中。

3.工厂模式不适合在常规开发中大量使用，它会增加代码的复杂度，同时不利于代码的维护。

谢谢观看！

我是星星课堂老师：周小周