# JavaScript面向对象

## 本章内容

* 类和对象
* 什么是面向对象
* 面向对象——封装
* 面向对象——继承

## 课程内容

### 1-2-1 类和对象

JavaScript中所有的数据都可以看成对象！

都是对象 == 面向对象？NO!

类和对象

* 类：类型的简称，数据类型的简称；是一个概念、代号，如人、电脑、飞机等等
* 对象：类的某个具体的实例，是一个实际存在的物体，如陈伟霆、dell电脑、波音747
* 类和对象，是面向对象编程的基础

### 1-2-2 什么是面向对象？

面向对象不是技术，而是一种处理问题的思路，一种编程思想。

面向对象，是将程序中的数据和功能进行封装，以达到代码复用性、低耦合、增强代码功能扩展。

面向对象已经被广泛应用于各种编程语言中，是当下流行的一种编程语言处理需求的理念。

面向对象的基础，就是类和对象的概念和使用！

### 1-2-3 面向对象——封装

#### 原始的封装模式：通过JSON数据来模拟一种数据类型

|  |
| --- |
| var People = {  name:””,  age:0,  eat:function(){  console.log(name + “吃饭了”);  }  } |

上面就是一个模拟的人的类型，正常情况下要根据这个类型的模板，来创建具体的实例对象！

|  |
| --- |
| var houZi = {}  houZi.name = “侯局长”;  houZi.age = 43;  houZi.eat = function() {  console.log(this.name + “吃饭了…”);  } |

数据已经封装完成，将name、age、eat()描述一个人的属性特征和行为操作被封装在一起，也就是将零散的数据封装整合到一起方便程序中进行统一的管理和使用。

但是上面的封装方式使用的时候一定要注意适用范围

* 比较适用于程序中某个类型只存在唯一实例对象的形式。
* 不适合某个类型有多个对象的形式，多个实例书写方式就会很麻烦，实例和类之间没有关联关系、实例和实例之间没有关联关系

#### 函数封装模式

另外一种情况下通过函数进行改进，函数通过返回一个JSON对象的形式来优化创建对象的复杂度，提高代码的复用性

|  |
| --- |
| function People(name, age) {  return {  name:name,  age:age  }  } |

在这样的情况下，创建对象就会变得非常简单

|  |
| --- |
| var leaderGao = People(“高经理”, 30);  var chaiQianChang = People(“拆迁常”, 35); |

这样的优化，一定程度上降低了代码的复杂度，提高了代码的可用性。不过只是对上一种方式的封装，依然体现不出leaderGao、chaiQianChang实例和People类型之间的关系

#### 构造函数

JavaScript提供了一种模拟面向对象的语法，通过函数的方式来模拟定义类型，定义好类型之后，通过new关键字来创建对象

备注：ES6以后提供了面向对象的语法结构！

|  |
| --- |
| 定义类型的语法结构：  function People (name, age){  this.name = name;  this.age = age;  this.eat = function() {  console.log(this.name + “吃饭了…”);  }  } |

此时，有了构造函数之后，我们对于创建对象有了新的看法

|  |
| --- |
| var jianChaJi = new People(“季检察长”, 50);  var chenHai = new People(“陈海”, 38); |

我们重新审视创建的对象jianChaJi和chenHai

通过new关键字创建的对象，就会有自己的一个constructor属性，指向对象的构造函数，也就是我们定义的类型；通过这样的方式来反映类型和对象之间的关系

|  |
| --- |
| console.log(jianChaJi.constructor);  console.log(chenHai.constructor); |

同样的，JS提供了另一个对象操作符号instanceof验证类型和对象之间的关系

|  |
| --- |
| console.log(jianChaJi instanceof People);  console.log(chenHai instanceof People); |

#### JS内存分析

程序是运行在计算中的进程，一旦程序要开始运行，必须要向操作系统申请内存空间以支持存取和计算程序运行过程中的各种数据

常规情况下，JS代码的运行，是基于浏览器引擎对象的。

* 沙盒隔离

由于浏览器的沙盒隔离，所以JS代码只能操作浏览器中的各种信息，如浏览器版本，访问的历史网页导航、访问的URL地址、访问的网页文档的操作等，JS不能直接操作浏览器之外的内容

* 内存分解

运行JS的引擎，在程序运行时，向操作系统申请内存。内存会被划分成一下的几个部分

|  |  |
| --- | --- |
| * 栈区 * 堆区 * 全局区 * 文字常量区 * 程序代码区 |  |

* 简要分析
* 栈区：由编译器自动分配释放 ，存放函数的参数值，局部变量的值等，内存的分配是连续的，类似于平时我们所说的栈，如果还不清楚，那么就把它想成数组，它的内存分配是连续分配的，即，所分配的内存是在一块连续的内存区域内．当我们声明变量时，那么编译器会自动接着当前栈区的结尾来分配内存．
* 堆区：一般由程序员分配释放， 若程序员不释放，程序结束时可能由操作系统回收．类似于链表，在内存中的分布不是连续的，它们是不同区域的内存块通过指针链接起来的．一旦某一节点从链中断开，我们要人为的把所断开的节点从内存中释放．
* 全局区：全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 程序结束后由系统释放
* 文字常量区：常量字符串就是放在这里的。 程序结束后由系统释放
* 程序代码区：存放函数体的二进制代码。
* 程序处理
* 常规情况下，声明的全局变量会存放在全局区，方便程序中的代码使用；但是局部变量和函数中的参数值会被存放在栈区，方便快速访问；对象则创建在堆内存区，方便程序的加载的稳定性。

|  |
| --- |
| var count = 1; # 存在在全局区  function subTotal(double price) {// price 存放在栈区  var discount = 0.8; // 函数运行后discount 存放在栈区  } |

* 字符串处理

字符串是程序中使用最频繁的一个对象。

所以各种编程语言对字符串进行了特殊的处理，增加了内存处理的字符串区[文字常量区]

通常，字符串一旦创建，就会在字符串文字常量区存储字符串的一个备份，下次再创建字符串时，首先在常量区查询是否存在，存在即使用；否则重新创建并备份到常量区

|  |
| --- |
| var s1 = “abc”;  var s2 = “abc”;  var s3 = new String(“abc”);  var s4 = new String(“abc”);  s1 == s2 ? true  s1 === s2 ? true  s1 == s3 ? true  s2 == s3 ? true  s1 === s3 ? false  s2 === s3 ? false  s3 == s4 ? false |

* JS程序中内存的管理问题

|  |
| --- |
| JS在设计之初，参考了Java的内存管理机制，使用垃圾自动回收机制，但是操作系统为了防止由于网页消耗太多内存导致系统崩溃，所以对浏览器使用内存进行了限制。  确保JS程序中，尽量少的出现全局变量；  一旦数据不再使用，通过设置null值释放引用； |

#### 构造函数模式下存在的问题

类的多个不同的实例对象，可能都会存在一些公共的特征或者行为，表现为实例对象的属性和函数/方法；

如chenHai和jianChaLi的职务都是属于检察官。按照构造函数创建的对象，会在每个对象中独立创建这样一个区域；造成了内存不必要的浪费。

* 原型对象：prototype

JS中提供了一个原型对象prototype，可以通过prototype原型进行对象公共扩展和继承使用，目的是为了提高代码的复用、降低代码的耦合并提升扩展性。

|  |
| --- |
|  |

原型对象的使用

|  |
| --- |
| var People = function(name, age) {  this.name = name;  this.age = age;  }  People.prototype.run = function() {  console.log(“跑步中…”)  } |

* 原型链：\_\_proto\_\_

JS中为了方便描述一个对象的创建过程，每个对象都有一个原型链\_\_proto\_\_，指向创建它的函数对象的prototype

通常将这个串起来的\_\_proto\_\_直到Object.prototpye.\_\_proto\_\_为null的处理链称为原型链

|  |
| --- |
|  |

#### 对象、函数、constructor、prototype、\_\_proto\_\_

|  |
| --- |
|  |

#### prototype验证

* A.isPrototypeOf(b)判断b对象是否为A类型创建

|  |
| --- |
|  |

* obj.hasOwnProperty(prop):判断prop是否Obj的本地属性[prototype上构建的属性为非本地属性]

|  |
| --- |
|  |

* prop in obj:判断某个属性Prop是否obj对象的属性[不区分是否本地属性]

|  |
| --- |
|  |

### 1-2-4 面向对象——继承

继承：体现的是A is a B的关系，同样也体现的是一种类型的关联关系

继承是为了提高对象之间的关联关系和提高代码的复用性和功能扩展性

JS中实现继承有非常多的方式，常见的方式如下：

* 通过apply/call的形式进行继承【熟悉】
* 通过prototype的方式进行继承【掌握】
* 直接通过prototype进行继承【熟悉】
* **通过空的中间对象进行间接的继承【必须掌握】**
* 拷贝继承【熟悉】

#### 1-4-1 通过apply/call的形式进行继承【构造函数绑定】

这是一种最简洁的方式，通过构造方法直接继承。

|  |
| --- |
| function Animal() {  this.species = “动物”;  } |

定义其他对象的类型

|  |
| --- |
| function Cat(name, color) {  Animal.apply(this, arguments);  this.name = name;  this.color = color;  } |

|  |
| --- |
| var cat1 = new Cat(“tom”, “white”);  cat1.species;// 动物 |

|  |
| --- |
| call和apply方法实现的效果一致，只是参数列表和类型不同而已。  Function.apply:  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）.    Function.call:  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. |

#### 1-4-2 prototype模式继承对象

通过prototype的形式，进行指定对象的继承。

|  |
| --- |
| var Animal = function() {  this.species = “动物”  } |
| var Cat = function(name, color) {  this.name = name;  this.color = color;  } |
| Cat.prototype = new Animal();  **Cat.prototype.constructor = Cat;**  var cat = new Cat(“tom”, “black”); |

|  |
| --- |
| 以上代码在继承过程中，通过修改Cat.prototype指向Animal来实现继承。但是同时也改变了Cat的原型链。为了修正手工指定Cat.prototype.constructor=Cat; |

#### 1-4-3 直接继承prototype

通过指定类型和被继承类型的prototype进行绑定来实现继承。

通过Prototype原型进行继承，这样的方式是在第二种方式基础上进行的优化。

|  |
| --- |
| var Animal = function(){}  Animal.prototype.species = “动物”  var Cat = function(name, color) {  this.name = name;  this.color = color;  }  Cat.prototype = Animal.prototype;  Cat.prototype.constructor = Cat;  var cat = new Cat(“tom”, “orange”); |

|  |
| --- |
| 以上代码在继承过程中，通过修改Cat.prototype和Animal.prototype绑定关系来实现的。这样的绑定关系回造成Cat和Animal之间的关联，修改Cat.prototype的属性，就会直接影响到Animal.prototype属性。 |

#### 1-4-4 通过空对象进行继承

第三种方式实现了继承，但是继承者和被继承者之间出现了prototype原型共同指向的问题，这里通过一个中间的空对象进行处理【结合第二种和第三种方式】，提升代码质量。

|  |
| --- |
| var Animal = function() {}  Animal.prototype.species = “动物”;  var Cat = function(name, age) {  this.name = name;  this.age = age;  }  var F = function(){};  F.prototype = Animal.prototype;  Cat.prototype = new F();  Cat.prototype.constructor = Cat;  ##### 实现了Cat和Animal之间的继承关系 |

通常在项目应用的时候，将继承关系封装成函数方便调用

|  |
| --- |
| function extends (Child, Parent) {  var F = function() {};//空对象  F.prototype = Parent.prototype;// 类型prototype关联  Child.prototype.constructor = Child;// 重新设置  Child.super = Parent.prototype;// 打开父对象的访问途径  } |

#### 1-4-5 拷贝继承

这里说的继承，已经不在是一种继承。而是一种内在的数据交换了。

|  |
| --- |
| var \_extend = function(Child, Parent) {  var \_p = Parent.prototype;  var \_c = Child.prototype;  for(var i in \_p){  c[i] = p[i]  }  \_c.super = \_p;  } |