在某些场景下 this 的绑定行为会出乎意料，你认为应当应用其他绑定规则时，实际上应用的可能是默认绑定规则。

2.4.1 被忽略的this

如果你把 null 或者 undefined 作为 this 的绑定对象传入 call、apply 或者 bind，这些值在调用时会被忽略，实际应用的是默认绑定规则：

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

var a = 2;

foo.call( null ); // 2
```

那么什么情况下你会传入 null 呢？

一种非常常见的做法是使用 apply(..) 来“展开”一个数组，并当作参数传入一个函数。类似地，bind(..) 可以对参数进行柯里化（预先设置一些参数），这种方法有时非常有用：

```
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
```

这两种方法都需要传入一个参数当作 this 的绑定对象。如果函数并不关心 this 的话，你仍然需要传入一个占位值，这时 null 可能是一个不错的选择，就像代码所示的那样。

尽管本书中未提到，但在 ES6 中，可以用 ... 操作符代替 apply(..) 来“展开”数组，foo(...[1,2]) 和 foo(1,2) 是一样的，这样可以避免不必要的 this 绑定。可惜，在 ES6 中没有柯里化的相关语法，因此还是需要使用 bind(..)。

然而，总是使用 null 来忽略 this 绑定可能产生一些副作用。如果某个函数确实使用了 this（比如第三方库中的一个函数），那默认绑定规则会把 this 绑定到全局对象（在浏览器中这个对象是 window），这将导致不可预计的后果（比如修改全局对象）。

显而易见，这种方式可能会导致许多难以分析和追踪的 bug。

更安全的this

一种“更安全”的做法是传入一个特殊的对象，把 this 绑定到这个对象不会对你的程序产生任何副作用。就像网络（以及军队）一样，我们可以创建一个“DMZ”（demilitarized zone，非军事区）对象——它就是一个空的非委托的对象（委托在第 5 章和第 6 章介绍）。

如果我们在忽略 this 绑定时总是传入一个 DMZ 对象，那就什么都不用担心了，因为任何对于 this 的使用都会被限制在这个空对象中，不会对全局对象产生任何影响。

由于这个对象完全是一个空对象，我自己喜欢用变量名 ø（这是数学中表示空集合符号的小写形式）来表示它。在大多数键盘（比如说 Mac 的 US 布局键盘）上都可以使用⌥+o（Option-o）来打出这个符号。有些系统允许你为特殊符号设定快捷键。如果你不喜欢 ø 符号或者你的键盘不太容易打出这个符号，那你可以换一个喜欢的名字来称呼它。

无论你叫它什么，在 JavaScript 中创建一个空对象最简单的方法都是 Object.create(null)（详细介绍请看第 5 章）。Object.create(null) 和 {} 很像，但是并不会创建 Object.prototype 这个委托，所以它比 {}“更空”：

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// 我们的 DMZ 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
```

使用变量名 ø 不仅让函数变得更加“安全”，而且可以提高代码的可读性，因为 ø 表示“我希望 this 是空”，这比 null 的含义更清楚。不过再说一遍，你可以用任何喜欢的名字来命名 DMZ 对象。

2.4.2 间接引用

另一个需要注意的是，你有可能（有意或者无意地）创建一个函数的“间接引用”，在这种情况下，调用这个函数会应用默认绑定规则。

间接引用最容易在赋值时发生：

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

```
var a = 2;
var o = { a: 3, foo: foo };
var p = { a: 4 };

o.foo(); // 3
(p.foo = o.foo)(); // 2
```

赋值表达式 p.foo = o.foo 的返回值是目标函数的引用，因此调用位置是 foo() 而不是 p.foo() 或者 o.foo()。根据我们之前说过的，这里会应用默认绑定。

注意：对于默认绑定来说，决定 this 绑定对象的并不是调用位置是否处于严格模式，而是函数体是否处于严格模式。如果函数体处于严格模式，this 会被绑定到 undefined，否则 this 会被绑定到全局对象。

2.4.3 软绑定

之前我们已经看到过，硬绑定这种方式可以把 this 强制绑定到指定的对象（除了使用 new 时），防止函数调用应用默认绑定规则。问题在于，硬绑定会大大降低函数的灵活性，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。

如果可以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式绑定修改 this 的能力。

可以通过一种被称为软绑定的方法来实现我们想要的效果：

```
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;
    };
}
```

除了软绑定之外，softBind(..) 的其他原理和 ES5 内置的 bind(..) 类似。它会对指定的函数进行封装，首先检查调用时的 this，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。此外，这段代码还支持可选的柯里化（详情请查看之前和 bind(..) 相关的介绍）。

下面我们看看 softBind 是否实现了软绑定功能：

```
function foo() {
   console.log("name: " + this.name);
}

var obj = { name: "obj" },
    obj2 = { name: "obj2" },
    obj3 = { name: "obj3" };

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

obj2.foo = foo.softBind(obj);
obj2.foo(); // name: obj2 <---- 看！！！

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

setTimeout( obj2.foo, 10 );
// name: obj   <---- 应用了软绑定
```

可以看到，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。

2.5 this词法

我们之前介绍的四条规则已经可以包含所有正常的函数。但是 ES6 中介绍了一种无法使用这些规则的特殊函数类型：箭头函数。

箭头函数并不是使用 function 关键字定义的，而是使用被称为“胖箭头”的操作符 => 定义的。箭头函数不使用 this 的四种标准规则，而是根据外层（函数或者全局）作用域来决定 this。

我们来看看箭头函数的词法作用域：

```
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 !
```

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！）

箭头函数最常用于回调函数中，例如事件处理器或者定时器：

```
function foo() {
    setTimeout(() => {
        // 这里的 this 在此法上继承自 foo()
        console.log( this.a );
    },100);
}

var obj = {
    a:2
};

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

箭头函数可以像 bind(..) 一样确保函数的 this 被绑定到指定对象，此外，其重要性还体现在它用更常见的词法作用域取代了传统的 this 机制。实际上，在 ES6 之前我们就已经在使用一种几乎和箭头函数完全一样的模式。

```
function foo() {
    var self = this; // lexical capture of this
    setTimeout( function(){
        console.log( self.a );
    }, 100 );
}

var obj = {
    a: 2
};

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

虽然 self = this 和箭头函数看起来都可以取代 bind(..)，但是从本质上来说，它们想替代的是 this 机制。

如果你经常编写 this 风格的代码，但是绝大部分时候都会使用 self = this 或者箭头函数来否定 this 机制，那你或许应当：

1. 只使用词法作用域并完全抛弃错误 this 风格的代码；
2. 完全采用 this 风格，在必要时使用 bind(..)，尽量避免使用 self = this 和箭头函数。

当然，包含这两种代码风格的程序可以正常运行，但是在同一个函数或者同一个程序中混合使用这两种风格通常会使代码更难维护，并且可能也会更难编写。

2.6 小结

如果要判断一个运行中函数的 this 绑定，就需要找到这个函数的直接调用位置。找到之后就可以顺序应用下面这四条规则来判断 this 的绑定对象。

1. 由 new 调用？绑定到新创建的对象。
2. 由 call 或者 apply（或者 bind）调用？绑定到指定的对象。
3. 由上下文对象调用？绑定到那个上下文对象。
4. 默认：在严格模式下绑定到 undefined，否则绑定到全局对象。

一定要注意，有些调用可能在无意中使用默认绑定规则。如果想“更安全”地忽略 this 绑定，你可以使用一个 DMZ 对象，比如 ø = Object.create(null)，以保护全局对象。

ES6 中的箭头函数并不会使用四条标准的绑定规则，而是根据当前的词法作用域来决定 this，具体来说，箭头函数会继承外层函数调用的 this 绑定（无论 this 绑定到什么）。这其实和 ES6 之前代码中的 self = this 机制一样。

第3章

对象

在第 1 章和第 2 章中，我们介绍了函数调用位置的不同会造成 this 绑定对象的不同。但是对象到底是什么，为什么我们需要绑定它们呢？本章会详细介绍对象。

3.1 语法

对象可以通过两种形式定义：声明（文字）形式和构造形式。

对象的文字语法大概是这样：

```
var myObj = {
    key: value
    // ...
};
```

构造形式大概是这样：

```
var myObj = new Object();
myObj.key = value;
```

构造形式和文字形式生成的对象是一样的。唯一的区别是，在文字声明中你可以添加多个键 / 值对，但是在构造形式中你必须逐个添加属性。

用上面的“构造形式”来创建对象是非常少见的，一般来说你会使用文字语法，绝大多数内置对象也是这样做的（稍后解释）。

3.2 类型

对象是 JavaScript 的基础。在 JavaScript 中一共有六种主要类型（术语是“语言类型”）：

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `object`

注意，简单基本类型（`string`、`boolean`、`number`、`null` 和 `undefined`）本身并不是对象。`null` 有时会被当作一种对象类型，但是这其实只是语言本身的一个 bug，即对 `null` 执行 `typeof null` 时会返回字符串 `"object"`。[1] 实际上，`null` 本身是基本类型。

有一种常见的错误说法是“JavaScript 中万物皆是对象”，这显然是错误的。

实际上，JavaScript 中有许多特殊的对象子类型，我们可以称之为复杂基本类型。

函数就是对象的一个子类型（从技术角度来说就是“可调用的对象”）。JavaScript 中的函数是“一等公民”，因为它们本质上和普通的对象一样（只是可以调用），所以可以像操作其他对象一样操作函数（比如当作另一个函数的参数）。

数组也是对象的一种类型，具备一些额外的行为。数组中内容的组织方式比一般的对象要稍微复杂一些。

内置对象

JavaScript 中还有一些对象子类型，通常被称为内置对象。有些内置对象的名字看起来和简单基础类型一样，不过实际上它们的关系更复杂，我们稍后会详细介绍。

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`

注 1：原理是这样的，不同的对象在底层都表示为二进制，在 JavaScript 中二进制前三位都为 0 的话会被判断为 `object` 类型，`null` 的二进制表示是全 0，自然前三位也是 0，所以执行 `typeof` 时会返回“`object`”。——译者注

- Date
- RegExp
- Error

这些内置对象从表现形式来说很像其他语言中的类型（type）或者类（class），比如 Java 中的 String 类。

但是在 JavaScript 中，它们实际上只是一些内置函数。这些内置函数可以当作构造函数（由 new 产生的函数调用——参见第 2 章）来使用，从而可以构造一个对应子类型的新对象。举例来说：

```
var strPrimitive = "I am a string";
typeof strPrimitive; // "string"
strPrimitive instanceof String; // false

var strObject = new String( "I am a string" );
typeof strObject; // "object"
strObject instanceof String; // true

// 检查 sub-type 对象
Object.prototype.toString.call( strObject ); // [object String]
```

在之后的章节中我们会详细介绍 Object.prototype.toString... 是如何工作的，不过简单来说，我们可以认为子类型在内部借用了 Object 中的 toString() 方法。从代码中可以看到，strObject 是由 String 构造函数创建的一个对象。

原始值 "I am a string" 并不是一个对象，它只是一个字面量，并且是一个不可变的值。如果要在这个字面量上执行一些操作，比如获取长度、访问其中某个字符等，那需要将其转换为 String 对象。

幸好，在必要时语言会自动把字符串字面量转换成一个 String 对象，也就是说你并不需要显式创建一个对象。JavaScript 社区中的大多数人都认为能使用文字形式时就不要使用构造形式。

思考下面的代码：

```
var strPrimitive = "I am a string";

console.log( strPrimitive.length ); // 13

console.log( strPrimitive.charAt( 3 ) ); // "m"
```

使用以上两种方法，我们都可以直接在字符串字面量上访问属性或者方法，之所以可以这样做，是因为引擎自动把字面量转换成 String 对象，所以可以访问属性和方法。

同样的事也会发生在数值字面量上，如果使用类似 42.359.toFixed(2) 的方法，引擎会把

42 转换成 new Number(42)。对于布尔字面量来说也是如此。

null 和 undefined 没有对应的构造形式，它们只有文字形式。相反，Date 只有构造，没有文字形式。

对于 Object、Array、Function 和 RegExp（正则表达式）来说，无论使用文字形式还是构造形式，它们都是对象，不是字面量。在某些情况下，相比用文字形式创建对象，构造形式可以提供一些额外选项。由于这两种形式都可以创建对象，所以我们首选更简单的文字形式。建议只在需要那些额外选项时使用构造形式。

Error 对象很少在代码中显式创建，一般是在抛出异常时被自动创建。也可以使用 new Error(..) 这种构造形式来创建，不过一般来说用不着。

3.3 内容

之前我们提到过，对象的内容是由一些存储在特定命名位置的（任意类型的）值组成的，我们称之为属性。

需要强调的一点是，当我们说“内容”时，似乎在暗示这些值实际上被存储在对象内部，但是这只是它的表现形式。在引擎内部，这些值的存储方式是多种多样的，一般并不会存在对象容器内部。存储在对象容器内部的是这些属性的名称，它们就像指针（从技术角度来说就是引用）一样，指向这些值真正的存储位置。

思考下面的代码：

```
var myObject = {
    a: 2
};

myObject.a; // 2

myObject["a"]; // 2
```

如果要访问 myObject 中 a 位置上的值，我们需要使用 . 操作符或者 [] 操作符。.a 语法通常被称为“属性访问”，["a"] 语法通常被称为“键访问”。实际上它们访问的是同一个位置，并且会返回相同的值 2，所以这两个术语是可以互换的。在本书中我们会使用最常见的术语“属性访问”。

这两种语法的主要区别在于 . 操作符要求属性名满足标识符的命名规范，而 [".."] 语法可以接受任意 UTF-8/Unicode 字符串作为属性名。举例来说，如果要引用名称为 "Super-Fun!" 的属性，那就必须使用 ["Super-Fun!"] 语法访问，因为 Super-Fun! 并不是一个有效的标识符属性名。

此外，由于 [".."] 语法使用字符串来访问属性，所以可以在程序中构造这个字符串，比如说：