认绑定，从而把 this 绑定到全局对象或者 undefined 上，取决于是否是严格模式。

思考下面的代码：

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

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

var bar = obj.foo; // 函数别名！

var a = "oops, global"; // a 是全局对象的属性

bar(); // "oops, global"
```

虽然 bar 是 obj.foo 的一个引用，但是实际上，它引用的是 foo 函数本身，因此此时的 bar() 其实是一个不带任何修饰的函数调用，因此应用了默认绑定。

一种更微妙、更常见并且更出乎意料的情况发生在传入回调函数时：

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

function doFoo(fn) {
    // fn 其实引用的是 foo

    fn(); // <-- 调用位置！
}

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

var a = "oops, global"; // a 是全局对象的属性

doFoo( obj.foo ); // "oops, global"
```

参数传递其实就是一种隐式赋值，因此我们传入函数时也会被隐式赋值，所以结果和上一个例子一样。

如果把函数传入语言内置的函数而不是传入你自己声明的函数，会发生什么呢？结果是一样的，没有区别：

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

```
}

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

var a = "oops, global"; // a 是全局对象的属性

setTimeout( obj.foo, 100 ); // "oops, global"
```

JavaScript 环境中内置的 `setTimeout()` 函数实现和下面的伪代码类似：

```
function setTimeout(fn,delay) {
    // 等待 delay 毫秒
    fn(); // <-- 调用位置!
}
```

就像我们看到的那样，回调函数丢失 `this` 绑定是非常常见的。除此之外，还有一种情况 `this` 的行为会出乎我们意料：调用回调函数的函数可能会修改 `this`。在一些流行的 JavaScript 库中事件处理器常会把回调函数的 `this` 强制绑定到触发事件的 DOM 元素上。这在一些情况下可能很有用，但是有时它可能会让你感到非常郁闷。遗憾的是，这些工具通常无法选择是否启用这个行为。

无论是哪种情况，`this` 的改变都是意想不到的，实际上你无法控制回调函数的执行方式，因此就没有办法控制会影响绑定的调用位置。之后我们会介绍如何通过固定 `this` 来修复（这里是双关，“修复”和“固定”的英语单词都是 fixing）这个问题。

### 2.2.3 显式绑定

就像我们刚才看到的那样，在分析*隐式绑定*时，我们必须在一个对象内部包含一个指向函数的属性，并通过这个属性间接引用函数，从而把 `this` 间接（隐式）绑定到这个对象上。

那么如果我们不想在对象内部包含函数引用，而想在某个对象上强制调用函数，该怎么做呢？

JavaScript 中的“所有”函数都有一些有用的特性（这和它们的 [[ 原型 ]] 有关——之后我们会详细介绍原型），可以用来解决这个问题。具体点说，可以使用函数的 `call(..)` 和 `apply(..)` 方法。严格来说，JavaScript 的宿主环境有时会提供一些非常特殊的函数，它们并没有这两个方法。但是这样的函数非常罕见，JavaScript 提供的绝大多数函数以及你自己创建的所有函数都可以使用 `call(..)` 和 `apply(..)` 方法。

这两个方法是如何工作的呢？它们的第一个参数是一个对象，它们会把这个对象绑定到 `this`，接着在调用函数时指定这个 `this`。因为你可以直接指定 `this` 的绑定对象，因此我们称之为显式绑定。

思考下面的代码：

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

var obj = {
    a:2
};

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

通过 foo.call(..)，我们可以在调用 foo 时强制把它的 this 绑定到 obj 上。

如果你传入了一个原始值（字符串类型、布尔类型或者数字类型）来当作 this 的绑定对象，这个原始值会被转换成它的对象形式（也就是 new String(..)、new Boolean(..) 或者 new Number(..)）。这通常被称为“装箱”。

从 this 绑定的角度来说，call(..) 和 apply(..) 是一样的，它们的区别体现在其他的参数上，但是现在我们不用考虑这些。

可惜，显式绑定仍然无法解决我们之前提出的丢失绑定问题。

### 1. 硬绑定

但是显式绑定的一个变种可以解决这个问题。

思考下面的代码：

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

var obj = {
    a:2
};

var bar = function() {
    foo.call( obj );
};

bar(); // 2
setTimeout( bar, 100 ); // 2

// 硬绑定的 bar 不可能再修改它的 this
bar.call( window ); // 2
```

我们来看看这个变种到底是怎样工作的。我们创建了函数 bar()，并在它的内部手动调用了 foo.call(obj)，因此强制把 foo 的 this 绑定到了 obj。无论之后如何调用函数 bar，它总会手动在 obj 上调用 foo。这种绑定是一种显式的强制绑定，因此我们称之为*硬绑定*。

硬绑定的典型应用场景就是创建一个包裹函数，传入所有的参数并返回接收到的所有值：

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a:2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

另一种使用方法是创建一个 i 可以重复使用的辅助函数：

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// 简单的辅助绑定函数
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
    };
}

var obj = {
    a:2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

由于*硬绑定*是一种非常常用的模式，所以在 ES5 中提供了内置的方法 Function.prototype.bind，它的用法如下：

```
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
```

bind(..) 会返回一个硬编码的新函数，它会把参数设置为 this 的上下文并调用原始函数。

#### 2. API调用的“上下文”

第三方库的许多函数，以及 JavaScript 语言和宿主环境中许多新的内置函数，都提供了一个可选的参数，通常被称为“上下文”（context），其作用和 bind(..) 一样，确保你的回调函数使用指定的 this。

举例来说：

```
function foo(el) {
    console.log( el, this.id );
}

var obj = {
    id: "awesome"
};

// 调用 foo(..) 时把 this 绑定到 obj
[1, 2, 3].forEach( foo, obj );
// 1 awesome 2 awesome 3 awesome
```

这些函数实际上就是通过 call(..) 或者 apply(..) 实现了显式绑定，这样你可以少些一些代码。

### 2.2.4 new绑定

这是第四条也是最后一条 this 的绑定规则，在讲解它之前我们首先需要澄清一个非常常见的关于 JavaScript 中函数和对象的误解。

在传统的面向类的语言中，“构造函数”是类中的一些特殊方法，使用 new 初始化类时会调用类中的构造函数。通常的形式是这样的：

```
something = new MyClass(..);
```

JavaScript 也有一个 new 操作符，使用方法看起来也和那些面向类的语言一样，绝大多数开发者都认为 JavaScript 中 new 的机制也和那些语言一样。然而，JavaScript 中 new 的机制实际上和面向类的语言完全不同。

首先我们重新定义一下 JavaScript 中的“构造函数”。在 JavaScript 中，构造函数只是一些使用 new 操作符时被调用的函数。它们并不会属于某个类，也不会实例化一个类。实际上，它们甚至都不能说是一种特殊的函数类型，它们只是被 new 操作符调用的普通函数而已。

举例来说，思考一下 Number(..) 作为构造函数时的行为，ES5.1 中这样描述它：

> 15.7.2　Number 构造函数
> 当 Number 在 new 表达式中被调用时，它是一个构造函数：它会初始化新创建的对象。

所以，包括内置对象函数（比如 Number(..)，详情请查看第 3 章）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。这里有一个重要但是非常细微的区别：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）一个全新的对象。
2. 这个新对象会被执行 [[ 原型 ]] 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。

我们现在关心的是第 1 步、第 3 步、第 4 步，所以暂时跳过第 2 步，第 5 章会详细介绍它。

思考下面的代码：

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

var bar = new foo(2);
console.log( bar.a ); // 2
```

使用 new 来调用 foo(..) 时，我们会构造一个新对象并把它绑定到 foo(..) 调用中的 this 上。new 是最后一种可以影响函数调用时 this 绑定行为的方法，我们称之为 new 绑定。

## 2.3　优先级

现在我们已经了解了函数调用中 this 绑定的四条规则，你需要做的就是找到函数的调用位置并判断应当应用哪条规则。但是，如果某个调用位置可以应用多条规则该怎么办？为了解决这个问题就必须给这些规则设定优先级，这就是我们接下来要介绍的内容。

毫无疑问，默认绑定的优先级是四条规则中最低的，所以我们可以先不考虑它。

隐式绑定和显式绑定哪个优先级更高？我们来测试一下：

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

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

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

可以看到，显式绑定优先级更高，也就是说在判断时应当先考虑是否可以应用显式绑定。

现在我们需要搞清楚 new 绑定和隐式绑定的优先级谁高谁低：

```
function foo(something) {
    this.a = something;
}

var obj1 = {
    foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

可以看到 new 绑定比隐式绑定优先级高。但是 new 绑定和显式绑定谁的优先级更高呢？

new 和 call/apply 无法一起使用，因此无法通过 new foo.call(obj1) 来直接进行测试。但是我们可以使用硬绑定来测试它俩的优先级。

在看代码之前先回忆一下硬绑定是如何工作的。Function.prototype.bind(..) 会创建一个新的包装函数，这个函数会忽略它当前的 this 绑定（无论绑定的对象是什么），并把我们提供的对象绑定到 this 上。

这样看起来硬绑定（也是显式绑定的一种）似乎比 new 绑定的优先级更高，无法使用 new 来控制 this 绑定。

我们看看是不是这样：

```
function foo(something) {
    this.a = something;
}

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar(3);
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

出乎意料！ bar 被硬绑定到 obj1 上，但是 new bar(3) 并没有像我们预计的那样把 obj1.a 修改为 3。相反，new 修改了硬绑定（到 obj1 的）调用 bar(..) 中的 this。因为使用了 new 绑定，我们得到了一个名字为 baz 的新对象，并且 baz.a 的值是 3。

再来看看我们之前介绍的“裸”辅助函数 bind：

```
function bind(fn, obj) {
    return function() {
        fn.apply( obj, arguments );
    };
}
```

非常令人惊讶，因为看起来在辅助函数中 new 操作符的调用无法修改 this 绑定，但是在刚才的代码中 new 确实修改了 this 绑定。

实际上，ES5 中内置的 Function.prototype.bind(..) 更加复杂。下面是 MDN 提供的一种 bind(..) 实现，为了方便阅读我们对代码进行了排版：

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(oThis) {
        if (typeof this !== "function") {
            // 与 ECMAScript 5 最接近的
            // 内部 IsCallable 函数
            throw new TypeError(
                "Function.prototype.bind - what is trying " +
                "to be bound is not callable"
```

```
            );
        }

        var aArgs = Array.prototype.slice.call( arguments, 1 ),
            fToBind = this,
            fNOP = function(){},
            fBound = function(){
                return fToBind.apply(
                    (
                        this instanceof fNOP &&
                        oThis ? this : oThis
                    ),
                    aArgs.concat(
                       Array.prototype.slice.call( arguments )
                    );
            }
        ;

        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP();

        return fBound;
    };
}
```

这种 bind(..) 是一种 polyfill 代码（polyfill 就是我们常说的刮墙用的腻子，polyfill 代码主要用于旧浏览器的兼容，比如说在旧的浏览器中并没有内置 bind 函数，因此可以使用 polyfill 代码在旧浏览器中实现新的功能），对于 new 使用的硬绑定函数来说，这段 polyfill 代码和 ES5 内置的 bind(..) 函数并不完全相同（后面会介绍为什么要在 new 中使用硬绑定函数）。由于 polyfill 并不是内置函数，所以无法创建一个不包含 .prototype 的函数，因此会具有一些副作用。如果你要在 new 中使用硬绑定函数并且依赖 polyfill 代码的话，一定要非常小心。

下面是 new 修改 this 的相关代码：

```
this instanceof fNOP &&
oThis ? this : oThis

// ... 以及：

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();
```

我们并不会详细解释这段代码做了什么（这非常复杂并且不在我们的讨论范围之内），不过简单来说，这段代码会判断硬绑定函数是否是被 new 调用，如果是的话就会使用新创建的 this 替换硬绑定的 this。

那么，为什么要在 new 中使用硬绑定函数呢？直接使用普通函数不是更简单吗？

之所以要在 new 中使用硬绑定函数，主要目的是预先设置函数的一些参数，这样在使用 new 进行初始化时就可以只传入其余的参数。bind(..) 的功能之一就是可以把除了第一个参数（第一个参数用于绑定 this）之外的其他参数都传给下层的函数（这种技术称为“部分应用”，是“柯里化”的一种）。举例来说：

```
function foo(p1,p2) {
        this.val = p1 + p2;
}

// 之所以使用 null 是因为在本例中我们并不关心硬绑定的 this 是什么
// 反正使用 new 时 this 会被修改
var bar = foo.bind( null, "p1" );

var baz = new bar( "p2" );

baz.val; // p1p2
```

## 判断this

现在我们可以根据优先级来判断函数在某个调用位置应用的是哪条规则。可以按照下面的顺序来进行判断：

1. 函数是否在 new 中调用（new 绑定）？如果是的话 this 绑定的是新创建的对象。
   ```
   var bar = new foo()
   ```
2. 函数是否通过 call、apply（显式绑定）或者硬绑定调用？如果是的话，this 绑定的是指定的对象。
   ```
   var bar = foo.call(obj2)
   ```
3. 函数是否在某个上下文对象中调用（隐式绑定）？如果是的话，this 绑定的是那个上下文对象。
   ```
   var bar = obj1.foo()
   ```
4. 如果都不是的话，使用默认绑定。如果在严格模式下，就绑定到 undefined，否则绑定到全局对象。
   ```
   var bar = foo()
   ```

就是这样。对于正常的函数调用来说，理解了这些知识你就可以明白 this 的绑定原理了。不过……凡事总有例外。

# 2.4 绑定例外

规则总有例外，这里也一样。