****1、使用typeof bar ===“object”来确定bar是否是一个对象时有什么潜在的缺陷？这个陷阱如何避免？****

尽管typeof bar ===“object”是检查bar是否是对象的可靠方法，但JavaScript中令人惊讶的问题null也被认为是一个对象！

因此，对于大多数开发人员来说，下面的代码会将true（而不是false）打印到控制台：

var bar = null;console.log(typeof bar === "object"); // logs true!

只要知道这一点，就可以通过检查bar是否为空来轻松避免该问题：

console.log((bar !== null) && (typeof bar === "object")); // logs false

为了让我们的答案更加的完整，还有两件事值得注意：首先，如果bar是一个函数，上面的解决方案将返回false。在大多数情况下，这是所期望的行为，但是在您希望函数返回true的情况下，您可以将上述解决方案修改为：

console.log((bar !== null) && ((typeof bar === "object") || (typeof bar === "function")));

其次，如果bar是数组，则上述解决方案将返回true（例如，如果var bar = [];）。在大多数情况下，这是所希望的行为，因为数组确实是对象，但是在您想要对数组也是false的情况下，可以将上述解决方案修改为：

console.log((bar !== null) && (typeof bar === "object") && (toString.call(bar) !== "[object Array]"));

但是，还有一个替代方法对空值，数组和函数返回false，但对于对象则为true：

console.log((bar !== null) && (bar.constructor === Object));

或者，如果您使用jQuery：

console.log((bar !== null) && (typeof bar === "object") && (! $.isArray(bar)));

ES5使得数组的情况非常简单，包括它自己的空检查：

console.log(Array.isArray(bar));

### ****2、下面的代码将输出到控制台的是什么，为什么？****

(function(){ var a = b = 3;})();

console.log("a defined? " + (typeof a !== 'undefined'));console.log("b defined? " + (typeof b !== 'undefined'));

由于a和b都在函数的封闭范围内定义，并且由于它们所在的行以var关键字开头，因此大多数JavaScript开发人员会希望typeof a和typeof b在上面的示例中都未定义。

但是，情况并非如此。这里的问题是大多数开发人员错误地理解语句var a = b = 3;以下简写为：

var b = 3;var a = b;

但实际上，var a = b = 3;其实是速记：

b = 3;var a = b;

因此（如果您不使用严格模式），代码片段的输出将为：

a defined? falseb defined? true

但是如何在封闭函数的范围之外定义b？那么，因为声明var a = b = 3;是语句b = 3的简写;并且var a = b; b最终成为一个全局变量（因为它不在var关键字后面），因此它仍然在作用域内，即使在封闭函数之外。

注意，在严格模式下（即，使用strict），语句var a = b = 3;会产生一个ReferenceError的运行时错误：b没有定义，从而避免了可能导致的任何头headfakes/bugs。 （这就是为什么你应该在你的代码中使用strict，一个重要的例子！）

### ****3、下面的代码将输出到控制台的是什么？，为什么？****

var myObject = { foo: "bar", func: function() { var self = this; console.log("outer func: this.foo = " + this.foo); console.log("outer func: self.foo = " + self.foo); (function() { console.log("inner func: this.foo = " + this.foo); console.log("inner func: self.foo = " + self.foo); }()); }};myObject.func();

以上代码将输出到控制台：

outer func: this.foo = barouter func: self.foo = barinner func: this.foo = undefinedinner func: self.foo = bar

在外部函数中，this和self都引用myObject，因此都可以正确地引用和访问foo。

但在内部函数中，这不再指向myObject。因此，this.foo在内部函数中是未定义的，而对局部变量self的引用仍然在范围内并且可以在那里访问。

### ****4、在功能块中封装JavaScript源文件的全部内容的重要性和原因是什么？****

这是一种日益普遍的做法，被许多流行的JavaScript库（jQuery，Node.js等）所采用。这种技术在文件的全部内容周围创建一个闭包，这可能最重要的是创建一个私有名称空间，从而有助于避免不同JavaScript模块和库之间的潜在名称冲突。

这种技术的另一个特点是为全局变量提供一个容易引用（可能更短）的别名。例如，这通常用于jQuery插件。 jQuery允许您使用jQuery.noConflict()来禁用对jQuery名称空间的$引用。如果这样做了，你的代码仍然可以使用$使用闭包技术，如下所示：

(function($) { /\* jQuery plugin code referencing $ \*/ } )(jQuery);

### ****5、在JavaScript源文件的开头包含'use strict'的意义和有什么好处？****

这里最简单也是最重要的答案是use strict是****一种在运行时自动执行更严格的JavaScript代码解析和错误处理的方法****。如果代码错误被忽略或失败，将会产生错误或抛出异常。总的来说，这是一个很好的做法。

严格模式的一些主要优点包括：

* ****使调试更容易。**** 如果代码错误本来会被忽略或失败，那么现在将会产生错误或抛出异常，从而更快地发现代码中的问题，并更快地指引它们的源代码。
* ****防止意外全局。**** 如果没有严格模式，将值赋给未声明的变量会自动创建一个具有该名称的全局变量。这是JavaScript中最常见的错误之一。在严格模式下，尝试这样做会引发错误。
* ****消除隐藏威胁。****在没有严格模式的情况下，对null或undefined的这个值的引用会自动强制到全局。这可能会导致许多*headfakes*和*pull-out-your-hair*类型的错误。在严格模式下，引用null或undefined的这个值会引发错误。
* ****不允许重复的参数值。**** 严格模式在检测到函数的重复命名参数（例如，函数foo（val1，val2，val1）{}）时会引发错误，从而捕获代码中几乎可以肯定存在的错误，否则您可能会浪费大量的时间追踪。
  + 注意：它曾经是（在ECMAScript 5中）strict模式将禁止重复的属性名称（例如var object = {foo：“bar”，foo：“baz”};）但是从ECMAScript 2015 开始，就不再有这种情况了。
* ****使eval()更安全。**** eval()在严格模式和非严格模式下的行为方式有些不同。最重要的是，在严格模式下，在eval()语句内部声明的变量和函数不会在包含范围中创建（它们是以非严格模式在包含范围中创建的，这也可能是问题的常见来源）。
* ****抛出无效的使用错误的删除符。**** 删除操作符（用于从对象中删除属性）不能用于对象的不可配置属性。当试图删除一个不可配置的属性时，非严格代码将自动失败，而在这种情况下，严格模式会引发错误。

### ****6、考虑下面的两个函数。他们都会返回同样的值吗？为什么或者为什么不？****

function foo1(){ return { bar: "hello" };}function foo2(){ return { bar: "hello" };}

令人惊讶的是，这两个函数不会返回相同的结果。而是：

console.log("foo1 returns:");console.log(foo1());console.log("foo2 returns:");console.log(foo2());

会产生：

foo1 returns:Object {bar: "hello"}foo2 returns:undefined

这不仅令人惊讶，而且特别令人烦恼的是，foo2()返回未定义而没有引发任何错误。

原因与JavaScript中分号在技术上是可选的事实有关（尽管忽略它们通常是非常糟糕的形式）。因此，在foo2()中遇到包含return语句的行（没有其他内容）时，*会在return语句之后立即自动插入分号。*

由于代码的其余部分是完全有效的，即使它没有被调用或做任何事情（它只是一个未使用的代码块，它定义了一个属性栏，它等于字符串“hello”），所以不会抛出任何错误。

这种行为也被认为是遵循了在JavaScript中将一行开头大括号放在行尾的约定，而不是在新行的开头。如此处所示，这不仅仅是JavaScript中的一种风格偏好。

### ****7、什么是NaN？它的类型是什么？如何可靠地测试一个值是否等于NaN？****

NaN属性表示“不是数字”的值。这个特殊值是由于一个操作数是非数字的（例如“abc”/ 4）或者因为操作的结果是非数字而无法执行的。

虽然这看起来很简单，但NaN有一些令人惊讶的特征，如果人们没有意识到这些特征，就会导致bug。

一方面，虽然NaN的意思是“不是数字”，但它的类型是，数字：

console.log(typeof NaN === "number"); // logs "true"

此外，NaN相比任何事情 - 甚至本身！ - 是false：

console.log(NaN === NaN); // logs "false"

测试数字是否等于NaN的半可靠方法是使用内置函数isNaN()，但即使使用 isNaN()也不是一个好的解决方案。.

一个更好的解决方案要么是使用value!==值，如果该值等于NaN，那么只会生成true。另外，ES6提供了一个新的Number.isNaN()函数 ，它与旧的全局isNaN()函数不同，也更加可靠。

### ****8、下面的代码输出什么？解释你的答案。****

console.log(0.1 + 0.2);console.log(0.1 + 0.2 == 0.3);

对这个问题的一个有教养的回答是：“你不能确定。它可能打印出0.3和true，或者可能不打印。 JavaScript中的数字全部用浮点精度处理，因此可能不会总是产生预期的结果。“

上面提供的示例是演示此问题的经典案例。令人惊讶的是，它会打印出来：

0.30000000000000004false

一个典型的解决方案是比较两个数字与特殊常数Number.EPSILON之间的绝对差值：

function areTheNumbersAlmostEqual(num1, num2) { return Math.abs( num1 - num2 ) < Number.EPSILON;}console.log(areTheNumbersAlmostEqual(0.1 + 0.2, 0.3));

讨论写函数的可能方法isInteger(x)，它确定x是否是一个整数。

这听起来很平凡，事实上，ECMAscript 6为此正好引入了一个新的Number.isInteger()函数，这是微不足道的。但是，在ECMAScript 6之前，这有点复杂，因为没有提供与Number.isInteger()方法等价的方法。

问题在于，在ECMAScript规范中，整数只在概念上存在;即数值始终作为浮点值存储。

考虑到这一点，最简单，最清洁的ECMAScript-6之前的解决方案（即使将非数字值（例如字符串或空值）传递给该函数，该解决方案也具有足够的可靠性以返回false）将成为以下用法按位异或运算符：

function isInteger(x) { return (x ^ 0) === x; }

下面的解决方案也可以工作，尽管不如上面那样高雅

function isInteger(x) { return Math.round(x) === x; }

请注意，在上面的实现中Math.ceil()或Math.floor()可以同样使用（而不是Math.round()）。

或者：

function isInteger(x) { return (typeof x === 'number') && (x % 1 === 0); }

一个相当常见的不正确的解决方案如下：

function isInteger(x) { return parseInt(x, 10) === x; }

虽然这个基于parseInt的方法对许多x值很有效，但一旦x变得相当大，它将无法正常工作。问题是parseInt()在解析数字之前将其第一个参数强制转换为字符串。因此，一旦数字变得足够大，其字符串表示将以指数形式呈现（例如1e + 21）。因此，parseInt()将尝试解析1e + 21，但是当它到达e字符时将停止解析，因此将返回值1.观察：

> String(1000000000000000000000)'1e+21'

> parseInt(1000000000000000000000, 10)1

> parseInt(1000000000000000000000, 10) === 1000000000000000000000false

### ****9、执行下面的代码时，按什么顺序将数字1-4记录到控制台？为什么？****

(function() { console.log(1); setTimeout(function(){console.log(2)}, 1000); setTimeout(function(){console.log(3)}, 0); console.log(4);})();

这些值将按以下顺序记录：

1432

我们先来解释一下这些可能更为明显的部分：

* 首先显示1和4，因为它们是通过简单调用console.log()而没有任何延迟记录的
* 在3之后显示，因为在延迟1000毫秒（即1秒）之后记录2，而在0毫秒的延迟之后记录3。

好的。但是，如果在延迟0毫秒后记录3，这是否意味着它正在被立即记录？而且，如果是这样，不应该在4之前记录它，因为4是由后面的代码行记录的吗？

答案与正确理解JavaScript事件和时间有关。

浏览器有一个事件循环，它检查事件队列并处理未决事件。例如，如果在浏览器繁忙时（例如，处理onclick）在后台发生事件（例如脚本onload事件），则该事件被附加到队列中。当onclick处理程序完成时，将检查队列并处理该事件（例如，执行onload脚本）。

同样，如果浏览器繁忙，setTimeout()也会将其引用函数的执行放入事件队列中。

当值为零作为setTimeout()的第二个参数传递时，它将尝试“尽快”执行指定的函数。具体来说，函数的执行放置在事件队列中，以在下一个计时器滴答时发生。但请注意，这不是直接的;该功能不会执行，直到下一个滴答声。这就是为什么在上面的例子中，调用console.log(4)发生在调用console.log(3)之前（因为调用console.log(3)是通过setTimeout调用的，所以稍微延迟了一点）。

### ****10、编写一个简单的函数（少于160个字符），返回一个布尔值，指示字符串是否是palindrome。****

如果str是回文，以下一行函数将返回true;否则，它返回false。

function isPalindrome(str) { str = str.replace(/W/g, '').toLowerCase(); return (str == str.split('').reverse().join(''));}

例如：

console.log(isPalindrome("level")); // logs 'true'console.log(isPalindrome("levels")); // logs 'false'console.log(isPalindrome("A car, a man, a maraca")); // logs 'true'

### ****11、写一个sum方法，当使用下面的语法调用时它将正常工作。****

console.log(sum(2,3)); // Outputs 5console.log(sum(2)(3)); // Outputs 5

有（至少）两种方法可以做到这一点：

****METHOD 1****

function sum(x) { if (arguments.length == 2) { return arguments[0] + arguments[1]; } else { return function(y) { return x + y; }; }}

在JavaScript中，函数提供对参数对象的访问，该对象提供对传递给函数的实际参数的访问。这使我们能够使用length属性在运行时确定传递给函数的参数的数量

如果传递两个参数，我们只需将它们相加并返回。

否则，我们假设它是以sum(2)(3)的形式被调用的，所以我们返回一个匿名函数，它将传递给sum()（在本例中为2）的参数和传递给匿名函数的参数（这种情况3）。

****METHOD 2****

function sum(x, y) { if (y !== undefined) { return x + y; } else { return function(y) { return x + y; }; }}

当函数被调用时，JavaScript不需要参数的数量来匹配函数定义中参数的数量。如果传递的参数数量超过了函数定义中参数的数量，则超出的参数将被忽略。另一方面，如果传递的参数数量少于函数定义中的参数数量，则在函数内引用时，缺少的参数将具有未定义的值。因此，在上面的例子中，通过简单地检查第二个参数是否未定义，我们可以确定函数被调用的方式并相应地继续。

### ****12、考虑下面的代码片段****

for (var i = 0; i < 5; i++) { var btn = document.createElement('button'); btn.appendChild(document.createTextNode('Button ' + i)); btn.addEventListener('click', function(){ console.log(i); }); document.body.appendChild(btn);}

(a) 当用户点击“按钮4”时，什么被记录到控制台？为什么？

(b) 提供一个或多个可按预期工作的替代实现。

答：

(a) 无论用户点击哪个按钮，数字5将始终记录到控制台。这是因为，在调用onclick方法（对于任何按钮）时，for循环已经完成，并且变量i已经具有值5.（如果受访者知道足够的话就可以获得奖励点数关于执行上下文，变量对象，激活对象和内部“范围”属性如何影响闭包行为。）

(b) 使这项工作的关键是通过将它传递给新创建的函数对象来捕获每次通过for循环的i的值。以下是四种可能的方法来实现这一点：

for (var i = 0; i < 5; i++) { var btn = document.createElement('button'); btn.appendChild(document.createTextNode('Button ' + i)); btn.addEventListener('click', (function(i) { return function() { console.log(i); }; })(i)); document.body.appendChild(btn);}

或者，您可以将新的匿名函数中的整个调用包装为btn.addEventListener：

for (var i = 0; i < 5; i++) { var btn = document.createElement('button'); btn.appendChild(document.createTextNode('Button ' + i)); (function (i) { btn.addEventListener('click', function() { console.log(i); }); })(i); document.body.appendChild(btn);}

或者，我们可以通过调用数组对象的原生forEach方法来替换for循环：

['a', 'b', 'c', 'd', 'e'].forEach(function (value, i) { var btn = document.createElement('button'); btn.appendChild(document.createTextNode('Button ' + i)); btn.addEventListener('click', function() { console.log(i); }); document.body.appendChild(btn);});

最后，最简单的解决方案，如果你在ES6 / ES2015上下文中，就是使用let i而不是var i：

for (let i = 0; i < 5; i++) { var btn = document.createElement('button'); btn.appendChild(document.createTextNode('Button ' + i)); btn.addEventListener('click', function(){ console.log(i); }); document.body.appendChild(btn);}

### ****13、假设d是范围内的“空”对象：****

var d = {};

使用下面的代码完成了什么？

[ 'zebra', 'horse' ].forEach(function(k) { d[k] = undefined;});

上面显示的代码片段在对象d上设置了两个属性。理想情况下，对具有未设置键的JavaScript对象执行的查找评估为未定义。但是运行这段代码会将这些属性标记为对象的“自己的属性”。

这是确保对象具有一组给定属性的有用策略。将该对象传递给Object.keys将返回一个包含这些设置键的数组（即使它们的值未定义）。

### ****14、下面的代码将输出到控制台，为什么？****

var arr1 = "john".split('');var arr2 = arr1.reverse();var arr3 = "jones".split('');arr2.push(arr3);console.log("array 1: length=" + arr1.length + " last=" + arr1.slice(-1));console.log("array 2: length=" + arr2.length + " last=" + arr2.slice(-1));

记录的输出将是：

"array 1: length=5 last=j,o,n,e,s""array 2: length=5 last=j,o,n,e,s"

arr1和arr2是相同的（即['n'，'h'，'o'，'j'，['j'，'o'，'n'，'e'，'s']]）上述代码由于以下原因而被执行：

* 调用数组对象的reverse()方法不仅以相反的顺序返回数组，它还颠倒了数组本身的顺序（即在这种情况下，arr1）。
* reverse()方法返回对数组本身的引用（即，在这种情况下为arr1）。因此，arr2仅仅是对arr1的引用（而不是副本）。因此，当对arr2做任何事情时（即，当我们调用arr2.push(arr3);）时，arr1也会受到影响，因为arr1和arr2只是对同一个对象的引用。

这里有几个观点可以让人们回答这个问题：

* 将数组传递给另一个数组的push()方法会将整个数组作为单个元素推入数组的末尾。结果，声明arr2.push(arr3);将arr3作为一个整体添加到arr2的末尾（即，它不连接两个数组，这就是concat()方法的用途）。
* 像Python一样，JavaScript在调用像slice()这样的数组方法时，会承认负面下标，以此作为在数组末尾引用元素的方式;例如，下标-1表示数组中的最后一个元素，依此类推。

### ****15、下面的代码将输出到控制台，为什么？****

console.log(1 + "2" + "2");console.log(1 + +"2" + "2");console.log(1 + -"1" + "2");console.log(+"1" + "1" + "2");console.log( "A" - "B" + "2");console.log( "A" - "B" + 2);

以上代码将输出到控制台：

"122""32""02""112""NaN2"NaN

这是为什么...

这里的基本问题是JavaScript（ECMAScript）是一种松散类型的语言，它对值执行自动类型转换以适应正在执行的操作。让我们来看看这是如何与上面的每个例子进行比较。

示例1：1 +“2”+“2”输出：“122”说明：第一个操作在1 +“2”中执行。由于其中一个操作数（“2”）是一个字符串，所以JavaScript假定需要执行字符串连接，因此将1的类型转换为“1”，1 +“2”转换为“12”。然后，“12”+“2”产生“122”。

示例2：1 + +“2”+“2”输出：“32”说明：根据操作顺序，要执行的第一个操作是+“2”（第一个“2”之前的额外+被视为一个一元运算符）。因此，JavaScript将“2”的类型转换为数字，然后将一元+符号应用于它（即将其视为正数）。结果，下一个操作现在是1 + 2，当然这会产生3.但是，我们有一个数字和一个字符串之间的操作（即3和“2”），所以JavaScript再次转换数值赋给一个字符串并执行字符串连接，产生“32”。

示例3：1 + - “1”+“2”输出：“02”说明：这里的解释与前面的示例相同，只是一元运算符是 - 而不是+。因此，“1”变为1，然后在应用 - 时将其变为-1，然后将其加1到产生0，然后转换为字符串并与最终的“2”操作数连接，产生“02”。

示例4：+“1”+“1”+“2”输出：“112”说明：尽管第一个“1”操作数是基于其前面的一元+运算符的数值类型转换的，当它与第二个“1”操作数连接在一起时返回一个字符串，然后与最终的“2”操作数连接，产生字符串“112”。

示例5：“A” - “B”+“2”输出：“NaN2”说明：由于 - 运算符不能应用于字符串，并且既不能将“A”也不能将“B”转换为数值， “ - ”B“产生NaN，然后与字符串”2“串联产生”NaN2“。

例6：“A” - “B”+2输出：NaN说明：在前面的例子中，“A” - “B”产生NaN。但是任何运算符应用于NaN和其他数字操作数仍然会产生NaN。

### ****16、如果数组列表太大，以下递归代码将导致堆栈溢出。你如何解决这个问题，仍然保留递归模式？****

var list = readHugeList();var nextListItem = function() { var item = list.pop();

if (item) { // process the list item... nextListItem(); }};

通过修改nextListItem函数可以避免潜在的堆栈溢出，如下所示：

var list = readHugeList();var nextListItem = function() { var item = list.pop();

if (item) { // process the list item... setTimeout( nextListItem, 0); }};

堆栈溢出被消除，因为事件循环处理递归，而不是调用堆栈。当nextListItem运行时，如果item不为null，则将超时函数（nextListItem）推送到事件队列，并且函数退出，从而使调用堆栈清零。当事件队列运行超时事件时，将处理下一个项目，并设置一个计时器以再次调用nextListItem。因此，该方法从头到尾不经过直接递归调用即可处理，因此调用堆栈保持清晰，无论迭代次数如何。

### ****17、什么是JavaScript中的“闭包”？举一个例子。****

闭包是一个内部函数，它可以访问外部（封闭）函数的作用域链中的变量。闭包可以访问三个范围内的变量;具体来说：（1）变量在其自己的范围内，（2）封闭函数范围内的变量（3）全局变量。

这里是一个例子：

var globalVar = "xyz";(function outerFunc(outerArg) { var outerVar = 'a';

(function innerFunc(innerArg) { var innerVar = 'b';

console.log( "outerArg = " + outerArg + " " + "innerArg = " + innerArg + " " + "outerVar = " + outerVar + " " + "innerVar = " + innerVar + " " + "globalVar = " + globalVar);

})(456);})(123);

在上面的例子中，innerFunc，outerFunc和全局名称空间的变量都在innerFunc的范围内。上面的代码将产生以下输出：

outerArg = 123innerArg = 456outerVar = ainnerVar = bglobalVar = xyz

### ****18、以下代码的输出是什么：****

for (var i = 0; i < 5; i++) { setTimeout(function() { console.log(i); }, i \* 1000 );}

解释你的答案。如何在这里使用闭包？

显示的代码示例不会显示值0,1,2,3和4，这可能是预期的;而是显示5,5,5,5。

这是因为循环内执行的每个函数将在整个循环完成后执行，因此所有函数都会引用存储在i中的最后一个值，即5。

通过为每次迭代创建一个唯一的作用域，可以使用闭包来防止这个问题，并将该变量的每个唯一值存储在其作用域中，如下所示：

for (var i = 0; i < 5; i++) { (function(x) { setTimeout(function() { console.log(x); }, x \* 1000 ); })(i);}

这会产生将0,1,2,3和4记录到控制台的可能结果。

在ES2015上下文中，您可以在原始代码中简单地使用let而不是var：

for (let i = 0; i < 5; i++) { setTimeout(function() { console.log(i); }, i \* 1000 );}

### ****19、以下几行代码输出到控制台？****

console.log("0 || 1 = "+(0 || 1));console.log("1 || 2 = "+(1 || 2));console.log("0 && 1 = "+(0 && 1));console.log("1 && 2 = "+(1 && 2));

解释你的答案。

该代码将输出以下四行：

0 || 1 = 11 || 2 = 10 && 1 = 01 && 2 = 2

在JavaScript中，都是||和&&是逻辑运算符，当从左向右计算时返回第一个完全确定的“逻辑值”。

或（||）运算符。在形式为X || Y的表达式中，首先计算X并将其解释为布尔值。如果此布尔值为真，则返回true（1），并且不计算Y，因为“或”条件已经满足。但是，如果此布尔值为“假”，我们仍然不知道X || Y是真还是假，直到我们评估Y，并将其解释为布尔值。

因此，0 || 1评估为真（1），正如1 || 2。

和（&&）运算符。在X && Y形式的表达式中，首先评估X并将其解释为布尔值。如果此布尔值为false，则返回false（0）并且不评估Y，因为“and”条件已失败。但是，如果这个布尔值为“真”，我们仍然不知道X && Y是真还是假，直到我们评估Y，并将其解释为布尔值。

然而，&&运算符的有趣之处在于，当表达式评估为“真”时，则返回表达式本身。这很好，因为它在逻辑表达式中被视为“真”，但也可以用于在您关心时返回该值。这解释了为什么，有点令人惊讶的是，1 && 2返回2（而你可能会期望它返回true或1）。

### ****20 、下面的代码执行时输出是什么？说明。****

console.log(false == '0')console.log(false === '0')

该代码将输出：

truefalse

在JavaScript中，有两套相等运算符。三重相等运算符===的行为与任何传统的相等运算符相同：如果两侧的两个表达式具有相同的类型和相同的值，则计算结果为true。然而，双等号运算符在比较它们之前试图强制这些值。因此，通常使用===而不是==。对于！== vs！=也是如此。

### ****21、以下代码的输出是什么？解释你的答案。****

var a={}, b={key:'b'}, c={key:'c'};

a[b]=123;a[c]=456;

console.log(a[b]);

此代码的输出将是456（不是123）。

原因如下：设置对象属性时，JavaScript会隐式地将参数值串联起来。在这种情况下，由于b和c都是对象，它们都将被转换为“[object Object]”。因此，a [b]和a [c]都等价于[“[object Object]”]，并且可以互换使用。因此，设置或引用[c]与设置或引用[b]完全相同。

### ****22、以下代码将输出到控制台中.****

console.log((function f(n){return ((n > 1) ? n \* f(n-1) : n)})(10));

该代码将输出10阶乘的值（即10！或3,628,800）。

原因如下：

命名函数f()以递归方式调用自身，直到它调用f(1)，它简单地返回1.因此，这就是它的作用：

f(1): returns n, which is 1f(2): returns 2 \* f(1), which is 2f(3): returns 3 \* f(2), which is 6f(4): returns 4 \* f(3), which is 24f(5): returns 5 \* f(4), which is 120f(6): returns 6 \* f(5), which is 720f(7): returns 7 \* f(6), which is 5040f(8): returns 8 \* f(7), which is 40320f(9): returns 9 \* f(8), which is 362880f(10): returns 10 \* f(9), which is 3628800

### ****23 、考虑下面的代码片段。控制台的输出是什么，为什么？****

(function(x) { return (function(y) { console.log(x); })(2)})(1);

输出将为1，即使x的值从未在内部函数中设置。原因如下：

正如我们的JavaScript招聘指南中所解释的，闭包是一个函数，以及创建闭包时在范围内的所有变量或函数。在JavaScript中，闭包被实现为“内部函数”;即在另一功能的主体内定义的功能。闭包的一个重要特征是内部函数仍然可以访问外部函数的变量。

因此，在这个例子中，因为x没有在内部函数中定义，所以在外部函数的作用域中搜索一个定义的变量x，该变量的值为1。

### ****24、以下代码将输出到控制台以及为什么****

var hero = { \_name: 'John Doe', getSecretIdentity: function (){ return this.\_name; }};var stoleSecretIdentity = hero.getSecretIdentity;

console.log(stoleSecretIdentity());console.log(hero.getSecretIdentity());

这段代码有什么问题，以及如何解决这个问题。

该代码将输出：

undefinedJohn Doe

第一个console.log打印未定义，因为我们从hero对象中提取方法，所以stoleSecretIdentity()在\_name属性不存在的全局上下文（即窗口对象）中被调用。

修复stoleSecretIdentity()函数的一种方法如下：

var stoleSecretIdentity = hero.getSecretIdentity.bind(hero);

### ****25、创建一个函数，给定页面上的DOM元素，将访问元素本身及其所有后代（***不仅仅是它的直接子元素***）。对于每个访问的元素，函数应该将该元素传递给提供的回调函数。****

该函数的参数应该是：

* 一个 DOM 元素
* 一个回调函数（以DOM元素作为参数）

访问树中的所有元素（DOM）是经典的深度优先搜索算法应用程序。以下是一个示例解决方案：

function Traverse(p\_element,p\_callback) { p\_callback(p\_element); var list = p\_element.children; for (var i = 0; i < list.length; i++) { Traverse(list[i],p\_callback); // recursive call }}

### ****27、在JavaScript中测试您的这些知识：以下代码的输出是什么？****

var length = 10;function fn() { console.log(this.length);}var obj = { length: 5, method: function(fn) { fn(); arguments[0](); }};

obj.method(fn, 1);

输出：

102

为什么不是10和5？

首先，由于fn作为函数方法的参数传递，函数fn的作用域（this）是窗口。 var length = 10;在窗口级别声明。它也可以作为window.length或length或this.length来访问（当这个===窗口时）。

方法绑定到Object obj，obj.method用参数fn和1调用。虽然方法只接受一个参数，但调用它时已经传递了两个参数;第一个是函数回调，其他只是一个数字。

当在内部方法中调用fn()时，该函数在全局级别作为参数传递，this.length将有权访问在Object obj中定义的var length = 10（全局声明）而不是length = 5。

现在，我们知道我们可以使用arguments []数组访问JavaScript函数中的任意数量的参数。

因此arguments0只不过是调用fn()。在fn里面，这个函数的作用域成为参数数组，并且记录参数[]的长度将返回2。

因此输出将如上所述。

### ****28、考虑下面的代码。输出是什么，为什么？****

(function () { try { throw new Error(); } catch (x) { var x = 1, y = 2; console.log(x); } console.log(x); console.log(y);})();

1undefined2

var语句被挂起（没有它们的值初始化）到它所属的全局或函数作用域的顶部，即使它位于with或catch块内。但是，错误的标识符只在catch块内部可见。它相当于：

(function () { var x, y; // outer and hoisted try { throw new Error(); } catch (x /\* inner \*/) { x = 1; // inner x, not the outer one y = 2; // there is only one y, which is in the outer scope console.log(x /\* inner \*/); } console.log(x); console.log(y);})();

### ****29、这段代码的输出是什么？****

var x = 21;var girl = function () { console.log(x); var x = 20;};girl ();

21，也不是20，结果是‘undefined’的

这是因为JavaScript初始化没有被挂起。

（为什么它不显示21的全局值？原因是当函数执行时，它检查是否存在本地x变量但尚未声明它，因此它不会查找全局变量。）

### ****30、你如何克隆一个对象？****

var obj = {a: 1 ,b: 2}var objclone = Object.assign({},obj);

现在objclone的值是{a：1，b：2}，但指向与obj不同的对象。

但请注意潜在的缺陷：Object.clone()只会执行浅拷贝，而不是深拷贝。这意味着嵌套的对象不会被复制。他们仍然引用与原始相同的嵌套对象：

let obj = { a: 1, b: 2, c: { age: 30 }};var objclone = Object.assign({},obj);console.log('objclone: ', objclone);

obj.c.age = 45;console.log('After Change - obj: ', obj); // 45 - This also changesconsole.log('After Change - objclone: ', objclone); // 45

### ****31、此代码将打印什么？****

for (let i = 0; i < 5; i++) { setTimeout(function() { console.log(i); }, i \* 1000 );}

它会打印0 1 2 3 4，因为我们在这里使用let而不是var。变量i只能在for循环的块范围中看到。

### ****32、以下几行输出什么，为什么？****

console.log(1 < 2 < 3);console.log(3 > 2 > 1);

第一条语句返回true，如预期的那样。

第二个返回false是因为引擎如何针对<和>的操作符关联性工作。它比较从左到右，所以3> 2> 1 JavaScript翻译为true> 1. true具有值1，因此它比较1> 1，这是错误的。

### ****33、如何在数组的开头添加元素？最后如何添加一个？****

var myArray = ['a', 'b', 'c', 'd'];myArray.push('end');myArray.unshift('start');console.log(myArray); // ["start", "a", "b", "c", "d", "end"]

使用ES6，可以使用扩展运算符：

myArray = ['start', ...myArray];myArray = [...myArray, 'end'];

或者，简而言之：

myArray = ['start', ...myArray, 'end'];

### ****34、想象一下你有这样的代码:****

var a = [1, 2, 3];

a）这会导致崩溃吗？

a[10] = 99;

b）这个输出是什么？

console.log(a[6]);

a）它不会崩溃。 JavaScript引擎将使阵列插槽3至9成为“空插槽”。

b）在这里，a [6]将输出未定义的值，但时隙仍为空，而不是未定义的。在某些情况下，这可能是一个重要的细微差别。例如，使用map()时，map()的输出中的空插槽将保持为空，但未定义的插槽将使用传递给它的函数重映射：

var b = [undefined];b[2] = 1;console.log(b); // (3) [undefined, empty × 1, 1]console.log(b.map(e => 7)); // (3) [7, empty × 1, 7]

### ****35、typeof undefined == typeof NULL的值是什么？****

该表达式将被评估为true，因为NULL将被视为任何其他未定义的变量。

注意：JavaScript区分大小写，我们在这里使用NULL而不是null。

### ****36、代码返回后会怎么样？****

console.log(typeof typeof 1);

打印结果：string

typeof 1将返回“number”，typeof“number”将返回字符串。

### ****37、以下代码输出什么？为什么？****

var b = 1;function outer(){ var b = 2 function inner(){ b++; var b = 3; console.log(b) } inner();}outer();

输出到控制台将是“3”。

在这个例子中有三个闭包，每个都有它自己的var b声明。当调用变量时，将按照从本地到全局的顺序检查闭包，直到找到实例。由于内部闭包有自己的b变量，这就是输出。

此外，由于提升内部的代码将被解释如下：

function inner () { var b; // b is undefined b++; // b is NaN b = 3; // b is 3 console.log(b); // output "3"}

面试比棘手的技术问题要多，所以这些仅仅是作为指导。并不是每个值得聘用的“A”候选人都能够回答所有问题，能够回答所有问题的也不能保证是“A”候选人。最后，招聘仍然是一门艺术，一门科学 - 还有很多工作要做。