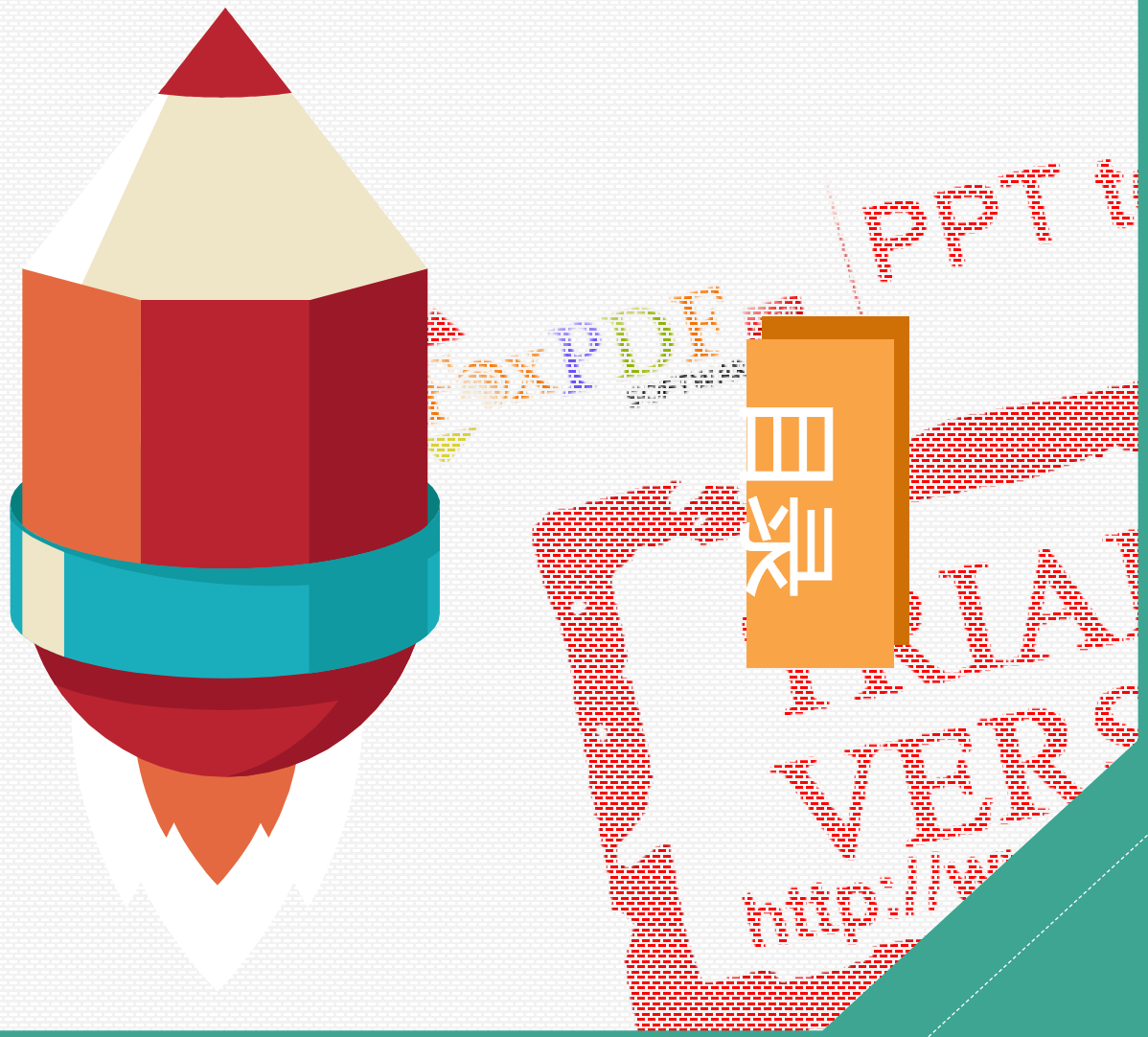

第23课：面向对象编程中级
主讲老师：万章



目录

- 组合使用构造函数和原型
- 动态原型模式
- 寄生构造函数模式
- 稳妥构造函数模式
- 原型继承

组合使用构造函数和原型

组合使用构造函数和原型

创建自定义类型的最常见方式，就是组合使用构造函数模式与原型模式。

构造函数模式用于定义实例属性，而原型模式用于定义方法和共享的属性。结果，每个实例都会有自己的一份实例属性的副本，但同时又共享着对方法的引用，最大限度地节省了内存

另外，这种混成模式还支持向构造函数传递参数；可谓是集两种模式之长

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.girlfriend=["石原里美", "新恒结衣"];
    this.company="潭州教育";
    this,sayName=function () {
        console.log(this.name);
    }
}
let wanzhang = new Person("万章", 18);
let yinshi = new Person("银时", 18);
```

构造函数模式

```
function Person(){}

Person.prototype = {
    constructor: Person,
    name:"万章",
    age:"18",
    girlfriend:["石原里美","新垣结衣"],
    company: "潭州教育",
    sayName: function () {
        console.log(this.name);
    }
}

let wanzhang = new Person();
let yinshi = new Person();
```

原型模式

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.girlfriend=["石原里美", "新恒结衣"]
}

Person.prototype = {
    constructor: Person,
    company: "潭州教育",
    sayName: function () {
        console.log(this.name);
    }
}

let wanzhang = new Person("万章", 18);
let yinshi = new Person("银时", 18);
```

组合模式

组合使用构造函数和原型

组合使用构造函数和原型时有这么几个注意点：

1. 对于对象内后期需要修改的引用类型的继承的值，这个值以构造函数的this.属性名称的方式来实现继承，这样可以避免某个示例操作继承来的对象而引起其他实例的变化

这种构造函数与原型混成的模式，是目前在ECMAScript中使用最广泛、认同度最高的一种创建自定义类型的方法。可以说，这是用来定义引用类型的一种默认模式

动态原型模式

动态原型模式

动态原型模式把所有信息都封装在了构造函数中，而通过在构造函数中初始化原型（仅在必要的情况下），又保持了同时使用构造函数和原型的优点。可以通过检查某个应该存在的方法是否有效，来决定是否需要初始化原型

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.company = "潭州教育";
    if (typeof this.sayName !== "function") {
        Person.prototype.sayName = function () {
            console.log(this.name);
        }
    }
}

let wanzhang = new Person("万章", 18);
let yinshi = new Person("银时", 18);
```

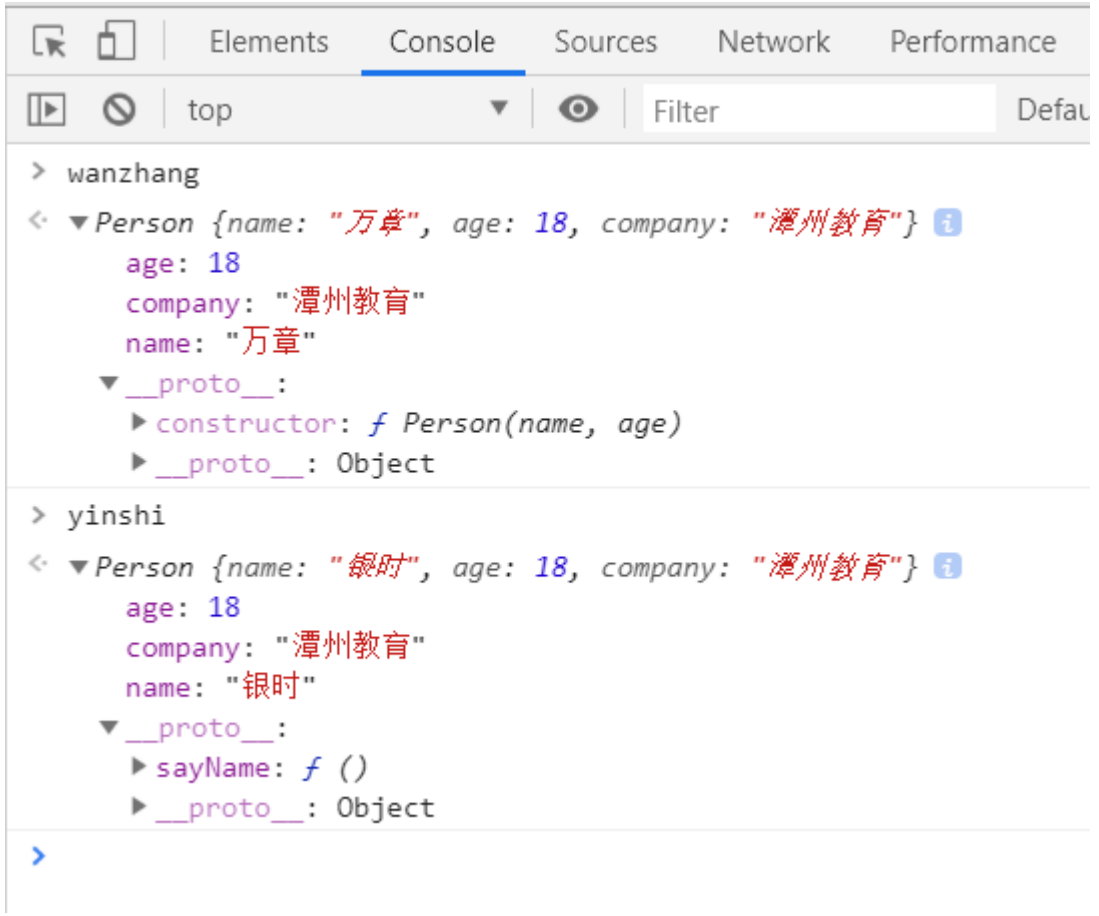
这里只在sayName()方法不存在的情况下，才会将它添加到原型中。这段代码只会在初次调用构造函数时才会执行。此后，原型已经完成初始化，不需要再做什么修改了

动态原型模式

使用动态原型模式时，不能使用对象字面量重写原型。前面已经解释过了，如果在已经创建了实例的情况下重写原型，那么就会切断现有实例与新原型之间的联系。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.company="潭州教育";
    if (typeof this.sayName !== "function") {
        Person.prototype = {
            sayName: function () {
                console.log(this.name);
            }
        }
    }
}

let wanzhang = new Person("万章", 18);
let yinshi = new Person("银时", 18);
```



寄生构造函数模式

寄生构造函数模式

这种模式的基本思想是创建一个函数，该函数的作用仅仅是封装创建对象的代码，然后再返回新创建的对象；但从表面上看，这个函数又很像是典型的构造函数

```
function Person(name, age) {
    var o = new Object();
    o.name = name;
    o.age = age;
    o.company = "潭州教育";
    o.sayName = function () {
        alert(this.name);
    };
    return o;
}

let wanzhang = new Person("万章", 18);
let yinshi = new Person("银时", 18);
```

寄生构造函数模式

在这个例子中，Person 函数创建了一个新对象，并以相应的属性和方法初始化该对象，然后又返回了这个对象。**除了使用new 操作符并把使用的包装函数叫做构造函数之外，这个模式跟工厂模式其实是一模一样的**

```
> yinshi instanceof Object
< true
> yinshi instanceof Person
< false
>
```

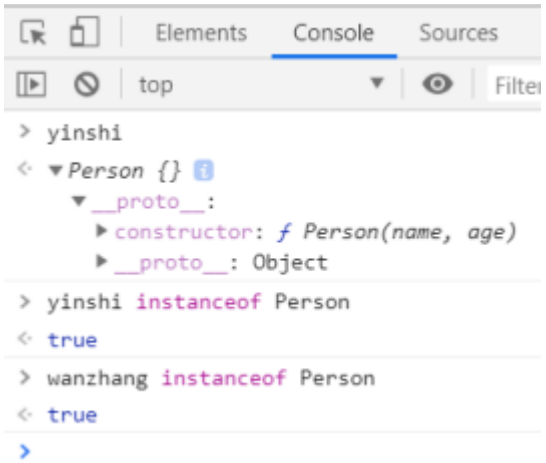
寄生构造函数模式

构造函数在不返回值的情况下，默认会返回新对象实例。而通过在构造函数的末尾添加一个return 语句，可以重写调用构造函数时返回的值。

```
function Person(name, age) {
    var o = new Object();
    o.name = name;
    o.age = age;
    o.company = "潭州教育";
    o.sayName = function () {
        alert(this.name);
    };

}

let wanzhang = new Person("万章", 18);
let yinshi = new Person("银时", 18);
```



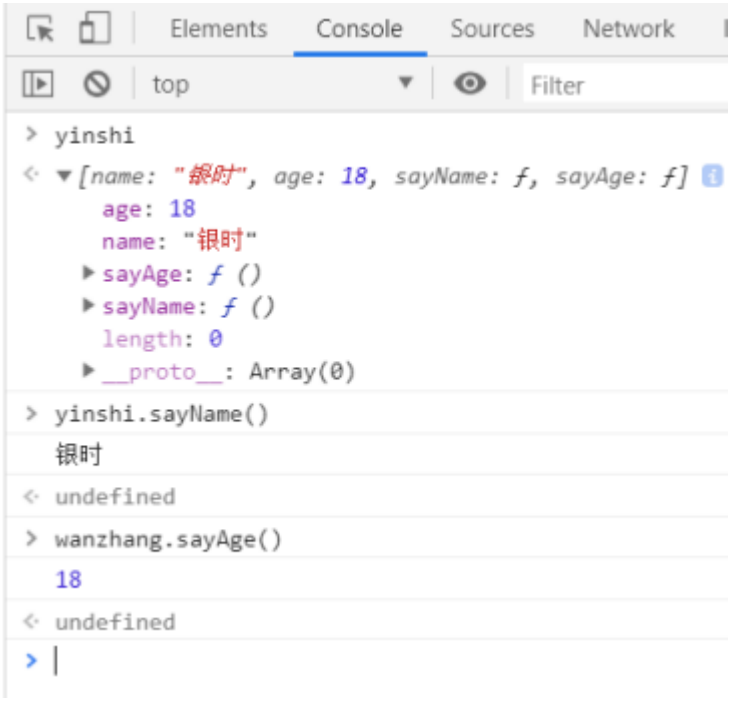
如果不返回一个对象的话，那么就如同是对一个普通的函数调用的new操作符，此时Person的每一行代码照样会执行,只是丝毫不会影响新的对象

寄生构造函数模式

这个模式可以在特殊的情况下用来为对象创建构造函数(可以将现在已经有的构造函数拿来用)。假设我们想创建一个具有额外方法的特殊数组。由于直接修改Array 构造函数造成的后果和影响太大,我们就可以用这个方式来实现

```
function Person(name, age) {
    var a = new Array();
    a.name=name;
    a.age=age;
    a.sayName = function () {
        console.log(this.name);
    };
    a.sayAge=function(){
        console.log(this.age);
    }
    return a;

}
let wanzhang = new Person("万章", 18);
let yinshi = new Person("银时", 18);
```



此模式的问题：

返回的对象与构造函数(Person)或者与构造函数的原型属性之间没有关系；也就是说，构造函数返回的对象与在构造函数外部创建的对象没有什么不同。为此，不能依赖instanceof操作符来确定对象类型。

由于存在上述问题，我们建议在可以使用其他模式的情况下，不要使用这种模式。

稳妥构造函数模式

稳妥构造函数模式

所谓稳妥对象，指的是没有公共属性，而且其方法也不引用this 的对象。稳妥对象最适合在一些安全的环境中（这些环境中会禁止使用this 和new），或者在防止数据被其他应用程序改动时使用

```
function Person(name, age) {
    var a = new Array();
    a.sayName = function () {
        console.log(name);
    };
    a.sayAge = function () {
        console.log(age);
    }
    return a;
}
let wanzhang = Person("万章", 18);
let yinshi = Person("银时", 18);
```

稳妥构造函数遵循与寄生构造函数类似的模式，但有两点不同：

一是新创建对象的实例方法不引用this

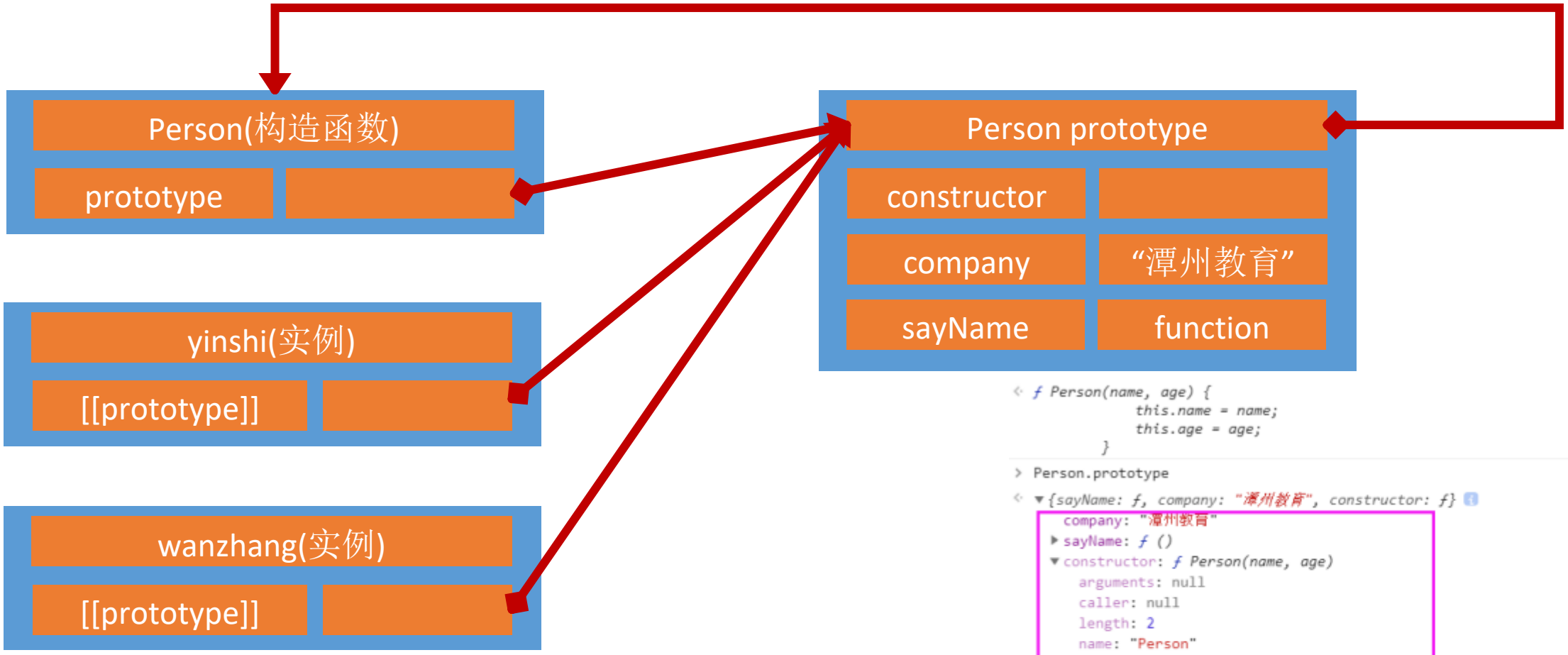
二是不使用new 操作符调用构造函数

在以这种模式创建的对象中，除了使用sayName()方法之外，没有其他办法访问name 的值,和闭包的理念类似

用稳妥构造函数模式创建的对象与构造函数之间也没有什么关系，因此instanceof 操作符对这种对象也没有意义。

继承

原型链深入分析




```
ƒ Person(name, age) {
        this.name = name;
        this.age = age;
    }
> Person.prototype
{sayName: ƒ, company: "潭州教育", constructor: ƒ}
  company: "潭州教育"
▶ sayName: ƒ ()
▼ constructor: ƒ Person(name, age)
    arguments: null
    caller: null
    length: 2
    name: "Person"
  ▼ prototype:
      company: "潭州教育"
    ▶ sayName: ƒ ()
    ▶ constructor: ƒ Person(name, age)
    ▶ __proto__: Object
  ▶ __proto__: ƒ ()
    [[FunctionLocation]]: demo.html:15
  ▶ [[Scopes]]: Scopes[2]
▶ __proto__: Object
```

构造函数、原型和实例的关系：每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个指向原型对象的内部指针

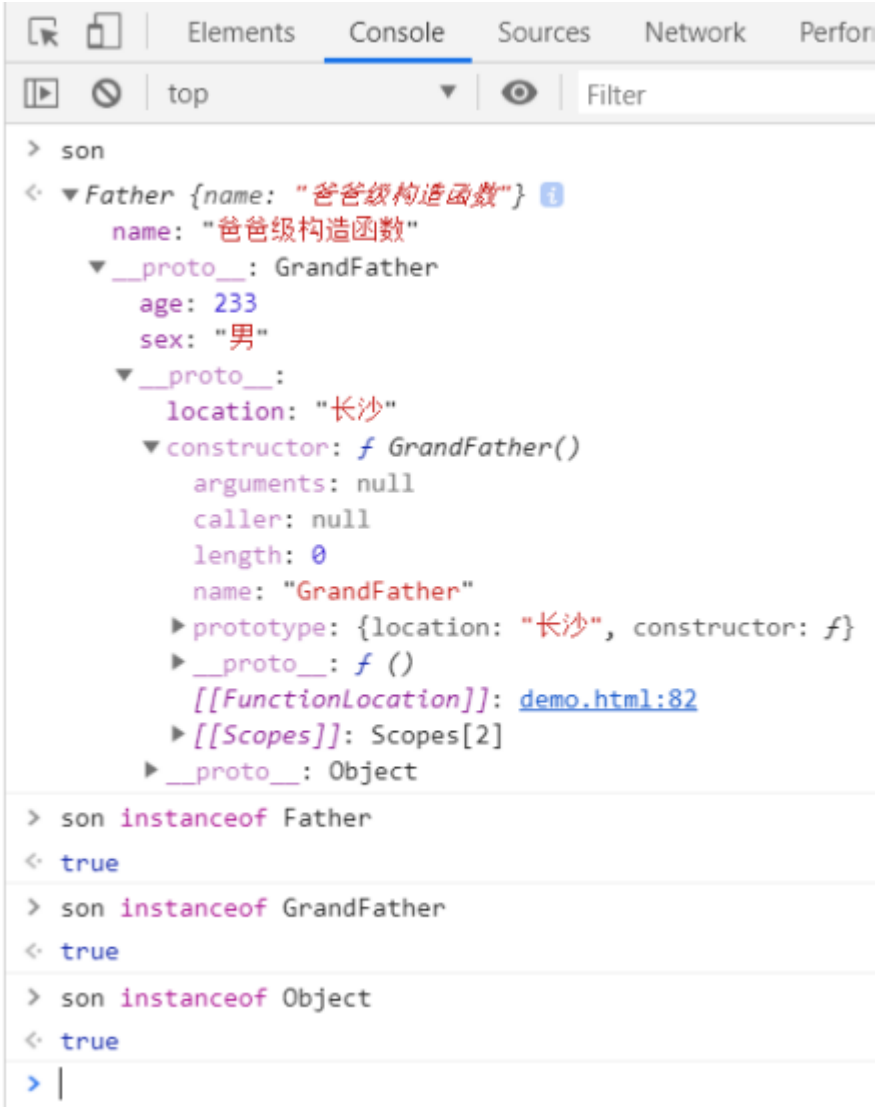
原型链深入分析

假如我们让原型对象等于另一个类型的实例，结果会怎么样呢？

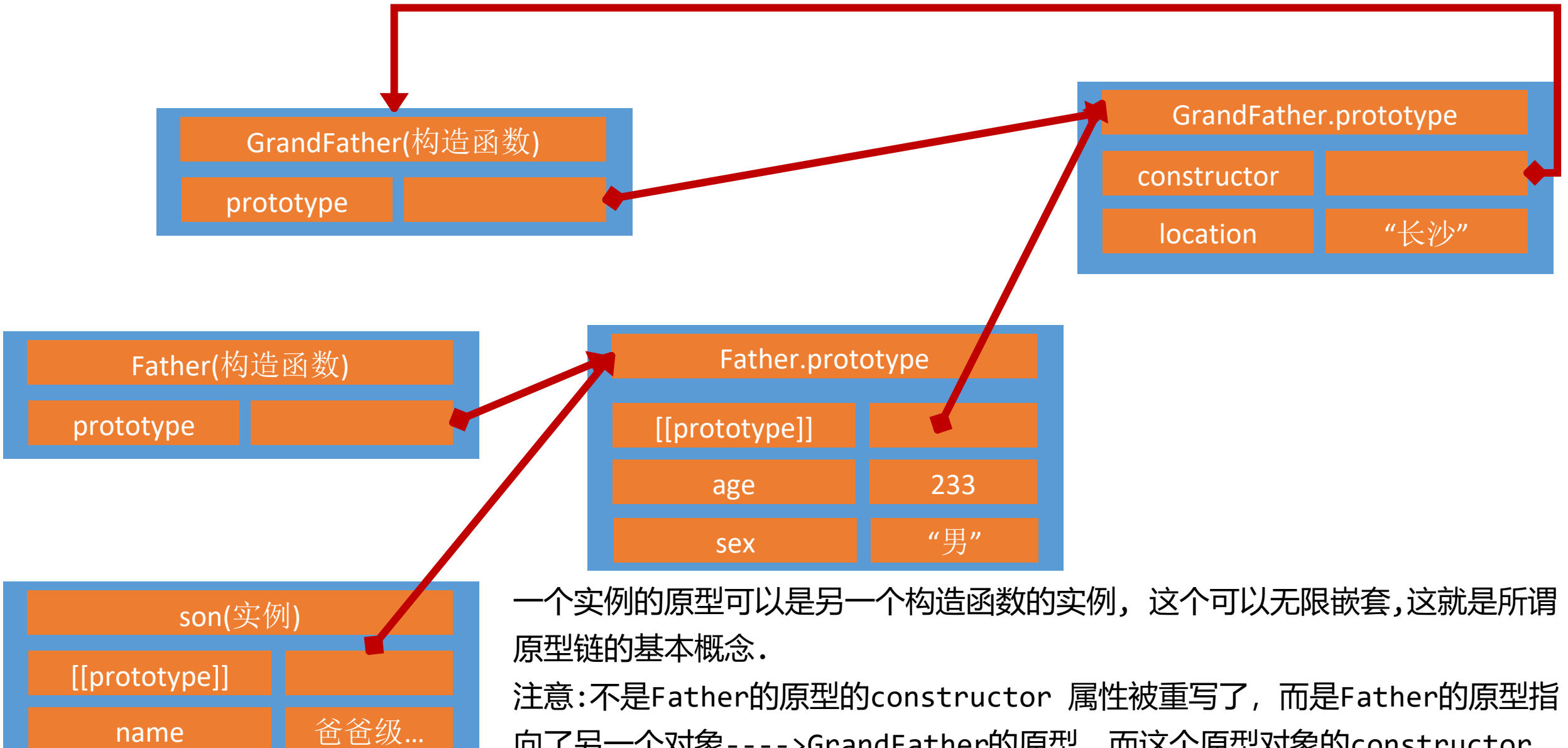
```
function GrandFather(){
    this.age=233;
}
GrandFather.prototype.location="长沙"

function Father(){
    this.name="爸爸级构造函数"
}
Father.prototype=new GrandFather();
Father.prototype.sex="男";

let son=new Father();
```



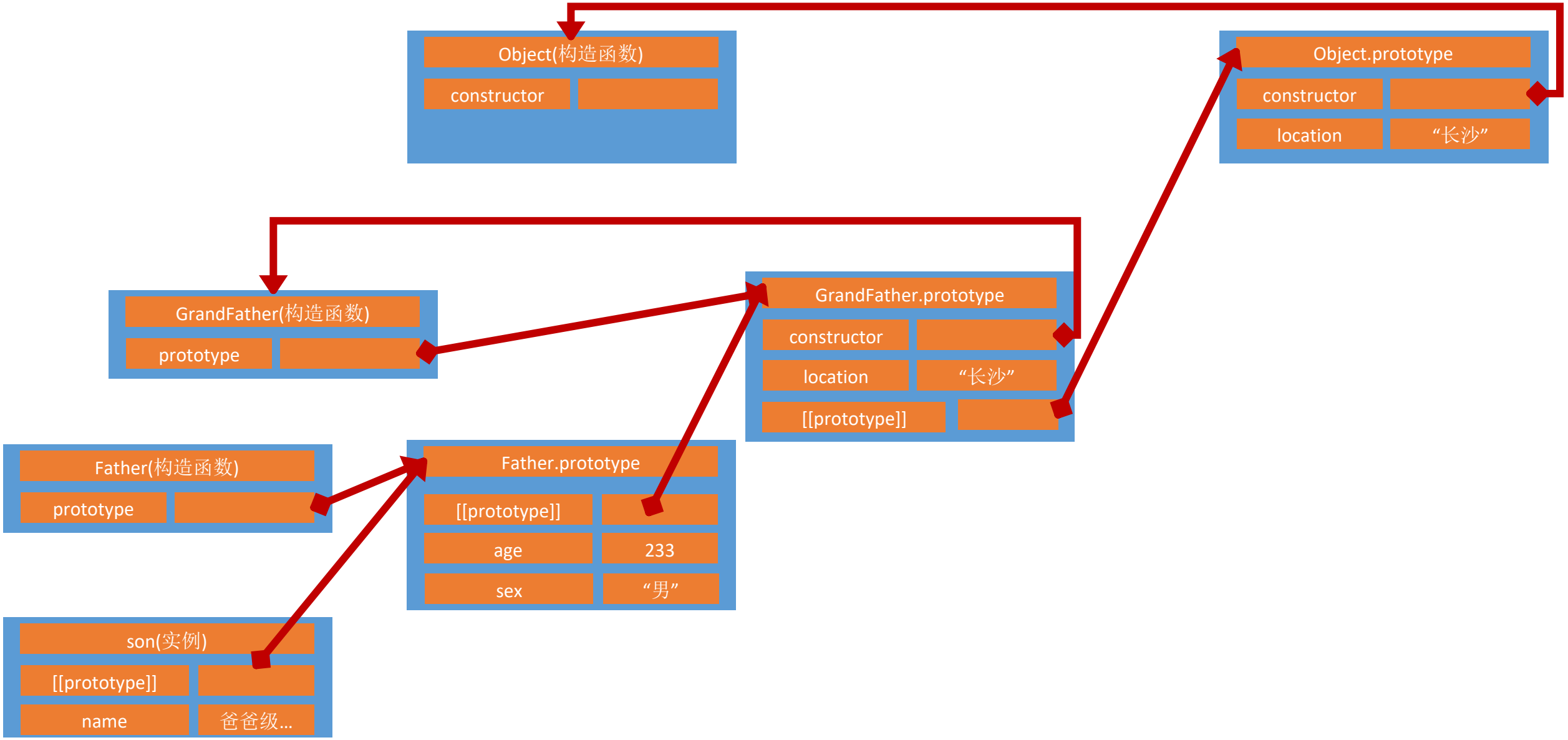
原型链深入分析



一个实例的原型可以是另一个构造函数的实例，这个可以无限嵌套,这就是所谓原型链的基本概念.

注意:不是Father的原型的constructor 属性被重写了，而是Father的原型指向了另一个对象---->GrandFather的原型，而这个原型对象的constructor

原型链深入分析(究极版本)所有的对象继承自Object



检测原型与实例之间的关系

可以通过两种方式来确定原型和实例之间的关系。

第一种方式是使用instanceof 操作符，只要用这个操作符来测试实例与原型链中出现过的构造函数，结果就会返回true

```
> son instanceof Father
< true
> son instanceof GrandFather
< true
> son instanceof Object
< true
>
```

第二种方式是使用isPrototypeOf()方法。同样，只要是原型链中出现过的原型，都可以说是该原型链所派生的实例的原型，因此isPrototypeOf()方法也会返回true

```
> Object.prototype.isPrototypeOf(son)
< true
> GrandFather.prototype.isPrototypeOf(son)
< true
> Father.prototype.isPrototypeOf(son)
< true
>
```

原型的注意点

在通过原型链实现继承时，不能使用对象字面量创建原型方法。因为这样做就会重写原型链

```
function GrandFather() {
    this.age = 233;
}
GrandFather.prototype.location = "长沙"

function Father() {
    this.name = "爸爸级构造函数"
}

Father.prototype = new GrandFather();

Father.prototype={//坚决不能用这种写法，这种写法会导致上面原型的赋值操作无效化
    sex : "男"
}

let son = new Father();
```