# 第21课：面向对象编程初级

主讲老师：万章

# 目录

# 构造函数，原型和继承

# 什么是构造函数?

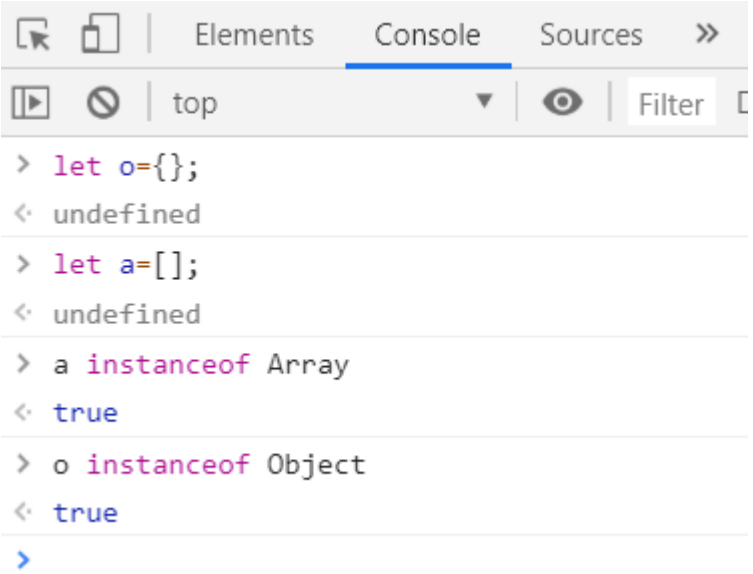
构造函数就是用来创造新对象的(函数,数组,正则表达式,日期等都是对象)，它必须用过关键字NEW来创造,如果将构造函数用作普通函数的话,往往不会正常工作的．按照一惯的约定，我们开发者把构造函数的首字母大写用作辨别．一个构造函数创造的对象被称为该构造函数的**实例**

常见的构造函数:

1. Object()
2. Array()
3. RegExp()
4. Function()
5. Date()

变量　instanceof　构造函数

可以辨别该变量的数据是否是由改构造函数所创造



# 什么是构造函数?

每一个对象都拥有一个_proto_属性(即为prototype原型属性),该属性内部有一个不可枚举的属性constructor,constructor的属性的值是一个函数对象(即为该对象的构造函数)

```
> a
<- ▼[]
    length: 0
    ▼__proto__: Array(0)
      ▶concat: ƒ concat()
      ▼constructor: ƒ Array()
          arguments: (...)
          caller: (...)
        ▶from: ƒ from()
        ▶isArray: ƒ isArray()
          length: 1
          name: "Array"
        ▶of: ƒ of()
        ▶prototype: [constructor: ƒ, concat: ƒ,…
          Symbol(Symbol.species): (...)
        ▶get Symbol(Symbol.species): ƒ [Symbol.…
        ▶__proto__: ƒ ()
        ▶[[Scopes]]: Scopes[0]
      ▶copyWithin: ƒ copyWithin()
      ▶entries: ƒ entries()
      ▶every: ƒ every()
      ▶fill: ƒ fill()
      ▶filter: ƒ filter()
      ▶find: ƒ find()
      ▶findIndex: ƒ findIndex()
      ▶flat: ƒ flat()
      ▶flatMap: ƒ flatMap()
```

```
> let o={};
<- undefined
> o
<- ▼{}
    ▼__proto__:
      ▼constructor: ƒ Object()
          arguments: (...)
        ▶assign: ƒ assign()
          caller: (...)
        ▶create: ƒ create()
        ▶defineProperties: ƒ defineProperties()
        ▶defineProperty: ƒ defineProperty()
        ▶entries: ƒ entries()
        ▶freeze: ƒ freeze()
        ▶fromEntries: ƒ fromEntries()
        ▶getOwnPropertyDescriptor: ƒ getOwnProp…
        ▶getOwnPropertyDescriptors: ƒ getOwnPro…
        ▶getOwnPropertyNames: ƒ getOwnPropertyN…
        ▶getOwnPropertySymbols: ƒ getOwnPropert…
        ▶getPrototypeOf: ƒ getPrototypeOf()
        ▶is: ƒ is()
        ▶isExtensible: ƒ isExtensible()
        ▶isFrozen: ƒ isFrozen()
        ▶isSealed: ƒ isSealed()
        ▶keys: ƒ keys()
          length: 1
          name: "Object"
        ▶preventExtensions: ƒ preventExtensions…
        ▶prototype: {constructor: ƒ, __defineGe…
        ▶seal: ƒ seal()
        ▶setPrototypeOf: ƒ setPrototypeOf()
```

# 什么是构造函数？

```
> let o={};
<· undefined
> o
<· ▼{}
    ▼__proto__:
      ▶constructor: ƒ Object()
      ▶hasOwnProperty: ƒ hasOwnProperty()
      ▶isPrototypeOf: ƒ isPrototypeOf()
      ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶toLocaleString: ƒ toLocaleString()
      ▶toString: ƒ toString()
      ▶valueOf: ƒ valueOf()
      ▶__defineGetter__: ƒ __defineGetter__()
      ▶__defineSetter__: ƒ __defineSetter__()
      ▶__lookupGetter__: ƒ __lookupGetter__()
      ▶__lookupSetter__: ƒ __lookupSetter__()
      ▶get __proto__: ƒ __proto__()
      ▶set __proto__: ƒ __proto__()
> o.__proto__.constructor
<· ƒ Object() { [native code] }
> o.constructor
<· ƒ Object() { [native code] }
>
```

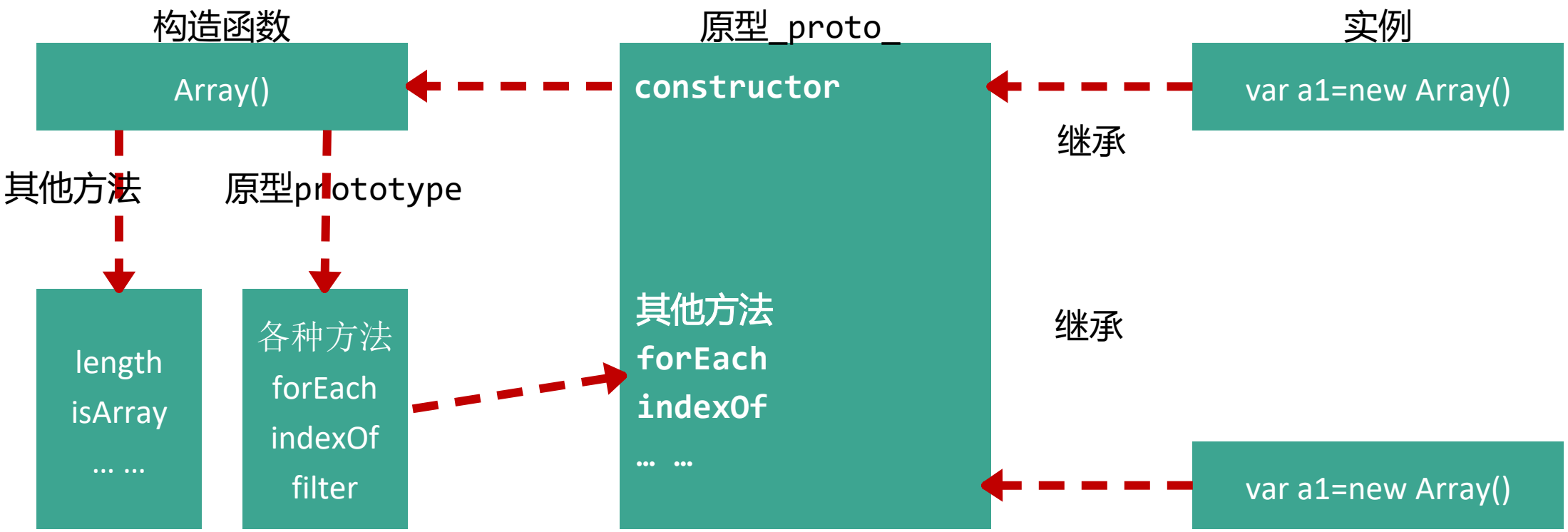
# 原型和构造函数的关系

每一个构造函数都有一个prototype属性，该属性的值是一个对象，对象内的各种方法，就是该构造函数创造示例时，子元素继承过去的方法

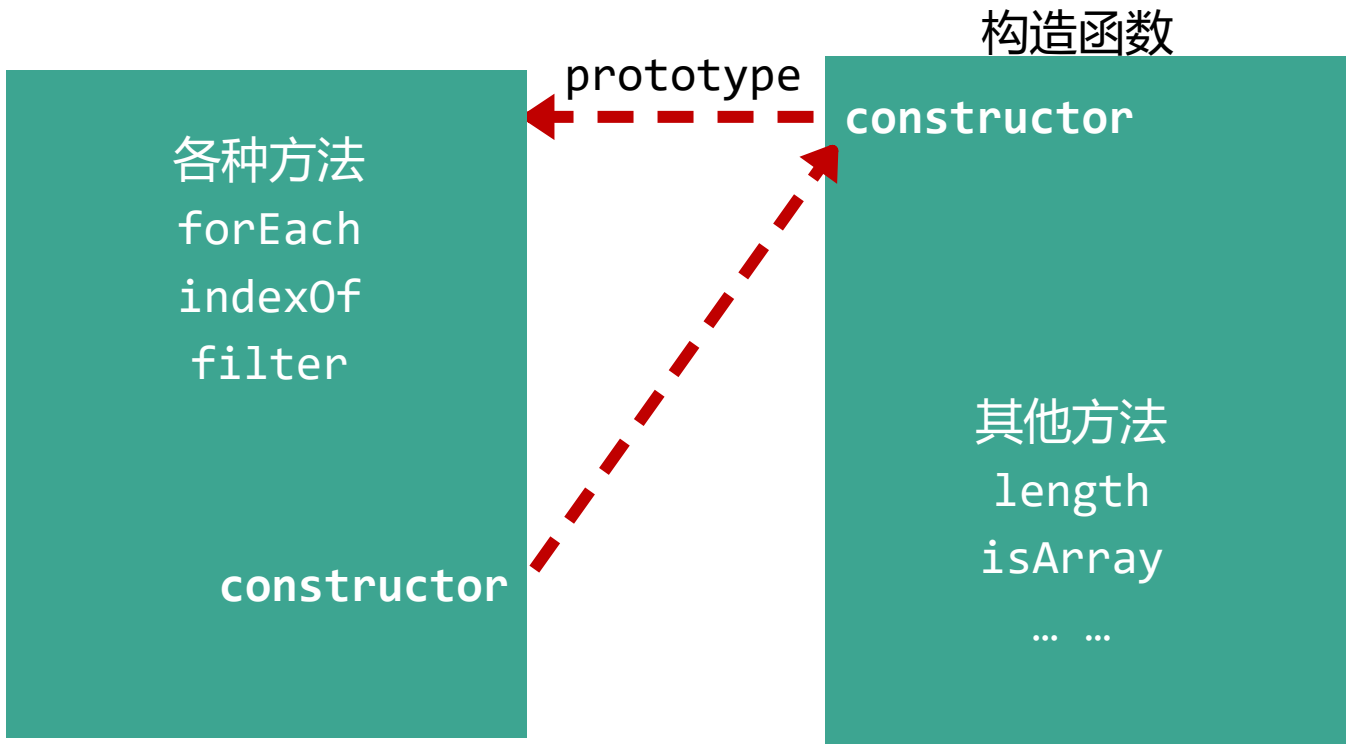
```
Elements  Console  Sources  Network
top                Filter
> let a=[];
  undefined
> a
  ▼[]
    length: 0
    ▼__proto__: Array(0)
      ▶concat: ƒ concat()
      ▶constructor: ƒ Array()
      ▶copyWithin: ƒ copyWithin()
      ▶entries: ƒ entries()
      ▶every: ƒ every()
      ▶fill: ƒ fill()
      ▶filter: ƒ filter()
      ▶find: ƒ find()
      ▶findIndex: ƒ findIndex()
      ▶flat: ƒ flat()
      ▶flatMap: ƒ flatMap()
      ▶forEach: ƒ forEach()
      ▶includes: ƒ includes()
      ▶indexOf: ƒ indexOf()
      ▶join: ƒ join()
      ▶keys: ƒ keys()
      ▶lastIndexOf: ƒ lastIndexOf()
       length: 0
      ▶map: ƒ map()
      ▶pop: ƒ pop()
      ▶push: ƒ push()
      ▶reduce: ƒ reduce()
      ▶reduceRight: ƒ reduceRight()
      ▶reverse: ƒ reverse()
      ▶shift: ƒ shift()
      ▶slice: ƒ slice()
      ▶some: ƒ some()
      ▶sort: ƒ sort()
      ▶splice: ƒ splice()
      ▶toLocaleString: ƒ toLocaleString()
      ▶toString: ƒ toString()
      ▶unshift: ƒ unshift()
      ▶values: ƒ values()
      ▶Symbol(Symbol.iterator): ƒ values()
      ▶Symbol(Symbol.unscopables): {copyWithin: tr
      ▶__proto__: Object
```

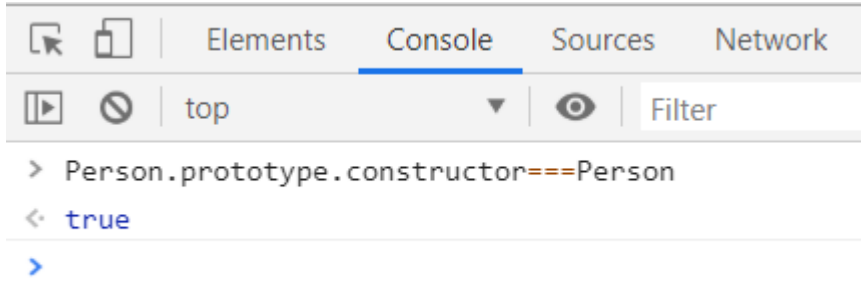
```
▼constructor: ƒ Array()
   arguments: (...)
   caller: (...)
 ▶from: ƒ from()
 ▶isArray: ƒ isArray()
   length: 1
   name: "Array"
 ▶of: ƒ of()
 ▼prototype: Array(0)
   ▶concat: ƒ concat()
   ▶constructor: ƒ Array()
   ▶copyWithin: ƒ copyWithin()
   ▶entries: ƒ entries()
   ▶every: ƒ every()
   ▶fill: ƒ fill()
   ▶filter: ƒ filter()
   ▶find: ƒ find()
   ▶findIndex: ƒ findIndex()
   ▶flat: ƒ flat()
   ▶flatMap: ƒ flatMap()
   ▶forEach: ƒ forEach()
   ▶includes: ƒ includes()
   ▶indexOf: ƒ indexOf()
   ▶join: ƒ join()
   ▶keys: ƒ keys()
   ▶lastIndexOf: ƒ lastIndexOf()
    length: 0
   ▶map: ƒ map()
   ▶pop: ƒ pop()
   ▶push: ƒ push()
   ▶reduce: ƒ reduce()
   ▶reduceRight: ƒ reduceRight()
   ▶reverse: ƒ reverse()
   ▶shift: ƒ shift()
   ▶slice: ƒ slice()
   ▶some: ƒ some()
   ▶sort: ƒ sort()
   ▶splice: ƒ splice()
   ▶toLocaleString: ƒ toLocaleString()
   ▶toString: ƒ toString()
   ▶unshift: ƒ unshift()
```

# 原型和构造函数的关系


构造函数
Array()
原型_proto_
constructor
实例
var a1=new Array()
继承
其他方法
原型prototype
length
isArray
… …
各种方法
forEach
indexOf
filter
其他方法
forEach
indexOf
… …
继承
var a1=new Array()

# 原型和构造函数的关系


构造函数
prototype
constructor
各种方法
forEach
indexOf
filter
constructor
其他方法
length
isArray
… …


Elements
Console
Sources
Network
top
Filter
> Person.prototype.constructor===Person
<· true
>

# 工厂模式

# 普通的对象创建模式

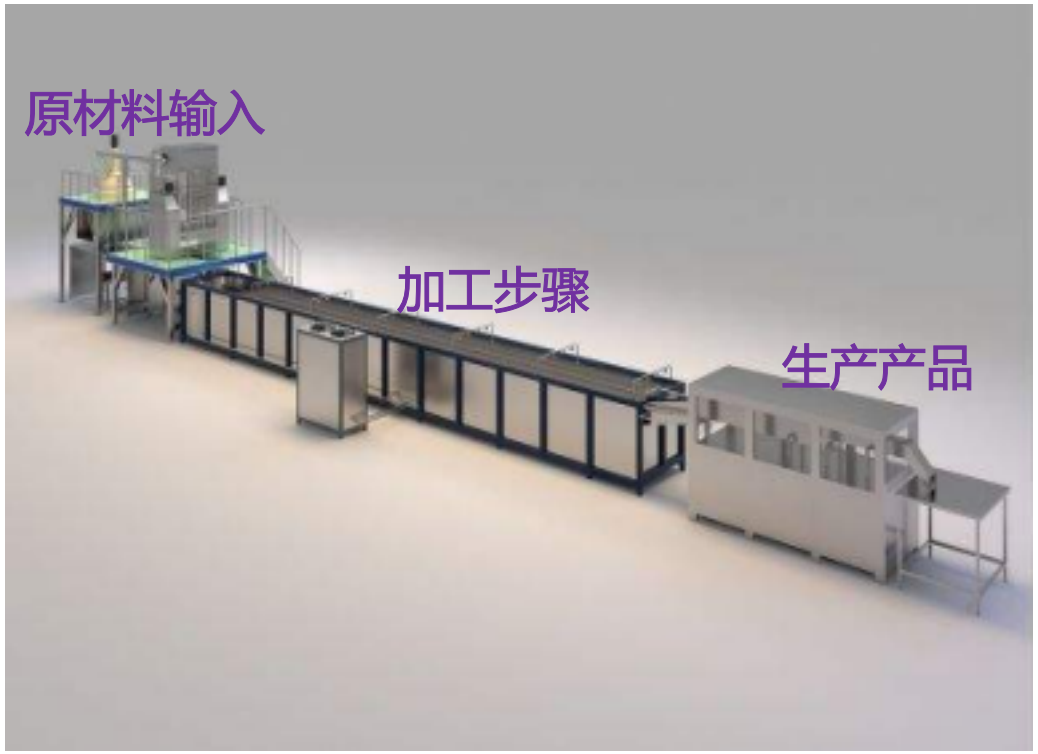
```
var wanzhang = {
    name: "万章",
    age: 18,
    company: "潭州教育",
    sayName: function () {
        alert(this.name);
    }
}
```

```
var yinshi = {
    name: "银时",
    age: 18,
    company: "潭州教育",
    sayName: function () {
        alert(this.name);
    }
}
```

```
var afei = {
    name: "阿飞",
    age: 18,
    company: "潭州教育",
    sayName: function () {
        alert(this.name);
    }
}
```

当我们在创建大量的结构相同的对象时，工作量大且重复性高

# 工厂模式的概念模型




```
function createPerson(name, age, company) {
    var obj = {};
    obj.name = name;
    obj.age = age;
    obj.company = company;
    obj.sayName = function () {
        alert(this.name);
    }
    return obj;
}
```

辣条工厂
（工厂名称）

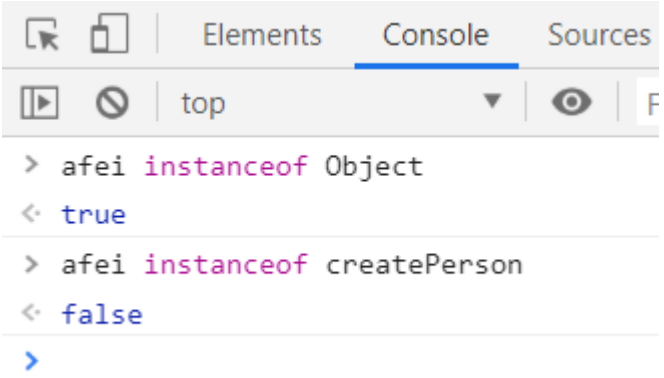
函数createPerson()能够根据接受的参数来构建一个包含所有必要信息的Person 对象。可以无数次地调用这个函数，而每次它都会返回一个包含三个属性一个方法的对象

# 工厂模式的问题

```
var afei = createPerson("阿飞", 18, "潭州教育");
var wanzhang = createPerson("万章", 18, "潭州教育");
var yinshi = createPerson("银时", 18, "潭州教育");
```

因为createPerson内创造对象的方式还是对象字面量，所以就相当于是new Object创造一样

这就导致createPerson函数创造的所有的对象的构造函数还是Object，这样咱们就**无法得知某些对象是不是同一个函数创造出来的了**




# 构造函数模式

# 构造函数模式

```
function Person(name, age,company) {
    this.name = name;
    this.age = age;
    this.company = company;
    this.sayName = function () {
        alert(this.name);
    }
}
```

在构造函数模式中，Person()函数取代了createPerson()函数。Person函数和createPerson函数存在以下差别：

没有显式地创建对象；

直接将属性和方法赋给了this 对象；

没有return 语句。

此外，还应该注意到函数名Person 使用的是大写字母P。按照惯例，构造函数始终都应该以一个大写字母开头，而非构造函数则应该以一个小写字母开头。这个做法借鉴自其他OO 语言，主要是为了区别于ECMAScript中的其他函数；

**构造函数本身也是函数，只不过可以用来创建对象而已!!!!!!**

# 构造函数模式

```
function Person(name, age,company) {
    this.name = name;
    this.age = age;
    this.company = company;
    this.sayName = function () {
        alert(this.name);
    }
}
```

```
let afei = new Person("阿飞", 18, "潭州教育");
let wanzhang = new Person("万章", 18, "潭州教育");
let yinshi = new Person("银时", 18, "潭州教育");
```

```
function Person(name, age,company) {
    let o=new Object();//此处只是为了模拟创造对象的过程，其实该对象是由函数Person创造的
    let this=o;//此处只是为了表明这个this是指每一次咱们新生成的对象本身. this是不能作为新的变量名称
    this.name = name;
    this.age = age;
    this.company = company;
    this.sayName = function () {
        alert(this.name);
    }
    return o;
}
```

要创建Person 的新实例，必须使用new 操作符。以这种方式调用构造函数实际上会经历以下4个步骤：

（1）创建一个新对象；

（2）将构造函数的作用域赋给新对象（因此this 就指向了这个新对象）；

（3）执行构造函数中的代码（为这个新对象添加属性）；

（4）返回新对象。

# 构造函数模式

```
> wanzhang instanceof Person
  true
> yinshi instanceof Person
  true
> afei instanceof Person
  true
> wanzhang instanceof Object
  true
> yinshi instanceof Object
  true
>
```

```
> afei
  ▼Person {name: "阿飞", age: 18, company: "潭州教育", sayName: ƒ}
      age: 18
      company: "潭州教育"
      name: "阿飞"
    ▶sayName: ƒ ()
    ▼__proto__:
      ▶constructor: ƒ Person(name, age,company)
      ▶__proto__: Object
> wanzhang
  ▼Person {name: "万章", age: 18, company: "潭州教育", sayName: ƒ}
      age: 18
      company: "潭州教育"
      name: "万章"
    ▶sayName: ƒ ()
    ▼__proto__:
      ▶constructor: ƒ Person(name, age,company)
      ▶__proto__: Object
> yinshi
  ▼Person {name: "银时", age: 18, company: "潭州教育", sayName: ƒ}
      age: 18
      company: "潭州教育"
      name: "银时"
    ▶sayName: ƒ ()
    ▼__proto__:
      ▶constructor: ƒ Person(name, age,company)
      ▶__proto__: Object
>
```

我们在这个例子中创建的所有对象既是Object 的实例，同时也是Person的实例，这一点通过instanceof 操作符可以得到验证。

之所以wanzhang,yinshi都是Object的实例是因为所有的对象都继承自Object

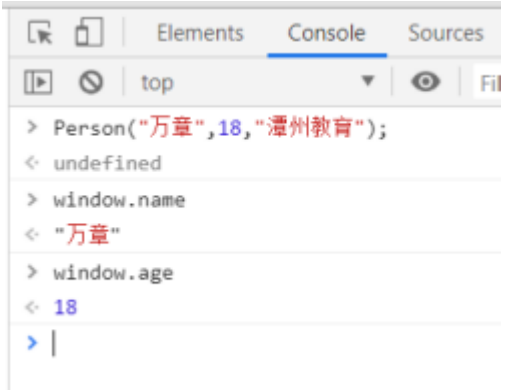
# 构造函数模式的深入辨析1

**构造函数与其他函数的唯一区别，就在于调用它们的方式不同。**不过，构造函数毕竟也是函数，不存在定义构造函数的特殊语法。

- ◆ 任何函数，只要通过new操作符来调用，那它就可以作为构造函数；
- ◆ 任何函数，如果不通过new操作符来调用，那它跟普通函数也不会有什么两样

```
function Person(name, age,company) {
    this.name = name;
    this.age = age;
    this.company = company;
    this.sayName = function () {
        alert(this.name);
    }
}
```

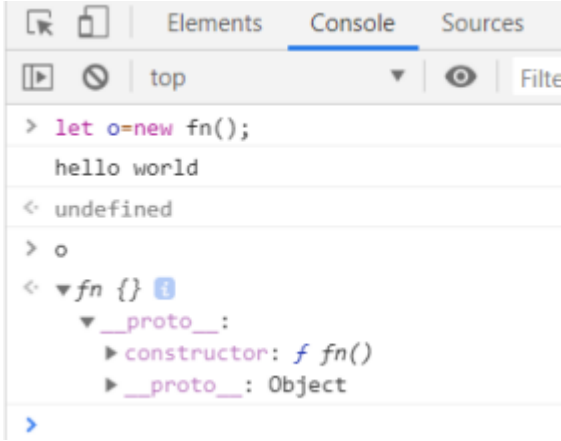
```
function fn(){
    console.log("hello world");
}
```




直接运行构造函数,那么构造函数里面的this自动指向window

把普通函数当做构造函数使用时1:首先函数内的所有代码会自动执行一次

2:然后返回一个新的对象,该对象的constructor就是这个普通函数



# 构造函数模式的深入辨析2

构造函数模式虽然好用，但也并非没有缺点。使用构造函数的主要问题，**就是每个方法都要在每个实例上重新创建一遍**。

在前面的例子中，yinshi 和wanzhang 都有一个名为sayName()的方法，但那两个方法不是同一个Function的实例。不要忘了——ECMAScript 中的函数是对象，因此每定义一个函数，也就是实例化了一个对象。

```
function Person(name, age,company) {
    this.name = name;
    this.age = age;
    this.company = company;
    this.sayName = function () {
        alert(this.name);
    }
}
```


```
function Person(name, age,company) {
    this.name = name;
    this.age = age;
    this.company = company;
    this.sayName = new Function ("alert(this.name)");
}
```

从逻辑角度讲，此时的构造函数也可以这样定义

# 构造函数模式的深入辨析2

```
function Person(name, age,company) {
    this.name = name;
    this.age = age;
    this.company = company;
    this.sayName = fn;
}

function fn(){
    console.log(this.name);
}
```

```
Elements  Console  Sources  Network
top                Filter
> yinshi.sayName
<- ƒ fn(){
            console.log(this.name);
        }
> wanzhang.sayName
<- ƒ fn(){
            console.log(this.name);
        }
> yinshi.sayName===wanzhang.sayName
<- true
>
```

在这个例子中，我们把sayName()函数的定义转移到了构造函数外部。而在构造函数内部，我们将sayName 属性设置成等于全局的sayName 函数。这样一来，由于sayName 包含的是一个指向函数的指针，因此yinshi和wanzhang对象就共享了在全局作用域中定义的同一个sayName()函数。

新问题：在全局作用域中定义的函数实际上只能被某个对象调用，这让全局作用域有点名不副实。如果对象需要定义很多方法，那么就要定义很多个全局函数，于是我们这个自定义的引用类型就丝毫没有封装性可言