###### 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

**Math.random() 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);

###### 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**
2. **原型链继承**

简单来说，就是直接将父类的一个实例赋给子类的原型。

function SuperType(){

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

}

function SubType(){

}

SubType.prototype = new SuperType();

var instance1 = new SubType();

instance1.colors.push("black");

console.log(instance1.colors);//"red,blue,green,black"

var instance2 = new SubType();

console.log(instance1.colors);//"red,blue,green,black"

上例中，SubType实例化出来的对象都会共享这一个colors属性，所以**最大问题**是子类无法通过父类创建私有属性，实践中很少会单独使用原型链。

1. **借用构造函数，**思想是在子类型构造函数的内部调用超类型构造函数。

function Person(name){

this.name=name;

this.className="person"

}

Person.prototype.getName=function(){

console.log(this.name)

}

function Man(name){

Person.apply(this,arguments)

}

var man1=new Man("Davin");

var man2=new Man("Jack");

>man1.name

>"Davin"

>man2.name

>"Jack"

**>man1.getName() //1 报错**

>man1 instanceof Person

>true

这样一来，每new一个子类的实例，构造函数执行完后，都会有自己的一份资源(name)。但是这种办法只能继承父类构造函数中声明的实例属性，并没有继承父类原型的属性和方法，所以就找不到getName方法，所以1处会报错。为了同时继承父类原型，从而诞生了组合继承的方式:

1. **组合继承（原型链+借用构造函数）**弥补了以上两者的缺点，融合了优点。

function Person(name){

this.name=name||"default name"; //1

this.className="person"

}

Person.prototype.getName=function(){

console.log(this.name)

}

function Man(name){

Person.apply(this,arguments)

}

//继承原型

Man.prototype = new Person();

var man1=new Man("Davin");

> man1.name

>"Davin"

> man1.getName()

>"Davin"

这样不仅会继承构造函数中的属性，也会复制父类原型链中的属性。

**缺点：父类会被调用两次，且Man.prototype = new Person(); 这句执行后，Man的原型如下：**

**> Man.prototype**

**> {name: "default name", className: "person"}**

也就是说Man的原型中已经有了一个name属性，而之后创建man1时传给构造的函数的name则是通过this重新定义了一个name属性，相当于只是覆盖掉了原型的name属性（原型中的name依然还在），这样很不优雅。（即调用子类型构造函数时会重写属性）

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));