# JS基础

## 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。

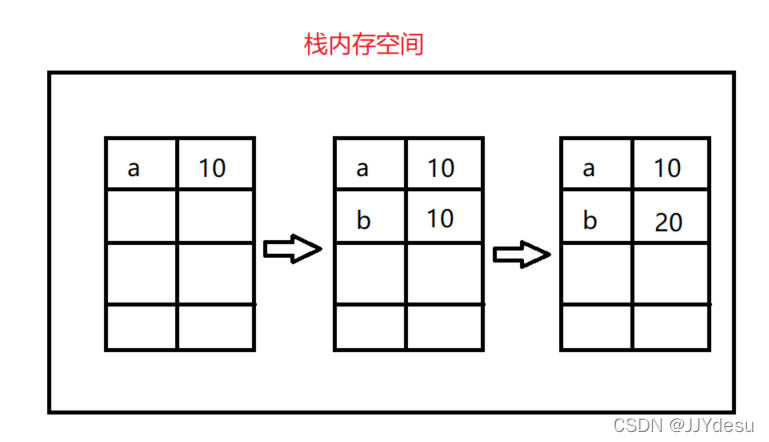
引用数据类型是存储在堆内存中，占据空间大。引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址，当解释器寻找引用值时，会检索其在栈中的地址，取得地址后从堆中获得实体。

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。

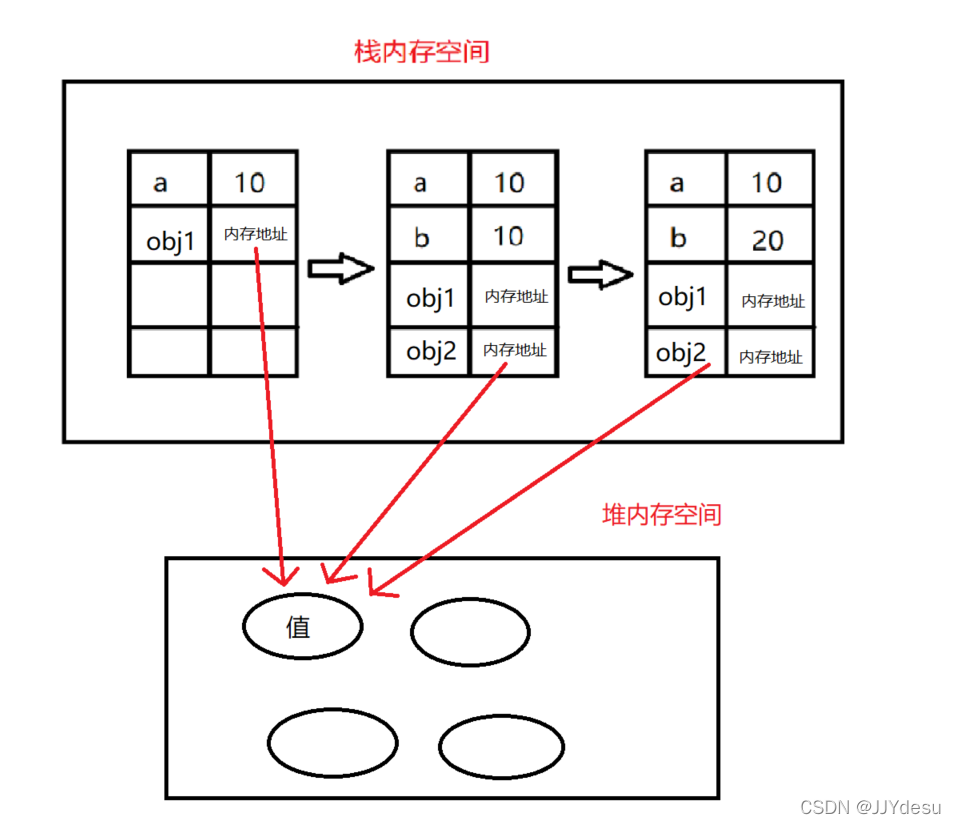
BigInt是ES10新出的一种数据类型，这种数据类型的特点就是数据涵盖的范围大，能够解决超出普通数据类型范围报错的问题。

### 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。

## js实现常见的数据结构---链表，队列，栈，树

### 链表

链表是由多个节点（node）连接起来的，每个节点包含了一些存储的数据和指向下一个节点的指针，

* 链表：多个连续的节点构成，
* 节点：包含一串数据，以及下一个节点的数据

注意：链表是由顺序的数据集结构，访问链表只能从头到尾的访问

### 队列

先进先出的数据结构：每次只能从队列的尾部添加数据，从队列的头部访问数据，

* 一头放数据，一头拿数据，不可直接访问中间的数据

### 栈

后进先出的数据结构：只有一个出入口，数据的增减都在尾部

* 只能在链式数据的一端进行访问数据和修改数据

### 树

树是一个二维结构，一颗树从根节点出发，连接着多个子节点，每个子节点又连接着子孙节点，循环往复就组成了一棵树；

注意：在树中子节点只能指向子孙节点不能指向父节点，根节点只有一个，他是所有节点的父节点，不能被任何节点指向

* 根节点：每个数的根节点只能有一个（最顶层的节点）
* 叶节点：没有子节点的节点（最底层的节点）

### 总结

通过js的class的功能，可以很清晰的实现这些数据结构，本质上看数据结构表示的是按照一定的关系顺序存放数据，每个数据的单位都是节点，节点包含了一个单位数据（任意大小）和下一个节点的地址（引用），这里是使用js对象属性的方式来实现的；

一维的数据结构中，链表是最基本的数据结构，它按顺序连接了多个节点，通过头节点就可以获取到整个链表数据，其他的数据结构（队列，栈）都类似链表结构的变式

树是二维的数据结构，它的关系是一对多，一个节点指向了多个节点，同时树是开放的，一颗树是没有环状结构的（子节点不会引用父节点），一旦出现环状结构，就变成了图（封闭的二维数据结构）

这些基本结构的映射关系： 点（节点）--- 线（一维结构） --- 面 （二维结构）

补充：

和数组的关系：数组也是一种数据结构，但它是连续的空间（连续的变量集合），而链表这样的结构它的数据是分散的（各处的变量集合）

## 谈谈this对象的理解

this的值是在函数调用时确定的，它的值取决于函数的调用方式。

在JavaScript中，this的指向可以通过四种调用模式来判断：

**1. 函数调用模式：**

当一个函数不是一个对象的属性时，直接作为函数来调用时，this指向全局对象（在浏览器中是window对象，在Node.js中是global对象）。

function test() {

console.log(this)

}

test() // 输出全局对象，一般是window对象

**2. 方法调用模式：**

如果一个函数作为一个对象的方法来调用时，this指向这个对象。

var obj = {

name: 'ggg',

sayName: function () {

console.log(this.name)

}

}

obj.sayName() // 输出ggg

**3. 构造器调用模式：**

如果一个函数用new调用时，函数执行前会新创建一个对象，this指向这个新创建的对象。

function Person(name) {

this.name = name

}

var person = new Person('ggg')

console.log(person.name) // 输出ggg

**4. apply、call和bind调用模式：**

这三个方法都可以显示的指定调用函数的this指向。其中apply方法接收两个参数：一个是this绑定的对象，一个是参数数组。call方法接收的参数，第一个是this绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用call()方法时，传递给函数的参数必须逐个列举出来。bind方法通过传入一个对象，返回一个this绑定了传入对象的新函数。这个函数的this指向除了使用new时会被改变，其他情况下都不会改变。

function sayName() {

console.log(this.name)

}

var obj1 = {

name: 'John'

}

var obj2 = {

name: 'Mike'

}

sayName.call(obj1) // 输出John

sayName.call(obj2) // 输出Mike

var boundSayName = sayName.bind(obj1)

boundSayName() // 输出John

## 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

### 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

### 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。

### 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。

## 说说new操作符具体干了什么?

在前开发中，new操作符是用来创建一个新的对象例的。当使用new操作符时，它会执行以下几个步骤：

* 创建一个空的简单JavaScript对象（即{}）。
* 将这个空对象的\_\_proto\_\_属性指向构造函数的原型对象。
* 将构造函数内部的this关键字指向这个新创建的空对象。
* 执行构造函数内部的代码，为这个新对象添加属性和方法。
* 如果构造函数返回了一个对象，则返回这个对象；否则返回这个新创建的对象。

详细分析一下这个过程：

创建一个空的简单JavaScript对象：这个空对象将会成为将要创建的实例对象。这个对象会继承构造函数的原型对象上的属性和方法。

将这个空对象的\_\_proto\_\_属性指向构造函数的原型对象：原型对象是一个包含可继承属性和方法的对象。在JavaScript中，每个函数都有一个prototype属性，指向它的原型对象。当我们使用new操作符创建一个对象时，这个新对象的\_\_proto\_\_属性就会指向构造函数的原型对象。

将构造函数内部的this关键字指向这个新创建的空对象：在构造函数内部，通过使用this关键字，我们可以向这个新对象添加属性和方法。this关键字在这里指代了当前正在创建的实例对象。

执行构造函数内部的代码，为这个新对象添加属性和方法：在构造函数内部，我们可以向这个新对象添加属性和方法，以便将其初始化为我们想要的状态。

如果构造函数返回了一个对象，则返回这个对象；否则返回这个新创建的对象：如果构造函数内部有返回语句，并且返回的是一个对象，则返回这个对象；否则返回刚刚创建的新对象。这意味着我们可以在构造函数内部返回一个自定义的对象，而不一定是this所指向的对象。

使用new操作符创建一个对象实例时，我们会创建一个空对象，并将它的\_\_proto\_\_属性指向构造函数的原型对象。然后，构造函数内部的代码会被执行，以便向这个新对象添加属性和方法。最后，如果构造函数内部有返回语句，则返回指定的对象；否则返回这个新创建的对象。这样，我们就可以使用构造函数来创建多个实例对象，它们都可以共享构造函数的原型对象上的属性和方法。

## JavaScript中执行上下文和执行栈是什么?

### 一、执行上下文（Execution Context）

简而言之，执行上下文就是当前 JavaScript 代码被解析和执行时所在环境的抽象概念， JavaScript 中运行任何的代码都是在执行上下文中运行

2.执行上下文的类型

执行上下文总共有三种类型：

全局执行上下文： 这是默认的、最基础的执行上下文。不在任何函数中的代码都位于全局执行上下文中。它做了两件事：1. 创建一个全局对象，在浏览器中这个全局对象就是 window 对象。2. 将 this 指针指向这个全局对象。一个程序中只能存在一个全局执行上下文。

函数执行上下文： 每次调用函数时，都会为该函数创建一个新的执行上下文。每个函数都拥有自己的执行上下文，但是只有在函数被调用的时候才会被创建。一个程序中可以存在任意数量的函数执行上下文。每当一个新的执行上下文被创建，它都会按照特定的顺序执行一系列步骤，具体过程将在本文后面讨论。

Eval 函数执行上下文： 运行在 eval 函数中的代码也获得了自己的执行上下文，但由于 Javascript 开发人员不常用 eval 函数，所以在这里不再讨论。

### 二、执行上下文的生命周期

执行上下文的生命周期包括三个阶段：创建阶段→执行阶段→回收阶段，本文重点介绍创建阶段。

1.创建阶段

当函数被调用，但未执行任何其内部代码之前，会做以下三件事：

创建变量对象：首先初始化函数的参数 arguments，提升函数声明和变量声明。下文会详细说明。

创建作用域链（Scope Chain）：在执行期上下文的创建阶段，作用域链是在变量对象之后创建的。作用域链本身包含变量对象。作用域链用于解析变量。当被要求解析变量时，JavaScript 始终从代码嵌套的最内层开始，如果最内层没有找到变量，就会跳转到上一层父作用域中查找，直到找到该变量。

确定 this 指向：包括多种情况，下文会详细说明

在一段 JS 脚本执行之前，要先解析代码（所以说 JS 是解释执行的脚本语言），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。变量先暂时赋值为 undefined，函数则先声明好可使用。这一步做完了，然后再开始正式执行程序。

另外，一个函数在执行之前，也会创建一个函数执行上下文环境，跟全局上下文差不多，不过 函数执行上下文中会多出 this arguments 和函数的参数。

2.执行阶段

执行变量赋值、代码执行

3.回收阶段

执行上下文出栈等待虚拟机回收执行上下文

### 三、变量提升和 this 指向的细节

1.变量声明提升

大部分编程语言都是先声明变量再使用，但在 JS 中，事情有些不一样：

console.log(a)// undefinedvar a = 10

上述代码正常输出 undefined 而不是报错 Uncaught ReferenceError: a is not defined ,这是因为声明提升（hoisting），相当于如下代码：

var a; //声明 默认值是 undefined “准备工作”console.log(a);a=10; //赋值

2.函数声明提升

我们都知道，创建一个函数的方法有两种，一种是通过函数声明 function foo(){}

另一种是通过函数表达式 var foo = function(){} ,那这两种在函数提升有什么区别呢？

console.log(f1) // function f1(){}function f1() {} // 函数声明console.log(f2) // undefinedvar f2 = function() {} // 函数表达式

接下来我们通过一个例子来说明这个问题：

function test() { foo(); // Uncaught TypeError "foo is not a function" bar(); // "this will run!" var foo = function () { // function expression assigned to local variable 'foo' alert("this won't run!"); } function bar() { // function declaration, given the name 'bar' alert("this will run!"); }}test();

在上面的例子中，foo()调用的时候报错了，而 bar 能够正常调用。

我们前面说过变量和函数都会上升，遇到函数表达式 var foo = function(){} 时，首先会将 var foo 上升到函数体顶部，然而此时的 foo 的值为 undefined,所以执行 foo() 报错。

而对于函数 bar() , 则是提升了整个函数，所以 bar() 才能够顺利执行。

有个细节必须注意：当遇到函数和变量同名且都会被提升的情况，函数声明优先级比较高，因此变量声明会被函数声明所覆盖，但是可以重新赋值。

alert(a);//输出：function a(){ alert('我是函数') }function a(){ alert('我是函数') }//var a = '我是变量';alert(a); //输出：'我是变量'

function 声明的优先级比 var 声明高，也就意味着当两个同名变量同时被 function 和 var 声明时，function 声明会覆盖 var 声明

这代码等效于：

function a(){alert('我是函数')} var a; //hoistingalert(a); //输出：function a(){ alert('我是函数') }a = '我是变量';//赋值alert(a); //输出：'我是变量'

最后我们看个复杂点的例子：

function test(arg){ // 1. 形参 arg 是 "hi" // 2. 因为函数声明比变量声明优先级高，所以此时 arg 是 function console.log(arg); var arg = 'hello'; // 3.var arg 变量声明被忽略， arg = 'hello'被执行 function arg(){ console.log('hello world') } console.log(arg); }test('hi');/\* 输出：function arg(){ console.log('hello world') }hello \*/

这是因为当函数执行的时候,首先会形成一个新的私有的作用域，然后依次按照如下的步骤执行：

如果有形参，先给形参赋值

进行私有作用域中的预解释，函数声明优先级比变量声明高，最后后者会被前者所覆盖，但是可以重新赋值

私有作用域中的代码从上到下执行

3.确定 this 的指向

先搞明白一个很重要的概念 —— this 的值是在执行的时候才能确认，定义的时候不能确认！ 为什么呢 —— 因为 this 是执行上下文环境的一部分，而执行上下文需要在代码执行之前确定，而不是定义的时候。看如下例子：

// 情况 1function foo() { console.log(this.a) //1}var a = 1foo()// 情况 2function fn(){ console.log(this);}var obj={fn:fn};obj.fn(); //this->obj// 情况 3function CreateJsPerson(name,age){//this 是当前类的一个实例 p1this.name=name; //=>p1.name=namethis.age=age; //=>p1.age=age}var p1=new CreateJsPerson("尹华芝",48);// 情况 4function add(c, d){ return this.a + this.b + c + d;}var o = {a:1, b:3};add.call(o, 5, 7); // 1 + 3 + 5 + 7 = 16add.apply(o, [10, 20]); // 1 + 3 + 10 + 20 = 34// 情况 5<button>箭头函数 this</button><script type="text/javascript"> let btn1 = document.getElementById('btn1'); let obj = { name: 'kobe', age: 39, getName: function () { btn1.onclick = () => { console.log(this);//obj }; } }; obj.getName();</script>

接下来我们逐一解释上面几种情况

对于直接调用 foo 来说，不管 foo 函数被放在了什么地方，this 一定是 window

对于 obj.foo() 来说，我们只需要记住，谁调用了函数，谁就是 this，所以在这个场景下 foo 函数中的 this 就是 obj 对象

在构造函数模式中，类中（函数体中）出现的 this.xxx=xxx 中的 this 是当前类的一个实例

call、apply 和 bind：this 是第一个参数

箭头函数 this 指向:箭头函数没有自己的 this，看其外层的是否有函数，如果有，外层函数的 this 就是内部箭头函数的 this，如果没有，则 this 是 window。

### 四、执行上下文栈（Execution Context Stack）

函数多了，就有多个函数执行上下文，每次调用函数创建一个新的执行上下文，那如何管理创建的那么多执行上下文呢？

JavaScript 引擎创建了执行上下文栈来管理执行上下文。可以把执行上下文栈认为是一个存储函数调用的栈结构，遵循先进后出的原则。

从上面的流程图，我们需要记住几个关键点：

JavaScript 执行在单线程上，所有的代码都是排队执行。

一开始浏览器执行全局的代码时，首先创建全局的执行上下文，压入执行栈的顶部。

每当进入一个函数的执行就会创建函数的执行上下文，并且把它压入执行栈的顶部。当前函数执行完成后，当前函数的执行上下文出栈，并等待垃圾回收。

浏览器的 JS 执行引擎总是访问栈顶的执行上下文。

全局上下文只有唯一的一个，它在浏览器关闭时出栈。

我们再来看个例子：

var color = 'blue';

function changeColor() {

var anotherColor = 'red';

function swapColors() {

var tempColor = anotherColor;

anotherColor = color;

color = tempColor;

}

swapColors();

}

changeColor();

上述代码运行按照如下步骤：

当上述代码在浏览器中加载时，JavaScript 引擎会创建一个全局执行上下文并且将它推入当前的执行栈

调用 changeColor 函数时，此时 changeColor 函数内部代码还未执行，js 执行引擎立即创建一个 changeColor 的执行上下文（简称 EC），然后把这执行上下文压入到执行栈（简称 ECStack）中。

执行 changeColor 函数过程中，调用 swapColors 函数，同样地，swapColors 函数执行之前也创建了一个 swapColors 的执行上下文，并压入到执行栈中。

swapColors 函数执行完成，swapColors 函数的执行上下文出栈，并且被销毁。

changeColor 函数执行完成，changeColor 函数的执行上下文出栈，并且被销毁。

## JavaScript有几种方法判断变量的类型

* typeof 运算符
* instanceof 运算符
* Object.prototype.toString 方法(对象原型链判断方法)
* constructor (用于引用数据类型)

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。

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

## 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

## 字面量

字面量（Literal），又名直接量，即程序中直接使用的数据值。

## 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.)

闭包内部又创建了一个函数

## js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

### 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.

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

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.

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。

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

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

## map和forEach的异同

### 相同点

* 都只能遍历数组
* 都是循环遍历数组中的每一项。
* 每次执行匿名函数都支持三个参数，参数分别为item(当前每一项)，index(索引值)，arr(原数组)。
* 匿名函数中的this都是指向window。

不同点

* forEach()方法没有返回值，会更改原数组。
* map() 有返回值，返回一个新数组，不会改变原数组，map() 不会对空数组进行检测。
* 总的来说 map 的速度大于 forEach()
* 性能上来说for > forEach > map
* 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。

## 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

## 对象常用的函数

Object.assign() 方法用于将所有可枚举自有 Property 的值从一个或多个源对象拷贝到目标对象。

Object.create() 方法用于创建指定对象为原型对象的新对象。

Object.defineProperties() 方法用于为一个对象定义 Properties 和/或修改已有的 Properties 的 Attributes。

Object.defineProperty() 方法用于为一个对象定义一个自有 Property 和/或修改已有 Property 的 Attributes。

Object.entries() 方法用于枚举指定对象并返回以键值对组成的数组为元素的二维数组。

Object.freeze() 方法用于冻结一个对象,冻结一个对象是 JavaScript 提供的最高完整性级别保护措施。

Object.getOwnPropertyDescriptor() 方法可以获取对象自有 Property 的某个 Attributes。

Object.getOwnPropertyDescriptors() 方法用于获取一个对象的所有自身 Property 的 Attributes。

Object.getOwnPropertyNames() 方法用于获取指定对象的所有自身 Property 的键名（包括不可枚举属性但不包括 Symbol 值作为名称的属性）组成的数组。

Object.getOwnPropertySymbols() 方法用于获取一个给定对象自身的所有 Symbol Property 的数组。

Object.getPrototypeOf() 方法用于获取指定对象的原型（内部 [[Prototype]] 属性的值）。

Object.is() 方法用于判断两个值是否是相同的值。

Object.isExtensible() 方法用于检测指定对象是否可扩展。

Object.isFrozen() 方法用来检测指定对象是否已被冻结。

Object.isSealed() 方法用于检测指定对象是否已被密封。

Object.keys() 方法用于获取指定对象自身可枚举 Property 组成的键名数组。

Object.preventExtensions() 方法用于令指定对象无法再添加新的属性。

Object.seal 方法用于标识指定对象为不可扩展，且所有现有 Property 均不可配置。

Object.setPrototypeOf() 方法用于设置一个指定的对象的原型 ( 即，内部 [[Prototype]] 属性）到另一个对象或 null。

Object.values() 方法用于指定对象自身的所有可枚举 Property 值的数组。

Object.prototype.hasOwnProperty 方法用于检测指定对象自有 Properties 中是否具有指定的 Property。

Object.prototype.isPrototypeOf() 方法用于测试指定对象是否存在于目标对象的原型链上。

Object.prototype.prototypeIsEnumerable() 方法用于检测指定 Property 是否可枚举。

Object.prototype.toString() 方法用于表示指定对象的字符串。

Object.fromEntries() 方法把键值对列表转换为一个对象。

## Object,Map,Set的异同

### 1.定义

#### 1.1 Object

* 本质上是键值对的集合（Hash 结构）
* 只能用字符串当作键

#### 1.2 Map

* 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。
* 本身是一个构造函数(要 new)，接受数组作为参数（原理上Array.forEach）
* 如果对同一个键多次赋值，后面的值将覆盖前面的值，如果读取一个未知的键，则返回undefined
* Map 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键
* 继承自Object

#### 1.3 Set

* 类似于数组，但成员值是唯一的，没有重复的值（可以接受一个数组作为参数，进行初始化）
* 本身是一个构造函数(要 new)，用来生成Set数据结构
* Set 对象允许你储存任何类型的唯一值，无论是原始值或者是对象引用。
* 向 Set 加入值的时候，不会发生类型转换，所以5和"5"是两个不同的值

### 结构区别

// Object

const obj = { a: 1, b: 2 }

// Map

const map = new Map()

map.set(1, 2);

map.set({ a: 1 }, { b: 2 })

map.has(1)

map.get(1)

// map.delete(1)

// map.clear()

// Set

const set = new Set([1, 2, 3, 4, 4]);

[...set]

// [1, 2, 3, 4]

set.has(5)

set.add(5)

set.delete(1)

set.clear()

### 3.相互转换

#### 3.1 Set > 数组

const set1 = new Set([1, 2, 3, 4, 5]);

const arr1 = Array.from(set1);

// [ 1, 2, 3, 4, 5 ]

#### 3.2 数组 > Set

const arr2 = [1, 2, 3, 4, 5];

const set2 = new Set(arr2);

// Set { 1, 2, 3, 4, 5 }

#### 3.3 Map > 数组

const keys = [...map.keys()];

const values = [...map.values()];

const entries = [...map.entries()];

const map2 = [...map];

// keys: [ 1, '1', { a: 1 } ],

// values: [ 2, 222, { b: 2 } ],

// entries: [ [ 1, 2 ], [ '1', 222 ], [ [Object], [Object] ] ],

// map2: [ [ 1, 2 ], [ '1', 222 ], [ [Object], [Object] ] ]

#### 3.4 数组 > Map

const arr1 = [ [1, 2],[2, 3],[3, 4] ];

const arr2map = new Map(arr1);

// arr2map: Map { 1 => 2, 2 => 3, 3 => 4 }

#### 3.5 Map > 对象

// 自定义

function map2Obj(m) {

const obj5 = {};

for (const [k, v] of m) {

obj5[k] = v;

}

return obj5;

}

const obj5\_1 = map2Obj(arr2map);

// 键值对转为对象

const map5\_1 = Object.fromEntries([

['foo', 'bar'],

['baz', 42]

]);

// map转为对象

const map5\_2 = new Map([

['foo', 'bar'],

['baz', 42]

]);

const obj5\_2 = Object.fromEntries(map5\_2);

// obj5\_1: { '1': 2, '2': 3, '3': 4 },

// map5\_1: { foo: 'bar', baz: 42 },

// obj5\_2: { foo: 'bar', baz: 42 }

#### 3.6 对象 > 数组

const obj6 = { a: 1, b: 2 };

const map6 = new Map(Object.entries(obj6));

// Map { 'a' => 1, 'b' => 2 }

#### 3.7 将查询字符串转为对象

Object.fromEntries(new URLSearchParams('foo=1&baz=2))

// { foo: 1, baz: 2}

### 4.遍历操作

set遍历操作

* Set.prototype.keys()：返回键名的遍历器
* Set.prototype.values()：返回键值的遍历器
* Set.prototype.entries()：返回键值对的遍历器
* Set.prototype.forEach()：使用回调函数遍历每个成员

map遍历操作

Map 的遍历顺序就是插入顺序

* Map.prototype.keys()：返回键名的遍历器。
* Map.prototype.values()：返回键值的遍历器。
* Map.prototype.entries()：返回所有成员的遍历器。
* Map.prototype.forEach()：遍历 Map 的所有成员。forEach方法还可以接受第二个参数，用来绑定this。

### 5.判断长度

* Object：Object.key ().length
* Set：set.size
* map:map.size

### 6.使用场景

Set

* 数组去重（使用Set对象的唯一性值特性方便的给我们数组去重）
* 去除字符串里面的重复字符
* 去除数组中的重复
* 实现并集（Union）、交集（Intersect）和差集（Difference）
* 遍历操作中，同步改变原来的 Set 结构目前没有直接的方法，但有两种变通方法。一种是利用原 Set 结构映射出一个新的结构，然后赋值给原来的 Set 结构；另一种是利用Array.from方法。

Map

* 使用Map对象建立一个请求状态码对象字典、保证对象的顺序
* 结合数组的map方法、filter方法，可以实现 Map 的遍历和过滤（Map 本身没有map和filter方法）。

### 7.Map和Object的区别

* Object的key 必须是简单数据类型（整数、字符串、symbol），map的key可以是任何类型
* Map元素插入顺序是FIFO，object没有
* Map继承Object
* Map在存储大量元素的时候性能表现更好
* 写入删除密集的情况应该使用 Map
* Map删除: map.delete(key), Object删除: delete key
* Map有内置的迭代器，Object没有内置的迭代器
* 循环遍历: Map使用for of或 forEach, Object使用for in或 Object.keys(obj)

#### 如何判断某种类型是否可迭代，可以通过以下方式实现

//typeof <obj>[Symbol.iterator] === “function”

console.log(typeof obj[Symbol.iterator]); //undefined

console.log(typeof map[Symbol.iterator]); //function

### 8.Map和Set的区别

* Set以[value, value]的形式储存元素,Map以[key,value]的形式储存元素
* Map的值不作为键，键和值是分开的

## 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。

## Async和Await原理

async、await从字面上理解，async就是异步的意思，await就是等待的意思，而两者的用法上也是这样的，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

过程就是：

1，async就是一个async函数，而await只能在这个函数中使用

2，await表示这里等待await后面的操作执行完毕，再执行下一句代码

3，await后面紧跟的最好是一个定时操作或者一个异步操作

async和await的优点

1，解决了回调地狱的问题

2，支持并发执行

3，可以添加返回值 return xxx

4，可以在代码中添加try/catch捕获错误

## 说一说promise是什么与使用方法？

九大方法: resolve、reject、then、catch、finally、all、allSettled、race、any

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

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。

Promise.allSettled()：入参和Promise.all、Promise.race一样，接受一个promise 对象的数组作为参数,也是同时开始、并行执行的。但是Promise.allSettled的返回值需要注意以下几点：Promise.allSettled不会走进catch，当所有输入Promise都被履行或者拒绝时， statusesPromise 会解析一个具有具体完成状态的数组

{ status: 'fulfilled', value:value } ：如果相应的promise被履行

{ status: 'rejected', reason: reason }：如果相应的promise被拒绝

## 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。

## 类数组对象与 arguments

### 类数组对象

所谓类数组对象，就是指可以通过索引属性访问元素并且拥有 length 属性的对象。

var arrLike = {

0: 'name',

1: 'age',

2: 'job',

length: 3

}

它所对应的数组应该是这样子的

var arr = ['name', 'age', 'job'];

所以说类数组对象与数组的性质相似，是因为类数组对象在**访问、赋值、获取长度**上的操作与数组是一致的。

数组与类数组对象的访问

console.log(arr[0]); // name

console.log(arrLike[0]); // name

数组与类数组对象的赋值

arr[0] = 'new name';

arrLike[0] = 'new name';

获取数组与类数组对象的长度

console.log(arr.length); // 3

console.log(arrLike.length); // 3

arrLike.push('hobby'); // Uncaught TypeError: arrLike.push is not a function

一般是通过 Function.call 或者 Function.apply 方法来间接调用

// 使用 call

Array.prototype.push.call(arrLike, 'hobby');

console.log(arrLike); // { '0': 'name', '1': 'age', '2': 'job', '3': 'hobby', length: 4 }

var arrLikeStr = Array.prototype.join.call(arrLike, '&')

console.log(arrLikeStr); // name&age&job&hobby

// 使用 apply

Array.prototype.push.apply(arrLike, ['hobby']);

console.log(arrLike); // { '0': 'name', '1': 'age', '2': 'job', '3': 'hobby', length: 4 }

var arrLikeStr = Array.prototype.join.apply(arrLike, ['&'])

console.log(arrLikeStr); // name&age&job&hobby

在实际的开发过程中，为了方便对数据的操作，我们还经常会通过 Array.prototype.slice 或 Array.prototype.splice 等方法把类数组对象转换成真正的数组。

// 使用 call

console.log(Array.prototype.slice.call(arrLike,0));

console.log(Array.prototype.splice.call(arrLike,0)); // 会改变原先的类数组对象

// 使用 apply

console.log(Array.prototype.slice.apply(arrLike,[0]));

console.log(Array.prototype.splice.apply(arrLike,[0])); // 会改变原先的类数组对象

需要注意的是，Array.prototype.slice 方法不会改变原先类数组对象，而 Array.prototype.splice 会改变原先的类数组对象，这一点跟数组调用 slice 或 splice 方法时的行为是一致的。

### arguments

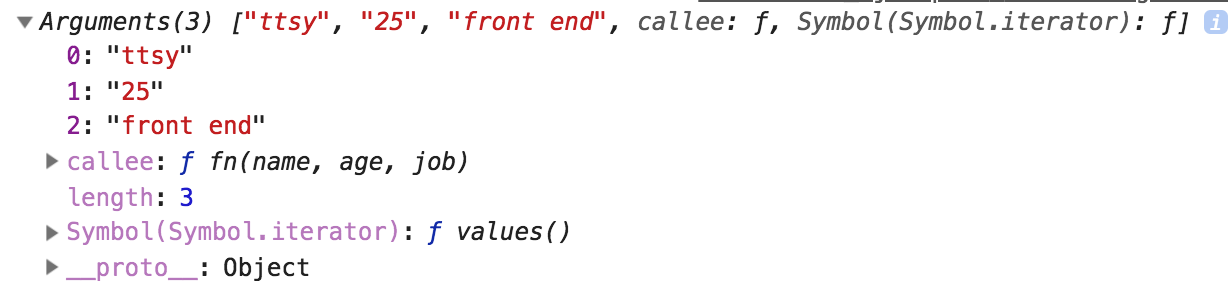
在函数体中定义了 Arguments 对象，其包含函数的参数和其它属性，以 arguments 变量来指代。

function fn(name, age, job) {

console.log(arguments);

}

fn('ttsy', '25', 'front end')



length 属性表示的是实参的长度，即调用函数的时候传入的参数个数。

function fn(name, age, job) {

console.log(arguments.length); // 2

}

fn('ttsy', '25')

arguments 是一个经典的类数组对象，我们可以通过上述的 Function.call 或者 Function.apply 方法来间接调用数组的方法，也可以直接通过 Array.prototype.slice 或 Array.prototype.splice 等方法把类数组对象转换成真正的数组。

## 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）

## 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。

## 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

## js阻塞怎么处理

Js 单线程肯定会出现阻塞

两个层面：

页面阻塞：要么js代码放在body最后，

<script defer src> <script async src>引入外部js代码的时候，可以延迟引入。异步引入

代码阻塞：避免使用alert comfirm 、异步解决方案

## 什么是事件委托

事件委托也称之为事件代理（Event Delegation）。是JavaScript中常用绑定事件的常用技巧。顾名思义，“事件代理”即是把原本需要绑定在子元素的响应事件委托给父元素，让父元素担当事件监听的职务。事件代理的原理是DOM元素的事件冒泡。

## 浏览器跨标签页通信的方式都有什么？

1使用localStorage或sessionStorage存储共享数据，并通过监听storage事件来实现数据的变化检测和同步更新。

2使用BroadcastChannel API，它提供了一种跨窗口通信的机制，可以在不同标签页之间发送消息。

3使用window.postMessage()方法，该方法允许在不同的窗口或标签页之间安全地传递消息。

4借助服务端的实时通信技术，如WebSocket，通过服务器作为中介来实现标签页之间的消息传递和数据同步。

## Object.keys 与 Object.getOwnPropertyNames() 有何区别【热度: 210】

Object.keys()和Object.getOwnPropertyNames()都是用于获取对象自身属性名的方法，但它们之间存在一些区别：

### 一、返回值类型

* Object.keys()：

返回一个由对象自身可枚举属性名组成的数组。

可枚举属性是指那些可以通过for...in循环遍历到的属性。

* Object.getOwnPropertyNames()：

返回一个由对象自身所有属性名组成的数组，无论属性是否可枚举。

### 二、可枚举性处理

* Object.keys()：

只返回可枚举属性的名称。如果一个属性被设置为不可枚举，它将不会出现在Object.keys()的返回结果中。

例如，使用Object.defineProperty()定义的不可枚举属性不会被包含在Object.keys()的结果中。

* Object.getOwnPropertyNames()：

返回所有属性的名称，包括可枚举和不可枚举的属性。

这使得它在需要获取对象的所有属性，无论其可枚举性如何时非常有用。