```
$( 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 );
} );
```

毫无疑问，使用 ES6 的 class 之后，上一段代码中许多丑陋的语法都不见了，super(..) 函数棒极了。（尽管深入探究就会发现并不是那么完美！）

尽管语法上得到了改进，但实际上这里并没有真正的类，class 仍然是通过 [[Prototype]] 机制实现的，因此我们仍然面临第 4 章至第 6 章提到的思维模式不匹配问题。附录 A 会详细介绍 ES6 的 class 语法及其实现细节，我们会看到为什么解决语法上的问题无法真正解除对于 JavaScript 中类的误解，尽管它看起来非常像一种解决办法！

无论你使用的是传统的原型语法还是 ES6 中的新语法糖，你仍然需要用“类”的概念来对问题（UI 控件）进行建模。就像前几章试图证明的一样，这种做法会为你带来新的麻烦。

6.2.2 委托控件对象

下面的例子使用对象关联风格委托来更简单地实现 Widget/Button：

```
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){
    // 委托调用
    this.init( width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
};
Button.build = function($where) {
```

```
    // 委托调用
    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 );
} );
```

使用对象关联风格来编写代码时不需要把 `Widget` 和 `Button` 当作父类和子类。相反，`Widget` 只是一个对象，包含一组通用的函数，任何类型的控件都可以委托，`Button` 同样只是一个对。（当然，它会通过委托关联到 `Widget`！）

从设计模式的角度来说，我们并没有像类一样在两个对象中都定义相同的方法名 `render(..)`，相反，我们定义了两个更具描述性的方法名（`insert(..)` 和 `build(..)`）。同理，*初始化*方法分别叫作 `init(..)` 和 `setup(..)`。

在委托设计模式中，除了建议使用不相同并且更具描述性的方法名之外，还要通过对象关联避免丑陋的显式伪多态调用（`Widget.call` 和 `Widget.prototype.render.call`），代之以简单的相对委托调用 `this.init(..)` 和 `this.insert(..)`。

从语法角度来说，我们同样没有使用任何构造函数、`.prototype` 或 `new`，实际上也没必要使用它们。

如果你仔细观察就会发现，之前的一次调用（`var btn1 = new Button(..)`）现在变成了两次（`var btn1 = Object.create(Button)` 和 `btn1.setup(..)`）。乍一看这似乎是一个缺点（需要更多代码）。

但是这一点其实也是对象关联风格代码相比传统原型风格代码有优势的地方。为什么呢？

使用类构造函数的话，你需要（并不是硬性要求，但是强烈建议）在同一个步骤中实现构造和初始化。然而，在许多情况下把这两步分开（就像对象关联代码一样）更灵活。

举例来说，假如你在程序启动时创建了一个实例池，然后一直等到实例被取出并使用时才执行特定的初始化过程。这个过程中两个函数调用是挨着的，但是完全可以根据需要让它

们出现在不同的位置。

对象关联可以更好地支持关注分离（separation of concerns）原则，创建和初始化并不需要合并为一个步骤。

6.3 更简洁的设计

对象关联除了能让代码看起来更简洁（并且更具扩展性）外还可以通过行为委托模式简化代码结构。我们来看最后一个例子，它展示了对象关联如何简化整体设计。

在这个场景中我们有两个控制器对象，一个用来操作网页中的登录表单，另一个用来与服务器进行验证（通信）。

我们需要一个辅助函数来创建 Ajax 通信。我们使用的是 jQuery（尽管其他框架也做得不错），它不仅可以处理 Ajax 并且会返回一个类 Promise 的结果，因此我们可以使用 `.then(..)` 来监听响应。

这里我们不会介绍 Promise，但是在本系列之后的书中会介绍。

在传统的类设计模式中，我们会把基础的函数定义在名为 `Controller` 的类中，然后派生两个子类 `LoginController` 和 `AuthController`，它们都继承自 `Controller` 并且重写了一些基础行为：

```
// 父类
function Controller() {
    this.errors = [];
}
Controller.prototype.showDialog(title,msg) {
    // 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 );
}
// 把子类关联到父类
```

```
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 (user.length < 5) {
        return this.failure(
           "Password must be 5+ characters!"
        );
    }

    // 如果执行到这里说明通过验证
    return true;
};
// 重写基础的 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 );
    // 合成
    this.login = login;
}
// 把子类关联到父类
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 ) );
        }
    };
    // 重写基础的 success()
    AuthController.prototype.success = function() {
        // “super” 调用
        Controller.prototype.success.call( this, "Authenticated!" );
    };
    // 重写基础的 failure()
    AuthController.prototype.failure = function(err) {
        // “super” 调用
        Controller.prototype.failure.call(
            this,
            "Auth Failed: " + err
        );
    };

    var auth = new AuthController();
    auth.checkAuth(
        // 除了继承，我们还需要合成
        new LoginController()
    );
```

所有控制器共享的基础行为是 `success(..)`、`failure(..)` 和 `showDialog(..)`。子类 `LoginController` 和 `AuthController` 通过重写 `failure(..)` 和 `success(..)` 来扩展默认基础类行为。此外，注意 `AuthController` 需要一个 `LoginController` 的实例来和登录表单进行交互，因此这个实例变成了一个数据属性。

另一个需要注意的是我们在继承的基础上进行了一些合成。`AuthController` 需要使用 `LoginController`，因此我们实例化后者（`new LoginController()`）并用一个类成员属性 `this.login` 来引用它，这样 `AuthController` 就可以调用 `LoginController` 的行为。

你可能想让 `AuthController` 继承 `LoginController` 或者相反，这样我们就通过继承链实现了真正的合成。但是这就是类继承在问题领域建模时会产生的问题，因为 `AuthController` 和 `LoginController` 都不具备对方的基础行为，所以这种继承关系是不恰当的。我们的解决办法是进行一些简单的合成从而让它们既不必互相继承又可以互相合作。

如果你熟悉面向类设计，你一定会觉得以上内容非常亲切和自然。

反类

但是，我们真的需要用一个 Controller 父类、两个子类加上合成来对这个问题进行建模吗？能不能使用对象关联风格的行为委托来实现更简单的设计呢？当然可以！

```
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 (user.length < 5) {
            return this.failure(
               "Password must be 5+ characters!"
            );
        }

        // 如果执行到这里说明通过验证
        return true;
    },
    showDialog: function(title,msg) {
        // 给用户显示标题和消息
    },
    failure: function(err) {
        this.errors.push( err );
        this.showDialog( "Error", "Login invalid: " + err );
    }
};
// 让 AuthController 委托 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 );
};
```

由于 AuthController 只是一个对象（LoginController 也一样），因此我们不需要实例化（比如 new AuthController()），只需要一行代码就行：

```
AuthController.checkAuth();
```

借助对象关联，你可以简单地向委托链上添加一个或多个对象，而且同样不需要实例化：

```
var controller1 = Object.create( AuthController );
var controller2 = Object.create( AuthController );
```

在行为委托模式中，AuthController 和 LoginController 只是对象，它们之间是*兄弟关系*，并不是父类和子类的关系。代码中 AuthController 委托了 LoginController，反向委托也完全没问题。

这种模式的重点在于只需要两个实体（LoginController 和 AuthController），而之前的模式需要三个。

我们不需要 Controller 基类来“共享”两个实体之间的行为，因为委托足以满足我们需要的功能。同样，前面提到过，我们也不需要实例化类，因为它们根本就不是类，它们只是对象。此外，我们也不需要合成，因为两个对象可以通过委托进行合作。

最后，我们避免了面向类设计模式中的多态。我们在不同的对象中没有使用相同的函数名 success(..) 和 failure(..)，这样就不需要使用丑陋的显示伪多态。相反，在 AuthController 中它们的名字是 accepted(..) 和 rejected(..)——可以更好地描述它们的行为。

总结：我们用一种（极其）简单的设计实现了同样的功能，这就是对象关联风格代码和*行为委托*设计模式的力量。

6.4 更好的语法

ES6 的 `class` 语法可以简洁地定义类方法，这个特性让 `class` 乍看起来更有吸引力（附录 A 会介绍为什么要避免使用这个特性）：

```
class Foo {
    methodName() { /* .. */ }
}
```

我们终于可以抛弃定义中的关键字 `function` 了，对所有 JavaScript 开发者来说真是大快人心！

你可能注意到了，在之前推荐的对象关联语法中出现了许多 `function`，看起来违背了对象关联的简洁性。但是实际上大可不必如此！

在 ES6 中我们可以在任意对象的字面形式中使用*简洁方法声明*（concise method declaration），所以对象关联风格的对象可以这样声明（和 `class` 的语法糖一样）：

```
var LoginController = {
    errors: [],
    getUser() { // 妈妈再也不用担心代码里有 function 了！
        // ...
    },
    getPassword() {
        // ...
    }
    // ...
};
```

唯一的区别是对象的字面形式仍然需要使用“,”来分隔元素，而 `class` 语法不需要。这个区别对于整体的设计来说无关紧要。

此外，在 ES6 中，你可以使用对象的字面形式（这样就可以使用简洁方法定义）来改写之前繁琐的属性赋值语法（比如 `AuthController` 的定义），然后用 `Object.setPrototypeOf(..)` 来修改它的 `[[Prototype]]`：

```
// 使用更好的对象字面形式语法和简洁方法
var AuthController = {
    errors: [],
    checkAuth() {
        // ...
    },
    server(url,data) {
        // ...
    }
    // ...
};

// 现在把 AuthController 关联到 LoginController
Object.setPrototypeOf( AuthController, LoginController );
```

使用 ES6 的简洁方法可以让对象关联风格更加人性化（并且仍然比典型的原型风格代码更加简洁和优秀）。你完全不需要使用类就能享受整洁的对象语法！

反词法

简洁方法有一个非常小但是非常重要的缺点。思考下面的代码：

```
var Foo = {
    bar() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

去掉语法糖之后的代码如下所示：

```
var Foo = {
    bar: function() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

看到区别了吗？由于函数对象本身没有名称标识符，所以 `bar()` 的缩写形式（`function()..`）实际上会变成一个*匿名函数表达式*并赋值给 `bar` 属性。相比之下，*具名函数表达式*（`function baz()..`）会额外给 `.baz` 属性附加一个词法名称标识符 `baz`。

然后呢？在本书第一部分“作用域和闭包”中我们分析了匿名函数表达式的三大主要缺点，下面我们会简单介绍一下这三个缺点，然后和简洁方法定义进行对比。

匿名函数没有 `name` 标识符，这会导致：

1. 调试栈更难追踪；
2. 自我引用（递归、事件（解除）绑定，等等）更难；
3. 代码（稍微）更难理解。

简洁方法没有第 1 和第 3 个缺点。

去掉语法糖的版本使用的是*匿名函数表达式*，通常来说并不会在追踪栈中添加 `name`，但是简洁方法很特殊，会给对应的函数对象设置一个内部的 `name` 属性，这样理论上可以用在追踪栈中。（但是追踪的具体实现是不同的，因此无法保证可以使用。）

很不幸，简洁方法无法避免第 2 个缺点，它们不具备可以自我引用的词法标识符。思考下面的代码：

```
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;
        }
    };
```

在本例中使用 `Foo.bar(x*2)` 就足够了，但是在许多情况下无法使用这种方法，比如多个对象通过代理共享函数、使用 `this` 绑定，等等。这种情况下最好的办法就是使用函数对象的 `name` 标识符来进行真正的自我引用。

使用简洁方法时一定要小心这一点。如果你需要自我引用的话，那最好使用传统的*具名函数表达式*来定义对应的函数（`· baz: function baz(){..}·`），不要使用简洁方法。

6.5 内省

如果你写过许多面向类的程序（无论是使用 JavaScript 还是其他语言），那你可能很熟悉*自省*。自省就是检查实例的类型。类实例的自省主要目的是通过创建方式来判断对象的结构和功能。

下面的代码使用 `instanceof`（参见第 5 章）来推测对象 `a1` 的功能：

```
function Foo() {
    // ...
}
Foo.prototype.something = function(){
    // ...
}

var a1 = new Foo();

// 之后

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

因为 `Foo.prototype`（不是 `Foo`！）在 `a1` 的 `[[Prototype]]` 链上（参见第 5 章），所以 `instanceof` 操作（会令人困惑地）告诉我们 `a1` 是 `Foo`“类”的一个实例。知道了这点后，我们就可以认为 `a1` 有 `Foo`“类”描述的功能。

当然，`Foo` 类并不存在，只有一个普通的函数 `Foo`，它引用了 `a1` 委托的对象（`Foo.prototype`）。从语法角度来说，`instanceof` 似乎是检查 `a1` 和 `Foo` 的关系，但是实际上它想说的是 `a1` 和 `Foo.prototype`（引用的对象）是互相关联的。