JavaScript | 复制代码

```javascript
function mergeSort(arr) {  // 采用自上而下的递归方法
    const len = arr.length;
    if(len < 2) {
        return arr;
    }
    let middle = Math.floor(len / 2),
        left = arr.slice(0, middle),
        right = arr.slice(middle);
    return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right)
{
    const result = [];

    while (left.length && right.length) {
        if (left[0] <= right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }

    while (left.length)
        result.push(left.shift());

    while (right.length)
        result.push(right.shift());

    return result;
}
```

上述归并分成了分、合两部分，在处理分过程中递归调用两个分的操作，所花费的时间为2乘 `T(n/2)`，合的操作时间复杂度则为 `O(n)`，因此可以得到以下公式：

总的执行时间 = 2 × 输入长度为 `n/2` 的 `sort` 函数的执行时间 + `merge` 函数的执行时间 `O(n)`

当只有一个元素时，`T(1) = O(1)`

如果对 `T(n) = 2 * T(n/2) + O(n)` 进行左右 / n的操作，得到 `T(n) / n = (n / 2) * T(n/2) + O(1)`

现在令 `S(n) = T(n)/n`，则 `S(1) = O(1)`，然后利用表达式带入得到 `S(n) = S(n/2) + O(1)`

所以可以得到：`S(n) = S(n/2) + O(1) = S(n/4) + O(2) = S(n/8) + O(3) = S(n/2^k) + O(k) = S(1) + O(logn) = O(logn)`

综上可得，`T(n) = n * log(n) = nlogn`

关于归并排序的稳定性，在进行合并过程，在1个或2个元素时，1个元素不会交换，2个元素如果大小相等也不会交换，由此可见归并排序是稳定的排序算法

17.3. 应用场景

在外排序中通常使用排序–归并的策略，外排序是指处理超过内存限度的数据的排序算法，通常将中间结果放在读写较慢的外存储器，如下分成两个阶段：

- 排序阶段：读入能够放进内存中的数据量，将其排序输出到临时文件，一次进行，将带排序数据组织为多个有序的临时文件
- 归并阶段：将这些临时文件组合为大的有序文件

例如，使用100m内存对900m的数据进行排序，过程如下：

- 读入100m数据内存，用常规方式排序
- 将排序后的数据写入磁盘
- 重复前两个步骤，得到9个100m的临时文件
- 将100m的内存划分为10份，将9份为输入缓冲区，第10份为输出缓冲区
- 进行九路归并排序，将结果输出到缓冲区
    - 若输出缓冲区满，将数据写到目标文件，清空缓冲区
    - 若缓冲区空，读入相应文件的下一份数据

18. 说说你对贪心算法、回溯算法的理解？应用场景？

18.1. 贪心算法

贪心算法，又称贪婪算法，是算法设计中的一种思想

其期待每一个阶段都是局部最优的选择，从而达到全局最优，但是结果并不一定是最优的

举个零钱兑换的例子，如果你有1元、2元、5元的钱币数张，用于兑换一定的金额，但是要求兑换的钱币张数最少

如果现在你要兑换11元，按照贪心算法的思想，先选择面额最大的5元钱币进行兑换，那么就得到11 = 5 + 5 + 1 的选择，这种情况是最优的

但是如果你手上钱币的面额为1、3、4，想要兑换6元，按照贪心算法的思路，我们会 6 = 4 + 1 + 1这样选择，这种情况结果就不是最优的选择

从上面例子可以看到，贪心算法存在一些弊端，使用到贪心算法的场景，都会存在一个特性：

一旦一个问题可以通过贪心法来解决，那么贪心法一般是解决这个问题的最好办法

至于是否选择贪心算法，主要看是否有如下两大特性：

- 贪心选择：当某一个问题的整体最优解可通过一系列局部的最优解的选择达到，并且每次做出的选择可以依赖以前做出的选择，但不需要依赖后面需要做出的选择
- 最优子结构：如果一个问题的最优解包含其子问题的最优解，则此问题具备最优子结构的性质。问题的最优子结构性质是该问题是否可以用贪心算法求解的关键所在

18.2. 回溯算法

回溯算法，也是算法设计中的一种思想，是一种渐进式寻找并构建问题解决方式的策略

回溯算法会先从一个可能的工作开始解决问题，如果不行，就回溯并选择另一个动作，知道将问题解决

使用回溯算法的问题，有如下特性：

- 有很多路，例如一个矩阵的方向或者树的路径
- 在这些的路里面，有死路也有生路，思路即不符合题目要求的路，生路则符合
- 通常使用递归来模拟所有的路

常见的伪代码如下：

```
result = []
function backtrack(路径, 选择列表):
  if 满足结束条件:
    result.add(路径)
  return

  for 选择 of 选择列表:
    做选择
    backtrack(路径, 选择列表)
    撤销选择
```

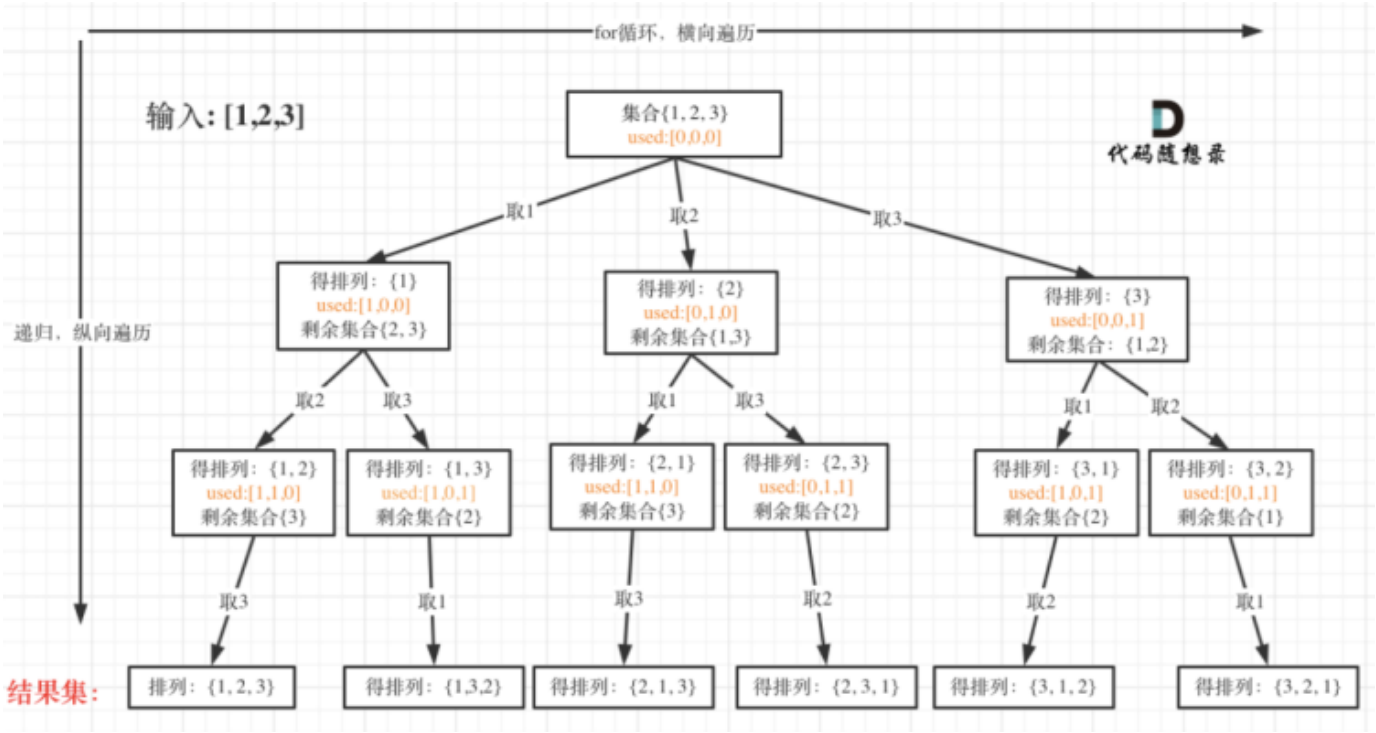
重点解决三个问题：

- 路径：也就是已经做出的选择
- 选择列表
- 结束条件

例如经典使用回溯算法为解决全排列的问题，如下：

一个不含重复数字的数组 `nums` ，我们要返回其所有可能的全排列，解决这个问题的思路是：

- 用递归模拟所有的情况
- 遇到包含重复元素的情况则回溯
- 收集到所有到达递归终点的情况，并返回、



用代码表示则如下：

JavaScript | 复制代码

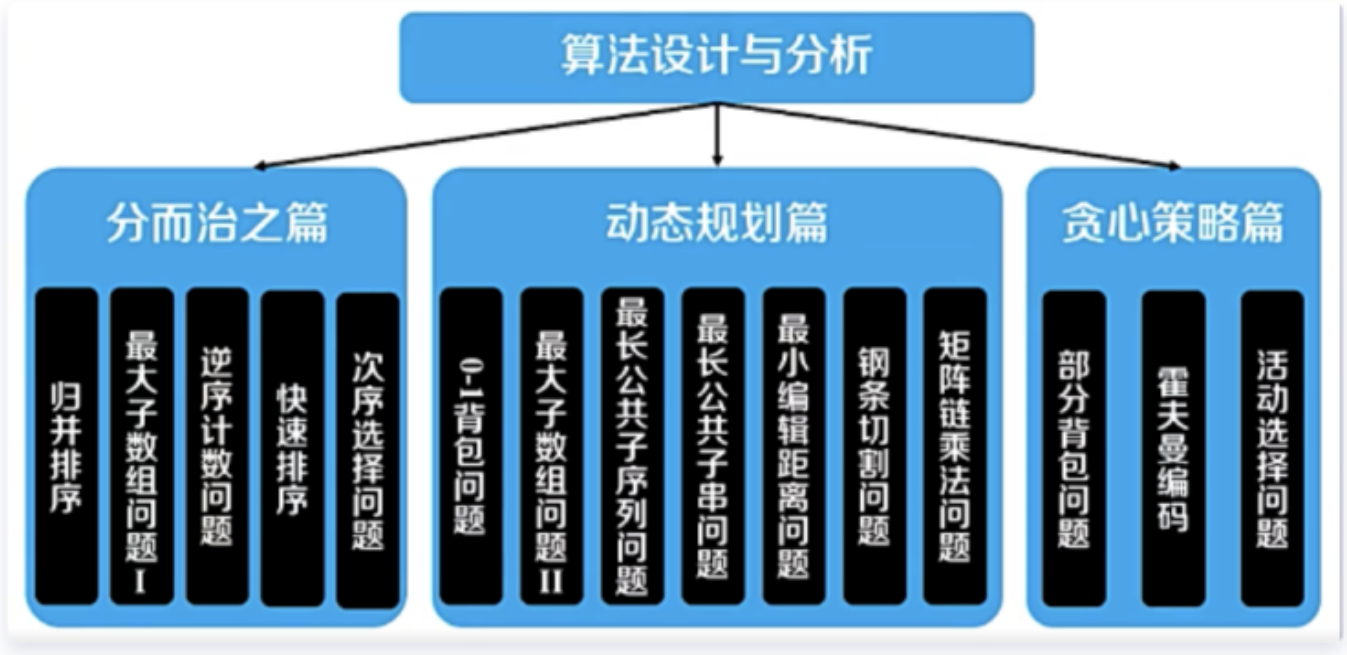
```javascript
var permute = function(nums) {
    const res = [], path = [];
    backtracking(nums, nums.length, []);
    return res;

    function backtracking(n, k, used) {
        if(path.length === k) {
            res.push(Array.from(path));
            return;
        }
        for (let i = 0; i < k; i++ ) {
            if(used[i]) continue;
            path.push(n[i]);
            used[i] = true; // 同支
            backtracking(n, k, used);
            path.pop();
            used[i] = false;
        }
    }
};
```

18.3. 总结

前面也初步了解到分而治之、动态规划，现在再了解到贪心算法、回溯算法

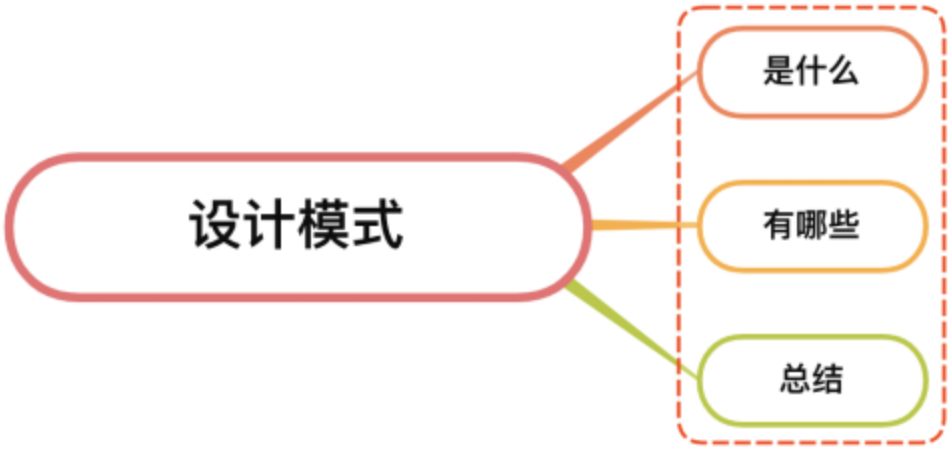
其中关于分而治之、动态规划、贪心策略三者的求解思路如下：

其中三者对应的经典问题如下图：



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：

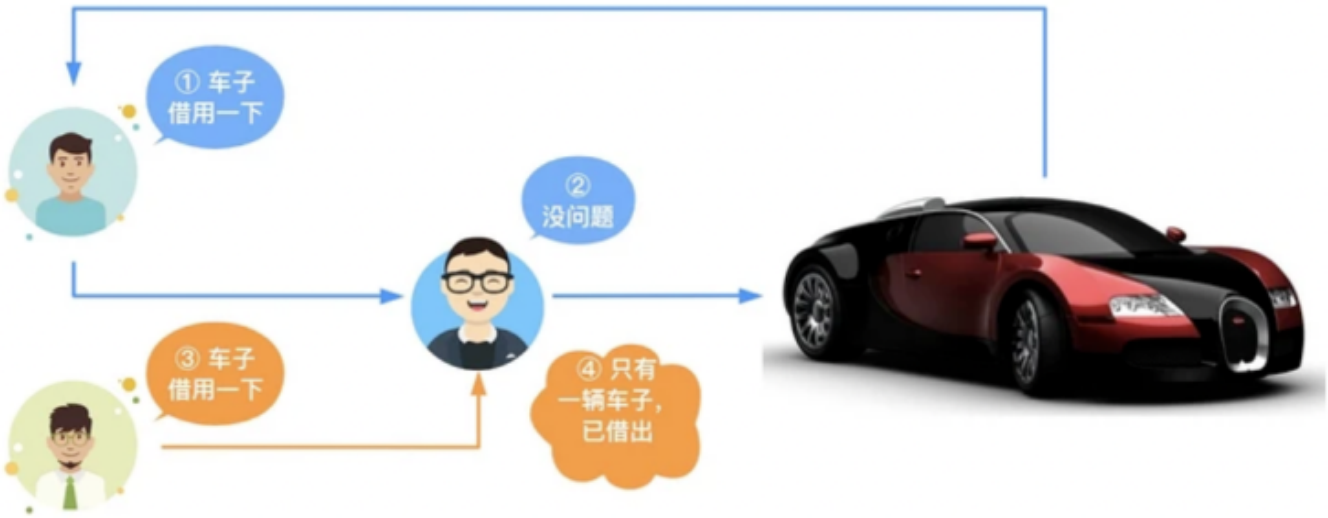
- 单例模式
- 工厂模式
- 策略模式
- 代理模式

- 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

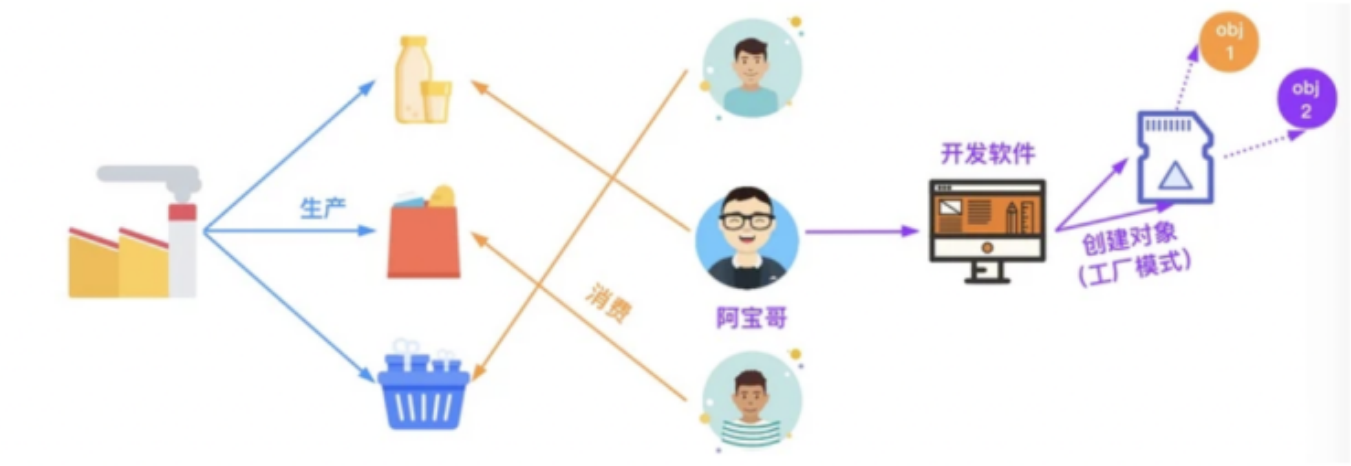
如下图的车，只有一辆，一旦借出去则不能再借给别人：



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

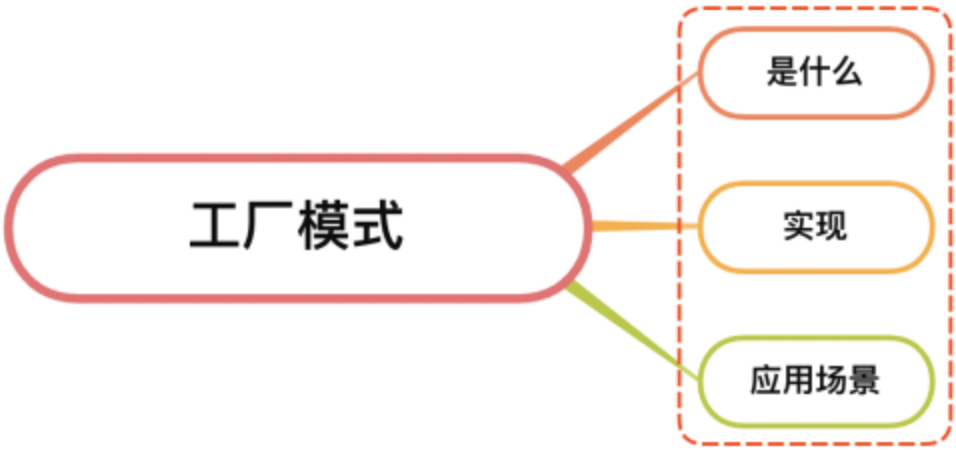
通常运用在原有方法维持不变，在原有方法上再挂载其他方法来满足现有需求

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

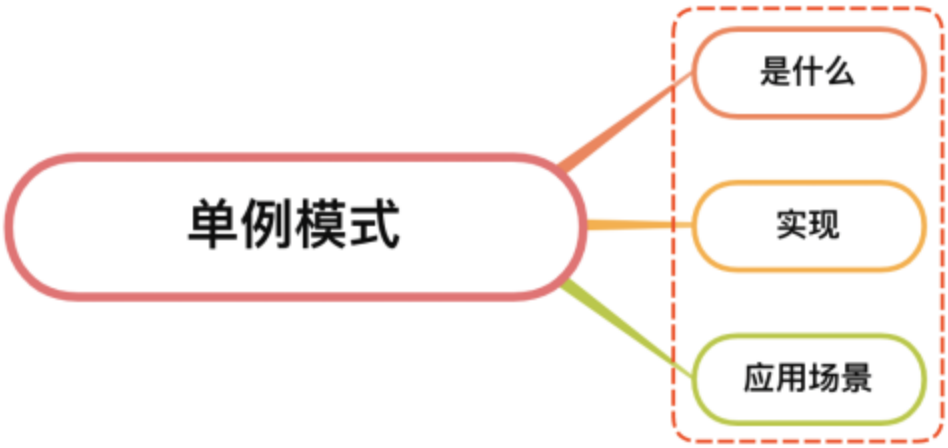
应用场景也容易识别，有构造函数的地方，就应该考虑简单工厂，但是如果函数构建函数太多与复杂，会导致工厂函数变得复杂，所以不适合复杂的情况

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

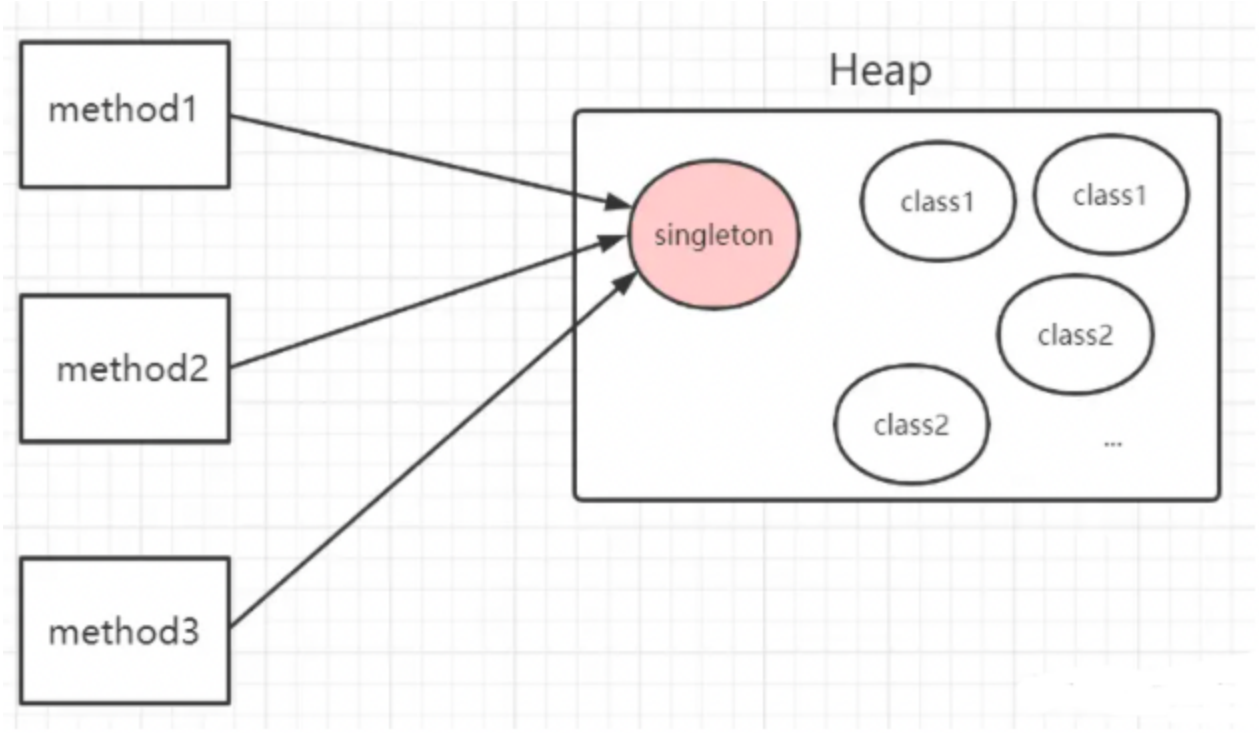
3. 说说你对单例模式的理解？如何实现？



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';
};
```

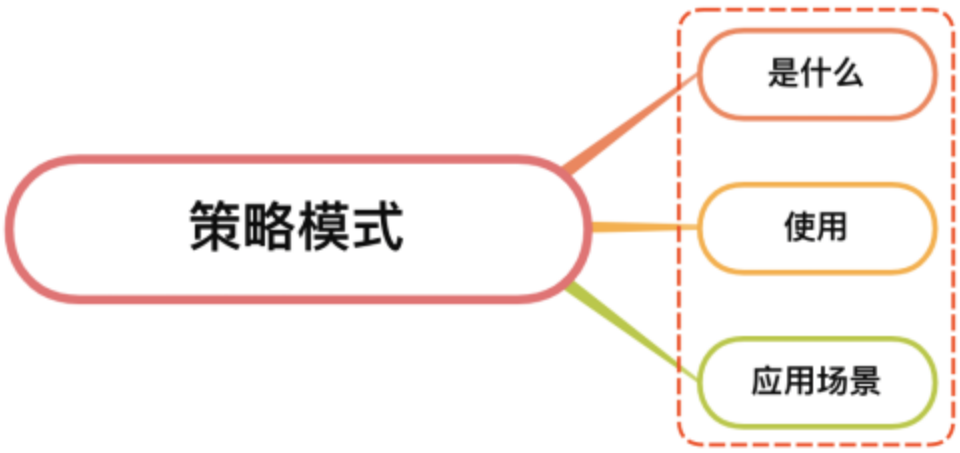
上述这种实现称为惰性单例，意图解决需要时才创建类实例对象

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”

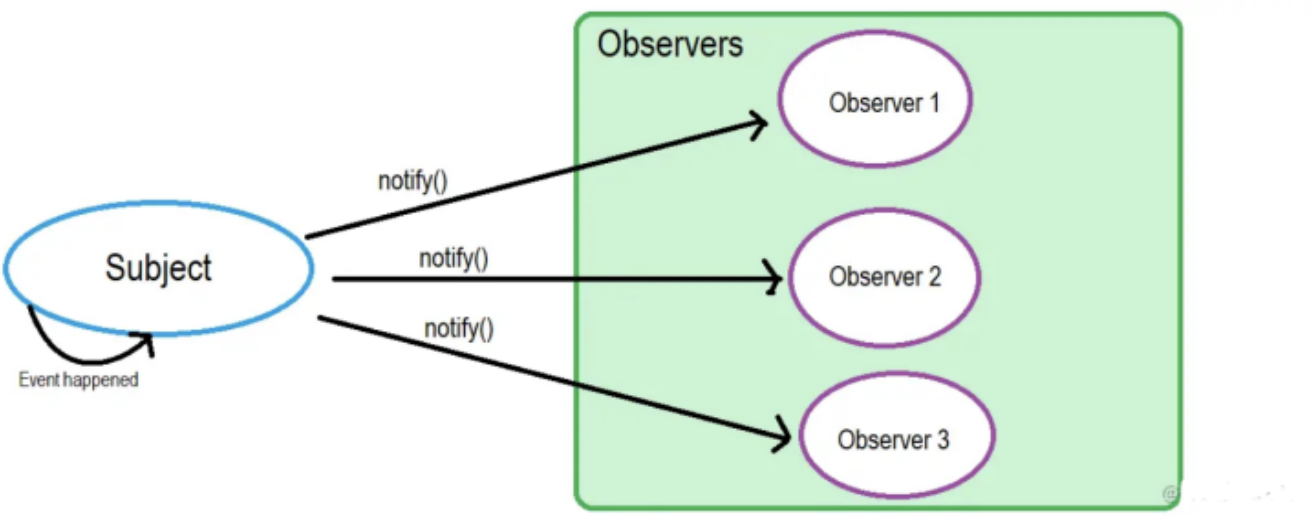
只要这些业务规则指向的目标一致，并且可以被替换使用，我们就可以用策略模式来封装它们

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
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
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