全文 !!!! 表示必考必记，!!! 表示必记，!! 表示理解，！表示 了解

一、数据类型

1.Javascript 的数据类型都有哪些 !!!!

> 基本值类型：undefined、null、number、string、boolean、symbol(独一无二且不可变的数据类型)、bigint(可以表示任意精度格式的整数,可以安全的存储和操作大整数)
> 引用类型：Object、Array、Function。

这些数据可以分为原始数据类型和引用数据类型：

- 栈(原始数据类型):占据空间小、大小固定，属于被频繁使用的数据。
- 堆(引用数据类型):占据空间大、大小不固定，会在栈中存储一个指针,指向堆中该数据的起始位置。

2.数据类型检测方式 !!!!

- typeof
    - typeof [] => object
    - typeof null => object

  typeof 不能用于检测 null 和数组
- instanceof
    - 1 instanceof Number => false
    - true instanceof Boolean => false
    - 'str' instanceof String => false

  instanceof 只能用于判断引用数据类型,主要原理是检测一个对象的原型是否存在于一个构造函数的 prototype 属性
- constructor

  不能用于判断 undefined、null。需要注意，如果一个对象的原型被改变了，它的 constructor 属性也会改变。
- Object.prototype.toString.call

所有数据类型均能检测返回值是一个 [object 类型] 格式的字符串。

注意：obj.toString() 的 toString 可能被重写了,所以需要调用 Object 原型上的 toString 方法。

```
[1,2,3].toString() => 1,2,3; ({a:1}).toString() => [object Object];
(function(){return{a:1}}).toString() => function(){return{a:1}}
```

3.判断数组的方式 !!!!

- Object.prototype.toString.call
- Object.getPrototypeOf(obj) === Array.prototype
- Array.isArray(obj)
- obj instanceof Array
- Array.prototype.isPrototypeOf(obj),检测 Array.prototype 是否在 obj 的原型链上

4.null 和 undefined 区别 !!!!

- undefined 的含义是未定义，一般变量声明了但还没有定义时它的值为 undefined;
- null 的含义是空对象，主要用于赋值给一些可能会赋值为对象的变量，作为初始化的值。
- undefined 在 javascript 中不是一个标识符,可以用 undefined 作为变量名但非常危险不推荐用这种做法。
- 用双等号比较 undefined 和 null 会返回 true,用严格相等号比较会返回 false

5.typeof null 的结果及原因 !!!

object,因为 javascript 在第一个版本时，所有值都存储在 32 位的单元中，每个单元的低 3 位都表示类型标签，其他位表示真实数据。
而对象的类型标签为 000，null 的值是机器码 null 指针也就是全 0，所以 null 的类型标签也是 000 和对象的类型标签相同，所以会被判定为 object。
注意：undefined 的值是(-2)^30,超出整数范围的数字。

6.为什么 0.1+0.2 !== 0.3,如何让其相等 !!!!

为什么 0.1+0.2 !== 0.3,如何让其相等 !!!!

- 原因
    - 一个原因是浮点数进制转换,js 在做数字运算时,0.1 和 0.2 会被转为无限循环的二进制数,但是 js 采用的 IEEE754 二进制浮点运算最大只能存储 53 位的有效数字,于是大于 53 后面的会被全部截掉,导致精度损失。

- 另一个原因是对接运算，由于指数位数不相同，运算时需要对阶运算，阶小的位数要根据阶差右移，位数位移时可能会发生数丢失的情况，影响精度。

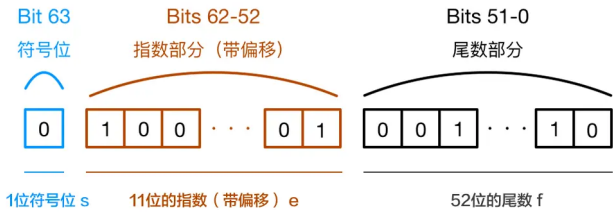
总结：简单来说因为进制转换和对阶运算过程中会出现精度损失所以就会出现偏差。

- 解决方法

  将浮点数转为整数后运算

  通过 (0.1+0.2).toFixed(2) 将其四舍五入为指定小数位数的数字；

- IEEE754 补充
  第 0 位：符号位，0 表示正数，1 表示负数(s)
  第 1 位到第 11 位：储存指数部分（e）
  第 12 位到第 63 位：储存小数部分（即有效数字）f



注意：Number.MAX_SAFE_INTEGER == Math.pow(2,53) - 1 因为二进制表示有效数字的形式总是为 1.xx 的形式，尾数部分默认第一位为 1 省略不写,所以 js 提供的有效数字最长为 53 个二进制位。

7.如何获取安全的 undefined 值 !!

```
undefined 不是一个标识符，可以被当作变量来使用和赋值，但是这样会影响对 undefined 的正常判断。
可以通过 void 0 获取 undefined,因为 void 不能返回值,因此返回结果为 undefined。
```

8.typeof NaN 的结果 !!

```
number, NaN 与自身不严格相等,用于表示执行的数学运算不成功，这是失败的返回结果。
```

9.isNaN 和 Number.isNaN 函数的区别 !!!

```
比较的逻辑不同, Number.isNaN 会先比较传入的参数类型,是 number 类型再判断是否为 NaN;
而 isNaN 接收参数后会将参数转换为数值，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。(不能转为number的类型可以看隐式转换)
```

10. == 操作符的强制类型转换规则 !!!!

1. 判断 x、y 是否为正常值，不是中断执行
2. 类型相同执行严格相等运算
3. 判断是否在比较 undefined 和 null,是返回 true
4. 判断是否在比较 string 和 number,是将字符串转为 number
5. 判断是否有一方为 boolean ,是的话将 boolean 转为 number 再比较
6. 判断是否一方为对象(包括数组),且另一方为 string、number 或 Symbol,是的话将 object 转为与目标相同的原始数据类型再比较。
   '1' == { name: 'js' } => '1' == '[object Object]'; // false
   Symbol({}) == {} => Symbol({}) == Symbol({}) // false
7. 返回 false

简写 == 类型转换规则：

1. 判断 x、y 是否为正常值，不是中断执行
2. 类型相同执行严格相等运算
3. x,y 不是对象执行 Number(x) == Number(y)
4. x 或 y 是对象 toPrimitive(x|y, type(x|y)) == x|y
5. false

11.其他值到 string 的转换规则 !!!

- null 或 undefined 会直接转为 null 和 undefined 字符串

  null->'null',undefined->'undefined'
- Boolean 类型,true、false 会直接转为字符串

  true->'true'
- Number 类型的值直接转换，不过极大和极小的数字会采用指数形式再转为字符串

  123->'123';
- Symbol 类型的值可以直接转换，但仅支持显示强制类型转换，使用隐式强制类型转换会报错

  Symbol(0).toString() -> 'Symbol(0)';String(Symbol(0));
  Symbol(0)+" // 隐式转换 error
- BigInt 类型的值可以直接转换，只是转换后的值没有代表是 BigInt 类型的尾标识 n
  BigInt(123).toString() -> 123;
- 对象类型除非重写了 toString ,否则会变为[object Object]

  比如，数组类型转换为字符串会将数组的内容直接转为字符串,如果内容中存在对象类型也会转为[object Object] 因为内部数据调用的是该数据的 toString。

  [1,2,{a:1}] -> '1,2,[object Object]'; []->'';

12.其他值到 number 的转换规则 !!!

- undefined -> NaN
- null -> 0
- true->1; false->0;
- string 类型的值如果包含字非数值的字符会转为 NaN，'' 为 0
- Symbol 不能转为 number , 会报错 显示隐式都相同
- Number(BigInt(123)) -> 123 显示强制类型转换

  +BigInt(123) 隐式强制类型转换报错
- 对象类型的数据转换见 14.Javascript 中如何进行隐式类型转换

13.其他值到布尔类型的转换规则 !!!

```
NaN、undefined、null、0、'' 会转为 false;
其他为 true
```

14.Javascript 中如何进行隐式类型转换 !!!!

Javascript 中调用 [[ToPrimitive]] 方法将值转化为基本类型值。

基本结构如下：

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

ToPrimitive 操作规则：

1. 目标类型 为 number
    1. 调用 valueOf ,值为原始值则返回
    2. 调用 toString, 值为原始值则返回
    3. 抛出错误
2. 目标类型 为 string
    1. 调用 toString, 值为原始值则返回
    2. 调用 valueOf ,值为原始值则返回
    3. 抛出错误

主要区别在于 toString 和 valueOf 调用顺序

- 如果需要转换的对象为 Date 对象，则目标类型默认为 string;
- 其他情况下，目标类型默认为 number;

隐式类型转换主要发生在 `+、-、*、/ 及 ==、>、<` 这些运算符之间。

而这些运算符只能操作基本类型值，所以在进行这些运算前的第一步就是将两边的值用 [[ToPrimitive]] 转换成基本类型，再进行操作。

- + 操作符
    - 有一方为 string 另一方也转为 string(字符串拼接)
    - 有一方为 number,另一方为原始值,另一方转为 number(加法)
    - 有一方为 number,另一方为引用类型,另一方转为 string,转换完后再重新执行流程 1-3

- -、*、\ 操作符

```
1 * "23"; // 23
1 * false; // 0
1 / "aa"; // NaN
```

- == 操作符
    - 判断 x、y 是否为正常值，不是中断执行
    - 类型相同执行严格相等运算
    - x,y 不是对象执行 Number(x) == Number(y)
    - x 或 y 是对象 toPrimitive(x|y, type(x|y)) == x|y

- < 或 >
    - 两边都是字符串，比较字典序
    - 其他情况,转为 number 比较

```
var a = {};
a > 2; // false,需要注意这和直接用 {} > 2 是不同的!!!!!!
// {}, ToPrimitive 默认 type 为 number，所以先 valueOf，结果还是个对象，下一步
a.valueOf();
// "[object Object]"，现在是一个字符串了
a.toString();
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

var a = { name: "Jack" };
var b = { age: 18 };
a + b; // "[object Object][object Object]"
a.valueOf(); // { name: "Jack" }，上面提到过，ToPrimitive默认type为number，所以先valueOf，结果还是个对象，下一步
a.toString(); // "[object Object]"
b.valueOf(); // 同理
b.toString(); // "[object Object]"
a + b; // "[object Object][object Object]"
```

总结：

如果某个操作数是字符串或者能够通过以下步骤转换为字符串的话，+ 将进行拼接操作。

- 其中一个操作数是对象（包括数组），则首先对其调用 ToPrimitive 操作
- 该操作再调用 [[DefaultValue]]，以数字作为上下文。
- 如果不能转换为字符串，则会将其转换为数字类型来进行计算。

如果 + 的其中一个操作数是字符串（或者通过以上步骤最终得到字符串），则执行字符串拼接，否则执行数字加法。

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。

15.|| 和 && 操作符的返回值 !!

- 两者都会先对第一个操作数执行条件进行判断，如果不是 boolean 值就转为 boolean 再判断；
- 对于 || 如果第一个操作数执行结果为 true 就返回第一个操作数的值，执行结果为 false 返回第二个操作数的值；
- 对于 && 如果为第一个操作数执行结果为 true 就返回第二个操作数的值，执行结果为 false 就返回第一个操作数的值。
- || 和 && 返回他们其中一个**操作数的执行结果**，而非条件判断结果

16.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，但处理了一些特殊的情况，如 -0 和 +0 不再相等，两个 NaN 是相等的。

17.什么是 Javascript 包装类型 !

- javascript 中的基本类型没有属性和方法，为了方便操作基本类型的值；
- 在调用基本类型的属性和方法时 javascript 会在后台隐式地将基本类型的值转为对象。
- 也可以使用 Object 显示地将**基本类型转为包装类型**(undefined、null 会转为空对象)。
- 也可以使用 valueOf 将**包装类型倒转成基本类型**(undefined、null 会转为空对象)。
- 包装类型就是对象，它的各种转换遵循对象的转换规则。

18.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。

总结：解决超过最大安全数字的数,因超过范围导致精度丢失需要借助第三方库的问题

19.object.assign 与 扩展运算符 !!

- 均为浅拷贝
- 均不复制继承的属性或类的属性，均会复制 Symbol
- 均不可赋值不可迭代的属性,enumerable 为 false 的属性

20.JS 内存管理机制

21. JS 基本数据类型和引用数据类型在内存中存储方式

大多数说法，基本数据类型如 number、string、boolean 存放在栈中，而引用数据类型存放在堆中。

实际上 Javascript 底层是用 c 实现的，c 语言处理字符串通过 char[] 数组实现，因此 string 是不可能存放在栈中的而是存放在堆中，之所以 `a='a',b='a',a===b` 的原因是因为 js 会把相同的字符串存放在同一内存地址中，也就是只在堆中存储一次，之后如果某个变量赋值了这个字符串就会在该变量指向的栈中位置存储这个字符串的起始地址。

而对于一些大整数，也是存储在堆中的。因为我们知道，栈中能存放的数据的位数是有限的，一般为存放一个地址的最大位数。因此如果这个数在这个最大位数的范围内能够表示，那它就能够直接存储在栈中。

二、Javascript 基础

1.new 操作符原理(手写 new) !!!!

见手写篇

```
function MyNew(constructor, ...rest) {
  if (typeof constructor !== "function") throw new
TypeError("err");
  // 注意，是创建继承于 (constructor 原型) 的对象
  let obj = Object.create(constructor.prototype);
  let res = constructor.call(obj, ...rest);
  if (res && (typeof res === "object" || typeof res ===
"function")) {
    return res;
  }
  return obj;
}
```

2.map 和 Object 的区别 !!!

- Object 的键只能为字符串或 Symbol，而 map 的键可以为任何类型。
- Object 中迭代的顺序是排序后的数值属性,然后其他属性按插入顺序迭代，而 map 迭代的顺序与插入顺序相同
- map 有定义 iterator 接口是可迭代的，而 Object 需要使用一些特殊方式才能迭代
- map 的键值对个数可以通过 size 属性获取，而 Object 的键值对个数只能通过一些方法处理后再 通过 length 属性获取
- map 在频繁的操作键值对的场景下性能更好

3.map 和 weakMap 的区别 !!!

- map 的键可以为任何类型， weakMap 的键只能为对象。
- weakmap 的键是弱引用，而 map 的键是强引用；
- 为 map 创建一个类型为对象的键，通过手动将该对象内存进行释放，但是因为 map 仍然和这个对象存在强引用关系，所以 map 中这部分内存依然无法释放。
- 但如果是为 weakmap 进行同样的操作时，因为 weakmap 和键之间是弱引用关系，当下一次垃圾回收
  机制执行时，这块内存就会被释放。
- 当需要拷贝一个很大的对象时，使用 map 会对内存造成非常大的消耗，而且需要通过手动清除 map 上的相应键才能释放这块内存，而 weakmap 的机制可以化解这个问题。

补充：

> 弱引用的概念，弱引用与强引用相对，弱引用指的是其引用的对象可能在任何时刻被垃圾回收机制回收。

4.Javascript 的内置对象 !

- 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

- 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
- 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
- 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
- 字符串，用来表示和操作字符串的对象。例如 String、RegExp
- 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
- 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
- 反射。例如 Reflect、Proxy
- 控制抽象对象 例如 Promise、Generator 等

5.常用的正则表达式有哪些！

手机号、QQ 号、邮箱、中文、16 进制颜色值、匹配日期，如 yyyy-mm-dd 格式

6.对 JSON 的理解 !!

- 作为前后端数据交换的方法
- JSON 全称是(JavaScript Object Notation),语法基于 js ,但 JSON 的语法更严格。

  比如说：json 对象的属性名任何时候都必须加双引号、json 对象没有末尾的分号、JSON 不支持变量、函数和对象实例,且 json 文件中不能添加注释

包括两个内置方法

- JSON.stringify 函数, 传入一个符合 JSON 格式的数据结构，将其转化为一个 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。
- JSON.parse 函数, 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。

7.延迟 Javascript 脚本执行的方式 !!!

- 将 js 脚本防止文档的底部，使 js 脚本尽可能在最后执行
- 用外链式引入 js 脚本并且设置标签的 async 属性或 defer 属性
- 动态创建 DOM 标签，对文档的 load 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，设定一个定时器延时加载 js 脚本

8.Javascript 类数组对象的定义 !!!

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。

常见的类数组对象有 arguments、 获取 DOM 元素的返回结果，函数也可以被看作是类数组对象，因为它含有 length 属性值

常见的类数组转换为数组的方法：

- Array.prototype.slice.call(arrayLike);
- Array.prototype.splice.call(arrayLike, 0);
- Array.prototype.concat.apply([], arrayLike);
- Array.from(arrayLike);
- [...arrayLike] (类数组需要配置 iterator 接口)

9.数组的原生方法 !!!!

toString、join
push、pop、shift、unshift、slice、splice、split、concat、reverse、sort
forEach、reduce、reduceRight、some、every、map、filter、indexOf、includes、
fill、find、findIndex(这两个都能找到 NaN)

10.Unicode、UTF-8、UTF-16、UTF-32 的区别 !

Unicode、UTF-8、UTF-16、UTF-32 的区别

11.常见的位运算符有哪些？其计算规则是什么 !!

|、&、^、~(符号位也会取反)、>>、<<

>>1 除 2，<<1 乘 2

12.js 的运算符优先级 !!!

优先级	运算类型	关联性	运算符
20	圆括号	n/a（不相关）	`( … )`
19	成员访问	从左到右	`… . …`
	需计算的成员访问	从左到右	`… [ … ]`
	`new` (带参数列表)	n/a	`new … ( … )`
	函数调用	从左到右	`… ( … )`
	可选链（Optional chaining）	从左到右	`?.`
18	new (无参数列表)	从右到左	`new …`
17	后置递增(运算符在后)	n/a	`… ++`
	后置递减(运算符在后)		`… --`
16	逻辑非	从右到左	`! …`
	按位非		`~ …`
	一元加法		`+ …`
	一元减法		`- …`
	前置递增		`++ …`
	前置递减		`-- …`
	typeof		`typeof …`
	void		`void …`
	delete		`delete …`
	await		`await …`

13.为什么函数的 arguments 参数是类数组而不是数组？如何遍历类数组? !!!

arguments 实质是一个对象,并且包括了处本身的值和 length 以外的属性,如 callee、Symbol.iterator。
Symbol.iterator 是 javascript 中内置的属性用户配置迭代器。
可以将类数组转换为普通数组后遍历

- Array.prototype.slice.call(arrayLike);
- Array.prototype.splice.call(arrayLike, 0);
- Array.prototype.concat.apply([], arrayLike);
- Array.from(arrayLike);
- [...arrayLike] (类数组需要配置 iterator 接口)

14.什么是 DOM 和 BOM !!!!

- DOM 指的是文档对象模型，把整个文档当做一个对象，这个对象提供了处理网页内容的接口。
- BOM 指的是浏览器对象模型，定义了与浏览器进行交互的接口，本质是用 Javascript 开发 Web 应用程序的核心。

15.escape、encodeURI、encodeURIComponent 的区别 !!

- 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 中有特殊含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转义。
- escape 和 encodeURI 的作用相同，不过它们对于 unicode 编码为 0xff 之外的字符会有区别，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 %。

16.对 AJAX 的理解，实现一个 AJAX 请求(手写 ajax) !!!!

```
// 简易版
function sendAJAX(url) {
  let xhr = new XMLHttpRequest();
  xhr.open('GET', url);
  xhr.setRequestHeader('Content-Type',
'application/json');
  xhr.onreadystatechange = function(res) {
    if(xhr.readystate === 4)  return ;
    if(xhr.status === 200) {
      console.log(res.responseText)
    } else {
      console.error(res.responseText)
    }
  });
  // 设置请求失败时的监听函数
  xhr.onerror = function() {
    console.error(this.statusText);
  }
  xhr.send();
}
// promise 版
function sendAJAX(url) {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);
    xhr.onreadystatechange = function() {
      if(this.readystate === 4) return;
      if(this.status === 200) {
        resolve(this.responseText);
      } else {
        reject(this.responseText);
      }
    }
    xhr.onerror = function() {
      reject(this.statusText);
    }
    xhr.responseType = 'json';
    xhr.setRequestHeader('Content-Type',
'application/json');
    xhr.send(null);
  })
}
```

17.Javascript 为什么要变量提升，以及导致的问题 !!!!

- 提高性能

  解析和预编译过程中的声明提升可以提高性能，让函数可以在执行时预先为变量分配栈空间。
- 提高代码容错性

  例如在声明变量前就定义了变量，如果没有变量提升就会报错。

变量提升导致所有的变量都会成为一个全局变量，并定义在 window 上，如果定义的变量太多会导致 window 对象过为繁重

18.什么是尾调用 !!

尾调用指的是某个函数的最后一步操作是调用另一个函数。

19.尾调用优化 !!

函数调用会在内存形成一个“调用记录”，又称为调用帧，保存函数调用的一些信息，所有的调用帧形成一个调用栈。

尾调用是函数的最后一步操作，不需要保留外层函数的调用帧，直接由内层函数的调用帧替代。

尾调用优化就是只保留内层函数的调用帧，如果所有函数都尾调用就可以做到每次执行调用帧都只有一项，大大节省了内存。

注意: 只有不再用到外层函数的变量，内层函数的调用帧才会取代外层函数的调用帧，否则就无法进行尾调用优化。ES6 的尾调用优化只在严格模式下开启 (ES6 模块默认为严格模式)。

函数里调用另一个函数时，会保留当前的执行上下文，然后再新建另外一个执行上下文加入栈中。

使用尾调用的话，因为已经是函数的最后一步，所以这时不必再保留当前函数的执行上下文，从而节省了内存，这就是尾调用优化。

20.ES6 模块与 CommonJs 模块的区别 !!!!

- CommonJS 模块输出的是值的复制，ES6 模块输出的是值的只读引用。

  CommonJS 如果想得到内部变量的变化需要写成一个函数的形式，如果变量只是一个原始类型的值，该值会被缓存。
- CommonJS 模块是运行时加载，ES6 模块是编译时加载。

  这个差异是因为 CommonJS 加载的是一个对象，该对象只有在脚本执行时才会产生；

  而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码解

析阶段便会建立接口的引用。

- CommonJS 中顶层 this 指向当前模块，ES6 模块中顶层 this 指向 undefined。

21.常见的 DOM 操作 !!!

1. DOM 节点获取
    - getElementById
    - getElementByTagName
    - getElementByClassName
    - querySelect
    - querySelectAll
2. DOM 节点创建
    - createElement
    - createTextElement
3. DOM 节点删除

```
// 获取目标元素的父元素
let container =
document.getElementById("container");
// 获取目标元素
let targetNode =
document.getElementById("title");
// 删除目标元素
container.removeChild(targetNode);
```

4. 修改 DOM 元素

```
// 获取父元素
let container =
document.getElementById("container");
// 获取两个需要被交换的元素
let title = document.getElementById("title");
let content =
document.getElementById("content");
// 交换两个元素，把 content 置于 title 前面
container.insertBefore(content, title);
let a = document.createElement("a");
// 将 a 标签添加到 container 尾部
container.appenChild(a);
```

22.use strict 严格模式 !!!

使得 Javascript 在更严格的条件下运行。

- 变量必须先声明后使用
- 函数参数不能有同名
- 不能使用 0 表示八进制
- 不能对只读属性赋值
- 不能删除不可删除的属性
- 不能使用 delete 删除变量
- 禁止使用 this 指向全局对象
- eval 不会在它的外层作用域引入变量
- eval 和 arguments 不能被重新赋值
- arguments 不会自动反应函数参数的变化
- 不能使用 with 语句
- 不能使用 arguments.callee
- 不能使用 arguments.caller
- 不能使用 fn.caller 和 fn.arguments 获取函数调用的堆栈
- 增加了保留字 (protected、static 和 interface)

23.如何判断一个对象是否属于某个类 !!!

instanceof、constructor、Object.prototype.toString.call

24.强类型语言和弱类型语言的区别 !!

静态代码编译，类型预检测

25.解释性语言和编译性语言区别 !!

js 是解释性语言，js 的数据类型是动态的在运行时检查。
编译型语言的类型是静态的，在编译期间检查。

26.for...in 和 for...of 区别 !!!

- for...of 内部调用的是 Symbol.iterator 属性指向的 iterator 接口
- 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,性能较差,for...of 只遍历当前对象

补充：

- for...in,遍历对象自身和继承的可枚举属性(不包括 Symbol)
- Object.keys,获取自身的可枚举属性(不包括 Symbol)
- Object.getOwnPropertyNames,获取对象自身所有属性(包括不可枚举,不包括 Symbol)
- Object.getOwnPropertySymbols,获取自身所有 Symbol 属性
- Reflect.ownKeys,获取自身所有属性(包括不可枚举，Symbol)

  顺序: 数值属性名 > 字符串属性名 > Symbol 值属性名

27.如何使用 for...of 遍历对象 !!

- 配置该对象的 Symbol.iterator 属性，为其添加 generator 函数
- Object.keys、Object.values、Object.entries

28.ajax、axios、fetch 的区别 !!!

- AJAX

  Ajax 是一种在无需重新加载整个网页的情况下，能够对网页某部分进行更新的技术。
  缺点：

  - 本身是针对 MVC 编程，不符合前端 MVVM
  - JQuery 整个项目太大，如果只是要使用 Ajax 需要引入整个 JQuery
  - 不符合关注分离的原则
  - 配置和调用方式非常混乱，想做封装处理的时候配置不好处理，需要进行判断，如果方法不公用就每次都调用 ajax 一次。

- Axios

  Axios 是一种基于 Promise 封装的 HTTP 客户端，其特点如下：

  - 浏览器发起 XMLHttpRequests 请求
  - node 端发起 http 请求
  - 支持 Promise API
  - 客户端支持抵御 CSRF 攻击
  - 可以拦截请求和响应
  - axios 提供了并发请求的 api (axios.all 配合 axios.spread 不过可以用 Promise.all 替换)

```
axios.all(arr).then(
  axios.spread(function (a, b) {
    if (a.data.code == 0 && b.data.code == 0) {
      console.log("success");
    } else {
      console.log("failed");
    }
  })
);
```

  - 取消请求
  - 自动转换 json 数据

- Fetch

  - Fetch 是原生的 js，没有使用 XMLHttpRequest
  - 语法简洁，更加语义化
  - 基于 Promise 实现，支持 async/await
  - 更加底层，提供了丰富的 API，脱离了 XHR
  - Fetch 只对网络请求保错，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

在网络错误这些导致请求不能完成时 fetch 才会被 reject
- Fetch 不支持中断请求、不能监测请求的进度，兼容性不好

29.数组的遍历方法 !!!

```
普通的 for 循环、forEach、for...of、for...in
reduce、every、some、map
```

30.forEach 和 map 方法有什么区别 !!!

- forEach 和 map 对数据的操作不会改变原数组
- map 返回一个新数组，forEach 没有返回值
- forEach 和 map 都会对数组中每一个元素执行提供的函数
- map 可以使用 return 拿到函数调用的返回值，forEach 中使用 return 无效

31. try..catch 什么情况使用 !!

- async / await 请求时捕获错误
- JSON.stringfy / JSON.parse 解析失败时的异常处理
- 连续的 `.` 导致从 undefined/null 上获取属性值报错

32. 多标签之间如何通信 !!!

- localStorage：在一个标签页监听 localStorage 的变化，然后当另一个标签页修改时可以通过监听获取新数据。
- WebSocket：可以实现实时服务器推送，所有服务器就可以当做消息的转发人。
- postMessage

三、原型与原型链

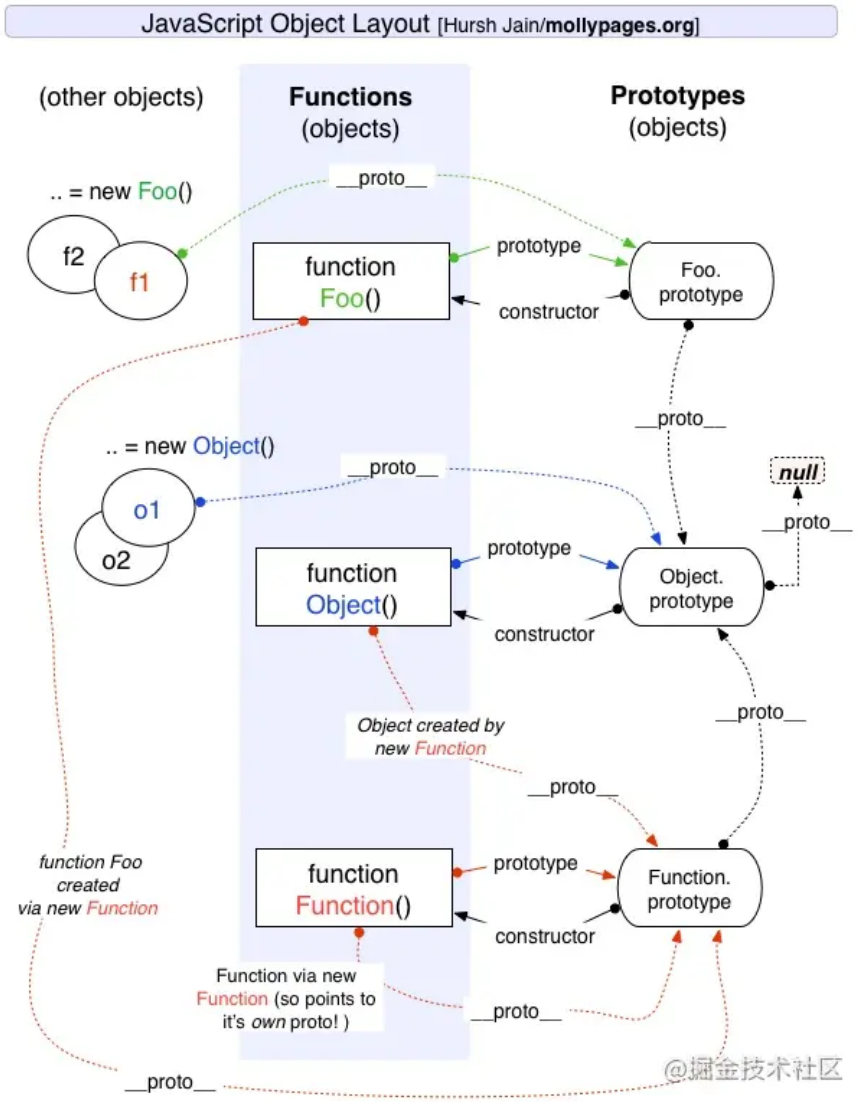
---

1. 对原型、原型链的理解 !!!!

在 ES6 之前 Javascript 通过构造函数来创建一个新的对象，每个函数内部都有一个 prototype 属性，它的属性值是一个对象，通常称为原型对象，这个对象包含了由该构造函数创建的所有实例共享的属性和方法。当使用构造函数创建一个新的对象后，这个对象内部会有一个 __proto__ 属性，该属性指向构造函数的 prototype 属性对应的值。对象上的这个 proto 属性通常称为对象原型属性，一般来说不应该能获取到这个属性的值，但是大多数浏览器都实现了 proto 属性来访问这个属性的值，但最好使用 Object.getPrototypeOf 和 Object.setPrototypeOf 来获取和修改这个属性。

当访问对象的一个属性时，如果这个对象内部不存在这个属性，就会去它的原型中找这个属性，每个原型对象又会有自己的原型，于是就这样一直找下去，这就是原型链的概念。原型链的尽头是 null 。

Javascript 对象是通过引用传递的，通过构造函数创建的每个实例中并没有一份属于自己的原型的拷贝。当修改原型对象时，与之相关的对象也会继承这一改变。



2. 原型修改、重写 !!!

```
function Person(name) {
  this.name = name;
}
// 修改原型
Person.prototype.getName = function () {};
var p = new Person("hello");
console.log(Object.getPrototypeOf(p) ===
Person.prototype); // true
console.log(Object.getPrototypeOf(p) ===
p.constructor.prototype); // true
// 重写原型
Person.prototype = {
  getName: function () {}
};
var p = new Person("hello");
console.log(p.__proto__ === Person.prototype); // true
console.log(p.__proto__ === p.constructor.prototype); //
false
```

之所以修改了原型后，对象的 constructor 属性不指向原来的构造函数的原因，是因为构造函数的原型被修改了，所以原型上指向构造函数的属性 constructor 的值也随之修改，变成了指向 Object 的构造函数。如果想要它指回原来的构造函数，就需要重新覆盖构造函数原型上的 constructor 属性为原本的构造函数。

3. 原型链指向 !!!!

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

4. 原型链的终点是什么？如何打印出原型链的终点 !!

因为 Object 是构造函数，所以它也有自己的原型，而他的原型又是一个对象，这个对象有自己的对象原型，这个对象原型为 null，所以原型链的终点是 null，即 Object.prototype.__proto__ 为 null。

5. 如何获得对象非原型链上的属性 !!

可以通过 hasOwnPrototype 来判断该属性是否属于原型链上的属性。

hasOwnPrototype 会忽略掉从原型链上继承到的属性。

四、执行上下文 !!!!

---

1. 对作用域、作用域链的理解

作用域规定当前执行代码对变量的访问权限。

- js 的作用域分为全局作用域、函数作用域和块级作用域
    - 全局作用域
        - 最外层函数和最外层函数外面声明定义的变量处于全局作用域
        - 所有未声明直接定义的变量自动声明为全局作用域
        - 所有 window 对象的内置属性都处于全局作用域
        - 弊端：过多全局作用域变量会污染全局命名空间,容易引起命名冲突
    - 函数作用域

      声明在函数体内部的变量处于该函数的函数作用域，只有在该函数体内才能访问到该变量
    - 变量提升
        - 所有未声明直接定义的变量，会提升为全局变量并提升到 **全局作用域顶部**
        - 通过 var 在函数内声明的变量具有函数作用域，该变量会提升到它声明时所处于的**函数的顶部**
    - 块级作用域
        - 块级作用域在一个代码块内部被创建，也就是一对花括号 `{}`
        - 通过 let、const 声明的变量具有块级作用域，var 声明的变量不具有块级作用域
        - 块级作用域声明的变量不会被提升到代码顶部、且一个块级作用域内不允许重复声明变量

作用域链就是从当前作用域开始一层一层往上找某个变量，如果找到全局作用域还没找到就放弃寻找，这种一层一层的关系就称为作用域链。

- js 采用的是**静态作用域**，所以**作用域在变量定义时就已经确定**好了并不会改变。

```
var value = 1;
function foo() {
  console.log(value);
}
function bar() {
  var value = 2;
  foo();
}
bar();
// 结果是 1
```

这是因为 js 采用的是**静态作用域**，当调用函数时，如果它的作用域内没有这个局部变量，就会依据函数 **编写的位置** 去查找上一层代码；

如果采用的是动态作用域，在它在自己的作用域内没有找到相应的局部变量时，会在函数的 **调用的位置** 的作用域查找

2. 对执行上下文的理解

执行上下文详解

当 js 执行一段可执行代码时，会创建对应的执行上下文，对于每个执行上下文都会有三个重要属性。

- 三个重要属性
    - 变量对象(Variable Object，VO)

      变量对象 VO 是与执行上下文相关的 **数据作用域** ，存储了在上下文中 **定义的变量和函数**
    - 作用域链
    - this

- 执行上下文类型
    - 全局上下文，全局上下文中的变量对象 VO 就是全局对象
    - 函数上下文，函数上下文中的变量对象 VO 称为活动对象 AO

      只有到当进入一个执行上下文中，这个执行上下文的变量对象才会被激活，所以叫 activation object ，而只有被激活的变量对象(也就是活动对象)上的各种属性才能被访问。活动对象是在进入函数上下文时被创建，它通过函数的 arguments 属性初始化，arguments 属性的值是 Arguments 对象。
    - eval 执行上下文，执行在 eval 函数中的代码会有属于他自己的执行上下文

- 函数 `执行过程` 分为 `进入函数` 和 `代码执行` 两个阶段
    - 进入函数，这时还没有执行代码，此时的 VO 会包括
        - 函数形参 (函数上下文才有)
            - 由名称和对应值组成的一个变量对象的属性被创建
            - 没有传入实参，函数形参对应属性值为 undefined
        - 函数声明
            - 由名称和对应的值 (函数对象 function-object) 组成一个变量对象的属性被创建
            - 如果变量对象已经 **存在相同名称的属性，则完全替换这个属性**
        - 变量声明
            - 由名称和对应值(undefined)组成一个变量对象的属性被创建
            - 如果变量名跟已经声明的形参或函数相同，**则变量声明不会干扰这类属性**，保持原来的变量

```
function foo(a) {
  var b = 2;
  function c() {}
  var d = function () {};
  b = 3;
}
foo(1);

AO = {
  arguments: {
      0: 1,
      length: 1
  },
  a: 1,
  b: undefined,
  c: reference to function c(){},
  d: undefined
}
```

    - 代码执行，顺序执行代码，根据代码修改 AO 上属性的值

```
AO = {
  arguments: {
      0: 1,
      length: 1
  },
  a: 1,
  b: 3,
  c: reference to function c(){},
  d: reference to FunctionExpression "d"
}
```

- 变量对象创建过程总结：
    - 全局上下文的 VO 初始化是全局对象
    - 函数上下文的 VO 初始化 只包括 **Arguments** 对象

```
AO = {
  arguments: {
    0: 1,
    length: 1
  }
};
```

- 在进入函数创建执行上下文时会给 VO 添加 `形参` 、 `函数声明` 、 `变量声明` 等初始的属性值

```
  AO = {
    arguments: {
      0: 1,
      length: 1
    },
    a: 1,
    b: undefined,
    c: reference to function c() {},
    d: undefined
  }
```

- 在代码执行阶段，会再次修改 VO 的属性值

```
AO = {
  arguments: {
    0: 1,
    length: 1
  },
  a: 1,
  b: 3,
  c: reference to function c() {},
  d: reference to FunctionExpress 'd'
}
```

- 思考题
  - 第一题

```
function bar() {
  a = 1;
  console.log(a); // ???
}
bar();

function foo() {
  console.log(a);
  var a = 1; // ???
}
foo();
```

第一段执行 console 的时候， AO 的值是：

```
AO = {
  arguments: {
    length: 0
  }
};
```

因为 a 没有通过 var 声明，不会存到 AO 中，AO 中没有 a 的值，然后就会到全局去找，全局也没有，所以会报错。

  - 第二题

```
console.log(foo);
function foo() {
  console.log("foo");
}
var foo = 1;
```

会打印函数，而不是 undefined 。因为在进入执行上下文时，首先会处理函数声明，其次会处理变量声明，如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性。

结合作用域链的函数执行过程总结

1. 执行到函数的声明语句，保存**函数所处作用域链**到的函数的内部属性[[scope]]，这个属性的值就是该**函数**的所处作用域
2. 执行到函数调用语句，创建函数执行上下文，将函数执行上下文压入执行上下文栈中
3. 作进入函数的准备工作(预编译阶段)
    1. 复制函数作用域的 [[scope]] 属性到函数**执行上下文**中的作用域属性 `Scope`
    2. 用 arguments 创建 AO ，随后加入形参、函数声明、变量声明
    3. 将 AO 压入自己的函数执行上下文的作用域属性 `Scope` 中

个人认为在 2-3 之间又或是 3 这个准备工作完后会有一个将 this 复制到当前函数执行上下文中的 this 属性的过程。
因为此时如果有绑定 this 的操作也该绑定完了，如果没有绑定也该继承外层执行上下文的 this 了。
这个绑定操作就是将 this 复制到执行上下文的 this 上，并且将 this 压入到 Scope 属性的过程。
所以最后 Scope 里是包括 this、AO、[[scope]] 三部分的。
这样在后面的执行阶段，便可以去该执行上下文的作用域 Scope 属性里找相应的变量，并且在函数执行完毕后这个执行上下文就会从执行栈中弹出。

需要注意这里和闭包的联系，如果在当前函数中创建了一个函数，在函数执行时，执行到声明这个函数的语句时会把作用域放到函数的 [[scope]] 属性上。
而如果这个函数是个闭包，他就会保存着这份来自外层执行上下文的作用域，直到闭包函数执行完毕。

4. 执行函数，修改 AO 的属性值(执行阶段)
5. 函数执行完毕，弹出该函数的执行上下文

总结：
注：每一个箭头属于一个不同阶段分别是：执行到函数声明、函数调用后(创建=>预编译=>执行，这三个的工作对象都是函数上下文)、执行完毕

创建全局执行上下文，压入栈->**执行到函数的声明语句**，添加作用域链到**函数的** `[[scope]]` **内部属性**
->函数调用后，**创建**函数执行上下文，将函数执行上下文压入栈函数
=>**预编译**阶段，复制**函数** `[[scope]]` **属性**到**执行上下文的 Scope 属性**，用 arguments 创建并初始化 AO、加入形参、函数声明、变量声明；
=>**执行**阶段，(this 是在执行阶段加入到 AO 中的)，将 AO 压入**上下文的 Scope 属性**
->执行函数，修改 AO 值->执行完毕，执行栈中弹出执行上下文

3. 作用域与执行上下文区别理解

JavaScript 属于解释型语言，JavaScript 的执行分为：解释和执行两个阶段

- 解释阶段：
    - 词法分析
    - 语法分析
    - 作用域规则确定
- 执行阶段：
    - 调用函数后创建执行上下文
    - 执行函数代码
    - 垃圾回收

JavaScript 解释阶段便会确定作用域规则，因此作用域在函数定义时就已经确定了，而不是在函数调用时确定，作用域访问的变量是由编写代码时的结构确定。

但是执行上下文是函数执行之前创建的，执行上下文最明显的就是 this 的指向是执行时确定。

区别：

- 作用域在定义时就确定，并且永远不会改变；执行上下文在执行时确定，随时可能改变。
- 一个作用域下可能包含若干个上下文环境，也有可能从来没有过上下文环境（函数没有被调用）；也有可能有过，但是因为函数被调用完毕后，上下文环境被销毁了；也有可能同时存在一个或多个（闭包）；同一个作用域下，不同的调用会产生不同的执行上下文环境，继而产生不同的变量的值。

4. 对 this 的理解

MDN

```
let obj = {
  foo: function () {
    console.log(this.bar);
  },
  bar: 2
};
let bar = 1;
obj.foo(); // 2
const foo = obj.foo;
foo(); // 1
```

当我们通过对象访问到该对象上的某一个方法，并将其赋值到一个新的变量上，此时这个函数就会变为在全局环境中执行，也就是这个函数的外层执行上下文为全局上下文，此时因为该函数它没有自己的 this，所以默认会继承外层执行上下文中的 this，就指向了全局对象(也有可能为 undefined)。当我们尝试调用这个处于全局环境中的函数时，它会通过 this 去全局对象上找变量 bar 如果找到就输出否则为 undefined。

而如果采用 对象的.语法 去调用一个函数，这个函数内部就会隐式的绑定 this 为当前的调用对象，因此这个函数有了自己的 this 就不需要去外层执行上下文中继承 this。

需要注意，js 中原始对象以字典结构保存，每一个属性都对应一个属性描述对象，而如果一个属性的值是一个对象(函数也是一个对象)，那么这个属性的[[value]]的值就是这个对象在堆中的地址。

5. 对闭包的理解

闭包是指那些能够访问自由变量的函数，每当创建一个函数，闭包就会在函数创建的同时被创建出来。

自由变量指的是在函数中使用的，但既不是函数参数也不是函数的局部变量的变量(外层的 this、局部变量、外层。闭包=函数+函数能够访问的自由变量。

理论上，所有的函数都是闭包，因为他们都在创建的时候将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于在访问自由变量，这时候使用最外层的作用域。

而从实践角度上讲，闭包是一个即使创建它的**上下文**已经销毁但是它仍然存在，且函数体中引用了自由变量的函数。

它能访问到创建它的上下文中的作用域的原因，是因为外层函数在执行到创建函数的语句时，会在函数上的[[scope]]属性保留**外层执行上下文**的作用域链，这个作用域链也就是**执行上下文的 Scope 属性**，这个 Scope 属性包括该执行上下文的三个重要属性 this、VO、作用域链，所以即使外层函数的执行上下文已经销毁，但它仍能够访问到属于外层作用域的自由变量。

- 必会题!!!!

```
var data = [];
for (var i = 0; i < 3; i++) {
  data[i] = function () {
    console.log(i);
  };
}
data[0](); // 3
data[1](); // 3
data[2](); // 3
```

当执行到 data[0] 函数之前，此时全局上下文的 VO 为：

```
globalContext = {
    VO: {
        data: [...],
        i: 3
    }
}
```

当执行 data[0] 函数的时候，data[0] 函数的作用域链为：

```
data[0]Context = {
    Scope: [AO, globalContext.VO]
}
```

data[0]Context 的 AO 并没有 i 值，所以会从 globalContext.VO 中查找，i 为 3，所以打印的结果就是 3。
所以改成闭包看看：

```
var data = [];
for (var i = 0; i < 3; i++) {
  data[i] = (function (i) {
    return function () {
      console.log(i);
    };
  })(i);
}
data[0]();
data[1]();
data[2]();
```

当执行到 data[0] 函数之前，此时全局上下文的 VO 为：

```
globalContext = {
    VO: {
        data: [...],
        i: 3
    }
}
```

跟没改之前一模一样。
当执行 data[0] 函数的时候，data[0] 函数的作用域链发生了改变：

```
data[0]Context = {
  Scope: [AO, 匿名函数Context.AO
globalContext.VO]
}
```

匿名函数执行上下文的 AO 为：

```
匿名函数Context = {
  AO: {
    arguments: {
      0: 0,
      length: 1
    },
    i: 0
  }
};
```

data[0]Context 的 AO 并没有 i 值，所以会沿着作用域链从匿名函数 Context.AO 中查找，这时候就会找 i 为 0，找到了就不会往 globalContext.VO 中查找了，即使 globalContext.VO 也有 i 的值(值为 3)，所以打印的结果就是 0。

五、各种手写

1. this 指向篇 !!!!

- call

```
Function.prototype.call = function (obj, ...params) {
  // 调用对象处理
  obj = obj ? obj : globalThis;
  // 拿到方法
  let fn = this;
  // 在对象上绑定方法
  const key = Symbol();
  obj[key] = fn;
  // 参数处理
  // 调用
  const res = obj[key](...params);
  delete obj[key];
  return res;
};
```

- apply

```
Function.prototype.apply = function (obj,
params) {
  obj = obj ? obj : globalThis;
  let fn = this;
  const key = Symbol();
  obj[key] = fn;
  const res = obj[key](...params);
  delete obj[key];
  return res;
};
```

- bind

```
Function.prototype.bind = function (obj,
...params) {
  obj = obj ? obj : globalThis;
  let fn = this;
  const key = Symbol();
  obj[key] = fn;
  let _this = this;
  const newFunc = function (...newParams) {
    if (this instanceof _this) {
      this[key] = _this;
      return this[key](...params, ...newParams);
    }
    return obj[key](...params, ...newParams);
  };
  newFunc.prototype =
Object.create(this.prototype);
  return newFunc;
};
```

2. 模拟实现篇 !!!

- setTimeout 模拟 setInterval
  为什么要用 setTimeout 模拟 setInterval ?
  setInterval 是一个宏任务。
  浏览器的时钟触发线程仅管理时钟的计时操作，时钟的时间到了时钟触发线程就会将该时钟的回调函数加入到事件触发线程的事件队列中。
  而事件触发线程可能任务较多又或是前面某个任务等待时间较长,而 JS 引擎来不及去处理。所以造成了 setInterval 计时不准确的现象
- 交集并集
  `[ ...arr1.filter((item)=>new Set(arr2).has(item))]`
  `[ ...arr1, ...arr2.filter((item)=>!new Set(arr1).has(item)) ]`
- flat

- 深拷贝
- 函数柯里化
- compose
- 防抖
  防抖是指在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。

  使用在一些点击请求的事件上，避免因为用户的多次点击向后端发送多次请求。
- 节流
  节流是指规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行.
  如果在同一个单位时间内某事件被触发多次，只有一次能生效。
  可以使用在 scroll 函数的事件监听上，通过事件节流来降低事件调用的频率。
- LazyMan
- Object.is
- Object.freeze
  freeze 是浅冻结，目标对象上 `某个属性上的属性` 仍然是可以改变的。
  freeze 调用了 preventExtensions 这个方法，preventExtensions 会将对象的原型也变为不可扩展的对象
- 运用 Promise 手写 AJAX 实现

```
function sendAJAX(url) {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open("GET", url, false);
    // 设置请求头
    xhr.setRequestHeader("Content-Type",
"application/json");
    // 监听状态变化
    xhr.onreadystateChange = function () {
      if (xhr.readyState !== 4) return;
      if (xhr.status === 200) {
        resolve(xhr.responseText);
      } else {
        reject(new Error(xhr.responseText));
      }
    };
    // 发送请求
    xhr.send();
  });
}
```

- 虚拟 DOM 转化为真实 DOM

- 真实 DOM 转化为 JSON 数据
- 列表转树形结构 listToTree
- 树形结构转成列表
- URL 参数提取

```
function parseUrlSearchToJson () => {
  const params = {}
  const query = window.location.href.split('?')
[1]
  if (query.length > 1) {
    const buf = query.split('&')
    for (let i = 0; i < buf.length; i++) {
      const tmp = buf[i].split('=')
      params[tmp[0]] =
decodeURIComponent(tmp[1])
    }
  }
  return params
}
```

- 对象字符串转树形结构(objectStrToTree)

```
let strarr = {
  "a-b-c-d": 1,
  "a-b-c-e": 2,
  "a-b-f": 3,
  "a-j": 4
};
let obj = {
  a: {
    b: {
      c: {
        d: 1,
        e: 2
      },
      f: 3
    },
    j: 4
  }
};
```

3. 继承篇 !!!!

- Object.create
- 写一个你认为不错的 js 继承方式
- new

- instanceof, instanceof 的原理就是查找目标对象的原型链上是否存在该构造函数的原型。

4. 排序篇(详细解各种排序) !!!!

- 冒泡

```
function bubbleSort(arr) {
  const len = arr.length;
  for (let i = 0; i < len; i++) {
    for (let j = 1; j < len - i; j++) {
      if (arr[j] < arr[j - 1]) {
        // 前面的大于后面的
        [arr[j], arr[j - 1]] = [arr[j - 1],
arr[j]];
      }
    }
  }
}
```

- 选择

```
function selectSort(arr) {
  const n = arr.length;
  for (let i = 0; i < n; i++) {
    let pos = i;
    for (let j = i + 1; j < n; j++) {
      if (arr[j] < arr[pos]) {
        // 找到后面那个最小的
        pos = j;
      }
    }
    if (pos !== i) {
      // 有找到最小的，插入到前面去
      [arr[pos], arr[i]] = [arr[i], arr[pos]];
    }
  }
}
```

- 插入

```
function insertSort(arr) {
  const n = arr.length;
  for (let i = 1; i < n; i++) {
    let t = arr[i];
    let j = i - 1;
    while (j >= 0 && arr[j] > t) {
      arr[j + 1] = arr[j--];
    }
    arr[j + 1] = t;
  }
}
```

- 堆排序

```
function heapSort(arr) {
  let heapSize = arr.length - 1;
  // 构造大顶堆
  const getMaxHeap = (arr, i, heapSize) => {
    let l = i * 2 + 1; // 它的左儿子
    let r = i * 2 + 2; // 它的右儿子
    let largest = i; // 记录父节点、两个子节点中最大
一个
    if (l <= heapSize && arr[l] > arr[largest])
{
      largest = l;
    }
    if (r <= heapSize && arr[r] > arr[largest])
{
      largest = r;
    }
    if (largest !== i) {
      // 将最大的那个换到父节点位置
      [arr[i], arr[largest]] = [arr[largest],
arr[i]];
      // 这个主要在下沉后的重新找堆顶元素会用到
      getMaxHeap(arr, largest, heapSize);
    }
  };

  // 初始化大顶堆
  const initMaxHeap = (arr, heapSize) => {
    // 利用了完全二叉树的性质，完全二叉树的节点只能是
2n 个
    // 而每一个`右结点`的父节点的下标值为 n/2-1
    // 将 i 值初始化为第一个末尾节点
    for (let i = (heapSize >> 1) - 1; i >= 0; i-
-) {
      getMaxHeap(arr, i, heapSize);
    }
  };
  initMaxHeap(arr, heapSize);

  // 执行大顶堆的堆顶值下沉
  while (heapSize >= 0) {
    // 与堆顶元素交换
    // 将堆顶值换到末尾
    [arr[heapSize], arr[0]] = [arr[0],
arr[heapSize]];
    heapSize--; // 缩小下一次需要查找的堆范围
    getMaxHeap(arr, 0, heapSize);
  }

}
```

```
}
```

5. 设计模式篇 !!!

- 发布订阅
- 单例模式

6. promise 篇 !!!

- 并发限制的异步调度器

六、ES6

1.let、const、var 区别 !!!!

1. var 声明存在变量提升，let 和 const 不存在变量提升。
2. var 声明的变量会变为全局变量，并且会将该变量添加为全局对象的属性，但 let 和 const 不会。
3. 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间声明的变量。let 和 const 不存在变量的重复声明。
4. 用 let 和 const 声明的变量会存在暂时性死区，就是在改变了声明之前都是不可用的。
5. let 和 const 具有块级作用域，当一个块级作用域中声明了一个变量，那么这个变量就会和这个作用域绑定，这个作用域中不能再出现同名的变量。块级作用域解决了内层变量可能会覆盖外层变量的问题，以及用来计数的循环遍历泄露为全局变量的问题。
6. 使用 var 和 let 声明变量可以不赋予初始值，但使用 const 声明变量必须赋予初始值。因为 const 表达的含义为常量，即该变量声明之后它所指向的内存地址中的值不可改变。

2.const 对象的属性可以修改吗 !!!

可以修改

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。对于基本数据类型的数据(number、string、boolean)，其值保存在变量指向的栈中相应的内存地址，因此等同于常量。

但对于引用类型的数据，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，置于它指向的数据结构是不是可变的就不完全不能控制了。

3.如果 new 一个箭头函数会怎样 !!!

箭头函数没有 prototype，也没有自己的 this 所有不能对箭头函数使用 new 语法。

new 操作符的步骤如下:

1. 创建一个对象
2. 将构造函数的原型赋予新对象，也就是将对象的 proto 属性指向构造函数的 prototype 属性
3. 调用构造函数加工这个对象，也就是为这个对象添加实例属性和实例方法
4. 如果构造函数有返回值，并且返回值是一个对象或者函数，将得到的返回值返回，否则返回加工后的对象

4.箭头函数和普通函数的区别 !!!!

- 箭头函数的语法比普通函数更加简洁
- 箭头函数没有自己的 this，它只会从自己作用域的上一层继承 this，所以在定义时它的 this 指向就已经确定且之后不会改变
- 使用 call、apply、bind 并不能改变箭头函数的 this 指向
- 箭头函数没有 prototype
- 箭头函数不能作为构造函数使用，所以不能对箭头函数使用 new 语法会报错
- 箭头函数没有自己的 arguments，如果在箭头函数中使用 arguments 其实使用的它外层函数的 arguments
- 箭头函数不能用作 generator 函数，不能使用 yield 关键字

5.箭头函数的 this 指向 !!!!

箭头函数总是捕获其所在上下文的 this 的值作为自己的 this 值

6.扩展运算符作用及运用场景 !!

- 将一个参数列表转化为一个参数序列
- 合并数组或对象、实现浅拷贝、将类数组对象转为数组、与解构赋值结合接收剩余序列
- 将实现了 Iterator 接口的对象转为数组

它能转换对象的原因本质是因为该对象配置了 iterator 接口

7.Proxy 可以实现什么功能 !!

修改操作的默认行为。

比如在 Vue3 中通过 Proxy 来替换原本 Vue2 用 Object.definedProperty 定义响应式数据。

之所以 Vue3 使用 Proxy 替换原本的 API 根本原因在于 Proxy 无需一层一层递归为每个属性添加代理，并且 Vue2 的响应式数据监听对于数组的监听并不是

那么的理想

Proxy 属于 ES6 新增的语法，所以兼容性不大好。

8.对对象与数组的解构的理解 !!

- 数组解构采用 [] 语法，对象解构采用 {} 语法
- 对象解构时，以键名作为解构时的匹配模式，无需考虑键值对在原对象中的顺序
- 数组解构时，以元素的位置作为匹配条件，也就是按数组中的顺序进行解构
- 不过数组也是一个对象，也可以用对象的解构形式来解构，只是解构时的模式是一个数值，也就是数组中的下标。

9.如何提取高度嵌套的对象中的指定属性 !

可以通过多层解构赋值的形式

```javascript
const {a:{b:{c:{d:{e}}}}} = {
    a:{
        b:{
            c:{
                d:{
                    e:1
                }
            }
        }
    }
}
```

10.对 rest 参数的理解 !!!

- rest 参数常用来把一个参数序列转化为参数列表
- 经常用于获取函数的多余参数，又或是处理函数参数个数不确定的情况。
- rest 只能是函数的最后一个参数否则会报错，函数的 length 属性不包括 rest 参数。

11.ES6 中模板语法与字符串的处理 !!!

模板语法常用于转义字符串，可以防止xss攻击。

七、异步编程

1. 异步编程实现方式 !!!

- 回调函数

  多个回调函数嵌套形成回调地狱，代码间耦合度高不利于维护。

- Promise

  解决回调地狱的问题，将嵌套的回调函数改为链式调用。
  Promise 一旦新建它就会立即执行，无法中途取消。
  当 promise 对象处于 pending 状态时，我们无法得知它进展到哪一个阶段。

- Generator

  在函数执行的过程中可以将函数的执行权转交到函数体外，并且在函数体外也能将执行权归还。
  通过 generator 可以使用同步的顺序书写异步任务。

- Async

  结合 Promise 和 Generator 的一个语法糖，本质是一个自动执行机。
  当函数体内部执行到 await 语句时，如果 await 后面的是一个 promise 对象封装的异步任务，那么函数将会等待 Promise 对象状态变为 resolved 后再继续向下执行。

2. 宏任务微任务 !!!!

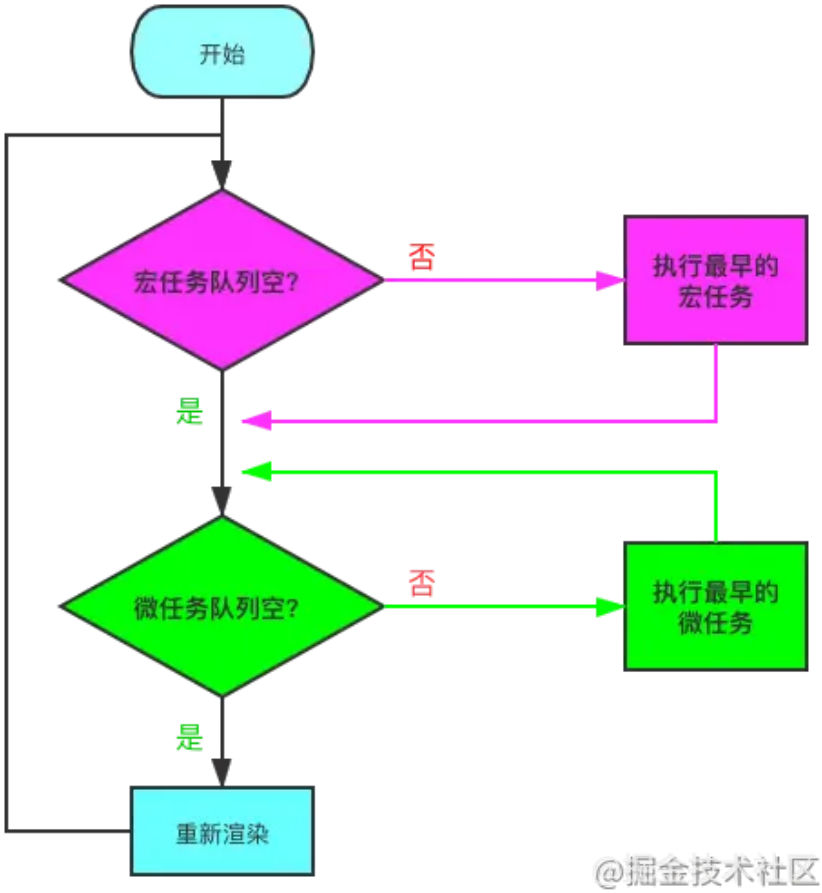
Js 中有两种任务分别是同步任务和异步任务，而异步任务又可以划分为宏任务和微任务。宏任务是由宿主环境发起的，而微任务由 JS 自身发起。

- 常见的宏任务有：script、setTimeout、setInterval、setImmediate、requestAnimationFrame、I/O、UI 渲染
- 常见的微任务有：Promise.then/catch/finally、MutationObserver、process.nextTick(node)

通常将一个 script 代码块看做是一个宏任务，例如两个 script 代码块，这时候 js 引擎会先执行第一个 script 代码块并且将其代码中的宏任务和微任务执行完后，再去执行第二个 script 代码块。

3. 事件循环 !!!!

宏任务和微任务的执行就需要提到浏览器的事件循环，事件循环的整个流程大致是这样的：



每次一有微任务就会直接将它加入到微任务队列中，无需等待下一次循环。

因为宏任务的执行时间一般比较长，所以微任务的执行优先级要比宏任务更高。

4. 对 Promise 的理解 !!!!

Promise 是异步编程的一种解决方案，它是一个对象，从它可以获取异步操作的消息。

简单来说 Promise 就是一个容器，保存着未来某个时间才会结束的事件的结果。

Promise 有三个状态分别是 `pending、fullfilled、reject` 。Promise 对象的状态**不会受外界影响**，只有异步操作的 `结果` 才可以 `决定` 当前是哪一个 `状态` 。并且 Promise 对象的状态一旦**改变了就不会再变**，它的状态只能从 `pending 变为 fullfilled` 或 `pending 变为 rejected` ，如果 Promise 对象的状态已经改变了再去监听它也是拿不到结果的。

Promise 对象状态的改变，是通过它的构造函数提供的两个函数类型的参数 resolve 和 reject 来实现的，可以在异步操作结束后调用这两个函数改变 promise 对象的状态。Promise 的 prototype 属性上定义了一个 then 方法，可以通过 then 方法，为两个状态的改变，注册回调函数，这两个回调函数均属于

微任务，会在本轮事件循环的宏任务执行完后执行。

另外，Promise 一旦创建就会立即执行，无法中途取消。其次，如果没有为 promise 设置回调函数，promise 内部抛出的错误不会反应到外部。并且当 promise 对象处于 pending 状态时，无法得知它执行到哪一阶段。

可以在回调函数中返回一个 pendding 状态的 promise 对象来中断链式调用

5. Promise 解决的问题 !!!

在传统的回调函数解决异步任务的方法中，如果后一个任务的执行需要依赖前一个任务的返回结果，就会导致多个回调函数嵌套的情况，也就是常说的回调地狱。Promise 解决了回调地狱的问题，将嵌套的回调函数转换为链式调用。降低了代码间的耦合度，更有利于维护。

6. Promise 用法 !!!!

- then

```
// 28-38 √
function resolvePromise (promise, result,
resolve, reject) {
  if (promise === result) {
    return reject(new TypeError('Chaining cycle
detected for promise #<Promise>'));
  }
  // 28. 如果 result 是对象或函数
  if (typeof result === 'object' || typeof
result === 'function') {
    // 29.如果 x 是 null ，直接作为结果 resolve
    if (result === null) {
      resolve(result);
    }
    // 30. 如果 result 是一个 thenable 对象
    let then;
    try {
      then = result.then;
    } catch (err) {
      // 31. result 不是 thenable 对象
      reject(err);
    }
    // 32.调用 thenable 对象的 then 方法
    if (typeof then === 'function') {
      // 35. 防止循环调用
      let called = false;
      try {
        // 33. 将 result 作为 then 方法 this
        then.call(
          result,
          v => {  // 34.  then 方法的成功回调
resolvePromise
            if (called) return;
            called = true;
            resolvePromise(promise, v, resolve,
reject);
          },
          err => {// 34.  then 方法的失败回调
rejectPromise
            if (called) return;
            called = true;
            reject(err);
          }
        )
      } catch (err) {
        // 36. then 方法执行错误 抛出异常


        // 如果 resolvePromise/rejectPromise 已经
```

```
        // 如果 resolvePromise/rejectPromise 已经
被调用，直接返回
        if (called) return;
        reject(err);
      }
    } else {
      // 37. 如果 then 不是函数，则将 result 作为参
数执行 promise
      resolve(result);
    }
  } else {
    // 38. 如果  result 既不是函数也不是对象，将
result 作为参数执行 promise
    resolve(result);
  }
}
/**
 * 成功的回调函数、失败的回调函数
 * @param {*} onFullfilled
 * @param {*} onRejected
 */
then (onFullfilled, onRejected) {
  // 26. 使用默认函数
  onFullfilled = typeof onFullfilled ===
'function' ? onFullfilled : v => v;
  onRejected = typeof onRejected === 'function'
? onRejected : e => { throw e };

  // 16. 为了实现链式调用需要返回一个 Promise 对象
  const promise = new MyPromise((resolve,
reject) => {
    // 8. 判断promise对象状态
    if (this.status === FULLFILLED) {

      // 20. 创建一个微任务等待 promise 完成初始化
      queueMicrotask(() => {
        // 23. 捕获 then 执行时的错误
        try {
          // 17. 获取成功回调函数执行结果
          // 9. 调用成功回调，传入结果
          const result =
onFullfilled(this.value);

          // 18. 传入 resolvePromise 集中处理
          // 其实就是借助这个函数判断 onFullfilled
的返回值是不是一个 Promise 对象
          // 是的话再次调用 then 方法，不是直接
resolve/reject(返回值)

          // resolvePromise(result, resolve,
```

```
            // resolvePromise(result, resolve,
reject);

            // 19.将 promise 传入用于判断与 result 是
否为同一个
            // 如果不将成功回调也就是 then 里的回调更改
为异步调用
            // 此时的 promise 并未初始化
            resolvePromise(promise, result,
resolve, reject);
          } catch (err) {
            // 24. then 执行时出现错误直接调用 reject
            // 这里其实有一个细节很多人没有注意到(其实是
自己犯蠢了)
            // 就是这个 reject 不能通过 this 调用
            // 如果通过 this 调用的话使用的就是 this
上的 reject 函数了
            // 而不是 new MyPromise 时传入的 reject
            // 它要改变的是当前 then 方法返回的
Promise 对象的状态
            reject(err);
          }
        })
      } else if (this.status === REJECTED) {
        // 25. 补充 reject
        queueMicrotask(() => {
          try {
            // 9. 调用失败回调，传入原因
            const result =
onRejected(this.reason);
            resolvePromise(promise, result,
resolve, reject);
          } catch (err) {
            reject(err);
          }
        })
      } else {
        // 11. 不清楚异步任务结果，需要存储回调函数
        // 等到异步任务结束再调用
        // this.onFufilledCallback = onFullfilled;
        // this.onRejectedCallback = onRejected;

        // 26. 补充 pending，包装回调函数将其变为微任
务，捕获 then 执行时的错误
        // 14. 添加多个回调函数
        this.onFufilledCallbacks.push(() => {
          queueMicrotask(() => {
            try {
              const result =
```

```
            const result =
onFullfilled(this.value);
            resolvePromise(promise, result,
resolve, reject);
          } catch (err) {
            reject(err);
          }
        })
      });
      this.onRejectedCallbacks.push(() => {
        queueMicrotask(() => {
          try {
            const result =
onRejected(this.reason);
            resolvePromise(promise, result,
resolve, reject);
          } catch (err) {
            reject(err)
          }
        })
      });
    }
  })
  return promise;
}
```

- catch

  如果 promise 执行的是一个异步任务，则 try...catch 不能捕获到这个异步任务的异常，因为 try...catch 捕获的是一个同步任务的异常，而 promise 内部执行的是一个异步任务，当异步任务执行时，同步任务已经执行完毕了，所以 try...catch 并不能捕获 promise 的异常。

  catch 相当于 then 方法的第二个参数，promise.then(null, onRejected);

- finally

  不管 Promise 对象最后的状态如何，传入 finally 的回调函数都会执行。并且就算 finally 返回了新的值，也还是会将之前的结果包装成 promise 后 return 而不是 return 自己的新值。

- all

  接收一个 promise 的可迭代类型的输入，并且只返回一个 Promise 实例。只有在接收的所有 promise 对象都变为 resolved 后，返回的对象状态才会变为 resolved，并且将这些 promise 对象的返回值组成一个数组，传递给返回的 promise 对象的回调。如果接受的 promise 对象中有一个被 reject 那么返回的 promise 对象将变为 rejected，并且这个被 rejected 对象的返回值会传递给返回的 promise 对象的回调。

- race

将多个 promise 对象包裹为一个 promise 对象，并在一个 promise 对象 resolve 或 reject 后，改变其返回的 promise 对象状态，并将其返回值作为后续方法的参数

- allSettled

  返回一个在所有给定的 promise 都已经 fulfilled 或 rejected 后的 promise，并带有一个对象数组，每个对象表示对应的 promise 结果

- resolve

  关于 return Promise.resolve() 详解

  resolve 会根据传入参数的类型，进行不一样的操作。

  本质是因为 Promise.resolve 内部调用了 then 并且这个 then 方法会调用 resolvePromise ，

  resolvePromise 内部对传入参数进行了类型的判断，如果是 thenable 对象会调用它的 then 方法，

  并给它的 then 方法传入 resolve/reject，并且如果这个参数是个 Promise 对象的话，因为 Promise

  对象也有 then 方法，并且它的 then 方法是一个微任务，所以会导致后面的 thenable 对象后面的 then

  回调被延迟了，需要等待这个 (thenable Promise 对象) 的 then 方法，执行完毕。

- reject

  返回一个 rejected 状态的 promise 对象，原封不动的将参数作为 reject 的理由，变成后续方法的参数

7. Promise.all、Promise.race、Promise.allSettled 区别 !!

- 这三者都会将多个 Promis 实例包装成一个新的 Promise 实例
- Promise.all 只有当所有任务均 resolved 时，返回的 promise 对象才会为 resolved；并且 Promise.all 成功时返回的结果数组与原数组中的顺序一致；如果有一个任务失败，返回的 promise 对象的状态将会立马变为 rejected。
- Promise.race 只要传入的所有实例中有一个实例先改变了状态，则返回值的状态就会立马改变。
- Promise.allSettled 会在所有的任务完成时才返回一个对象数组，每个对象表示对应的 promise 结果；并且返回的对象数组与原数组中顺序一致。

8. generator !!!!

Generator 是 ES6 提供的一种异步编程解决方案。

9. async/await !!!!

async/await 解决的问题

- Promise 虽然解决了回调地狱的问题，但多个 then 的链式调用也会带来额外的阅读负担。
- 通过 async/await 可以更加优雅的将异步任务用同步的写法。
- async/await 可以用 try/catch 捕获异常，而 Promise 只能通过 catch 捕获错误。

10. 并发与并行区别 !

- 并发是宏观概念，比如分别有任务 A 和任务 B，在一段时间内任务间不断切换完成了这两个任务，这种情况就称之为并发。
- 并行是微观概念，假设 CPU 中存在两个核心，那么就可以同时执行任务 A、B，这种处于多个不同的执行器上同时执行多个任务就成为并行。

11. 传统的回调函数缺点 !!

- 嵌套函数存在耦合性，一旦有所改动，就需要改变很多地方
- 嵌套函数一多，就很难处理错误
- 回调函数不能使用 try..catch 捕获错误，不能直接 return

12. 定时器 setTimeout、setInterval、requestAnimationFrame !!!!

setTimeout

1. setTimeout 的执行时间并不是确定的，在 Javascript 中，setTimeout 任务被放进了异步队列中，只有当主线程上的任务执行完以后，才会去检查该队列里的任务是否需要开始执行，所以 setTimeout 的实际执行实际一般要比设定的时间晚一些。
2. 刷新频率受屏幕分辨率和屏幕尺寸的影响，不同设备的屏幕绘制频率可能会不同，而 setTimeout 只能设置一个 `固定的时间间隔` ，这个时间不一定和屏幕的刷新时间相同。
3. setTimeout 的执行只是在 `内存中` 对元素属性进行改变，这个变化必须要等待屏幕下次绘制时才会被更新到屏幕上。如果两者的步调不一致，就可能会导致中间某一帧的操作被跨域过去，而直接更新下一帧的元素。

假设屏幕刷新率 fps 为 60HZ，即每隔 16.7ms 刷新一次，而 setTimeout 每隔 10ms 设置图像向左移动 1px，就会出现如下绘制过程。

1. 第 0ms：屏幕未绘制，等待中，setTimeout 也未执行，等待中；
2. 第 10ms：屏幕未绘制，等待中，setTimeout 开始执行并设置元素属性 left=1px；
3. 第 16.7ms：屏幕开始绘制，屏幕上的元素向左移动了 1px，setTimeout 未执行，继续等待中；

4. 第 20ms：屏幕未绘制，等待中，setTimeout 开始执行并设置 left=2px；
5. 第 30ms：屏幕未绘制，等待中，setTimeout 开始执行并设置 left=3px；
6. 第 33.4ms：屏幕开始绘制，屏幕上的元素向左移动了 3px，setTimeout 未执行，继续等待中
7. ...

从上面的绘制过程中可以看出，屏幕没有绘制 left=2px 的那一帧，元素直接从 left=1px 的位置跳到了 left=3px 的位置，也就是 丢帧现象 ，这种现象就会引起动画卡顿。

setInterval

1. setInterval 的回调函数调用之间的实际延迟小于代码中设置的延迟，因为回调函数执行所需的时间 “消耗” 了间隔的一部分， 如果回调函数执行时间过长、执行次数多的话，误差也会越来越大 ；
2. 嵌套的 setTimeout 可以 保证固定的延迟 ；

requestAnimationFrame

requestAnimationFrame 不同于 setTimeout 或 setInterval，由系统来决定回调函数的执行时机，会请求浏览器在 下一次重新渲染之前 执行回调函数，也就是 requestAnimationFrame 可以 保证在每一帧开始时 执行回调。
requestAnimationFrame 回调执行的时间间隔都会紧跟屏幕刷新一次所需要的时间，requestAnimationFrame 能够保证回调函数在每一帧内只渲染一次，但是如果这一帧有太多任务执行，还是会造成卡顿，因此只能 保证重新渲染的时间间隔最短是屏幕的刷新时间 。

requestAnimationFrame 的回调执行次数依据显示器的刷新率，显示器有固定的刷新频率（60Hz 或 75Hz），也就是一般为每秒刷新 60 次，每次刷新花费 16.6 ms，也就是说每秒最大只能重绘 60/75 次，requestAnimationFrame 的基本思想就是与这个刷新频率保持同步，使用这个 API，一旦页面不处于浏览器的当前标签，就会自动停止刷新。

1. CPU 节能：使用 setTimeout 实现的动画，当页面被隐藏或最小化时，setTimeout 仍然在后台执行动画任务，由于此时页面处于不可见或不可用状态，刷新动画是没有意义的，完全是浪费 CPU 资源。而 requestAnimationFrame 则完全不同，**当页面处于未激活的状态下，该页面的屏幕刷新任务也会被系统暂停，因此跟着系统步伐走的 requestAnimationFrame 也会停止渲染**，当页面被激活时，动画就从上次停留的地方继续执行，有效节省了 CPU 开销。
2. 因为 requestAnimationFrame 可 **保证每个刷新间隔内，函数只被执行一次** ，这样能保证流畅性，也能更好的节省函数执行的开销。因为一个刷新间隔内执行多次是没有意义的，因为显示器每隔 16.6ms 刷新一次进行重绘，在这个间隔内多次执行回调并不会在屏幕上体现出来。

13. 代码题 !!!!

解答

```
//1 promise构造函数的参数函数
new Promise(resolve => {
  resolve(1); //2
  Promise.resolve({
    //3  Promise.resolve
    then: function (resolve, reject) {
      //4 内then函数
      console.log(2);
      resolve(3); //8
    }
  }).then(t => console.log(t)); //9 内then1函数
  console.log(4); // 5
}).then(t => console.log(t)); //6  外then函数
console.log(5); // 7
```

八、面向对象

---

1. 对象创建的方式有哪些 !!!

- 对象字面量 `{}`，需要手动创建，适用于临时对象变量
- 内置构造函数 `new Object()`，复用性差，容易创建多个相同内容的对象，造成代码冗余
- 工厂模式，无法识别对象是由哪个工厂创造的，相同工厂产出的实例的原型不是同一个

```
function Person(name, age) {
  const obj = {};
  obj.name = name;
  obj.age = age;
  return obj;
}
const person = Person("dz", 23);
const person1 = Person("dz1", 24);
console.log(person instanceof Person); // false
console.log(person1.__proto__ ===
person.__proto__); // false
```

- 构造函数

  简化了工厂模式的加工过程，并且通过实例化对象，可以知道该对象的标识。

  能识别是被哪一个构造函数创造的。但是构造函数内部存在方法，就意味着每次创建实例时

  都会重新为每个实例创建方法，没有实现方法的复用。

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.sayname = () => {
    // 如果使用的是普通函数，在通过 obj.sayname 调用时是没有区别的
    // 但是如果将 obj.sayname 保存在一个变量 sayname 中
    // 这时通过 sayname() 直接调用，sayname 里的 this 就不是指向 obj 了
    // 所以这里使用了 箭头函数，让 this 一直为构造函数创建对象后的对象
    console.log(this.name);
  };
}
const p1 = new Person("dz1", 23);
const p2 = new Person("dz2", 24);
console.log(p1 instanceof Person); // true
```

- 原型模式

  在构造函数的原型上添加属性和方法，每一个实例都能共享原型上的属性和方法。

  避免了重复创建相同的方法。但是存在一个问题，所有的属性和方法都是共享的，因此每一个实例
  都可能更改原型里面的属性，也就导致后面创建的对象包含的属性会覆盖上一次创建的对象的属性，
  即每个对象对原型上属性的修改会同步到所有的对象。

```
function Person(name, age) {
  Person.prototype.name = name;
  Person.prototype.age = age;
  Person.prototype.likes = ["apple", "banana",
"watermelon"];
  Person.prototype.sayname = () => {
    // 这里的箭头函数与构造函数同理
    console.log(this.name);
  };
}
const p1 = new Person("dz", 23);
const p2 = new Person("dz1", 24);
p1.likes.pop();
console.log(p1.name === p2.name); // true
console.log(p1.likes); // ['apple', 'banana']
console.log(p2.likes); // ['apple', 'banana']
```

- 构造函数+原型模式的组合模式

  结合了构造函数和原型模式的优点，每个实例都拥有自己的属性和方法，以及共享相同的方法。

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype.sayname = funtion () {
  console.log(this.name);
};
const p1 = new Person("dz", 23);
const p2 = new Person("dz2", 24);
console.log(p1.name, p2.name);
p1.sayname();
sayname = p2.sayname;
// 如果将对象的方法提取出来的话内部的 this 指向会改变
// 只能通过绑定 this 的方法来调用
sayname();
// sayname.call(p32);
```

- 动态原型模式

```
function Person4(name, age) {
  this.name = name;
  this.age = age;
  console.log(this.__proto__,
Person4.prototype);
  if (typeof this.sayname !== "function") {
    Person4.prototype.sayname = () => {
      console.log(this.name);
    };
    /*
    // 改为普通函数后就能正常执行
    Person4.prototype.sayname = function () {
      console.log(this.name);
    };
    */
  }
  console.log(this.__proto__,
Person4.prototype);
}
// 这个对象创建时原型上没有 sayname 方法，所以在原型上
添加了sayname
// 并且因为 sayname 使用的是箭头函数，所以 this 一直
指向为 p41
const p41 = new Person4("dz", 23);
// 这个对象创建时原型上有了 sayname 方法，所以不再在原
型上添加
// 因此 p42 使用的 sayname 是在 p41 创建时候添加进去
的
// 并且由于使用的是箭头函数所以 this 指向为 p41而非
p42
const p42 = new Person4("dz2", 24);
p41.sayname(); // dz
p42.sayname(); // dz2

function Person4(name, age) {
  this.name = name;
  this.age = age;
  // p41 { test: '1' } { test: '1' }
  // p42 { constructor: [Function: Person4],
sayname: [Function: sayname] } { constructor:
[Function: Person4], sayname: [Function:
sayname] }
  console.log(this.__proto__,
Person4.prototype);
  if (typeof this.sayname !== "function") {
    // 我们知道，在 new 中继承和放到构造函数加工是两个
不同的步骤

    // 因此在进入这里之前 p41 就已经继承了修改前原来的
```

```
    // 因此在进入这里之前 p41 就已经继承了修改前原来的
原型
    // 但是这里直接修改了 Person4 的原型属性为一个新
的引用数据类型
    // 因此 Person4.prototype 所指向的栈中的位置保存
的地址变为了一个'新的内存的地址'
    // 而此时 p41.__proto__ 指向的栈中的位置保存的内
存地址仍然是'原来那个地址'
    // 所以 p41.__proto__ 就变得孤立不再是 Person4
的实例了

    // 而后来 p42 初始化的时候
    // 因为 prototype 已经被更改了，并且上面有
sayname 方法，所以不再进入这个 if
    // 所以 p42.sayname 能够正常输出

    Person4.prototype = {
      constructor: Person4,
      sayname: function () {
        console.log(this.name);
      }
    };
  }
  // p41 { test: '1' } { constructor: [Function:
Person4], sayname: [Function: sayname] }
  // p42 { constructor: [Function: Person4],
sayname: [Function: sayname] } { constructor:
[Function: Person4], sayname: [Function:
sayname] }
  console.log(this.__proto__,
Person4.prototype);
}
Person4.prototype.test = "1";
const p41 = new Person4("dz", 23);
const p42 = new Person4("dz2", 24);
try {
  p41.sayname();
} catch (e) {
  console.log(e.message); // p41.sayname is not
a function
}
p42.sayname(); // dz2
```

- 寄生构造函数

  创建一个函数,基于一个已有的类型，在实例化时对实例化的对象进行扩展,然后再返回新创建的对象。这个对象与生产他的这个函数并入任何关系，构造该对象时只是借助了 new 方法的第二步，
  也就是将继承的对象放入构造函数加工这一步，它并不返回继承后

的对象，而是返回构造函数返回的对象或函数。

```
function SpecialArray() {
  var array = new Array();
  array.push.apply(array, arguments);
  array.toPipedString = function () {
    return this.join("|");
  };
  return array;
}
var colors = new SpecialArray("red", "green",
"pink");
console.log(colors.toPipedString()); //
red|green|pink
console.log(colors instanceof SpecialArray); //
false
```

- 稳妥构造函数模式

  当我们使用构造函数又或是工厂模式创建一个对象并给它添加属性时，我们可以在后期为它添加新的方法，并通过 this.属性名 的方式访问到它身上的值。其实这个模式和工厂模式差不多，但是工厂模式会给加工的对象添加属性，还是可以通过后期添加方法的形式来访问。

  但如果使用稳妥构造函数，除了一开始创建对象时添加方法，不然在后期任何时候对它添加方法都是不能访问到一开始初始化的值的。

```
function Person(name, age, gender) {
  let obj = new Object();
  obj.sayname = function () {
    console.log(name);
  };
  return obj;
}
let person = Person("stan", 0000, "male");
person.sayname(); // stan
```

- class

  constructor 是构造方法,类似构造函数, 定义这个方法里面的内容都是实例自身的属性和方法, 不会被其他实例共享, 而写在 constructor 外的 sayname 表示原型上的方法, 是会被共享的.

  static 表示静态，加了 static 的函数不会挂载到 prototype 上,而是挂载到 Person 这个对象上 。

```
class Person {
  constructor(name, age) {
    // constructor构造函数
    this.name = name;
    this.age = age;
  }
  sayname() {
    //原型上的
    console.log(this.name);
  }
  static sayAge() {
    console.log(this.age);
  }
}
const per = new Person("dz", 23);
per.sayname(); // -> dz
Person.sayAge(); // 23
```

2. 对象继承的方式有哪些 !!!!

- 原型链继承

  子构造函数.prototype = new 父构造函数()
  子构造函数.prototype = Object.create(父构造函数)

  注意：修改了子构造函数的原型后，其原型上的 constructor 属性也会修改，最好让其重新指回子构造函数，子构造函数.prototype.constructor = 子构造函数

  !!!!注意：原型链存在缺点多个实例对引用类型的操作会被篡改。

```
function SuperType() {
  this.colors = ["red", "blue", "green"];
  this.a = 1;
}
function SubType() {}

SubType.prototype = new SuperType();

var instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors);
//"red,blue,green,black"
console.log(++instance1.a); // 2

var instance2 = new SubType();
console.log(instance2.colors);
//"red,blue,green,black"
console.log(instance2.a); // 1
```

- 借用构造函数配合 call 继承

  核心代码是 SuperType.call(this)，创建子类实例时调用 SuperType 构造函数，于是 SubType 的每个实例都会将 SuperType 中的属性复制一份。

  只能继承父构造函数的实例属性，不能继承原型属性/方法。
  每个子类都有父类**实例函数**的副本，没有实现复用，影响性能

```
function SuperType() {
  this.color = ["red", "green", "blue"];
}
function SubType() {
  //继承自SuperType
  SuperType.call(this); // 这是继承的关键语句
}
var instance1 = new SubType();
instance1.color.push("black");
console.log(instance1.color);
//"red,green,blue,black"

var instance2 = new SubType();
console.log(instance2.color); //"red,green,blue"
```

- 组合继承

```
function SuperType(name) {
  this.name = name;
  this.colors = ["red", "blue", "green"];
}
SuperType.prototype.sayName = function () {
  console.log(this.name);
};

function SubType(name, age) {
  // 继承属性
  // 第二次调用SuperType()
  SuperType.call(this, name);
  this.age = age;
}

// 继承方法
// 构建原型链
// 第一次调用SuperType()
SubType.prototype = new SuperType();
// 重写SubType.prototype的constructor属性，指向自己的构造函数SubType
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function () {
  console.log(this.age);
};

let instance1 = new SubType("Nicholas", 29);
instance1.colors.push("black");
console.log(instance1.colors);
//"red,blue,green,black"
instance1.sayName(); //"Nicholas";
instance1.sayAge(); //29

let instance2 = new SubType("Greg", 27);
alert(instance2.colors); //"red,blue,green"
instance2.sayName(); //"Greg";
instance2.sayAge(); //27
```

第一次调用 SuperType 时，给 SubType.prototype 写入了两个属性 name,color

第二次调用 SuperType 时，给 instance1 写入了两个属性 name, color

一个对象上有两个重复的属性，一个是复制父构造函数的，一个是继承于父构造函数的原型链的。依据就近原则，实例上的属性就屏蔽了其原型对象的两个重复属性。

- 原型式继承

  类似于 Object.create 函数，object()对传入其中的对象执行了一次浅复制，将构造函数 F 的原型直接指向传入的对象。

  缺点：

  - 原型链继承多个实例的引用类型属性指向相同，存在篡改的可能。
  - 无法传递参数。

```
function object(obj) {
  function F() {}
  F.prototype = obj;
  return new F();
}
```

- 寄生式继承

  在原型式继承的基础上，增强对象，返回构造函数。

  缺点与原型式继承相同。

```
function createAnother(original) {
  var clone = object(original); // 通过调用
object() 函数创建一个新对象
  clone.sayHi = function () {
    // 以某种方式来增强对象
    alert("hi");
  };
  return clone; // 返回这个对象
}
```

- 寄生组合式继承 !!!!

```
function Father(name) {
  this.name = name;
  this.hobbies = [];
}
Father.prototype.sayName = function () {
  console.log(this.name);
};
function Son(name, age) {
  // 调用父类构造函数，复制父类的实例属性
  Father.call(this, name);
  // 添加自己的实例属性
  this.age = age;
}
// 让子类的原型对象的原型继承于父类
// 这里相当于
Object.setPrototypeOf(Son.prototype,
Father.prototype)
// 即 Son.prototype.__proto__ = Father.prototype
Son.prototype = Object.create(Father.prototype);
Son.prototype.constructor = Son;
// 可以封装为下面的函数
`function inheritPrototype(subType, superType){
  let prototype =
Object.create(superType.prototype); // 创建对象，
创建父类原型的一个副本
  prototype.constructor = subType;
     // 增强对象，弥补因重写原型而失去的默认的
constructor 属性
  subType.prototype = prototype;
     // 指定对象，将新创建的对象赋值给子类的原型
}`;
Son.prototype.sayAge = function () {
  console.log(this.age);
};
```

- 混入方式继承多个对象

```
function MyClass() {
  SuperClass.call(this);
  otherSuperClass.call(this);
}

// 继承一个类
MyClass.prototype =
Object.create(SuperClass.prototype);
// 混入其它类原型上的属性和方法
Object.assign(MyClass.prototype,
otherSuperClass.prototype);
// 重新制定 constructor
MyClass.prototype.constructor = MyClass;
MyClass.prototype.myMethod = function () {
  // do something;
};
```

- ES6 类继承 extends !!!!

  extends 的核心代码

```
function _inherits = function(Sub, Sup) {
  Sub.prototype = Object.create(
    Sup && Sup.prototype,
    {
      constuctor: {
        value: Sub,
        enumerable: false,
        configuable: true,
        writable: true,
      }
    }
  );
  if(Sup) {
    Object.setPrototypeOf ?
    Object.setPrototypeOf(Sub, Sup) :
    Sub.__proto__ = Sup;
  }
}
```

关于继承牢记下面的原型链

```
// 1、构造器原型链
Child.__proto__ === Parent; // true
Parent.__proto__ === Function.prototype; // true
Function.prototype.__proto__ === Object.prototype; //
true
Object.prototype.__proto__ === null; // true
// 2、实例原型链
child.__proto__ === Child.prototype; // true
Child.prototype.__proto__ === Parent.prototype; // true
Parent.prototype.__proto__ === Object.prototype; // true
Object.prototype.__proto__ === null; // true
```

九、扩展

1 Web Worker

Web Worker 使用其他工作线程从而独立于主线程之外，它可以执行任务耳部感染用户界面。一个 worker 线程可以将消息发送到创建它的 Javascript 代码。主线程通过 worker.postMessage(msg) 的方式向 worker 线程发送消息，worker 线程通过监听 message 事件接收主线程发送的消息。

```
// main.js
const worker = new Worker("worker.js");
worker.postMessage(msg); // 向 worker 线程发送消息
worker.onmessage = function (e) {
  console.log(`from work ${e.data}`);
};

// worker.js
onmessage = function (e) {
  // 监听消息事件
  console.log(e.data);
  postMessage("receive"); // 向主线程发送消息
};
```

在 worker 内不能操作 DOM 节点，也不能使用 window 对象的默认方法和属性，但能使用 window 对象之下的 WebSockets 、indexedDB。