JavaScript 面试问题及答案 （一）

1.使用 typeof bar === "object" 来确定 bar 是否是对象的潜在陷阱是什么?如何避免这个陷阱?

尽管 typeof bar === "object" 是检查 bar 是否对象的可靠方法，令人惊讶的是在JavaScript中 null 也被认为是对象!

因此，令大多数开发人员惊讶的是，下面的代码将输出 true (而不是false) 到控制台：

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

只要清楚这一点，同时检查 bar 是否为 null，就可以很容易地避免问题：

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

要答全问题，还有其他两件事情值得注意：

首先，上述解决方案将返回 false，当 bar 是一个函数的时候。在大多数情况下，这是期望行为，但当你也想对函数返回 true 的话，你可以修改上面的解决方案为：

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

第二，上述解决方案将返回 true，当 bar 是一个数组(例如，当 var bar = [];)的时候。在大多数情况下，这是期望行为，因为数组是真正的对象，但当你也想对数组返回 false 时，你可以修改上面的解决方案为：

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

或者，如果你使用jQuery的话：

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

2.下面的代码将输出什么到控制台，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 都定义在函数的封闭范围内，并且都始于 var关键字，大多数JavaScript开发(http://web.tedu.cn/data/js/)人员期望 typeof a 和 typeof b 在上面的例子中都是 undefined。

然而，事实并非如此。这里的问题是，大多数开发人员将语句 var a = b = 3; 错误地理解为是以下声明的简写：

var b = 3;var a = b;

但事实上，var a = b = 3; 实际是以下声明的简写：

b = 3;var a = b;

因此(如果你不使用严格模式的话)，该代码段的输出是：

a defined? falseb defined? true

但是， b 如何才能被定义在封闭函数的范围之外呢?是的，既然语句 var a = b = 3; 是语句 b = 3; 和 var a = b;的简写， b 最终成为了一个全局变量(因为它没有前缀 var 关键字)，因此仍然在范围内甚至封闭函数之外。

需要注意的是，在严格模式下(即使用 use strict)，语句var a = b = 3; 将生成 ReferenceError: b is not defined的运行时错误，从而避免任何否则可能会导致的 headfakes /bug。 (还是你为什么应该理所当然地在代码中使用 use strict 的最好例子!)

3.下面的代码将输出什么到控制台，为什么?

var myObject = {

foo: "bar",};

myObject.func();

上面的代码将输出以下内容到控制台：

outer func: this.foo = bar

outer func: self.foo = bar

inner func: this.foo = undefined

inner func: self.foo = bar

在外部函数中， this 和self 两者都指向了 myObject，因此两者都可以正确地引用和访问 foo。

在内部函数中， this 不再指向 myObject。其结果是，this.foo 没有在内部函数中被定义，相反，指向到本地的变量self 保持在范围内，并且可以访问。 (在ECMA 5之前，在内部函数中的this 将指向全局的 window 对象;反之，因为作为ECMA 5，内部函数中的功能this 是未定义的。)

4.封装JavaScript源文件的全部内容到一个函数块有什么意义及理由?

这是一个越来越普遍的做法，被许多流行的JavaScript库(jQuery，Node.js等)采用。这种技术创建了一个围绕文件全部内容的闭包，也许是最重要的是，创建了一个私有的命名空间，从而有助于避免不同JavaScript模块和库之间潜在的名称冲突。

这种技术的另一个特点是，允许一个易于引用的(假设更短的)别名用于全局变量。这通常用于，例如，jQuery插件中。jQuery允许你使用jQuery.noConflict()，来禁用 $ 引用到jQuery命名空间。在完成这项工作之后，你的代码仍然可以使用$ 利用这种闭包技术，如下所示：

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

5.在JavaScript源文件的开头包含 use strict 有什么意义和好处?

对于这个问题，既简要又最重要的答案是，use strict 是一种在JavaScript代码运行时自动实行更严格解析和错误处理的方法。那些被忽略或默默失败了的代码错误，会产生错误或抛出异常。通常而言，这是一个很好的做法。

严格模式的一些主要优点包括：

使调试更加容易。那些被忽略或默默失败了的代码错误，会产生错误或抛出异常，因此尽早提醒你代码中的问题，你才能更快地指引到它们的源代码。

防止意外的全局变量。如果没有严格模式，将值分配给一个未声明的变量会自动创建该名称的全局变量。这是JavaScript中最常见的错误之一。在严格模式下，这样做的话会抛出错误。

消除 this 强制。如果没有严格模式，引用null或未定义的值到 this 值会自动强制到全局变量。这可能会导致许多令人头痛的问题和让人恨不得拔自己头发的bug。在严格模式下，引用 null或未定义的 this 值会抛出错误。

不允许重复的属性名称或参数值。当检测到对象(例如，var object = {foo: "bar", foo: "baz"};)中重复命名的属性，或检测到函数中(例如，function foo(val1, val2, val1){})重复命名的参数时，严格模式会抛出错误，因此捕捉几乎可以肯定是代码中的bug可以避免浪费大量的跟踪时间。

使eval() 更安全。在严格模式和非严格模式下，eval() 的行为方式有所不同。最显而易见的是，在严格模式下，变量和声明在 eval() 语句内部的函数不会在包含范围内创建(它们会在非严格模式下的包含范围中被创建，这也是一个常见的问题源)。

在 delete使用无效时抛出错误。delete操作符(用于从对象中删除属性)不能用在对象不可配置的属性上。当试图删除一个不可配置的属性时，非严格代码将默默地失败，而严格模式将在这样的情况下抛出异常。

6.考虑以下两个函数。它们会返回相同的东西吗？为什么相同或为什么不相同？

```
function foo1(){ return {

bar: "hello"

};
```

```
}function foo2(){ return

{

bar: "hello"

};

}
```

出人意料的是，这两个函数返回的内容并不相同。更确切地说是：

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

将产生：

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

这不仅是令人惊讶，而且特别让人困惑的是， foo2()返回undefined却没有任何错误抛出。

原因与这样一个事实有关，即分号在JavaScript中是一个可选项(尽管省略它们通常是非常糟糕的形式)。其结果就是，当碰到 foo2()中包含 return语句的代码行(代码行上没有其他任何代码)，分号会立即自动插入到返回语句之后。

也不会抛出错误，因为代码的其余部分是完全有效的，即使它没有得到调用或做任何事情(相当于它就是是一个未使用的代码块，定义了等同于字符串 "hello"的属性 bar)。

这种行为也支持放置左括号于JavaScript代码行的末尾，而不是新代码行开头的约定。正如这里所示，这不仅仅只是JavaScript中的一个风格偏好。

7. NaN 是什么?它的类型是什么?你如何可靠地测试一个值是否等于 NaN ?

NaN 属性代表一个“不是数字”的值。这个特殊的值是因为运算不能执行而导致的，不能执行的原因要么是因为其中的运算对象之一非数字(例如， "abc" / 4)，要么是因为运算的结果非数字(例如，除数为零)。

虽然这看上去很简单，但 NaN 有一些令人惊讶的特点，如果你不知道它们的话，可能会导致令人头痛的bug。

首先，虽然 NaN 意味着“不是数字”，但是它的类型，不管你信不信，是 Number：

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

此外， NaN 和任何东西比较——甚至是它自己本身!——结果是false：

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

一种半可靠的方法来测试一个数字是否等于 NaN，是使用内置函数 isNaN()，但即使使用 isNaN() 依然并非是一个完美的解决方案。

一个更好的解决办法是使用 value !== value，如果值等于NaN，只会产生true。另外，ES6提供了一个新的 Number.isNaN() 函数，这是一个不同的函数，并且比老的全局 isNaN() 函数更可靠。

8.下列代码将输出什么?并解释原因。

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

一个稍微有点编程基础的回答是：“你不能确定。可能会输出“0.3”和“true”，也可能不会。JavaScript中的数字和浮点精度的处理相同，因此，可能不会总是产生预期的结果。”

以上所提供的例子就是一个演示了这个问题的典型例子。但出人意料的是，它会输出：

0.30000000000000004false

9.讨论写函数 isInteger(x) 的可能方法，用于确定x是否是整数。

这可能听起来是小菜一碟，但事实上，这很琐碎，因为ECMAScript 6引入了一个新的正以此为目的 Number.isInteger() 函数。然而，之前的ECMAScript 6，会更复杂一点，因为没有提供类似的 Number.isInteger() 方法。

问题是，在ECMAScript规格说明中，整数只概念上存在：即，数字值总是存储为浮点值。

考虑到这一点，最简单又最干净的ECMAScript6之前的解决方法(同时也非常稳健地返回 false ，即使一个非数字的值，如字符串或 null ，被传递给函数)如下：

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

下面的解决方法也是可行的，虽然不如上面那个方法优雅：

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

请注意 Math.ceil() 和 Math.floor() 在上面的实现中等同于 Math.round()。

或：

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

相当普遍的一个不正确的解决方案是：

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

虽然这个以 parseInt函数为基础的方法在 x 取许多值时都能工作良好，但一旦 x 取值相当大的时候，就会无法正常工作。问题在于 parseInt() 在解析数字之前强制其第一个参数到字符串。因此，一旦数目变得足够大，它的字符串就会表达为指数形式(例如， 1e+21)。因此，parseInt() 函数就会去解析 1e+21，但当到达 e字符串的时候，就会停止解析，因此只会返回值 1。注意：

```
> String(1000000000000000000000)'1e+21'> parseInt(1000000000000000000000, 10)1> parseInt(1000000000000000000000, 10) === 1000000000000000000000false
```

10.下列代码行1-4如何排序，使之能够在执行代码时输出到控制台？为什么？

```
(function() { console.log(1);

setTimeout(function(){console.log(2)}, 1000);

setTimeout(function(){console.log(3)}, 0);

console.log(4);

})();
```

序号如下：

1

4

3

2

让我们先来解释比较明显而易见的那部分：

1 和 4之所以放在前面，是因为它们是通过简单调用 console.log() 而没有任何延迟输出的

2 之所以放在 3的后面，是因为 2 是延迟了1000毫秒(即，1秒)之后输出的，而 3 是延迟了0毫秒之后输出的。

好的。但是，既然 3 是0毫秒延迟之后输出的，那么是否意味着它是立即输出的呢?如果是的话，那么它是不是应该在 4 之前输出，既然 4 是在第二行输出的?

要回答这个问题，你需要正确理解JavaScript的事件和时间设置。

浏览器有一个事件循环，会检查事件队列和处理未完成的事件。例如，如果时间发生在后台(例如，脚本的 onload 事件)时，浏览器正忙(例如，处理一个 onclick)，那么事件会添加到队列中。当onclick处理程序完成后，检查队列，然后处理该事件(例如，执行 onload 脚本)。

同样的， setTimeout() 也会把其引用的函数的执行放到事件队列中，如果浏览器正忙的话。

当setTimeout()的第二个参数为0的时候，它的意思是“尽快”执行指定的函数。具体而言，函数的执行会放置在事件队列的下一个计时器开始。但是请注意，这不是立即执行：函数不会被执行除非下一个计时器开始。这就是为什么在上述的例子中，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被调用的，因此会稍微延迟)。

11.写一个简单的函数(少于80个字符)，要求返回一个布尔值指明字符串是否为回文结构。

下面这个函数在 str 是回文结构的时候返回true，否则，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'

12.写一个 sum方法，在使用下面任一语法调用时，都可以正常工作。

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

(至少)有两种方法可以做到：

方法1

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

} else { return function(y) { return x + y; };

}

}

在JavaScript中，函数可以提供到 arguments 对象的访问，arguments 对象提供传递到函数的实际参数的访问。这使我们能够使用 length 属性来确定在运行时传递给函数的参数数量。

如果传递两个参数，那么只需加在一起，并返回。

否则，我们假设它被以 sum(2)(3)这样的形式调用，所以我们返回一个匿名函数，这个匿名函数合并了传递到 sum()的参数和传递给匿名函数的参数。

方法2

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

} else { return function(y) { return x + y; };

}

}

当调用一个函数的时候，JavaScript不要求参数的数目匹配函数定义中的参数数量。如果传递的参数数量大于函数定义中参数数量，那么多余参数将简单地被忽略。另一方面，如果传递的参数数量小于函数定义中的参数数量，那么缺少的参数在函数中被引用时将会给一个

undefined值。所以，在上面的例子中，简单地检查第2个参数是否未定义，就可以相应地确定函数被调用以及进行的方式。

13.请看下面的代码片段：

```
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“Button 4”的时候会输出什么到控制台，为什么?(b)提供一个或多个备用的可按预期工作的实现方案。

(a)无论用户点击什么按钮，数字5将总会输出到控制台。这是因为，当 onclick 方法被调用(对于任何按钮)的时候， for 循环已经结束，变量 i 已经获得了5的值。(面试者如果能够谈一谈有关如何执行上下文，可变对象，激活对象和内部“范围”属性贡有助于闭包行为，则可以加分)。

(b)要让代码工作的关键是，通过传递到一个新创建的函数对象，在每次传递通过 for 循环时，捕捉到 i 值。下面是三种可能实现的方法：

```
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，你可以封装全部调用到在新匿名函数中的 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);

}
```

也可以调用数组对象的本地 forEach 方法来替代 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);

});
```

14.下面的代码将输出什么到控制台，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));
```

输出结果是:

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

arr1 和 arr2 在上述代码执行之后，两者相同了，原因是:

调用数组对象的 reverse() 方法并不只返回反顺序的阵列，它也反转了数组本身的顺序(即，在这种情况下，指的是 arr1)。

reverse() 方法返回一个到数组本身的引用(在这种情况下即，arr1)。其结果为，arr2 仅仅是一个到 arr1的引用(而不是副本)。因此，当对 arr2做了任何事情(即当我们调用 arr2.push(arr3);)时，arr1 也会受到影响，因为 arr1 和 arr2 引用的是同一个对象。

这里有几个侧面点有时候会让你在回答这个问题时，阴沟里翻船：

传递数组到另一个数组的 push() 方法会让整个数组作为单个元素映射到数组的末端。其结果是，语句 arr2.push(arr3); 在其整体中添加 arr3 作为一个单一的元素到 arr2 的末端(也就是说，它并没有连接两个数组，连接数组是 concat() 方法的目的)。

和Python一样，JavaScript标榜数组方法调用中的负数下标，例如 slice() 可作为引用数组末尾元素的方法：例如，-1下标表示数组中的最后一个元素，等等。

15.下面的代码将输出什么到控制台，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);

上面的代码将输出以下内容到控制台：

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

原因是...

这里的根本问题是，JavaScript(ECMAScript)是一种弱类型语言，它可对值进行自动类型转换，以适应正在执行的操作。让我们通过上面的例子来说明这是如何做到的。

例1：1 + "2" + "2" 输出："122" 说明： 1 + "2" 是执行的第一个操作。由于其中一个运算对象("2")是字符串，JavaScript会假设它需要执行字符串连接，因此，会将 1 的类型转换为 "1"， 1 + "2"结果就是 "12"。然后， "12" + "2" 就是 "122"。

例2： 1 + +"2" + "2" 输出： "32" 说明：根据运算的顺序，要执行的第一个运算是 +"2" (第一个 "2" 前面的额外 + 被视为一元运算符)。因此，JavaScript将 "2" 的类型转换为数字，然后应用一元 + 号(即，将其视为一个正数)。其结果是，接下来的运算就是 1 + 2 ，这当然是 3。然后我们需要在一个数字和一个字符串之间进行运算(即， 3 和 "2")，同样的，JavaScript会将数值类型转换为字符串，并执行字符串的连接，产生 "32"。

例3： 1 + -"1" + "2" 输出： "02" 说明：这里的解释和前一个例子相同，除了此处的一元运算符是 - 而不是 +。先是 "1" 变为 1，然后当应用 - 时又变为了 -1 ，然后将其与 1相加，结果为 0，再将其转换为字符串，连接最后的 "2" 运算对象，得到 "02"。

例4： +"1" + "1" + "2" 输出： "112" 说明：虽然第一个运算对象 "1"因为前缀的一元 + 运算符类型转换为数值，但又立即转换回字符串，当连接到第二个运算对象 "1" 的时候，然后又和最后的运算对象"2" 连接，产生了字符串 "112"。

例5： "A" - "B" + "2" 输出： "NaN2" 说明：由于运算符 - 不能被应用于字符串，并且 "A" 和 "B" 都不能转换成数值，因此，"A" - "B"的结果是 NaN，然后再和字符串 "2" 连接，得到 "NaN2" 。

例6： "A" - "B" + 2 输出： NaN 说明：参见前一个例子， "A" - "B" 结果为 NaN。但是，应用任何运算符到NaN与其他任何的数字运算对象，结果仍然是 NaN。

16.下面的递归代码在数组列表偏大的情况下会导致堆栈溢出。在保留递归模式的基础上，你怎么解决这个问题?

var list = readHugeList();var nextListItem = function() { var item = list.pop(); if (item) { // process the list item...

nextListItem();

}

};

潜在的堆栈溢出可以通过修改nextListItem 函数避免：

var list = readHugeList();var nextListItem = function() { var item = list.pop(); if (item) { // process the list item...

setTimeout(nextListItem, 0);

}

};

堆栈溢出之所以会被消除，是因为事件循环操纵了递归，而不是调用堆栈。当 nextListItem 运行时，如果 item不为空，timeout函数(nextListItem)就会被推到事件队列，该函数退出，因此就清空调用堆栈。当事件队列运行其timeout事件，且进行到下一个 item 时，定时器被设置为再次调用 extListItem。因此，该方法从头到尾都没有直接的递归调用，所以无论迭代次数的多少，调用堆栈保持清空的状态。

17.JavaScript中的“闭包”是什么?请举一个例子。

闭包是一个可以访问外部(封闭)函数作用域链中的变量的内部函数。闭包可以访问三种范围中的变量：这三个范围具体为：(1)自己范围内的变量，(2)封闭函数范围内的变量，以及(3)全局变量。

下面是一个简单的例子：

var globalVar = "xyz";

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

(function innerFunc(innerArg) { var innerVar = 'b'; console.log("outerArg = " + outerArg + "\n" + "innerArg = " + innerArg + "\n" + "outerVar = " + outerVar + "\n" + "innerVar = " + innerVar + "\n" + "globalVar = " + globalVar);

```
})(456);

})(123);
```

在上面的例子中，来自于 innerFunc， outerFunc和全局命名空间的变量都在 innerFunc的范围内。因此，上面的代码将输出如下：

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

18.下面的代码将输出什么：

```
for (var i = 0; i < 5; i++) {

setTimeout(function() { console.log(i); }, i * 1000 );

}
```

解释你的答案。闭包在这里能起什么作用?

上面的代码不会按预期显示值0，1，2，3，和4，而是会显示5，5，5，5，和5。

原因是，在循环中执行的每个函数将整个循环完成之后被执行，因此，将会引用存储在 i中的最后一个值，那就是5。

闭包可以通过为每次迭代创建一个唯一的范围，存储范围内变量的每个唯一的值，来防止这个问题，如下：

```
for (var i = 0; i < 5; i++) {

(function(x) {

setTimeout(function() { console.log(x); }, x * 1000 );

})(i);

}
```

这就会按预期输出0，1，2，3，和4到控制台。

19.以下代码行将输出什么到控制台?

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

并解释。

该代码将输出：

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

在JavaScript中， || 和 &&都是逻辑运算符，用于在从左至右计算时，返回第一个可完全确定的“逻辑值”。

或(||)运算符。在形如 X||Y的表达式中，首先计算X 并将其解释执行为一个布尔值。如果这个布尔值true，那么返回true(1)，不再计算 Y，因为“或”的条件已经满足。如果这个布尔值为false，那么我们仍然不能知道 X||Y是真是假，直到我们计算 Y，并且也把它解释执行为一个布尔值。

因此， 0 || 1 的计算结果为true(1)，同理计算1 || 2。

与(&&)运算符。在形如 X&&Y的表达式中，首先计算 X并将其解释执行为一个布尔值。如果这个布尔值为 false，那么返回 false(0)，不再计算 Y，因为“与”的条件已经失败。如果这个布尔值为true，但是，我们仍然不知道 X&&Y 是真是假，直到我们去计算 Y，并且也把它解释执行为一个布尔值。

不过，关于 &&运算符有趣的地方在于，当一个表达式计算为“true”的时候，那么就返回表达式本身。这很好，虽然它在逻辑表达式方面计算为“真”，但如果你希望的话也可用于返回该值。这就解释了为什么，有些令人奇怪的是， 1 && 2返回 2(而不是你以为的可能返回 true 或 1)。

20.执行下面的代码时将输出什么?请解释。

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

代码将输出：

truefalse

在JavaScript中，有两种等式运算符。三个等于运算符 === 的作用类似传统的等于运算符：如果两侧的表达式有着相同的类型和相同的值，那么计算结果为true。而双等于运算符，会只强制比较它们的值。因此，总体上而言，使用 ===而不是 ==的做法更好。 !==vs !=亦是同理。

21.以下代码将输出什么?并解释你的答案。

var a={},

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

a[b]=123;

a[c]=456;

console.log(a[b]);

这段代码将输出 456(而不是 123)。

原因为：当设置对象属性时，JavaScript会暗中字符串化参数值。在这种情况下，由于 b 和c都是对象，因此它们都将被转换为"[object Object]"。结果就是， a[b]和a[c]均相当于a["[object Object]"]，并可以互换使用。因此，设置或引用 a[c]和设置或引用 a[b]完全相同。

22.以下代码行将输出什么到控制台?

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

并解释你的答案。

代码将输出10!的值(即10!或3628800)。

原因是：

命名函数 f()递归地调用本身，当调用 f(1)的时候，只简单地返回1。下面就是它的调用过程：

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.请看下面的代码段。控制台将输出什么，为什么?

(function(x) { return (function(y) { console.log(x);

})(2)

})(1);

控制台将输出 1，即使从来没有在函数内部设置过x的值。原因是：

闭包是一个函数，连同在闭包创建的时候，其范围内的所有变量或函数一起。在JavaScript中，闭包是作为一个“内部函数”实施的：即，另一个函数主体内定义的函数。闭包的一个重要特征是，内部函数仍然有权访问外部函数的变量。

因此，在本例中，由于 x未在函数内部中定义，因此在外部函数范围中搜索定义的变量 x，且被发现具有1的值。

24.下面的代码将输出什么到控制台，为什么：

var hero = {

_name: 'John Doe',

getSecretIdentity: function (){ return this._name;

}

}; var stoleSecretIdentity = hero.getSecretIdentity;console.log(stoleSecretIdentity());console.log(hero.getSecretIdentity());

代码有什么问题，以及应该如何修复。

代码将输出：

undefinedJohn Doe

第一个 console.log之所以输出 undefined，是因为我们正在从 hero对象提取方法，所以调用了全局上下文中(即窗口对象)的 stoleSecretIdentity()，而在此全局上下文中， _name属性不存在。

其中一种修复stoleSecretIdentity() 函数的方法如下：

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

25.创建一个给定页面上的一个DOM (http://web.tedu.cn/courses/4003.html)元素，就会去访问元素本身及其所有子元素(不只是它的直接子元素)的函数。对于每个被访问的元素，函数应该传递元素到提供的回调函数。

此函数的参数为：

DOM元素

回调函数(将DOM元素作为其参数)

访问树(DOM)的所有元素是经典的深度优先搜索算法应用。下面是一个示范的解决方案：

```
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

}

}
```

感谢大家阅读由java培训机构 (http://java.tedu.cn/)分享的“JavaScript 面试问题及答案”希望对大家有所帮助，更多精彩内容请关注Java培训 (http://java.tedu.cn/)官网

上一篇：Java面试经验，兼谈互联网公司后端面试经验 (http://java.tedu.cn/data/topic/256623.html)
下一篇：面试笔记Java NIO 核心组件 (http://java.tedu.cn/data/topic/257649.html)