instanceof 语法会产生语义困惑而且非常不直观。如果你想检查对象 a1 和某个对象的关系，那必须使用另一个引用该对象的函数才行——你不能直接判断两个对象是否关联。

还记得本章之前介绍的抽象的 Foo/Bar/b1 例子吗，简单来说是这样的：

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

function Bar() { /* .. */ }
Bar.prototype = Object.create( Foo.prototype );

var b1 = new Bar( "b1" );
```

如果要使用 instanceof 和 .prototype 语义来检查本例中实体的关系，那必须这样做：

```
// 让 Foo 和 Bar 互相关联
Bar.prototype instanceof Foo; // true
Object.getPrototypeOf( Bar.prototype )
   === Foo.prototype; // true
Foo.prototype.isPrototypeOf( Bar.prototype ); // true

// 让 b1 关联到 Foo 和 Bar
b1 instanceof Foo; // true
b1 instanceof Bar; // true
Object.getPrototypeOf( b1 ) === Bar.prototype; // true
Foo.prototype.isPrototypeOf( b1 ); // true
Bar.prototype.isPrototypeOf( b1 ); // true
```

显然这是一种非常糟糕的方法。举例来说，（使用类时）你最直观的想法可能是使用 Bar instanceof Foo（因为很容易把“实例”理解成“继承”），但是在 JavaScript 中这是行不通的，你必须使用 Bar.prototype instanceof Foo。

还有一种常见但是可能更加脆弱的内省模式，许多开发者认为它比 instanceof 更好。这种模式被称为“鸭子类型”。这个术语源自这句格言“如果看起来像鸭子，叫起来像鸭子，那就一定是鸭子。”

举例来说：

```
if (a1.something) {
    a1.something();
}
```

我们并没有检查 a1 和委托 something() 函数的对象之间的关系，而是假设如果 a1 通过了测试 a1.something 的话，那 a1 就一定能调用 .something()（无论这个方法存在于 a1 自身还是委托到其他对象）。这个假设的风险其实并不算很高。

但是“鸭子类型”通常会在测试之外做出许多关于对象功能的假设，这当然会带来许多风险（或者说脆弱的设计）。

ES6 的 Promise 就是典型的“鸭子类型”（之前解释过，本书并不会介绍 Promise）。

出于各种各样的原因，我们需要判断一个对象引用是否是 Promise，但是判断的方法是检查对象是否有 then() 方法。换句话说，如果对象有 then() 方法，ES6 的 Promise 就会认为这个对象是“可持续”（thenable）的，因此会期望它具有 Promise 的所有标准行为。

如果有一个不是 Promise 但是具有 then() 方法的对象，那你千万不要把它用在 ES6 的 Promise 机制中，否则会出错。

这个例子清楚地解释了“鸭子类型”的危害。你应该尽量避免使用这个方法，即使使用也要保证条件是可控的。

现在回到本章想说的对象关联风格代码，其内省更加简洁。我们先来回顾一下之前的 Foo/Bar/b1 对象关联例子（只包含关键代码）：

```
var Foo = { /* .. */ };

var Bar = Object.create( Foo );
Bar...

var b1 = Object.create( Bar );
```

使用对象关联时，所有的对象都是通过 [[Prototype]] 委托互相关联，下面是内省的方法，非常简单：

```
// 让 Foo 和 Bar 互相关联
Foo.isPrototypeOf( Bar ); // true
Object.getPrototypeOf( Bar ) === Foo; // true

// 让 b1 关联到 Foo 和 Bar
Foo.isPrototypeOf( b1 ); // true
Bar.isPrototypeOf( b1 ); // true
Object.getPrototypeOf( b1 ) === Bar; // true
```

我们没有使用 instanceof，因为它会产生一些和类有关的误解。现在我们想问的问题是“你是我的原型吗？”我们并不需要使用间接的形式，比如 Foo.prototype 或者繁琐的 Foo.prototype.isPrototypeOf(..)。

我觉得和之前的方法比起来，这种方法显然更加简洁并且清晰。再说一次，我们认为 JavaScript 中对象关联比类风格的代码更加简洁（而且功能相同）。

6.6 小结

在软件架构中你可以选择是否使用类和继承设计模式。大多数开发者理所当然地认为类是唯一（合适）的代码组织方式，但是本章中我们看到了另一种更少见但是更强大的设计模

式：行为委托。

行为委托认为对象之间是兄弟关系，互相委托，而不是父类和子类的关系。JavaScript 的 [[Prototype]] 机制本质上就是行为委托机制。也就是说，我们可以选择在 JavaScript 中努力实现类机制（参见第 4 和第 5 章），也可以拥抱更自然的 [[Prototype]] 委托机制。

当你只用对象来设计代码时，不仅可以让语法更加简洁，而且可以让代码结构更加清晰。

对象关联（对象之前互相关联）是一种编码风格，它倡导的是直接创建和关联对象，不把它们抽象成类。对象关联可以用基于 [[Prototype]] 的行为委托非常自然地实现。

附录 A

ES6中的Class

可以用一句话总结本书的第二部分（第 4 章至第 6 章）：类是一种可选（而不是必须）的设计模式，而且在 JavaScript 这样的 [[Prototype]] 语言中实现类是很别扭的。

这种别扭的感觉不只是来源于语法，虽然语法是很重要的原因。第 4 章和第 5 章介绍了许多语法的缺点：繁琐杂乱的 .prototype 引用、试图调用原型链上层同名函数时的显式伪多态（参见第 4 章）以及不可靠、不美观而且容易被误解成“构造函数”的 .constructor。

除此之外，类设计其实还存在更深刻的问题。第 4 章指出，传统面向类的语言中父类和子类、子类和实例之间其实是复制操作，但是在 [[Prototype]] 中并没有复制，相反，它们之间只有委托关联。

对象关联代码和行为委托（参见第 6 章）使用了 [[Prototype]] 而不是将它藏起来，对比其简洁性可以看出，类并不适用于 JavaScript。

A.1 class

不过我们并不需要再纠结于这个问题，这里提到只是让你简单回忆一下；现在我们来看看 ES6 的 class 机制。我们会介绍它的工作原理并分析 class 是否改进了之前提到的那些缺点。

首先回顾一下第 6 章中的 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( $where );
            this.$elem.click( this.onClick.bind( this ) );
        }
        onClick(evt) {
            console.log( "Button '" + this.label + "' clicked!" );
        }
    }
```

除了语法更好看之外，ES6 还解决了什么问题呢？

1. （基本上，下面会详细介绍）不再引用杂乱的 `.prototype` 了。
2. `Button` 声明时直接“继承”了 `Widget`，不再需要通过 `Object.create(..)` 来替换 `.prototype` 对象，也不需要设置 `.__proto__` 或者 `Object.setPrototypeOf(..)`。
3. 可以通过 `super(..)` 来实现相对多态，这样任何方法都可以引用原型链上层的同名方法。这可以解决第 4 章提到过的那个问题：构造函数不属于类，所以无法互相引用——`super()` 可以完美解决构造函数的问题。
4. `class` 字面语法不能声明属性（只能声明方法）。看起来这是一种限制，但是它会排除掉许多不好的情况，如果没有这种限制的话，原型链末端的“实例”可能会意外地获取其他地方的属性（这些属性隐式被所有“实例”所“共享”）。所以，`class` 语法实际上可以帮助你避免犯错。
5. 可以通过 `extends` 很自然地扩展对象（子）类型，甚至是内置的对象（子）类型，比如 `Array` 或 `RegExp`。没有 `class ..extends` 语法时，想实现这一点是非常困难的，基本上只有框架的作者才能搞清楚这一点。但是现在可以轻而易举地做到！

平心而论，`class` 语法确实解决了典型原型风格代码中许多显而易见的（语法）问题和缺点。

A.2 class陷阱

然而，class 语法并没有解决所有的问题，在 JavaScript 中使用“类”设计模式仍然存在许多深层问题。

首先，你可能会认为 ES6 的 class 语法是向 JavaScript 中引入了一种新的“类”机制，其实不是这样。class 基本上只是现有 [[Prototype]]（委托！）机制的一种语法糖。

也就是说，class 并不会像传统面向类的语言一样在声明时静态复制所有行为。如果你（有意或无意）修改或者替换了父“类”中的一个方法，那子“类”和所有实例都会受到影响，因为它们在定义时并没有进行复制，只是使用基于 [[Prototype]] 的实时委托：

```
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" ——噢!
```

如果你已经明白委托的原理所以并不会期望得到“类”的副本的话，那这种行为才看起来比较合理。所以你需要问自己：为什么要使用本质上不是类的 class 语法呢？

ES6 中的 class 语法不是会让传统类和委托对象之间的区别更加难以发现和理解吗？

class 语法无法定义类成员属性（只能定义方法），如果为了跟踪实例之间共享状态必须要这么做，那你只能使用丑陋的 .prototype 语法，像这样：

```
class C {
    constructor() {
        // 确保修改的是共享状态而不是在实例上创建一个屏蔽属性!
        C.prototype.count++;

        // this.count 可以通过委托实现我们想要的功能
        console.log( "Hello: " + this.count );
    }
```

```
}

// 直接向 prototype 对象上添加一个共享状态
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
```

这种方法最大的问题是，它违背了class语法的本意，在实现中暴露（泄露！）了.prototype。

如果使用this.count++的话，我们会很惊讶地发现在对象c1和c2上都创建了.count属性，而不是更新共享状态。class没有办法解决这个问题，并且干脆就不提供相应的语法支持，所以你根本就不应该这样做。

此外，class语法仍然面临意外屏蔽的问题：

```
class C {
    constructor(id) {
        // 噢，郁闷，我们的 id 属性屏蔽了 id() 方法
        this.id = id;
    }
    id() {
        console.log( "Id: " + id );
    }
}

var c1 = new C( "c1" );
c1.id(); // TypeError -- c1.id 现在是字符串 "c1"
```

除此之外，super也存在一些非常细微的问题。你可能认为super的绑定方法和this类似（参见第2章），也就是说，无论目前的方法在原型链中处于什么位置，super总会绑定到链中的上一层。

然而，出于性能考虑（this绑定已经是很大的开销了），super并不是动态绑定的，它会在声明时“静态”绑定。没什么大不了的，是吧？

呃……可能，可能不是这样。如果你和大多数JavaScript开发者一样，会用许多不同的方法把函数应用在不同的（使用class定义的）对象上，那你可能不知道，每次执行这些操作时都必须重新绑定super。

此外，根据应用方式的不同，super可能不会绑定到合适的对象（至少和你想的不一样），

所以你可能（写作本书时，TC39 正在讨论这个话题）需要用 toMethod(..) 来手动绑定 super（类似用 bind(..) 来绑定 this——参见第 2 章）。

你已经习惯了把方法应用到不同的对象上，从而可以*自动*利用 this 的*隐式*绑定规则（参见第 2 章）。但是这对于 super 来说是行不通的。

思考下面代码中 super 的行为（D 和 E 上）：

```
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
};

// 把 E 委托到 D
Object.setPrototypeOf( E, D );

E.foo(); // "P.foo"
```

如果你认为 super 会动态绑定（非常合理！），那你可能期望 super() 会自动识别出 E 委托了 D，所以 E.foo() 中的 super() 应该调用 D.foo()。

但事实并不是这样。出于性能考虑，super 并不像 this 一样是*晚绑定*（late bound，或者说动态绑定）的，它在 [[HomeObject]].[[Prototype]] 上，[[HomeObject]] 会在创建时静态绑定。

在本例中，super() 会调用 P.foo()，因为方法的 [[HomeObject]] 仍然是 C，C.[[Prototype]] 是 P。

确实可以手动修改 super 绑定，使用 toMethod(..) 绑定或重新绑定方法的 [[HomeObject]]（就像设置对象的 [[Prototype]] 一样！）就可以解决本例的问题：

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

```
};

// 把 E 委托到 D
var E = Object.create( D );

// 手动把 foo 的 [[HomeObject]] 绑定到 E，E.[[Prototype]] 是 D，所以 super() 是 D.foo()
E.foo = C.prototype.foo.toMethod( E, "foo" );

E.foo(); // "D.foo"
```

`toMethod(..)` 会复制方法并把 `homeObject` 当作第一个参数（也就是我们传入的 `E`），第二个参数（可选）是新方法的名称（默认是原方法名）。

除此之外，开发者还有可能会遇到其他问题，这有待观察。无论如何，对于引擎自动绑定的 `super` 来说，你必须时刻警惕是否需要进行手动绑定。唉！

A.3 静态大于动态吗

通过上面的这些特性可以看出，ES6 的 `class` 最大的问题在于，（像传统的类一样）它的语法有时会让你认为，定义了一个 `class` 后，它就变成了一个（未来会被实例化的）东西的静态定义。你会彻底忽略 `C` 是一个对象，是一个具体的可以直接交互的东西。

在传统面向类的语言中，类定义之后就不会进行修改，所以类的设计模式就不支持修改。但是 JavaScript 最强大的特性之一就是它的动态性，任何对象的定义都可以修改（除非你把它设置成不可变）。

`class` 似乎不赞成这样做，所以强制让你使用丑陋的 `.prototype` 语法以及 `super` 问题，等等。而且对于这种动态产生的问题，`class` 基本上都没有提供解决方案。

换句话说，`class` 似乎想告诉你：“动态太难实现了，所以这可能不是个好主意。这里有一种看起来像静态的语法，所以编写静态代码吧。”

对于 JavaScript 来说这是多么悲伤的评论啊：*动态太难实现了，我们假装成静态吧。（但是实际上并不是！）*

总地来说，ES6 的 `class` 想伪装成一种很好的语法问题的解决方案，但是实际上却让问题更难解决而且让 JavaScript 更加难以理解。

如果你使用 `.bind(..)` 函数来硬绑定函数（参见第 2 章），那么这个函数不会像普通函数那样被 ES6 的 `extend` 扩展到子类中。

A.4 小结

`class` 很好地伪装成 JavaScript 中类和继承设计模式的解决方案，但是它实际上起到了反作用：它隐藏了许多问题并且带来了更多更细小但是危险的问题。

`class` 加深了过去 20 年中对于 JavaScript 中“类”的误解，在某些方面，它产生的问题比解决的多，而且让本来优雅简洁的 `[[Prototype]]` 机制变得非常别扭。

结论：如果 ES6 的 `class` 让 `[[Prototype]]` 变得更加难用而且隐藏了 JavaScript 对象最重要的机制——对象之间的实时委托关联，我们难道不应该认为 `class` 产生的问题比解决的多吗？难道不应该抵制这种设计模式吗？

我无法替你回答这些问题，但是我希望本书能从前所未有的深度分析这些问题，并且能够为你提供回答问题所需的所有信息。