###### 同步和异步

**两者区别：**

1.同步会阻塞代码执行，而异步不会。

2.比如alert是同步，setTimeout是异步。

**前端使用异步的场景：（这些场景的特点都是需要等待）**

1.定时任务：setTimeout,setInterval

2.网络请求：ajax请求，动态<img>加载

3.事件绑定

**例子**：

console.log(1);

console.log(setTimeout(function(){

console.log(2);

},1000));

console.log(3);

结果：

1

2850

3

2

解释：setTimeout执行后会立马返回一个数值id，但setTimeout中定义的函数会先暂存起来，继续执行后续代码，最后再执行暂存起来的函数。

###### 日期

Date.now(); //获取当前时间毫秒数

var dt = new Date();

dt.getTime(); //获取毫秒数

dt.getFullYear(); //年 ，四位数，真实的月份要加上1

dt.getMonth(); //月（0-11）

dt.getDate(); //日（0-31）

dt.getHours(); //小时（0-23）

dt.getMinutes(); //分钟（0-59）

dt.getSeconds(); //秒（0-59）

**获取2017-06-10格式的日期**

function formatDate(dt){

var year = dt.getFullYear();

var month = dt.getMonth() + 1;

var date = dt.getDate();

if(month < 10){

month = '0' + month;

}

if(date < 10){

date = '0' + date;

}

return year + '-' + month + '-' + date;

}

var dt = new Date();

var formatDate = formatDate(dt);

console.log(formatDate);

###### 随机数 、数组的随机排序

Math.random() 产生一个[0，1)之间的随机数，每次执行，小数点后的位数并不是一致的

**获取随机数，要求是长度一致的字符串格式**

var random = Math.random();

var random = random + "0000000000";

var random = random.slice(0,10); //要求长度为10

console.log(random);

**数组随机排序方法：**

**法一：**

var arr = [1,2,3,4,5,6,7];

function randomSort(arr){

for(var i = 0;i < arr.length;i++){

var index = parseInt(Math.random() \*　arr.length);

var temp = arr[index];

arr[index] = arr[i],

arr[i] = temp;

}

return arr;

}

console.log(randomSort(arr));

**法二：**

var arr = [1,2,3,4,5,6,7];

function randomSort(arr){

var newArr = [];

while(arr.length > 0){

var index = parseInt(Math.random() \*　arr.length);

newArr.push(arr[index]);

arr.splice(index,1);

}

return newArr;

}

console.log(randomSort(arr));

**法三：**

var arr = [1,2,3,4,5,6,7];

function randomSort(arr){

arr.sort(function(){

return Math.random() - 0.5;

});

return arr;

}

console.log(randomSort(arr));

###### JS面向对象——封装、继承（原型链继承、借用构造函数、组合继承）、多态

1. **封装**：

我们将一类实体共同具有的属性和行为提取出来抽象为一个类，将这些属性和行为都放在这个类里。就好像人类，可以具有name,sex,age等属性，同时也具有eat(),sleep()等行为，这就可以把他们封装成一个Person类。JS这种弱类型语言不像强类型语言中通过class关键字实现类的封装（ES6另谈），而是通过声明一个函数，在函数内部通过为this变量添加属性或者方法来实现对类添加属性和方法。

如：（Book例子）

function Book(id,name){

this.id = id;

this.name = name

}

**也可以在类的原型上添加属性和方法：**

Book.prototype.display = function(){};

//或者

Book.prototype = {

display : function(){}

};

通过this添加属性和在protoytpe中添加属性的**区别**：通过类创建一个新对象时，this指向的属性和方法都会得到相应的创建，而通过prototype继承的属性或方法是每个对象通过prototype访问到，所以每次创建一个新对象时这些属性和方法不会再次创建。

1. **继承——父类的特性传给子类**

**1.借用构造函数继承**

function Parent0(){

this.name = "parent0";

this.colors = ["red","blue","yellow"];

}

function Child0(){

**Parent0.call( this ); // 或apply**

this.type = "child0";

}

第6行，在子类（Child0）中执行父类（Parent0）的构造函数，通过这种调用，把父类构造函数的this指向为子类实例化对象引用，从而导致父类执行的时候父类里面的属性都会被挂载到子类的实例上去。

new Child0().name; // Parent0

new Child0().colors; // (3) ["red", "blue", "yellow"]

但是通过这种方式，父类原型上的东西是没法继承的

Parent0.prototype.sex = "男";

Parent0.prototype.say = function() {

console.log(" Oh,My God! ");

}

**new Child0().sex; // undefined**

**// Uncaught TypeError: (intermediate value).say is not a function**

**new Child0().say();**

缺点：Child1无法继承Parent1的原型对象，并没有真正的实现继承（部分继承）

**2.原型链式继承（借用原型链实现继承）**

function Parent1(){

this.name = "parent1";

this.colors = ["red","blue","yellow"];

}

function Child1(){

this.name = "child1";

}

**Child1.prototype = new Parent1();**

这种方式能否解决借用构造函数继承的缺点呢？来看下面代码，我们依然为父类的原型添加sex属性和say方法：

Parent1.prototype.sex = "男";

Parent1.prototype.say = function() {

console.log(" Oh,My God! ");

}

**new Child1().sex; // 男**

**new Child1().say(); // Oh,My God!**

这种方式确实解决了上面借用构造函数继承方式的缺点。

但是，这种方式仍有缺点，我们来看如下代码：

var s1 = new Child1();

s1.colors.push("black");

var s2 = new Child1();

**s1.colors; // (4) ["red", "blue", "yellow", "balck"]**

**s2.colors; // (4) ["red", "blue", "yellow", "balck"]**

我们实例化了两个Child1，在实例s1中为父类的colors属性push了一个颜色，但是s2也被跟着改变了。造成这种现象的原因就是原型链上中的原型对象它俩是共用的。

这不是我们想要的，s1和s2这个两个对象应该是隔离的，这是这种继承方式的缺点。

**3.组合式继承**

这里所谓的组合是指组合借用构造函数和原型链继承两种方式。

function Parent2(){

this.name = "parent2";

this.colors = ["red","blue","yellow"];

}

function Child2(){

**Parent2.call(this);**

this.type = "child2";

}

**Child2.prototype = new Parent2()**

注意第6，9行，这种方式结合了借用构造函数继承和原型链继承的有点，能否解决上述两个实例对象没有被隔离的问题呢？

var s1 = new Child2();

s1.colors.push("black");

var s2 = new Child2();

s1.colors; // (4) ["red", "blue", "yellow", "balck"]

s2.colors; // (3) ["red", "blue", "yellow"]

可以看到，s2和s1两个实例对象已经被隔离了。

但这种方式仍有缺点。父类的构造函数被执行了两次，第一次是Child2.prototype = new Parent2()，第二次是在实例化的时候，这是没有必要的。

**3.1组合式继承优化1**

直接把父类的原型对象赋给子类的原型对象

function Parent3(){

this.name = "parent3";

this.colors = ["red","blue","yellow"];

}

Parent3.prototype.sex = "男";

Parent3.prototype.say = function(){console.log("Oh, My God！")}

function Child3(){

**Parent3.call(this);**

this.type = "child3";

}

**Child3.prototype = Parent3.prototype;**

var s1 = new Child3();

var s2 = new Child3();

console.log(s1, s2);

但是，我们来看如下代码：

console.log(s1 instanceof Child3); // true

console.log(s1 instanceof Parent3); // true

可以看到，我们无法区分实例对象s1到底是由Child3直接实例化的还是Parent3直接实例化的。用instanceof关键字来判断是否是某个对象的实例就基本无效了。

我们还可以用.constructor来观察对象是不是某个类的实例：

console.log(s1.constructor.name); // Parent3

从这里可以看到，s1的构造函数居然是父类，而不是子类Child3，这显然不是我们想要的。

**3.2组合式继承优化2**

这是继承的最完美方式

function Parent4(){

this.name = "parent4";

this.colors = ["red","blue","yellow"];

}

Parent4.prototype.sex = "男";

Parent4.prototype.say = function(){console.log("Oh, My God！")}

function Child4(){

**Parent4.call(this);**

this.type = "child4";

}

**Child4.prototype = Object.create(Parent4.prototype)；**

**Child4.prototype.constructor = Child4;**

Object.create是一种创建对象的方式，它会创建一个中间对象

var p = {name: "p"}

var obj = Object.create(p)

// Object.create({ name: "p" })

通过这种方式创建对象，新创建的对象obj的原型就是p，同时obj也拥有了属性name，这个新创建的中间对象的原型对象就是它的参数。

这种方式解决了上面的所有问题，是继承的最完美实现方式。

**4.ES6中继承**

Class 可以通过**extends**关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。

class Parent {

}

class Child1 extends Parent {

constructor(x, y, colors) {

super(x, y); // 调用父类的constructor(x, y)

this.colors = colors;

}

toString() {

return this.colors + ' ' + super.toString(); // 调用父类的toString()

}

}

上面代码中，constructor方法和toString方法之中，都出现了super关键字，它在这里表示父类的构造函数，用来新建父类的this对象。

子类必须在constructor方法中调用super方法，否则新建实例时会报错。如果子类没有定义constructor方法，这个方法会被默认添加，不管有没有显式定义，任何一个子类都有constructor方法。

ES5 的继承，实质是先创造子类的实例对象this，然后再将父类的方法添加到this上面（Parent.apply(this)）。ES6 的继承机制完全不同，实质是先创造父类的实例对象this（所以必须先调用super方法），然后再用子类的构造函数修改this。

1. **多态（多种调用方式）**

摘自设计模式：就是同一个方法多种调用方式。比如在js中对传入的参数做判断以实现多种调用方式，如定义一个add方法

function add(){

var arg = arguments,

len = arg.length;

switch(len){

case 0 : return 10;

case 1 : return + arg[0];

case 2 :return arg[0] + arg[1];

}

}

console.log(add()); //10

console.log(add(5)); //15

console.log(add(6,7)); //13

转化为类的形式：

function Add(){

function zero(){

return 10

}

function one(num){

return 10 + num;

}

function two(num1,num2){

return num1 + num2;

}

function add(){

var arg = arguments,

len = arg.length;

switch(len){

case 0 : return zero();

case 1 : return one(arg[0]);

case 2 : return two(arg[0],arg[1]);

}

}

var A = new Add();

console.log(A.add());

console.log(A.add(5));

console.log(A.add(6,7));