### get请求传参长度的误区

误区：我们经常说get请求参数的大小存在限制，而post请求的参数大小是无限制的。

实际上HTTP 协议从未规定 GET/POST 的请求长度限制是多少。对get请求参数的限制是来源与浏览器或web服务器，浏览器或web服务器限制了url的长度。为了明确这个概念，我们必须再次强调下面几点:

**HTTP 协议 未规定** GET 和POST的长度限制

GET的最大长度显示是因为 **浏览器和 web服务器限制了 URI的长度**

不同的浏览器和WEB服务器，限制的最大长度不一样

要支持IE，则最大长度为2083byte，若只支持Chrome，则最大长度 8182byte

### 补充get和post请求在缓存方面的区别

### **参考回答：**

post/get的请求区别，具体不再赘述。

补充补充一个get和post在缓存方面的区别：

get请求类似于查找的过程，用户获取数据，可以不用每次都与数据库连接，所以可以使用缓存。

post不同，post做的一般是修改和删除的工作，所以必须与数据库交互，所以不能使用缓存。因此get请求适合于请求缓存。

## **● 说一下闭包**

### **参考回答：**

一句话可以概括：闭包就是能够读取其他函数内部变量的函数，或者子函数在外调用，子函数所在的父函数的作用域不会被释放。

## **● 说一下类的创建和继承**

### **参考回答：**

（1）类的创建（es5）：new一个function，在这个function的prototype里面增加属性和方法。

下面来创建一个Animal类：

// 定义一个动物类

function Animal (name) {

// 属性

this.name = name || 'Animal';

// 实例方法

this.sleep = function(){

console.log(this.name + '正在睡觉！');

}

}

// 原型方法

Animal.prototype.eat = function(food) {

console.log(this.name + '正在吃：' + food);

};

这样就生成了一个Animal类，实力化生成对象后，有方法和属性。

（2）类的继承——原型链继承

--原型链继承

function Cat(){ }

Cat.prototype = new Animal();

Cat.prototype.name = 'cat';

//　Test Code

var cat = new Cat();

console.log(cat.name);

console.log(cat.eat('fish'));

console.log(cat.sleep());

console.log(cat instanceof Animal); //true

console.log(cat instanceof Cat); //true

介绍：在这里我们可以看到new了一个空对象,这个空对象指向Animal并且Cat.prototype指向了这个空对象，这种就是基于原型链的继承。

特点：基于原型链，既是父类的实例，也是子类的实例

缺点：无法实现多继承

（3）构造继承：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

// Test Code

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // false

console.log(cat instanceof Cat); // true

特点：可以实现多继承

缺点：只能继承父类实例的属性和方法，不能继承原型上的属性和方法。

（4）实例继承和拷贝继承

实例继承：为父类实例添加新特性，作为子类实例返回

拷贝继承：拷贝父类元素上的属性和方法

上述两个实用性不强，不一一举例。

（5）组合继承：相当于构造继承和原型链继承的组合体。通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

Cat.prototype = new Animal();

Cat.prototype.constructor = Cat;

// Test Code

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // true

console.log(cat instanceof Cat); // true

特点：可以继承实例属性/方法，也可以继承原型属性/方法

缺点：调用了两次父类构造函数，生成了两份实例

（6）寄生组合继承：通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

(function(){

// 创建一个没有实例方法的类

|  |  |
| --- | --- |
| 1  2 | var Super = function(){};  Super.prototype = Animal.prototype; |

//将实例作为子类的原型

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Cat.prototype = new Super();  })();  // Test Code  var cat = new Cat();  console.log(cat.name);  console.log(cat.sleep());  console.log(cat instanceof Animal); // true  console.log(cat instanceof Cat); //true |

较为推荐

## **● 如何解决异步回调地狱**

### **参考回答：**

|  |  |
| --- | --- |
| 1 | promise、generator、async/await |

## **● 说说前端中的事件流**

### **参考回答：**

HTML中与javascript交互是通过事件驱动来实现的，例如鼠标点击事件onclick、页面的滚动事件onscroll等等，可以向文档或者文档中的元素添加事件侦听器来预订事件。想要知道这些事件是在什么时候进行调用的，就需要了解一下“事件流”的概念。

什么是事件流：事件流描述的是从页面中接收事件的顺序,DOM2级事件流包括下面几个阶段。

事件捕获阶段

处于目标阶段

事件冒泡阶段

addEventListener：addEventListener 是DOM2 级事件新增的指定事件处理程序的操作，这个方法接收3个参数：要处理的事件名、作为事件处理程序的函数和一个布尔值。最后这个布尔值参数如果是true，表示在捕获阶段调用事件处理程序；如果是false，表示在冒泡阶段调用事件处理程序。

IE只支持事件冒泡。

## **● 如何让事件先冒泡后捕获**

### **参考回答：**

在DOM标准事件模型中，是先捕获后冒泡。但是如果要实现先冒泡后捕获的效果，对于同一个事件，监听捕获和冒泡，分别对应相应的处理函数，监听到捕获事件，先暂缓执行，直到冒泡事件被捕获后再执行捕获之间。

## **● 说一下事件委托**

### **参考回答：**

简介：事件委托指的是，不在事件的发生地（直接dom）上设置监听函数，而是在其父元素上设置监听函数，通过事件冒泡，父元素可以监听到子元素上事件的触发，通过判断事件发生元素DOM的类型，来做出不同的响应。

举例：最经典的就是ul和li标签的事件监听，比如我们在添加事件时候，采用事件委托机制，不会在li标签上直接添加，而是在ul父元素上添加。

好处：比较合适动态元素的绑定，新添加的子元素也会有监听函数，也可以有事件触发机制。

## **● 说一下图片的懒加载和预加载**

### **参考回答：**

预加载：提前加载图片，当用户需要查看时可直接从本地缓存中渲染。

懒加载：懒加载的主要目的是作为服务器前端的优化，减少请求数或延迟请求数。

两种技术的本质：两者的行为是相反的，一个是提前加载，一个是迟缓甚至不加载。  
懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

## **● mouseover和mouseenter的区别**

### **参考回答：**

mouseover：当鼠标移入元素或其子元素都会触发事件，所以有一个重复触发，冒泡的过程。对应的移除事件是mouseout

mouseenter：当鼠标移除元素本身（不包含元素的子元素）会触发事件，也就是不会冒泡，对应的移除事件是mouseleave

## **● js的new操作符做了哪些事情**

### **参考回答：**

new 操作符新建了一个空对象，这个对象原型指向构造函数的prototype，执行构造函数后返回这个对象。

## **● 改变函数内部this指针的指向函数（bind，apply，call的区别）**

### **参考回答：**

通过apply和call改变函数的this指向，他们两个函数的第一个参数都是一样的表示要改变指向的那个对象，第二个参数，apply是数组，而call则是arg1,arg2...这种形式。通过bind改变this作用域会返回一个新的函数，这个函数不会马上执行。

## **● js的各种位置，比如clientHeight,scrollHeight,offsetHeight ,以及scrollTop, offsetTop,clientTop的区别？**

### **参考回答：**

clientHeight：表示的是可视区域的高度，不包含border和滚动条

offsetHeight：表示可视区域的高度，包含了border和滚动条

scrollHeight：表示了所有区域的高度，包含了因为滚动被隐藏的部分。

clientTop：表示边框border的厚度，在未指定的情况下一般为0

scrollTop：滚动后被隐藏的高度，获取对象相对于由offsetParent属性指定的父坐标(css定位的元素或body元素)距离顶端的高度。

## **● js拖拽功能的实现**

### **参考回答：**

首先是三个事件，分别是mousedown，mousemove，mouseup

当鼠标点击按下的时候，需要一个tag标识此时已经按下，可以执行mousemove里面的具体方法。

clientX，clientY标识的是鼠标的坐标，分别标识横坐标和纵坐标，并且我们用offsetX和offsetY来表示元素的元素的初始坐标，移动的举例应该是：

鼠标移动时候的坐标-鼠标按下去时候的坐标。

也就是说定位信息为：

鼠标移动时候的坐标-鼠标按下去时候的坐标+元素初始情况下的offetLeft.

还有一点也是原理性的东西，也就是拖拽的同时是绝对定位，我们改变的是绝对定位条件下的left  
以及top等等值。

补充：也可以通过html5的拖放（Drag 和 drop）来实现

## **● 异步加载js的方法**

### **参考回答：**

defer：只支持IE如果您的脚本不会改变文档的内容，可将 defer 属性加入到<script>标签中，以便加快处理文档的速度。因为浏览器知道它将能够安全地读取文档的剩余部分而不用执行脚本，它将推迟对脚本的解释，直到文档已经显示给用户为止。

async，HTML5属性仅适用于外部脚本，并且如果在IE中，同时存在defer和async，那么defer的优先级比较高，脚本将在页面完成时执行。

创建script标签，插入到DOM中

## **● Ajax解决浏览器缓存问题**

### **参考回答：**

在ajax发送请求前加上 anyAjaxObj.setRequestHeader("If-Modified-Since","0")。

在ajax发送请求前加上 anyAjaxObj.setRequestHeader("Cache-Control","no-cache")。

在URL后面加上一个随机数： "fresh=" + Math.random()。

在URL后面加上时间搓："nowtime=" + new Date().getTime()。

如果是使用jQuery，直接这样就可以了 $.ajaxSetup({cache:false})。这样页面的所有ajax都会执行这条语句就是不需要保存缓存记录。

## **● js的节流和防抖**

### **参考回答：**

[http://www.cnblogs.com/coco1s/p/5499469.html](http://www.cnblogs.com/coco1s/p/5499469.html" \t "https://www.nowcoder.com/tutorial/96/_blank)

## **● JS中的垃圾回收机制**

### **参考回答：**

必要性：由于字符串、对象和数组没有固定大小，所有当他们的大小已知时，才能对他们进行动态的存储分配。JavaScript程序每次创建字符串、数组或对象时，解释器都必须分配内存来存储那个实体。只要像这样动态地分配了内存，最终都要释放这些内存以便他们能够被再用，否则，JavaScript的解释器将会消耗完系统中所有可用的内存，造成系统崩溃。

这段话解释了为什么需要系统需要垃圾回收，JS不像C/C++，他有自己的一套垃圾回收机制（Garbage Collection）。JavaScript的解释器可以检测到何时程序不再使用一个对象了，当他确定了一个对象是无用的时候，他就知道不再需要这个对象，可以把它所占用的内存释放掉了。例如：

var a="hello world";

var b="world";

var a=b;

//这时，会释放掉"hello world"，释放内存以便再引用

垃圾回收的方法：标记清除、计数引用。

标记清除

这是最常见的垃圾回收方式，当变量进入环境时，就标记这个变量为”进入环境“,从逻辑上讲，永远不能释放进入环境的变量所占的内存，永远不能释放进入环境变量所占用的内存，只要执行流程进入相应的环境，就可能用到他们。当离开环境时，就标记为离开环境。

垃圾回收器在运行的时候会给存储在内存中的变量都加上标记（所有都加），然后去掉环境变量中的变量，以及被环境变量中的变量所引用的变量（条件性去除标记），删除所有被标记的变量，删除的变量无法在环境变量中被访问所以会被删除，最后垃圾回收器，完成了内存的清除工作，并回收他们所占用的内存。

引用计数法

另一种不太常见的方法就是引用计数法，引用计数法的意思就是每个值没引用的次数，当声明了一个变量，并用一个引用类型的值赋值给改变量，则这个值的引用次数为1,；相反的，如果包含了对这个值引用的变量又取得了另外一个值，则原先的引用值引用次数就减1，当这个值的引用次数为0的时候，说明没有办法再访问这个值了，因此就把所占的内存给回收进来，这样垃圾收集器再次运行的时候，就会释放引用次数为0的这些值。

用引用计数法会存在内存泄露，下面来看原因：

function problem() {

var objA = new Object();

var objB = new Object();

objA.someOtherObject = objB;

objB.anotherObject = objA;

}

在这个例子里面，objA和objB通过各自的属性相互引用，这样的话，两个对象的引用次数都为2，在采用引用计数的策略中，由于函数执行之后，这两个对象都离开了作用域，函数执行完成之后，因为计数不为0，这样的相互引用如果大量存在就会导致内存泄露。

特别是在DOM对象中，也容易存在这种问题：

var element=document.getElementById（’‘）；

var myObj=new Object();

myObj.element=element;

element.someObject=myObj;

这样就不会有垃圾回收的过程。

## **● eval是做什么的**

### **参考回答：**

它的功能是将对应的字符串解析成js并执行，应该避免使用js，因为非常消耗性能（2次，一次解析成js，一次执行）

## **● 如何理解前端模块化**

### **参考回答：**

前端模块化就是复杂的文件编程一个一个独立的模块，比如js文件等等，分成独立的模块有利于重用（复用性）和维护（版本迭代），这样会引来模块之间相互依赖的问题，所以有了commonJS规范，AMD，CMD规范等等，以及用于js打包（编译等处理）的工具webpack

## **● 说一下Commonjs、AMD和CMD**

### **参考回答：**

一个模块是能实现特定功能的文件，有了模块就可以方便的使用别人的代码，想要什么功能就能加载什么模块。

Commonjs：开始于服务器端的模块化，同步定义的模块化，每个模块都是一个单独的作用域，模块输出，modules.exports，模块加载require()引入模块。

AMD：中文名异步模块定义的意思。

requireJS实现了AMD规范，主要用于解决下述两个问题。

1.多个文件有依赖关系，被依赖的文件需要早于依赖它的文件加载到浏览器

2.加载的时候浏览器会停止页面渲染，加载文件越多，页面失去响应的时间越长。

语法：requireJS定义了一个函数define，它是全局变量，用来定义模块。

requireJS的例子：

//定义模块

define(['dependency'], function(){

var name = 'Byron';

function printName(){

console.log(name);

}

return {

printName: printName

};

});

//加载模块

require(['myModule'], function (my){

my.printName();

}

requirejs定义了一个函数define,它是全局变量，用来定义模块：

define(id?dependencies?,factory)

在页面上使用模块加载函数：

require([dependencies],factory)；

总结AMD规范：require（）函数在加载依赖函数的时候是异步加载的，这样浏览器不会失去响应，它指定的回调函数，只有前面的模块加载成功，才会去执行。  
因为网页在加载js的时候会停止渲染，因此我们可以通过异步的方式去加载js,而如果需要依赖某些，也是异步去依赖，依赖后再执行某些方法。

## **● 对象深度克隆的简单实现**

### **参考回答：**

function deepClone(obj){

var newObj= obj instanceof Array ? []:{};

for(var item in obj){

var temple= typeof obj[item] == 'object' ? deepClone(obj[item]):obj[item];

newObj[item] = temple;

}

return newObj;

}

ES5的常用的对象克隆的一种方式。注意数组是对象，但是跟对象又有一定区别，所以我们一开始判断了一些类型，决定newObj是对象还是数组~

## **● 实现一个once函数，传入函数参数只执行一次**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | function ones(func){  var tag=true;  return function(){  if(tag==true){  func.apply(null,arguments);  tag=false;  }  return undefined  }  } |

## **● 将原生的ajax封装成promise**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var  myNewAjax=function(url){  return new Promise(function(resolve,reject){  var xhr = new XMLHttpRequest();  xhr.open('get',url);  xhr.send(data);  xhr.onreadystatechange=function(){  if(xhr.status==200&&readyState==4){  var json=JSON.parse(xhr.responseText);  resolve(json)  }else if(xhr.readyState==4&&xhr.status!=200){  reject('error');  }  }  })  } |

## **● js监听对象属性的改变**

### **参考回答：**

我们假设这里有一个user对象,

(1)在ES5中可以通过Object.defineProperty来实现已有属性的监听

Object.defineProperty(user,'name',{

set：function(key,value){

}

})

缺点：如果id不在user对象中，则不能监听id的变化

(2)在ES6中可以通过Proxy来实现

var  user = new Proxy({}，{

set：function(target,key,value,receiver){

}

})

这样即使有属性在user中不存在，通过user.id来定义也同样可以这样监听这个属性的变化哦~

## **● 如何实现一个私有变量，用getName方法可以访问，不能直接访问**

### **参考回答：**

(1)通过defineProperty来实现

obj={

name:yuxiaoliang,

getName:function(){

return this.name

}

}

object.defineProperty(obj,"name",{

//不可枚举不可配置

});

(2)通过函数的创建形式

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function product(){  var name='yuxiaoliang';  this.getName=function(){  return name;  }  }  var obj=new product(); |

## **● ==和===、以及Object.is的区别**

### **参考回答：**

(1) ==

主要存在：强制转换成number,null==undefined

" "==0  //true

"0"==0  //true

" " !="0" //true

123=="123" //true

null==undefined //true

(2)Object.js

主要的区别就是+0！=-0 而NaN==NaN  
(相对比===和==的改进)

## **● setTimeout、setInterval和requestAnimationFrame之间的区别**

### **参考回答：**

这里有一篇文章讲的是requestAnimationFrame：[http://www.cnblogs.com/xiaohuochai/p/5777186.html](http://www.cnblogs.com/xiaohuochai/p/5777186.html" \t "https://www.nowcoder.com/tutorial/96/_blank)

与setTimeout和setInterval不同，requestAnimationFrame不需要设置时间间隔，  
大多数电脑显示器的刷新频率是60Hz，大概相当于每秒钟重绘60次。大多数浏览器都会对重绘操作加以限制，不超过显示器的重绘频率，因为即使超过那个频率用户体验也不会有提升。因此，最平滑动画的最佳循环间隔是1000ms/60，约等于16.6ms。

RAF采用的是系统时间间隔，不会因为前面的任务，不会影响RAF，但是如果前面的任务多的话，  
会响应setTimeout和setInterval真正运行时的时间间隔。

特点：  
（1）requestAnimationFrame会把每一帧中的所有DOM操作集中起来，在一次重绘或回流中就完成，并且重绘或回流的时间间隔紧紧跟随浏览器的刷新频率。

（2）在隐藏或不可见的元素中，requestAnimationFrame将不会进行重绘或回流，这当然就意味着更少的CPU、GPU和内存使用量

（3）requestAnimationFrame是由浏览器专门为动画提供的API，在运行时浏览器会自动优化方法的调用，并且如果页面不是激活状态下的话，动画会自动暂停，有效节省了CPU开销。

## **● 实现一个两列等高布局，讲讲思路**

### **参考回答：**

为了实现两列等高，可以给每列加上 padding-bottom:9999px;

margin-bottom:-9999px;同时父元素设置overflow:hidden;

## **● 自己实现一个bind函数**

### **参考回答：**

原理：通过apply或者call方法来实现。

(1)初始版本

Function.prototype.bind=function(obj,arg){

var arg=Array.prototype.slice.call(arguments,1);

var context=this;

return function(newArg){

arg=arg.concat(Array.prototype.slice.call(newArg));

return context.apply(obj,arg);

}

}

(2) 考虑到原型链

为什么要考虑？因为在new 一个bind过生成的新函数的时候，必须的条件是要继承原函数的原型

Function.prototype.bind=function(obj,arg){

var arg=Array.prototype.slice.call(arguments,1);

var context=this;

var bound=function(newArg){

arg=arg.concat(Array.prototype.slice.call(newArg));

return context.apply(obj,arg);

}

var F=function(){}

//这里需要一个寄生组合继承

|  |  |
| --- | --- |
| 1  2  3  4 | F.prototype=context.prototype;  bound.prototype=**new** F();  **return** bound;  } |

## **● 用setTimeout来实现setInterval**

### **参考回答：**

(1)用setTimeout()方法来模拟setInterval()与setInterval()之间的什么区别？

首先来看setInterval的缺陷，使用setInterval()创建的定时器确保了定时器代码规则地插入队列中。这个问题在于：如果定时器代码在代码再次添加到队列之前还没完成执行，结果就会导致定时器代码连续运行好几次。而之间没有间隔。不过幸运的是：javascript引擎足够聪明，能够避免这个问题。当且仅当没有该定时器的如何代码实例时，才会将定时器代码添加到队列中。这确保了定时器代码加入队列中最小的时间间隔为指定时间。

这种重复定时器的规则有两个问题：1.某些间隔会被跳过 2.多个定时器的代码执行时间可能会比预期小。

下面举例子说明：

假设，某个onclick事件处理程序使用啦setInterval()来设置了一个200ms的重复定时器。如果事件处理程序花了300ms多一点的时间完成。

<img width="626" alt="2018-07-10 11 36 43" src="[https://user-images.githubusercontent.com/17233651/42487876-92656f2c-8435-11e8-8a5f-0a97918039da.png](https://user-images.githubusercontent.com/17233651/42487876-92656f2c-8435-11e8-8a5f-0a97918039da.png" \t "https://www.nowcoder.com/tutorial/96/_blank)">

这个例子中的第一个定时器是在205ms处添加到队列中，但是要过300ms才能执行。在405ms又添加了一个副本。在一个间隔，605ms处，第一个定时器代码还在执行中，而且队列中已经有了一个定时器实例，结果是605ms的定时器代码不会添加到队列中。结果是在5ms处添加的定时器代码执行结束后，405处的代码立即执行。

function say(){

//something

setTimeout(say,200);

}

setTimeout(say,200)

或者

|  |  |
| --- | --- |
| 1  2  3  4 | setTimeout(function(){  //do something  setTimeout(arguments.callee,200);  },200); |

## **● js怎么控制一次加载一张图片，加载完后再加载下一张**

### **参考回答：**

(1)方法1

<script type="text/javascript">

var obj=new Image();

obj.src="[http://www.phpernote.com/uploadfiles/editor/201107240502201179.jpg](http://www.phpernote.com/uploadfiles/editor/201107240502201179.jpg" \t "https://www.nowcoder.com/tutorial/96/_blank)";

obj.onload=function(){

alert('图片的宽度为：'+obj.width+'；图片的高度为：'+obj.height);

document.getElementById("mypic").innnerHTML="<img src='"+this.src+"' />";

}

</script>

<div id="mypic">onloading……</div>

(2)方法2

<script type="text/javascript">

var obj=new Image();

obj.src="[http://www.phpernote.com/uploadfiles/editor/201107240502201179.jpg](http://www.phpernote.com/uploadfiles/editor/201107240502201179.jpg" \t "https://www.nowcoder.com/tutorial/96/_blank)";

obj.onreadystatechange=function(){

if(this.readyState=="complete"){

alert('图片的宽度为：'+obj.width+'；图片的高度为：'+obj.height);

|  |  |
| --- | --- |
| 1  2  3  4  5 | document.getElementById("mypic").innnerHTML="<img src='"+this.src+"' />";  }  }  </script>  <div id="mypic">onloading……</div> |

## **● 代码的执行顺序**

### **参考回答：**

setTimeout(function(){console.log(1)},0);

new Promise(function(resolve,reject){

console.log(2);

resolve();

}).then(function(){console.log(3)

}).then(function(){console.log(4)});

process.nextTick(function(){console.log(5)});

console.log(6);

//输出2,6,5,3,4,1

为什么呢？具体请参考我的文章： [从promise、process.nextTick、setTimeout出发，谈谈Event Loop中的Job queue](https://github.com/forthealllight/blog/issues/5" \t "https://www.nowcoder.com/tutorial/96/_blank)

## **● 如何实现sleep的效果（es5或者es6）**

### **参考回答：**

(1)while循环的方式

function sleep(ms){

var start=Date.now(),expire=start+ms;

while(Date.now()<expire);

console.log('1111');

return;

}

执行sleep(1000)之后，休眠了1000ms之后输出了1111。上述循环的方式缺点很明显，容易造成死循环。

(2)通过promise来实现

function sleep(ms){

var temple=new Promise(

(resolve)=>{

console.log(111);setTimeout(resolve,ms)

});

return temple

}

sleep(500).then(function(){

//console.log(222)

})

//先输出了111，延迟500ms后输出222

(3)通过async封装

function sleep(ms){

return new Promise((resolve)=>setTimeout(resolve,ms));

}

async function test(){

var temple=await sleep(1000);

console.log(1111)

return temple

}

test();

//延迟1000ms输出了1111

(4).通过generate来实现

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function\* sleep(ms){  yield new Promise(function(resolve,reject){  console.log(111);  setTimeout(resolve,ms);  })  }  sleep(500).next().value.then(function(){console.log(2222)}) |

## **● 简单的实现一个promise**

### **参考回答：**

[首先明确什么是promiseA+规范，参考规范的地址：primise](https://promisesaplus.com/" \t "https://www.nowcoder.com/tutorial/96/_blank)

A+规范

如何实现一个promise，参考我的文章：

[实现一个完美符合Promise/A+规范的Promise](https://github.com/forthealllight/blog/issues/4" \t "https://www.nowcoder.com/tutorial/96/_blank)

一般不会问的很详细，只要能写出上述文章中的v1.0版本的简单promise即可。

## **● Function.\_proto\_(getPrototypeOf)是什么？**

### **参考回答：**

获取一个对象的原型，在chrome中可以通过\_proto\_的形式，或者在ES6中可以通过Object.getPrototypeOf的形式。

那么Function.proto是什么么？也就是说Function由什么对象继承而来，我们来做如下判别。

Function.\_\_proto\_\_==Object.prototype //false

Function.\_\_proto\_\_==Function.prototype//true

我们发现Function的原型也是Function。

我们用图可以来明确这个关系：

<img width="646" alt="2018-07-10 2 38 27" src="[https://user-images.githubusercontent.com/17233651/42493275-f55d0860-844e-11e8-983f-e04189a4f3d8.png](https://user-images.githubusercontent.com/17233651/42493275-f55d0860-844e-11e8-983f-e04189a4f3d8.png" \t "https://www.nowcoder.com/tutorial/96/_blank)">

## **● 实现js中所有对象的深度克隆（包装对象，Date对象，正则对象）**

### **参考回答：**

通过递归可以简单实现对象的深度克隆，但是这种方法不管是ES6还是ES5实现，都有同样的缺陷，就是只能实现特定的object的深度复制（比如数组和函数），不能实现包装对象Number，String ， Boolean，以及Date对象，RegExp对象的复制。

(1)前文的方法

function deepClone(obj){

var newObj= obj instanceof Array?[]:{};

for(var i in obj){

newObj[i]=typeof obj[i]=='object'?

deepClone(obj[i]):obj[i];

}

return newObj;

}

这种方法可以实现一般对象和数组对象的克隆，比如：

var arr=[1,2,3];

var newArr=deepClone(arr);

// newArr->[1,2,3]

var obj={

x:1,

y:2

}

var newObj=deepClone(obj);

// newObj={x:1,y:2}

但是不能实现例如包装对象Number,String,Boolean,以及正则对象RegExp和Date对象的克隆，比如：

//Number包装对象

var num=new Number(1);

typeof num // "object"

var newNum=deepClone(num);

//newNum ->  {} 空对象

//String包装对象

var str=new String("hello");

typeof str //"object"

var newStr=deepClone(str);

//newStr->  {0:'h',1:'e',2:'l',3:'l',4:'o'};

//Boolean包装对象

var bol=new Boolean(true);

typeof bol //"object"

var newBol=deepClone(bol);

// newBol ->{} 空对象

....

(2)valueof()函数

所有对象都有valueOf方法，valueOf方法对于：如果存在任意原始值，它就默认将对象转换为表示它的原始值。对象是复合值，而且大多数对象无法真正表示为一个原始值，因此默认的valueOf()方法简单地返回对象本身，而不是返回一个原始值。数组、函数和正则表达式简单地继承了这个默认方法，调用这些类型的实例的valueOf()方法只是简单返回这个对象本身。

对于原始值或者包装类：

function baseClone(base){

return base.valueOf();

}

//Number

var num=new Number(1);

var newNum=baseClone(num);

//newNum->1

//String

var str=new String('hello');

var newStr=baseClone(str);

// newStr->"hello"

//Boolean

var bol=new Boolean(true);

var newBol=baseClone(bol);

//newBol-> true

其实对于包装类，完全可以用=号来进行克隆，其实没有深度克隆一说，

这里用valueOf实现，语法上比较符合规范。

对于Date类型：

因为valueOf方法，日期类定义的valueOf()方法会返回它的一个内部表示：1970年1月1日以来的毫秒数.因此我们可以在Date的原型上定义克隆的方法：

Date.prototype.clone=function(){

return new Date(this.valueOf());

}

var date=new Date('2010');

var newDate=date.clone();

// newDate->  Fri Jan 01 2010 08:00:00 GMT+0800

对于正则对象RegExp：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | RegExp.prototype.clone = function() {  var pattern = this.valueOf();  var flags = '';  flags += pattern.global ? 'g' : '';  flags += pattern.ignoreCase ? 'i' : '';  flags += pattern.multiline ? 'm' : '';  return new RegExp(pattern.source, flags);  };  var reg=new RegExp('/111/');  var newReg=reg.clone();  //newReg->  /\/111\// |

## **● 简单实现Node的Events模块**

### **参考回答：**

简介：观察者模式或者说订阅模式，它定义了对象间的一种一对多的关系，让多个观察者对象同时监听某一个主题对象，当一个对象发生改变时，所有依赖于它的对象都将得到通知。

node中的Events模块就是通过观察者模式来实现的：

var events=require('events');

var eventEmitter=new events.EventEmitter();

eventEmitter.on('say',function(name){

console.log('Hello',name);

})

eventEmitter.emit('say','Jony yu');

这样，eventEmitter发出say事件，通过On接收，并且输出结果，这就是一个订阅模式的实现，下面我们来简单的实现一个Events模块的EventEmitter。

(1)实现简单的Event模块的emit和on方法

function Events(){

this.on=function(eventName,callBack){

if(!this.handles){

this.handles={};

}

if(!this.handles[eventName]){

this.handles[eventName]=[];

}

this.handles[eventName].push(callBack);

}

this.emit=function(eventName,obj){

if(this.handles[eventName]){

for(var i=0;o<this.handles[eventName].length;i++){

this.handles[eventName][i](obj);

}

}

}

return this;

}

这样我们就定义了Events，现在我们可以开始来调用：

var events=new Events();

events.on('say',function(name){

console.log('Hello',nama)

});

events.emit('say','Jony yu');

//结果就是通过emit调用之后，输出了Jony yu

(2)每个对象是独立的

因为是通过new的方式，每次生成的对象都是不相同的，因此：

var event1=new Events();

var event2=new Events();

event1.on('say',function(){

console.log('Jony event1');

});

event2.on('say',function(){

console.log('Jony event2');

})

event1.emit('say');

event2.emit('say');

//event1、event2之间的事件监听互相不影响

//输出结果为'Jony event1' 'Jony event2'

## **● 箭头函数中this指向举例**

### **参考回答：**

var a=11;

function test2(){

this.a=22;

let b=()=>{console.log(this.a)}

b();

}

var x=new test2();

//输出22

定义时绑定。

## **js判断类型**

### **参考回答：**

判断方法：typeof()，instanceof，Object.prototype.toString.call()等

## **● 数组常用方法**

### **参考回答：**

push()，pop()，shift()，unshift()，splice()，sort()，reverse()，map()等

## **● 数组去重**

### **参考回答：**

法一：indexOf循环去重

法二：ES6 Set去重；Array.from(new Set(array))

法三：Object 键值对去重；把数组的值存成 Object 的 key 值，比如 Object[value1] = true，在判断另一个值的时候，如果 Object[value2]存在的话，就说明该值是重复的。

## **● 闭包 有什么用**

### **参考回答：**

（1）什么是闭包：

闭包是指有权访问另外一个函数作用域中的变量的函数。

闭包就是函数的局部变量集合，只是这些局部变量在函数返回后会继续存在。闭包就是就是函数的“堆栈”在函数返回后并不释放，我们也可以理解为这些函数堆栈并不在栈上分配而是在堆上分配。当在一个函数内定义另外一个函数就会产生闭包。

（2）为什么要用：

匿名自执行函数：我们知道所有的变量，如果不加上var关键字，则默认的会添加到全局对象的属性上去，这样的临时变量加入全局对象有很多坏处，比如：别的函数可能误用这些变量；造成全局对象过于庞大，影响访问速度(因为变量的取值是需要从原型链上遍历的)。除了每次使用变量都是用var关键字外，我们在实际情况下经常遇到这样一种情况，即有的函数只需要执行一次，其内部变量无需维护，可以用闭包。

结果缓存：我们开发中会碰到很多情况，设想我们有一个处理过程很耗时的函数对象，每次调用都会花费很长时间，那么我们就需要将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。闭包正是可以做到这一点，因为它不会释放外部的引用，从而函数内部的值可以得以保留。

封装：实现类和继承等。

## **● 事件代理在捕获阶段的实际应用**

### **参考回答：**

可以在父元素层面阻止事件向子元素传播，也可代替子元素执行某些操作。

## **● 去除字符串首尾空格**

### **参考回答：**

使用正则(^\s\*)|(\s\*$)即可

## **● 性能优化**

### **参考回答：**

减少HTTP请求

使用内容发布网络（CDN）

添加本地缓存

压缩资源文件

将CSS样式表放在顶部，把javascript放在底部（浏览器的运行机制决定）

避免使用CSS表达式

减少DNS查询

使用外部javascript和CSS

避免重定向

图片lazyLoad

## **● 来讲讲JS的闭包吧**

### **参考回答：**

闭包是指有权访问另外一个函数作用域中的变量的函数。

闭包就是函数的局部变量集合，只是这些局部变量在函数返回后会继续存在。闭包就是就是函数的“堆栈”在函数返回后并不释放，我们也可以理解为这些函数堆栈并不在栈上分配而是在堆上分配。当在一个函数内定义另外一个函数就会产生闭包。

（2）为什么要用：

匿名自执行函数：我们知道所有的变量，如果不加上var关键字，则默认的会添加到全局对象的属性上去，这样的临时变量加入全局对象有很多坏处，比如：别的函数可能误用这些变量；造成全局对象过于庞大，影响访问速度(因为变量的取值是需要从原型链上遍历的)。除了每次使用变量都是用var关键字外，我们在实际情况下经常遇到这样一种情况，即有的函数只需要执行一次，其内部变量无需维护，可以用闭包。

结果缓存：我们开发中会碰到很多情况，设想我们有一个处理过程很耗时的函数对象，每次调用都会花费很长时间，那么我们就需要将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。闭包正是可以做到这一点，因为它不会释放外部的引用，从而函数内部的值可以得以保留。

## **● 能来讲讲JS的语言特性吗**

### **参考回答：**

运行在客户端浏览器上；

不用预编译，直接解析执行代码；

是弱类型语言，较为灵活；

与操作系统无关，[跨平台](https://www.baidu.com/s?wd=%E8%B7%A8%E5%B9%B3%E5%8F%B0&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://www.nowcoder.com/tutorial/96/_blank)的语言；

脚本语言、解释性语言

## **● 如何判断一个数组(讲到typeof差点掉坑里)**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2 | Object.prototype.call.toString()  instanceof |

## **● 你说到typeof，能不能加一个限制条件达到判断条件**

### **参考回答：**

typeof只能判断是object,可以判断一下是否拥有数组的方法

## **● JS实现跨域**

### **参考回答：**

JSONP：通过动态创建script，再请求一个带参网址实现跨域通信。document.domain + iframe跨域：两个页面都通过js强制设置document.domain为基础主域，就实现了同域。

location.hash + iframe跨域：a欲与b跨域相互通信，通过中间页c来实现。 三个页面，不同域之间利用iframe的location.hash传值，相同域之间直接js访问来通信。

window.name + iframe跨域：通过iframe的src属性由外域转向本地域，跨域数据即由iframe的window.name从外域传递到本地域。

postMessage跨域：可以跨域操作的window属性之一。

CORS：服务端设置Access-Control-Allow-Origin即可，前端无须设置，若要带cookie请求，前后端都需要设置。

代理跨域：启一个代理服务器，实现数据的转发

参考[https://segmentfault.com/a/1190000011145364](https://segmentfault.com/a/1190000011145364" \t "https://www.nowcoder.com/tutorial/96/_blank)

## **● Js基本数据类型**

### **参考回答：**

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

## **●  js的命名方式**

## **● js深度拷贝一个元素的具体实现**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var deepCopy = function(obj) {  if (typeof obj !== 'object') return;  var newObj = obj instanceof Array ? [] : {};  for (var key in obj) {  if (obj.hasOwnProperty(key)) {  newObj[key] = typeof obj[key] === 'object' ? deepCopy(obj[key]) : obj[key];  }  }  return newObj;  } |

## **● 之前说了ES6set可以数组去重，是否还有数组去重的方法**

### **参考回答：**

法一：indexOf循环去重

法二：Object 键值对去重；把数组的值存成 Object 的 key 值，比如 Object[value1] = true，在判断另一个值的时候，如果 Object[value2]存在的话，就说明该值是重复的。

## **● 重排和重绘，讲讲看**

### **参考回答：**

重绘（repaint或redraw）：当盒子的位置、大小以及其他属性，例如颜色、字体大小等都确定下来之后，浏览器便把这些原色都按照各自的特性绘制一遍，将内容呈现在页面上。重绘是指一个元素外观的改变所触发的浏览器行为，浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。

触发重绘的条件：改变元素外观属性。如：color，background-color等。

注意：table及其内部元素可能需要多次计算才能确定好其在渲染树中节点的属性值，比同等元素要多花两倍时间，这就是我们尽量避免使用table布局页面的原因之一。

重排（重构/回流/reflow）：当渲染树中的一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建, 这就称为回流(reflow)。每个页面至少需要一次回流，就是在页面第一次加载的时候。

重绘和重排的关系：在回流的时候，浏览器会使渲染树中受到影响的部分失效，并重新构造这部分渲染树，完成回流后，浏览器会重新绘制受影响的部分到屏幕中，该过程称为重绘。所以，重排必定会引发重绘，但重绘不一定会引发重排。

## **● JS的全排列**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | function permutate(str) {  var result = [];  if(str.length > 1) {  var left = str[0];  var rest = str.slice(1, str.length);  var preResult = permutate(rest);  for(var i=0; i<preResult.length; i++) {  for(var j=0; j<preResult[i].length; j++) {  var tmp = preResult[i],slice(0, j) + left + preResult[i].slice(j, preResult[i].length);  result.push(tmp);  }  }  } else if (str.length == 1) {  return [str];  }  return result;  } |

## **●  ES6中用过哪些**

## **● 跨域的原理**

### **参考回答：**

跨域，是指浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的，是浏览器对JavaScript实施的安全限制，那么只要协议、域名、端口有任何一个不同，都被当作是不同的域。跨域原理，即是通过各种方式，避开浏览器的安全限制。

## **● 不同数据类型的值的比较，是怎么转换的，有什么规则**

### **参考回答：**



## **● null == undefined为什么**

### **参考回答：**

要比较相等性之前，不能将null 和 undefined 转换成其他任何值，但 null == undefined 会返回 true 。ECMAScript规范中是这样定义的。

## **● this的指向 哪几种**

### **参考回答：**

默认绑定：全局环境中，this默认绑定到window。

隐式绑定：一般地，被直接对象所包含的函数调用时，也称为方法调用，this隐式绑定到该直接对象。

隐式丢失：隐式丢失是指被隐式绑定的函数丢失绑定对象，从而默认绑定到window。显式绑定：通过call()、apply()、bind()方法把对象绑定到this上，叫做显式绑定。

new绑定：如果函数或者方法调用之前带有关键字new，它就构成构造函数调用。对于this绑定来说，称为new绑定。

【1】构造函数通常不使用return关键字，它们通常初始化新对象，当构造函数的函数体执行完毕时，它会显式返回。在这种情况下，构造函数调用表达式的计算结果就是这个新对象的值。

【2】如果构造函数使用return语句但没有指定返回值，或者返回一个原始值，那么这时将忽略返回值，同时使用这个新对象作为调用结果。

【3】如果构造函数显式地使用return语句返回一个对象，那么调用表达式的值就是这个对象。

## **● 暂停死区**

### **参考回答：**

在代码块内，使用let、const命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”

## **● AngularJS双向绑定原理**

### **参考回答：**

Angular将双向绑定转换为一堆watch表达式，然后递归这些表达式检查是否发生过变化，如果变了则执行相应的watcher函数（指view上的指令，如ng-bind，ng-show等或是{{}}）。等到model中的值不再发生变化，也就不会再有watcher被触发，一个完整的digest循环就完成了。

Angular中在view上声明的事件指令，如：ng-click、ng-change等，会将浏览器的事件转发给$scope上相应的model的响应函数。等待相应函数改变model，紧接着触发脏检查机制刷新view。

watch表达式：可以是一个函数、可以是$scope上的一个属性名，也可以是一个字符串形式的表达式。$watch函数所监听的对象叫做watch表达式。watcher函数：指在view上的指令（ngBind，ngShow、ngHide等）以及{{}}表达式，他们所注册的函数。每一个watcher对象都包括：监听函数，上次变化的值，获取监听表达式的方法以及监听表达式，最后还包括是否需要使用深度对比（angular.equals()）

## **● 写一个深度拷贝**

### **参考回答：**

function clone( obj ) {

var copy;

switch( typeof obj ) {

case "undefined":

break;

case "number":

copy = obj - 0;

break;

case "string":

copy = obj + "";

break;

case "boolean":

copy = obj;

break;

case "object":  //object分为两种情况 对象（Object）和数组（Array）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | if(obj === null) {  copy = null;  } else {  if( Object.prototype.toString.call(obj).slice(8, -1) === "Array") {  copy = [];  for( var i = 0 ; i < obj.length ; i++ ) {  copy.push(clone(obj[i]));  }  } else {  copy = {};  for( var j in obj) {  copy[j] = clone(obj[j]);  }  }  }  break;  default:  copy = obj;  break;  }  return copy;  } |

## **● 简历中提到了requestAnimationFrame，请问是怎么使用的**

### **参考回答：**

requestAnimationFrame() 方法告诉浏览器您希望执行动画并请求浏览器在下一次重绘之前调用指定的函数来更新动画。该方法使用一个回调函数作为参数，这个回调函数会在浏览器重绘之前调用。

## **● 有一个游戏叫做Flappy Bird，就是一只小鸟在飞，前面是无尽的沙漠，上下不断有钢管生成，你要躲避钢管。然后小明在玩这个游戏时候老是卡顿甚至崩溃，说出原因（3-5个）以及解决办法（3-5个）**

### **参考回答：**

原因可能是：

1.内存溢出问题。

2.资源过大问题。

3.资源加载问题。

4.canvas绘制频率问题

解决办法：

1.针对内存溢出问题，我们应该在钢管离开可视区域后，销毁钢管，让垃圾收集器回收钢管，因为不断生成的钢管不及时清理容易导致内存溢出游戏崩溃。

2.针对资源过大问题，我们应该选择图片文件大小更小的图片格式，比如使用webp、png格式的图片，因为绘制图片需要较大计算量。

3.针对资源加载问题，我们应该在可视区域之前就预加载好资源，如果在可视区域生成钢管的话，用户的体验就认为钢管是卡顿后才生成的，不流畅。

4.针对canvas绘制频率问题，我们应该需要知道大部分显示器刷新频率为60次/s,因此游戏的每一帧绘制间隔时间需要小于1000/60=16.7ms，才能让用户觉得不卡顿。

（注意因为这是单机游戏，所以回答与网络无关）

## **● 编写代码，满足以下条件： （1）Hero("37er");执行结果为 Hi! This is 37er （2）Hero("37er").kill(1).recover(30);执行结果为 Hi! This is 37er Kill 1 bug Recover 30 bloods （3）Hero("37er").sleep(10).kill(2)执行结果为 Hi! This is 37er //等待10s后 Kill 2 bugs  //注意为bugs （双斜线后的为提示信息，不需要打印）**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | function Hero(name){  let o=new Object();  o.name=name;  o.time=0;  console.log("Hi! This is "+o.name);  o.kill=function(bugs) {  if(bugs==1){  console.log("Kill "+(bugs)+" bug");  }else {  setTimeout(function () {  console.log("Kill " + (bugs) + " bugs");  }, 1000 \* this.time);  }  return o;  };  o.recover=function (bloods) {  console.log("Recover "+(bloods)+" bloods");  return o;  }  o.sleep=function (sleepTime) {  o.time=sleepTime;  return o;  }  return o;  } |

## **●  jit;jc**

## **● es6新特性用过哪些**

## **● 什么是按需加载**

### **参考回答：**

当用户触发了动作时才加载对应的功能。触发的动作，是要看具体的业务场景而言，包括但不限于以下几个情况：鼠标点击、输入文字、拉动滚动条，鼠标移动、窗口大小更改等。加载的文件，可以是JS、图片、CSS、HTML等。

## **● 说一下什么是virtual dom**

### **参考回答：**

用JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中 当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异 把所记录的差异应用到所构建的真正的DOM树上，视图就更新了。Virtual DOM 本质上就是在 JS 和 DOM 之间做了一个缓存。

## **● webpack用来干什么的**

### **参考回答：**

webpack 是一个现代 JavaScript 应用程序的静态模块打包器(module bundler)。当 webpack 处理应用程序时，它会递归地构建一个依赖关系图(dependency graph)，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个bundle。

## **● ant-design优点和缺点**

### **参考回答：**

优点：组件非常全面，样式效果也都比较不错。

缺点：框架自定义程度低，默认UI风格修改困难。

## **● JS中继承实现的几种方式，**

### **参考回答：**

1、原型链继承，将父类的实例作为子类的原型，他的特点是实例是子类的实例也是父类的实例，父类新增的原型方法/属性，子类都能够访问，并且原型链继承简单易于实现，缺点是来自原型对象的所有属性被所有实例共享，无法实现多继承，无法向父类构造函数传参。

2、构造继承，使用父类的构造函数来增强子类实例，即复制父类的实例属性给子类，

构造继承可以向父类传递参数，可以实现多继承，通过call多个父类对象。但是构造继承只能继承父类的实例属性和方法，不能继承原型属性和方法，无法实现函数服用，每个子类都有父类实例函数的副本，影响性能

3、实例继承，为父类实例添加新特性，作为子类实例返回，实例继承的特点是不限制调用方法，不管是new 子类（）还是子类（）返回的对象具有相同的效果，缺点是实例是父类的实例，不是子类的实例，不支持多继承

4、拷贝继承：特点：支持多继承，缺点：效率较低，内存占用高（因为要拷贝父类的属性）无法获取父类不可枚举的方法（不可枚举方法，不能使用for in 访问到）

5、组合继承：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

6、寄生组合继承：通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，避免的组合继承的缺点

## **● 写一个函数，第一秒打印1，第二秒打印2**

### **参考回答：**

两个方法，第一个是用let块级作用域

for(let i=0;i<5;i++){

setTimeout(function(){

console.log(i)

},1000\*i)

}

第二个方法闭包

[复制代码](https://www.nowcoder.com/tutorial/96/4b00158bec04471ea22e7713e0e4301b)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for(var i=0;i<5;i++){  (function(i){  setTimeout(function(){  console.log(i)  },1000\*i)  })(i)  } |

## **vue的生命周期**

### **参考回答：**

Vue实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模板、挂载Dom、渲染→更新→渲染、销毁等一系列过程，我们称这是Vue的生命周期。通俗说就是Vue实例从创建到销毁的过程，就是生命周期。

每一个组件或者实例都会经历一个完整的生命周期，总共分为三个阶段：初始化、运行中、销毁。

实例、组件通过new Vue() 创建出来之后会初始化事件和生命周期，然后就会执行beforeCreate钩子函数，这个时候，数据还没有挂载呢，只是一个空壳，无法访问到数据和真实的dom，一般不做操作

挂载数据，绑定事件等等，然后执行created函数，这个时候已经可以使用到数据，也可以更改数据,在这里更改数据不会触发updated函数，在这里可以在渲染前倒数第二次更改数据的机会，不会触发其他的钩子函数，一般可以在这里做初始数据的获取

接下来开始找实例或者组件对应的模板，编译模板为虚拟dom放入到render函数中准备渲染，然后执行beforeMount钩子函数，在这个函数中虚拟dom已经创建完成，马上就要渲染,在这里也可以更改数据，不会触发updated，在这里可以在渲染前最后一次更改数据的机会，不会触发其他的钩子函数，一般可以在这里做初始数据的获取

接下来开始render，渲染出真实dom，然后执行mounted钩子函数，此时，组件已经出现在页面中，数据、真实dom都已经处理好了,事件都已经挂载好了，可以在这里操作真实dom等事情...

当组件或实例的数据更改之后，会立即执行beforeUpdate，然后vue的虚拟dom机制会重新构建虚拟dom与上一次的虚拟dom树利用diff算法进行对比之后重新渲染，一般不做什么事儿

当更新完成后，执行updated，数据已经更改完成，dom也重新render完成，可以操作更新后的虚拟dom

当经过某种途径调用$destroy方法后，立即执行beforeDestroy，一般在这里做一些善后工作，例如清除计时器、清除非指令绑定的事件等等

组件的数据绑定、监听...去掉后只剩下dom空壳，这个时候，执行destroyed，在这里做善后工作也可以

## **● 简单介绍一下symbol**

### **参考回答：**

Symbol是ES6 的新增属性，代表用给定名称作为唯一标识，这种类型的值可以这样创建，let id=symbol(“id”)

Symbl确保唯一，即使采用相同的名称，也会产生不同的值，我们创建一个字段，仅为知道对应symbol的人能访问，使用symbol很有用，symbol并不是100%隐藏，有内置方法Object.getOwnPropertySymbols(obj)可以获得所有的symbol。   
也有一个方法Reflect.ownKeys(obj)返回对象所有的键，包括symbol。

所以并不是真正隐藏。但大多数库内置方法和语法结构遵循通用约定他们是隐藏的，

## **● 什么是事件监听**

### **参考回答：**

addEventListener()方法，用于向指定元素添加事件句柄，它可以更简单的控制事件，语法为

element.addEventListener(event, function, useCapture);

第一个参数是事件的类型(如 "click" 或 "mousedown").

第二个参数是事件触发后调用的函数。

第三个参数是个布尔值用于描述事件是冒泡还是捕获。该参数是可选的。

事件传递有两种方式，冒泡和捕获

事件传递定义了元素事件触发的顺序，如果你将P元素插入到div元素中，用户点击P元素，

在冒泡中，内部元素先被触发，然后再触发外部元素，

捕获中，外部元素先被触发，在触发内部元素，

## **● 介绍一下promise，及其底层如何实现**

### **参考回答：**

Promise是一个对象，保存着未来将要结束的事件，她有两个特征:

1、对象的状态不受外部影响，Promise对象代表一个异步操作，有三种状态，pending进行中，fulfilled已成功，rejected已失败，只有异步操作的结果，才可以决定当前是哪一种状态，任何其他操作都无法改变这个状态，这也就是promise名字的由来

2、一旦状态改变，就不会再变，promise对象状态改变只有两种可能，从pending改到fulfilled或者从pending改到rejected，只要这两种情况发生，状态就凝固了，不会再改变，这个时候就称为定型resolved,

Promise的基本用法，

let promise1 = new Promise(function(resolve,reject){

setTimeout(function(){

resolve('ok')

},1000)

})

promise1.then(function success(val){

console.log(val)

})

最简单代码实现promise

class PromiseM {

constructor (process) {

this.status = 'pending'

this.msg = ''

process(this.resolve.bind(this), this.reject.bind(this))

return this

}

resolve (val) {

this.status = 'fulfilled'

this.msg = val

}

reject (err) {

this.status = 'rejected'

this.msg = err

}

then (fufilled, reject) {

if(this.status === 'fulfilled') {

fufilled(this.msg)

}

if(this.status === 'rejected') {

reject(this.msg)

}

}

}

//测试代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var mm=new PromiseM(function(resolve,reject){  resolve('123');  });  mm.then(function(success){  console.log(success);  },function(){  console.log('fail!');  }); |

## **● 说说C++,Java，JavaScript这三种语言的区别**

### **参考回答：**

从静态类型还是动态类型来看

静态类型，编译的时候就能够知道每个变量的类型，编程的时候也需要给定类型，如Java中的整型int，浮点型float等。C、C++、Java都属于静态类型语言。

动态类型，运行的时候才知道每个变量的类型，编程的时候无需显示指定类型，如JavaScript中的var、PHP中的$。JavaScript、Ruby、Python都属于动态类型语言。

静态类型还是动态类型对语言的性能有很大影响。

对于静态类型，在编译后会大量利用已知类型的优势，如int类型，占用4个字节，编译后的代码就可以用内存地址加偏移量的方法存取变量，而地址加偏移量的算法汇编很容易实现。

对于动态类型，会当做字符串通通存下来，之后存取就用字符串匹配。

从编译型还是解释型来看

编译型语言，像C、C++，需要编译器编译成本地可执行程序后才能运行，由开发人员在编写完成后手动实施。用户只使用这些编译好的本地代码，这些本地代码由系统加载器执行，由操作系统的CPU直接执行，无需其他额外的虚拟机等。

源代码=》抽象语法树=》中间表示=》本地代码

解释性语言，像JavaScript、Python，开发语言写好后直接将代码交给用户，用户使用脚本解释器将脚本文件解释执行。对于脚本语言，没有开发人员的编译过程，当然，也不绝对。

源代码=》抽象语法树=》解释器解释执行。

对于JavaScript，随着Java虚拟机JIT技术的引入，工作方式也发生了改变。可以将抽象语法树转成中间表示（字节码），再转成本地代码，如JavaScriptCore，这样可以大大提高执行效率。也可以从抽象语法树直接转成本地代码，如V8

Java语言，分为两个阶段。首先像C++语言一样，经过编译器编译。和C++的不同，C++编译生成本地代码，Java编译后，生成字节码，字节码与平台无关。第二阶段，由Java的运行环境也就是Java虚拟机运行字节码，使用解释器执行这些代码。一般情况下，Java虚拟机都引入了JIT技术，将字节码转换成本地代码来提高执行效率。

注意，在上述情况中，编译器的编译过程没有时间要求，所以编译器可以做大量的代码优化措施。

对于JavaScript与Java它们还有的不同：

对于Java，Java语言将源代码编译成字节码，这个同执行阶段是分开的。也就是从源代码到抽象语法树到字节码这段时间的长短是无所谓的。

对于JavaScript，这些都是在网页和JavaScript文件下载后同执行阶段一起在网页的加载和渲染过程中实施的，所以对于它们的处理时间有严格要求。

## **● js原型链，原型链的顶端是什么？Object的原型是什么？Object的原型的原型是什么？在数组原型链上实现删除数组重复数据的方法**

### **参考回答：**

能够把这个讲清楚弄明白是一件很困难的事，

首先明白原型是什么，在ES6之前，JS没有类和继承的概念，JS是通过原型来实现继承的，在JS中一个构造函数默认带有一个prototype属性，这个的属性值是一个对象，同时这个prototype对象自带有一个constructor属性，这个属性指向这个构造函数，同时每一个实例都会有一个\_proto\_属性指向这个prototype对象，我们可以把这个叫做隐式原型，我们在使用一个实例的方法的时候，会先检查这个实例中是否有这个方法，没有的话就会检查这个prototype对象是否有这个方法，

基于这个规则，如果让原型对象指向另一个类型的实例，即constructor1.protoytpe=instance2，这时候如果试图引用constructor1构造的实例instance1的某个属性p1,

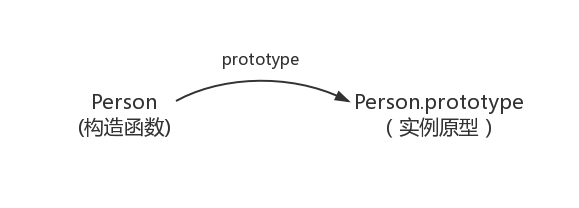
首先会在instance1内部属性中找一遍，

接着会在instance1.\_proto\_（constructor1.prototype）即是instance2中寻找p1

搜寻轨迹：instance1->instance2->constructor2.prototype……->Object.prototype;这即是原型链，原型链顶端是Object.prototype

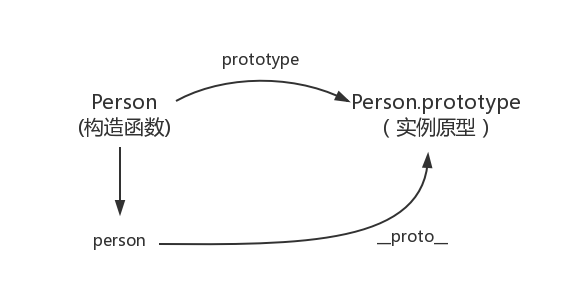
补充学习：

每个函数都有一个prototype属性，这个属性指向了一个对象，这个对象正是调用该函数而创建的实例的原型，那么什么是原型呢，可以这样理解，每一个JavaScript对象在创建的时候就会预制管理另一个对象，这个对象就是我们所说的原型，每一个对象都会从原型继承属性，如图：



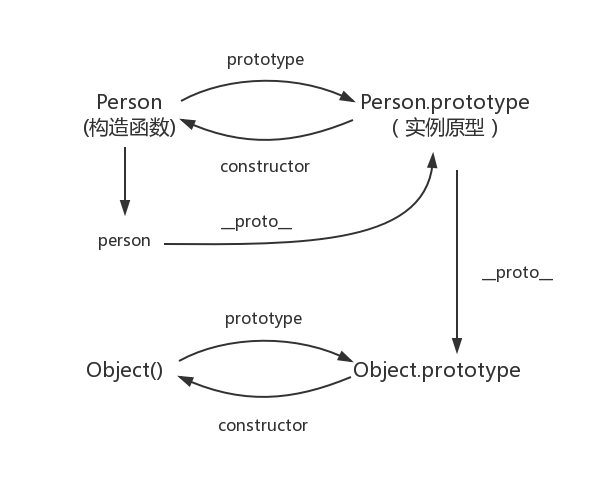
那么怎么表示实例与实例原型的关系呢，这时候就要用到第二个属性\_proto\_

这是每一个JS对象都会有的一个属性，指向这个对象的原型，如图：



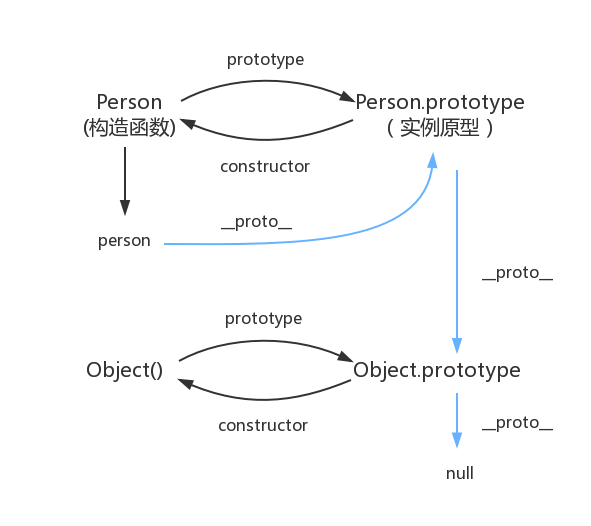
既然实例对象和构造函数都可以指向原型，那么原型是否有属性指向构造函数或者实例呢，指向实例是没有的，因为一个构造函数可以生成多个实例，但是原型有属性可以直接指向构造函数，通过constructor即可

接下来讲解实例和原型的关系：

当读取实例的属性时，如果找不到，就会查找与对象相关的原型中的属性，如果还查不到，就去找原型的原型，一直找到最顶层，那么原型的原型是什么呢，首先，原型也是一个对象，既然是对象，我们就可以通过构造函数的方式创建它，所以原型对象就是通过Object构造函数生成的，如图：  


那么Object.prototype的原型呢，我们可以打印console.log(Object.prototype.\_\_proto\_\_ === null)，返回true

null表示没有对象，即该处不应有值，所以Object.prototype没有原型，如图：



图中这条蓝色的线即是原型链，

最后补充三点：

constructor：

function Person(){

}

var person = new Person();

console.log(Person === person.constructor);

原本person中没有constructor属性，当不能读取到constructor属性时，会从person的原型中读取，所以指向构造函数Person

\_\_proto\_\_：

绝大部分浏览器支持这个非标准的方法访问原型，然而它并不存在与Person.prototype中，实际上它来自Object.prototype，当使用obj.\_\_proto\_\_时，可以理解为返回来Object.getPrototype(obj)

继承：

前面说到，每个对象都会从原型继承属性，但是引用《你不知道的JS》中的话，继承意味着复制操作，然而JS默认不会复制对象的属性，相反，JS只是在两个对象之间创建一个关联，这样子一个对象就可以通过委托访问另一个对象的属性和函数，所以与其叫继承，叫委托更合适，

## **● 什么是js的闭包？有什么作用，用闭包写个单例模式**

### **参考回答：**

MDN对闭包的定义是：闭包是指那些能够访问自由变量的函数，自由变量是指在函数中使用的，但既不是函数参数又不是函数的局部变量的变量，由此可以看出，闭包=函数+函数能够访问的自由变量，所以从技术的角度讲，所有JS函数都是闭包，但是这是理论上的闭包，还有一个实践角度上的闭包，从实践角度上来说，只有满足1、即使创建它的上下文已经销毁，它仍然存在，2、在代码中引入了自由变量，才称为闭包

闭包的应用：

模仿块级作用域。2、保存外部函数的变量。3、封装私有变量

单例模式：

var Singleton = (function(){

var instance;

var CreateSingleton = function (name) {

this.name = name;

if(instance) {

return instance;

}

// 打印实例名字

this.getName();

// instance = this;

// return instance;

return instance = this;

}

// 获取实例的名字

CreateSingleton.prototype.getName = function() {

console.log(this.name)

}

return CreateSingleton;

})();

// 创建实例对象1

|  |  |
| --- | --- |
| 1 | var a = new Singleton('a'); |

// 创建实例对象2

|  |  |
| --- | --- |
| 1  2 | var b = **new** Singleton('b');  console.log(a===b); |

## **● promise+Generator+Async的使用**

### **参考回答：**

Promise

解决的问题:回调地狱

Promise规范:

promise有三种状态，等待（pending）、已完成（fulfilled/resolved）、已拒绝（rejected）.Promise的状态只能从“等待”转到“完成”或者“拒绝”，不能逆向转换，同时“完成”和“拒绝”也不能相互转换.

promise 必须提供一个 then方法以访问其当前值、终值和据因。promise.then(resolve, reject),resolve 和 reject都是可选参数。如果 resolve 或reject 不是函数，其必须被忽略.

then 方法必须返回一个 promise 对象.

使用:

实例化promise对象需要传入函数(包含两个参数),resolve和reject,内部确定状态.resolve和reject函数可以传入参数在回调函数中使用.  
resolve和reject都是函数,传入的参数在then的回调函数中接收.

|  |  |
| --- | --- |
| 1  2 | var promise = **new** Promise(function(resolve, reject) {  setTimeout(function(){ |

resolve('好哈哈哈哈');

});

});

promise.then(function(val){

console.log(val)

})

then接收两个函数,分别对应resolve和reject状态的回调,函数中接收实例化时传入的参数.

promise.then(val=>{

//resolved

},reason=>{

//rejected

})

catch相当于.then(null, rejection)  
当then中没有传入rejection时,错误会冒泡进入catch函数中,若传入了rejection,则错误会被rejection捕获,而且不会进入catch.此外,then中的回调函数中发生的错误只会在下一级的then中被捕获,不会影响该promise的状态.

|  |  |
| --- | --- |
| 1 | **new** Promise((resolve,reject)=>{ |

throw new Error('错误')

|  |  |
| --- | --- |
| 1 | }).then(null,(err)=>{ |

console.log(err,1);//此处捕获

}).catch((err)=>{

console.log(err,2);

});

// 对比

|  |  |
| --- | --- |
| 1 | new Promise((resolve,reject)=>{ |

throw new Error('错误')

|  |  |
| --- | --- |
| 1 | }).then(null,null).catch((err)=>{ |

console.log(err,2);//此处捕获

});

// 错误示例

new Promise((resolve,reject)=>{

resolve('正常');

}).then((val)=>{

throw new Error('回调函数中错误')

},(err)=>{

console.log(err,1);

}).then(null,(err)=>{

console.log(err,2);//此处捕获,也可用catch

});

两者不等价的情况:  
此时，catch捕获的并不是p1的错误，而是p2的错误，

|  |  |
| --- | --- |
| 1 | p1().then(res=>{ |

return p2()//p2返回一个promise对象

|  |  |
| --- | --- |
| 1 | }).catch(err=> console.log(err)) |

一个错误捕获的错误用例:  
该函数调用中即使发生了错误依然会进入then中的resolve的回调函数,因为函数p1中实例化promise对象时已经调用了catch,若发生错误会进入catch中,此时会返回一个新的promise,因此即使发生错误依然会进入p1函数的then链中的resolve回调函数.

function p1(val){

return new Promise((resolve,reject)=>{

if(val){

var len = val.length;//传入null会发生错误,进入catch捕获错误

resolve(len);

}else{

reject();

}

}).catch((err)=>{

console.log(err)

})

};

p1(null).then((len)=>{

console.log(len,'resolved');

},()=>{

console.log('rejected');

}).catch((err)=>{

console.log(err,'catch');

})

Promise回调链:

promise能够在回调函数里面使用 return 和 throw， 所以在then中可以return出一个promise对象或其他值，也可以throw出一个错误对象，但如果没有return，将默认返回 undefined，那么后面的then中的回调参数接收到的将是undefined.

function p1(val){

return new Promise((resolve,reject)=>{

val==1?resolve(1):reject()

})

};

function p2(val){

return new Promise((resolve,reject)=>{

val==2?resolve(2):reject();

})

};

let promimse = new Promise(function(resolve,reject){

resolve(1)

})

.then(function(data1) {

return p1(data1)//如果去掉return,则返回undefined而不是p1的返回值,会导致报错

})

.then(function(data2){

return p2(data2+1)

})

.then(res=>console.log(res))

Generator函数：

generator函数使用:  
1、分段执行，可以暂停

2、可以控制阶段和每个阶段的返回值  
3、可以知道是否执行到结尾

function\* g() {

var o = 1;

yield o++;

yield o++;

}

var gen = g();

console.log(gen.next()); //  Object {value: 1, done: false}

var xxx = g();

console.log(gen.next()); // Object {value: 2, done: false}

console.log(xxx.next()); // Object {value: 1, done: false}

console.log(gen.next()); // Object {value: undefined, done: true}

generator和异步控制:  
利用Generator函数的暂停执行的效果，可以把异步操作写在yield语句里面，等到调用next方法时再往后执行。这实际上等同于不需要写回调函数了，因为异步操作的后续操作可以放在yield语句下面，反正要等到调用next方法时再执行。所以，Generator函数的一个重要实际意义就是用来处理异步操作，改写回调函数。

async和异步:

用法:

async 表示这是一个async函数，await只能用在这个函数里面。

await 表示在这里等待异步操作返回结果，再继续执行。

await 后一般是一个promise对象

示例:async用于定义一个异步函数，该函数返回一个Promise。  
如果async函数返回的是一个同步的值，这个值将被包装成一个理解resolve的Promise，等同于return Promise.resolve(value)。

await用于一个异步操作之前，表示要“等待”这个异步操作的返回值。await也可以用于一个同步的值。

let timer = async function timer(){

return new Promise((resolve,reject) => {

setTimeout(() => {

resolve('500');

},500);

});

}

timer().then(result => {

console.log(result);  //500

}).catch(err => {

console.log(err.message);

});

//返回一个同步的值

|  |  |
| --- | --- |
| 1  2 | let sayHi = async function sayHi(){  let hi = await 'hello world'; |

return hi;  //等同于return Promise.resolve(hi);

|  |  |
| --- | --- |
| 1  2  3  4 | }  sayHi().then(result => {  console.log(result);  }); |

## **● 事件委托以及冒泡原理。**

### **参考回答：**

事件委托是利用冒泡阶段的运行机制来实现的，就是把一个元素响应事件的函数委托到另一个元素，一般是把一组元素的事件委托到他的父元素上，委托的优点是

减少内存消耗，节约效率

动态绑定事件

事件冒泡，就是元素自身的事件被触发后，如果父元素有相同的事件，如onclick事件，那么元素本身的触发状态就会传递，也就是冒到父元素，父元素的相同事件也会一级一级根据嵌套关系向外触发，直到document/window，冒泡过程结束。

## **● 写个函数，可以转化下划线命名到驼峰命名**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static String UnderlineToHump(String para){  StringBuilder result=new StringBuilder();  String a[]=para.split("\_");  for(String s:a){  if(result.length()==0){  result.append(s.toLowerCase());  }else{  result.append(s.substring(0, 1).toUpperCase());  result.append(s.substring(1).toLowerCase());  }  }  return result.toString();  }  } |

## **● 深浅拷贝的区别和实现**

### **参考回答：**

数组的浅拷贝：

如果是数组，我们可以利用数组的一些方法，比如slice，concat方法返回一个新数组的特性来实现拷贝，但假如数组嵌套了对象或者数组的话，使用concat方法克隆并不完整，如果数组元素是基本类型，就会拷贝一份，互不影响，而如果是对象或数组，就会只拷贝对象和数组的引用，这样我们无论在新旧数组进行了修改，两者都会发生变化，我们把这种复制引用的拷贝方法称为浅拷贝，

深拷贝就是指完全的拷贝一个对象，即使嵌套了对象，两者也互相分离，修改一个对象的属性，不会影响另一个

如何深拷贝一个数组

1、这里介绍一个技巧，不仅适用于数组还适用于对象！那就是：

var arr = ['old', 1, true, ['old1', 'old2'], {old: 1}]

var new\_arr = JSON.parse( JSON.stringify(arr) );

console.log(new\_arr);

原理是JOSN对象中的stringify可以把一个js对象序列化为一个JSON字符串，parse可以把JSON字符串反序列化为一个js对象，通过这两个方法，也可以实现对象的深复制。

但是这个方法不能够拷贝函数

浅拷贝的实现：

以上三个方法concat,slice ,JSON.stringify都是技巧类，根据实际项目情况选择使用，我们可以思考下如何实现一个对象或数组的浅拷贝，遍历对象，然后把属性和属性值都放在一个新的对象里即可

|  |  |
| --- | --- |
| 1 | var shallowCopy = function(obj) { |

// 只拷贝对象

|  |  |
| --- | --- |
| 1 | if (typeof obj !== 'object') return; |

// 根据obj的类型判断是新建一个数组还是对象

|  |  |
| --- | --- |
| 1 | var newObj = obj instanceof Array ? [] : {}; |

// 遍历obj，并且判断是obj的属性才拷贝

for (var key in obj) {

if (obj.hasOwnProperty(key)) {

newObj[key] = obj[key];

}

}

return newObj;

}

深拷贝的实现

那如何实现一个深拷贝呢？说起来也好简单，我们在拷贝的时候判断一下属性值的类型，如果是对象，我们递归调用深拷贝函数不就好了~

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var deepCopy = function(obj) {  if (typeof obj !== 'object') return;  var newObj = obj instanceof Array ? [] : {};  for (var key in obj) {  if (obj.hasOwnProperty(key)) {  newObj[key] = typeof obj[key] === 'object' ? deepCopy(obj[key]) : obj[key];  }  }  return newObj;  } |

## **● JS中string的startwith和indexof两种方法的区别**

### **参考回答：**

JS中startwith函数，其参数有3个，stringObj,要搜索的字符串对象，str，搜索的字符串，position，可选，从哪个位置开始搜索，如果以position开始的字符串以搜索字符串开头，则返回true，否则返回false

Indexof函数，indexof函数可返回某个指定字符串在字符串中首次出现的位置，

## **● js字符串转数字的方法**

### **参考回答：**

通过函数parseInt（），可解析一个字符串，并返回一个整数，语法为parseInt（string ,radix）

string：被解析的字符串

radix：表示要解析的数字的基数，默认是十进制，如果radix<2或>36,则返回NaN

## **● let const var的区别 ，什么是块级作用域，如何用ES5的方法实现块级作用域（立即执行函数），ES6 呢**

### **参考回答：**

提起这三个最明显的区别是var声明的变量是全局或者整个函数块的，而let,const声明的变量是块级的变量，var声明的变量存在变量提升，let,const不存在，let声明的变量允许重新赋值，const不允许，

## **● ES6箭头函数的特性**

### **参考回答：**

ES6 增加了箭头函数，基本语法为

let func = value => value;

相当于

let func = function (value) {

return value;

};

箭头函数与普通函数的区别在于：

1、箭头函数没有this，所以需要通过查找作用域链来确定this的值，这就意味着如果箭头函数被非箭头函数包含，this绑定的就是最近一层非箭头函数的this，

2、箭头函数没有自己的arguments对象，但是可以访问外围函数的arguments对象

3、不能通过new关键字调用，同样也没有new.target值和原型

## **● setTimeout和Promise的执行顺序**

### **参考回答：**

首先我们来看这样一道题：

setTimeout(function() {

console.log(1)

}, 0);

new Promise(function(resolve, reject) {

console.log(2)

for (var i = 0; i < 10000; i++) {

if(i === 10) {console.log(10)}

i == 9999 && resolve();

}

console.log(3)

}).then(function() {

console.log(4)

})

console.log(5);

输出答案为2 10 3 5 4 1

要先弄清楚settimeout（fun,0）何时执行，promise何时执行，then何时执行

settimeout这种异步操作的回调，只有主线程中没有执行任何同步代码的前提下，才会执行异步回调，而settimeout（fun,0）表示立刻执行，也就是用来改变任务的执行顺序，要求浏览器尽可能快的进行回调

promise何时执行，由上图可知promise新建后立即执行，所以promise构造函数里代码同步执行的，

then方法指向的回调将在当前脚本所有同步任务执行完成后执行，

那么then为什么比settimeout执行的早呢，因为settimeout（fun,0）不是真的立即执行，

经过测试得出结论：执行顺序为：同步执行的代码-》promise.then->settimeout

## **● 有了解过事件模型吗，DOM0级和DOM2级有什么区别，DOM的分级是什么**

### **参考回答：**

JSDOM事件流存在如下三个阶段：

事件捕获阶段

处于目标阶段

事件冒泡阶段

JSDOM标准事件流的触发的先后顺序为：先捕获再冒泡，点击DOM节点时，事件传播顺序：事件捕获阶段，从上往下传播，然后到达事件目标节点，最后是冒泡阶段，从下往上传播

DOM节点添加事件监听方法addEventListener，中参数capture可以指定该监听是添加在事件捕获阶段还是事件冒泡阶段，为false是事件冒泡，为true是事件捕获，并非所有的事件都支持冒泡，比如focus，blur等等，我们可以通过event.bubbles来判断

事件模型有三个常用方法：

event.stopPropagation:阻止捕获和冒泡阶段中，当前事件的进一步传播，

event.stopImmediatePropagetion，阻止调用相同事件的其他侦听器，

event.preventDefault，取消该事件（假如事件是可取消的）而不停止事件的进一步传播，

event.target：指向触发事件的元素，在事件冒泡过程中这个值不变

event.currentTarget = this，时间帮顶的当前元素，只有被点击时目标元素的target才会等于currentTarget，

最后，对于执行顺序的问题，如果DOM节点同时绑定了两个事件监听函数，一个用于捕获，一个用于冒泡，那么两个事件的执行顺序真的是先捕获在冒泡吗，答案是否定的，绑定在被点击元素的事件是按照代码添加顺序执行的，其他函数是先捕获再冒泡

## **● 平时是怎么调试JS的**

### **参考回答：**

一般用Chrome自带的控制台

## **● JS的基本数据类型有哪些，基本数据类型和引用数据类型的区别，NaN是什么的缩写，JS的作用域类型，undefined==null返回的结果是什么，undefined与null的区别在哪，写一个函数判断变量类型**

### **参考回答：**

JS的基本数据类型有字符串，数字，布尔，数组，对象，Null，Undefined,基本数据类型是按值访问的，也就是说我们可以操作保存在变量中的实际的值，

基本数据类型和引用数据类型的区别如下：

基本数据类型的值是不可变的，任何方法都无法改变一个基本类型的值，当这个变量重新赋值后看起来变量的值是改变了，但是这里变量名只是指向变量的一个指针，所以改变的是指针的指向改变，该变量是不变的，但是引用类型可以改变

基本数据类型不可以添加属性和方法，但是引用类型可以

基本数据类型的赋值是简单赋值，如果从一个变量向另一个变量赋值基本类型的值，会在变量对象上创建一个新值，然后把该值复制到为新变量分配的位置上，引用数据类型的赋值是对象引用，

基本数据类型的比较是值的比较，引用类型的比较是引用的比较，比较对象的内存地址是否相同

基本数据类型是存放在栈区的，引用数据类型同事保存在栈区和堆区

NaN是JS中的特殊值，表示非数字，NaN不是数字，但是他的数据类型是数字，它不等于任何值，包括自身，在布尔运算时被当做false，NaN与任何数运算得到的结果都是NaN，党员算失败或者运算无法返回正确的数值的就会返回NaN，一些数学函数的运算结果也会出现NaN ,

JS的作用域类型：

一般认为的作用域是词法作用域，此外JS还提供了一些动态改变作用域的方法，常见的作用域类型有：

函数作用域，如果在函数内部我们给未定义的一个变量赋值，这个变量会转变成为一个全局变量，

块作用域：块作用域吧标识符限制在{}中，

改变函数作用域的方法：

eval（），这个方法接受一个字符串作为参数，并将其中的内容视为好像在书写时就存在于程序中这个位置的代码，

with关键字：通常被当做重复引用同一个对象的多个属性的快捷方式

undefined与null：目前null和undefined基本是同义的，只有一些细微的差别，null表示没有对象，undefined表示缺少值，就是此处应该有一个值但是还没有定义，因此undefined==null返回false

此外了解== 和===的区别：

在做==比较时。不同类型的数据会先转换成一致后在做比较，===中如果类型不一致就直接返回false，一致的才会比较

类型判断函数，使用typeof即可，首先判断是否为null，之后用typeof哦按段，如果是object的话，再用array.isarray判断是否为数组，如果是数字的话用isNaN判断是否是NaN即可  
扩展学习：

JS采用的是词法作用域，也就是静态作用域，所以函数的作用域在函数定义的时候就决定了，

看如下例子：

var value = 1;

function foo() {

console.log(value);

}

function bar() {

var value = 2;

foo();

}

bar();

假设JavaScript采用静态作用域，让我们分析下执行过程：

执行foo 函数，先从 foo 函数内部查找是否有局部变量 value，如果没有，就根据书写的位置，查找上面一层的代码，也就是 value 等于 1，所以结果会打印 1。

假设JavaScript采用动态作用域，让我们分析下执行过程：

执行foo 函数，依然是从 foo 函数内部查找是否有局部变量 value。如果没有，就从调用函数的作用域，也就是 bar 函数内部查找 value 变量，所以结果会打印 2。

前面我们已经说了，JavaScript采用的是静态作用域，所以这个例子的结果是 1。

## **● setTimeout(fn,100);100毫秒是如何权衡的**

### **参考回答：**

setTimeout()函数只是将事件插入了任务列表，必须等到当前代码执行完，主线程才会去执行它指定的回调函数，有可能要等很久，所以没有办法保证回调函数一定会在setTimeout指定的时间内执行，100毫秒是插入队列的时间+等待的时间

## **● JS的垃圾回收机制**

### **参考回答：**

GC（garbage collection），GC执行时，中断代码，停止其他操作，遍历所有对象，对于不可访问的对象进行回收，在V8引擎中使用两种优化方法，

分代回收，2、增量GC，目的是通过对象的使用频率，存在时长来区分新生代和老生代对象，多回收新生代区，少回收老生代区，减少每次遍历的时间，从而减少GC的耗时

回收方法：

引用计次，当对象被引用的次数为零时进行回收，但是循环引用时，两个对象都至少被引用了一次，因此导致内存泄漏，

标记清除

## **● 写一个newBind函数，完成bind的功能。**

### **参考回答：**

bind（）方法，创建一个新函数，当这个新函数被调用时，bind（）的第一个参数将作为它运行时的this，之后的一序列参数将会在传递的实参前传入作为它的参数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Function.prototype.bind2 = function (context) {  if (typeof this !== "function") {  throw new Error("Function.prototype.bind - what is trying to be bound is not callable");  }  var self = this;  var args = Array.prototype.slice.call(arguments, 1);  var fNOP = function () {};  var fbound = function () {  self.apply(this instanceof self ? this : context, args.concat(Array.prototype.slice.call(arguments)));  }  fNOP.prototype = this.prototype;  fbound.prototype = new fNOP();  return fbound;  } |

## **● 怎么获得对象上的属性：比如说通过Object.key（）**

### **参考回答：**

从ES5开始，有三种方法可以列出对象的属性

for（let I in obj）该方法依次访问一个对象及其原型链中所有可枚举的类型

object.keys:返回一个数组，包括所有可枚举的属性名称

object.getOwnPropertyNames:返回一个数组包含不可枚举的属性

## **● 简单讲一讲ES6的一些新特性**

### **参考回答：**

ES6在变量的声明和定义方面增加了let、const声明变量，有局部变量的概念，赋值中有比较吸引人的结构赋值，同时ES6对字符串、 数组、正则、对象、函数等拓展了一些方法，如字符串方面的模板字符串、函数方面的默认参数、对象方面属性的简洁表达方式，ES6也 引入了新的数据类型symbol，新的数据结构set和map,symbol可以通过typeof检测出来，为解决异步回调问题，引入了promise和 generator，还有最为吸引人了实现Class和模块，通过Class可以更好的面向对象编程，使用模块加载方便模块化编程，当然考虑到 浏览器兼容性，我们在实际开发中需要使用babel进行编译

重要的特性：

块级作用域：ES5只有全局作用域和函数作用域，块级作用域的好处是不再需要立即执行的函数表达式，循环体中的闭包不再有问题

rest参数：用于获取函数的多余参数，这样就不需要使用arguments对象了，

promise:一种异步编程的解决方案，比传统的解决方案回调函数和事件更合理强大

模块化：其模块功能主要有两个命令构成，export和import，export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能

## **● call和apply是用来做什么？**

### **参考回答：**

Call和apply的作用是一模一样的，只是传参的形式有区别而已

1、改变this的指向

2、借用别的对象的方法，

3、调用函数，因为apply，call方法会使函数立即执行

## **● 了解事件代理吗，这样做有什么好处**

### **参考回答：**

事件代理/事件委托：利用了事件冒泡，只指定一个事件处理程序，就可以管理某一类型的事件，

简而言之：事件代理就是说我们将事件添加到本来要添加的事件的父节点，将事件委托给父节点来触发处理函数，这通常会使用在大量的同级元素需要添加同一类事件的时候，比如一个动态的非常多的列表，需要为每个列表项都添加点击事件，这时就可以使用事件代理，通过判断e.target.nodeName来判断发生的具体元素，这样做的好处是减少事件绑定，同事动态的DOM结构任然可以监听，事件代理发生在冒泡阶段

## **● 如何使不同页面之间进行通信**

## **● 如何写一个继承？**

### **参考回答：**

原型链继承

核心： 将父类的实例作为子类的原型

特点：

非常纯粹的继承关系，实例是子类的实例，也是父类的实例

父类新增原型方法/原型属性，子类都能访问到

简单，易于实现

缺点：

要想为子类新增属性和方法，不能放到构造器中

无法实现多继承

来自原型对象的所有属性被所有实例共享

创建子类实例时，无法向父类构造函数传参

构造继承

核心：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

特点：

解决了子类实例共享父类引用属性的问题

创建子类实例时，可以向父类传递参数

可以实现多继承（call多个父类对象）

缺点：

实例并不是父类的实例，只是子类的实例

只能继承父类的实例属性和方法，不能继承原型属性/方法

无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

实例继承

核心：为父类实例添加新特性，作为子类实例返回

特点：

不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

缺点：

实例是父类的实例，不是子类的实例

不支持多继承

拷贝继承

特点：

支持多继承

缺点：

效率较低，内存占用高（因为要拷贝父类的属性）

组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

特点：

可以继承实例属性/方法，也可以继承原型属性/方法

既是子类的实例，也是父类的实例

不存在引用属性共享问题

可传参

函数可复用

寄生组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

参考[https://www.cnblogs.com/humin/p/4556820.html](https://www.cnblogs.com/humin/p/4556820.html" \t "https://www.nowcoder.com/tutorial/96/_blank)

## **● 给出以下代码，输出的结果是什么？原因？ for(var i=0;i<5;i++) { setTimeout(function(){ console.log(i); },1000); } console.log(i)**

### **参考回答：**

在一秒后输出5个5

每次for循环的时候setTimeout都会执行，但是里面的function则不会执行被放入任务队列，因此放了5次；for循环的5次执行完之后不到1000毫秒；1000毫秒后全部执行任务队列中的函数，所以就是输出5个5。

## **● 给两个构造函数A和B，如何实现A继承B？**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2  3 | function A(...) {}  A.prototype...  function B(...) {}  B.prototype...  A.prototype = Object.create(B.prototype); |

// 再在A的构造函数里new B(props);

|  |  |
| --- | --- |
| 1  2  3  4  5 | for(var i = 0; i < lis.length; i++) {  lis[i].addEventListener('click', function(e) {  alert(i);  }, false)  } |

## **● 问能不能正常打印索引**

### **参考回答：**

在click的时候，已经变成length了

## **● 如果已经有三个promise，A、B和C，想串行执行，该怎么写？**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // promise  A.then(B).then(C).catch(...)  // async/await  (async ()=>{  await a();  await b();  await c();  })() |

## **● 知道private和public吗**

### **参考回答：**

public：public表明该数据成员、成员函数是对所有用户开放的，所有用户都可以直接进行调用

private：private表示私有，私有的意思就是除了class自己之外，任何人都不可以直接使用

## **● 基础的js**

### **参考回答：**

Function.prototype.a = 1;

Object.prototype.b = 2;

function A() {}

var a = new A();

console.log(a.a, a.b); // undefined, 2

console.log(A.a, A.b); // 1, 2

## **● async和await具体该怎么用？**

### **参考回答：**

|  |  |
| --- | --- |
| 1  2  3 | (async () = > {  await new promise();  })() |

## **知道哪些ES6，ES7的语法**

### **参考回答：**

promise，await/async，let、const、块级作用域、箭头函数

## **● promise和await/async的关系**

### **参考回答：**

都是异步编程的解决方案

## **● 问了一段js代码，输出结果是什么**

## **● js的数据类型**

### **参考回答：**

字符串，数字，布尔，数组，null，Undefined，symbol，对象。

## **● js加载过程阻塞，解决方法。**

### **参考回答：**

指定script标签的async属性。

如果async="async"，脚本相对于页面的其余部分异步地执行（当页面继续进行解析时，脚本将被执行）

如果不使用async 且 defer="defer"：脚本将在页面完成解析时执行

## **● js对象类型，基本对象类型以及引用对象类型的区别**

### **参考回答：**

分为基本对象类型和引用对象类型

基本数据类型：按值访问，可操作保存在变量中的实际的值。基本类型值指的是简单的数据段。基本数据类型有这六种:undefined、null、string、number、boolean、symbol。

引用类型：当复制保存着对象的某个变量时，操作的是对象的引用，但在为对象添加属性时，操作的是实际的对象。引用类型值指那些可能为多个值构成的对象。

引用类型有这几种：Object、Array、RegExp、Date、Function、特殊的基本包装类型(String、Number、Boolean)以及单体内置对象(Global、Math)。

## **● JavaScript中的轮播实现原理？假如一个页面上有两个轮播，你会怎么实现？**

### **参考回答：**

图片轮播的原理就是图片排成一行，然后准备一个只有一张图片大小的容器，对这个容器设置超出部分隐藏，在控制定时器来让这些图片整体左移或右移，这样呈现出来的效果就是图片在轮播了。

如果有两个轮播，可封装一个轮播组件，供两处调用

## **● 怎么实现一个计算一年中有多少周？**

### **参考回答：**

首先你得知道是不是闰年，也就是一年是365还是366.

其次你得知道当年1月1号是周几。假如是周五，一年365天把1号 2号3号减去，也就是把第一个不到一周的天数减去等于362  
还得知道最后一天是周几，加入是周五，需要把周一到周五减去，也就是362-5=357.正常情况 357这个数计算出来是7的倍数。357/7=51 。即为周数。

## **● 面向对象的继承方式**

### **参考回答：**

原型链继承

核心： 将父类的实例作为子类的原型

特点：

非常纯粹的继承关系，实例是子类的实例，也是父类的实例

父类新增原型方法/原型属性，子类都能访问到

简单，易于实现

缺点：

要想为子类新增属性和方法，不能放到构造器中

无法实现多继承

来自原型对象的所有属性被所有实例共享

创建子类实例时，无法向父类构造函数传参

构造继承

核心：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

特点：

解决了子类实例共享父类引用属性的问题

创建子类实例时，可以向父类传递参数

可以实现多继承（call多个父类对象）

缺点：

实例并不是父类的实例，只是子类的实例

只能继承父类的实例属性和方法，不能继承原型属性/方法

无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

实例继承

核心：为父类实例添加新特性，作为子类实例返回

特点：

不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

缺点：

实例是父类的实例，不是子类的实例

不支持多继承

拷贝继承

特点：

支持多继承

缺点：

效率较低，内存占用高（因为要拷贝父类的属性）

组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

特点：

可以继承实例属性/方法，也可以继承原型属性/方法

既是子类的实例，也是父类的实例

不存在引用属性共享问题

可传参

函数可复用

寄生组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

参考[https://www.cnblogs.com/humin/p/4556820.html](https://www.cnblogs.com/humin/p/4556820.html" \t "https://www.nowcoder.com/tutorial/96/_blank)

## **● JS的数据类型**

### **参考回答：**

字符串，数字，布尔，数组，null，Undefined，symbol，对象。

## **● 引用类型常见的对象**

### **参考回答：**

Object、Array、RegExp、Date、Function、特殊的基本包装类型(String、Number、Boolean)以及单体内置对象(Global、Math)等

## **● es6的常用**

### **参考回答：**

promise，await/async，let、const、块级作用域、箭头函数

## **● class**

### **参考回答：**

ES6提供了更接近传统语言的写法，引入了Class（类）这个概念，作为对象的模板。通过class关键字，可以定义类。

## **● 口述数组去重**

### **参考回答：**

法一：indexOf循环去重

法二：ES6 Set去重；Array.from(new Set(array))

法三：Object 键值对去重；把数组的值存成 Object 的 key 值，比如 Object[value1] = true，在判断另一个值的时候，如果 Object[value2]存在的话，就说明该值是重复的。

## **● 继承**

### **参考回答：**

原型链继承

核心： 将父类的实例作为子类的原型

特点：

非常纯粹的继承关系，实例是子类的实例，也是父类的实例

父类新增原型方法/原型属性，子类都能访问到

简单，易于实现

缺点：

要想为子类新增属性和方法，不能放到构造器中

无法实现多继承

来自原型对象的所有属性被所有实例共享

创建子类实例时，无法向父类构造函数传参

构造继承

核心：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

特点：

解决了子类实例共享父类引用属性的问题

创建子类实例时，可以向父类传递参数

可以实现多继承（call多个父类对象）

缺点：

实例并不是父类的实例，只是子类的实例

只能继承父类的实例属性和方法，不能继承原型属性/方法

无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

实例继承

核心：为父类实例添加新特性，作为子类实例返回

特点：

不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

缺点：

实例是父类的实例，不是子类的实例

不支持多继承

拷贝继承

特点：

支持多继承

缺点：

效率较低，内存占用高（因为要拷贝父类的属性）

组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

特点：

可以继承实例属性/方法，也可以继承原型属性/方法

既是子类的实例，也是父类的实例

不存在引用属性共享问题

可传参

函数可复用

寄生组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

参考[https://www.cnblogs.com/humin/p/4556820.html](https://www.cnblogs.com/humin/p/4556820.html" \t "https://www.nowcoder.com/tutorial/96/_blank)

## **● call和apply的区别**

### **参考回答：**

apply：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.apply(A, arguments);即A对象应用B对象的方法。

call：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.call(A, args1,args2);即A对象调用B对象的方法。

## **● es6的常用特性**

### **参考回答：**

promise，await/async，let、const、块级作用域、箭头函数

## **● 箭头函数和function有什么区别**

### **参考回答：**

箭头函数根本就没有绑定自己的this，在箭头函数中调用 this 时，仅仅是简单的沿着作用域链向上寻找，找到最近的一个 this 拿来使用

## **● new操作符原理**

### **参考回答：**

1. 创建一个类的实例：创建一个空对象obj，然后把这个空对象的\_\_proto\_\_设置为构造函数的prototype。

2. 初始化实例：构造函数被传入参数并调用，关键字this被设定指向该实例obj。

3. 返回实例obj。

## **● bind,apply,call**

### **参考回答：**

apply：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.apply(A, arguments);即A对象应用B对象的方法。

call：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.call(A, args1,args2);即A对象调用B对象的方法。

bind除了返回是函数以外，它的参数和call一样。

## **● bind和apply的区别**

### **参考回答：**

返回不同：bind返回是函数

参数不同：apply(A, arguments)，bind(A, args1,args2)

## **● 数组的去重**

### **参考回答：**

法一：indexOf循环去重

法二：ES6 Set去重；Array.from(new Set(array))

法三：Object 键值对去重；把数组的值存成 Object 的 key 值，比如 Object[value1] = true，在判断另一个值的时候，如果 Object[value2]存在的话，就说明该值是重复的。

## **● 闭包**

### **参考回答：**

（1）什么是闭包：

闭包是指有权访问另外一个函数作用域中的变量的函数。

闭包就是函数的局部变量集合，只是这些局部变量在函数返回后会继续存在。闭包就是就是函数的“堆栈”在函数返回后并不释放，我们也可以理解为这些函数堆栈并不在栈上分配而是在堆上分配。当在一个函数内定义另外一个函数就会产生闭包。

（2）为什么要用：

匿名自执行函数：我们知道所有的变量，如果不加上var关键字，则默认的会添加到全局对象的属性上去，这样的临时变量加入全局对象有很多坏处，比如：别的函数可能误用这些变量；造成全局对象过于庞大，影响访问速度(因为变量的取值是需要从原型链上遍历的)。除了每次使用变量都是用var关键字外，我们在实际情况下经常遇到这样一种情况，即有的函数只需要执行一次，其内部变量无需维护，可以用闭包。

结果缓存：我们开发中会碰到很多情况，设想我们有一个处理过程很耗时的函数对象，每次调用都会花费很长时间，那么我们就需要将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。闭包正是可以做到这一点，因为它不会释放外部的引用，从而函数内部的值可以得以保留。

## **● promise实现**

### **参考回答：**

Promise实现如下

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | function Promise(fn) {  var state = 'pending',  value = null,  callbacks = [];  this.then = function (onFulfilled, onRejected) {  return new Promise(function (resolve, reject) {  handle({  onFulfilled: onFulfilled || null,  onRejected: onRejected || null,  resolve: resolve,  reject: reject  });  });  };  function handle(callback) {  if (state === 'pending') {  callbacks.push(callback);  return;  }  var cb = state === 'fulfilled' ? callback.onFulfilled : callback.onRejected,  ret;  if (cb === null) {  cb = state === 'fulfilled' ? callback.resolve : callback.reject;  cb(value);  return;  }  ret = cb(value);  callback.resolve(ret);  }  function resolve(newValue) {  if (newValue && (typeof newValue === 'object' || typeof newValue === 'function')) {  var then = newValue.then;  if (typeof then === 'function') {  then.call(newValue, resolve, reject);  return;  }  }  state = 'fulfilled';  value = newValue;  execute();  }  function reject(reason) {  state = 'rejected';  value = reason;  execute();  }  function execute() {  setTimeout(function () {  callbacks.forEach(function (callback) {  handle(callback);  });  }, 0);  }  fn(resolve, reject);  } |

## **● assign的深拷贝**

### **参考回答：**

function clone( obj ) {

var copy;

switch( typeof obj ) {

case "undefined":

break;

case "number":

copy = obj - 0;

break;

case "string":

copy = obj + "";

break;

case "boolean":

copy = obj;

break;

case "object":  //object分为两种情况 对象（Object）和数组（Array）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | if(obj === null) {  copy = null;  } else {  if( Object.prototype.toString.call(obj).slice(8, -1) === "Array") {  copy = [];  for( var i = 0 ; i < obj.length ; i++ ) {  copy.push(clone(obj[i]));  }  } else {  copy = {};  for( var j in obj) {  copy[j] = clone(obj[j]);  }  }  }  break;  default:  copy = obj;  break;  }  return copy;  } |

## **● 说promise，没有promise怎么办**

### **参考回答：**

没有promise，可以用回调函数代替

## **● 事件委托**

### **参考回答：**

把一个元素响应事件（click、keydown......）的函数委托到另一个元素；

优点：减少内存消耗、动态绑定事件。

## **●  怎么用原生的js实现jquery的一个特定方法**

## **● 箭头函数和function的区别**

### **参考回答：**

箭头函数根本就没有绑定自己的this，在箭头函数中调用 this 时，仅仅是简单的沿着作用域链向上寻找，找到最近的一个 this 拿来使用

## **● arguments**

### **参考回答：**

arguments是类数组对象，有length属性，不能调用数组方法

可用Array.from()转换

## **● 箭头函数获取arguments**

### **参考回答：**

可用…rest参数获取

## **● Promise**

### **参考回答：**

Promise对象是CommonJS工作组提出的一种规范，目的是为异步编程提供统一接口。每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数。

f1().then(f2);

一个promise可能有三种状态：等待（pending）、已完成（resolved，又称fulfilled）、已拒绝（rejected）。

promise必须实现then方法（可以说，then就是promise的核心），而且then必须返回一个promise，同一个promise的then可以调用多次，并且回调的执行顺序跟它们被定义时的顺序一致。

then方法接受两个参数，第一个参数是成功时的回调，在promise由“等待”态转换到“完成”态时调用，另一个是失败时的回调，在promise由“等待”态转换到“拒绝”态时调用。同时，then可以接受另一个promise传入，也接受一个“类then”的对象或方法，即thenable对象。

## **● 模块化开发（require）**

## **● 事件代理**

### **参考回答：**

事件代理是利用事件的冒泡原理来实现的，何为事件冒泡呢？就是事件从最深的节点开始，然后逐步向上传播事件，举个例子：页面上有这么一个节点树，div>ul>li>a;比如给最里面的a加一个click点击事件，那么这个事件就会一层一层的往外执行，执行顺序a>li>ul>div，有这样一个机制，那么我们给最外面的div加点击事件，那么里面的ul，li，a做点击事件的时候，都会冒泡到最外层的div上，所以都会触发，这就是事件代理，代理它们父级代为执行事件。

## **● Eventloop**

### **参考回答：**

任务队列中，在每一次事件循环中，macrotask只会提取一个执行，而microtask会一直提取，直到microsoft队列为空为止。

也就是说如果某个microtask任务被推入到执行中，那么当主线程任务执行完成后，会循环调用该队列任务中的下一个任务来执行，直到该任务队列到最后一个任务为止。而事件循环每次只会入栈一个macrotask,主线程执行完成该任务后又会检查microtasks队列并完成里面的所有任务后再执行macrotask的任务。

macrotasks: setTimeout, setInterval, setImmediate, I/O, UI rendering  
microtasks: process.nextTick, Promise, MutationObserver

[上一节](https://www.nowcoder.com/tutorial/96/614ed46bbd9c483597ef89b8abfc9084)

## **bootstrap清除浮动的方法**

### **参考回答：**

.clearfix:before,

.clearfix:after {

content: " ";

display: table;

}

.clearfix:after {

clear: both;

}

/\*\*

\* For IE 6/7 only

\*/

.clearfix {

\*zoom: 1;

}

:after伪类在元素末尾插入了一个包含空格的字符，并设置display为table

display:table会创建一个匿名的table-cell，从而触发块级上下文（BFC），因为容器内float的元素也是BFC，由于BFC有不能互相重叠的特性，并且设置了clear: both，:after插入的元素会被挤到容器底部，从而将容器撑高。并且设置display:table后，content中的空格字符会被渲染为0\*0的空白元素，不会占用页面空间。

content包含一个空格，是为了解决Opera浏览器的BUG。当HTML中包含contenteditable属性时，如果content没有包含空格，会造成清除浮动元素的顶部、底部有一个空白（设置font-size：0也可以解决这个问题）。

:after伪类的设置已经达到了清除浮动的目的，但还要设置:before伪类，原因如下

:before的设置也触发了一个BFC，由于BFC有内部布局不受外部影响的特性，因此:before的设置可以阻止margin-top的合并。

这样做，其一是为了和其他清除浮动的方式的效果保持一致；其二，是为了与ie6/7下设置zoom：1后的效果一致（即阻止margin-top合并的效果）。

zoom: 1用于在ie6/7下触发haslayout和contain floats