我所理解的this

先来看一个：

```
let obj = {
  msg: 'obj\'s message',
  sendMessage () {
    console.log(this.msg)
  }
}
obj.sendMessage()
```

执行这段代码的结果是什么？运行结果就是输出内容：`obj's message`。如果想让它执行结束过一秒再执行一次的话，可能会认为代码应该是这个样子的：

```
let obj = {
  msg: 'obj\'s message',
  sendMessage () {
    console.log(this.msg)
  }
}
obj.sendMessage()
setTimeout(obj.sendMessage, 1000)
```

然而中执行结果却是：`obj's message`，一秒后：`undefined`

立即执行的没有问题，过一秒再执行的就成了`undefined`了？如果在代码中再加一行，然后在浏览器中运行的话就更奇怪了：

```
// 注意是var，而不是let
var msg = 'Global Message!'
let obj = {
  msg: 'obj\'s message',
  sendMessage () {
    console.log(this.msg)
  }
}
obj.sendMessage()
setTimeout(obj.sendMessage, 1000)
```

结果是：`obj's message`，一秒后：`Global Message!`

现在就来看看这个`this`为什么这么古怪，它究竟是什么。

this是什么?

`this`代表的是执行函数时自动生成的一个内部对象，只存在于函数体中。函数在调用的时候，这个函数就会有一个执行环境，这个执行环境描述了包括函数在哪里被调用，调用的方法，变量对象等等信息。也就是说，这个this的值是在函数调用时才有的，也只有在函数调用时才存在，**也就是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。**

我们通过不同的函数调用情况来看`this`的值。

纯粹的函数调用

函数调用 即`functionName ()`模式，这也是我们使用的最多的一种方式，其属于全局调用，默认情况下函数内部的`this`指向`window`，当然是在非严格模式下

```
var name = 'aaa'
function showName () {
  console.log(this.name) // aaa
  console.log(this === window) // true
}
showName()
```

对象的方法调用

当一个函数作为对象的某个属性方法被调用的时候

```
let obj1 = {
  msg: 'obj1\'s message',
  sendMessage () {
    console.log(this.msg)
  }
}
obj1.sendMessage() // obj1's message

// 或者
function sendMessage () {
  console.log(this.msg)
}
let obj2 = {
  msg: 'obj2\'s message',
  sendMessage
}
obj2.sendMessage() // obj2's message
```

因为这个函数被调用时的执行环境就是`obj`对象，所以函数调用的`this`会绑定到这个对象上

假如想给`obj`的`sendMessage`方法再起一个名字叫`anotherFuncName`，然后通过`anotherFuncName`来调用这个方法：

```
function sendMessage () {
  console.log(this.msg)
}
const obj = {
  msg: 'obj\'s message',
  sendMessage
}
let anotherFuncName = obj.sendMessage
anotherFuncName() // undefined
```

返回的结果是`undefined`。为什么呢？实际上`anotherFuncName`引用的依然是`sendMessage`函数本身。在调用的时候就相当于普通的函数调用，并没有额外的信息，所以`this`绑定的`Window`对象，`window`上此事并没有`msg`属性，自然输出是`undefined`。

这个例子想要说明什么？把函数赋值给或当做参数传递的时候也会意外的发生同样的事情，只是会给函数再加一个名称，并不会真正的把函数执行的相关信息也一同复制过来。所以在调用的时候就如同普通的函数调用一样，而函数中的如果有`this`的话，自然就绑定的是`undefined`。这也就是文章开头看到的延时一秒执行出问题的原因。问题怎么解决呢?

call、apply和bind

使用`call`和`apply`方式去调用一个函数的时候，内部的`this`指向的是传进来的第一个参数，当第一个参数是`undefined`或者`null`的时候，依旧指向`window`，所以说这三个函数是用来**改变函数执行时的上下文**，再具体一点就是**改变函数运行时的this指向**。

```
let showName = function () {
  console.log(this)
}
showName() // window
showName.call(undefined) // window
showName.call(null) // window
showName.call({name: 'aaa'}) // {name: 'aaa'}
```

所以文章开头的问题可以这样来解决：

```
function sendMessage () {
  console.log(this.msg)
}
let obj = {
  msg: 'obj\'s message',
  sendMessage
}
let anotherFuncName = function () {
  sendMessage.call(obj)
}
setTimeout(anotherFuncName, 1000)
```

这样一来，通过`anotherFuncName`来调用函数，会在执行的时候将`sendMessage`函数的`this`绑定到`obj`对象上了。

call、apply与bind的差别

`call`和`apply`改变了函数的`this`上下文后便执行该函数，是立即执行函数，而`bind`则是返回改变了上下文后的一个函数，也就是创建一个新的包装函数，并且返回，而不是立刻执行。

call、apply的区别

他们俩之间的差别在于参数的区别，`call`和`aplly`的第一个参数都是要改变上下文的对象，而`call`从第二个参数开始以参数列表的形式展现，`apply`则是把除了改变上下文对象的参数放在一个数组里面作为它的第二个参数。

```
fn.call(obj, arg1, arg2, arg3...)
fn.apply(obj, [arg1, arg2, arg3...])
```

构造函数调用

当使用`new`去调用一个构造函数的时候，内部的`this`，指向的是实例化出来的对象

```
function Say (msg) {
  this.msg = msg
}
let obj = new Say('Hello')
console.log(obj.msg) // Hello
```

在执行`new`时，会将`this`绑定到新建的`obj`上，所以`obj`会获得一个`msg`属性。

构造函数也是函数，所以当你用普通调用方式调用时：

```
function Say (msg) {
  this.msg = msg
}
Say('hello')

window.msg // hello
```

箭头函数

在 ES6 的新规范中，加入了箭头函数，它和普通函数最不一样的一点就是`this`的指向，普通函数中的`this`，是运行时候决定的，而箭头函数却是定义时候就决定了。

```
let obj = {
  name: 'aaa',
  showName1: () => {
    console.log(this.name)
  },
  showName2: function () {
    console.log(this.name)
  }
}
obj.showName1() // undefined
obj.showName2() // aaa
```

箭头函数没有自己的this，导致内部的this就是外部的this，此时当方法被调用的时候，箭头函数的this总是指向调用方法的对象，也就是window。所以箭头函数体内的this对象就是定义时所在的对象，而不是使用时所在的对象。

```
function asd () {
  return () => {
    console.log(this, this.id)
  }
}
asd()() // window, undefined
asd.call({id: 123})() // {id: 123} 123
```

一些坑

setTimeout

```
let obj = {
  name: 'aaa',
  showName: function () {
    console.log(this.name)
  },
  showNameLater: function () {
    setTimeout(this.showName, 1000)
  }
}
obj.showNameLater() // undefined
```

这里在执行setTimeout这个函数的时候传了obj的showName函数作为第一个参数，其效果与`let showName = obj.showName`是相同的。而setTimeout内部其实也是执行了传进去这个函数而已，即：`showName()`，所以这个时候输入为`undefined`也就好理解了。那么怎么解决这个问题呢，毕竟期望的效果是输出`aaa`

```
let obj1 = {
  name: 'aaa',
  showName: function () {
```

```
    console.log(this.name)
  },
  showNameLater: function () {
    let _this = this
    setTimeout(function () {
      _this.showName()
    }, 1000)
  }
}
obj1.showNameLater() // aaa

// 或者
let obj2 = {
  name: 'aaa',
  showName: function () {
    console.log(this.name)
  },
  showNameLater: function () {
    setTimeout(this.showName.bind(obj2), 1000)
  }
}
obj2.showNameLater() // aaa
```

为构造函数指定this

```
let Person = function (name, sex) {
  this.name = name
  this.sex = sex
}
// 这里报错了，原因是我们去 new 了 Person.call 函数 ，这里的函数不是一个构造函数；
let p1 = new Person.call({}) // Person.call is not a constructor
```

解决方式：

```
let Person = function (name, sex) {
  this.name = name
  this.sex = sex
}
let p1 = new (Person.bind({}))('aaa', 'male')
console.log(p1.name, p1.sex) // aaa male
```

为箭头函数指定this

```
let show = () => {
  console.log(this)
}
show() // window
show.call({name: 'aaa'}) // window
```

可以看到使用call来手动改变箭头函数中的this的时候，无法成功。 箭头函数中的 this 在定义它的时候已经决定了（执行定义它的作用域中的 this），与如何调用以及在哪里调用它无关，包括 (call, apply, bind) 等操作都无法改变它的 this。