```
class ABC inherits Task {
    // ...
}
```

现在你可以实例化子类 XYZ 的一些副本然后使用这些实例来执行任务“XYZ”。这些实例会复制 Task 定义的通用行为以及 XYZ 定义的特殊行为。同理，ABC 类的实例也会复制 Task 的行为和 ABC 的行为。在构造完成后，你通常只需要操作这些实例（而不是类），因为每个实例都有你需要完成任务的所有行为。

## 6.1.2 委托理论

但是现在我们试着来使用*委托行为*而不是类来思考同样的问题。

首先你会定义一个名为 Task 的*对象*（和许多 JavaScript 开发者告诉你的不同，它既不是类也不是函数），它会包含所有任务都可以使用（写作使用，读作委托）的具体行为。接着，对于每个任务（“XYZ”、“ABC”）你都会定义一个对象来存储对应的数据和行为。你会把特定的任务对象都关联到 Task 功能对象上，让它们在需要的时候可以进行委托。

基本上你可以想象成，执行任务“XYZ”需要两个兄弟对象（XYZ 和 Task）协作完成。但是我们并不需要把这些行为放在一起，通过类的复制，我们可以把它们分别放在各自独立的对象中，需要时可以允许 XYZ 对象委托给 Task。

下面是推荐的代码形式，非常简单：

```
Task = {
    setID: function(ID) { this.id = ID; },
    outputID: function() { console.log( this.id ); }
};

// 让 XYZ 委托 Task
XYZ = Object.create( Task );

XYZ.prepareTask = function(ID,Label) {
    this.setID( ID );
    this.label = Label;
};

XYZ.outputTaskDetails = function() {
    this.outputID();
    console.log( this.label );
};

// ABC = Object.create( Task );
// ABC ... = ...
```

在这段代码中，Task 和 XYZ 并不是类（或者函数），它们是对象。XYZ 通过 Object.create(..) 创建，它的 [[Prototype]] 委托了 Task 对象（参见第 5 章）。

相比于面向类（或者说面向对象），我会把这种编码风格称为“对象关联”（OLOO，objects linked to other objects）。我们真正关心的只是 XYZ 对象（和 ABC 对象）委托了 Task 对象。

在 JavaScript 中，[[Prototype]] 机制会把对象关联到其他对象。无论你多么努力地说服自己，JavaScript 中就是没有类似“类”的抽象机制。这有点像逆流而上：你确实可以这么做，但是如果你选择对抗事实，那要达到目的就显然会更加困难。

对象关联风格的代码还有一些不同之处。

1. 在上面的代码中，id 和 label 数据成员都是直接存储在 XYZ 上（而不是 Task）。通常来说，在 [[Prototype]] 委托中最好把状态保存在委托者（XYZ、ABC）而不是委托目标（Task）上。

2. 在类设计模式中，我们故意让父类（Task）和子类（XYZ）中都有 outputTask 方法，这样就可以利用重写（多态）的优势。在委托行为中则恰好相反：我们会尽量避免在 [[Prototype]] 链的不同级别中使用相同的命名，否则就需要使用笨拙并且脆弱的语法来消除引用歧义（参见第 4 章）。

   这个设计模式要求尽量少使用容易被重写的通用方法名，提倡使用更有描述性的方法名，尤其是要写清相应对象行为的类型。这样做实际上可以创建出更容易理解和维护的代码，因为方法名（不仅在定义的位置，而是贯穿整个代码）更加清晰（自文档）。

3. this.setID(ID)；XYZ 中的方法首先会寻找 XYZ 自身是否有 setID(..)，但是 XYZ 中并没有这个方法名，因此会通过 [[Prototype]] 委托关联到 Task 继续寻找，这时就可以找到 setID(..) 方法。此外，由于调用位置触发了 this 的隐式绑定规则（参见第 2 章），因此虽然 setID(..) 方法在 Task 中，运行时 this 仍然会绑定到 XYZ，这正是我们想要的。在之后的代码中我们还会看到 this.outputID()，原理相同。

   换句话说，我们和 XYZ 进行交互时可以使用 Task 中的通用方法，因为 XYZ 委托了 Task。

委托行为意味着某些对象（XYZ）在找不到属性或者方法引用时会把这个请求委托给另一个对象（Task）。

这是一种极其强大的设计模式，和父类、子类、继承、多态等概念完全不同。在你的脑海中对象并不是按照父类到子类的关系垂直组织的，而是通过任意方向的委托关联并排组织的。

在 API 接口的设计中，委托最好在内部实现，不要直接暴露出去。在之前的例子中我们并没有让开发者通过 API 直接调用 XYZ.setID()。（当然，可以这么做！）相反，我们把委托隐藏在了 API 的内部，XYZ.prepareTask(..) 会委托 Task.setID(..)。更多细节参见 5.4.2 节。

### 1. 互相委托（禁止）

你无法在两个或两个以上互相（双向）委托的对象之间创建循环委托。如果你把 B 关联到 A 然后试着把 A 关联到 B，就会出错。

很遗憾（并不是非常出乎意料，但是有点烦人）这种方法是被禁止的。如果你引用了一个两边都不存在的属性或者方法，那就会在 [[Prototype]] 链上产生一个无限递归的循环。但是如果所有的引用都被严格限制的话，B 是可以委托 A 的，反之亦然。因此，互相委托理论上是可以正常工作的，在某些情况下这是非常有用的。

之所以要禁止互相委托，是因为引擎的开发者们发现在设置时检查（并禁止！）一次无限循环引用要更加高效，否则每次从对象中查找属性时都需要进行检查。

### 2. 调试

我们来简单介绍一个容易让开发者感到迷惑的细节。通常来说，JavaScript 规范并不会控制浏览器中开发者工具对于特定值或者结构的表示方式，浏览器和引擎可以自己选择合适的方式来进行解析，因此浏览器和工具的解析结果并不一定相同。比如，下面这段代码的结果只能在 Chrome 的开发者工具中才能看到。

这段传统的“类构造函数”JavaScript 代码在 Chrome 开发者工具的控制台中结果如下所示：

```
function Foo() {}

var a1 = new Foo();

a1; // Foo {}
```

我们看代码的最后一行：表达式 a1 的输出是 Foo {}。如果你在 Firefox 中运行同样的代码会得到 Object {}。为什么会这样呢？这些输出是什么意思呢？

Chrome 实际上想说的是“{} 是一个空对象，由名为 Foo 的函数构造”。Firefox 想说的是“{} 是一个空对象，由 Object 构造”。之所以有这种细微的差别，是因为 Chrome 会动态跟踪并把实际执行构造过程的函数名当作一个内置属性，但是其他浏览器并不会跟踪这些额外的信息。

看起来可以用 JavaScript 的机制来解释 Chrome 的跟踪原理：

```
function Foo() {}

var a1 = new Foo();

a1.constructor; // Foo(){}
a1.constructor.name; // "Foo"
```

Chrome 是不是直接输出了对象的 .constructor.name 呢？令人迷惑的是，答案是“既是又不是”。

思考下面的代码：

```
function Foo() {}

var a1 = new Foo();

Foo.prototype.constructor = function Gotcha(){};

a1.constructor; // Gotcha(){}
a1.constructor.name; // "Gotcha"

a1; // Foo {}
```

即使我们把 a1.constructor.name 修改为另一个合理的值（Gotcha），Chrome 控制台仍然会输出 Foo。

看起来之前那个问题（是否使用 .constructor.name？）的答案是“不是”；Chrome 在内部肯定是通过另一种方式进行跟踪。

别着急！我们先看看下面这段代码：

```
var Foo = {};

var a1 = Object.create( Foo );

a1; // Object {}

Object.defineProperty( Foo, "constructor", {
    enumerable: false,
    value: function Gotcha(){}
});

a1; // Gotcha {}
```

啊哈！抓到你了（Gotcha 的意思就是抓到你了）！本例中 Chrome 的控制台*确实*使用了 .constructor.name。实际上，在编写本书时，这个行为被认定是 Chrome 的一个 bug，当你读到此书时，它可能已经被修复了。所以你看到的可能是 a1; // Object {}。

除了这个 bug，Chrome 内部跟踪（只用于调试输出）“构造函数名称”的方法是 Chrome 自身的一种扩展行为，并不包含在 JavaScript 的规范中。

如果你并不是使用“构造函数”来生成对象，比如使用本章介绍的对象关联风格来编写代码，那 Chrome 就*无法*跟踪对象内部的“构造函数名称”，这样的对象输出是 Object {}，意思是“Object() 构造出的对象”。

当然，这并不是对象关联风格代码的缺点。当你使用对象关联风格来编写代码并使用行为委托设计模式时，并不需要关注是谁“构造了”对象（就是使用 new 调用的那个*函数*）。只有使用类风格来编写代码时 Chrome 内部的“构造函数名称”跟踪才有意义，使用对象关联时这个功能不起任何作用。

### 6.1.3 比较思维模型

现在你已经明白了“类”和“委托”这两种设计模式的理论区别，接下来我们看看它们在思维模型方面的区别。

我们会通过一些示例（Foo、Bar）代码来比较一下两种设计模式（面向对象和对象关联）具体的实现方法。下面是典型的（“原型”）面向对象风格：

```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call( this, who );
}
Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

子类 Bar 继承了父类 Foo，然后生成了 b1 和 b2 两个实例。b1 委托了 Bar.prototype，后者委托了 Foo.prototype。这种风格很常见，你应该很熟悉了。

下面我们看看如何使用对象关联风格来编写功能完全相同的代码：

```
Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};
Bar = Object.create( Foo );

Bar.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = Object.create( Bar );
b1.init( "b1" );
var b2 = Object.create( Bar );
```

```
b2.init( "b2" );

b1.speak();
b2.speak();
```

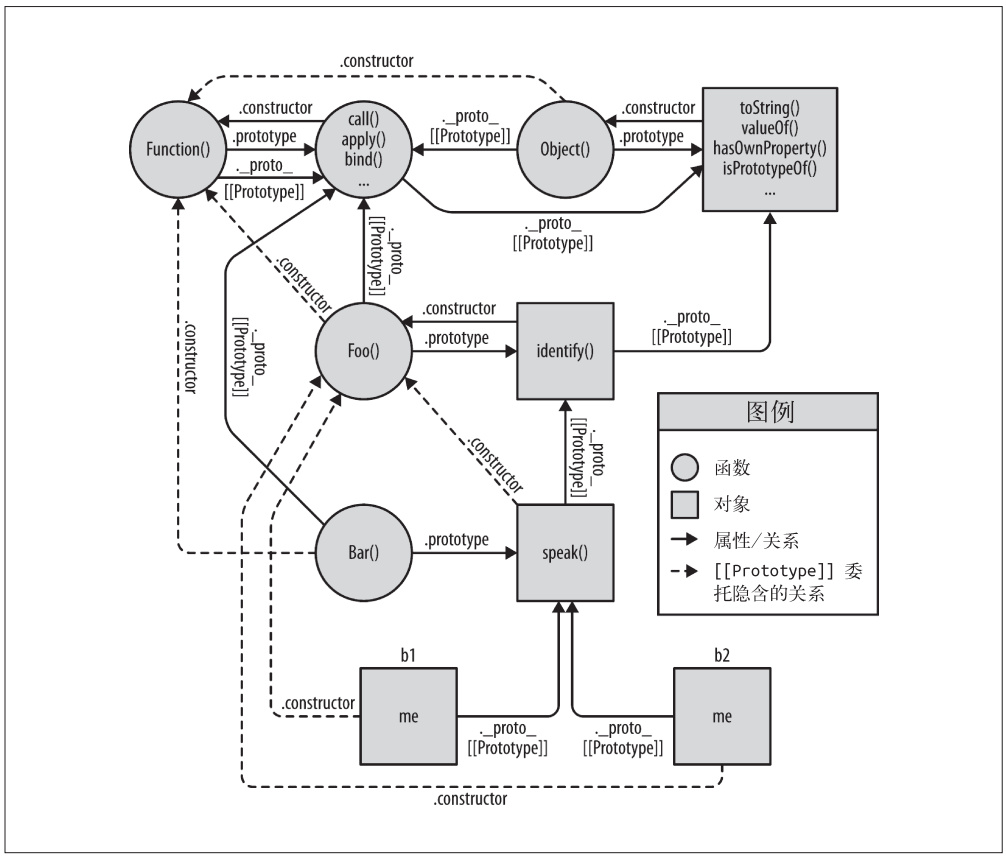
这段代码中我们同样利用 [[Prototype]] 把 b1 委托给 Bar 并把 Bar 委托给 Foo，和上一段代码一模一样。我们仍然实现了三个对象之间的关联。

但是非常重要的一点是，这段代码简洁了许多，我们只是把对象关联起来，并不需要那些既复杂又令人困惑的模仿类的行为（构造函数、原型以及 new）。

问问你自己：如果对象关联风格的代码能够实现类风格代码的所有功能并且更加简洁易懂，那它是不是比类风格更好？

下面我们看看两段代码对应的思维模型。

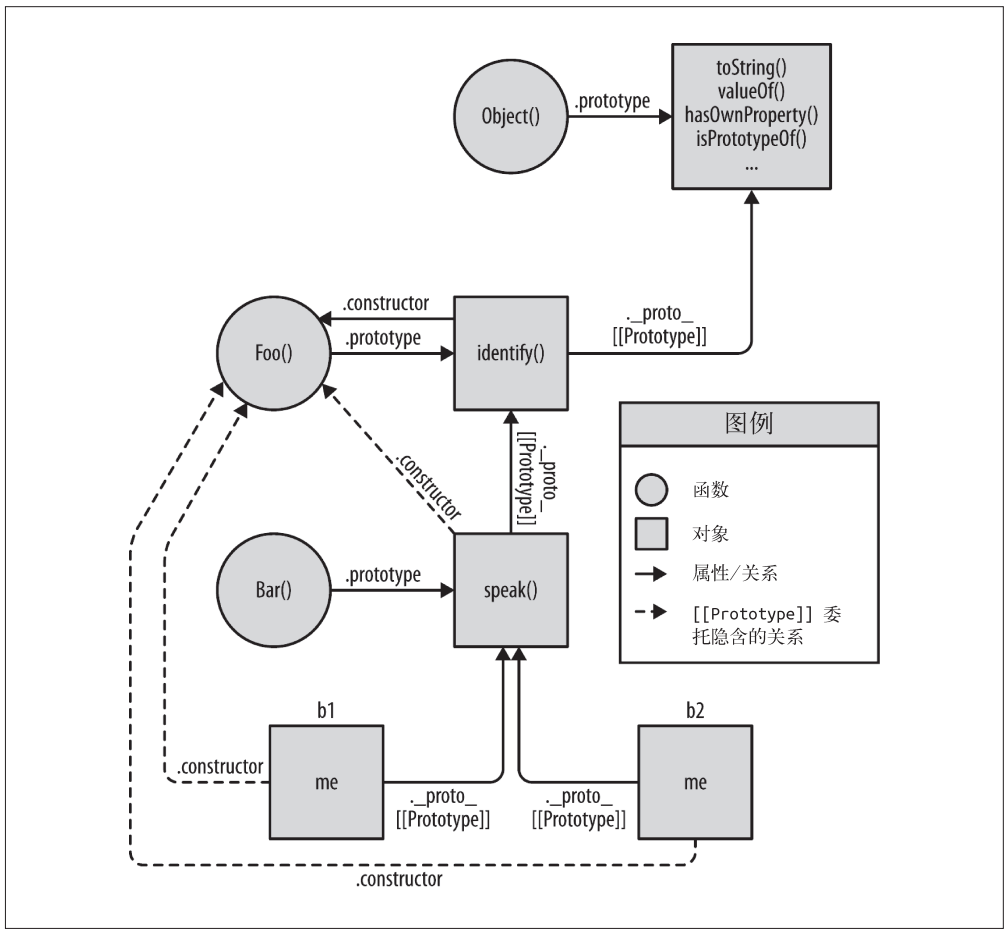
首先，类风格代码的思维模型强调实体以及实体间的关系：



实际上这张图有点不清晰 / 误导人，因为它还展示了许多技术角度不需要关注的细节（但是你必须理解它们）！从图中可以看出这是一张十分复杂的关系网。此外，如果你跟着图中的箭头走就会发现，JavaScript 机制有很强的内部连贯性。

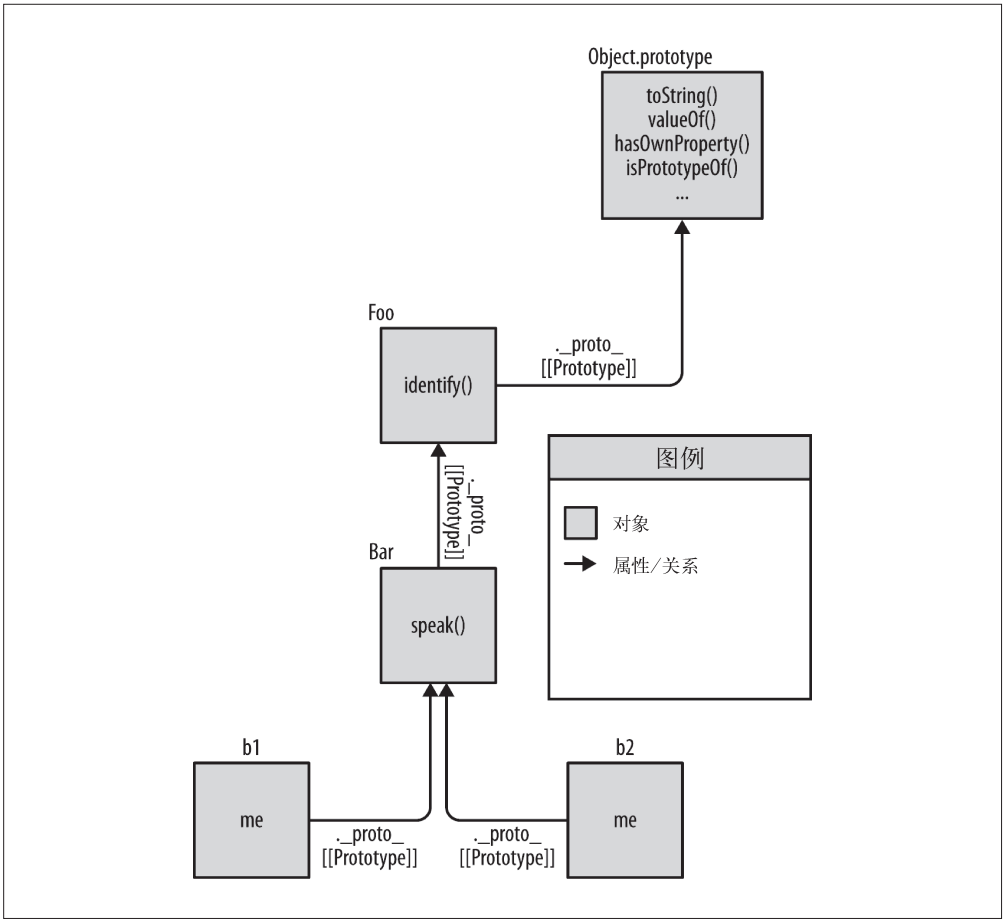
举例来说，JavaScript 中的函数之所以可以访问 `call(..)`、`apply(..)` 和 `bind(..)`（参见第 2 章），就是因为函数本身是对象。而函数对象同样有 `[[Prototype]]` 属性并且关联到 `Function.prototype` 对象，因此所有函数对象都可以通过委托调用这些默认方法。JavaScript 能做到这一点，你也可以！

好，下面我们来看一张简化版的图，它更“清晰”一些——只展示了必要的对象和关系：




仍然很复杂，是吧？虚线表示的是 `Bar.prototype` 继承 `Foo.prototype` 之后丢失的 `.constructor` 属性引用（参见 5.2.3 节的“回顾‘构造函数’”部分），它们还没有被修复。即使移除这些虚线，这个思维模型在你处理对象关联时仍然非常复杂。

现在我们看看对象关联风格代码的思维模型：




通过比较可以看出，对象关联风格的代码显然更加简洁，因为这种代码只关注一件事：对象之间的关联关系。

其他的“类”技巧都是非常复杂并且令人困惑的。去掉它们之后，事情会变得简单许多（同时保留所有功能）。

## 6.2 类与对象

我们已经看到了“类”和“行为委托”在理论和思维模型方面的区别，现在看看在真实场景中如何应用这些方法。

首先看看 Web 开发中非常典型的一种前端场景：创建 UI 控件（按钮、下拉列表，等等）。

### 6.2.1 控件“类”

你可能已经习惯了面向对象设计模式，所以很快会想到一个包含所有通用控件行为的父类（可能叫作 `Widget`）和继承父类的特殊控件子类（比如 `Button`）。

这里将使用 jQuery 来操作 DOM 和 CSS，因为这些操作和我们现在讨论的内容没有关系。这些代码并不关注你是否使用，或使用哪种 JavaScript 框架（jQuery、Dojo、YUI，等等）来解决问题。

下面这段代码展示的是如何在不使用任何“类”辅助库或者语法的情况下，使用纯 JavaScript 实现类风格的代码：

```
// 父类
function Widget(width,height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
}

Widget.prototype.render = function($where){
    if (this.$elem) {
        this.$elem.css( {
            width: this.width + "px",
            height: this.height + "px"
        } ).appendTo( $where );
    }
};

// 子类
function Button(width,height,label) {
    // 调用“super”构造函数
    Widget.call( this, width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
}

// 让 Button“继承”Widget
Button.prototype = Object.create( Widget.prototype );

// 重写 render(..)
Button.prototype.render = function($where) {
    // “super”调用
    Widget.prototype.render.call( this, $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.prototype.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
```

```
};

$( 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 );
} );
```

在面向对象设计模式中我们需要先在父类中定义基础的 `render(..)`，然后在子类中重写它。子类并不会替换基础的 `render(..)`，只是添加一些按钮特有的行为。

可以看到代码中出现了丑陋的显式伪多态（参见第 4 章），即通过 `Widget.call` 和 `Widget.prototype.render.call` 从“子类”方法中引用“父类”中的基础方法。呸！

### ES6的class语法糖

附录 A 会详细介绍 ES6 的 `class` 语法糖，不过这里可以简单介绍一下如何使用 `class` 来实现相同的功能：

```
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!" );
    }
}
```