this关键字是javascript最复杂的机制之一，如果对this没有一个清晰的认识，this对开发者来说简直是一种魔法。

**为什么要用this**

看一段代码：

function a(context) {

return context.name.toUpperCse()

}

var you = {

name: ‘xixi’

}

a(you) // XIXI

如果不用this的话，对于函数a，我们需要传入you对象作为上下文对象，这种显式传递上下文的方式，会随着你的使用模式越来越复杂，让代码变得越来越混乱。简单的说，就是函数的上下文完全需要开发者去手动传参配置，并不能做到函数自动应用合适的上下文对象。

**this是什么？**

对于this的认知，开发者很容易有以下两种错误认知。

错误认知一：

有开发者认为，this就是函数本身。看代码：

function foo(num) {

console.log(“foo”, num)

this.count++

}

foo.count = 0

for(var i = 0; i < 10; i++){

if(i > 5) {

foo(i)

}

}

//foo:6

//foo:7

//foo:8

//foo:9

console.log(foo.count)//0

解析：

在上述例子中，foo函数被调用了四次，那么foo函数内的 this.count++也被执行了四次，如果this指向函数本身的话，那么打印console.log(foo.count)时，应该打印出4，然而运行结果是0，所以函数中的this不等于其自身。

错误认知二：

this指向函数的作用域。

function foo(){

var a = 2

this.bar()

}

function bar() {

console.log(this.a)

}

foo()//referenceError:a is not defined

解析：

这段代码运行直接抛错了，证明this指向作用域是不对的。改例子试图使用this来联通foo和bar函数的作用域，从而让bar可以访问foo中作用域的变量a。这样做是不可能实现的，因为使用this并不能再作用域中查到到什么。**每当你想把this和词法作用域的查找混合使用时，你要告诉自己这是无法实现的**

那么this到底是什么呢？根据执行上下文的知识，我们知道当一个函数被调用的时候，会创建一个活动记录，，这个记录会包含函数在哪里被调用、函数的调用方式、传入参数等信息。而**this就是这个记录的一个属性，会在函数执行的过程中用到，指向调用函数的对象。**

**this的绑定规则**

1. **默认绑定**

在开发中，我们最常用的函数调用方式是：独立函数调用。比如：

function foo() {

console.log(this.a)

}

var a = 2

foo()

解析：

我们可以看到this.a被解析为了全局变量的a。为什么？因为foo函数的调用位置是在全局执行上下文中调用的，并且不带任何修饰的函数调用，只能是默认绑定。由于默认绑定，foo（）等同于window.foo（），那么结合我们之前说过的this**指向调用函数的对象，this.a其实就是window.a，也就是2。**

如果使用严格模式，则全局对象不能用于默认绑定，那么this会绑定到undefined。

function foo() {

“use strict”

console.log(this.a)

}

var a = 2

foo()// typeError:this is undefined

值的注意的是，函数内部使用严格模式才会影响到默认绑定，但是在严格模式下调用函数是不影响的：

function foo() {

console.log(this.a)

}

var a = 2

（function（）{

“use strict”

foo()// 2

}）（）

理论上，并不推荐混合使用严格和非严格模式，但是有时候用到第三方库，其严格程度会与你自身写的代码不同。这种严格模式下调用函数是不影响的特性，会使得第三方库与你代码互相兼容。

1. **隐式绑定**

调用函数的时候，存在上下文对象。

function foo() {

console.log(this.a)

}

var obj = {

a:2,

foo:foo

}

obj.foo() //2

解析：

当foo函数被调用时，前面加上了obj的引用。当函数引用有上下文对象时，隐式绑定规则会把函数调用中的this绑定到这个上下文对象。因此调用obj.foo（）时，foo函数中的this.a指向obj.a。

需注意的是，对象属性引用只在最后一层的调用中起作用，什么意思？看代码：

function foo() {

console.log(this.a)

}

var obj2 = {

a:42,

foo:foo

}

var obj1 = {

a:2,

obj2 :obj2

}

obj1.obj2.foo() //42

解析：

obj1.obj2.foo()调用之后，foo函数的this指向了最后一层调用obj2，因此this.a也就是obj2.a 等于42

值得注意的是，隐式绑定容易造成隐式丢失，比如：

function foo() {

console.log(this.a)

}

var obj = {

a:2,

foo:foo

}

var bar = obj.foo

var a = ‘i am all’

bar()// i am all

解析：

bar = obj.foo实际上是把foo函数的内存地址赋值给了bar，bar（）函数调用其实可以看做foo（）调用，而不是obj.foo（），因此bar（）实际上是一个不带修饰的的函数调用，应用的是默认绑定方式。

1. **显示绑定**

隐式绑定中this是意想不到的，那么有没有方法能固定this指向呢？那肯定有的啊，就是使用call和apply这两个方法。

怎么使用这两个方法呢？它们的第一个参数是一个对象，用于this，在调用函数时，把该对象绑定到this。这种方式，区别就是call后面参数都作为函数的入参，而apply的第二个参数为一个数组，数组元素为函数的入参。使用app和call方式的this绑定称为显式绑定。

function foo() {

console.log(this.a)

}

var obj = {

a:2

}

foo.call(obj)//2

解析：

由于call的原因，foo函数内部的this指向了传入的obj。

**硬绑定**

在上述例子中，由于foo中this还是动态变化的，无法解决之前的丢失绑定问题，因此我们有一种方案，就是返回一个绑定了this的函数，这种方案叫硬绑定，显式硬绑定

function foo() {

console.log(this.a)

}

var obj = {

a:2

}

var bar = function() {

foo.call(obj)

}

bar() // 2

bar.call(window)//2

解析：

函数bar内部手动调用了foo.call(obj)，强制把foo的this绑定到obj。无论之后如何调用bar函数，它总是会手动在obj上调用foo，这就是硬绑定。

由于硬绑定是一种非常常用的模式，ES5提供了内置方法bind，看代码：

function foo(something) {

console.log(this.a，something)

return this.a +something

}

var obj = {

a:2

}

var bar = foo.bind(obj)

var b = bar(3) //2 3

console.log(b) //5

解析：

var bar = foo.bind(obj)调用之后返回一个 内部this绑定到了obj的函数，bar（3）则是以3为入参调用该函数。

实际上，在javascript中很多内置函数都提供了一个可选参数，来确保你回调函数使用指定的this，比如 ：

function(el){

console.log(el,this.id)

}

var obj = {

id: “abc”

}

[1,2,3].forEach(foo,obj)// 1 abc 2 abc 3 abc

1. **new绑定**

在JS中，构造函数只是使用new操作符时被调用的普通函数，他们不属于某个类，也不会实例化一个类。包括内置对象函数（比如Number(..)）在内的所有函数都可以用new来调用，这种函数调用被称为构造函数调用。

实际上并不存在所谓的“构造函数”，只有对于函数的“构造调用”。

使用new来调用函数，或者说发生构造函数调用时，会自动执行下面的操作：

1、创建（或者说构造）一个新对象。

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。

4、如果函数没有返回其他对象，那么new表达式中的函数调用会自动返回这个新对象。

使用new来调用foo(..)时，会构造一个新对象并把它（bar）绑定到foo(..)调用中的this。

function foo(a) {

this.a = a;

}

var bar = new foo(2); // bar和foo(..)调用中的this进行绑定

console.log( bar.a ); // 2

手写一个new实现

function create() {

// 创建一个空的对象

var obj = new Object(),

// 获得构造函数，arguments中去除第一个参数

Con = [].shift.call(arguments);

// 链接到原型，obj 可以访问到构造函数原型中的属性

obj.\_\_proto\_\_ = Con.prototype;

// 绑定 this 实现继承，obj 可以访问到构造函数中的属性

var ret = Con.apply(obj, arguments);

// 优先返回构造函数返回的对象

return ret instanceof Object ? ret : obj;

};

使用这个手写的new

function Person() {...}

// 使用内置函数new

var person = new Person(...)

// 使用手写的new，即create

var person = create(Person, ...)

代码原理解析：

1、用new Object()的方式新建了一个对象obj

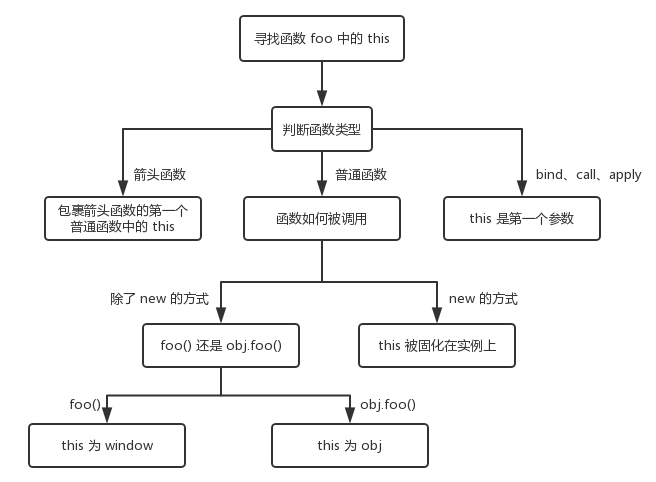
2、取出第一个参数，就是我们要传入的构造函数。此外因为 shift 会修改原数组，所以 arguments会被去除第一个参数

3、将 obj的原型指向构造函数，这样obj就可以访问到构造函数原型中的属性

4、使用apply，改变构造函数this 的指向到新建的对象，这样 obj就可以访问到构造函数中的属性

1. 返回 obj
2. **绑定优先级**

首先，new 的方式优先级最高，接下来是 bind 这些函数，然后是 obj.foo() 这种调用方式，最后是 foo 这种调用方式，如果你还是觉得有点绕，那么就看以下的这张流程图吧



1. **绑定例外**

**被忽略的this**

call、apply或者bind传入的绑定this对象为null或者undefined时，会导致这些值在调用时会被忽略，实际应用是默认绑定规则。

有两种会传入null的应用场景：

1. 使用apply(..)来“展开”一个数组，并当作参数传入一个函数
2. bind(..)可以对参数进行柯里化（预先设置一些参数）

function foo(a, b) {

console.log( "a:" + a + "，b:" + b );

}

// 把数组”展开“成参数

foo.apply( null, [2, 3] ); // a:2，b:3

// 使用bind(..)进行柯里化

var bar = foo.bind( null, 2 );

bar( 3 ); // a:2，b:3

总是传入null来忽略this绑定可能产生一些副作用：默认绑定规则会把this绑定到全局对象中。

更安全的做法就是传入一个特殊的对象（空对象），把this绑定到这个对象不会对你的程序产生任何副作用。

JS中创建一个空对象最简单的方法是Object.create(null)，这个和{}很像，但是并不会创建Object.prototype这个委托，所以比{}更空。

function foo(a, b) {

console.log( "a:" + a + "，b:" + b );

}

// 我们的空对象

var ø = Object.create( null );

// 把数组”展开“成参数

foo.apply( ø, [2, 3] ); // a:2，b:3

// 使用bind(..)进行柯里化

var bar = foo.bind( ø, 2 );

bar( 3 ); // a:2，b:3

**软绑定**

硬绑定可以把this强制绑定到指定的对象（new除外），防止函数调用应用默认绑定规则。但是会降低函数的灵活性，使用硬绑定之后就无法使用隐式绑定或者显式绑定来修改this。

如果给默认绑定指定一个全局对象和undefined以外的值，那就可以实现和硬绑定相同的效果，同时保留隐式绑定或者显示绑定修改this的能力。

// 默认绑定规则，优先级排最后

// 如果this绑定到全局对象或者undefined，那就把指定的默认对象obj绑定到this,否则不会修改this

if(!Function.prototype.softBind) {

Function.prototype.softBind = function(obj) {

var fn = this;

// 捕获所有curried参数

var curried = [].slice.call( arguments, 1 );

var bound = function() {

return fn.apply(

(!this || this === (window || global)) ?

obj : this,

curried.concat.apply( curried, arguments )

);

};

bound.prototype = Object.create( fn.prototype );

return bound;

};

}

使用：软绑定版本的foo()可以手动将this绑定到obj2或者obj3上，但如果应用默认绑定，则会将this绑定到obj。

function foo() {

console.log("name:" + this.name);

}

var obj = { name: "obj" },

obj2 = { name: "obj2" },

obj3 = { name: "obj3" };

// 默认绑定，应用软绑定，软绑定把this绑定到默认对象obj

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

// 隐式绑定规则

obj2.foo = foo.softBind( obj );

obj2.foo(); // name: obj2 <---- 看！！！

// 显式绑定规则

fooOBJ.call( obj3 ); // name: obj3 <---- 看！！！

// 绑定丢失，应用软绑定

setTimeout( obj2.foo, 10 ); // name: obj

1. **箭头函数中的this绑定**

5 this词法

ES6新增一种特殊函数类型：箭头函数，箭头函数无法使用上述四条规则，而是根据外层（函数或者全局）作用域（词法作用域）来决定this。

foo()内部创建的箭头函数会捕获调用时foo()的this。由于foo()的this绑定到obj1，bar(引用箭头函数)的this也会绑定到obj1，箭头函数的绑定无法被修改(new也不行)。

function foo() {

// 返回一个箭头函数

return (a) => {

// this继承自foo()

console.log( this.a );

};

}

var obj1 = {

a: 2

};

var obj2 = {

a: 3

}

var bar = foo.call( obj1 );

bar.call( obj2 ); // 2，不是3！

ES6之前和箭头函数类似的模式，采用的是词法作用域取代了传统的this机制。

function foo() {

var self = this; // lexical capture of this

setTimeout( function() {

console.log( self.a ); // self只是继承了foo()函数的this绑定

}, 100 );

}

var obj = {

a: 2

};

foo.call(obj); // 2

**小结：**

到此this的指向解析已经都分析完了，是不是你的脑袋已经快扛不住了，混乱了？那就把文章多读几遍，你就会慢慢体会到this指向的精髓，如果想偷懒的话那就记住这个流程图吧。

