第4章

混合对象“类”

上一章介绍了对象，这章自然要介绍和类相关的面向对象编程。在研究类的具体机制之前，我们首先会介绍面向类的设计模式：实例化（instantiation）、继承（inheritance）和（相对）多态（polymorphism）。

我们将会看到，这些概念实际上无法直接对应到 JavaScript 的对象机制，因此我们会介绍许多 JavaScript 开发者所使用的解决方法（比如混入，mixin）。

本章用很大的篇幅（整整半章）介绍面向对象编程理论。在后半章介绍混入时会把这些概念落实到 JavaScript 代码上。但是首先我们会看到许多概念和伪代码，因此千万不要迷路——跟紧了！

4.1 类理论

类 / 继承描述了一种代码的组织结构形式——一种在软件中对真实世界中问题领域的建模方法。

面向对象编程强调的是数据和操作数据的行为本质上是互相关联的（当然，不同的数据有不同的行为），因此好的设计就是把数据以及和它相关的行为打包（或者说封装）起来。这在正式的计算机科学中有时被称为数据结构。

举例来说，用来表示一个单词或者短语的一串字符通常被称为字符串。字符就是数据。但是你关心的往往不是数据是什么，而是可以对数据做什么，所以可以应用在这种数据上的行为（计算长度、添加数据、搜索，等等）都被设计成 `String` 类的方法。

所有字符串都是 String 类的一个实例，也就是说它是一个包裹，包含字符数据和我们可以应用在数据上的函数。

我们还可以使用类对数据结构进行分类，可以把任意数据结构看作范围更广的定义的一种特例。

我们来看一个常见的例子，“汽车”可以被看作“交通工具”的一种特例，后者是更广泛的类。

我们可以在软件中定义一个 Vehicle 类和一个 Car 类来对这种关系进行建模。

Vehicle 的定义可能包含推进器（比如引擎）、载人能力等等，这些都是 Vehicle 的行为。我们在 Vehicle 中定义的是（几乎）所有类型的交通工具（飞机、火车和汽车）都包含的东西。

在我们的软件中，对不同的交通工具重复定义“载人能力”是没有意义的。相反，我们只在 Vehicle 中定义一次，定义 Car 时，只要声明它继承（或者扩展）了 Vehicle 的这个基础定义就行。Car 的定义就是对通用 Vehicle 定义的特殊化。

虽然 Vehicle 和 Car 会定义相同的方法，但是实例中的数据可能是不同的，比如每辆车独一无二的 VIN（Vehicle Identification Number，车辆识别号码），等等。

这就是类、继承和实例化。

类的另一个核心概念是*多态*，这个概念是说父类的通用行为可以被子类用更特殊的行为重写。实际上，相对多态性允许我们从重写行为中引用基础行为。

类理论强烈建议父类和子类使用相同的方法名来表示特定的行为，从而让子类重写父类。我们之后会看到，在 JavaScript 代码中这样做会降低代码的可读性和健壮性。

4.1.1 “类”设计模式

你可能从来没把类作为设计模式来看待，讨论得最多的是面向对象设计模式，比如*迭代器*模式、*观察者*模式、*工厂*模式、*单例*模式，等等。从这个角度来说，我们似乎是在（低级）面向对象类的基础上实现了*所有*（高级）设计模式，似乎面向对象是优秀代码的基础。

如果你之前接受过正规的编程教育的话，可能听说过*过程化编程*，这种代码只包含过程（函数）调用，没有高层的抽象。或许老师还教过你最好使用类把过程化风格的“意大利面代码”转换成结构清晰、组织良好的代码。

当然，如果你有*函数式编程*（比如 Monad）的经验就会知道类也是非常常用的一种设计模式。但是对于其他人来说，这可能是第一次知道类并不是必须的编程基础，而是一种可选的代码抽象。

有些语言（比如 Java）并不会给你选择的机会，类并不是可选的——万物皆是类。其他语言（比如 C/C++ 或者 PHP）会提供过程化和面向类这两种语法，开发者可以选择其中一种风格或者混用两种风格。

4.1.2 JavaScript中的“类”

JavaScript 属于哪一类呢？在相当长的一段时间里，JavaScript 只有一些近似类的语法元素（比如 `new` 和 `instanceof`），不过在后来的 ES6 中新增了一些元素，比如 `class` 关键字（参见附录 A）。

这是不是意味着 JavaScript 中实际上有类呢？简单来说：不是。

由于类是一种设计模式，所以你可以用一些方法（本章之后会介绍）近似实现类的功能。为了满足对于类设计模式的最普遍需求，JavaScript 提供了一些近似类的语法。

虽然有近似类的语法，但是 JavaScript 的机制似乎一直在阻止你使用类设计模式。在近似类的表象之下，JavaScript 的机制其实和类完全不同。语法糖和（广泛使用的）JavaScript“类”库试图掩盖这个现实，但是你迟早会面对它：其他语言中的类和 JavaScript 中的“类”并不一样。

总结一下，在软件设计中类是一种可选的模式，你需要自己决定是否在 JavaScript 中使用它。由于许多开发者都非常喜欢面向类的软件设计，我们会在本章的剩余部分中介绍如何在 JavaScript 中实现类以及存在的一些问题。

4.2 类的机制

在许多面向类的语言中，“标准库”会提供 `Stack` 类，它是一种“栈”数据结构（支持压入、弹出，等等）。`Stack` 类内部会有一些变量来存储数据，同时会提供一些公有的可访问行为（“方法”），从而让你的代码可以和（隐藏的）数据进行交互（比如添加、删除数据）。

但是在这些语言中，你实际上并不是直接操作 `Stack`（除非创建一个静态类成员引用，这超出了我们的讨论范围）。`Stack` 类仅仅是一个抽象的表示，它描述了所有“栈”需要做的事，但是它本身并不是一个“栈”。你必须先实例化 `Stack` 类然后才能对它进行操作。

4.2.1 建造

“类”和“实例”的概念来源于房屋建造。

建筑师会规划出一个建筑的所有特性：多宽、多高、多少个窗户以及窗户的位置，甚至连建造墙和房顶需要的材料都要计划好。在这个阶段他并不需要关心建筑会被建在哪，也不需要关心会建造多少个这样的建筑。

建筑师也不太关心建筑里的内容——家具、壁纸、吊扇等——他只关心需要用什么结构来容纳它们。

建筑蓝图只是建筑*计划*，它们并不是真正的建筑，我们还需要一个建筑工人来建造建筑。建筑工人会按照蓝图*建造*建筑。实际上，他会把规划好的特性从蓝图中*复制*到现实世界的建筑中。

完成后，建筑就成为了蓝图的物理实例，本质上就是对蓝图的复制。之后建筑工人就可以到下一个地方，把所有工作都重复一遍，再创建一份副本。

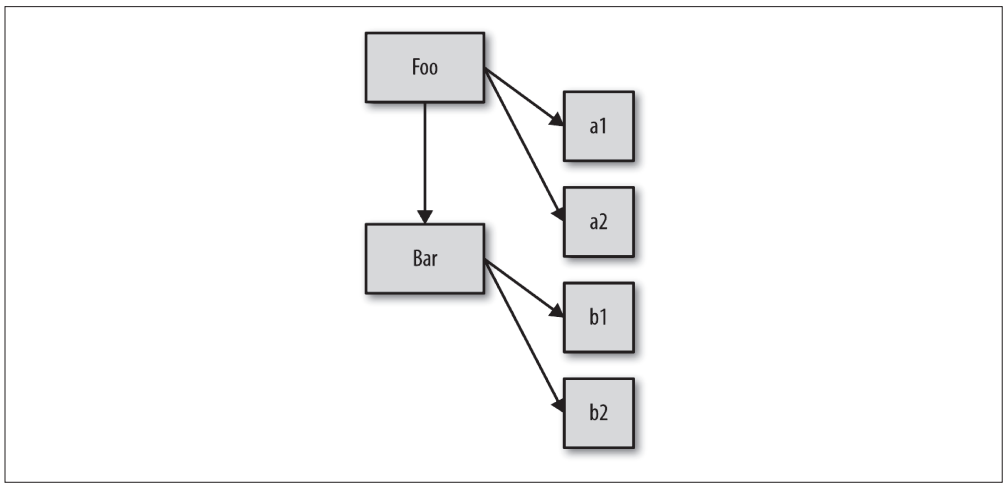
建筑和蓝图之间的关系是间接的。你可以通过蓝图了解建筑的结构，只观察建筑本身是无法获得这些信息的。但是如果你想打开一扇门，那就必须接触真实的建筑才行——蓝图只能*表示*门应该在哪，但并不是真正的门。

一个类就是一张蓝图。为了获得真正可以交互的对象，我们必须按照类来建造（也可以说*实例化*）一个东西，这个东西通常被称为实例，有需要的话，我们可以直接在实例上调用方法并访问其所有公有数据属性。

这个对象就是类中描述的所有特性的一份副本。

你走进一栋建筑时，它的蓝图不太可能挂在墙上（尽管这个蓝图可能会保存在公共档案馆中）。类似地，你通常也不会使用一个实例对象来直接访问并操作它的类，不过至少可以判断出这个实例对象来自哪个类。

把类和实例对象之间的关系看作是直接关系而不是间接关系通常更有助于理解。类通过复制操作被实例化为对象形式：



如你所见，箭头的方向是从左向右、从上向下，它表示概念和物理意义上发生的复制操作。

4.2.2 构造函数

类实例是由一个特殊的类方法构造的，这个方法名通常和类名相同，被称为构造函数。这个方法的任务就是初始化实例需要的所有信息（状态）。

举例来说，思考下面这个关于类的伪代码（编造出来的语法）：

```
class CoolGuy {
    specialTrick = nothing

    CoolGuy( trick ) {
        specialTrick = trick
    }

    showOff() {
        output( "Here's my trick: ", specialTrick )
    }
}
```

我们可以调用类构造函数来生成一个 CoolGuy 实例：

```
Joe = new CoolGuy( "jumping rope" )

Joe.showOff() // 这是我的绝技：跳绳
```

注意，CoolGuy 类有一个 CoolGuy() 构造函数，执行 new CoolGuy() 时实际上调用的就是它。构造函数会返回一个对象（也就是类的一个实例），之后我们可以在这个对象上调用 showOff() 方法，来输出指定 CoolGuy 的特长。

显然，跳绳让乔成为了一个非常酷的家伙。

类构造函数属于类，而且通常和类同名。此外，构造函数大多需要用 new 来调，这样语言引擎才知道你想要构造一个新的类实例。

4.3 类的继承

在面向类的语言中，你可以先定义一个类，然后定义一个继承前者的类。

后者通常被称为“子类”，前者通常被称为“父类”。这些术语显然是类比父母和孩子，不过在意思上稍有扩展，你很快就会看到。

对于父母的亲生孩子来说，父母的基因特性会被复制给孩子。显然，在大多数生物的繁殖系统中，双亲都会贡献等量的基因给孩子。但是在编程语言中，我们假设只有一个父类。

一旦孩子出生，他们就变成了单独的个体。虽然孩子会从父母继承许多特性，但是他是一个独一无二的存在。如果孩子的头发是红色，父母的头发未必是红的，也不会随之变红，

二者之间没有直接的联系。

同理，定义好一个子类之后，相对于父类来说它就是一个独立并且完全不同的类。子类会包含父类行为的原始副本，但是也可以重写所有继承的行为甚至定义新行为。

非常重要的一点是，我们讨论的父类和子类并不是实例。父类和子类的比喻容易造成一些误解，实际上我们应当把父类和子类称为父类 DNA 和子类 DNA。我们需要根据这些 DNA 来创建（或者说实例化）一个人，然后才能和他进行沟通。

好了，我们先抛开现实中的父母和孩子，来看一个稍有不同的例子：不同类型的交通工具。这是一个非常典型（并且经常被抱怨）的讲解继承的例子。

首先回顾一下本章前面部分提出的 `Vehicle` 和 `Car` 类。思考下面关于类继承的伪代码：

```
class Vehicle {
    engines = 1

    ignition() {
        output( "Turning on my engine." );
    }

    drive() {
        ignition();
        output( "Steering and moving forward!" )
    }
}

class Car inherits Vehicle {
    wheels = 4

    drive() {
        inherited:drive()
        output( "Rolling on all ", wheels, " wheels!" )
    }
}

class SpeedBoat inherits Vehicle {
    engines = 2

    ignition() {
        output( "Turning on my ", engines, " engines." )
    }

    pilot() {
        inherited:drive()
        output( "Speeding through the water with ease!" )
    }
}
```

为了方便理解并缩短代码，我们省略了这些类的构造函数。

我们通过定义 Vehicle 类来假设一种发动机，一种点火方式，一种驾驶方法。但是你不可能制造一个通用的“交通工具”，因为这个类只是一个抽象的概念。

接下来我们定义了两类具体的交通工具：Car 和 SpeedBoat。它们都从 Vehicle 继承了通用的特性并根据自身类别修改了某些特性。汽车需要四个轮子，快艇需要两个发动机，因此它必须启动两个发动机的点火装置。

4.3.1 多态

Car 重写了继承自父类的 drive() 方法，但是之后 Car 调用了 inherited:drive() 方法，这表明 Car 可以引用继承来的原始 drive() 方法。快艇的 pilot() 方法同样引用了原始 drive() 方法。

这个技术被称为*多态*或者*虚拟多态*。在本例中，更恰当的说法是*相对多态*。

多态是一个非常广泛的话题，我们现在所说的“相对”只是多态的一个方面：任何方法都可以引用继承层次中高层的方法（无论高层的方法名和当前方法名是否相同）。之所以说“相对”是因为我们并不会定义想要访问的绝对继承层次（或者说类），而是使用相对引用“查找上一层”。

在许多语言中可以使用 super 来代替本例中的 inherited:，它的含义是“超类”（superclass），表示当前类的父类 / 祖先类。

多态的另一个方面是，在继承链的不同层次中一个方法名可以被多次定义，当调用方法时会自动选择合适的定义。

在之前的代码中就有两个这样的例子：drive() 被定义在 Vehicle 和 Car 中，ignition() 被定义在 Vehicle 和 SpeedBoat 中。

在传统的面向类的语言中 super 还有一个功能，就是从子类的构造函数中通过 super 可以直接调用父类的构造函数。通常来说这没什么问题，因为对于真正的类来说，构造函数是属于类的。然而，在 JavaScript 中恰好相反——实际上“类”是属于构造函数的（类似 Foo.prototype... 这样的类型引用）。由于 JavaScript 中父类和子类的关系只存在于两者构造函数对应的 .prototype 对象中，因此它们的构造函数之间并不存在直接联系，从而无法简单地实现两者的相对引用（在 ES6 的类中可以通过 super 来“解决”这个问题，参见附录 A）。

我们可以在 ignition() 中看到多态非常有趣的一点。在 pilot() 中通过相对多态引用了（继承来的）Vehicle 中的 drive()。但是那个 drive() 方法直接通过名字（而不是相对引用）引用了 ignotion() 方法。

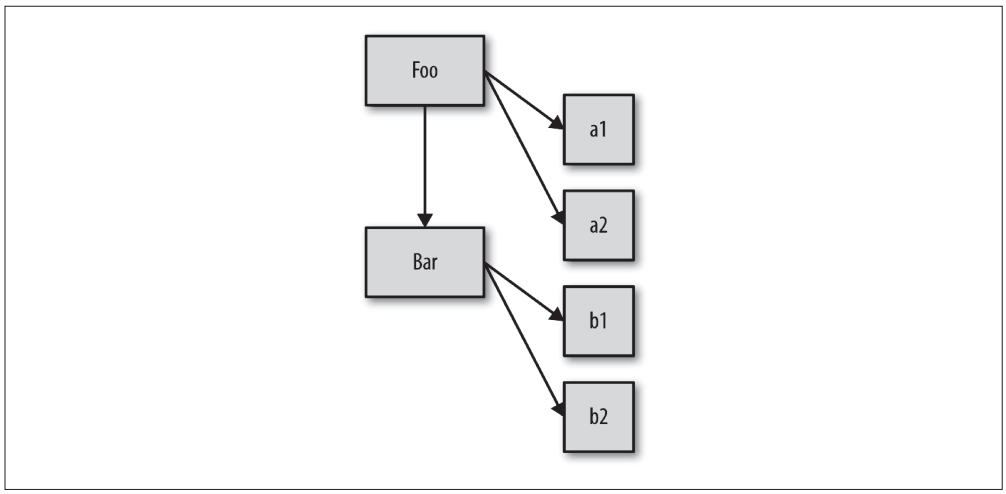
那么语言引擎会使用哪个 ignition() 呢，Vehicle 的还是 SpeedBoat 的？实际上它会使用 SpeedBoat 的 ignition()。如果你直接实例化了 Vehicle 类然后调用它的 drive()，那语言引擎就会使用 Vehicle 中的 ignition() 方法。

换言之，ignition() 方法定义的*多态性*取决于你是在哪个类的实例中引用它。

这似乎是一个过于深入的学术细节，但是只有理解了这个细节才能理解 JavaScript 中类似（但是并不相同）的 [[Prototype]] 机制。

在子类（而不是它们创建的实例对象！）中也可以相对引用它继承的父类，这种*相对引用*通常被称为 super。

还记得之前的那张图吗？



注意这些实例（a1、a2、b1 和 b2）和继承（Bar），箭头表示复制操作。

从概念上来说，子类 Bar 应当可以通过相对多态引用（或者说 super）来访问父类 Foo 中的行为。需要注意，子类得到的仅仅是继承自父类行为的一份副本。子类对继承到的一个方法进行“重写”，不会影响父类中的方法，这两个方法互不影响，因此才能使用相对多态引用访问父类中的方法（如果重写会影响父类的方法，那重写之后父类中的原始方法就不存在了，自然也无法引用）。

多态并不表示子类和父类有关联，子类得到的只是父类的一份副本。类的继承其实就是复制。

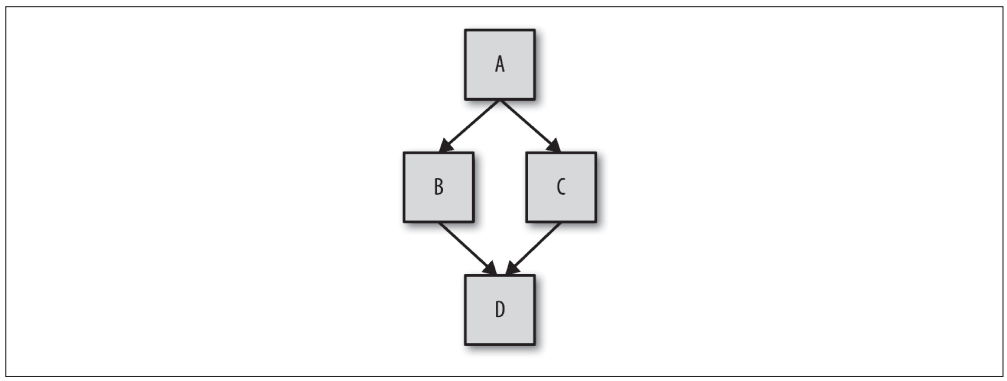
4.3.2 多重继承

还记得我们之前关于父类、子类和 DNA 的讨论吗？当时我们说这个比喻不太恰当，因为在现实中绝大多数后代是由双亲产生的。如果类可以继承两个类，那看起来就更符合现实的比喻了。

有些面向类的语言允许你继承多个“父类”。多重继承意味着所有父类的定义都会被复制到子类中。

从表面上来，对于类来说这似乎是一个非常有用的功能，可以把许多功能组合在一起。然而，这个机制同时也会带来很多复杂的问题。如果两个父类中都定义了 `drive()` 方法的话，子类引用的是哪个呢？难道每次都需要手动指定具体父类的 `drive()` 方法吗？这样多态继承的很多优点就存在了。

除此之外，还有一种被称为*钻石问题*的变种。在钻石问题中，子类 `D` 继承自两个父类（`B` 和 `C`），这两个父类都继承自 `A`。如果 `A` 中有 `drive()` 方法并且 `B` 和 `C` 都重写了这个方法（多态），那当 `D` 引用 `drive()` 时应当选择哪个版本呢（`B:drive()` 还是 `C:drive()`）？



这些问题远比看上去要复杂得多。之所以要介绍这些问题，主要是为了和 JavaScript 的机制进行对比。

相比之下，JavaScript 要简单得多：它本身并不提供“多重继承”功能。许多人认为这是件好事，因为使用多重继承的代价太高。然而这无法阻挡开发者们的热情，他们会尝试各种各样的办法来实现多重继承，我们马上就会看到。

4.4 混入

在继承或者实例化时，JavaScript 的对象机制并不会*自动*执行复制行为。简单来说，JavaScript 中只有对象，并不存在可以被实例化的“类”。一个对象并不会被复制到其他对象，它们会被*关联起来*（参见第 5 章）。

由于在其他语言中类表现出来的都是复制行为，因此 JavaScript 开发者也想出了一个方法来模拟类的复制行为，这个方法就是混入。接下来我们会看到两种类型的混入：显式和隐式。

4.4.1 显式混入

首先我们来回顾一下之前提到的 Vehicle 和 Car。由于 JavaScript 不会自动实现 Vehicle 到 Car 的复制行为，所以我们需要手动实现复制功能。这个功能在许多库和框架中被称为 extend(..)，但是为了方便理解我们称之为 mixin(..)。

```
// 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;
}

var Vehicle = {
    engines: 1,

    ignition: function() {
        console.log( "Turning on my engine." );
    },

    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log(
           "Rolling on all " + this.wheels + " wheels!"
        );
    }
} );
```

有一点需要注意，我们处理的已经不再是类了，因为在 JavaScript 中不存在类，Vehicle 和 Car 都是对象，供我们分别进行复制和粘贴。