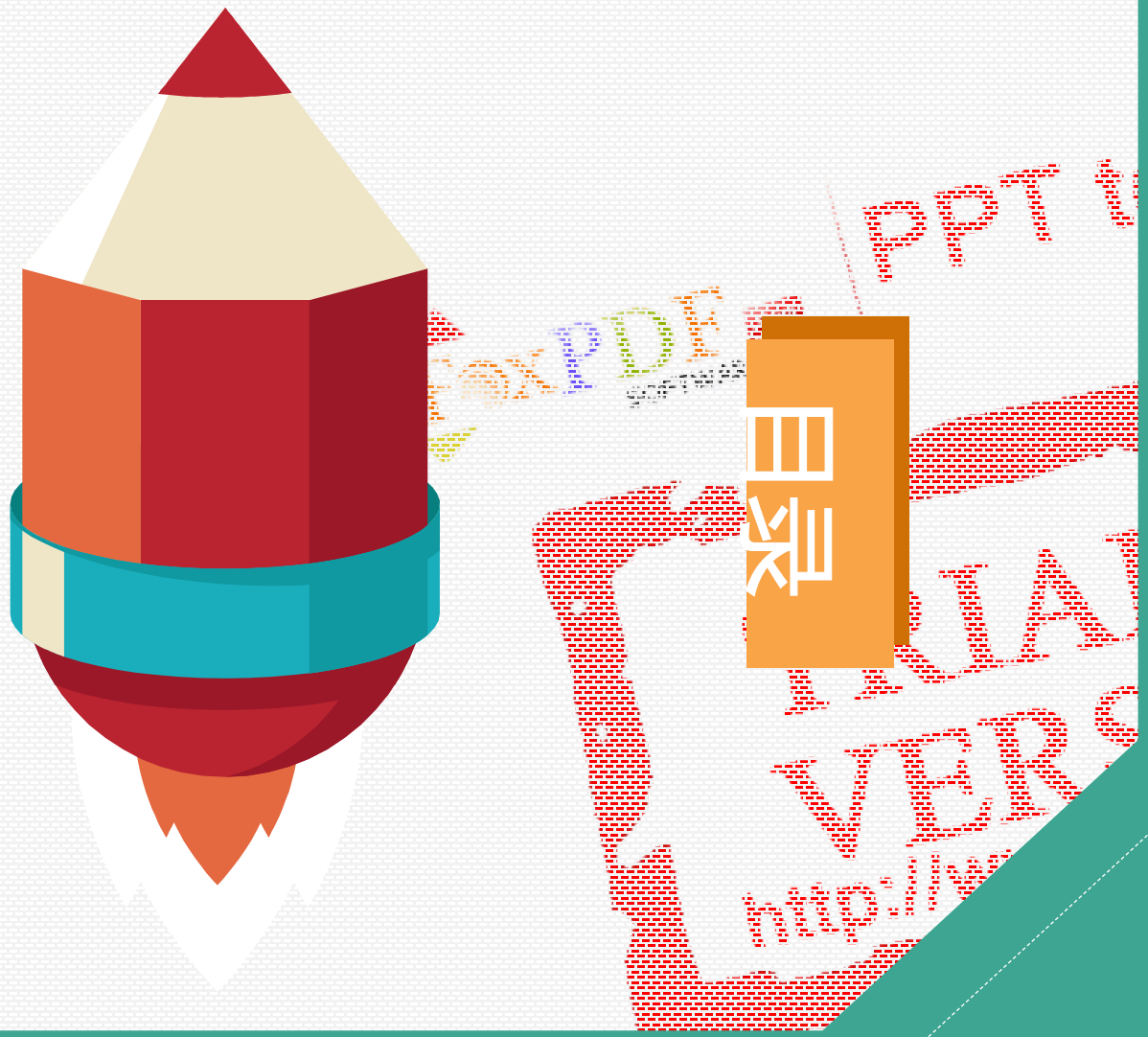
第22课：面向对象编程进阶

主讲老师：万章



- 原型模式的核心概念
- 原型模式的操作及原型链
- 原型的覆盖问题
- 原型的动态继承
- 原生对象的问题

原型模式的核心概念

原型模式

我们创建的每个函数都有一个prototype（原型）属性，这个属性是一个指针，指向一个对象，而这个对象的用途是包含可以由特定类型的**所有实例共享的属性和方法**。 如果按照字面意思来理解，那么prototype 就是通过调用构造函数而创建的那个对象实例的原型对象。

使用原型对象的好处是可以让所有对象实例共享它所包含的属性和方法

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

Person.prototype.sayName=function(){
    console.log(this.name);
}

Person.prototype.company="潭州教育"

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

在构造函数中：

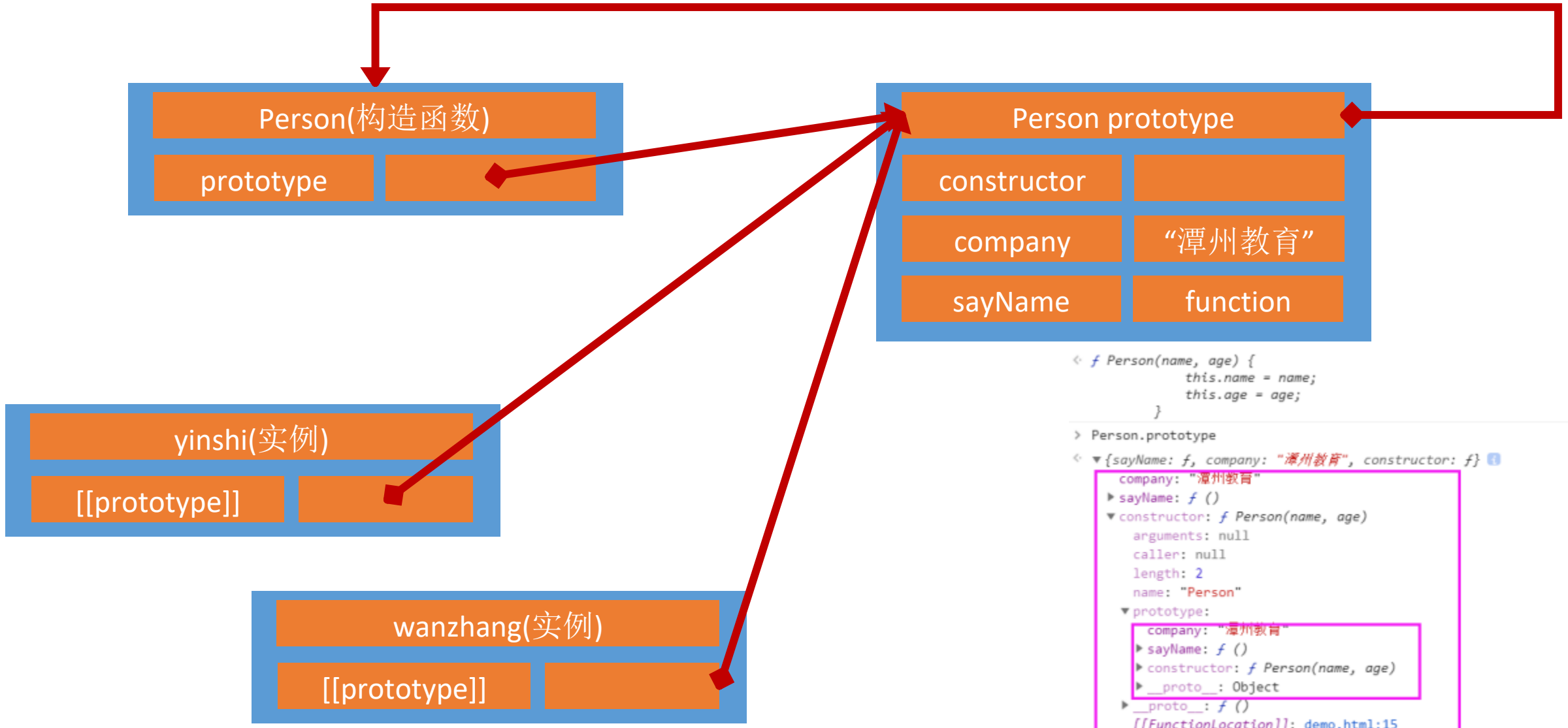
1. 通过this.属性来定义的参数会在实例中直接看到
2. 通过构造函数.prototype定义的参数会在实例的._proto_属性中看到，这代表._proto_里面列举的属性都是从构造函数的prototype中获取的.

理解原型模式

无论什么时候，只要创建了一个新函数，就会根据一组特定的规则为该函数创建一个prototype属性，这个属性指向函数的原型对象。在默认情况下，所有原型对象都会自动获得一个constructor（构造函数）属性，这个属性包含一个指向prototype 属性所在函数的指针。

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

Person.prototype.sayName=function(){
    console.log(this.name);
}

Person.prototype.company="潭州教育"

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

原型模式的概念模型图


Person(构造函数)
prototype
Person prototype
constructor
company
“潭州教育”
sayName
function
yinshi(实例)
[[prototype]]
wanzhang(实例)
[[prototype]]
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

原型模式的操作及原型链

原型的检测

正如在数据特性那一章节所讲[[Prototype]]属性是用户自身无法访问的，但可以通过isPrototypeOf()方法来确定对象之间是否存在这种关系。

```
> Person.prototype.isPrototypeOf(wanzhang)
< true
> Person.prototype.isPrototypeOf(yinshi)
< true
```

```
> let o={};
< undefined
> Person.prototype.isPrototypeOf(o)
< false
>
```

构造函数.prototype.isPrototypeof(待检测对象)

如果被检测对象的[[Prototype]]指向构造函数的prototype属性的，那么这个方法就返回true

原型的获取

ECMAScript 5 增加了一个新方法，叫Object.getPrototypeOf()，在所有支持的实现中，这个方法返回[[Prototype]]的值。

```
> Object.getPrototypeOf(wanzhang)
< ▼{sayName: ƒ, company: "潭州教育", constructor: ƒ}
    company: "潭州教育"
  ▶ sayName: ƒ ()
  ▶ constructor: ƒ Person(name, age)
  ▶ __proto__: Object
> Object.getPrototypeOf(wanzhang)===Person.prototype
< true
>
```

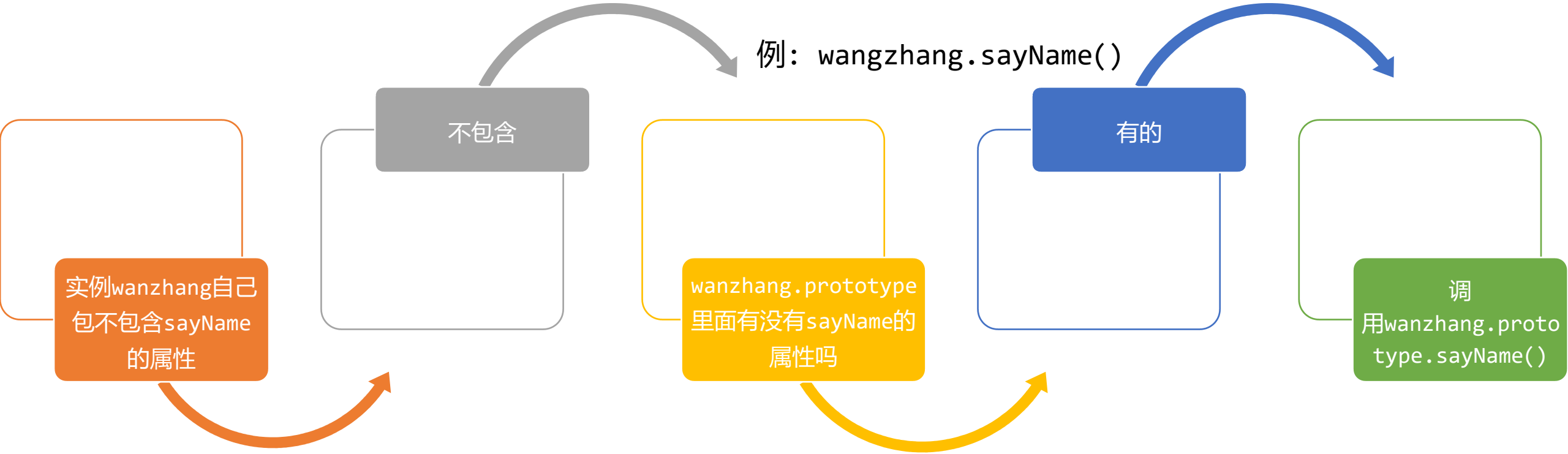
原型链的基本概念

每当代码读取某个对象的某个属性时，都会执行一次搜索，目标是具有给定名字的属性。

1．搜索首先从对象实例本身开始。如果在实例中找到了具有给定名字的属性，则返回该属性的值；

2．如果没有找到，则继续搜索指针指向的原型对象，在原型对象中查找具有给定名字的属性。

3．如果在原型对象中找到了这个属性，则返回该属性的值。



原型的覆盖问题

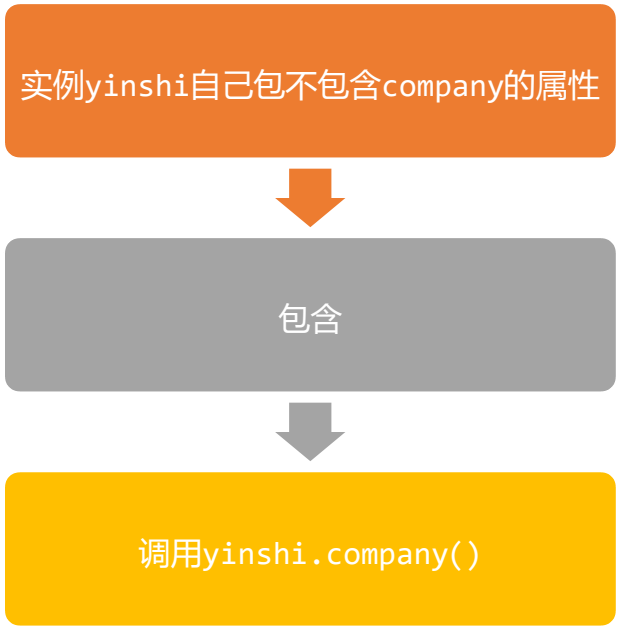
原型的覆盖问题

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

Person.prototype.sayName=function(){
    console.log(this.name);
}

Person.prototype.company="潭州教育"

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

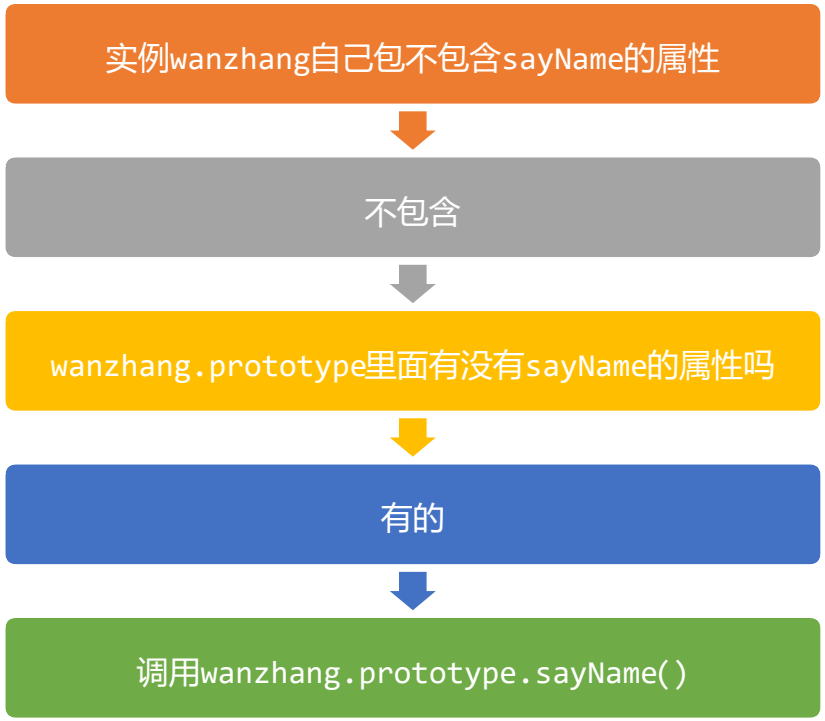
虽然可以通过对象实例访问保存在原型中的值，但却不能通过对象实例重写原型中的值

如果我们在实例中添加了一个属性，而该属性与实例原型中的一个属性同名，那我们就在实例中创建该属性，该属性将会屏蔽原型中的那个属性(原型的属性依旧还在),其他继承原型属性的实例不受影响

原型的覆盖问题




```
> yinshi
◂ ▾Person {name: "银时", age: 18}
    age: 18
    name: "银时"
  ▾__proto__:
    company: "潭州教育"
  ▸sayName: ƒ ()
  ▸constructor: ƒ Person(name, age)
  ▸__proto__: Object
> yinshi.company="怡红院"
◂ "怡红院"
> yinshi
◂ ▾Person {name: "银时", age: 18, company: "怡红院"}
    age: 18
    company: "怡红院"
    name: "银时"
  ▾__proto__:
    company: "潭州教育"
  ▸sayName: ƒ ()
  ▸constructor: ƒ Person(name, age)
  ▸__proto__: Object
> wanzhang.company
◂ "潭州教育"
>
```



在这个例子中，yinshi 的company 被一个新值给屏蔽了。但无论访问yinshi.company还是访问wanzhang.company都能够正常地返回值，即分别是“怡红院”（来自对象实例）和“潭州教育"（来自原型）

当访问yinshi.company 时，需要读取它的值，因此就会在这个实例上搜索一个名为company的属性。这个属性确实存在，于是就返回它的值而不必再搜索原型了。当以同样的方式访问wanzhang.company时，并没有在实例上发现该属性，因此就会继续搜索原型，结果在那里找到了company 属性。

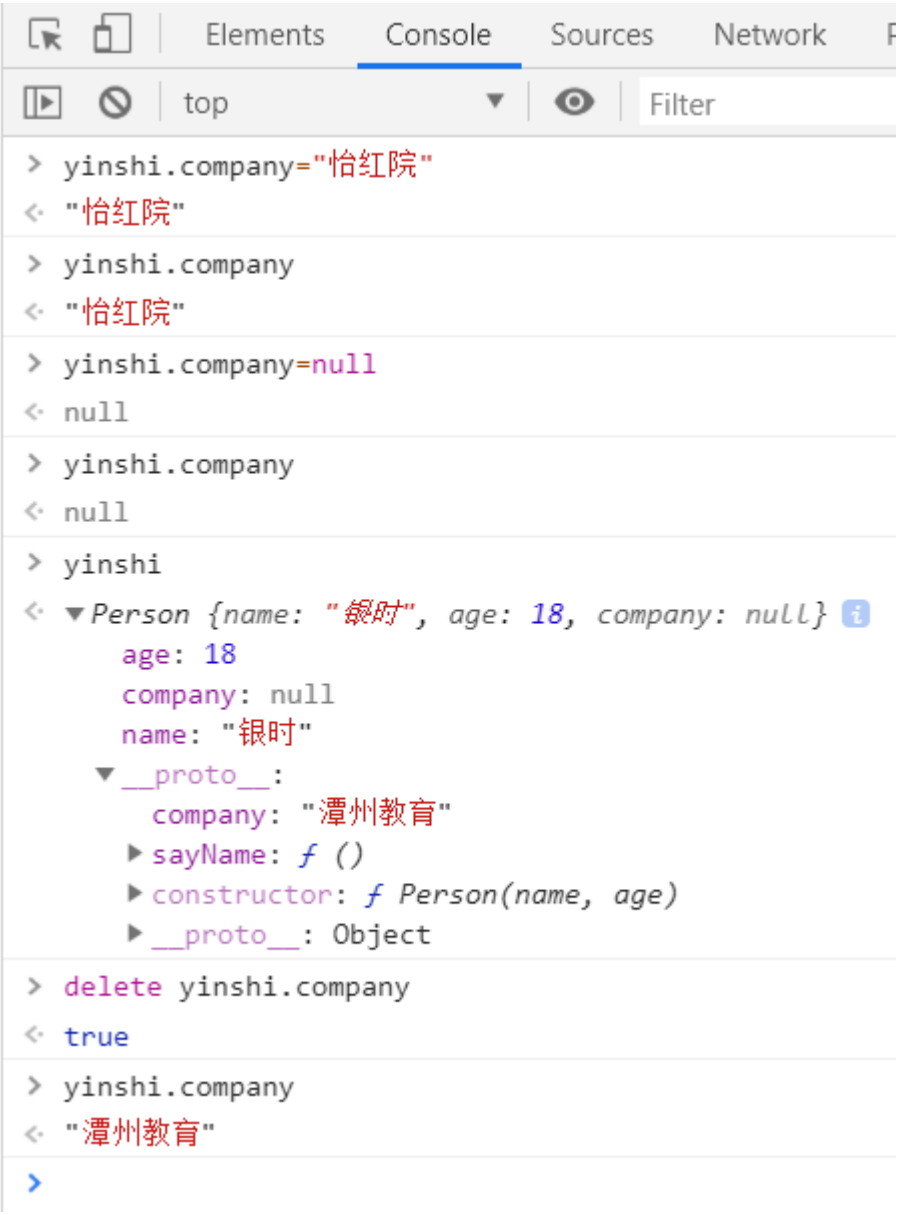
原型的覆盖问题

当为对象实例添加一个属性时，这个属性就会屏蔽原型对象中保存的同名属性；

简单来说，添加这个属性只会阻止我们访问原型中的那个属性，但不会修改那个属性。即使将这个属性设置为null，也只会在实例中设置这个属性，而不会恢复其指向原型的连接。

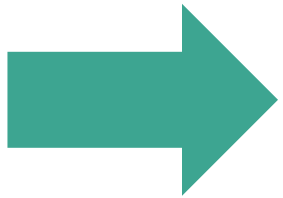
不过，使用delete 操作符则可以完全删除实例属性,这样就可以恢复指向原型的链接

我们可以用obj.hasOwnProperty(“属性名称”)方法判断该属性是否是obj的自有属性还是继承的属性(先用in运算符来判断对象是否拥有该属性)



原型的覆盖问题

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

Person.prototype.sayName = function () {
    console.log(this.name);
}

Person.prototype.company = "潭州教育"

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



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

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

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

前面例子中每添加一个属性和方法就要敲一遍Person.prototype。为减少不必要的输入，也为了从视觉上更好地封装原型的功能，更常见的做法是用一个包含所有属性和方法的对象字面量来重写整个原型对象

原型的覆盖问题

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

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

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

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

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

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

```
> yinshi
  ▼Person {name: "银时", age: 18}
      age: 18
      name: "银时"
    ▼__proto__:
      ▶constructor: ƒ Person(name, age)
      ▶__proto__: Object
>
```

top | Filter

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

构造函数木有了！！！！！！

每创建一个函数，就会同时创建它的prototype 对象，这个对象也会自动获得constructor 属性。

我们在这里使用的语法，本质上完全重写了默认的prototype 对象，constructor 属性也就变成了新对象的constructor 属性（指向Object 构造函数），不再指向Person 函数。

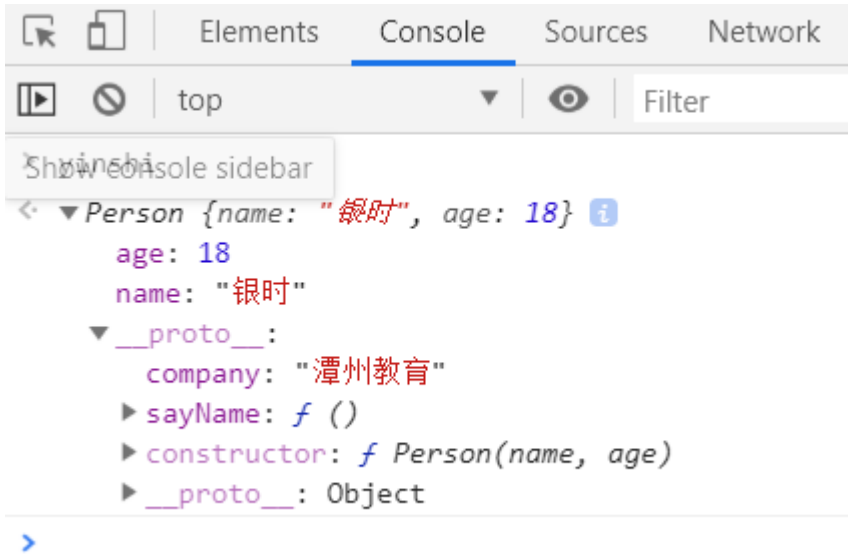
尽管instanceof操作符还能返回正确的结果，但通过constructor 已经无法确定对象的类型了

原型的覆盖问题

用instanceof 操作符测试Object 和Person 仍然返回true，但constructor 属性则等于Object 而不等于Person 了。如果constructor 的值真的很重要，可以像下面这样特意将它设置回适当的值。

（但是此时的constructor是可枚举的，为了保证不会被枚举，可以同时定义数据特性）

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

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

Object.defineProperty(Person.prototype,"constructor",{
    enumerable:false,
    value:Person
})

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

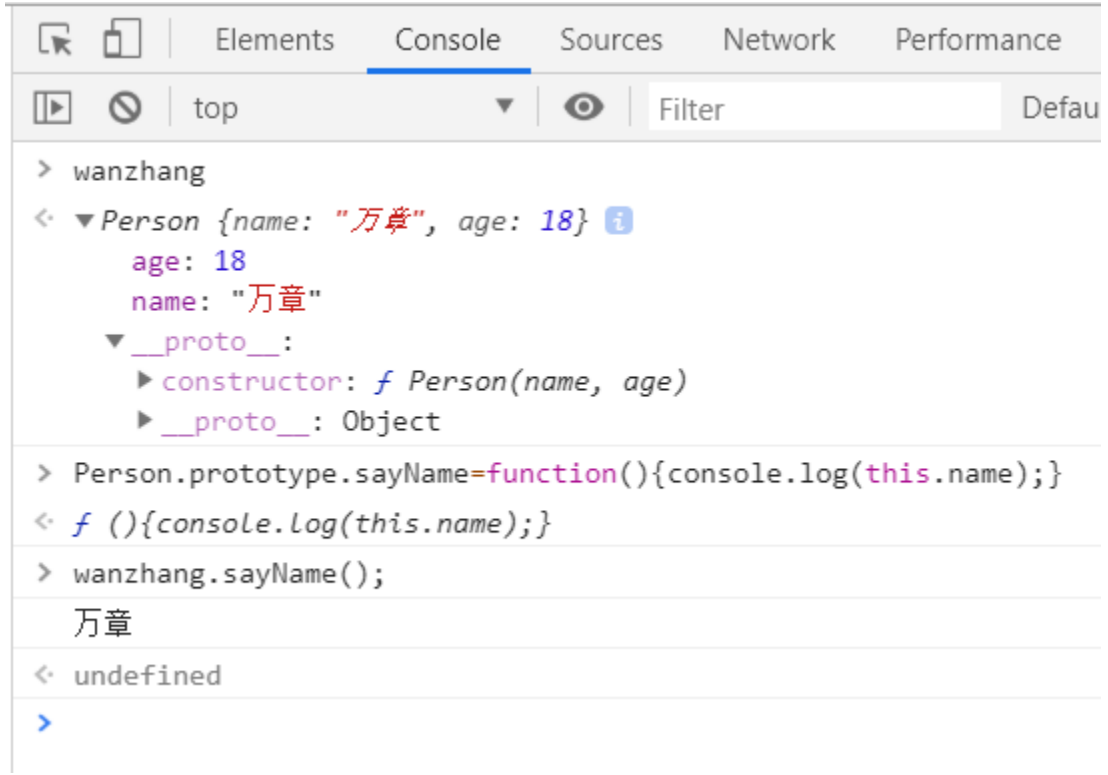


原型的动态继承

原型的动态性问题

由于在原型中查找值的过程是一次搜索，因此我们对原型对象所做的任何修改都能够立即从实例上反映出来—即使是先创建了实例后修改原型也照样如此

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

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



实例与原型之间的连接只不过是一个指针

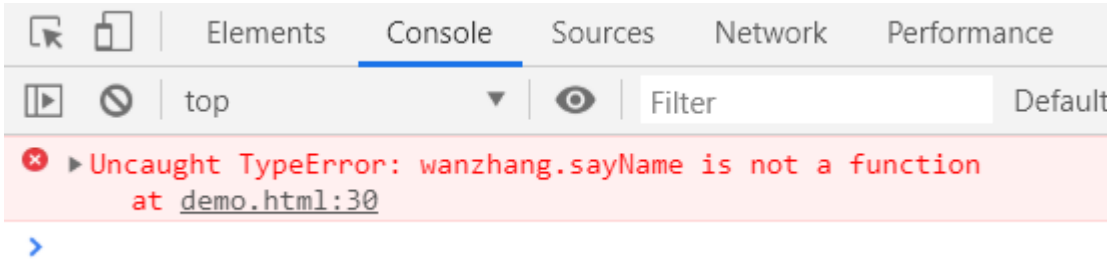
（见原型的概念模型图，所以每次调用原型的方法都是一次在内存数据的中搜索，一旦内存数据中的

原型重写时的动态性问题

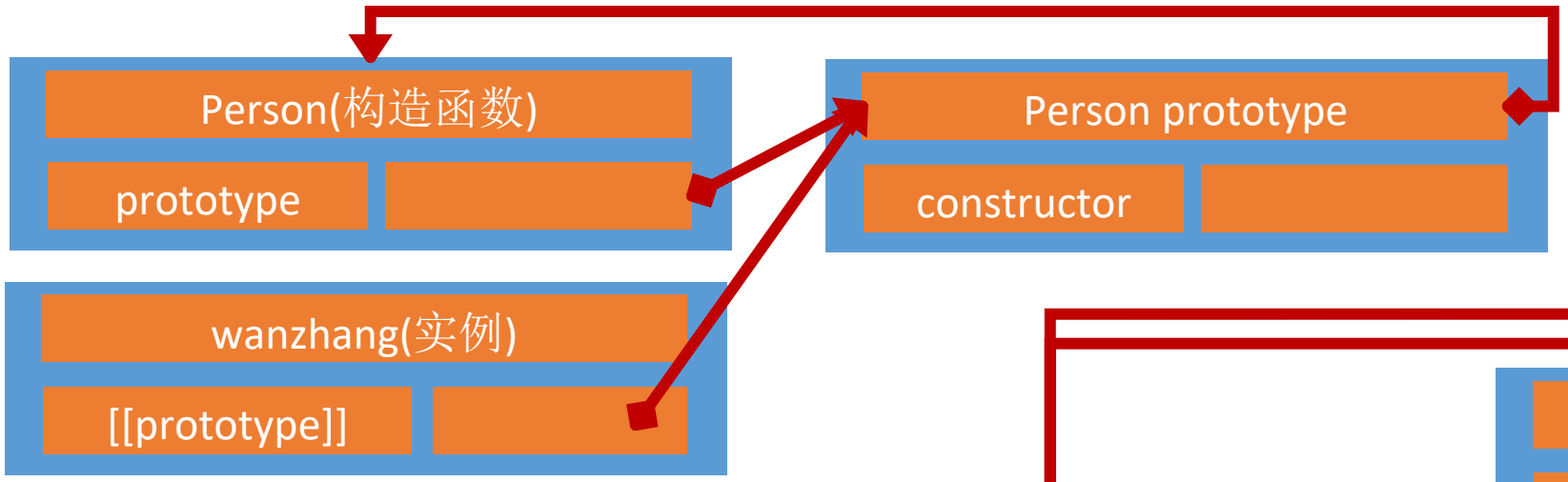
尽管可以随时为原型添加属性和方法，并且修改能够立即在所有对象实例中反映出来，但如果是重写整个原型对象，那么情况就不一样了。我们知道，调用构造函数时会为实例添加一个指向最初原型的[[Prototype]]指针，而把原型修改为另外一个对象就等于切断了构造函数与最初原型之间的联系。

请记住：实例中的指针仅指向原型，而不指向构造函数

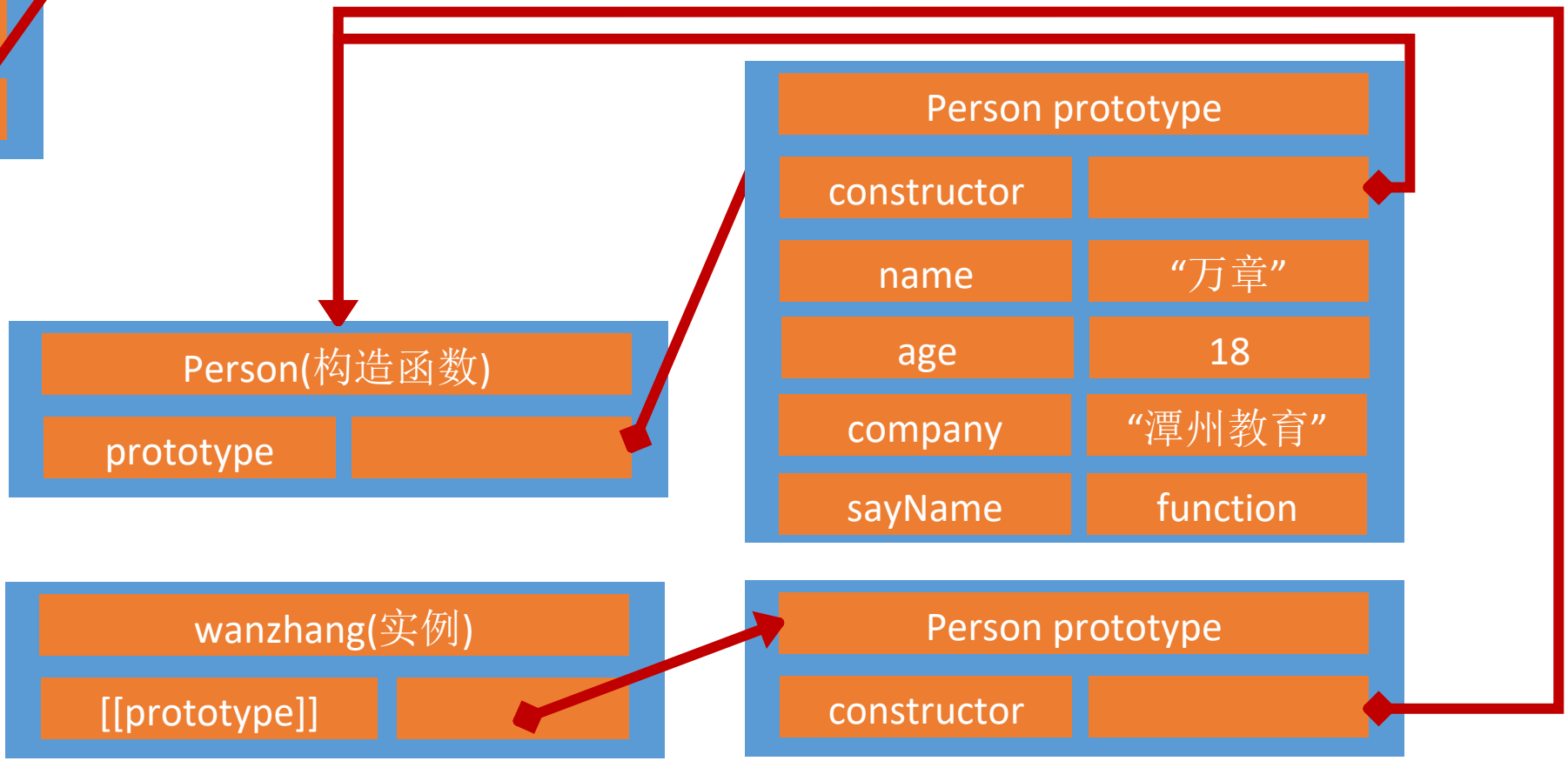
```
function Person(name, age) {

}

let wanzhang = new Person("万章", 18);

Person.prototype = {
    constructor: Person,
    name:"万章",
    age:"18",
    company: "潭州教育",
    sayName: function () {
        console.log(this.name);
    }
}

wanzhang.sayName();
```



原型重写时的动态性问题



重写原型对象之前



重写原型对象之后

重写原型对象切断了现有原型与任何之前已经存在的对象实例之间的联系；之前创造的对象引用的仍然是最初的原型。

原生对象学习

原生对象的原型

所有原生引用类型（Object、Array、Date，等等）都在其构造函数的原型上定义了方法。

```
> Array.prototype
▼[constructor: ƒ, concat: ƒ, copyWithin: ƒ, fill: ƒ, find: ƒ, …]
  ▶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: true, entries: true, fil
  ▶__proto__: Object
```

```
> Object.prototype
◀{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__
  pGetter__: ƒ, …}
  ▶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__()
```

```
Date.prototype
▼{constructor: ƒ, toString: ƒ, toDateString: ƒ, toTimeString: ƒ, toISOString
  ▶constructor: ƒ Date()
  ▶getDate: ƒ getDate()
  ▶getDay: ƒ getDay()
  ▶getFullYear: ƒ getFullYear()
  ▶getHours: ƒ getHours()
  ▶getMilliseconds: ƒ getMilliseconds()
  ▶getMinutes: ƒ getMinutes()
  ▶getMonth: ƒ getMonth()
  ▶getSeconds: ƒ getSeconds()
  ▶getTime: ƒ getTime()
  ▶getTimezoneOffset: ƒ getTimezoneOffset()
  ▶getUTCDate: ƒ getUTCDate()
  ▶getUTCDay: ƒ getUTCDay()
  ▶getUTCFullYear: ƒ getUTCFullYear()
  ▶getUTCHours: ƒ getUTCHours()
  ▶getUTCMilliseconds: ƒ getUTCMilliseconds()
  ▶getUTCMinutes: ƒ getUTCMinutes()
  ▶getUTCMonth: ƒ getUTCMonth()
  ▶getUTCSeconds: ƒ getUTCSeconds()
  ▶getYear: ƒ getYear()
  ▶setDate: ƒ setDate()
  ▶setFullYear: ƒ setFullYear()
  ▶setHours: ƒ setHours()
  ▶setMilliseconds: ƒ setMilliseconds()
  ▶setMinutes: ƒ setMinutes()
  ▶setMonth: ƒ setMonth()
  ▶setSeconds: ƒ setSeconds()
  ▶setTime: ƒ setTime()
  ▶setUTCDate: ƒ setUTCDate()
  ▶setUTCFullYear: ƒ setUTCFullYear()
  ▶setUTCHours: ƒ setUTCHours()
  ▶setUTCMilliseconds: ƒ setUTCMilliseconds()
  ▶setUTCMinutes: ƒ setUTCMinutes()
  ▶setUTCMonth: ƒ setUTCMonth()
  ▶setUTCSeconds: ƒ setUTCSeconds()
  ▶setYear: ƒ setYear()
  ▶toDateString: ƒ toDateString()
  ▶toGMTString: ƒ toUTCString()
  ▶toISOString: ƒ toISOString()
  ▶toJSON: ƒ toJSON()
```

原生对象原型的拓展

通过原生对象的原型，不仅可以取得所有默认方法的引用，而且也可以定义新方法。可以像修改自定义对象的原型一样修改原生对象的原型，因此可以随时添加方法

```
Object.prototype.sayMyName=function(){
    console.log("万章大帅比");
}

let a=[];
let o={};
function fn(){}
```

尽管可以这样做，但我们不推荐在程序中修改原生对象的原型。如果因某个实现中缺少某个方法，就在原生对象的原型中添加这个方法，那么当在另一个支持该方法(比如不同版本的浏览器)的实现中运行代码时，就可能会导致命名冲突。

原型对象模式的问题

- 它省略了为构造函数传递初始化参数这一环节，结果所有实例在默认情况下都将取得相同的属性值(如果所有的属性都在prototype上定义的话)

- 原型中所有属性是被很多实例共享的，这种共享对于函数非常合适。然而，对于包含引用类型值的属性来说，就有些问题

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

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

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

还是因为引用类型的锅

实例查询到原型的属性是直接指向内存中的原始数据，增删改查都是在原始数据上