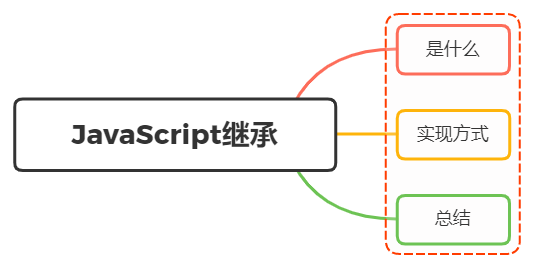
# 面试官：Javascript如何实现继承？



## 一、是什么

继承（inheritance）是面向对象软件技术当中的一个概念。

如果一个类别B“继承自”另一个类别A，就把这个B称为“A的子类”，而把A称为“B的父类别”也可以称“A是B的超类”

* 继承的优点

继承可以使得子类具有父类别的各种属性和方法，而不需要再次编写相同的代码

在子类别继承父类别的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类别的原有属性和方法，使其获得与父类别不同的功能

虽然JavaScript并不是真正的面向对象语言，但它天生的灵活性，使应用场景更加丰富

关于继承，我们举个形象的例子：

定义一个类（Class）叫汽车，汽车的属性包括颜色、轮胎、品牌、速度、排气量等

class Car{  
 constructor(color,speed){  
 this.color = color  
 this.speed = speed  
 // ...  
 }  
}

由汽车这个类可以派生出“轿车”和“货车”两个类，在汽车的基础属性上，为轿车添加一个后备厢、给货车添加一个大货箱

// 货车  
class Truck extends Car{  
 constructor(color,speed){  
 super(color,speed)  
 this.Container = true // 货箱  
 }  
}

这样轿车和货车就是不一样的，但是二者都属于汽车这个类，汽车、轿车继承了汽车的属性，而不需要再次在“轿车”中定义汽车已经有的属性

在“轿车”继承“汽车”的同时，也可以重新定义汽车的某些属性，并重写或覆盖某些属性和方法，使其获得与“汽车”这个父类不同的属性和方法

class Truck extends Car{  
 constructor(color,speed){  
 super(color,speed)  
 this.color = "black" //覆盖  
 this.Container = true // 货箱  
 }  
}

从这个例子中就能详细说明汽车、轿车以及卡车之间的继承关系

## 二、实现方式

下面给出JavaScripy常见的继承方式：

* 原型链继承
* 构造函数继承（借助 call）
* 组合继承
* 原型式继承
* 寄生式继承
* 寄生组合式继承

### 原型链继承

原型链继承是比较常见的继承方式之一，其中涉及的构造函数、原型和实例，三者之间存在着一定的关系，即每一个构造函数都有一个原型对象，原型对象又包含一个指向构造函数的指针，而实例则包含一个原型对象的指针

举个例子

function Parent() {  
 this.name = 'parent1';  
 this.play = [1, 2, 3]  
 }  
 function Child() {  
 this.type = 'child2';  
 }  
 Child1.prototype = new Parent();  
 console.log(new Child())

上面代码看似没问题，实际存在潜在问题

var s1 = new Child2();  
var s2 = new Child2();  
s1.play.push(4);  
console.log(s1.play, s2.play); // [1,2,3,4]

改变s1的play属性，会发现s2也跟着发生变化了，这是因为两个实例使用的是同一个原型对象，内存空间是共享的

### 构造函数继承

借助 call调用Parent函数

function Parent(){  
 this.name = 'parent1';  
}  
  
Parent.prototype.getName = function () {  
 return this.name;  
}  
  
function Child(){  
 Parent1.call(this);  
 this.type = 'child'  
}  
  
let child = new Child();  
console.log(child); // 没问题  
console.log(child.getName()); // 会报错

可以看到，父类原型对象中一旦存在父类之前自己定义的方法，那么子类将无法继承这些方法

相比第一种原型链继承方式，父类的引用属性不会被共享，优化了第一种继承方式的弊端，但是只能继承父类的实例属性和方法，不能继承原型属性或者方法

### 组合继承

前面我们讲到两种继承方式，各有优缺点。组合继承则将前两种方式继承起来

function Parent3 () {  
 this.name = 'parent3';  
 this.play = [1, 2, 3];  
}  
  
Parent3.prototype.getName = function () {  
 return this.name;  
}  
function Child3() {  
 // 第二次调用 Parent3()  
 Parent3.call(this);  
 this.type = 'child3';  
}  
  
// 第一次调用 Parent3()  
Child3.prototype = new Parent3();  
// 手动挂上构造器，指向自己的构造函数  
Child3.prototype.constructor = Child3;  
var s3 = new Child3();  
var s4 = new Child3();  
s3.play.push(4);  
console.log(s3.play, s4.play); // 不互相影响  
console.log(s3.getName()); // 正常输出'parent3'  
console.log(s4.getName()); // 正常输出'parent3'

这种方式看起来就没什么问题，方式一和方式二的问题都解决了，但是从上面代码我们也可以看到Parent3 执行了两次，造成了多构造一次的性能开销

### 原型式继承

这里主要借助Object.create方法实现普通对象的继承

同样举个例子

let parent4 = {  
 name: "parent4",  
 friends: ["p1", "p2", "p3"],  
 getName: function() {  
 return this.name;  
 }  
 };  
  
 let person4 = Object.create(parent4);  
 person4.name = "tom";  
 person4.friends.push("jerry");  
  
 let person5 = Object.create(parent4);  
 person5.friends.push("lucy");  
  
 console.log(person4.name); // tom  
 console.log(person4.name === person4.getName()); // true  
 console.log(person5.name); // parent4  
 console.log(person4.friends); // ["p1", "p2", "p3","jerry","lucy"]  
 console.log(person5.friends); // ["p1", "p2", "p3","jerry","lucy"]

这种继承方式的缺点也很明显，因为Object.create方法实现的是浅拷贝，多个实例的引用类型属性指向相同的内存，存在篡改的可能

### 寄生式继承

寄生式继承在上面继承基础上进行优化，利用这个浅拷贝的能力再进行增强，添加一些方法

let parent5 = {  
 name: "parent5",  
 friends: ["p1", "p2", "p3"],  
 getName: function() {  
 return this.name;  
 }  
};  
  
function clone(original) {  
 let clone = Object.create(original);  
 clone.getFriends = function() {  
 return this.friends;  
 };  
 return clone;  
}  
  
let person5 = clone(parent5);  
  
console.log(person5.getName()); // parent5  
console.log(person5.getFriends()); // ["p1", "p2", "p3"]

其优缺点也很明显，跟上面讲的原型式继承一样

### 寄生组合式继承

寄生组合式继承，借助解决普通对象的继承问题的Object.create 方法，在前面几种继承方式的优缺点基础上进行改造，这也是所有继承方式里面相对最优的继承方式

function clone (parent, child) {  
 // 这里改用 Object.create 就可以减少组合继承中多进行一次构造的过程  
 child.prototype = Object.create(parent.prototype);  
 child.prototype.constructor = child;  
}  
  
function Parent6() {  
 this.name = 'parent6';  
 this.play = [1, 2, 3];  
}  
Parent6.prototype.getName = function () {  
 return this.name;  
}  
function Child6() {  
 Parent6.call(this);  
 this.friends = 'child5';  
}  
  
clone(Parent6, Child6);  
  
Child6.prototype.getFriends = function () {  
 return this.friends;  
}  
  
let person6 = new Child6();  
console.log(person6); //{friends:"child5",name:"child5",play:[1,2,3],\_\_proto\_\_:Parent6}  
console.log(person6.getName()); // parent6  
console.log(person6.getFriends()); // child5

可以看到 person6 打印出来的结果，属性都得到了继承，方法也没问题

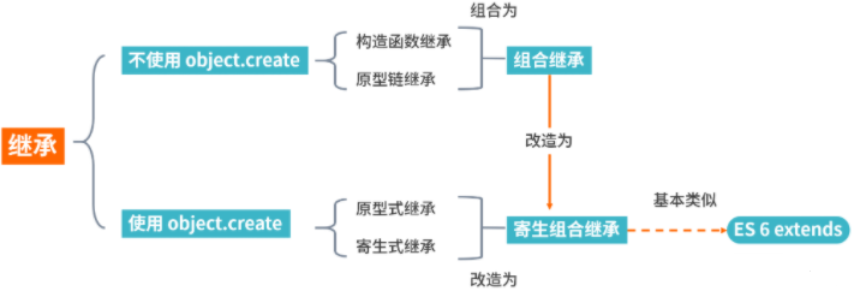
文章一开头，我们是使用ES6 中的extends关键字直接实现 JavaScript的继承

class Person {  
 constructor(name) {  
 this.name = name  
 }  
 // 原型方法  
 // 即 Person.prototype.getName = function() { }  
 // 下面可以简写为 getName() {...}  
 getName = function () {  
 console.log('Person:', this.name)  
 }  
}  
class Gamer extends Person {  
 constructor(name, age) {  
 // 子类中存在构造函数，则需要在使用“this”之前首先调用 super()。  
 super(name)  
 this.age = age  
 }  
}  
const asuna = new Gamer('Asuna', 20)  
asuna.getName() // 成功访问到父类的方法

利用babel工具进行转换，我们会发现extends实际采用的也是寄生组合继承方式，因此也证明了这种方式是较优的解决继承的方式

## 三、总结

下面以一张图作为总结：



通过Object.create 来划分不同的继承方式，最后的寄生式组合继承方式是通过组合继承改造之后的最优继承方式，而 extends 的语法糖和寄生组合继承的方式基本类似

## 相关链接

https://zh.wikipedia.org/wiki/%E7%BB%A7%E6%89%BF