检查一个实例（JavaScript 中的对象）的继承祖先（JavaScript 中的委托关联）通常被称为*内省*（或者*反射*）。

思考下面的代码：

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

Foo.prototype.blah = ...;

var a = new Foo();
```

我们如何通过内省找出 a 的“祖先”（委托关联）呢？第一种方法是站在“类”的角度来判断：

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

instanceof 操作符的左操作数是一个普通的对象，右操作数是一个函数。instanceof 回答的问题是：在 a 的整条 [[Prototype]] 链中是否有指向 Foo.prototype 的对象？

可惜，这个方法只能处理对象（a）和函数（带 .prototype 引用的 Foo）之间的关系。如果你想判断两个对象（比如 a 和 b）之间是否通过 [[Prototype]] 链关联，只用 instanceof 无法实现。

如果使用内置的 .bind(..) 函数来生成一个硬绑定函数（参见第 2 章）的话，该函数是没有 .prototype 属性的。在这样的函数上使用 instanceof 的话，目标函数的 .prototype 会代替硬绑定函数的 .prototype。

通常我们不会在“构造函数调用”中使用硬绑定函数，不过如果你这么做的话，实际上相当于直接调用目标函数。同理，在硬绑定函数上使用 instanceof 也相当于直接在目标函数上使用 instanceof。

下面这段荒谬的代码试图站在“类”的角度使用 instanceof 来判断两个对象的关系：

```
// 用来判断 o1 是否关联到（委托）o2 的辅助函数
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(..) 内部我们声明了一个一次性函数 F，把它的 .prototype 重新赋值并指向对象 o2，然后判断 o1 是否是 F 的一个“实例”。显而易见，o1 实际上并没有继承 F 也不是由 F 构造，所以这种方法非常愚蠢并且容易造成误解。问题的关键在于思考的角度，强行在 JavaScript 中应用类的语义（在本例中就是使用 instanceof）就会造成这种尴尬的局面。

下面是第二种判断 [[Prototype]] 反射的方法，它更加简洁：

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

注意，在本例中，我们实际上并不关心（甚至不需要）Foo，我们只需要一个可以用来判断的对象（本例中是 Foo.prototype）就行。isPrototypeOf(..) 回答的问题是：在 a 的整条 [[Prototype]] 链中是否出现过 Foo.prototype？

同样的问题，同样的答案，但是在第二种方法中并不需要间接引用函数（Foo），它的 .prototype 属性会被自动访问。

我们只需要两个对象就可以判断它们之间的关系。举例来说：

```
// 非常简单：b 是否出现在 c 的 [[Prototype]] 链中？
b.isPrototypeOf( c );
```

注意，这个方法并不需要使用函数（“类”），它直接使用 b 和 c 之间的对象引用来判断它们的关系。换句话说，语言内置的 isPrototypeOf(..) 函数就是我们的 isRelatedTo(..) 函数。

我们也可以直接获取一个对象的 [[Prototype]] 链。在 ES5 中，标准的方法是：

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

可以验证一下，这个对象引用是否和我们想的一样：

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

绝大多数（不是所有！）浏览器也支持一种非标准的方法来访问内部 [[Prototype]] 属性：

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

这个奇怪的 .__proto__（在 ES6 之前并不是标准！）属性“神奇地”引用了内部的 [[Prototype]] 对象，如果你想直接查找（甚至可以通过 .__proto__.__ptoto__... 来遍历）原型链的话，这个方法非常有用。

和我们之前说过的 .constructor 一样，.__proto__ 实际上并不存在于你正在使用的对象中（本例中是 a）。实际上，它和其他的常用函数（.toString()、.isPrototypeOf(..)，等等）

一样，存在于内置的 Object.prototype 中。（它们是不可枚举的，参见第 2 章。）

此外，.__proto__ 看起来很像一个属性，但是实际上它更像一个 getter/setter（参见第 3 章）。

.__proto__ 的实现大致上是这样的（对象属性的定义参见第 3 章）：

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

因此，访问（获取值）a.__proto__ 时，实际上是调用了 a.__proto__()（调用 getter 函数）。虽然 getter 函数存在于 Object.prototype 对象中，但是它的 this 指向对象 a（this 的绑定规则参见第 2 章），所以和 Object.getPrototypeOf(a) 结果相同。

.__proto__ 是可设置属性，之前的代码中使用 ES6 的 Object.setPrototypeOf(..) 进行设置。然而，通常来说你不需要修改已有对象的 [[Prototype]]。

一些框架会使用非常复杂和高端的技术来实现“子类”机制，但是通常来说，我们不推荐这种用法，因为这会极大地增加代码的阅读难度和维护难度。

ES6 中的 class 关键字可以在内置的类型（比如 Array）上实现类似“子类”的功能。详情参考附录 A 中关于 ES6 中 class 语法的介绍。

我们只有在一些特殊情况下（我们前面讨论过）需要设置函数默认 .prototype 对象的 [[Prototype]]，让它引用其他对象（除了 Object.prototype）。这样可以避免使用全新的对象替换默认对象。此外，最好把 [[Prototype]] 对象关联看作是只读特性，从而增加代码的可读性。

JavaScript 社区中对于双下划线有一个非官方的称呼，他们会把类似 __proto__ 的属性称为“笨蛋（dunder）”。所以，JavaScript 潮人会把 __proto__ 叫作“笨蛋 proto”。

5.4 对象关联

现在我们知道了，[[Prototype]] 机制就是存在于对象中的一个内部链接，它会引用其他对象。

通常来说，这个链接的作用是：如果在对象上没有找到需要的属性或者方法引用，引擎就会继续在 [[Prototype]] 关联的对象上进行查找。同理，如果在后者中也没有找到需要的引用就会继续查找它的 [[Prototype]]，以此类推。这一系列对象的链接被称为“原型链”。

5.4.1 创建关联

我们已经明白了为什么 JavaScript 的 [[Prototype]] 机制和类不一样，也明白了它如何建立对象间的关联。

那 [[Prototype]] 机制的意义是什么呢？为什么 JavaScript 开发者费这么大的力气（模拟类）在代码中创建这些关联呢？

还记得吗，本章前面曾经说过 Object.create(..) 是一个大英雄，现在是时候来弄明白为什么了：

```
var foo = {
    something: function() {
        console.log( "Tell me something good..." );
    }
};

var bar = Object.create( foo );

bar.something(); // Tell me something good...
```

Object.create(..) 会创建一个新对象（bar）并把它关联到我们指定的对象（foo），这样我们就可以充分发挥 [[Prototype]] 机制的威力（委托）并且避免不必要的麻烦（比如使用 new 的构造函数调用会生成 .prototype 和 .constructor 引用）。

Object.create(null) 会创建一个拥有空（或者说 null）[[Prototype]] 链接的对象，这个对象无法进行委托。由于这个对象没有原型链，所以 instanceof 操作符（之前解释过）无法进行判断，因此总是会返回 false。这些特殊的空 [[Prototype]] 对象通常被称作“字典”，它们完全不会受到原型链的干扰，因此非常适合用来存储数据。

我们并不需要类来创建两个对象之间的关系，只需要通过委托来关联对象就足够了。而 Object.create(..) 不包含任何“类的诡计”，所以它可以完美地创建我们想要的关联关系。

Object.create()的polyfill代码

Object.create(..)是在ES5中新增的函数，所以在ES5之前的环境中（比如旧IE）如果要支持这个功能的话就需要使用一段简单的polyfill代码，它部分实现了Object.create(..)的功能：

```
if (!Object.create) {
    Object.create = function(o) {
        function F(){}
        F.prototype = o;
        return new F();
    };
}
```

这段polyfill代码使用了一个一次性函数F，我们通过改写它的.prototype属性使其指向想要关联的对象，然后再使用new F()来构造一个新对象进行关联。

由于Object.create(..c)可以被模拟，因此这个函数被应用得非常广泛。标准ES5中内置的Object.create(..)函数还提供了一系列附加功能，但是ES5之前的版本不支持这些功能。通常来说，这些功能的应用范围要小得多，但是出于完整性考虑，我们还是介绍一下：

```
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(..)的第二个参数指定了需要添加到新对象中的属性名以及这些属性的属性

描述符（参见第 3 章）。因为 ES5 之前的版本无法模拟属性操作符，所以 polyfill 代码无法实现这个附加功能。

通常来说并不会使用 `Object.create(..)` 的附加功能，所以对于大多数开发者来说，上面那段 polyfill 代码就足够了。

有些开发者更加严谨，他们认为只有能被完全模拟的函数才应该使用 polyfill 代码。由于 `Object.create(..)` 是只能部分模拟的函数之一，所以这些狭隘的人认为如果你需要在 ES5 之前的环境中使用 `Object.create(..)` 的特性，那不要使用 polyfill 代码，而是使用一个自定义函数并且名字不能是 `Object.create`。你可以把你自己的函数定义成这样：

```
function createAndLinkObject(o) {
    function F(){}
    F.prototype = o;
    return new F();
}

var anotherObject = {
    a:2
};

var myObject = createAndLinkObject( anotherObject );

myObject.a; // 2
```

我并不赞同这个严格的观点，相反，我很赞同在 ES5 中使用上面那段 polyfill 代码。如何选择取决于你。

5.4.2 关联关系是备用

看起来对象之间的关联关系是处理“缺失”属性或者方法时的一种备用选项。这个说法有点道理，但是我认为这并不是 `[[Prototype]]` 的本质。

思考下面的代码：

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.cool(); // "cool!"
```

由于存在 `[[Prototype]]` 机制，这段代码可以正常工作。但是如果你这样写只是为了让 `myObject` 在无法处理属性或者方法时可以使用备用的 `anotherObject`，那么你的软件就会

变得有点“神奇”，而且很难理解和维护。

这并不是说任何情况下都不应该选择备用这种设计模式，但是这在 JavaScript 中并不是很常见。所以如果你使用的是这种模式，那或许应当退后一步并重新思考一下这种模式是否合适。

在 ES6 中有一个被称为“代理”（Proxy）的高端功能，它实现的就是“方法无法找到”时的行为。代理超出了本书的讨论范围，但是在本系列之后的书中会介绍。

千万不要忽略这个微妙但是非常重要的区别。

当你给开发者设计软件时，假设要调用 `myObject.cool()`，如果 `myObject` 中不存在 `cool()` 时这条语句也可以正常工作的话，那你的 API 设计就会变得很“神奇”，对于未来维护你软件的开发者来说这可能不太好理解。

但是你可以让你的 API 设计不那么“神奇”，同时仍然能发挥 `[[Prototype]]` 关联的威力：

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.doCool = function() {
    this.cool(); // 内部委托!
};

myObject.doCool(); // "cool!"
```

这里我们调用的 `myObject.doCool()` 是*实际存在于* `myObject` 中的，这可以让我们的 API 设计更加清晰（不那么“神奇”）。从*内部*来说，我们的实现遵循的是*委托设计模式*（参见第 6 章），通过 `[[Prototype]]` 委托到 `anotherObject.cool()`。

换句话说，内部委托比起直接委托可以让 API 接口设计更加清晰。下一章我们会详细解释委托。

5.5 小结

如果要访问对象中并不存在的一个属性，`[[Get]]` 操作（参见第 3 章）就会查找对象内部 `[[Prototype]]` 关联的对象。这个关联关系实际上定义了一条“原型链”（有点像嵌套的作

用域链），在查找属性时会对它进行遍历。

所有普通对象都有内置的 `Object.prototype`，指向原型链的顶端（比如说全局作用域），如果在原型链中找不到指定的属性就会停止。`toString()`、`valueOf()` 和其他一些通用的功能都存在于 `Object.prototype` 对象上，因此语言中所有的对象都可以使用它们。

关联两个对象最常用的方法是使用 `new` 关键词进行函数调用，在调用的 4 个步骤（第 2 章）中会创建一个关联其他对象的新对象。

使用 `new` 调用函数时会把新对象的 `.prototype` 属性关联到“其他对象”。带 `new` 的函数调用通常被称为“构造函数调用”，尽管它们实际上和传统面向类语言中的*类构造函数*不一样。

虽然这些 JavaScript 机制和传统面向类语言中的“类初始化”和“类继承”很相似，但是 JavaScript 中的机制有一个核心区别，那就是不会进行复制，对象之间是通过内部的 `[[Prototype]]` 链关联的。

出于各种原因，以“继承”结尾的术语（包括“原型继承”）和其他面向对象的术语都无法帮助你理解 JavaScript 的*真实机制*（不仅仅是限制我们的思维模式）。

相比之下，“委托”是一个更合适的术语，因为对象之间的关系不是*复制*而是委托。

第 6 章

行为委托

第 5 章详细介绍了 [[Prototype]] 机制并说明了为什么在“类”或者“继承”的背景下讨论 [[Prototype]] 容易产生误解（这种不恰当的方式已经持续了 20 年）。我们搞清楚了繁杂的语法（各种 .prototype 代码），也见识了各种各样的陷阱（比如出人意料的 .constructor 和丑陋的伪多态语法），我们还看到了用来解决这些问题的各种“混入”方法。

你可能会很好奇，为什么看起来简单的事情会变得这么复杂。现在我们会把帘子拉开，看看后面到底有什么。不出意外，绝大多数 JavaScript 开发者从来没有如此深入地了解过 JavaScript，他们只是把这些交给一个“类”库来处理。

现在，我希望你不仅满足于掩盖这些细节并把它们交给一个“黑盒”库。忘掉令人困惑的类，我们用一种更加简单直接的方法来深入发掘一下 JavaScript 中对象的 [[Prototype]] 机制到底是什么。

首先简单回顾一下第 5 章的结论：[[Prototype]] 机制就是指对象中的一个内部链接引用另一个对象。

如果在第一个对象上没有找到需要的属性或者方法引用，引擎就会继续在 [[Prototype]] 关联的对象上进行查找。同理，如果在后者中也没有找到需要的引用就会继续查找它的 [[Prototype]]，以此类推。这一系列对象的链接被称为“原型链”。

换句话说，JavaScript 中这个机制的本质就是对象之间的关联关系。

这个观点对于理解本章的内容来说是非常基础并且非常重要的。

6.1 面向委托的设计

为了更好地学习如何更直观地使用 [[Prototype]]，我们必须认识到它代表的是一种不同于类（参见第 4 章）的设计模式。

面向类的设计中有些原则依然有效，因此不要把所有知识都抛掉。（只需要抛掉大部分就够了！）举例来说，封装是非常有用的，它同样可以应用在委托中（虽然不太常见）。

我们需要试着把思路从类和继承的设计模式转换到委托行为的设计模式。如果你在学习或者工作的过程中几乎一直在使用类，那转换思路可能不太自然并且不太舒服。你可能需要多重复几次才能熟悉这种思维模式。

首先我会带你们进行一些理论训练，然后再传授一些能够应用在代码中的具体实例。

6.1.1 类理论

假设我们需要在软件中建模一些类似的任务（“XYZ”、“ABC”等）。

如果使用类，那设计方法可能是这样的：定义一个通用父（基）类，可以将其命名为 Task，在 Task 类中定义所有任务都有的行为。接着定义子类 XYZ 和 ABC，它们都继承自 Task 并且会添加一些特殊的行为来处理对应的任务。

非常重要的是，类设计模式鼓励你在继承时使用方法重写（和多态），比如说在 XYZ 任务中重写 Task 中定义的一些通用方法，甚至在添加新行为时通过 super 调用这个方法的原始版本。你会发现许多行为可以先“抽象”到父类然后再用子类进行特殊化（重写）。

下面是对应的伪代码：

```
class Task {
    id;

    // 构造函数 Task()
    Task(ID) { id = ID; }
    outputTask() { output( id ); }
}

class XYZ inherits Task {
    label;

    // 构造函数 XYZ()
    XYZ(ID,Label) { super( ID ); label = Label; }
    outputTask() { super(); output( label ); }
}
```