看完这几道 JavaScript 面试题，让你与考官对答如流（上）

原创 前端小智 大迁世界 2020-01-07 18:40

> 作者：Mark A
> 译者：前端小智
> 来源：dev

考题列表

- 1. undefined 和 null 有什么区别?
- 2. && 运算符能做什么
- 3. || 运算符能做什么
- 4. 使用 + 或一元加运算符是将字符串转换为数字的最快方法吗?
- 5. DOM 是什么?
- 6. 什么是事件传播?
- 7. 什么是事件冒泡?
- 8. 什么是事件捕获?
- 9. event.preventDefault() 和 event.stopPropagation()方法之间有什么区别?
- 10. 如何知道是否在元素中使用了event.preventDefault()方法?
- 11. 为什么此代码obj.someprop.x会引发错误?
- 12. 什么是event.target?
- 13. 什么是event.currentTarget?
- 14. == 和 === 有什么区别?
- 15. 为什么在 JS 中比较两个相似的对象时返回 false?
- 16. !! 运算符能做什么?
- 17. 如何在一行中计算多个表达式的值?
- 18. 什么是提升?
- 19. 什么是作用域?
- 20. 什么是闭包?
- 21. JavaScript中的虚值是什么?
- 22. 如何检查值是否虚值?
- 23. 'use strict' 是干嘛用的?
- 24. JavaScript中 this 值是什么?
- 25. 对象的 prototype 是什么?

1.undefined 和 null 有什么区别?

在理解 `undefined` 和 `null` 之间的差异之前，我们先来看看它们的相似类。

它们属于 JavaScript 的 7 种基本类型。

```
let primitiveTypes = ['string','number','null','undefined','boolean','symbol', 'bigint
```

它们是属于虚值，可以使用 `Boolean(value)` 或 `!!value` 将其转换为布尔值时，值为 `false` 。

```
console.log(!!null); // false
console.log(!!undefined); // false

console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false
```

接着来看看它们的区别。

`undefined` 是未指定特定值的变量的默认值，或者没有显式返回值的函数，如：`console.log(1)` ，还包括对象中不存在的属性，这些 JS 引擎都会为其分配 `undefined` 值。

```
let _thisIsUndefined;
const doNothing = () => {};
const someObj = {
  a : "ay",
  b : "bee",
  c : "si"
};

console.log(_thisIsUndefined); // undefined
console.log(doNothing()); // undefined
console.log(someObj["d"]); // undefined
```

`null` 是**“不代表任何值的值”**。 `null` 是已明确定义给变量的值。在此示例中，当 `fs.readFile` 方法未引发错误时，我们将获得 `null` 值。

```
fs.readFile('path/to/file', (e,data) => {
   console.log(e); // 当没有错误发生时，打印 null
   if(e){
     console.log(e);
   }
   console.log(data);
 });
```

在比较 `null` 和 `undefined` 时，我们使用 `==` 时得到 `true` ，使用 `===` 时得到 `false` ：

```
 console.log(null == undefined); // true
 console.log(null === undefined); // false
```

2. && 运算符能做什么

`&&` 也可以叫**逻辑与**，在其操作数中找到第一个虚值表达式并返回它，如果没有找到任何虚值表达式，则返回最后一个真值表达式。它采用短路来防止不必要的工作。

```
console.log(false && 1 && []); // false
console.log(" " && true && 5); // 5
```

使用 if 语句

```
const router: Router = Router();

router.get('/endpoint', (req: Request, res: Response) => {
   let conMobile: PoolConnection;
   try {
      //do some db operations
   } catch (e) {
   if (conMobile) {
    conMobile.release();
   }
  }
});
```

使用 && 操作符

```
const router: Router = Router();

router.get('/endpoint', (req: Request, res: Response) => {
  let conMobile: PoolConnection;
  try {
     //do some db operations
  } catch (e) {
    conMobile && conMobile.release()
  }
});
```

3. || 运算符能做什么

|| 也叫或 逻辑或 ，在其操作数中找到第一个真值表达式并返回它。这也使用了短路来防止不必要的工作。在支持 ES6 默认函数参数之前，它用于初始化函数中的默认参数值。

```
console.log(null || 1 || undefined); // 1

function logName(name) {
  var n = name || "Mark";
  console.log(n);
}

logName(); // "Mark"
```

4. 使用 + 或一元加运算符是将字符串转换为数字的最快方法吗?

根据MDN文档， + 是将字符串转换为数字的最快方法，因为如果值已经是数字，它不会执行任何操作。

5. DOM 是什么?

DOM 代表**文档对象模型**，是 HTML 和 XML 文档的接口(API)。当浏览器第一次读取(解析)HTML文档时，它会创建一个大对象，一个基于 HTM L文档的非常大的对象，这就是**DOM**。它是一个从 HTML 文档中建模的树状结构。DOM 用于交互和修改DOM结构或特定元素或节点。

假设我们有这样的 HTML 结构：

```
<!DOCTYPE html>
<html lang="en">

<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta http-equiv="X-UA-Compatible" content="ie=edge">
   <title>Document Object Model</title>
</head>

<body>
   <div>
      <p>
         <span></span>
      </p>
      <label></label>
      <input>
   </div>
</body>

</html>
```

等价的**DOM**是这样的：

JS 中的 `document` 对象表示DOM。它为我们提供了许多方法，我们可以使用这些方法来选择元素来更新元素内容，等等。

6. 什么是事件传播？

当**事件**发生在**DOM**元素上时，该**事件**并不完全发生在那个元素上。在**“冒泡阶段”**中，事件冒泡或向上传播至父级，祖父母，祖父母或父级，直到到达 `window` 为止；而在**“捕获阶**

段”中，事件从 `window` 开始向下触发元素 事件或 `event.target` 。

事件传播有三个阶段：

1. **捕获阶段**–事件从 `window` 开始，然后向下到每个元素，直到到达目标元素。
2. **目标阶段**–事件已达到目标元素。
3. **冒泡阶段**–事件从目标元素冒泡，然后上升到每个元素，直到到达 `window` 。

7. 什么是事件冒泡？

当**事件**发生在**DOM**元素上时，该**事件**并不完全发生在那个元素上。在冒泡阶段，事件冒泡，或者事件发生在它的父代，祖父母，祖父母的父代，直到到达 `window` 为止。

假设有如下的 HTML 结构：

```
<div class="grandparent">
  <div class="parent">
    <div class="child">1</div>
  </div>
</div>
```

对应的 JS 代码：

```
function addEvent(el, event, callback, isCapture = false) {
  if (!el || !event || !callback || typeof callback !== 'function') return;
  if (typeof el === 'string') {
    el = document.querySelector(el);
  };
  el.addEventListener(event, callback, isCapture);
}

addEvent(document, 'DOMContentLoaded', () => {
  const child = document.querySelector('.child');
  const parent = document.querySelector('.parent');
  const grandparent = document.querySelector('.grandparent');

  addEvent(child, 'click', function (e) {
    console.log('child');
  });

  addEvent(parent, 'click', function (e) {
    console.log('parent');
  });

  addEvent(grandparent, 'click', function (e) {
    console.log('grandparent');
  });

  addEvent(document, 'click', function (e) {
    console.log('document');
  });

  addEvent('html', 'click', function (e) {
    console.log('html');
  })

  addEvent(window, 'click', function (e) {
    console.log('window');
  })

});
```

`addEventListener` 方法具有第三个可选参数 `useCapture`，其默认值为 `false`，事件将在冒泡阶段中发生，如果为 `true`，则事件将在捕获阶段中发生。如果单击 `child` 元素，它将分别在控制台上记录 `child`，`parent`，`grandparent`，`html`，`document` 和 `window`，这就是事件冒泡。

8. 什么是事件捕获？

当事件发生在 **DOM** 元素上时，该事件并不完全发生在那个元素上。在捕获阶段，事件从 `window` 开始，一直到触发事件的元素。

假设有如下的 HTML 结构：

```
<div class="grandparent">
  <div class="parent">
    <div class="child">1</div>
  </div>
</div>
```

对应的 JS 代码：

```
function addEvent(el, event, callback, isCapture = false) {
  if (!el || !event || !callback || typeof callback !== 'function') return;
  if (typeof el === 'string') {
    el = document.querySelector(el);
  };
  el.addEventListener(event, callback, isCapture);
}

addEvent(document, 'DOMContentLoaded', () => {
  const child = document.querySelector('.child');
  const parent = document.querySelector('.parent');
  const grandparent = document.querySelector('.grandparent');

  addEvent(child, 'click', function (e) {
    console.log('child');
  });

  addEvent(parent, 'click', function (e) {
    console.log('parent');
  });

  addEvent(grandparent, 'click', function (e) {
    console.log('grandparent');
  });

  addEvent(document, 'click', function (e) {
    console.log('document');
  });

  addEvent('html', 'click', function (e) {
    console.log('html');
  })

  addEvent(window, 'click', function (e) {
    console.log('window');
  })

});
```

`addEventListener` 方法具有第三个可选参数 `useCapture`，其默认值为 `false`，事件将在冒泡阶段中发生，如果为 `true`，则事件将在捕获阶段中发生。如果单击 `child` 元素，它将分别在控制台上打印 `window`、`document`、`html`、`grandparent` 和 `parent`，这就是**事件捕获**。

9. event.preventDefault() 和 event.stopPropagation()方法之间有什么区别？

`event.preventDefault()` 方法可防止元素的默认行为。如果在表单元素中使用，它将阻止其提交。如果在锚元素中使用，它将阻止其导航。如果在上下文菜单中使用，它将

阻止其显示或显示。 `event.stopPropagation()` 方法用于阻止捕获和冒泡阶段中当前事件的进一步传播。

10. 如何知道是否在元素中使用了`event.preventDefault()`方法?

我们可以在事件对象中使用 `event.defaultPrevented` 属性。它返回一个布尔值用来表明是否在特定元素中调用了 `event.preventDefault()` 。

11. 为什么此代码 `obj.someprop.x` 会引发错误?

```
const obj = {};
console.log(obj.someprop.x);
```

显然，由于我们尝试访问 `someprop` 属性中的 `x` 属性，而 someprop 并没有在对象中，所以值为 `undefined` 。记住对象本身不存在的属性，并且其原型的默认值为 `undefined` 。因为 `undefined` 没有属性 `x` ，所以试图访问将会报错。

12. 什么是 event.target ?

简单来说， `event.target` 是发生事件的元素或触发事件的元素。

假设有如下的 HTML 结构:

```
<div onclick="clickFunc(event)" style="text-align: center;margin:15px;
border:1px solid red;border-radius:3px;">
    <div style="margin: 25px; border:1px solid royalblue;border-radius:3px;">
        <div style="margin:25px;border:1px solid skyblue;border-radius:3px;">
          <button style="margin:10px">
             Button
          </button>
        </div>
    </div>
 </div>
```

JS 代码如下:

```
function clickFunc(event) {
  console.log(event.target);
}
```

如果单击 `button` ，即使我们将事件附加在最外面的 `div` 上，它也将打印 `button` 标签，因此我们可以得出结论 `event.target` 是触发事件的元素。

13. 什么是 event.currentTarget? ?

`event.currentTarget` 是我们在其上显式附加事件处理程序的元素。

假设有如下的 HTML 结构:

```
<div onclick="clickFunc(event)" style="text-align: center;margin:15px;
border:1px solid red;border-radius:3px;">
    <div style="margin: 25px; border:1px solid royalblue;border-radius:3px;">
        <div style="margin:25px;border:1px solid skyblue;border-radius:3px;">
          <button style="margin:10px">
             Button
          </button>
        </div>
    </div>
 </div>
```

JS 代码如下：

```
function clickFunc(event) {
  console.log(event.currentTarget);
}
```

如果单击 `button`，即使我们单击该 `button`，它也会打印最外面的 `div` 标签。在此示例中，我们可以得出结论，`event.currentTarget` 是附加事件处理程序的元素。

14. == 和 === 有什么区别？

`==` 用于一般比较，`===` 用于严格比较，`==` 在比较的时候可以转换数据类型，`===` 严格比较，只要类型不匹配就返回 `flase`。

先来看看 `==` 这兄弟：

强制是将值转换为另一种类型的过程。在这种情况下，`==` 会执行隐式强制。在比较两个值之前，`==` 需要执行一些规则。

假设我们要比较 `x == y` 的值。

1. 如果 `x` 和 `y` 的类型相同，则 JS 会换成 `===` 操作符进行比较。
2. 如果 `x` 为 `null`，`y` 为 `undefined`，则返回 `true`。
3. 如果 `x` 为 `undefined` 且 `y` 为 `null`，则返回 `true`。
4. 如果 `x` 的类型是 `number`，`y` 的类型是 `string`，那么返回 `x == toNumber(y)`。
5. 如果 `x` 的类型是 `string`，`y` 的类型是 `number`，那么返回 `toNumber(x) == y`。
6. 如果 `x` 为类型是 `boolean`，则返回 `toNumber(x)== y`。
7. 如果 `y` 为类型是 `boolean`，则返回 `x == toNumber(y)`。
8. 如果 `x` 是 `string`、`symbol` 或 `number`，而 `y` 是 `object` 类型，则返回 `x == toPrimitive(y)`。
9. 如果 `x` 是 `object`，`y` 是 `string`，`symbol` 则返回 `toPrimitive(x) == y`。
10. 剩下的 返回 `false`

注意：`toPrimitive` 首先在对象中使用 `valueOf` 方法，然后使用 `toString` 方法来获取该对象的原始值。

举个例子。

x	y	x == y
5	5	true
1	'1'	true
null	undefined	true
0	false	true
'1,2'	[1,2]	true
'[object Object]'	{}	true

这些例子都返回 `true` 。

第一个示例符合 `条件1` ，因为 `x` 和 `y` 具有相同的类型和值。

第二个示例符合 `条件4` ，在比较之前将 `y` 转换为数字。

第三个例子符合 `条件2` 。

第四个例子符合 `条件7` ，因为 `y` 是 `boolean` 类型。

第五个示例符合 `条件8` 。使用 `toString()` 方法将数组转换为字符串，该方法返回 `1,2` 。

最后一个示例符合 `条件8` 。使用 `toString()` 方法将对象转换为字符串，该方法返回 `[object Object]` 。

x	y	x === y
5	5	true
1	'1'	false
null	undefined	false
0	false	false
'1,2'	[1,2]	false
'[object Object]'	{}	false

如果使用 `===` 运算符，则第一个示例以外的所有比较将返回 `false` ，因为它们的类型不同，而第一个示例将返回 `true` ，因为两者的类型和值相同。

具体更多规则可以对参考我之前的文章：

我对 JS 中相等和全等操作符转化过程一直很迷惑，直到有了这份算法

15. 为什么在 JS 中比较两个相似的对象时返回 false?

先看下面的例子：

```
let a = { a: 1 };
let b = { a: 1 };
let c = a;

console.log(a === b); // 打印 false，即使它们有相同的属性
console.log(a === c); // true
```

JS 以不同的方式比较对象和基本类型。在基本类型中，JS 通过值对它们进行比较，而在对象中，JS 通过引用或存储变量的内存中的地址对它们进行比较。这就是为什么第一个 `console.log` 语句返回 `false` ，而第二个 `console.log` 语句返回 `true` 。 `a` 和 `c` 有相同的引用地址，而 `a` 和 `b` 没有。

16. !! 运算符能做什么?

`!!` 运算符可以将右侧的值强制转换为布尔值，这也是将值转换为布尔值的一种简单方法。

```
console.log(!!null); // false
console.log(!!undefined); // false
console.log(!!''); // false
console.log(!!0); // false
console.log(!!NaN); // false
console.log(!!' '); // true
console.log(!!{}); // true
console.log(!![]); // true
console.log(!!1); // true
console.log(!![].length); // false
```

17. 如何在一行中计算多个表达式的值?

可以使用 逗号 运算符在一行中计算多个表达式。它从左到右求值，并返回右边最后一个项目或最后一个操作数的值。

```
let x = 5;

x = (x++ , x = addFive(x), x *= 2, x -= 5, x += 10);

function addFive(num) {
  return num + 5;
}
```

上面的结果最后得到 `x` 的值为 `27` 。首先，我们将 `x` 的值增加到 `6` ，然后调用函数 `addFive(6)` 并将 `6` 作为参数传递并将结果重新分配给 `x` ，此时 `x` 的值为 `11` 。之后，将 `x` 的当前值乘以 `2` 并将其分配给 `x` ， `x` 的更新值为 `22` 。然后，将 `x` 的当前值减去 `5`

并将结果分配给 x x 更新后的值为 17 。最后，我们将 x 的值增加 10 ，然后将更新的值分配给 x ，最终 x 的值为 27 。

18. 什么是提升?

提升是用来描述变量和函数移动到其(全局或函数)作用域顶部的术语。

为了理解提升，需要来了解一下**执行上下文**。**执行上下文**是当前正在执行的**“代码环境”**。执行上下文有两个阶段: 编译 和 执行 。

编译-在此阶段，JS 引荐获取所有**函数声明**并将其**提升**到其作用域的顶部，以便我们稍后可以引用它们并获取所有变量声明（使用 var 关键字进行声明），还会为它们提供默认值： undefined 。

执行——在这个阶段中，它将值赋给之前提升的变量，并执行或调用函数(对象中的方法)。

注意:只有使用 var 声明的变量，或者函数声明才会被提升，相反，函数表达式或箭头函数， let 和 const 声明的变量，这些都不会被提升。

假设在全局使用域，有如下的代码:

```
console.log(y);
y = 1;
console.log(y);
console.log(greet("Mark"));

function greet(name){
  return 'Hello ' + name + '!';
}

var y;
```

上面分别打印： undefined , 1 , Hello Mark! 。

上面代码在编译阶段其实是这样的:

```
function greet(name) {
  return 'Hello ' + name + '!';
}

var y; // 默认值 undefined

// 等待“编译”阶段完成，然后开始“执行”阶段

/*
console.log(y);
y = 1;
console.log(y);
console.log(greet("Mark"));
*/
```

编译阶段完成后，它将启动执行阶段调用方法，并将值分配给变量。

```
function greet(name) {
  return 'Hello ' + name + '!';
}

var y;

//start "execution" phase

console.log(y);
y = 1;
console.log(y);
console.log(greet("Mark"));
```

19. 什么是作用域？

JavaScript 中的作用域是我们可以有效访问变量或函数的区域。JS 有三种类型的作用域：**全局作用域**、**函数作用域**和**块作用域(ES6)**。

- **全局作用域**——在全局命名空间中声明的变量或函数位于全局作用域中，因此在代码中的任何地方都可以访问它们。

```
//global namespace
var g = "global";

function globalFunc(){
  function innerFunc(){
    console.log(g); // can access "g" because "g" is a global variable
  }
 innerFunc();
}
```

- **函数作用域**——在函数中声明的变量、函数和参数可以在函数内部访问，但不能在函数外部访问。

```
function myFavoriteFunc(a) {
  if (true) {
    var b = "Hello " + a;
  }
  return b;
}

myFavoriteFunc("World");

console.log(a); // Throws a ReferenceError "a" is not defined
console.log(b); // does not continue here
```

- **块作用域**-在块 `{}` 中声明的变量（ `let, const` ）只能在其中访问。

```
function testBlock(){
  if(true){
    let z = 5;
  }
  return z;
}

testBlock(); // Throws a ReferenceError "z" is not defined
```

作用域也是一组用于查找变量的规则。如果变量在当前作用域中不存在，它将向外部作用域中查找并搜索，如果该变量不存在，它将再次查找直到到达全局作用域，如果找到，则可以使用它，否则引发错误，这种查找过程也称为**作用域链**。

```
/* 作用域链
```

```
   内部作用域->外部作用域-> 全局作用域
*/

// 全局作用域
var variable1 = "Comrades";
var variable2 = "Sayonara";

function outer(){
// 外部作用域
  var variable1 = "World";
  function inner(){
  // 内部作用域
    var variable2 = "Hello";
    console.log(variable2 + " " + variable1);
  }
  inner();
}
outer(); // Hello World
```

20. 什么是闭包？

这可能是所有问题中最难的一个问题，因为闭包是一个有争议的话题，这里从个人角度来谈谈，如果不妥，多多海涵。

闭包就是一个函数在声明时能够记住当前作用域、父函数作用域、及父函数作用域上的变量和参数的引用，直至通过作用域链上全局作用域，基本上闭包是在声明函数时创建的作用域。

看看小例子：

```
// 全局作用域
var globalVar = "abc";

function a(){
  console.log(globalVar);
}

a(); // "abc"
```

在此示例中，当我们声明 `a` 函数时，全局作用域是 `a` 闭包的一部分。

变量 `globalVar` 在图中没有值的原因是该变量的值可以根据调用函数 `a` 的位置和时间而改变。但是在上面的示例中， `globalVar` 变量的值为 `abc` 。

来看一个更复杂的例子：

```
var globalVar = "global";
var outerVar = "outer"

function outerFunc(outerParam) {
  function innerFunc(innerParam) {
    console.log(globalVar, outerParam, innerParam);
  }
  return innerFunc;
}

const x = outerFunc(outerVar);
outerVar = "outer-2";
globalVar = "guess"
x("inner");
```

上面打印结果是 `guess outer inner` 。

当我们调用 `outerFunc` 函数并将返回值 `innerFunc` 函数分配给变量 `x` 时，即使我们为 `outerVar` 变量分配了新值 `outer-2` ， `outerParam` 也继续保留 `outer` 值，因为重新分配是在调用 `outerFunc` 之后发生的，并且当我们调用 `outerFunc` 函数时，它会在作用域链中查找 `outerVar` 的值，此时的 `outerVar` 的值将为 `"outer"` 。

现在，当我们调用引用了 `innerFunc` 的 `x` 变量时， `innerParam` 将具有一个 `inner` 值，因为这是我们在调用中传递的值，而 `globalVar` 变量值为 `guess` ，因为在调用 `x` 变量之前，我们将一个新值分配给 `globalVar` 。

下面这个示例演示没有理解好闭包所犯的错误：

```
const arrFuncs = [];
for(var i = 0; i < 5; i++){
  arrFuncs.push(function (){
    return i;
  });
}
console.log(i); // i is 5

for (let i = 0; i < arrFuncs.length; i++) {
  console.log(arrFuncs[i]()); // 都打印 5
}
```

由于闭包，此代码无法正常运行。 `var` 关键字创建一个全局变量，当我们 `push` 一个函数时，这里返回的全局变量 `i` 。因此，当我们在循环后在该数组中调用其中一个函数时，它会打印 `5` ，因为我们得到 `i` 的当前值为 `5` ，我们可以访问它，因为它是全局变量。

因为闭包在创建变量时会保留该变量的引用而不是其值。我们可以使用**IIFES**或使用 `let` 来代替 `var` 的声明。

21. JavaScript 中的虚值是什么?

```
const falsyValues = ['', 0, null, undefined, NaN, false];
```

简单的来说虚值就是是在转换为布尔值时变为 `false` 的值。

22. 如何检查值是否虚值?

使用 `Boolean` 函数或者 `!!` 运算符。

23. 'use strict' 是干嘛用的?

`"use strict"` 是 **ES5** 特性，它使我们的代码在函数或整个脚本中处于**严格模式**。**严格模式**帮助我们在代码的早期避免 bug，并为其添加限制。

严格模式的一些限制：

1. 变量必须声明后再使用
2. 函数的参数不能有同名属性，否则报错
3. 不能使用 `with` 语句
4. 不能对只读属性赋值，否则报错
5. 不能使用前缀 0 表示八进制数，否则报错
6. 不能删除不可删除的属性，否则报错
7. 不能删除变量 `delete prop`，会报错，只能删除属性 `delete global[prop]`
8. `eval` 不能在它的外层作用域引入变量
9. `eval` 和 `arguments` 不能被重新赋值
10. `arguments` 不会自动反映函数参数的变化
11. 不能使用 `arguments.callee`
12. 不能使用 `arguments.caller`
13. 禁止 `this` 指向全局对象
14. 不能使用 `fn.caller` 和 `fn.arguments` 获取函数调用的堆栈
15. 增加了保留字（比如 `protected`、`static` 和 `interface`）

设立"严格模式"的目的，主要有以下几个：

1. 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;
2. 消除代码运行的一些不安全之处，保证代码运行的安全;
3. 提高编译器效率，增加运行速度;
4. 为未来新版本的Javascript做好铺垫。

24. JavaScript 中 `this` 值是什么?

基本上，`this` 指的是当前正在执行或调用该函数的对象的值。`this` 值的变化取决于我们使用它的上下文和我们在哪里使用它。

```
const carDetails = {
  name: "Ford Mustang",
  yearBought: 2005,
```

```
  getName(){
    return this.name;
  },
  isRegistered: true
};

console.log(carDetails.getName()); // Ford Mustang
```

这通常是我们期望结果的，因为在 `getName` 方法中我们返回 `this.name` ，在此上下文中， `this` 指向的是 `carDetails` 对象，该对象当前是执行函数的“所有者”对象。

接下我们做些奇怪的事情：

```
var name = "Ford Ranger";
var getCarName = carDetails.getName;

console.log(getCarName()); // Ford Ranger
```

上面打印 `Ford Ranger` ，这很奇怪，因为在第一个 `console.log` 语句中打印的是 `Ford Mustang` 。这样做的原因是 `getCarName` 方法有一个不同的“所有者”对象，即 `window` 对象。在全局作用域中使用 `var` 关键字声明变量会在 `window` 对象中附加与变量名称相同的属性。请记住，当没有使用 `“use strict”` 时，在全局作用域中 `this` 指的是 `window` 对象。

```
console.log(getCarName === window.getCarName); // true
console.log(getCarName === this.getCarName); // true
```

本例中的 `this` 和 `window` 引用同一个对象。

解决这个问题的一种方法是在函数中使用 `apply` 和 `call` 方法。

```
console.log(getCarName.apply(carDetails)); // Ford Mustang
console.log(getCarName.call(carDetails));  // Ford Mustang
```

`apply` 和 `call` 方法期望第一个参数是一个对象，该对象是函数内部 `this` 的值。

`IIFE` 或**立即执行的函数表达式**，在全局作用域内声明的函数，对象内部方法中的匿名函数和内部函数的 `this` 具有默认值，该值指向 `window` 对象。

```
(function (){
  console.log(this);
})(); // 打印 "window" 对象

function iHateThis(){
   console.log(this);
}

iHateThis(); // 打印 "window" 对象

const myFavoriteObj = {
  guessThis(){
     function getName(){
       console.log(this.name);
     }
     getName();
```

```
    },
    name: 'Marko Polo',
    thisIsAnnoying(callback){
      callback();
    }
  };


  myFavoriteObj.guessThis(); // 打印 "window" 对象
  myFavoriteObj.thisIsAnnoying(function (){
    console.log(this); // 打印 "window" 对象
  });
```

如果我们要获取 `myFavoriteObj` 对象中的 `name` 属性（即**Marko Polo**）的值，则有两种方法可以解决此问题。

一种是将 `this` 值保存在变量中。

```
const myFavoriteObj = {
 guessThis(){
  const self = this; // 把 this 值保存在 self 变量中
  function getName(){
    console.log(self.name);
  }
  getName();
 },
 name: 'Marko Polo',
 thisIsAnnoying(callback){
   callback();
  }
};
```

第二种方式是使用箭头函数

```
const myFavoriteObj = {
  guessThis(){
     const getName = () => {
       console.log(this.name);
     }
     getName();
  },
  name: 'Marko Polo',
  thisIsAnnoying(callback){
   callback();
  }
};
```

箭头函数没有自己的 `this` 。它复制了这个封闭的词法作用域中 `this` 值，在这个例子中， `this` 值在 `getName` 内部函数之外，也就是 `myFavoriteObj` 对象。

25. 对象的 prototype(原型) 是什么?

简单地说，原型就是对象的蓝图。如果它存在当前对象中，则将其用作属性和方法的回退。它是在对象之间共享属性和功能的方法，这也是JavaScript实现继承的核心。

```
const o = {};
console.log(o.toString()); // logs [object Object]
```

即使 `o` 对象中不存在 `o.toString` 方法，它也不会引发错误，而是返回字符串 `[object Object]` 。当对象中不存在属性时，它将查看其原型，如果仍然不存在，则将其查找到原型的原型，依此类推，直到在原型链中找到具有相同属性的属性为止。原型链的末尾是 `Object.prototype` 。

```
console.log(o.toString === Object.prototype.toString); // logs true
```

由于篇幅过长，我将此系列分成上中下三篇，下篇我们在见。

原文：

https://dev.to/macmacky/70-javascript-interview-questions-5gfi#1-whats-the-difference-between-undefined-and-null

---

交流

面试 10

面试 · 目录

下一篇 · 看完这几道 JavaScript 面试题，让你与考官对答如流（中）

喜欢此内容的人还喜欢

90后程序员辞职搞灰产，一年获利超700万，结局很刑！
大迁世界

100 个鲜为人知的 CSS 技巧汇总整理合集
大迁世界

最近前端爆了？？？
大迁世界