1.2 误解

我们之后会解释 this 到底是如何工作的，但是首先需要消除一些关于 this 的错误认识。

太拘泥于“this”的字面意思就会产生一些误解。有两种常见的对于 this 的解释，但是它们都是错误的。

1.2.1 指向自身

人们很容易把 this 理解成指向函数自身，这个推断从英语的语法角度来说是说得通的。

那么为什么需要从函数内部引用函数自身呢？常见的原因是递归（从函数内部调用这个函数）或者可以写一个在第一次被调用后自己解除绑定的事件处理器。

JavaScript 的新手开发者通常会认为，既然函数看作一个对象（JavaScript 中的所有函数都是对象），那就可以在调用函数时存储状态（属性的值）。这是可行的，有些时候也确实有用，但是在本书即将介绍的许多模式中你会发现，除了函数对象还有许多更合适存储状态的地方。

不过现在我们先来分析一下这个模式，让大家看到 this 并不像我们所想的那样指向函数本身。

我们想要记录一下函数 foo 被调用的次数，思考一下下面的代码：

```
function foo(num) {
    console.log( "foo: " + num );

    // 记录 foo 被调用的次数
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// foo 被调用了多少次?
console.log( foo.count ); // 0 -- WTF?
```

console.log 语句产生了 4 条输出，证明 foo(..) 确实被调用了 4 次，但是 foo.count 仍然是 0。显然从字面意思来理解 this 是错误的。

执行 foo.count = 0 时，的确向函数对象 foo 添加了一个属性 count。但是函数内部代码 this.count 中的 this 并不是指向那个函数对象，所以虽然属性名相同，根对象却并不相同，困惑随之产生。

负责的开发者一定会问“如果我增加的 count 属性和预期的不一样，那我增加的是哪个 count？”实际上，如果他深入探索的话，就会发现这段代码在无意中创建了一个全局变量 count（原理参见第 2 章），它的值为 NaN。当然，如果他发现了这个奇怪的结果，那一定会接着问：“为什么它是全局的，为什么它的值是 NaN 而不是其他更合适的值？”（参见第 2 章。）

遇到这样的问题时，许多开发者并不会深入思考为什么 this 的行为和预期的不一致，也不会试图回答那些很难解决但却非常重要的问题。他们只会回避这个问题并使用其他方法来达到目的，比如创建另一个带有 count 属性的对象。

```
function foo(num) {
    console.log( "foo: " + num );

    // 记录 foo 被调用的次数
    data.count++;
}

var data = {
    count: 0
};

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// foo 被调用了多少次?
console.log( data.count ); // 4
```

从某种角度来说这个方法确实“解决”了问题，但可惜它忽略了真正的问题——无法理解 this 的含义和工作原理——而是返回舒适区，使用了一种更熟悉的技术：词法作用域。

词法作用域是一种非常优秀并且有用的技术。我丝毫没有贬低它的意思（可以参考本书第一部分“作用域和闭包”）。但是如果你仅仅是因为无法猜对 this 的用法，就放弃学习 this 而去使用词法作用域，就不能算是一种很好的解决办法了。

如果要从函数对象内部引用它自身，那只使用 this 是不够的。一般来说你需要通过一个指向函数对象的词法标识符（变量）来引用它。

思考一下下面这两个函数：

```
function foo() {
    foo.count = 4; // foo 指向它自身
}

setTimeout( function(){
    // 匿名（没有名字的）函数无法指向自身
}, 10 );
```

第一个函数被称为具名函数，在它内部可以使用 foo 来引用自身。

但是在第二个例子中，传入 setTimeout(..) 的回调函数没有名称标识符（这种函数被称为匿名函数），因此无法从函数内部引用自身。

还有一种传统的但是现在已经被弃用和批判的用法，是使用 arguments.callee 来引用当前正在运行的函数对象。这是唯一一种可以从匿名函数对象内部引用自身的方法。然而，更好的方式是避免使用匿名函数，至少在需要自引用时使用具名函数（表达式）。arguments.callee 已经被弃用，不应该再使用它。

所以，对于我们的例子来说，另一种解决方法是使用 foo 标识符替代 this 来引用函数对象：

```
function foo(num) {
    console.log( "foo: " + num );

    // 记录 foo 被调用的次数
    foo.count++;
}
foo.count=0
var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
```

```
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// foo 被调用了多少次?
console.log( foo.count ); // 4
```

然而，这种方法同样回避了 this 的问题，并且完全依赖于变量 foo 的词法作用域。

另一种方法是强制 this 指向 foo 函数对象：

```
function foo(num) {
    console.log( "foo: " + num );

    // 记录 foo 被调用的次数
    // 注意，在当前的调用方式下（参见下方代码），this 确实指向 foo
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        // 使用 call(..) 可以确保 this 指向函数对象 foo 本身
        foo.call( foo, i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// foo 被调用了多少次?
console.log( foo.count ); // 4
```

这次我们接受了 this，没有回避它。如果你仍然感到困惑的话，不用担心，之后我们会详细解释具体的原理。

1.2.2 它的作用域

第二种常见的误解是，this 指向函数的作用域。这个问题有点复杂，因为在某种情况下它是正确的，但是在其他情况下它却是错误的。

需要明确的是，this 在任何情况下都不指向函数的词法作用域。在 JavaScript 内部，作用域确实和对象类似，可见的标识符都是它的属性。但是作用域“对象”无法通过 JavaScript 代码访问，它存在于 JavaScript *引擎* 内部。

思考一下下面的代码，它试图（但是没有成功）跨越边界，使用 this 来隐式引用函数的词法作用域：

```
function foo() {
    var a = 2;
    this.bar();
}

function bar() {
    console.log( this.a );
}

foo(); // ReferenceError: a is not defined
```

这段代码中的错误不止一个。虽然这段代码看起来好像是我们故意写出来的例子，但是实际上它出自一个公共社区中互助论坛的精华代码。这段代码非常完美（同时也令人伤感）地展示了 this 多么容易误导人。

首先，这段代码试图通过 this.bar() 来引用 bar() 函数。这是绝对不可能成功的，我们之后会解释原因。调用 bar() 最自然的方法是省略前面的 this，直接使用词法引用标识符。

此外，编写这段代码的开发者还试图使用 this 联通 foo() 和 bar() 的词法作用域，从而让 bar() 可以访问 foo() 作用域里的变量 a。这是不可能实现的，你不能使用 this 来引用一个词法作用域内部的东西。

每当你想要把 this 和词法作用域的查找混合使用时，一定要提醒自己，这是无法实现的。

1.3 this到底是什么

排除了一些错误理解之后，我们来看看 this 到底是一种什么样的机制。

之前我们说过 this 是在运行时进行绑定的，并不是在编写时绑定，它的上下文取决于函数调用时的各种条件。this 的绑定和函数声明的位置没有任何关系，只取决于函数的调用方式。

当一个函数被调用时，会创建一个活动记录（有时候也称为执行上下文）。这个记录会包含函数在哪里被调用（调用栈）、函数的调用方法、传入的参数等信息。this 就是记录的其中一个属性，会在函数执行的过程中用到。

在下一章我们会学习如何寻找函数的调用位置，从而判断函数在执行过程中会如何绑定 this。

1.4 小结

对于那些没有投入时间学习 this 机制的 JavaScript 开发者来说，this 的绑定一直是一件非

常令人困惑的事。this 是非常重要的，但是猜测、尝试并出错和盲目地从 Stack Overflow 上复制和粘贴答案并不能让你真正理解 this 的机制。

学习 this 的第一步是明白 this 既不指向函数自身也不指向函数的词法作用域，你也许被这样的解释误导过，但其实它们都是错误的。

this 实际上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。

第2章

this全面解析

在第 1 章中，我们排除了一些对于 this 的错误理解并且明白了每个函数的 this 是在调用时被绑定的，完全取决于函数的调用位置（也就是函数的调用方法）。

2.1 调用位置

在理解 this 的绑定过程之前，首先要理解*调用位置*：调用位置就是函数在代码中被调用的位置（而不是声明的位置）。只有仔细分析调用位置才能回答这个问题：这个 this 到底引用的是什么？

通常来说，寻找调用位置就是寻找“函数被调用的位置”，但是做起来并没有这么简单，因为某些编程模式可能会隐藏真正的调用位置。

最重要的是要分析*调用栈*（就是为了到达当前执行位置所调用的所有函数）。我们关心的调用位置就在当前正在执行的函数的*前一个*调用*中*。

下面我们来看看到底什么是调用栈和调用位置：

```
function baz() {
    // 当前调用栈是：baz
    // 因此，当前调用位置是全局作用域

    console.log( "baz" );
    bar(); // <-- bar 的调用位置
}

function bar() {
```

```
    // 当前调用栈是 baz -> bar
    // 因此，当前调用位置在 baz 中

    console.log( "bar" );
    foo(); // <-- foo 的调用位置
}

function foo() {
    // 当前调用栈是 baz -> bar -> foo
    // 因此，当前调用位置在 bar 中

    console.log( "foo" );
}

baz(); // <-- baz 的调用位置
```

注意我们是如何（从调用栈中）分析出真正的调用位置的，因为它决定了 this 的绑定。

你可以把调用栈想象成一个函数调用链，就像我们在前面代码段的注释中所写的一样。但是这种方法非常麻烦并且容易出错。另一个查看调用栈的方法是使用浏览器的调试工具。绝大多数现代桌面浏览器都内置了开发者工具，其中包含 JavaScript 调试器。就本例来说，你可以在工具中给 foo() 函数的第一行代码设置一个断点，或者直接在第一行代码之前插入一条 debugger; 语句。运行代码时，调试器会在那个位置暂停，同时会展示当前位置的函数调用列表，这就是你的调用栈。因此，如果你想要分析 this 的绑定，使用开发者工具得到调用栈，然后找到栈中第二个元素，这就是真正的调用位置。

2.2 绑定规则

我们来看看在函数的执行过程中调用位置如何决定 this 的绑定对象。

你必须找到调用位置，然后判断需要应用下面四条规则中的哪一条。我们首先会分别解释这四条规则，然后解释多条规则都可用时它们的优先级如何排列。

2.2.1 默认绑定

首先要介绍的是最常用的函数调用类型：独立函数调用。可以把这条规则看作是无法应用其他规则时的默认规则。

思考一下下面的代码：

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

```
var a = 2;

foo(); // 2
```

你应该注意到的第一件事是，声明在全局作用域中的变量（比如 var a = 2）就是全局对象的一个同名属性。它们本质上就是同一个东西，并不是通过复制得到的，就像一个硬币的两面一样。

接下来我们可以看到当调用 foo() 时，this.a 被解析成了全局变量 a。为什么？因为在本例中，函数调用时应用了 this 的默认绑定，因此 this 指向全局对象。

那么我们怎么知道这里应用了默认绑定呢？可以通过分析调用位置来看看 foo() 是如何调用的。在代码中，foo() 是直接使用不带任何修饰的函数引用进行调用的，因此只能使用默认绑定，无法应用其他规则。

如果使用严格模式（strict mode），那么全局对象将无法使用默认绑定，因此 this 会绑定到 undefined：

```
function foo() {
    "use strict";

    console.log( this.a );
}

var a = 2;

foo(); // TypeError: this is undefined
```

这里有一个微妙但是非常重要的细节，虽然 this 的绑定规则完全取决于调用位置，但是只有 foo() 运行在非 strict mode 下时，默认绑定才能绑定到全局对象；严格模式下与 foo() 的调用位置无关：

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

var a = 2;

(function(){
    "use strict";

    foo(); // 2
})();
```

通常来说你不应该在代码中混合使用 strict mode 和 non-strict mode。整个程序要么严格要么非严格。然而，有时候你可能会用到第三方库，其严格程度和你的代码有所不同，因此一定要注意这类兼容性细节。

2.2.2 隐式绑定

另一条需要考虑的规则是调用位置是否有上下文对象，或者说是否被某个对象拥有或者包含，不过这种说法可能会造成一些误导。

思考下面的代码：

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

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

首先需要注意的是 foo() 的声明方式，及其之后是如何被当作引用属性添加到 obj 中的。但是无论是直接在 obj 中定义还是先定义再添加为引用属性，这个函数严格来说都不属于 obj 对象。

然而，调用位置会使用 obj 上下文来引用函数，因此你可以说函数被调用时 obj 对象“拥有”或者“包含”它。

无论你如何称呼这个模式，当 foo() 被调用时，它的落脚点确实指向 obj 对象。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。因为调用 foo() 时 this 被绑定到 obj，因此 this.a 和 obj.a 是一样的。

对象属性引用链中只有最顶层或者说最后一层会影响调用位置。举例来说：

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

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

隐式丢失

一个最常见的 this 绑定问题就是被隐式绑定的函数会丢失绑定对象，也就是说它会应用默