第13章 面向对象

面向对象编程(Object Oriented Programming，OOP)是一种编程范式，它将代码分为具有属性和方法的对象。这种方式的好处是，将相关代码片段封装到对象中，由对象来维护在程序里面的生命状态。对象可以按需被重用，或者修改。我们在第6章已经看到，JavaScript里面是支持对象的，所以它也支持面向对象的编程风格。在本章中，我们就将一起来看一下在JavaScript中如何实现面向对象的编程。

本章中我们将学习以下内容：

- 编程范式介绍
- JavaScript中的原型对象
- 类和对象的创建
- 与原型相关的方法
- 面向对象的3大特性，封装、继承和多态
- 给JavaScript中的内置对象添加方法
- 对象的属性特性以及属性描述符
- 基于对象来创建对象
- 混入技术
- 改变this的指向

13-1 面向对象简介

13-1-1 编程范式

在正式介绍面向对象之前，我们先来看一下什么叫做编程范式。

所谓编程范式(programming paradigm)，指的是计算机编程的基本风格或典范模式。借用哲学的术语，如果说每个编程者都在创造虚拟世界，那么编程范式就是他们置身其中自觉不自觉采用的世界观和方法论。

在我们的编程语言里面，根据编程范式来分类大致可以分为两个大类：命令式编程和声明式编程

1. 命令式编程

一直以来，比较流行的都是命令式编程。所谓命令式编程，就是是以命令为主的，给机器提供一条又一条的命令序列让其原封不动的执行。程序执行的效率取决于执行命令的数量。
一句话概括的话，那就是命令式编程：命令"机器"如何去做事情(how)，这样不管你想要的是什么

(what)，它都会按照你的命令实现。

我们常见的命令式编程有C语言，C++，Java，C#
而在命令式编程语言里面，又可以分为两个大类：面向过程和面向对象。

面向过程
就是分析出解决问题所需要的步骤，然后把这些步骤一步一步实现。

面向对象
面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。需要说明的是：面向对象并不是完全脱离面向过程的，也就是说，就算是采用面向对象的思想，里面也会有面向过程的步骤，只不过通过面向对象可以给我们节省了很多步骤，因为这些步骤通过对象本身实现了。

2. 声明式编程

那什么是声明式编程呢？声明式编程：就是告诉"机器"你想要的是什么(what)，让机器想出如何去做(how)。
在声明式编程里面又可以分为两个大类：DSL和函数式编程。

DSL
英语全称为Domain Specific Language(领域专用语言，简称DSL)。DSL主要是指一些对应专门领域的高层编程语言，和通用编程语言的概念相对。
DSL对应的专门领域(Domain)一般比较狭窄，或者对应于某个行业，或者对应于某一类具体应用程序，比如数据库等。
我们常见的DSL有HTML，CSS，SQL等

函数式编程
函数式编程，前面在函数进阶一章也有所触及。所谓函数式编程，简单来讲，就是一种编程模型，他将计算机运算看做是数学中函数的计算。在函数式编程中，函数是基本单位，是第一型，他几乎被用作一切，包括最简单的计算，甚至连变量都被计算所取代。在函数式编程中，变量只是一个名称，而不是一个存储单元，这是函数式编程与传统的命令式编程最典型的不同之处。

随着Web开发中高并发的需求越来越多，多核并行的程序设计被推到了前线。而传统的命令式编程天生的缺陷使得并行编程模型变得非常复杂，使程序员不堪其重。函数式编程在解决并发程序上面却有着天然的优势，代码简洁优雅，但是缺点就是学习门槛高。

13-1-2 面向对象

面向对象程序设计，英语全称(Object Oriented Programming，OOP)。这里的Object，中文翻译成了对象，实际上就是指物体，或者更简单一点来讲就是指东西。事实上这是站在哲学的角度

上，将人类思维融入到了程序里面的一种编程范式。因为对于我们人类来讲，这个世界就是由千千万万的东西组成的，每个东西与东西之间又彼此产生了联系。

例如：汽车，人，大树，建筑，我们的世界由这些成千上万的东西组成。而东西与东西之间，或者说物体与物体之间又存在着彼此的联系。例如汽车在马路上跑，而人又在开汽车。

在面向对象思想提出之前，有很长一段时间是使用的面向过程的方式来编写程序。所谓面向过程，就是分析完成一件事的每一个步骤。例如从ATM机上面取钱，那么就要从插卡 -> 是存钱还是取钱 -> 如果是取钱，则输入取钱金额 -> 验证余额是否不足 -> 取出钞票 -> 退卡，每一个步骤都需要考虑，每一个步骤具体要干一些什么事也需要考虑。

然而按照我们现实生活中的思维，当我们要取钱的时候只会考虑ATM机在哪儿，找到ATM机这个东西，存钱或者取钱这个事儿基本上就搞定了。因为无论是存钱还是取钱，这都是属于ATM机这个东西的一个功能。所以不需要去考虑其他多余的步骤。这就是区别！

13-1-3 描述对象

在现实生活中，我们要描述或者向别人介绍一个东西时，一般会分为两个方面来进行介绍，分别是这个东西的外观和功能。而在程序中，我们要描述一个对象也是通过这两个方面来进行描述。

- 对象有什么特征
  对象的特征，又被称之为对象的成员属性，一般通过变量来进行存储。例如一个电视机这个对象，我要描述这个电视机的颜色时，可以将颜色存储到名为color的变量里面，这个color变量就可以被称之为电视机对象的一个属性。要描述电视机的尺寸时，可以将尺寸存储到名为size的变量里面，这个size同样也是电视机对象的一个属性。

- 对象有什么功能
  描述一个对象除了描述它的特征以外，我们往往还需要告诉别人这个对象它有什么样的作用。例如上面提到的电视机这个对象，在描述了它的外观之后，接下来我们就要开始描述它有哪些功能。而功能，在程序里面一般是通过方法(函数)来进行描述的。例如电视机有一个播放视频的功能，那么一般就会书写一个播放视频的函数挂载到电视机这个对象上面，而这个播放视频的函数就被称之为电视机的成员方法。

具体示例如下：

```
电视机
成员属性：颜色，尺寸
成员方法：播放视频
```

13-1-4 面向对象特征

一般来讲，面向对象会被分为三大特点，分别是封装，继承，多态。下面将对这三个特点进行一个简单的介绍。放心，不会太难，上面在介绍面向对象的时候有说过，面向对象是一种源自于现实生活的思想，所以这里我们就以现实生活中的事物，**电视机**为例，来看一下面向对象的这些特征。

1. 封装

当我们在看电视的时候，只需要按下遥控器的开关，就可以打开电视，按下音量键就可以调节音量，按下节目键，就可以更换电视频道。显然，当我们按下这些按钮时，电视机内部是做了一些处理才能达到这些效果的。但是，我们并不知道电视机内部具体做了什么样的处理，这其实就是演示了封装的概念：内部运作隐藏在对象里面，只有基本功能暴露给最终用户，比如这里的按钮。

在OOP中，这包括把所有编程逻辑放在一个对象中，让方法对实现功能可用，而外部世界不需要知道它是如何做的。

2. 继承

我非常喜欢我现在的这台电视机，但是它还是有一些缺点，比如音量不够大，屏幕也偏小。所以我很期待这台电视机的下一个型号，分别解决了这些问题，音量更加大，屏幕也更加艳丽。不过即使新型号的电视机有了这些优点，但是我相信内部同样使用了和我现型号电视机同样的零部件。这里其实就演示了继承的概念：采用一个对象的特性，然后添加一些新的特性。

在OOP中，这意味着我们可以继承已存在对象的所有属性和方法。然后可以通过添加新属性和方法来改进其功能。

3. 多态

我的电视机不仅仅只有看电视的功能，例如当我插上我的PS4以后，电视机屏幕就能显示出PS4里面的游戏画面。甚至我可以将我的电脑主机插在电视机上面，那么这个时候电视机就充当了一个显示器的作用。这里，我们电视机除了观看电视节目以外，还可以用作其他的用途。这里就演示了多态的概念：通过电视屏幕输出的这个行为是相同的，但是却可以用于不同的对象。

在OOP中，这意味着不同的对象可以共享这些方法。但是也能用更特定的实现来覆盖共享的方法。

13-1-5 类的概念

在上面我们介绍了面向对象的基本概念。我们的世界是由成千上万的对象组成的，但是有这么多的对象，或者说这么多的东西，应该怎么管理呢？试想一下，假设现在给你一万本书，你会怎样来管理呢？没错，按照我们人的思维，首先就会给这一万本书进行一个分类，小说分一类，杂志

分一类，IT分一类，外语分一类。其实我们现实生活中的书店都是这么来分类的。

而在面向对象编程中，我们的对象也是需要分类的，或者更确切的来讲，我们的对象就是从类里面产生出来的。类可以看作是一个铸件的模具，对象可以看作由模具具体生产出来的产品。或者可以将类比喻为生产汽车的图纸，而按照这个图纸生产出来的汽车就是一个个实际的对象。

所以，类与对象的关系我们可以总结为：类是对对象的一种概括，而对象是类的一种具体实现。

13-2 对象与原型对象

13-2-1 对象的分类

在ES6之前，对象可以分为2大类，分别是原生对象和宿主对象。

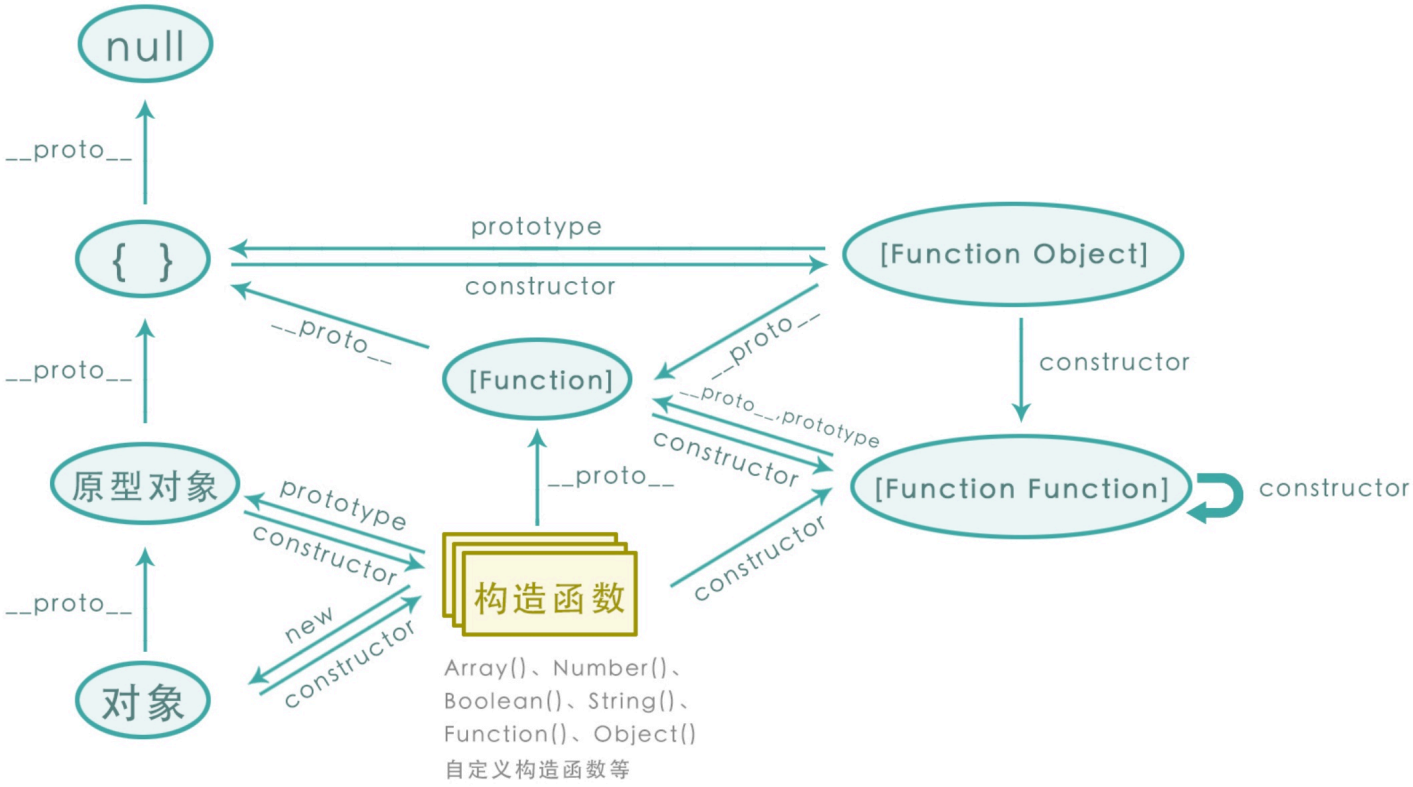
- **原生对象**
  原生对象又可以分为2类：内置对象和自定义对象。例如前面我们学习过的Date、Math、正则、数组等，这些就是典型的内置对象。它们是JavaScript这门语言本身所内置的，我们直接使用即可。
  而自定义对象则是我们开发人员自己定义的对象，例如在JS基础中介绍过的使用`{}`快速生成对象。这样的对象就可以被称之为自定义对象。

- **宿主对象**
  顾名思义，就是依存于某一个特定的环境才会有的对象。一旦离开了特定的环境，则这些对象将不存在。例如前面我们在讲解DOM编程时介绍过
  的`widnow`、`navigator`、`history`等对象，这些都是依赖于浏览器环境的。一旦脱离了浏览器环境，这些对象也就不存在了。

在ES6中，对象的类别得到了扩充，分为了4个类别，分别是普通对象，外来对象，标准对象和内置对象。

13-2-2 原型对象

在JavaScript里面，没有类这个概念，只有原型的概念。在JavaScript里面的每一个对象，都有一个原型对象，而原型对象上面也有一个自己的原型对象，一层一层向上找，最终到达`null`。这听起来感觉有点复杂，这里我们将通过一张图来进行解释，如下图：



(对象与原型对象关系图)

通过上面的图我们可以得出以下的结论：

- 每一个对象都有一个原型对象。我们可以通过 `__proto__` 来访问到某个对象的原型对象
- 通过 `__proto__` 一直向上寻找原型对象的话，最终会找到null
- 一个构造函数的 `prototype` 属性指向一个对象，而这个对象是通过该构造函数实例化出来的对象的原型对象
- JavaScript中的根对象是 `Object.prototype` 对象。 `Object.prototype` 对象是一个空对象。我们在 JavaScript中遇到的每个对象，实际上都是从 `Object.prototype` 对象克隆而来的。 `Object.prototype` 对象就是它们的原型。而 `Object.prototype` 对象的原型为null。

13-3 类与对象的创建

在前面的第一小节，我们讲述了什么是面向对象，以及什么是类，还有类和对象之间的关系。紧接着在第二小节我们话锋一转，阐述了JavaScript里面不存在类的概念，而是基于一门基于原型的语言。

既然JavaScript里面不存在类，那么为什么我们还要在第一小节花大量的篇幅来介绍面向对象里面的类呢？实际上，在JavaScript中虽然没有类，但是可以通过构造函数来模拟其他编程语言里面的类。从而从构造函数里面实例化对象出来。所以在这一小节，我们就来看一下JavaScript中如何书写构造函数来模拟其他语言中的类的。

13-3-1 构造函数

JavaScript是一门很特殊的语言，在ES6之前都没有类的概念(注：ES6新增了class关键字)，而是通过构造函数来模拟其他编程语言里面的类的。构造函数实际上也是函数，不过这种函数是专门用于生产对象的，所以被称之为构造函数。它的外表和普通函数一模一样，区别只是在于被调用的方式上面。

构造函数的函数名有一个不成文的规定，就是首字母要大写，以便和普通函数进行区分。下面的例子就演示了在JavaScript中如何书写一个构造函数：

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
```

这里，我们创建了一个`Computer`类，这个类有两个成员属性，分别是`name`和`price`，有一个成员方法，是`showSth()`。但是可以看到，这里的成员方法`showSth()`是书写在`Computer`类的prototype上面的。

在前面我们也已经介绍过了，prototype将会指向原型对象。之所以将方法添加在原型对象上面，是因为对于每个对象来讲方法体是相同的，没有必要每个对象都像属性那样单独拥有一份，所以将方法添加至原型上面，这样就达到了方法共享的效果。

好了，既然这里我们的构造函数已经书写好了，那么接下来就可以从该构造函数里面来实例化对象出来了。通过`new`运算符，可以从构造函数中实例化出来一个对象。换句话说，当使

用 `new` 运算符调用函数时，该函数总会返回一个对象，通常情况下，构造函数里面的this就指向返回的这个对象。示例如下：

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(this); // 打印出this所指向的对象
    console.log(`这是一台${this.name}电脑`);
}
let apple = new Computer("苹果",12000);
console.log(apple.name);//苹果
console.log(apple.price);//12000
apple.showSth();//Computer { name: '苹果', price: 12000 } 这是一台苹果电脑
let asus = new Computer("华硕",5000);
console.log(asus.name);//华硕
console.log(asus.price);//5000
asus.showSth();//Computer { name: '华硕', price: 5000 } 这是一台华硕电脑
```

这里，我们分别实例化出来了两个对象apple和asus。那么，这两个对象就有各自的属性值。

但用 `new` 调用构造函数时，还要注意一个问题，如果构造函数显式地返回了一个object类型的对象，那么此次运算结果最终会返回这个对象，而不是我们之前所期待的this，这里我们通过下面的两段代码来搞清楚构造函数是否显式返回object类型对象之间的区别，如下：

正常情况，构造函数没有返回object类型对象，this指向实例化出来对象

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(this); // 打印出this所指向的对象
}
let apple = new Computer("苹果",12000);
console.log(apple.name); // 苹果
apple.showSth(); // Computer { name: '苹果', price: 12000 }
```

构造函数显式的返回一个object类型的对象，那么最终使用的就是手动返回的这个对象。

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
```

```
    // 显式返回一个object类型对象
    return {
        name : "xiejie",
        showSth : function(){
            console.log("another object");
            console.log(this); // 打印出this所指向的对象
        }
    }
}
Computer.prototype.showSth = function(){
    console.log(this); // 打印出this所指向的对象
}
let apple = new Computer("苹果",12000);
console.log(apple.name); // xiejie
apple.showSth();
// another object
// { name: 'xiejie', showSth: [Function: showSth] }
```

通过上面的例子，我们可以看到，如果构造器函数不显式返回任何数据，this则就指向实例化出来的对象，而如果显式的返回一个object类型对象，那么最终使用的就是手动返回的那个对象。为什么我这里一直强调是object类型对象呢？实际上，如果返回的只是普通数据，那么this也是指向实例化出来的对象，如下：

```
let myClass = function(){
    this.name = 'xiejie';
    return {name : 'song'};
}
let obj = new myClass();
console.log(obj.name); // song
```

```
let myClass = function(){
    this.name = 'xiejie';
    return 'song';
}
let obj = new myClass();
console.log(obj.name); // xiejie
```

13-3-2 ES6类的声明

在ES6中，已经开始越来越贴近其他的高级语言了。在ES6中有了类这个概念，使用class来创建类，然后从类里面实例化对象。
但是，需要说明的是，虽然有了class这种关键字，但是这只是一种语法糖，背后对象的创建，还

是使用的是原型的方式。

具体示例如下：

```
class Computer
{
    //构造器
    constructor(name,price){
        this.name = name;
        this.price = price;
    }
    //原型方法
    showSth(){
        console.log(`这是一台${this.name}电脑`);
    }
}
let apple = new Computer("苹果",12000);
console.log(apple.name);//苹果
console.log(apple.price);//12000
apple.showSth();//这是一台苹果电脑
```

13-3-3 静态方法

所谓静态方法，又被称之为类方法。顾名思义，就是通过类来调用的方法。静态方法的好处在于不需要实例化对象，直接通过类就能够进行方法的调用。

在ES6创建类的方法的时候，可以给方法前面添加一个关键字static，来创建一个静态方法。

```
class Computer
{
    //构造器
    constructor(name,price){
        this.name = name;
        this.price = price;
    }
    //原型方法
    showSth(){
        console.log(`这是一台${this.name}电脑`);
    }
    //静态方法
    static comStruct(){
        console.log("电脑由显示器，主机，键鼠组成");
    }
}
Computer.comStruct();
```

```
//电脑由显示器，主机，键鼠组成
```

通过构造函数创建对象的时候，也有办法模拟静态方法。就是为构造函数动态的添加方法。

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
//静态方法 直接通过Computer这个构造函数来调用
Computer.comStruct = function(){
    console.log("电脑由显示器，主机，键鼠组成");
}
Computer.comStruct();
//电脑由显示器，主机，键鼠组成
```

13-4 与原型相关的方法

前面已经介绍了JS里面比较有特色的原型对象，也介绍了在JS里面如何创建构造函数或者类，然后通过构造函数和类来实例化出对象。接下来，我们就来看一下JS中和原型对象相关的一些方法。

1. prototype和__proto__

prototype是构造函数上面的一个属性，指向一个对象，这个对象是构造函数实例化出来的对象的原型对象。实例化出来的对象可以通过__proto__来找到自己的原型对象。

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
let apple = new Computer("苹果",12000);
console.log(Computer.prototype);//Computer { showSth: [Function] }
console.log(apple.__proto__);//Computer { showSth: [Function] }
console.log(Computer.prototype === apple.__proto__);//true
```

2. Object.getPrototypeOf()

除了上面介绍的通过__proto__来找到对象的原型对象以外，也可以通过`Object.getPrototypeOf()`来查找一个对象的原型对象

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
let apple = new Computer("苹果",12000);
console.log(Object.getPrototypeOf(apple));//Computer { showSth: [Function] }
console.log(apple.__proto__);//Computer { showSth: [Function] }
console.log(Object.getPrototypeOf(apple) === apple.__proto__);//true
```

3. constructor属性

通过`constructor`属性，我们可以查看到一个对象的构造函数是什么。换句话说，就是这个对象是如何得来的，示例如下：

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
let apple = new Computer("苹果",12000);
console.log(apple.constructor);//[Function: Computer]
```

4. instanceof操作符

判断一个对象是否是一个构造函数的实例。如果是返回true，否则就返回false

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
let apple = new Computer("苹果",12000);
console.log(apple instanceof Computer);//true
console.log(apple instanceof Array);//false
```

5. isPrototypeOf()

该方法将会返回一个布尔值，主要用于检测一个对象是否是一个实例对象的原型对象

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
let apple = new Computer("苹果",12000);
```

```
let test = Computer.prototype;
console.log(test.isPrototypeOf(apple));//true
```

6. hasOwnProperty()

判断一个属性是定义在对象本身上面还是从原型对象上面继承而来的。如果是本身的，则返回true，如果是继承而来的，则返回false

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
Computer.prototype.test = "this is a test";
let apple = new Computer("苹果",12000);
console.log(apple.test);//this is a test
console.log(apple.hasOwnProperty("test"));//false
console.log(apple.hasOwnProperty("name"));//true
```

13-5 封装

13-5-1 封装基本介绍

在前面介绍面向对象三大特征时，使用了现实中电视机的例子介绍了封装的概念。所谓封装，就是指隐藏内部的细节，不暴露在外面。目的就是将信息隐藏。

然而之前我们创建的对象，所有的属性都是暴露在外面的，外部可以很轻松的访问到对象内部的所有属性，如下：

```
let Computer = function(name,price){
    this.name = name;
    this.price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
let apple = new Computer("苹果",12000);
//所有属性在外部都可以访问到
console.log(apple.name);//苹果
console.log(apple.price);//12000
```

如果我们不希望所有的属性外部都能访问到，而是部分属性对外隐藏，这个时候我们就需要对这些属性进行封装了。

在许多面向对象语言中，封装由语法解析来实现。这些语言提供了private，public，protected等关键字来提供不同的访问权限。但是在JavaScript中并没有提供对这些关键字的支持。

不过在JavaScript中要实现封装的方法也非常简单，可以利用函数的作用域来进行模拟。在声明属性的时候，添加关键字(let、const、var)即可。原因也非常简单，因为构造函数也是函数，既然是函数就会有函数作用域，所以在属性的前面添加了关键字之后使其成为一个局部变量，外部自然而然也就访问不到了。示例如下：

```
let Computer = function(name,price){
    this.name = name;
    let _price = price;
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
let apple = new Computer("苹果",12000);
```

```
console.log(apple.name);//苹果
console.log(apple.price);//undefined
console.log(apple._price);//undefined
```

一般来讲，对于私有属性，有一个不成文的规定，那就是习惯使用_开头来命名私有属性。这里我们可以看到，由于我们对price属性进行了封装，所以外部无法访问到。

封装后的属性，我们可以将其称之为私有属性。对于外部来讲，私有属性是不可见的，这个我们已经知道了，但是对于内部来讲，私有属性是可见的。这就好比电视机里面的零部件封装后外部无法直接访问到，但是电视机内部是可以使用这个零部件来完成相应的功能的。示例如下：

```
let Computer = function(name,price){
    this.name = name;
    let _price = price;
    this.sayPrice = function(){
        console.log(`价格为${_price}`);
    }
}
Computer.prototype.showSth = function(){
    console.log(`这是一台${this.name}电脑`);
}
let apple = new Computer("苹果",12000);
apple.sayPrice();//价格为12000
```

13-5-2 存取器

很多时候，我们对对象的属性进行封装，并不是为了外部访问不到，而是为了当外部对该对象的属性进行访问或者设置的时候，给予一定的限制。也就是说，外部能够正常的访问到进行了限制的属性。在JS中，可以通过`Object.defineProperty()`方法来给对象添加属性，而添加的属性我们就可以为其设置一定的属性特性，这个我们在后面会详细来看。属性特性里面其中就包含了存取器的设置。一个是get，另一个是set。

- get：一旦目标属性被访问时，就会调用相应的方法
- set：一旦目标属性被设置时，就会调用相应的方法

接下来我们来看一个具体的示例：

```
let stu = {};
let age = 18;
Object.defineProperty(stu,"stuAge",{
    //get:当外界获取stuAge属性时会自动调用
    get:function(){
```

```
        return age;
    },
    //set:当外界要设置stuAge属性时会自动调用
    set:function(value){
        if(value > 100 || value < 0)
        {
            age = 20;
        }
        else{
            age = value;
        }
    }
})
console.log(stu.stuAge);//18;
stu.stuAge = 30;
console.log(stu.stuAge);//30
stu.stuAge = 1000;
console.log(stu.stuAge);//20
console.log(stu.hasOwnProperty("stuAge"));//true
```

这里我们为stu对象自定义了一个stuAge属性。然后为其设置了get和set存取器。当用户获取stu对象的stuAge属性值时，会自动调用get所对应的函数，然后返回age变量。当用户要设置age的值时，这时就有一个限制，如果用户设置的值大于100或者小于0，则将age的值设置为20，否则就将用户设置的值赋值给age。

如果要设置多个属性，只需要将`defineProperty`修改为`defineProperties`，这样就可以设置多个属性了。不过使用该方法接收的参数也有一定的变化，示例如下：

```
let stu = {};
Object.defineProperties(stu,{
    stuAge : {
        get:function(){
            return 18;
        }
    },
    stuName : {
        get:function(){
            return "xiejie";
        }
    }
});
console.log(stu.stuName);//xiejie
console.log(stu.stuAge);//18
```

这里，我们就为stu对象同时添加了两个属性，分别是stuName和stuAge，并设置了get所对应的

方法。

13-6 继承

13-6-1 继承基本介绍

在程序语言中，面向对象里面的继承是指一个子类去继承一个父类。子类继承了父类之后，父类所有的属性和方法都自动都拥有了。

继承的好处

继承最大的好处，就在于代码复用。例如有两个类，这两个类的属性和方法基本相同，那么我们就可以考虑给这两个类写一个父类

继承的缺点

首先如果继承设计得非常复杂，那么整个程序的设计也会变得庞大和臃肿。甚至经常会出现"大猩猩与香蕉"的问题。"大猩猩与香蕉"这个名字来自于Erlang编程语言的创始人Joe Armstrong的文章：你想要一个香蕉，但是得到的是拿着香蕉和整个丛林的大猩猩。

还有一个问题就是，如果是像C++那样的多继承语言，那么还可能会遇到菱形继承的问题。
这里我们首先来看一下什么叫做多继承。既然有多继承，自然就有单继承。
单继承：所谓单继承，就是指只有一个父类
多继承：所谓多继承，就是指有多个父类

菱形继承问题：首先需要说明，菱形继承问题只会在多继承里面才会出现。例如：A和B都继承Base类，假设Base类里面有一个成员方法，那么A和B里面都会有这个成员方法，这个时候A和B两个类又都对这个成员方法进行了修改。接下来让一个C来同时继承A和B，那么这个时候就会产生一个问题，对于同名的方法，究竟使用哪一个，这就是所谓的菱形继承问题。这个是属于设计上的问题。

不过，JavaScript是一门单继承的语言，所以一定程度上避免了菱形继承的问题。

13-6-2 对象冒充

最早的时候，在JavaScript里面采用的是对象冒充的方式来实现的继承。所谓对象冒充，就是用父类(构造函数)去充当子类的属性，示例代码如下：

```
let Person = function(name,age){
    this.name = name;
    this.age = age;
```

```
}
let Student = function(name,age,gender,score){
    this.temp = Person;//将Person构造函数作为Student的一个属性
    this.temp(name,age);//给Person构造函数里面的this.name以及this.age赋值
    delete this.temp;//this.temp已经无用，将其删除
    this.gender = gender;
    this.score = score;
}
let xiejie = new Student("谢杰",18,"男",100);
console.log(xiejie.name);//谢杰
console.log(xiejie.age);//18
console.log(xiejie.gender);//男
console.log(xiejie.score);//100
```

可以看到，这里Student确实已经就继承到了Person，成功的访问到了name和age属性。但是，使用对象冒充来实现继承也有一个缺陷，那就是无法继承到原型对象上面的属性，证明如下：

```
let Person = function(name,age){
    this.name = name;
    this.age = age;
}
Person.prototype.test = function(){
    console.log("this is a test");
}
let Student = function(name,age,gender,score){
    this.temp = Person;//将Person构造函数作为Student的一个属性
    this.temp(name,age);//给Person构造函数里面的this.name以及this.age赋值
    delete this.temp;//this.temp已经无用，将其删除
    this.gender = gender;
    this.score = score;
}
let xiejie = new Student("谢杰",18,"男",100);
console.log(xiejie.name);//谢杰
console.log(xiejie.age);//18
console.log(xiejie.gender);//男
console.log(xiejie.score);//100
xiejie.test();//TypeError: xiejie.test is not a function
```

方法借用模式

现在书写JavaScript继承，基本上不会再使用对象冒充了，但是上面之所以要介绍对象冒充这种继承方式，是为了顺带引出JS中一个非常有特色的方法借用模式。这在JS中是相当灵活的一种模式，不需要继承，就可以从父类或者原型上面借用相应的属性和方法。这里，我们先来介绍两个方法，分别是 `call()` 和 `apply()` 方法，call和apply方法都能继承另外一个对象的方法和属

性，具体的语法如下：

```
Function.apply(obj, args)方法能接收两个参数
obj：这个对象将代替Function类里this对象
args：这个是数组，它将作为参数传给Function(args->arguments)

call:和apply的意思一样,只不过是参数列表不一样.
Function.call(obj, [param1[, param2[, ...[, paramN]]]])
obj：这个对象将代替Function类里this对象
params：这个是一个参数列表
```

这两个方法除了参数的形式外基本上是相同的。`call()`方法后面接收的是参数列表，而`apply()`方法后面则是接收的是一个数组。简单概括起来，可以写作如下的形式：

```
A.call(B, args)
```

代表着将A应用到B里面，但是作用域还是保留B的作用域。这就是JS里面经典的方法借用。我们来看一个具体的示例，如下：

```
let One = function(){
    this.name = "one";
    this.showName = function(){
        console.log(this.name);
    }
}
let two = {
    name : "two"
};
let one = new One();
one.showName.call(two);//two
//将one的showName方法应用到two上面
//two原本没有showName方法，找one借用后有了该方法
```

接下来我们再来看一个经典的例子，为数组求最值

```
let arr = [3,15,8,1,2,17,11];
let max = Math.max.apply(this,arr);
console.log(max);//17
```

这里我们借用了Math对象的max方法，将其应用到全局对象上面。注意这里的this是指向的全局对象，然后将数组作为apply的第二个参数传入进去。

接下来我们来看一下通过call和apply来实现的继承
call示例：

```
let Person = function(name,age){
    this.name = name;
    this.age = age;
}
Person.prototype.test = function(){
    console.log("this is a test");
}
let Student = function(name,age,gender,score){
    //将Person构造函数应用到this里面
    //this后面是参数
    Person.call(this,name,age);
    this.gender = gender;
    this.score = score;
}
let xiejie = new Student("谢杰",18,"男",100);
console.log(xiejie.name);//谢杰
console.log(xiejie.age);//18
console.log(xiejie.gender);//男
console.log(xiejie.score);//100
xiejie.test();//TypeError: xiejie.test is not a function
```

apply示例：

```
let Person = function(name,age){
    this.name = name;
    this.age = age;
}
Person.prototype.test = function(){
    console.log("this is a test");
}
let Student = function(name,age,gender,score){
    //将Person构造函数应用到this里面
    //this后面是参数
    Person.apply(this,[name,age]);
    this.gender = gender;
    this.score = score;
}
let xiejie = new Student("谢杰",18,"男",100);
console.log(xiejie.name);//谢杰
console.log(xiejie.age);//18
console.log(xiejie.gender);//男
console.log(xiejie.score);//100
xiejie.test();//TypeError: xiejie.test is not a function
```

可以看到，通过方法借用模式也成功借用到了Person类的属性，从而实现了继承的效果。但是还是老问题，就是处于原型对象上面的属性和方法无法继承到。所以，才有了后面的通过原型链来实现继承。

13-6-3 原型继承

这种方式的核心思路就是改变构造函数的prototype的指向，使其指定到我们想要继承的类的实例对象上面，示例如下：

```
let Person = function(name,age){
    this.name = name;
    this.age = age;
}
Person.prototype.test = function(){
    console.log("this is a test");
}
let Student = function(name,age,gender,score){
    //将Person构造函数应用到this里面
    //this后面是参数
    Person.apply(this,[name,age]);
    this.gender = gender;
    this.score = score;
}
Student.prototype = new Person();//改变Student构造函数的原型对象
let xiejie = new Student("谢杰",18,"男",100);
console.log(xiejie.name);//谢杰
console.log(xiejie.age);//18
console.log(xiejie.gender);//男
console.log(xiejie.score);//100
xiejie.test();//this is a test
```

可以看到，这里我们使用`apply()`方法来实现对象本身的属性和方法的继承，然后通过将Person的实例化对象赋值给Student的prototype的方法来改变其原型，从而实现了Person原型上面的属性和方法也可以被继承下来。

13-6-4 ES6继承方式

从ES6开始，可以使用extends关键字来实现继承了，示例如下：

```
class Person
{
    constructor(name,age){
```

```
        this.name = name;
        this.age = age;
    }
    sayName(){
        console.log(`my name is ${this.name}`);
    }
}
class Student extends Person
{
    constructor(name,age,gender,score){
        super(name,age);//super代表访问父类的构造函数
        this.gender = gender;
        this.score = score;
    }
    learn(){
        console.log("I\'m learning");
    }
}
let xiejie = new Student("xiejie",18,"male",100);
console.log(xiejie.name);//xiejie
console.log(xiejie.age);//18
console.log(xiejie.gender);//male
console.log(xiejie.score);//100
xiejie.sayName();//my name is xiejie
xiejie.learn();//I'm learning
```

13-7 多态

13-7-1 多态简介

"多态"一词源于希腊文的polymorphism，拆开来看poly(复数) + morph(形态) + ism，从字面上我们可以理解为复数形态。

多态的实际含义是：**同一操作作用于不同的对象上面，可以产生不同的解释和不同的执行结果。换句话说，给不同的对象发送同一个消息的时候，这些对象会根据这个消息分别给出不同的反馈。**

从字面上来理解多态不太容易，下面我们来举例说明一下：
主人家里养了两只动物，分别是一只鸭和一只鸡，当主人向它们发出"叫"的命令时，鸭会"嘎嘎嘎"地叫，而鸡会"咯咯咯"地叫。这两只动物都会以自己的方式来发出叫声。在这里，鸭和鸡是两个不同的对象，但是它们都拥有"叫"这个方法。当主人调用"叫"这个方法时，不同的对象给出了不同的反馈。

其实，这里面就隐含了多态的思想。

在我们所接触的JavaScript代码中，最常见的多态，就是前面我们所见到过的`toString()`方法，虽然每个对象都拥有`toString()`方法，但是并不是所有对象的表现形式都是一样的，如下：

```
let i = 3;
console.log(i.toString());//"3"
let j = [1,2,3];
console.log(j.toString());//"1,2,3"
```

同样都是`toString()`方法，但是对于普通数字来讲就是单纯的返回数字，而对于数组对象而言则是将每个值以逗号分隔的形式来返回。

13-7-2 实现多态

由于JavaScript是一门动态语言，可以动态的为对象添加属性和方法，并且是实时的，所以JavaScript里面的对象可以说天生就是多态的。示例如下：

```
let duck = {
    name : "鸭子",
    makeSound : function(){
```

```
        console.log("嘎嘎嘎");
    }
};
let chicken = {
    name : "鸡",
    makeSound : function(){
        console.log("咯咯咯");
    }
};
let animalSound = function(obj){
    obj.makeSound();
}
animalSound(duck); // 嘎嘎嘎
animalSound(chicken); // 咯咯咯
```

即使是从父类继承而来的方法，也可以通过方法覆盖的方式来实现父类实例对象和子类实例对象同名方法不同行为的多态表现，示例代码如下：

```
// 父类构造函数
let Person = function(name,age){
    this.name = name;
    this.age = age;
}
Person.prototype.sayName = function(){
    console.log(`my name is ${this.name}`);
}
// 子类构造函数
let Student = function(name,age,gender,score){
    Person.call(this,name,age);
    this.gender = gender;
    this.score = score;
}
Student.prototype = new Person();//改变Student构造函数的原型对象
//改写父类的sayName()方法
Student.prototype.sayName = function(){
    console.log(`haha,I'm ${this.name}`);
}
let xiejie = new Student("谢杰",18,"男",100);
xiejie.sayName();//haha,I'm 谢杰
let song = new Person("宋",20);
song.sayName();//my name is 宋
```

在ES6中我们也可以在子类里面书写和父类相同的方法名来进行覆盖，如下：

```
class Person
```

```
{
    constructor(name,age){
        this.name = name;
        this.age = age;
    }
    sayName(){
        console.log(`my name is ${this.name}`);
    }
}
class Student extends Person
{
    constructor(name,age,gender,score){
        super(name,age);//super代表访问父类的构造函数
        this.gender = gender;
        this.score = score;
    }
    sayName(){
        console.log(`haha,I'm ${this.name}`);
    }
    learn(){
        console.log("I\'m learning");
    }
}
let xiejie = new Student("谢杰",18,"男",100);
xiejie.sayName();//haha,I'm 谢杰
let song = new Person("宋",20);
song.sayName();//my name is 宋
```

13-7-3 多态的意义

Martin Fowler在《重构：改善既有代码的设计》里写道：

多态的最根本的好处在于，你不必再向对象询问"你是什么类型"后根据得到的答案调用对象的某个行为，你只需要调用该行为就是了，其他的一切多态机制都会为你安排妥当。

换句话说，多态最根本的作用就是通过把过程化的条件分支语句转化为了对象的多态性，从而消除了这些条件分支语句。

Martin Fowler的话可以用下面这个现实生活中的例子来很好的进行诠释：

在电影的拍摄现场，当导演喊出"action"时，主角开始说台词，照明师负责打灯光，后面的群众演员假装中枪倒地，道具师往镜头里撒上雪花。在得到同一个消息时，每个对象都知道自己应该做什么。如果不利用对象的多态性，而是使用面向过程的方式来编写这一段代码，那么相当于在电影开始拍摄之后，导演每次都要走到每个人面前，确认它们的职业分工(类型)，然后告诉他们要做什么。如果映射到程序中，那么程序中将充斥着大量的条件分支语句。

利用对象的多态性，导演在发布消息时，就不必考虑每个对象接到消息后应该做什么。对象应该做什么并不是临时决定的，而是已经事先约定和排练完毕的。每个对象应该做什么，已经成为了该对象的一个方法，被安装在对象的内部，每个对象负责它们自己的行为。所以这些对象可以根据同一个消息，有条不紊地分别进行各自的工作。

将行为分布在各个对象中，并让这些对象各自负责自己的行为，这正是面向对象设计的优点。

13-8 内置对象添加方法

前面在介绍多态的时候我们有提到过，JS由于是动态语言，所以我们可以实时的为对象添加属性和方法，甚至是JS中的内置对象，我们也可以很轻松的为其添加更多的属性和方法，从而带来更多的功能。这种做法往往被称之猴子补丁(monkey-patching)。不过，尽管这是一项非常强大的技术，但是JS社区的大部分人都不推荐这么做，大部分人的观点是"被耍流氓，不是你的对象别动手动脚"。

下面我们来举一个例子，给Number对象的原型添加`isOdd()`和`isEven()`方法，然后这些方法就可以被基本数据类型的数字值所使用：

```
Number.prototype.isEven = function(){
    return this%2 === 0;
}
Number.prototype.isOdd = function(){
    return this%2 === 1;
}
let i = 42;
console.log(i.isEven());//true
let j = 13;
console.log(j.isOdd());//true
```

有些方法规范中有，但是部分浏览器并不支持。比如，`String.prototype`中方法之一的`trim()`方法。这是个所有字符串都会继承的方法，它删除字符串开头和结尾的所有空白，但是不幸的是，IE8及以下版本并没有实现该方法。这个时候我们就可以使用猴子补丁来进行添加。

```
String.prototype.trim = String.prototype.trim || function(){
    return this.replace(/^\s+|\s+$/,'');
}
console.log("   hello   ".trim().length);//5
```

在上述的代码中，如果trim()方法存在，就用内置的String.prototype.trim，否则就用猴子补丁提供的函数对String的原型添加该方法。

虽然对内置对象添加猴子补丁看似是添加额外的活着遗漏的功能的一种好办法，不过它也会带来不可预期的行为。目前JavaScript社区的共识为：不应该这样做。所以除非有很好的理由，否则应该避免给任何内置对象添加猴子补丁。如果我们添加的方法后面被语言原生实现了，就会出现更多的问题。

避免引起问题的另一种方式使用`extends`子类化一个已经创建好了的类，从而创建自己的类。例如，我们可以像下面一样，通过继承内置的Number类来创建自己的Number类

```
class myNum extends Number
{
    constructor(...args){
        super(...args);
    }
    isEven(){
        return this%2 === 0
    }
}
let i = new myNum(42);
//无论是新类添加的方法还是Number类里面的方法都可以使用
console.log(i.isEven());//true
console.log(i.toFixed(2));//42.00
```

13-9 属性特性以及描述符

在前面我们介绍封装的时候，有接触过get和set方法。这两个方法是通过`Object.defineProperty()`给对象添加属性时，挂在属性上面的。实际上，每个属性除了get和set以外，还有一些其他的特性，这些特性提供了有关该属性的相关信息。

对象属性的所有特性如下：

- value：这是属性的值，默认是`undefined`
- writable：这是一个布尔值，表示一个属性是否可以被修改，默认是true
- enumerable：这是一个布尔值，表示在用for-in循环遍历对象的属性时，该属性是否可以显示出来，默认值为true
- configurable：这是一个布尔值，表示我们是否能够删除一个属性或者修改属性的特性，默认值为true

可以通过`Object.getOwnPropertyDescriptor()`来获取到一个对象的属性的特性，示例如下：

```
let xiejie = {
    name : "xiejie"
}
let i = Object.getOwnPropertyDescriptor(xiejie,"name");
console.log(i);
// { value: 'xiejie',
//   writable: true,
//   enumerable: true,
//   configurable: true }
```

这里我们来使用`Object.defineProperty()`方法来添加对象属性的方式为对象的属性赋一个值，如下：

```
let xiejie = {
    name : "xiejie"
}
Object.defineProperty(xiejie,"age",{
    value : 18,
    writable : false
})
console.log(xiejie.name);//xiejie
console.log(xiejie.age);//18
xiejie.age = 28;
console.log(xiejie.age);//18
```

可以看到，这里我们通过`Object.defineProperty()`方法来为xiejie这个对象设置了一个age属性，并且将value值设置为18。与此同时将writable设置为了false，所以外界无法修改xiejie这个对象的age属性的属性值。

最后需要注意的一个地方就是，关于对象属性的特性设置时，并不是所有的特性都是可以同时设置的，有一些特性是不能够同时进行设置的。示例如下：

```
let xiejie = {
    name : "xiejie"
}
Object.defineProperty(xiejie,"age",{
    value : 18,
    writable : false,
    get : function(){
        return 18;
    }
})
console.log(xiejie.age);
//Cannot both specify accessors and a value or writable attribute
```

13-10 基于对象创建对象

前面我们也有提到过，JavaScript是一门基于原型的语言，不存在"类"的概念。在以类为中心的面向对象编程语言中，类和对象的关系可以想象成铸模和铸件的关系，对象总是从类中创建而来的。而在原型编程的思想中，类并不是必须的，对象未必需要从类中创建而来，一个对象是通过克隆另外一个对象所得到的。

原型模式实现的关键，是语言本身是否提供了clone方法。ECMAScript5提供了`Object.create`方法，可以用来克隆对象。这让我们可以完全避免使用类，转而基于另一个充当蓝图或者原型的对象来创建新的对象。这个新的对象就是作为参数所提供的对象的一个精确复制。而作为参数所提供的那个对象则充当新对象的原型对象。示例代码如下：

```
let person = {
    arms : 2,
    legs : 2,
    walk(){
        console.log('walking');
    }
}
let xiejie = Object.create(person);
console.log(xiejie.arms);//2
console.log(xiejie.legs);//2
xiejie.walk();//walking
console.log(xiejie.__proto__ === person);//true
```

我们在使用`create()`方法来创建对象的时候，还可以传入第二个参数，第二个参数是一个JSON对象，JSON里面书写的是对象属性的一些设定，示例如下：

```
let person = {
    arms : 2,
    legs : 2,
    walk(){
        console.log('walking');
    }
}
let xiejie = Object.create(person,{
    name : {
        value : "xiejie",
        writable : false,
        enumerable : true
    },
    age : {
```

```
        value : 18,
        enumerable : false
    }
});
console.log(xiejie.name);//xiejie
console.log(xiejie.age);//18
console.log(xiejie.arms);//2
console.log(xiejie.legs);//2
for(let i in xiejie)
{
    console.log(i);
    // name
    // arms
    // legs
    // walk
}
```

基于对象创建的对象同样存在继承。下面的例子演示了通过`Object.create()`方法来创建的对象，里面所存在的继承关系，如下：

```
let person = {
    arms : 2,
    legs : 2,
    walk(){
        console.log('walking');
    }
}
let xiejie = Object.create(person,{
    name : {
        value : "xiejie",
        writable : false,
        enumerable : true
    },
    age : {
        value : 18,
        enumerable : false
    },
    born : {
        value : "chengdu"
    }
});
let son = Object.create(xiejie,{
    name : {
        value : "xizhi"
    },
    age : {
        value : 0
```

```
    }
})
console.log(son.name);//xizhi
console.log(son.age);//0
console.log(son.born);//chengdu
console.log(son.arms);//2
console.log(son.legs);//2
son.walk();//walking
console.log(xiejie.isPrototypeOf(son));//true
console.log(person.isPrototypeOf(son));//true
```

最后需要说明的是，当我们使用new运算符从构造函数实例化对象时，实际上在JavaScript引擎的内部，也是先克隆了`Object.prototype`对象，然后再进行一些其他的额外操作。

13-11 混入技术

混入(mixins)是一种不用继承，就将某些对象的属性和方法添加到另一个对象上面的技术。它允许通过混合基础对象在一起，来创建更加复杂的对象。

基础的混入功能是由`Object.assign()`方法提供的。这个方法会把目标的对象作为第一个参数，所有需要混入进来的对象作为后面的参数，然后这些对象的属性和方法就被合并到目标对象上面。示例如下：

```
let a = {};
let b = {
    name : "xiejie",
    age : 18
}
let c = {
    gender : "male",
    age : 20
}
Object.assign(a,b,c);
console.log(a.name);//xiejie
console.log(a.age);//20
console.log(a.gender);//male
```

13-11-1 浅复制与深复制

上面虽然我们实现了对象的混入，但是，这种混入有一个缺点，那就是混入时如果对象的属性是一个引用类型的数据，例如是一个数组或者是一个对象，那么只会发生浅复制，也就是说只会复制其引用，示例如下：

```
let a = {};
let b = {
    test : [1,2,3]
}
Object.assign(a,b);
console.log(a.test);//[ 1, 2, 3 ]
b.test.push(4);
console.log(a.test);//[ 1, 2, 3, 4 ]
console.log(b.test);//[ 1, 2, 3, 4 ]
```

可以看到，这里我们向a对象混入了b对象，但是b对象里面的test所对应的是一个数组，而这里

由于只是浅复制，所以当我们改变b对象的test属性，往数组里面添加一个元素时，a对象的test属性也会发生相应的变化。

要避免只是浅复制，我们可以自己创建一个`mixin()`函数，该函数会对一个对象的所有属性进行深度的复制，书写如下：

```
let mixin = function(target,...objects){
    //首先遍历要混入的对象 用object来进行接收每一个要混入的对象
    for(const object of objects)
    {
        //要混入的必须是对象 所以先进行一个类型的判断
        if(typeof object === 'object')
        {
            //取出混入对象的每一个属性
            for(const key of Object.keys(object))
            {
                //如果属性所对应的值为对象 继续递归调用mixin
                if(typeof object[key] === 'object')
                {
                    target[key] = Array.isArray(object[key]) ? [] : {};
                    mixin(target[key],object[key]);
                }
                else{
                    //否则就直接混入到目标对象里面
                    Object.assign(target,object);
                }
            }
        }
    }
    return target;
}
```

有了该函数以后，我们就可以使用该函数来进行对象的混入，还是演示上面的那个例子：

```
let a = {};
let b = {
    test : [1,2,3]
}
mixin(a,b);
console.log(a.test);//[ 1, 2, 3 ]
b.test.push(4);
console.log(a.test);//[ 1, 2, 3 ]
console.log(b.test);//[ 1, 2, 3, 4 ]
```

可以看到，这一次我们再次修改b对象的test属性值，为数组推入一个元素时，没有影响到a对象

的test属性值了。说明我们这一次是进行了深度的复制。

13-11-2 用mixin函数添加模块化功能

继承允许我们通过继承其他对象的属性和方法，来给对象添加功能。虽然这个很有用，不过创建一个继承链是不可取的，因为这很容易出现"大猩猩与香蕉"的问题。有时我们只是想给对象添加一些属性和方法，并不想把两个对象链在一起。`mixin()`函数就可以帮助我们解决这个问题，让我们将属性和方法封装在一个对象里面，然后把它们添加到另一个对象，而不用付出创建继承链的代价。

例如，我要创建一个班级对象，而这个班级对象是2名学生组成。那么我们优先进行考虑的应该是混入，因为如果使用继承的话，一个学生可能会继承他的父亲，而他的父亲又有可能继承他的爷爷，这里就回到了"大猩猩与香蕉"的问题，我只想得到的是两个学生，却得到的是两大家人。

使用`mixin()`添加模块的示例如下：

```
let a = {a : "张三"};
let b = {b : "李四"};
let c = {c : "王五"};
let banji = {};
mixin(banji,a,b,c);
console.log(banji);//{ a: '张三', b: '李四', c: '王五' }
```

13-12 改变this的指向

在前面的JS基础部分，我们有介绍过JavaScript中关于this的指向，默认有两种情况：

- 如果是在对象里面使用`this`，一般来讲，关键字`this`是指它所在的对象。`this`可以用在方法内，获取对对象属性的访问。
- 还有一种情况，是`this`指向全局对象。当this是在全局中调用，或者是在函数中调用的时候，指向的就是全局对象(node里面是global对象，浏览器环境里面是window对象)。

但是，这个this的指向，我们是可以进行修改的。接下来我们来介绍几种修改this指向的方法。

13-12-1 使用call或者apply来修改this指向

前面我们有介绍过JS中的方法借用模式，使用`call()`或者`apply()`可以借用其他对象的方法而不用通过继承。实际上，`call()`和`apply()`也算是间接修改了this的指向。示例如下：

```
let Test = function(){
    this.name = "JavaScript";
    this.say = function(){
        console.log(`这是${this.name}`);
    }
}
let test = new Test();
test.say();//这是JavaScript
let a = {name : "PHP"};
test.say.call(a);//这是PHP
```

这里我们借用了test对象的`say()`方法，本来对于test对象的`say()`方法来讲，this是指向test对象的，但是现在a对象借用了`say()`方法以后，this是指向a对象的，所以打印出了"这是PHP"

13-12-2 通过bind()方法来强行绑定this指向

第二种方式是可以通过`bind()`方法来强行绑定this的指向。`bind()`方法的语法如下：
`fun.bind(thisArg[, arg1[, arg2[, ...]]])`
参数：

- thisArg 当绑定函数被调用时，该参数会作为原函数运行时的 this 指向。
- arg1, arg2, ... 当绑定函数被调用时，这些参数将置于实参之前传递给被绑定的方法。

我们来看一个具体的示例，如下：

```
let a = {name : "PHP"};
let Test = function(){
    this.name = "JavaScript";
    this.say = function(){
        console.log(`这是${this.name}`);
    }.bind(a);
}
let test = new Test();
test.say();//这是PHP
```

这里我们在`say()`方法后面使用了`bind()`方法，并将a对象作为参数传递进去。这个时候我们就会发现，当我们以test对象的身份来调用`say()`方法时，打印出来的仍然是"这是PHP"，这就是因为`bind()`方法在这里强行改变了this的指向，使this指向了a对象。

13-12-3 箭头函数的this指向

当我们的this是以函数的形式调用时，this指向的是全局对象。不过对于箭头函数来讲，却比较特殊。箭头函数的this指向与其他函数不一样，它的this指向始终是指向的外层作用域。

我们先来看一个普通函数作为对象的一个方法被调用时，this的指向，如下：

```
let obj = {
    x : 10,
    test : function(){
        console.log(this);//指向obj对象
        console.log(this.x);//10
    }
}
obj.test();
// { x: 10, test: [Function: test] }
// 10
```

可以看到，普通函数作为对象的一个方法被调用时，this的指向是指向的这个对象，上面的例子中就是指向的obj这个对象，this.x的值为10。当然这是我们最早在JS基础部分的对象一章就已经介绍过的内容。

接下来我们来看一下箭头函数以对象的方式被调用的时候的this指向，如下：

```
let obj = {
    x : 10,
    test : ()=>{
        console.log(this);// {}
```

```
        console.log(this.x);// undefined
    }
}
obj.test();
// {}
// undefined
```

这里打印出来的结果上面明显不一样。this打印出来的是`{}`，而this.x的值则为undefined。为什么呢？实际上刚才我们有讲过，箭头函数的this指向与其他函数不一样，它的this指向始终是指向的外层作用域。所以这里的this实际上是指向的全局对象。但是怎么证明呢？方法非常简单，如下：

```
var x = 20;//全局环境中添加一个变量x 赋值为20
let obj = {
    x : 10,
    test : ()=>{
        console.log(this);// Window
        console.log(this.x);// 20
    }
}
obj.test();
```

这里我们在全局环境下面添加了一个变量x，并赋了一个值20。既然你说是指向全局对象的，那么this.x应该能够打印出20才对。我们在浏览器中执行该代码，效果如下：

```
> var x = 20;//全局对象中添加一个变量x 赋值为20
  let obj = {
      x : 10,
      test : ()=>{
          console.log(this);// Window
          console.log(this.x);// 20
      }
  }
  obj.test();
                                                          VM84:5
  ▶ Window
    {postMessage: f, blur: f, focus: f, close: f, frames: Window, …}
  20                                                      VM84:6
< undefined
```

接下来我们再来看一个例子，来证明箭头函数的this指向始终是指向的外层作用域。

```
var name = "JavaScript";
let obj = {
    name : "PHP",
    test : function(){
        let i = function(){
            console.log(this.name);
```

```
            //i是以函数的形式被调用的，所以this指向全局
            //在浏览器环境中打印出JavaScript，node里面为undeifned
        }
        i();
    }
}
obj.test();//JavaScript
```

接下来我们将 `i()` 函数修改为箭头函数

```
var name = "JavaScript";
let obj = {
    name : "PHP",
    test : function(){
        let i = ()=>{
            console.log(this.name);
            //由于i为一个箭头函数，所以this是指向外层的
            //所以this.name将会打印出PHP
        }
        i();
    }
}
obj.test();//PHP
```

最后需要说一点的就是，箭头函数不能用做构造函数，如下：

```
let Test = (name,age) => {
    this.name = name;
    this.age = age;
};
let test = new Test("xiejie",18);
//TypeError: Test is not a constructor
```

总结

1. 所谓编程范式，指的是计算机编程的基本风格或典范模式。大致可以分为命令式编程和声明式编程。
2. 面向对象的程序设计是站在哲学的角度上，将人类思维融入到了程序里面的一种编程范式。
3. 描述对象的时候可以通过两个方面来进行描述，分别是对象的外观和功能。在程序中与之对象的就是属性和方法。
4. 面向对象的3大特征有封装，继承和多态。

5. 在其他语言中，对象从类中产生。而在JavaScript中，通过构造函数来模拟其他语言中的类。
6. 类与对象的关系可以总结为，类是对对象的一种概括，而对象是类的一种具体实现。
7. 在JavaScript中每个对象都有一个原型对象。可以通过__proto__属性来找到一个对象的原型对象。
8. 封装是指对象的属性或者方法隐藏于内部，不暴露给外部。
9. 继承是指一个子类继承一个父类。在继承了父类以后，子类就拥有了父类所有的属性和方法。
10. 多态是指不同的对象可以拥有相同的方法，不过是以不同的方式来实现它。
11. 可以给内置的对象添加属性和方法，这种行为被称之为猴子补丁。一般不推荐这么做。
12. 在JavaScript中，可以为对象的属性制定一系列的特性。
13. 除了通过构造函数来产生对象，还可以通过对象来生产一个对象。
14. 混入技术是一种不需要继承就能够将某些对象的属性和方法添加到另一个对象上面的技术。
15. this的指向默认有两种指向，但是我们可以通过一些方法来修改这个this的指向。