22、Bootstrap 中按钮组件颜色？（必会）

```
<div><a href="" class="btn btn-primary">编辑</a></div>
<div><a href="" class="btn btn-info">编辑</a></div>
<div><a href="" class="btn btn-success">编辑</a></div>
<div><a href="" class="btn btn-warning">编辑</a></div>
<div><a href="" class="btn btn-danger">编辑</a></div>
<div><a href="" class="btn btn-inverse">编辑</a></div>
<div><a href="" class="btn btn-default">编辑</a></div>
```

编辑
编辑
编辑
编辑
编辑
编辑
编辑

23、Bootstrap 常用组件有哪些？（了解）

常用组件

下拉菜单、按钮组、按钮式下拉菜单、输入框组、导航、导航条、路径导航、分页、标签、徽章、巨幕、页头、缩略图、警告框、进度条、媒体对象、列表组面板、具有响应式特性的嵌入内容、Well

JavaScript 基础

1、JavaScript 的基本类型有哪些？引用类型有哪些？null 和 undefined 的区别？（必会）

数据类型

基本数据类型：number、string、boolean、null、undefined

引用数据类型：function、object、Array

区别

undefined:表示变量声明但未初始化时的值

null：表示准备用来保存对象，还没有真正保存对象的值。从逻辑角度看，null 值表示一个空对象指针

JavaScript（ECMAScript 标准）里共有 5 种基本类型：Undefined, Null, Boolean, Number, String，和一种复杂类型 Object。可以看到 null 和 undefined 分属不同的类型，未初始化定义的值用 typeof 检测出来是"undefined"(字符串)，而 null 值用 typeof 检测出来是"object"（字符串）。任何时候都不建议显式的设置一个变量为 undefined，但是如果保存对象的变量还没有真正保存对象，应该设置成 null。实际上，undefined 值是派生自 null 值的，ECMAScript 标准规定对二者进行相等性测试要返回 true

2、如何判断 JavaScript 的数据类型？（必会）

JavaScript 数据类型一共有 7 种：

Undefined、Null、Boolean、String、Symbol、Number、Object

除了 Object 之外的 6 种属于原始数据类型。有时，我们还会细分 Object 的类型，比如 Array， Function，Date，RegExp 等

判断方法

typeof

typeof 可以用来区分除了 Null 类型以外的原始数据类型，对象类型的可以从普通对象里面识别出函数：

```
typeof undefined // "undefined"
typeof null // "object"
typeof 1 // "number"

typeof "1" // "string"
typeof Symbol() // "symbol"
typeof function() {} // "function"
typeof {} // "object"
```

问题一：typeof 不能识别 null，如何识别 null？

答案：如果想要判断是否为 null，可以直接使用===全等运算符来判断（或者使用下面的 Object.prototype.toString 方法）：

```
let a = null
a === null // true
```

问题二：typeof 作用于未定义的变量，会报错吗？

答案：不会报错，返回"undefined"。

```
typeof randomVariable // "undefined"
```

问题三：typeof Number(1)的返回值是什么？

答案："number"。注意 Number 和 String 作为普通函数调用的时候，是把参数转化为相应的原始数据类型，也就是类似于做一个强制类型转换的操作，而不是默认当做构造函数调用。注意和 Array 区分，Array(...)等价于 new Array(...)

```
typeof Number(1) // "number"
typeof String("1") // "string"
Array(1, 2, 3)
// 等价于
new Array(1, 2, 3)
```

问题四：typeof new Number(1)的返回值是什么？

答案："object"。

```
typeof new Number(1) // "object"
typeof new String(1) // "object"
```

instanceof

instanceof 不能用于判断原始数据类型的数据：

```
3 instanceof Number // false
'3' instanceof String // false
true instanceof Boolean // false
```

instanceof 可以用来判断对象的类型：

```
var date = new Date()
date instanceof Date // true
var number = new Number()
number instanceof Number // true
var string = new String()
string instanceof String // true
```

需要注意的是，instanceof 的结果并不一定是可靠的，因为在 ECMAScript7 规范中可以通过自定义 Symbol.hasInstance 方法来覆盖默认行为。

Object.prototype.toString

```
Object.prototype.toString.call(undefined).slice(8, -1) // "Undefined"
Object.prototype.toString.call(null).slice(8, -1) // "Null"
Object.prototype.toString.call(3).slice(8, -1) // "Number"
Object.prototype.toString.call(new Number(3)).slice(8, -1) // "Number"
Object.prototype.toString.call(true).slice(8, -1) // "Boolean"
Object.prototype.toString.call('3').slice(8, -1) // "String"
Object.prototype.toString.call(Symbol()).slice(8, -1) // "Symbol"
```

由上面的示例可知，该方法没有办法区分数字类型和数字对象类型，同理还有字符串类型和字符串对象类型、布尔类型和布尔对象类型

另外，ECMAScript7 规范定义了符号 Symbol.toStringTag，你可以通过这个符号自定义 Object.prototype.toString 方法的行为：

```
'use strict'
var number = new Number(3)
number[Symbol.toStringTag] = 'Custom'
Object.prototype.toString.call(number).slice(8, -1) // "Custom"
function a () {}
a[Symbol.toStringTag] = 'Custom'
Object.prototype.toString.call(a).slice(8, -1) // "Custom"
var array = []
array[Symbol.toStringTag] = 'Custom'
Object.prototype.toString.call(array).slice(8, -1) // "Custom"
```

因为 Object.prototype.toString 方法可以通过 Symbol.toStringTag 属性来覆盖默认行为，所以使用这个方法来判断数据类型也不一定是可靠的

Array.isArray

Array.isArray(value)可以用来判断 value 是否是数组：

```
Array.isArray([]) // true
Array.isArray({}) // false
(function () {console.log(Array.isArray(arguments))}()) // false
```

2、创建对象有几种方法（必会）

创建方法

1、字面量对象 // 默认这个对象的原型链指向 object

var o1 = {name: '01'};

2、通过 new Object 声明一个对象

var o11 = new Object({name: '011'});

3、使用显式的构造函数创建对象

var M = function(){this.name='o2'};

var o2 = new M();

o2.__proto__=== M.prototype

o2 的构造函数是 M

o2 这个普通函数，是 M 这个构造函数的实例

4、object.create()

var P = {name:'o3'};

var o3 = Object.create(P);

4、简述创建函数的几种方式？（必会）

第一种（函数声明）：

```
function sum1(num1,num2){
return num1+num2;
}
```

第二种（函数表达式）

```
var sum2 = function(num1,num2){
return num1+num2;
}
```

第三种（函数对象方式）

```
var sum3 = new Function("num1","num2","return num1+num2");
```

5、Javascript 创建对象的几种方式？（必会）

1、简单对象的创建 使用对象字面量的方式{}

创建一个对象（最简单，好理解，推荐使用）

代码如下

```
var Cat = {};//JSON
Cat.name="kity";//添加属性并赋值
Cat.age=2;
Cat.sayHello=function(){
alert("hello "+Cat.name+",今年"+Cat["age"]+"岁了");//可以使用 “.” 的方式访问属性，
```

```
也可以使用 HashMap 的方式访问
}
Cat.sayHello();//调用对象的（方法）函数
```

2、用 function(函数)来模拟 class

2.1） 创建一个对象，相当于 new 一个类的实例(无参构造函数)

代码如下

```
function Person(){
}
var personOne=new Person();//定义一个 function，如果有 new 关键字去"实例化",那
么该 function 可以看作是一个类
personOne.name="dylan";
personOne.hobby="coding";
personOne.work=function(){
alert(personOne.name+" is coding now...");
}
personOne.work();
```

2.2）可以使用有参构造函数来实现，这样定义更方便，扩展性更强（推荐使用）

代码如下

```
function Pet(name,age,hobby){
this.name=name;//this 作用域：当前对象
this.age=age;
this.hobby=hobby;
this.eat=function(){
alert("我叫"+this.name+",我喜欢"+this.hobby+",也是个吃货");
}
}
var maidou =new Pet("麦兜",5,"睡觉");//实例化/创建对象
maidou.eat();//调用 eat 方法(函数)
```

3、使用工厂方式来创建（Object 关键字）

代码如下：

```
var wcDog =new Object();
wcDog.name="旺财";
wcDog.age=3;
wcDog.work=function(){
alert("我是"+wcDog.name+",汪汪汪.....");
}
wcDog.work();
```

4、使用原型对象的方式 prototype 关键字

代码如下：

```
function Dog(){
}
Dog.prototype.name="旺财";
```

```
Dog.prototype.eat=function(){
alert(this.name+"是个吃货");
}
var wangcai =new Dog();
wangcai.eat();
```

5、混合模式(原型和构造函数)

代码如下：

```
function Car(name,price){
this.name=name;
this.price=price;
}
Car.prototype.sell=function(){
alert("我是"+this.name+"，我现在卖"+this.price+"万元");
}
var camry =new Car("凯美瑞",27);
camry.sell();
```

6、动态原型的方式(可以看作是混合模式的一种特例)

代码如下：

```
function Car(name,price){
this.name=name;
this.price=price;
if(typeof Car.sell=="undefined"){
Car.prototype.sell=function(){
alert("我是"+this.name+"，我现在卖"+this.price+"万元");
}
Car.sell=true;
}
}
var camry =new Car("凯美瑞",27);
camry.sell();
```

以上几种，是 javascript 中最常用的创建对象的方式

6、请指出 JavaScript 宿主对象和原生对象的区别？（必会）

原生对象

ECMA-262 把本地对象（native object）定义为 “独立于宿主环境的 ECMAScript 实现提供的对象”

“本地对象”包含哪些内容：Object、Function、Array、String、Boolean、Number、Date、RegExp、Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError

由此可以看出，本地对象就是 ECMA-262 定义的类（引用类型）

内置对象

ECMA-262 把内置对象（built-in object）定义为“由 ECMAScript 实现提供的、独立 于宿主环境的所有对象，在 ECMAScript 程序开始执行时出现”。这意味着开发者不必明 确实例化内置对象，它已被实例化了

同样是“独立于宿主环境”。根据定义我们似乎很难分清“内置对象”与“本地对象”的区别。而 ECMA-262 只定义了两个内置对象，即 Global 和 Math （它们也是本地对象，根据定义，每个内置对象都是本地对象）。如此就可以理解了。内置对象是本地对象的一种

宿主对象

何为“宿主对象”？主要在这个“宿主”的概念上，ECMAScript 中的“宿主”当然就是我们网页的运行环境，即“操作系统”和“浏览器”

所有非本地对象都是宿主对象（host object），即由 ECMAScript 实现的宿主环境提供的对象。所有的 BOM 和 DOM 都是宿主对象。因为其对于不同的“宿主”环境所展示的内容不同。其实说白了就是，ECMAScript 官方未定义的对象都属于宿主对象，因为其未定义的对象大多数是自己通过 ECMAScript 程序创建的对象

7、JavaScript 内置的常用对象有哪些？并列举该对象常用的方法？（必会）

对象及方法

Arguments 函数参数集合

Arguments[ ] 函数参数的数组

Arguments 一个函数的参数和其他属性

Arguments.callee 当前正在运行的函数

Arguments.length 传递给函数的参数的个数

Array 数组

length 属性 动态获取数组长度

join() 将一个数组转成字符串。返回一个字符串

reverse() 将数组中各元素颠倒顺序

delete 运算符 只能删除数组元素的值，而所占空间还在，总长度没变(arr.length)

shift() 删除数组中第一个元素，返回删除的那个值，并将长度减 1

pop() 删除数组中最后一个元素，返回删除的那个值，并将长度减 1

unshift() 往数组前面添加一个或多个数组元素，长度要改变。arrObj.unshift(“a”，“b，“c”)

push() 往数组结尾添加一个或多个数组元素，长度要改变。arrObj.push(“a”，“b”，“c”)

concat( ) 连接数组
slice( ) 返回数组的一部分
sort( ) 对数组元素进行排序
splice( ) 插入、删除或替换数组的元素
toLocaleString( ) 把数组转换成局部字符串
toString( ) 将数组转换成一个字符串
forEach 遍历所有元素

```
var arr = [1, 2, 3];
arr.forEach(function(item, index) {
    //    遍历数组的所有元素
    console.log(index, item);
});
```

every 判断所有元素是否都符合条件

```
    var arr = [1, 2, 3];
    var arr1 = arr.every(function(item, index) {
        if (item < 4) {
            return true;
        }
    })
    console.log(arr1); // true
```

sort 排序

```
    var arr = [1, 5, 2, 7, 3, 4];
    var arr2 = arr.sort(function(a, b) {
        // 从小到大
        return a-b;
        // 从大到小
        return b-a;
    })
    console.log(arr2); // 1,2,3,4,5,7
```

map 对元素重新组装，生成新数组

```
var arr = [1, 5, 2, 7, 3, 4];
var arr2 = arr.map(function(item, index) {
    return '<b>' + item + '</br>';
})
console.log(arr2);
```

filter 过滤符合条件的元素

```
var arr = [1, 2, 3, 4];
var arr2 = arr.filter(function(item, index) {
    if (item>2) {
        return true;
    }
})
```

console.log(arr2); // [3, 4]

Boolean 布尔对象

Boolean.toString( ) 将布尔值转换成字符串

Boolean.valueOf( ) Boolean 对象的布尔值

Date 日期时间

创建 Date 对象的方法

（1）创建当前(现在)日期对象的实例，不带任何参数

var today = new Date()；

（2）创建指定时间戳的日期对象实例，参数是时间戳。

时间戳：是指某一个时间距离 1970 年 1 月 1 日 0 时 0 分 0 秒，过去了多少毫秒值(1 秒
=1000 毫秒)

var timer = new Date(10000)； //时间是 1970 年 1 月 1 日 0 时 0 分 10 秒

（3）指定一个字符串的日期时间信息，参数是一个日期时间字符串

var timer = new Date(“2015/5/25 10：00：00”)；

（4）指定多个数值参数

var timer = new Date(2015+100，4，25，10，20，0)； //顺序为：年、月、日、时、分、秒，年、月、日是必须的

方法：

Date.getDate( ) 返回一个月中的某一天

Date.getDay( ) 返回一周中的某一天

Date.getFullYear( ) 返回 Date 对象的年份字段

Date.getHours( ) 返回 Date 对象的小时字段

Date.getMilliseconds( ) 返回 Date 对象的毫秒字段

Date.getMinutes( ) 返回 Date 对象的分钟字段

Date.getMonth( ) 返回 Date 对象的月份字段

Date.getSeconds( ) 返回 Date 对象的秒字段

Date.getTime( ) 返回 Date 对象的毫秒表示

Date.getTimezoneOffset( ) 判断与 GMT 的时间差

Date.getUTCDate( ) 返回该天是一个月的哪一天(世界时)

Date.getUTCDay( ) 返回该天是星期几(世界时)

Date.getUTCFullYear( ) 返回年份(世界时)

Date.getUTCHours( ) 返回 Date 对象的小时字段(世界时)

Date.getUTCMilliseconds( ) 返回 Date 对象的毫秒字段(世界时)

Date.getUTCMinutes( ) 返回 Date 对象的分钟字段(世界时)

Date.getUTCMonth( ) 返回 Date 对象的月份(世界时)

Date.getUTCSeconds( ) 返回 Date 对象的秒字段(世界时)

Date.getYear( ) 返回 Date 对象的年份字段(世界时)

Date.parse( ) 解析日期/时间字符串

Date.setDate( ) 设置一个月的某一天

Date.setFullYear( ) 设置年份，也可以设置月份和天

Date.setHours( ) 设置 Date 对象的小时字段、分钟字段、秒字段和毫秒字段

Date.setMilliseconds( ) 设置 Date 对象的毫秒字段
Date.setMinutes( ) 设置 Date 对象的分钟字段和秒字段
Date.setMonth( ) 设置 Date 对象的月份字段和天字段
Date.setSeconds( ) 设置 Date 对象的秒字段和毫秒字段
Date.setTime( ) 以毫秒设置 Date 对象
Date.setUTCDate( ) 设置一个月中的某一天(世界时)
Date.setUTCFullYear( ) 设置年份、月份和天(世界时)
Date.setUTCHours( ) 设置 Date 对象的小时字段、分钟字段、秒字段和毫秒字段(世界时)
Date.setUTCMilliseconds( ) 设置 Date 对象的毫秒字段(世界时)
Date.setUTCMinutes( ) 设置 Date 对象的分钟字段和秒字段(世界时)
Date.setUTCMonth( ) 设置 Date 对象的月份字段和天数字段(世界时)
Date.setUTCSeconds( ) 设置 Date 对象的秒字段和毫秒字段(世界时)
Date.setYear( ) 设置 Date 对象的年份字段
Date.toDateString( ) 返回 Date 对象日期部分作为字符串
Date.toGMTString( ) 将 Date 转换为世界时字符串
Date.toLocaleDateString( ) 回 Date 对象的日期部分作为本地已格式化的字符串
Date.toLocaleString( ) 将 Date 转换为本地已格式化的字符串
Date.toLocaleTimeString( ) 返回 Date 对象的时间部分作为本地已格式化的字符串
Date.toString( ) 将 Date 转换为字符串
Date.toTimeString( ) 返回 Date 对象日期部分作为字符串
Date.toUTCString( ) 将 Date 转换为字符串(世界时)
Date.UTC( ) 将 Date 规范转换成毫秒数
Date.valueOf( ) 将 Date 转换成毫秒表示
Error 异常对象
Error.message 可以读取的错误消息
Error.name 错误的类型
Error.toString( ) 把 Error 对象转换成字符串
EvalError 在不正确使用 eval()时抛出
SyntaxError 抛出该错误用来通知语法错误
RangeError 在数字超出合法范围时抛出
ReferenceError 在读取不存在的变量时抛出
TypeError 当一个值的类型错误时，抛出该异常
URIError 由 URI 的编码和解码方法抛出
Function 函数构造器
Function.apply( ) 将函数作为一个对象的方法调用
Function.arguments[] 传递给函数的参数
Function.call( ) 将函数作为对象的方法调用
Function.caller 调用当前函数的函数
Function.length 已声明的参数的个数
Function.prototype 对象类的原型

Function.toString( ) 把函数转换成字符串
Math 数学对象
Math 对象是一个静态对象
Math.PI 圆周率
Math.abs() 绝对值
Math.ceil() 向上取整(整数加 1，小数去掉)
Math.floor() 向下取整(直接去掉小数)
Math.round() 四舍五入
Math.pow(x，y) 求 x 的 y 次方
Math.sqrt() 求平方根
Number 数值对象
Number.MAX_VALUE 最大数值
Number.MIN_VALUE 最小数值
Number.NaN 特殊的非数字值
Number.NEGATIVE_INFINITY 负无穷大
Number.POSITIVE_INFINITY 正无穷大
Number.toExponential( ) 用指数计数法格式化数字
Number.toFixed( ) 采用定点计数法格式化数字
Number.toLocaleString( ) 把数字转换成本地格式的字符串
Number.toPrecision( ) 格式化数字的有效位
Number.toString( ) 将一个数字转换成字符串
Number.valueOf( ) 返回原始数值
Object 基础对象
Object 含有所有 JavaScript 对象的特性的超类
Object.constructor 对象的构造函数
Object.hasOwnProperty( ) 检查属性是否被继承
Object.isPrototypeOf( ) 一个对象是否是另一个对象的原型
Object.propertyIsEnumerable( ) 是否可以通过 for/in 循环看到属性
Object.toLocaleString( ) 返回对象的本地字符串表示
Object.toString( ) 定义一个对象的字符串表示
Object.valueOf( ) 指定对象的原始值
RegExp 正则表达式对象
RegExp.exec( ) 通用的匹配模式
RegExp.global 正则表达式是否全局匹配
RegExp.ignoreCase 正则表达式是否区分大小写
RegExp.lastIndex 下次匹配的起始位置
RegExp.source 正则表达式的文本
RegExp.test( ) 检测一个字符串是否匹配某个模式
RegExp.toString( ) 把正则表达式转换成字符串
String 字符串对象
Length 获取字符串的长度。如：var len = strObj.length
toLowerCase() 将字符串中的字母转成全小写。如：strObj.toLowerCase()

toUpperCase() 将字符串中的字母转成全大写。如：strObj.toUpperCase()
charAt(index) 返回指定下标位置的一个字符。如果没有找到，则返回空字符串
substr() 在原始字符串，返回一个子字符串
substring() 在原始字符串，返回一个子字符串
区别：'''
“abcdefgh” .substring(2，3) = “c”
“abcdefgh” .substr(2，3) = “cde”
'''
split() 将一个字符串转成数组
charCodeAt( ) 返回字符串中的第 n 个字符的代码
concat( ) 连接字符串
fromCharCode( ) 从字符编码创建一个字符串
indexOf( ) 返回一个子字符串在原始字符串中的索引值(查找顺序从左往右查找)。如果没有找到，则返回-1
lastIndexOf( ) 从后向前检索一个字符串
localeCompare( ) 用本地特定的顺序来比较两个字符串
match( ) 找到一个或多个正则表达式的匹配
replace( ) 替换一个与正则表达式匹配的子串
search( ) 检索与正则表达式相匹配的子串
slice( ) 抽取一个子串
toLocaleLowerCase( ) 把字符串转换小写
toLocaleUpperCase( ) 将字符串转换成大写
toLowerCase( ) 将字符串转换成小写
toString( ) 返回字符串
toUpperCase( ) 将字符串转换成大写
valueOf( )
Function

8、=== 和 ==的区别？（必会）

区别

===：三个等号我们称为等同符，当等号两边的值为相同类型的时候，直接比较等号两边的值，值相同则返回 true，若等号两边的值类型不同时直接返回 false。也就是说三个等号既要判断值也要判断类型是否相等

==：两个等号我们称为等值符，当等号两边的值为相同类型时比较值是否相同，类型不同时会发生类型的自动转换，转换为相同的类型后再作比较。也就是说两个等号只要值相等就可以

9、null，undefined 的区别（必会）

区别

null 表示一个对象被定义了，值为 “空值” ；

undefined 表示不存在这个值

typeof undefined //"undefined"

undefined :是一个表示"无"的原始值或者说表示"缺少值"，就是此处应该有一个值，但还没有定义。当尝试读取时会返回 undefined；

例如变量被声明了，但没有赋值时，就等于 undefined

typeof null //"object"

null：是一个对象(空对象，没有任何属性和方法)；

例如作为函数的参数，表示该函数的参数不是对象；

注意：

在验证 null 时，一定要使用=== ，因为 == 无法分别 null 和 undefined

undefined 表示"缺少值"，就是此处应该有一个值，但是还没有定义。典型用法是：

1、变量被声明了，但没有赋值时，就等于 undefined

2)、调用函数时，应该提供的参数没有提供，该参数等于 undefined

3、对象没有赋值的属性，该属性的值为 undefined

4、函数没有返回值时，默认返回 undefined

null 表示"没有对象"，即该处不应该有值。典型用法是：

4.1） 作为函数的参数，表示该函数的参数不是对象

4.2） 作为对象原型链的终点

10、JavaScript 中什么情况下会返回 undefined 值？（必会）

1、访问声明，但是没有初始化的变量

```
var aaa;
console.log(aaa); // undefined
```

2、访问不存在的属性

```
var aaa = {};
console.log(aaa.c);
```

3、访问函数的参数没有被显式的传递值

```
(function (b){
    console.log(b); // undefined
})();
```

4、访问任何被设置为 undefined 值的变量

```
var aaa = undefined;console.log(aaa); // undefined
```

5、没有定义 return 的函数隐式返回

```
function aaa(){}console.log(aaa()); // undefined
```

6、函数 return 没有显式的返回任何内容

```
function aaa(){
    return;
}console.log(aaa()); // undefined
```

11、如何区分数组和对象？（必会）

方法一：通过 ES6 中的 Array.isArray 来识别

```
Array.isArray([])   //true
Array.isArray({})   //false
```

方法二：通过 instanceof 来识别

```
[] instanceof Array     //true
{} instanceof Array     //false
```

方法三：通过调用 constructor 来识别

```
{}.constructor      //返回 object
[].constructor      //返回 Array
```

方法四：通过 Object.prototype.toString.call 方法来识别

```
Object.prototype.toString.call([])     //["object Array"]
Object.prototype.toString.call({})     //["object Object"]
```

12、怎么判断两个对象相等？（必会）

ES6 中有一个方法判断两个对象是否相等，这个方法判断是两个对象引用地址是否一致

```
let obj1= {
    a: 1
  }
  let obj2 = {
    a: 1
  }
console.log(Object.is(obj1, obj2)) // false
let obj3 = obj1
console.log(Object.is(obj1, obj3)) // true
console.log(Object.is(obj2, obj3)) // false
```

当需求是比较两个对象内容是否一致时就没用了

想要比较两个对象内容是否一致，思路是要遍历对象的所有键名和键值是否都一致：

1、判断两个对象是否指向同一内存

2、使用 Object.getOwnPropertyNames 获取对象所有键名数组

3、判断两个对象的键名数组是否相等

4、遍历键名，判断键值是否都相等

```
let obj1 = {
    a: 1,
    b: {
        c: 2
    }
}
let obj2 = {
    b: {
        c: 3
    },
    a: 1
}
function isObjectValueEqual(a, b) {
    // 判断两个对象是否指向同一内存，指向同一内存返回 true
    if (a === b) return true
    // 获取两个对象键值数组
    let aProps = Object.getOwnPropertyNames(a)
    let bProps = Object.getOwnPropertyNames(b)
    // 判断两个对象键值数组长度是否一致，不一致返回 false
    if (aProps.length !== bProps.length) return false
    // 遍历对象的键值
    for (let prop in a) {
        // 判断 a 的键值，在 b 中是否存在，不存在，返回 false
        if (b.hasOwnProperty(prop)) {
            // 判断 a 的键值是否为对象，是则递归，不是对象直接判断键值是否相等，不相等返回 false
            if (typeof a[prop] === 'object') {
                if (!isObjectValueEqual(a[prop], b[prop])) return false
            } else if (a[prop] !== b[prop]) {
                return false
            }
        } else {
            return false
        }
    }
    return true
}
console.log(isObjectValueEqual(obj1, obj2))    // false
```

13、列举三种强制类型转换和两种隐式类型转换？（必会）

强制

转化成字符串 toString() String()

转换成数字 Number()、 parseInt()、 parseFloat()

转换成布尔类型 Boolean()

隐式

拼接字符串

例子 var str = "" + 18

- / % === ==

14、JavaScript 中怎么获取当前日期的月份？（必会）

方法

JavaScript 中获得当前日期是使用 new Date 这个内置对象的实例，其他一些进阶的操作也是基于这个内置对象的实例。

获取完整的日期（默认格式）：

var date = new Date; // Sat Jul 06 2019 19:59:27 GMT+0800 (中国标准时间)

获取当前年份：

var year = date.getFullYear(); // 2019

获取当前月份：

var month = date.getMonth() + 1; // 7

获取当前日：

var day = date.getDay(); // 6

获取当前日期（年-月-日）：

month = (month > 9) ? month : ("0" + month);

day = (day < 10) ? ("0" + day) : day;var today = year + "-" + month + "-" + day; // 2019-07-06

另外的一些操作：

date.getYear(); // 获取当前年份(2 位)

date.getFullYear(); // 获取完整的年份(4 位, 1970-????)

date.getMonth(); // 获取当前月份(0-11,0 代表 1 月)

date.getDate(); // 获取当前日(1-31)

date.getDay(); // 获取当前星期 X(0-6,0 代表星期天)

date.getTime(); // 获取当前时间(从 1970.1.1 开始的毫秒数)

date.getHours(); // 获取当前小时数(0-23)

date.getMinutes(); // 获取当前分钟数(0-59)

date.getSeconds(); // 获取当前秒数(0-59)

```
date.getMilliseconds(); // 获取当前毫秒数(0-999)
date.toLocaleDateString(); // 获取当前日期
date.toLocaleTimeString(); // 获取当前时间
date.toLocaleString( ); // 获取日期与时间
```

15、什么是类数组（伪数组），如何将其转化为真实的数组？（必会）

伪数组

1、具有 length 属性

2、按索引方式存储数据

3、不具有数组的 push.pop 等方法

伪数组（类数组）：无法直接调用数组方法或期望 length 属性有什么特殊的行为，不具有数组的 push.pop 等方法，但仍可以对真正数据遍历方法来遍历它们。典型的是函数 document.childnodes 之类的，它们返回的 nodeList 对象都属于伪数组

伪数组-->真实数组

1.使用 Arrray.from()--ES6

2.[].slice.call(eleArr) 或则 Array.prototype.slice.call(eleArr)

示例：

```
let eleArr = document.querySelectorAll('li');
  Array.from(eleArr).forEach(function(item){
  alert(item);
});

let eleArr = document.querySelectorAll('li');
  [].slice.call(eleArr).forEach(function(item){
  alert(item);
});
```

16、如何遍历对象的属性？（必会）

1、遍历自身可枚举的属性（可枚举，非继承属性) Object.keys() 方法

该方法会返回一个由一个给定对象的自身可枚举属性组成的数组，数组中的属性名的排列顺序和使用 for..in 遍历该对象时返回的顺序一致（两者的区别是 for ..in 还会枚举其原型链上的属性 ）

```
/**Array 对象**/
var arr = ['a','b','c'];
console.log(Object.keys(arr));
// ['0','1','2']
```

```
/**Object 对象**/
var obj = {foo:'bar',baz:42};
console.log(Object.keys(obj));
// ["foo","baz"]
/**类数组 对象  随机 key 排序**/
var anObj ={100:'a',2:'b',7:'c'};
console.log(Object.keys);
//['2','7','100']
/***getFoo 是一个不可枚举的属性**/
var my_obj = Object.create(
    {},
        { getFoo : { value : function () { return this.foo } } }
    );
my_obj.foo = 1;
console.log(Object.keys(my_obj));
// ['foo']
```

2、遍历自身的所有属性(可枚举，不可枚举，非继承属性) Object.getOwnPropertyNames() 方法，该方法返回一个由指定对象的所有自身属性组成的数组(包括不可枚举属性但不包括 Symbol 值作为名称的属性)

```
var arr = ["a", "b", "c"];
console.log(Object.getOwnPropertyNames(arr).sort()); // ["0", "1", "2", "length"]
// 类数组对象
var obj = { 0: "a", 1: "b", 2: "c"};
console.log(Object.getOwnPropertyNames(obj).sort()); // ["0", "1", "2"]
// 使用 Array.forEach 输出属性名和属性值
Object.getOwnPropertyNames(obj).forEach(function(val, idx, array) {
  console.log(val + " -> " + obj[val]);
});
// 输出
// 0 -> a
// 1 -> b
// 2 -> c
//不可枚举属性
var my_obj = Object.create({}, {
  getFoo: {
    value: function() { return this.foo; },
    enumerable: false
  }
});
my_obj.foo = 1;
console.log(Object.getOwnPropertyNames(my_obj).sort()); // ["foo", "getFoo"]
```

3、遍历可枚举的自身属性和继承属性 （可枚举，可继承的属性） for in

遍历对象的属性

```
var obj={
    name：'张三',
    age : '24',
    getAge:function(){
        console.log(this.age);
    }
}
var arry ={};
for(var i in obj){
    if(obj.hasOwnProperty(i)&& typeOf obj[i] != 'function'){
        arry[i] = obj[i];
    }
}
console.log(arry);
{name:'张三',age:24}
```

注: hasOwnProperty()方法判断对象是有某个属性(本身的属性，不是继承的属性)

4、遍历所有的自身属性和继承属性

```
(function () {
    var getAllPropertyNames = function (obj) {
        var props = [];
        do {
            props = props.concat(Object.getOwnPropertyNames(obj));
        } while (obj = Object.getPrototypeOf(obj));
        return props;
    }
    var propertys = getAllPropertyNames(window);
    alert(propertys.length);              //276
    alert(propertys.join("\n"));          //toString 等
})()
```

17、src 和 href 的区别是？（必会）

区别

src（source）指向外部资源的位置，指向的内容将会嵌入到文档中当前标签所在位置；在请求 src 资源时会将其指向的资源下载并应用到文档中，如 JavaScript 脚本，img 图片和 iframe 等元素

当浏览器解析到该元素时，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，类似于将所指向资源嵌入当前标签内

href（hypertext reference/超文本引用）指向网络资源所在位置，建立和当前元素（锚点）

或当前文档（链接）之间的链接，如果我们在文档中添加<link href="common.css" rel="stylesheet"/>那么浏览器会识别该文档为 CSS 文件，就会并行下载资源并且不会停止对当前文档的处理

18、如何使用原生 JavaScript 给一个按钮绑定两个 onclick 事件？（必会）

```
Var   btn=document.getElementById( 'btn' );
//事件监听 绑定多个事件
 var btn4 = document.getElementById("btn4");
 btn4.addEventListener("click",hello1);
 btn4.addEventListener("click",hello2);
 function hello1(){
    alert("hello 1");
 }
function hello2(){
   alert("hello 2");
 }
```

19、如何在 JavaScript 中比较两个对象？（必会）

比较方法

使用 == 或 === 对两个不同却具有相同属性及属性值的对象进行比较，他们的结果却不会相等。这是因为等号比较的是他们的引用（内存地址），而不是基本类型

为了测试两个对象在结构上是否相等，需要一个辅助函数。 他将遍历每个对象的所有属性，然后测试他们是否具有相同的值，嵌套对象也需如此。当然，也可以使用参数来控制是否对原型链进行比较

注意：此代码只对普通对象、数组、函数、日期和基本类型的数据结构进行对比

```
function isDeepEqual(obj1, obj2, testPrototypes = false) {
  if (obj1 === obj2) {
    return true
  }
  if (typeof obj1 === "function" && typeof obj2 === "function") {
    return obj1.toString() === obj2.toString()
  }
  if (obj1 instanceof Date && obj2 instanceof Date) {
    return obj1.getTime() === obj2.getTime()
  }
  if (
```

```
        Object.prototype.toString.call(obj1) !==
          Object.prototype.toString.call(obj2) ||
        typeof obj1 !== "object"
      ) {
        return false
      }
      const prototypesAreEqual = testPrototypes
        ? isDeepEqual(
            Object.getPrototypeOf(obj1),
            Object.getPrototypeOf(obj2),
            true
          )
        : true
      const obj1Props = Object.getOwnPropertyNames(obj1)
      const obj2Props = Object.getOwnPropertyNames(obj2)
      return (
        obj1Props.length === obj2Props.length &&
        prototypesAreEqual &&
        obj1Props.every(prop => isDeepEqual(obj1[prop], obj2[prop]))
      )
    }
```

加分回答：

像数字和字符串这样的基本类型只需对比他们的值

当一个对象赋值给另一个新对象时，使用等号进行对比，他们就会相等。因为他们的引用（内存地址）是同一个。如：

```
    const obj1 = {
        name: "Benjamin",
        sex : "male"
    };
    const obj2 = {
        name: "Benjamin",
        sex : "male"
    };
    const obj3 = obj1;
    console.log(obj1 === obj3); // true
```

想要完整的实现对比，可参见 Lodash 中的_.isEqual 和_.isEqualWith 的实现

20、JavaScript 中的作用域、预解析与变量声明提升？（必会）

作用域

就是变量的有效范围。 在一定的空间里可以对数据进行读写操作，这个空间就是数据的作用域

1、全局作用域： 最外层函数定义的变量拥有全局作用域，即对任何内部函数来说，都是可以访问的；

2、局部作用域： 局部作用域一般只在固定的代码片段内可访问到，而对于函数外部是无法访问的，最常见的例如函数内部。在 ES6 之前，只有函数可以划分变量的作用域，所以 在函数的外面无法访问函数内的变量

3、块级作用域：凡是代码块就可以划分变量的作用域，这种作用域的规则就叫块级作用域

块级作用域 函数作用域 词法作用域之间的区别：

3.1）块级作用域和函数作用域描述的是，什么东西可以划分变量的作用域

3.2）词法作用域描述的是，变量的查找规则

之间的关系：

1、 块级作用域 包含 函数作用域

2、 词法作用域 与 块级作用域、函数作用域之间没有任何交集， 他们从两个角度描述了作用域的规则

ES6 之前 JavaScript 采用的是函数作用域+词法作用域，ES6 js 采用的是块级作用域+词法作用域

预解析

JavaScript 代码的执行是由浏览器中的 JavaScript 解析器来执行的。JavaScript 解析器执行 JavaScript 代码的时候，分为两个过程：预解析过程和代码执行过程

预解析过程：

1.把变量的声明提升到当前作用域的最前面，只会提升声明，不会提升赋值

2.把函数的声明提升到当前作用域的最前面，只会提升声明，不会提升调用

3.先提升 var，在提升 function

JavaScript 的执行过程：

```
1
2
3
4
5
6
7
8
9
```

```
// 案例 1
var a = 25;
function abc() {
  alert(a);
  var a = 10;
}
abc();
```

```
// 案例 2
console.log(a);
function a() {
  console.log('aaaaa');
}
var a = 1;
console.log(a);
```

变量提升

变量提升：定义变量的时候，变量的声明会被提升到作用域的最上面，变量的赋值不会提升

函数提升：JavaScript 解析器首先会把当前作用域的函数声明提前到整个作用域的最前面

```
// 1、----------------------------------
var num = 10;
fun();
function fun() {
  console.log(num);
  var num = 20;
}
//2、----------------------------------
var a = 18;
f1();
function f1() {
  var b = 9;
  console.log(a);
  console.log(b);
  var a = '123';
}
// 3、----------------------------------
f1();
console.log(c);
console.log(b);
console.log(a);
function f1() {
  var a = b = c = 9;
  console.log(a);
  console.log(b);
  console.log(c);
}
```

变量声明提升：

使用 var 关键字定义的变量，被称为变量声明；

函数声明提升的特点是，在函数声明的前面，可以调用这个函数

21、变量提升与函数提升的区别？（必会）

变量提升

简单说就是在 JavaScript 代码执行前引擎会先进行预编译，预编译期间会将变量声明与函数声明提 升至其对应作用域的最顶端，函数内声明的变量只会提升至该函数作用域最顶层
当函数内部定义的一个变量与外部相同时，那么函数体内的这个变量就会被上升到最顶端
举例来说：

```
console.log(a); //
undefined var a = 3;
//预编译后的代码结构可以看做如下运行顺序 var a; // 将变量 a 的声明提升至最顶端，赋值逻辑不提升。 console.log(a);
// undefined a = 3; // 代码执行到原位置即执行原赋值逻辑
```

函数提升

函数提升只会提升函数声明式写法，函数表达式的写法不存在函数提升
函数提升的优先级大于变量提升的优先级，即函数提升在变量提升之上

22、什么是作用域链？（必会）

作用域链

当代码在一个环境中执行时，会创建变量对象的一个作用域链
由子级作用域返回父级作用域中寻找变量，就叫做作用域链
作用域链中的下一个变量对象来自包含环境，也叫外部环境。而再下一个变量对象则来自下一个包含环境，一直延续到全局执行环境。全局执行环境的变量对象始终都是作用域链中的最后一个对象
作用域链前端始终都是当前执行的代码所在环境的变量对象，如果环境是函数，则将其活动对象作为变量对象

23、如何延长作用域链？（必会）

作用域链是可以延长的
延长作用域链：
执行环境的类型只有两种，全局和局部（函数）。但是有些语句可以在作用域链的前端临时增加一个变量对象，该变量对象会在代码执行后被移除
具体来说就是执行这两个语句时，作用域链都会得到加强
1、try - catch 语句的 catch 块；会创建一个新的变量对象，包含的是被抛出的错误对象的声明
2、with 语句。with 语句会将指定的对象添加到作用域链中

23、判断一个值是什么类型有哪些方法？（必会）

方法

1、typeof 运算符

2、instanceof 运算符

instanceof 严格来说是 Java 中的一个双目运算符，用来测试一个对象是否为一个类的实例，用法为：

```
// 判断 foo 是否是 Foo 类的实例
function Foo(){}
var foo = new Foo();
console.log(foo instanceof Foo)    //true
```

3、Object.prototype.toString 方法

- 在 JavaScript 里使用 typeof 来判断数据类型，只能区分基本类型，即“number”，”string”，”undefined”，”boolean”，”object”，“function”，“symbol” (ES6 新增)七种
- 对于数组、null、对象来说，其关系错综复杂，使用 typeof 都会统一返回 “object”字符串
- 要想区别对象、数组、函数单纯使用 typeof 是不行的，JavaScript 中,通过 Object.prototype.toString 方法，判断某个对象值属于哪种内置类型。
- 在介绍 Object.prototype.toString 方法之前，我们先把 toString()方法和 Object.prototype.toString.call()方法进行对比
- toString()方法和 Object.prototype.toString.call()方法对比

```
var arr=[1,2];
//直接对一个数组调用 toString()
arr.toString();// "1,2"
//通过 call 指定 arr 数组为 Object.prototype 对象中的 toString 方法的上下文
Object.prototype.toString.call(arr); //"[object Array]"
```

25、JavaScript 变量按照存储方式区分为哪些类型，并描述其特点？（必会）

1、值类型和引用类型

2、值类型存储的是值 ， 赋值之后原变量的值不改变

3、引用类型存储的是地址 ， 赋值之后是把原变量的引用地址赋值给新变量 ， 新变量改变 原来的会跟着改变

26、如何实现数组的随机排序？（必会）

方法一：

```
var arr = [1,2,3,4,5,6,7,8,9,10];
function randSort1(arr){
    for(var i = 0,len = arr.length;i &lt; len; i++ ){
        var rand = parseInt(Math.random()*len);
        var temp = arr[rand];
        arr[rand] = arr[i];
        arr[i] = temp;
    }
    return arr;
}
console.log(randSort1(arr));
```

方法二：

```
var arr = [1,2,3,4,5,6,7,8,9,10];
    function randSort2(arr){
        var mixedArray = [];
        while(arr.length &gt; 0){
            var randomIndex = parseInt(Math.random()*arr.length);
            mixedArray.push(arr[randomIndex]);
            arr.splice(randomIndex, 1);
        }
     return mixedArray;
 }
console.log(randSort2(arr));
```

方法三：

```
var arr = [1,2,3,4,5,6,7,8,9,10];
    arr.sort(function(){
        return Math.random() - 0.5;
    })
    console.log(arr);
```

27、Function foo() {}和 var foo = function() {}之间 foo 的用法上的区别？（必会）

区别

1、var foo = function () {}

这种方式是声明了个变量，而这个变量是个方法，变量在 JavaScript 中是可以改变的。

2、function foo() {}

这种方式是声明了个方法，foo 这个名字无法改变

例：

```
function b(){
document.write("aa");
}var a=function(){
document.write("123");
}
b();
a();
```

好像并没有什么区别，看下边

```
b();
a();function b(){
document.write("aa");
}var a=function(){
document.write("123");
}
```

是不是有区别了

function b(){} 为函数声明，程序运行前就已存在

var a = function(){} 为函数表达式，是变量的声明，属于按顺序执行，所以 a 为 undefined

28、索引有哪几种类型，有什么区别？（了解）

索引是对数据库表中一列或多列的值进行排序的一种结构，例如 employee 表的姓（name）列。如果要按姓查找特定职员，与必须搜索表中的所有行相比，索引会帮助您更快地获得该信息

索引是一个单独的、物理的数据库结构，它是某个表中一列或若干列值的集合和相应的指向表中物理标识这些值的数据页的逻辑指针清单。索引提供指向存储在表的指定列中的数据值的指针，然后根据您指定的排序顺序对这些指针排序。数据库使用索引的方式与您使用书籍中的索引的方式很相似：它搜索索引以找到特定值，然后顺指针找到包含该值的行。在数据库关系图中，您可以在选定表的“索引/键”属性页中创建、编辑或删除每个索引类型。当保存索引所附加到的表，或保存该表所在的关系图时，索引将保存在数据库中。

可以基于数据库表中的单列或多列创建索引。多列索引使您可以区分其中一列可能有相同值的行。如果经常同时搜索两列或多列或按两列或多列排序时，索引也很有帮助。例如，如果经常在同一查询中为姓和名两列设置判据，那么在这两列上创建多列索引将很有意义。确定索引的有效性：检查查询的 WHERE 和 JOIN 子句。在任一子句中包括的每一列都是索引可以选择的对象。对新索引进行试验以检查它对运行查询性能的影响。考虑已在表上创建的索引数量。最好避免在单个表上有很多索引。检查已在表上创建的索引的定义。最好避免包含共享列的重叠索引。　检查某列中唯一数据值的数量，并将该数量与表中的行　数进行比较。比较的结果就是该列的可选择性，这有助于确定该列是否适合建立索引，如果适合，确定索引的类型

建立索引的优点：

1、大大加快数据的检索速度;

2、创建唯一性索引，保证数据库表中每一行数据的唯一性;

3、加速表和表之间的连接;

4、在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间

索引类型

根据数据库的功能，可以在数据库设计器中创建四种索引：唯一索引、非唯一索引、主键索引和聚集索引。 尽管唯一索引有助于定位信息，但为获得最佳性能结果，建议改用主键 或唯一约束

唯一索引：

唯一索引是不允许其中任何两行具有相同索引值的索引。当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在 employee 表中职员的姓 (lname) 上创建了唯一索引，则任何两个员工都不能同姓

非唯一索引：

非唯一索引是相对唯一索引，允许其中任何两行具有相同索引值的索引。当现有数据中存在重复的键值时，数据库是允许将新创建的索引与表一起保存。这时数据库不能防止添加将在表中创建重复键值的新数据

主键索引：

数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

聚集索引（也叫聚簇索引）：

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度

29、简述 Array.form 和 Array.of 的使用及区别？（了解）

一、 Array.from()：将伪数组对象或可遍历对象转换为真数组

Array.from()用法

Array.from 接受三个参数，但只有 input 是必须的：

input: 你想要转换的类似数组对象和可遍历对象

map: 类似于数组的 map 方法，用来对每个元素进行处理，将处理后的值放入返回的数组

context: 绑定 map 中用到的 this

只要是部署了 iterator 接口的数据结构，Array.from 都能将其转为数组:

```
let arr = Array.from('juejin');
console.log(arr); //["j", "u", "e", "j", "i", "n"]
```

Array.from 还可以接受第二个参数，作用类似于数组的 map 方法，用来对每个元素进行处理，处理后的值放入返回的数组

```
Array.from([1, 2, 3], (x) => x * x)// [1, 4, 9]
// 等同于
Array.from([1,2,3].map(x => x * x))
```

如果 map 函数里面用到了 this 关键字，还可以传入 Array.from 的第三个参数，用来绑定 this

Array.from()可以将各种值转为真正的数组，并且还提供 map 功能。这实际上意味着，只要有一个原始的数据结构，你就可以先对它的值进行处理，然后转成规范的数组结构，进而就可以使用数量众多的数组方法

```
Array.from({ length: 2 }, () => 'jack')// ['jack', 'jack']
```

二、Array.of(v1, v2, v3)：将一系列值转换成数组

当调用 new Array( )构造器时，根据传入参数的类型与数量的不同，实际上会导致一些不同的结果， 例如：

```
let items = new Array(2) ;
console.log(items.length) ; // 2
console.log(items[0]) ; // undefined
console.log(items[1]) ;
let items = new Array(1, 2) ;
console.log(items.length) ; // 2
console.log(items[0]) ; // 1
console.log(items[1]) ; // 2
```

当使用单个数值参数来调用 Array 构造器时，数组的长度属性会被设置为该参数。 如果使用多个参数(无论是否为数值类型)来调用，这些参数也会成为目标数组的项。数组的这种 行为既混乱又有风险，因为有时可能不会留意所传参数的类型

ES6 引入了 Array.of( )方法来解决这个问题。该方法的作用非常类似 Array 构造器，但在使用单个数值参数的时候并不会导致特殊结果。Array.of( )方法总会创建一个包含所有传入参数的数组，而不管参数的数量与类型：

```
let items = Array.of(1, 2);
console.log(items.length); // 2
console.log(items[0]); // 1
console.log(items[1]); // 2
items = Array.of(2);
console.log(items.length); // 1
console.log(items[0]); // 2
```

Array.of 基本上可以用来替代 Array()或 newArray()，并且不存在由于参数不同而导致的重载，而且他们的行为非常统一

30、根据你的理解,请简述 JavaScript 脚本的执行原理（了解）

原理

JavaScript 是一种动态、弱类型、基于原型的语言，通过浏览器可以直接执行

当浏览器遇到\<script> 标记的时候，浏览器会执行之间的 javascript 代码。嵌入的 JavaScript

代　码是顺序执行的，每个脚本定义的全局变量和函数，都可以被后面执行的脚本所调用。
变量的调用，必须是前面已经声明，否则获取的变量值是 undefined

WebAPI

1、什么是 dom？（必会）

什么是 dom

1、DOM 是 W3C（万维网联盟）的标准

2、DOM 定义了访问 HTML 和 XML 文档的标准

什么是 W3C

1、 “W3C 文档对象模型 （DOM） 是中立于平台和语言的接口，它允许程序和脚本动态地访问和更新文档的内容、结构和样式。”

2、W3C DOM 标准被分为 3 个不同的部分

2.1）核心 DOM - 针对任何结构化文档的标准模型

2.2）XML DOM - 针对 XML 文档的标准模型

2.3）HTML DOM - 针对 HTML 文档的标准模型

备注：DOM 是 Document Object Model（文档对象模型）的缩写

2、dom 是哪种基本的数据结构？（必会）

DOM 是一种树形结构的数据结构

3、dom 操作的常用 api 有哪些？（必会）

dom 操作的常用 api 有以下几种

1、获取 dom 节点

1.1）document.getElementById('div1')

1.2）document.getElementsByTagName('div')

1.3）document.getElementsByClassName('container')

1.4）document.querySelector('p')

1.5）document.querySelectorAll('p')

2、property（js 对象的 property）

```
var p = document.getElementByTagName('p')[0]
console.log(p.nodeName); // nodeName 是 p 的 property，即 nodeName 是 p 的属性
```

3、attribute

3.1）p.getAttribute('data-name');

3.2）p.setAttribute('data-name', 'imooc');