前端面试之彻底搞懂this指向

原创 coderwhy coderwhy 2020-04-23 15:07

收录于话题
#前端面试精讲

10个

> this是JavaScript中的一个关键字，但是又一个相对比较特别的关键字，不像function、var、for、if这些关键字一样，可以很清楚的搞清楚它到底是如何使用的。
>
> this会在执行上下文中绑定一个对象，但是是根据什么条件绑定的呢？在不同的执行条件下会绑定不同的对象，这也是让人捉摸不定的地方。
>
> 这一次，我们一起来彻底搞定this到底是如何绑定的吧！

一. 理解this

1.1. 为什么使用this

在常见的编程语言中，几乎都有this这个关键字（Objective-C中使用的是self），但是JavaScript中的this和常见的面向对象语言中的this不太一样：

- 常见面向对象的编程语言中，比如Java、C++、Swift、Dart等等一系列语言中，this通常只会出现在 类的方法 中。
- 也就是你需要有一个类，类中的方法（特别是实例方法）中，this代表的是当前调用对象。
- 但是JavaScript中的this更加灵活，无论是它出现的位置还是它代表的含义。

使用this有什么意义呢？下面的代码中，我们通过对象字面量创建出来一个对象，当我们调用对象的方法时，希望将对象的名称一起进行打印。

如果没有this，那么我们的代码会是下面的写法：

- 在方法中，为了能够获取到name名称，必须通过obj的引用（变量名称）来获取。
- 但是这样做有一个很大的弊端：如果我将obj的名称换成了info，那么所有的方法中的obj都需要换成info。

```
var obj = {
  name: "why"
```

```
  name: "why",
  running: function() {
    console.log(obj.name + " running");
  },
  eating: function() {
    console.log(obj.name + " eating");
  },
  studying: function() {
    console.log(obj.name + " studying");
  }
}
```

事实上，上面的代码，在实际开发中，我们都会使用this来进行优化：

- 当我们通过obj去调用running、eating、studying这些方法时，this就是指向的obj对象

```
var obj = {
  name: "why",
  running: function() {
    console.log(this.name + " running");
  },
  eating: function() {
    console.log(this.name + " eating");
  },
  studying: function() {
    console.log(this.name + " studying");
  }
}
```

所以我们会发现，在某些函数或者方法的编写中，this可以让我们更加便捷的方式来引用对象，在进行一些API设计时，代码更加的简洁和易于复用。

当然，上面只是应用this的一个场景而已，开发中使用到this的场景到处都是，这也是为什么它不容易理解的原因。

1.2. this指向什么

我们先说一个最简单的，this在全局作用域下指向什么？

- 这个问题非常容易回答，在浏览器中测试就是指向window
- 所以，在全局作用域下，我们可以认为this就是指向的window

```
console.log(this); // window
```

```
var name = "why";
console.log(this.name); // why
console.log(window.name); // why
```

但是，开发中很少直接在全局作用域下去使用this，通常都是在**函数中使用**。

所有的函数在被调用时，都会创建一个执行上下文：

- 这个上下文中记录着函数的调用栈、函数的调用方式、传入的参数信息等；
- this也是其中的一个属性；

我们先来看一个让人困惑的问题：

- 定义一个函数，我们采用三种不同的方式对它进行调用，它产生了三种不同的结果

```
// 定义一个函数
function foo() {
  console.log(this);
}

// 1.调用方式一：直接调用
foo(); // window

// 2.调用方式二：将foo放到一个对象中,再调用
var obj = {
  name: "why",
  foo: foo
}
obj.foo() // obj对象

// 3.调用方式三：通过call/apply调用
foo.call("abc"); // String {"abc"}对象
```

上面的案例可以给我们什么样的启示呢?

- 1.函数在调用时，JavaScript会默认给this绑定一个值；
- 2.this的绑定和定义的位置（编写的位置）没有关系；
- 3.this的绑定和调用方式以及调用的位置有关系；
- 4.this是在运行时被绑定的；

那么this到底是怎么样的绑定规则呢？一起来学习一下吧

二. this绑定规则

> 我们现在已经知道this无非就是在函数调用时被绑定的一个对象，我们就需要知道它在不同的场景下的绑定规则即可。

2.1. 默认绑定

什么情况下使用默认绑定呢？独立函数调用。

- 独立的函数调用我们可以理解成函数没有被绑定到某个对象上进行调用；

案例一：普通函数调用

- 该函数直接被调用，并没有进行任何的对象关联；
- 这种独立的函数调用会使用默认绑定，通常默认绑定时，函数中的this指向全局对象（window）；

```
function foo() {
  console.log(this); // window
}

foo();
```

案例二：函数调用链（一个函数又调用另外一个函数）

- 所有的函数调用都没有被绑定到某个对象上；

```
// 2.案例二:
function test1() {
  console.log(this); // window
  test2();
}

function test2() {
  console.log(this); // window
  test3()
}

function test3() {
  console.log(this); // window
```

```
}
test1();
```

案例三：将函数作为参数，传入到另一个函数中

```
function foo(func) {
  func()
}

function bar() {
  console.log(this); // window
}

foo(bar);
```

我们对案例进行一些修改，考虑一下打印结果是否会发生变化：

- 这里的结果依然是window，为什么呢?
- 原因非常简单，在真正函数调用的位置，并没有进行任何的对象绑定，只是一个独立函数的调用；

```
function foo(func) {
  func()
}

var obj = {
  name: "why",
  bar: function() {
    console.log(this); // window
  }
}

foo(obj.bar);
```

2.2. 隐式绑定

另外一种比较常见的调用方式是通过某个对象进行调用的：

- 也就是它的调用位置中，是通过某个对象发起的函数调用。

案例一：通过对象调用函数

- foo的调用位置是obj.foo()方式进行调用的

- 那么foo调用时this会隐式的被绑定到obj对象上

```
function foo() {
  console.log(this); // obj对象
}

var obj = {
  name: "why",
  foo: foo
}

obj.foo();
```

案例二：案例一的变化

- 我们通过obj2又引用了obj1对象，再通过obj1对象调用foo函数；
- 那么foo调用的位置上其实还是obj1被绑定了this；

```
function foo() {
  console.log(this); // obj对象
}

var obj1 = {
  name: "obj1",
  foo: foo
}

var obj2 = {
  name: "obj2",
  obj1: obj1
}

obj2.obj1.foo();
```

案例三：隐式丢失

- 结果最终是window，为什么是window呢?
- 因为foo最终被调用的位置是bar，而bar在进行调用时没有绑定任何的对象，也就没有形成隐式绑定；
- 相当于是一种默认绑定；

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

```
}

var obj1 = {
  name: "obj1",
  foo: foo
}

// 讲obj1的foo赋值给bar
var bar = obj1.foo;
bar();
```

2.3. 显示绑定

隐式绑定有一个前提条件：

- 必须在调用的 对象内部 有一个对函数的引用（比如一个属性）；
- 如果没有这样的引用，在进行调用时，会报找不到该函数的错误；
- 正是通过这个引用，间接的将this绑定到了这个对象上；

如果我们不希望在 **对象内部** 包含这个函数的引用，同时又希望在这个对象上进行强制调用，该怎么做呢?

- JavaScript所有的函数都可以使用call和apply方法（这个和Prototype有关）。
  - 它们两个的区别这里不再展开；
  - 其实非常简单，第一个参数是相同的，后面的参数，apply为数组，call为参数列表；
- 这两个函数的第一个参数都要求是一个对象，这个对象的作用是什么呢？就是给this准备的。
- 在调用这个函数时，会将this绑定到这个传入的对象上。

因为上面的过程，我们明确的绑定了this指向的对象，所以称之为 **显示绑定**。

2.3.1. call、apply

通过call或者apply绑定this对象

- 显示绑定后，this就会明确的指向绑定的对象

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

foo.call(window); // window
```

```
foo.call(window); // window
foo.call({name: "why"}); // {name: "why"}
foo.call(123); // Number对象,存放时123
```

2.3.2. bind函数

如果我们希望一个函数总是显示的绑定到一个对象上，可以怎么做呢?

方案一：自己手写一个辅助函数（了解）

- 我们手动写了一个bind的辅助函数
- 这个辅助函数的目的是在执行foo时，总是让它的this绑定到obj对象上

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

var obj = {
  name: "why"
}

function bind(func, obj) {
  return function() {
    return func.apply(obj, arguments);
  }
}

var bar = bind(foo, obj);

bar(); // obj对象
bar(); // obj对象
bar(); // obj对象
```

方案二：使用Function.prototype.bind

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

var obj = {
  name: "why"
}
```

```
var bar = foo.bind(obj);

bar(); // obj对象
bar(); // obj对象
bar(); // obj对象
```

2.3.3. 内置函数

有些时候，我们会调用一些JavaScript的内置函数，或者一些第三方库中的内置函数。

- 这些内置函数会要求我们传入另外一个函数；
- 我们自己并不会显示的调用这些函数，而且JavaScript内部或者第三方库内部会帮助我们执行；
- 这些函数中的this又是如何绑定的呢?

案例一：setTimeout

- setTimeout中会传入一个函数，这个函数中的this通常是window

```
setTimeout(function() {
  console.log(this); // window
}, 1000);
```

为什么这里是window呢?

- 这个和setTimeout源码的内部调用有关；
- setTimeout内部是通过apply进行绑定的this对象，并且绑定的是全局对象；

案例二：数组的forEach

数组有一个高阶函数forEach，用于函数的遍历：

- 在forEach中传入的函数打印的也是Window对象；
- 这是因为默认情况下传入的函数是自动调用函数（默认绑定）；

```
var names = ["abc", "cba", "nba"];
names.forEach(function(item) {
  console.log(this); // 三次window
});
```

我们是否可以改变该函数的this指向呢?

forEach参数

```
var names = ["abc", "cba", "nba"];
var obj = {name: "why"};
names.forEach(function(item) {
  console.log(this); // 三次obj对象
}, obj);
```

案例三：div的点击

如果我们有一个div元素：

- 注意：省略了部分代码

```
<style>
  .box {
    width: 200px;
    height: 200px;
    background-color: red;
  }
</style>

<div class="box"></div>
```

获取元素节点，并且监听点击：

- 在点击事件的回调中，this指向谁呢？box对象；
- 这是因为在发生点击时，执行传入的回调函数被调用时，会将box对象绑定到该函数中；

```
var box = document.querySelector(".box");
box.onclick = function() {
```

```
  console.log(this); // box对象
}
```

所以传入到内置函数的回调函数this如何确定呢?

- 某些内置的函数，我们很难确定它内部是如何调用传入的回调函数；
- 一方面可以通过分析源码来确定，另一方面我们可以通过经验（见多识广）来确定；
- 但是无论如何，通常都是我们之前讲过的规则来确定的；

2.4. new绑定

JavaScript中的函数可以当做一个类的构造函数来使用，也就是使用new关键字。

使用new关键字来调用函数时，会执行如下的操作：

- 1.创建一个全新的对象；
- 2.这个新对象会被执行Prototype连接；
- 3.这个新对象会绑定到函数调用的this上（this的绑定在这个步骤完成）；
- 4.如果函数没有返回其他对象，表达式会返回这个新对象；

```
// 创建Person
function Person(name) {
  console.log(this); // Person {}
  this.name = name; // Person {name: "why"}
}

var p = new Person("why");
console.log(p);
```

2.5. 规则优先级

学习了四条规则，接下来开发中我们只需要去查找函数的调用应用了哪条规则即可，但是如果一个函数调用位置应用了多条规则，优先级谁更高呢?

1.默认规则的优先级最低

毫无疑问，默认规则的优先级是最低的，因为存在其他规则时，就会通过其他规则的方式来绑定this

2.显示绑定优先级高于隐式绑定

显示绑定和隐式绑定哪一个优先级更高呢？这个我们可以测试一下：

- 结果是obj2，说明是显示绑定生效了

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

var obj1 = {
  name: "obj1",
  foo: foo
}

var obj2 = {
  name: "obj2",
  foo: foo
}

// 隐式绑定
obj1.foo(); // obj1
obj2.foo(); // obj2

// 隐式绑定和显示绑定同时存在
obj1.foo.call(obj2); // obj2, 说明显式绑定优先级更高
```

3.new绑定优先级高于隐式绑定

- 结果是foo，说明是new绑定生效了

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

var obj = {
  name: "why",
  foo: foo
}

new obj.foo(); // foo对象, 说明new绑定优先级更高
```

4.new绑定优先级高于bind

new绑定和call、apply是不允许同时使用的，所以不存在谁的优先级更高

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

var obj = {
  name: "obj"
}

var foo = new foo.call(obj);
```


new和call同时使用

但是new绑定是否可以和bind后的函数同时使用呢？可以

- 结果显示为foo，那么说明是new绑定生效了

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

var obj = {
  name: "obj"
}

// var foo = new foo.call(obj);
var bar = foo.bind(obj);
var foo = new bar(); // 打印foo，说明使用的是new绑定
```

优先级总结：

- new绑定 > 显示绑定（bind） > 隐式绑定 > 默认绑定

三. this规则之外

> 我们讲到的规则已经足以应付平时的开发，但是总有一些语法，超出了我们的规则之外。（神话故事和动漫中总是有类似这样的人物）

3.1. 忽略显示绑定

如果在显示绑定中，我们传入一个null或者undefined，那么这个显示绑定会被忽略，使用默认规则：

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

var obj = {
  name: "why"
}

foo.call(obj); // obj对象
foo.call(null); // window
foo.call(undefined); // window

var bar = foo.bind(null);
bar(); // window
```

3.2. 间接函数引用

另外一种情况，创建一个函数的 间接引用 ，这种情况使用默认绑定规则。

我们先来看下面的案例结果是什么?

- (num2 = num1)的结果是num1的值；

```
var num1 = 100;
var num2 = 0;
var result = (num2 = num1);
console.log(result); // 100
```

我们来下面的函数赋值结果：

- 赋值(obj2.foo = obj1.foo)的结果是foo函数；
- foo函数被直接调用，那么是默认绑定；

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

var obj1 = {
```

```
  name: "obj1",
  foo: foo
};

var obj2 = {
  name: "obj2"
}

obj1.foo(); // obj1对象
(obj2.foo = obj1.foo)();  // window
```

3.3. ES6箭头函数

在ES6中新增一个非常好用的函数类型：箭头函数

- 这里不再具体介绍箭头函数的用法，可以自行学习。

箭头函数不使用this的四种标准规则（也就是不绑定this），而是根据外层作用域来决定this。

我们来看一个模拟网络请求的案例：

- 这里我使用setTimeout来模拟网络请求，请求到数据后如何可以存放到data中呢?
- 我们需要拿到obj对象，设置data；
- 但是直接拿到的this是window，我们需要在外层定义： `var _this = this`
- 在setTimeout的回调函数中使用_this就代表了obj对象

```
var obj = {
  data: [],
  getData: function() {
    var _this = this;
    setTimeout(function() {
      // 模拟获取到的数据
      var res = ["abc", "cba", "nba"];
      _this.data.push(...res);
    }, 1000);
  }
}

obj.getData();
```

上面的代码在ES6之前是我们最常用的方式，从ES6开始，我们会使用箭头函数：

- 为什么在setTimeout的回调函数中可以直接使用this呢?
- 因为箭头函数并不绑定this对象，那么this引用就会从上层作用域中找到对应的this

```
var obj = {
  data: [],
  getData: function() {
    setTimeout(() => {
      // 模拟获取到的数据
      var res = ["abc", "cba", "nba"];
      this.data.push(...res);
    }, 1000);
  }
}

obj.getData();
```

思考：如果getData也是一个箭头函数，那么setTimeout中的回调函数中的this指向谁呢?

- 答案是window；
- 依然是不断的从上层作用域找，那么找到了全局作用域；
- 在全局作用域内，this代表的就是window

```
var obj = {
  data: [],
  getData: () => {
    setTimeout(() => {
      console.log(this); // window
    }, 1000);
  }
}

obj.getData();
```

四. this面试题

4.1. 面试题一：

```
var name = "window";
var person = {
  name: "person",
```

```
  sayName: function () {
    console.log(this.name);
  }
};
function sayName() {
  var sss = person.sayName;
  sss();
  person.sayName();
  (person.sayName)();
  (b = person.sayName)();
}
sayName();
```

这道面试题非常简单，无非就是绕一下，希望把面试者绕晕：

```
function sayName() {
  var sss = person.sayName;
  // 独立函数调用，没有和任何对象关联
  sss(); // window
  // 关联
  person.sayName(); // person
  (person.sayName)(); // person
  (b = person.sayName)(); // window
}
```

4.2. 面试题二：

```
var name = 'window'
var person1 = {
  name: 'person1',
  foo1: function () {
    console.log(this.name)
  },
  foo2: () => console.log(this.name),
  foo3: function () {
    return function () {
      console.log(this.name)
    }
  },
  foo4: function () {
    return () => {
```

```
      console.log(this.name)
    }
  }
}

var person2 = { name: 'person2' }

person1.foo1();
person1.foo1.call(person2);

person1.foo2();
person1.foo2.call(person2);

person1.foo3()();
person1.foo3.call(person2)();
person1.foo3().call(person2);

person1.foo4()();
person1.foo4.call(person2)();
person1.foo4().call(person2);
```

下面是代码解析：

```
// 隐式绑定，肯定是person1
person1.foo1(); // person1
// 隐式绑定和显示绑定的结合，显示绑定生效，所以是person2
person1.foo1.call(person2); // person2

// foo2()是一个箭头函数，不适用所有的规则
person1.foo2() // window
// foo2依然是箭头函数，不适用于显示绑定的规则
person1.foo2.call(person2) // window

// 获取到foo3，但是调用位置是全局作用于下，所以是默认绑定window
person1.foo3()() // window
// foo3显示绑定到person2中
// 但是拿到的返回函数依然是在全局下调用，所以依然是window
person1.foo3.call(person2)() // window
// 拿到foo3返回的函数，通过显示绑定到person2中，所以是person2
person1.foo3().call(person2) // person2

// foo4()的函数返回的是一个箭头函数
// 箭头函数的执行找上层作用域，是person1
person1.foo4()() // person1
// foo4()显示绑定到person2中，并且返回一个箭头函数
```

```
// 箭头函数找上层作用域，是person2
person1.foo4.call(person2)() // person2
// foo4返回的是箭头函数，箭头函数只看上层作用域
person1.foo4().call(person2) // person1
```

4.3. 面试题三:

```
var name = 'window'
function Person (name) {
  this.name = name
  this.foo1 = function () {
    console.log(this.name)
  },
  this.foo2 = () => console.log(this.name),
  this.foo3 = function () {
    return function () {
      console.log(this.name)
    }
  },
  this.foo4 = function () {
    return () => {
      console.log(this.name)
    }
  }
}
var person1 = new Person('person1')
var person2 = new Person('person2')

person1.foo1()
person1.foo1.call(person2)

person1.foo2()
person1.foo2.call(person2)

person1.foo3()()
person1.foo3.call(person2)()
person1.foo3().call(person2)

person1.foo4()()
person1.foo4.call(person2)()
person1.foo4().call(person2)
```

下面是代码解析：

```
// 隐式绑定
person1.foo1() // peron1
// 显示绑定优先级大于隐式绑定
person1.foo1.call(person2) // person2

// foo是一个箭头函数，会找上层作用域中的this，那么就是person1
person1.foo2() // person1
// foo是一个箭头函数，使用call调用不会影响this的绑定，和上面一样向上层查找
person1.foo2.call(person2) // person1

// 调用位置是全局直接调用，所以依然是window（默认绑定）
person1.foo3()() // window
// 最终还是拿到了foo3返回的函数，在全局直接调用（默认绑定）
person1.foo3.call(person2)() // window
// 拿到foo3返回的函数后，通过call绑定到person2中进行了调用
person1.foo3().call(person2) // person2

// foo4返回了箭头函数，和自身绑定没有关系，上层找到person1
person1.foo4()() // person1
// foo4调用时绑定了person2，返回的函数是箭头函数，调用时，找到了上层绑定的person2
person1.foo4.call(person2)() // person2
// foo4调用返回的箭头函数，和call调用没有关系，找到上层的person1
person1.foo4().call(person2) // person1
```

4.4. 面试题四：

```
var name = 'window'
function Person (name) {
  this.name = name
  this.obj = {
    name: 'obj',
    foo1: function () {
      return function () {
        console.log(this.name)
      }
    },
    foo2: function () {
      return () => {
        console.log(this.name)
      }
```

```
    }
  }
}
var person1 = new Person('person1')
var person2 = new Person('person2')

person1.obj.foo1()()
person1.obj.foo1.call(person2)()
person1.obj.foo1().call(person2)

person1.obj.foo2()()
person1.obj.foo2.call(person2)()
person1.obj.foo2().call(person2)
```

下面是代码解析：

```
// obj.foo1()返回一个函数
// 这个函数在全局作用于下直接执行（默认绑定）
person1.obj.foo1()() // window
// 最终还是拿到一个返回的函数（虽然多了一步call的绑定）
// 这个函数在全局作用于下直接执行（默认绑定）
person1.obj.foo1.call(person2)() // window
person1.obj.foo1().call(person2) // person2

// 拿到foo2()的返回值，是一个箭头函数
// 箭头函数在执行时找上层作用域下的this，就是obj
person1.obj.foo2()() // obj
// foo2()的返回值，依然是箭头函数，但是在执行foo2时绑定了person2
// 箭头函数在执行时找上层作用域下的this，找到的是person2
person1.obj.foo2.call(person2)() // person2
// foo2()的返回值，依然是箭头函数
// 箭头函数通过call调用是不会绑定this，所以找上层作用域下的this是obj
person1.obj.foo2().call(person2) // obj
```

文章已于2021/05/27修改