# 深入浅出this，JavaScript中神奇的存在！

## 前言

只要你踏入 JavaScript 的世界，那么你一定会遇到 this 关键词。有许多人所 this 是 JavaScript 中最复杂的东西之一，也有人说 this 其实很简单.....但是事实确实，有许多工作了好多年的小伙伴，在 this 指向问题上也常常出现错误。

总之，我们本篇文章的目的就是为了让大家彻底理解 this，遇到 this 不再害怕！

## 1.为什么要有 this?

既然 this 这么多小伙伴都觉得难，那为什么还要使用它呢？根据哲学思想：存在即合理。既然 this 被提了出来，那么它肯定帮助我们解决了一些问题，又或者提升了开发效率。

我们先使用一句比较官方的一句话来总结 this 解决了什么问题。

**较为官方的解释：**

> this 被自动定义在所有函数的作用域中，它提供了一种更好的方式来“隐式”的传递对象引用，这样使得我们的 API 设计或者函数变得更加简洁，而且还更容易复用。

看了上面那样官方的一段话是不是感觉脑子变成了一团浆糊，没看懂要紧。我们可以结合一段代码再来理解。

**代码如下：**

```
function say() {
  console.log("你好！", this.name);
}
let person1 = {
  name: '小猪课堂'
}
let person2 = {
  name: '张三'
}

say.call(person1); // 你好！ 小猪课堂
say.call(person2); // 你好！ 张三
```

上面这段代码非常简单，我们在函数内部使用了 person1 和 person2 对象中的那么属性，但是我们的函数实际上并没有接收参数，而是调用 this 隐式的使用了 name 属性，即隐式使用上下

文对象中 name，我们利用了 call 方法将函数内部的 this 指向了 person1 和 person2，这使得我们的函数变得简洁且容易复用。

大家想一想，如果我们没有 this，那么我们就需要显式的将上下文对象传入函数，即显式传入 person1 和 person2 对象。

**代码如下：**

```
function say(context) {
  console.log("你好！", context.name);
}
let person1 = {
  name: '小猪课堂'
}
let person2 = {
  name: '张三'
}

say(person1); // 你好！ 小猪课堂
say(person2); // 你好！ 张三
```

上段代码中没有使用 this，所以我们直接显式的将上下文对象传入了函数，虽然目前代码看起来不复杂，但是随着我们的业务逻辑逐渐复杂，或者说函数变得复杂起来，那么我们传入的 context 上下文对象只会让代码变得越来越混乱。

但是如果我们使用了 this，便不会这样，前提是我们需要清楚的知道 this 指代的上下文对象是谁。

当然，如果你对上面的代码不太理解，别急，慢慢来，看完本篇文章！

## 2.this 错误的理解

对于很多初学者，刚开始接触到 this 关键词时，常常踏入很多误区。很大一部分原因确实是由于 this 有很多坑，但是最终原因还是没有搞懂 this 的指向原理。这里我们举出初学者常见的 this 误区，也是很多面试题里面常常喜欢挖坑的地方。

### 2.1 this 指向函数自身?

这一个误区是很多初学者都会踏入的，毕竟 this 关键词英译过来就是“这里”的意思，我们在函数里面使用 this，理所当然任务 this 指代的是当前函数。

但是事实果真如此吗？我们一起来看一段代码。

**代码如下：**

```
function say(num) {
  console.log("函数执行：", num);
  this.count++;
}
```

```
say.count = 0;
say(1); // 函数执行：1
say(2); // 函数执行：2
say(3); // 函数执行：3
console.log(say.count); // 0
```

上段代码中我们给 say 函数添加了一个 count 属性，因为在 JS 中函数也是一个对象。然后我们执行了 3 次函数，并且每次执行都调用了 count++。

如果我们认为 this 指向的是函数本身，那么 this.count++执行的便是 say.count，所以按理来说我们最终打印 say.count 结果应该是 3，但是结果却是 0。说明 this.count 并不是 say.count。

所以我们最终得出结论：**say 函数内部的 this 并不执行函数本身！**

那么我们上段代码中的 this.count 是哪里的 count 呢？实际上执行 this.count++的时候，会声明一个全局变量 count，至于为什么，本篇文章和后面会解释。

**打印 count：**

```
console.log(say.count); // 0
console.log(count); // NaN
```

## 2.2 this 作用域问题

作用域也是 JS 中比较难的知识点之一了，我们这里不会展开说作用域问题。我们只给出 this 指向在作用域方面的误区，这个误区很多初学者甚至好多年经验的开发者也会踏入此误区。

我们可以先来看一段非常经典的代码：

```
function foo() {
  var a = 2;
  this.bar();
}
function bar() {
  console.log(this.a);
}
foo(); // undefined
```

上段代码中我们在 foo 函数内部使用 this 调用了 bar 函数，然后在 bar 函数内部打印 a 变量，如果我们按照作用域链的思想思考的话，此时的 a 变量按道理是能够读取到的，但是事实却是 undefined。

造成上述问题的原因有多个，其中有一个就是 **this 在任何情况下都不指向函数的词法作用域**，上段代码就使用使用 this 将 foo 和 bar 函数的词法作用域联通，这是不可行的。

至于词法作用域是什么，这里不展开说，需要大家自行下去学习，简单来说词法作用域是由你在写代码时将变量和块作用域写在哪来决定的。

## 3.this 的定义

看了前面两章节，我们大概能理解 this 是什么？它其实就是一个执行上下文中的一个属性，大家也可以简单的把 this 当作一个对象，只不过该对象指向哪儿是在函数调用的时候确定的。

我们简单总结一下 this 的特点：

- this 是在运行时绑定的，不是在编写时绑定
- this 的绑定与函数的声明和位置没有任何关系
- 函数在调用时，会创建一个执行上下文，this 就是这个执行上下文中的一个属性，在函数执行的时候可以用到 this。所以 this 是在函数调用的时候确定绑定关系的，也就是运行时。

所以，**总结出来大概就一句话：**

> this 就是一个对象，this 是在函数被调用时发生的绑定，它指向什么完全取决于函数在哪里被调用。

## 4.this 绑定规则

到这里我们知道了 this 的绑定是在函数调用的时候确定的，以及 this 不指向函数自身等等问题。那么，函数在某个位置被调用时，我们怎么确定 this 该绑定到哪里呢？这个时候我们就需要一些绑定规则来帮助我们明确 this 绑定到哪里了，当然，想要运用绑定规则的前提是，我们需要知道函数的调用位置。

有些情况下，函数的调用位置我们可以直接观察出来，但是有些情况稍显复杂，这个时候我们就需要借助**调用栈来**来分析出函数的实际调用位置了。

我们可以通过浏览器来查看调用栈，简单来说调用栈就相当于函数的调用链，和作用域链有异曲同工之妙，只是我们直接看代码分析可能不太容易。所以我们可以通过打断点的方式，然后借助浏览器来查看调用栈，如下图所示：


调用栈的具体用法还需要大家下来仔细学习。

接下来就来学习具体的 this 绑定规则。

### 4.1 默认绑定

我们比较常见的一种函数调用类型就是独立函数的调用，形如 foo()等。这个时候的 this 绑定就是采用的默认绑定规则。

**代码如下：**

```
let name = '小猪课堂';
function foo(){
  console.log(this) // Window{}
  console.log(this.name) // 小猪课堂
}
foo(); // 小猪课堂
```

上段代码非常简单，我们在全局作用域中定义了一个变量 name，然后我们在函数 foo 中使用 this.name，输出的结果就是全局变量 name，这说明我们 this 指向了全局作用域，也就是说 this 绑定到了 window 对象上。

**输出结果：**


函数的这种调用方式就被称为默认绑定，默认绑定规则下的 this 指向全局对象。

我们可以给默认绑定给个定义：

> 当函数不带用任何修饰进行调用时，此时 this 的绑定就是默认绑定规则，this 指向全局对象。

**注意：**

严格模式下上段代码的 this 是 undefined，比如下面这段代码：

```
let name = '小猪课堂';
function foo(){
  'use strict'
  console.log(this.name)
}
foo(); // Uncaught TypeError: Cannot read properties of undefined (reading '
```

从上段代码可以看出，默认绑定规则下，this 绑定到了全局对象，当然这与函数调用位置有关。

但是严格模式下，this 的绑定与函数调用位置无关。

## 4.2 隐式绑定

前面的默认绑定规则很好理解，因为我们的函数执行上下文就是全局作用域，this 自然而然绑定到了全局对象上。

独立函数的调用我们可以直接看出执行上下文在哪里，但如果不是独立函数调用，比如下面代码。

**代码如下：**

```
function foo() {
  console.log(this.name) // 小猪课堂
}
let obj = {
```

```
  name: '小猪课堂',
  foo: foo
}
obj.foo();
```

上段代码我们在 obj 对象中引用了函数 foo，然后我们使用 obj.foo（函数别名）的方式调用了该函数，此时不是独立函数调用，我们不能使用默认绑定规则。

此时 this 的绑定规则称为隐式绑定规则，因为我们不能直接看出函数的调用位置，它的实际调用位置在 obj 对象里面，调用 foo 时，它的执行上下文对象为 obj 对象，所以 this 将会被绑定到 obj 对象上，所以我们函数中的 this.nam 其实就是 obj.name。这就是我们的隐式绑定规则。

**注意：**

如果我们调用函数时有多个引用调用，比如 obj1.obj2.foo()。这个时候函数 foo 中的 this 指向哪儿呢？其实不管引用链多长，this 的绑定都由最顶层调用位置确定，即 obj1.obj2.foo()的 this 还是绑定带 obj2。

**隐式绑定中 this 丢失**

在隐式绑定规则中，我们认为谁调用了函数，this 就绑定谁，比如 obj.foo 中 this 就绑定到 obj，但是有一些情况比较特殊，即使采用的隐式绑定规则，但是 this 并没有按照我们的想法去绑定，这就是所谓的隐式绑定 this 丢失，常见于回调函数中。

**代码如下：**

```
function foo() {
  console.log(this.name) // 小猪课堂
}

function doFoo(fn) {
  fn(); // 函数调用位置
}

let obj = {
  name: '张三',
  foo: foo
}
let name = '小猪课堂';
doFoo(obj.foo); // 小猪课堂
```

上段代码中我们很容易会以为 foo 绑定的 this 是 obj 对象，因为我们使用了 obj.foo 的方式，这种方式就是遵循隐式绑定规则。但是事实上 this 却绑定到了全局对象上去，这是因为我们在 doFoo 函数中调用 fn 时，这里才是函数的实际调用位置，此时是独立函数调用，所以 this 指向了全局对象。

实际项目中我们容易遇到这种问题的场景可能就是定时器了，比如下面的代码：

```
setTimeout(obj.foo, 100)
```

这种写法就很容易造成 this 丢失。

### 4.3 显式绑定

前面我们已经说了默认绑定和隐式绑定，其中隐式绑定我们通常是以 obj.foo 这种形式来调用函数的，目的就是为了让 foo 的 this 绑定到 obj 对象上。

这个时候，如果我们不想通过 obj.foo 的形式调用函数，我们想要很明确的将函数的 this 绑定在某个对象上。那么可以使用 call、apply 等方法，这就是所谓的显式绑定规则。

**代码如下：**

```
function foo() {
  console.log(this.name) // 小猪课堂
}

let obj = {
  name: '小猪课堂',
}

foo.call(obj);
```

上段代码我们利用 call 方法直接将 foo 函数内部的 this 指向了 obj 对象，这就是显式绑定。

虽然显式绑定让我们很清楚的知道了函数中的 this 绑定到了哪个对象上，但是它还是无法结局我们 this 绑定丢失的问题，就比如下面这种写法：

```
function foo() {
  console.log(this.name) // 小猪课堂
}

function doFoo(fn) {
  fn(); // 函数调用位置
}

let obj = {
  name: '张三',
  foo: foo
}
let name = '小猪课堂';
doFoo.call(obj, obj.foo); // 小猪课堂
```

上段代码我们虽然使用 call 来更改 this 绑定，但是最终结果却是没有用的。

虽然显式绑定本身不能解决 this 绑定丢失的问题，但是我们可以通过变通的方式来解决这个问题，也被称作**硬绑定**。

**硬绑定：**

```
function foo() {
```

```
  console.log(this.name) // 小猪课堂
}

function doFoo(fn) {
  fn(); // 函数调用位置
}

let obj = {
  name: '张三',
}
let bar = function () {
  foo.call(obj)
}
let name = '小猪课堂';
doFoo(bar); // 张三
setTimeout(bar, 100); // 张三
```

其实思路也比较简单，出现 this 绑定丢失原因无非就是我们传入的回调函数在被执行时，this 绑定规则变为了默认绑定，那么为了解决这个问题，我们不妨在封装一个函数，将 foo 函数的 this 显式绑定到 obj 对象上去即可。

这里提一点，下面写法是错误的：

```
doFoo(foo.call(obj));
```

因为回调函数是在 doFoo 里面执行的，上面的写法相当于 foo 函数立即执行了。

**补充：**

其实我们的 bind 函数就是一个硬绑定，大家想一想，bind 函数是不是创建一个新的函数，然后将 this 指定，是不是就和我们下面这段代码的效果一样。

```
let bar = function () {
  foo.call(obj)
}

// bind 形式
let bar = foo.bind(obj)
```

## 4.4 new 绑定

new 关键词相信大家都知道或者使用过吧，这就是我们将要将的第 4 种 this 绑定，叫做 new 绑定。

想要知道 new 绑定规则，我们就很有必要知道一个当我们 new 一个对象的时候做了什么，或者说 new 关键词会做哪些操作。我们这里简单总结一下，具体的 new 的过程还需要大家自行下来好好学学。

**使用 new 来调用函数时，会执行下面操作：**

- 创建一个全新的对象
- 这个新对象会被执行原型连接
- 这个新对象会绑定到函数调用的 this
- 如果函数没有返回其它对象，那么 new 表达式种的函数调用会自动返回这个新对象

我们可以看到 new 的操作中就有 this 的绑定，我们在来看看代码。

**代码如下：**

```
function foo(name) {
  this.name = name;
}
let bar = new foo('小猪课堂');
console.log(bar.name); // 小猪课堂
```

上段代码我们使用 new 关键词调用了 foo 函数，大家注意这不是默认调用规则，这是 new 绑定规则。

## 5.优先级

前面我们总结了 4 条 this 绑定的规则，在大多数情况下我们只需要找到函数的调用位置，然后再判断采用哪条 this 绑定规则，最终确定 this 绑定。

我们这里可以先简单总结一下 4 条规则以及 this 绑定确定流程。

**this 绑定确定流程：**

> 先确定函数调用位置，然后确定使用哪条规则，然后根据规则确定 this 绑定。

**this 绑定规则：**

- 默认绑定：this 绑定到全局对象
- 隐式绑定：一般绑定到调用对象，如 obj.foo 绑定到 obj
- 显式绑定：通过 call、apply 指定 this 绑定到哪里
  - 硬绑定：使用 bind 函数
- new 绑定：使用 new 关键词，绑定到当前函数对象

可以看到，我们确认 this 绑定的时候有 4 条规则，在通常情况下，我们可以根据这 4 条规则来判断出 this 的绑定。但是有时候某个函数的调用位置对应了多个绑定规则，这个时候我们该选用哪一条规则来确定 this 绑定呢？这个时候就需要明确每一条绑定规则的优先级了！

首先我们要明确的式默认绑定规则的优先级是最低的，所以我们考虑的时候暂时不考虑默认绑定规则。

### 5.1 隐式绑定与显式绑定

如果函数调用的时候出现了隐式绑定和显式绑定，那么具体采用哪一个规则，我们通过代码来实验一下。

**代码如下：**

```
function foo(){
  console.log(this.name);
}
let obj1 = {
  name: '小猪课堂',
  foo: foo
}
let obj2 = {
  name: '李四',
  foo: foo
}

obj1.foo(); // 小猪课堂
obj2.foo(); // 李四

obj1.foo.call(obj2); // 李四
obj2.foo.call(obj1); // 小猪课堂
```

上段代码中我们涉及到了两种 this 绑定，obj.foo 为隐式绑定，this 绑定给 obj 对象，而 foo.call(obj)为显示绑定，this 绑定给 obj 对象。

从上段代码看出，当两个绑定规则都存在的时候，我们采用的是显式绑定规则。

**总结：**

显式绑定 > 隐式绑定

## 5.2 new 绑定与隐式绑定

接下来我们看看 new 绑定与隐式绑定的优先级。

**代码如下：**

```
function foo(name) {
  this.name = name;
}
let obj1 = {
  foo: foo
}

obj1.foo('小猪课堂');
let bar = new obj1.foo("张三");
console.log(obj1.name); // 小猪课堂
console.log(bar.name); // 张三
```

上段代码中在在使用 new 关键词的时候又使用了 obj1.foo 隐式绑定，但是最终结果 this 并没有绑定到 obj1 对象上，所以隐式绑定优先级低于 new 绑定。

**总结：**

隐式绑定 < new 绑定

### 5.3 显式绑定与 new 绑定

接下来我们比较显式绑定与 new 绑定规则，但是我们 new 绑定与显式绑定的 call、apply 方法不能一起使用，所以我们无法通过 new foo.call(obj)来进行测试。

但是我们前面讲解显式绑定的时候，提到一种绑定叫做硬绑定，它也是显式绑定中的一种，所以说我们可以利用硬绑定与 new 绑定来进行比较。

**代码如下：**

```
function foo(name) {
  this.name = name;
}
let obj1 = {};
let bar = foo.bind(obj1);
bar('小猪课堂');
console.log(obj1.name); // 小猪课堂

let baz = new bar('张三');
console.log(obj1.name); // 小猪课堂
console.log(baz.name); // 张三
```

上段代码中我们把 obj 硬绑定到了 bar 上，然后我们通过 new 绑定调用了函数，但是 obj1.name 还是小猪课堂，并没有改为张三，但是，我们的 new 操作修改了硬绑定（到 obj1 的）调用 bar 中的 this。因为使用了 new 绑定，我们得到了新的 baz 对象，并且 baz.name 为张三。

**总结：**

new 绑定 > 显式绑定，需要注意的是，new 操作时将 this 绑定到了新创建的对象。

## 6.this 绑定总结

到这儿，我们基本上能够确定一个函数内部的 this 指向哪儿了，我们这里做出一些总结，以供在项目实践中判断 this 绑定。

**this 绑定规则优先级：**

默认绑定 < 隐式绑定 < 显式绑定 < new 绑定

**判断 this 最终指向，总体流程：**

1. 判断函数调用时是否使用了 new，即 new 绑定，如果使用了，则 this 绑定的是新创建的对象。

2. 函数调用是否使用了 call、apply 等显式绑定，或者硬绑定（bind），如果是的话，this 指向指定的对象。
3. 函数是否在某个上下文对象中调用，即隐式绑定，如 obj1.foo，如果是的话，this 指向绑定的那个上下文对象。
4. 以上 3 点都不涉及的话，则采用默认绑定，但是需要注意的是，在严格模式下，默认绑定的 this 是 undefined，非严格模式下绑定到全局对象。

结合上面的绑定优先级以及判断流程，我们在一般的项目中以及能够判断出 this 指向哪儿了！

## 总结

this 绑定虽然是一个比较难的知识点，但是我们作为一个前端开发者，必须要学会如何理解和使用它，因为它确实能给我们带来很多的便利和好处。

当然，本篇文章只讲解了最常规的 this 绑定及原理，除此之外，this 绑定还有一些意外的情况，这里不做更多解释，感兴趣的小伙伴可以自行下来查询资料，比如说软绑定、间接引用等等。

**总之，常规判断如下：**

1. 由 new 调用函数，绑定到新对象
2. 由 call 等调用，绑定到指定对象
3. 由上下文对象效用，绑定到上下文对象
4. 默认调用，绑定到全局对象或 undefined

参考资料： 《你不知道的JavaScript》