## 节流与防抖的原理和应用场景

防抖：当一个动作连续触发，只执行最后一次

节流：限制一个动作在一段时间内只能执行一次

## JS 数据类型 ?

### 数据类型主要包括两部分：

基本数据类型(也叫简单数据类型)：Number 、String、Boolean、BigInt、Symbol、Null、Undefined

引用数据类型(复杂数据类型)： 用Object代表，普通对象，数组，正则，日期，Math数学函数都属于Object。

两大类的**本质区别**：基本数据类型和引用数据类型它们在**内存中的存储方式不同**。

基本数据类型是直接存储在栈内存中的简单数据段，占据空间小，属于被频繁使用的数据。

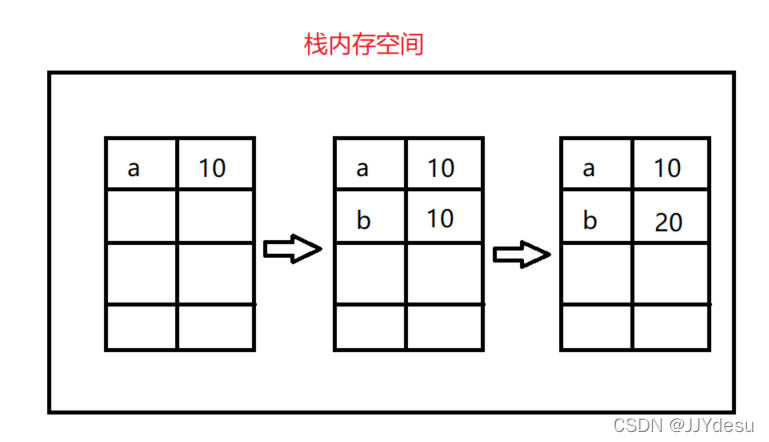
引用数据类型是存储在堆内存中，占据空间大。引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址，当解释器寻找引用值时，会检索其在栈中的地址，取得地址后从堆中获得实体。

加分回答： Symbol是ES6新出的一种数据类型，这种数据类型的特点就是没有重复的数据，可以作为object的key。

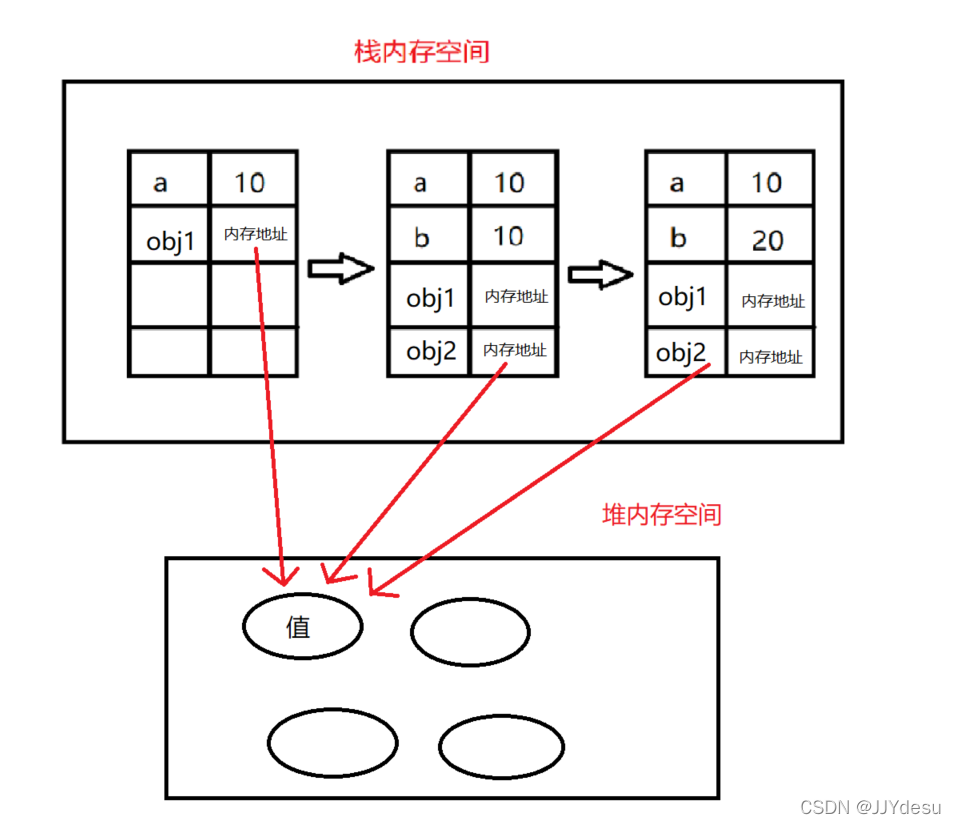
BigInt是ES10新出的一种数据类型，这种数据类型的特点就是数据涵盖的范围大，能够解决超出普通数据类型范围报错的问题。

### 存储地址

对**基本类型**而言，它的变量值是直接存储在**栈内存**中的



对**引用类型**而言，声明变量的时候，**栈内存存储的是值所在的内存地址**，而真正的值存放在**堆内存**中的



所以直接复制一份对象，其实只是把obj1中的内存地址复制了过来，改变obj2的时候根据obj2存放的内存地址可以找到和obj1在堆内存中同样的位置，所以obj1的内容也会同时改变。

### 总结

基本类型和引用类型的区别如下：

**基本类型变量存放值，引用类型变量存放内存地址。**

js中不能直接访问保存在堆内存中的对象，所以访问一个对象的时候，都是先在栈内存中得到该对象在堆内存中的地址，再根据地址访问堆内存中的对象值。而基本类型直接访问栈内存中的值即可。

基本类型的值大小固定，引用类型的值大小会改变。（所以一个放在栈，一个放在堆）

PS：这也解释了为什么对象作为函数参数传入进去的时候，在函数内部改变对象会体现在外部的原因。还有就是为什么const定义的数组和对象的值可以被改变。

## 判断一个值是什么类型有哪些方法？

* typeof 运算符
* instanceof 运算符
* Object.prototype.toString 方法

## null 和 undefined 的区别？

null 表示一个对象被定义了，值为“空值”；

undefined 表示不存在这个值。

（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。

## 怎么判断一个变量arr的话是否为数组（此题用 typeof 不行）？

arr instanceof Array  
arr.constructor == Array  
Object.protype.toString.call(arr) == ‘[Object Array]’

## “ ===”、“ ==”的区别？

==，当且仅当两个运算数相等时，它返回 true，即不检查数据类型  
===，只有在无需类型转换运算数就相等的情况下，才返回 true，需要检查数据类型

## 解释一下 JavaScript 的同源策略？

概念:同源策略是客户端脚本（尤其是Netscape Navigator2.0，其目的是防止某个文档或脚本从多个不同源装载。  
这里的同源策略指的是：协议，域名，端口相同，同源策略是一种安全协议。  
指一段脚本只能读取来自同一来源的窗口和文档的属性。

## javascript的内存(垃圾)回收机制？

* 垃圾回收器会每隔一段时间找出那些不再使用的内存，然后为其释放内存
* 一般使用标记清除方法(mark and sweep), 当变量进入环境标记为进入环境，离开环境标记为离开环境垃圾回收器会在运行的时候给存储在内存中的所有变量加上标记，然后去掉环境中的变量以及被环境中变量所引用的变量（闭包），在这些完成之后仍存在标记的就是要删除的变量了
* 还有引用计数方法(reference counting), 在低版本IE中经常会出现内存泄露，很多时候就是因为其采用引用计数方式进行垃圾回收。引用计数的策略是跟踪记录每个值被使用的次数，当声明了一个 变量并将一个引用类型赋值给该变量的时候这个值的引用次数就加1，如果该变量的值变成了另外一个，则这个值得引用次数减1，当这个值的引用次数变为0的时 候，说明没有变量在使用，这个值没法被访问了，因此可以将其占用的空间回收，这样垃圾回收器会在运行的时候清理掉引用次数为0的值占用的空间。
* 在IE中虽然JavaScript对象通过标记清除的方式进行垃圾回收，但BOM与DOM对象却是通过引用计数回收垃圾的， 也就是说只要涉及BOM及DOM就会出现循环引用问题。

## js内存泄漏以及如何避免

### 什么是内存泄漏

内存泄漏可以定义为程序不再使用或不需要的一块内存，但是由于某种原因没有被释放仍然被不必要的占有。在代码中创建对象和变量会占用内存，但是javaScript是有自己的内存回收机制，可以确定那些变量不再需要，并将其清除。但是当你的代码存在逻辑缺陷的时候，你以为你已经不需要，但是程序中还存在着引用，导致程序运行完后并没有合适的回收所占用的空间，导致内存不断的占用，运行的时间越长占用的就越多，随之出现的是，性能不佳，高延迟，频繁崩溃。 在深入了解内存泄漏之前，我们需要知道一下几点：

* 内存生命周期
* 内存管理系统
* 垃圾回收算法

### 内存生命周期

不管什么程序语言，内存生命周期基本是一致的：

1. 分配你所需要的内存
2. 使用分配到的内存（读、写）
3. 不需要时将其释放\归还

### 常见的四种内存泄漏

1.全局变量

2.被遗忘的定时器和回调函数

3.DOM引用

4.闭包(闭包本身没有错,不会引起内存泄漏.而是使用错误导致.)

## 函数柯里化

什么叫函数柯里化？其实就是将使用多个参数的函数转换成一系列使用一个参数的函数的技术。还不懂？来举个例子。

function add(a, b, c) {

return a + b + c

}

add(1, 2, 3)

let addCurry = curry(add)

addCurry(1)(2)(3)

现在就是要实现 curry 这个函数，使函数从一次调用传入多个参数变成多次调用每次传一个参数。

function curry(fn) {

let judge = (...args) => {

if (args.length == fn.length) return fn(...args)

return (...arg) => judge(...args, ...arg)

}

return judge

}

## dom事件流原理是什么，分为那几个阶段？

原理

DOM事件流相当于将事件捕获与事件冒泡两者结合起来，事件触发的顺序是先进行事件捕获阶段 => 目标元素阶段 => 事件冒泡阶段

事件捕获 处于目标阶段 事件冒泡

## js的两种回收机制

1标记清除（mark and sweep）

当变量进入执行环境时,就将这个变量标记为“进入环境”,当变量离开环境时会被标记“离开环境”,离开环境的变量内存被释放.

function f1(){

//被标记已进入执行环境

var a=1

var b=2

}

f1() //执行完毕，a，b被标记离开执行环境，内存释放

2引用计数（reference counting）

跟踪记录每个值被引用的次数，当某个值的引用次数变为0时，说明没有方法在访问该值了，则可将其占用的内存收回.

function f1(){

//跟踪a的引用计数

var a={} //a的引用次数 0

var b=a //a的引用次数 1

var c=a //a的引用次数 2

var b={} //a的引用次数 1

var c=[] //a的引用次数 0

}

3手工

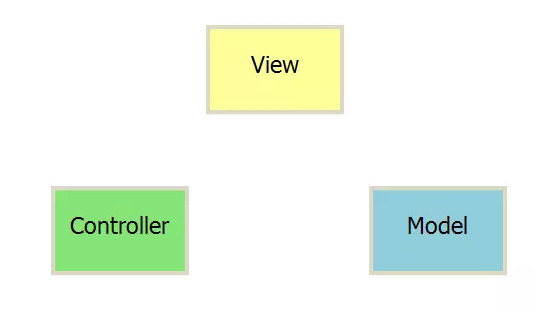
直接置空，GC下次再运行时会删除这些值

a=null

## JS常用设计模式（MVC、MVP、MVVM及其他设计模式）

### MVC

MVC模式的意思是，软件可以分成三个部分。

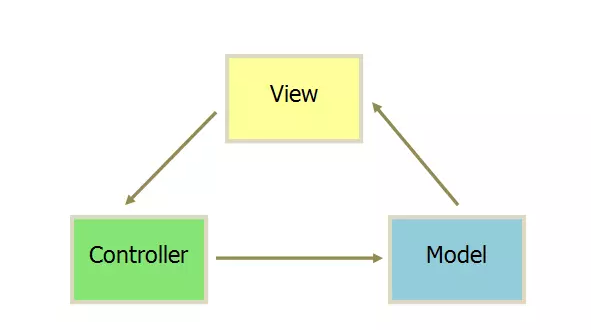


视图（View）：用户界面。

控制器（Controller）：业务逻辑

模型（Model）：数据保存

各部分之间的通信方式如下。



View 传送指令到 Controller

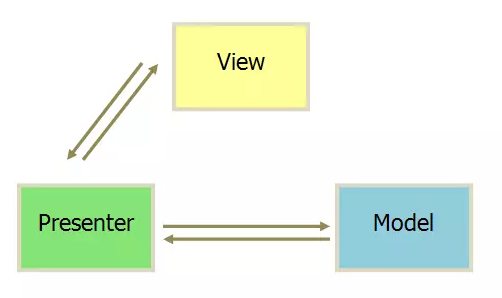
Controller 完成业务逻辑后，要求 Model 改变状态

Model 将新的数据发送到 View，用户得到反馈

所有通信都是单向的。

### MVP

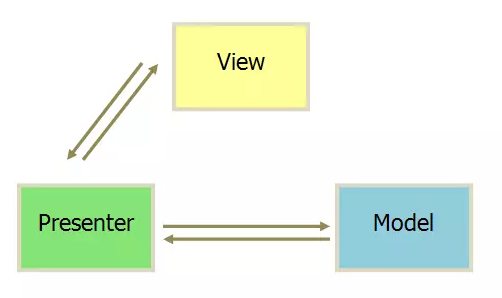
MVP 模式将 Controller 改名为 Presenter，同时改变了通信方向。

1. 各部分之间的通信，都是双向的。

2. View 与 Model 不发生联系，都通过 Presenter 传递。

3. View 非常薄，不部署任何业务逻辑，称为"被动视图"（Passive View），即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。

MVP 模式将 Controller 改名为 Presenter，同时改变了通信方向。

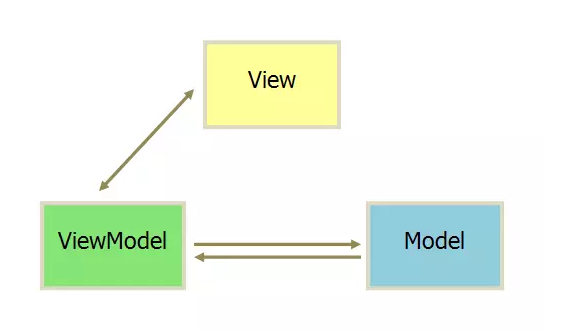
1. 各部分之间的通信，都是双向的。

2. View 与 Model 不发生联系，都通过 Presenter 传递。

3. View 非常薄，不部署任何业务逻辑，称为"被动视图"（Passive View），即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。

### MVVM

MVVM 模式将 Presenter 改名为 ViewModel，基本上与 MVP 模式完全一致。



唯一的区别是，它采用双向绑定（data-binding）：View的变动，自动反映在 ViewModel，反之亦然。[Angular](https://angularjs.org/" \t "_blank)和[Ember](http://emberjs.com/" \t "_blank)都采用这种模式。

## 闭包问题

得分点： 变量背包、作用域链、局部变量不销毁、函数体外访问函数的内部变量、内存泄漏、内存溢出、形成块级作用域、柯里化、构造函数中定义特权方法、Vue中数据响应式Observer

标准回答：

**一个函数和词法作用域的引用捆绑在一起，这样的组合**就是闭包（closure）。一般就是一个函数A，return其内部的函数B，被return出去的B函数能够在外部访问A函数内部的变量，这时候就形成了一个B函数的变量背包，A函数执行结束后这个变量背包也不会被销毁，并且这个变量背包在A函数外部只能通过B函数访问。

闭包形成的原理：作用域链，当前作用域可以访问上级作用域中的变量

闭包解决的问题：能够让函数作用域中的变量在函数执行结束之后不被销毁，同时也能在函数外部可以访问函数内部的局部变量。

闭包带来的问题：由于垃圾回收器不会将闭包中变量销毁，于是就造成了内存泄露，内存泄露积累多了就容易导致内存溢出。

加分回答： 闭包的应用，能够模仿块级作用域，能够实现柯里化，在构造函数中定义特权方法、Vue中数据响应式Observer中使用闭包等。

### 闭包的应用

封装私有变量、创建模块化的代码、实现柯里化和高阶函数等

## ajax、axios 前后端通信的过程（原生的ajax怎么写）

原生ajax请求步骤get：

1.创建核心的对象（new xhrhttprequest对象）

2.编写回调函数（判断是否请求成功 状态码等于4或者等于200）

3.调用open方法 设置请求方式和请求路径

4.调用send方法 发送

post：要多设置一个请求头setrequestheader（在发送的前面）

## 跨域问题（具体实现）

答：跨域解决方式：

1.jsonp（script这个标签本来就有跨域的能力，然后将ajax中datatype中类型改为jsonp）缺点只支持get

2.利用cors设置头 允许跨域（头有哪些）优点：支持get和post两种

使用普通的XMLHttpRequest发起请求和获得数据，比起JSONP有更好的错误处理

3.利用proxy代理（具体实现不管是否跨域，在自己服务器上中转一次，让它允许跨域，类似于一个中转站的意思）

## map和forEach的区别

forEach返回undefined，map会返回新的数组。

forEach没办法中止循环，但是map可以通过返回false或者出错来中止。

## call、apply以及bind的区别。

call、aply和bind都可以改变函数中的this指向。

**this指向：**

**this永远指向最后一个调用它的那个对象**

* apply和call基本类似，他们的区别只是传入的参数不同。
* apply传入的参数是包含多个参数的数组
* call传入的参数是若干个参数列表
* bind方法会创建一个新的函数，当被调用的时候，将其this关键字设置为提供的值，我们必须手动去调用

### apply：

apply() 方法调用一个函数，其具有一个指定的this值，以及作为一个数组（或者类似数组的对象）提供的参数，fun.apply(thisArg, [argsArray])

thisArg：在fun函数运行时指定的this值。指定this的值并不一定是函数执行时真正的this值，如果是原始值的this会指向该原始值的自动包装对象。

argsArray：一个数组或者类数组对象，其中的数组元素将作为单独的参数传给fun函数。参数为null或者undefined，则表示不需要传入任何参数。

### call：

call() 调用一个函数，其具有一个指定的this值，以及若干个参数列表，fun.call(thisArg, arg1, arg2, …)

thisArg：在fun函数运行时指定的this值。指定this的值并不一定是函数执行时真正的this值，如果是原始值的this会指向该原始值的自动包装对象。

arg1, arg2, …：若干个参数列表

### bind：

bind() 创建一个新的函数，当被调用时，将其this的关键字设置为提供的值，在调用新函数时，在任何提供一个给定的参数序列。

bind创建了一个新函数，必须手动去调用。

## 数组常用函数

join()：用指定的分隔符将数组每一项拼接为字符串

push() ：向数组的末尾添加新元素

pop()：删除数组的最后一项

shift()：删除数组的第一项

unshift()：向数组首位添加新元素

slice()：按照条件查找出其中的部分元素

splice()：对数组进行增删改

fill(): 方法能使用特定值填充数组中的一个或多个元素

filter():“过滤”功能

concat()：用于连接两个或多个数组

indexOf()：检测当前值在数组中第一次出现的位置索引

lastIndexOf()：检测当前值在数组中最后一次出现的位置索引

every()：判断数组中每一项都是否满足条件

some()：判断数组中是否存在满足条件的项

includes()：判断一个数组是否包含一个指定的值

sort()：对数组的元素进行排序

reverse()：对数组进行倒序

forEach()：ES5 及以下循环遍历数组每一项

map()：ES6 循环遍历数组每一项

copyWithin():用于从数组的指定位置拷贝元素到数组的另一个指定位置中

find():返回匹配的值

findIndex():返回匹配位置的索引

toLocaleString()、toString():将数组转换为字符串

flat()、flatMap()：扁平化数组

entries() 、keys() 、values():遍历数组

## 事件循环

JavaScript是单线程执行模型，执行的时候将会区分为主线程和任务队列。主线程执行完毕，会从任务队列中读取新的任务放入主线程进行执行，这个读取过程是**循环读取**，所以也叫**事件循环**。

任务队列分为宏任务和微任务，同层次，**先执行微任务，再执行宏任务**。

#### 宏任务和微任务

事件循环由宏任务和在执行宏任务期间产生的所有微任务组成。完成当下的宏任务后，会立刻执行所有在此期间入队的微任务。

这种设计是为了给紧急任务一个插队的机会，否则新入队的任务永远被放在队尾。区分了微任务和宏任务后，本轮循环中的微任务实际上就是在插队，这样微任务中所做的状态修改，在下一轮事件循环中也能得到同步。

常见的**宏任务**有：

script（整体代码）

setTimout

setInterval

setImmediate(node 独有)

requestAnimationFrame(浏览器独有)

IO

UI render（浏览器独有）

常见的**微任务**有：

process.nextTick(node 独有)

Promise.then()

Object.observe

MutationObserver

## get和post请求区别

GET在浏览器回退时是无害的，而POST会再次提交请求。

GET产生的URL地址可以被添加书签(Bookmark)，而POST不可以。

GET请求会被浏览器访问，而post不能

GET请求只能进行url编码，而POST支持多种编码方式。

GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

GET请求在URL中传送的参数是有长度限制的，而POST没有。

对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

GET参数通过URL传递，POST放在Request body中。

GET产生一个TCP数据包；POST产生两个TCP数据包。

GET与POST都有自己的语义，不能随便混用。

## promise 与 async await

async/await是写异步代码的新方式，以前的方法有回调函数和Promise。

　　async/await是基于Promise实现的，它不能用于普通的回调函数。

　　async/await与Promise一样，是非阻塞的。

　　async/await使得异步代码看起来像同步代码，这正是它的魔力所在

函数前面多了一个aync关键字。await关键字只能用在aync定义的函数内。async函数会隐式地返回一个promise，该promise的reosolve值就是函数return的值。(示例中reosolve值就是字符串”done”)

为什么Async/Await更好？

1）使用async函数可以让代码简洁很多，不需要像Promise一样需要些then，不需要写匿名函数处理Promise的resolve值，也不需要定义多余的data变量，还避免了嵌套代码。

2）错误处理：

　　　　Async/Await 让 try/catch 可以同时处理同步和异步错误。在下面的promise示例中，try/catch 不能处理 JSON.parse 的错误，因为它在Promise中。我们需要使用 .catch，这样错误处理代码非常冗余。并且，在我们的实际生产代码会更加复杂

3）使用async/await的话，代码会变得异常简单和直观。

4）错误栈

　 如果 Promise 连续调用，对于错误的处理是很麻烦的。你无法知道错误出在哪里。

async/await中的错误栈会指向错误所在的函数。在开发环境中，这一点优势并不大。但是，当你分析生产环境的错误日志时，它将非常有用。这时，知道错误发生在makeRequest比知道错误发生在then链中要好。

5）调试

　 async/await能够使得代码调试更简单。2个理由使得调试Promise变得非常痛苦:

《1》不能在返回表达式的箭头函数中设置断点

《2》如果你在.then代码块中设置断点，使用Step Over快捷键，调试器不会跳到下一个.then，因为它只会跳过异步代码。

使用await/async时，你不再需要那么多箭头函数，这样你就可以像调试同步代码一样跳过await语句。

## 说一说promise是什么与使用方法？

得分点： pendding、rejected、resolved、微任务、then、catch、Promise.resolve()、Promise.reject()、Promise.all() Promise.any()、Promise.race()

标准回答：

Promise的作用：Promise是异步微任务，解决了异步多层嵌套回调的问题，让代码的可读性更高，更容易维护

Promise使用：Promise是ES6提供的一个构造函数，可以使用Promise构造函数new一个实例，Promise构造函数接收一个函数作为参数，这个函数有两个参数，分别是两个函数 resolve和reject resolve将Promise的状态由等待变为成功，将异步操作的结果作为参数传递过去； reject则将状态由等待转变为失败，在异步操作失败时调用，将异步操作报出的错误作为参数传递过去。 实例创建完成后，可以使用then方法分别指定成功或失败的回调函数，也可以使用catch捕获失败，then和catch最终返回的也是一个Promise，所以可以链式调用。

Promise的特点：

对象的状态不受外界影响（Promise对象代表一个异步操作，有三种状态）。 - pending（执行中） - Resolved（成功，又称Fulfilled） - rejected（拒绝） 其中pending为初始状态，fulfilled和rejected为结束状态（结束状态表示promise的生命周期已结束）。

一旦状态改变，就不会再变，任何时候都可以得到这个结果。 Promise对象的状态改变，只有两种可能（状态凝固了，就不会再变了，会一直保持这个结果）： - 从Pending变为Resolved - 从Pending变为Rejected

resolve 方法的参数是then中回调函数的参数，reject 方法中的参数是catch中的参数

then 方法和 catch方法 只要不报错，返回的都是一个fullfilled状态的promise

加分回答 ：Promise的其他方法：

Promise.resolve() :返回的Promise对象状态为fulfilled，并且将该value传递给对应的then方法.

Promise.reject()：返回一个状态为失败的Promise对象，并将给定的失败信息传递给对应的处理方法。

Promise.all()：返回一个新的promise对象，该promise对象在参数对象里所有的promise对象都成功的时候才会触发成功，一旦有任何一个iterable里面的promise对象失败则立即触发该promise对象的失败。

Promise.any()：接收一个Promise对象的集合，当其中的一个 promise 成功，就返回那个成功的promise的值。

Promise.race()：当参数里的任意一个子promise被成功或失败后，父promise马上也会用子promise的成功返回值或失败详情作为参数调用父promise绑定的相应句柄，并返回该promise对象。

## 说一说跨域是什么？如何解决跨域问题？

得分点： 同源限制、协议、域名、端口、CORS、node中间件、JSONP、postmessage

标准回答 跨域：当前页面中的某个接口请求的地址和当前页面的地址如果协议、域名、端口其中有一项不同，就说该接口跨域了。 跨域限制的原因：浏览器为了保证网页的安全，出的同源协议策略。

跨域解决方案

**cors**：目前最常用的一种解决办法，通过设置后端允许跨域实现。 res.setHeader(‘Access-Control-Allow-Origin’,\*); res.setHeader(“Access-Control-Allow-Methods”, “GET, PUT, OPTIONS,POST”);

**node中间件、nginx反向代理**：跨域限制的时候浏览器不能跨域访问服务器，node中间件和nginx反向代理，都是让请求发给代理服务器，静态页面面和代理服务器是同源的，然后代理服务器再向后端服务器发请求，服务器和服务器之间不存在同源限制。

**JSONP**：利用的原理是script标签可以跨域请求资源，将回调函数作为参数拼接在url中。后端收到请求，调用该回调函数，并将数据作为参数返回去，注意设置响应头返回文档类型，应该设置成javascript。

**postmessage**：H5新增API，通过发送和接收API实现跨域通信。

加分回答 跨域场景：前后端分离式开发、调用第三方接口

## 说一说JavaScript有几种方法判断变量的类型？

得分点： typeof、instanceof、Object.prototype.toString.call()（对象原型链判断方法）、 constructor (用于引用数据类型)

标准回答：

JavaScript有4种方法判断变量的类型，分别是typeof、instanceof、Object.prototype.toString.call()（对象原型链判断方法）、 constructor (用于引用数据类型)

typeof：常用于判断基本数据类型，对于引用数据类型除了function返回’function‘，其余全部返回’object’。

instanceof：主要用于区分引用数据类型，检测方法是检测的类型在当前实例的原型链上，用其检测出来的结果都是true，不太适合用于简单数据类型的检测，检测过程繁琐且对于简单数据类型中的undefined, null, symbol检测不出来。

constructor：用于检测引用数据类型，检测方法是获取实例的构造函数判断和某个类是否相同，如果相同就说明该数据是符合那个数据类型的，这种方法不会把原型链上的其他类也加入进来，避免了原型链的干扰。

Object.prototype.toString.call()：

适用于所有类型的判断检测，检测方法是Object.prototype.toString.call(数据) 返回的是该数据类型的字符串。

这四种判断数据类型的方法中，各种数据类型都能检测且检测精准的就是Object.prototype.toString.call()这种方法。

加分回答

instanceof的实现原理：验证当前类的原型prototype是否会出现在实例的原型链\_\_proto\_\_上，只要在它的原型链上，则结果都为true。因此，instanceof 在查找的过程中会遍历左边变量的原型链，直到找到右边变量的 prototype，找到返回true，未找到返回false。

Object.prototype.toString.call()原理：Object.prototype.toString 表示一个返回对象类型的字符串，call()方法可以改变this的指向，那么把Object.prototype.toString()方法指向不同的数据类型上面，返回不同的结果

## 模块化

### 1.什么是模块化

* 将一个复杂的程序，依据一定的规则(规范)封装成一个或多个块(文件), 并进行组合在一起
* 块的内部数据与实现是私有的, 只是向外部暴露一些接口(方法)与外部其它模块通信

模块化是一种将系统分离成独立功能部分的方法，可将系统分割成独立的功能部分，严格定义模块接口、模块间具有透明性。

### 2.模块化的进化过程

#### a.无模块时代

在ajax还未提出之前，js还只是用来在网页上进行表单校验、提交，对DOM渲染操作。

缺点：

全局变量污染

函数命名冲突

文件依赖顺序

#### b.模块雏形时代

2006年，ajax的概念被提出，前端拥有了主动向服务端发送请求并操作返回数据的能力，传统的网页向“富客户端”发展，出现了简单的功能对象封装。

#### c.namespace模式

优点: 减少了全局变量，解决命名冲突

缺点: 数据不安全(外部可以直接修改模块内部的数据)，模块名称会暴露在全局，存在命名冲突，依赖顺序问题。

#### d.自执行匿名函数（闭包）模式

优点：变量、方法全局隐藏，模块私有化

缺点：模块名称会暴露在全局，存在命名冲突，依赖顺序问题。

#### 面临的问题

从以上的尝试中，可以归纳出js模块化需要解决那些问题：

如何安全的包装一个模块的代码？（不污染模块外的任何代码）

如何唯一标识一个模块？

如何优雅的把模块的API暴漏出去？（不能增加全局变量）

如何方便的使用所依赖的模块？

### 3.模块化的规范

#### 1、CommonJS

2009年Nodejs发布，采用 CommonJS 模块规范。

特点：

每个文件都是一个模块实例，代码运行在模块作用域，不会污染全局作用域。

文件内通过require对象引入指定模块，通过exports对象来向往暴漏API，文件内定义的变量、函数，都是私有的，对其他文件不可见。

每个模块加载一次之后就会被缓存。

所有文件加载均是同步完成，加载的顺序，按照其在代码中出现的顺序。

模块输出的是一个值的拷贝，模块内部的变化不会影响该值。

缺点：模块同步加载，资源消耗和等待时间，适用于服务器编程

#### 2、AMD/RequireJS

Commonjs局限性很明显：

基于Node原生api在服务端可以实现模块同步加载，但是仅仅局限于服务端，客户端如果同步加载依赖的话时间消耗非常大，所以需要一个在客户端上基于Commonjs但是对于加载模块做改进的方案，于是AMD规范诞生了。

AMD是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义"。它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到所有依赖加载完成之后（依赖前置），这个回调函数才会运行。

RequireJS是一个工具库，主要用于客户端的模块管理。它的模块管理遵守AMD规范，RequireJS的基本思想是，通过define方法将代码定义为模块，通过require方法实现代码的模块加载。

特点：浏览器直接运行无需编译，异步加载，依赖关系清晰。

#### 3、CMD/SeaJS

CMD规范专门用于浏览器端，同样是受到Commonjs的启发，国内（阿里）诞生了一个CMD（Common Module Definition）规范。该规范借鉴了Commonjs的规范与AMD规范，在两者基础上做了改进。

与AMD相比非常类似，CMD规范（2011）具有以下特点：

define定义模块，require加载模块，exports暴露变量。

不同于AMD的依赖前置，CMD推崇依赖就近（需要的时候再加载）

推崇api功能单一，一个模块干一件事。

SeaJs是CMD规范的实现，跟RequireJs类似，CMD是SeaJs推广过程中诞生的规范。CMD借鉴了很多AMD和Commonjs优点。

#### AMD、CMD区别

AMD 推崇依赖前置

CMD 推崇依赖就近

### 4.ES6

2015年，ES6规范中，终于将模块化纳入JavaScript标准，从此js模块化被ECMA官方扶正，也是后来js的标准。

ES6中的模块化在CommonJS的基础上有所不同，关键字有import，export，default，as，from。

### CommonJS和ES6区别

CommonJS 模块输出的是一个值的拷贝，即原来模块中的值改变不会影响已经加载的该值。

ES6 模块输出的是值的只读引用，模块内值改变，引用也改变。

CommonJS 模块是运行时加载，加载的是整个模块，即将所有的接口全部加载进来。

ES6 模块是编译时输出接口，可以单独加载其中的某个接口。

## 前端模块化commonJS,CMD,AMD,UMD,es6 module

### 总结

CommonJS规范主要用于服务端编程，加载模块是同步的，不适合在浏览器环境，存在阻塞加载，浏览器资源是异步加载的，因此有了AMD、CMD解决方案。

AMD规范在浏览器环境中异步加载模块，而且可以并行加载多个模块。

CMD规范与AMD规范很相似，都用于浏览器编程，依赖就近，代码更简单。

ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

### 1、Commonjs

用于nodejs环境，它使用 require 引用和加载模块，exports 定义和导出模块，module 标识模块。有人试图将其引入浏览器环境，由于node服务器环境上文件总是位于服务器上，加载速度较快，因此使用同步加载即可；但浏览器需要从服务端下载文件，所以说如果用CommonJS的require方式加载模块，需要等代码模块下载完毕，并运行之后才能得到所需要的API。如果我们在某个代码模块里使用CommonJS的方法require了一个模块，而这个模块需要通过http请求从服务器去取，如果网速很慢，而CommonJS又是同步的，所以将阻塞后面代码的执行，从而阻塞浏览器渲染页面，使得页面出现假死状态。因此产生了异步加载，主要是AMD和CMD

* CommonJS 是一种用于服务器端的模块化规范，最初由Node.js采用并推广。
* 使用require()函数导入模块，使用module.exports导出模块。
* 同步加载模块，适用于服务器端的模块加载。
* 不能在浏览器端直接使用，需要通过工具转换成浏览器可用的代码。

### 2、AMD

用于浏览器环境，依赖前置，提前执行。主要有requirejs，它主要有两个接口：define 和 require。define 是模块开发者关注的方法，而 require 则是模块使用者关注的方法。

优点：1、首先，尽早执行依赖可以尽早发现错误。

2、尽早执行依赖通常可以带来更好的用户体验。

缺点：1、容易产生浪费。有些模块加载后未使用或者使用较晚

1. 开发成本高，代码的阅读和书写比较困难，模块定义方式的语义不顺畅；不符合通用的模块化思维方式，是一种妥协的实现。

* AMD 是一种用于浏览器端的模块化规范，最著名的实现是RequireJS。
* 使用define()函数定义模块，使用require()函数异步加载模块。
* 异步加载模块，适用于浏览器端的模块加载，可以提高页面加载速度。
* 适用于浏览器端的异步加载，可以并行加载多个模块。

### 3、CMD：

用于浏览器环境，依赖就近，延迟执行。主要实现为sea.js

CMD是SeaJS在推广过程中生产的对模块定义的规范，在Web浏览器端的模块加载器中，SeaJS与RequireJS并称，SeaJS作者为阿里的玉伯。CMD规范专门用于浏览器端，模块的加载是异步的，模块使用时才会加载执行。CMD规范整合了CommonJS和AMD规范的特点。在 Sea.js 中，所有 JavaScript 模块都遵循 CMD模块定义规范。

优点：依赖就近，延迟执行 可以很容易在 Node.js 中运行；

缺点：依赖 SPM 打包，模块的加载逻辑偏重

* CMD 也是一种用于浏览器端的模块化规范，最著名的实现是SeaJS。
* 使用define()函数定义模块，使用require()函数加载模块。
* 与AMD类似，但CMD更加倾向于依赖就近，需要使用时再加载。
* 适用于浏览器端的模块加载，更加符合国内开发者的使用习惯。

### 4、UMD

nodejs和浏览器环境都支持，是common和AMD的统一，能够兼容他们，这样我们在开发时，就不需要再去考虑依赖模块所遵循的规范了，UMD的出现就是为了解决这个问题。(没有兼容CMD吧)

* UMD是一种兼容多种模块规范的解决方案，可以在CommonJS、AMD和全局变量环境下使用。
* 通过判断当前环境，选择使用CommonJS、AMD或全局变量的方式导出模块。

### 5、ES6 Module

设计思想是尽量的静态化，使得编译时就能确定模块的依赖关系，以及导入和导出的变量，也就是所谓的"编译时加载"。主要语法是import和export

* ES6 模块是 ECMAScript 6 标准中新增的模块化规范，已成为 JavaScript 的官方模块化标准。
* 使用import关键字导入模块，使用export关键字导出模块。
* 支持静态分析，可以进行更好的优化。
* 可以在浏览器端和服务器端直接使用，不需要额外的转换工具。

### 6、对比

common引入的是一个拷贝，而es6module引入的是引用，可以实时变化，比如文件中有一个settimeout，延时后再调用可获得最新值，但是commonjs不能获取最新值。

commonjs或AMD，都是运行时加载，在编译阶段只判断语法层面对不对，如果引入了不存在的变量等只有在执行时才会发现。

ES6 可以因为是一边编译一边执行，在编译阶段会检查引入变量是否存在、引入模块是否存在等，问题更早暴露。

### 编译时和运行时知识点扩展

编译时

编译：将源代码翻译成机器能识别的语言（二进制）。计算机还是那个计算机，他至今还是只能看懂01,但随着时间的推移和人类的进步，逐渐出现了很多高级语言，高级语言之所以高级就在于我们可以用一种简单的方式比如if elese 来实现我们的代码逻辑，但最终在执行的时候计算机还是没办法识别的，这个时候就需要一个处理过程"编译"，将大家平时写的if else 翻译成机器可以看懂的语言。那负责编译的这部分，我们一般称之为 编译器。

那编译时就会做一些简单的翻译工作,比如检查你有没有粗心写错啥关键字了啊。会进行词法分析,语法分析之类的。就像个老师检查学生的作文中有没有错别字和病句一样.如果发现啥错误编译器就告诉你。开始编译时,如果有errors或者warning信息,都是编译器检查出来的。所谓这时的错误就叫编译时错误,这个过程中做的类型检查也就叫编译时类型检查,或静态类型检查(所谓静态嘛就是没把真把代码放内存中运行起来,而只是把代码当作文本来扫描下)。

运行时

所谓运行时就是代码跑起来了。被装载到内存中去了。(你的代码保存在磁盘上没装入内存之前是个文件。只有跑到内存中才变成活的)。而运行时类型检查就与前面讲的编译时类型检查(或者静态类型检查)不一样。不是简单的扫描代码.而是在内存中做些操作,做些判断以确定我们的程序是否存在错误。

## 原型链

每个对象（object）都有一个私有属性指向另一个名为原型（prototype）的对象

js分为函数对象和普通对象，每个对象都有\_\_proto\_\_属性，但是只有函数对象才有prototype属性

Object、Function都是js内置的函数, 类似的还有我们常用到的Array、RegExp、Date、Boolean、Number、String

属性\_\_proto\_\_是一个对象，它有两个属性，constructor和\_\_proto\_\_；

原型对象prototype有一个默认的constructor属性，用于记录实例是由哪个构造函数创建；

js原型、原型链默认遵从以下两个准则

* 准则1：原型对象（即Person.prototype）的constructor指向构造函数本身
* 准则2：实例（即per0）的\_\_proto\_\_和原型对象指向同一个地方

## 作用域链

当执行一段JavaScript代码（全局代码或函数）时，JavaScript引擎会创建为其创建一个作用域又称为执行上下文（Execution Context），在页面加载后会首先创建一个全局的作用域，然后每执行一个函数，会建立一个对应的作用域，从而形成了一条作用域链。每个作用域都有一条对应的作用域链，链头是全局作用域，链尾是当前函数作用域。

作用域链的作用是用于**解析标识符**，当函数被创建时（不是执行），会将this、arguments、命名参数和该函数中的所有局部变量添加到该当前作用域中，当JavaScript需要查找变量X的时候（这个过程称为变量解析），它首先会从作用域链中的链尾也就是当前作用域进行查找是否有X属性，如果没有找到就顺着作用域链继续查找，直到查找到链头，也就是全局作用域链，仍未找到该变量的话，就认为这段代码的作用域链上不存在x变量，并抛出一个引用错误（ReferenceError）的异常。

## js作用域的销毁、不立即销毁、不销毁

JavaScript中的函数执行会形成私有的作用域。

（1）作用域的销毁

　　一般情况下，函数执行形成一个私有的作用域，当执行完成后就销毁了->节省内存空间

（2）作用域的不立即销毁

function fn(){

var i=10;

return function(n){

console.log(n+i++);

　　}

}

fn()(15);//->先执行fn,有一个私有的变量i=10,返回一个堆内存地址 xxxfff111,我们发现这个地址还用到了一次,那么当前的这个fn形成的私有作用域（A）就不能立即销毁了,

　　　　　　xxxfff111(15)->输出25,A中的i变为11；当xxxfff111执行完了,发现这个地址没用了,浏览器就把A、xxxfff111都释放了

fn()(20);//->在执行fn的时候一切都从新开始了,和上面的步骤是一样的->输出30

（3）作用域的立即销毁

function fn(){

var i=10;

return function(n){

console.log(n+i++);

}

}

var f=fn();//->fn执行形成一个私有的作用域A,A中有一个私有的变量i=10,A中返回一个地址xxxfff11,被外面的f占用了,那么当前的A就不能销毁了

f(15);//->输出25,让A中的i=11

f(20);//->输出31,让A中的i=12

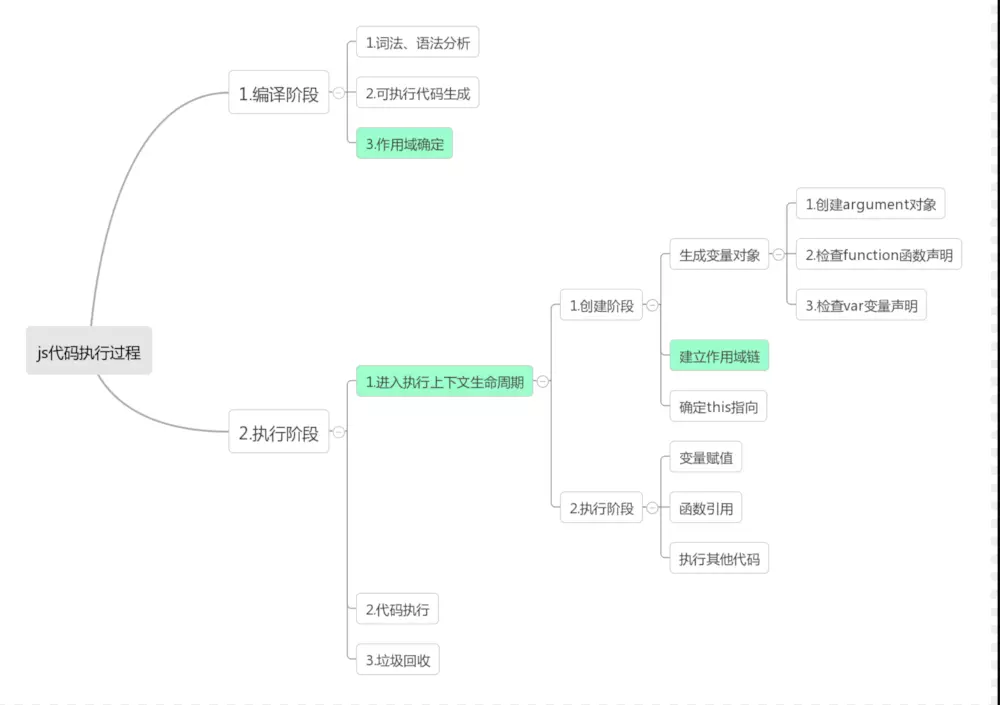
### 几种不销毁常用到的形式:

（1）函数执行,返回一个引用数据类型的值,并且在函数的外面被别人接收了,那么当前函数形成的私有作用域就不在销毁了-->例如上面的案例 var f=fn();

（2）在函数执行的时候,里面的一个小函数的地址赋值给了外面元素的点击事件,那么当前小函数也相当于被外面占用了,大函数执行形成的私有的作用域也不能销毁了

（3）在使用setTimeout实现轮询动画的时候,我们如果move需要传递参数值,那么像下面这样的写法会行成很多的不销毁的作用域,非常的耗性能

## JavaScript 执行过程



JavaScript整个执行过程，分为两个阶段，**代码编译阶段**和**代码执行阶段**

### 一、编译阶段

编译阶段由编译器执行，将代码翻译成可执行代码，这个阶段确定作用域规则。

变量的预编译只作声明，不作初始化，初始化在执行时

function语句定义的函数，不仅声明了函数名，而且函数体也进行了处理

### 二、执行阶段

执行阶段由引擎完成，主要任务是执行可执行代码，执行上下文在这个阶段创建。执行上下文也分为创建阶段和执行阶段。

1、执行上下文

执行上下文可以理解为当前代码的执行环境，它会形成一个作用域。

2、三种类型

a、全局执行上下文：浏览器的全局对象是window对象，this指向这个全局对象

b、函数执行上下文：存在无数个，只有在函数被调用的时候才会被创建，每次调用函数都会创建一个新的执行上下文，函数上下文的变量对象初始化只包括 Arguments 对象。

c、Eval函数执行上下文：指运行在eval函数的代码，很少用而且不建议使用。。

创建阶段：

确定this的值，也被成为 This Binding；

a、全局执行上下文中，this 指向全局对象，在浏览器中this 指向window对象，在nodejs中指向这个文件的module 对象。

b、函数执行上下文中，this的值取决于函数的调用方式。具体有：默认绑定、隐式绑定、显式绑定、new绑定、箭头函数。

需要注意的是同一作用域下存在多个同名函数声明，后面的会替换前面的函数声明。

### 三、执行上下文栈

因为JS引擎创建了很多的执行上下文，所以JS引擎创建了执行上下文栈（Execution context stack，ECS）来管理执行上下文。

当 JavaScript 初始化的时候会向执行上下文栈压入一个全局执行上下文，我们用 globalContext 表示它，并且只有当整个应用程序结束的时候，执行栈才会被清空，所以程序结束之前， 执行栈最底部永远有个 globalContext。

当执行一个函数的时候，就会创建一个执行上下文，并且压入执行上下文栈，当函数执行完毕的时候，就会将函数的执行上下文从栈中弹出。

## 匿名函数,箭头函数

### 匿名函数

#### 1，基本概念

匿名函数顾名思义指的是没有名字的函数，在实际开发中使用的频率非常高！（在[ES5](https://so.csdn.net/so/search?q=ES5&spm=1001.2101.3001.7020" \t "/Users/gdj/Documents\\x/_blank)中就已经有了匿名函数）

#### 2，匿名函数的应用场景

1自调用

匿名函数想要自调用需要放在（）里面并后面使用小括号调用，里面可以传参；

2事件的绑定函数

3在对象中使用

4函数表达式  
JS中定义函数的方式有两种：一种是函数声明，另一种就是函数表达式。

注意：函数声明，它的一个重要特征就是函数会进行声明提升，意思是在执行代码之前会先读取函数声明。这就意味着可以把函数声明放在调用它的语句后面；

函数表达式的写法则不会进行js预编译的函数提升；

5、回调函数

概念：回调函数就是一个被作为参数传递的函数；

6、作为返回值返回

### 箭头函数

#### 1，基本概念

箭头函数是ES6中新增一种语法，属于匿名函数； 将函数更加简单化的写法；也是非常的常用。

#### 2，箭头函数写法的特殊性

他有一些特殊写法 在编辑函数的时候可以适当的做出一些省略：

1，当只有一个参数的时候 可以省略小括号

2，当没有参数或者多个参数的时候 ，不能省略小括号；

3，当我们只有一句函数体的时候, 并且这句话是返回值的时候 ，可以同时省略return和大括号（函数体） ；

5，如果返回值是对象 ，在省略大括号和return的情况下，需要在对象外面加上一层小括号（因为有歧义无法确定大括号是箭头函数的还是对象的 ）

#### 3，箭头函数的特点

只能定义匿名函数

箭头函数没有原型对象

箭头函数不能定义构造函数

箭头函数不能使用arguments new super

箭头函数没有自己的this （调用的是他的父级的this）

#### 箭头函数this总是指向父级的this

## 普通函数和箭头函数

### 1.普通函数到箭头函数

function sum(a, b) {

return a + b;

}

const sum = (a, b) => {

return a + b;

}

const sum = (a, b) => a + b;

注意：若函数没有参数，或者多个参数的情况下，必须要使用括号。

### 2.箭头函数与普通函数的区别

1. 箭头函数的 this 是父级作用域的 this

const a = {

a: 1,

obj: () => {

console.log(' a this => ', this);

}

}

const b = {

b: 1,

obj: function () {

console.log(' b this => ', this);

}

}

a.obj() // window

b.obj() // { b: 1, obj: [Function: obj] }

2. call、apply、bind无法改变箭头函数的 this

const obj = { name: '111' }

const getName1 = () => {

console.log(this.name);

}

function getName2() {

console.log(this.name);

}

getName1.call(obj) // undefined

getName2.call(obj) // 111

1. 不可以作为构造函数

const Person = () => {

this.name = 'ggg'

}

new Person() // TypeError: Person is not a constructor

1. 不可以使用 arguments

const func1 = () => {

console.log('func1 arguments', arguments);

}

function func2() {

console.log('func2 arguments', arguments);

}

func1(1, 2, 3) // undefined

func2(1, 2, 3) // [Arguments] { '0': 1, '1': 2, '2': 3 }

1. 箭头函数不支持 new.target

const Cat1 = () => {

console.log('cat1 new.target', new.target);

}

function Cat2() {

console.log('cat2 new.target', new.target);

}

// new Cat1() // 浏览器环境 Uncaught SyntaxError: new.target expression is not allowed here

new Cat2() // new.target [Function: Cat2]

1. 箭头函数没有原型对象

const Dog1 = () => { }

function Dog2() { }

const dog2 = new Dog2();

console.log('--- Dog2 --->', {

// 'Dog1.prototype.constructor == Dog1': Dog1.prototype.constructor == Dog1,

// Cannot read property 'constructor' of undefined

'Dog2.prototype.constructor == Dog2': Dog2.prototype.constructor == Dog2, // true

'dog2.\_\_proto\_\_ == Dog2.prototype': dog2.\_\_proto\_\_ == Dog2.prototype, // true

});

## var、let、const

**var:**

1.声明变量的作用域限制于函数作用域

2.变量的声明会被提升，总是被提升到函数作用域顶部

**let：**

1.声明变量的作用域限制于块级

2.变量的声明不会被提升，处于“暂时性死区”

**const:**

1.声明变量的同时必须初始化，修改值时指针就会发生改变，要报错

2.如果是复合类型，只修改某个value值是不会报错的。

在开发过程中，程序员优先使用const，let次之，一般不使用var

var只有全局作用域和函数作用域的的概念，如果用var在函数里面声明变量，则这个变量属于当前函数的作用域，而如果是在函数外的任何地方声明，则属于全局作用域。而let是块级作用域。

var存在变量提升，无论var出现在作用域的哪个位置，这个声明都属于当前的整个作用域，在任何地方都能访问到，而let没有。

var声明的变量可以重复声明，而let不可以。

const一般用来声明常量，且其值不可变。

但是在实际使用的时候会发现，const的值其实是可以"变"的：

const arr = ['a']

arr.push('b')

arr[0] = 'c'

const obj = {

name:'jjy'

}

obj.name = 'wjh'

console.log(arr)

console.log(obj)

可以发现数组和对象内部的值被改变了，实际上这是因为const**保证的不是变量的值不动，而是变量指向的那个内存地址所保存的数据不能改变**。对于**基本数据类型，值就保存在变量指向的那个内存地址**，所以用const定义的基本数据类型的值是不能改变了，而对于数组和对象，它们属于**引用数据类型，变量保存的只是一个指向实际数据的指针**，而这个指针指向的内存地址才是真正保存数据的值的地方，所以const只能保证这个指针不能变，但不能保证他所指向的这个内存地址里面的值不能变。

## 深拷贝&浅拷贝&赋值

### 1.深拷贝

深拷贝是将一个对象从内存中完整的拷贝一份出来，从堆内存中开辟一个新的区域存放新对象（新旧对象不共享同一块内存），且修改新对象不会影响原对象（`深拷贝采用了在堆内存中申请新的空间来存储数据，这样每个可以避免指针悬挂`）

### 2.浅拷贝

如果属性是基本类型，拷贝的就是基本类型的值。如果属性是引用类型，那么拷贝的就是内存地址（新旧对象共享同一块内存），所以如果其中一个对象改变了这个地址，就会影响到另一个对象（`只是拷贝了指针，使得两个指针指向同一个地址`）

### 3.赋值与浅拷贝的区别

1. 把一个对象赋值给一个新的变量的时候，赋的其实是该对象在栈中的地址，而不是栈中的数据。也就是这两个对象指的是同一个储存空间，不论哪个对象发生改变，其实都是改变储存空间里的内容。因此，两个对象是联动的。

2. 浅拷贝是按位拷贝对象的，他会创建一个新的对象，这个对象有着原始对象属性值的一份精确拷贝。

* 如果属性是基本类型，拷贝的就是基本类型的值。
* 如果属性是内存地址（引用类型），拷贝的就是内存地址，因此如果一个对象改变了这个地址，就会影响到另一个对象。
* 即默认拷贝构造函数只是对对象进行浅拷贝复制（逐个成员依次拷贝），即只复制对象空间而不复制资源。

### 4.浅拷贝的实现【当拷贝对象只有一层的时候，是深拷贝】

* 展开运算符...
* Object.assign()
* Array.prototype.concat()
* Array.prototype.slice()

### 5.深拷贝的实现

* JSON.parse(JSON.stringify())
* 手写递归方法：(递归方法实现深度克隆原理：遍历对象、数组直到里边都是基本数据类型，然后再去复制，就是深度拷贝)
* jQuery的extend方法实现深拷贝
* 函数库lodash的\_.cloneDeep方法