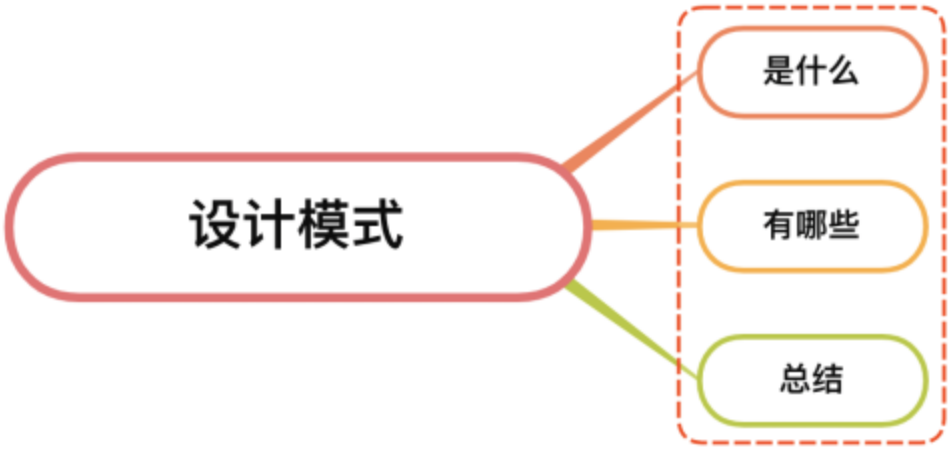
# 设计模式面试真题（6题）

## 1. 说说对设计模式的理解？常见的设计模式有哪些?




### 1.1. 是什么

在软件工程中，设计模式是对软件设计中普遍存在的各种问题所提出的解决方案

设计模式并不直接用来完成代码的编写，而是描述在各种不同情况下，要怎么解决问题的一种方案

设计模式能使不稳定依赖于相对稳定、具体依赖于相对抽象，避免会引起麻烦的紧耦合，以增强软件设计面对并适应变化的能力

因此，当我们遇到合适的场景时，我们可能会条件反射一样自然而然想到符合这种场景的设计模式

比如，当系统中某个接口的结构已经无法满足我们现在的业务需求，但又不能改动这个接口，因为可能原来的系统很多功能都依赖于这个接口，改动接口会牵扯到太多文件

因此应对这种场景，我们可以很快地想到可以用适配器模式来解决这个问题

### 1.2. 有哪些

常见的设计模式有：

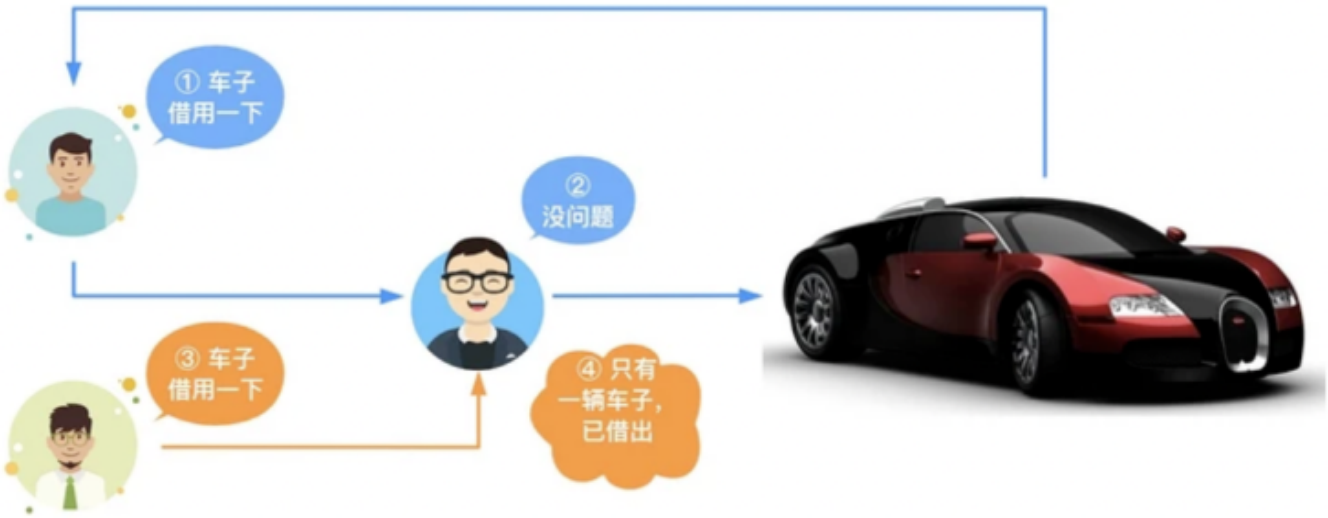
- 单例模式
- 工厂模式
- 策略模式
- 代理模式

- 中介者模式
- 装饰者模式
- ……

### 1.2.1. 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。实现的方法为先判断实例存在与否，如果存在则直接返回，如果不存在就创建了再返回，这就确保了一个类只有一个实例对象

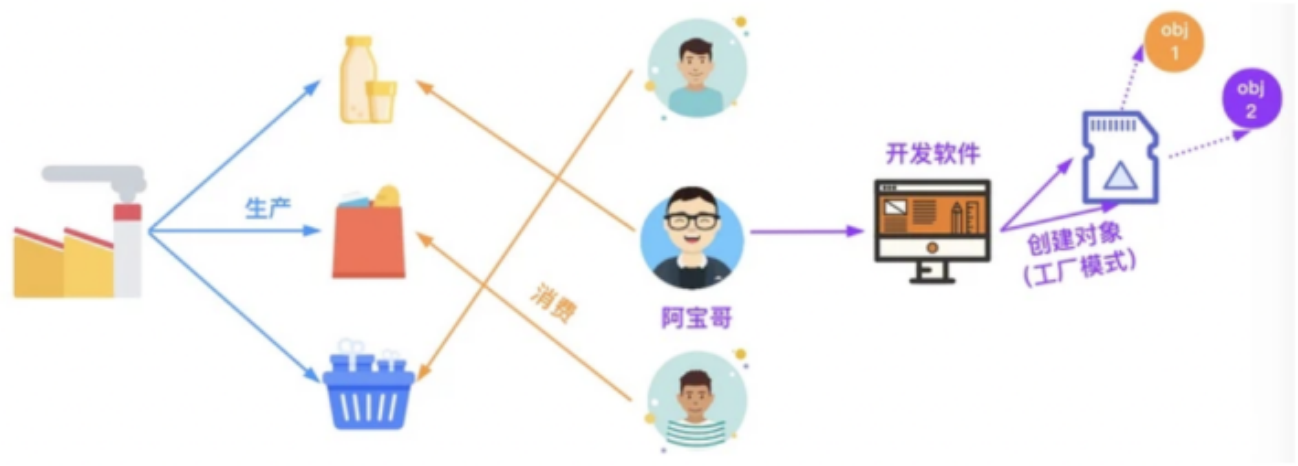
如下图的车，只有一辆，一旦借出去则不能再借给别人：




### 1.2.2. 工厂模式

工厂模式通常会分成3个角色：

- 工厂角色–负责实现创建所有实例的内部逻辑.
- 抽象产品角色–是所创建的所有对象的父类，负责描述所有实例所共有的公共接口
- 具体产品角色–是创建目标，所有创建的对象都充当这个角色的某个具体类的实例




### 1.2.3. 策略模式

策略模式，就是定义一系列的算法，把他们一个个封装起来，并且使他们可以相互替换

至少分成两部分：

- 策略类（可变），策略类封装了具体的算法，并负责具体的计算过程
- 环境类（不变），接受客户的请求，随后将请求委托给某一个策略类

### 1.2.4. 代理模式

代理模式：为对象提供一个代用品或占位符，以便控制对它的访问

例如实现图片懒加载的功能，先通过一张 `loading` 图占位，然后通过异步的方式加载图片，等图片加载好了再把完成的图片加载到 `img` 标签里面

### 1.2.5. 中介者模式

中介者模式的定义：通过一个中介者对象，其他所有的相关对象都通过该中介者对象来通信，而不是相互引用，当其中的一个对象发生改变时，只需要通知中介者对象即可

通过中介者模式可以解除对象与对象之间的紧耦合关系

### 1.2.6. 装饰者模式

装饰者模式的定义：在不改变对象自身的基础上，在程序运行期间给对象动态地添加方法

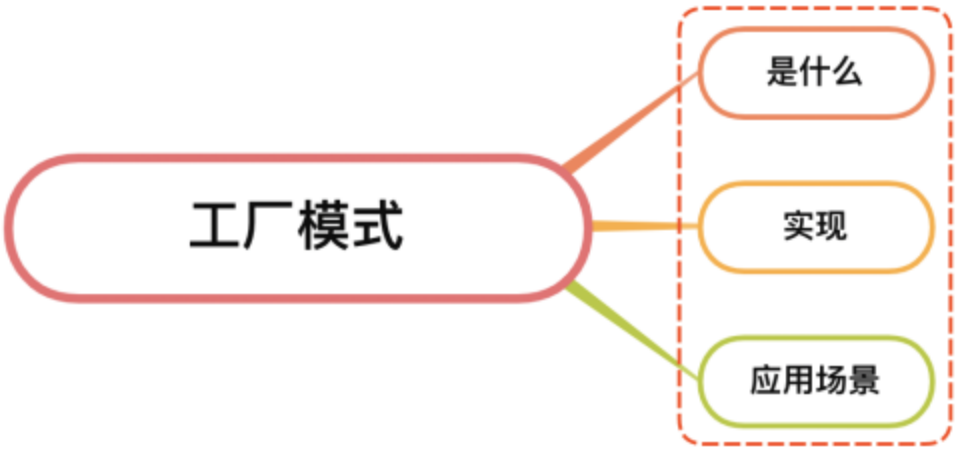
通常运用在原有方法维持不变，在原有方法上再挂载其他方法来满足现有需求

## 1.3. 总结

不断去学习设计模式，会对我们有着极大的帮助，主要如下：

- 从许多优秀的软件系统中总结出的成功的、能够实现可维护性、复用的设计方案，使用这些方案将可以让我们避免做一些重复性的工作
- 设计模式提供了一套通用的设计词汇和一种通用的形式来方便开发人员之间沟通和交流，使得设计方案更加通俗易懂
- 大部分设计模式都兼顾了系统的可重用性和可扩展性，这使得我们可以更好地重用一些已有的设计方案、功能模块甚至一个完整的软件系统，避免我们经常做一些重复的设计、编写一些重复的代码
- 合理使用设计模式并对设计模式的使用情况进行文档化，将有助于别人更快地理解系统
- 学习设计模式将有助于初学者更加深入地理解面向对象思想

# 2. 说说你对工厂模式的理解？应用场景？




## 2.1. 是什么

工厂模式是用来创建对象的一种最常用的设计模式，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

其就像工厂一样重复的产生类似的产品，工厂模式只需要我们传入正确的参数，就能生产类似的产品

举个例子：

- 编程中，在一个 A 类中通过 new 的方式实例化了类 B，那么 A 类和 B 类之间就存在关联（耦合）
- 后期因为需要修改了 B 类的代码和使用方式，比如构造函数中传入参数，那么 A 类也要跟着修改，

一个类的依赖可能影响不大，但若有多个类依赖了 B 类，那么这个工作量将会相当的大，容易出现修改错误，也会产生很多的重复代码，这无疑是件非常痛苦的事；

- 这种情况下，就需要将创建实例的工作从调用方（A类）中分离，与调用方**解耦**，也就是使用工厂方法创建实例的工作封装起来（**减少代码重复**），由工厂管理对象的创建逻辑，调用方不需要知道具体的创建过程，只管使用，**而降低调用者因为创建逻辑导致的错误**；

## 2.2. 实现

工厂模式根据抽象程度的不同可以分为：

- 简单工厂模式（Simple Factory）
- 工厂方法模式（Factory Method）
- 抽象工厂模式（Abstract Factory）

### 2.2.1. 简单工厂模式

简单工厂模式也叫静态工厂模式，用一个工厂对象创建同一类对象类的实例

假设我们要开发一个公司岗位及其工作内容的录入信息，不同岗位的工作内容不一致

代码如下：

```javascript
function Factory(career) {
    function User(career, work) {
        this.career = career
        this.work = work
    }
    let work
    switch(career) {
        case 'coder':
            work =  ['写代码', '修Bug']
            return new User(career, work)
            break
        case 'hr':
            work = ['招聘', '员工信息管理']
            return new User(career, work)
            break
        case 'driver':
            work = ['开车']
            return new User(career, work)
            break
        case 'boss':
            work = ['喝茶', '开会', '审批文件']
            return new User(career, work)
            break
    }
}
let coder = new Factory('coder')
console.log(coder)
let boss = new Factory('boss')
console.log(boss)
```

`Factory` 就是一个简单工厂。当我们调用工厂函数时，只需要传递name、age、career就可以获取到包含用户工作内容的实例对象

### 2.2.2. 工厂方法模式

工厂方法模式跟简单工厂模式差不多，但是把具体的产品放到了工厂函数的 `prototype` 中

这样一来，扩展产品种类就不必修改工厂函数了，和心累就变成抽象类，也可以随时重写某种具体的产品

也就是相当于工厂总部不生产产品了，交给下辖分工厂进行生产；但是进入工厂之前，需要有个判断来验证你要生产的东西是否是属于我们工厂所生产范围，如果是，就丢给下辖工厂来进行生产

如下代码：

JavaScript | 复制代码

```javascript
// 工厂方法
function Factory(career){
    if(this instanceof Factory){
        var a = new this[career]();
        return a;
    }else{
        return new Factory(career);
    }
}
// 工厂方法函数的原型中设置所有对象的构造函数
Factory.prototype={
    'coder': function(){
        this.careerName = '程序员'
        this.work = ['写代码', '修Bug']
    },
    'hr': function(){
        this.careerName = 'HR'
        this.work = ['招聘', '员工信息管理']
    },
    'driver': function () {
        this.careerName = '司机'
        this.work = ['开车']
    },
    'boss': function(){
        this.careerName = '老板'
        this.work = ['喝茶', '开会', '审批文件']
    }
}
let coder = new Factory('coder')
console.log(coder)
let hr = new Factory('hr')
console.log(hr)
```

工厂方法关键核心代码是工厂里面的判断this是否属于工厂，也就是做了分支判断，这个工厂只做我能做的产品

## 2.2.3. 抽象工厂模式

上述简单工厂模式和工厂方法模式都是直接生成实例，但是抽象工厂模式不同，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

通俗点来讲就是：简单工厂和工厂方法模式的工作是生产产品，那么抽象工厂模式的工作就是生产工厂的

由于 `JavaScript` 中并没有抽象类的概念，只能模拟，可以分成四部分：

- 用于创建抽象类的函数
- 抽象类
- 具体类
- 实例化具体类

上面的例子中有 `coder` 、 `hr` 、 `boss` 、 `driver` 四种岗位，其中 `coder` 可能使用不同的开发语言进行开发，比如 `JavaScript` 、 `Java` 等等。那么这两种语言就是对应的类簇

示例代码如下：

JavaScript | 复制代码

```
let CareerAbstractFactory = function(subType, superType) {
  // 判断抽象工厂中是否有该抽象类
  if (typeof CareerAbstractFactory[superType] === 'function') {
    // 缓存类
    function F() {}
    // 继承父类属性和方法
    F.prototype = new CareerAbstractFactory[superType]()
    // 将子类的constructor指向父类
    subType.constructor = subType;
    // 子类原型继承父类
    subType.prototype = new F()
  } else {
    throw new Error('抽象类不存在')
  }
}
```

上面代码中 `CareerAbstractFactory` 就是一个抽象工厂方法，该方法在参数中传递子类和父类，在方法体内部实现了子类对父类的继承

## 2.3. 应用场景

从上面可看到，简单简单工厂的优点就是我们只要传递正确的参数，就能获得所需的对象，而不需要关心其创建的具体细节

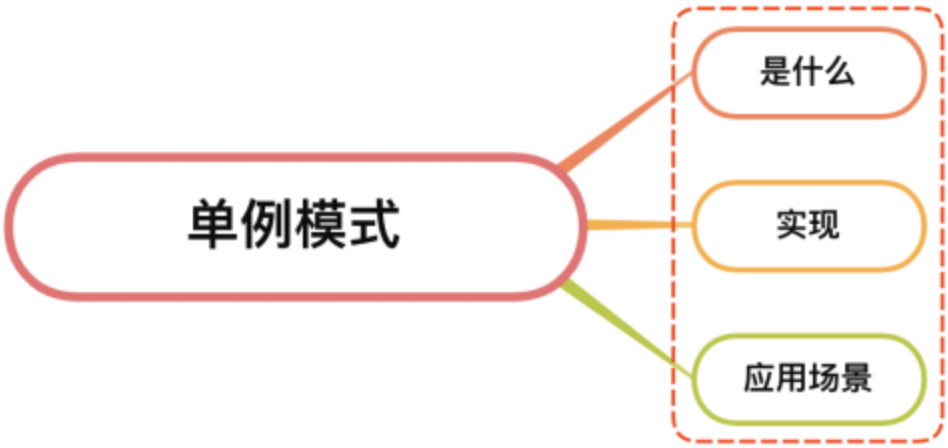
应用场景也容易识别，有构造函数的地方，就应该考虑简单工厂，但是如果函数构建函数太多与复杂，会导致工厂函数变得复杂，所以不适合复杂的情况

抽象工厂模式一般用于严格要求以面向对象思想进行开发的超大型项目中，我们一般常规的开发的话一般就是简单工厂和工厂方法模式会用的比较多一些

综上，工厂模式适用场景如下：

- 如果你不想让某个子系统与较大的那个对象之间形成强耦合，而是想运行时从许多子系统中进行挑选的话，那么工厂模式是一个理想的选择
- 将new操作简单封装，遇到new的时候就应该考虑是否用工厂模式；
- 需要依赖具体环境创建不同实例，这些实例都有相同的行为,这时候我们可以使用工厂模式，简化实现的过程，同时也可以减少每种对象所需的代码量，有利于消除对象间的耦合，提供更大的灵活性

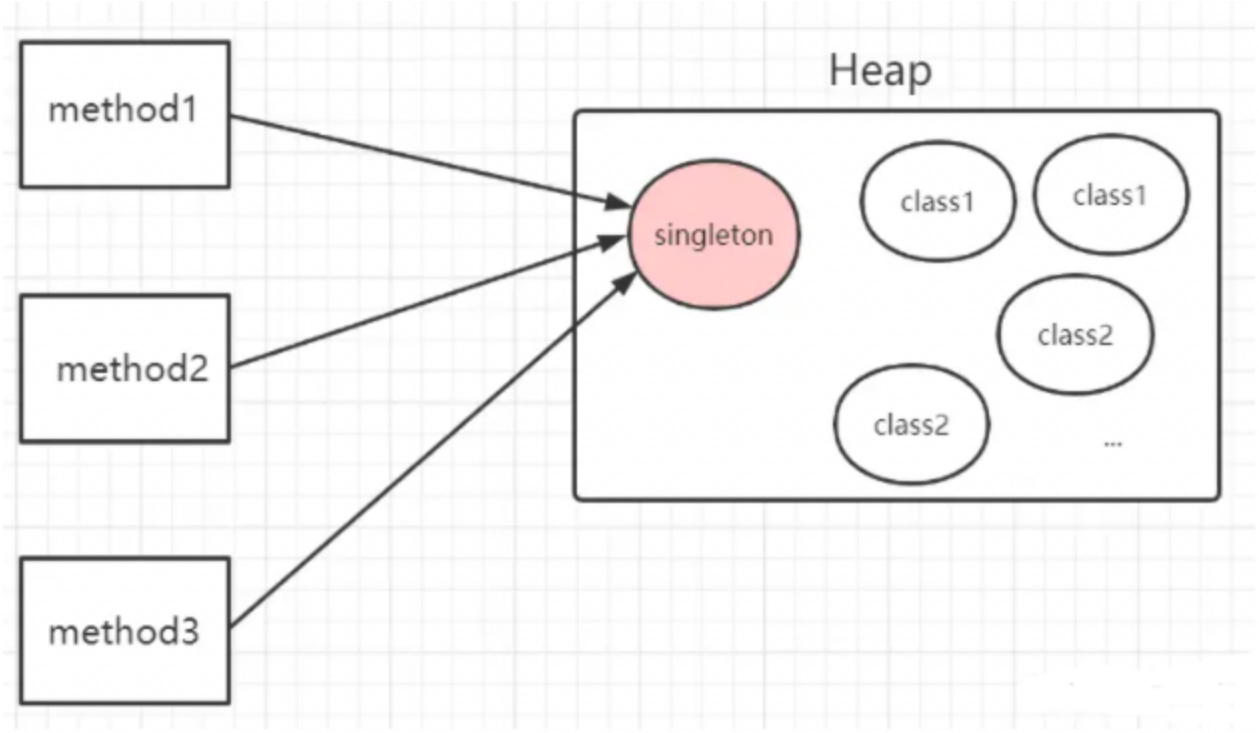
# 3. 说说你对单例模式的理解？如何实现？




## 3.1. 是什么

单例模式（Singleton Pattern）：创建型模式，提供了一种创建对象的最佳方式，这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建

在应用程序运行期间，单例模式只会在全局作用域下创建一次实例对象，让所有需要调用的地方都共享这一单例对象，如下图所示：




从定义上来看，全局变量好像就是单例模式，但是一般情况我们不认为全局变量是一个单例模式，原因是：

- 全局命名污染
- 不易维护，容易被重写覆盖

## 3.2. 实现

在 `javascript` 中，实现一个单例模式可以用一个变量来标志当前的类已经创建过对象，如果下次获取当前类的实例时，直接返回之前创建的对象即可，如下：

JavaScript | 复制代码

```javascript
// 定义一个类
function Singleton(name) {
    this.name = name;
    this.instance = null;
}
// 原型扩展类的一个方法getName()
Singleton.prototype.getName = function() {
    console.log(this.name)
};
// 获取类的实例
Singleton.getInstance = function(name) {
    if(!this.instance) {
        this.instance = new Singleton(name);
    }
    return this.instance
};

// 获取对象1
const a = Singleton.getInstance('a');
// 获取对象2
const b = Singleton.getInstance('b');
// 进行比较
console.log(a === b);
```

使用闭包也能够实现，如下：

JavaScript | 复制代码

```javascript
function Singleton(name) {
    this.name = name;
}
// 原型扩展类的一个方法getName()
Singleton.prototype.getName = function() {
    console.log(this.name)
};
// 获取类的实例
Singleton.getInstance = (function() {
    var instance = null;
    return function(name) {
        if(!this.instance) {
            this.instance = new Singleton(name);
        }
        return this.instance
    }
})();

// 获取对象1
const a = Singleton.getInstance('a');
// 获取对象2
const b = Singleton.getInstance('b');
// 进行比较
console.log(a === b);
```

也可以将上述的方法稍作修改，变成构造函数的形式，如下：

JavaScript | 复制代码

```javascript
// 单例构造函数
function CreateSingleton (name) {
    this.name = name;
    this.getName();
};

// 获取实例的名字
CreateSingleton.prototype.getName = function() {
    console.log(this.name)
};
// 单例对象
const Singleton = (function(){
    var instance;
    return function (name) {
        if(!instance) {
            instance = new CreateSingleton(name);
        }
        return instance;
    }
})();

// 创建实例对象1
const a = new Singleton('a');
// 创建实例对象2
const b = new Singleton('b');

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

## 3.3. 使用场景

在前端中，很多情况都是用到单例模式，例如页面存在一个模态框的时候，只有用户点击的时候才会创建，而不是加载完成之后再创建弹窗和隐藏，并且保证弹窗全局只有一个

可以先创建一个通常的获取对象的方法，如下：

```javascript
const getSingle = function( fn ){
  let result;
  return function(){
    return result || ( result = fn .apply(this, arguments ) );
  }
};
```

创建弹窗的代码如下：

```javascript
const createLoginLayer = function(){
  var div = document.createElement( 'div' );
  div.innerHTML = '我是浮窗';
  div.style.display = 'none';
  document.body.appendChild( div );
  return div;
};

const createSingleLoginLayer = getSingle( createLoginLayer );

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

上述这种实现称为惰性单例，意图解决需要时才创建类实例对象

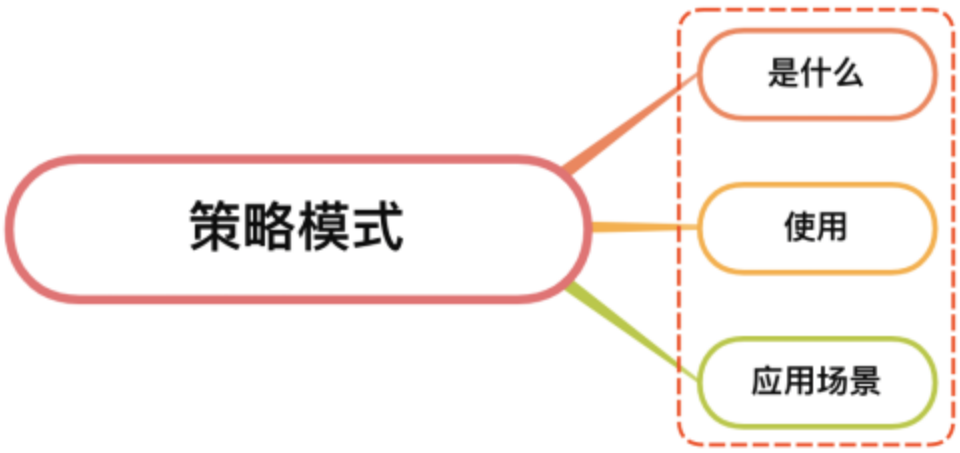
并且 `Vuex` 、 `redux` 全局态管理库也应用单例模式的思想，如下图：

```
1   import applyMixin from './mixin'
2   import devtoolPlugin from './plugins/devtool'
3   import ModuleCollection from './module/module-colle
4   import { forEachValue, isObject, isPromise, assert,
5
6   let Vue // bind on install

539 export function install (_Vue) {
540   if (Vue && _Vue === Vue) {
541     if (__DEV__) {
542       console.error(
543         '[vuex] already installed. Vue.use(Vuex)
544       )
545     }
546     return
547   }
548   Vue = _Vue
549   applyMixin(Vue)
550 }
```

现在很多第三方库都是单例模式，多次引用只会使用同一个对象，
如 `jquery` 、 `lodash` 、 `moment` ...

# 4. 说说你对策略模式的理解？应用场景？




## 4.1. 是什么

策略模式（Strategy Pattern）指的是定义一系列的算法，把它们一个个封装起来，目的就是将算法的使用与算法的实现分离开来

一个基于策略模式的程序至少由两部分组成：

- 策略类，策略类封装了具体的算法，并负责具体的计算过程
- 环境类Context，Context 接受客户的请求，随后 把请求委托给某一个策略类

## 4.2. 使用

举个例子，公司的年终奖是根据员工的工资和绩效来考核的，绩效为A的人，年终奖为工资的4倍，绩效为B的人，年终奖为工资的3倍，绩效为C的人，年终奖为工资的2倍

若使用 `if` 来实现，代码则如下：

JavaScript | 复制代码

```javascript
var calculateBouns = function(salary,level) {
    if(level === 'A') {
        return salary * 4;
    }
    if(level === 'B') {
        return salary * 3;
    }
    if(level === 'C') {
        return salary * 2;
    }
};
// 调用如下：
console.log(calculateBouns(4000,'A')); // 16000
console.log(calculateBouns(2500,'B')); // 7500
```

从上述可有看到，函数内部包含过多 `if...else` ，并且后续改正的时候，需要在函数内部添加逻辑，违反了开放封闭原则

而如果使用策略模式，就是先定义一系列算法，把它们一个个封装起来，将不变的部分和变化的部分隔开，如下：

```javascript
var obj = {
        "A": function(salary) {
            return salary * 4;
        },
        "B" : function(salary) {
            return salary * 3;
        },
        "C" : function(salary) {
            return salary * 2;
        }
};
var calculateBouns =function(level,salary) {
    return obj[level](salary);
};
console.log(calculateBouns('A',10000)); // 40000
```

上述代码中，`obj` 对应的是策略类，而 `calculateBouns` 对应上下通信类

又比如实现一个表单校验的代码，常常会像如下写法：

```javascript
var registerForm = document.getElementById("registerForm");
registerForm.onsubmit = function(){
    if(registerForm.userName.value === '') {
        alert('用户名不能为空');
        return;
    }
    if(registerForm.password.value.length < 6) {
        alert("密码的长度不能小于6位");
        return;
    }
    if(!/(^1[3|5|8][0-9]{9}$)/.test(registerForm.phoneNumber.value)) {
        alert("手机号码格式不正确");
        return;
    }
}
```

上述代码包含多处 `if` 语句，并且违反了开放封闭原则，如果应用中还有其他的表单，需要重复编写代码

此处也可以使用策略模式进行重构校验，第一步确定不变的内容，即策略规则对象，如下：

JavaScript | 复制代码

```javascript
var strategy = {
    isNotEmpty: function(value,errorMsg) {
        if(value === '') {
            return errorMsg;
        }
    },
    // 限制最小长度
    minLength: function(value,length,errorMsg) {
        if(value.length < length) {
            return errorMsg;
        }
    },
    // 手机号码格式
    mobileFormat: function(value,errorMsg) {
        if(!/(^1[3|5|8][0-9]{9}$)/.test(value)) {
            return errorMsg;
        }
    }
};
```

然后找出变的地方，作为环境类 `context` ，负责接收用户的要求并委托给策略规则对象，如下 `Validator` 类：

```javascript
var Validator = function(){
        this.cache = [];  // 保存效验规则
};
Validator.prototype.add = function(dom,rule,errorMsg) {
    var str = rule.split(":");
    this.cache.push(function(){
        // str 返回的是 minLength:6
        var strategy = str.shift();
        str.unshift(dom.value); // 把input的value添加进参数列表
        str.push(errorMsg);  // 把errorMsg添加进参数列表
        return strategys[strategy].apply(dom,str);
    });
};
Validator.prototype.start = function(){
    for(var i = 0, validatorFunc; validatorFunc = this.cache[i++]; ) {
        var msg = validatorFunc(); // 开始效验 并取得效验后的返回信息
        if(msg) {
            return msg;
        }
    }
};
```

通过 `validator.add` 方法添加校验规则和错误信息提示，使用如下：

```javascript
var validateFunc = function(){
    var validator = new Validator(); // 创建一个Validator对象
    /* 添加一些效验规则 */
    validator.add(registerForm.userName,'isNotEmpty','用户名不能为空');
    validator.add(registerForm.password,'minLength:6','密码长度不能小于6位');
    validator.add(registerForm.userName,'mobileFormat','手机号码格式不正确');

    var errorMsg = validator.start(); // 获得效验结果
    return errorMsg; // 返回效验结果
};
var registerForm = document.getElementById("registerForm");
registerForm.onsubmit = function(){
    var errorMsg = validateFunc();
    if(errorMsg){
        alert(errorMsg);
        return false;
    }
}
```

上述通过策略模式完成表单的验证，并且可以随时调用，在修改表单验证规则的时候，也非常方便，通过传递参数即可调用

## 4.3. 应用场景

从上面可以看到，使用策略模式的优点有如下：

- 策略模式利用组合，委托等技术和思想，有效的避免很多if条件语句
- 策略模式提供了开放–封闭原则，使代码更容易理解和扩展
- 策略模式中的代码可以复用

策略模式不仅仅用来封装算法，在实际开发中，通常会把算法的含义扩散开来，使策略模式也可以用来封装 一系列的“业务规则”

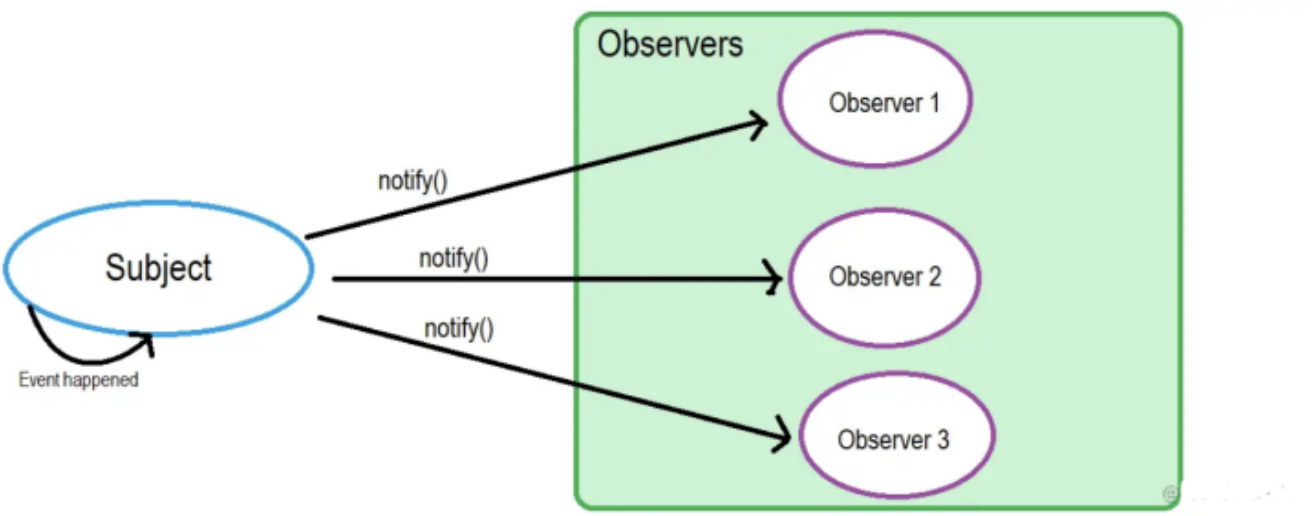
只要这些业务规则指向的目标一致，并且可以被替换使用，我们就可以用策略模式来封装它们

# 5. 说说你对发布订阅、观察者模式的理解？区别？


## 5.1. 观察者模式

观察者模式定义了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，并自动更新

观察者模式属于行为型模式，行为型模式关注的是对象之间的通讯，观察者模式就是观察者和被观察者之间的通讯




例如生活中，我们可以用报纸期刊的订阅来形象的说明，当你订阅了一份报纸，每天都会有一份最新的报纸送到你手上，有多少人订阅报纸，报社就会发多少份报纸

报社和订报纸的客户就形成了一对多的依赖关系

实现代码如下：

被观察者模式

JavaScript | 复制代码

```
class Subject {

  constructor() {
    this.observerList = [];
  }

  addObserver(observer) {
    this.observerList.push(observer);
  }

  removeObserver(observer) {
    const index = this.observerList.findIndex(o => o.name === observer.name);
    this.observerList.splice(index, 1);
  }

  notifyObservers(message) {
    const observers = this.observeList;
    observers.forEach(observer => observer.notified(message));
  }

}
```

观察者：

```
C | 复制代码
class Observer {

  constructor(name, subject) {
    this.name = name;
    if (subject) {
      subject.addObserver(this);
    }
  }

  notified(message) {
    console.log(this.name, 'got message', message);
  }
}
```

使用代码如下：

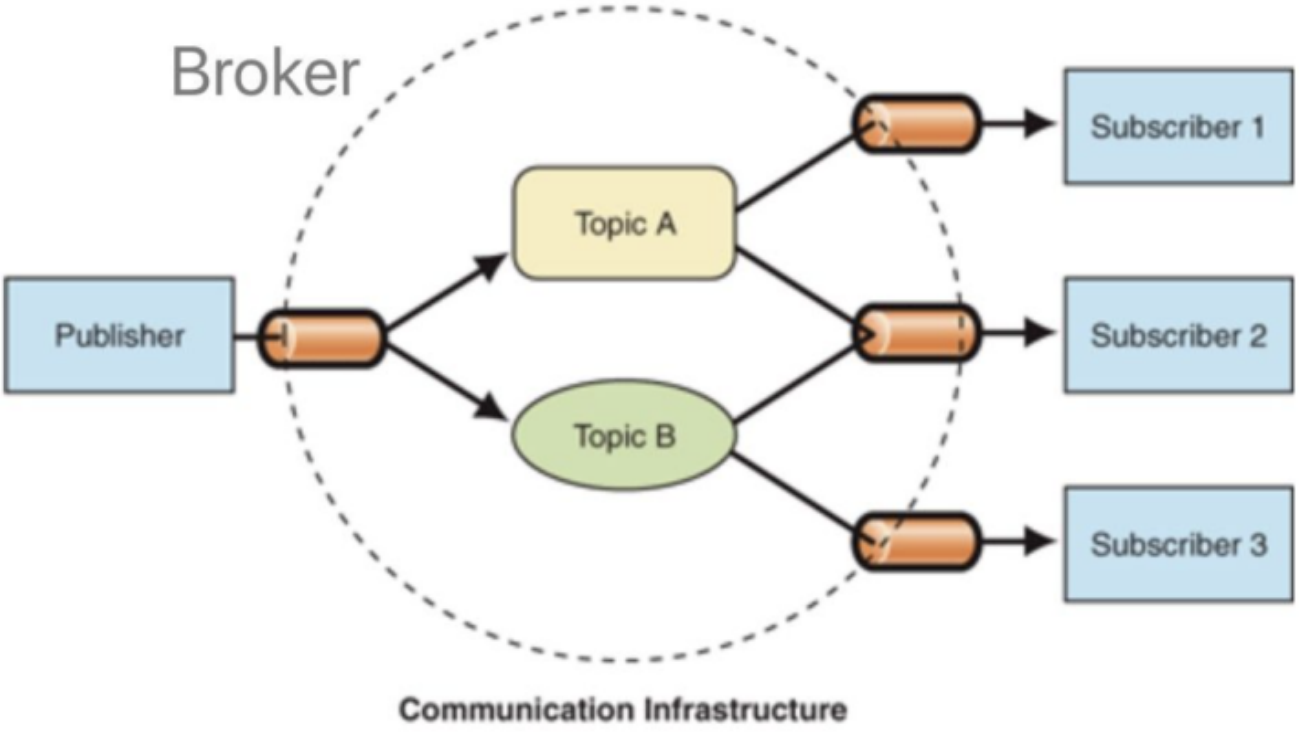
```
JavaScript | 复制代码
const subject = new Subject();
const observerA = new Observer('observerA', subject);
const observerB = new Observer('observerB');
subject.addObserver(observerB);
subject.notifyObservers('Hello from subject');
subject.removeObserver(observerA);
subject.notifyObservers('Hello again');
```

上述代码中，观察者主动申请加入被观察者的列表，被观察者主动将观察者加入列表

## 5.2. 发布订阅模式

发布–订阅是一种消息范式，消息的发送者（称为发布者）不会将消息直接发送给特定的接收者（称为订阅者）。而是将发布的消息分为不同的类别，无需了解哪些订阅者（如果有的话）可能存在

同样的，订阅者可以表达对一个或多个类别的兴趣，只接收感兴趣的消息，无需了解哪些发布者存在


Broker
Publisher
Topic A
Topic B
Subscriber 1
Subscriber 2
Subscriber 3
Communication Infrastructure

实现代码如下：

JavaScript | 复制代码

```javascript
class PubSub {
  constructor() {
    this.messages = {};
    this.listeners = {};
  }
  // 添加发布者
  publish(type, content) {
    const existContent = this.messages[type];
    if (!existContent) {
      this.messages[type] = [];
    }
    this.messages[type].push(content);
  }
  // 添加订阅者
  subscribe(type, cb) {
    const existListener = this.listeners[type];
    if (!existListener) {
      this.listeners[type] = [];
    }
    this.listeners[type].push(cb);
  }
  // 通知
  notify(type) {
    const messages = this.messages[type];
    const subscribers = this.listeners[type] || [];
    subscribers.forEach((cb, index) => cb(messages[index]));
  }
}
```

发布者代码如下：

JavaScript | 复制代码

```javascript
class Publisher {
  constructor(name, context) {
    this.name = name;
    this.context = context;
  }
  publish(type, content) {
    this.context.publish(type, content);
  }
}
```

订阅者代码如下：

JavaScript 复制代码

```
class Subscriber {
  constructor(name, context) {
    this.name = name;
    this.context = context;
  }
  subscribe(type, cb) {
    this.context.subscribe(type, cb);
  }
}
```

使用代码如下：

JavaScript 复制代码

```
const TYPE_A = 'music';
const TYPE_B = 'movie';
const TYPE_C = 'novel';

const pubsub = new PubSub();

const publisherA = new Publisher('publisherA', pubsub);
publisherA.publish(TYPE_A, 'we are young');
publisherA.publish(TYPE_B, 'the silicon valley');
const publisherB = new Publisher('publisherB', pubsub);
publisherB.publish(TYPE_A, 'stronger');
const publisherC = new Publisher('publisherC', pubsub);
publisherC.publish(TYPE_C, 'a brief history of time');

const subscriberA = new Subscriber('subscriberA', pubsub);
subscriberA.subscribe(TYPE_A, res => {
  console.log('subscriberA received', res)
});
const subscriberB = new Subscriber('subscriberB', pubsub);
subscriberB.subscribe(TYPE_C, res => {
  console.log('subscriberB received', res)
});
const subscriberC = new Subscriber('subscriberC', pubsub);
subscriberC.subscribe(TYPE_B, res => {
  console.log('subscriberC received', res)
});

pubsub.notify(TYPE_A);
pubsub.notify(TYPE_B);
pubsub.notify(TYPE_C);
```

上述代码，发布者和订阅者需要通过发布订阅中心进行关联，发布者的发布动作和订阅者的订阅动作相互独立，无需关注对方，消息派发由发布订阅中心负责

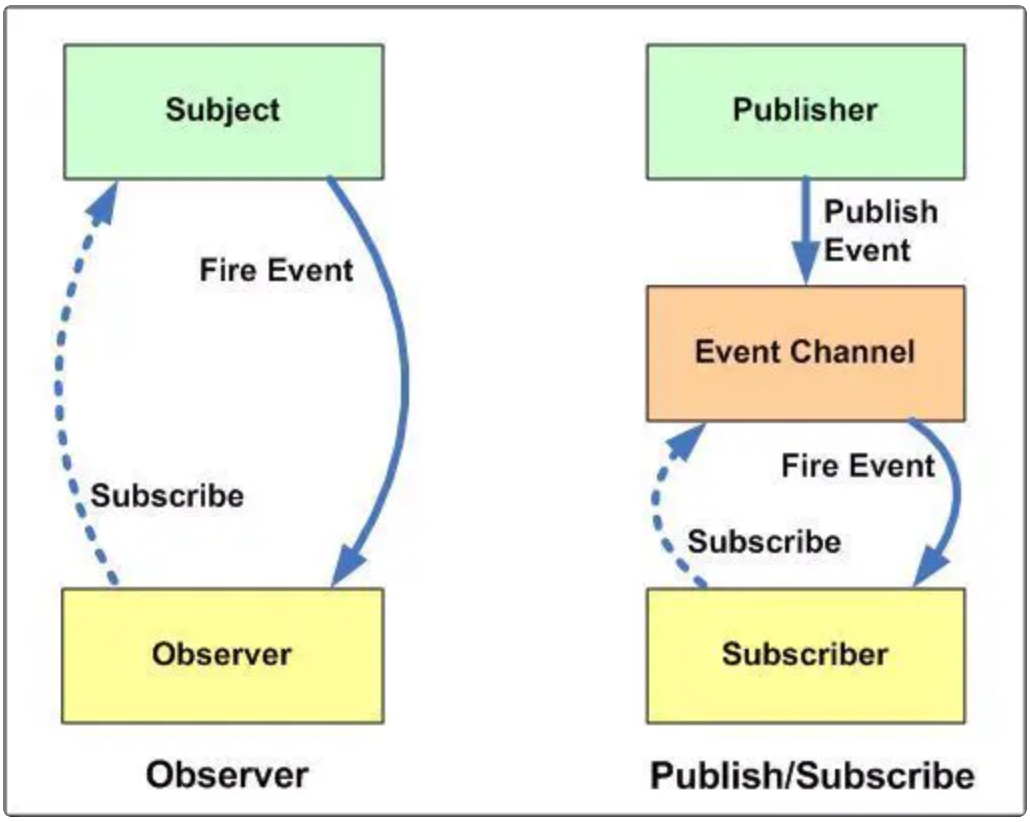
## 5.3. 区别

两种设计模式思路是一样的，举个生活例子：

- 观察者模式：某公司给自己员工发月饼发粽子，是由公司的行政部门发送的，这件事不适合交给第三方，原因是“公司”和“员工”是一个整体
- 发布-订阅模式：某公司要给其他人发各种快递，因为“公司”和“其他人”是独立的，其唯一的桥梁是“快递”，所以这件事适合交给第三方快递公司解决

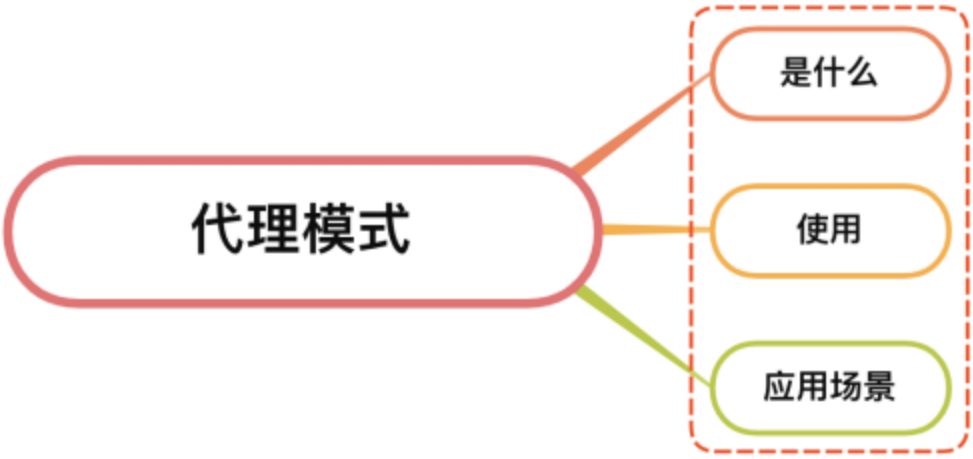
上述过程中，如果公司自己去管理快递的配送，那公司就会变成一个快递公司，业务繁杂难以管理，影响公司自身的主营业务，因此使用何种模式需要考虑什么情况两者是需要耦合的

两者区别如下图：




- 在观察者模式中，观察者是知道Subject的，Subject一直保持对观察者进行记录。然而，在发布订阅模式中，发布者和订阅者不知道对方的存在。它们只有通过消息代理进行通信。
- 在发布订阅模式中，组件是松散耦合的，正好和观察者模式相反。
- 观察者模式大多数时候是同步的，比如当事件触发，Subject就会去调用观察者的方法。而发布-订阅模式大多数时候是异步的（使用消息队列）

# 6. 说说你对代理模式的理解？应用场景？




## 6.1. 是什么

代理模式（Proxy Pattern）是为一个对象提供一个代用品或占位符，以便控制对它的访问

代理模式的关键是，当客户不方便直接访问一个对象或者不满足需要时，提供一个替身对象来控制这个对象的访问，客户实际上访问的是替身对象


在生活中，代理模式的场景是十分常见的，例如我们现在如果有租房、买房的需求，更多的是去找链家等房屋中介机构，而不是直接寻找想卖房或出租房的人谈。此时，链家起到的作用就是代理的作用

## 6.2. 使用

在 `ES6` 中，存在 `proxy` 构建函数能够让我们轻松使用代理模式：

JavaScript | 复制代码

```
const proxy = new Proxy(target, handler);
```

关于 `Proxy` 的使用可以翻看以前的文章

而按照功能来划分， `javascript` 代理模式常用的有：

- 缓存代理
- 虚拟代理

### 6.2.1. 缓存代理

缓存代理可以为一些开销大的运算结果提供暂时的存储，在下次运算时，如果传递进来的参数跟之前一致，则可以直接返回前面存储的运算结果

如实现一个求积乘的函数，如下：

JavaScript | 复制代码

```javascript
var muti = function () {
  console.log("开始计算乘积");
  var a = 1;
  for (var i = 0, l = arguments.length; i < l; i++) {
    a = a * arguments[i];
  }
  return a;
};
```

现在加入缓存代理，如下：

JavaScript | 复制代码

```javascript
var proxyMult = (function () {
  var cache = {};
  return function () {
    var args = Array.prototype.join.call(arguments, ",");
    if (args in cache) {
      return cache[args];
    }
    return (cache[args] = mult.apply(this, arguments));
  };
})();

proxyMult(1, 2, 3, 4); // 输出:24
proxyMult(1, 2, 3, 4); // 输出:24
```

当第二次调用 `proxyMult(1, 2, 3, 4)` 时，本体 `mult` 函数并没有被计算，`proxyMult` 直接返回了之前缓存好的计算结果

### 6.2.2. 虚拟代理

虚拟代理把一些开销很大的对象，延迟到真正需要它的时候才去创建

常见的就是图片预加载功能：

未使用代理模式如下：

JavaScript | 复制代码

```
let MyImage = (function(){
    let imgNode = document.createElement( 'img' );
    document.body.appendChild( imgNode );
    // 创建一个Image对象，用于加载需要设置的图片
    let img = new Image;

    img.onload = function(){
        // 监听到图片加载完成后，设置src为加载完成后的图片
        imgNode.src = img.src;
    };

    return {
        setSrc: function( src ){
            // 设置图片的时候，设置为默认的loading图
            imgNode.src = 'https://img.zcool.cn/community/01deed576019060000018c1bd2352d.gif';
            // 把真正需要设置的图片传给Image对象的src属性
            img.src = src;
        }
    }
})();

MyImage.setSrc( 'https://xxx.jpg' );
```

`MyImage` 对象除了负责给 `img` 节点设置 `src` 外，还要负责预加载图片，违反了面向对象设计的原则——单一职责原则

上述过程 `loding` 则是耦合进 `MyImage` 对象里的，如果以后某个时候，我们不需要预加载显示loading这个功能了，就只能在 `MyImage` 对象里面改动代码

使用代理模式，代码则如下：

JavaScript | 复制代码

```javascript
// 图片本地对象，负责往页面中创建一个img标签，并且提供一个对外的setSrc接口
let myImage = (function(){
    let imgNode = document.createElement( 'img' );
    document.body.appendChild( imgNode );

    return {
        //setSrc接口，外界调用这个接口，便可以给该img标签设置src属性
        setSrc: function( src ){
            imgNode.src = src;
        }
    }
})();
// 代理对象，负责图片预加载功能
let proxyImage = (function(){
    // 创建一个Image对象，用于加载需要设置的图片
    let img = new Image;
    img.onload = function(){
        // 监听到图片加载完成后，给被代理的图片本地对象设置src为加载完成后的图片
        myImage.setSrc( this.src );
    }
    return {
        setSrc: function( src ){
            // 设置图片时，在图片未被真正加载好时，以这张图作为loading，提示用户图片正在加载
            myImage.setSrc( 'https://img.zcool.cn/community/01deed576019060000018c1bd2352d.gif' );
            img.src = src;
        }
    }
})();

proxyImage.setSrc( 'https://xxx.jpg' );
```

使用代理模式后，图片本地对象负责往页面中创建一个 `img` 标签，并且提供一个对外的 `setSrc` 接口；

代理对象负责在图片未加载完成之前，引入预加载的 `loading` 图，负责了图片预加载的功能

上述并没有改变或者增加 `MyImage` 的接口，但是通过代理对象，实际上给系统添加了新的行为

并且上述代理模式可以发现，代理和本体接口的一致性，如果有一天不需要预加载，那么就不需要代理对象，可以选择直接请求本体。其中关键是代理对象和本体都对外提供了 `setSrc` 方法

## 6.3. 应用场景

现在的很多前端框架或者状态管理框架都使用代理模式，用与监听变量的变化

使用代理模式代理对象的访问的方式，一般又被称为拦截器，比如我们在项目中经常使用 `Axios` 的实例来进行 HTTP 的请求，使用拦截器 `interceptor` 可以提前对 请求前的数据 服务器返回的数据进行一些预处理

以及上述应用到的缓存代理和虚拟代理