构造函数和原型

构造函数

> 创建对象的三种方式：

字面量方式

```
const o ={
name : 'pink'
}
```

new创建

```
const obj = new Object({uname: 'pink'})//实例化
```

自定义构造函数创建对象

> 构造函数是特殊的函数，用来初始化对象

```
function Pig (name ,age, gender ) {
this.name = name
this.age = age
this.gender = gener
//不需要写return 返回的就是一个对象因为有new实现返回新对象
}

const Peppa = new Pig('佩奇',6,'女')
```

为了区分构造函数和其他函数：约定

命名以大写字母开始

创建对象的时候一定要new一下

实例对象和实例方法

实例成员：实例对象上的属性和方法就是实例成员

在实例对象中新加的属性和方法也算实例成员

在构造函数的属性和方法叫做静态成员（静态属性和静态方法）只有对构造函数才能使用的

```
function Pig (a, b){}
Pig.fun = () => {console.log(这是一个静态方法)}
Pig.con = '这是一个静态属性'
```

基本包装类型

> 在js底层中，对基本的数据类型进行了包装，所以这些基本数据类型也有方法和属性

```
const str = 'pink'
包装成了：
const str = new String('pink')
```

js中所有数据基本都可以由构造函数创建

内置构造函数

引用类型：Object , Array ,RegExp , Date

包装类型：String ,Number ,Boolean ….

Object

```
const user = new Object ()
```

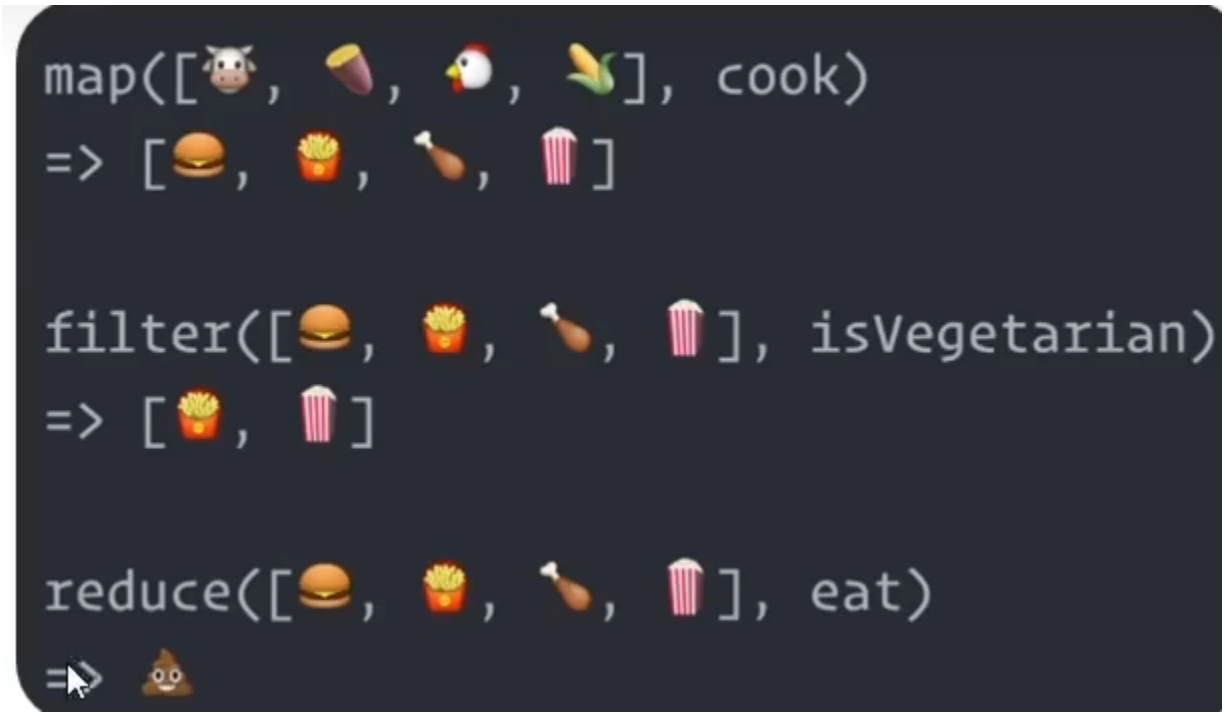
```
//静态方法
Object.keys//只能对Object用
Object.keys(user)//o是实例化对象
Object.values(user)

//拷贝
const newuser = {}
Object.assign(newuser,user)
Object.assign(user,{gender :female})//用来给对象添加属性
```

Array

```
cosnt arr = new Array()
```

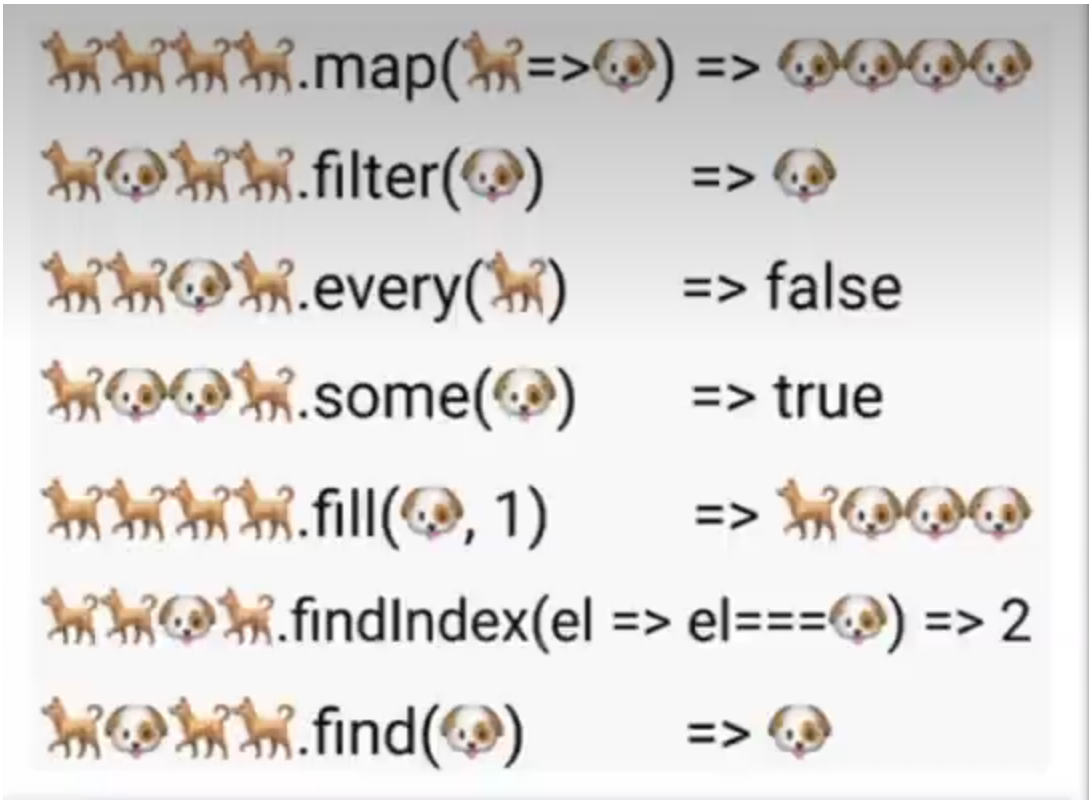
```
//实例方法
arr.forEach()//遍历数组，不返回数组，用来查找遍历元素
arr.filter() //过滤数组
arr.map()//迭代数组，返回数组，用于处理并得到新的数组
arr.reduce(function(上一次的值prev，当前的值current){
return prev + current},起始值start)//用于返回累计处理的结果，常用于求和（返回prev + current + start)
arr.join()
arr.find(callback(){})
arr.some()
arr.concat ()
arr.sort()
arr.splice()
arr.reverse()
arr.findIndex()
```



reduce :

	上一次值pre	当前值current	返回值
1	start	arr[0]	start + arr[0]
2	start + arr[0]	arr[1]	start + arr[0] +arr[1]
3	start + arr[0] +arr[1]	arr[2]	start + arr[0] +arr[1] +arr[2]

> ⚠️注意如果遍历的数组里面放的是对象，初始值要写0，否则就会变成数加对象



String

```
//实例方法
str = 'pink,red'
str.split('')//['pink','red'] 和.join(',')相反
str.substring(indexstart[,indexEnd])//返回一个子集,前闭后开
str.substr()//避免使用
str.startsWith(检测的索引号)//是否用给定的字符串开头
str.endsWith()
str.includes ()//判断字符串是否包含另一个字符串
```

原型

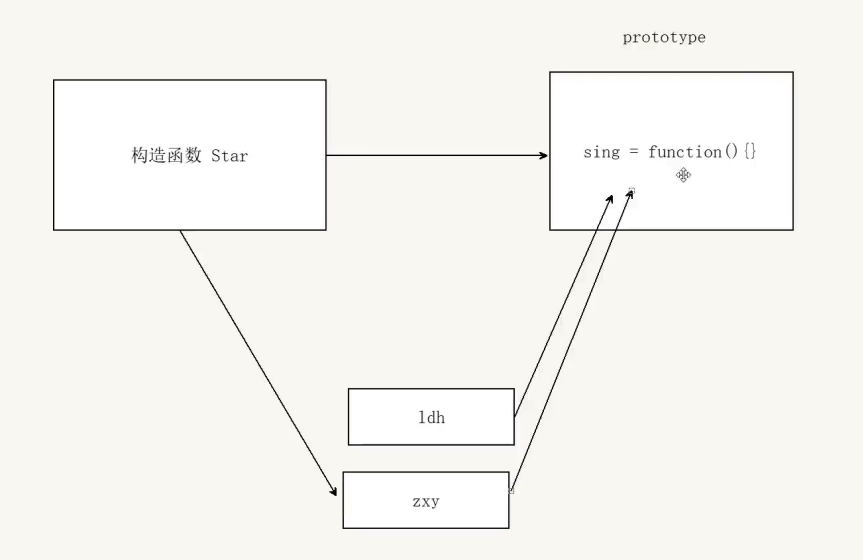
> 解决构造函数里面会浪费内存的问题（就是同一个函数再不同的对象里面不是同一个地址）

prototype

> 每个构造函数都有一个prototype属性，指向另一个对象，也成为**原型对象**
> ，这个对象可以挂载函数，实现函数的共享

```
function Star(uname,age) {
this.uname = uname
this.age = age
//this.sing = function () {}这里的函数每次实例化都会开辟新的空间
}
Star.prototype.sing = function (){//挂载在原型上就不会再次开辟
  console.log('唱歌')
}

const ldh = new Star('ldh',55)
const zxy = new Star('zxy',58)
console.log(ldh.sing === zxy.sing)//true 这个函数是共享的
```



总结：公共属性写到构造函数里面，公共方法写在原型里面

> 构造函数和对象原型里面的this都是指向当前实例化的对象

demo

给数组拓展方法

```
Array.prototype.max= function () {
  return Math.max(...this)//展开运算符
}
Array.prototype.sum = function() {
  return this.reduce((prev,item) => prev + item ,0)
}
```

constructor

> 构造器constructor，原型对象prototype里面都有一个constructor属性，指向构造函数，就是指回去了


```
Star.prototype.constructor === Star//true
```

注意 ⚠️

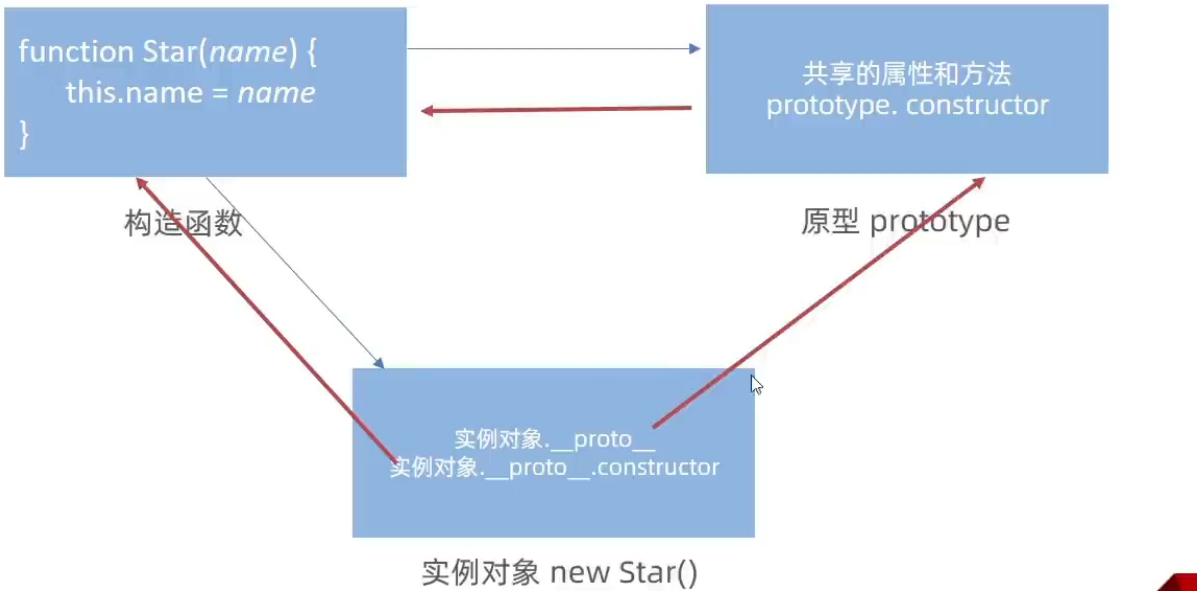
```
Array.prototype = {
max :function (){}
sum :function (){}
}//这种方法会找不到爹constructor
```

正确写法

```
Array.prototype = {
constructor:Star,
max :function (){},
sum :function (){}
}
```

对象原型 _ _proto_ _

> 实例化对象是如何访问原型prototype的?注意区分原型对象和对象原型
> __proto__是只读的，[[prototype]]意义相同




```
Star.prototype === ldh.__proto__//true
ldh.__proto__.prototype === Star
```

原型继承

> js中大多是借助原型对象来实现的

```
const Person = {
eyes : 2,
head : 1
}

function Women () {}
const pink = New Man ()
Woman.prototype = Person//可以继承Person的属性
Woman.prototype.constructor = Woman //把她指回原型
```

📌出现问题，因为women和man的原型是一样的，所以给Women的原型挂载的函数挂载在Person上，Man也会继承：

```
Women.prototype.baby = function () {}//不能这样做
Man.baby//存在
```

继续构造函数

```
//父类
function Person() {
this.eyes = 2 ,
this.heads = 1
}

//子类
Man.prototype = New Person()
Woman.prototype = New Person()//这样就隔离了两个prototype就可以挂载baby方法了
Woman.prototype.baby = function() {}
```

> 子类的原型 = New 父类

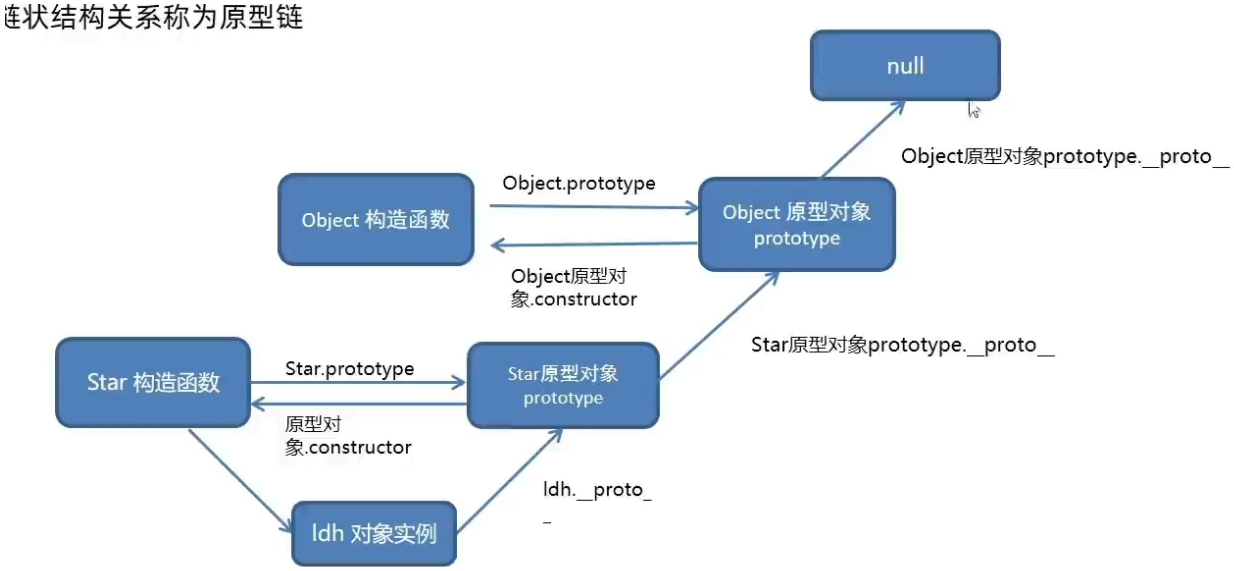
js的原生缺陷导致js的继承比较不好用

原型链

> 类比作用域链，这是一种查找规则

每个构造函数的prototype（原型对象）里的的__proto__（对象原型0指向Object.prototype，Object.prototype.__proto__是null

链状结构关系称为原型链



instanceof

> 检测某个对象是否在另一个对象的原型链下 （是否属于）

```
ldh instanceof Object //true
ldh instanceof Array //false
Array instanceof Object //ture
```