Js 中奇妙的this值

在 `JS` 中最常见的莫过于函数了，在函数(方法)中 `this` 的出现频率特别高，那么 `this` 到底是什么呢，今天就和大家一起学习总结一下 `JS` 中的 `this`。

1. 初探this

`this` 在 `JS` 中是一个关键字，不是变量也不是属性名， `JS` 中不允许给this赋值。

它是函数运行时，在函数体内部自动生成的一个对象，只能在函数体内部使用。

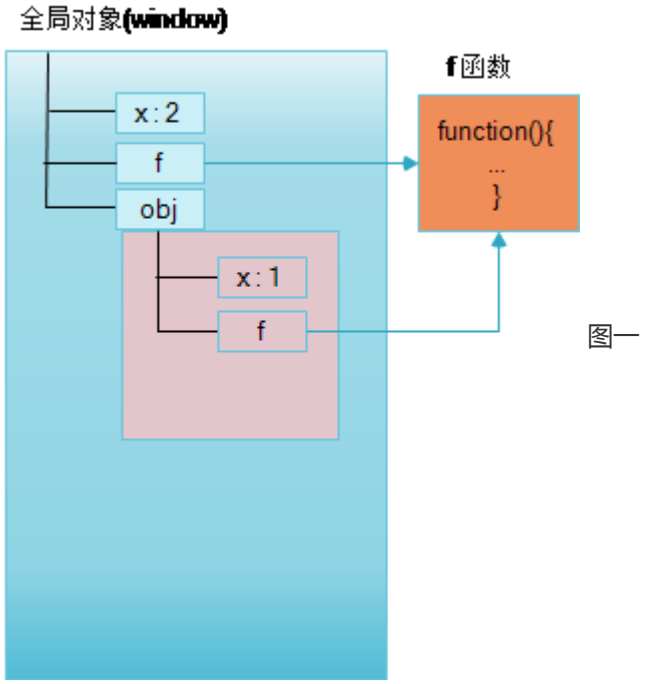
`this` 指向的是函数运行时所在的环境，也就是说函数在哪个环境中运行，`this` 的值就指向哪个环境。 先看下面这段代码的输出结果：

```
function f() {
    console.log(this.x);
}
var obj = {
    f: f,
    x: 1
};
var x = 2;

f(); // 2
obj.f(); // 1
```

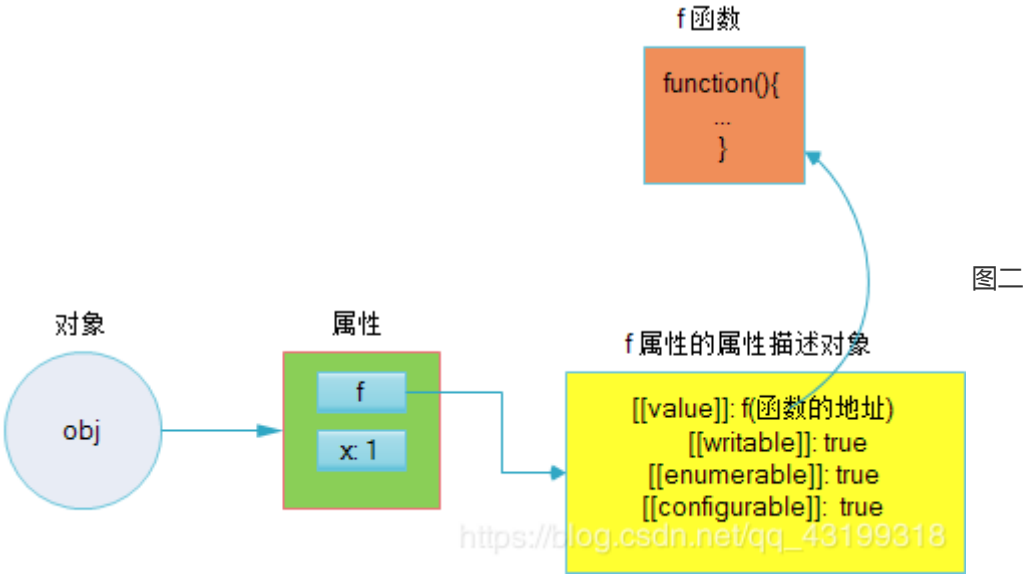
有点奇怪，`obj.f` 和 `f` 明明指向的是同一个函数为什么执行结果是不同的呢？

原因就在于这两个函数运行时所在的环境是不同的。

可以结合下面的两张图来理解 图一描述了上面这段代码的作用域链 图二描述了运行 `obj.f()` 时的部分执行过程



图一



图二

如图二所示，执行`obj.f()` 时，`obj` 对象需要先找到 `f` 属性，然后通过 `f` 属性中的 `value` 值获取到 `f` 函数的地址，通过这个地址再获取到 `f` 函数实际的代码开始运行，因此此时 `f` 函数运行时所在的环境是 `obj` 环境。因为 `obj` 环境下 `x` 的值是 `1` ，所以最终输出的值为 `1`。

执行 `f()` 时，实际上是从全局对象 `window` 中找到 `f` 函数，然后再执行。此时 `f` 函数运行时所在的环境是全局环境，因为全局环境下的 `x` 的值为 `2` ，因此最终输出的值为 `2`。

下面是另外一个值得注意的地方：

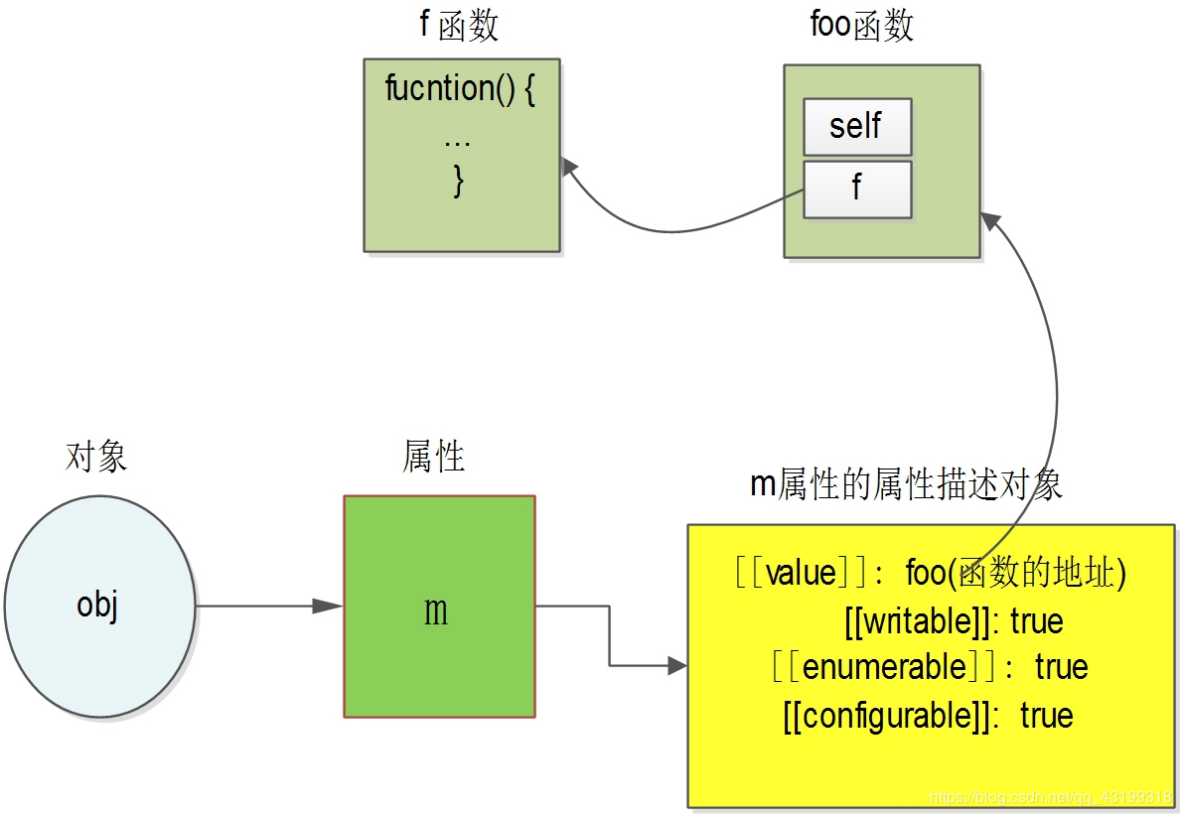
`this` 值没有作用域的限制，嵌套函数不会从它的包含函数中继承 `this`，**很多人误以为调用嵌套函数时 `this` 值会指向它的外层函数的变量对象**，其实并不是这样的。 如果想访问这个外层函数的 `this` 值，需要将 `this` 值保存在一个变量里，通常使用 `self` 来保存`this`。

再看下面这段代码：

```
let foo = function() {
    var self = this;
    console.log(this === obj); // true, this就是obj对象
    f(); // 嵌套函数f当做普通函数调用

```

```
    function f() {
    // 上面f()是被当做普通函数调用的，执行环境是全局作用域，因此f内部的this的值指向全局对象window
        console.log(this === obj) // false,this在这里指向全局对象
        // self保存的是外部方法中的this，指向对象obj
        console.log(self === obj) // true，  self中保存的是外层函数中的this值
        }
    };

var obj = {
    m: foo
};

obj.m();
```

下面这张图描述了执行 `obj.m()` 时内部运行的部分流程：



图三

执行`obj.m()` 时，obj对象需要先找到 `m` 属性，然后通过读取 `m` 属性中的 `value` 值来调用 `foo` 函数，所以此时 `foo` 函数运行时所在的环境是 `obj` 环境，所以 `foo` 内部的 `this` 指向 `obj` 环境，所以第一个 `console.log` 的输出结果为 `true` 。

在 `foo` 函数内部调用 `f` 时，直接写成了 `f()` 这种普通函数调用的方式，记住当被当做普通函数调用时，`f` 内部的 `this` 在是指向全局环境的。(严格模式下是 `undefined` 非严格模式下指向全局环境，一般情况下都是用的非严格模式)。 因此，`f` 函数内部的 `this` 是全局对象 `window` 而不是 `obj` ，这也说明了内层函数不会继承外部函数的 `this` 。

所以，第二个 `console.log` 会输出 `false` ，因为此时 `f` 内部的 `this` 指向全局对象 `window` 。第三个 `console.log` 会输出 `true` ，因为 `self` 里存放的是外层函数的 `this` ，外层函数的 `this` 指向 `obj` 环境。

看到这里可能有的小伙伴还是对于 `this` 的值到底是什么还是有一点疑惑，能不能再归纳一下呢？好，那接下来就根据不同的情况再做一下总结，其实这个总结是之前看的阮一峰老师归纳的，在这里加上一点自己的理解，拿过来借花献佛。

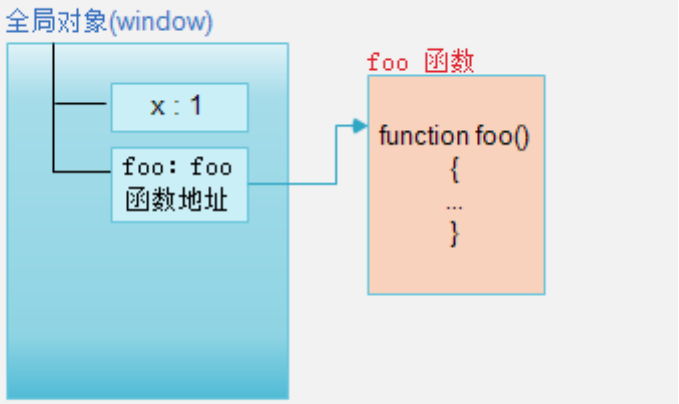
2. this指向总结

再重申一下，`this` 是在函数运行时，自动生成的一个对象，`this` 的指向不同，归根结底在于函数调用方式的不同，下面就以四种不同的函数调用方式来分析 `this` 的指向问题。

2.1 普通函数调用

如果一个函数被当做普通函数调用，在非严格模式下这个函数中的 `this` 值就指向全局对象 `window`，在严格模式下 `this` 值就是 `undefined`。 下面结合代码和配图来说明一下：

```
var x = 1;
function foo() {
    console.log(this.x);
}
foo();
```



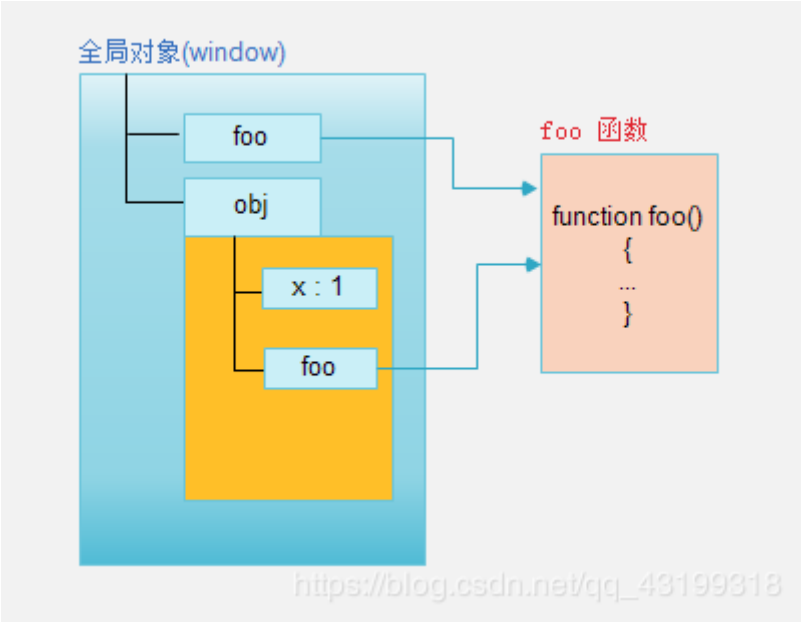
图四

运行 `foo()` 时 `foo` 是被当做普通函数调用，`window` 对象需要先找到 `foo` 属性，然后通过里面保存的地址找到 `foo` 函数的代码开始运行，因此 `foo` 函数的运行环境是 `window` 环境，此时 `this` 的值指向 `window` 环境。因为 `window` 环境中 `x` 属性的值为 `1` ，因此最终的输出结果为 `1`。

2.2 对象的方法调用

当某个函数被某个对象当做方法来调用时，`this` 就指向这个对象。

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

var obj = {
    x : 1,
    foo : foo
}

obj.foo();
```



图五

运行 `obj.foo()` 时 `foo` 函数被当做 `obj` 对象的方法来调用，此时 `foo` 函数的运行环境是 `obj` 环境，因此 `this` 指向 `obj` ，因为 `obj.x = 1` ， 所以最终输出 `1`。

2.3 构造函数调用

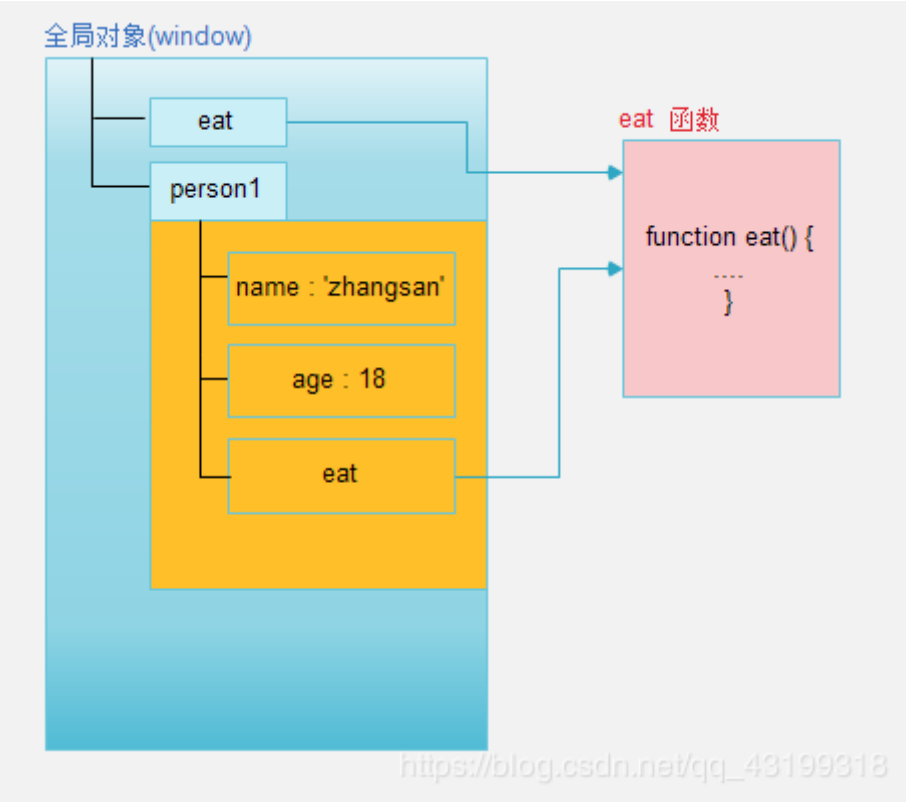
使用 `new 构造函数` 的语法会创建一个新的对象，此时 `this` 就指向这个新的对象。

要想明白其中的原理，就要从 `new` 操作符说起， 使用 `new` 操作符时实际上 `JS` 引擎做了四件事：

1. 创建一个新对象(创建 `person1` 对象)
2. 将构造函数的环境赋给新对象(`this` **指向了** `person1`)
3. 执行构造函数中的代码(为 `person1` 对象添加属性和方法，即 `name` ，`age` 属性，`eat` 方法)
4. 返回这个新对象(将新创建的对象的地址赋给 `person1`)

> 注：上面的1,2,3步中不应该出现 `person1` ，因为最后一步才将新创建的对象的地址赋给 `person1`，上面那样写是为了理解方便。

```
function eat() {
    console.log('I am eating');
}

function Person(name, age) {
    this.name = name;
    this.age = age;
    this.eat = eat;
}

let person1 = new Person('zhangsan', '18');
console.log(person1.name); // 'zhangsan'
console.log(person1.age); // 18
person1.eat(); // 'I am eating'
```



图六

通过 `new` 操作符的第二步，我们就可以看出 `Js` 引擎将构造函数的环境赋给了新的对象(`person1`)，因此 `this` 就指向了那个新创建的对象(`person1`)。

2.4 利用call，apply，bind方法调用函数

这几个都是函数的方法，它们可以改变函数运行时的环境， `this` 就指向它们的参数所指定的运行环境。

```
var obj1 = {
    x : 1
};

var obj2 = {
    x : 2
};

var obj3 = {
    x : 3
};

var x = 4;

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

var foo1 = foo.bind(obj1);
foo1(); // 1
foo.call(obj2); // 2
foo.apply(obj3); // 3
foo(); // 4
```

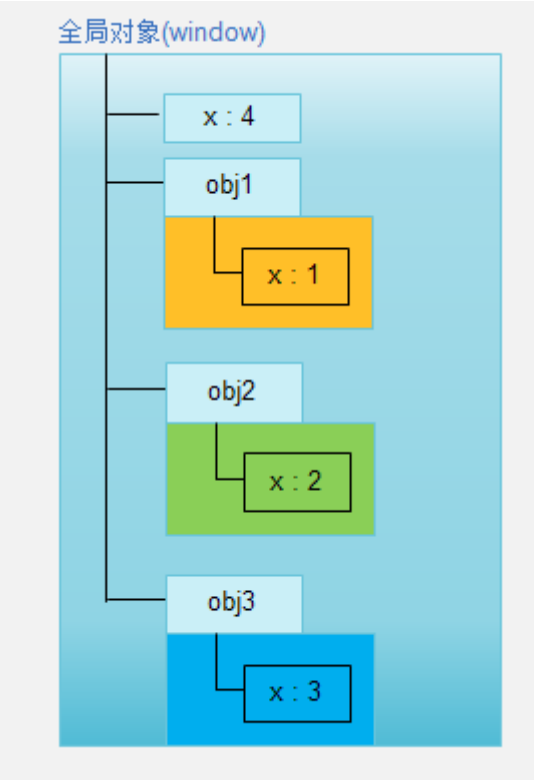


图 七

`var foo1 = foo.bind(obj1); foo1();` 将函数运行的环境修改为 `obj1` ，`this` 指向 `obj1`，因此输出 `1`。 `foo.call(obj2);` 将函数的运行环境修改为 `obj2`，`this` 指向 `obj2`，因此输出为 `2`。 `foo.apply(obj3)` 将函数的运行环境修改为`obj3`，`this` 指向 `obj3`，因此输出为 `3`。 `foo()` 纯粹的函数调用，运行环境为 全局对象window， `this` 指向 `obj4`，因此输出为 `4`。

完 如有不恰当之处，欢迎指正哦.