WEB开发二题库

**1. JavaScript 中的强制转型是指什么？**

难度星级: \*  
在 JavaScript 中，两种不同的内置类型间的转换被称为强制转型。强制转型在 JavaScript 中有两种形式：显式和隐式。

这是一个显式强制转型的例子：

var a = "42";

var b = Number( a );

a; // "42"

b; // 42 -- 是个数字!

这是一个隐式强制转型的例子：

var a = "42";

var b = a \* 1; // "42" 隐式转型成 42

a; // "42"

b; // 42 -- 是个数字!

**2. JavaScript 中的作用域（scope）是指什么？**

难度星级: \*  
在 JavaScript 中，每个函数都有自己的作用域。作用域基本上是变量以及如何通过名称访问这些变量的规则的集合。只有函数中的代码才能访问函数作用域内的变量。

同一个作用域中的变量名必须是唯一的。一个作用域可以嵌套在另一个作用域内。如果一个作用域嵌套在另一个作用域内，最内部作用域内的代码可以访问另一个作用域的变量。

**3. 解释 JavaScript 中的相等性。**

难度星级: \*  
JavaScript 中有严格比较和类型转换比较：

* 严格比较（例如 ===）在不允许强制转型的情况下检查两个值是否相等；
* 抽象比较（例如 ==）在允许强制转型的情况下检查两个值是否相等。

var a = "42";

var b = 42;

a == b; // true

a === b; // false

一些简单的规则：

* 如果被比较的任何一个值可能是 true 或 false，要用 ===，而不是 ==；
* 如果被比较的任何一个值是这些特定值（0、“”或 []），要用 ===，而不是 ==；
* 在其他情况下，可以安全地使用 ==。它不仅安全，而且在很多情况下，它可以简化代码，并且提升代码可读性。

**4. 解释什么是回调函数，并提供一个简单的例子。**

难度星级: \*\*  
回调函数是可以作为参数传递给另一个函数的函数，并在某些操作完成后执行。下面是一个简单的回调函数示例，这个函数在某些操作完成后打印消息到控制台。

function modifyArray(arr, callback) {

// 对 arr 做一些操作

arr.push(100);

// 执行传进来的 callback 函数

callback();

}

var arr = [1, 2, 3, 4, 5];

modifyArray(arr, function() {

console.log("array has been modified", arr);

});

**5. “use strict”的作用是什么？**

难度星级: \*\*  
use strict 出现在 JavaScript 代码的顶部或函数的顶部，可以帮助你写出更安全的 JavaScript 代码。如果你错误地创建了全局变量，它会通过抛出错误的方式来警告你。例如，以下程序将抛出错误：

function doSomething(val) {

"use strict";

x = val + 10;

}

它会抛出一个错误，因为 x 没有被定义，并使用了全局作用域中的某个值对其进行赋值，而 use strict 不允许这样做。下面的小改动修复了这个错误：

function doSomething(val) {

"use strict";

var x = val + 10;

}

**6. 解释 JavaScript 中的 null 和 undefined。**

难度星级: \*\*  
JavaScript 中有两种底层类型：null 和 undefined。它们代表了不同的含义：

* 尚未初始化：undefined；
* 空值：null。

null和undefined是两个不同的对象, 有图为证:

image.png

**7. 编写一个可以执行如下操作的函数。**

难度星级: \*\*

var addSix = createBase(6);

addSix(10); // 返回 16

addSix(21); // 返回 27

可以创建一个闭包来存放传递给函数 createBase 的值。被返回的内部函数是在外部函数中创建的，内部函数就成了一个闭包，它可以访问外部函数中的变量，在本例中是变量 baseNumber。

function createBase(baseNumber) {

return function(N) {

// 我们在这里访问 baseNumber，即使它是在这个函数之外声明的。

// JavaScript 中的闭包允许我们这么做。

return baseNumber + N;

}

}

var addSix = createBase(6);

addSix(10);

addSix(21);

**8. 解释 JavaScript 中的值和类型。**

难度星级: \*\*  
JavaScript提供两种数据类型: 基本数据类型和引用数据类型  
基本数据类型有:

* String
* Number
* Boolean
* Null
* Undefined
* Symbol

引用数据类型有:

* Object
* Array
* Function

**9. 解释事件冒泡以及如何阻止它？**

难度星级: \*\*  
事件冒泡是指嵌套最深的元素触发一个事件，然后这个事件顺着嵌套顺序在父元素上触发。

防止事件冒泡的一种方法是使用 event.cancelBubble 或 event.stopPropagation()（低于 IE 9）。

**10. JavaScript 中的 let 关键字有什么用？**

难度星级: \*\*  
除了可以在函数级别声明变量之外，ES6 还允许你使用 let 关键字在代码块（{..}）中声明变量。

**11. 如何检查一个数字是否为整数？**

难度星级: \*\*  
检查一个数字是小数还是整数，可以使用一种非常简单的方法，就是将它对 1 进行取模，看看是否有余数。

function isInt(num) {

return num % 1 === 0;

}

console.log(isInt(4)); // true

console.log(isInt(12.2)); // false

console.log(isInt(0.3)); // false

**12. 什么是 IIFE（立即调用函数表达式）？**

难度星级: \*\*  
它是立即调用函数表达式（Immediately-Invoked Function Expression），简称 IIFE。函数被创建后立即被执行：

(function IIFE(){

console.log( "Hello!" );

})();

// "Hello!"

在避免污染全局命名空间时经常使用这种模式，因为 IIFE（与任何其他正常函数一样）内部的所有变量在其作用域之外都是不可见的。

**13. 如何在 JavaScript 中比较两个对象？**

难度星级: \*\*\*  
对于两个非原始值，比如两个对象（包括函数和数组），== 和 === 比较都只是检查它们的引用是否匹配，并不会检查实际引用的内容。

例如，默认情况下，数组将被强制转型成字符串，并使用逗号将数组的所有元素连接起来。所以，两个具有相同内容的数组进行 == 比较时不会相等：

var a = [1,2,3];

var b = [1,2,3];

var c = "1,2,3";

a == c; // true

b == c; // true

a == b; // false

对于对象的深度比较，可以使用 deep-equal 这个库，或者自己实现递归比较算法

**14. 你能解释一下 ES5 和 ES6 之间的区别吗？**

难度星级: \*\*\*

* ECMAScript 5（ES5）：ECMAScript 的第 5 版，于 2009 年标准化。这个标准已在所有现代浏览器中完全实现。
* ECMAScript 6（ES6）或 ECMAScript 2015（ES2015）：第 6 版 ECMAScript，于 2015 年标准化。这个标准已在大多数现代浏览器中部分实现。

以下是 ES5 和 ES6 之间的一些主要区别：

**箭头函数和字符串插值：**

const greetings = (name) => {

return `hello ${name}`;

}

const greetings = name => `hello ${name}`;

**常量**

常量在很多方面与其他语言中的常量一样，但有一些需要注意的地方。常量表示对值的“固定引用”。因此，在使用常量时，你实际上可以改变变量所引用的对象的属性，但无法改变引用本身。

const NAMES = [];

NAMES.push("Jim");

console.log(NAMES.length === 1); // true

NAMES = ["Steve", "John"]; // error

**块作用域变量。**

新的 ES6 关键字 let 允许开发人员声明块级别作用域的变量。let 不像 var 那样可以进行提升。

**默认参数值**

默认参数允许我们使用默认值初始化函数。如果省略或未定义参数，则使用默认值，也就是说 null 是有效值。

// 基本语法

function multiply (a, b = 2) {

return a \* b;

}

multiply(5); // 10

**类定义和继承**

ES6 引入了对类（关键字 class）、构造函数（关键字 constructor）和用于继承的 extend 关键字的支持。

**for…of 操作符**

for…of 语句将创建一个遍历可迭代对象的循环。

**用于对象合并的 Spread 操作**

const obj1 = { a: 1, b: 2 }

const obj2 = { a: 2, c: 3, d: 4}

const obj3 = {...obj1, ...obj2}

**promise**

promise 提供了一种机制来处理异步操作结果。你可以使用回调来达到同样的目的，但是 promise 通过方法链接和简洁的错误处理带来了更高的可读性。

const isGreater = (a, b) => {

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

if(a > b) {

resolve(true)

} else {

reject(false)

}

})

}

isGreater(1, 2)

.then(result => {

console.log('greater')

})

.catch(result => {

console.log('smaller')

})

**模块导出和导入**

const myModule = { x: 1, y: () => { console.log('This is ES5') }}

export default myModule;

import myModule from './myModule';

**15. 解释 JavaScript 中“undefined”和“not defined”之间的区别。**

难度星级: \*\*\*  
在 JavaScript 中，如果你试图使用一个不存在且尚未声明的变量，JavaScript 将抛出错误“var name is not defined”，让后脚本将停止运行。但如果你使用 typeof undeclared\_variable，它将返回 undefined。

在进一步讨论之前，先让我们理解声明和定义之间的区别。

“var x”表示一个声明，因为你没有定义它的值是什么，你只是声明它的存在。

var x; // 声明 x

console.log(x); // 输出: undefined

“var x = 1”既是声明又是定义（我们也可以说它是初始化），x 变量的声明和赋值相继发生。在 JavaScript 中，每个变量声明和函数声明都被带到了当前作用域的顶部，然后进行赋值，这个过程被称为提升（hoisting）。

当我们试图访问一个被声明但未被定义的变量时，会出现 undefined 错误。

var x; // 声明

if(typeof x === 'undefined') // 将返回 true

当我们试图引用一个既未声明也未定义的变量时，将会出现 not defined 错误。

console.log(y); // 输出: ReferenceError: y is not defined

**16. 匿名和命名函数有什么区别？**

难度星级: \*\*\*

var foo = function() { // 赋给变量 foo 的匿名函数

// ..

};

var x = function bar(){ // 赋给变量 x 的命名函数 bar

// ..

};

foo(); // 实际执行函数

x();

**17. Javascript 中的“闭包”是什么？举个例子？**

难度星级: \*\*\*\*  
闭包是在另一个函数（称为父函数）中定义的函数，并且可以访问在父函数作用域中声明和定义的变量。

闭包可以访问三个作用域中的变量：

* 在自己作用域中声明的变量；
* 在父函数中声明的变量；
* 在全局作用域中声明的变量。

var globalVar = "abc";

// 自调用函数

(function outerFunction (outerArg) { // outerFunction 作用域开始

// 在 outerFunction 函数作用域中声明的变量

var outerFuncVar = 'x';

// 闭包自调用函数

(function innerFunction (innerArg) { // innerFunction 作用域开始

// 在 innerFunction 函数作用域中声明的变量

var innerFuncVar = "y";

console.log(

"outerArg = " + outerArg + "

" +

"outerFuncVar = " + outerFuncVar + "

" +

"innerArg = " + innerArg + "

" +

"innerFuncVar = " + innerFuncVar + "

" +

"globalVar = " + globalVar);

// innerFunction 作用域结束

})(5); // 将 5 作为参数

// outerFunction 作用域结束

})(7); // 将 7 作为参数

innerFunction 是在 outerFunction 中定义的闭包，可以访问在 outerFunction 作用域内声明和定义的所有变量。除此之外，闭包还可以访问在全局命名空间中声明的变量。

上述代码的输出将是：

outerArg = 7

outerFuncVar = x

innerArg = 5

innerFuncVar = y

globalVar = abc

**18. 如何在 JavaScript 中创建私有变量？**

难度星级: \*\*\*\*  
要在 JavaScript 中创建无法被修改的私有变量，你需要将其创建为函数中的局部变量。即使这个函数被调用，也无法在函数之外访问这个变量。例如：

function func() {

var priv = "secret code";

}

console.log(priv); // throws error

要访问这个变量，需要创建一个返回私有变量的辅助函数。

function func() {

var priv = "secret code";

return function() {

return priv;

}

}

var getPriv = func();

console.log(getPriv()); // => secret code

**19. 请解释原型设计模式。**

难度星级: \*\*\*\*  
原型模式可用于创建新对象，但它创建的不是非初始化的对象，而是使用原型对象（或样本对象）的值进行初始化的对象。原型模式也称为属性模式。

原型模式在初始化业务对象时非常有用，业务对象的值与数据库中的默认值相匹配。原型对象中的默认值被复制到新创建的业务对象中。

经典的编程语言很少使用原型模式，但作为原型语言的 JavaScript 在构造新对象及其原型时使用了这个模式。

**20. “this”关键字的原理是什么？请提供一些代码示例。**

难度星级: \*\*\*\*  
在 JavaScript 中，this 是指正在执行的函数的“所有者”，或者更确切地说，指将当前函数作为方法的对象。

function foo() {

console.log( this.bar );

}

var bar = "global";

var obj1 = {

bar: "obj1",

foo: foo

};

var obj2 = {

bar: "obj2"

};

foo(); // "global"

obj1.foo(); // "obj1"

foo.call( obj2 ); // "obj2"

new foo(); // undefined

**21. 如何向 Array 对象添加自定义方法，让下面的代码可以运行？**

难度星级: \*\*\*\*

var arr = [1, 2, 3, 4, 5];

var avg = arr.average();

console.log(avg);

JavaScript 不是基于类的，但它是基于原型的语言。这意味着每个对象都链接到另一个对象（也就是对象的原型），并继承原型对象的方法。你可以跟踪每个对象的原型链，直到到达没有原型的 null 对象。我们需要通过修改 Array 原型来向全局 Array 对象添加方法。

Array.prototype.average = function() {

// 计算 sum 的值

var sum = this.reduce(function(prev, cur) { return prev + cur; });

// 将 sum 除以元素个数并返回

return sum / this.length;

}

var arr = [1, 2, 3, 4, 5];

var avg = arr.average();

console.log(avg); // => 3

**22. 什么是 JavaScript 中的提升操作？**

难度星级: \*\*\*\*  
提升（hoisting）是 JavaScript 解释器将所有变量和函数声明移动到当前作用域顶部的操作。有两种类型的提升：

* 变量提升——非常少见
* 函数提升——常见

无论 var（或函数声明）出现在作用域的什么地方，它都属于整个作用域，并且可以在该作用域内的任何地方访问它。

var a = 2;

foo(); // 因为`foo()`声明被"提升"，所以可调用

function foo() {

a = 3;

console.log( a ); // 3

var a; // 声明被"提升"到 foo() 的顶部

}

console.log( a ); // 2

**23. 以下代码输出的结果是什么？**

难度星级: \*\*\*\*\*

0.1 + 0.2 === 0.3

这段代码的输出是 false，这是由浮点数内部表示导致的。0.1 + 0.2 并不刚好等于 0.3，实际结果是 0.30000000000000004。解决这个问题的一个办法是在对小数进行算术运算时对结果进行舍入。