代码工程仓库：

[shAdow-XJY/js\_coding: js 编程练习记录 代码工程 (github.com)](https://github.com/shAdow-XJY/js_coding)

# 算法题：

* 两数之和

[1. 两数之和 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/two-sum/submissions/)

|  |
| --- |
| function twoNum(nums , target){  var hash = new ***Map***();  for(let i = 0 ;i < nums.length; i++){  if(hash.has(target-nums[i])){  return [hash.get(target-nums[i]),i];  }else {  hash.set(nums[i],i);  }  }  return []; }  ***console***.log(twoNum([2,7,11,15],9)) |

* 有效括号

[20. 有效的括号 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/valid-parentheses/)

|  |
| --- |
| function valid(s){  if(s.length%2 !== 0){  return false;  }   const map = new ***Map***([  [')','('],  [']','['],  ['}','{']  ]);  let stack = [];  for(let item of s){  if(!map.has(item)){ //item = ( { [  stack.push(item);  }else { //item = ) } ]  if(map.get(item) !== stack.pop()){  return false;  }  }  }   if(stack.length === 0){  return true;  } } |

* 数组去除/去重

[(4条消息) js 数组去除的几种方法\_活着就得有野心的博客-CSDN博客\_数组去除](https://blog.csdn.net/huoniu007/article/details/49529693)

|  |
| --- |
| //indexOf() function deleteMore(inputList){  let res = [];  for (let item of inputList){  if(res.indexOf(item) === -1){  res.push(item);  }  }  return res; } |
| // indexOf() // array.filter(function(currentValue,index,arr), thisValue)返回新数组  function deleteMore2(inputList){  return inputList.filter((value,index,array)=>{  return array.indexOf(value) === index  }) } |
| // sort()方法会改变原始数组  function deleteMore3(inputList) {  inputList.sort();  let res = [inputList[0]];  for(let item of inputList){  if(item !== res[res.length-1]){  res.push(item)  }  }  return res } |
| // set去重 function deleteMore4(inputList) {  const set = new ***Set***(inputList);  return [...set]; } |
| // map 方法 function deleteMore5(inputList) {  let res = [];  const hash = new ***Map***();   for (let item of inputList){  if (!hash.has(item)){  hash.set(item,true);  res.push(item);  }  }   return res; } |

* RMB 形式：12345678 转化成 12,345,678

[19道常见的JS面试算法题 - tiana\_Z - 博客园 (cnblogs.com)](https://www.cnblogs.com/djw12333/p/11647413.html)

|  |
| --- |
| // 12345678 转化成 12,345,678 let ***str*** = '12345678'  function RMB(str) {  let res = '';   let strList = str.split(''); // ['1', '2', '3', '4', '5', '6', '7', '8']   for (let i in strList) {   res += strList[i];   if ((i+1)%3 === 0){  res += ',';  }  }   return res; }  ***console***.log(RMB(***str***)) |

* 实现版本号比较

[js中版本号的比较 - 掘金 (juejin.cn)](https://juejin.cn/post/6844903942812336142)

|  |
| --- |
| // 关键是将点分割，每一部分化成number比较，分割后没有的用0补充 function compareVersion(version1, version2) {  let v1 = version1.split('.'), v2 = version2.split('.');  let len1 = v1.length, len2 = v2.length;  let index1 = 0, index2 = 0;   while (index1 < len1 || index2 < len2){  let v1Item = 0, v2Item = 0;   if (index1 < len1) v1Item = parseInt(v1[index1++]);  if (index2 < len2) v2Item = parseInt(v2[index2++]);   if (v1Item !== v2Item) return v1Item > v2Item ? 1 : -1;  }   return 0; } let ***version1*** = '1.2.4',***version2*** = '1.1.5'; let ***version11*** = '1.2',***version22*** = '1.10.5'; let ***version111*** = '1.00.03',***version222*** = '1.0.03';  ***console***.log(compareVersion(***version1***,***version2***)) ***console***.log(compareVersion(***version11***,***version22***)) ***console***.log(compareVersion(***version111***,***version222***)) |

* 求最大公约数

[19道常见的JS面试算法题 - tiana\_Z - 博客园 (cnblogs.com)](https://www.cnblogs.com/djw12333/p/11647413.html)

|  |
| --- |
| // 辗转相除法 // a / b // b / a%b // 直到除数为0，被除数即为最大公约数 function greatestCommonDivisor(a,b) {  if (b === 0){  return a;  }   return greatestCommonDivisor(b,a%b); } ***console***.log(greatestCommonDivisor(81,5)) ***console***.log(greatestCommonDivisor(112,8)) |

* 实现数组打乱

[如何将一个 JavaScript 数组打乱顺序 - 简书 (jianshu.com)](https://www.jianshu.com/p/4454eaf4bdf8)

|  |
| --- |
| // Fisher–Yates shuffle 洗牌算法 // 该方法就是每次在数组中随机产生一个位置，依次将数组中的每一项与该次产生的随机位置上的元素交换位置：  // Math.random () Math.random () 函数返回一个浮点数， 伪随机数在范围从 0 到 小于 1 ，也就是说，从 0（包括 0）往上，但是不包括 1（排除 1） var ***list*** = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15];  function fisher(list) {  let len = list.length;  for (let i = 0; i < len; ++i){  let randomIndex = ***Math***.floor(***Math***.random() \* len);   let temp = list[i];  list[i] = list[randomIndex];  list[randomIndex] = temp;  }  return list; }  ***console***.log(fisher(***list***)); |

* 斐波拉契

[js 实现斐波那契数列(数组缓存、动态规划、尾调用优化) - 简书 (jianshu.com)](https://www.jianshu.com/p/bbc7e54a98d6)

|  |
| --- |
| // 斐波那契数列是以下一系列数字： // 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ... // 在种子数字 0 和 1 之后，后续的每一个数字都是前面两个数字之和。  // 在数学上，斐波那契数列是以递归的方法来定义： // F(0)=0, F(1)=1, // n>1时，F(n)=F(n-1)+F(n-2)。 // 根据该规则，返回第n个斐波那契数。  // 递归 function Fibonacci\_di(n) {  if (n === 0 || n === 1){  return n;  }   return Fibonacci\_di(n-2) + Fibonacci\_di(n-1); } // 递推（动态规划 、 迭代） function Fibonacci\_dynamic(n) {  let pre = 0;  let next = 1;  for (let i = 0; i < n; ++i){  [pre , next] = [next, pre + next];  }  return pre; } // 在ES6规范中，有一个尾调用优化，可以实现高效的尾递归方案。 // ES6的尾调用优化只在严格模式下开启，正常模式是无效的。 `use strict` function Fibonacci\_tail(n, pre = 0, next = 1) {  if (n === 0) return 0;  if (n === 1) return next;   return Fibonacci\_tail(n-1, next, pre + next); }  ***console***.time('递归'); ***console***.log(Fibonacci\_di(30)); ***console***.timeEnd('递归');  ***console***.time('动态规划'); ***console***.log(Fibonacci\_dynamic(30)); ***console***.timeEnd('动态规划');  ***console***.time('尾递归'); ***console***.log(Fibonacci\_tail(30)); ***console***.timeEnd('尾递归');  // 输出结果： 第一次运行用时： 动态规划 < 尾递归 < 递归 // 但是多次运行之后，可以发现 尾递归 的用时 会变得比 动态规划 小了 |

* sleep 函数

[(4条消息) javascript里的sleep()方法\_clschen的博客-CSDN博客\_js sleep()](https://blog.csdn.net/clschen/article/details/51727599)

|  |
| --- |
| function sleep(time) {  return new ***Promise***((resolve) => {  setTimeout(resolve,time);  }) }  // 异步的操作 ***console***.time('异步'); sleep(1000).then(()=>{  ***console***.log('异步使用sleep');  // 不会阻塞console.timeEnd('异步') 打印 }) ***console***.timeEnd('异步');  // 同步操作 (async function() {  ***console***.time('同步');  await sleep(1000).then(()=>{  ***console***.log('同步使用sleep');  // 会阻塞console.timeEnd('同步')打印  });  ***console***.timeEnd('同步'); })();  // 异步的打印时间不会包括sleep的1s，同步的打印时间会包括sleep的1s |

* 查找数组中第二大元素/求数组的两个最大值

[(2条消息) 数组：找出数组中第二大的数\_加油呐的博客-CSDN博客\_求数组中第二大的数](https://blog.csdn.net/weixin_38108266/article/details/81269216)

|  |
| --- |
| let ***list*** = [0,5,4,9,8,11,10]  // 1. function secondMax(list) {  if (list.length < 2) return false;   let max = list[0];  let sec = max - 1;   for (let item of list){  if (item > max){  sec = max;  max = item;  }else if (item > sec){  sec = item;  }  }   return [max,sec]; }  // 2. sort // sort 方法失效：为什么我们省略回调函数这个参数，就不行了呢？ // 原因很简单，当你没有传递这个参数的时候，会触发sort的默认行为，它的默认行为就是把数组中所有的项转化为字符串，然后进行unicode编码比较。 function secondMax\_sort(list) {   list.sort(function (a,b){  return a-b  });   let index = list.length-2;  while (list[index] === list[index+1]){  index--;  }    return [list[list.length-1],list[index]]; }  ***console***.log(secondMax(***list***)); ***console***.log(secondMax\_sort(***list***)); |

* 最长不重复字符串长度

[3. 无重复字符的最长子串 - 力扣（LeetCode）](https://leetcode.cn/problems/longest-substring-without-repeating-characters/)

|  |
| --- |
| // 滑动窗口 + 对于每一次的判断是否有重复字符，用set的has来检索判断 // 关键是用set当成窗口，delete调整左边界，add调整右边界，has判断是否存在重复 function longestSubstring(s) {  let maxSubstring = '', maxSublength = 0;   const windowSet = new ***Set***();  let right\_index = -1;  for (let left\_index = 0;left\_index < s.length; left\_index++){  if (left\_index !== 0){ //说明进入下一次for循环了，要调整左边界向右移一位，即删除左边第一位字符串  windowSet.delete(s.charAt(left\_index-1));  }   while ( !windowSet.has(s.charAt(right\_index+1)) && right\_index+1 < s.length){  windowSet.add(s.charAt(right\_index+1));  right\_index ++;  }   if (right\_index - left\_index + 1 > maxSublength){  maxSublength = right\_index -left\_index + 1;  maxSubstring = s.substring(left\_index, right\_index + 1);  }  }   return [maxSublength, maxSubstring]; } |

* 判断字符串回文

[19道常见的JS面试算法题 - tiana\_Z - 博客园 (cnblogs.com)](https://www.cnblogs.com/djw12333/p/11647413.html)

|  |
| --- |
| let ***str*** = 'abcdefedcba'; let ***str2*** = 'abcdeffedcba'; let ***str3*** = 'abcdefFedcba';  function isMirrorString(str) {  if (!str) return;   for (let i = 0; i < str.length/2; ++i){  if (str[i] !== str[str.length - i - 1]){  return false;  }  }  return true; }  ***console***.log(isMirrorString(***str***)); ***console***.log(isMirrorString(***str2***)); ***console***.log(isMirrorString(***str3***)); |

* 最长前缀

[14. 最长公共前缀 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/longest-common-prefix/submissions/)

|  |
| --- |
| let ***strs*** = [  'abcde',  'abcdeFGHIJK',  'abcdefghijk',  'abcdE' //,'x' ]  // 随便找一个作为标准比较就行，因为最长都不会超过任何一个的长度，这里直接找strs[0] function longestCommonPrefix(strs) {  if (strs.length === 0 ) return '';   let commonPrefix = strs[0];  for (let str of strs) {  for (let index = 0; index < commonPrefix.length; ++index) {  if (str[index] !== commonPrefix[index]){  if (index === 0) return '';  commonPrefix = commonPrefix.substring(0,index);  break;  }  }  }   return commonPrefix; }  ***console***.log(longestCommonPrefix(***strs***)) |

* 连续序列最长长度

[128. 最长连续序列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/longest-consecutive-sequence/submissions/)

set 去重+起点遍历（！set.has(item-1)）

|  |
| --- |
| ****输入：****nums = [100,4,200,1,3,2]  ****输出：****4  ****解释：****最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。  // set 去重+起点遍历（！set.has(item-1)） function longestSubList(list) {  let length = 0;   const set = new ***Set***(list);  for (let setElement of set) {  if (!set.has(setElement - 1)){ //set中没有比当前setElement小1的数，这个数是起点，开始遍历。  let templength = 1;  while (set.has(setElement + templength)){  templength++;  }   length = ***Math***.max(length,templength);  }  }   return length; }   let ***list*** = [12,2,3,887,5,6,9,7,11,15,3,5,2,6,15]; ***console***.log(longestSubList(***list***)); |

* 去掉字符串前后的空格

[JavaScript 正则表达式 | 菜鸟教程 (runoob.com)](https://www.runoob.com/js/js-regexp.html)

去除所有空格:

str = str.replace(/\s+/g,"")

去除两头空格:

str = str.replace(/^\s+|\s+$/g,"")

去除左空格：

str=str.replace( /^\s/, '')

去除右空格：

str=str.replace(/(\s$)/g, "")

* 大数相乘（字符串相乘）

|  |
| --- |
| var multiply = function(num1, num2) {      let m = num1.length;      let n = num2.length;      let res = new Array(m+n).fill(0);      for(let i = m-1; i>=0; i--){          for(let j=n-1; j>=0; j--){              let mul = (num1[i]-0)\*(num2[j]-0);              let p1 = i+j;              let p2 = i+j+1;              let sum = res[p2]+mul;              res[p2] = sum % 10;              res[p1] += parseInt(sum/10);          }      }      let i = 0;      while(i<res.length&&res[i]==0){          i++;      }      let str = "";      for(;i<res.length;i++){          str += res[i];      }      return str.length == 0? "0":str;  }; |

* 朋友圈 (set)

|  |
| --- |
| obj={  a:[‘b’,’c’],  b:[‘a’],  c:[‘a’],  d:[]  }  输出2，有两个朋友圈（[a,b,c],[d]） |
| var obj = {  a:['b','c'],  b:['a','e'],  c:['a'],  d:[],  e:['b'],  f:[] }  function friendCircle(obj){  var circles = [];  for(let key in obj){  var bool = true;   for (let circle of circles){  if (circle.has(key)){  for(let item of obj[key]){ //[b,c]  circle.add(item);  }  bool = false;  }  }  if(bool){  var set = new Set(key);  for(let item of obj[key]){ //[b,c]  set.add(item);  }  circles.push(set);  }   }  console.log(circles);  console.log(circles.length); }  friendCircle(obj) |

* 判断素数

[5种你不知道的素数的判断方法 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/104314640)

# JS函数实现：

* 数组平铺/扁平flat

[js数组拍平(数组扁平化)的八种方式 - 掘金 (juejin.cn)](https://juejin.cn/post/6844903998223286285)

|  |
| --- |
| var ***arr*** = [1,2,[3,4,5,[6,[7,8]],9],10,[11,12]]; //toString + map function stringFlat(arr){  //console.log(arr.toString().split(','))  //['1', '2', '3', '4','5', '6', '7', '8','9', '10', '11', '12']  arr = arr.toString().split(',').map((value,index,array)=>{  return ***Number***(value)  })  return arr; } ***console***.log(stringFlat(***arr***))  // 原生flat // reduce((prev,cur,index,arr)=>{},init)用法 // 两个参数，一个callback,一个初始值 // 回调的四个参数分别为，上一次 return的结果，当前项，当前索引，数组，还有初始值。 // reduce实现什么功能取决于回调内部做了什么 function flat(arr){  if(! arr instanceof ***Array***){  return false;  }   let res = arr.reduce((pre,cur)=>{  if(***Array***.isArray(cur)){  return [...pre,...flat(cur)]  }else{  return [...pre,cur]  }  },[])    return res; } ***console***.log(flat(***arr***))  //选择扁平层次 function selectFlat(arr,level = 1) {  if(! arr instanceof ***Array***){  return false;  }   let res = arr.reduce((pre,cur)=>{  if(***Array***.isArray(cur) && level>0){  return [...pre,...selectFlat(cur,level-1)]  }else {  return [...pre,cur]  }  },[])   return res; } ***console***.log(selectFlat(***arr***,2)) |

* instanceof

|  |
| --- |
| function instanceOf(leftObj,rightFunction){  let leftProto = ***Object***.getPrototypeOf(leftObj);  let rightPrototype = rightFunction.prototype;  while(true){  if(leftProto === rightPrototype){  return true  }else if(leftProto === null){  return false  }  leftProto = ***Object***.getPrototypeOf(leftProto)  } } const ***array*** = [1,2,3,4,5,6] ***console***.log(instanceOf(***array***,***Array***)) ***console***.log(instanceOf(***array***,***Object***)) ***console***.log(instanceOf(***array***,***Number***)) |

* new函数

|  |
| --- |
| function Person(name, age) {  this.name = name;  this.age = age; }  function newFunc(Func,...args) {  //new obj  let obj = {};  //obj.\_\_proto\_\_ = Func.prototype  ***Object***.setPrototypeOf(obj,Func.prototype)  //console.log(Object.getPrototypeOf(obj) === Func.prototype)  //this  let res = Func.apply(obj,args);  //constructor  return res instanceof Func ? res : obj; }  var ***ob*** = newFunc(Person,'name',10086); ***console***.log(***ob***.name) ***console***.log(***ob***.age) |

* bind函数
* url 解析 query

[获取地址中的query参数，完美版 - 简书 (jianshu.com)](https://www.jianshu.com/p/0f8167363148)

|  |
| --- |
| const urlString = 'https://www.example.com/test.html?a=param1&b=param2';//decodeURL(urlString)  function parseSearchParams(urlString){  const list = urlString.split('?');    if(list.length>1){  var obj = {};  var jquryList = list[1].split('&');  for(let i = 0; i < jquryList.length; i++){  let cur = jquryList[i].split('=');  obj[cur[0]] = cur[1];//decodeURLComponent(cur[1])  }  return obj;  }  return ;  }  parseSearchParams(urlString); |

* JS 闭包修改值

[JavaScript之更改闭包内的变量值 - Lowki - 博客园 (cnblogs.com)](https://www.cnblogs.com/lowki/p/10406320.html)

|  |  |
| --- | --- |
| |  | | --- | | 正确写法：  var foo = function(){      let val = 1;      function fo(){          console.log(val);          val++;      }      fo.reset = function (){          val = 1;      }      return fo;  }  var a = foo();  a();  a();  a.reset();  a(); | |

* 防抖节流

[JavaScript---节流与防抖 - 掘金 (juejin.cn)](https://juejin.cn/post/7030787304696315918)

[面试官：什么是防抖和节流？有什么区别？如何实现？ · Issue #83 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/83)（防抖debounce、节流throttle）

|  |
| --- |
| **节流**  使用时间戳写法，事件会立即执行，停止触发后没有办法再次执行  function throttled1(fn, delay = 500) {  let oldtime = Date.now()  return function (...args) {  let newtime = Date.now()  if (newtime - oldtime >= delay) {  fn.apply(null, args)  oldtime = Date.now()  }  }  }  **使用定时器写法，delay毫秒后第一次执行，第二次事件停止触发后依然会再一次执行**  **function throttled2(fn, delay = 500) {**  **let timer = null**  **return function (...args) {**  **if (!timer) {**  **timer = setTimeout(() => {**  **fn.apply(this, args)**  **timer = null**  **}, delay);**  **}**  **}**  **}**  可以将时间戳写法的特性与定时器写法的特性相结合，实现一个更加精确的节流。实现如下  function throttled(fn, delay) {  let timer = null  let starttime = Date.now()  return function () {  let curTime = Date.now() // 当前时间  let remaining = delay - (curTime - starttime) // 从上一次到现在，还剩下多少多余时间  let context = this  let args = arguments  clearTimeout(timer)  if (remaining <= 0) {  fn.apply(context, args)  starttime = Date.now()  } else {  timer = setTimeout(fn, remaining);  }  }  } |
| **防抖**  简单版本的实现  **function debounce(func, wait) {**  **let timeout;**  **return function () {**  **let context = this; // 保存this指向**  **let args = arguments; // 拿到event对象**  **clearTimeout(timeout)**  **timeout = setTimeout(function(){**  **func.apply(context, args)**  **}, wait);**  **}**  **}**  防抖如果需要立即执行，可加入第三个参数用于判断，实现如下：  function debounce(func, wait, immediate) {  let timeout;  return function () {  let context = this;  let args = arguments;  if (timeout) clearTimeout(timeout); // timeout 不为null  if (immediate) {  let callNow = !timeout; // 第一次会立即执行，以后只有事件执行后才会再次触发  timeout = setTimeout(function () {  timeout = null;  }, wait)  if (callNow) {  func.apply(context, args)  }  }  else {  timeout = setTimeout(function () {  func.apply(context, args)  }, wait);  }  }  } |

# JS继承实现：

* 原型链继承

[【前端工程师面试宝典】学习说明\_互联网校招面试真题面经汇总\_牛客网 (nowcoder.com)](https://www.nowcoder.com/tutorial/96/1678a0fd35cd4db486af18589e34e4d4)

[JavaScript 继承的八种方式 - 云+社区 - 腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1851143)

|  |
| --- |
| 原型链继承  实现方式：将子类的原型链指向父类的对象实例  function Parent(){  this.name = "parent";  this.list = ['a'];  }  Parent.prototype.sayHi = function(){  console.log('hi');  }  function Child(){  }  Child.prototype = new Parent();  var child = new Child();  console.log(child.name);  child.sayHi();  原理：子类实例child的\_\_proto\_\_指向Child的原型链prototype，而Child.prototype指向Parent类的对象实例，该父类对象实例的\_\_proto\_\_指向Parent.prototype,所以Child可继承Parent的构造函数属性、方法和原型链属性、方法 优点：可继承构造函数的属性，父类构造函数的属性，父类原型的属性 缺点：无法向父类构造函数传参；且所有实例共享父类实例的属性，若父类共有属性为引用类型，一个子类实例更改父类构造函数共有属性时会导致继承的共有属性发生变化；实例如下：  var a = new Child();  var b = new Child();  a.list.push('b');  console.log(b.list); // ['a','b'] |
| // 原理：子类实例child的\_\_proto\_\_指向Child的原型链prototype，而Child.prototype指向Parent类的对象实例， // 该父类对象实例的\_\_proto\_\_指向Parent.prototype,所以Child可继承Parent的构造函数属性、方法和原型链属性、方法。  // 优点：可继承构造函数的属性，父类构造函数的属性，父类原型的属性 // 缺点：无法向父类构造函数传参；且所有实例共享父类实例的属性，若父类共有属性为引用类型，一个子类实例更改父类构造函数共有属性时会导致继承的共有属性发生变化；实例如下：  // 1. 父类声明，属性和方法实现 function Parent(name) {  //父类的 非引用 属性  this.name = name;  //父类的 引用 属性  this.school = ['小学','初中','高中'] }  Parent.prototype.printInfomation = function () {  ***console***.log('名字:');  ***console***.log(this.name);  ***console***.log('学历:');  ***console***.log(this.school); }  // 2. 子类声明 // (子类中不重写的属性直接调用父类的属性，如Child可以不写name，school) function Child(name) {  this.name = name; }  // 3. 原型链继承 —— Child.prototype指向Parent类的对象实例 (子类指向父实例) Child.prototype = new Parent();  // 4. 输出验证结果 const ***studentA*** = new Child('studentA'); const ***studentB*** = new Child('studentB');  ***studentA***.printInfomation(); ***studentB***.printInfomation();  ***studentB***.school.push('大学'); // 改了引用类型，所有child实例的school都被改了  ***studentA***.printInfomation(); ***studentB***.printInfomation(); |

* 构造函数继承

|  |
| --- |
| 构造函数继承  实现方式：在子类构造函数中使用call或者apply劫持父类构造函数方法，并传入参数  function Parent(name, id){  this.id = id;  this.name = name;  this.printName = function(){  console.log(this.name);  }  }  Parent.prototype.sayName = function(){  console.log(this.name);  };  function Child(name, id){  Parent.call(this, name, id);  // Parent.apply(this, arguments);  }  var child = new Child("jin", "1");  child.printName(); // jin  child.sayName() // Error  原理：使用call或者apply更改子类函数的作用域，使this执行父类构造函数，子类因此可以继承父类共有属性 优点：可解决原型链继承的缺点 缺点：不可继承父类的原型链方法，构造函数不可复用 |
| // 原理：使用call或者apply更改子类函数的作用域，使this执行父类构造函数，子类因此可以继承父类共有属性 // 优点：可解决原型链继承的缺点 // 缺点：不可继承父类的原型链方法，构造函数不可复用  // 1. 父类声明，属性和方法实现 function Parent(name) {  //父类的 非引用 属性  this.name = name;  //父类的 引用 属性  this.school = ['小学','初中','高中'];  //  this.print = function () {  ***console***.log('名字:');  ***console***.log(this.name);  ***console***.log('学历:');  ***console***.log(this.school);  } }  // 2. 子类声明 使用call或者apply更改子类函数的作用域，使this执行父类构造函数，子类因此可以继承父类共有属性 function Child(name) {  Parent.call(this,name);  // Parent.call(this,...arguments);  // Parent.apply(this,arguments); }  let ***studentA*** = new Child('studentA'); let ***studentB*** = new Child('studentB');  // ***studentA***.print(); ***studentB***.print();  ***studentB***.school.push('大学');  // 结果不互相影响 ***studentA***.print(); ***studentB***.print();  // 不能调用 // Parent.prototype.printInfomation = function () { // console.log('名字:'); // console.log(this.name); // console.log('学历:'); // console.log(this.school); // } // // studentA.printInfomation(); // studentB.printInfomation(); |

* 组合继承

|  |
| --- |
| 组合继承  原理：综合使用构造函数继承和原型链继承  function Parent(name, id){  this.id = id;  this.name = name;  this.list = ['a'];  this.printName = function(){  console.log(this.name);  }  }  Parent.prototype.sayName = function(){  console.log(this.name);  };  function Child(name, id){  Parent.call(this, name, id);  // Parent.apply(this, arguments);  }  Child.prototype = new Parent();  var child = new Child("jin", "1");  child.printName(); // jin  child.sayName() // jin  var a = new Child();  var b = new Child();  a.list.push('b');  console.log(b.list); // ['a']  优点：可继承父类原型上的属性，且可传参；每个新实例引入的构造函数是私有的 缺点：会执行两次父类的构造函数，消耗较大内存，子类的构造函数会代替原型上的那个父类构造函数 |
| // 组合继承 // 原理：综合使用构造函数继承和原型链继承  // 优点：可继承父类原型上的属性，且可传参；每个新实例引入的构造函数是私有的 // 缺点：会执行两次父类的构造函数，消耗较大内存，子类的构造函数会代替原型上的那个父类构造函数  function Parent(name) {  //父类的 非引用 属性  this.name = name;  //父类的 引用 属性  this.school = ['小学','初中','高中'];  //  this.print = function () {  ***console***.log('名字:');  ***console***.log(this.name);  ***console***.log('学历:');  ***console***.log(this.school);  } }  Parent.prototype.printInfomation = function () {  ***console***.log('名字:');  ***console***.log(this.name);  ***console***.log('学历:');  ***console***.log(this.school); }  // 构造函数继承 function Child(name) {  Parent.call(this,name); }  // 原型链继承 Child.prototype = new Parent();   // 输出验证 let ***studentA*** = new Child('studentA'); let ***studentB*** = new Child('studentB');  ***console***.log('------------------------父类的内部函数print：---------------------------'); ***studentA***.print(); ***studentB***.print(); ***console***.log('------------------------父类的原型链函数printInfomation：---------------'); ***studentA***.printInfomation(); ***studentB***.printInfomation();  ***studentB***.school.push('大学');  // 结果不互相影响 ***console***.log('------------------------父类的内部函数print：---------------------------'); ***studentA***.print(); ***studentB***.print(); ***console***.log('------------------------父类的原型链函数printInfomation：---------------'); ***studentA***.printInfomation(); ***studentB***.printInfomation(); |

* 原型式继承

|  |
| --- |
| 原型式继承  原理：类似Object.create，用一个函数包装一个对象，然后返回这个函数的调用，这个函数就变成了个可以随意增添属性的实例或对象，结果是将子对象的\_\_proto\_\_指向父对象  var parent = {  names: ['a']  }  function copy(object) {  function F() {}  F.prototype = object;  return new F();  }  var child = copy(parent);  缺点：共享引用类型 |
| // 原型式继承 // 原理：类似Object.create，用一个函数包装一个对象，然后返回这个函数的调用，这个函数就变成了个可以随意增添属性的实例或对象，结果是将子对象的\_\_proto\_\_指向父对象  // 缺点：共享引用类型  // 1. 父对象实例声明（不是父类） var ***parent*** = {  name: 'parent',  school: ['小学','初中','高中'],  print: function () {  ***console***.log('名字:');  ***console***.log(this.name);  ***console***.log('学历:');  ***console***.log(this.school);  } }  // 2. 返回对父对象实例的函数的调用 function createChild(parentObj) {  function f() {   }   f.prototype = parentObj;   return new f();  }  // 3. 实例化子对象 var ***studentA*** = createChild(***parent***); var ***studentB*** = createChild(***parent***);  // 4. 输出验证结果 ***console***.log('------------------------修改前：---------------------------'); ***studentA***.print(); ***studentB***.print();  ***studentA***.name = 'studentA'; ***studentA***.school.push('大学');  ***console***.log('------------------------修改后：---------------------------'); ***studentA***.print(); ***studentB***.print(); |

* 寄生式继承

|  |
| --- |
| 寄生式继承  原理：二次封装原型式继承，并拓展  function createObject(obj) {  var o = copy(obj);  o.getNames = function() {  console.log(this.names);  return this.names;  }  return o;  }  优点：可添加新的属性和方法 |
| // 寄生式继承 // 原理：二次封装原型式继承，并拓展  // 优点：可添加新的属性和方法  var ***parent*** = {  name: 'parent',  school: ['小学','初中','高中'],  print: function () {  ***console***.log('名字:');  ***console***.log(this.name);  ***console***.log('学历:');  ***console***.log(this.school);  } }  function createChild(parentObj) {  function f() {}  f.prototype = parentObj;  return new f(); }  // 再次封装creatChild function inherit(parentObj) {  var obj = createChild(parentObj);   // 新增函数  obj.printName = function () {  ***console***.log(this.name);  }  obj.printSchool = function () {  ***console***.log(this.school);  }   return obj; }  // 3. 实例化子对象 var ***studentA*** = inherit(***parent***); var ***studentB*** = inherit(***parent***);  // 4. 输出验证结果 ***console***.log('------------------------修改前：--------------------------'); ***studentA***.print(); ***studentB***.print();  ***studentA***.name = 'studentA'; ***studentA***.school.push('大学');  ***console***.log('------------------------修改后：--------------------------'); ***studentA***.print(); ***studentB***.print(); ***console***.log('---------------------依旧存在共享引用问题---------------------');  ***console***.log('---------------------使用新的拓展方法：--------------------------'); ***studentA***.printName(); ***studentA***.printSchool(); ***studentB***.printName(); ***studentB***.printSchool(); |

* 寄生组合式继承

|  |
| --- |
| 寄生组合式继承  原理：改进组合继承，利用寄生式继承的思想继承原型  function inheritPrototype(subClass, superClass) {  // 复制一份父类的原型  var p = copy(superClass.prototype);  // 修正构造函数  p.constructor = subClass;  // 设置子类原型  subClass.prototype = p;  }  function Parent(name, id){  this.id = id;  this.name = name;  this.list = ['a'];  this.printName = function(){  console.log(this.name);  }  }  Parent.prototype.sayName = function(){  console.log(this.name);  };  function Child(name, id){  Parent.call(this, name, id);  // Parent.apply(this, arguments);  }  inheritPrototype(Child, Parent); |
| // 寄生组合式继承 // 原理：改进组合继承，利用寄生式继承的思想继承原型  // 组合继承 function Parent(name) {  //父类的 非引用 属性  this.name = name;  //父类的 引用 属性  this.school = ['小学','初中','高中'];  //  this.print = function () {  ***console***.log('名字:');  ***console***.log(this.name);  ***console***.log('学历:');  ***console***.log(this.school);  } }  Parent.prototype.printInfomation = function () {  ***console***.log('名字:');  ***console***.log(this.name);  ***console***.log('学历:');  ***console***.log(this.school); }  function Child(name) {  Parent.call(this,name); }  // 组合继承的这一句用寄生式替代 // Child.prototype = new Parent();  // 寄生式继承 function createChild(parentObj) {  function f() {}  f.prototype = parentObj;  return new f(); }  function inherit(childClass, ParentClass) {  // 复制一份父类的原型  var p = createChild(ParentClass.prototype);  // 修正构造函数  p.constructor = childClass;  // 设置子类原型  childClass.prototype = p; }  inherit(Child, Parent);  // 输出验证结果 const ***studentA*** = new Child('studentA'); const ***studentB*** = new Child('studentB');  ***console***.log('------------------------------------初始状态----------------------------------'); ***console***.log('------------studentA:print()---------------------'); ***studentA***.print(); ***console***.log('------------studentA:printInfomation()-----------'); ***studentA***.printInfomation(); ***console***.log('------------studentB:print()---------------------'); ***studentB***.print(); ***console***.log('------------studentB:printInfomation()-----------'); ***studentB***.printInfomation();  ***studentA***.name = 'studentAstudentAstudentA'; ***studentA***.school.push('大学'); ***studentB***.name = 'studentBstudentBstudentB'; ***studentB***.school.push('大学'); ***studentB***.school.push('毕业寄');  ***console***.log('------------------------------------修改后状态----------------------------------'); ***console***.log('------------studentA:print()---------------------'); ***studentA***.print(); ***console***.log('------------studentA:printInfomation()-----------'); ***studentA***.printInfomation(); ***console***.log('------------studentB:print()---------------------'); ***studentB***.print(); ***console***.log('------------studentB:printInfomation()-----------'); ***studentB***.printInfomation(); |

* ES6 class继承

|  |
| --- |
| **class 继承**  ES6 中，通过 class 关键字来定义类，子类可以通过 extends 继承父类。  代码示例：  class Parent{  constructor(name) {  this.name = name;  this.hobbies = ["sing", "dance", "rap"];  }  getHobbies() {  return this.hobbies;  }  static getCurrent() {  console.log(this);  }}  class Child extends Parent {  constructor(name) {  super(name);  }}  var c1 = new Child('c1');var c2 = new Child('c2');  console.log(c1 instanceof Child); // true  console.log(c1 instanceof Parent); // true  要点：   * constructor 为构造函数，即使未定义也会自动创建。 * 在父类构造函数内 this 定义的都是实例属性和方法，其他方法包括 、getHobbies 都是原型方法。 * static 关键字定义的静态方法都必须通过类名调用，其 this 指向调用者而并非实例。 * 通过 extends 可以继承父类的所有原型属性及 static 类方法，子类 constructor 调用 super 父类构造函数实现实例属性和方法的继承。   对比：   * ES5 的继承，实质是先创造子类的实例对象 this，然后再将父类的方法添加到 this 上面（Parent.apply(this)）。   ES6 的继承机制完全不同，实质是先将父类实例对象的属性和方法，加到 this 上面（所以必须先调用 super 方法），然后再用子类的构造函数修改 this。 |
| // 要点： // constructor 为构造函数，即使未定义也会自动创建。 // 在父类构造函数内 this 定义的都是实例属性和方法，其他方法包括 printInfomation、getHobbies 都是原型方法。 // static 关键字定义的静态方法都必须通过类名调用，其 this 指向调用者而并非实例。 // 通过 extends 可以继承父类的所有原型属性及 static 类方法，子类 constructor 调用 super 父类构造函数实现实例属性和方法的继承。 // // 对比： // ES5 的继承，实质是先创造子类的实例对象 this，然后再将父类的方法添加到 this 上面（Parent.apply(this)）。 // ES6 的继承机制完全不同，实质是先将父类实例对象的属性和方法，加到 this 上面（所以必须先调用 super 方法），然后再用子类的构造函数修改 this。  class Parent {  // 构造函数内 this 是实例属性和方法，即每一个实例都会新创建一份name和school，解决引用问题。  constructor(name) {  this.name = name;  this.school = ['小学','初中','高中'];  }   // 不在构造函数里面，是原型方法(或属性)，即理解的在原型链上写方法函数，不用每一个实例内部都创建一个方法。  // 相当于前面写过的 Parent.prototype.printInfomation = function () {}  printInfomation() {  ***console***.log('名字:');  ***console***.log(this.name);  ***console***.log('学历:');  ***console***.log(this.school);  }   // static方法必须用Parent.printThis()  static *printThis*() {  ***console***.log(this);  } }  // 通过 extends 可以继承父类的所有原型属性及 static 类方法，子类 constructor 调用 super 父类构造函数实现实例属性和方法的继承。 class Child extends Parent {  constructor(name) {  super(name);  } }  // 输出验证结果 const ***studentA*** = new Child('studentA'); const ***studentB*** = new Child('studentB');  ***console***.log('------------------------------------初始状态----------------------------------'); ***console***.log('------------studentA--------------'); ***console***.log('实例属性name，shcool：'); ***console***.log(***studentA***.name); ***console***.log(***studentA***.school); ***console***.log('原型方法printInfomation：'); ***studentA***.printInfomation(); ***console***.log('static 方法printThis，调用的this为：'); Parent.*printThis*(); ***console***.log('------------studentB--------------'); ***console***.log('实例属性name，shcool：'); ***console***.log(***studentB***.name); ***console***.log(***studentB***.school); ***console***.log('原型方法printInfomation：'); ***studentB***.printInfomation(); ***console***.log('static 方法printThis，调用的this为：'); Parent.*printThis*();   ***studentA***.name = 'studentAstudentAstudentA'; ***studentA***.school.push('大学'); ***studentB***.name = 'studentBstudentBstudentB'; ***studentB***.school.push('大学'); ***studentB***.school.push('毕业寄');   ***console***.log('------------------------------------修改后状态----------------------------------'); ***console***.log('------------studentA--------------'); ***console***.log('实例属性name，shcool：'); ***console***.log(***studentA***.name); ***console***.log(***studentA***.school); ***console***.log('原型方法printInfomation：'); ***studentA***.printInfomation(); ***console***.log('static 方法printThis，调用的this为：'); Parent.*printThis*(); ***console***.log('------------studentB--------------'); ***console***.log('实例属性name，shcool：'); ***console***.log(***studentB***.name); ***console***.log(***studentB***.school); ***console***.log('原型方法printInfomation：'); ***studentB***.printInfomation(); ***console***.log('static 方法printThis，调用的this为：'); Parent.*printThis*(); |

# 正则匹配：

[正则表达式在线测试 - 站长工具 (chinaz.com)](https://tool.chinaz.com/regex)

[js正则表达式及练习（超详细）\_小生听雨园的博客-CSDN博客\_js正则表达式练习](https://blog.csdn.net/weixin_44154094/article/details/112685412)

[正则表达式练习（js）\_白小宇的博客-CSDN博客](https://blog.csdn.net/b954960630/article/details/82634008)

* 匹配规则

|  |
| --- |
| 1. | 表示或 2. [] 中括号里也是或的关系 [ab] == ***a***|***b***; 3. [***a***-z] 任意小写字母 4. [A-z] 任意字母 5. [^ ] 除了\*\*\*以外 6. [0-9] 任意数字 7. . 匹配任意1个字符，除了换行和行结束符(\n) 8. \w 任意字母数字下划线——匹配非特殊字符，即a-z、A-Z、0-9、\_、汉字 9. \W 除了字母数字下划线——匹配特殊字符，即非字母、非数字、非汉字、非下划线 10. \d [0-9]——匹配数字，即0-9 11. \D [^0-9]——匹配非数字，即不是数字 12. \***s*** 空格——匹配空白，即 空格，tab键 13. \S 除了空格——匹配非空白 14. \***b*** 单词边界 15. \B 除了单词边界 16. /***b***{1,3}/ :b出现一次到三次 17. /***b***{3}/ :b出现三次 18. /(ab){3}/ :ab一起出现三次 19. /***b***{m,}/ :b出现m次以上 20. + :至少一个 21. \* :0个或多个 22. ? :0个或一个 23. ^ :以\*\*\*开头，他和[]里的^不一样 24. $ :以\*\*\*结尾 25. 查找.和/的话需要\进行转义 |

* 匹配规则使用实例1~7

|  |
| --- |
| // \* 1. | 表示或 // 匹配 包含 ab或Ab 的字符串 var ***regex1*** = /ab|Ab/ ; ***console***.log(1); ***console***.log(***regex1***.test("---abcd---")); // T ***console***.log(***regex1***.test("---Abcd---")); // T ***console***.log(***regex1***.test("---ABcd---")); // F ***console***.log(***regex1***.test("---aBcd---")); // F  // \* 2. [] 中括号里也是或的关系 [ab] == a|b; // 匹配 包含 abd或acd 的字符串 var ***regex2*** = /a[bc]d/ ; ***console***.log(2); ***console***.log(***regex2***.test("---abd---")); // T ***console***.log(***regex2***.test("---acd---")); // T ***console***.log(***regex2***.test("---abcd---")); // F ***console***.log(***regex2***.test("---add---")); // F  // \* 3. [a-z] 任意小写字母 // 匹配 ”[a-z]是小写字母“ 的字符串 var ***regex3*** = /[a**-**z]是小写字母/ ***console***.log(3); ***console***.log(***regex3***.test("a是小写字母")); // T ***console***.log(***regex3***.test("z是小写字母")); // T ***console***.log(***regex3***.test("az是小写字母")); // T ***console***.log(***regex3***.test("a?是小写字母")); // F ***console***.log(***regex3***.test("?是小写字母")); // F  // \* 4. [A-z] 任意字母 // 匹配 ”[A-z]是字母“ 的字符串 var ***regex4*** = /[A**-**z]是字母/ ***console***.log(4); ***console***.log(***regex4***.test("a是字母")); // T ***console***.log(***regex4***.test("Z是字母")); // T ***console***.log(***regex4***.test("Az是字母")); // T ***console***.log(***regex4***.test("a8是字母")); // F ***console***.log(***regex4***.test("8是字母")); // F  // \* 5. [^ ] 除了\*\*\*以外 // 匹配 除 ”打死你“ 以外的 ”打？你“ 的字符串 var ***regex5*** = /打[^死]你/ ; ***console***.log(5); ***console***.log(***regex5***.test("我打死你")); // F ***console***.log(***regex5***.test("我打败你")); // T ***console***.log(***regex5***.test("我打打你")); // T ***console***.log(***regex5***.test("我打你")); // F  // \* 6. [0-9] 任意数字 // 匹配 ”我的排名是[0-9]“ 的字符串 var ***regex6*** = /我的排名是[0**-**9]/ ; ***console***.log(6); ***console***.log(***regex6***.test("我的排名是1")); // T ***console***.log(***regex6***.test("我的排名是 1")); // F ***console***.log(***regex6***.test("我的排名是:2")); // F ***console***.log(***regex6***.test("我的排名是一")); // F ***console***.log(***regex6***.test("哇哦！我的排名是1")); // T  // \* 7. . 匹配任意1个字符，除了换行和行结束符(\n) // 匹配 形式为 ”-->？<--“ 的字符串 var ***regex7*** = /-->.<--/ ; ***console***.log(7); ***console***.log(***regex7***.test("-->.<--")); // T ***console***.log(***regex7***.test("--> <--")); // T ***console***.log(***regex7***.test("--><--")); // F ***console***.log(***regex7***.test("-->W<--")); // T ***console***.log(***regex7***.test("-->QWQ<--")); // F ***console***.log(***regex7***.test("-->\n<--")); // F |

* 匹配规则使用实例8~15

|  |
| --- |
| // \* 8. \w 任意字母数字下划线——匹配非特殊字符，即a-z、A-Z、0-9、\_ //w 能不能匹配汉字要视你的操作系统和你的应用环境而定 var ***regex8*** = /\w\w\w\w/ ; ***console***.log(8); ***console***.log(***regex8***.test("aZ9\_")); // T ***console***.log(***regex8***.test("aZ9我")); // F ***console***.log(***regex8***.test("aZ9?")); // F  // \* 9. \W 除了字母数字下划线——匹配特殊字符，即非字母、非数字、非汉字、非下划线 var ***regex9*** = /\W\W\W/; ***console***.log(9); ***console***.log(***regex9***.test("aA0")); // F ***console***.log(***regex9***.test("aA0\_")); // F ***console***.log(***regex9***.test("aA0?!?")); // T ***console***.log(***regex9***.test("{?}")) // T  // \* 10. \d [0-9]——匹配数字，即0-9 var ***regex10*** = /我的排名是\d/ ; ***console***.log(10); ***console***.log(***regex10***.test("我的排名是2")); // T ***console***.log(***regex10***.test("我的排名是 2")); // F ***console***.log(***regex10***.test("我的排名是:2")); // F ***console***.log(***regex10***.test("我的排名是一")); // F ***console***.log(***regex10***.test("哇哦！我的排名是1")); // T  // \* 11. \D [^0-9]——匹配非数字，即不是数字 var ***regex11*** = /\D不是数字/ ; ***console***.log(11); ***console***.log(***regex11***.test("100不是数字")); // F ***console***.log(***regex11***.test("一百不是数字")); // T ***console***.log(***regex11***.test("？？？不是数字")); // T  // \* 12. \s 空格——匹配空白，即 空格 var ***regex12*** = /1\s2\s3/ ; ***console***.log(12); ***console***.log(***regex12***.test("1 2 3")); // T ***console***.log(***regex12***.test("1 2 3")); // F ***console***.log(***regex12***.test("123")); // F  // \* 13. \S 除了空格——匹配非空白 var ***regex13*** = /1\S2\S3/ ; ***console***.log(13); ***console***.log(***regex13***.test("1?2?3")); // T ***console***.log(***regex13***.test("1 2 3")); // F ***console***.log(***regex13***.test("123")); // F ***console***.log(***regex13***.test("1??2??3")); // F  // \* 14. \b 单词边界 var ***regex14*** = /\bshoe\b/ ; ***console***.log(14); ***console***.log(***regex14***.test("Here is one shoe.")); // T ***console***.log(***regex14***.test("Here are two shoes.")); // F ***console***.log(***regex14***.test("Hereisoneshoe")); // F ***console***.log(***regex14***.test("Herearetwoshoes")); // F  // \* 15. \B 除了单词边界 var ***regex15*** = /\Bshoe\B/ ; ***console***.log(15); ***console***.log(***regex15***.test("Here is one shoe.")); // F ***console***.log(***regex15***.test("Here are two shoes.")); // F ***console***.log(***regex15***.test("Hereisoneshoe")); // F ***console***.log(***regex15***.test("Herearetwoshoes")); // T |

* 匹配规则使用实例16~25

|  |
| --- |
| // \* 16. /b{1,3}/ :b出现一次到三次 var ***regex16*** = /ab{1,3}/ ; //匹配 a后面跟着1~3个b 的字符串 ***console***.log(16); ***console***.log(***regex16***.test("aa")); // F ***console***.log(***regex16***.test("ab")); // T ***console***.log(***regex16***.test("abb")); // T ***console***.log(***regex16***.test("abbb")); // T ***console***.log(***regex16***.test("abbbb")); // T  // \* 17. /b{3}/ :b出现三次 var ***regex17*** = /ab{3}/ ; //匹配 a后面跟着3个b 的字符串 ***console***.log(17); ***console***.log(***regex17***.test("ab")); // F ***console***.log(***regex17***.test("abb")); // F ***console***.log(***regex17***.test("abbb")); // T ***console***.log(***regex17***.test("abbbb")); // T ***console***.log(***regex17***.test("abAbb")); // F  // \* 18. /(ab){3}/ :ab一起出现三次 var ***regex18*** = /(ab){3}/ ; ***console***.log(18); ***console***.log(***regex18***.test("ab")); // F ***console***.log(***regex18***.test("abab")); // F ***console***.log(***regex18***.test("ababab")); // T ***console***.log(***regex18***.test("abABabABab")); // F  // \* 19. /b{m,}/ :b出现m次以上 var ***regex19*** = /b{3,}/ ; ***console***.log(19); ***console***.log(***regex19***.test("b")); // F ***console***.log(***regex19***.test("bb")); // F ***console***.log(***regex19***.test("bbb")); // T ***console***.log(***regex19***.test("bbbb")); // T ***console***.log(***regex19***.test("bababab")); // F  // \* 20. + :至少一个 var ***regex20*** = /ab+/ ; ***console***.log(20); ***console***.log(***regex20***.test("a")); // F ***console***.log(***regex20***.test("ab")); // T ***console***.log(***regex20***.test("abb")); // T ***console***.log(***regex20***.test("acb")); // F  // \* 21. \* :0个或多个 var ***regex21*** = /ab\*/ ; ***console***.log(21); ***console***.log(***regex21***.test("a")); // T ***console***.log(***regex21***.test("ac")); // T ***console***.log(***regex21***.test("ab")); // T ***console***.log(***regex21***.test("abb")); // T  // \* 22. ? :0个或一个 var ***regex22*** = /ab?/ ; ***console***.log(22); ***console***.log(***regex22***.test("a")); // T ***console***.log(***regex22***.test("ac")); // T ***console***.log(***regex22***.test("ab")); // T ***console***.log(***regex22***.test("abb")); // T  // \* 23. ^ :以\*\*\*开头，他和[]里的^不一样 var ***regex23*** = /^a/; // 匹配以 a为开头的字符串 ***console***.log(23); ***console***.log(***regex23***.test("apple")); // T ***console***.log(***regex23***.test("a shoe")); // T ***console***.log(***regex23***.test("two shoes")); // F  // \* 24. $ :以\*\*\*结尾 var ***regex24*** = /a$/ ; // 匹配以 a为结尾的字符串 ***console***.log(24); ***console***.log(***regex24***.test("banana")); // T ***console***.log(***regex24***.test("a banana")); // T ***console***.log(***regex24***.test("two bananas")); // F  // \* 25. 查找.和/的话需要\进行转义 var ***regex25*** = /[0**-**9]\.[0**-**9]/ ; ***console***.log(25); ***console***.log(***regex25***.test("12")); // F ***console***.log(***regex25***.test("1.2")); // T |