## 1. 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。

**2.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. 所以对于象Number，String ， Boolean。。都可以先将其包装为对象，在进行克隆

//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：

|  |  |
| --- | --- |
|  | 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\// |

## 3. ● 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。

匿名函数的执行环境具有全局性，this常常指向window

Self是this的副本，匿名执行函数的作用域处于全局变量中，找不到self的副本，如果这个时候需要self的副本，可以向上访问。

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。

**4● 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。

**5● 去除字符串首尾空格**

**参考回答：**

使用正则(^\s\*)|(\s\*$)即可

**6● 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

## 7. ● 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)的语言；

脚本语言、解释性语言

**8● 如何判断一个数组(讲到typeof差点掉坑里)**

**参考回答：**

Typeof的回答有：

1. typeof null =object
2. typeof symbol() //symbol
3. typeof undefined /undefined

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

## 9. ● 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

**10● JS实现跨域**

**参考回答：**

JSONP：通过动态创建jscript，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>

**11● Js基本数据类型**

**参考回答：**

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

**12●  js的命名方式**

**13● js深度拷贝一个元素的具体实现**

**参考回答：**

|  |  |
| --- | --- |
|  | 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;  } |

**14● 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。

**● 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。

**● 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

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”

**● 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();  }  } |

**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。

**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

**● 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。

**● 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>

**● 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>

**● 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。

**●  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

**● get请求传参长度的误区**

**参考回答：**

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。

**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，

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>

**● 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。

注意：Js调用时，加括号和不加括号的区别，

不加括号相当于把函数代码，赋值给等号左边，

加括号是把函数返回值赋值给等号左边

1. 函数声明提升：

函数声明定义的函数会优先解析，而不是顺序解析；因此在整个过程中，首先进行依次解析，由于同名，会进行函数覆盖，

函数声明和函数表达式的区别：

解析器在向执行环境中加载数据时，会率先读取函数声明，并 使其在执行任何代码之前可用

1. 变量提升：

在同一个作用域中，变量声明会被提前，但是赋值不会

在函数里，不用var声明的函数，相当于全局变量，但是如果这个函数传参时，使用了这个变量，相当于这个变量已经被声明了var b

函数体内，bb并没有使用var来定义，按理说这个bb在预处理的时候应该是window的属性。但在这里，函数声明的时候，带了一个参数bb，也就是相当于在函数体内声明了var bb。所以，函数里的bb就是函数活动对象的属性。所以函数执行时会输出2。函数执行完后，函数的活动对象被销毁，也就是局部的这个bb被删除了，执行流进入到window，再输出bb，值就是1了。

**var *bb*** = 1;  
**function** *aa*() {  
 ***bb*** = 2;  
 alert(***bb***);  
};  
*aa*(***bb***);  
alert(***bb***);

// 2 2

**var *bb*** = 1;  
**function** *aa*() {  
 ***bb*** = 2;  
 alert(***bb***);  
};  
*aa*(***bb***);  
alert(***bb***);

// 2 1

Js运行机制：

务，js是单线程的，一个时间上只会执行一件事情，优先处理同步任务，只有同步任务做完了才去看看有没有异步任务，然后顺序执行。

加号优先于三目操作的优先级：

**（三目操作 ）条件表达式的优先级别仅高于赋值运算符,而低于前面遇到过的所有运算符.**

**模块开发的好处是什么：**

模块化程序设计是指在进行程序设计时将一个大程序按照功能划分为若干小程序模块，每个小程序模块完成一个确定的功能，并在这些模块之间建立必要的联系，通过模块的互相协作完成整个功能的程序设计方法。

比如 java 的 import，C# 的 using。我的理解是通过模块化编程，可以将不同的功能独立出来，修改某个功能时不会对其他功能产生影响。

**AMD和CMD的区别**

1. 都是为了达成浏览器端模块开发的目的

区别：

AMD（异步模块定义）。采取异步加载的方式加载模块，模块的加载不会影响它后面的语句执行。

1. AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。  
   CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。
2. 1. 对于依赖的模块，AMD 是**提前执行**，CMD 是**延迟执行**。不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）。CMD 推崇 as lazy as possible.
3. CMD 推崇**依赖就近**，AMD 推崇**依赖前置**。看代码：
4. // CMD  
   define(function(require, exports, module) {  
   var a = require('./a')  
   a.doSomething()  
   // 此处略去 100 行  
   var b = require('./b') // 依赖可以就近书写  
   b.doSomething()  
   // ...   
   })
5. // AMD 默认推荐的是  
   define(['./a', './b'], function(a, b) { // 依赖必须一开始就写好  
   a.doSomething()  
   // 此处略去 100 行  
   b.doSomething()  
   ...  
   })
6. 虽然 AMD 也支持 CMD 的写法，同时还支持将 require 作为依赖项传递，但 RequireJS 的作者默认是最喜欢上面的写法，也是官方文档里默认的模块定义写法。
8. 3. AMD 的 API 默认是**一个当多个用**，CMD 的 API 严格区分，推崇**职责单一**。比如 AMD 里，require 分全局 require 和局部 require，都叫 require。CMD 里，没有全局 require，而是根据模块系统的完备性，提供 seajs.use 来实现模块系统的加载启动。CMD 里，每个 API 都**简单纯粹**。