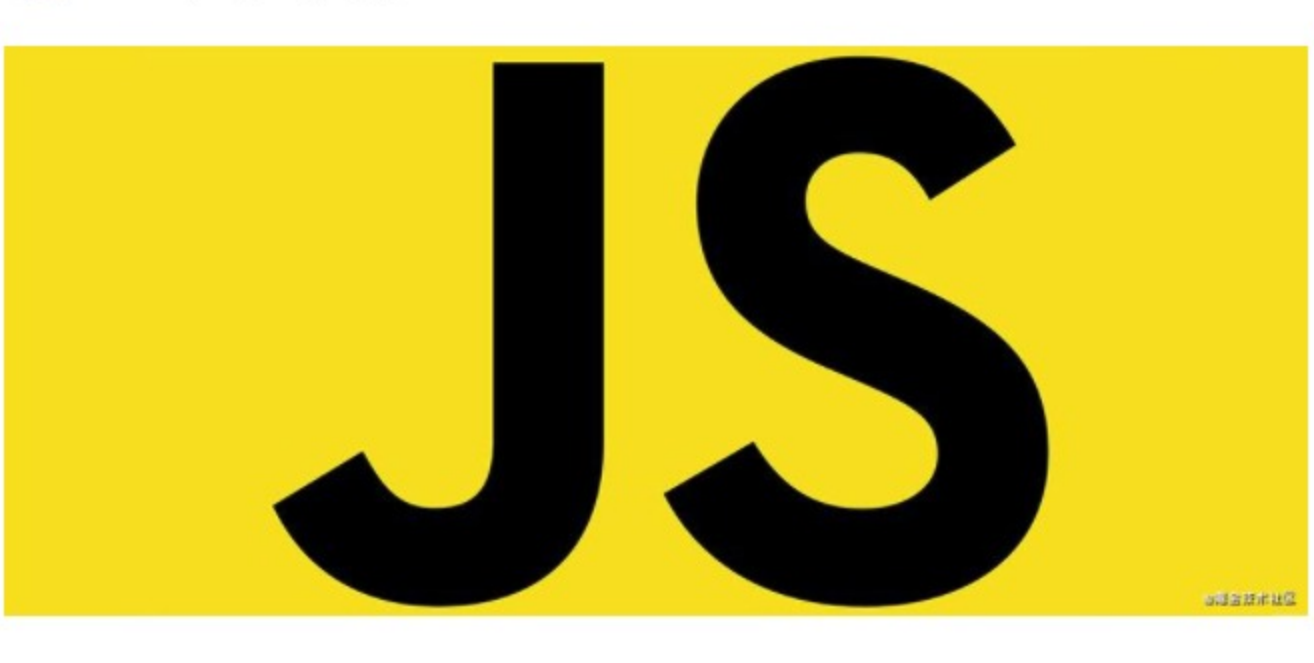
random_ Lv2
2021年03月23日 阅读 30
关注



前端面试系列-JavaScript-this指向问题(附面试例题)

一、对this的产生原因分析和了解

> this指的是函数运行时所在的环境（即调用的对象）。
> JavaScript 语言之所以有this的设计，跟内存里面的数据结构有关系

对于普通对象的保存：实际对象属性的值就是值；
对于函数的保存：实际函数属性的值是函数的地址；（而函数本身，可理解为：它不属于任何一个对象，相当于一个全局对象。所以，函数在不同场景下运行，this就是不同的场景了，不过都是执行时的环境）
同时，函数，可以作为一个参数（值）被调用，被传播。同时，在函数体内部，允许引用当前环境的其他变量（可以引用当前环境的其他变量。但当前环境是不确定的，所以会出现不同的值）

当函数被作为某个对象调用时，this等于那个对象，
匿名函数具有全局性，this对象通常指向window。

二、this绑定详解

this 实际上是在函数被调用时发生的绑定，它指向什么地方完全取决于函数在哪里被「调用」。

「注意」：箭头函数并没有自己的this，被「定义」在哪里，this就指向谁。

1 默认绑定（绑定在全局作用域）

即在调用函数时，函数不带任何修饰，也就是"光秃秃"的调用（即没有任何修饰的调用），就会应用默认绑定规则，默认绑定指向全局作用域。

2 隐式绑定

当函数在调用时，如果函数有所谓的"落脚点"，即有上下文对象（即调用时.前面的对象）时，隐式绑定规则会把函数中的 this 绑定到这个上下文对象。

在非严格模式下this为undefined时会返回window，严格模式下 this 返回undefined。

3 显式绑定：apply,call,bind

三种显式绑定方法：

> fun.apply(thisArg, [argsArray])
> func.call(thisArg, arg1, arg2, ...)
> func.bind(thisArg[, arg1[, arg2[, ...]]])

「注意」：bind返回一个新函数，这个函数已经制定了执行上下文（这表示执行上下文在之后不可改变了），而返回这个新函数可以接受参数（即在绑定的执行上下文中，以参数为入参执行函数）。

4 new 绑定

通过 new 操作符调用构造函数时发生的 this 绑定。

实际上：new 和 bind应该很类似。只是一个在创建时执行，一个在执行过程中执行。（不过还是很不同的，见下文区别）

5 bind 和 new 的区别

bind只能被函数调用。而new 返回的是一个对象。即new 之后，函数的执行上下文就不可以被改变了，bind不行，显示绑定的三种形式都不行。所以new 绑定 的优先级最高。

6 四种绑定的优先级和区别

默认绑定 < 隐式绑定 < 显式绑定 < new 绑定

7.几种调用场景

```
var obj = {
        a: 1,
        b: function () {
                console.log(this);
        }
}
```

> 作为对象调用时，指向该对象 obj.b(); // 指向obj
> 作为函数调用, var b = obj.b; b(); // 指向全局window
> 作为构造函数调用 var b = new Fun(); // this指向当前实例对象
> 作为call与apply调用 obj.b.apply(object, []); // this指向当前的object

再来个例子：

```
function say() {
    console.log(this.name)
}
var name = "global"
var obj = {
    name: "inside",
    say: say
}
obj.say()//作为对象调用时，this指向该对象，输出"inside"
var alias = obj.say;
alias() //作为函数调用,this指向全局window，输出"global"
```

三、当 this 碰到 return

如果返回值是一个对象，那么this指向的就是那个返回的对象；
如果返回值不是一个对象那么this还是指向函数的实例。（null是特例，虽然null是对象，但是返回时指向的还是函数的实例。）

即：函数返回对象时，那么this指向的是这个返回对象的运行环境； 函数返回不是对象时，那么this指向的就是这个函数本身。

四、例题实战

```
function foo() {
    console.log(this.bar);
}
var bar = "bar1";
var o2 = {bar: "bar2", foo: foo};
var o3 = {bar: "bar3", foo: foo};

foo();//bar1
o2.foo();//bar2
foo.call(o3); //bar3
```

这三者分别是默认绑定、隐式绑定和显式绑定。

```
var name = 'Nicolas';
function Person(){
    this.name = 'Smiley';
    this.sayName=function(){
        console.log(this);
        console.log(this.name);
    };
    setTimeout(this.sayName, 0);     // 第二次输出:window,Nicolas
}

var person = new Person();
person.sayName();              // 第一次输出:Person ,Smiley
```

第一次输出的是Person, Smiley;对象调用场景 第二次输出的结果是window, Nicolas。尽管setTimeout是在构造函数中定义的，但是调用的时候，是在window中调用。SetTimeout等许多之后被触发的事件当中，一定要注意this的指向，这是基于调用点（call stack）的 。

```
function Person() {
  this.name = "Smiley";
  this.sayName = function(){
    console.log(this);
    console.log(this.name);
  };
}

let person = new Person();
let sayNameCopy = person.sayName;
sayNameCopy();//window,Nicolas
```

函数调用场景

```
function Person() {
  this.name = "Smiley";
  this.sayName = ()=> {
    console.log(this);
    console.log(this.name);
  };
}

let person = new Person();
person.sayName.call({name: "Nicolas"});
```

我们只改动了一处：把sayName改为箭头函数。结果则变成了Person和"Smiley"。这是因为箭头函数并没有自己的this，被定义在哪里，this就指向谁，且优先级比显式调用高，因此，this仍指向Person。

五、拓展-从ECMAScript规范解读this

看了一篇大佬的文章感觉很棒： JavaScript深入之从ECMAScript规范解读this

从ECMAScript规范解读this,从底层原理上讲了this指向。

感兴趣的可以看一下。

一些概念

GetValue，GetBase以及Reference类型等文章这部分提到的概念都属于浏览器底层的实现，只是为了从原理上来解读this指向问题的

1.Reference

Reference 是一个 Specification Type，也就是 "只存在于规范里的抽象类型"。它们是为了更好地描述语言的底层行为逻辑才存在的，但并不存在于实际的 js 代码中。
Reference 由三个组成部分，分别是：

> base value
> referenced name
> strict reference

我们简单的理解：

base value 就是属性所在的对象或者就是 EnvironmentRecord(环境数据)，它的值只可能是 undefined, an Object, a Boolean, a String, a Number, or an environment record 其中的一种。 referenced name 就是属性的名称。

例子：

```
var foo = 1;

// 对应的Reference是:
var fooReference = {
    base: EnvironmentRecord,
    name: 'foo',
    strict: false
};
```

再举个例子：

```
var foo = {
    bar: function () {
        return this;
    }
};

foo.bar(); // foo

// bar对应的Reference是:
var BarReference = {
    base: foo,
    propertyName: 'bar',
    strict: false
};
```

规范中还提供了一些方法，比如 「GetBase」、「IsPropertyReference」和「GetValue:」。

2.GetBase

返回 reference 的 base value。

3.IsPropertyReference

用于判断，如果 base value 是一个对象，就返回true。

4.GetValue

用于从 Reference 类型获取对应值的方法。

简单模拟 GetValue 的使用：

```
var foo = 1;

var fooReference = {
    base: EnvironmentRecord,
    name: 'foo',
    strict: false
};

GetValue(fooReference) // 1;
```

GetValue 返回对象属性真正的值，而不再是一个 Reference

5.MemberExpression (复杂表达式)

> PrimaryExpression // 原始表达式 可以参见《JavaScript权威指南第四章》
> FunctionExpression // 函数定义表达式
> MemberExpression [Expression] // 属性访问表达式
> MemberExpression . IdentifierName // 属性访问表达式
> new MemberExpression Arguments // 对象创建表达式

如何确定this的值

> 计算 MemberExpression(复杂表达式)的结果赋值给 ref
>
> 判断 ref 是不是一个 Reference 类型
>
> > 如果 ref 是 Reference，并且 IsPropertyReference(ref) 是 true, 那么 this 的值为 GetBase(ref)
> > 如果 ref 是 Reference，并且 base value 值是 Environment Record, 那么this的值为 ImplicitThisValue(ref)
> > 如果 ref 不是 Reference，那么 this 的值为 undefined

1.计算 MemberExpression的结果赋值给 ref

举个例子：

```
function foo() {
    console.log(this)
}

foo(); // MemberExpression 是 foo

function foo() {
    return function() {
        console.log(this)
    }
}

foo()(); // MemberExpression 是 foo()

var foo = {
    bar: function () {
        return this;
    }
}

foo.bar(); // MemberExpression 是 foo.bar
```

所以简单理解 MemberExpression 其实就是()左边的部分。

2.判断 ref 是不是一个 Reference 类型

关键就在于看规范是如何处理各种 MemberExpression，返回的结果是不是一个Reference类型。 判断 ref 是不是一个 Reference 类型

> 如果 ref 是 Reference，并且 IsPropertyReference(ref) 是 true, 那么 this 的值为 GetBase(ref)
> 如果 ref 是 Reference，并且 base value 值是 Environment Record, 那么this的值为 ImplicitThisValue(ref)
> 如果 ref 不是 Reference，那么 this 的值为 undefined

根据例子来分析

例子：

```
var value = 1;

var foo = {
  value: 2,
  bar: function () {
    return this.value;
  }
}

//示例1
console.log(foo.bar()); // 2
//示例2
console.log((foo.bar)()); // 2
//示例3
console.log((foo.bar = foo.bar)()); // 1
//示例4
console.log((false || foo.bar)()); // 1
//示例5
console.log((foo.bar, foo.bar)()); // 1
```

「示例1」

```
console.log(foo.bar());//2
```

第一步，计算 MemberExpression的结果赋值给 ref： MemberExpression 计算的结果是 foo.barx 第二步，判断 ref 是不是一个 Reference 类型 根据之前的内容，我们知道该值为：

```
var Reference = {
  base: foo,
  name: 'bar',
  strict: false
};
```

接下来按照 2.1 的判断流程走：

2.1 如果 ref 是 Reference，并且 IsPropertyReference(ref) 是 true，那么 this 的值为 GetBase(ref) 这里base value 为 foo，是一个对象，所以 IsPropertyReference(ref) 结果为 true。 所以this = GetBase(ref) = foo 所以示例1的结果就是 2

「示例2」

```
console.log((foo.bar)());//2
```

foo.bar 被 () 包住，实际上 () 并没有对 MemberExpression 进行计算， 所以其实跟示例 1 的结果是一样的。

「示例3」

```
console.log((foo.bar = foo.bar)());//1
```

有赋值操作符，查看规范 11.13.1 Simple Assignment (=):

```
计算的第三步:
3.Let rval be GetValue(rref).
```

因为使用了 GetValue，所以返回的值不是 Reference 类型，

按照之前讲的判断逻辑：

2.3 如果 ref 不是Reference，那么 this 的值为 undefined

this 为 undefined，非严格模式下，this 的值为 undefined 的时候，其值会被隐式转换为全局对象。 所以输出为1

「示例4」

```
console.log((false || foo.bar)());//1
```

逻辑与算法，查看规范 11.11 Binary Logical Operators:

```
计算第二步:
2.Let lval be GetValue(lref).
```

因为使用了 GetValue，所以返回的不是 Reference 类型，this 为 undefined

「示例5」

```
console.log((foo.bar, foo.bar)());//1
```

逗号操作符，查看规范11.14 Comma Operator (,)

```
计算第二步:
2.Call GetValue(lref).
```

因为使用了 GetValue，所以返回的不是 Reference 类型，this 为 undefined

「综上，有赋值操作符，逻辑与算法，逗号操作符时，都会使用GetValue，所以返回的不是 Reference 类型，this 为 undefined，非严格模式下，this指向全局对象。」

文章分类 前端 文章标签 JavaScript

目录