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

myObject.a++; // 隐式屏蔽!

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

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

尽管 myObject.a++ 看起来应该（通过委托）查找并增加 anotherObject.a 属性，但是别忘了 ++ 操作相当于 myObject.a = myObject.a + 1。因此 ++ 操作首先会通过 [[Prototype]] 查找属性 a 并从 anotherObject.a 获取当前属性值 2，然后给这个值加 1，接着用 [[Put]] 将值 3 赋给 myObject 中新建的屏蔽属性 a，天呐!

修改委托属性时一定要小心。如果想让 anotherObject.a 的值增加，唯一的办法是 anotherObject.a++。

5.2 “类”

现在你可能会很好奇：为什么一个对象需要关联到另一个对象？这样做有什么好处？这个问题非常好，但是在回答之前我们首先要理解 [[Prototype]]“不是”什么。

第 4 章中我们说过，JavaScript 和面向类的语言不同，它并没有类来作为对象的抽象模式或者说蓝图。JavaScript 中只有对象。

实际上，JavaScript 才是真正应该被称为“面向对象”的语言，因为它是少有的可以不通过类，直接创建对象的语言。

在 JavaScript 中，类无法描述对象的行，（因为根本就不存在类！）对象直接定义自己的行为。再说一遍，JavaScript 中只有对象。

5.2.1 “类”函数

多年以来，JavaScript 中有一种奇怪的行为一直在被无耻地滥用，那就是*模仿*类。我们会仔细分析这种方法。

这种奇怪的“类似类”的行为利用了函数的一种特殊特性：所有的函数默认都会拥有一个名为 prototype 的公有并且不可枚举（参见第 3 章）的属性，它会指向另一个对象：

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

Foo.prototype; // { }
```

这个对象通常被称为 Foo 的*原型*，因为我们通过名为 Foo.prototype 的属性引用来访问它。然而不幸的是，这个术语对我们造成了极大的误导，稍后我们就会看到。如果是我的话就会叫它“之前被称为 Foo 的原型的那个对象”。好吧我是开玩笑的，你觉得“被贴上‘Foo 点 prototype’标签的对象”这个名字怎么样？

抛开名字不谈，这个对象到底是什么？

最直接的解释就是，这个对象是在调用 new Foo()（参见第 2 章）时创建的，最后会被（有点武断地）关联到这个“Foo 点 prototype”对象上。

我们来验证一下：

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

var a = new Foo();

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

调用 new Foo() 时会创建 a（具体的 4 个步骤参见第 2 章），其中的一步就是给 a 一个内部的 [[Prototype]] 链接，关联到 Foo.prototype 指向的那个对象。

暂停一下，仔细思考这条语句的含义。

在面向类的语言中，类可以被*复制*（或者说实例化）多次，就像用模具制作东西一样。我们在第 4 章中看到过，之所以会这样是因为实例化（或者继承）一个类就意味着“把类的行为复制到物理对象中”，对于每一个新实例来说都会重复这个过程。

但是在 JavaScript 中，并没有类似的复制机制。你不能创建一个类的多个实例，只能创建多个对象，它们 [[Prototype]] 关联的是同一个对象。但是在默认情况下并不会进行复制，因此这些对象之间并不会完全失去联系，它们是互相关联的。

new Foo() 会生成一个新对象（我们称之为 a），这个新对象的内部链接 [[Prototype]] 关联的是 Foo.prototype 对象。

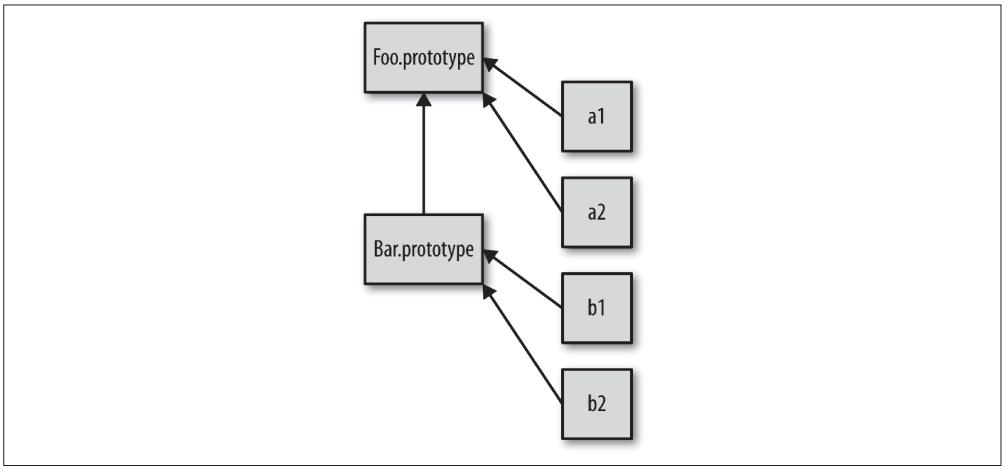
最后我们得到了两个对象，它们之间互相关联，就是这样。我们并没有初始化一个类，实际上我们并没有从“类”中复制任何行为到一个对象中，只是让两个对象互相关联。

实际上，绝大多数 JavaScript 开发者不知道的秘密是，new Foo() 这个函数调用实际上并没有*直接*创建关联，这个关联只是一个意外的副作用。new Foo() 只是间接完成了我们的目标：一个关联到其他对象的新对象。

那么有没有*更直接*的方法来做到这一点呢？当然！功臣就是 Object.create(..)，不过我们现在暂时不介绍它。

关于名称

在 JavaScript 中，我们并不会将一个对象（“类”）复制到另一个对象（“实例”），只是将它们关联起来。从视觉角度来说，`[[Prototype]]` 机制如下图所示，箭头从右到左，从下到上：



这个机制通常被称为*原型继承*（稍后我们会分析具体代码），它常常被视为动态语言版本的类继承。这个名称主要是为了对应面向类的世界中“继承”的意义，但是*违背*（写作违背，读作推翻）了动态脚本中对应的语义。

“继承”这个词会让人产生非常强的心理预期（参见第 4 章）。仅仅在前面加上“原型”并不能区分出 JavaScript 中和类继承*几乎完全相反*的行为，因此在过去 20 年中造成了极大的误解。

在我看来，在“继承”前面加上“原型”对于事实的曲解就好像一只手拿橘子一只手拿苹果然后把苹果叫作“红橘子”一样。无论添加什么标签都无法改变*事实*：一种水果是苹果，另一种是橘子。

更好的方法是直接把苹果叫作苹果——使用更加准确并且直接的术语。这样有助于理解它们的相似之处以及不同之处，因为我们大家都明白“苹果”的含义。

因此我认为这个容易混淆的组合术语“原型继承”（以及使用其他面向类的术语比如“类”、“构造函数”、“实例”、“多态”，等等）严重影响了大家对于 JavaScript 机制真实原理的理解。

*继承*意味着复制操作，JavaScript（默认）并不会复制对象属性。相反，JavaScript 会在两个对象之间创建一个关联，这样一个对象就可以通过*委托*访问另一个对象的属性和函数。*委托*（参见第 6 章）这个术语可以更加准确地描述 JavaScript 中对象的关联机制。

还有个偶尔会用到的 JavaScript 术语*差异继承*。基本原则是在描述对象行为时，使用其不

同于普遍描述的特质。举例来说，描述汽车时你会说汽车是有四个轮子的一种交通工具，但是你不会重复描述交通工具具备的通用特性（比如引擎）。

如果你把 JavaScript 中对象的所有委托行为都归结到对象本身并且把对象看作是实物的话，那就（差不多）可以理解差异继承了。

但是和原型继承一样，差异继承会更多是你脑中构建出的模型，而非真实情况。它忽略了一个事实，那就是对象 B 实际上并不是被差异构造出来的，我们只是定义了 B 的一些指定特性，其他没有定义的东西都变成了“洞”。而这些洞（或者说缺少定义的空白处）最终会被委托行为“填满”。

默认情况下，对象并不会像差异继承暗示的那样通过复制生成。因此，差异继承也不适合用来描述 JavaScript 的 [[Prototype]] 机制。

当然，如果你喜欢，完全可以使用差异继承这个术语，但是无论如何它只适用于你脑中的模型，并不符合引擎的真实行为。

5.2.2 “构造函数”

好了，回到之前的代码：

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

var a = new Foo();
```

到底是什么让我们认为 Foo 是一个“类”呢？

其中一个原因是我们看到了关键字 new，在面向类的语言中构造类实例时也会用到它。另一个原因是，看起来我们执行了类的构造函数方法，Foo() 的调用方式很像初始化类时类构造函数的调用方式。

除了令人迷惑的“构造函数”语义外，Foo.prototype 还有另一个绝招。思考下面的代码：

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

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

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

Foo.prototype 默认（在代码中第一行声明时！）有一个公有并且不可枚举（参见第 3 章）的属性 .constructor，这个属性引用的是对象关联的函数（本例中是 Foo）。此外，我们

可以看到通过“构造函数”调用 new Foo() 创建的对象也有一个 .constructor 属性，指向“创建这个对象的函数”。

实际上 a 本身并没有 .constructor 属性。而且，虽然 a.constructor 确实指向 Foo 函数，但是这个属性并不是表示 a 由 Foo“构造”，稍后我们会解释。

哦耶，好吧……按照 JavaScript 世界的惯例，“类”名首字母要大写，所以名字写作 Foo 而非 foo 似乎也提示它是一个“类”。显而易见，是吧 ?!

这个惯例影响力非常大，以至于如果你用 new 来调用小写方法或者不用 new 调用首字母大写的函数，许多 JavaScript 开发者都会责怪你。这很令人吃惊，我们竟然会如此努力地维护 JavaScript 中（假）“面向类”的权力，尽管对于 JavaScript 引擎来说首字母大写没有任何意义。

1. 构造函数还是调用

上一段代码很容易让人认为 Foo 是一个构造函数，因为我们使用 new 来调用它并且看到它“构造”了一个对象。

实际上，Foo 和你程序中的其他函数没有任何区别。函数本身并不是构造函数，然而，当你在普通的函数调用前面加上 new 关键字之后，就会把这个函数调用变成一个“构造函数调用”。实际上，new 会劫持所有普通函数并用构造对象的形式来调用它。

举例来说：

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

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

a; // {}
```

NothingSpecial 只是一个普通的函数，但是使用 new 调用时，它就会构造一个对象并赋值给 a，这看起来像是 new 的一个副作用（无论如何都会构造一个对象）。这个调用是一个构造函数调用，但是 NothingSpecial 本身并不是一个构造函数。

换句话说，在 JavaScript 中对于“构造函数”最准确的解释是，所有带 new 的函数调用。

函数不是构造函数，但是当且仅当使用 new 时，函数调用会变成“构造函数调用”。

5.2.3 技术

我们是不是已经介绍了 JavaScript 中所有和“类”相关的问题了呢？

不是。JavaScript 开发者绞尽脑汁想要模仿类的行为：

```
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”的技巧：

1. `this.name = name` 给每个对象（也就是 `a` 和 `b`，参见第 2 章中的 `this` 绑定）都添加了 `.name` 属性，有点像类实例封装的数据值。
2. `Foo.prototype.myName = ...` 可能个更有趣的技巧，它会给 `Foo.prototype` 对象添加一个属性（函数）。现在，`a.myName()` 可以正常工作，但是你可能会觉得很惊讶，这是什么原理呢？

在这段代码中，看起来似乎创建 `a` 和 `b` 时会把 `Foo.prototype` 对象复制到这两个对象中，然而事实并不是这样。

在本章开头介绍默认 `[[Get]]` 算法时我们介绍过 `[[Prototype]]` 链，以及当属性不直接存在于对象中时如何通过它来进行查找。

因此，在创建的过程中，`a` 和 `b` 的内部 `[[Prototype]]` 都会关联到 `Foo.prototype` 上。当 `a` 和 `b` 中无法找到 `myName` 时，它会（通过委托，参见第 6 章）在 `Foo.prototype` 上找到。

回顾“构造函数”

之前讨论 `.constructor` 属性时我们说过，看起来 `a.constructor === Foo` 为真意味着 `a` 确实有一个指向 `Foo` 的 `.constructor` 属性，但是事实不是这样。

这是一个很不幸的误解。实际上，`.constructor` 引用同样被委托给了 `Foo.prototype`，而 `Foo.prototype.constructor` 默认指向 `Foo`。

把 `.constructor` 属性指向 `Foo` 看作是 `a` 对象由 `Foo`“构造”非常容易理解，但这只不过是一种虚假的安全感。`a.constructor` 只是通过默认的 `[[Prototype]]` 委托指向 `Foo`，这和

“构造”毫无关系。相反，对于 .constructor 的错误理解很容易对你自己产生误导。

举例来说，Foo.prototype 的 .constructor 属性只是 Foo 函数在声明时的默认属性。如果你创建了一个新对象并替换了函数默认的 .prototype 对象引用，那么新对象并不会自动获得 .constructor 属性。

思考下面的代码：

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

Foo.prototype = { /* .. */ }; // 创建一个新原型对象

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

Object(..) 并没有“构造”a1，对吧？看起来应该是 Foo()“构造”了它。大部分开发者都认为是 Foo() 执行了构造工作，但是问题在于，如果你认为“constructor”表示“由……构造”的话，a1.constructor 应该是 Foo，但是它并不是 Foo！

到底怎么回事？ a1 并没有 .constructor 属性，所以它会委托 [[Prototype]] 链上的 Foo.prototype。但是这个对象也没有 .constructor 属性（不过默认的 Foo.prototype 对象有这个属性！），所以它会继续委托，这次会委托给委托链顶端的 Object.prototype。这个对象有 .constructor 属性，指向内置的 Object(..) 函数。

错误观点已被摧毁。

当然，你可以给 Foo.prototype 添加一个 .constructor 属性，不过这需要手动添加一个符合正常行为的不可枚举（参见第 3 章）属性。

举例来说：

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

Foo.prototype = { /* .. */ }; // 创建一个新原型对象

// 需要在 Foo.prototype 上“修复”丢失的 .constructor 属性
// 新对象属性起到 Foo.prototype 的作用
// 关于 defineProperty(..)，参见第 3 章
Object.defineProperty( Foo.prototype, "constructor" , {
    enumerable: false,
    writable: true,
    configurable: true,
    value: Foo // 让 .constructor 指向 Foo
} );
```

修复 .constructor 需要很多手动操作。所有这些工作都是源于把“constructor”错误地理解为“由……构造”，这个误解的代价实在太高了。

实际上，对象的 .constructor 会默认指向一个函数，这个函数可以通过对象的 .prototype 引用。“constructor”和“prototype”这两个词本身的含义可能适用也可能不适用。最好的办法是记住这一点“constructor 并不表示被构造”。

.constructor 并不是一个不可变属性。它是不可枚举（参见上面的代码）的，但是它的值是可写的（可以被修改）。此外，你可以给任意 [[Prototype]] 链中的任意对象添加一个名为 constructor 的属性或者对其进行修改，你可以任意对其赋值。

和 [[Get]] 算法查找 [[Prototype]] 链的机制一样，.constructor 属性引用的目标可能和你想的完全不同。

现在你应该明白这个属性多么随意了吧？

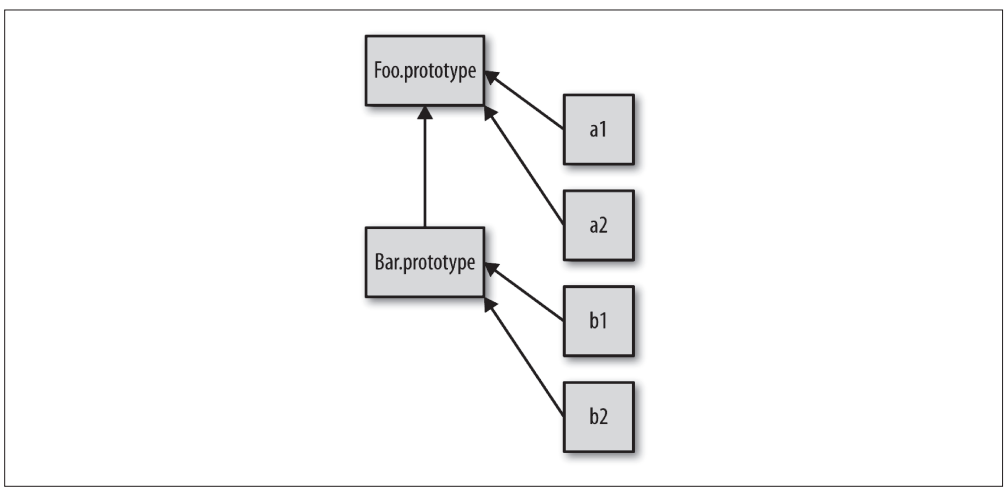
结论？一些随意的对象属性引用，比如 a1.constructor，实际上是不被信任的，它们不一定会指向默认的函数引用。此外，很快我们就会看到，稍不留神 a1.constructor 就可能会指向你意想不到的地方。

a1.constructor 是一个非常不可靠并且不安全的引用。通常来说要尽量避免使用这些引用。

5.3 （原型）继承

我们已经看过了许多 JavaScript 程序中常用的模拟类行为的方法，但是如果没有“继承”机制的话，JavaScript 中的类就只是一个空架子。

实际上，我们已经了解了通常被称作原型继承的机制，a 可以“继承”Foo.prototype 并访问 Foo.prototype 的 myName() 函数。但是之前我们只把继承看作是类和类之间的关系，并没有把它看作是类和实例之间的关系：



还记得这张图吗，它不仅展示出对象（实例）a1 到 Foo.prototype 的委托关系，还展示出 Bar.prototype 到 Foo.prototype 的委托关系，而后者和类继承很相似，只有箭头的方向不同。图中由下到上的箭头表明这是委托关联，不是复制操作。

下面这段代码使用的就是典型的“原型风格”：

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

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

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

// 我们创建了一个新的 Bar.prototype 对象并关联到 Foo.prototype
Bar.prototype = Object.create( Foo.prototype );

// 注意！现在没有 Bar.prototype.constructor 了
// 如果你需要这个属性的话可能需要手动修复一下它

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

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

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

如果不明白为什么 this 指向 a 的话，请查看第 2 章。

这段代码的核心部分就是语句 Bar.prototype = Object.create(Foo.prototype)。调用 Object.create(..) 会凭空创建一个“新”对象并把新对象内部的 [[Prototype]] 关联到你指定的对象（本例中是 Foo.prototype）。

换句话说，这条语句的意思是：“创建一个新的 Bar.prototype 对象并把它关联到 Foo.prototype”。

声明 function Bar() { .. } 时，和其他函数一样，Bar 会有一个 .prototype 关联到默认的对象，但是这个对象并不是我们想要的 Foo.prototype。因此我们创建了一个新对象并把

它关联到我们希望的对象上，直接把原始的关联对象抛弃掉。

注意，下面这两种方式是常见的错误做法，实际上它们都存在一些问题：

```
// 和你想要的机制不一样!
Bar.prototype = Foo.prototype;

// 基本上满足你的需求，但是可能会产生一些副作用 :(
Bar.prototype = new Foo();
```

`Bar.prototype = Foo.prototype` 并不会创建一个关联到 `Bar.prototype` 的新对象，它只是让 `Bar.prototype` 直接引用 `Foo.prototype` 对象。因此当你执行类似 `Bar.prototype.myLabel = ...` 的赋值语句时会直接修改 `Foo.prototype` 对象本身。显然这不是你想要的结果，否则你根本不需要 `Bar` 对象，直接使用 `Foo` 就可以了，这样代码也会更简单一些。

`Bar.prototype = new Foo()` 的确会创建一个关联到 `Bar.prototype` 的新对象。但是它使用了 `Foo(..)` 的“构造函数调用”，如果函数 `Foo` 有一些副作用（比如写日志、修改状态、注册到其他对象、给 `this` 添加数据属性，等等）的话，就会影响到 `Bar()` 的“后代”，后果不堪设想。

因此，要创建一个合适的关联对象，我们必须使用 `Object.create(..)` 而不是使用具有副作用的 `Foo(..)`。这样做唯一的缺点就是需要创建一个新对象然后把旧对象抛弃掉，不能直接修改已有的默认对象。

如果能有一个标准并且可靠的方法来修改对象的 `[[Prototype]]` 关联就好了。在 ES6 之前，我们只能通过设置 `.__proto__` 属性来实现，但是这个方法并不是标准并且无法兼容所有浏览器。ES6 添加了辅助函数 `Object.setPrototypeOf(..)`，可以用标准并且可靠的方法来修改关联。

我们来对比一下两种把 `Bar.prototype` 关联到 `Foo.prototype` 的方法：

```
// ES6 之前需要抛弃默认的 Bar.prototype
Bar.ptototype = Object.create( Foo.prototype );

// ES6 开始可以直接修改现有的 Bar.prototype
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

如果忽略掉 `Object.create(..)` 方法带来的轻微性能损失（抛弃的对象需要进行垃圾回收），它实际上比 ES6 及其之后的方法更短而且可读性更高。不过无论如何，这是两种完全不同的语法。

检查“类”关系

假设有对象 `a`，如何寻找对象 `a` 委托的对象（如果存在的话）呢？在传统的面向类环境中，