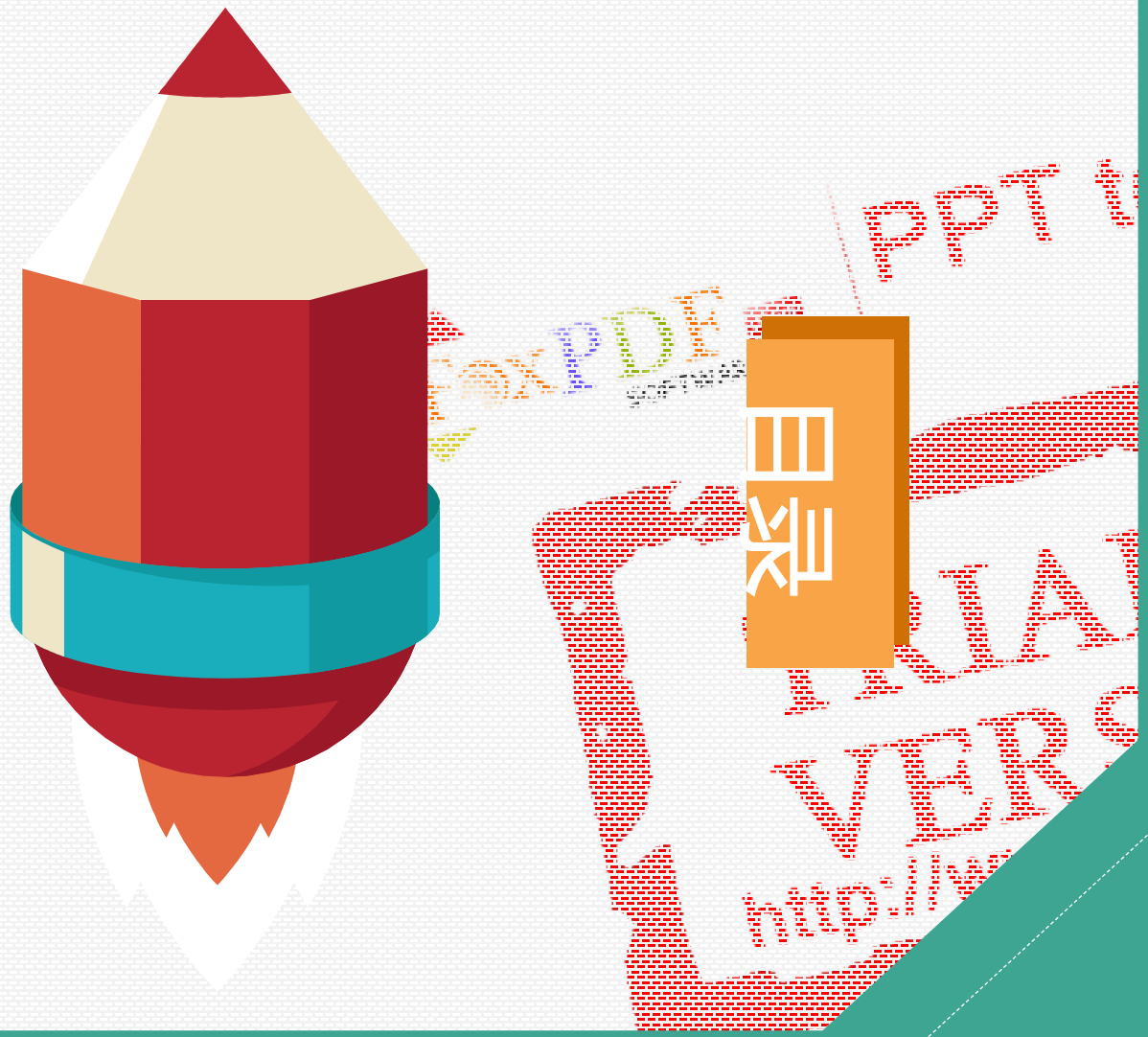
第24课：面向对象编程高级

主讲老师：万章



目录

- 伪造对象(call,apply)
- 组合继承
- 原型式继承
- 寄生式继承模式
- 寄生组合式继承

经典继承(伪造对象)

经典继承(伪造对象)

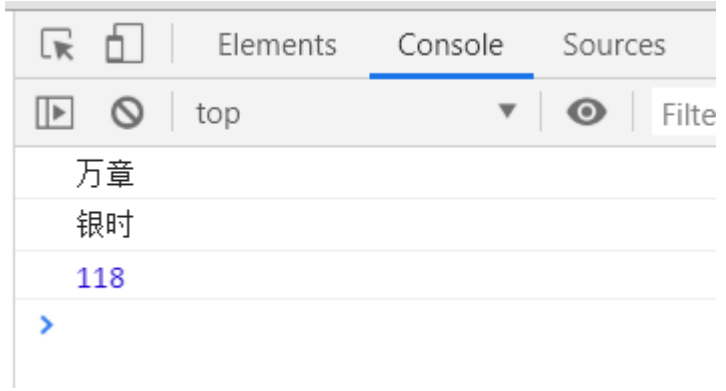
函数只不过是在特定环境中执行代码的对象!!!

经典继承(伪造对象)之call

当我们在一个全局环境中已经创造了若干的对象及其对应方法时，有些时候需要实现一个方法借用的功能：

例:对象o有一个方法a是输出对象的name值，而对象o2没有该方法，那么如果我们也想输出o2的值的话，那么我们就可以把对象o的值借过来用用

```
let o1={
    name:"万章",
    age:18,
    sayName:function(){
        console.log(this.name);
    },
    add:function(num1,num2){
        console.log(num1+num2+this.age);
    }
}
let o2={
    name:"银时",
    age:88,
    sayAge:function(){
        console.log(this.age);
    }
}
o1.sayName();//输出万章
o1.sayName.call(o2);//输出银时
o1.add.call(o2,14,16);//输+出14+16+88=118
```



经典继承(伪造对象)之call

fn.call({ },参数1,参数2…)

等价于

{ }.fn(参数1,参数2,…)

经典继承(伪造对象)之call

```
let o1={
    name:"万章",
    age:18,
    sayName:function(){
        console.log(this.name);
    },
    add:function(num1,num2){
        console.log(num1+num2+this.age);
    }
}
let o2={
    name:"银时",
    age:88,
    sayAge:function(){
        console.log(this.age);
    }
}

o1.sayName.call(o2);//输出银时
```

等价于

```
function(){
        console.log(this.name);
}.call(o2);
```

等价于

```
o2.function(){
        console.log(this.name);
}();
```

函数内的this永远指向调用该函数的对象，所以此时该函数是由对象o2调用的，所以函数内的this指的就是o2

经典继承(伪造对象)之call

```
let o1={
    name:"万章",
    age:18,
    sayName:function(){
        console.log(this.name);
    },
    add:function(num1,num2){
        console.log(num1+num2+this.age);
    }
}
let o2={
    name:"银时",
    age:88,
    sayAge:function(){
        console.log(this.age);
    }
}

o1.add.call(o2,14,16);//输出118
```

等价于

```
function(num1,num2){
        console.log(num1+num2+this.age);
}.call(o2,14,16)
```

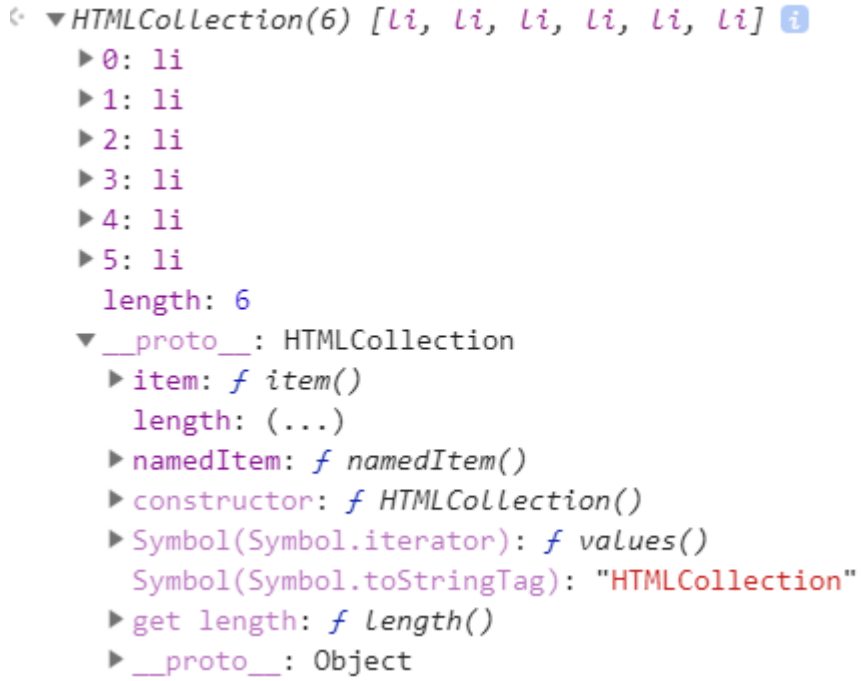
等价于

```
o2.function(num1,num2){
        console.log(num1+num2+this.age);
}(14,16)
```

此处的参数可以是任意多个

经典继承(伪造对象)之call经典应用

在数组一章我们学到了很多的数组方法，但是我们日常应用中最常见的却是由DOM节点组成的类数组,而类数组中的方法极少,使用时极不方便

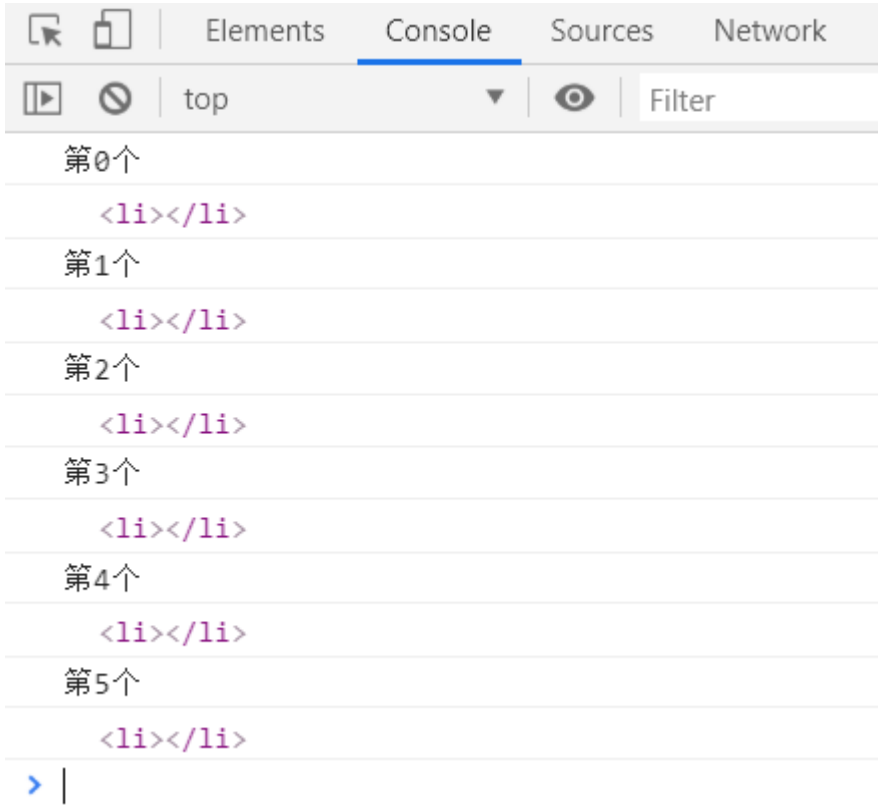


类数组

数组

经典继承(伪造对象)之call经典应用

```
<ul>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
</ul>
<script>

    let aLi=document.getElementsByTagName("li");

    Array.prototype.forEach.call(aLi,function(item,index,array){
        console.log("第"+index+"个");
        console.log(item);
    });
```



在类数组上调用数组的方法

经典继承(伪造对象)之apply

```
fn.apply({ },[参数1,参数2…])
```

等价于

```
{ }.fn(参数1,参数2,…)
```

apply方法和call方法最大的区别就是**apply方法的参数都放在一个数组里面**,这个数组可以是一个类数组或是标准数组

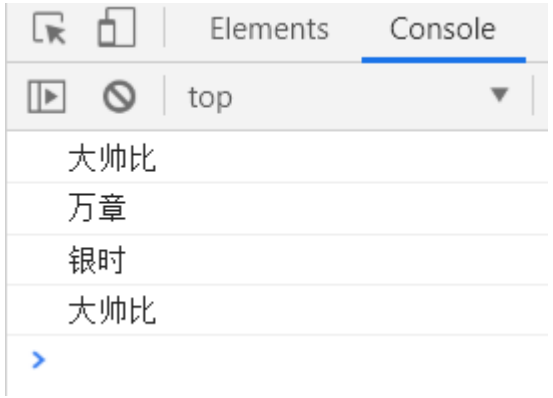
经典继承(伪造对象)之bind

```
var name="大帅比";

let o1 = {
    name: "万章",
}
let o2 = {
    name: "银时",
}

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

sayName();//输出大帅比

sayName.call(o1);//输出万章

let newFn=sayName.bind(o2);

newFn();//输出银时

sayName();//输出
```

fn.bind({ } ,参数1,参数2...)

等价于

{ } .fn(参数1,参数2...)



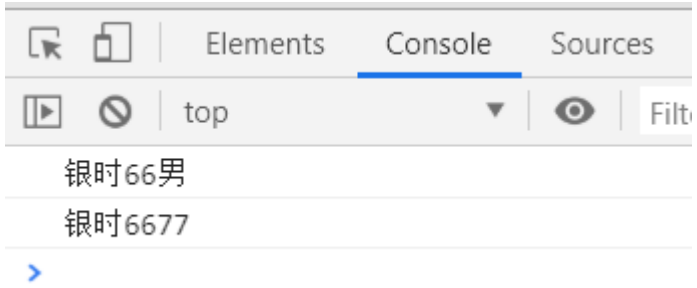
bind()方法**创建一个新的函数**，在调用时设置this关键字为提供的值。

并在调用新函数时，将**绑定时给定参数列表**作为原函数的参数序列的前若干项。

经典继承(伪造对象)之bind

```
let o2 = {
    name: "银时",
}

function sayName(age,sex){
    console.log(this.name+age+sex);
}

let newFn=sayName.bind(o2,66);

newFn("男");//输出银时66男
newFn(77);//输出银时6677
```

注意：调用新函数时，原来在**绑定时给定的参数列表**将会**永远**作为参数组的前几项输入进来



函数绑定时传进来的参数66，将永远作为函数里面的age变量的值，永不会变

经典继承(伪造对象)之bind

```
let newFn=sayName.bind(o2,66);
```

```
o2={
    name: "银时",
    sayName:function(){
        console.log(this.name+66+sex);
    }
}

newFn=o2.sayName

o2 = {
    name: "银时",
}
```

```
newFn("男");

o2={
    name: "银时",
    sayName:function(){
        console.log(this.name+66+sex);
    }
}

newFn=o2.sayName

newFn("男");//输出银时66男

o2 = {
    name: "银时",
}
```

以上只是为了大致模拟bind的运行过程思路

经典继承(伪造对象)

这种技术的基本思想相当简单，即在子类型构造函数的内部调用超类型构造函数 ，因此通过使用apply()和call()方法也可以在将来新创建的对象上执行构造函数

```
function SuperType() {
    this.colors = ["red", "blue", "green"];
}

function SubType() {
    console.log(this);
    //输出SubType,this是函数的运行环境， 在函数当做构造函数使用的时候的环境是SubType自己
    SuperType.call(this);
}

var instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors); //"red,blue,green,black"

var instance2 = new SubType();
console.log(instance2.colors); //"red,blue,green"
```

我们实际上是在（未来将要）新创建SubType 实例的环境下调用了SuperType 构造函数。这样一来，就会在新SubType 对象上执SuperType()函数中定义的所有对象初始化代码。

结果，SubType 的每个实例就都会具有自己的colors 属性的副本了

经典继承(伪造对象)的优势和劣势

相对于原型链而言，借用构造函数有一个很大的优势，即可以在子类型构造函数中向父类型构造函数传递参数

```
function SuperType(name) {
    this.name = name;
}

function SubType() {
    //继承了SuperType，同时还传递了参数
    SuperType.call(this, "Nicholas");
    //实例属性
    this.age = 29;
}
var instance = new SubType();
alert(instance.name); //"Nicholas";
alert(instance.age); //29
```

在SubType 构造函数内部调用SuperType 构造函数时，实际上是为SubType 的实例设置了name 属性。

为了确保SuperType 构造函数不会重写子类型的属性，可以在调用超类型构造函数后，再添加应该在子类型中定义的属性。

如果仅仅是借用构造函数，那么也将无法避免构造函数模式存在的问题—方法都在构造函数中定义，因此函数复用就无从谈起了。

组合继承

组合继承

组合继承（combination inheritance），有时候也叫做伪经典继承，指的是将原型链和借用构造函数的技术组合到一块，从而发挥二者之长的一种继承模式。思路是使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又能够保证每个实例都有它自己的属性

```
> instance1
< ▼SubType {name: "万章", colors: Array(3), age: 29}
    age: 29
  ▶ colors: (3) ["red", "blue", "green"]
    name: "万章"
  ▼ __proto__: SuperType
    ▶ colors: (3) ["red", "blue", "green"]
    ▶ constructor: ƒ SubType(name, age)
      name: undefined
    ▶ sayAge: ƒ ()
    ▼ __proto__:
      ▶ sayName: ƒ ()
      ▶ constructor: ƒ SuperType(name)
      ▶ __proto__: Object
>
```

```
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}
SuperType.prototype.sayName = function () {
    console.log(this.name);
};

//当用SuperType作为构造函数时，会创造三个属性,name,colors,sayName

function SubType(name, age) {
    //继承属性
    SuperType.call(this, name);
    this.age = age;//当用SuperType作为构造函数时，会创造age属性
}

//继承方法
SubType.prototype = new SuperType();
//用SuperType的实例当做原型，原有的构造函数属性被覆盖
//此时没有传参,所以原型上的name是undefined,而且有一个color和sayAge
SubType.prototype.constructor = SubType;
 //手动定义构造函数
SubType.prototype.sayAge = function () {
    console.log(this.age);
};
//在SuperType的实例当做原型的基础上添加sayAge和constructor

var instance1 = new SubType("万章", 29);
//instance1实例拥有的是name,colors,age,sayName,sayAge,constructor
```

原型式继承

原型式继承

道格拉斯·克罗克福德在2006 年写了一篇文章，题为Prototypal Inheritance in JavaScript（JavaScript中的原型式继承）。在这篇文章中，他介绍了一种实现继承的方法，这种方法并没有使用严格意义上的构造函数。他的想法是借助原型可以基于已有的对象创建新对象，同时还不必因此创建自定义类型。为了达到这个目的，他给出了如下函数。

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

在object()函数内部，先创建了一个临时性的构造函数，然后将传入的对象作为这个构造函数的原型，最后返回了这个临时类型的一个新实例

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}

var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};
var anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");//会直接操作原型上的对象
var yetAnotherPerson = object(person);//此时的person已经变化了
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");//会直接操作原型上的对象
console.log(person.friends); //"Shelby,Court,Van,Rob,Barbie"
console.log(anotherPerson.friends); //"Shelby,Court,Van,Rob,Barbie"
console.log(yetAnotherPerson.friends); //"Shelby,Court,Van,Rob,Barbie"
```

从本质上讲，object()对传入其中的对象执行了一次浅复制

寄生式继承

寄生式继承

寄生式（parasitic）继承是与原型式继承紧密相关的一种思路，并且同样也是由克罗克福德推而广之的。寄生式继承的思路与寄生构造函数和工厂模式类似：
即创建一个仅用于封装继承过程的函数，该函数在内部以某种方式来增强对象，最后再像真地是它做了所有工作一样返回对象

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}

function createAnother(original) {
    var clone = object(original); //通过调用函数创建一个新对象
    clone.sayHi = function () { //以某种方式来增强这个对象
        console.log("hi");
    };
    return clone; //返回这个对象
}
```

在主要考虑对象而不是自定义类型和构造函数的情况下，寄生式继承也是一种有用的模式。示范继承模式时使用的object()函数不是必需的，任何能够返回新对象的函数都适用于此模式。

使用寄生式继承来为对象添加函数，会由于不能做到函数复用而降低效率；这一点与构造函数模式类似。

寄生组合式继承

寄生组合式继承

前面说过，组合继承是JavaScript 最常用的继承模式；不过，它也有自己的不足。组合继承最大的问题就是无论什么情况下，都会调用两次超类型构造函数：一次是在创建子类型原型的时候，另一次是在子类型构造函数内部。没错，子类型最终会包含超类型对象的全部实例属性，但我们不得不在调用子类型构造函数时重写这些属性

在第一次调用SuperType 构造函数时，SubType.prototype 会得到两个属性：name 和colors；它们都是SuperType 的实例属性，只不过现在位于SubType 的原型中。

当调用SubType 构造函数时，又会调用一次SuperType 构造函数，这一次又在新对象上创建了实例属性name 和colors。于是，这两个属性就屏蔽了原型中的两个同名属性

第一次调用

第二次调用


```
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}
SuperType.prototype.sayName = function () {
    console.log(this.name);
};

//当用SuperType作为构造函数时，会创造三个属性,name,colors,sayName

function SubType(name, age) {
    //继承属性
    SuperType.call(this, name);
    this.age = age;//当用SuperType作为构造函数时，会创造age属性
}

//继承方法
SubType.prototype = new SuperType();
//用SuperType的实例当做原型，原有的构造函数属性被覆盖
//此时没有传参,所以原型上的name是undefined,而且有一个color和sayAge
SubType.prototype.constructor = SubType;
//手动定义构造函数
SubType.prototype.sayAge = function () {
    console.log(this.age);
};
//在SuperType的实例当做原型的基础上添加sayAge和constructor

var instance1 = new SubType("万章", 29);
//instance1实例拥有的是name,colors,age,sayName,sayAge,constructor
```

寄生组合式继承

所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法。其背后的基本思路是：不必为了指定子类型的原型而调用超类型的构造函数，我们所需要的无非就是超类型原型的一个副本而已。本质上，就是使用寄生式继承来继承超类型的原型，然后再将结果指定给子类型的原型。

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}

function inheritPrototype(subType, superType) {
    var prototype = object(superType.prototype); //创建对象
    prototype.constructor = subType; //增强对象
    subType.prototype = prototype; //指定对象
}
```

这个函数接收两个参数：子类型构造函数和超类型构造函数。在函数内部

1. 第一步是创建超类型原型的一个副本。
2. 第二步是为创建的副本添constructor 属性，从而弥补因重写原型而失去的默认的constructor 属性。
3. 最后一步，将新创建的对象（即副本）赋值给子类型的原型。这样，我们就可以用调用inherit-Prototype()函数的语句，去替换前面例子中为子类型原型赋值的语句了

寄生组合式继承

```
function SuperType(name) {
    this.name = name;
    this.colors = ["red", "blue", "green"];
}
SuperType.prototype.sayName = function () {
    alert(this.name);
};

function SubType(name, age) {
    SuperType.call(this, name);
    this.age = age;
}
inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function () {
    alert(this.age);
};

let instance=new SubType("万章",18);
//只会调用SubType和SuberType一次
```

```
> instance
< ▼SubType {name: "万章", colors: Array(3), age: 18}
    age: 18
  ▶ colors: (3) ["red", "blue", "green"]
    name: "万章"
  ▼ __proto__: SuperType
    ▶ constructor: ƒ SubType(name, age)
    ▶ sayAge: ƒ ()
    ▼ __proto__:
      ▶ sayName: ƒ ()
      ▶ constructor: ƒ SuperType(name)
      ▶ __proto__: Object
>
```

这个例子的高效率体现在它只调用了一次SuperType 构造函数，并且因此避免了在SubType.prototype 上面创建不必要的、多余的属性。与此同时，原型链还能保持不变；因此，还能够正常使用instanceof 和isPrototypeOf()。

开发人员普遍认为寄生组合式继承是引用类型最理想的继承范式