JavaScript知识点记录

写在前面：本文档主要用于记录在学习JavaScript过程中遇到的重要知识点和各类通俗易懂的案例

零、基础知识

1.强语言与弱语言

- **强类型语言**：强类型语言也称为强类型定义语言，是一种总是强制类型定义的语言，要求变量的使用要严格符合定义，所有变量都必须先定义后使用。Java和C++等语言都是强制类型定义的，也就是说，一旦一个变量被指定了某个数据类型，如果不经过强制转换，那么它就永远是这个数据类型了。例如你有一个整数，如果不显式地进行转换，你不能将其视为一个字符串。
- **弱类型语言**：弱类型语言也称为弱类型定义语言，与强类型定义相反。JavaScript语言就属于弱类型语言。简单理解就是一种变量类型可以被忽略的语言。比如JavaScript是弱类型定义的，在JavaScript中就可以将字符串'12'和整数3进行连接得到字符串'123'，在相加的时候会进行强制类型转换。

2.解释性语言与编译性语言

（1）解释型语言

使用专门的解释器对源程序逐行解释成特定平台的机器码并立即执行。是代码在执行时才被解释器一行行动态翻译和执行，而不是在执行之前就完成翻译。解释型语言不需要事先编译，其直接将源代码解释成机器码并立即执行，所以只要某一平台提供了相应的解释器即可运行该程序。其特点总结如下

- 解释型语言每次运行都需要将源代码解释称机器码并执行，效率较低；
- 只要平台提供相应的解释器，就可以运行源代码，所以可以方便源程序移植；
- JavaScript、Python等属于解释型语言。

（2）编译型语言

使用专门的编译器，针对特定的平台，将高级语言源代码一次性的编译成可被该平台硬件执行的机器码，并包装成该平台所能识别的可执行性程序的格式。在编译型语言写的程序执行之前，需要一个专门的**编译过程**，把源代码编译成机器语言的文件，如exe格式的文件，以后要再运行时，直接使用编译结果即可，如直接运行exe文件。因为只需编译一次，以后运行时不需要编译，所以编译型语言执行效率高。其特点总结如下：

- 一次性的编译成平台相关的机器语言文件，运行时脱离开发环境，运行效率高；
- 与特定平台相关，一般无法移植到其他平台；
- C、C++等属于编译型语言。

两者对比：强类型语言在速度上可能略逊色于弱类型语言，但是强类型语言带来的严谨性可以有效地帮助避免许多错误。

两者主要区别在于：前者源程序编译后即可在该平台运行，后者是在运行期间才编译。所以前者运行速度快，后者跨平台性好。

一、数据类型

1. JavaScript中的变量

JavaScript的**原始数据类型–> 栈** ，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储

1. number
2. string
3. boolean
4. null
5. undefined
6. symbol（ES6 引入了一种新的原始数据类型，表示独一无二的值，主要是为了解决全局变量冲突的问题）

引用数据类型（对象类型）：对象(Object)、数组(Array)、函数(Function)，还有两个特殊的对象：正则（RegExp）和日期（Date）。

javascript中的全部变量- 8

1. number
2. string
3. boolean
4. null
5. undefined
6. symbol（ES6）
7. Object（Array，function，object **引用数据类型 –> 堆**，占据空间大、大小不固定。如果存储在栈中，将会影响程序运行的性能；引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。）
8. Bigint (ES10，是一种数字类型的数据，它可以表示任意精度格式的整数，使用 BigInt 可以安全地存储和操作大整数，即使这个数已经超出了 Number 能够表示的安全整数范围。数据类型的目的是比`Number`数据类型支持的范围更大的整数值。

   引出原因：为什么会有**BigInt**的提案?

   JavaScript中Number.MAX_SAFE_INTEGER表示最大安全数字，计算结果是9007199254740991，即在这个数范围内不会出现精度丢失（小数除外）。但是一旦超过这个范围，js就会出现计算不准确的情况，这在大数计算的时候不得不依靠一些第三方库进行解决，因此官方提出了BigInt来解决此问题)

在操作系统中，**内存被分为栈区和堆区**：

- 栈区内存由**编译器**自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
- 堆区内存一般由**开发者**分配释放，若开发者不释放，程序结束时可能由**垃圾回收机制**回收

2. 检查变量类型的几种方式

2.1 typeof

```
console.log(typeof 2);               // number
console.log(typeof true);            // boolean
console.log(typeof 'str');           // string
console.log(typeof []);              // object
console.log(typeof function(){});    // function
console.log(typeof {});              // object
console.log(typeof undefined);       // undefined
console.log(typeof null);            // object（历史遗留问题）
console.log(typeof NaN);              //number
// 当使用双等号对null和undefined的值进行比较时会返回 true，使用三个等号时会返回 false。
```

其中数组、对象、null都会被判断为object，其他判断都正确

2.2 instanceof及其实现原理

`instanceof`可以正确判断对象的类型，其内部运行机制是**判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。**

```
function myInstanceof(left, right) {
  // 获取对象的原型
  let proto = Object.getPrototypeOf(left)
  // 获取构造函数的 prototype 对象
  let prototype = right.prototype;

  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
    if (proto === prototype) return true;
    // 如果没有找到，就继续从其原型上找，Object.getPrototypeOf方法用来获取指定对象的原型
    proto = Object.getPrototypeOf(proto);
  }
}
```

PS： prototype属性：返回对象类型原型的引用

```
function Test(){}
alert(Test.prototype); // 输出 "Object"
```

```
console.log(2 instanceof Number);                    // false
console.log(true instanceof Boolean);                // false
console.log('str' instanceof String);                // false

console.log([] instanceof Array);                    // true
console.log(function(){} instanceof Function);       // true
console.log({} instanceof Object);                   // true
```

可以看到，`instanceof`**只能正确判断引用数据类型**，而不能判断基本数据类型。`instanceof` 运算符可以用来测试一个对象在其原型链中是否存在一个构造函数的 `prototype` 属性。

2.3 constructor0

`constructor`有两个作用，一是判断数据的类型，二是对象实例通过 `constrcutor` 对象访问它的构造函数。

```
console.log((2).constructor === Number); // true
console.log((true).constructor === Boolean); // true
console.log(('str').constructor === String); // true
console.log(([]).constructor === Array); // true
console.log((function() {}).constructor === Function); // true
console.log(({}).constructor === Object); // true
```

需要注意，如果创建一个对象来改变它的原型，`constructor`就不能用来判断数据类型了

```
function Fn(){};

Fn.prototype = new Array();        //改变原型

var f = new Fn();

console.log(f.constructor===Fn);    // false
console.log(f.constructor===Array); // true
```

2.4 Object.prototype.toString.call()

使用 Object 对象的原型方法 toString 来判断数据类型

```
var a = Object.prototype.toString;

console.log(a.call(2));
console.log(a.call(true));
console.log(a.call('str'));
console.log(a.call([]));
console.log(a.call(function(){}));
console.log(a.call({}));
console.log(a.call(undefined));
console.log(a.call(null));
```

3. null和undefined的区别是什么?

null： Null类型，代表“空值”，代表一个空对象指针(null 的值是机器码 NULL 指针(null 指针的值全是 0)，使用typeof运算得到 “object”，所以可以认为它是一个特殊的对象值。场景如下：

（1） 作为函数的参数，表示该函数的参数不是对象。

（2） 作为对象原型链的终点

(3) 赋值给一些可能会返回对象的变量，作为初始化

undefined： undefined表示**缺少值**，就是此处应该有一个值，但是还没有定义（当一个声明了一个变量未初始化时）,undefined的值是 (-2)^30(一个超出整数范围的数字)，场景如下：

（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。

安全的获得undefined：

因为 undefined 是一个标识符，所以可以被当作变量来使用和赋值，但是这样会影响 undefined 的正常判断。表达式 void ___ 没有返回值，因此返回结果是 undefined。void 并不改变表达式的结果，只是让表达式不返回值。因此可以用 void 0 来获得 undefined。

4. JavaScript最大安全数字与最小安全数字?

```
console.log(Number.MAX_SAFE_INTEGER)
// 9007199254740991

console.log(Number.MIN_SAFE_INTEGER)
// -9007199254740991
```

5.为什么0.1+0.2 ! == 0.3，如何让其相等

```
let n1 = 0.1, n2 = 0.2
console.log(n1 + n2)  // 0.30000000000000004

(n1 + n2).toFixed(2) // 注意，toFixed为四舍五入
```

原因：0.1的二进制是`0.0001100110011001100...`（1100循环），0.2的二进制是：`0.00110011001100...`（1100循环），这两个数的二进制都是无限循环的数。一般我们认为数字包括整数和小数，但是在 JavaScript 中只有一种数字类型：Number，它的实现遵循IEEE 754标准，使用64位固定长度来表示，也就是标准的**double双精度浮点数**。在二进制科学表示法中，双精度浮点数的小数部分最多只能保留52位，再加上前面的1，其实就是保留53位有效数字，剩余的需要舍去，遵从“0舍1入”的原则。所以，0.1+0.2 ！ =0.3

解决办法：对于这个问题，一个直接的解决方法就是设置一个误差范围，通常称为“机器精度”。对JavaScript来说，这个值通常为2-52，在ES6中，提供了`Number.EPSILON`（ε）属性，而它的值就是2^(-52)，只要判断`0.1+0.2-0.3`是否小于`Number.EPSILON`，如果小于，就可以判断为0.1+0.2 ===0.3

```
function numberepsilon(arg1,arg2){
  return Math.abs(arg1 - arg2) < Number.EPSILON;
}

console.log(numberepsilon(0.1 + 0.2, 0.3)); // true
```

6.isNaN 和 Number.isNaN 函数的区别?

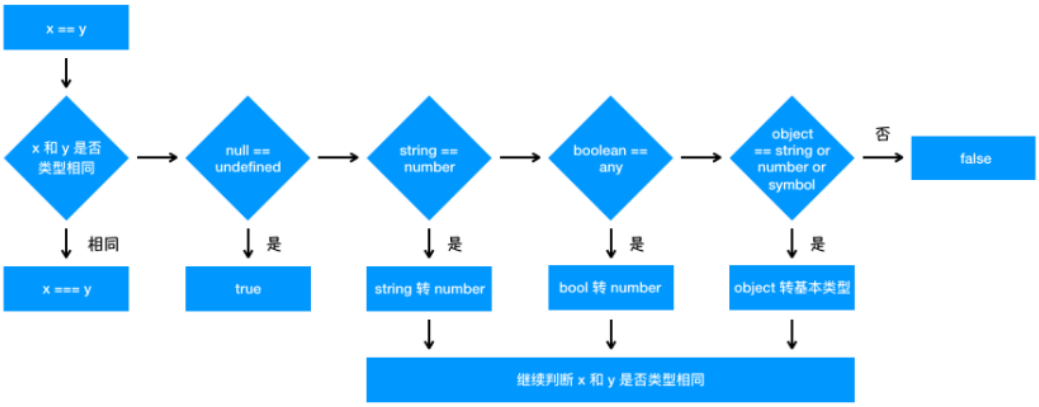
NaN - not a number NaN 是一个特殊值，它和自身不相等，是唯一一个非自反的值。而 NaN !== NaN 为 true。

- 函数 isNaN 接收参数后，会尝试将这个参数转换为数值，任何不能被转换为数值的的值都会返回 true，因此非数字值传入也会返回 true ，会影响 NaN 的判断。
- 函数 Number.isNaN 会首先判断**传入参数是否为数字**，如果是数字再继续判断是否为 NaN ，不会进行数据类型的转换，这种方法对于 NaN 的判断更为准确。

7. `==` 操作符的强制类型转换规则?

对于 `==` 来说，如果对比双方的类型**不一样**，就会进行**类型转换**。假如对比 `x` 和 `y` 是否相同，就会进行如下判断流程：

1. 首先会判断两者类型是否**相同**，相同的话就比较两者的大小；
2. 类型不相同的话，就会进行类型转换；
3. 会先判断是否在对比 `null` 和 `undefined`，是的话就会返回 `true`
4. 判断两者类型是否为 `string` 、 `number`、 `Boolean`，是的话就会将字符串转换为 `number`



8 .其他类型到字符串的转换规则

- Null 和 Undefined 类型 ， null 转换为 "null"，undefined 转换为 "undefined"，
- Boolean 类型，true 转换为 "true"，false 转换为 "false"。
- Number 类型的值直接转换，不过那些极小和极大的数字会使用指数形式。
- Symbol 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误。
- 对普通对象来说，除非自行定义 toString() 方法，否则会调用 toString()（Object.prototype.toString()）来返回内部属性 [[Class]] 的值，如"[object Object]"。如果对象有自己的 toString() 方法，字符串化时就会调用该方法并使用其返回值。

转换方法：

2. toString （） =》用于数值、布尔值和字符串

```
var found = true;
var str2 = found.toString();            //字符串 “true”

//括号中的参数表示进制，来表示输出数值的基数
var num = 10;
num.toString();   // "10"

num.toString(2);  // "1010"

num.toString(8);  // "12"

num.toString(16);  // "a"
```

3. 在不知道变量是否为null或者undefined时，使用String()函数来转换为字符串类型。

   String()函数可以将任何类型的数值转换为字符，使用规则：

   - 如果转换值有toString()方法的话，就直接调用该方法，并返回相应的结果
   - 如果转换值是null,则返回"null"
   - 如果转换值是undefined,则返回"undefined"
   - Symbol 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误

```
var value1 = 10;
 var value2 = true;
 var value3 = null;
 var value4;    //只定义未初始化的变量，自动赋值为undefined

  String(value1);   // 10"
  String(value2);   //"true"
  String(value3);   // "null"
  String(value4);   // "undefined"
```

3. 利用+“”， 把转换的值与一个字符串 “” 加在一起

9.其他类型到数字型的转化规则

- Undefined 类型的值转换为 NaN。
- Null 类型的值转换为 0。
- Boolean 类型的值，true 转换为 1，false 转换为 0。
- String 类型的值转换如同使用 Number() 函数进行转换，如果包含非数字值则转换为 NaN，空字符串为 0。（见后续示例）
- Symbol 类型的值不能转换为数字，会报错。
- 对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字

9.1 转换函数

parseInt()：转换为整数； parseFloat()：转换为浮点数（只有String能用，其他类型用这两个函数返回NaN）

```
//1. 当字符串为纯数字时，直接转换即可
var s = '234'
parseInt(s); // 234
parseFloat(s); //234

// 2. 当字符串是数字加字母等非数字时：
var s = '234string';
parseInt(s);  //234
parseFloat(s);  //234.0

//3. 字符串第一个不是数字
parseInt('bb cc 12')       //返回 NaN
parseInt("AF", 16)        //返回 175；会自动把二进制十六进制八进制的转化成数字
//4. 返回 1；字符串的情况，自会返回第一个数
parseInt('1 2 3')
```

9.2 强制转换

通过String（），Number（），Boolean（）函数强制转换

Boolean(value)——把给定的值转换成Boolean型;
Number(value)——把给定的值转换成数字（可以是整数或浮点数）;
String(value)——把给定的值转换成字符串

```
var str=123;
var str1='123';

console.log(typeof str);             // number
console.log(typeof str1);             //String
console.log(typeof String(str));      // String
console.log(typeof Number(str1));    //Number
```

9.3 数组与字符串的相互转换

1、数组转字符串

需要将数组元素用某个字符连接成字符串，示例代码如下：

```
var a, b,c;
a = new Array(a,b,c,d,e);
b = a.join('-'); //a-b-c-d-e  使用-拼接数组元素
c = a.join(''); //abcde
```

2、字符串转数组

实现方法为将字符串按某个字符切割成若干个字符串，并以数组形式返回，示例代码如下：

```
var str = 'ab+c+de';
var a = str.split('+'); // [ab, c, de]
var b = str.split(''); //[a, b, +, c, +, d, e]
```

10.其他类型到布尔类型的转换规则

以下这些是假值：

• undefined

• null

• false

• +0、-0 和 NaN

• ""

假值的布尔强制类型转换结果为 false。从逻辑上说，假值列表以外的都应该是真值。

11. || （与）和 &&（或） 操作符的返回值?

|| 和 && 首先会对第一个操作数执行条件判断，如果其不是布尔值就先强制转换为布尔类型，然后再执行条件判断。

- 对于 || 来说，如果条件判断结果为 true 就返回第一个操作数的值，如果为 false 就返回第二个操作数的值。

- && 则相反，如果条件判断结果为 true 就返回第二个操作数的值，如果为 false 就返回第一个操作数的值。

|| 和 &&返回它们其中一个操作数的值，而非条件判断的结果

12.Object.is() 与比较操作符 “===”、“==” 的区别?

- 使用双等号（==）进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。
- 使用三等号（===）进行相等判断时，如果两边的类型不一致时，不会做强制类型准换，直接返回 false。
- 使用 Object.is 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 -0 和 +0 不再相等，两个 NaN 是相等的。

13. 什么是 JavaScript 中的包装类型?

在 JavaScript 中，基本类型是没有属性和方法的，但是为了便于操作基本类型的值，在调用基本类型的属性或方法时 JavaScript 可以隐式or显式的将基本类型的值转换为对象，如：

```
// 隐式，后台转化
const a = "abc";
a.length; // 3
a.toUpperCase(); // "ABC"

// 显式：JavaScript也可以使用`Object`函数显式地将基本类型转换为包装类型：
var a = 'abc'
Object(a) // String {"abc"}

//使用valueOf方法将包装类型倒转成基本类型：
var a = 'abc'
var b = Object(a)
var c = b.valueOf() // 'abc'
```

14. JavaScript 中如何进行隐式类型转换?

首先要介绍`ToPrimitive`方法，这是 JavaScript 中每个值隐含的自带的方法，用来将值 （无论是基本类型值还是对象）转换为基本类型值。如果值为基本类型，则直接返回值本身；如果值为对象，其看起来大概是这样:

```
/**
* @obj 需要转换的对象
* @type 期望的结果类型
*/
ToPrimitive(obj,type)
```

1): 对象和布尔值比较

对象和布尔值进行比较时，对象先转换为字符串，然后再转换为数字，布尔值直接转换为数字

```
[] == true; //false []转换为字符串'',然后转换为数字0,true转换为数字1，所以为false
```

2): 对象和字符串比较

对象和字符串进行比较时，对象转换为字符串，然后两者进行比较。

```
[1,2,3] == '1,2,3' // true [1,2,3]转化为'1,2,3'，然后和'1,2,3'， so结果为true;
```

3): 对象和数字比较

对象和数字进行比较时，对象先转换为字符串，然后转换为数字，再和数字进行比较。

```
[1] == 1; // true `对象先转换为字符串再转换为数字，二者再比较 [1] => '1' => 1 所以结果为true
```

4): 字符串和数字比较

字符串和数字进行比较时，字符串转换成数字，二者再比较。

5): 字符串和布尔值比较

字符串和布尔值进行比较时，二者全部转换成数值再比较。

6): 布尔值和数字比较

布尔值和数字进行比较时，布尔转换为数字，二者比较。

```
[] == false;
![] == false;
// 第二个前边多了个!，则直接转换为布尔值再取反，转换为布尔值时，空字符串(''),NaN,0,null,undefined这几个外返回的都是true， 所以! []这个[] => true 取反为false,所以[] == false为true
```

转换规则:

1. `+`操作符`+`操作符的两边有至少一个`string`类型变量时，两边的变量都会被隐式转换为字符串；其他情况下两边的变量都会被转换为数字（有字符串优先转换字符串）
2. `==`操作符：两边的值尽量转换为number
3. `<`和`>`:如果两边都是字符串，则比较字母顺序；其他情况下，转换为数字再比较
4. 对象比较：

```
a.valueOf() // {}，ToPrimitive默认type为number，所以先valueOf，结果还是个对象，下一步
a.toString() // "[object Object]"，现在是一个字符串了
Number(a.toString()) // NaN，根据上面 < 和 > 操作符的规则，要转换成数字
NaN > 2 //false，得出比较结果
```

15. `+`操作符什么时候用于字符串拼接?

如果 + 的其中一个操作数是字符串（或者通过上述对象比较的步骤得到字符串），则执行字符串拼接，否则执行数字加法。对于除了加法的运算符来说，只要其中一方是数字，那么另一方就会被转为数字

16.object.assign和扩展运算法是深拷贝还是浅拷贝，两者区别（待补充）

两者都是浅拷贝

- ES6专门为合并对象提供了Object.assign()方法，其接收的第一个参数作为目标对象，后面的所有参数作为源对象。然后把所有的源对象合并到目标对象中。它会修改了一个对象，因此会触发 ES6 setter。

```
let outObj = {
  inObj: {a: 1, b: 2}
}
let newObj = Object.assign({}, outObj)
newObj.inObj.a = 2
console.log(outObj) // {inObj: {a: 2, b: 2}}
```

- 扩展操作符（...）使用它时，数组或对象中的每一个值都会被拷贝到一个新的数组或对象中。它不复制继承的属性或类的属性，但是它会复制ES6的 symbols 属性。

```
let outObj = {
  inObj: {a: 1, b: 2}
}
let newObj = {...outObj}
newObj.inObj.a = 2
console.log(outObj) // {inObj: {a: 2, b: 2}}
```

17.如何判断一个对象是空对象

- 使用JSON自带的.stringify方法来判断：

```
if(Json.stringify(Obj) == '{}' ){
    console.log('空对象');
}
```

- 使用ES6新增的方法Object.keys()来判断：

```
if(Object.keys(Obj).length < 0){
    console.log('空对象');
}
```

18.如何判断一个对象是否属于某个类?

- 第一种方式，使用 instanceof 运算符来判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。（`instanceof` **只能正确判断引用数据类型**，而不能判断基本数据类型。）
- 第二种方式，通过对象的 constructor 属性来判断，对象的 constructor 属性指向该对象的构造函数，但是这种方式不是很安全，因为 constructor 属性可以被改写。
- 第三种方式，如果需要判断的是某个内置的引用类型的话，可以使用 Object.prototype.toString() 方法来打印对象的[[Class]] 属性来进行判断。

二、常见的DOM操作有哪些

1.DOM节点和BOM

- DOM 指的是**文档对象模型**（Document Object Model），它指的是把文档当做一个对象，这个对象主要定义了处理网页内容的方法和接口。DOM是W3C的标砖，定义了访问HTML和XML文档的标准，即独立于平台和语言的接口。W3C的DOM标准主要分为以下三种

    1. 核心DOM-> 针对任何结构化文档的标准模型
    2. XML DOM -> 针对XML文档的标准模型
    3. HTML DOM-> 针对HTML文档的标准模型

- BOM 指的是浏览器对象模型(Browser Object Model)，它指的是把浏览器当做一个对象来对待，这个对象主要定义了与浏览器进行交互的方法和接口。BOM的核心是 window，而**window 对象具有双重角色**，它既是通过**js 访问浏览器窗口的一个接口**，又是一个 **Global（全局）对象。**这意味着在**网页**中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。window 对象含有 location 对象、navigator 对象、screen 对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象

2.DOM节点的获取

```
 // 按照 id 查询 getElementById
// 按照标签名查询 getElementsByTagName
// 按照类名查询 getElementsByClassName
// 按照 css 选择器查询 querySelectorAll

// 按照 id 查询
var imooc = document.getElementById('imooc') // 查询到 id 为 imooc 的元素
// 按照标签名查询
var pList = document.getElementsByTagName('p')  // 查询到标签为 p 的集合
console.log(divList.length)
console.log(divList[0])
// 按照类名查询
var moocList = document.getElementsByClassName('mooc') // 查询到类名为 mooc 的集合
// 按照 css 选择器查询
var pList = document.querySelectorAll('.mooc') // 查询到类名为 mooc 的集合
var pList = document.querySelectAll('#.mooc')  // 查询到ID名为mooc的元素
```

3.DOM节点的创建

```
/*有html结构1如下*/
<html>
  <head>
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
      <h2 id="title2">我是二级标题</h2>
    </div>
  </body>
</html>
```

例：添加一个有内容的 span 节点到 id 为 title 的节点后面，做法就是：

```
//先获取父元素
const container = document.getElementById('container')
const createSpan = document.createElement('span')
createSpan.innerHTML = "这是新添加的span标签"
container.appendChild(createSpan)
```

4.DOM节点的删除

在html结构1的基础上，删除id为title的元素

```
//方法一：
const container = doucment.getElementById('container')
const targetElement = document.getElementById('title')
container.removeChild(targetElement)

//方法二：通过子节点数组完成删除
const container = document.getElementById('container')
const targetElement = container.childNodes[1]
container.removeChild(targetElement)

//输出获取的数组中，发现List数组为[text, h1#title, text, h2#title2, text]
//初步猜测元素的次序可能为1，3,5,7,9....
```

5.修改DOM元素

修改 DOM 元素这个动作可以分很多维度，比如说移动 DOM 元素的位置，修改 DOM 元素的属性等。以HTML结构1为例，交换h1标签和h2标签的顺序：

```
const container = document.getElementById('container')
const targetElement = document.getElementByTagName('h1')
const sourceElement = doucument.querySelectAll('h2')
//交换元素，将sourceElement放置于targetElement前面,类似的还有replaceChild()操作
container.insertBefore(sourceElement，targetElement)

//给span标签添加新的class,覆盖原来的样式
document.getElementByTagName('span').setAttribute("class","test1")
//追加样式不覆盖原来的class
doucment.getElementByTagName('span').classList.add('test1','test2')
//删除class
doucment.getElementByTagName('span').classList.remove('test1')
//修改DOM元素样式
targetElement.style.color = '#fff'
targetElement.style.border = '1px solid black'
// 。。。。
```

6.什么是事件代理（事件委托）？

事件代理（事件委托），是JavaScript中绑定事件的常用技巧。顾名思义，“事件代理”，就是**把原本需要绑定的事件委托给父元素，让父元素负责事件监听。**事件代理的原理是DOM元素的**事件冒泡。**事件委托的好处：

- 减少事件数量，提高性能
- 预测未来元素，新添加的元素仍然可以触发该事件
- 避免内存外泄，在低版本的IE中，防止删除元素而没有移除事件造成的内存溢出

7.什么是事件冒泡?

事件流是指从页面接收事件的顺序。也就是说，当一个事件发生时，这个事件的传播过程就是事件流。

事件冒泡是由事件开始时由最具体的元素接收，然后逐级向上传播到较为不具体的节点。对于HTML来说当以一个元素产生一个事件时，它会把这个事件传递给它的父元素，父元素接收到之后，还要传递给它上一级的元素，就这样**一直传播到document对象。**

事件冒泡机制：当一个对象上触发了某种事件，此对象定义了此事件的处理程序，那么就会调用这个处理程序；如果没有定义处理程序或者事件返回true，那么这个事件会向父级对象传播，从里到外，直到它被处理，或者到达对象层次的最顶层

事件捕获：就是值不太具体的元素更早的接收到事件，而最具体的节点最后接收到事件。它们的用意就是在事件达到目标之前就捕获它，过程与冒泡相反

```
//例：
(function(){
  console.log(1);
  setTimeout(function(){console.log(2)},1000);
  setTimeout(function(){console.log(3)},0);
  console.log(4);
})()

//上述输出结构为1,4,3,2。1,4在前面是因为console.log没有延迟输出，而2,3在事件中，存在事件循环
```

事件的分类：鼠标事件、键盘事件、表单事件、窗口事件、触屏事件、剪贴板事件、打印事件、多媒体事件、CSS3（动画、过渡等）事件、其他事件。

三、JavaScript基础

1.new操作符的实现原理

new操作符通过构造函数创建出来的实例可以访问到构造函数中的属性，也能够访问到构造函数原型链中的属性。即通过new操作符，实例与构造函数通过原型链连接了起来。

- 如果构造函数**没有**返回`对象类型Object`(包含`Functoin`，`Array`，`Date`，`RegExg`，`Error`)，那么new表达式中的函数调用会自动返回这个新的对象(即返回非对象类型，不会产生任何影响)。
- 如果构造函数返回`对象类型Object`，那么new表达式中的函数调用会返回这个我们设置好的对象，那么这个返回值会被正常使用

1.1 new操作符的执行过程：

1. 首先创建了一个新的空对象
2. 设置原型，将对象的原型设置为函数的 prototype 对象。
3. 让函数的 this 指向这个对象，执行构造函数的代码（为这个新对象添加属性）
4. 判断函数的返回值类型，如果构造函数返回了一个Object（`return { age: 21}`），则此Object会取代整个new出来的结果；如果构造函数返回的是非对象，则new出来的结果则为创建的新对象

1.2 new操作符的作用

1. 创建了一个全新的对象。
2. 这个对象会被执行`[[Prototype]]`（也就是`__proto__`）链接。
3. 生成的新对象会绑定到函数调用的this。
4. 通过new创建的每个对象将最终被`[[Prototype]]`链接到这个函数的`prototype`对象上。
5. 如果构造函数没有返回对象类型Object，那么new表达式中的函数调用会自动返回这个新的对象。【返回原始值需要忽略，返回对象需要正常处理】

2.map和weakMap的区别

2.1 Map

传统的Object中的键只能是字符串、数字或symbol，而ES6提供的map本质上就是键值对（key-value）的集合，键值对中的键不限制范围，可以是任意类型，是一种更加完善的Hash结构。（PS：如果Map的键是一个原始数据类型，只要两个键严格相同，就视为是同一个键。）

实际上Map是一个数组，它的每一个数据也都是一个数组，其形式如下：

```
//定义空Map:
    let map = new Map();
    map.set('Tom',22);
    map.set('Jerry',20);
    console.log(map);//Map(2) {"Tom" => 22, "Jerry" => 20}
    //定义时直接初始化（数据为二维数组）
    let map2 = new Map([['Tom',22],['Jerry',20]]);
    console.log(map2);//Map(2) {"Tom" => 22, "Jerry" => 20}

//存储形式
const map = [
     ["name","张三"],
     ["age",18],
]
```

Map有以下操作方法：

- **size**：`map.size` 返回Map结构的成员总数。
- **set(key,value)**：设置键名key对应的键值value，然后返回整个Map结构，如果key已经有值，则键值会被更新，否则就新生成该键。（因为返回的是当前Map对象，所以可以链式调用）
- **get(key)**：该方法读取key对应的键值，如果找不到key，返回undefined。
- **has(key)**：该方法返回一个布尔值，表示某个键是否在当前Map对象中。
- **delete(key)**：该方法删除某个键，返回true，如果删除失败，返回false。
- **clear()**：map.clear()清除所有成员，没有返回值
- **map.keys()**：返回键名的遍历器。
- **map.values()**：返回键值的遍历器。
- **map.entries()**：返回所有成员的遍历器。
- **map.forEach()**：遍历Map的所有成员

2.2 Map与Object的比较：

1. 对于 `Object` 而言，它键（key）的类型只能是字符串，数字或者 `Symbol`；而对于 `Map` 而言，它可以是**任何类型。**（包括 Date，Map，或者自定义对象）
2. `Map` 中的元素会**保持其插入时的顺序**；而 `Object` 则不会完全保持插入时的顺序，按照一定的规则进行排序

   Object的排序规则：

   - **非负整数**会最先被列出，排序是从小到大的数字顺序
   - 然后所有字符串，负整数，浮点数会被列出，顺序是根据插入的顺序
   - 最后才会列出 `Symbol`，`Symbol` 也是根据插入的顺序进行排序的
3. 读取 `Map` 的长度很简单，只需要调用其 `.size()` 方法即可；而读取 `Object` 的长度则需要额外的计算：`Object.keys(obj).length`
4. Map 是**可迭代对象**，所以其中的键值对是可以通过 `for of` 循环或 `.foreach()` 方法来循环迭代的；而普通的Object键值对则默认是不可迭代的，只能通过 `for in` 循环来访问
5. `JSON` 默认支持 `Object` 而不支持 `Map`。若想要通过 `JSON` 传输 `Map` 则需要使用到 `.toJSON()` 方法，然后在 `JSON.parse()` 中传入**复原函数**来将其复原

6. 创建时语法不通，读取/删减/删除元素时存在不同
7. Map性能优于Object，在拥有一定数量的元素时，Object所需内存大于Map

Map	Object
Map默认情况不包含任何键，只包含显式插入的键。	Object 有一个原型, 原型链上的键名有可能和自己在对象上的设置的键名产生冲突。
Map的键可以是任意值，包括函数、对象或任意基本类型。	Object 的键必须是 String 或是Symbol。
Map 中的 key 是有序的。因此，当迭代的时候， Map 对象以插入的顺序返回键值。	Object 的键是无序的（按照一定规则排序）
Map 的键值对个数可以轻易地通过size 属性获取	Object 的键值对个数只能手动计算 `Object.keys(obj).length`
Map 是 iterable 的，所以可以直接被迭代。	迭代Object需要以某种方式获取它的键然后才能迭代。
在频繁增删键值对的场景下表现更好。	在频繁添加和删除键值对的场景下未作出优化。

2.3 weakMa

WeakMap 对象也是一组键值对的集合，其中的键是弱引用的。**其键必须是Object**类型，原始数据类型不能作为key值，而值可以是任意的。WeakMap的设计目的在于，有时想在某个对象上面存放一些数据，但是这会形成对于这个对象的引用。一旦不再需要这两个对象，就必须手动删除这个引用，否则垃圾回收机制就不会释放对象占用的内存。

```
const wm = new WeakMap();
const arr = ["Tom"];
//添加操作
wm.set(arr, 20);
console.log(wm.has(arr)); //true
//删除操作
wm.delete(arr);
//检索判断
console.log(wm.has(arr)); //false
```

weakMap的操作方法：

- **set(key,value)**：设置键名key对应的键值value，然后返回整个Map结构，如果key已经有值，则键值会被更新，否则就新生成该键。（因为返回的是当前Map对象，所以可以链式调用）
- **get(key)**：该方法读取key对应的键值，如果找不到key，返回undefined。
- **has(key)**：该方法返回一个布尔值，表示某个键是否在当前Map对象中。
- **delete(key)**：该方法删除某个键，返回true，如果删除失败，返回false

使用weakMap注意：

- 键名必须是对象（ null 除外）
- WeaMap对键名是弱引用的，键值是正常引用（弱引用：在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。）
- 垃圾回收不考虑WeaMap的键名，不会改变引用计数器，键在其他地方不被引用时即删除

- 因为WeakMap 是弱引用，由于其他地方操作成员可能会不存在，所以不可以进行forEach( )遍历等操作
- 也是因为弱引用，WeaMap 结构没有keys( )，values( )，entries( )等方法和 size 属性
- 当键的外部引用删除时，希望自动删除数据时使用 WeakMap

2.4 总结：

- Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。
- WeakMap 结构与 Map 结构类似，也是用于生成键值对的集合。但是 WeakMap 只接受对象作为键名（ null 除外），不接受其他类型的值作为键名。而且 WeakMap 的键名所指向的对象，不计入垃圾回收机制。

3.JavaScript有哪些内置对象?

js 中的内置对象主要指的是在程序执行前存在全局作用域里的由 js 定义的一些全局值属性、函数和用来实例化其他对象的构造函数对象。一般经常用到的如全局变量值 NaN、undefined，全局函数如 parseInt()、parseFloat() 用来实例化对象的构造函数如 Date、Object 等，还有提供数学计算的单体内置对象如 Math 对象。主要包括全局的对象（全局作用域里的对象），或称标准内置对象

1. 值属性，这些全局属性返回一个简单值，这些值没有自己的属性和方法。例如 Infinity、NaN、undefined、null 字面量
2. 函数属性，全局函数可以直接调用，不需要在调用时指定所属对象，执行结束后会将结果直接返回给调用者。例如 eval()、parseFloat()、parseInt() 等
3. 基本对象，基本对象是定义或使用其他对象的基础。基本对象包括一般对象、函数对象和错误对象。例如 Object、Function、Boolean、Symbol、Error 等
4. 数字和日期对象，用来表示数字、日期和执行数学计算的对象。例如 Number、Math、Date
5. 字符串，用来表示和操作字符串的对象。例如 String、RegExp
6. 可索引的集合对象，这些对象表示按照索引值来排序的数据集合，包括数组和类型数组，以及类数组结构的对象。例如 Array
7. 使用键的集合对象，这些集合对象在存储数据时会使用到键，支持按照插入顺序来迭代元素。例如 Map、Set、WeakMap、WeakSet
8. 矢量集合，SIMD 矢量集合中的数据会被组织为一个数据序列。例如 SIMD 等
9. 结构化数据，这些对象用来表示和操作结构化的缓冲区数据，或使用 JSON 编码的数据。例如 JSON 等
10. 控制抽象对象，例如 Promise、Generator 等
11. 反射，例如 Reflect、Proxy
12. 国际化，为了支持多语言处理而加入 ECMAScript 的对象。例如 Intl、Intl.Collator 等
13. WebAssembly
14. 其他，例如 arguments

4.对JSON的理解

JSON 是一种**基于文本的轻量级的数据交换格式。**它可以被**任何的编程语言**读取和作为数据格式来传递。在项目开发中，使用 JSON 作为前后端数据交换的方式。在前端通过将一个符合 JSON 格式的数据结构序列化为 JSON 字符串，然后将它传递到后端，后端通过 JSON 格式的字符串解析后生成对应的数据结构，以此来实现前后端数据的一个传递。

因为 JSON 的语法是基于 js 的，因此很容易将 JSON 和 js 中的对象弄混，但是应该注意的是 JSON 和 js 中的对象不是一回事，**JSON 中对象格式更加严格**，比如说在 JSON 中属性值不能为函数，不能出现 NaN 这样的属性值等，因此大多数的 js 对象是不符合 JSON 对象的格式的。

在 js 中提供了两个函数来实现 js 数据结构和 JSON 格式的转换处理，

- **JSON.stringify** 函数，通过传入一个符合 JSON 格式的数据结构，将其转换为一个 JSON 字符串。如果传入的数据结构不符合 JSON 格式，那么在序列化的时候会对这些值进行对应的特殊处理，使

其符合规范。在前端向后端发送数据时，可以调用这个函数将数据对象转化为 JSON 格式的字符串。

- **JSON.parse()** 函数，这个函数用来将 JSON 格式的字符串转换为一个 js 数据结构，如果传入的字符串不是标准的 JSON 格式的字符串的话，将会抛出错误。当从后端接收到 JSON 格式的字符串时，可以通过这个方法来将其解析为一个 js 数据结构，以此来进行数据的访问

```
// js数据结构转JSON字符串
var str = {"name":"菜鸟教程", "site":"http://www.runoob.com"}
str_pretty1 = JSON.stringify(str)
document.write("<pre>" + str_pretty1 + "</pre>" );    //使用

// JSON字符串转js数据结构
{ "name":"runoob", "alexa":10000, "site":"www.runoob.com" } //转换对象一定得是JSON，否则会出现转换错误
var obj = JSON.parse('{ "name":"runoob", "alexa":10000, "site":"www.runoob.com" }');
document.getElementById("demo").innerHTML = obj.name + ": " + obj.site;  //使用
```

5.JavaScript的同步加载和异步加载

同步加载，又称阻塞模式，是我们平时使用最多的方式，也**就是直接将`<script>`写在`<head>`里。**这种方式会阻止浏览器的后续处理，停止后续的解析，直到当前的加载完成。一般来说，**同步加载是安全的**，但如果我们js里设计到document内容输出、获取或修改DOM结构等行为，就会产生页面阻塞代码出错。所以一般就会建议把`<script>`写在页面最底部,以减少页面阻塞。(这种方式可能也是我们刚开始接触到js优化，最常使用的一种方式。)

异步加载，又称为非阻塞加载，在浏览器下载执行js的同时，还会继续后续页面的处理。这里也是一般面试会问到的一点，即js延迟加载的方式有哪些?

6.JavaScript脚本延迟加载的方式有哪些?

延迟加载就是等页面加载完成之后再加载 JavaScript 文件。 js 延迟加载有助于提高页面加载速度。 **之所以要优化是因为HTML元素是按其在页面中出现的次序调用的，**如果用javascript来管理页面上的元素（使用文档对象模型dom），并且js加载于欲操作的HTML元素之前，则代码将出错。也就是说，**我们写了js语句来获取DOM对象，但由于DOM结构还没有加载完成，因此获取到的是空对象。**一般有以下几种方式:

- **defer 属性：**给 js 脚本添加 defer 属性，表明脚本在执行时不会影响页面的构造，浏览器会立即下载，但延迟执行，即**脚本会被延迟到整个页面都解析完毕之后再执行。**defer属性只适用于外部脚本文件，只有 Internet Explorer 支持 defer 属性。并且defer属性解决了async引起的脚本顺序问题，使用defer属性，脚本将按照在页面中出现的顺序加载和运行。
- **async 属性：**给 js 脚本添加 async 属性，这个属性会使**脚本异步加载**(HTML和脚本一并加载)，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。async只适用于外部脚本
- **动态创建 DOM 方式：**动态创建 DOM 标签的方式，可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。
- **使用 setTimeout 延迟方法：**设置一个定时器来延迟加载js脚本文件
- **让 JS 最后加载：**将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行

defer和async的使用区别：

- 如果脚本无需等待页面解析，且无依赖独立运行，那么应使用`async`。也就是每一个async属性的脚本都在它下载结束之后立即执行，同时会在window的load事件之前执行。

- 如果脚本需要等待解析，且依赖于其它脚本，调用这些脚本时应使用 `defer`，将关联的脚本按所需顺序置于 HTML 中。

7.JavaScript中的类数组对象

类数组：一个长得像数组的对象，一个拥有 length 属性和若干索引属性的对象就可以被称为类数组对象

7.1 类数组和数组的区别：

1、都有length属性
2、类数组也可以for循环遍历，有的类数组还可以通过 for of 遍历
3、类数组不具备数组的原型方法，因此类数组不可调用相关数组方法（如，push,slicec,concat等等）

7.2 常见的类数组：

1. **argummments对象**：arguments对象用于保存数组中的实参，可以直接通过arguments[0]访问函数的第一个实参
2. 通过getElementsByTagName，getElementsByClassName，getElementsByName等方法获取的dom列表（也叫 HTMLCollection）

7.3 类数组向数组转换

```
//定义类数组
var arrObj = { 0: 'dog', 1: 'cat', 2: 'rabbit', 'length': 3 }
//方法1：通过call调用数组的slice方法
Array.prototype.slice.call(arrObj)
//方法2：通过call调用数组的splice方法
Array.prototype.splice.call(arrObj, 0)
//方法3：array.from方法
Array.from(arrObj)
//方法4：通过apply调用concat方法
Array.prototype.concat.apply([], arrObj)
//方法5：for循环一个一个的取值
var arr = []
for(var i = 0; i < arrObj.length; i++) {
    arr.push(arrObj[i])
}
console.log(arr)

// ["dog", "cat", "rabbit"]
```

8.数组有哪些原生方法?

- 数组和字符串的转换方法：toString()、toLocalString()、join() 。其中 join() 方法可以指定转换为字符串时的分隔符。（当数字是四位及以上时，toLocaleString()会让数字三位三位一分隔，且和toString在转换时间格式上存在差异）
- 数组尾部操作的方法 pop() 和 push()，push 方法可以传入多个参数。
- 数组首部操作的方法 shift() 和 unshift()
- 重排序的方法 reverse() （倒置数组）和 sort()，sort() 方法可以传入一个函数来进行比较，传入前后两个值，如果返回值为正数，则交换两个参数的位置。
- 数组连接的方法 concat() ，返回的是拼接好的数组，不影响原数组。
- 数组截取办法 slice()，用于截取数组中的一部分返回，不影响原数组。
- 数组插入方法 splice()，影响原数组
- 查找特定项的索引的方法，indexOf() (该方法接受一个数据，返回该数据在数组中的从左往右第一个索引)和 lastIndexOf() (从右往左)
- 迭代方法 every()、some()、filter()、map() 和 forEach() 方法

> map和forEach都是用来遍历数组的，两者区别如下：
>
> - forEach()方法会针对每一个元素执行提供的函数，对数据的操作会改变原数组，该方法没有返回值；
> - map()方法不会改变原数组的值，返回一个新数组，新数组中的值为原数组调用函数处理之后的值

```
let array =[1,2,3,4,5,6]
  array.forEach((value,key)=>{
      return array[key] = value * key
  })

  console.log(array)  //[0, 2, 6, 12, 20, 30]
  let array2 =[1,2,3,4,5,6]
  let array3=array2.map(value=>{
      return value * value
  })
  console.log(array2)  //[1,2,3,4,5,6] 原函数没变
  console.log(array3)  //[1, 4, 9, 16, 25, 36] 返回新的数组
```

- 数组归并方法 reduce() 和 reduceRight() 方法

9. GET、GETTER、SET、SETTER

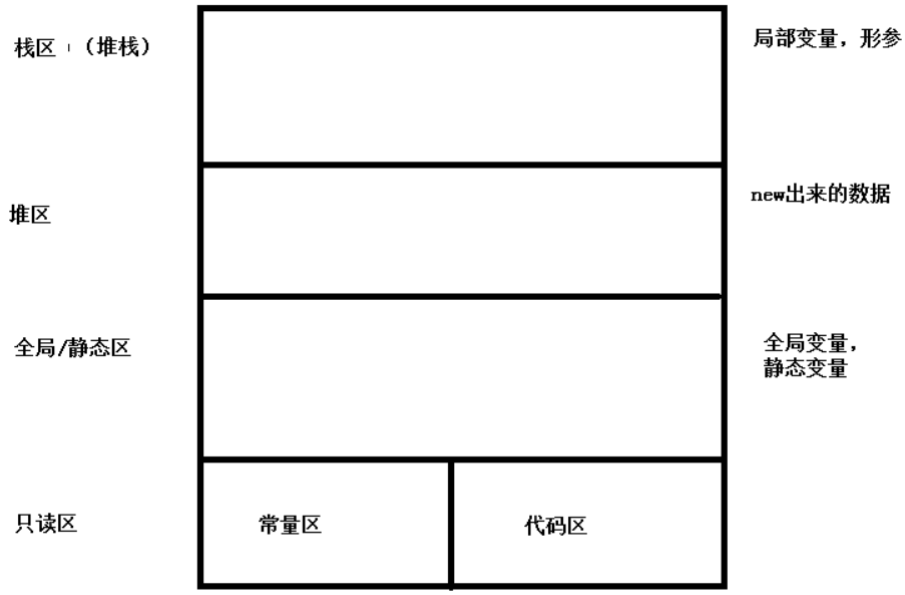
- getter 是一种获得属性值的方法，负责查询值，它不带任何参数
- setter是一种设置属性值的方法，值以参数的形式传递，在他的函数体中，一切的return都是无效的
- get/set访问器不是对象的属性，而是属性的特性，特性只有内部才用

```
// set和get的使用
var obj3 = {
        a:100,
        b:8,
        set val(n){
            this.a = n
        },
        get val2(){
            return this.a
        },
    }

    console.log(obj3.a)   //100
    console.log(obj3.val2)   //100  此时还未赋值
    obj3.val = 5
    console.log(obj3.val2)   //5

//setter和getter的使用
```

10.浅拷贝与深拷贝

10.1 内存的分区

分为四个区域：栈区（堆栈），堆区，全局静态区，只读区（常量区和代码区）。



1、栈区

1. 存放的数据：局部变量，形参，被调用函数的地址（这个可以不用管）等等
2. 特点：

   读取速度快，存储和释放的思路是按照数据结构中的栈进行的，存数据就是压栈，释放就是弹栈；

   空间小，基本类型的数据占用空间的大小不会随着值的改变而改变，而且占用空间小。

2、堆区：

1. 存放数据：new出来的数据。
2. 特点：

   读取速度慢

   空间大：引用类型的数据大小是动态的，会随着数据的增加而改变大小

3、全局静态区：

1. 存放数据：全局变量和静态变量
2. 特点：在程序运行过程中，数据会一直在内存中。

4、只读区：

1. 存放数据：常量区存放常量，代码区存放程序的代码（程序运行时是需要载入到内存中允许的）
2. 特点：此区域的数据在程序运行过程中肯定不能改变。

10.2 浅拷贝与深拷贝

```
var p = {
    "id":"007",
    "name":"刘德华",
    "books":new Array("三国演义","红楼梦","水浒传")//这是引用类型
}

//浅拷贝
var p2 = {};
for(let key in p){
```

```
    p2[key] = p[key];
}
p2.books[0] ="四国";
console.log(p2);   //“四国”，"红楼梦","水浒传"
console.log(p);   //“四国”，"红楼梦","水浒传"     //因为两者的books仍指向同一个内存地址，一改则全改

//深拷贝
var p2 = {};
for(let key in p){
    if(typeof p[key]=='object'){
        p2[key]=[];//因为,我上面写的是数组,所以,暂时赋值一个空数组.
        for(let i in p[key]){
            p2[key][i] = p[key][i]
        }
    }else{
        p2[key] = p[key];
    }
}
p2.books[0] ="四国";
console.log(p2);  //“四国”，"红楼梦","水浒传"
console.log(p);  // "三国演义","红楼梦","水浒传"

//深拷贝：如果属性都是json对象，则采用递归方式
var p = {
    "id":"007",
    "name":"刘德华",
    "wife":{
        "id":"008",
        "name":"刘德的妻子",
        "address":{
            "city":"北京",
            "area":"海淀区"
        }
    }
}

//写函数
function copyObj(obj){
    let newObj={};
    for(let key in obj){
        if(typeof obj[key] =='object'){//如:key是引用类型,那就递归
            newObj[key] = copyObj(obj[key])
        }else{//基本类型,直接赋值
            newObj[key] = obj[key];
        }
    }
    return newObj;
}

let pNew = copyObj(p);
pNew.wife.name="张三疯";
pNew.wife.address.city = "香港";
console.log(pNew);
console.log(p);
```

11.事件监听addEventListener()

- **EventTarget . addEventListener()** 方法将指定的监听器注册到 EventTarget 上，当该对象触发指定的事件时，指定的回调函数就会被执行。 事件目标可以是一个文档上的元素 Element，Document和Window或者任何其他支持事件的对象。
- addEventListener()的工作原理是将实现EventListener的函数或对象添加到调用它的EventTarget上的指定事件类型的事件侦听器列表中。

```
target.addEventListener(type, listener, options);
target.addEventListener(type, listener, useCapture);
target.addEventListener(type, listener, useCapture, wantsUntrusted);
document.getElementById("myBtn").addEventListener("click", function(){
    document.getElementById("demo").innerHTML = "Hello World";
});
/*
（1）type
表示监听事件类型的字符串。
（2）listener
当所监听的事件类型触发时，会接收到一个事件通知（实现了 Event 接口的对象）对象。listener 必须是一个实现了 EventListener 接口的对象，或者是一个函数。
（3）options 可选
一个指定有关 listener 属性的可选参数对象。可用的选项如下：
• capture： Boolean，表示 listener 会在该类型的事件捕获阶段传播到该 EventTarget 时触发。
• once： Boolean，表示 listener 在添加之后最多只调用一次。如果是 true， listener 会在其被调用之后自动移除。
• passive: Boolean，设置为true时，表示 listener 永远不会调用 preventDefault()。如果 listener 仍然调用了这个函数，客户端将会忽略它并抛出一个控制台警告。
• signal：AbortSignal，该 AbortSignal 的 abort() 方法被调用时，监听器会被移除。
（4）useCapture 可选
Boolean，在DOM树中，注册了listener的元素， 是否要先于它下面的EventTarget，调用该listener。 当useCapture(设为true) 时，沿着DOM树向上冒泡的事件，不会触发listener。当一个元素嵌套了另一个元素，并且两个元素都对同一事件注册了一个处理函数时，所发生的事件冒泡和事件捕获是两种不同的事件传播方式。事件传播模式决定了元素以哪个顺序接收事件。如果没有指定， useCapture 默认为 false 。
（5）wantsUntrusted
如果为 true ，则事件处理程序会接收网页自定义的事件。此参数只适用于 Gecko（chrome的默认值为true，其他常规网页的默认值为false），主要用于附加组件的代码和浏览器本身。
*/
```

四、ES6

1.let、const、var的区别

（1）块级作用域：块作用域由 `{ }`包括，let和const具有块级作用域，var不存在块级作用域。块级作用域解决了ES5中的两个问题：

- 内层变量可能覆盖外层变量
- 用来计数的循环变量泄露为全局变量

（2）变量提升：var存在变量提升，let和const不存在变量提升，即在变量只能在声明之后使用，否在会报错。

（3）给全局添加属性：浏览器的全局对象是window，Node的全局对象是global。var声明的变量为全局变量，并且会将该变量添加为全局对象的属性，但是let和const不会。

（4）重复声明：var声明变量时，可以重复声明变量，后声明的同名变量会覆盖之前声明的遍历。const和let不允许重复声明变量。

（5）暂时性死区：在使用let、const命令声明变量之前，该变量都是不可用的。这在语法上，称为**暂时性死区**。使用var声明的变量不存在暂时性死区。

（6）初始值设置：在变量声明时，var 和 let 可以不用设置初始值。而const声明变量必须设置初始值。

（7）指针指向：let和const都是ES6新增的用于创建变量的语法。 let创建的变量是可以更改指针指向（可以重新赋值）。但const声明的变量是不允许改变指针的指向

2.const对象的属性可以修改吗

- const保证的并不是变量的值不能改动，而是**变量指向的那个内存地址**不能改动。对于基本类型的数据（数值、字符串、布尔值），其值就保存在变量指向的那个内存地址，因此等同于常量。
- 但对于引用类型的数据（主要是对象和数组）来说，变量指向数据的内存地址，保存的只是一个指针，const只能保证这个指针是固定不变的，至于它指向的数据结构是不是可变的，就完全不能控制了。

3.arguments为什么是一个类数组而不是数组

`arguments`是一个对应于传递给函数的参数的类数组对象，它的属性是从 0 开始依次递增的数字，还有`callee`和`length`等属性，与数组相似；但是它却没有数组常见的方法属性，如`forEach`，`reduce`等，所以它是一个**类数组**。

- arguments对象和Function是分不开的,因为arguments这个对象不能显式创建
- arguments对象只有函数开始时才可用

```
function test() {
        var s = "";
        for (var i = 0; i < arguments.length; i++) {
            alert(arguments[i]);
            s += arguments[i] + ",";
        }
        return s;
}
test("name", "age");

输出结果：
name,age
```

4.箭头函数与普通函数的区别

1. 箭头函数比普通函数更加简洁

- 如果没有参数，就直接写一个空括号即可
- 如果只有一个参数，可以省去参数的括号
- 如果有多个参数，用逗号分割
- 如果函数体的返回值只有一句，可以省略大括号
- 如果函数体不需要返回值，且只有一句话，可以给这个语句前面加一个void关键字。最常见的就是调用一个函数

```
let fn = () => void doesNotReturn();
```

2. 箭头函数没有自己的this且继承来的this指向永远不会变

箭头函数不会创建自己的this， 所以它没有自己的this，它只会在自己作用域的上一层继承this。所以箭头函数中this的指向在它在定义时已经确定了，之后不会改变

```
var id = 'GLOBAL';
var obj = {
  id: 'OBJ',
  a: function(){
    console.log(this.id);
  },
  b: () => {
    console.log(this.id);
  }
};
obj.a();    // 'OBJ'
obj.b();    // 'GLOBAL'    =>  这个函数中的this就永远指向它定义时所处的全局执行环境中的this，即便这个函数是作为对象obj的方法调用，this依旧指向Window对象
new obj.a()  // undefined
new obj.b()  // Uncaught TypeError: obj.b is not a constructor
```

3. call()、apply()、bind()等方法不能改变箭头函数中this的指向

```
var id = 'Global';
let fun1 = () => {
    console.log(this.id)
};
fun1();                     // 'Global'
fun1.call({id: 'Obj'});     // 'Global'
fun1.apply({id: 'Obj'});    // 'Global'
fun1.bind({id: 'Obj'})();   // 'Global'
```

4. 箭头函数不能作为构造函数使用

构造函数在new的步骤在上面已经说过了，实际上第二步就是将函数中的this指向该对象。 但是由于箭头函数时没有自己的this的，且this指向外层的执行环境，且不能改变指向，所以不能当做构造函数使用。

5. 箭头函数没有自己的arguments

箭头函数没有自己的arguments对象。在箭头函数中访问arguments实际上获得的是它外层函数的arguments值。

6. 箭头函数没有prototype
7. 箭头函数不能用作Generator函数，不能使用yeild关键字

5. 如果new一个箭头函数的会怎么样

箭头函数是ES6中的提出来的，它没有prototype，也没有自己的this指向，更不可以使用arguments参数，所以不能New一个箭头函数。new操作符的实现步骤如下：

1. 创建一个对象
2. 将构造函数的作用域赋给新对象（也就是将对象的**proto**属性指向构造函数的prototype属性）
3. 指向构造函数中的代码，构造函数中的this指向该对象（也就是为这个对象添加属性和方法）
4. 返回新的对象

所以，上面的第二、三步，箭头函数都是没有办法执行的

6.箭头函数的this指向哪里?

箭头函数不同于传统JavaScript中的函数，箭头函数并没有属于自己的this，它所谓的this是捕获其所在上下文的 this 值，作为自己的 this 值，并且由于没有属于自己的this，所以是不会被new调用的，这个所谓的this也不会被改变。

7.扩展运算符作用及其使用场景

7.1 对象扩展运算符

对象的扩展运算符(...)用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中，属于浅拷贝

```
let bar = { a: 1, b: 2 };
let baz = { ...bar }; // { a: 1, b: 2 }
//等价于Object.assign方法
let bar = { a: 1, b: 2 };
let baz = Object.assign({}, bar); // { a: 1, b: 2 }
//被覆盖
let bar = {a: 1, b: 2};
let baz = {...bar, ...{a:2, b: 4}};  // {a: 2, b: 4}
```

`Object.assign`方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。`Object.assign`方法的第一个参数是目标对象，后面的参数都是源对象。(**如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性**)。

同样，如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉

7.2 数组扩展运算符

数组的扩展运算符可以将一个数组转为用逗号分隔的参数序列，且每次只能展开一层数组。

```
console.log(...[1, 2, 3])
// 1 2 3
console.log(...[1, [2, 3, 4], 5])
// 1 [2, 3, 4] 5

//主要应用于：1.将数组转换为参数序列
function add(x, y) {
  return x + y;
}
const numbers = [1, 2];
add(...numbers) // 3

//2. 复制数组
const arr1 = [1,2]
const arr2 = [...arr1]

//3. 合并数组
const arr1 = ['two', 'three'];
const arr2 = ['one', ...arr1, 'four', 'five'];
// ["one", "two", "three", "four", "five"]

//4. 生成数组（如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错）
const [first, ...rest] = [1, 2, 3, 4, 5];
first // 1
rest  // [2, 3, 4, 5]
```

```
//5. 将字符串转为真正的数组
[...'hello']    // [ "h", "e", "l", "l", "o" ]

//6. 任何 Iterator 接口的对象，都可以用扩展运算符转为真正的数组
function foo() {
  const args = [...arguments];  //用于替换es5中的
Array.prototype.slice.call(arguments)写法
}

//7. 使用Math函数获取数组中特定的值
const numbers = [9, 4, 7, 1];
Math.min(...numbers); // 1
Math.max(...numbers); // 9
```

8.Proxy可以实现什么功能（待补充）

在 Vue3.0 中通过 `Proxy` 来替换原本的 `Object.defineProperty` 来实现数据响应式。Proxy 是 ES6 中新增的功能，它可以用来自定义对象中的操作，实现getter和setter的以上所有内容（getter和setter是属性级别的拦截，代理是对象级别的拦截，功能更强大）。其中，getter负责获取值，不带任何参数；setter负责设置值，在它的函数体中，一些的return都是无效的

8.1 Proxy定义

MDN定义：Proxy 对象用于定义基本操作的自定义行为（如属性查找，赋值，枚举，函数调用等）。

通俗的讲Proxy是一个对象操作的拦截器，拦截对目标对象的操作，进行一些自定义的行为，一种分层的思想有点类似spring的AOP。

需要实现一个 Vue 中的响应式，需要在 `get` 中收集依赖，在 `set` 派发更新，之所以 **Vue3.0 要使用 `Proxy` 替换原本的 API 原因在于 `Proxy` 无需一层层递归为每个属性添加代理，一次即可完成以上操作，性能上更好，并且原本的实现有一些数据更新不能监听到，但是 `Proxy` 可以完美监听到任何方式的数据改变**，唯一缺陷就是浏览器的兼容性不好。

8.2 Proxy的使用

```
let p = new Proxy(target,handler)
// target：用Proxy包装的目标对象（可以是任何类型的对象，包括原生数组，函数，甚至另一个代理）。
// handler：自定义对象中的操作

let test2={
      name:'SFONE',
      age:'24'
}

test2 = new Proxy(test2,{
     get(target,key){
          let result = target[key]
          if( key === 'age' ){
              result += '岁'
          }
          return result
      },
      set(target,key,value) {
          if( key === 'age' && typeof value != Number){
              throw Error('age字段必须为Number类型')
          }
```

```
    //Reflect.set(): 在一个对象上设置一个属性。返回一个boolean值表示是否设置成功
    // 异常时抛出TypeError，如果对象不是Object
        return Reflect.set(target,key,value)
      }
});
  console.log(`我叫${test2.name}，我今年${test2.age}了`)
  test2.school = 'SWUST'
  console.log(test2)   //Proxy {name: 'SFONE', age: '24', school: 'SWUST'}
```

9.对象与数组的解构

解构是 ES6 提供的一种新的提取数据的模式，这种模式能够从对象或数组里有针对性地拿到想要的数值。

1）数组的解构

解构数组时，以元素的位置为匹配条件来提取想要的数据的

```
const [a, b, c] = [1, 2, 3]
const [a,,c] = [1,2,3]  // a:1,c:3
```

2）对象的解构

在解构对象时，是以属性的名称为匹配条件，来提取想要的数据的。

```
//原型对象
const stu = {
  name: 'Bob',
  age: 24
}
const { name, age } = stu  // name:bob   age:24
// 得到name 和 age 两个和 stu 平级的变量
```

10.如何提取高度嵌套的对象里的指定属性

```
const school = {
   classes: {
      stu: {
         name: 'Bob',
         age: 24,
      }
   }
}

//老方法:
const { classes } = school
const { stu } = classes
const { name } = stu
name // 'Bob'

//解构:在解构出来的变量名右侧，通过冒号+{目标属性名}这种形式，进一步解构它，一直解构到拿到目标数据为止。
const { classes:{ stu: { name } }} = school
console.log( name )   //bob
```

11.对rest参数的理解

扩展运算符被用在函数形参上时，**它还可以把一个分离的参数序列整合成一个数组。**这一点**经常用于获取函数的多余参数，或者像上面这样处理函数参数个数不确定的情况**

```
function mutiple(...args) {
    console.log(args)    //[1,2,3,4]
  let result = 1;
  for ( var val of args) {
    result *= val;
  }
  return result;
}
mutiple(1, 2, 3, 4) // 24
```

12.ES6中模板语法与字符串处理

1. 模板字符串优点：

- 允许用${}的方式嵌入变量
- 在模板字符串中，空格、缩进、换行都会被保留
- 模板字符串完全支持"运算"式的表达式，可以在${}里完成一些计算

```
var name = 'css'
var career = 'coder'
var hobby = ['coding', 'writing']
// ES5
var finalString = 'my name is ' + name + ', I work as a ' + career + ', I love '
+ hobby[0] + ' and ' + hobby[1]
//ES6
var finalString = `my name is ${name}, I work as a ${career} I love ${hobby[0]}
and ${hobby[1]}`
// 在模板字符串中写html代码
let list = `
    <ul>
        <li>列表项1</li>
        <li>列表项2</li>
    </ul>
`;
console.log(message); // 正确输出，不存在报错

// 在模板字符串中进击一些简单的计算和调用
function add(a, b) {
  const finalString = `${a} + ${b} = ${a+b}`
  console.log(finalString)
}
add(1, 2) // 输出 '1 + 2 = 3'
```

2.模板字符串中的一些方法

```
//1.includes：判断字符串与子串的包含关系
const son = 'haha'
const father = 'xixi haha hehe'
father.includes(son) // true
```

```
//2.startsWith：判断字符串是否以某个/某串字符开头
const father = 'xixi haha hehe'
father.startsWith('haha') // false
father.startsWith('xixi') // true

//3. endsWith：判断字符串是否以某个/某串字符结尾：
const father = 'xixi haha hehe'
father.endsWith('hehe') // true

//4. 自动重复：可以使用 repeat 方法来使同一个字符串输出多次（被连续复制多次）：
const sourceCode = 'repeat for 3 times;'
const repeated = sourceCode.repeat(3)
console.log(repeated) // repeat for 3 times;repeat for 3 times;repeat for 3
times;
```

13.ES6模块、CommonJS模块、AMD模块（重点关注，待补充）

- CommonJS和ES6 Module都可以对引入的对象进行赋值，即对对象内部属性的值进行改变
- CommonJS是对模块的浅拷贝，ES6 Module是对模块的引用，即ES6 Module只存只读，不能改变其值，也就是指针指向不能变，类似const；
-

14.尾调用（Tail Call）与尾递归

尾调用：指某个函数的最后一步是调用另一个函数

```
function f(x){
  return g(x);
}

//下列均不属于尾调用
// 情况一：
function f(x){
  let y = g(x);
  return y;
}

// 情况二
function f(x){
  return g(x) + 1;
}
```

尾递归：递归非常耗费内存，因为需要同时保存成千上百个调用记录，很容易发生"栈溢出"错误（stack overflow）。但对于尾递归来说，由于只存在一个调用记录，所以永远不会发生"栈溢出"错误，相对节省内存

ES6的尾调用优化只在严格模式下开启，正常模式是无效的。这是因为在正常模式下，函数内部有两个变量，可以跟踪函数的调用栈。

> - `arguments`：返回调用时函数的参数。
> - `func.caller`：返回调用当前函数的那个函数。

尾调用优化发生时，函数的调用栈会改写，因此上面两个变量就会失真。严格模式禁用这两个变量，所以尾调用模式仅在严格模式下生效。

15.严格模式（use strict）

use strict 是一种 ECMAscript5 添加的（严格模式）运行模式，这种模式使得 Javascript 在更严格的条件下运行。设立严格模式的目的如下：

- 消除 Javascript 语法的不合理、不严谨之处，减少怪异行为;
- 消除代码运行的不安全之处，保证代码运行的安全；
- 提高编译器效率，增加运行速度；
- 为未来新版本的 Javascript 做好铺垫。

区别：

- 禁止使用 with 语句。
- 禁止 this 关键字指向全局对象。
- 对象不能有重名的属性。
- ES6下的尾调用仅在严格模式下开启

16. for...in 和 for... of的区别

for...of 是ES6新增的遍历方式，允许遍历一个含有**iterator接口**的数据结构（数组、对象等）并且返回各项的值，和ES3中的for...in的区别如下：

- **for...of 遍历获取的是对象的键值，for...in 获取的是对象的键名**；
- for... in 会遍历对象的整个原型链，性能非常差不推荐使用，而 for ... of 只遍历当前对象不会遍历原型链；
- 对于数组的遍历，for...in 会返回数组中所有可枚举的**属性**(包括原型链上可枚举的属性)，for...of 只返回数组的下标对应的属性值；

总结：for...in 循环主要是为了遍历对象而生，不适用于遍历数组；for...of 循环可以用来遍历数组、类数组对象，字符串、Set、Map 以及 Generator 对象

> **iterator**：原有的表示集合的数据结构，主要是数组和对象，ES6又添加了Map跟Set，Iterator是一种统一的接口机制，用来处理不同的数据结构
>
> - iterator是一种数据接口，为各种不同的数据结构提供统一的访问机制
> - 任何数据结构只要部署了iterator接口，就称这种数据结构是可遍历的
> - ES6规定，默认的iterator接口部署在数据结构的Symbol. iterator属性上
> - 原生具备iterator接口的数据结构有：Array, Map, Set, TypedArray, 函数的arguements对象，NodeList对象

17.属性枚举顺序

for...in循环、Object.keys()、Objcet.getOwnPrototypeNames()、Object.getOwnPrototypeSymbols、Object.assign在属性枚举方面有很大的差异性，for...in循环和Object.keys()的枚举顺序不确定，取决于JS引擎，可能因浏览器而异。
Objcet.getOwnPrototypeNames()、Object.getOwnPrototypeSymbols、Object.assign()的枚举顺序是确定的。

Object.values()返回对象值的数组，Object.entries()返回键/值对的数组

17.数据传输ajax/axios/fetch

17.1 AJAX

AJAX是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的 异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。通过在后台与服务器进行少量数据交换，Ajax 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。传统的网页（不使用 Ajax）如果需要更新内容，必须重载整个网页页面。其缺点

如下：

- 本身是针对MVC编程，不符合前端MVVM的浪潮
- 基于原生XHR开发，XHR本身的架构不清晰
- 不符合关注分离（Separation of Concerns）的原则
- 配置和调用方式非常混乱，而且基于事件的异步模型不友好。

> MVC,MVP,MVVM是三种常见的前端架构模式(Architectural Pattern),它通过分离关注点来改进代码组织方式。不同于设计模式(Design Pattern),只是为了解决一类问题而总结出的抽象方法，一种架构模式往往能使用多种设计模式。
>
> MVC模式是MVP,MVVM模式的基础，这两种模式更像是MVC模式的优化改良版,他们三个的MV即Model，view相同，不同的是MV之间的纽带部分。本文主要介绍MVC与MVVM的应用与区别，因为MVP好像不是很常用。

创建AJAX请求的步骤：

- **创建一个 XMLHttpRequest 对象。**
- 在这个对象上**使用 open 方法创建一个 HTTP 请求**，open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。
- 在发起请求前，可以为这个对象**添加一些信息和监听函数。**比如说可以通过 setRequestHeader 方法来为请求添加头信息。还可以为这个对象添加一个状态监听函数。一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发onreadystatechange 事件，可以通过设置监听函数，来处理请求成功后的结果。当对象的 readyState 变为 4 的时候，代表服务器返回的数据接收完成，这个时候可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候就可以通过 response 中的数据来对页面进行更新了。
- 当对象的属性和监听函数设置完成后，最后调**用 sent 方法来向服务器发起请求**，可以传入参数作为发送的数据体

```
// promise 封装实现ajax:
function getJSON(url) {
  // 创建一个 promise 对象
  let promise = new Promise(function(resolve, reject) {
    let xhr = new XMLHttpRequest();
    // 新建一个 http 请求
    xhr.open("GET", url, true);
    // 设置状态的监听函数
    xhr.onreadystatechange = function() {
      if (this.readyState !== 4) return;
      // 当请求成功或失败时，改变 promise 的状态
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    // 设置错误监听函数
    xhr.onerror = function() {
      reject(new Error(this.statusText));
    };
    // 设置响应的数据类型
    xhr.responseType = "json";
    // 设置请求头信息
    xhr.setRequestHeader("Accept", "application/json");
    // 发送 http 请求
    xhr.send(null);
  });
```

```
  return promise;
}
```

7.2 Fetch

fetch号称是AJAX的替代品，是在ES6出现的，使用了ES6中的promise对象。Fetch是基于promise设计的。Fetch的代码结构比起ajax简单多。**fetch不是ajax的进一步封装，而是原生js，没有使用XMLHttpRequest对象。**

fetch的优点：

- 语法简洁，更加语义化
- 基于标准 Promise 实现，支持 async/await
- 更加底层，提供的API丰富（request, response）
- 脱离了XHR，是ES规范里新的实现方式

fetch的缺点：

- fetch只对网络请求报错，对400，500都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。
- fetch默认不会带cookie，需要添加配置项： fetch(url, {credentials: 'include'})
- fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费
- fetch没有办法原生监测请求的进度，而XHR可以

7.3 Axios

Axios 是一种基于Promise封装的HTTP客户端，其特点如下：

- 浏览器端发起XMLHttpRequests请求
- node端发起http请求
- 支持Promise API
- 监听请求和返回
- 对请求和返回进行转化
- 取消请求
- 自动转换json数据
- 客户端支持抵御XSRF攻击

7.4 URL的转义

- encodeURI 是对整个 URI 进行转义，将 URI 中的非法字符转换为合法字符，所以对于一些在 URI 中有特殊意义的字符不会进行转义。
- encodeURIComponent 是对 URI 的组成部分进行转义，所以一些特殊字符也会得到转义。
- escape 和 encodeURI 的作用相同，不过它们对于 unicode 编码为 0xff 之外字符的时候会有区别，escape 是直接在字符的 unicode 编码前加上 %u，而 encodeURI 首先会将字符转换为 UTF-8 的格式，再在每个字节前加上 %

五、原型与原型链

1.构造函数

构造函数模式的目的就是为了创建一个自定义类，并且创建这个类的实例。构造函数模式中拥有了类和实例的概念，并且实例和实例之间是相互独立的，即实例识别。构造函数就是一个普通的函数，创建方式和普通函数没有区别，不同的是构造函数习惯上首字母大写。

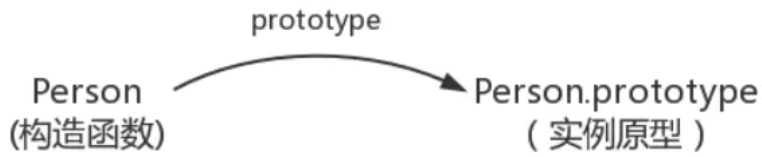
另外就是调用方式的不同，普通函数是直接调用，而构造函数需要使用**new关键字**来调用。

```
function Person(name, age, gender) {     //创建一个构造函数
        this.name = name
        this.age = age
        this.gender = gender
        this.sayName = function () {     //定义构造函数中的一个方法
            alert(this.name);
        }
    }
    var per = new Person("孙悟空", 18, "男");      //调用创建的构造函数，必须用new关键字调用

    console.log(per)              //当我们直接在页面中打印一个对象时，事件上是输出的对象的toString()方法的返回值

//toSting()：  可以把一个Number转换为字符串
```

每创建一个Person构造函数，在Person构造函数中，为每一个对象都添加了一个sayName方法，也就是说构造函数每执行一次就会创建一个新的sayName方法。这样就导致了构造函数执行一次就会创建一个新的方法，执行10000次就会创建10000个新的方法，而10000个方法都是一摸一样的，为什么不把这个方法单独放到一个地方，并让所有的实例都可以访问到呢?这就需要**原型(`prototype`)**

2.原型对象

原型对象就相当于一个公共的区域，所有同一个类的实例都可以访问到这个原型对象，我们可以将对象中共有的内容，统一设置到原型对象中在JavaScript中。每当定义一个函数数据类型(普通函数、类)时候，都会天生自带一个`prototype`属性，这个属性**指向函数的原型对象**，并且**这个属性是一个对象数据类型的值。**



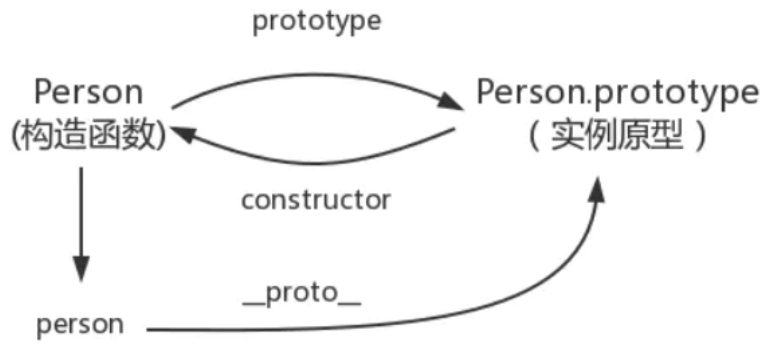
> 在JavaScript中是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性，它的**属性值是一个对象**，这个对象**包含了可以由该构造函数的所有实例共享的属性和方法**。当使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。一般来说不应该能够获取到这个值的，但是现在浏览器中实现了 `__proto__` 属性来访问这个属性，但是最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，可以通过这个方法来获取对象的原型。

3.原型链

3.1 .`__proto__`和`constructor`

每一个对象数据类型(普通的对象、实例、`prototype`.....)也天生自带一个属性`__proto__`，属性值是当前实例所属类的原型(`prototype`)。原型对象中有一个属性`constructor`,它指向函数对象。

```
function Person() {}
    var person = new Person()
    console.log(person.__proto__ === Person.prototype)//true
    console.log(Person.prototype.constructor===Person)//true
    //顺便学习一个ES5的方法,可以获得对象的原型
    console.log(Object.getPrototypeOf(person) === Person.prototype) // true
```

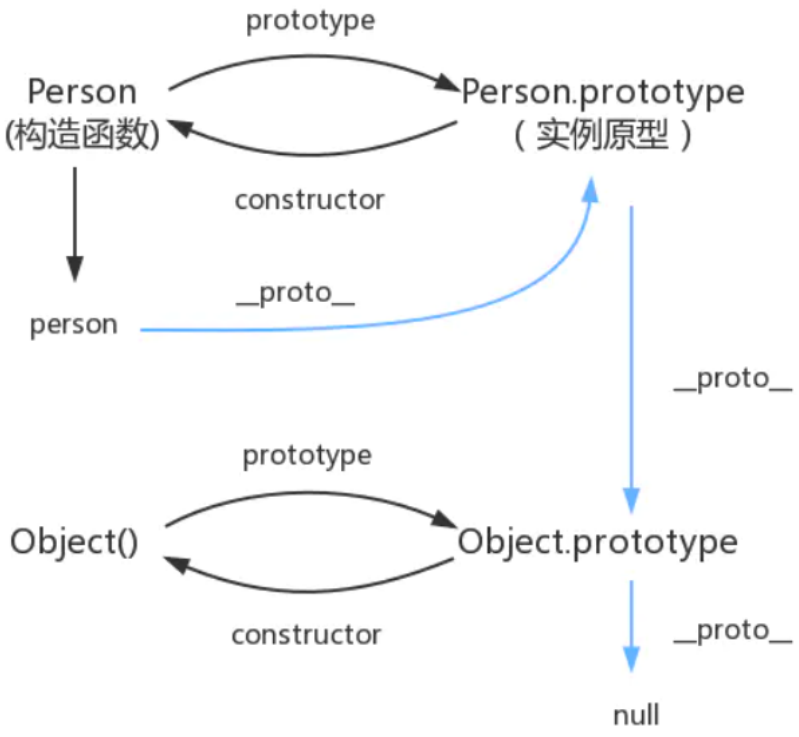


3.2 何为原型链?

在JavaScript中万物都是对象，对象和对象之间也有关系，并不是孤立存在的。对象之间的继承关系，**在JavaScript中是通过prototype对象指向父类对象，直到指向Object对象为止**（person → Person → Object），这样就形成了一个原型指向的链条，专业术语称之为原型链

当我们访问对象的一个属性或方法时，它会先在对象自身中寻找，如果有则直接使用，如果没有则会去原型对象中寻找，如果找到则直接使用。如果没有则去原型的原型中寻找,直到找到Object对象的原型，Object对象的原型没有原型，如果在Object原型中依然没有找到，则返回undefined。注意：Object对象是老祖宗，没人比他更大了，所以Object的`_proto_`为空, 即**原型链的尽头一般来说都是Object.prototype**

```
console.log(Object.prototype.__proto__ === null) // true
```



我们可以使用对象的`hasOwnProperty()`来检查对象自身中是否含有该属性；使用`in`检查对象中是否含有某个属性时，如果对象中没有但是原型中有，也会返回true

```
 function Person() {}                                                    //创建构造函数
    Person.prototype.a = 123;                                            //给构造函数的
原型实例中添加a属性，而不是在构造函数中添加a属性
    Person.prototype.sayHello = function () {                          //给构造函数的原
型实例中添加sayHello方法，而不是在构造函数中添加该方法
      alert("hello");
    };

    var person = new Person()                                            //创建一个构造函
数实例对象
    console.log(person.a)//123                                         // 虽然在构造函数中
没有a属性，但是在其父函数（实例原型）中有该属性，所以构造
函数继承了a属性，仍能正常输出而不报错
    console.log(person.hasOwnProperty('a'));//false                      //
hasOwnProperty方法仅检查对象自身是否含有某属性，a属性是继承的，自身中没
有，所以不能正常输出
    console.log('a'in person)//true                                    // 若当前实例对象的
构造函数中没有属性a时，in方法可以查询其父函数或祖父函数中
是否有a属性，即查户口查到祖宗十八代去了，23333


//getPrototypeOf():返回参数内部特性Prototype的值
console.log(Object.getPrototypeOf(person1) == Person.prototype)  //true
```

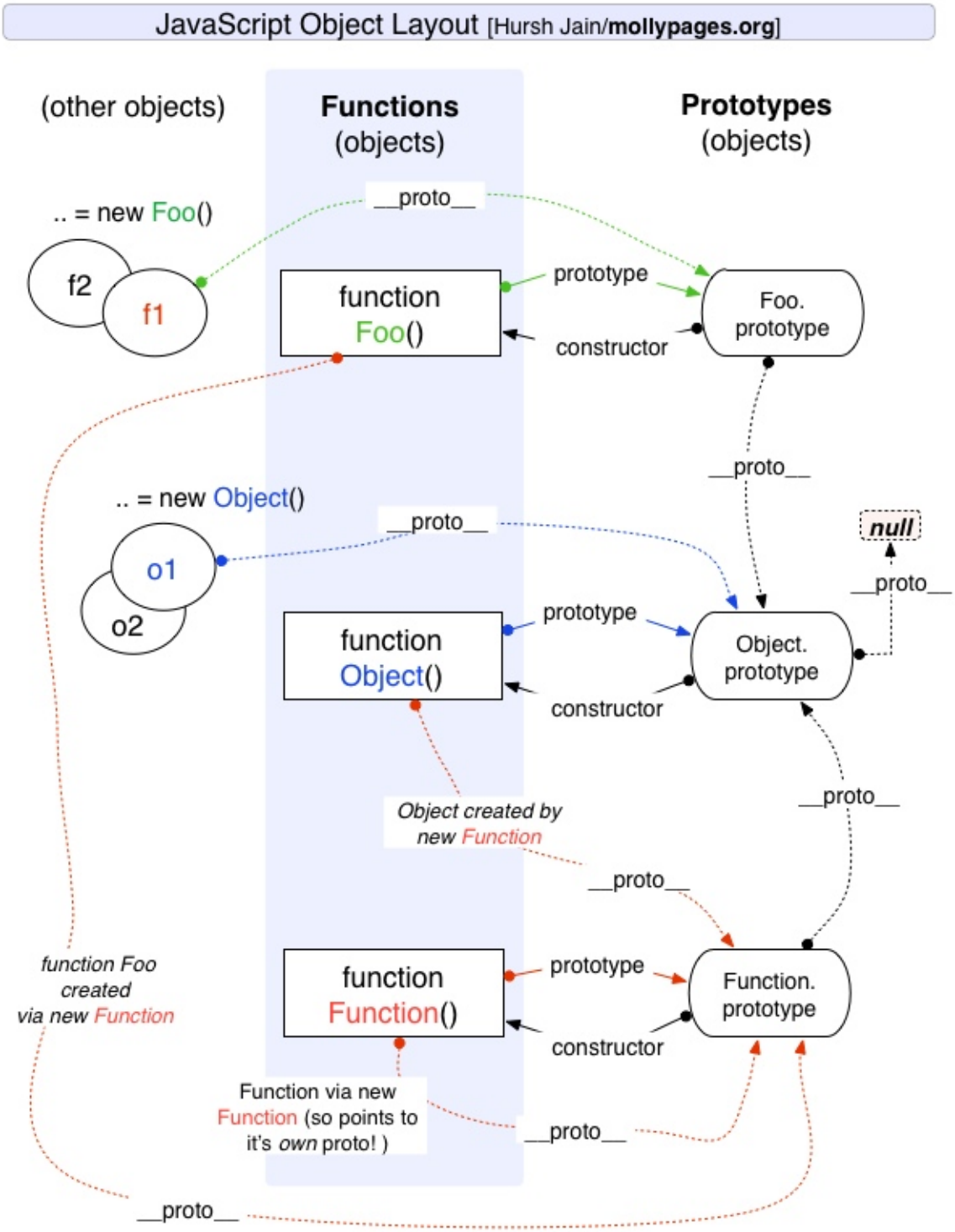
4.原型链指向

```
p.__proto__  // Person.prototype
Person.prototype.__proto__  // Object.prototype
p.__proto__.__proto__ //Object.prototype
p.__proto__.constructor.prototype.__proto__ // Object.prototype
Person.prototype.constructor.prototype.__proto__ // Object.prototype
p1.__proto__.constructor // Person
Person.prototype.constructor  // Person
```

5.如何获取对象中非原型链上的属性?

使用后`hasOwnProperty()`方法来判断属性是否属于原型链的属性

```
function iterate(obj){
   var res=[];
   for(var key in obj){
        if(obj.hasOwnProperty(key))
           res.push(key+': '+obj[key]);
   }
   return res;
}
```



function关键词本身也有一个原型对象"Function"，其他函数在使用"function"关键词时，一定存在属性继承于原型对象"Function.prototype"

六、 闭包/ 执行上下文/ 作用域链

1.闭包（需要多理解代码题）

1.1 定义

闭包是指有权访问另一个函数作用域中变量的函数，创建闭包的最常见的方式就是在一个函数内创建另一个函数(即嵌套函数)，创建的函数可以访问到当前函数的局部变量

```
// 在函数中包含着函数B，函数B可以访问函数A的私有变量a，函数B为闭包
function A() {
  let a = 1
  window.B = function () {
      console.log(a)
  }
}
A()
B() // 1
```

1.2 作用

- 在函数外部能够访问到函数内部的变量。通过使用闭包，可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。
- 使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收.

1.3 经典面试题：循环中使用闭包解决 var 定义函数的问题

```
for (var i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)                          //在setTimeout中的函数引用了外层 for循环的变量i，导致i一直存在内存中，不被回收，所以等到JS队列执行函数时，
i已经是6了
  }, i * 1000)
}
```

因为 `setTimeout` 是个异步函数，所以会先把循环全部执行完毕，这时候 `i` 就是 6 了，所以会输出一堆 6。解决方如下：

方法一：使用let定义i变量（最推荐？）

```
for (let i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)
  }, i * 1000)
}
```

方法二：使用闭包的方式

```
//思路：使用立即函数，将i传入立即函数内部，此时i被固定在了参数j上。当下次执行timer闭包时，直接可以使用参数j以达到目的
for (var i = 1; i <= 5; i++) {
  ;(function(j) {                                   //这种写法是一个立即执行函数，写法一：用括号把整个函数定义和调用包裹起来(function(){}())
    setTimeout(function timer() {                   // 写法二：用括号把函数定义包裹起来，后面再加括号（本段代码的写法）
      console.log(j)                                //立即函数的写法都是以圆括号开头，引擎为认定后面跟着表达式，而不是函数定义语句，以避免错误
    }, j * 1000)
  })(i)
}

//立即执行函数的作用：立即执行函数会形成一个单独的作用域，我们可以封装一些临时变量或者局部变量，避免污染全局变量
```

```
//不必为函数命名，避免了污染全局变量
//立即执行函数内部形成了一个独立的作用域，可以封装一些外部无法读取的私有变量，这个作用域里面的变量，外面访问不到，这样就可以避免变量污染
//封装变量
//闭包和私有数据
//https://blog.csdn.net/weixin_43876206/article/details/106025126
```

方法三：使用setTimeout第三个参数, 这个参数会被当成 `timer` 函数的参数传入

```
for (var i = 1; i <= 5; i++) {
  setTimeout(
    function timer(j) {
      console.log(j)
    },
    i * 1000,
    i
  )}
```

2.作用域与作用域链

2.1 全局作用域

- 最外层函数和最外层函数外面定义的变量拥有全局作用域
- 所有未定义直接赋值的变量自动声明为全局作用域
- 所有window对象的属性拥有全局作用域
- 全局作用域有很大的弊端，过多的全局作用域变量会污染全局命名空间，容易引起命名冲突

2.2 函数作用域

- 函数作用域声明在函数内部的变零，一般只有固定的代码片段可以访问到
- 作用域是分层的，内层作用域可以访问外层作用域，反之不行

2.3 块级作用域

- 使用ES6中新增的let和const指令可以声明块级作用域，块级作用域可以在函数中创建也可以在一个代码块中创建（由 `{ }` 包裹的代码片段）
- let和const声明的变量不会有**变量提升**，也不可以重复声明
- 在循环中比较适合绑定块级作用域，这样就可以把声明的计数器变量限制在循环内部
- ES6 引入了块级作用域，明确允许在块级作用域之中声明函数。ES6 规定，块级作用域之中，函数声明语句的行为类似于 `let`，在块级作用域之外不可引用。

2.4 变量提升

变量提升的表现是，无论在函数中何处位置声明的变量，好像都被提升到了函数的首部，可以在变量声明前访问到而不会报错。

本质原因：是 js 引擎在代码执行前有一个解析的过程，创建了**执行上下文**，初始化了一些代码执行时需要用到的对象。当访问一个变量时，会到当前执行上下文中的作用域链中去查找，而**作用域链的首端指向的是当前执行上下文的变量对象**，这个变量对象是执行上下文的一个属性，它包含了函数的形参、所有的函数和变量声明，这个对象的是在代码解析的时候创建的

解析阶段：JS会检查语法，并对函数进行预编译。解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来，变量先赋值为undefined，函数先声明好可使用。

提升变量的好处：

1）提高性能

在JS代码执行之前，会进行语法检查和预编译，并且这一操作只进行一次。这么做就是为了提高性能，**如果没有这一步，那么每次执行代码前都必须重新解析一遍该变量（函数）**，而这是没有必要的，因为变量（函数）的代码并不会改变，解析一遍就够了。

在解析的过程中，还会为函数生成预编译代码。在预编译时，会统计声明了哪些变量、创建了哪些函数，并对函数的代码进行压缩，去除注释、不必要的空白等。这样做的好处就是**每次执行函数时都可以直接为该函数分配栈空间**（不需要再解析一遍去获取代码中声明了哪些变量，创建了哪些函数），并且因为代码压缩的原因，代码执行也更快了。

2）容错性

```
a = 1;
var a;
console.log(a);
```

缺点：

ES6中提出了let、const来定义变量，它们就没有变量提升的机制。

```
// case1：在这个函数中，原本是要打印出外层的tmp变量，但是因为变量提升的问题，内层定义的tmp被提到函数内部的最顶部，相当于覆盖了外层的tmp，所以打印结果为undefined。
var tmp = new Date();
function fn(){
    console.log(tmp);
    if(false){
        var tmp = 'hello world';
    }
}:
fn();  // undefined

//case2：由于遍历时定义的i会变量提升成为一个全局变量，在函数结束之后不会被销毁，所以打印出来11。
var tmp = 'hello world';
for (var i = 0; i < tmp.length; i++) {
    console.log(tmp[i]);
}
console.log(i); // 11
```

2.5 作用域链

在当前作用域中查找所需变量，但是该作用域没有这个变量，那这个变量就是自由变量。如果在自己作用域找不到该变量就去父级作用域查找，依次向上级作用域查找，直到访问到window对象就被终止，这一层层的关系就是作用域链。

作用：保证对执行环境中有权访问的 所有变量和函数的**有序访问**，通过作用域链，可以访问到外层环境的变量和函数。

作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的**前端始终都是当前执行**上下文的变量对象。全局执行上下文的变量对象（也就是**全局对象）始终是作用域链的最后**一个对象。

3.执行上下文

执行上下文定义：执行上下文就是**当前 JavaScript 代码被解析和执行时所在的环境**， 给js执行提供了基础和必备资源。JavaScript 中运行任何的代码都是在执行上下文中运行

3.1 全局执行上下文（重点）

任何不在函数内部的都是全局执行上下文，在客户端中一般由浏览器创建。首先会创建一个全局的window对象，并且**设置this的值等于这个全局对象**（即：能通过this直接访问到window对象），一个程序中**只有一个全局执行上下文。**

全局对象window上预定义了大量的方法和属性，我们在全局环境的任意处都能直接访问这些属性方法，同时window对象还是var声明的全局变量的载体。我们通过var创建的全局对象，都可以通过window直接访问

3.2 函数执行上下文（重点）

当一个函数被调用时，就会为该函数创建一个新的执行上下文，函数执行上下文可存在无数多个，每当一个函数被调用时都会创建一个函数上下文；需要注意的是，**同一个函数被多次调用，都会创建一个新的上下文**，不过函数执行上下文会多出this、arguments和函数的参数。

- 全局上下文：变量定义，函数声明
- 函数上下文：变量定义，函数声明，`this`，`arguments`

3.3 `eval`函数执行上下文

执行在eval函数中的代码有属于他自己的执行上下文，不过eval函数不常使用，不做介绍。

3.4 执行上下文栈（调用栈）

执行栈用于存储代码执行期间创建的所有上下文，具有**LIFO（Last In First Out后进先出，也就是先进后出）**的特性。原理如下：

1. JS代码首次运行，都会先创建一个全局执行上下文并压入到执行栈中，之后每当有函数被调用，都会创建一个新的函数执行上下文并压入栈内；
2. 引擎会执行位于执行上下文栈顶的函数，当函数执行完成之后，执行上下文从栈中弹出，继续执行下一个上下文。
3. 当所有的代码都执行完毕之后，从栈中弹出全局执行上下文。由于执行栈LIFO的特性，所以可以理解为，JS代码执行完毕前在**执行栈底部永远有个全局执行上下文。**

```
let a = 'Hello World!';
function first() {
  console.log('Inside first function');
  second();
  console.log('Again inside first function');
}
function second() {
  console.log('Inside second function');
}
first();
//执行顺序
//先压入全局执行上下文，先执行second(),再执行first()
```

创建执行上下文：

创建执行上下文有两个阶段：**创建阶段**和**执行阶段**

1）创建阶段

```
ExecutionContext = {
    // 确定this的值
    ThisBinding = <this value>,
    // 创建词法环境组件
    LexicalEnvironment = {},
    // 创建变量环境组件
    VariableEnvironment = {},
};
```

（1）this绑定

- 在全局执行上下文中，this指向全局对象（window对象）
- 在函数执行上下文中，this指向取决于函数如何调用。如果它被一个引用对象调用，那么 this 会被设置成那个对象，否则 this 的值被设置为全局对象或者 undefined

（2）创建词法环境组件

- 词法环境是一种有**标识符—>变量**映射的数据结构，标识符是指变量/函数名，变量是对实际对象或原始数据的引用。
- 词法环境的内部有两个组件：加粗样式和环境记录器:用来储存变量个函数声明的实际位置

  外部环境的引用：可以访问父级作用域

（3）创建变量环境组件

- 变量环境也是一个词法环境，其环境记录器持有变量声明语句在执行上下文中创建的绑定关系。

2）执行阶段

此阶段会完成对变量的分配，最后执行完代码

总结：

在执行JS代码之前，需要先解析代码。解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来，变量先赋值undefined，函数先声明好可使用。这一步执行完了，才开始正式的执行程序。

七、JS中的this指向

1.this理解

1.1 隐式绑定：

this的指向在函数定义的时候是确定不了的，只有**函数执行的时候才能确定this到底指向谁**，实际上this的最终指向的是那个调用它的对象

1、当函数被单独定义和调用的时候，应用的规则就是绑定全局变量window

```
function fn() {
    console.log( this.a );
}
var a = 2;
fn(); // 2 -- fn单独调用，this引用window
```

2、如果一个函数中有this，这个函数有被上一级的对象所调用，那么this指向的就是上一级的对象(官话：当函数引用有上下文对象时，隐式绑定规则会把函数调用中的this绑定到这个上下文对象。)

3、如果一个函数中有this，这个函数中包含多个对象，尽管这个函数是被最外层的对象所调用，this指向的也只是它**上一级**的对象

```
var o = {
    a:10,
    b:{
        a:12,
        fn:function(){
            console.log(this.a); //undefined
            console.log(this); //window
        }
    }
}

o.b.fn()     //this的调用对象是她的上一级，b
window.o.b.fun()     //此时this指向仍未b，因为this指向上一级的对象所调用
var j = o.b.fn()
j()                //this的调用对象是window，因为this永远指向最后调用它的对象，此时fn虽然是被b引用，但赋值给j时并未执行，所以最终指向还是window
```

4、如果返回值是一个对象，那么this指向的就是那个返回的对象，如果返回值不是一个对象那么this还是指向函数的实例.(特殊情况：虽然null也是对象，但是在这里this还是指向那个函数的实例，因为null比较特殊)

1.2 显示绑定

1：构造函数版this

new关键字可以改变this的指向，将其指向构造的对象实例中

```
function Fn(){
    this.user = "追梦子";
}
var a = new Fn();
console.log(a.user);            //此时this指向对象a
```

2：call()/ apply() /bind()

使用call()/ apply() /bind()函数，他们接受的第一个参数即是上下文对象并将其赋值给this。如果传递为null，那么结果就是在绑定默认全局变量

call() 、apply()、bind()的区别

- apply 接受两个参数，第一个参数指定了函数体内 this 对象的指向，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，apply 方法把这个集合中的元素作为参数传递给被调用的函数。
- call 传入的参数数量不固定，跟 apply 相同的是，第一个参数也是代表函数体内的 this 指向，从第二个参数开始往后，每个参数被**依次**传入函数。即call()方法使用它自有的实参列表作为函数的实参，apply()方法则要求以数组的形式传入参数。
- call和apply都是改变上下文中的this并立即执行这个函数，bind方法可以让对应的函数想什么时候调就什么时候调用（bind 返回的是一个新的函数，你必须调用它才会被执行）。
- call()改过this的指向后，会再执行函数，bind()改过this后，不执行函数，会返回一个绑定新this的函数

```
var a = {
    user:"追梦子",
    fn:function(e,ee){
        console log(this.user); //追梦子
```

```
        console.log(e+ee); //3
    }
}

// call方法
var b = a.fn;
b.call(a,1,2);

// apply方法
var b = a.fn;
b.apply(a,[10,1]);

//bind()方法
function f(){
console.log("看我怎么被调用");
console.log(this) //指向this
}
var obj = {};
f.call(obj) //直接调用函数

var g = f.bind(obj); //bind()不能调用函数
g();  //此时才调用函数
```

八、常见的位运算符

运算符	描述	运算规则
`&`	与	两个位都为1时，结果才为1
`|`	或	两个位都为0时，结果才为0
`^`	异或	两个位相同为0，相异为1
`~`	取反	0变1，1变0
`<<`	左移	各二进制位全部左移若干位，高位丢弃，低位补0
`>>`	右移	各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃

九、异步编程

0. 并发、并行与回调函数

- 并发是宏观概念，我分别有任务 A 和任务 B，在一段时间内通过任务间的切换完成了这两个任务，这种情况就可以称之为并发。
- 并行是微观概念，假设 CPU 中存在两个核心，那么我就可以同时完成任务 A、B。同时完成多个任务的情况就可以称之为并行。
- 回调函数：回调函数有一个致命的弱点，就是容易写出回调地狱（Callback hell）。回调地狱的根本问题就是：
    1. 嵌套函数存在耦合性，一旦有所改动，就会牵一发而动全身
    2. 嵌套函数一多，就很难处理错误
    3. 回调函数不能使用 `try catch` 捕获错误，不能直接 `return`

```
ajax(url, () => {
    // 处理逻辑
    ajax(url1, () => {
        // 处理逻辑
        ajax(url2, () => {
            // 处理逻辑
        })
    })
})
```

1.异步编程的实现方式

- **回调函数** 的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成**回调函数地狱**，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。（callback回调函数***定义***: 回调函数就是一个通过函数指针调用的函数。如果你把函数的**指针（地址）作为参数**传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。）
- **Promise** 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。
- **generator** 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。
- **async 函数** 的方式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行

2.对回调函数的理解

2.1 回调函数的种类

1. 同步回调

   理解： 立即执行，完全执行了才会结束，不会放入回调队列中

   例子：数组遍历相关的回调函数/ Promise的excutor函数

2. 异步回调

   理解：不会立即执行，会放入回调队列中将来执行

   例子：定时器回调/ ajax回调/ Promise的成功| 失败的回调

```
//1.同步回调函数
const arr = [1,3,5]
arr.forEach(item =>{        //遍历回调
    console.log(item)
})
console.log('foeEach()之后')
// 打印结果1,3,5,foeEach()之后

//2.异步回调函数
setTimeout(()=>{
```

```
        console.log('timeout now')
    },0)
    console.log('timeout()之后')
    // 打印结果 timeout()之后，timeout now
```

2.2 js中error的错误

1. 错误的类型

   Error：所有错误的父类型

   ReferenceError：引用的变量不存在

   TypeError：数据类型不正确的错误

   RangeError：数据值不在其所允许的范围内

   SyntaxError：语法错误

2. 错误处理

   捕获错误：try...catch

   ```
   try{
       let b
       console.log(d.xxx)
   }catch(error){
       console.log(error.message)
   }

   //错误对象
   // message属性：错误相关信息
   //stack属性：函数调用栈记录信息
   ```

   抛出错误：throw error

   ```
   //抛出异常
   function something(){
       if(Data.now()%2 === 1){
           console.log('当前时间为奇数，可以执行任务')
       }else{
           throw new Error('当前时间为偶数，无法执行任务')
       }
   }

   //捕获异常
   try{
       something()
   }catch(error){
       alert(error.message)
   }
   ```

3.Promise的理解和使用

3.1 Promise是什么

1.抽象表达

Promise是JS中进行异步编程的一种**新解决方案**（旧的方法：纯回调形式）

```
// 下述代码为akax发送数据，● 后一个请求需要依赖于前一个请求成功后，将数据往下传递，会导致多个ajax请求嵌套的情况，代码不够直观。如果前后两个请求不需要传递参数的情况下，那么后一个请求也需要前一个请求成功后再执行下一步操作，这种情况下，那么也需要如上编写代码，导致代码不够直观
let fs = require('fs')
fs.readFile('./a.txt','utf8',function(err,data){
  fs.readFile(data,'utf8',function(err,data){
    fs.readFile(data,'utf8',function(err,data){
      console.log(data)
    })
  })
})

// promise写法，更简洁，解决了地狱回调问题
let fs = require('fs')
function read(url){
  return new Promise((resolve,reject)=>{
    fs.readFile(url,'utf8',function(error,data){
      error && reject(error)
      resolve(data)
    })
  })
}
read('./a.txt').then(data=>{
  return read(data)
}).then(data=>{
  return read(data)
}).then(data=>{
  console.log(data)
})
```

2.具体表达

从**语法**上来说，Promise是一个构造函数；从**功能**上来说，Promise对象用来封装一个异步操作并可以获取其结果

3.2 Promise的状态改变

Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。一个 Promise 实例有三种状态，分别是pending、resolved 和 rejected，分别代表了进行中、已成功和已失败。

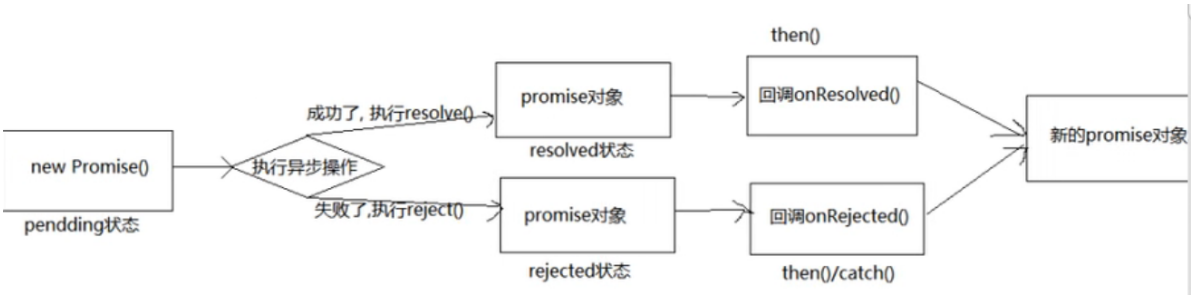
Promise一共有两种变化状态，且**状态一经改变，就凝固了，无法变形**

1.pending -> resolved 成功

2.pending -> rejected 失败

Promise的状态变化只有上述两种，且一个Promise对象只能改变一次，无论变为成功还是失败，都会有一个结果数据。成功的结果数据一般称为value，失败的结果数据一般称为reason

3.3 Promise的基本流程与使用




```
//1. 创建一个新的Promise对象
    const p = new Promise((resolve,reject)=>{     // 执行器函数
        //2. 执行异步操作任务
        setTimeout(()=>{
            const time = Data.now()
                //3.1 如果成功了，调用resolve(value)
            if(time%2 === 0){
                resolve('成功的数据，time=' + time)
            }else {
                //3.2 如果失败了，调用reject（reason）
                reject('失败的数据，time=' + time)
            }
        },1000)
    })
// value和reason都会返回一个新的Promise对象，这是链式调用的前提
    p.then(         //.then是同步执行的，里面的回调函数是异步回调
        value => {   //接收得到成功的数据  onResolved
            console.log('成功的回调',value)
        },
        reason =>{   //接收得到失败的数据  onRejected
            console.log('失败的回调',reason)
        }
    )

//Promsie.prototype.catch方法:(onRejected)=>{
    //onRejected函数：失败的回调函数（reason）=>{}
    //说明：then()的语法糖，相当于：then(undefined,onRejected)

}
```

3.4 为什么使用Promise

- Promise是异步编程的一种解决方案，它是一个对象，可以获取异步操作的消息，他的出现大大改善了异步编程的困境，避免了地狱回调，它比传统的解决方案回调函数和事件更合理和更强大。
- 所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

3.5 Promise的缺点:

- 无法取消Promise，一旦新建它就会立即执行，无法中途取消。
- 如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。
- 当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

3.6 Promise的链式调用

```
    // 1. 如果抛出异常，新Promise变为rejected，reason为抛出的异常值
    // 2. 如果返回的是非Promise的任意值（return 2），新Promise变为resolved，value为返回的值
    // 3. 如果返回的是另一个新Promise（return Promise.resolve(3)），此Promise的结果就会成为新Promise的结果
    const p = new Promise((resolve,reject)=>{
        resolve(1)
        //reject(2)

    }).then(
        value => {
            console.log('onResolved1()',value)  //成功输出1
            // return 2
            // return Promise.resolve(3)  //成功 -> 调用下一个then函数中的resolve
            return Promise.reject(4)  //失败 ->调用下一个then函数中的reject
            // throw 5 //失败
        },
        reason => {
            console.log('onRejected1()',reason)
        }
    )
    p.then(
        value => {
            console.log('onResolved2()',value)  // 上一个then函数的成功返回值
            // 中断Promise链，返回一个pending的promise即可
            return new Promise(()=>{})
        },
        reason => {
            console.log('onRejected2()',reason)  // 上一个then函数的失败返回值
        }
    )
```

3.7 Promise方法- then()

`then`方法返回的是一个新的Promise实例（不是原来那个Promise实例）。因此可以采用链式写法，即`then`方法后面再调用另一个then方法。

```
let promise = new Promise((resolve,reject)=>{
    ajax('first').success(function(res){
        resolve(res);
    })
})
promise.then(res=>{
    return new Promise((resovle,reject)=>{
        ajax('second').success(function(res){
            resolve(res)
        })
    })
}).then(res=>{
    return new Promise((resovle,reject)=>{
        ajax('second').success(function(res){
            resolve(res)
        })
    })
```

```
}).then(res=>{

})
```

3.8 Promise方法-catch()

Promise对象除了有then方法，还有一个catch方法，该方法相当于`then`方法的第二个参数，指向`reject`的回调函数。不过`catch`方法还有一个作用，就是在执行`resolve`回调函数时，如果出现错误，抛出异常，不会停止运行，而是进入`catch`方法中。

```
p.then((data) =>{
    console.log('resolved',data)
},(err)=>{
    console.log('rejected',err)
}
);
p.then((data)=>{
    console.log('resolved',data)
}).catch((err)=>{
    console.log('rejected',err)
})
```

3.9 Promise方法- all()

1. `all`方法可以完成并行任务， 它接收一个数组，数组的每一项都是一个`promise`对象。当数组中所有的`promise`的状态都达到`resolved`的时候，`all`方法的状态就会变成`resolved`，如果有一个状态变成了`rejected`，那么`all`方法的状态就会变成`rejected`。
2. **适用场景**：适用于处理多个异步任务，且所有的异步任务都得到结果。=》对话框中呈现两部分数据，只有当两部分数据都从后台拿到数据时，才显示两部分数据，否则，显示"数据加载中。。。"

```
let promise1 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
       resolve(1);
    },2000)
});
let promise2 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
       resolve(2);
    },1000)
});
let promise3 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
       resolve(3);
    },3000)
});
Promise.all([promise1,promise2,promise3]).then(res=>{
    console.log(res);
    //结果为：[1,2,3]
})clickFn() {
      this.alertMask = true; // 打开弹出框
      this.loading = true; // 暂时还没数据，所以就呈现loading加载中效果

      // 第一个异步任务
      function asyncOne() {
        let async1 = new Promise(async (resolve, reject) => {
          setTimeout(() => {
```

```
            // 这里我们用定时器模拟后端发请求的返回的结果，毕竟都是异步的
            let apiData1 = "第一个接口返回数据啦";
            resolve(apiData1);
          }, 800);
        });
        return async1;
      }
      console.log("异步任务一", asyncOne());  // 返回的是一个Promise对象

      // 第二个异步任务
      function asyncTwo() {
        let async2 = new Promise(async (resolve, reject) => {
          setTimeout(() => {
            let apiData2 = "第二个接口返回数据啦";
            resolve(apiData2);
          }, 700);
        });
        return async2;
      }
      console.log("异步任务二", asyncTwo()); // 返回的是一个Promise对象

      let paramsArr = [asyncOne(), asyncTwo()]

      // Promise.all方法接收的参数是一个数组，数组中的每一项是一个个的Promise对象
      // 我们在 .then方法里面可以取到 .all的结果。这个结果是一个数组，数组中的每一项
      // 对应的就是 .all数组中的每一项的请求结果返回的值
      Promise
      .all(paramsArr)
      .then((value) => {
        console.log("Promise.all方法的结果", value);
        this.loading = false; // 现在有数据了，所以就关闭loading加载中效果
      });
    },
  },
//调用all方法时的结果成功的时候是回调函数的参数也是一个数组，这个数组按顺序保存着每一个promise对象resolve执行时的值。
```

3.10 Promise方法-race()

1.`race`方法和`all`一样，接受的参数是一个每项都是`promise`的数组，但是与`all`不同的是，当最先执行完的事件执行完之后，就直接返回该`promise`对象的值。如果第一个`promise`对象状态变成`resolved`，那自身的状态变成了`resolved`；反之第一个`promise`变成`rejected`，那自身状态就会变成`rejected`。

2.race赛跑机制：当做一件事，超多一定时间就不做了，可用race。把异步操作和定时器放在一起，如果定时器先触发，认为超时，告知用户。（*点击按钮发请求，当后端的接口超过一定时间，假设超过三秒，没有返回结果，我们就提示用户请求超时*）

2.**适用场景**：

```
async clickFn() {

      // 第一个异步任务
      function asyncOne() {
        let async1 = new Promise(async (resolve, reject) => {
          setTimeout(() => {
            // 这里我们用定时器模拟后端发请求的返回的结果，毕竟都是异步的
```

```
            let apiData1 = "某个请求";
            resolve(apiData1);
          }, 4000);
        });
        return async1;
      }
      console.log("异步任务一", asyncOne());  // 返回的是pending状态的Promise对象

      // 第二个异步任务
      function asyncTwo() {
        let async2 = new Promise(async (resolve, reject) => {
          setTimeout(() => {
            let apiData2 = "超时提示";
            resolve(apiData2);
          }, 3000);
        });
        return async2;
      }
      console.log("异步任务二", asyncTwo()); // 返回的是pending状态的Promise对象

      // Promise.race接收的参数也是数组，和Promise.all类似。只不过race方法得到的结果只有一
个
      // 就是谁跑的快，结果就使用谁的值
      let paramsArr = [asyncOne(), asyncTwo()]

      Promise
      .race(paramsArr)
      .then((value) => {
        console.log("Promise.race方法的结果", value);
        if (value == "超时提示") {
          this.$message({
            type:"warning",
            message:"接口请求超时了"
          })
        }else{
          console.log('正常操作即可');
        }
      })
    }
  }
```

3.11 Promise方法-finally()

`finally`方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。`finally`方法的回调函数不接受任何参数，这意味着没有办法知道，前面的 Promise 状态到底是`fulfilled`还是`rejected`。这表明，`finally`**方法里面的操作，应该是与状态无关的，不依赖于 Promise 的执行结果。**

```
promise
.then(result => {···})
.catch(error => {···})
.finally(() => {···});
//适用场景：服务器适用Promise处理请求，然后使用finally关掉服务器
```

4.Async 和Await的理解和使用

async/await是`generator`的语法糖，它能实现的效果都能用then链来实现，它是为**优化then链**而开发出来的。从字面上来看，async是"异步"的简写，await则为等待，所以很好理解async 用于申明一个 function 是异步的，而 await 用于等待一个异步方法执行完成。当然语法上强制规定await只能出现在asnyc函数中

4.1 Async

async返回值为promise对象，promise对象的结果由async函数执行的返回值决定

4.2 Await表达式

await右侧的表达式一般为promise对象，但也可以是其他的值。（注意：await必须写在asunc函数中，但async函数中可以没有await；如果await的promise失败了，就会抛出异常，需要通过try...catch来捕获异常）

- 如果表达式是promise对象，await返回的是promise成功的值，并阻塞后续的代码；
- 如果表达式是其他值，直接将此值作为await的返回值。

```
  //2. await表达式
    function fn2() {
        return new  Promise((resolve ,reject)=>{
            setTimeout(()=>{
                resolve(5)
            },1000)
        })
    }

    function fn22(){
        return 6
    }

    async function fn3() {
        // 情况1：await右侧结果是promise对象，则返回promise成功的value
        const value = await fn2()
        console.log('value',value)
        // 情况2：  await右侧结果为非promise对象，则直接返回该值
        const value2 = await  fn22()
        console.log('value2',value2)
    }
    fn3()

// 得到失败的结果
 function fn23() {
        return new  Promise((resolve ,reject)=>{
            setTimeout(()=>{
                reject(6)
            },1000)
        })
    }

    async function fn4() {
        try{
            const value = await fn23()
            console.log('value',value)
        } catch (error) {
            console.log('得到失败的结果', error)
```

```
        }
    }
    fn4()
```

4.3 async/await的优势

- 代码读起来更加同步，Promise虽然摆脱了回调地狱，但是then的链式调用也会带来额外的阅读负担
- Promise传递中间值非常麻烦，而async/await几乎是同步的写法，非常优雅
- 错误处理友好，async/await可以用成熟的try/catch，Promise的错误捕获非常冗余
- 调试友好，Promise的调试很差，由于没有代码块，你不能在一个返回表达式的箭头函数中设置断点，如果你在一个.then代码块中使用调试器的步进(step-over)功能，调试器并不会进入后续的.then代码块，因为调试器只能跟踪同步代码的每一步

```
//例： 假设一个业务，分多个步骤完成，每个步骤都是异步的，而且依赖于上一个步骤的结果。仍然用
setTimeout 来模拟异步操作
/**
 * 传入参数 n，表示这个函数执行的时间（毫秒）
 * 执行的结果是 n + 200，这个值将用于下一步骤
 */
function takeLongTime(n) {
    return new Promise(resolve => {
        setTimeout(() => resolve(n + 200), n);
    });
}
function step1(n) {
    console.log(`step1 with ${n}`);
    return takeLongTime(n);
}
function step2(n) {
    console.log(`step2 with ${n}`);
    return takeLongTime(n);
}
function step3(n) {
    console.log(`step3 with ${n}`);
    return takeLongTime(n);
}

//Promise写法：----------------------------------------------------------------------
---
function doIt() {
    console.time("doIt");
    const time1 = 300;
    step1(time1)
        .then(time2 => step2(time2))
        .then(time3 => step3(time3))
        .then(result => {
            console.log(`result is ${result}`);
            console.timeEnd("doIt");
        });
}
doIt();
// c:\var\test>node --harmony_async_await .
// step1 with 300
// step2 with 500
// step3 with 700
// result is 900
```

```
// doIt: 1507.251ms

// async写法--------------------------------------------------------------------------------
async function doIt2(){
    console.log("doIt")
    const time1 =300
    const time2 = await step1(time1)
    const time3 = await step2(time2)
    const result = await step2(time3)
    console.log(`result is ${result}`)
    console.timeEnd("doIt")
    // 上述两个输出相同
}
doIt2()
```

4.4 async/await 捕获异常

```
async function fn(){
    try{
        let a = await Promise.reject('error')
    }catch(error){
        console.log(error)
    }
}
```

5.setTimeout、Promise、Async/Await的区别

2.1 setTimeout

```
console.log('script start') //1. 打印 script start
setTimeout(function(){
    console.log('settimeout')   // 4. 打印 settimeout
})  // 2. 调用 setTimeout 函数，并定义其完成后执行的回调函数
console.log('script end')   //3. 打印 script start
// 输出顺序：script start->script end->settimeout
```

2.2 Promise

Promise本身是**同步的立即执行函数**（Promise(excutoe)：其中， excutor会在Promise内部立即同步回调，异步操作在执行器中执行）， 当在executor中执行resolve或者reject的时候, 此时是异步操作， 会先执行then/catch等，当主栈完成后，才会去调用resolve/reject中存放的方法执行，打印p的时候，是打印的返回结果，一个Promise实例。

```
console.log('script start')
let promise1 = new Promise(function (resolve) {
    console.log('promise1')
    resolve()            //异步操作，会先执行then中的函数
    console.log('promise1 end')
}).then(function () {
    console.log('promise2')
})
setTimeout(function(){
    console.log('settimeout')
})
console.log('script end')
// 输出顺序: script start->promise1->promise1 end->script end->promise2-
>settimeout
```

当JS主线程执行到Promise对象时：

- promise1.then() 的回调就是一个 task
- promise1 是 resolved或rejected: 那这个 task 就会放入当前事件循环回合的 microtask queue
- promise1 是 pending: 这个 task 就会放入 事件循环的未来的某个(可能下一个)回合的 microtask queue 中
- setTimeout 的回调也是个 task ， 它会被放入 macrotask queue 即使是 0ms 的情况

2.3 async/await

await的含义为等待，也就是 async 函数需要**等待await后的函数执行完成并且有了返回结果**（Promise 对象）之后，才能继续执行下面的代码。await通过返回一个Promise对象来实现同步的效果。

```
async function async1(){
   console.log('async1 start');
    await async2();
    console.log('async1 end')
}
async function async2(){
    console.log('async2')
}
console.log('script start');
async1();
console.log('script end')

// 输出顺序：script start->async1 start->async2->script end->async1 end
```

6. JS中的宏队列和微队列（执行顺序）

同步代码 -> 队列中的回调函数（ 微任务 -> 宏任务）

同步任务	宏任务	微任务
直接执行的函数or程序	DOM事件回调函数	MutationObserve的回调函数
事件的回调函数	ajax回调函数	Promise的回调函数（5个方法）
setTimeout和setInterval()，I/O操作等	定时器回调函数	Object.observe

执行顺序：

1. JS引擎先执行所有的初始化**同步**代码
2. 每次准备取出第一个宏任务执行前，都要将所有的微任务一个一个取出来执行

重点例题看webstorm上的面试题！必考，一定要搞懂

```
setTimeout(()=>{   //立即放入宏队列
        console.log('timeout callback1()')
        Promise.resolve(3).then(
            value => {   //会立即放入微队列中
                console.log('Promise onResolved3()',value)
            }
        )
    },0)
    setTimeout(()=>{
        console.log('timeout callback2()')
    },0)
    Promise.resolve(1).then(
        value => {   //会立即放入微队列中
            console.log('Promise onResolved1()',value)
        }
    )
    Promise.resolve(2).then(
        value => {
            console.log('Promise onResolved2()',value)
        }
    )

    //输出顺序：Promise onResolved1()-》Promise onResolved2()
    // -》timeout callback1() =》Promise onResolved3()
    // -》timeout callback2()
```

7.常见的定时器（setTimeout、setInterval、requestAnimationFrame）

7.1 setTimeout

setTimeout()用于在指定在一定时间后执行某些代码，setTimeout第一个参数表示执行内容，第二个参数是执行回调函数前等待的时间（毫秒）。因为JS是单线程，只维护了一个任务队列，**第二个参数告诉JS引擎在指定的毫秒数才将任务添加到队列中**。如果队列是空的，则立即执行该代码；如果队列不是空的，则等到前面任务执行完毕后才能执行。所以，setTimeout并不能表示代码在延期XX秒后一定会执行

可以使用clearTimeout方法取消超时任务

```
//调用setTimeout时，会返回一个表示该超时排期的数值ID，是执行代码的唯一标识符
setTimeID = setTimeour(()=>
    (console.log("test"),
     1000)

clearTimout(setTimeID)
```

7.2 setInterval

setInterval表示每隔一段时间执行一次回调函数，直到取消或页面卸载。浏览器不管回调函数函数什么时候执行或者执行多久，只要一到指定时间，便会向队列中添加一个任务，所以，执行时间短、非阻塞的回调函数适合setInterval。取消循环时，可以使用clearInterval方法。

通常来说不建议使用 `setInterval`，因为一个任务和下一个任务之间的时间间隔是无法保证的，有些循环定时任务可能因此会跳过。还存在执行积累的问题

```
//以上代码在浏览器环境中，如果定时器执行过程中出现了耗时操作，多个回调函数会在耗时操作结束以后同时执行，这样可能就会带来性能上的问题。
function demo() {
  setInterval(function(){
    console.log(2)
  },1000)
  sleep(2000)
}
demo()
```

7.3 requestAnimationFrame

HTML新增的 `requestAnimationFrame` 自带函数节流功能，不需要设置时间间隔

引入：

- 计时器一直是javascript动画的核心技术。而编写动画循环的关键是要知道延迟时间多长合适。一方面，循环间隔必须足够短，这样才能让不同的动画效果显得平滑流畅；另一方面，循环间隔还要足够长，这样才能确保浏览器有能力渲染产生的变化。
- 大多数电脑显示器的刷新频率是60Hz，大概相当于每秒钟重绘60次。大多数浏览器都会对重绘操作加以限制，不超过显示器的重绘频率，因为即使超过那个频率用户体验也不会有提升。因此，最平滑动画的最佳循环间隔是1000ms/60，约等于16.6ms
- 而setTimeout和setInterval的问题是，它们都不精确。它们的内在运行机制决定了时间间隔参数实际上只是指定了把动画代码添加到浏览器UI线程队列中以等待执行的时间。如果队列前面已经加入了其他任务，那动画代码就要等前面的任务完成后再执行。
- requestAnimationFrame采用系统时间间隔，保持最佳绘制效率，不会因为间隔时间过短，造成过度绘制，增加开销；也不会因为间隔时间太长，使用动画卡顿不流畅，让各种网页动画效果能够有一个统一的刷新机制，从而节省系统资源，提高系统性能，改善视觉效果。

特点：

1. requestAnimationFrame会把每一帧中的所有DOM操作集中起来，在一次重绘或回流中就完成，并且重绘或回流的时间间隔紧紧跟随浏览器的刷新频率
2. 在隐藏或不可见的元素中，requestAnimationFrame将不会进行重绘或回流，这当然就意味着更少的CPU、GPU和内存使用量
3. requestAnimationFrame是由浏览器专门为动画提供的API，在运行时浏览器会自动优化方法的调用，并且如果页面不是激活状态下的话，动画会自动暂停，有效节省了CPU开销

使用：

requestAnimationFrame的用法与settimeout很相似，只是不需要设置时间间隔而已。
requestAnimationFrame使用一个回调函数作为参数，这个回调函数会在浏览器重绘之前调用。它返回一个整数，表示定时器的编号，这个值可以传递给cancelAnimationFrame用于取消这个函数的执行

```
const requestID = requestAnimationFrame(callback)
const timer = requestAnimationFrame(functionn(){
      console.log(0)
})
console.log(timer)  //1

//取消该定时器
cancelAnimationFrame(timer);
cancelAnimationFrame(1)
```

十、面向对象

1. 数据属性

数据属性包含一个保存数据值的位置。值会从这个位置读取，也会写入到这个位置。数据属性共有4个特性（通过Object.definePrototype修改）：

1. [[Configurable]]:表示数据是否可以通过delete删除并重新定义，是否可以修改它的特性以及是否可以将其改为访问器属性。默认情况下，为true。若定义该属性为false，则表示该属性不能从对象上删除
2. [[Enumberable]] : 表示属性是否可以通过for..in循环。默认情况下为true
3. [[Writable]]: 表示属性的值是否可以被修改，默认情况下为true。若设置为只读模式，则在严格模式下，修改该位置的属性会报错，非严格模式下则会忽略后续的赋值
4. [[Value]]: 属性实际的值，默认为undefined

```
let person ={
    name:'SFONE'  //这里设置value为SFONE。 在严格模式下，尝试修改
}
```

2.访问器属性

访问器属性不包含数据属性，而是包含getter、setter函数。在读取访问器属性时，会调用getter函数，并返回一个有效的值，写入时，会调用setter函数并传入新值，共有4个属性：

1. [[Configurable]]: 表示属性是否通过delete重新删除或定义，与上述相似
2. [[Enumberable]]: 表示属性是否可以通过for..in循环。默认情况下为true
3. [[get]]: 获取函数，默认为undefined
4. [[set]]: 设置函数，默认为undefined

```
let person ={
    name:'SFONE'  //这里设置value为SFONE。 在严格模式下，尝试修改
    age:'2018'
}
Objec.definePrototype(person,"age",{
    get(){
        return this.name
    },
    set(newvalue){
        if(age>2017){
            this.age = newvalue +1
        }
    }
})
```

3.创建对象的方法有哪些?

ES5并没有正式支持面向对象的结构，比如类和继承，但是使用原型链可以模拟同样的行为。ES6正式支持类和继承。一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象的时候，会产生大量的重复代码。主要方式有：

3.1 工厂模式

工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。

```
function createPerson(name,age,job){
    let o = new Object();   // 显示的创建了对象
    o.name = name
    o.age = age;
    o.job = job;
    o.sayNmae = function(){
        console.log( this.name)
    };
    return o;
}

let person1 = createPerson('SFONE','24','Software Engineer')
let person2 = createPerson('Grace','28','Software Engineer')
// 这种工厂模式虽然解决创建多个类似对象的问题，但是没有解决对象标识问题
```

3.2 构造函数模式

构造函数模式相对于工厂模式的**优点**是，所创建的对象和构造函数建立起了联系，因此可以通过**原型**来识别对象的类型。但是构造函数存在一个**缺点**就是，造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

与构造函数的区别：

1. 没有显示的创建对象
2. 属性和方法直接赋值给了this
3. 没有return

```
function Person(name,age,job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = function(){
        console.log(this.name)
    };
}

let person1 = new Person('SFONE','24','Software Engineer')
let person2 = new Person('Grace','28','Software Engineer')

person1.sayName()  //SFONE
person2.sayName()  //Grace
```

3.3 原型模式

因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如 Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例

```
function Person(){
    Person.prototype.name = 'SFONE',
    Person.prototype.age = 29,
    Prson.prototype.sayName =  function(){
        console.log(this.name)
    }
}

//等价于
function Person(){}
Person.prototype.name = 'SFONE',
Person.prototype.age = 29,
Prson.prototype.sayName =  function(){
     console.log(this.name)
}

let person1 = new Person()
person1.sayName()

let person2= new Person()
person2.sayName()   //可以调用主要是由于对象查找机制，链式向上查找

//1：原型和方法都添加到了Person的Prototype属性上（指向原型对象）。
//2：默认情况下，所有原型对象自动获得一个constructor属性，指向与之关联的构造函数（Person.prototype.constructor 指向Person）
//3：  自定义构造函数时，原型对象默认获得constructor属性，其他所有属性来源于Object。每次调用构造函数创建一个实例时，实例内部的[[prototype]]指针被赋值为构造函数原型对象。脚本中不能访问[[prototype]]，但chrome，firefox，Safari中会暴露_proto_属性，通过这个属性可以访问对象原型。
```

3.4 组合使用构造函数和原型模式

3.5 动态原型模式

3.6 寄生构造函数

4.对象继承的方式

4.1 原型链

这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

```
function Father(){
      this.FatherName = "father's 构造函数";
 }
Father.prototype.age = 40;
function Son(){
    this.SonNname = "Son's 构造函数";
}
//Son的原型继承Father创建出来的对象，相当于继承了Father的全部内容，同时全部都存在Son__proto__属性里
Son.prototype = new Father();
Son.prototype.getSubValue = false;
Son.prototype.age1 = 20;
var example =new Son();
console.log(example.age);
```

4.2 构造函数

使用call()和apply()将父类构造函数引入到子类函数,使用父类的构造函数来增强子类的实例,等同于复制父类的实例给子类, 所以子类的任何操作都不会影响到父类。

这种方式是通过在子类型的函数中调用父类型的构造函数来实现的，这一种方法**解决了不能向父类型传递参数的缺点**，但是它存在的一个问题就是无法实现函数方法的复用，并且父类型原型定义的方法子类型也没有办法访问到。

```
 function Father1(name){
        this.name = name;
        this.colors = ["red"];
 }
    Father1.prototype.sex = 1;
 function Son1(name,age){
      Father1.call(this,name);
      //利用call改变this的指向将this指向父类中的name，这样就把Father中方法给引入到子类中了
        this.age = age;
 }
var obj1 = new Son1("张三",25);
obj1.colors.push("write");   //引用类型的也不会改变父类中的内容
console.log(obj1.colors);// [red write]

var obj2= new Son1("jike",10);
console.log(obj2.colors);//[red]
console.log(obj2.sex);//undefined 无法继承到父类原型中的内容
```

4.3 组合继承

将原型链继承和构造函数继承这两种模式的有点组合在一起,通过调用父类构造,继承父类的属性并保留传参,然后通过将父类实例作为子类原型,实现函数复用，解决了上面的两种模式单独使用时的问题

缺点:

父类中的实例属性,方法即存在子类的实例中,也存在于子类的原型中,所以调用了两次超类的构造函数,造成了子类型的原型中多了很多不必要的属性，占内存

```
function Father3(name){
      this.name = name;
      this.colors = ["red", "blue", "green"];
}

Father3.prototype.sayName=function(){
      console.log(this.name);
}

function Son3(name,age){
    //构造函数继承
    Father3.call(this,name);   //继承this.name = name
    this.age = age;
 }

//原型链继承
Son3.prototype = new Father3();  //son3现在有name,color,sayname,重复继承了name
var obj = new Son3("jike",25);
console.log(obj);
```

```
obj.sayName();//jike 可以调用父类的原型上的方法
// obj.colors.push("write");

var obj3 = new Son3("mike",30);
obj3.colors.push("write");
console.log(obj3.colors);//["red", "blue", "green", "write"]  不会修改原型中的color数组
console.log(obj.colors);// ["red", "blue", "green"]
```

4.4 原型式继承

原型式继承的主要思路就是基于已有的对象来创建新的对象。 这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承

重点:用一个函数包装一个对象,然后返回这个函数的调用,这个函数就变成了可以随意增添属性的实力或者对象. ES5中将object包装为object.create()方法

```
function object(o) {   //封装一个函数，用来输出对象和承载继承的原型
 function F() {}   //创建临时性构造函数，将传入的对象o作为这个构造函数的原型
 F.prototype = o   //本质上，是对传入对象o执行了一次浅复制
 return new F()   //最后返回这个临时类型的一个新实例
}

var person = {
 name: 'Gaosirs',
 friends: ['Shelby', 'Court']
}
var anotherPerson = object(person)
console.log(anotherPerson.friends) // ['Shelby', 'Court']
//和原型链继承一样,同样是将自身的对象原型继承一个完整的对象,也可以说是将一个对象作为自身的原型对象,对于引用类型的数据一样继承的是引用地址,那么一个子实例对引用类型的数据进行操作的话,所有实例都会受到影响.
```

4.5 寄生式继承

寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，复制一个对象的副本，然后对象进行扩展，最后返回这个对象。这个扩展的过程就可以理解是一种继承,和工厂模式类似

这种继承的优点就是一个简单对象实现继承，如果这个对象不是自定义类型时。缺点是没有办法实现函数的复用，效率降低

```
function createAnother(o) {  //封装继承过程的函数
 var clone = Object.create(o) // 创建一个新对象
 clone.sayHi = function() { // 添加方法
 console.log('hi')
 }
 return clone // 返回这个对象
}
var person = {
 name: 'GaoSirs'
}
var anotherPeson = createAnother(person)
anotherPeson.sayHi()
```

4.6 寄生式组合继承

在前面说的组合模式(原型链+构造函数)中，继承的时候需要调用两次父类构造函数，导致添加了不必要的原型属性。使用寄生组合式继承，可以规避这些问题。这种模式通过借用构造函数来继承属性，通过原型链的形式来继承方法。本质上就是使用寄生式继承来继承父类的原型，在将结果指定给子类型的原型。

优点：只调用了一次 supertype 构造函数，因此避免在subtype.prototype上创建不必要的，多余的属性，与此同时，原型链还能保持不变，还能正常使用instanceof 和isPrototypeOf()，因此，寄生组合式继承被认为是引用类型最理想的继承范式

```
//父类
function SuperType(name) {
 this.name = name
 this.colors = ['red', 'blue', 'green']
}
//给父类原型添加方法sayName
SuperType.prototype.sayName = function () {
 console.log(this.name)
}

function SubType(name, job) {
 // 继承属性
 SuperType.call(this, name)
 this.job = job
}

//继承函数
function inheritPrototype(subType, superType) {
 var prototype = Object.create(superType.prototype);  // 创建父类原型副本
 prototype.constructor = subType;                     // 给创建副本添加constructor属性
 subType.prototype = prototype;                       // 将子类原型指向这个副本
}

// 继承
inheritPrototype(SubType, SuperType)
var instance = new SubType('Gaosirs', 'student')
instance.sayName()
```

十一、垃圾回收与内存泄漏

1.浏览器的垃圾回收机制

垃圾回收：JavaScript代码运行时，需要分配内存空间来储存变量和值。当变量不在参与运行时，就需要系统收回被占用的内存空间，这就是垃圾回收。

回收机制：

- Javascript 具有自动垃圾回收机制，会定期对那些不再使用的变量、对象所占用的内存进行释放，原理就是找到不再使用的变量，然后释放掉其占用的内存。
- JavaScript中存在两种变量：局部变量和全局变量。全局变量的生命周期会持续要页面卸载；而局部变量声明在函数中，它的生命周期从函数执行开始，直到函数执行结束，在这个过程中，局部变量会在堆或栈中存储它们的值，当函数执行结束后，这些局部变量不再被使用，它们所占有的空间就会被释放。

- 不过，当局部变量被外部函数使用时，其中一种情况就是闭包，在函数执行结束后，函数外部的变量依然指向函数内部的局部变量，此时局部变量依然在被使用，所以不会回收。

垃圾回收的方式

浏览器通常使用的垃圾回收方法有两种：标记清除，引用计数。

1）标记清除

- 标记清除是浏览器常见的垃圾回收方式，当变量进入执行环境时，就标记这个变量“进入环境”，被标记为“进入环境”的变量是不能被回收的，因为他们正在被使用。当变量离开环境时，就会被标记为“离开环境”，被标记为“离开环境”的变量会被内存释放。
- 垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量以及被环境中的变量引用的标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后。垃圾收集器完成内存清除工作，销毁那些带标记的值，并回收他们所占用的内存空间。

2）引用计数

- 另外一种垃圾回收机制就是引用计数，这个用的相对较少。引用计数就是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数就减1。当这个引用次数变为0时，说明这个变量已经没有价值，因此，在在机回收期下次再运行时，这个变量所占有的内存空间就会被释放出来。
- 这种方法会引起**循环引用**的问题：例如：`obj1`和`obj2`通过属性进行相互引用，两个对象的引用次数都是2。当使用循环计数时，由于函数执行完后，两个对象都离开作用域，函数执行结束，`obj1`和`obj2`还将会继续存在，因此它们的引用次数永远不会是0，就会引起循环引用。

2.减少垃圾回收

虽然浏览器可以进行垃圾自动回收，但是当代码比较复杂时，垃圾回收所带来的代价比较大，所以应该尽量减少垃圾回收。

- **对数组进行优化**：在清空一个数组时，最简单的方法就是给其赋值为[]，但是与此同时会创建一个新的空对象，可以将数组的长度设置为0，以此来达到清空数组的目的。
- **对`object`进行优化**：对象尽量复用，对于不再使用的对象，就将其设置为null，尽快被回收。
- **对函数进行优化**：在循环中的函数表达式，如果可以复用，尽量放在函数的外面。

3.哪些情况会导致内存泄漏

以下四种情况会造成内存的泄漏：

- **意外的全局变量**：由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。
- **被遗忘的计时器或回调函数**：设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。
- **脱离 DOM 的引用**：获取一个 DOM 元素的引用，而后面这个元素被删除，由于一直保留了对这个元素的引用，所以它也无法被回收。
- **闭包**：不合理的使用闭包，从而导致某些变量一直被留在内存当中。

十二、其他重要的问题

js手写防抖函数，异步

JavaScript NEW一个对象的过程

this相关知识点

立即执行函数知识点

window 对象含有 location 对象、navigator 对象、screen 对象等子对象

MVC,MVP,MVVM

关注分离（Separation of Concerns）的原则

Modern.js