现在 Car 中就有了一份 Vehicle 属性和函数的副本了。从技术角度来说，函数实际上没有被复制，复制的是函数引用。所以，Car 中的属性 ignition 只是从 Vehicle 中复制过来的对于 ignition() 函数的引用。相反，属性 engines 就是直接从 Vehicle 中复制了值 1。

Car 已经有了 drive 属性（函数），所以这个属性引用并没有被 mixin 重写，从而保留了 Car 中定义的同名属性，实现了“子类”对“父类”属性的重写（参见 mixin(..) 例子中的 if 语句）。

1. 再说多态

我们来分析一下这条语句：Vehicle.drive.call(this)。这就是我所说的显式多态。还记得吗，在之前的伪代码中对应的语句是 inherited:drive()，我们称之为相对多态。

JavaScript（在 ES6 之前；参见附录 A）并没有相对多态的机制。所以，由于 Car 和 Vehicle 中都有 drive() 函数，为了指明调用对象，我们必须使用绝对（而不是相对）引用。我们通过名称显式指定 Vehicle 对象并调用它的 drive() 函数。

但是如果直接执行 Vehicle.drive()，函数调用中的 this 会被绑定到 Vehicle 对象而不是 Car 对象（参见第 2 章），这并不是我们想要的。因此，我们会使用 .call(this)（参见第 2 章）来确保 drive() 在 Car 对象的上下文中执行。

如果函数 Car.drive() 的名称标识符并没有和 Vehicle.drive() 重叠（或者说“屏蔽”；参见第 5 章）的话，我们就不需要实现方法多态，因为调用 mixin(..) 时会把函数 Vehicle.drive() 的引用复制到 Car 中，因此我们可以直接访问 this.drive()。正是由于存在标识符重叠，所以必须使用更加复杂的显式伪多态方法。

在支持相对多态的面向类的语言中，Car 和 Vehicle 之间的联系只在类定义的开头被创建，从而只需要在这一个地方维护两个类的联系。

但是在 JavaScript 中（由于屏蔽）使用显式伪多态会在所有需要使用（伪）多态引用的地方创建一个函数关联，这会极大地增加维护成本。此外，由于显式伪多态可以模拟多重继承，所以它会进一步增加代码的复杂度和维护难度。

使用伪多态通常会导致代码变得更加复杂、难以阅读并且难以维护，因此应当尽量避免使用显式伪多态，因为这样做往往得不偿失。

2. 混合复制

回顾一下之前提到的 mixin(..) 函数：

```
// 非常简单的 mixin(..) 例子：
function mixin( sourceObj, targetObj ) {
```

```
        for (var key in sourceObj) {
            // 只会在不存在的情况下复制
            if (!(key in targetObj)) {
                targetObj[key] = sourceObj[key];
            }
        }

        return targetObj;
    }
```

现在我们来分析一下 `mixin(..)` 的工作原理。它会遍历 `sourceObj`（本例中是 `Vehicle`）的属性，如果在 `targetObj`（本例中是 `Car`）没有这个属性就会进行复制。由于我们是在目标对象初始化之后才进行复制，因此一定要小心不要覆盖目标对象的原有属性。

如果我们是先进行复制然后对 `Car` 进行特殊化的话，就可以跳过存在性检查。不过这种方法并不好用并且效率更低，所以不如第一种方法常用：

```
    // 另一种混入函数，可能有重写风险
    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并把 Vehicle 的内容复制进去
    var Car = mixin( Vehicle, { } );

    // 然后把新内容复制到 Car 中
    mixin( {
        wheels: 4,

        drive: function() {
            // ...
        }
    }, Car );
```

这两种方法都可以把不重叠的内容从 `Vehicle` 中显性复制到 `Car` 中。“混入”这个名字来源于这个过程的另一种解释：`Car` 中混合了 `Vehicle` 的内容，就像你把巧克力片混合到你最喜欢的饼干面团中一样。

复制操作完成后，`Car` 就和 `Vehicle` 分离了，向 `Car` 中添加属性不会影响 `Vehicle`，反之亦然。

这里跳过了一些小细节，实际上，在复制完成之后两者之间仍然有一些巧妙的方法可以“影响”到对方，例如引用同一个对象（比如一个数组）。

由于两个对象引用的是同一个函数，因此这种复制（或者说混入）实际上并不能完全模拟面向类的语言中的复制。

JavaScript 中的函数无法（用标准、可靠的方法）真正地复制，所以你只能复制对共享函数对象的引用（函数就是对象；参见第 3 章）。如果你修改了共享的函数对象（比如 `ignition()`），比如添加了一个属性，那 `Vehicle` 和 `Car` 都会受到影响。

显式混入是 JavaScript 中一个很棒的机制，不过它的功能也没有看起来那么强大。虽然它可以把一个对象的属性复制到另一个对象中，但是这其实并不能带来太多的好处，无非就是少几条定义语句，而且还会带来我们刚才提到的函数对象引用问题。

如果你向目标对象中显式混入超过一个对象，就可以部分模仿多重继承行为，但是仍没有直接的方式来处理函数和属性的同名问题。有些开发者 / 库提出了“晚绑定”技术和其他的一些解决方法，但是从根本上来说，使用这些“诡计”通常会（降低性能并且）得不偿失。

一定要注意，只在能够提高代码可读性的前提下使用显式混入，避免使用增加代码理解难度或者让对象关系更加复杂的模式。

如果使用混入时感觉越来越困难，那或许你应该停止使用它了。实际上，如果你必须使用一个复杂的库或者函数来实现这些细节，那就标志着你的方法是有问题的或者是不必要的。第 6 章会试着提出一种更简单的方法，它能满足这些需求并且可以避免所有的问题。

3. 寄生继承

显式混入模式的一种变体被称为“寄生继承”，它既是显式的又是隐式的，主要推广者是 Douglas Crockford。

下面是它的工作原理：

```
// “传统的 JavaScript 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();

    // 接着我们对 car 进行定制
    car.wheels = 4;

    // 保存到 Vehicle::drive() 的特殊引用
    var vehDrive = car.drive;

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

    return car;
}

var myCar = new Car();

myCar.drive();
// 发动引擎。
// 手握方向盘！
// 全速前进！
```

如你所见，首先我们复制一份 Vehicle 父类（对象）的定义，然后混入子类（对象）的定义（如果需要的话保留到父类的特殊引用），然后用这个复合对象构建实例。

调用 new Car() 时会创建一个新对象并绑定到 Car 的 this 上（参见第 2 章）。但是因为我们没有使用这个对象而是返回了我们自己的 car 对象，所以最初被创建的这个对象会被丢弃，因此可以不使用 new 关键字调用 Car()。这样做得到的结果是一样的，但是可以避免创建并丢弃多余的对象。

4.4.2 隐式混入

隐式混入和之前提到的显式伪多态很像，因此也具备同样的问题。

思考下面的代码：

```
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() {
        // 隐式把 Something 混入 Another
        Something.cool.call( this );
    }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1（count 不是共享状态）
```

通过在构造函数调用或者方法调用中使用 `Something.cool.call( this )`，我们实际上“借用”了函数 `Something.cool()` 并在 `Another` 的上下文中调用了它（通过 `this` 绑定；参加第 2 章）。最终的结果是 `Something.cool()` 中的赋值操作都会应用在 `Another` 对象上而不是 `Something` 对象上。

因此，我们把 `Something` 的行为“混入”到了 `Another` 中。

虽然这类技术利用了 `this` 的重新绑定功能，但是 `Something.cool.call( this )` 仍然无法变成相对（而且更灵活的）引用，所以使用时千万要小心。通常来说，尽量避免使用这样的结构，以保证代码的整洁和可维护性。

4.5 小结

类是一种设计模式。许多语言提供了对于面向类软件设计的原生语法。JavaScript 也有类似的语法，但是和其他语言中的类完全不同。

类意味着复制。

传统的类被实例化时，它的行为会被复制到实例中。类被继承时，行为也会被复制到子类中。

多态（在继承链的不同层次名称相同但是功能不同的函数）看起来似乎是从子类引用父类，但是本质上引用的其实是复制的结果。

JavaScript 并不会（像类那样）自动创建对象的副本。

混入模式（无论显式还是隐式）可以用来模拟类的复制行为，但是通常会产生丑陋并且脆弱的语法，比如显式伪多态（`OtherObj.methodName.call(this, ...)`），这会让代码更加难懂并且难以维护。

此外，显式混入实际上无法完全模拟类的复制行为，因为对象（和函数！别忘了函数也是对象）只能复制引用，无法复制被引用的对象或者函数本身。忽视这一点会导致许多问题。

总地来说，在 JavaScript 中模拟类是得不偿失的，虽然能解决当前的问题，但是可能会埋下更多的隐患。

第 5 章

原型

第 3 章和第 4 章多次提到了 [[Prototype]] 链，但没有说它到底是什么。现在我们来详细介绍一下它。

第 4 章中介绍的所有模拟类复制行为的方法，如各种混入，都没有使用 [[Prototype]] 链机制。

5.1 [[Prototype]]

JavaScript 中的对象有一个特殊的 [[Prototype]] 内置属性，其实就是对于其他对象的引用。几乎所有的对象在创建时 [[Prototype]] 属性都会被赋予一个非空的值。

注意：很快我们就可以看到，对象的 [[Prototype]] 链接可以为空，虽然很少见。

思考下面的代码：

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

myObject.a; // 2
```

[[Prototype]] 引用有什么用呢？在第 3 章中我们说过，当你试图引用对象的属性时会触发

[[Get]] 操作，比如 myObject.a。对于默认的 [[Get]] 操作来说，第一步是检查对象本身是否有这个属性，如果有的话就使用它。

ES6 中的 Proxy 超出了本书的范围（但是在本系列之后的书中会介绍），但是要注意，如果包含 Proxy 的话，我们这里对 [[Get]] 和 [[Put]] 的讨论就不适用。

但是如果 a 不在 myObject 中，就需要使用对象的 [[Prototype]] 链了。

对于默认的 [[Get]] 操作来说，如果无法在对象本身找到需要的属性，就会继续访问对象的 [[Prototype]] 链：

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

// 创建一个关联到 anotherObject 的对象
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

稍后我们会介绍 Object.create(..) 的原理，现在只需要知道它会创建一个对象并把这个对象的 [[Prototype]] 关联到指定的对象。

现在 myObject 对象的 [[Prototype]] 关联到了 anotherObject。显然 myObject.a 并不存在，但是尽管如此，属性访问仍然成功地（在 anotherObject 中）找到了值 2。

但是，如果 anotherObject 中也找不到 a 并且 [[Prototype]] 链不为空的话，就会继续查找下去。

这个过程会持续到找到匹配的属性名或者查找完整条 [[Prototype]] 链。如果是后者的话，[[Get]] 操作的返回值是 undefined。

使用 for..in 遍历对象时原理和查找 [[Prototype]] 链类似，任何可以通过原型链访问到（并且是 enumerable，参见第 3 章）的属性都会被枚举。使用 in 操作符来检查属性在对象中是否存在时，同样会查找对象的整条原型链（无论属性是否可枚举）：

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

```
// 创建一个关联到 anotherObject 的对象
var myObject = Object.create( anotherObject );

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

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

因此，当你通过各种语法进行属性查找时都会查找 [[Prototype]] 链，直到找到属性或者查找完整条原型链。

5.1.1 Object.prototype

但是到哪里是 [[Prototype]] 的“尽头”呢？

所有普通的 [[Prototype]] 链最终都会指向内置的 Object.prototype。由于所有的“普通”（内置，不是特定主机的扩展）对象都“源于”（或者说把 [[Prototype]] 链的顶端设置为）这个 Object.prototype 对象，所以它包含 JavaScript 中许多通用的功能。

有些功能你应该已经很熟悉了，比如说 .toString() 和 .valueOf()，第 3 章还介绍过 .hasOwnProperty(..)。稍后我们还会介绍 .isPrototypeOf(..)，这个你可能不太熟悉。

5.1.2 属性设置和屏蔽

第 3 章提到过，给一个对象设置属性并不仅仅是添加一个新属性或者修改已有的属性值。现在我们完整地讲解一下这个过程：

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

如果 myObject 对象中包含名为 foo 的普通数据访问属性，这条赋值语句只会修改已有的属性值。

如果 foo 不是直接存在于 myObject 中，[[Prototype]] 链就会被遍历，类似 [[Get]] 操作。如果原型链上找不到 foo，foo 就会被直接添加到 myObject 上。

然而，如果 foo 存在于原型链上层，赋值语句 myObject.foo = "bar" 的行为就会有些不同（而且可能很出人意料）。稍后我们会进行介绍。

如果属性名 foo 既出现在 myObject 中也出现在 myObject 的 [[Prototype]] 链上层，那么就会发生屏蔽。myObject 中包含的 foo 属性会屏蔽原型链上层的所有 foo 属性，因为 myObject.foo 总是会选择原型链中最底层的 foo 属性。

屏蔽比我们想象中更加复杂。下面我们分析一下如果 foo 不直接存在于 myObject 中而是存

在于原型链上层时 myObject.foo = "bar" 会出现的三种情况。

1. 如果在 [[Prototype]] 链上层存在名为 foo 的普通数据访问属性（参见第 3 章）并且没有被标记为只读（writable:false），那就会直接在 myObject 中添加一个名为 foo 的新属性，它是*屏蔽属性*。
2. 如果在 [[Prototype]] 链上层存在 foo，但是它被标记为只读（writable:false），那么无法修改已有属性或者在 myObject 上创建屏蔽属性。如果运行在严格模式下，代码会抛出一个错误。否则，这条赋值语句会被忽略。总之，不会发生屏蔽。
3. 如果在 [[Prototype]] 链上层存在 foo 并且它是一个 setter（参见第 3 章），那就一定会调用这个 setter。foo 不会被添加到（或者说屏蔽于）myObject，也不会重新定义 foo 这个 setter。

大多数开发者都认为如果向 [[Prototype]] 链上层已经存在的属性（[[Put]]）赋值，就一定会触发屏蔽，但是如你所见，三种情况中只有一种（第一种）是这样的。

如果你希望在第二种和第三种情况下也屏蔽 foo，那就不能使用 = 操作符来赋值，而是使用 Object.defineProperty(..)（参见第 3 章）来向 myObject 添加 foo。

第二种情况可能是最令人意外的，*只读*属性会阻止 [[Prototype]] 链下层隐式创建（屏蔽）同名属性。这样做主要是为了模拟类属性的继承。你可以把原型链上层的 foo 看作是父类中的属性，它会被 myObject 继承（复制），这样一来 myObject 中的 foo 属性也是只读，所以无法创建。但是一定要注意，实际上并不会发生类似的继承复制（参见第 4 章和第 5 章）。这看起来有点奇怪，myObject 对象竟然会因为其他对象中有一个只读 foo 就不能包含 foo 属性。更奇怪的是，这个限制只存在于 = 赋值中，使用 Object.defineProperty(..) 并不会受到影响。

如果需要对屏蔽方法进行委托的话就不得不使用丑陋的显式伪多态（参见第 4 章）。通常来说，使用屏蔽得不偿失，所以应当尽量避免使用。第 6 章会介绍另一种不使用屏蔽的更加简洁的设计模式。

有些情况下会隐式产生屏蔽，一定要当心。思考下面的代码：

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

var myObject = Object.create( anotherObject );

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