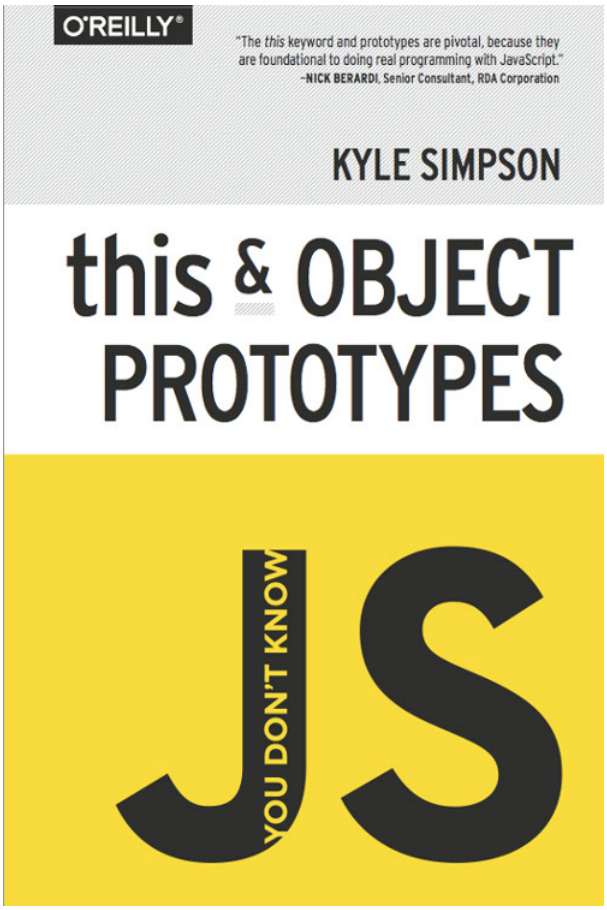


从 O'Reilly 购买数字/印刷版

目录

序

在我读这本书为写这篇序言做准备时，我被迫反思我是如何学习 JavaScript，和在我用它进行编程和开发的最近 15 年它改变了多少。

当我 15 年前开始使用 JavaScript 时，在你的网页上使用 CSS 和 JS 这样的非 HTML 技术的做法称为 DHTML 或动态 HTML。回到那时，JavaScript 的用途有很大的不同，并且倾向于在你的网页上加入动画雪花，或者在状态栏上显示告知时间的动态时钟。可以说，在我的职业生涯早期，因为这些我经常能在因特网上找到的新奇小玩意儿，我真的没有太注意 JavaScript。

直到 2005 年我第一次重新认识到 JavaScript 是一个我需要更加重视的真正的编程语言。在挖掘研究了 Google Maps 的第一个 beta 版后，我被它的潜力吸引住了。那时，Google Maps 是第一个同种类的应用 —— 它允许你用鼠标移动地图，放缩，请求服务器而不必刷新页面 —— 都是通过 JavaScript。它看起来就像魔法！

当什么东西看起来像魔法时，这通常都一个好的信号：你正处在用新方法做事的黎明。噢，我没有错 —— 快进到今天，我敢说 JavaScript 是同时用于客户端和服务器端编程的主要语言之一，而且我不会用其他方式这么说。

在我回顾过去的 15 年时，我的一个遗憾是在 2005 年以前我没有给 JavaScript 更多机会，或者更确切地说，我缺乏远见来看到 JavaScript 是一个真正的编程语言，就像 C++，C#，Java 和许多其他语言一样有用。

如果我在自己的职业生涯一开始就拥有这套 *你不懂 JS* 系列丛书，我们的职业经历将和今天有很大的不同。我喜欢这个系列的一个地方是：当你通读这个系列时，它在建立你的理解的水平上，而且用一种有趣且信息丰富的方式讲解 JS。

*this 与对象原型*是这个系列的一个绝妙的续作。它漂亮且自然地建立于前一本书之上，作用域与闭包，将知识扩展至 JS 语言中十分重要的部分，`this` 关键字和原型。这两个简单的东西是你将在未来的书中学到的东西的枢纽，因为他们是用 JavaScript 进行真正的编程的基础。如何创建对象，关联它们，和扩展它们来表达你的应用中的东西，是用 JavaScript 建立大型和复杂应用程序所必要的。没有它们，用 JavaScript 制造复杂应用程序（比如 Google Maps）将是不可能的。

我敢说绝大多数 web 开发者可能从没建立过 JavaScript 对象，而只是将这个语言当做按钮和 AJAX 请求的事件绑定胶水。我曾经在我职业生涯的某一点上属于这个群体，但是当我学习了如何掌握原型和在 JavaScript 中创建对象后，一个充满可能性的世界向我打开了大门。如果你属于仅仅会写事件绑定胶水代码的那一类，这本书是必读的；如果你只是需要进修，这本书是你一定会用到的资源。不管怎样，你不会失望的。相信我！

Nick Berardi nickberardi.com, [@nberardi](http://twitter.com/nberardi)

第一章：`this` 是什么？

JavaScript 中最令人困惑的机制之一就是 `this` 关键字。它是一个在每个函数作用域中自动定义的特殊标识符关键字，但即便是一些老练的 JavaScript 开发者也对它到底指向什么感到困

扰。

> 任何足够先进的技术都跟魔法没有区别。– Arthur C. Clarke

JavaScript 的 this 机制实际上没有那么先进，但是开发者们总是在大脑中插入“复杂”和“混乱”来解释这句话，毫无疑问，如果没有清晰的理解，在你的困惑中 this 可能看起来就是彻头彻尾的魔法。

注意： “this”这个词是在一般的论述中极常用的代词。所以，特别是在口头论述中，很难确定我们是在将“this”作为一个代词使用，还是在将它作为一个实际的关键字标识符使用。为了表意清晰，我会总是使用 this 来代表特殊的关键字，而在其他情况下使用“this”或 *this* 或 this。

为什么要用 this?

如果对于那些老练的 JavaScript 开发者来说 this 机制都是如此的令人费解，那么有人会问为什么这种机制会有用? 它带来的麻烦不是比好处多吗? 在讲解如何有用之前，我们应当先来看看为什么有用。

让我们试着展示一下 this 的动机和用途:

```
function identify() {
    return this.name.toUpperCase();
}

function speak() {
    var greeting = "Hello, I'm " + identify.call( this );
    console.log( greeting );
}

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify.call( me ); // KYLE
identify.call( you ); // READER
```

```
speak.call( me ); // Hello, I'm KYLE
speak.call( you ); // Hello, I'm READER
```

如果这个代码段 如何工作让你困惑，不要担心！我们很快就会讲解它。只是简要地将这些问题放在旁边，以便于我们可以更清晰的探究 为什么。

这个代码片段允许 identify() 和 speak() 函数对多个 环境对象（me 和 you）进行复用，而不是针对每个对象定义函数的分离版本。

与使用 this 相反地，你可以明确地将环境对象传递给 identify() 和 speak()。

```
function identify(context) {
    return context.name.toUpperCase();
}

function speak(context) {
    var greeting = "Hello, I'm " + identify( context );
    console.log( greeting );
}

identify( you ); // READER
speak( me ); // Hello, I'm KYLE
```

然而，this 机制提供了更优雅的方式来隐含地 “传递” 一个对象引用，导致更加干净的 API 设计和更容易的复用。

你的使用模式越复杂，你就会越清晰地看到：将执行环境作为一个明确参数传递，通常比传递 this 执行环境要乱。当我们探索对象和原型时，你将会看到一组可以自动引用恰当执行环境对象的函数是多么有用。

困惑

我们很快就要开始讲解 this 是如何 实际工作的，但我们首先要摒弃一些误解——它实际上 不是如何工作的。

在开发者们用太过于字面的方式考虑 “this” 这个名字时就会产生困惑。这通常会产生两种臆测，但都是不对的。

它自己

第一种常见的倾向是认为 this 指向函数自己。至少，这是一种语法上的合理推测。

为什么你想要在函数内部引用它自己？最常见的理由是递归（在函数内部调用它自己）这样的情形，或者是一个在第一次被调用时会解除自己绑定的事件处理器。

初次接触 JS 机制的开发者们通常认为，将函数作为一个对象（JavaScript 中所有的函数都是对象!），可以让你在方法调用之间储存 状态（属性中的值）。这当然是可能的，而且有一些有

限的用处，但这本书的其余部分将会阐述许多其他的模式，提供比函数对象 更好的地方来存储状态。

过一会儿我们将探索一个模式，来展示 this 是如何不让一个函数像我们可能假设的那样，得到它自身的引用的。

考虑下面的代码，我们试图追踪函数 (foo) 被调用了多少次：

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

    // 追踪 `foo` 被调用了多少次
    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 -- 这他妈怎么回事.....?
```

foo.count 依然是 0, 即便四个 console.log 语句明明告诉我们 foo(..) 实际上被调用了四次。这种挫败来源于对于 this（在 this.count++ 中）的含义进行了 过于字面化的解释。

当代码执行 foo.count = 0 时，它确实向函数对象 foo 添加了一个 count 属性。但是对于函数内部的 this.count 引用，this 其实 根本就不指向那个函数对象，即便属性名称一样，但根对象也不同，因而产生了混淆。

注意：一个负责任的开发者 应当在这里提出一个问题：“如果我递增的 count 属性不是我以为的那个，那是哪个 count 被我递增了？”。实际上，如果他再挖的深一些，他会发现自己不小心创建了一个全局变量 count（第二章解释了这是 如何发生的！），而且它当前的值是 NaN。

当然，一旦他发现这个不寻常的结果后，他会有一堆其他的问题："它怎么是全局的？为什么它是 NaN 而不是某个正确的计数值？"。（见第二章）

与停在这里来深究为什么 this 引用看起来不是如我们 期待的那样工作，并且回答那些尖锐且重要的问题相反，许多开发者简单地完全回避这个问题，转向一些其他的另类解决方法，比如创建另一个对象来持有 count 属性：

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

    // 追踪 `foo` 被调用了多少次
    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
```

虽然这种方式"解决"了问题是事实，但不幸的是它简单地忽略了真正的问题 —— 缺乏对于 this 的含义和其工作方式上的理解 —— 反而退回到了一个他更加熟悉的机制的舒适区：词法作用域。

注意：词法作用域是一个完善且有用的机制；我不是在用任何方式贬低它的作用（参见本系列的"作用域与闭包"）。但在如何使用 this 这个问题上总是靠 猜，而且通常都猜 错，并不是一个退回到词法作用域，而且从不学习 为什么 this 不跟你合作的好理由。

为了从函数对象内部引用它自己，一般来说通过 this 是不够的。你通常需要通过一个指向它的词法标识符（变量）得到函数对象的引用。

考虑这两个函数:

```
function foo() {
    foo.count = 4; // `foo` 引用它自己
}

setTimeout( function(){
    // 匿名函数（没有名字）不能引用它自己
}, 10 );
```

第一个函数，称为 “命名函数”，foo 是一个引用，可以用于在它内部引用自己。

但是在第二个例子中，传递给 setTimeout(..) 的回调函数没有名称标识符（所以被称为 “匿名函数”），所以没有合适的办法引用函数对象自己。

注意: 在函数中有一个老牌儿但是现在被废弃的，而且令人皱眉头的 arguments.callee 引用 也指向当前正在执行的函数的函数对象。这个引用通常是匿名函数在自己内部访问函数对象的唯一方法。然而，最佳的办法是完全避免使用匿名函数，至少是对于那些需要自引用的函数，而使用命名函数（表达式）。arguments.callee 已经被废弃而且不应该再使用。

对于当前我们的例子来说，另一个 好用的解决方案是在每一个地方都使用 foo 标识符作为函数对象的引用，而根本不用 this:

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

    // 追踪 `foo` 被调用了多少次
    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，这种方法也类似地回避了对 this 的 真正理解，而且完全依靠变量 foo 的词法作用域。

另一种解决这个问题的方法是强迫 this 指向 foo 函数对象:

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

    // 追踪 `foo` 被调用了多少次
    // 注意: 由于 `foo` 的被调用方式（见下方），`this` 现在确实是 `foo`
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        // 使用 `call(..)`，我们可以保证 `this` 指向函数对象 (`foo`)
        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
```

与回避 this 相反，我们接受它。我们马上将会更完整地讲解这样的技术 如何工作，所以如果你依然有点儿糊涂，不要担心!

它的作用域

对 this 的含义第二常见的误解，是它不知怎的指向了函数的作用域。这是一个刁钻的问题，因为在某一种意义上它有正确的部分，而在另外一种意义上，它是严重的误导。

明确地说，this 不会以任何方式指向函数的 **词法作用域**。作用域好像是一个将所有可用标识符作为属性的对象，这从内部来说是对的。但是 JavasScript 代码不能访问作用域 “对象”。它是 *引擎*的内部实现。

考虑下面代码，它（失败的）企图跨越这个边界，用 this 来隐含地引用函数的词法作用域：

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

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

foo(); //undefined
```

这个代码段里不只有一个错误。虽然它看起来是在故意瞎搞，但你看到的这段代码，提取自在公共社区的帮助论坛中被交换的真实代码。真是难以想象对 this 的臆想是多么的误导人。

首先，试图通过 this.bar() 来引用 bar() 函数。它几乎可以说是 *碰巧*能够工作，我们过一会儿再解释它是 *如何*工作的。调用 bar() 最自然的方式是省略开头的 this.，而仅使用标识符进行词法引用。

然而，写下这段代码的开发者试图用 this 在 foo() 和 bar() 的词法作用域间建立一座桥，使得 bar() 可以访问 foo() 内部作用域的变量 a。**这样的桥是不可能的**。你不能使用 this 引用在词法作用域中查找东西。这是不可能的。

每当你感觉自己正在试图使用 this 来进行词法作用域的查询时，提醒你自己：*这里没有桥*。

什么是 this?

我们已经列举了各种不正确的臆想，现在让我们把注意力转移到 this 机制是如何真正工作的。

我们早先说过，this 不是编写时绑定，而是运行时绑定。它依赖于函数调用的上下文条件。this 绑定与函数声明的位置没有任何关系，而与函数被调用的方式紧密相连。

当一个函数被调用时，会建立一个称为执行环境的活动记录。这个记录包含函数是从何处（调用栈 —— call-stack）被调用的，函数是 *如何*被调用的，被传递了什么参数等信息。这个记录的属性之一，就是在函数执行期间将被使用的 this 引用。

下一章中，我们将会学习寻找函数的 **调用点（call-site）**来判定它的执行如何绑定 `this`。

复习

对于那些没有花时间学习 `this` 绑定机制如何工作的 JavaScript 开发者来说，`this` 绑定一直是困惑的根源。对于 `this` 这么重要的机制来说，猜测、试错、或者盲目地从 Stack Overflow 的回答中复制粘贴，都不是有效或正确利用它的方法。

为了学习 `this`，你必须首先学习 `this` *不是*什么，不论是哪种把你误导至何处的臆测或误解。`this` 既不是函数自身的引用，也不是函数 *词法*作用域的引用。

`this` 实际上是在函数被调用时建立的一个绑定，它指向 *什么*是完全由函数被调用的调用点来决定的。

第二章: this 豁然开朗!

在第一章中，我们摒弃了种种对 `this` 的误解，并且知道了 `this` 是一个完全根据**调用点**（函数是如何被调用的）而为每次函数调用建立的绑定。

调用点（Call-site）

为了理解 `this` 绑定，我们不得不理解调用点：函数在代码中被调用的位置（**不是被声明的位置**）。我们必须考察调用点来回答这个问题：这个 `this` 指向什么?

一般来说寻找调用点就是："找到一个函数是在哪里被调用的"，但它不总是那么简单，比如某些特定的编码模式会使 *真正的*调用点变得不那么明确。

考虑 **调用栈（call-stack）**（使我们到达当前执行位置而被调用的所有方法的堆栈）是十分重要的。我们关心的调用点就位于当前执行中的函数 *之前*的调用。

我们来展示一下调用栈和调用点：

```
function baz() {
    // 调用栈是: `baz`
    // 我们的调用点是 global scope（全局作用域）

    console.log( "baz" );
    bar(); // <-- `bar` 的调用点
}

function bar() {
    // 调用栈是: `baz` -> `bar`
    // 我们的调用点位于 `baz`

    console.log( "bar" );
```

```
    foo(); // <-- `foo` 的 call-site
}

function foo() {
    // 调用栈是: `baz` -> `bar` -> `foo`
    // 我们的调用点位于 `bar`

    console.log( "foo" );
}

baz(); // <-- `baz` 的调用点
```

在分析代码来寻找（从调用栈中）真正的调用点时要小心，因为它是影响 this 绑定的唯一因素。

注意：你可以通过按顺序观察函数的调用链在你的大脑中建立调用栈的视图，就像我们在上面代码段中的注释那样。但是这很痛苦而且易错。另一种观察调用栈的方式是使用你的浏览器的调试工具。大多数现代的桌面浏览器都内建开发者工具，其中就包含 JS 调试器。在上面的代码段中，你可以在调试工具中为 foo() 函数的第一行设置一个断点，或者简单的在这第一行上插入一个 debugger 语句。当你运行这个网页时，调试工具将会停止在这个位置，并且向你展示一个到达这一行之前所有被调用过的函数的列表，这就是你的调用栈。所以，如果你想调查 this 绑定，可以使用开发者工具取得调用栈，之后从上向下找到第二个记录，那就是你真正的调用点。

仅仅是规则

现在我们将注意力转移到调用点 如何决定在函数执行期间 this 指向哪里。

你必须考察调用点并判定 4 种规则中的哪一种适用。我们将首先独立地解释一下这 4 种规则中的每一种，之后我们来展示一下如果有多种规则可以适用于调用点时，它们的优先顺序。

默认绑定（Default Binding）

我们要考察的第一种规则源于函数调用的最常见的情况：独立函数调用。可以认为这种 this 规则是在没有其他规则适用时的默认规则。

考虑这个代码段：

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

var a = 2;
```

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

第一点要注意的，如果你还没有察觉到，是在全局作用域中的声明变量，也就是 var a = 2，是全局对象的同名属性的同义词。它们不是互相拷贝对方，它们 就是彼此。正如一个硬币的两面。

第二，我们看到当 foo() 被调用时，this.a 解析为我们的全局变量 a。为什么? 因为在这种情况下，对此方法调用的 this 实施了 默认绑定，所以使 this 指向了全局对象。

我们怎么知道这里适用 默认绑定? 我们考察调用点来看看 foo() 是如何被调用的。在我们的代码段中，foo() 是被一个直白的，毫无修饰的函数引用调用的。没有其他的我们将要展示的规则适用于这里，所以 默认绑定在这里适用。

如果 strict mode 在这里生效，那么对于 默认绑定来说全局对象是不合法的，所以 this 将被设置为 undefined。

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

    console.log( this.a );
}

var a = 2;

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

一个微妙但是重要的细节是: 即便所有的 this 绑定规则都是完全基于调用点的，但如果 foo() 的 **内容**没有在 strict mode 下执行，对于 默认绑定来说全局对象是 **唯**一合法的; foo() 的调用点的 strict mode 状态与此无关。

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

var a = 2;

(function(){
    "use strict";

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

注意：在你的代码中故意混用 `strict mode` 和非 `strict mode` 通常是让人皱眉头的。你的程序整体可能应当不是 **Strict** 就是 **非 Strict**。然而，有时你可能会引用与你的 **Strict** 模式不同的第三方包，所以对这些微妙的兼容性细节要多加小心。

隐含绑定（Implicit Binding）

另一种要考虑的规则是：调用点是否有一个环境对象（context object），也称为拥有者（owning）或容器（containing）对象，虽然这些名词可能有些误导人。

考虑这段代码：

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

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

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

首先，注意 `foo()` 被声明然后作为引用属性添加到 `obj` 上的方式。无论 `foo()` 是否一开始就在 `obj` 上被声明，还是后来作为引用添加（如上面代码所示），这个 **函数**都不被 `obj` 所真正“拥有”或“包含”。

然而，调用点 *使用* `obj` *环境来* **引用**函数，所以你 *可以说* `obj` 对象在函数被调用的时间点上“拥有”或“包含”这个 **函数引用**。

不论你怎样称呼这个模式，在 `foo()` 被调用的位置上，它被冠以一个指向 `obj` 的对象引用。当一个方法引用存在一个环境对象时，*隐含绑定*规则会说：是这个对象应当被用于这个函数调用的 `this` 绑定。

因为 `obj` 是 `foo()` 调用的 `this`，所以 `this.a` 就是 `obj.a` 的同义词。

只有对象属性引用链的最后一层是影响调用点的。比如：

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

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

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

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

隐含丢失 (Implicitly Lost)

this 绑定最常让人沮丧的事情之一，就是当一个 隐含绑定丢失了它的绑定，这通常意味着它会退回到 默认绑定，根据 strict mode 的状态，其结果不是全局对象就是 undefined。

考虑这段代码:

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

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

var bar = obj.foo; // 函数引用!

var a = "oops, global"; // `a` 也是一个全局对象的属性

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

尽管 bar 似乎是 obj.foo 的引用，但实际上它只是另一个 foo 本身的引用而已。另外，起作用的调用点是 bar()，一个直白，毫无修饰的调用，因此 默认绑定适用于这里。

这种情况发生的更加微妙，更常见，而且更意外的方式，是当我们考虑传递一个回调函数时:

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

function doFoo(fn) {
    // `fn` 只不过 `foo` 的另一个引用
```

```
    fn(); // <-- 调用点!
}

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

var a = "oops, global"; // `a` 也是一个全局对象的属性

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

参数传递仅仅是一种隐含的赋值，而且因为我们在传递一个函数，它是一个隐含的引用赋值，所以最终结果和我们前一个代码段一样。

那么如果接收你所传递回调的函数不是你的，而是语言内建的呢？没有区别，同样的结果。

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

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

var a = "oops, global"; // `a` 也是一个全局对象的属性

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

把这个粗糙的，理论上的 `setTimeout()` 假想实现当做 JavaScript 环境内建的实现的话：

```
function setTimeout(fn,delay) {
  // (通过某种方法) 等待 `delay` 毫秒
    fn(); // <-- 调用点!
}
```

正如我们刚刚看到的，我们的回调函数丢掉他们的 `this` 绑定是十分常见的事情。但是 `this` 使我们吃惊的另一种方式是，接收我们回调的函数故意改变调用的 `this`。那些很流行的

JavaScript 库中的事件处理器就十分喜欢强制你的回调的 `this` 指向触发事件的 DOM 元素。虽然有时这很有用，但其他时候这简直能气死人。不幸的是，这些工具很少给你选择。

不管哪一种意外改变 `this` 的方式，你都不能真正地控制你的回调函数引用将如何被执行，所以你（还）没有办法控制调用点给你一个故意的绑定。我们很快就会看到一个方法，通过 `固定 this` 来解决这个问题。

明确绑定（Explicit Binding）

用我们刚看到的 `隐含绑定`，我们不得不改变目标对象使它自身包含一个对函数的引用，而后使用这个函数引用属性来间接地（隐含地）将 `this` 绑定到这个对象上。

但是，如果你想强制一个函数调用使用某个特定对象作为 `this` 绑定，而不在这个对象上放置一个函数引用属性呢?

JavaScript 语言中的 “所有” 函数都有一些工具（通过他们的 `[[Prototype]]` —— 待会儿详述）可以用于这个任务。具体地说，函数拥有 `call(..)` 和 `apply(..)` 方法。从技术上讲，JavaScript 宿主环境有时会提供一些（说得好听点儿!）很特别的函数，它们没有这些功能。但这很少见。绝大多数被提供的函数，当然还有你将创建的所有的函数，都可以访问 `call(..)` 和 `apply(..)`。

这些工具如何工作? 它们接收的第一个参数都是一个用于 `this` 的对象，之后使用这个指定的 `this` 来调用函数。因为你已经直接指明你想让 `this` 是什么，所以我们称这种方式为 `明确绑定`（*explicit binding*）。

考虑这段代码:

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

var obj = {
    a: 2
};

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

通过 `foo.call(..)` 使用 `明确绑定`来调用 `foo`，允许我们强制函数的 `this` 指向 `obj`。

如果你传递一个简单基本类型值（`string`，`boolean`，或 `number` 类型）作为 `this` 绑定，那么这个基本类型值会被包装在它的对象类型中（分别是 `new String(..)`，`new Boolean(..)`，或 `new Number(..)`）。这通常称为 “封箱（boxing）”。

注意: 就 `this` 绑定的角度讲，`call(..)` 和 `apply(..)` 是完全一样的。它们确实在处理其他参数上的方式不同，但那不是我们当前关心的。

不幸的是，单独依靠 `明确绑定`仍然不能为我们先前提到的问题提供解决方案，也就是函数 “丢失” 自己原本的 `this` 绑定，或者被第三方框架覆盖，等等问题。

硬绑定 (Hard Binding)

但是有一个 明确绑定的变种确实可以实现这个技巧。考虑这段代码:

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

var obj = {
    a: 2
};

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

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

// `bar` 将 `foo` 的 `this` 硬绑定到 `obj`
// 所以它不可以被覆盖
bar.call( window ); // 2
```

我们来看看这个变种是如何工作的。我们创建了一个函数 bar(), 在它的内部手动调用 foo.call(obj), 由此强制 this 绑定到 obj 并调用 foo。无论你过后怎样调用函数 bar, 它总是手动使用 obj 调用 foo。这种绑定即明确又坚定, 所以我们称之为 硬绑定 (*hard binding*)

用 硬绑定将一个函数包装起来的最典型的方法, 是为所有传入的参数和传出的返回值创建一个通道:

```
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
```

另一种表达这种模式的方法是创建一个可复用的帮助函数:

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

// 简单的 `bind` 帮助函数
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*是一个如此常用的模式,它已作为 ES5 的内建工具提供:`Function.prototype.bind`,像这样使用:

```
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(..) 返回一个硬编码的新函数，它使用你指定的 this 环境来调用原本的函数。

注意：在 ES6 中，bind(..) 生成的硬绑定函数有一个名为 .name 的属性，它源自于原始的 目标函数（*target function*）。举例来说：bar = foo.bind(..) 应该会有一个 bar.name 属性，它的值为 "bound foo"，这个值应当会显示在调用栈轨迹的函数调用名称中。

API 调用的“环境”

确实，许多库中的函数，和许多在 JavaScript 语言以及宿主环境中的内建函数，都提供一个可选参数，通常称为“环境（context）”，这种设计作为一种替代方案来确保你的回调函数使用特定的 this 而不必非得使用 bind(..)。

举例来说：

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

var obj = {
    id: "awesome"
};

// 使用 `obj` 作为 `this` 来调用 `foo(..)`
[1, 2, 3].forEach( foo, obj ); // 1 awesome  2 awesome  3 awesome
```

从内部来说，几乎可以确定这种类型的函数是通过 call(..) 或 apply(..) 来使用 明确绑定以节省你的麻烦。

new 绑定（new Binding）

第四种也是最后一种 this 绑定规则，要求我们重新思考 JavaScript 中关于函数和对象的常见误解。

在传统的面向类语言中，“构造器”是附着在类上的一种特殊方法，当使用 new 操作符来初始化一个类时，这个类的构造器就会被调用。通常看起来像这样：

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

JavaScript 拥有 `new` 操作符，而且使用它的代码模式看起来和我们在面向类语言中看到的基本一样；大多数开发者猜测 JavaScript 机制在做某种相似的事情。但是，实际上 JavaScript 的机制和 `new` 在 JS 中的用法所暗示的面向类的功能 *没有任何联系*。

首先，让我们重新定义 JavaScript 的“构造器”是什么。在 JS 中，构造器 **仅仅是一个函数**，它们偶然地与前置的 `new` 操作符一起调用。它们不依附于类，它们也不初始化一个类。它们甚至不是一种特殊的函数类型。它们本质上只是一般的函数，在被使用 `new` 来调用时改变了行为。

例如，引用 ES5.1 的语言规范，`Number(..)` 函数作为一个构造器来说：

> 15.7.2 Number 构造器
>
> 当 Number 作为 new 表达式的一部分被调用时，它是一个构造器：它初始化这个新创建的对象。

所以，可以说任何函数，包括像 `Number(..)`（见第三章）这样的内建对象函数都可以在前面加上 `new` 来被调用，这使函数调用成为一个 *构造器调用*（*constructor call*）。这是一个重要而微妙的区别：实际上不存在“构造器函数”这样的东西，而只有函数的构造器调用。

当在函数前面被加入 `new` 调用时，也就是构造器调用时，下面这些事情会自动完成：

1. 一个全新的对象会凭空创建（就是被构建）
2. *这个新构建的对象会被接入原形链*（*[[Prototype]]-linked*）
3. 这个新构建的对象被设置为函数调用的 `this` 绑定
4. 除非函数返回一个它自己的其他 **对象**，否则这个被 `new` 调用的函数将 *自动*返回这个新构建的对象。

步骤 1，3 和 4 是我们当下要讨论的。我们现在跳过第 2 步，在第五章回过头来讨论。

考虑这段代码：

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

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

通过在前面使用 `new` 来调用 `foo(..)`，我们构建了一个新的对象并把这个新对象作为 `foo(..)` 调用的 `this`。**`new` 是函数调用可以绑定 `this` 的最后一种方式**，我们称之为 *new* *绑定*（*new binding*）。

一切皆有顺序

如此，我们已经揭示了函数调用中的四种 this 绑定规则。你需要做的 一切就是找到调用点然后考察哪一种规则适用于它。但是，如果调用点上有多种规则都适用呢？这些规则一定有一个优先顺序，我们下面就来展示这些规则以什么样的优先顺序实施。

很显然，默认绑定在四种规则中优先权最低的。所以我们先把它放在一边。

隐含绑定和 明确绑定哪一个更优先呢？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
```

所以，明确绑定的优先权要高于 隐含绑定，这意味着你应当在考察 隐含绑定之前 **首先**考察 明确绑定是否适用。

现在，我们只需要搞清楚 *new* 绑定的优先级位于何处。

```
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
```

好了，*new* 绑定的优先级要高于 隐含绑定。那么你觉得 *new* 绑定的优先级较之于 明确绑定是高还是低呢?

注意: new 和 call/apply 不能同时使用，所以 new foo.call(obj1) 是不允许的，也就是不能直接对比测试 *new* 绑定和 明确绑定。但是我们依然可以使用 硬绑定来测试这两个规则的优先级。

在我们进入代码中探索之前,回想一下 硬绑定物理上是如何工作的,也就是 Function.prototype.bind(..) 创建了一个新的包装函数，这个函数被硬编码为忽略它自己的 this 绑定（不管它是什么），转而手动使用我们提供的。

因此，这似乎看起来很明显，硬绑定（明确绑定的一种）的优先级要比 *new* 绑定高，而且不能被 new 覆盖。

我们检验一下:

```
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
```

哇！bar 是硬绑定到 obj1 的，但是 new bar(3) 并 **没有**像我们期待的那样将 obj1.a 变为 3。反而，*硬绑定*（到 obj1）的 bar(..) 调用 **可以**被 new 所覆盖。因为 new 被实施，我们得到一个名为 baz 的新创建的对象，而且我们确实看到 baz.a 的值为 3。

如果你回头看看我们的“山寨”绑定帮助函数，这很令人吃惊：

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

如果你推导这段帮助代码如何工作，会发现对于 new 操作符调用来说没有办法去像我们观察到的那样，将绑定到 obj 的硬绑定覆盖。

但是 ES5 的内建 Function.prototype.bind(..) 更加精妙，实际上十分精妙。这里是 MDN 网页上为 bind(..) 提供的（稍稍格式化后的）polyfill（低版本兼容填补工具）：

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(oThis) {
        if (typeof this !== "function") {
            // 可能的与 ECMAScript 5 内部的 IsCallable 函数最接近的东西，
            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;
    };
}
```

注意: 就将与 new 一起使用的硬绑定函数（参照下面来看为什么这有用）而言，上面的 bind(..) polyfill 与 ES5 中内建的 bind(..) 是不同的。因为 polyfill 不能像内建工具那样，没有 .prototype 就能创建函数，这里使用了一些微妙而间接的方法来近似模拟相同的行为。如果你打算将硬绑定函数和 new 一起使用而且依赖于这个 polyfill，应当多加小心。

允许 new 进行覆盖的部分是这里:

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

// ... 和:

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

我们不会实际深入解释这个花招儿是如何工作的（这很复杂而且超出了我们当前的讨论范围），但实质上这个工具判断硬绑定函数是否是通过 new 被调用的（导致一个新构建的对象作为它的 this），如果是，它就用那个新构建的 this 而非先前为 this 指定的 硬绑定。

为什么 new 可以覆盖 硬绑定这件事很有用?

这种行为的主要原因是，创建一个实质上忽略 this 的 硬绑定而预先设置一部分或所有的参数的函数（这个函数可以与 new 一起使用来构建对象）。bind(..) 的一个能力是，任何在第一个 this 绑定参数之后被传入的参数，默认地作为当前函数的标准参数（技术上这称为 “局部应用（partial application）”，是一种 “柯里化（currying）”）。

例如:

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

// 在这里使用 `null` 是因为在这种场景下我们不关心 `this` 的硬绑定
// 而且反正它将会被 `new` 调用覆盖掉!
var bar = foo.bind( null, "p1" );
```

```
var baz = new bar( "p2" );

baz.val; // p1p2
```

判定 this

现在，我们可以按照优先顺序来总结一下从函数调用的调用点来判定 this 的规则了。按照这个顺序来问问题，然后在第一个规则适用的地方停下。

1. 函数是通过 new 被调用的吗（**new 绑定**）？如果是，this 就是新构建的对象。

   ```
   var bar = new foo()
   ```

2. 函数是通过 call 或 apply 被调用（**明确绑定**），甚至是隐藏在 bind 硬绑定之中吗？如果是，this 就是那个被明确指定的对象。

   ```
   var bar = foo.call( obj2 )
   ```

3. 函数是通过环境对象（也称为拥有者或容器对象）被调用的吗（**隐含绑定**）？如果是，this 就是那个环境对象。

   ```
   var bar = obj1.foo()
   ```

4. 否则，使用默认的 this（**默认绑定**）。如果在 strict mode 下，就是 undefined，否则是 global 对象。

   ```
   var bar = foo()
   ```

以上，就是理解对于普通的函数调用来说的 this 绑定规则 所需的全部。是的……几乎是全部。

绑定的特例

正如通常的那样，对于“规则”总有一些 例外。

在某些场景下 this 绑定会让人很吃惊，比如在你试图实施一种绑定，然而最终得到的却是默认绑定规则的绑定行为（见前面的内容）。

被忽略的 this

如果你传递 null 或 undefined 作为 call、apply 或 bind 的 this 绑定参数，那么这些值会被忽略掉，取而代之的是 默认绑定规则将适用于这个调用。

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

```
var a = 2;

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

为什么你会向 this 绑定故意传递像 null 这样的值?

一个很常见的做法是，使用 apply(..) 来将一个数组散开，从而作为函数调用的参数。相似地，bind(..) 可以柯里化参数（预设值），也可能非常有用。

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

// 将数组散开作为参数
foo.apply( null, [2, 3] ); // a:2, b:3

// 用 `bind(..)` 进行柯里化
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3
```

这两种工具都要求第一个参数是 this 绑定。如果目标函数不关心 this，你就需要一个占位值，而且正如这个代码段中展示的，null 看起来是一个合理的选择。

注意: 虽然我们在这本书中没有涵盖，但是 ES6 中有一个扩散操作符: ...，它让你无需使用 apply(..) 而在语法上将一个数组 “散开” 作为参数，比如 foo(...[1,2]) 表示 foo(1,2) —— 如果 this 绑定没有必要，可以在语法上回避它。不幸的是，柯里化在 ES6 中没有语法上的替代品，所以 bind(..) 调用的 this 参数依然需要注意。

可是，在你不关心 this 绑定而一直使用 null 的时候，有些潜在的 “危险”。如果你这样处理一些函数调用（比如，不归你管控的第三方包），而且那些函数确实使用了 this 引用，那么默认绑定规则意味着它可能会不经意间引用（或者改变，更糟糕!）global 对象（在浏览器中是 window）。

很显然，这样的陷阱会导致多种 非常难诊断和追踪的 Bug。

更安全的 this

也许某些 “更安全” 的做法是: 为了 this 而传递一个特殊创建好的对象，这个对象保证不会对你的程序产生副作用。从网络学（或军事）上借用一个词，我们可以建立一个 “DMZ”（非军事区）对象 —— 只不过是一个完全为空，没有委托（见第五，六章）的对象。

如果我们为了忽略自己认为不用关心的 this 绑定，而总是传递一个 DMZ 对象，那么我们就可以确定任何对 this 的隐藏或意外的使用将会被限制在这个空对象中，也就是说这个对象将 global 对象和副作用隔离开来。

因为这个对象是完全为空的，我个人喜欢给它一个变量名为 ø（空集合的数学符号的小写）。在许多键盘上（比如 Mac 的美式键盘），这个符号可以很容易地用 +o（option+o）打出来。有些系统还允许你为某个特殊符号设置快捷键。如果你不喜欢 ø 符号，或者你的键盘没那么好打，你当然可以叫它任意你希望的名字。

无论你叫它什么，创建 **完全为空的对象**的最简单方法就是 `Object.create(null)`（见第五章）。`Object.create(null)` 和 `{}` 很相似，但是没有指向 `Object.prototype` 的委托，所以它比 `{}` “空得更彻底”。

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

// 我们的 DMZ 空对象
var ø = Object.create( null );

// 将数组散开作为参数
foo.apply( ø, [2, 3] ); // a:2, b:3

// 用 `bind(..)` 进行 currying
var bar = foo.bind( ø, 2 );
bar( 3 ); // a:2, b:3
```

不仅在功能上更“安全”，ø 还会在代码风格上产生些好处，它在语义上可能会比 `null` 更清晰的表达“我想让 `this` 为空”。当然，你可以随自己喜欢来称呼你的 DMZ 对象。

间接

另外一个要注意的是，你可以（有意或无意地！）创建对函数的“间接引用（indirect reference）”，在那样的情况下，当那个函数引用被调用时，默认绑定规则也会适用。

一个最常见的 间接引用产生方式是通过赋值：

```
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 结果值是一个刚好指向底层函数对象的引用。如此，起作用的调用点就是 foo()，而非你期待的 p.foo() 或 o.foo()。根据上面的规则，默认绑定适用。

提醒：无论你如何得到适用 默认绑定的函数调用，被调用函数的 内容的 strict mode 状态 —— 而非函数的调用点 —— 决定了 this 引用的值：不是 global 对象（在非 strict mode 下），就是 undefined（在 strict mode 下）。

软化绑定（Softening Binding）

我们之前看到 硬绑定是一种通过将函数强制绑定到特定的 this 上，来防止函数调用在不经意间退回到 默认绑定的策略（除非你用 new 去覆盖它!）。问题是，硬绑定极大地降低了函数的灵活性，阻止我们手动使用 隐含绑定或后续的 明确绑定来覆盖 this。

如果有这样的办法就好了：为 默认绑定提供不同的默认值（不是 global 或 undefined），同时保持函数可以通过 隐含绑定或 明确绑定技术来手动绑定 this。

我们可以构建一个所谓的 软绑定工具来模拟我们期望的行为。

```
if (!Function.prototype.softBind) {
    Function.prototype.softBind = function(obj) {
        var fn = this,
            curried = [].slice.call( arguments, 1 ),
            bound = function bound() {
                return fn.apply(
                    (!this ||
                        (typeof window !== "undefined" &&
                            this === window) ||
                        (typeof global !== "undefined" &&
                            this === global)
                    ) ? obj : this,
                    curried.concat.apply( curried, arguments )
                );
            };
        bound.prototype = Object.create( fn.prototype );
        return bound;
    };
}
```

这里提供的 softBind(..) 工具的工作方式和 ES5 内建的 bind(..) 工具很相似，除了我们的 软绑定行为。它用一种逻辑将指定的函数包装起来，这个逻辑在函数调用时检查 this，如

果它是 global 或 undefined，就使用预先指定的 默认值（obj），否则保持 this 不变。它也提供了可选的柯里化行为（见先前的 bind(..) 讨论）。

我们来看看它的用法：

```
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   <---- 退回到软绑定
```

软绑定版本的 foo() 函数可以如展示的那样被手动 this 绑定到 obj2 或 obj3，如果 默认绑定适用时会退到 obj。

词法 this

我们刚刚涵盖了一般函数遵守的四种规则。但是 ES6 引入了一种不适用于这些规则特殊的函数：箭头函数（arrow-function）。

箭头函数不是通过 function 关键字声明的，而是通过所谓的 “大箭头” 操作符：=>。与使用四种标准的 this 规则不同的是，箭头函数从封闭它的（函数或全局）作用域采用 this 绑定。

我们来展示一下箭头函数的词法作用域：

```
function foo() {
  // 返回一个箭头函数
    return (a) => {
    // 这里的 `this` 是词法上从 `foo()` 采用的
        console.log( this.a );
    };
```

```
}

var obj1 = {
    a: 2
};

var obj2 = {
    a: 3
};

var bar = foo.call( obj1 );
bar.call( obj2 ); // 2, 不是 3!
```

在 foo() 中创建的箭头函数在词法上捕获 foo() 被调用时的 this，不管它是什么。因为 foo() 被 this 绑定到 obj1，bar（被返回的箭头函数的一个引用）也将会被 this 绑定到 obj1。一个箭头函数的词法绑定是不能被覆盖的（就连 new 也不行!）。

最常见的用法是用于回调，比如事件处理器或计时器:

```
function foo() {
    setTimeout(() => {
        // 这里的 `this` 是词法上从 `foo()` 采用
        console.log( this.a );
    },100);
}

var obj = {
    a: 2
};

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

虽然箭头函数提供除了使用 bind(..) 外，另外一种在函数上来确保 this 的方式，这看起来很吸引人，但重要的是要注意它们本质是使用广为人知的词法作用域来禁止了传统的 this 机制。在 ES6 之前，为此我们已经有了相当常用的模式，这些模式几乎和 ES6 的箭头函数的精神没有区别:

```
function foo() {
    var self = this; // 词法上捕获 `this`
    setTimeout( function(){
```

```
        console.log( self.a );
    }, 100 );
}

var obj = {
    a: 2
};

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

虽然对不想用 bind(..) 的人来说 self = this 和箭头函数都是看起来不错的“解决方案”，但它们实质上逃避了 this 而非理解和接受它。

如果你发现你在写 this 风格的代码，但是大多数或全部时候，你都用词法上的 self = this 或箭头函数“技巧”抵御 this 机制，那么也许你应该：

1. 仅使用词法作用域并忘掉虚伪的 this 风格代码。
2. 完全接受 this 风格机制，包括在必要的时候使用 bind(..)，并尝试避开 self = this 和箭头函数的“词法 this”技巧。

一个程序可以有效地同时利用两种风格的代码（词法和 this），但是在同一个函数内部，特别是对同种类型的查找，混合这两种机制通常是自找很难维护的代码，而且可能是聪明过了头。

复习

为执行中的函数判定 this 绑定需要找到这个函数的直接调用点。找到之后，四种规则将会以这种优先顺序施用于调用点：

1. 通过 new 调用？使用新构建的对象。
2. 通过 call 或 apply（或 bind）调用？使用指定的对象。
3. 通过持有调用的环境对象调用？使用那个环境对象。
4. 默认：strict mode 下是 undefined，否则就是全局对象。

小心偶然或不经意的 默认绑定规则调用。如果你想“安全”地忽略 this 绑定，一个像 ø = Object.create(null) 这样的“DMZ”对象是一个很好的占位值，以保护 global 对象不受意外的副作用影响。

与这四种绑定规则不同，ES6 的箭头方法使用词法作用域来决定 this 绑定，这意味着它们采用封闭他们的函数调用作为 this 绑定（无论它是什么）。它们实质上是 ES6 之前的 self = this 代码的语法替代品。

第三章：对象

在第一和第二章中，我们讲解了 this 绑定如何根据函数调用的调用点指向不同的对象。但究竟什么是对象，为什么我们需要指向它们？这一章我们就来详细探索一下对象。

语法

对象来自于两种形式：声明（字面）形式，和构造形式。

一个对象的字面语法看起来像这样：

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

构造形式看起来像这样：

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

构造形式和字面形式的结果是完全同种类的对象。唯一真正的区别在于你可以向字面声明一次性添加一个或多个键/值对，而对于构造形式，你必须一个一个地添加属性。

注意：像刚才展示的那样使用“构造形式”来创建对象是极其少见的。你很有可能总是想使用字面语法形式。这对大多数内建的对象也一样（后述）。

类型

对象是大多数 JS 程序依赖的基本构建块儿。它们是 JS 的六种主要类型（在语言规范中称为“语言类型”）中的一种：

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

注意 *简单基本类型*（string、number、boolean、null、和 undefined）自身 **不是** object。null 有时会被当成一个对象类型，但是这种误解源自于一个语言中的 Bug，它使得 typeof null 错误地（而且令人困惑地）返回字符串 "object"。实际上，null 是它自己的基本类型。

一个常见的错误论断是“JavaScript 中的一切都是对象”。这明显是不对的。

对比来看，存在几种特殊的对象子类型，我们可以称之为 *复杂基本类型*。

function 是对象的一种子类型（技术上讲，叫做“可调用对象”）。函数在 JS 中被称为“头等（first class）”类型，是因为它们基本上就是普通的对象（附带有可调用的行为语义），而且它们可以像其他普通的对象那样被处理。

数组也是一种形式的对象，带有特别的行为。数组在内容的组织上要稍稍比一般的对象更加结构化。

内建对象

有几种其他的对象子类型，通常称为内建对象。对于其中的一些来说，它们的名称看起来暗示着它们和它们对应的基本类型有着直接的联系，但事实上，它们的关系更复杂，我们一会儿就开始探索。

- String
- Number
- Boolean
- Object
- Function
- Array
- Date
- RegExp
- Error

如果你依照和其他语言的相似性来看的话，比如 Java 语言的 String 类，这些内建类型有着实际类型的外观，甚至是类（class）的外观，

但是在 JS 中，它们实际上仅仅是内建的函数。这些内建函数的每一个都可以被用作构造器（也就是一个可以通过 new 操作符调用的函数 —— 参照第二章），其结果是一个新 *构建*的相应子类型的对象。例如：

```
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

// 考察 object 子类型
Object.prototype.toString.call( strObject );    // [object String]
```

我们会在本章稍后详细地看到 `Object.prototype.toString...` 到底是如何工作的，但简单地说，我们可以通过借用基本的默认 `toString()` 方法来考察内部子类型，而且你可以看到它揭示了 `strObject` 实际上是一个由 `String` 构造器创建的对象。

基本类型值 `"I am a string"` 不是一个对象，它是一个不可变的基本字面值。为了对它进行操作，比如检查它的长度，访问它的各个独立字符内容等等，都需要一个 `String` 对象。

幸运的是，在必要的时候语言会自动地将 `"string"` 基本类型强制转换为 `String` 对象类型，这意味着你几乎从不需要明确地创建对象。JS 社区的绝大部分人都 **强烈推荐**尽可能地使用字面形式的值，而非使用构造的对象形式。

考虑下面的代码：

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

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

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

在这两个例子中，我们在字符串的基本类型上调用属性和方法，引擎会自动地将它强制转换为 `String` 对象，所以这些属性/方法的访问可以工作。

当使用如 `42.359.toFixed(2)` 这样的方法时，同样的强制转换也发生在数字基本字面量 `42` 和包装对象 `new Number(42)` 之间。同样的还有 `Boolean` 对象和 `"boolean"` 基本类型。

`null` 和 `undefined` 没有对象包装的形式，仅有它们的基本类型值。相比之下，`Date` 的值仅可以由它们的构造对象形式创建，因为它们没有对应的字面形式。

无论使用字面还是构造形式，`Object`、`Array`、`Function`、和 `RegExp`（正则表达式）都是对象。在某些情况下，构造形式确实会比对应的字面形式提供更多的创建选项。因为对象可以被任意一种方式创建，更简单的字面形式几乎是所有人的首选。**仅仅在你需要使用额外的选项时使用构建形式**。

`Error` 对象很少在代码中明示地被创建，它们通常在抛出异常时自动地被创建。它们可以由 `new Error(..)` 构造形式创建，但通常是不必要的。

内容

正如刚才提到的，对象的内容由存储在特定命名的 *位置*上的（任意类型的）值组成，我们称这些值为属性。

有一个重要的事情需要注意：当我们说“内容”时，似乎暗示着这些值 *实际上*存储在对象内部，但那只不过是表面现象。引擎会根据自己的实现来存储这些值，而且通常都不是把它们存储在容器对象 *内部*。在容器内存储的是这些属性的名称，它们像指针（技术上讲，叫 *引用*（*reference*））一样指向值存储的地方。

考虑下面的代码：

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

myObject.a;     // 2

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

为了访问 myObject 在 位置 a 的值，我们需要使用 . 或 [ ] 操作符。.a 语法通常称为“属性（property）”访问，而 ["a"] 语法通常称为“键（key）”访问。在现实中，它们俩都访问相同的 位置，而且会拿出相同的值，2，所以这些术语可以互换使用。从现在起，我们将使用最常见的术语 —— “属性访问”。

两种语法的主要区别在于，. 操作符后面需要一个 标识符（Identifier）兼容的属性名，而 ["..."] 语法基本可以接收任何兼容 UTF-8/unicode 的字符串作为属性名。举个例子，为了引用一个名为“Super-Fun!”的属性，你不得不使用 ["Super-Fun!"] 语法访问，因为 Super-Fun! 不是一个合法的 Identifier 属性名。

而且，由于 ["..."] 语法使用字符串的 **值**来指定位置，这意味着程序可以动态地组建字符串的值。比如：

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

var idx;

if (wantA) {
    idx = "a";
}

// 稍后

console.log( myObject[idx] ); // 2
```

在对象中，属性名 **总是**字符串。如果你使用 string 以外的（基本）类型值，它会首先被转换为字符串。这甚至包括在数组中常用于索引的数字，所以要小心不要将对象和数组使用的数字搞混了。

```
var myObject = { };
```

```
myObject[true] = "foo";
myObject[3] = "bar";
myObject[myObject] = "baz";

myObject["true"];               // "foo"
myObject["3"];                  // "bar"
myObject["[object Object]"];    // "baz"
```

计算型属性名

如果你需要将一个计算表达式 *作为*一个键名称，那么我们刚刚描述的 `myObject[..]` 属性访问语法是十分有用的，比如 `myObject[prefix + name]`。但是当使用字面对象语法声明对象时则没有什么帮助。

ES6 加入了 *计算型属性名*，在一个字面对象声明的键名称位置，你可以指定一个表达式，用 `[ ]` 括起来：

```
var prefix = "foo";

var myObject = {
    [prefix + "bar"]: "hello",
    [prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

*计算型属性名*的最常见用法，可能是用于 ES6 的 `Symbol`，我们将不会在本书中涵盖关于它的细节。简单地说，它们是新的基本数据类型，拥有一个不透明不可知的值（技术上讲是一个 `string` 值）。你将会被强烈地不鼓励使用一个 `Symbol` 的 *实际值*（这个值理论上会因 JS 引擎的不同而不同），所以 `Symbol` 的名称，比如 `Symbol.Something`（这是个瞎编的名称！），才是你会使用的：

```
var myObject = {
    [Symbol.Something]: "hello world"
};
```

属性（Property）vs. 方法（Method）

有些开发者喜欢在讨论对一个对象的属性访问时做一个区别，如果这个被访问的值恰好是一个函数的话。因为这诱使人们认为函数 *属于*这个对象，而且在其他语言中，属于对象（也就是“类”）的函数被称作“方法”，所以相对于“属性访问”，我们常能听到“方法访问”。

有趣的是，**语言规范也做出了同样的区别**。

从技术上讲，函数绝不会“属于”对象，所以，说一个偶然在对象的引用上被访问的函数就自动地成为了一个“方法”，看起来有些像是牵强附会。

有些函数内部确实拥有 this 引用，而且 *有时*这些 this 引用指向调用点的对象引用。但这个用法确实没有使这个函数比其他函数更像“方法”，因为 this 是在运行时在调用点动态绑定的，这使得它与这个对象的关系至多是间接的。

每次你访问一个对象的属性都是一个 **属性访问**，无论你得到什么类型的值。如果你 *恰好*从属性访问中得到一个函数，它也没有魔法般地在那时成为一个“方法”。一个从属性访问得来的函数没有任何特殊性（隐含的 this 绑定的情况在刚才已经解释过了）。

举个例子：

```
function foo() {
    console.log( "foo" );
}

var someFoo = foo;  // 对 `foo` 的变量引用


var myObject = {
    someFoo: foo
};

foo;                // function foo(){..}

someFoo;            // function foo(){..}

myObject.someFoo;   // function foo(){..}
```

someFoo 和 myObject.someFoo 只不过是同一个函数的两个分离的引用，它们中的任何一个都不意味着这个函数很特别或被其他对象所“拥有”。如果上面的 foo() 定义里面拥有一个 this 引用，那么 myObject.someFoo 的 *隐含绑定*将会是这个两个引用间 **唯**一可以观察到的不同。它们中的任何一个都没有称为“方法”的道理。

也许有人会争辩，函数 *变成了方法*，不是在定义期间，而是在调用的执行期间，根据它是如何在调用点被调用的（是否带有一个环境对象引用 —— 细节见第二章）。即便是这种解读也有些牵强。

可能最安全的结论是，在 JavaScript 中，“函数”和“方法”是可以互换使用的。

注意： ES6 加入了 super 引用，它通常是和 class（见附录 A）一起使用的。super 的行为方式（静态绑定，而非像 this 一样延迟绑定），给了这种说法更多的权重：一个被 super 绑

定到某处的函数比起“函数”更像一个“方法”。但是同样地，这仅仅是微妙的语义上的（和机制上的）细微区别。

就算你声明一个函数表达式作为字面对象的一部分，那个函数都不会魔法般地 *属于*这个对象 —— 仍然仅仅是同一个函数对象的多个引用罢了。

```
var myObject = {
    foo: function foo() {
        console.log( "foo" );
    }
};

var someFoo = myObject.foo;

someFoo;        // function foo(){..}

myObject.foo;   // function foo(){..}
```

注意：在第六章中，我们会为字面对象的 `foo: function foo(){ .. }` 声明语法介绍一种 ES6 的简化语法。

数组

数组也使用 `[ ]` 访问形式，但正如上面提到的，在存储值的方式和位置上它们的组织更加结构化（虽然仍然在存储值的 *类型*上没有限制）。数组采用 *数字索引*，这意味着值被存储的位置，通常称为 *下标*，是一个非负整数，比如 0 和 42。

```
var myArray = [ "foo", 42, "bar" ];

myArray.length;     // 3

myArray[0];         // "foo"

myArray[2];         // "bar"
```

数组也是对象，所以虽然每个索引都是正整数，你还可以在数组上添加属性：

```
var myArray = [ "foo", 42, "bar" ];

myArray.baz = "baz";
```

```
myArray.length; // 3

myArray.baz;    // "baz"
```

注意，添加命名属性（不论是使用 . 还是 [ ] 操作符语法）不会改变数组的 length 所报告的值。

你可以把一个数组当做普通的键/值对象使用，并且从不添加任何数字下标，但这不是一个好主意，因为数组对它本来的用途有着特定的行为和优化方式，普通对象也一样。使用对象来存储键/值对，而用数组在数字下标上存储值。

小心：如果你试图在一个数组上添加属性，但是属性名看起来像一个数字，那么最终它会成为一个数字索引（也就是改变了数组的内容）：

```
var myArray = [ "foo", 42, "bar" ];

myArray["3"] = "baz";

myArray.length; // 4

myArray[3];     // "baz"
```

复制对象

当开发者们初次拿起 Javascript 语言时，最常需要的特性就是如何复制一个对象。看起来应该有一个内建的 copy() 方法，对吧？但是事情实际上比这复杂一些，因为在默认情况下，复制的算法应当是什么，并不十分明确。

例如，考虑这个对象：

```
function anotherFunction() { /*..*/ }

var anotherObject = {
    c: true
};

var anotherArray = [];

var myObject = {
    a: 2,
    b: anotherObject,   // 引用，不是拷贝！
    c: anotherArray,    // 又一个引用！
    d: anotherFunction
```

```
};

anotherArray.push( anotherObject, myObject );
```

一个 myObject 的 拷贝究竟应该怎么表现？

首先，我们应该回答它是一个 浅（*shallow*）还是一个 深（*deep*）拷贝？一个 浅拷贝（*shallow copy*）会得到一个新对象，它的 a 是值 2 的拷贝，但 b、c 和 d 属性仅仅是引用，它们指向被拷贝对象中引用的相同位置。一个 深拷贝（*deep copy*）将不仅复制 myObject，还会复制 anotherObject 和 anotherArray。但之后我们让 anotherArray 拥有 anotherObject 和 myObject 的引用，所以 那些也应当被复制而不是仅保留引用。现在由于循环引用，我们得到了一个无限循环复制的问题。

我们应当检测循环引用并打破循环遍历吗（不管位于深处的，没有完全复制的元素）？我们应当报错退出吗？或者介于两者之间？

另外，“复制”一个函数意味着什么，也不是很清楚。有一些技巧，比如提取一个函数源代码的 toString() 序列化表达（这个源代码会因实现不同而不同，而且根据被考察的函数的类型，其结果甚至在所有引擎上都不可靠）。

那么我们如何解决所有这些刁钻的问题？不同的 JS 框架都各自挑选自己的解释并且做出自己的选择。但是哪一种（如果有的话）才是 JS 应当作为标准采用的呢？长久以来，没有明确答案。

一个解决方案是，JSON 安全的对象（也就是，可以被序列化为一个 JSON 字符串，之后还可以被重新解析为拥有相同的结构和值的对象）可以简单地这样 复制：

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

当然，这要求你保证你的对象是 JSON 安全的。对于某些情况，这没什么大不了的。而对另一些情况，这还不够。

同时，浅拷贝相当易懂，而且没有那么多问题，所以 ES6 为此任务已经定义了 Object.assign(..)。Object.assign(..) 接收 目标对象作为第一个参数，然后是一个或多个 源对象作为后续参数。它会在 源对象上迭代所有的 可枚举（*enumerable*），*owned keys*（**直接拥有的键**），并把它们拷贝到 目标对象上（仅通过 = 赋值）。它还会很方便地返回 目标对象，正如下面你可以看到的：

```
var newObj = Object.assign( {}, myObject );

newObj.a;                       // 2
newObj.b === anotherObject;     // true
newObj.c === anotherArray;      // true
newObj.d === anotherFunction;   // true
```

注意：在下一部分中，我们将讨论“属性描述符（property descriptors —— 属性的性质）”并展示 `Object.defineProperty(..)` 的使用。然而在 `Object.assign(..)` 中发生的复制是单纯的 `=` 式赋值，所以任何在源对象属性的特殊性质（比如 `writable`）在目标对象上 **都不会保留**。

属性描述符（Property Descriptors）

在 ES5 之前，JavaScript 语言没有给出直接的方法，让你的代码可以考察或描述属性性质间的区别，比如属性是否为只读。

在 ES5 中，所有的属性都用 **属性描述符（Property Descriptors）** 来描述。

考虑这段代码：

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

Object.getOwnPropertyDescriptor( myObject, "a" );
// {
//    value: 2,
//    writable: true,
//    enumerable: true,
//    configurable: true
// }
```

正如你所见，我们普通的对象属性 `a` 的属性描述符（称为“数据描述符”，因为它仅持有一个数据值）的内容要比 `value` 为 2 多得多。它还包含另外三个性质：`writable`、`enumerable`、和 `configurable`。

当我们创建一个普通属性时，可以看到属性描述符的各种性质的默认值，同时我们可以用 `Object.defineProperty(..)` 来添加新属性，或使用期望的性质来修改既存的属性（如果它是 `configurable` 的!）。

举例来说：

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: true,
    enumerable: true
} );
```

```
myObject.a; // 2
```

使用 defineProperty(..)，我们手动、明确地在 myObject 上添加了一个直白的，普通的 a 属性。然而，你通常不会使用这种手动方法，除非你想要把描述符的某个性质修改为不同的值。

可写性（Writable）

writable 控制着你改变属性值的能力。

考虑这段代码：

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // 不可写!
    configurable: true,
    enumerable: true
} );

myObject.a = 3;

myObject.a; // 2
```

如你所见，我们对 value 的修改悄无声息地失败了。如果我们在 strict mode 下进行尝试，会得到一个错误：

```
"use strict";

var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // 不可写!
    configurable: true,
    enumerable: true
} );

myObject.a = 3; // TypeError
```

这个 TypeError 告诉我们，我们不能改变一个不可写属性。

注意：我们一会儿就会讨论 getters/setters，但是简单地说，你可以观察到 writable:false 意味着值不可改变，和你定义一个空的 setter 是有些等价的。实际上，你的空 setter 在被调用时需要扔出一个 TypeError，来和 writable:false 保持一致。

可配置性（Configurable）

只要属性当前是可配置的，我们就可以使用相同的 defineProperty(..) 工具，修改它的描述符定义。

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

myObject.a = 3;
myObject.a;                 // 3

Object.defineProperty( myObject, "a", {
    value: 4,
    writable: true,
    configurable: false,    // 不可配置!
    enumerable: true
} );

myObject.a;                 // 4
myObject.a = 5;
myObject.a;                 // 5

Object.defineProperty( myObject, "a", {
    value: 6,
    writable: true,
    configurable: true,
    enumerable: true
} ); // TypeError
```

最后的 defineProperty(..) 调用导致了一个 TypeError，这与 strict mode 无关，如果你试图改变一个不可配置属性的描述符定义，就会发生 TypeError。要小心：如你所看到的，将 configurable 设置为 false 是 **一个单向操作，不可撤销！**

注意：这里有一个需要注意的微小例外：即便属性已经是 configurable:false，writable 总是可以没有错误地从 true 改变为 false，但如果已经是 false 的话不能变回 true。

configurable:false 阻止的另外一个事情是使用 delete 操作符移除既存属性的能力。

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

myObject.a;             // 2
delete myObject.a;
myObject.a;             // undefined

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: false,
    enumerable: true
} );

myObject.a;             // 2
delete myObject.a;
myObject.a;             // 2
```

如你所见，最后的 delete 调用（无声地）失败了，因为我们将 a 属性设置成了不可配置。

delete 仅用于直接从目标对象移除该对象的（可以被移除的）属性。如果一个对象的属性是某个其他对象/函数的最后一个现存的引用，而你 delete 了它，那么这就移除了这个引用，于是现在那个没有被任何地方所引用的对象/函数就可以被作为垃圾回收。但是，将 delete 当做一个像其他语言（如 C/C++）中那样的释放内存工具是 **不**恰当的。delete 仅仅是一个对象属性移除操作 —— 没有更多别的含义。

可枚举性（Enumerable）

我们将要在这里提到的最后一个描述符性质是 enumerable（还有另外两个，我们将在一会儿讨论 getter/setters 时谈到）。

它的名称可能已经使它的功能很明显了，这个性质控制着一个属性是否能在特定的对象-属性枚举操作中出现，比如 for..in 循环。设置为 false 将会阻止它出现在这样的枚举中，即使它依然完全是可以访问的。设置为 true 会使它出现。

所有普通的用户定义属性都默认是可 enumerable 的，正如你通常希望的那样。但如果你有一个特殊的属性，你想让它对枚举隐藏，就将它设置为 enumerable:false。

我们一会儿就更加详细地演示可枚举性，所以在大脑中给这个话题上打一个书签。

不可变性（Immutability）

有时我们希望将属性或对象（有意或无意地）设置为不可改变的。ES5 用几种不同的微妙方式，加入了对此功能的支持。

一个重要的注意点是：**所有**这些方法创建的都是浅不可变性。也就是，它们仅影响对象和它的直属属性的性质。如果对象拥有对其他对象（数组、对象、函数等）的引用，那个对象的 *内容*不会受影响，任然保持可变。

```
myImmutableObject.foo; // [1,2,3]
myImmutableObject.foo.push( 4 );
myImmutableObject.foo; // [1,2,3,4]
```

在这段代码中，我们假设 `myImmutableObject` 已经被创建，而且被保护为不可变。但是，为了保护 `myImmutableObject.foo` 的内容（也是一个对象 —— 数组），你将需要使用下面的一个或多个方法将 `foo` 设置为不可变。

注意：在 JS 程序中创建完全不可动摇的对象是不那么常见的。有些特殊情况当然需要，但作为一个普通的设计模式，如果你发现自己想要 *封印*（*seal*）或 *冻结*（*freeze*）你所有的对象，那么你可能想要退一步来重新考虑你的程序设计，让它对对象值的潜在变化更加健壮。

对象常量（Object Constant）

通过将 `writable:false` 与 `configurable:false` 组合，你可以实质上创建了一个作为对象属性的 *常量*（不能被改变，重定义或删除），比如：

```
var myObject = {};

Object.defineProperty( myObject, "FAVORITE_NUMBER", {
    value: 42,
    writable: false,
    configurable: false
} );
```

防止扩展（Prevent Extensions）

如果你想防止一个对象被添加新的属性，但另一方面保留其他既存的对象属性，可以调用 `Object.preventExtensions(..)`：

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

Object.preventExtensions( myObject );
```

```
myObject.b = 3;
myObject.b; // undefined
```

在非 strict mode 模式下，b 的创建会无声地失败。在 strict mode 下，它会抛出 TypeError。

封印（Seal）

Object.seal(..) 创建一个“封印”的对象，这意味着它实质上在当前的对象上调用 Object.preventExtensions(..)，同时也将它所有的既存属性标记为 configurable:false。

所以，你既不能添加更多的属性，也不能重新配置或删除既存属性（虽然你依然可以修改它们的值）。

冻结（Freeze）

Object.freeze(..) 创建一个冻结的对象，这意味着它实质上在当前的对象上调用 Object.seal(..)，同时也将它所有的“数据访问”属性设置为 writable:false，所以它们的值不可改变。

这种方法是你可以从对象自身获得的最高级别的不可变性，因为它阻止任何对对象或对象直属属性的改变（虽然，就像上面提到的，任何被引用的对象的内容不受影响）。

你可以“深度冻结”一个对象：在这个对象上调用 Object.freeze(..)，然后递归地迭代所有它引用的（目前还没有受过影响的）对象，然后也在它们上面调用 Object.freeze(..)。但是要小心，这可能会影响其他你并不打算影响的（共享的）对象。

[[Get]]

关于属性访问如何工作有一个重要的细节。

考虑下面的代码：

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

myObject.a; // 2
```

myObject.a 是一个属性访问，但是它并不是看起来那样，仅仅在 myObject 中寻找一个名为 a 的属性。

根据语言规范，上面的代码实际上在 myObject 上执行了一个 [[Get]] 操作（有些像 [[Get]]() 函数调用）。对一个对象进行默认的内建 [[Get]] 操作，会首先检查对象，寻找一个拥有被请求的名称的属性，如果找到，就返回相应的值。

然而，如果按照被请求的名称 没能找到属性，[[Get]] 的算法定义了另一个重要的行为。我们会在第五章来解释 接下来会发生什么（遍历 [[Prototype]] 链，如果有的话）。

但 [[Get]] 操作的一个重要结果是，如果它通过任何方法都不能找到被请求的属性的值，那么它会返回 undefined。

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

myObject.b; // undefined
```

这个行为和你通过标识符名称来引用 变量不同。如果你引用了一个在可用的词法作用域内无法解析的变量，其结果不是像对象属性那样返回 undefined，而是抛出一个 ReferenceError。

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

myObject.a; // undefined

myObject.b; // undefined
```

从 值的角度来说，这两个引用没有区别 —— 它们的结果都是 undefined。然而，在 [[Get]] 操作的底层，虽然不明显，但是比起处理引用 myObject.a，处理 myObject.b 的操作要多做一些潜在的 “工作”。

如果仅仅考察结果的值，你无法分辨一个属性是存在并持有一个 undefined 值，还是因为属性根本 不存在所以 [[Get]] 无法返回某个具体值而返回默认的 undefined。但是，你很快就能看到你其实 可以分辨这两种场景。

[[Put]]

既然为了从一个属性中取得值而存在一个内部定义的 [[Get]] 操作，那么很明显应该也存在一个默认的 [[Put]] 操作。

这很容易让人认为，给一个对象的属性赋值，将会在这个对象上调用 [[Put]] 来设置或创建这个属性。但是实际情况却有一些微妙的不同。

调用 [[Put]] 时，它根据几个因素表现不同的行为，包括（影响最大的）属性是否已经在对象中存在了。

如果属性存在，[[Put]] 算法将会大致检查：

1. 这个属性是访问器描述符吗（见下一节 “Getters 与 Setters”）？**如果是，而且是 setter，就调用 setter**。

2. 这个属性是 `writable` 为 `false` 数据描述符吗？**如果是，在非 `strict mode` 下无声地失败，或者在 `strict mode` 下抛出 `TypeError`。**
3. 否则，像平常一样设置既存属性的值。

如果属性在当前的对象中还不存在，`[[Put]]` 操作会变得更微妙和复杂。我们将在第五章讨论 `[[Prototype]]` 时再次回到这个场景，更清楚地解释它。

Getters 与 Setters

对象默认的 `[[Put]]` 和 `[[Get]]` 操作分别完全控制着如何设置既存或新属性的值，和如何取得既存属性。

注意：使用较先进的语言特性，覆盖整个对象（不仅是每个属性）的默认 `[[Put]]` 和 `[[Get]]` 操作是可能的。这超出了我们要在这本书中讨论的范围，但我们会在后面的“你不懂 JS”系列中涵盖此内容。

ES5 引入了一个方法来覆盖这些默认操作的一部分，但不是在对象级别而是针对每个属性，就是通过 getters 和 setters。Getter 是实际上调用一个隐藏函数来取得值的属性。Setter 是实际上调用一个隐藏函数来设置值的属性。

当你将一个属性定义为拥有 getter 或 setter 或两者兼备，那么它的定义就成为了“访问器描述符”（与“数据描述符”相对）。对于访问器描述符，它的 `value` 和 `writable` 性质因没有意义而被忽略，取而代之的是 JS 将会考虑属性的 `set` 和 `get` 性质（还有 `configurable` 和 `enumerable`）。

考虑下面的代码：

```
var myObject = {
    // 为 `a` 定义一个 getter
    get a() {
        return 2;
    }
};

Object.defineProperty(
    myObject,   // 目标对象
    "b",        // 属性名
    {           // 描述符
        // 为 `b` 定义 getter
        get: function(){ return this.a * 2 },

        // 确保 `b` 作为对象属性出现
        enumerable: true
```

```
    }
);

myObject.a; // 2

myObject.b; // 4
```

不管是通过在字面对象语法中使用 `get a() { .. }`，还是通过使用 `defineProperty(..)` 明确定义，我们都在对象上创建了一个没有实际持有值的属性，访问它们将会自动地对 getter 函数进行隐藏的函数调用，其返回的任何值就是属性访问的结果。

```
var myObject = {
    // 为 `a` 定义 getter
    get a() {
        return 2;
    }
};

myObject.a = 3;

myObject.a; // 2
```

因为我们仅为 `a` 定义了一个 getter，如果之后我们试着设置 `a` 的值，赋值操作并不会抛出错误而是无声地将赋值废弃。就算这里有一个合法的 setter，我们的自定义 getter 将返回值硬编码为仅返回 2，所以赋值操作是没有意义的。

为了使这个场景更合理，正如你可能期望的那样，每个属性还应当被定义一个覆盖默认 `[[Put]]` 操作（也就是赋值）的 setter。几乎可确定，你将总是想要同时声明 getter 和 setter（仅有它们中的一个经常会导致意外的行为）：

```
var myObject = {
    // 为 `a` 定义 getter
    get a() {
        return this._a_;
    },

    // 为 `a` 定义 setter
    set a(val) {
        this._a_ = val * 2;
    }
```

```
};

myObject.a = 2;

myObject.a; // 4
```

注意：在这个例子中，我们实际上将赋值操作（[[Put]] 操作）指定的值 2 存储到了另一个变量 _a_ 中。_a_ 这个名称只是用在这个例子中的单纯惯例，并不意味着它的行为有什么特别之处 —— 它和其他普通属性没有区别。

存在性（Existence）

我们早先看到，像 myObject.a 这样的属性访问可能会得到一个 undefined 值，无论是它明确存储着 undefined 还是属性 a 根本就不存在。那么，如果这两种情况的值相同，我们还怎么区别它们呢？

我们可以查询一个对象是否拥有特定的属性，而 不必取得那个属性的值：

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

("a" in myObject);              // true
("b" in myObject);              // false

myObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "b" ); // false
```

in 操作符会检查属性是否存在于对象 中，或者是否存在于 [[Prototype]] 链对象遍历的更高层中（详见第五章）。相比之下，hasOwnProperty(..) 仅仅检查 myObject 是否拥有属性，但 不会查询 [[Prototype]] 链。我们会在第五章详细讲解 [[Prototype]] 时，回来讨论这个两个操作重要的不同。

通过委托到 Object.prototype，所有的普通对象都可以访问 hasOwnProperty(..)（详见第五章） 。但是创建一个不链接到 Object.prototype 的对象也是可能的（通过 Object.create(null) —— 详见第五章）。这种情况下，像 myObject.hasOwnProperty(..) 这样的方法调用将会失败。

在这种场景下，一个进行这种检查的更健壮的方式是 Object.prototype.hasOwnProperty.call(myObject 它借用基本的 hasOwnProperty(..) 方法而且使用 明确的 *this* 绑定（详见第二章）来对我们的 myObject 实施这个方法。

注意： `in` 操作符看起来像是要检查一个值在容器中的存在性，但是它实际上检查的是属性名的存在性。在使用数组时注意这个区别十分重要，因为我们会有很强的冲动来进行 `4 in [2, 4, 6]` 这样的检查，但是这总是不像我们想象的那样工作。

枚举（Enumeration）

先前，在学习 `enumerable` 属性描述符性质时，我们简单地解释了“可枚举性（enumerability）”的含义。现在，让我们来更加详细地重新讲解它。

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // 使 `a` 可枚举，如一般情况
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // 使 `b` 不可枚举
    { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty( "b" ); // true

// .......

for (var k in myObject) {
    console.log( k, myObject[k] );
}
// "a" 2
```

你会注意到，`myObject.b` 实际上 **存在**，而且拥有可以访问的值，但是它不出现在 `for..in` 循环中（然而令人诧异的是，它的 `in` 操作符的存在性检查通过了）。这是因为“enumerable”基本上意味着“如果对象的属性被迭代时会被包含在内”。

注意：将 for..in 循环实施在数组上可能会给出意外的结果，因为枚举一个数组将不仅包含所有的数字下标，还包含所有的可枚举属性。所以一个好主意是：将 for..in 循环 仅用于对象，而为存储在数组中的值使用传统的 for 循环并用数字索引迭代。

另一个可以区分可枚举和不可枚举属性的方法是：

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // 使 `a` 可枚举，如一般情况
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // 使 `b` 不可枚举
    { enumerable: false, value: 3 }
);

myObject.propertyIsEnumerable( "a" ); // true
myObject.propertyIsEnumerable( "b" ); // false

Object.keys( myObject ); // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

propertyIsEnumerable(..) 测试一个给定的属性名是否直 接存在于对象上，并且是 enumerable:true。

Object.keys(..) 返回一个所有可枚举属性的数组，而 Object.getOwnPropertyNames(..) 返回一个 所有属性的数组，不论能不能枚举。

in 和 hasOwnProperty(..) 区别于它们是否查询 [[Prototype]] 链，而 Object.keys(..) 和 Object.getOwnPropertyNames(..) 都 只考察直接给定的对象。

（当下）没有与 in 操作符的查询方式（在整个 [[Prototype]] 链上遍历所有的属性，如我们在第五章解释的）等价的、内建的方法可以得到一个 **所有属性**的列表。你可以近似地模拟一个这样的工具：递归地遍历一个对象的 [[Prototype]] 链，在每一层都从 Object.keys(..) 中取得一个列表——仅包含可枚举属性。

迭代（Iteration）

`for..in` 循环迭代一个对象上（包括它的 `[[Prototype]]` 链）所有的可迭代属性。但如果你想要迭代值呢？

在数字索引的数组中，典型的迭代所有的值的办法是使用标准的 `for` 循环，比如：

```
var myArray = [1, 2, 3];

for (var i = 0; i < myArray.length; i++) {
    console.log( myArray[i] );
}
// 1 2 3
```

但是这并没有迭代所有的值，而是迭代了所有的下标，然后由你使用索引来引用值，比如 `myArray[i]`。

ES5 还为数组加入了几个迭代帮助方法，包括 `forEach(..)`、`every(..)`、和 `some(..)`。这些帮助方法的每一个都接收一个回调函数，这个函数将施用于数组中的每一个元素，仅在如何响应回调的返回值上有所不同。

`forEach(..)` 将会迭代数组中所有的值，并且忽略回调的返回值。`every(..)` 会一直迭代到最后，*或者*当回调返回一个 `false`（或 “falsy”）值，而 `some(..)` 会一直迭代到最后，*或者*当回调返回一个 `true`（或 “truthy”）值。

这些在 `every(..)` 和 `some(..)` 内部的特殊返回值有些像普通 `for` 循环中的 `break` 语句，它们可以在迭代执行到末尾之前将它结束掉。

如果你使用 `for..in` 循环在一个对象上进行迭代，你也只能间接地得到值，因为它实际上仅仅迭代对象的所有可枚举属性，让你自己手动地去访问属性来得到值。

注意：与以有序数字的方式（`for` 循环或其他迭代器）迭代数组的下标比较起来，迭代对象属性的顺序是 **不确定**的，而且可能会因 JS 引擎的不同而不同。对于需要跨平台环境保持一致的问题，**不要依赖**观察到的顺序，因为这个顺序是不可靠的。

但是如果你想直接迭代值，而不是数组下标（或对象属性）呢？ES6 加入了一个有用的 `for..of` 循环语法，用来迭代数组（和对象，如果这个对象有定义的迭代器）：

```
var myArray = [ 1, 2, 3 ];

for (var v of myArray) {
    console.log( v );
}
// 1
// 2
// 3
```

`for..of` 循环要求被迭代的 *东西*提供一个迭代器对象（从一个在语言规范中叫做 `@@iterator` 的默认内部函数那里得到），每次循环都调用一次这个迭代器对象的 `next()` 方法，循环迭代的内容就是这些连续的返回值。

数组拥有内建的 `@@iterator`，所以正如展示的那样，`for..of` 对于它们很容易使用。但是让我们使用内建的 `@@iterator` 来手动迭代一个数组，来看看它是怎么工作的：

```
var myArray = [ 1, 2, 3 ];
var it = myArray[Symbol.iterator]();

it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { done:true }
```

注意：我们使用一个 ES6 的 `Symbol`：`Symbol.iterator` 来取得一个对象的 `@@iterator` *内部属性*。我们在本章中简单地提到过 `Symbol` 的语义（见“计算型属性名”），同样的原理也适用于这里。你总是希望通过 `Symbol` 名称，而不是它可能持有的特殊的值，来引用这样特殊的属性。另外，尽管这个名称有这样的暗示，但 `@@iterator` 本身 **不是迭代器对象**，而是一个返回迭代器对象的 **方法** —— 一个重要的细节！

正如上面的代码段揭示的，迭代器的 `next()` 调用的返回值是一个 `{ value: .. , done: .. }` 形式的对象，其中 `value` 是当前迭代的值，而 `done` 是一个 `boolean`，表示是否还有更多内容可以迭代。

注意值 3 和 `done:false` 一起返回，猛地一看会有些奇怪。你不得不第四次调用 `next()`（在前一个代码段的 `for..of` 循环会自动这样做）来得到 `done:true`，以使自己知道迭代已经完成。这个怪异之处的原因超出了我们要在这里讨论的范围，但是它源自于 ES6 生成器（generator）函数的语义。

虽然数组可以在 `for..of` 循环中自动迭代，但普通的对象 **没有内建的 `@@iterator`**。这种故意省略的原因要比我们将在这里解释的更复杂，但一般来说，为了未来的对象类型，最好不要加入那些可能最终被证明是麻烦的实现。

但是 *可以*为你想要迭代的对象定义你自己的默认 `@@iterator`。比如：

```
var myObject = {
    a: 2,
    b: 3
};

Object.defineProperty( myObject, Symbol.iterator, {
    enumerable: false,
    writable: false,
```

```
    configurable: true,
    value: function() {
        var o = this;
        var idx = 0;
        var ks = Object.keys( o );
        return {
            next: function() {
                return {
                    value: o[ks[idx++]],
                    done: (idx > ks.length)
                };
            }
        };
    }
} );

// 手动迭代 `myObject`
var it = myObject[Symbol.iterator]();
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { value:undefined, done:true }

// 用 `for..of` 迭代 `myObject`
for (var v of myObject) {
    console.log( v );
}
// 2
// 3
```

注意：我们使用了 `Object.defineProperty(..)` 来自定义我们的 `@@iterator`（很大程度上是因为我们可以将它指定为不可枚举的），但是通过将 `Symbol` 作为一个 *计算型属性名*（在本章前面的部分讨论过），我们也可以直接声明它，比如 `var myObject = { a:2, b:3, [Symbol.iterator]: function(){ /* .. */ } }`。

每次 `for..of` 循环在 `myObject` 的迭代器对象上调用 `next()` 时，迭代器内部的指针将会向前移动并返回对象属性列表的下一个值（关于对象属性/值迭代顺序，参照前面的注意事项）。

我们刚刚演示的迭代，是一个简单的一个值一个值的迭代，当然你可以为你的自定义数据结构定义任意复杂的迭代方法，只要你觉得合适。对于操作用户自定义对象来说，自定义迭代器与 ES6 的 `for..of` 循环相组合，是一个新的强大的语法工具。

举个例子，一个 Pixel（像素）对象列表（拥有 x 和 y 的坐标值）可以根据距离原点 (0,0) 的直线距离决定它的迭代顺序，或者过滤掉那些“太远”的点，等等。只要你的迭代器从 next() 调用返回期望的 { value: .. } 返回值，并在迭代结束后返回一个 { done: true } 值，ES6 的 for..of 循环就可以迭代它。

其实，你甚至可以生成一个永远不会“结束”，并且总会返回一个新值（比如随机数，递增值，唯一的识别符等等）的“无穷”迭代器，虽然你可能不会将这样的迭代器用于一个没有边界的 for..of 循环，因为它永远不会结束，而且会阻塞你的程序。

```
var randoms = {
    [Symbol.iterator]: function() {
        return {
            next: function() {
                return { value: Math.random() };
            }
        };
    }
};

var randoms_pool = [];
for (var n of randoms) {
    randoms_pool.push( n );

    // 不要超过边界!
    if (randoms_pool.length === 100) break;
}
```

这个迭代器会“永远”生成随机数，所以我们小心地仅从中取出 100 个值，以使我们的程序不被阻塞。

复习

JS 中的对象拥有字面形式（比如 var a = { .. }）和构造形式（比如 var a = new Array(..)）。字面形式几乎总是首选，但在某些情况下，构造形式提供更多的构建选项。

许多人声称“Javascript 中的一切都是对象”，这是不对的。对象是六种（或七中，看你从哪个方面说）基本类型之一。对象有子类型，包括 function，还可以被行为特化，比如 [object Array] 作为内部的标签表示子类型数组。

对象是键/值对的集合。通过 .propName 或 ["propName"] 语法，值可以作为属性访问。不管属性什么时候被访问，引擎实际上会调用内部默认的 [[Get]] 操作（在设置值时调用 [[Put]] 操作），它不仅直接在对象上查找属性，在没有找到时还会遍历 [[Prototype]] 链（见第五章）。

属性有一些可以通过属性描述符控制的特定性质，比如 `writable` 和 `configurable`。另外，对象拥有它的不可变性（它们的属性也有），可以通过使用 `Object.preventExtensions(..)`、`Object.seal(..)`、和 `Object.freeze(..)` 来控制几种不同等级的不可变性。

属性不必非要包含值 —— 它们也可以是带有 getter/setter 的 “访问器属性”。它们也可以是可枚举或不可枚举的，这控制它们是否会在 `for..in` 这样的循环迭代中出现。

你也可以使用 ES6 的 `for..of` 语法，在数据结构（数组，对象等）中迭代 **值**，它寻找一个内建或自定义的 `@@iterator` 对象，这个对象由一个 `next()` 方法组成，通过这个 `next()` 方法每次迭代一个数据。

第四章: 混合（淆）“类”的对象

接着我们上一章对对象的探索，我们很自然的将注意力转移到 “面向对象（OO）编程”，与 “类（class）”。我们先将 “面向类” 作为设计模式来看看，之后我们再考察 “类” 的机制：“实例化（instantiation）”, “继承（inheritance）” 与 “（相对）多态 (relative polymorphism）”。

我们将会看到，这些概念并不是非常自然地映射到 JS 的对象机制上，以及许多 JavaScript 开发者为了克服这些挑战所做的努力（mixins 等）。

注意：这一章花了相当一部分时间（前一半!）在着重解释 “面向对象编程” 理论上。在后半部分讨论 “Mixins（混合）” 时，我们最终会将这些理论与真实且实际的 JavaScript 代码联系起来。但是这里首先要蹚过许多概念和假想代码，所以可别跟丢了 —— 坚持下去!

类理论

“类/继承” 描述了一种特定的代码组织和结构形式 —— 一种在我们的软件中对真实世界的建模方法。

OO 或者面向类的编程强调数据和操作它的行为之间有固有的联系（当然，依数据的类型和性质不同而不同!），所以合理的设计是将数据和行为打包在一起（也称为封装）。这有时在正式的计算机科学中称为 “数据结构”。

比如，表示一个单词或短语的一系列字符通常称为 “string（字符串）”。这些字符就是数据。但你几乎从来不关心数据，你总是想对数据 *做事情*，所以可以 *向*数据实施的行为（计算它的长度，在末尾添加数据，检索，等等）都被设计成为 `String` 类的方法。

任何给定的字符串都是这个类的一个实例，这个类是一个整齐的集合包装：字符数据和我们可以对它进行操作的功能。

类还隐含着对一个特定数据结构的一种 *分类*方法。其做法是将一个给定的结构考虑为一个更加泛化的基础定义的具体种类。

让我们通过一个最常被引用的例子来探索这种分类处理。一辆 *车*可以被描述为一 “类” 更泛化的东西 —— *载具* —— 的具体实现。

我们在软件中通过定义 `Vehicle` 类和 `Car` 类来模型化这种关系。

`Vehicle` 的定义可能会包含像动力(引擎等),载人能力等等,这些都是行为。我们在 `Vehicle` 中定义的都是所有（或大多数）不同类型的载具（飞机、火车、机动车）都共同拥有的东西。

在我们的软件中为每一种不同类型的载具一次又一次地重定义“载人能力”这个基本性质可能没有道理。反而，我们在 `Vehicle` 中把这个能力定义一次，之后当我们定义 `Car` 时，我们简单地指出它从基本的 `Vehicle` 定义中“继承”（或“扩展”）。于是 `Car` 的定义就被称为特化了更一般的 `Vehicle` 定义。

`Vehicle` 和 `Car` 用方法的形式集约地定义了行为，另一方面一个实例中的数据就像一个唯一的车牌号一样属于一辆具体的车。

这样，类，继承，和实例化就诞生了。

另一个关于类的关键概念是“多态（polymorphism）”，它描述这样的想法：一个来自于父类的泛化行为可以被子类覆盖，从而使它更加具体。实际上，相对多态允许我们在覆盖行为中引用基础行为。

类理论强烈建议父类和子类对相同的行为共享同样的方法名，以便于子类（差异化地）覆盖父类。我们即将看到，在你的 JavaScript 代码中这么做会导致种种困难和脆弱的代码。

“类”设计模式

你可能从没把类当做一种“设计模式”考虑过，因为最常见的是关于流行的“面向对象设计模式”的讨论，比如“迭代器（Iterator）”、“观察者（Observer）”、“工厂（Factory）”、“单例（Singleton）”等等。当以这种方式表现时，几乎可以假定 OO 的类是我们实现所有（高级）设计模式的底层机制，好像对所有代码来说 OO 是一个给定的基础。

取决于你在编程方面接受过的正规教育的水平，你可能听说过“过程式编程（procedural programming）”：一种不用任何高级抽象，仅仅由过程（也就是函数）调用其他函数构成的描述代码的方式。你可能被告知过，类是一个将过程式风格的“面条代码”转换为结构良好，组织良好代码的 *恰当*的方法。

当然，如果你有“函数式编程（functional programming）”的经验，你可能知道类只是几种常见设计模式中的一种。但是对于其他人来说，这可能是第一次你问自己，类是否真的是代码的根本基础，或者它们是在代码顶层上的选择性抽象。

有些语言（比如 Java）不给你选择，所以这根本没什么 *选择性* —— 一切都是类。其他语言如 C/C++ 或 PHP 同时给你过程式和面向类的语法，在使用哪种风格合适或混合风格上，留给开发者更多选择。

JavaScript 的“类”

在这个问题上 JavaScript 属于哪一边？JS 拥有 *一些*像类的语法元素（比如 `new` 和 `instanceof`）有一阵子了，而且在最近的 ES6 中，还有一些追加的东西，比如 `class` 关键字（见附录 A）。

但这意味着 JavaScript 实际上 *拥有*类吗？简单明了：**没有**。

由于类是一种设计模式，你 *可以*，用相当的努力（正如我们将在本章剩下的部分看到的），近似实现很多经典类的功能。JS 在通过提供看起来像类的语法，来努力满足用类进行设计的极其广泛的 *渴望*。

虽然我们好像有了看起来像类的语法，但是 JavaScript 机制好像在抵抗你使用 *类设计模式*，因为在底层，这些你正在上面工作的机制运行的十分不同。语法糖和（极其广泛被使用的）JS “Class” 库费了很大力气来把这些真实情况对你隐藏起来，但你迟早会面对现实：你在其他语言中遇到的 *类*和你在 JS 中模拟的 “类” 不同。

总而言之，类是软件设计中的一种可选模式，你可以选择在 JavaScript 中使用或不使用它。因为许多开发者都对面向类的软件设计情有独钟，我们将在本章剩下的部分中探索一下，为了使用 JS 提供的东西维护类的幻觉要付出什么代价，和我们经历的痛苦。

类机制

在许多面向类语言中，“标准库” 都提供一个叫 “栈”（压栈，弹出等）的数据结构，用一个 `Stack` 类表示。这个类拥有一组变量来存储数据，还拥有一组可公开访问的行为（“方法”），这些行为使你的代码有能力与（隐藏的）数据互动（添加或移除数据等等）。

但是在这样的语言中，你不是直接在 `Stack` 上操作（除非制造一个 **静态的**类成员引用，但这超出了我们要讨论的范围）。`Stack` 类仅仅是 *任何*的 “栈” 都会做的事情的一个抽象解释，但它本身不是一个 “栈”。为了得到一个可以对之进行操作的实在的数据结构，你必须 **实例化**这个 `Stack` 类。

建筑物

传统的 “类（class）” 和 “实例（instance）” 的比拟源自于建筑物的建造。

一个建筑师会规划出一栋建筑的所有性质：多宽，多高，在哪里有多少窗户，甚至墙壁和天花板用什么材料。在这个时候，她并不关心建筑物将会被建造在 *哪里*，她也不关心有 *多少*这栋建筑的拷贝将被建造。

同时她也不关心这栋建筑的内容 —— 家具、墙纸、吊扇等等 —— 她仅关心建筑物含有何种结构。

她生产的建筑学上的蓝图仅仅是建筑物的 “方案”。它们不实际构成我们可以实在进入其中并坐下的建筑物。为了这个任务我们需要一个建筑工人。建筑工人会拿走方案并精确地依照它们 *建造*这栋建筑物。在真正的意义上，他是在将方案中意图的性质 *拷贝*到物理建筑物中。

一旦完成，这栋建筑就是蓝图方案的一个物理实例，一个很可能实质完美的 *拷贝*。然后建筑工人就可以移动到隔壁将它再重做一遍，建造另一个 *拷贝*。

建筑物与蓝图间的关系是间接的。你可以检视蓝图来了解建筑物是如何构造的，但对于直接考察建筑物的每一部分，仅有蓝图是不够的。如果你想打开一扇门，你不得不走进建筑物自身 —— 蓝图仅仅是为了用来 *表示*门的位置而在纸上画的线条。

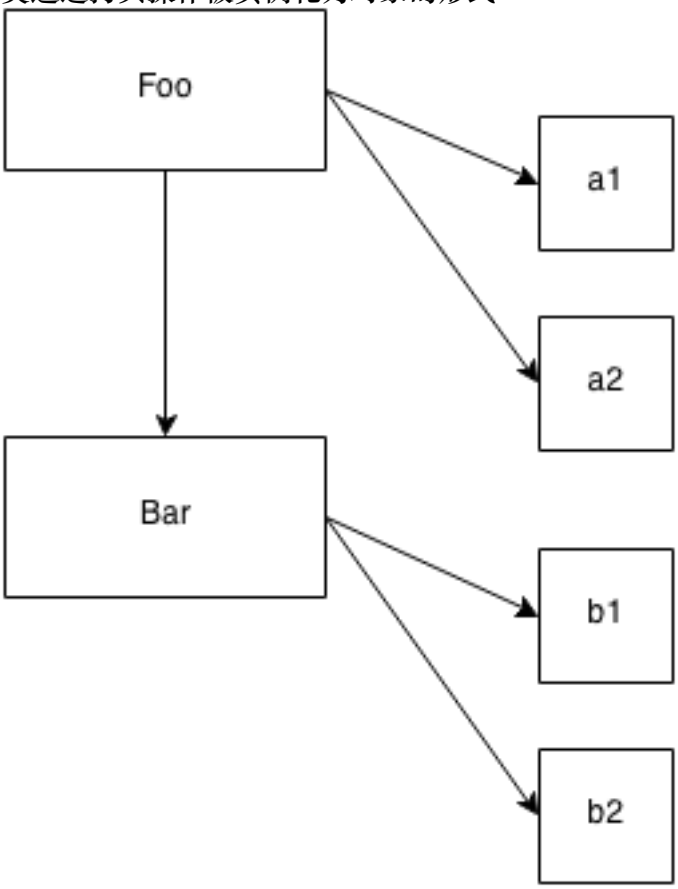
一个类就是一个蓝图。为了实际得到一个对象并与之互动，我们必须从类中建造（也就是实例化）某些东西。这种 “构建” 的最终结果是一个对象，通常称为一个 “实例”，我们可以按需

要直接调用它的方法，访问它的公共数据属性。

这个对象是所有在类中被描述的特性的 拷贝。

你不太可能会指望走进一栋建筑之后发现，一份用于规划这栋建筑物的蓝图被裱起来挂在墙上，虽然蓝图可能在办公室的公共记录的文件中。相似地，你一般不会使用对象实例来直接访问和操作类，但是对于判定对象实例来自于 哪个类至少是可能的。

与考虑对象实例与它源自的类的任何间接关系相比，考虑类和对象实例的直接关系更有用。**一个类通过拷贝操作被实例化为对象的形式。**



如你所见，箭头由左向右，从上至下，这表示着概念上和物理上发生的拷贝操作。

构造器（Constructor）

类的实例由类的一种特殊方法构建，这个方法的名称通常与类名相同，称为“构造器（*constructor*）”。这个方法的具体工作，就是初始化实例所需的所有信息（状态）。

比如，考虑下面这个类的假想代码（语法是自创的）：

```
class CoolGuy {
    specialTrick = nothing

    CoolGuy( trick ) {
        specialTrick = trick
```

```
    }

    showOff() {
        output( "Here's my trick: ", specialTrick )
    }
}
```

为了 制造一个 CoolGuy 实例，我们需要调用类的构造器:

```
Joe = new CoolGuy( "jumping rope" )

Joe.showOff() // Here's my trick: jumping rope
```

注意，CoolGuy 类有一个构造器 CoolGuy()，它实际上就是在我们说 new CoolGuy(..) 时调用的。我们从这个构造器拿回一个对象（类的一个实例），我们可以调用 showOff() 方法，来打印这个特定的 CoolGuy 的特殊才艺。

显然，跳绳使 *Joe* 看起来很酷。

类的构造器 属于那个类，几乎总是和类同名。同时，构造器大多数情况下总是需要用 new 来调用，以便使语言的引擎知道你想要构建一个 新的类的实例。

类继承

在面向类的语言中，你不仅可以定义一个能够初始化它自己的类，你还可以定义另外一个类 **继承**自第一个类。

这第二个类通常被称为“子类”，而第一个类被称为“父类”。这些名词显然来自于亲子关系的比拟，虽然这种比拟有些扭曲，就像你马上要看到的。

当一个家长拥有一个和他有血缘关系的孩子时，家长的遗传性质会被拷贝到孩子身上。明显地，在大多数生物繁殖系统中，双亲都平等地贡献基因进行混合。但是为了这个比拟的目的，我们假设只有一个亲人。

一旦孩子出现，他或她就从亲人那里分离出来。这个孩子受其亲人的继承因素的严重影响，但是独一无二。如果这个孩子拥有红色的头发，这并不意味这他的亲人的头发 曾经是红色，或者会自动 变成红色。

以相似的方式，一旦一个子类被定义，它就分离且区别于父类。子类含有一份从父类那里得来的行为的初始拷贝，但它可以覆盖这些继承的行为，甚至是定义新行为。

重要的是，要记住我们是在讨论父 **类**和子 **类**，而不是物理上的东西。这就是这个亲子比拟让人糊涂的地方，因为我们实际上应当说父类就是亲人的 DNA，而子类就是孩子的 DNA。我们不得不从两套 DNA 制造出（也就是“初始化”）人，用得到的物理上存在的人来与之进行谈话。

让我们把生物学上的亲子放在一边，通过一个稍稍不同的角度来看看继承：不同种类型的载具。这是用来理解继承的最经典（也是争议不断的）的比拟。

让我们重新审视本章前面的 Vehicle 和 Car 的讨论。考虑下面表达继承的类的假想代码：

```
class Vehicle {
    engines = 1

    ignition() {
        output( "Turning on my engine." )
    }

    drive() {
        ignition()
        output( "Steering and moving forward!" )
    }
}

class Car inherits Vehicle {
    wheels = 4

    drive() {
        inherited:drive()
        output( "Rolling on all ", wheels, " wheels!" )
    }
}

class SpeedBoat inherits Vehicle {
    engines = 2

    ignition() {
        output( "Turning on my ", engines, " engines." )
    }

    pilot() {
        inherited:drive()
        output( "Speeding through the water with ease!" )
    }
}
```

注意：为了简洁明了，这些类的构造器被省略了。

我们定义 Vehicle 类，假定它有一个引擎，有一个打开打火器的方法，和一个行驶的方法。但你永远也不会制造一个泛化的“载具”，所以在这里它只是一个概念的抽象。

然后我们定义了两种具体的载具：Car 和 SpeedBoat。它们都继承 Vehicle 的泛化性质，但之后它们都对这些性质进行了恰当的特化。一辆车有 4 个轮子，一艘快艇有两个引擎，意味着它需要在打火时需要特别注意要启动两个引擎。

多态（Polymorphism）

Car 定义了自己的 drive() 方法，它覆盖了从 Vehicle 继承来的同名方法。但是，Car 的 drive() 方法调用了 inherited:drive()，这表示 Car 可以引用它继承的，覆盖之前的原版 drive()。SpeedBoat 的 pilot() 方法也引用了它继承的 drive() 拷贝。

这种技术称为“多态（polymorphism）”，或“虚拟多态（virtual polymorphism）”。对我们当前的情况更具体一些，我们称之为“相对多态（relative polymorphism）”。

多态这个话题比我们可以在这里谈到的内容要宽泛的多，但我们当前的“相对”意味着一个特殊层面：任何方法都可以引用位于继承层级上更高一层的其他（同名或不同名的）方法。我们说“相对”，因为我们不绝对定义我们想访问继承的哪一层（也就是类），而实质上用“向上一层”来相对地引用。

在许多语言中，在这个例子中出现 inherited: 的地方使用了 super 关键字，它基于这样的想法：一个“超类（super class）”是当前类的父亲/祖先。

多态的另一个方面是，一个方法名可以在继承链的不同层级上有多种定义，而且在解析哪个方法在被调用时，这些定义可以适当地被自动选择。

在我们上面的例子中，我们看到这种行为发生了两次：drive() 在 Vehicle 和 Car 中定义，而 ignition() 在 Vehicle 和 SpeedBoat 中定义。

注意：另一个传统面向类语言通过 super 给你的能力，是从子类的构造器中直接访问父类构造器。这很大程度上是对的，因为对真正的类来说，构造器属于这个类。然而在 JS 中，这是相反的 —— 实际上认为“类”属于构造器（Foo.prototype... 类型引用）更恰当。因为在 JS 中，父子关系仅存在于它们各自的构造器的两个 .prototype 对象间，构造器本身不直接关联，而且没有简单的方法从一个中相对引用另一个（参见附录 A，看看 ES6 中用 super “解决”此问题的 class）。

可以从 ignition() 中具体看出多态的一个有趣的含义。在 pilot() 内部，一个相对多态引用指向了（被继承的）Vehicle 版本的 drive()。而这个 drive() 仅仅通过名称（不是相对引用）来引用 ignition() 方法。

语言的引擎会使用哪一个版本的 ignition()？是 Vehicle 的还是 SpeedBoat 的？**它会使用 SpeedBoat 版本的 ignition()**。如果你 能初始化 Vehicle 类自身，并且调用它的 drive()，那么语言引擎将会使用 Vehicle 的 ignition() 定义。

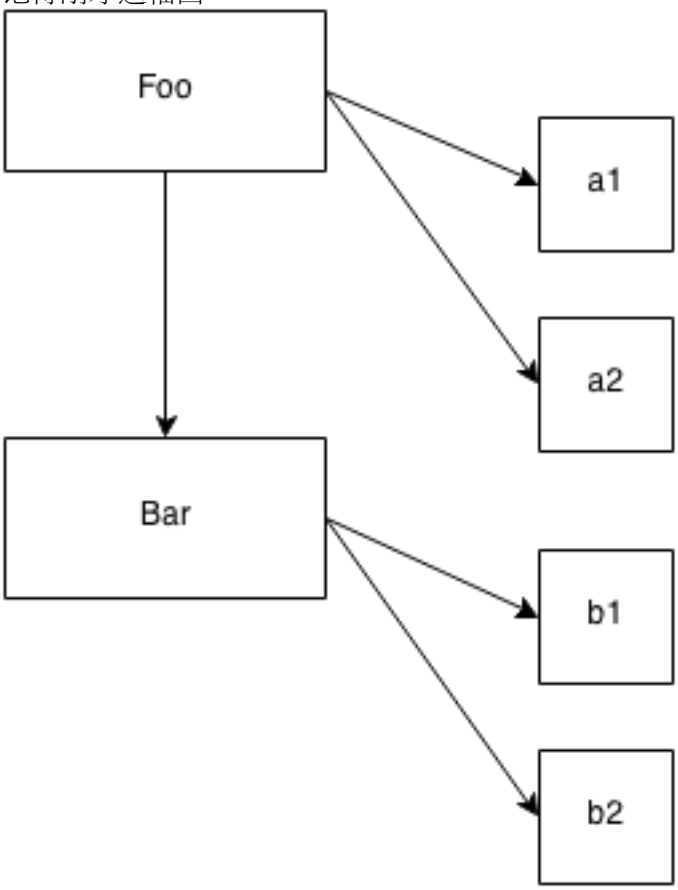
换句话说，ignition() 方法的定义，根据你引用的实例是哪个类（继承层级）而 *多态*（改变）。

这看起来过于深入学术细节了。不过为了好好地与 JavaScript 的 [[Prototype]] 机制的类

似行为进行对比，理解这些细节还是很重要的。

如果类是继承而来的，**对这些类本身**（不是由它们创建的对象!）有一个方法可以 相对地引用它们继承的对象，这个相对引用通常称为 `super`。

记得刚才这幅图：



注意对于实例化 (`a1`、`a2`、`b1`、和 `b2`) 和继承 (`Bar`)，箭头如何表示拷贝操作。

从概念上讲，看起来子类 `Bar` 可以使用相对多态引用（也就是 `super`）来访问它的父类 `Foo` 的行为。然而在现实中，子类不过是被给与了一份它从父类继承来的行为的拷贝而已。如果子类“覆盖”一个它继承的方法，原版的方法和覆盖版的方法实际上都是存在的，所以它们都是可以访问的。

不要让多态把你搞糊涂，使你认为子类是链接到父类上的。子类得到一份它需要从父类继承的东西的拷贝。**类继承意味着拷贝**。

多重继承（Multiple Inheritance）

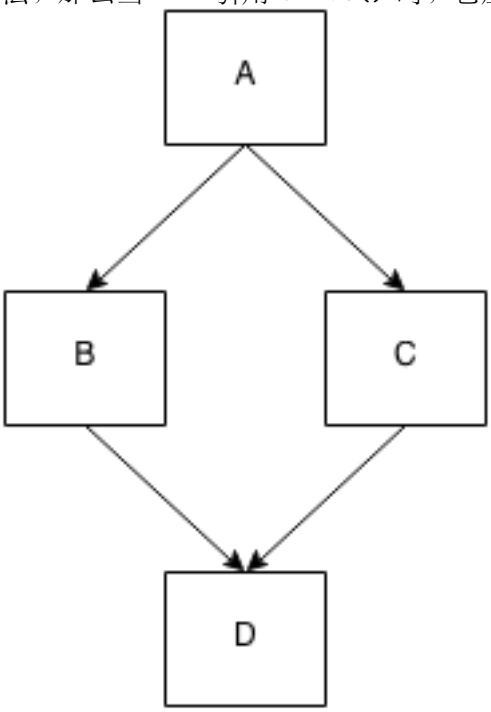
能回想起我们早先提到的亲子和 DNA 吗？我们说过这个比拟有些奇怪，因为生物学上大多数后代来自于双亲。如果类可以继承自其他两个类，那么这个亲子比拟会更合适一些。

有些面向类的语言允许你指定一个以上的“父类”来进行“继承”。多重继承意味着每个父类的定义都被拷贝到子类中。

表面上看来，这是对面向类的一个强大的加成，给我们能力去将更多功能组合在一起。然

而，这无疑会产生一些复杂的问题。如果两个父类都提供了名为 `drive()` 的方法，在子类中的 `drive()` 引用将会解析为哪个版本？你会总是不得不手动指明哪个父类的 `drive()` 是你想要的，从而失去一些多态继承的优雅之处吗？

还有另外一个所谓的“钻石问题”：子类“D”继承自两个父类（“B”和“C”），它们两个又继承自共通的父类“A”。如果“A”提供了方法 `drive()`，而“B”和“C”都覆盖（多态地）了这个方法，那么当“D”引用 `drive()` 时，它应当使用那个版本呢（`B:drive()` 还是 `C:drive()`）？



事情会比我们这样窥豹一斑能看到的复杂得多。我们在这里将它们提出来，只是便于我们可以将它和 JavaScript 机制的工作方式比较。

JavaScript 更简单：它不为“多重继承”提供原生机制。许多人认为这是好事，因为省去的复杂性要比“减少”的功能多得多。但是这并不能阻挡开发者们用各种方法来模拟它，我们接下来就看看。

混合（Mixin）

当你“继承”或是“实例化”时，JavaScript 的对象机制不会 *自动地*执行拷贝行为。很简单，在 JavaScript 中没有“类”可以拿来实例化，只有对象。而且对象也不会被拷贝到另一个对象中，而是被 *链接在一起*（详见第五章）。

因为在其他语言中观察到的类的行为意味着拷贝，让我们来看看 JS 开发者如何在 JavaScript 中 **模拟**这种 *缺失*的类的拷贝行为：mixins（混合）。我们会看到两种“mixin”：**明确的（explicit）**和 **隐含的（implicit）**。

明确的 Mixin（Explicit Mixins）

让我们再次回顾前面的 Vehicle 和 Car 的例子。因为 JavaScript 不会自动地将行为从 Vehicle 拷贝到 Car，我们可以建造一个工具来手动拷贝。这样的工具经常被许多库/框架称为 extend(..)，但为了便于说明，我们在这里叫它 mixin(..)。

```
// 大幅简化的 `mixin(..)` 示例:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // 仅拷贝非既存内容
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}

var Vehicle = {
    engines: 1,

    ignition: function() {
        console.log( "Turning on my engine." );
    },

    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    }
} );
```

注意：重要的细节：我们谈论的不再是类，因为在 JavaScript 中没有类。`Vehicle` 和 `Car` 分别只是我们实施拷贝的源和目标对象。

`Car` 现在拥有了一份从 `Vehicle` 得到的属性和函数的拷贝。技术上讲，函数实际上没有被复制，而是指向函数的 *引用*被复制了。所以，`Car` 现在有一个称为 `ignition` 的属性，它是一个 `ignition()` 函数引用的拷贝；而且它还有一个称为 `engines` 的属性，持有从 `Vehicle` 拷贝来的值 `1`。

`Car` *已经*有了 `drive` 属性（函数），所以这个属性引用没有被覆盖（参见上面 `mixin(..)` 的 `if` 语句）。

重温“多态（Polymorphism）”

我们来考察一下这个语句：`Vehicle.drive.call( this )`。我将之称为“显式假想多态（explicit pseudo-polymorphism）”。回想一下，我们前一段假想代码的这一行是我们称之为“相对多态（relative polymorphism）”的 `inherited:drive()`。

JavaScript 没有能力实现相对多态（ES6 之前，见附录 A）。所以，**因为 `Car` 和 `Vehicle` 都有一个名为 `drive()` 的函数**，为了在它们之间区别调用，我们必须使用绝对（不是相对）引用。我们明确地用名称指出 `Vehicle` 对象，然后在它上面调用 `drive()` 函数。

但如果我们说 `Vehicle.drive()`，那么这个函数调用的 `this` 绑定将会是 `Vehicle` 对象，而不是 `Car` 对象（见第二章），那不是我们想要的。所以，我们使用 `.call( this )`（见第二章）来保证 `drive()` 在 `Car` 对象的环境中被执行。

注意：如果 `Car.drive()` 的函数名称标识符没有与 `Vehicle.drive()` 的重叠（也就是“遮蔽（shadowed）”；见第五章），我们就不会有机会演示“方法多态（method polymorphism）”。因为那样的话，一个指向 `Vehicle.drive()` 的引用会被 `mixin(..)` 调用拷贝，而我们可以使用 `this.drive()` 直接访问它。被选用的标识符重叠 **遮蔽**就是为什么我们不得不使用更复杂的 *显式假想多态（explicit pseudo-polymorphism）* 的原因。

在拥有相对多态的面向类的语言中，`Car` 和 `Vehicle` 间的连接在类定义的顶端被建立一次，那里是维护这种关系的唯一场所。

但是由于 JavaScript 的特殊性，显式假想多态（因为遮蔽！）**在每一个你需要这种（假想）多态引用的函数中**建立了一种脆弱的手动/显式链接。这可能会显著地增加维护成本。而且，虽然显式假想多态可以模拟“多重继承”的行为，但这只会增加复杂性和代码脆弱性。

这种方法的结果通常是更加复杂，更难读懂，*而且*更难维护的代码。**应当尽可能地避免使用显式假想多态**，因为在大部分层面上它的代价要高于利益。

混合拷贝（Mixing Copies）

回忆一下上面的 `mixin(..)` 工具：

```
// 大幅简化的 `mixin()` 示例:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
```

```
        // 仅拷贝不存在的属性
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}
```

现在，我们考察一下 `mixin(..)` 如何工作。它迭代 `sourceObj`（在我们的例子中是 `Vehicle`）的所有属性，如果在 `targetObj`（在我们的例子中是 `Car`）中没有名称与之匹配的属性，它就进行拷贝。因为我们是在初始对象存在的情况下进行拷贝，所以我们要小心不要将目标属性覆盖掉。

如果在指明 `Car` 的具体内容之前，我们先进行拷贝，那么我们就可以省略对 `targetObj` 检查，但是这样做有些笨拙且低效，所以通常不优先选用：

```
// 另一种 mixin，对覆盖不太“安全”
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        targetObj[key] = sourceObj[key];
    }

    return targetObj;
}

var Vehicle = {
    // ...
};

// 首先，创建一个空对象
// 将 Vehicle 的内容拷贝进去
var Car = mixin( Vehicle, { } );

// 现在拷贝 Car 的具体内容
mixin( {
    wheels: 4,

    drive: function() {
        // ...
```

```
    }
}, Car );
```

不论哪种方法，我们都明确地将 `Vehicle` 中的非重叠内容拷贝到 `Car` 中。“mixin” 这个名称来自于解释这个任务的另一种方法：`Car` **混入** `Vehicle` 的内容，就像你吧巧克力碎片混入你最喜欢的曲奇饼面团。

这个拷贝操作的结果，是 `Car` 将会独立于 `Vehicle` 运行。如果你在 `Car` 上添加属性，它不会影响到 `Vehicle`，反之亦然。

注意：这里有几个小细节被忽略了。仍然有一些微妙的方法使两个对象在拷贝完成后还能互相“影响”对方，比如它们共享一个共通对象（比如数组）的引用。

由于两个对象还共享它们的共通函数的引用，这意味着 **即便手动将函数从一个对象拷贝（也就是混入）到另一个对象中，也不能** *实际上模拟发生在面向类的语言中的从类到实例的真正的复制*。

JavaScript 函数不能真正意义上地（以标准，可靠的方式）被复制，所以你最终得到的是同一个共享的函数对象（函数是对象；见第三章）的 **被复制的引用**。举例来说，如果你在一个共享的函数对象（比如 `ignition()`）上添加属性来修改它，`Vehicle` 和 `Car` 都会通过这个共享的引用而受“影响”。

在 JavaScript 中明确的 mixin 是一种不错的机制。但是它们显得言过其实。**和将一个属性定义两次相比**，将属性从一个对象拷贝到另一个对象并不会产生多少 *实际的*好处。而且由于我们刚才提到的函数对象引用的微妙之处，这显得尤为正确。

如果你明确地将两个或更多对象混入你的目标对象，你可以 **某种程度上模拟**“多重继承”的行为，但是在将方法或属性从多于一个源对象那里拷贝过来时，没有直接的办法可以解决名称的冲突。有些开发者/库使用“延迟绑定（late binding）”和其他诡异的替代方法来解决问题，但从根本上讲，这些“技巧”*通常*得不偿失（而且低效！）。

要小心的是，仅在明确的 mixin 能够实际提高代码可读性时使用它，而如果你发现它使代码变得更很难追溯，或在对象间建立了不必要或笨重的依赖性时，要避免使用这种模式。

如果正确使用 mixin 使你的问题变得比以前 *困难*，那么你可能应当停止使用 mixin。事实上，如果你不得不使用复杂的库/工具来处理这些细节，那么这可能标志着你正走在更困难，也许没必要的道路上。在第六章中，我们将试着提取一种更简单的方法来实现我们期望的结果，同时免去这些周折。

寄生继承（Parasitic Inheritance）

明确的 mixin 模式的一个变种，在某种意义上是明确的而在某种意义上是隐含的，称为“寄生继承（Parasitic Inheritance）”，它主要是由 Douglas Crockford 推广的。

这是它如何工作：

```
// “传统的 JS 类” `Vehicle`
function Vehicle() {
    this.engines = 1;
```

```
}
Vehicle.prototype.ignition = function() {
    console.log( "Turning on my engine." );
};
Vehicle.prototype.drive = function() {
    this.ignition();
    console.log( "Steering and moving forward!" );
};

// “寄生类” `Car`
function Car() {
    // 首先, `car` 是一个 `Vehicle`
    var car = new Vehicle();

    // 现在, 我们修改 `car` 使它特化
    car.wheels = 4;

    // 保存一个 `Vehicle::drive()` 的引用
    var vehDrive = car.drive;

    // 覆盖 `Vehicle::drive()`
    car.drive = function() {
        vehDrive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    };

    return car;
}

var myCar = new Car();

myCar.drive();
// Turning on my engine.
// Steering and moving forward!
// Rolling on all 4 wheels!
```

如你所见，我们一开始从“父类”（对象）`Vehicle` 制造了一个定义的拷贝，之后将我们的“子类”（对象）定义混入其中（按照需要保留父类的引用），最后将组合好的对象 `car` 作为子类

实例传递出去。

注意：当我们调用 `new Car()` 时，一个新对象被创建并被 `Car` 的 `this` 所引用（见第二章）。但是由于我们没有使用这个对象，而是返回我们自己的 `car` 对象，所以这个初始化创建的对象就被丢弃了。因此，`Car()` 可以不用 `new` 关键字调用，就可以实现和上面代码相同的功能，而且还可以省去对象的创建和回收。

隐含的 Mixin（Implicit Mixins）

隐含的 mixin 和前面解释的 显式假想多态是紧密相关的。所以它们需要注意相同的事项。

考虑这段代码：

```
var Something = {
    cool: function() {
        this.greeting = "Hello World";
        this.count = this.count ? this.count + 1 : 1;
    }
};

Something.cool();
Something.greeting; // "Hello World"
Something.count; // 1

var Another = {
    cool: function() {
        // 隐式地将 `Something` 混入 `Another`
        Something.cool.call( this );
    }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1 (不会和 `Something` 共享状态)
```

`Something.cool.call( this )` 既可以在“构造器”调用中使用（最常见的情况），也可以在方法调用中使用（如这里所示），我们实质上“借用”了 `Something.cool()` 函数并在 `Another` 环境下，而非 `Something` 环境下调用它（通过 `this` 绑定，见第二章）。结果是，`Something.cool()` 中进行的赋值被实施到了 `Another` 对象而非 `Something` 对象。

那么，这就是说我们将 `Something` 的行为“混入”了 `Another`。

虽然这种技术看起来有效利用了 this 再绑定的功能，也就是生硬地调用 Something.cool.call(this)，但是这种调用不能被作为相对（也更灵活的）引用，所以你应当 **提高警惕**。一般来说，**应当尽量避免使用这种结构**以保持代码干净而且易于维护。

复习

类是一种设计模式。许多语言提供语法来启用自然而然的面向类的软件设计。JS 也有相似的语法，但是它的行为和你在其他语言中熟悉的工作原理 **有很大的不同**。

类意味着拷贝。

当一个传统的类被实例化时，就发生了类的行为向实例中拷贝。当类被继承时，也发生父类的行为向子类的拷贝。

多态（在继承链的不同层级上拥有同名的不同函数）也许看起来意味着一个从子类回到父类的相对引用链接，但是它仍然只是拷贝行为的结果。

JavaScript **不会自动地**（像类那样）在对象间创建拷贝。

mixin 模式常用于在 某种程度上模拟类的拷贝行为，但是这通常导致像显式假想多态那样（OtherObj.methodName.call(this, ...)）难看而且脆弱的语法，这样的语法又常导致更难懂和更难维护的代码。

明确的 mixin 和类 拷贝又不完全相同，因为对象（和函数!）仅仅是共享的引用被复制，不是对象/函数自身被复制。不注意这样的微小之处通常是各种陷阱的根源。

一般来讲，在 JS 中模拟类通常会比解决当前 真正的问题埋下更多的坑。

第五章: 原型（Prototype）

在第三，四章中，我们几次提到了 [[Prototype]] 链，但我们没有讨论它到底是什么。现在我们就详细讲解一下原型（prototype）。

注意：所有模拟类拷贝行为的企图，也就是我们在前面第四章描述的内容，称为各种种类的 “mixin”，和我们要在本章中讲解的 [[Prototype]] 链机制完全不同。

[[Prototype]]

JavaScript 中的对象有一个内部属性，在语言规范中称为 [[Prototype]]，它只是一个其他对象的引用。几乎所有的对象在被创建时，它的这个属性都被赋予了一个非 null 值。

注意：我们马上就会看到，一个对象拥有一个空的 [[Prototype]] 链接是 可能的，虽然这有些不寻常。

考虑下面的代码：

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

```
myObject.a; // 2
```

[[Prototype]] 引用有什么用? 在第三章中，我们讲解了 [[Get]] 操作，它会在你引用一个对象上的属性时被调用，比如 myObject.a。对于默认的 [[Get]] 操作来说，第一步就是检查对象本身是否拥有一个 a 属性，如果有，就使用它。

注意: ES6 的代理 (Proxy) 超出了我们要在本书内讨论的范围 (将会在本系列的后续书目中涵盖!)，但是如果加入 Proxy，我们在这里讨论的关于普通 [[Get]] 和 [[Put]] 的行为都是不被采用的。

但是如果 myObject 上 **不**存在 a 属性时，我们就将注意力转向对象的 [[Prototype]] 链。

如果默认的 [[Get]] 操作不能直接在对象上找到被请求的属性，那么它会沿着对象的 [[Prototype]] **链**继续处理。

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

// 创建一个链接到 `anotherObject` 的对象
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

注意: 我们马上就会解释 Object.create(..) 是做什么，如何做的。眼下先假设，它创建了一个对象，这个对象带有一个链到指定对象的 [[Prototype]] 链接，这个链接就是我们要讲解的。

那么，我们现在让 myObject [[Prototype]] 链到了 anotherObject。虽然很明显 myObject.a 实际上不存在，但是无论如何属性访问成功了 (在 anotherObject 中找到了)，而且确实找到了值 2。

但是，如果在 anotherObject 上也没有找到 a，而且如果它的 [[Prototype]] 链不为空，就沿着它继续查找。

这个处理持续进行，直到找到名称匹配的属性，或者 [[Prototype]] 链终结。如果在链条的末尾都没有找到匹配的属性，那么 [[Get]] 操作的返回结果为 undefined。

和这种 [[Prototype]] 链查询处理相似，如果你使用 for..in 循环迭代一个对象，所有在它的链条上可以到达的 (并且是 enumerable —— 见第三章) 属性都会被枚举。如果你使用 in 操作符来测试一个属性在一个对象上的存在性，in 将会检查对象的整个链条 (不管 可枚举性)。

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

```
// 创建一个链接到 `anotherObject` 的对象
var myObject = Object.create( anotherObject );

for (var k in myObject) {
    console.log("found: " + k);
}
// 找到: a

("a" in myObject); // true
```

所以，当你以各种方式进行属性查询时，[[Prototype]] 链就会一个链接一个链接地被查询。一旦找到属性或者链条终结，这种查询就会停止。

Object.prototype

但是 [[Prototype]] 链到底在 哪里 “终结”?

每个 普通的 [[Prototype]] 链的最顶端，是内建的 Object.prototype。这个对象包含各种在整个 JS 中被使用的共通工具，因为 JavaScript 中所有普通（内建，而非被宿主环境扩展的）的对象都 “衍生自”（也就是，使它们的 [[Prototype]] 顶端为）Object.prototype 对象。

你会在这里发现一些你可能很熟悉的工具，比如 .toString() 和 .valueOf()。在第三章中，我们介绍了另一个：.hasOwnProperty(..)。还有另外一个你可能不太熟悉，但我们将在这一章里讨论的 Object.prototype 上的函数是 .isPrototypeOf(..)。

设置与遮蔽属性

回到第三章，我们提到过在对象上设置属性要比仅仅在对象上添加新属性或改变既存属性的值更加微妙。现在我们将更完整地重温这个话题。

```
myObject.foo = "bar";
```

如果 myObject 对象已经直接拥有了普通的名为 foo 的数据访问器属性，那么这个赋值就和改变既存属性的值一样简单。

如果 foo 还没有直接存在于 myObject，[[Prototype]] 就会被遍历，就像 [[Get]] 操作那样。如果在链条的任何地方都没有找到 foo，那么就会像我们期望的那样，属性 foo 就以指定的值被直接添加到 myObject 上。

然而，如果 foo 已经存在于链条更高层的某处，myObject.foo = "bar" 赋值就可能会发生微妙的（也许令人诧异的）行为。我们一会儿就详细讲解。

如果属性名 foo 同时存在于 myObject 本身和从 myObject 开始的 [[Prototype]] 链的更高层，这样的情况称为 遮蔽。直接存在于 myObject 上的 foo 属性会 遮蔽任何出现在链条高层的 foo 属性，因为 myObject.foo 查询总是在寻找链条最底层的 foo 属性。

正如我们被暗示的那样，在 myObject 上的 foo 遮蔽没有看起来那么简单。我们现在来考察 myObject.foo = "bar" 赋值的三种场景，当 foo **不直接存在**于 myObject，但 **存在**于 myObject 的 [[Prototype]] 链的更高层时：

1. 如果一个普通的名为 foo 的数据访问属性在 [[Prototype]] 链的高层某处被找到，**而且没有被标记为只读 (writable:false)**，那么一个名为 foo 的新属性就直接添加到 myObject 上，形成一个 **遮蔽属性**。
2. 如果一个 foo 在 [[Prototype]] 链的高层某处被找到，但是它被标记为 **只读 (writable:false)** ，那么设置既存属性和在 myObject 上创建遮蔽属性都是 **不允许** 的。如果代码运行在 strict mode 下，一个错误会被抛出。否则，这个设置属性值的操作会被无声地忽略。不论怎样，**没有发生遮蔽**。
3. 如果一个 foo 在 [[Prototype]] 链的高层某处被找到，而且它是一个 setter（见第三章），那么这个 setter 总是被调用。没有 foo 会被添加到（也就是遮蔽在）myObject 上，这个 foo setter 也不会被重定义。

大多数开发者认为，如果一个属性已经存在于 [[Prototype]] 链的高层，那么对它的赋值（[[Put]]）将总是造成遮蔽。但如你所见，这仅在刚才描述的三中场景中的一种（第一种）中是对的。

如果你想在第二和第三种情况中遮蔽 foo，那你就不能使用 = 赋值，而必须使用 Object.defineProperty(..)（见第三章）将 foo 添加到 myObject。

注意：第二种情况可能是三种情况中最让人诧异的了。只读属性的存在会阻止同名属性在 [[Prototype]] 链的低层被创建（遮蔽）。这个限制的主要原因是为了增强类继承属性的幻觉。如果你想象位于链条高层的 foo 被继承（拷贝）至 myObject，那么在 myObject 上强制 foo 属性不可写就有道理。但如果你将幻觉和现实分开，而且认识到 实际上没有这样的继承拷贝发生（见第四，五章），那么仅因为某些其他的对象上拥有不可写的 foo，而导致 myObject 不能拥有 foo 属性就有些不自然。而且更奇怪的是，这个限制仅限于 = 赋值，当使用 Object.defineProperty(..) 时不被强制。

如果你需要在方法间进行委托，**方法**的遮蔽会导致难看的 显式假想多态（见第四章）。一般来说，遮蔽与它带来的好处相比太过复杂和微妙了，**所以你应当尽量避免它**。第六章介绍另一种设计模式，它提倡干净而且不鼓励遮蔽。

遮蔽甚至会以微妙的方式隐含地发生，所以要想避免它必须小心。考虑这段代码：

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

var myObject = Object.create( anotherObject );

anotherObject.a; // 2
```

```
myObject.a; // 2

anotherObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "a" ); // false

myObject.a++; // 噢，隐式遮蔽!

anotherObject.a; // 2
myObject.a; // 3

myObject.hasOwnProperty( "a" ); // true
```

虽然看起来 myObject.a++ 应当（通过委托）查询并 原地递增 anotherObject.a 属性，但是 ++ 操作符相当于 myObject.a = myObject.a + 1。结果就是在 [[Prototype]] 上进行 a 的 [[Get]] 查询，从 anotherObject.a 得到当前的值 2，将这个值递增 1，然后将值 3 用 [[Put]] 赋值到 myObject 上的新遮蔽属性 a 上。噢!

修改你的委托属性时要非常小心。如果你想递增 anotherObject.a，那么唯一正确的方法是 anotherObject.a++。

"类"

现在你可能会想知道："为什么一个对象需要链到另一个对象？"真正的好处是什么？这是一个很恰当的问题，但在我们能够完全理解和体味它是什么和如何有用之前，我们必须首先理解 [[Prototype]] **不是**什么。

正如我们在第四章讲解的，在 JavaScript 中，对于对象来说没有抽象模式/蓝图，即没有面向类的语言中那样的称为类的东西。JavaScript **只有**对象。

实际上，在所有语言中，JavaScript **几乎是独一无二的**，也许是唯一的可以被称为"面向对象"的语言，因为可以根本没有类而直接创建对象的语言很少，而 JavaScript 就是其中之一。

在 JavaScript 中，类不能（因为根本不存在）描述对象可以做什么。对象直接定义它自己的行为。**这里 仅有对象**。

"类" 函数

在 JavaScript 中有一种奇异的行为被无耻地滥用了许多年来 山寨成某些 看起来像 "类" 的东西。我们来仔细看看这种方式。

"某种程度的类" 这种奇特的行为取决于函数的一个奇怪的性质：所有的函数默认都会得到一个公有的，不可枚举的属性，称为 prototype，它可以指向任意的对象。

```
function Foo() {
    // ...
```

```
}

Foo.prototype; // { }
```

这个对象经常被称为 “Foo 的原型”，因为我们通过一个不幸地被命名为 Foo.prototype 的属性引用来访问它。然而，我们马上会看到，这个术语命中注定地将我们搞糊涂。为了取代它，我将它称为 “以前被认为是 Foo 的原型的对象”。只是开个玩笑。“一个被随意标记为 ‘Foo 点儿原型’ 的对象”，怎么样?

不管我们怎么称呼它，这个对象到底是什么?

解释它的最直接的方法是，每个由调用 new Foo() (见第二章) 而创建的对象将最终 (有些随意地) 被 [[Prototype]] 链接到这个 “Foo 点儿原型” 对象。

让我们描绘一下:

```
function Foo() {
    // ...
}

var a = new Foo();

Object.getPrototypeOf( a ) === Foo.prototype; // true
```

当通过调用 new Foo() 创建 a 时，会发生的事情之一 (见第二章了解所有 四个步骤) 是，a 得到一个内部 [[Prototype]] 链接，此链接链到 Foo.prototype 所指向的对象。

停一会来思考一下这句话的含义。

在面向类的语言中，可以制造一个类的多个 **拷贝** (即 “实例”)，就像从模具中冲压出某些东西一样。我们在第四章中看到，这是因为初始化 (或者继承) 类的处理意味着，“将行为计划从这个类拷贝到物理对象中”，对于每个新实例这都会发生。

但是在 JavaScript 中，没有这样的拷贝处理发生。你不会创建类的多个实例。你可以创建多个对象，它们的 [[Prototype]] 连接至一个共通对象。但默认地，没有拷贝发生，如此这些对象彼此间最终不会完全分离和切断关系，而是 **链接在一起**。

new Foo() 得到一个新对象 (我们叫他 a)，这个新对象 a 内部地被 [[Prototype]] 链接至 Foo.prototype 对象。

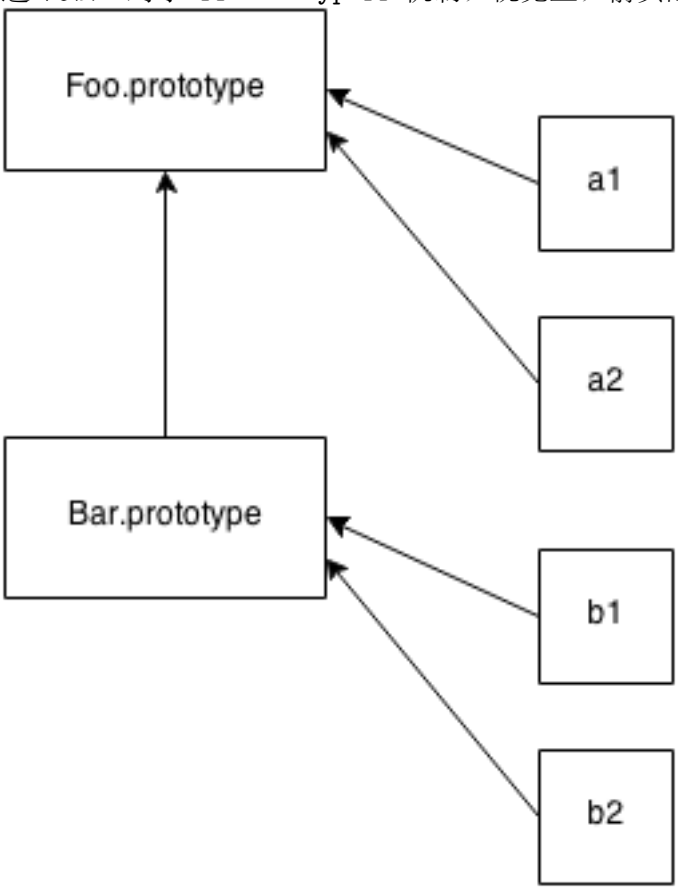
结果我们得到两个对象，彼此链接。如是而已。我们没有初始化一个对象。当然我们也没有做任何从一个 “类” 到一个实体对象的拷贝。我们只是让两个对象互相链接在一起。

事实上，这个使大多数 JS 开发者无法理解的秘密，是因为 new Foo() 函数调用实际上几乎和建立链接的处理没有任何 直接关系。**它是某种偶然的副作用**。 new Foo() 是一个间接的，迂回的方法来得到我们想要的: **一个被链接到另一个对象的对象**。

我们能用更直接的方法得到我们想要的吗? **可以!** 这位英雄就是 Object.create(..)。我们过会儿就谈到它。

名称的意义何在?

在 JavaScript 中，我们不从一个对象（“类”）向另一个对象（“实例”）拷贝。我们在对象之间制造 链接。对于 `[[Prototype]]` 机制，视觉上，箭头的移动方向是从右至左，由下至上。



这种机制常被称为“原型继承（prototypal inheritance）”（我们很快就用代码说明），它经常被说成是动态语言版的“类继承”。这种说法试图建立在面向类世界中对“继承”含义的共识上。但是 弄拧（**意思是：抹平**）了被理解的语义，来适应动态脚本。

先入为主，“继承”这个词有很强烈的含义（见第四章）。仅仅在它前面加入“原型”来区别于 JavaScript 中 实际上几乎相反的行为，使真相在泥泞般的困惑中沉睡了近二十年。

我想说，将“原型”贴在“继承”之前很大程度上搞反了它的实际意义，就像一只手拿着一个桔子，另一手拿着一个苹果，而坚持说苹果是一个“红色的桔子”。无论我在它前面放什么令人困惑的标签，那都不会改变一个水果是苹果而另一个是桔子的 事实。

更好的方法是直白地将苹果称为苹果——使用最准确和最直接的术语。这样能更容易地理解它们的相似之处和 **许多不同之处**，因为我们都对“苹果”的意义有一个简单的，共享的理解。

由于用语的模糊和歧义，我相信，对于解释 JavaScript 机制真正如何工作来说，“原型继承”这个标签（以及试图错误地应用所有面向类的术语，比如“类”，“构造器”，“实例”，“多态”等）本身带来的 **危害比好处多**。

“继承”意味着 拷贝操作，而 JavaScript 不拷贝对象属性（原生上，默认地）。相反，JS 在两个对象间建立链接，一个对象实质上可以将对属性/函数的访问 委托到另一个对象上。对于

描述 JavaScript 对象链接机制来说，“委托” 是一个准确得多的术语。

另一个有时被扔到 JavaScript 旁边的术语是 “差分继承”。它的想法是，我们可以用一个对象与一个更泛化的对象的 不同来描述一个它的行为。比如，你要解释汽车是一种载具，与其重新描述组成一个一般载具的所有特点，不如只说它有四个轮子。

如果你试着想象，在 JS 中任何给定的对象都是通过委托可用的所有行为的总和，而且 **在你思维中你扁平化**所有的行为到一个有形的 东西中，那么你就可以（八九不离十地）看到 “差分继承” 是如何自圆其说的。

但正如 “原型继承”，“差分继承” 假意使你的思维模型比在语言中物理发生的事情更重要。它忽视了这样一个事实：对象 B 实际上不是一个差异结构，而是由一些定义好的特定性质，与一些没有任何定义的 “漏洞” 组成的。正是通过这些 “漏洞”（缺少定义），委托可以接管并且动态地用委托行为 “填补” 它们。

对象不是像 “差分继承” 的思维模型所暗示的那样，原生默认地，**通过拷贝**扁平化到一个单独的差异对象中。因此，对于描述 JavaScript 的 [[Prototype]] 机制如何工作来说，“差分继承” 就不是自然合理。

你 可以选择偏向 “差分继承” 这个术语和思维模型，这是个人口味的问题，但是不能否认这个事实：它 仅仅符合你思维中的主观过程，不是引擎的物理行为。

“构造器”（Constructors）

让我们回到早先的代码：

```
function Foo() {
    // ...
}

var a = new Foo();
```

到底是什么导致我们认为 Foo 是一个 “类”?

其一，我们看到了 new 关键字的使用，就像面向类语言中人们构建类的对象那样。另外，它看起来我们事实上执行了一个类的 构造器方法，因为 Foo() 实际上是个被调用的方法，就像当你初始化一个真实的类时这个类的构造器被调用的那样。

为了使 “构造器” 的语义更令人糊涂，被随意贴上标签的 Foo.prototype 对象还有另外一招。考虑这段代码：

```
function Foo() {
    // ...
}

Foo.prototype.constructor === Foo; // true
```

```
var a = new Foo();
a.constructor === Foo; // true
```

Foo.prototype 对象默认地（就在代码段中第一行中声明的地方!）得到一个公有的，称为 .constructor 的不可枚举（见第三章）属性，而且这个属性回头指向这个对象关联的函数（这里是 Foo）。另外，我们看到被 “构造器” 调用 new Foo() 创建的对象 a 看起来也拥有一个称为 .constructor 的属性，也相似地指向 “创建它的函数”。

注意: 这实际上不是真的。a 上没有 .constructor 属性，而 a.constructor 确实解析成了 Foo 函数，“constructor” 并不像它看起来的那样实际意味着 “被 XX 创建”。我们很快就会解释这个奇怪的地方。

哦，是的，另外…… 根据 JavaScript 世界中的惯例，“类” 都以大写字母开头的单词命名，所以使用 Foo 而不是 foo 强烈地意味着我们打算让它成为一个 “类”。这对你来说太明显了，对吧!?

注意: 这个惯例是如此强大，以至于如果你在一个小写字母名称的方法上使用 new 调用，或并没有在一个大写字母开头的函数上使用 new，许多 JS 语法检查器将会报告错误。这是因为我们如此努力地想要在 JavaScript 中将（假的）“面向类” 搞对，所以我们建立了这些语法规则来确保我们使用了大写字母，即便对 JS 引擎来讲，大写字母根本没有 任何意义。

构造器还是调用?

上面的代码的段中，我们试图认为 Foo 是一个 “构造器”，是因为我们用 new 调用它，而且我们观察到它 “构建” 了一个对象。

在现实中，Foo 不会比你的程序中的其他任何函数 “更像构造器”。函数自身 **不是**构造器。但是，当你在普通函数调用前面放一个 new 关键字时，这就将函数调用变成了 “构造器调用”。事实上，new 在某种意义上劫持了普通函数并将它以另一种方式调用: 构建一个对象，**外加这个函数要做的其他任何事**。

举个例子:

```
function NothingSpecial() {
    console.log( "Don't mind me!" );
}

var a = new NothingSpecial();
// "Don't mind me!"

a; // {}
```

NothingSpecial 仅仅是一个普通的函数，但当用 new 调用时，几乎是一种副作用，它会构建一个对象，并被我们赋值到 a。这个 **调用**是一个 构造器调用，但是 NothingSpecial 本身并不是一个 构造器。

换句话说，在 JavaScrip t 中，更合适的说法是，“构造器” 是在前面 **用 new 关键字调用的任何函数**。

函数不是构造器，但是当且仅当 new 被使用时，函数调用是一个 “构造器调用”。

机制

仅仅是这些原因使得 JavaScript 中关于 “类” 的讨论变得命运多舛吗?

不全是。 JS 开发者们努力尽可能地模拟面向类:

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

var a = new Foo( "a" );
var b = new Foo( "b" );

a.myName(); // "a"
b.myName(); // "b"
```

这段代码展示了另外两种 “面向类” 的花招:

1. this.name = name: 在每个对象（分别在 a 和 b 上；参照第二章关于 this 绑定的内容）上添加了 .name 属性，和类的实例包装数据值很相似。
2. Foo.prototype.myName = ...: 这也许是更有趣的技术，它在 Foo.prototype 对象上添加了一个属性（函数）。现在，也许让人惊奇，a.myName() 可以工作。但是是如何工作的?

在上面的代码段中，有很强的倾向认为当 a 和 b 被创建时，Foo.prototype 上的属性/函数被 拷贝到了 a 与 b 俩个对象上。**但是，这没有发生**。

在本章开头，我们解释了 [[Prototype]] 链，以及它如何作为默认的 [[Get]] 算法的一部分，在不能直接在对象上找到属性引用时提供后备的查询步骤。

于是，得益于他们被创建的方式，a 和 b 都最终拥有一个内部的 [[Prototype]] 链接链到 Foo.prototype。当无法分别在 a 和 b 中找到 myName 时，就会在 Foo.prototype 上找到（通过委托，见第六章）。

复活 "构造器"

回想我们刚才对 .constructor 属性的讨论，怎么看起来 a.constructor === Foo 为 true 意味着 a 上实际拥有一个 .constructor 属性，指向 Foo? **不对**。

这只是一种不幸的混淆。实际上，.constructor 引用也 *委托*到了 Foo.prototype，它 **恰好**有一个指向 Foo 的默认属性。

这 *看起来*方便得可怕，一个被 Foo 构建的对象可以访问指向 Foo 的 .constructor 属性。但这只不过是安全感上的错觉。它是一个欢乐的巧合，几乎是误打误撞，通过默认的 [[Prototype]] 委托 a.constructor *恰好*指向 Foo。实际上 .constructor 意味着 "被 XX 构建" 这种注定失败的臆测会以几种方式来咬到你。

第一，在 Foo.prototype 上的 .constructor 属性仅当 Foo 函数被声明时才出现在对象上。如果你创建一个新对象，并用它替换函数默认的 .prototype 对象引用，这个新对象上将不会魔法般地得到 .contructor。

考虑这段代码:

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // 创建一个新的 prototype 对象

var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

Object(..) 没有 "构建" a1，是吧? 看起来确实是 Foo() "构建了" 它。许多开发者认为 Foo() 在执行构建，但当你认为 "构造器" 意味着 "被 XX 构建" 时，一切就都崩塌了，因为如果那样的话，a1.constructor 应当是 Foo，但它不是!

发生了什么? a1 没有 .constructor 属性，所以它沿着 [[Prototype]] 链向上委托到了 Foo.prototype。但是这个对象也没有 .constructor (默认的 Foo.prototype 对象就会有!)，所以它继续委托，这次轮到了 Object.prototype，委托链的最顶端。*那个*对象上确实拥有 .constructor，它指向内建的 Object(..) 函数。

误解，消除。

当然，你可以把 .constructor 加回到 Foo.prototype 对象上，但是要做一些手动工作，特别是如果你想要它与原生的行为吻合，并不可枚举时 (见第三章)。

举例来说:

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // 创建一个新的 prototype 对象

// 需要正确地 "修复" 丢失的 `.constructor`
```

```
// 新对象上的属性以 `Foo.prototype` 的形式提供。
// `defineProperty(..)` 的内容见第三章。
Object.defineProperty( Foo.prototype, "constructor" , {
    enumerable: false,
    writable: true,
    configurable: true,
    value: Foo    // 使 `.constructor` 指向 `Foo`
} );
```

修复 .constructor 要花不少功夫。而且，我们做的一切是为了延续“构造器”意味着“被 XX 构建”的误解。这是一种昂贵的假象。

事实上，一个对象上的 .constructor 默认地随意指向一个函数，而这个函数反过来拥有一个指向被这个对象称为 .prototype 的对象。“构造器”和“原型”这两个词仅有松散的默认含义，可能是真的也可能不是真的。最佳方案是提醒你自己，“构造器不是意味着被 XX 构建”。

.constructor 不是一个魔法般不可变的属性。它是不可枚举的（见上面的代码段），但是它的值是可写的（可以改变），而且，你可以用你感觉合适的任何值在 [[Prototype]] 链上的任何对象上添加或覆盖（有意或无意地）名为 constructor 的属性。

根据 [[Get]] 算法如何遍历 [[Prototype]] 链，在任何地方找到的一个 .constructor 属性引用解析的结果可能与你期望的十分不同。

看到它的实际意义有多随便了吗?

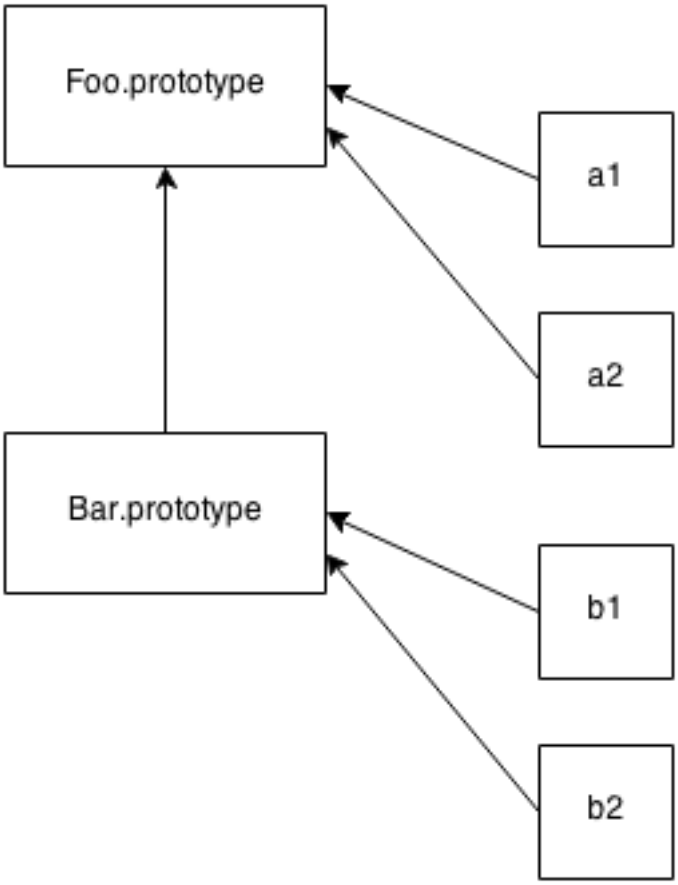
结果? 某些像 a1.constructor 这样随意的对象属性引用实际上不能被认为是默认的函数引用。还有，我们马上就会看到，通过一个简单的省略，a1.constructor 可以最终指向某些令人诧异，没道理的地方。

a1.constructor 是极其不可靠的，在你的代码中不应依赖的不安全引用。**一般来说，这样的引用应当尽量避免**。

“（原型）继承”

我们已经看到了一些近似的“类”机制黑进 JavaScript 程序。但是如果我们没有一种近似的“继承”，JavaScript 的“类”将会更空洞。

实际上，我们已经看到了一个常被称为“原型继承”的机制如何工作：a 可以“继承自” Foo.prototype，并因此可以访问 myName() 函数。但是我们传统的想法认为“继承”是两个“类”间的关系，而非“类”与“实例”的关系。



回想之前这幅图，它不仅展示了从对象（也就是“实例”）a1 到对象 Foo.prototype 的委托，而且从 Bar.prototype 到 Foo.prototype，这酷似类继承的亲子概念。酷似，除了方向，箭头表示的是委托链接，而不是拷贝操作。

这里是一段典型的创建这样的链接的“原型风格”代码：

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

function Bar(name,label) {
    Foo.call( this, name );
    this.label = label;
}

// 这里，我们创建一个新的 `Bar.prototype` 链接链到 `Foo.prototype`
```

```
Bar.prototype = Object.create( Foo.prototype );

// 注意! 现在 `Bar.prototype.constructor` 不存在了,
// 如果你有依赖这个属性的习惯的话, 它可以被手动“修复”。

Bar.prototype.myLabel = function() {
    return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"
```

注意: 要想知道为什么上面代码中的 this 指向 a，参见第二章。

重要的部分是 Bar.prototype = Object.create(Foo.prototype)。Object.create(..) 凭空 创建了一个“新”对象，并将这个新对象内部的 [[Prototype]] 链接到你指定的对象上（在这里是 Foo.prototype）。

换句话说，这一行的意思是：“做一个 新的链接到 ‘Foo 点儿 prototype’ 的 ‘Bar 点儿 prototype ’ 对象”。

当 function Bar() { .. } 被声明时，就像其他函数一样，拥有一个链到默认对象的 .prototype 链接。但是 那个对象没有链到我们希望的 Foo.prototype。所以，我们创建了一个 新对象，链到我们希望的地方，并将原来的错误链接的对象扔掉。

注意: 这里一个常见的误解/困惑是，下面两种方法 也能工作，但是他们不会如你期望的那样工作:

```
// 不会如你期望的那样工作!
Bar.prototype = Foo.prototype;

// 会如你期望的那样工作
// 但会带有你可能不想要的副作用 :(
Bar.prototype = new Foo();
```

Bar.prototype = Foo.prototype 不会创建新对象让 Bar.prototype 链接。它只是让 Bar.prototype 成为 Foo.prototype 的另一个引用，将 Bar 直接链到 Foo 链着的 **同一个对象**: Foo.prototype。这意味着当你开始赋值时，比如 Bar.prototype.myLabel = ...，你修改的 **不是一个分离的对象**而是那个被分享的 Foo.prototype 对象本身，它将影响到所有链接到 Foo.prototype 的对象。这几乎可以确定不是你想要的。如果这正是你想要的，那么你根本就不需要 Bar，你应当仅使用 Foo 来使你的代码更简单。

`Bar.prototype = new Foo()` **确实**创建了一个新的对象，这个新对象也的确链接到了我们希望的 `Foo.prototype`。但是，它是用 `Foo(..)` "构造器调用" 来这样做的。如果这个函数有任何副作用（比如 logging，改变状态，注册其他对象，**向 `this` 添加数据属性**，等等），这些副作用就会在链接时发生（而且很可能是对错误的对象!），而不是像可能希望的那样，仅最终在 `Bar()` 的 "后裔" 被创建时发生。

于是，我们剩下的选择就是使用 `Object.create(..)` 来制造一个新对象，这个对象被正确地链接，而且没有调用 `Foo(..)` 时所产生的副作用。一个轻微的缺点是，我们不得不创建新对象，并把旧的扔掉，而不是修改提供给我们的默认既存对象。

如果有一种标准且可靠地方法来修改既存对象的链接就好了。ES6 之前，有一个非标准的，而且不是完全对所有浏览器通用的方法：通过可以设置的 `.__proto__` 属性。ES6 中增加了 `Object.setPrototypeOf(..)` 辅助工具，它提供了标准且可预见的方法。

让我们一对一地比较一下 ES6 之前和 ES6 标准的技术如何处理将 `Bar.prototype` 链接至 `Foo.prototype`：

```
// ES6 以前
// 扔掉默认既存的 `Bar.prototype`
Bar.prototype = Object.create( Foo.prototype );

// ES6+
// 修改既存的 `Bar.prototype`
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

如果忽略 `Object.create(..)` 方式在性能上的轻微劣势（扔掉一个对象，然后被回收），其实它相对短一些而且可能比 ES6+ 的方式更易读。但两种方式可能都只是语法表面现象。

考察 "类" 关系

如果你有一个对象 `a` 并且希望找到它委托至哪个对象呢（如果有的话）? 考察一个实例（一个 JS 对象）的继承血统（在 JS 中是委托链接），在传统的面向类环境中称为 **自省** (*introspection*)（或 **反射** (*reflection*)）。

考虑下面的代码：

```
function Foo() {
    // ...
}

Foo.prototype.blah = ...;

var a = new Foo();
```

那么我们如何自省 a 来找到它的“祖先”（委托链）呢？一种方式是接受“类”的困惑：

```
a instanceof Foo; // true
```

instanceof 操作符的左侧操作数接收一个普通对象，右侧操作数接收一个 **函数**。instanceof 回答的问题是：**在 a 的整个 [[Prototype]] 链中，有没有出现那个被 Foo.prototype 所随便指向的对象？**

不幸的是，这意味着如果你拥有可以用于测试的 **函数**（Foo，和它带有的 .prototype 引用），你只能查询某些对象（a）的“祖先”。如果你有两个任意的对象，比如 a 和 b，而且你想调查是否 这些对象通过 [[Prototype]] 链相互关联，单靠 instanceof 帮不上什么忙。

注意：如果你使用内建的 .bind(..) 工具来制造一个硬绑定的函数（见第二章），这个被创建的函数将不会拥有 .prototype 属性。将 instanceof 与这样的函数一起使用时，将会透明地替换为创建这个硬绑定函数的 目标函数的 .prototype。

将硬绑定函数用于“构造器调用”十分罕见，但如果你这么做，它会表现得好像是 目标函数被调用了，这意味着将 instanceof 与硬绑定函数一起使用也会参照原版函数。

下面这段代码展示了试图通过“类”的语义和 instanceof 来推导 **两个对象**间的关系是多么荒谬：

```
// 用来检查 `o1` 是否关联到（委托至）`o2` 的帮助函数
function isRelatedTo(o1, o2) {
    function F(){}
    F.prototype = o2;
    return o1 instanceof F;
}

var a = {};
var b = Object.create( a );

isRelatedTo( b, a ); // true
```

在 isRelatedTo(..) 内部，我们借用一个一次性的函数 F，重新对它的 .prototype 赋值，使它随意地指向某个对象 o2，之后问 o1 是否是 F 的“一个实例”。很明显，o1 实际上不是继承或遗传自 F，甚至不是由 F 构建的，所以显而易见这种做法是愚蠢且让人困惑的。**这个问题归根结底是将类的语义强加于 JavaScript 的尴尬**，在这个例子中是由 instanceof 的间接语义揭露的。

第二种，也是更干净的方式，[[Prototype]] 反射：

```
Foo.prototype.isPrototypeOf( a ); // true
```

注意在这种情况下，我们并不真正关心（甚至 不需要）Foo，我们仅需要一个 **对象**（在我们的例子中被随意标志为 Foo.prototype）来与另一个 **对象**测试。isPrototypeOf(..) 回答的问题是：**在 a 的整个 [[Prototype]] 链中，Foo.prototype 出现过吗?**

同样的问题，和完全同样的答案。但是在第二种方式中，我们实际上不需要间接地引用一个 .prototype 属性将被自动查询的 **函数**（Foo）。

我们 只需要两个 **对象**来考察它们之间的关系。比如：

```
// 简单地: `b` 在 `c` 的 `[[Prototype]]` 链中出现过吗?
b.isPrototypeOf( c );
```

注意，这种方法根本不要求有一个函数（“类”）。它仅仅使用对象的直接引用 b 和 c，来查询他们的关系。换句话说，我们上面的 isRelatedTo(..) 工具是内建在语言中的，它的名字叫 isPrototypeOf(..)。

我们也可以直接取得一个对象的 [[Prototype]]。在 ES5 中，这么做的标准方法是：

```
Object.getPrototypeOf( a );
```

而且你将注意到对象引用是我们期望的：

```
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

大多数浏览器(不是全部!)还一种长期支持的,非标准方法可以访问内部的 [[Prototype]]：

```
a.__proto__ === Foo.prototype; // true
```

这个奇怪的 .__proto__（直到 ES6 才被标准化！）属性“魔法般地”取得一个对象内部的 [[Prototype]] 作为引用，如果你想要直接考察（甚至遍历：.__proto__.__proto__...）[[Prototype]] 链，这个引用十分有用。

和我们早先看到的 .constructor 一样，.__proto__ 实际上不存在于你考察的对象上（在我们的例子中是 a)。事实上，它和其他的共通工具在一起 (.toString(), .isPrototypeOf(..), 等等)，存在于（不可枚举地；见第二章）内建的 Object.prototype 上。

而且，.__proto__ 虽然看起来像一个属性，但实际上将它看做是一个 getter/setter（见第三章）更合适。

大致地，我们可以这样描述 .__proto__ 的实现（见第三章，对象属性的定义）：

```
Object.defineProperty( Object.prototype, "__proto__", {
    get: function() {
        return Object.getPrototypeOf( this );
    },
    set: function(o) {
        // ES6 的 setPrototypeOf(..)
```

```
        Object.setPrototypeOf( this, o );
        return o;
    }
} );
```

所以，当我们访问 `a.__proto__`（取得它的值）时，就好像调用 `a.__proto__()`（调用 getter 函数）一样。虽然 getter 函数存在于 `Object.prototype` 上（参照第二章，`this` 绑定规则），但这个函数调用将 `a` 用作它的 `this`，所以它相当于在说 `Object.getPrototypeOf( a )`。

`.__proto__` 还是一个可设置的属性，就像早先展示过的 ES6 `Object.setPrototypeOf(..)`。然而，一般来说你 **不应该改变一个既存对象的 `[[Prototype]]`**。

在某些允许对 `Array` 定义“子类”的框架中，深度地使用了一些非常复杂，高级的技术，但是这在一般的编程实践中经常是让人皱眉头的，因为这通常导致非常难理解/维护的代码。

注意：在 ES6 中，关键字 `class` 将允许某些近似方法，对像 `Array` 这样的内建类型“定义子类”。参见附录 A 中关于 ES6 中加入的 `class` 的讨论。

仅有一小部分例外（就像前面提到过的）会设置一个默认函数 `.prototype` 对象的 `[[Prototype]]`，使它引用其他的对象（`Object.prototype` 之外的对象）。它们会避免将这个默认对象完全替换为一个新的链接对象。否则，为了在以后更容易地阅读你的代码 **最好将对象的 `[[Prototype]]` 链接作为只读性质对待**。

注意：针对双下划线，特别是在像 `__proto__` 这样的属性中开头的部分，JavaScript 社区非官方地创造了一个术语：“dunder”。所以，那些 JavaScript 的“酷小子”们通常将 `__proto__` 读作“dunder proto”。

对象链接

正如我们看到的，`[[Prototype]]` 机制是一个内部链接，它存在于一个对象上，这个对象引用一些其他的对象。

这种链接（主要）在对一个对象进行属性/方法引用，但这样的属性/方法不存在时实施。在这种情况下，`[[Prototype]]` 链接告诉引擎在那个被链接的对象上查找这个属性/方法。接下来，如果这个对象不能满足查询，它的 `[[Prototype]]` 又会被查找，如此继续。这个在对象间的一系列链接构成了所谓的“原形链”。

创建链接

我们已经彻底揭露了为什么 JavaScript 的 `[[Prototype]]` 机制和 类 **不**一样，而且我们也看到了如何在正确的对象间创建 **链接**。

`[[Prototype]]` 机制的意义是什么？为什么总是见到 JS 开发者们费那么大力气（模拟类）在他们的代码中搞乱这些链接？

记得我们在本章很靠前的地方说过 `Object.create(..)` 是英雄吗？现在，我们准备好看看为什么了。

```
var foo = {
    something: function() {
        console.log( "Tell me something good..." );
    }
};

var bar = Object.create( foo );

bar.something(); // Tell me something good...
```

Object.create(..) 创建了一个链接到我们指定的对象（foo）上的新对象（bar），这给了我们 [[Prototype]] 机制的所有力量（委托），而且没有 new 函数作为类和构造器调用产生的所有没必要的复杂性，搞乱 .prototype 和 .constructor 引用，或任何其他的多余的东西。

注意： Object.create(null) 创建一个拥有空（也就是 null）[[Prototype]] 链接的对象，如此这个对象不能委托到任何地方。因为这样的对象没有原形链，instancof 操作符（前面解释过）没有东西可检查，所以它总返回 false。由于他们典型的用途是在属性中存储数据，这种特殊的空 [[Prototype]] 对象经常被称为“字典（dictionaries）”，这主要是因为它们不可能受到在 [[Prototype]] 链上任何委托属性/函数的影响，所以它们是纯粹的扁平数据存储。

我们不 *需要* 类来在两个对象间创建有意义的关系。我们需要 **真正关心** 的唯一问题是对象为了委托而链接在一起，而 Object.create(..) 给我们这种链接并且没有一切关于类的烂设计。

填补 Object.create()

Object.create(..) 在 ES5 中被加入。你可能需要支持 ES5 之前的环境（比如老版本的 IE），所以让我们来看一个 Object.create(..) 的简单 **部分** 填补工具，它甚至能在更老的 JS 环境中给我们所需的能力：

```
if (!Object.create) {
    Object.create = function(o) {
        function F(){}
        F.prototype = o;
        return new F();
    };
}
```

这个填补工具通过一个一次性的 F 函数并覆盖它的 .prototype 属性来指向我们想连接到的对象。之后我们用 new F() 构造器调用来制造一个将会链到我们指定对象上的新对象。

Object.create(..) 的这种用法是目前最常见的用法，因为它的这一部分是 *可以* 填补的。ES5 标准的内建 Object.create(..) 还提供了一个附加的功能，它是 **不能** 被 ES5 之前的版本填补的。如此，这个功能的使用远没有那么常见。为了完整性，让我们看看这个附加功能：

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

var myObject = Object.create( anotherObject, {
    b: {
        enumerable: false,
        writable: true,
        configurable: false,
        value: 3
    },
    c: {
        enumerable: true,
        writable: false,
        configurable: false,
        value: 4
    }
} );

myObject.hasOwnProperty( "a" ); // false
myObject.hasOwnProperty( "b" ); // true
myObject.hasOwnProperty( "c" ); // true

myObject.a; // 2
myObject.b; // 3
myObject.c; // 4
```

`Object.create(..)` 的第二个参数通过声明每个新属性的 *属性描述符*（见第三章）指定了要添加在新对象上的属性。因为在 ES5 之前的环境中填补属性描述符是不可能的，所以 `Object.create(..)` 的这个附加功能无法填补。

因为 `Object.create(..)` 的绝大多数用途都是使用填补安全的功能子集，所以大多数开发者在 ES5 之前的环境中使用这种 **部分填补**也没有问题。

有些开发者采取严格得多的观点，也就是除非能够被 *完全*填补，否则没有函数应该被填补。因为 `Object.create(..)` 是可以部分填补的工具之一，所以这种较狭窄的观点会说，如果你需要在 ES5 之前的环境中使用 `Object.create(..)` 的任何功能，你应当使用自定义的工具，而不是填补，而且应当彻底远离使用 `Object.create` 这个名字。你可以定义自己的工具，比如：

```
function createAndLinkObject(o) {
    function F(){}
```

```
    F.prototype = o;
    return new F();
}

var anotherObject = {
    a: 2
};

var myObject = createAndLinkObject( anotherObject );

myObject.a; // 2
```

我不会分享这种严格的观点。我完全拥护如上面展示的 `Object.create(..)` 的常见部分填补，甚至在 ES5 之前的环境下在你的代码中使用它。我将选择权留给你。

链接作为候补?

也许这么想很吸引人：这些对象间的链接 *主要*是为了给“缺失”的属性和方法提供某种候补。虽然这是一个可观察到的结果，但是我不认为这是考虑 `[[Prototype]]` 的正确方法。

考虑下面的代码：

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.cool(); // "cool!"
```

得益于 `[[Prototype]]`，这段代码可以工作，但如果你这样写是为了 **万一** `myObject` 不能处理某些开发者可能会调用的属性/方法，而让 `anotherObject` 作为一个候补，你的软件大概会变得有点儿“魔性”并且更难于理解和维护。

这不是说候补在任何情况下都不是一个合适的设计模式，但它不是一个在 JS 中很常见的用法，所以如果你发现自己在这么做，那么你可能想要退一步并重新考虑它是否真的是合适且合理的设计。

注意：在 ES6 中，引入了一个称为 `Proxy`（代理）的高级功能，它可以提供某种“方法未找到”类型的行为。`Proxy` 超出了本书的范围，但会在以后的“*你不懂 JS*”系列书目中详细讲解。

这里不要错过一个重要的细节。

例如，你打算为一个开发者设计软件，如果即使在 myObject 上没有 cool() 方法时调用 myObject.cool() 也能工作，会在你的 API 设计上引入一些 “魔法” 气息，这可能会使未来维护你的软件的开发者很吃惊。

然而你可以在你的 API 设计上少用些 “魔法”，而仍然利用 [[Prototype]] 链接的力量。

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.doCool = function() {
    this.cool(); // internal delegation!
};

myObject.doCool(); // "cool!"
```

这里，我们调用 myObject.doCool()，它是一个 *实际存在于* myObject 上的方法，这使我们的 API 设计更清晰（没那么 “魔性”）。*在它内部*，我们的实现依照 **委托设计模式**（见第六章），利用 [[Prototype]] 委托到 anotherObject.cool()。

换句话说，如果委托是一个内部实现细节，而非在你的 API 结构设计中简单地暴露出来，那么它将倾向于减少意外/困惑。我们会在下一章中详细解释 **委托**。

复习

当试图在一个对象上进行属性访问，而对象又没有该属性时，对象内部的 [[Prototype]] 链接定义了 [[Get]] 操作（见第三章）下一步应当到哪里寻找它。这种对象到对象的串行链接定义了对象的 “原形链”（和嵌套的作用域链有些相似），在解析属性时发挥作用。

所有普通的对象用内建的 Object.prototype 作为原形链的顶端(就像作用域查询的顶端是全局作用域),如果属性没能在链条的前面任何地方找到,属性解析就会在这里停止。toString()，valueOf()，和其他几种共同工具都存在于这个 Object.prototype 对象上，这解释了语言中所有的对象是如何能够访问他们的。

使两个对象相互链接在一起的最常见的方法是将 new 关键字与函数调用一起使用，在它的四个步骤中（见第二章），就会建立一个新对象链接到另一个对象。

那个用 new 调用的函数有一个被随便地命名为 .prototype 的属性，这个属性所引用的对象恰好就是这个新对象链接到的 “另一个对象”。带有 new 的函数调用通常被称为 “构造器”，尽管实际上它们并没有像传统的面向类语言那样初始化一个类。

虽然这些 JavaScript 机制看起来和传统面向类语言的 “初始化类” 和 “类继承” 类似，而在 JavaScript 中的关键区别是，没有拷贝发生。取而代之的是对象最终通过 `[[Prototype]]` 链链接在一起。

由于各种原因，不光是前面提到的术语，“继承”（和 “原型继承”）与所有其他的 OO 用语，在考虑 JavaScript 实际如何工作时都没有道理。

相反，“委托” 是一个更确切的术语，因为这些关系不是 *拷贝*而是委托 **链接**。

第六章: 行为委托

在第五章中，我们详细地讨论了 `[[Prototype]]` 机制，和 *为什么*对于描述 “类” 或 “继承” 来说它是那么使人糊涂和不合适。我们一路跋涉，不仅涉及了相当繁冗的语法（使代码凌乱的 `.prototype`），还有各种陷阱（比如使人吃惊的 `.constructor` 解析和难看的假想多态语法）。我们探索了许多人试图用抹平这些粗糙的区域而使用的各种 “mixin” 方法。

这时一个常见的反应是，想知道为什么这些看起来如此简单的事情这么复杂。现在我们已经拉开帷幕看到了它是多么麻烦，这并不奇怪：大多数 JS 开发者从不探究得这么深，而将这一团糟交给一个 “类” 包去帮他们处理。

我希望到现在你不会甘心于敷衍了事并把这样的细节丢给一个 “黑盒” 库。现在我们来深入讲解我们 *如何与应当如何*以一种比类造成的困惑 **简单得多而且更直接的方式**来考虑 JS 中对象的 `[[Prototype]]` 机制。

简单地复习一下第五章的结论，`[[Prototype]]` 机制是一种存在于一个对象上的内部链接，它指向一个其他对象。

当一个属性/方法引用在一个对象上发生，而这样的属性/方法又不存在时，这个链接就会被使用。在这种情况下，`[[Prototype]]` 链接告诉引擎去那个被链接的对象上寻找该属性/方法。接下来，如果那个对象也不能满足查询，就沿着它的 `[[Prototype]]` 查询，如此继续。这种对象间的一系列链接构成了所谓的 “原形链”。

换句话说，对于我们能在 JavaScript 中利用的功能的实际机制来说，其重要的实质 **全部在于被连接到其他对象的对象**。

这个观点是理解本章其余部分的动机和方法的重要基础!

迈向面向委托的设计

为了将我们的思想恰当地集中在如何用最直截了当的方法使用 `[[Prototype]]`，我们必须认识到它代表一种根本上与类不同的设计模式（见第四章）。

注意 *某些*面向类的设计依然是很有效的，所以不要扔掉你知道的每一件事（扔掉大多数就行了!）。比如，*封装*就十分强大，而且与委托是兼容的（虽然不那么常见）。

我们需要试着将我们的思维从类/继承的设计模式转变为行为代理设计模式。如果你已经使用你在教育/工作生涯中思考类的方式做了大多数或所有的编程工作，这可能感觉不舒服或不自然。你可能需要尝试这种思维过程好几次，才能适应这种非常不同的思考方式。

我将首先带你进行一些理论练习，之后我们会一对一地看一些更实际的例子来为你自己的代码提供实践环境。

类理论

比方说我们有几个相似的任务（“XYZ”，“ABC”，等）需要在我们的软件中建模。

使用类，你设计这个场景的方式是：定义一个泛化的父类（基类）比如 `Task`，为所有的“同类”任务定义共享的行为。然后，你定义子类 `XYZ` 和 `ABC`，它们都继承自 `Task`，每个都分别添加了特化的行为来处理各自的任务。

重要的是，类设计模式将鼓励你发挥继承的最大功效，当你在 `XYZ` 任务中覆盖 `Task` 的某些泛化方法的定义时，你将会想利用方法覆盖（和多态），也许会利用 `super` 来调用这个方法的泛化版本，为它添加更多的行为。**你很可能会找到几个这样的地方：可以“抽象”到父类中，并在子类中特化（覆盖）的一般化行为**。。

这是一些关于这个场景的假想代码：

```
class Task {
    id;

    // `Task()` 构造器
    Task(ID) { id = ID; }
    outputTask() { output( id ); }
}

class XYZ inherits Task {
    label;

    // `XYZ()` 构造器
    XYZ(ID,Label) { super( ID ); label = Label; }
    outputTask() { super(); output( label ); }
}

class ABC inherits Task {
    // ...
}
```

现在，你可以初始化一个或多个 `XYZ` 子类的 **拷贝**，并且使用这些实例来执行“XYZ”任务。这些实例已经 **同时拷贝**了泛化的 `Task` 定义的行为和具体的 `XYZ` 定义的行为。类似地，`ABC` 类的实例将拷贝 `Task` 的行为和具体的 `ABC` 的行为。在构建完成之后，你通常仅会与这些实例交互（而不是类），因为每个实例都拷贝了完成计划任务的所有行为。

委托理论

但是现在让我们试着用 行为委托代替 类来思考同样的问题。

你将首先定义一个称为 Task 的 **对象**（不是一个类，也不是一个大多数 JS 开发者想让你相信的 function），而且它将拥有具体的行为，这些行为包含各种任务可以使用的（读作：委托至!）工具方法。然后，对于每个任务（“XYZ”，“ABC”），你定义一个 **对象**来持有这个特定任务的数据/行为。你 **链接**你的特定任务对象到 Task 工具对象，允许它们在必要的时候可以委托到它。

基本上，你认为执行任务 “XYZ” 就是从两个兄弟/对等的对象（XYZ 和 Task）中请求行为来完成它。与其通过类的拷贝将它们组合在一起，我们可以将它们保持在分离的对象中，而且可以在需要的情况下允许 XYZ 对象 **委托到** Task。

这里是一些简单的代码，示意你如何实现它：

```
var Task = {
    setID: function(ID) { this.id = ID; },
    outputID: function() { console.log( this.id ); }
};

// 使 `XYZ` 委托到 `Task`
var XYZ = Object.create( Task );

XYZ.prepareTask = function(ID,Label) {
    this.setID( ID );
    this.label = Label;
};

XYZ.outputTaskDetails = function() {
    this.outputID();
    console.log( this.label );
};

// ABC = Object.create( Task );
// ABC ... = ...
```

在这段代码中，Task 和 XYZ 不是类（也不是函数），它们 **仅仅是对象**。XYZ 通过 Object.create() 创建，来 [[Prototype]] 委托到 Task 对象（见第五章）。

作为与面向类（也就是，OO —— 面向对象）的对比，我称这种风格的代码为 **“OLOO”**（objects-linked-to-other-objects（链接到其他对象的对象））。所有我们 真正关心的是，对象 XYZ 委托到对象 Task（对象 ABC 也一样）。

在 JavaScript 中，[[Prototype]] 机制将 **对象**链接到其他 **对象**。无论你多么想说服自己这不是真的，JavaScript 没有像 "类" 那样的抽象机制。这就像逆水行舟：你 可以做到，但你 选择了逆流而上，所以很明显地，**你会更困难地达到目的地**。

OLOO 风格的代码中有一些需要注意的不同：

1. 前一个类的例子中的 id 和 label 数据成员都是 XYZ 上的直接数据属性（它们都不在 Task 上）。一般来说，当 [[Prototype]] 委托引入时，**你想使状态保持在委托者上**（XYZ，ABC），不是在委托上（Task）。
2. 在类的设计模式中，我们故意在父类（Task）和子类（XYZ）上采用相同的命名 outputTask，以至于我们可以利用覆盖（多态）。在委托的行为中，我们反其道而行之：**我们尽一切可能避免在 [[Prototype]] 链的不同层级上给出相同的命名**（称为 "遮蔽" —— 见第五章），因为这些命名冲突会导致尴尬/脆弱的语法来消除引用的歧义（见第四章），而我们想避免它。这种设计模式不那么要求那些倾向于被覆盖的泛化的方法名，而是要求针对于每个对象的 具体行为类型给出更具描述性的方法名。**这实际上会产生更易于理解/维护的代码**，因为方法名（不仅在定义的位置，而是扩散到其他代码中）变得更加明白（代码即文档）。
3. this.setID(ID); 位于对象 XYZ 的一个方法内部，它首先在 XYZ 上查找 setID(..)，但因为它不能在 XYZ 上找到叫这个名称的方法，[[Prototype]] 委托意味着它可以沿着链接到 Task 来寻找 setID()，这样当然就找到了。另外，由于调用点的隐含 this 绑定规则（见第二章），当 setID() 运行时，即便方法是在 Task 上找到的，这个函数调用的 this 绑定依然是我们期望和想要的 XYZ。我们在代码稍后的 this.outputID() 中也看到了同样的事情。换句话说，我们可以使用存在于 Task 上的泛化工具与 XYZ 互动，因为 XYZ 可以委托至 Task。

行为委托意味着：在某个对象（XYZ）的属性或方法没能在这个对象（XYZ）上找到时，让这个对象（XYZ）为属性或方法引用提供一个委托（Task）。

这是一个 极其强大的设计模式，与父类和子类，继承，多态等有很大的不同。与其在你的思维中纵向地，从上面父类到下面子类地组织对象，你应当并列地，对等地考虑对象，而且对象间拥有方向性的委托链接。

注意：委托更适于作为内部实现的细节，而不是直接暴露在 API 接口的设计中。在上面的例子中，我们的 API 设计没必要有意地让开发者调用 XYZ.setID()（当然我们可以！）。我们以某种隐藏的方式将委托作为我们 API 的内部细节，即 XYZ.prepareTask(..) 委托到 Task.setID(..)。详细的内容，参照第五章的 "链接作为候补？" 中的讨论。

相互委托（不允许）

你不能在两个或多个对象间相互地委托（双向地）对方来创建一个 循环。如果你使 B 链接到 A，然后试着让 A 链接到 B，那么你将得到一个错误。

这样的事情不被允许有些可惜（不是非常令人惊讶，但稍稍有些恼人）。如果你制造一个在任意一方都不存在的属性/方法引用，你就会在 [[Prototype]] 上得到一个无限递归的循环。但

如果所有的引用都严格存在，那么 B 就可以委托至 A，或相反，而且它可以工作。这意味着你可以为了多种任务用这两个对象互相委托至对方。有一些情况这可能会有用。

但它不被允许是因为引擎的实现者发现，在设置时检查（并拒绝!）无限循环引用一次，要比每次你在一个对象上查询属性时都做相同检查的性能要高。

调试

我们将简单地讨论一个可能困扰开发者的微妙的细节。一般来说，JS 语言规范不会控制浏览器开发者工具如何向开发者表示指定的值/结构，所以每种浏览器/引擎都自由地按需要解释这个事情。因此，浏览器/工具 *不总是意见统一*。特别地，我们现在要考察的行为就是当前仅在 Chrome 的开发者工具中观察到的。

考虑这段传统的 “类构造器” 风格的 JS 代码，正如它将在 Chrome 开发者工具 *控制台* 中出现的：

```
function Foo() {}

var a1 = new Foo();

a1; // Foo {}
```

让我们看一下这个代码段的最后一行：对表达式 a1 进行求值的输出，打印 Foo {}。如果你在 FireFox 中试用同样的代码，你很可能会看到 Object {}。为什么会有不同？这些输出意味着什么？

Chrome 实质上在说 “{} 是一个由名为 ‘Foo’ 的函数创建的空对象”。Firefox 在说 “{} 是一个由 Object 普通构建的空对象”。这种微妙的区别是因为 Chrome 在像一个 *内部属性* 一样，动态跟踪执行创建的实际方法的名称，而其他浏览器不会跟踪这样的附加信息。

试图用 JavaScript 机制来解释它很吸引人：

```
function Foo() {}

var a1 = new Foo();

a1.constructor; // Foo(){}
a1.constructor.name; // "Foo"
```

那么，Chrome 就是通过简单地查看对象的 .Constructor.name 来输出 “Foo” 的？令人费解的是，答案既是 “是” 也是 “不”。

考虑下面的代码：

```
function Foo() {}
```

```
var a1 = new Foo();

Foo.prototype.constructor = function Gotcha(){};

a1.constructor; // Gotcha(){}
a1.constructor.name; // "Gotcha"

a1; // Foo {}
```

即便我们将 `a1.constructor.name` 合法地改变为其他的东西（“Gotcha”），Chrome 控制台依旧使用名称 “Foo”。

那么，说明前面问题（它使用 `.constructor.name` 吗?）的答案是 **不**，它一定在内部追踪其他的什么东西。

但是，且慢！让我们看看这种行为如何与 OLOO 风格的代码一起工作:

```
var Foo = {};

var a1 = Object.create( Foo );

a1; // Object {}

Object.defineProperty( Foo, "constructor", {
    enumerable: false,
    value: function Gotcha(){}
});

a1; // Gotcha {}
```

啊哈！**Gotcha**，Chrome 的控制台 **确实**寻找并且使用了 `.constructor.name`。实际上，就在写这本书的时候，这个行为被认定为是 Chrome 的一个 Bug，而且就在你读到这里的时候，它可能已经被修复了。所以你可能已经看到了被修改过的 `a1; // Object{}`。

这个 bug 暂且不论，Chrome 执行的（刚刚在代码段中展示的）“构造器名称” 内部追踪（目前仅用于调试输出的目的），是一个仅在 Chrome 内部存在的扩张行为，它已经超出了 JS 语言规范要求的范围。

如果你不使用 “构造器” 来制造你的对象，就像我们在本章的 OLOO 风格代码中不鼓励的那样，那么你将会得到一个 Chrome 不会为其追踪内部 “构造器名称” 的对象，所以这样的对象将正确地仅仅被输出 “Object {}”，意味着 “从 Object() 构建生成的对象”。

不要认为这代表一个 OLOO 风格代码的缺点。当你用 OLOO 编码而且用行为委托作为你的设计模式时，谁 “创建了”（也就是，哪个函数被和 `new` 一起调用了?）一些对象是一个无关的

细节。Chrome 特殊的内部 “构造器名称” 追踪仅仅在你完全接受 “类风格” 编码时才有用，而在你接受 OLOO 委托时是没有意义的。

思维模型比较

现在你至少在理论上可以看到 “类” 和 “委托” 设计模式的不同了，让我们看看这些设计模式在我们用来推导我们代码的思维模型上的含义。

我们将查看一些更加理论上的（“Foo”，“Bar”）代码，然后比较两种方法（OO vs. OLOO）的代码实现。第一段代码使用经典的（“原型的”）OO 风格：

```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call( this, who );
}
Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

父类 `Foo`，被子类 `Bar` 继承，之后 `Bar` 被初始化两次：`b1` 和 `b2`。我们得到的是 `b1` 委托至 `Bar.prototype`，`Bar.prototype` 委托至 `Foo.prototype`。这对你来说应当看起来十分熟悉。没有太具开拓性的东西发生。

现在，让我们使用 *OLOO* 风格的代码 **实现完全相同的功能**：

```
var Foo = {
    init: function(who) {
        this.me = who;
```

```
    },
    identify: function() {
        return "I am " + this.me;
    }
};

var Bar = Object.create( Foo );

Bar.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = Object.create( Bar );
b1.init( "b1" );
var b2 = Object.create( Bar );
b2.init( "b2" );

b1.speak();
b2.speak();
```

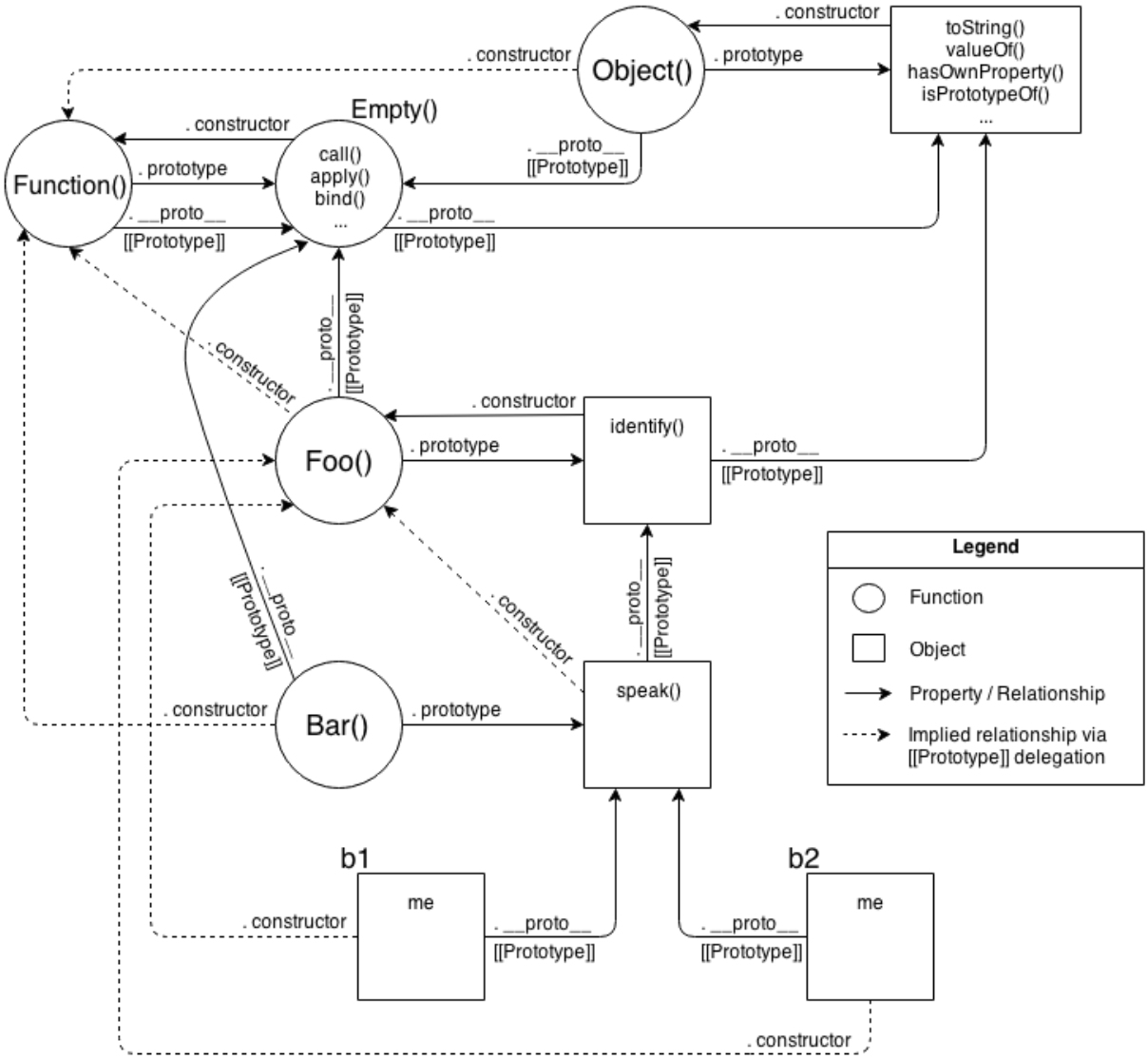
我们利用了完全相同的从 Bar 到 Foo 的 [[Prototype]] 委托，正如我们在前一个代码段中 b1，Bar.prototype，和 Foo.prototype 之间那样。**我们仍然有三个对象链接在一起**。

但重要的是，我们极大地简化了发生的 *所有其他事项*，因为我们现在仅仅建立了相互链接的 **对象**，而不需要所有其他讨厌且困惑的看起来像类（但动起来不像）的东西，还有构造器，原型和 new 调用。

问问你自己：如果我能用 OLOO 风格代码得到我用 “类” 风格代码得到的一样的东西，但 OLOO 更简单而且需要考虑的事情更少，**OLOO 不是更好吗**?

让我们讲解一下这两个代码段间涉及的思维模型。

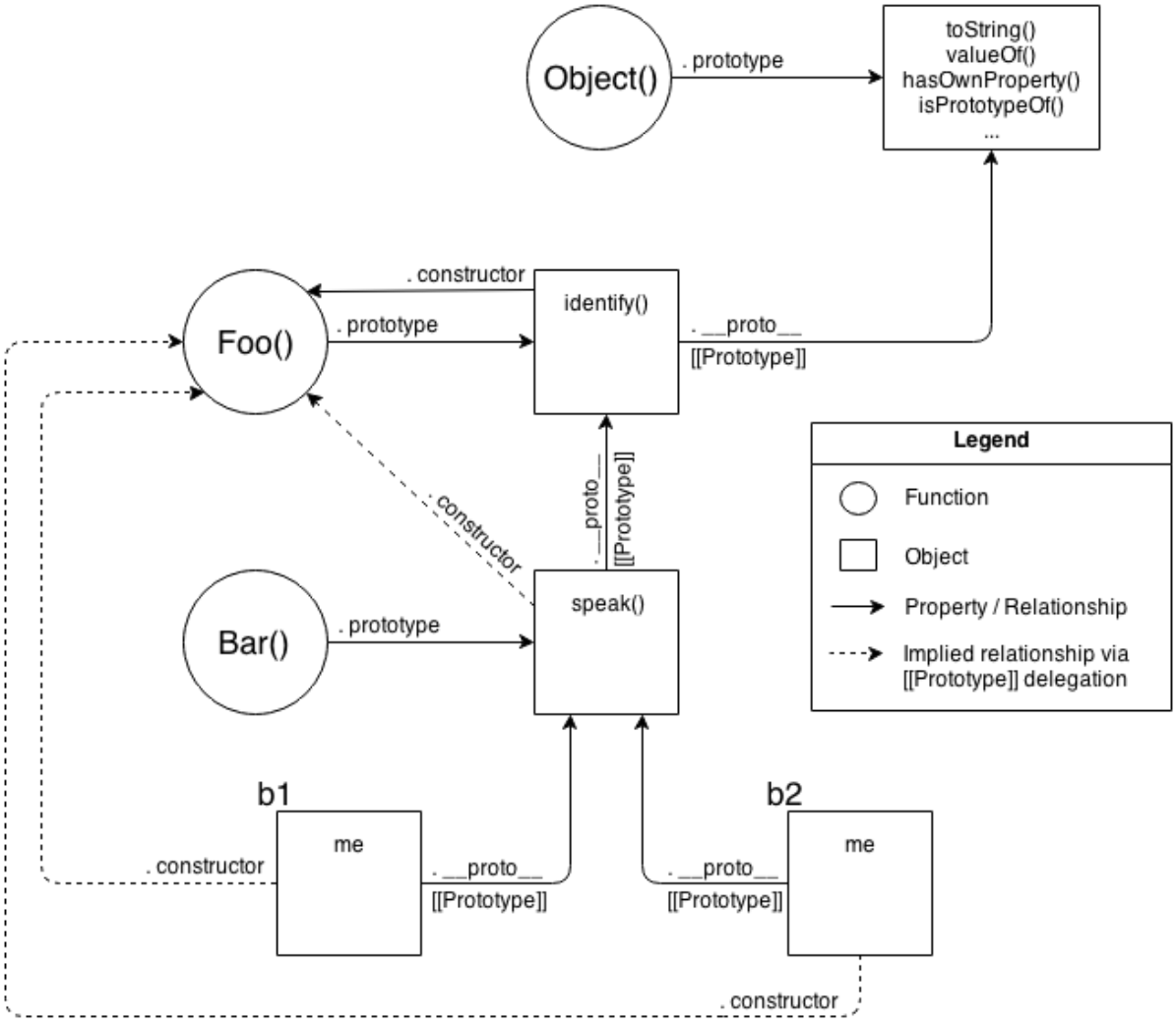
首先，类风格的代码段意味着这样的实体与它们的关系的思维模型：



实际上，这有点儿不公平/误导，因为它展示了许多额外的，你在 技术上一直不需要知道（虽然你 需要理解它）的细节。一个关键是，它是一系列十分复杂的关系。但另一个关键是：如果你花时间来沿着这些关系的箭头走，在 JS 的机制中 **有数量惊人的内部统一性**。

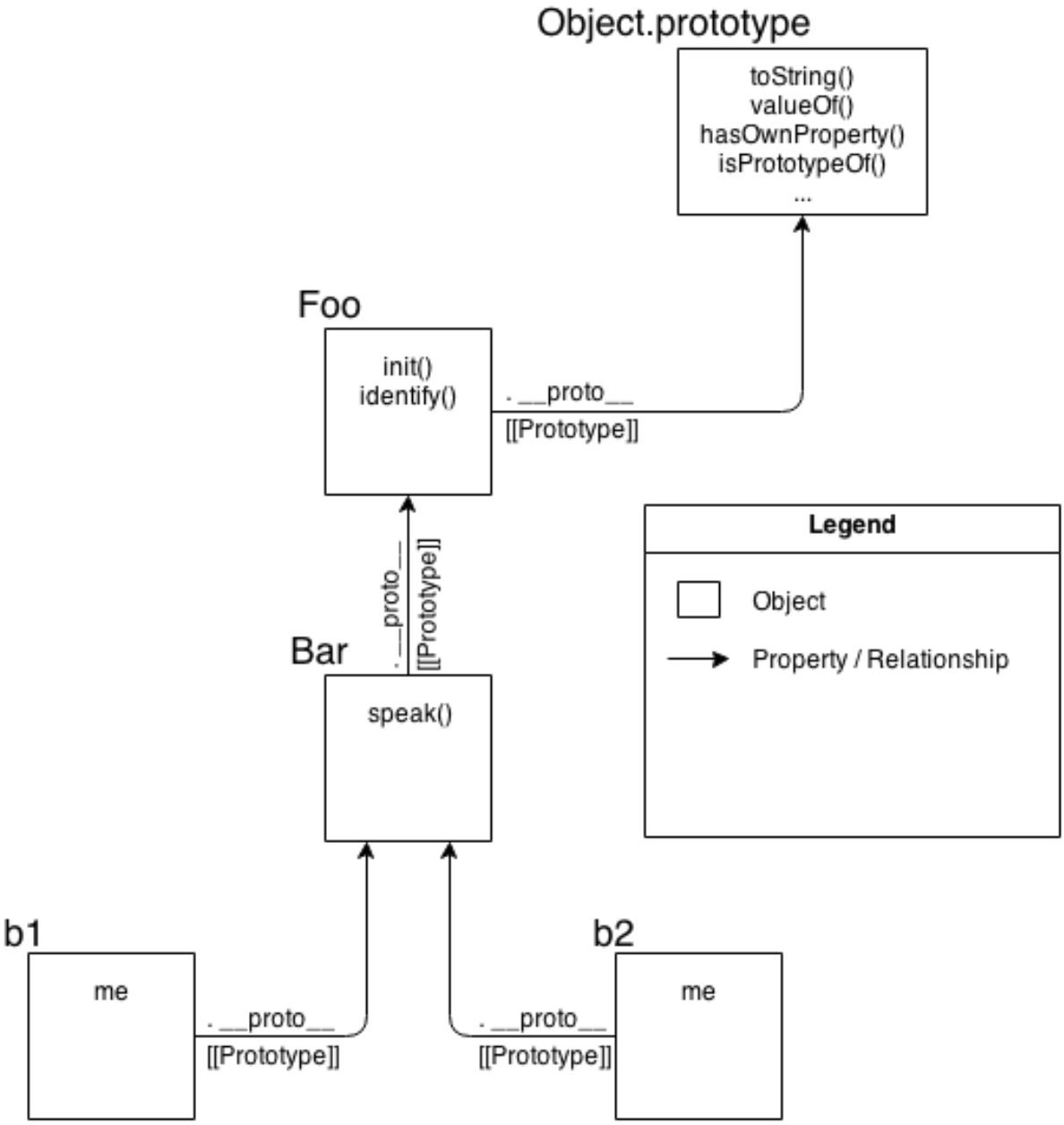
例如，JS 函数可以访问 `call(..)`，`apply(..)` 和 `bind(..)`（见第二章）的能力是因为函数本身是对象，而函数对象还拥有一个 `[[Prototype]]` 链接，链到 `Function.prototype` 对象，它定义了那些任何函数对象都可以委托到的默认方法。JS 可以做这些事情，你也能！

好了，现在让我们看一个这张图的 稍稍简化的版本，用它来进行比较稍微“公平”一点 —— 它仅展示了 相关的实体与关系。



仍然非常复杂，对吧? 虚线描绘了当你在 `Foo.prototype` 和 `Bar.prototype` 间建立“继承”时的隐含关系，而且还没有 修复 **丢失的** `.constructor` 属性引用（见第五章“复活构造器”）。即便将虚线去掉，每次你与对象链接打交道时，这个思维模型依然要变很多可怕的戏法。

现在，让我们看看 OLOO 风格代码的思维模型:



正如你比较它们所得到的，十分明显，OLOO 风格的代码 *需要关心的东西少太多了*，因为 OLOO 风格代码接受了 **事实**：我们唯一需要真正关心的事情是 **链接到其他对象的对象**。

所有其他 “类” 的烂设计用一种令人费解而且复杂的方式得到相同的结果。去掉那些东西，事情就变得简单得多（还不会失去任何功能）。

Classes vs. Objects

我们已经看到了各种理论的探索和 “类” 与 “行为委托” 的思维模型的比较。现在让我们来看看更具体的代码场景，来展示你如何实际应用这些想法。

我们将首先讲解一种在前端网页开发中的典型场景：建造 UI 部件（按钮，下拉列表等等）。

Widget “类”

因为你可能还是如此地习惯于 OO 设计模式，你很可能会立即这样考虑这个问题：一个父类（也许称为 `Widget`）拥有所有共通的基本部件行为，然后衍生的子类拥有具体的部件类型（比如 `Button`）。

注意：为了 DOM 和 CSS 的操作，我们将在这里使用 JQuery，这仅仅是因为对于我们现在的讨论，它不是一个我们真正关心的细节。这些代码中不关心你用哪个 JS 框架（JQuery，Dojo，YUI 等等）来解决如此无趣的问题。

让我们来看看，在没有任何 “类” 帮助库或语法的情况下，我们如何用经典风格的纯 JS 来实现 “类” 设计：

```
// 父类
function Widget(width,height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
}

Widget.prototype.render = function($where){
    if (this.$elem) {
        this.$elem.css( {
            width: this.width + "px",
            height: this.height + "px"
        } ).appendTo( $where );
    }
};

// 子类
function Button(width,height,label) {
    // "super" 构造器调用
    Widget.call( this, width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
}

// 使 `Button` “继承” `Widget`
Button.prototype = Object.create( Widget.prototype );
```

```
// 覆盖“继承来的” `render(..)`
Button.prototype.render = function($where) {
    // "super" 调用
    Widget.prototype.render.call( this, $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.prototype.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

OO 设计模式告诉我们要在父类中声明一个基础 `render(..)`，之后在我们的子类中覆盖它，但不是完全替代它，而是用按钮特定的行为增强这个基础功能。

注意 显式假想多态的丑态，`Widget.call` 和 `Widget.prototype.render.call` 引用是为了伪装从子“类”方法得到“父类”基础方法支持的“super”调用。呃。

ES6 `class` 语法糖

我们会在附录 A 中讲解 ES6 的 `class` 语法糖，但是让我们演示一下我们如何用 `class` 来实现相同的代码。

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
```

```
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

毋庸置疑，通过使用 ES6 的 `class`，许多前面经典方法中难看的语法被改善了。`super(..)` 的存在看起来非常适宜（但当你深入挖掘它时，不全是好事!）。

除了语法上的改进，**这些都不是 真正的类**，因为它们仍然工作在 `[[Prototype]]` 机制之上。它们依然会受到思维模型不匹配的拖累，就像我们在第四，五章中，和直到现在探索的那样。附录 A 将会详细讲解 ES6 `class` 语法和它的含义。我们将会看到为什么解决语法上的小问题不会实质上解决我们在 JS 中的类的困惑，虽然它做出了勇敢的努力假装解决了问题!

无论你是使用经典的原型语法还是新的 ES6 语法糖，你依然选择了使用 “类” 来对问题（UI 部件）进行建模。正如我们前面几章试着展示的，在 JavaScript 中做这个选择会带给你额外的

头疼和思维上的弯路。

委托部件对象

这是我们更简单的 Widget/Button 例子，使用了 **OLOO 风格委托**:

```
var Widget = {
    init: function(width,height){
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    },
    insert: function($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
};

var Button = Object.create( Widget );

Button.setup = function(width,height,label){
    // delegated call
    this.init( width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
};
Button.build = function($where) {
    // delegated call
    this.insert( $where );
    this.$elem.click( this.onClick.bind( this ) );
};
Button.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};
```

```
$( document ).ready( function(){
    var $body = $( document.body );

    var btn1 = Object.create( Button );
    btn1.setup( 125, 30, "Hello" );

    var btn2 = Object.create( Button );
    btn2.setup( 150, 40, "World" );

    btn1.build( $body );
    btn2.build( $body );
} );
```

使用这种 OLOO 风格的方法，我们不认为 `Widget` 是一个父类而 `Button` 是一个子类，`Widget` **只是一个对象**和某种具体类型的部件也许想要代理到的工具的集合，而且 `Button` **也只是一个独立的对象**（当然，带有委托至 `Widget` 的链接!）。

从设计模式的角度来看，我们 **没有**像类的方法建议的那样，在两个对象中共享相同的 `render(..)` 方法名称，而是选择了更能描述每个特定任务的不同的名称。同样的原因，*初始化*方法被分别称为 `init(..)` 和 `setup(..)`。

不仅委托设计模式建议使用不同而且更具描述性的名称，而且在 OLOO 中这样做会避免难看的显式假想多态调用，正如你可以通过简单，相对的 `this.init(..)` 和 `this.insert(..)` 委托调用看到的。

语法上，我们也没有任何构造器，`.prototype` 或者 `new` 出现，它们事实上是不必要的设计。

现在，如果你再细心考察一下，你可能会注意到之前仅有一个调用（`var btn1 = new Button(..)`），而现在有了两个（`var btn1 = Object.create(Button)` 和 `btn1.setup(..)`）。这猛地看起来像是一个缺点（代码变多了）。

然而，即便是这样的事情，和经典原型风格比起来也是 **OLOO 风格代码的优点**。为什么?

用类的构造器，你“强制”（不完全是这样，但是被强烈建议）构建和初始化在同一个步骤中进行。然而，有许多种情况，能够将这两步分开做（就像你在 OLOO 中做的）更灵活。

举个例子，我们假定你在程序的最开始，在一个池中创建所有的实例，但你等到在它们被从池中找出并使用之前再用指定的设置初始化它们。我们的例子中，这两个调用紧挨在一起，当然它们也可以按需要发生在非常不同的时间和代码中非常不同的部分。

OLOO 对关注点分离原则有 *更好*的支持，也就是创建和初始化没有必要合并在同一个操作中。

更简单的设计

OLOO 除了提供表面上更简单（而且更灵活!）的代码之外，行为委托作为一个模式实际上会带来更简单的代码架构。让我们讲解最后一个例子来说明 OLOO 是如何简化你的整体设计的。

这个场景中我们将讲解两个控制器对象，一个用来处理网页的登录 form（表单），另一个实际处理服务器的认证（通信）。

我们需要帮助工具来进行与服务器的 Ajax 通信。我们将使用 JQuery（虽然其他的框架都可以），因为它不仅为我们处理 Ajax，而且还返回一个类似 Promise 的应答，这样我们就可以在代码中使用 `.then(..)` 来监听这个应答。

注意：我们不会再这里讲到 Promise，但我们会在以后的 *你不懂 JS* 系列中讲到。

根据典型的类的设计模式，我们在一个叫做 `Controller` 的类中将任务分解为基本功能，之后我们会衍生出两个子类，`LoginController` 和 `AuthController`，它们都继承自 `Controller` 而且特化某些基本行为。

```
// 父类
function Controller() {
    this.errors = [];
}
Controller.prototype.showDialog = function(title,msg) {
    // 在对话框中给用户显示标题和消息
};
Controller.prototype.success = function(msg) {
    this.showDialog( "Success", msg );
};
Controller.prototype.failure = function(err) {
    this.errors.push( err );
    this.showDialog( "Error", err );
};

// 子类
function LoginController() {
    Controller.call( this );
}
// 将子类链接到父类
LoginController.prototype = Object.create( Controller.prototype );
LoginController.prototype.getUser = function() {
    return document.getElementById( "login_username" ).value;
};
```

```
LoginController.prototype.getPassword = function() {
    return document.getElementById( "login_password" ).value;
};
LoginController.prototype.validateEntry = function(user,pw) {
    user = user || this.getUser();
    pw = pw || this.getPassword();

    if (!(user && pw)) {
        return this.failure( "Please enter a username & password!" );
    }
    else if (pw.length < 5) {
        return this.failure( "Password must be 5+ characters!" );
    }

    // 到这里了? 输入合法!
    return true;
};
// 覆盖来扩展基本的 `failure()`
LoginController.prototype.failure = function(err) {
    // "super" 调用
    Controller.prototype.failure.call( this, "Login invalid: " + err );
};

// 子类
function AuthController(login) {
    Controller.call( this );
    // 除了继承外, 我们还需要合成
    this.login = login;
}
// 将子类链接到父类
AuthController.prototype = Object.create( Controller.prototype );
AuthController.prototype.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};
AuthController.prototype.checkAuth = function() {
```

```
    var user = this.login.getUser();
    var pw = this.login.getPassword();

    if (this.login.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
            pw: pw
        } )
        .then( this.success.bind( this ) )
        .fail( this.failure.bind( this ) );
    }
};
// 覆盖以扩展基本的 `success()`
AuthController.prototype.success = function() {
    // "super" 调用
    Controller.prototype.success.call( this, "Authenticated!" );
};
// 覆盖以扩展基本的 `failure()`
AuthController.prototype.failure = function(err) {
    // "super" 调用
    Controller.prototype.failure.call( this, "Auth Failed: " + err );
};

var auth = new AuthController(
    // 除了继承，我们还需要合成
    new LoginController()
);
auth.checkAuth();
```

我们有所有控制器分享的基本行为,它们是 `success(..)`,`failure(..)` 和 `showDialog(..)`。我们的子类 `LoginController` 和 `AuthController` 覆盖了 `failure(..)` 和 `success(..)` 来增强基本类的行为。还要注意的是，`AuthController` 需要一个 `LoginController` 实例来与登录 form 互动，所以它变成了一个数据属性成员。

另外一件要提的事情是，我们选择一些 合成散布在继承的顶端。`AuthController` 需要知道 `LoginController`，所以我们初始化它（`new LoginController()`），并用一个称为 `this.login` 的类属性成员来引用它，这样 `AuthController` 才可以调用 `LoginController` 上的行为。

注意：这里可能会存在一丝冲动，就是使 `AuthController` 继承 `LoginController`，或者反过来，这样的话我们就会通过继承链得到 虚拟合成。但是这是一个非常清晰的例子，表明对这个问题来讲，将类继承作为模型有什么问题，因为 `AuthController` 和 `LoginController` 都

不特化对方的行为，所以它们之间的继承没有太大的意义，除非类是你唯一的设计模式。与此相反的是，我们在一些简单的合成中分层，然后它们就可以合作了，同时它俩都享有继承自父类 Controller 的好处。

如果你熟悉面向类（OO）的设计，这都应该看起来十分熟悉和自然。

去类化

但是，我们真的需要用一个父类，两个子类，和一些合成来对这个问题建立模型吗？有办法利用 OLOO 风格的行为委托得到 *简单得多*的设计吗？**有的！**

```
var LoginController = {
    errors: [],
    getUser: function() {
        return document.getElementById( "login_username" ).value;
    },
    getPassword: function() {
        return document.getElementById( "login_password" ).value;
    },
    validateEntry: function(user,pw) {
        user = user || this.getUser();
        pw = pw || this.getPassword();

        if (!(user && pw)) {
            return this.failure( "Please enter a username & password!" );
        }
        else if (pw.length < 5) {
            return this.failure( "Password must be 5+ characters!" );
        }

        // 到这里了？输入合法！
        return true;
    },
    showDialog: function(title,msg) {
        // 在对话框中向用于展示成功消息
    },
    failure: function(err) {
        this.errors.push( err );
        this.showDialog( "Error", "Login invalid: " + err );
    }
```

```
};

// 链接 `AuthController`委托到 `LoginController`
var AuthController = Object.create( LoginController );

AuthController.errors = [];
AuthController.checkAuth = function() {
    var user = this.getUser();
    var pw = this.getPassword();

    if (this.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
            pw: pw
        } )
        .then( this.accepted.bind( this ) )
        .fail( this.rejected.bind( this ) );
    }
};
AuthController.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};
AuthController.accepted = function() {
    this.showDialog( "Success", "Authenticated!" )
};
AuthController.rejected = function(err) {
    this.failure( "Auth Failed: " + err );
};
```

因为 AuthController 只是一个对象（LoginController 也是），我们不需要初始化（比如 new AuthController()）就能执行我们的任务。所有我们要做的是：

```
AuthController.checkAuth();
```

当然，通过 OLOO，如果你确实需要在委托链上创建一个或多个附加的对象时也很容易，而且仍然不需要任何像类实例化那样的东西：

```
var controller1 = Object.create( AuthController );
var controller2 = Object.create( AuthController );
```

使用行为委托，AuthController 和 LoginController **仅仅是对象**，互相是 *水平*对等的，而且没有被安排或关联成面向类中的父与子。我们有些随意地选择让 AuthController 委托至 LoginController —— 相反方向的委托也同样是有效的。

第二个代码段的主要要点是，我们只拥有两个实体(LoginController and AuthController)，而 **不是之前的三个**。

我们不需要一个基本的 Controller 类来在两个子类间 "分享" 行为，因为委托是一种可以给我们所需功能的，足够强大的机制。同时，就像之前注意的，我们也不需要实例化我们的对象来使它们工作，因为这里没有类，**只有对象自身**。另外，这里不需要 *合成*作为委托来给两个对象 *差异化*地合作的能力。

最后，由于没有让名称 success(..) 和 failure(..) 在两个对象上相同，我们避开了面向类的设计的多态陷阱：它将会需要难看的显式假想多态。相反，我们在 AuthController 上称它们为 accepted() 和 rejected(..) —— 对于它们的具体任务来说，稍稍更具描述性的名称。

底线：我们最终得到了相同的结果，但是用了（显著的）更简单的设计。这就是 OLOO 风格代码和 *行为委托*设计模式的力量。

更好的语法

一个使 ES6 class 看似如此诱人的更好的东西是（见附录 A 来了解为什么要避免它!），声明类方法的速记语法：

```
class Foo {
    methodName() { /* .. */ }
}
```

我们从声明中扔掉了单词 function，这使所有的 JS 开发者欢呼!

你可能已经注意到，而且为此感到沮丧：上面推荐的 OLOO 语法出现了许多 function，这看起来像是对 OLOO 简化目标的诋毁。**但它不必是!**

在 ES6 中，我们可以在任何字面对象中使用 *简约方法声明*，所以一个 OLOO 风格的对象可以用这种方式声明（与 class 语法中相同的语法糖）：

```
var LoginController = {
    errors: [],
    getUser() { // 看，没有 `function`!
        // ...
    },
    getPassword() {
```

```
        // ...
    }
    // ...
};
```

唯一的区别是字面对象的元素间依然需要 ，逗号分隔符，而 class 语法不必如此。这是在整件事情上很小的让步。

还有，在 ES6 中，一个你使用的更笨重的语法（比如 AuthController 的定义中）：你一个一个地给属性赋值而不使用字面对象，可以改写为使用字面对象（于是你可以使用简约方法），而且你可以使用 Object.setPrototypeOf(..) 来修改对象的 [[Prototype]]，像这样：

```
// 使用更好的字面对象语法 w/ 简约方法!
var AuthController = {
    errors: [],
    checkAuth() {
        // ...
    },
    server(url,data) {
        // ...
    }
    // ...
};

// 现在，链接 `AuthController` 委托至 `LoginController`
Object.setPrototypeOf( AuthController, LoginController );
```

ES6 中的 OLOO 风格，与简明方法一起，变得比它以前 **友好得多**（即使在以前，它也比经典的原型风格代码简单好看的多）。**你不必非得选用类**（复杂性）来得到干净漂亮的对象语法!

没有词法

简约方法确实有一个缺点，一个重要的细节。考虑这段代码：

```
var Foo = {
    bar() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

这是去掉语法糖后，这段代码将如何工作：

```
var Foo = {
    bar: function() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

看到区别了? bar() 的速记法变成了一个附着在 bar 属性上的 *匿名函数表达式* (function()..) ，因为函数对象本身没有名称标识符。和拥有词法名称标识符 baz，附着在 .baz 属性上的手动指定的 *命名函数表达式* (function baz()..) 做个比较。

那又怎么样? 在 *"你不懂 JS"* 系列的 *"作用域与闭包"* 这本书中，我们详细讲解了 *匿名函数表达式*的三个主要缺点。我们简单地重复一下它们，以便于我们和简明方法相比较。

一个匿名函数缺少 name 标识符:

1. 使调试时的栈追踪变得困难
2. 使自引用（递归，事件绑定等）变得困难
3. 使代码（稍稍）变得难于理解

第一和第三条不适用于简明方法。

虽然去掉语法糖使用 *匿名函数表达式*一般会使栈追踪中没有 name。简明方法在语言规范中被要求去设置相应的函数对象内部的 name 属性，所以栈追踪应当可以使用它（这是依赖于具体实现的，所以不能保证）。

不幸的是，第二条 **仍然是简明方法的一个缺陷**。它们不会有词法标识符用来自引用。考虑:

```
var Foo = {
    bar: function(x) {
        if (x < 10) {
            return Foo.bar( x * 2 );
        }
        return x;
    },
    baz: function baz(x) {
        if (x < 10) {
            return baz( x * 2 );
        }
        return x;
    }
};
```

在这个例子中上面的手动 Foo.bar(x*2) 引用就足够了，但是在许多情况下，一个函数不一定能够这样做，比如使用 this 绑定，函数在委托中被分享到不同的对象，等等。你将会想要使用一个真正的自引用，而函数对象的 name 标识符是实现的最佳方式。

只要小心简明方法的这个注意点，而且如果当你陷入缺少自引用的问题时，**仅仅为这个声明**放弃简明方法语法，取代以手动的 *命名函数表达式*声明形式：`baz: function baz(){..}`。

自省

如果你花了很长时间在面向类的编程方式（不管是 JS 还是其他的语言）上，你可能会对 *类型自省*很熟悉：自省一个实例来找出它是什么 *种类*的对象。在类的实例上进行 *类型自省*的主要目的是根据 *对象是如何创建的*来推断它的结构/能力。

考虑这段代码，它使用 `instanceof`（见第五章）来自省一个对象 `a1` 来推断它的能力：

```
function Foo() {
    // ...
}
Foo.prototype.something = function(){
    // ...
}

var a1 = new Foo();

// 稍后

if (a1 instanceof Foo) {
    a1.something();
}
```

因为 `Foo.prototype`（不是 `Foo`!）在 `a1` 的 `[[Prototype]]` 链上（见第五章），`instanceof` 操作符（使人困惑地）假装告诉我们 `a1` 是一个 `Foo` “类” 的实例。有了这个知识，我们假定 `a1` 有 `Foo` “类” 中描述的能力。

当然，这里没有 `Foo` 类，只有一个普通的函数 `Foo`，它恰好拥有一个引用指向一个随意的对象（`Foo.prototype`），而 `a1` 恰好委托链接至这个对象。通过它的语法，`instanceof` 假装检查了 `a1` 和 `Foo` 之间的关系，但它实际上告诉我们的是 `a1` 和 `Foo.prototype`（这个随意被引用的对象）是否有关联。

`instanceof` 在语义上的混乱（和间接）意味着，要使用以 `instanceof` 为基础的自省来查询对象 `a1` 是否与讨论中的对象有关联，你 *不得不*拥有一个持有对这个对象引用的函数 —— 你不能直接查询这两个对象是否有关联。

回想本章前面的抽象 `Foo` / `Bar` / `b1` 例子，我们在这里缩写一下：

```
function Foo() { /* .. */ }
Foo.prototype...
```

```
function Bar() { /* .. */ }
Bar.prototype = Object.create( Foo.prototype );

var b1 = new Bar( "b1" );
```

为了在这个例子中的实体上进行 类型自省，使用 instanceof 和 .prototype 语义，这里有各种你可能需要实施的检查：

```
// `Foo` 和 `Bar` 互相的联系
Bar.prototype instanceof Foo; // true
Object.getPrototypeOf( Bar.prototype ) === Foo.prototype; // true
Foo.prototype.isPrototypeOf( Bar.prototype ); // true

// `b1` 与 `Foo` 和 `Bar` 的联系
b1 instanceof Foo; // true
b1 instanceof Bar; // true
Object.getPrototypeOf( b1 ) === Bar.prototype; // true
Foo.prototype.isPrototypeOf( b1 ); // true
Bar.prototype.isPrototypeOf( b1 ); // true
```

可以说，其中有些烂透了。举个例子，直觉上（用类）你可能想说这样的东西 Bar instanceof Foo（因为很容易混淆“实例”的意义认为它包含“继承”），但在 JS 中这不是一个合理的比较。你不得不说 Bar.prototype instanceof Foo。

另一个常见，但也许健壮性更差的 类型自省模式叫“duck typing（鸭子类型）”，比起 instanceof 来许多开发者都倾向于它。这个术语源自一则谚语，“如果它看起来像鸭子，叫起来像鸭子，那么它一定是一只鸭子”。

例如：

```
if (a1.something) {
    a1.something();
}
```

与其检查 a1 和一个持有可委托的 something() 函数的对象的关系，我们假设 a1.something 测试通过意味着 a1 有能力调用 .something()（不管是直接在 a1 上直接找到方法，还是委托至其他对象）。就其本身而言，这种假设没什么风险。

但是“鸭子类型”常常被扩展用于 **除了被测试关于对象能力以外的其他假设**，这当然会在测试中引入更多风险（比如脆弱的设计）。

“鸭子类型”的一个值得注意的例子来自于 ES6 的 Promises（就是我们前面解释过，将不再本书内涵盖的内容）。

由于种种原因，需要判定任意一个对象引用是否 *是一个 Promise*，但测试是通过检查对象是否恰好有 `then()` 函数出现在它上面来完成的。换句话说，**如果任何对象**恰好有一个 `then()` 方法，ES6 的 Promises 将会无条件地假设这个对象 **是 "thenable"** 的，而且因此会期望它按照所有的 Promises 标准行为那样一致地动作。

如果你有任何非 Promise 对象，而却不管因为什么它恰好拥有 `then()` 方法，你会被强烈建议使它远离 ES6 的 Promise 机制，来避免破坏这种假设。

这个例子清楚地展现了 "鸭子类型" 的风险。你应当仅在可控的条件下，保守地使用这种方式。

再次将我们的注意力转向本章中出现的 OLOO 风格的代码，*类型自省*变得清晰多了。让我们回想（并缩写）本章的 `Foo` / `Bar` / `b1` 的 OLOO 示例：

```
var Foo = { /* .. */ };

var Bar = Object.create( Foo );
Bar...

var b1 = Object.create( Bar );
```

使用这种 OLOO 方式，我们所拥有的一切都是通过 `[[Prototype]]` 委托关联起来的普通对象，这是我们可能会用到的大幅简化后的 *类型自省*：

```
// `Foo` 和 `Bar` 互相的联系
Foo.isPrototypeOf( Bar ); // true
Object.getPrototypeOf( Bar ) === Foo; // true

// `b1` 与 `Foo` 和 `Bar` 的联系
Foo.isPrototypeOf( b1 ); // true
Bar.isPrototypeOf( b1 ); // true
Object.getPrototypeOf( b1 ) === Bar; // true
```

我们不再使用 `instanceof`，因为它令人迷惑地假装与类有关系。现在，我们只需要（非正式地）问这个问题，"你是我的 一个原型吗？"。不再需要用 `Foo.prototype` 或者痛苦冗长的 `Foo.prototype.isPrototypeOf(..)` 来间接地查询了。

我想可以说这些检查比起前面一组自省检查，极大地减少了复杂性/混乱。**又一次，我们看到了在 JavaScript 中 OLOO 要比类风格的编码简单（但有着相同的力量）。**

复习

在你的软件体系结构中，类和继承是你可以 *选用*或 *不选用*的设计模式。多数开发者理所当然地认为类是组织代码的唯一（正确的）方法，但我们在这里看到了另一种不太常被提到的，但实际上十分强大的设计模式：**行为委托**。

行为委托意味着对象彼此是对等的，在它们自己当中相互委托，而不是父类与子类的关系。JavaScript 的 [[Prototype]] 机制的设计本质，就是行为委托机制。这意味着我们可以选择挣扎着在 JS 上实现类机制，也可以欣然接受 [[Prototype]] 作为委托机制的本性。

当你仅用对象设计代码时，它不仅能简化你使用的语法，而且它还能实际上引领更简单的代码结构设计。

OLOO（链接到其他对象的对像）是一种没有类的抽象，而直接创建和关联对象的代码风格。OLOO 十分自然地实现了基于 [[Prototype]] 的行为委托。

附录 A: ES6 class

如果说本书后半部分（第四到六章）有什么关键信息，那就是类是一种代码的可选设计模式（不是必要的），而且用像 JavaScript 这样的 [[Prototype]] 语言来实现它总是很尴尬。

虽然这种尴尬很大一部分关于语法，但 *不仅限于此*。第四和第五章审视了相当多的难看语法，从使代码杂乱的 .prototype 引用的繁冗，到 *显式假想多态*：当你在链条的不同层级上给方法相同的命名以试图实现从低层方法到高层方法的多态引用。.constructor 被错误地解释为“被 XX 构建”，这成为了一个不可靠的定义，也成为了另一个难看的语法。

但关于类的设计的问题要深刻多了。第四章指出在传统的面向类语言中，类实际上发生了从父类向子类，由子类向实例的 *拷贝*动作，而在 [[Prototype]] 中，动作 **不是**一个拷贝，而是相反 —— 一个委托链接。

OLOO 风格和行为委托接受了 [[Prototype]]，而不是将它隐藏起来，当比较它们的简单性时，类在 JS 中的问题就凸显出来。

class

我们 *不必*再次争论这些问题。我在这里简单地重提这些问题仅仅是为了使它们在你的头脑里保持新鲜，以使我们将注意力转向 ES6 的 class 机制。我们将在这里展示它如何工作，并且看看 class 是否实质上解决了任何这些“类”的问题。

让我们重温第六章的 Widget/Button 例子：

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
```

```
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}
```

除了语法上 *看起来*更好，ES6 还解决了什么?

1. 不再有（某种意义上的，继续往下看!）指向 `.prototype` 的引用来弄乱代码。
2. `Button` 被声明为直接“继承自”(也就是 `extends`)`Widget`,而不是需要用 `Object.create(..)` 来替换 `.prototype` 链接的对象，或者用 `__proto__` 和 `Object.setPrototypeOf(..)` 来设置它。
3. `super(..)` 现在给了我们非常有用的 **相对多态**的能力，所以在链条上某一个层级上的任何方法，可以引用链条上相对上一层的同名方法。第四章中有一个关于构造器的奇怪现象：构造器不属于它们的类，而且因此与类没有联系。`super(..)` 含有一个对此问题的解决方法 —— `super()` 会在构造器内部想如你期望的那样工作。
4. `class` 字面语法对指定属性没有什么启发（仅对方法有）。这看起来限制了某些东西，但是绝大多数情况下期望一个属性（状态）存在于链条末端的“实例”以外的地方，这通常是一个错误和令人诧异（因为这个状态被隐含地在所有“实例”中“分享”）的。所以，也可以说 `class` 语法防止你出现错误。
5. `extends` 甚至允许你用非常自然的方式扩展内建的对象（子）类型，比如 `Array` 或者 `RegExp`。在没有 `class .. extends` 的情况下这样做一直以来是一个极端复杂而令人沮丧的任务，只有最熟练的框架作者曾经正确地解决过这个问题。现在，它是小菜一碟!

凭心而论，对大多数明显的（语法上的）问题，和经典的原型风格代码使人诧异的地方，这些确实是实质上的解决方案。

class 的坑

然而，它不全是优点。在 JS 中将 “类” 作为一种设计模式，仍然有一些深刻和非常令人烦恼的问题。

首先，class 语法可能会说服你 JS 在 ES6 中存在一个新的 “类” 机制。**但不是这样**。class 很大程度上仅仅是一个既存的 [[Prototype]]（委托）机制的语法糖！

这意味着 class 实际上不是像传统面向类的语言那样，在声明时静态地拷贝定义。如果你在 “父类” 上更改/替换了一个方法（有意或无意地），子 “类” 和/或实例将会受到 “影响”，因为它们在声明时没有得到一份拷贝，它们依然都使用那个基于 [[Prototype]] 的实时委托模型。

```
class C {
    constructor() {
        this.num = Math.random();
    }
    rand() {
        console.log( "Random: " + this.num );
    }
}

var c1 = new C();
c1.rand(); // "Random: 0.4324299..."

C.prototype.rand = function() {
    console.log( "Random: " + Math.round( this.num * 1000 ));
};

var c2 = new C();
c2.rand(); // "Random: 867"

c1.rand(); // "Random: 432" -- oops!!!
```

这种行为只有在 *你已经知道了*关于委托的性质，而不是期待从 “真的类” 中 *拷贝*时，才看起来合理。那么你要问自己的问题是，为什么你为了根本上就和类不同的东西选择 class 语法？

ES6 的 class 语法不是使观察和理解传统的类和委托对象间的不同 **变得更困难**了吗？

class 语法 *没有*提供声明类的属性成员的方法（仅对方法有）。所以如果你需要跟踪对象间分享的状态，那么你最终会回到难看的 .prototype 语法，像这样：

```
class C {
    constructor() {
        // 确保修改的是共享状态
        // 不是设置实例上的遮蔽属性
        C.prototype.count++;

        // 这里，`this.count` 通过委托如我们期望的那样工作
        console.log( "Hello: " + this.count );
    }
}

// 直接在原型对象上添加一个共享属性
C.prototype.count = 0;

var c1 = new C();
// Hello: 1

var c2 = new C();
// Hello: 2

c1.count === 2; // true
c1.count === c2.count; // true
```

这里最大的问题是，由于它将 .prototype 作为实现细节暴露（泄露！）出来，而背叛了 class 语法的初衷。

而且，我们还依然面临着那个令人诧异的陷阱：this.count++ 将会隐含地在 c1 和 c2 两个对象上创建一个分离的遮蔽属性 .count，而不是更新共享的状态。class 没有在这个问题上给我们什么安慰，除了（大概是）通过缺少语法支持来暗示你 根本就不应该这么做。

另外，无意地遮蔽依然是个灾难：

```
class C {
    constructor(id) {
        // 噢，一个坑，我们用实例上的属性值遮蔽了 `id()`方法
        this.id = id;
    }
    id() {
        console.log( "Id: " + this.id );
    }
}
```

```
var c1 = new C( "c1" );
c1.id(); // TypeError -- `c1.id` 现在是字符串"c1"
```

还有一些关于 super 如何工作的微妙问题。你可能会假设 super 将会以一种类似与 this 得到绑定的方式（间第二章）来被绑定，也就是 super 总是会绑定到当前方法在 [[Prototype]] 链中的位置的更高一层。

然而，因为性能问题（this 绑定已经很耗费性能了），super 不是动态绑定的。它在声明时，被有些 “静态地” 绑定。不是什么大事儿，对吧?

恩…… 可能是，可能不是。如果你像大多数 JS 开发者那样，开始把函数以各种不同的方式赋值给不同的（来自于 class 定义的）对象，你可能不会意识到在所有这些情况下，底层的 super 机制会不得不每次都重新绑定。

而且根据你每次赋值采取的语法方式不同，很有可能在某些情况下 super 不能被正确地绑定（至少不会像你期望的那样），所以你可能（在写作这里时，TC39 正在讨论这个问题）会不得不用 toMethod(..) 来手动绑定 super（有点儿像你不得不用 bind(..) 绑定 this —— 见第二章）。

你曾经可以给不同的对象赋予方法，来通过 隐含绑定规则（见第二章），自动地利用 this 的动态性。但对于使用 super 的方法，同样的事情很可能不会发生。

考虑这里 super 应当怎样动作（对 D 和 E）:

```
class P {
    foo() { console.log( "P.foo" ); }
}

class C extends P {
    foo() {
        super();
    }
}

var c1 = new C();
c1.foo(); // "P.foo"

var D = {
    foo: function() { console.log( "D.foo" ); }
};

var E = {
    foo: C.prototype.foo
```

```
};

// E 链接到 D 来进行委托
Object.setPrototypeOf( E, D );

E.foo(); // "P.foo"
```

如果你（十分合理地!）认为 super 将会在调用时自动绑定，你可能会期望 super() 将会自动地认识到 E 委托至 D，所以使用 super() 的 E.foo() 应当调用 D.foo()。

不是这样。由于实用主义的性能原因，super 不像 this 那样 *延迟绑定*（也就是动态绑定）。相反它从调用时 [[HomeObject]].[[Prototype]] 派生出来，而 [[HomeObject]] 是在声明时静态绑定的。

在这个特定的例子中，super() 依然解析为 P.foo()，因为方法的 [[HomeObject]] 仍然是 C 而且 C.[[Prototype]] 是 P。

可能会有方法手动地解决这样的陷阱。在这个场景中使用 toMethod(..) 来绑定/重绑定方法的 [[HomeObject]]（设置这个对象的 [[Prototype]] 一起!）似乎会管用：

```
var D = {
    foo: function() { console.log( "D.foo" ); }
};

// E 链接到 D 来进行委托
var E = Object.create( D );

// 手动绑定 `foo` 的 `[[HomeObject]]` 到
// `E`，因为 `E.[[Prototype]]` 是 `D`，所以
// `super()` 是 `D.foo()`
E.foo = C.prototype.foo.toMethod( E, "foo" );

E.foo(); // "D.foo"
```

注意：toMethod() 克隆这个方法，然后将它的第一个参数作为 homeObject（这就是为什么我们传入 E），第二个参数（可选）用来设置新方法的 name（保持 “foo” 不变）。

除了这种场景以外，是否还有其他的极端情况会使开发者们落入陷阱还有待观察。无论如何，你将不得不费心保持清醒：在哪里引擎自动为你确定 super，和在哪里你不得不手动处理它。**噢!**

静态优于动态?

但是关于 ES6 的最大问题是，所有这些种种陷阱意味着 `class` 有点儿将你带入一种语法，它看起来暗示着（像传统的类那样）一旦你声明一个 `class`，它是一个东西的静态定义（将来会实例化）。使你完全忘记了这个事实：`C` 是一个对象，一个你可以直接互动的具体的东西。

在传统面向类的语言中，你从不会在晚些时候调整类的定义，所以类设计模式不提供这样的能力。但是 JS 的 **一个最强大的部分**就是它 *是*动态的，而且任何对象的定义都是（除非你将它设定为不可变）不固定的可变的 *东西*。

`class` 看起来在暗示你不应该做这样的事情，通过强制你使用 `.prototype` 语法才能做到，或强制你考虑 `super` 的陷阱，等等。而且它对这种动态机制可能带来的一切陷阱 *几乎*不提供任何支持。

换句话说，`class` 好像在告诉你："动态太坏了，所以这可能不是一个好主意。这里有看似静态语法，把你的东西静态编码。"

关于 JavaScript 的评论是多么悲伤啊：**动态太难了，让我们假装成（但实际上不是!）静态吧**。

这些就是为什么 ES6 的 `class` 伪装成一个语法头痛症的解决方案，但是它实际上把水搅得更浑，而且更不容易对 JS 形成清晰简明的认识。

注意：如果你使用 `.bind(..)` 工具制作一个硬绑定函数（见第二章），那么这个函数是不能像普通函数那样用 ES6 的 `extend` 扩展的。

复习

`class` 在假装修复 JS 中的类/继承设计模式的问题上做的很好。但它实际上做的却正相反：**它隐藏了许多问题，而且引入了其他微妙而且危险的东西**。

`class` 为折磨了 JavaScript 语言将近二十年的 "类" 的困扰做出了新的贡献。在某些方面，它问的问题比它解决的多，而且在 `[[Prototype]]` 机制的优雅和简单之上，它整体上感觉像是一个非常不自然的匹配。

底线：如果 ES6 `class` 使稳健地利用 `[[Prototype]]` 变得困难，而且隐藏了 JS 对象机制最重要的性质 —— **对象间的实时委托链接** —— 我们不应该认为 `class` 产生的麻烦比它解决的更多，并且将它贬低为一种反模式吗?

我真的不能帮你回答这个问题。但我希望这本书已经在你从未经历过的深度上完全地探索了这个问题，而且已经给出了 *你自己回答这个问题*所需的信息。

附录 B: 鸣谢

为了这本书和整个系列的诞生，我有很多人要感谢。

首先，我必须感谢我的妻子 Christen Simpson，和我的两个孩子 Ethan 和 Emily，忍受着老爹总是在电脑上敲打。即使在没有写书时，我对 JavaScript 的痴迷也将我的眼睛粘在屏幕上

太久了。我从家庭那里借来的时间是这些书可以如此深入和完整地向你，读者，解释 JavaScript 的原因。我欠我的家庭一切。

我要感谢我在 O'Reilly 的编辑，他们是 Simon St.Laurent 和 Brian MacDonald，还有其他的编辑和市场员工。和他们一起工作很棒，而且在这种 "开源" 写作，编辑，和生产的实验期间提供了特别的通融。

感谢许多通过提供编辑意见和订正来参与使这部丛书变得更好的朋友们，他们是 Shelley Powers，Tim Ferro，Evan Borden，Forrest L. Norvell，Jennifer Davis，Jesse Harlin，和许多其他人。一个巨大感谢送给为本书作序的 Nick Berardi。

感谢社区中无数的朋友们，包括 TC39 协会的成员，他们和我们分享了那么多的知识，特别是以耐心和细节容忍我无休止的问题和探究。John-David Dalton，Juriy "kangax" Zaytsev，Mathias Bynens，Axel Rauschmayer，Nicholas Zakas，Angus Croll，Reginald Braithwaite，Dave Herman，Brendan Eich，Allen Wirfs-Brock，Bradley Meck，Domenic Denicola，David Walsh，Tim Disney，Peter van der Zee，Andrea Giammarchi，Kit Cambridge，Eric Elliott，和其他许多我甚至不能接触到的人。

你不懂 JS 系列丛书诞生于 Kickstarter，所以我也要感谢我的所有（将近）500 位慷慨的支持者，没有他们这部丛书不可能诞生：

> Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hods-

den, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel - Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuţ Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clan-

ton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

这部丛书是以开源的风格书写的，包括编辑和生产。我们感激 GitHub 使这样的事情在社区中成为可能!

再次感谢所有无数的朋友，尽管我不能叫上名字但是我依然亏欠感谢。但愿这部丛书被我

们所有人 “拥有”，并为增进对 JavaScript 语言的意识和理解做出贡献，成为当下和未来所有社区贡献者的助益。