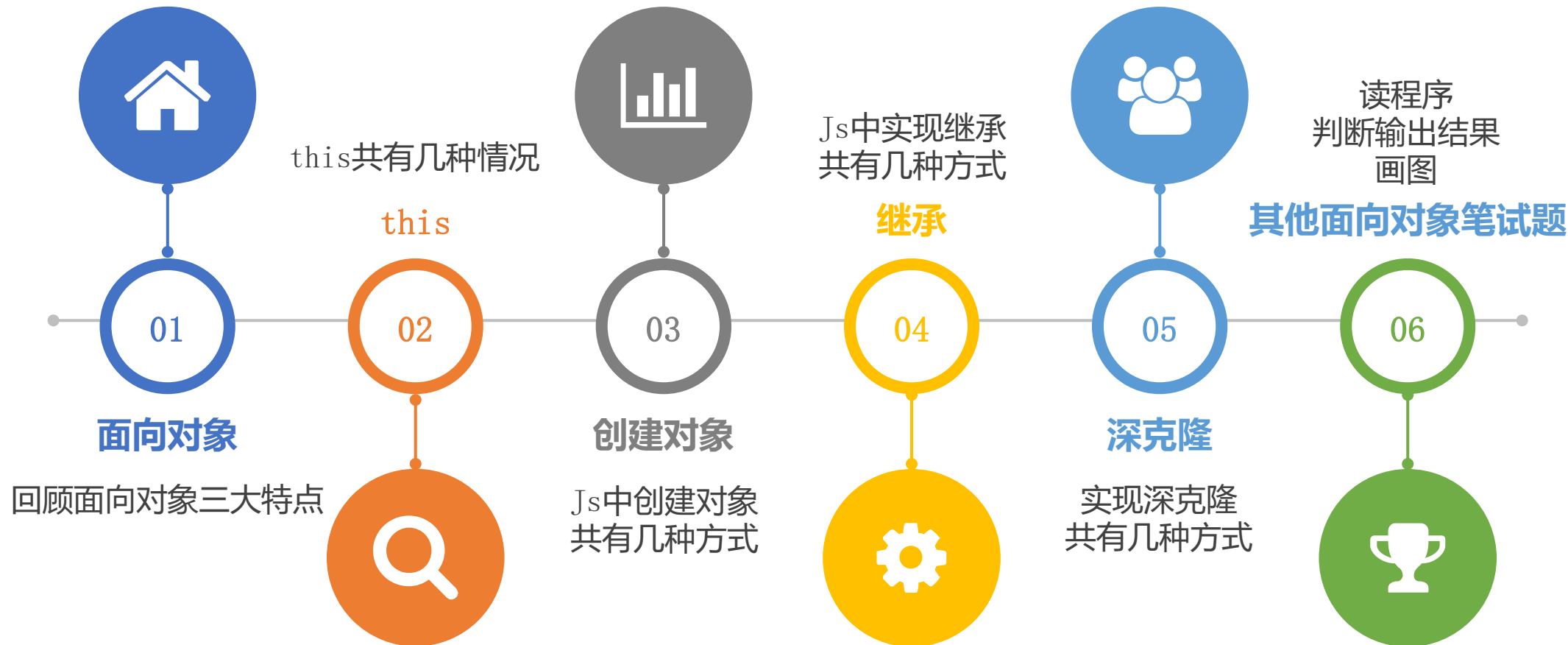


# JS高频笔试题精讲——面相对象

Web学科教学总监

张东（东神）

# 今天你能收获什么？





# Q1：面向对象

## ——回顾面向对象三大特点

# 问题:

程序中将来会保存大量的数据。

而大量数据如果零散的随意管理，极容易出错！

而且用着不方便。



电池

天线

遥控器

汽车



电池

天线

电池

遥控器

坦克

遥控器

汽车

天线

遥控器

电池

飞机

天线





**解決?**



电池

天线

电池

遥控器

坦克

遥控器

汽车

天线

遥控器

电池

飞机

天线





遥控器

天线

电池

汽车



# 遥控汽车

遥控器

天线

电池

汽车



遥控汽车

遥控器

天线

电池

汽车

遥控坦克

遥控器

坦克

电池

天线

遥控飞机

天线

电池

遥控器

飞机



# 今后程序中，也是

- 先使用一个个的对象结构集中存储现实中事物的属性和功能。
- 然后，再按需使用不同对象中的不同属性和功能。
- 这就是面向对象编程





- 极其便于大量数据的管理维护。



# 如何使用面向对象





如何使用面向对象：

三步/三大特点

封装、继承、多态



# 封装

- (1). 封装就是**创建**一个**对象**，集中**保存**现实中一个**事物**的**属性**和**功能**。
- (2). 将零散的数据封装进对象结构中，极其**便于大量数据的管理维护**。
- (3). 今后，只要使用**面向对象**思想开发时，**第一步**都是**先封装**各种各样的**对象**结构备用。



**如何封装对象:**

**3种方式:**

封装

继承

多态





# 第一种方式： 直接量 用{}:

```
var 对象名={ //new Object()的简写  
    属性名: 属性值,  
    ... : ... ,  
    方法名: function(){ ... }  
}
```

封装

继承

多态

## 如何访问对象中的成员:

对象名.属性名

对象名.方法名()



# 比如:

封装

继承

多态

```
var lilei={
  sname:"Li Lei",//姓名
  sage:11,//年龄
  intr:function(){
    console.log(`I'm ${sname}, I'm ${sage}`)
  }
}
```

Uncaught ReferenceError: sname is not defined

```
//输出lilei的年龄
console.log(lilei.sage);
//调用lilei的intr()方法, 请lilei做自我介绍
lilei.intr();
//过了一年, lilei的年龄涨了一岁
lilei.sage++;
//再次输出lilei的年龄
console.log(lilei.sage);
//再次调用lilei的intr()方法, 请lilei做自我介绍
lilei.intr();
```



封装

继承

多态

## 问题:

在对象自己的方法内,

想使用当前对象自己的另一个属性名时,

竟然报错!

说xxx属性名 is not defined——未定义!



# 原因

封装

继承

多态





开局

window

```
var lilei={
  sname: "Li Lei",
  sage:11,
  intr:function(){
    console.log(`I'm ${sname}`);
  }
}
lilei.intr();
```



定义对象时  
window

lilei : 0x1234

地址: 0x1234

{ //new Object()

}

var lilei={

sname: "Li Lei",

sage:11,

intr:function(){

console.log(`I'm \${sname}`);

}

}

lilei.intr();



定义对象时  
window

lilei : 0x1234

地址: 0x1234

```
{ //new Object()  
  sname: "Li Lei",  
  sage:11,  
}
```

```
var lilei={  
  sname: "Li Lei",  
  sage:11,  
  intr:function(){  
    console.log(`I'm ${sname}`);  
  }  
}  
lilei.intr();
```



定义对象时  
window

lilei : 0x1234

地址: 0x1234

{ //new Object()

    sname: "Li Lei",

    sage:11,

    intr: function(){ ... }


        new Function()

}

```
var lilei={
```

```
    sname: "Li Lei",
```

```
    sage:11,
```

```
    intr:function(){
```

```
        console.log(`I'm ${sname}`);
```

```
    }
```

```
}
```

```
lilei.intr();
```



定义对象时  
window

lilei : 0x1234

地址: 0x1234

```
{ //new Object()  
  sname: "Li Lei",  
  sage:11,  
  intr: 0x9091  
}
```

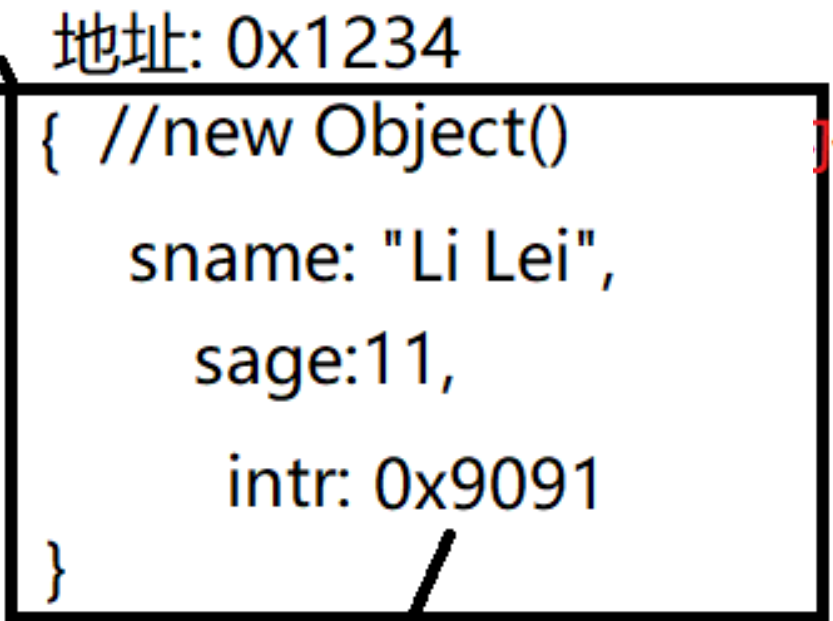
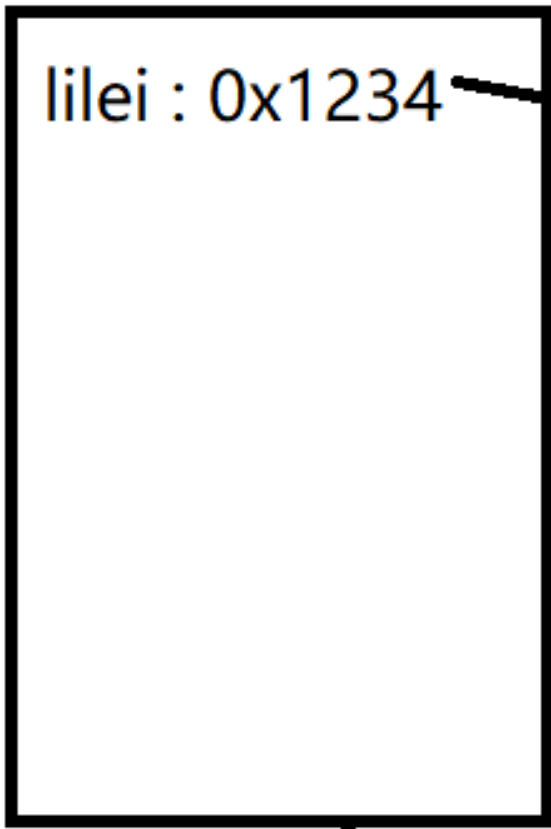
```
var lilei={  
  sname: "Li Lei",  
  sage:11,  
  intr:function(){  
    console.log(`I'm ${sname}`);  
  }  
}  
lilei.intr();
```

地址: 0x9091

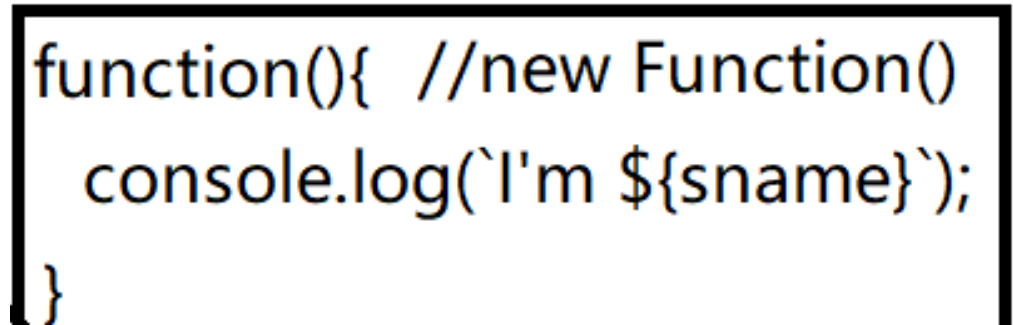
```
function(){ //new Function()  
  console.log(`I'm ${sname}`);  
}
```

# 定义对象时 window

快递小哥，踹门丢快递到家里沙发；  
在没有对象允许的情况，万万不能进入对象内部查找变量； lilei.sname



地址: 0x9091 即没有形参数，也没有var



2. 全局作用域window  
`var lilei={` *//不是作用域!*  
*//所以进不了函数作用域链*  
`sname: "Li Lei",`  
`sage:11,`  
`intr:function(){` *1. 函数作用域*  
 `console.log(`I'm ${sname}`);`  
`}`  
`}`  
`lilei.intr();`

定义对象时  
window

lilei : 0x1234

地址: 0x1234

```
{ //new Object()
```

```
  sname: "Li Lei",
```

```
  sage:11,
```

```
  intr: 0x9091
```

```
}
```

2. 全局作用域window

```
var lilei={ //不是作用域!
```

//所以进不了函数作用域链

```
  sname: "Li Lei",
```

```
  sage:11,
```

```
  intr:function(){
```

```
    console.log(`I'm ${sname}`);
```

```
}
```

```
lilei.intr();
```

1. 函数作用域

地址: 0x9091

```
function(){ //new Function()
```

```
  console.log(`I'm ${sname}`);
```

```
  作用域链
```

```
}
```

自己 空

window



调用lilei.intr()时

window

lilei : 0x1234

地址: 0x1234

```
{ //new Object()  
  sname: "Li Lei",  
  sage: 11,  
  intr: 0x9091  
}
```

不是作用域，进不了

函数的作用域链

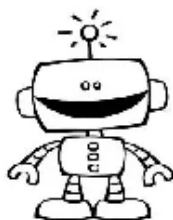
地址: 0x9091

```
function(){ //new Function()  
  console.log('I'm ${sname}');  
}
```

作用域链

自己

window



做菜三部曲:

1. 备料

封装

继承

多态

lilei.intr();

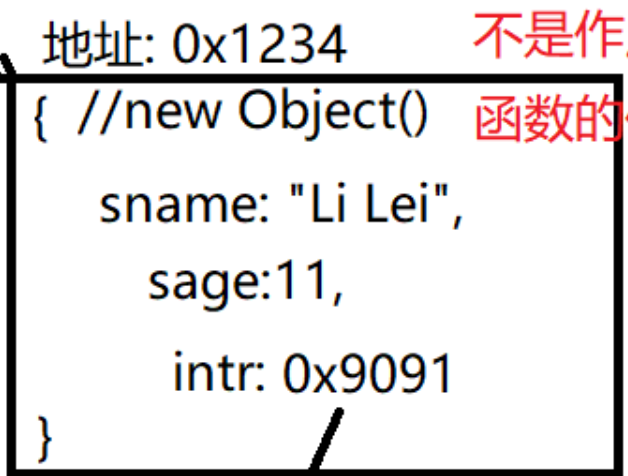
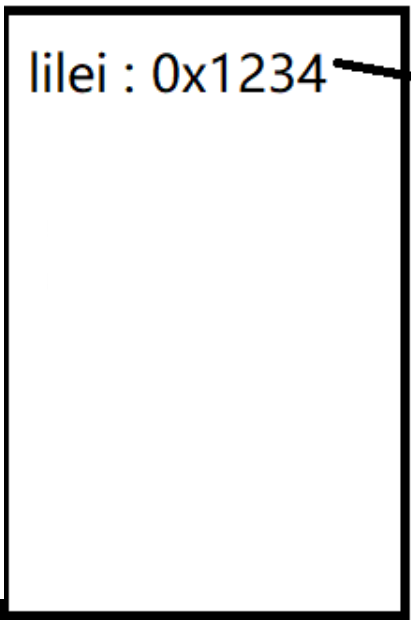
调用lilei.intr()时

封装

继承

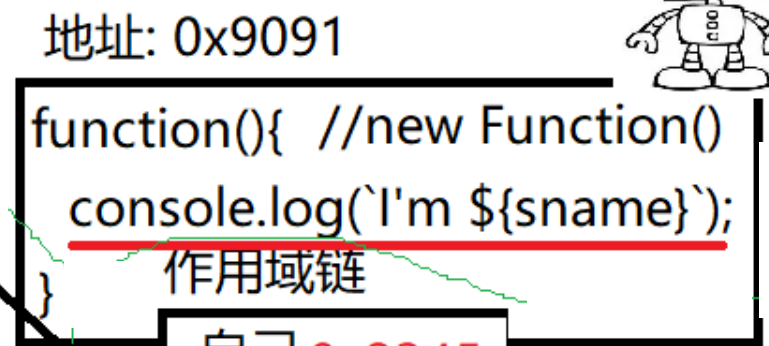
多态

window

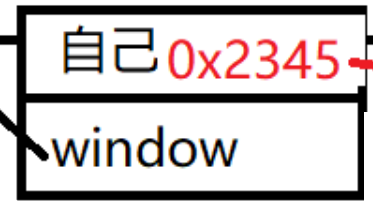


不是作用域，进不了  
函数的作用域链

lilei.intr();



做菜三部曲:  
1. 备料  
临时创建  
intr的函数作用域对象  
地址: 0x2345



调用lilei.intr()时

window

lilei : 0x1234

地址: 0x1234

```
{ //new Object()
```

sname: "Li Lei",

sage:11,

intr: 0x9091

}

不是作用域，进不了

函数的作用域链

封装

继承

多态

lilei.intr();

地址: 0x9091

```
function(){ //new Function()
```

console.log(`I'm \${sname}`);

}

作用域链

自己 0x2345

window

做菜三部曲:

2. 按菜谱做菜

临时创建

intr的函数作用域对象

地址: 0x2345

空





调用lilei.intr()时

window

lilei : 0x1234



地址: 0x1234

```
{ //new Object()
```

sname: "Li Lei",

sage:11,

intr: 0x9091

```
}
```

不是作用域，进不了

函数的作用域链

封装

继承

多态

lilei.intr();

地址: 0x9091

```
function(){ //new Function()
```

console.log(`I'm \${sname}`);

```
}
```

作用域链

自己 0x2345

window

做菜三部曲:

2. 按菜谱做菜

临时创建

intr的函数作用域对象

地址: 0x2345

空



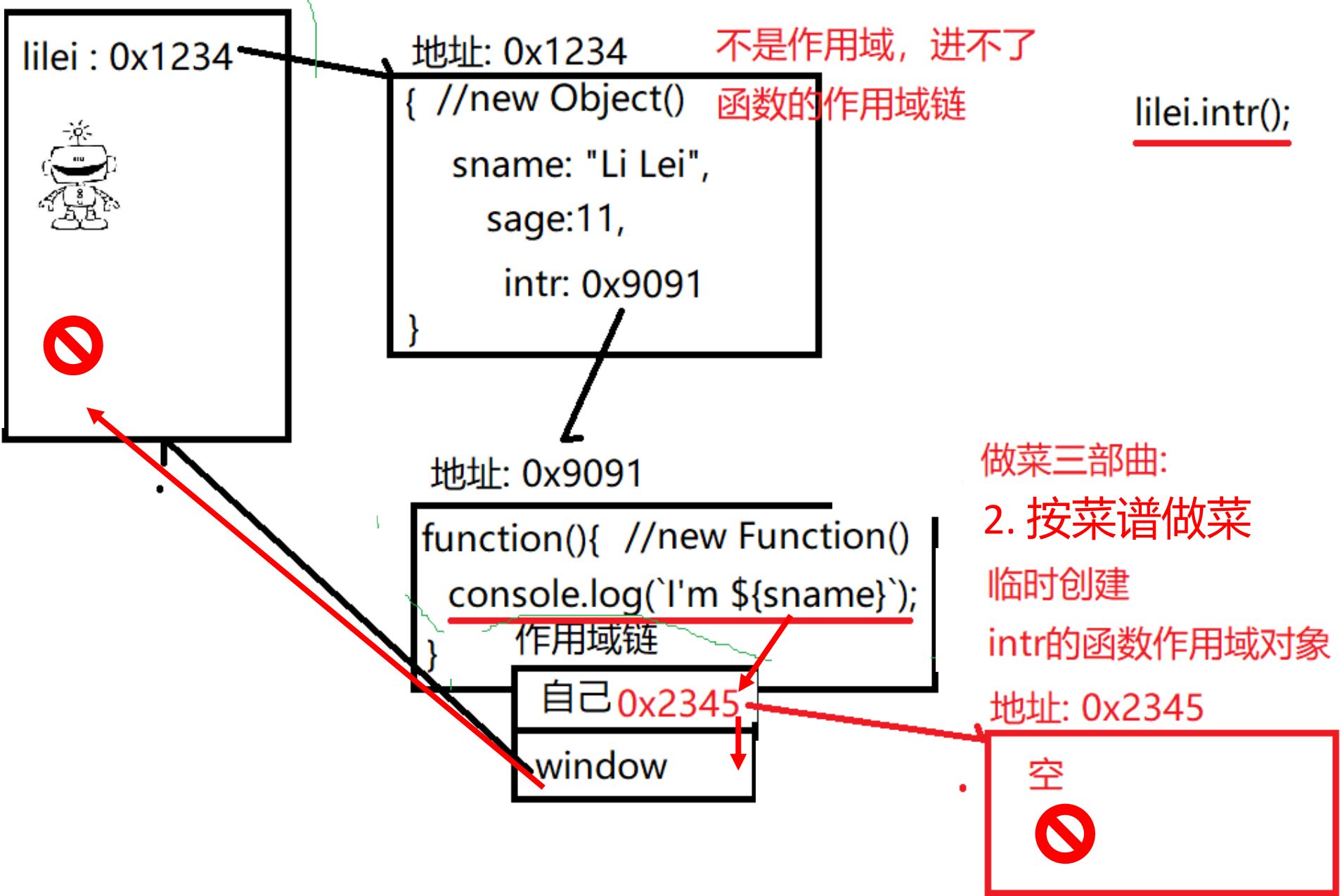
调用lilei.intr()时

封装

继承

多态

window



封装

继承

多态

## 强调:

Js引擎在没有你的程序授权情况下，  
不能擅自进入对象，读取对象的属性



调用lilei.intr()时

window

lilei : 0x1234



地址: 0x1234

{ //new Object()

sname: "Li Lei",

sage:11,

intr: 0x9091

}

不是作用域，进不了

函数的作用域链

封装

继承

多态

lilei.intr();

地址: 0x9091

function(){ //new Function()

console.log(`I'm \${sname}`);

}

作用域链

自己 0x2345

↓  
window

做菜三部曲:

2. 按菜谱做菜

临时创建

intr的函数作用域对象

地址: 0x2345

空



调用lilei.intr()时

封装



继承

多态



window

lilei : 0x1234



地址: 0x1234

不是作用域, 进不了

```
{ //new Object()
  sname: "Li Lei",
  sage:11,
  intr: 0x9091
}
```

函数的作用域链

lilei.intr();

报错: sname is not defined

地址: 0x9091

```
function(){ //new Function()
  console.log(`I'm ${sname}`);
}
```

作用域链

自己 0x2345

window

做菜三部曲:


2. 按菜谱做菜

临时创建

intr的函数作用域对象

地址: 0x2345

空





封装

继承

多态

# 解决1



# 解决1: 写死对象名: `lilei.sname` , `lilei.sage`

```
var lilei={
  sname:"Li Lei",//姓名
  sage:11,//年龄
  intr:function(){
    console.log(`I'm ${sname}, I'm ${sage}`)
  }
}
//输出lilei的年龄
console.log(lilei.sage);
//调用lilei的intr()方法, 请lilei做自我介绍
lilei.intr();
//过了一年, lilei的年龄涨了一岁
lilei.sage++;
//再次输出lilei的年龄
console.log(lilei.sage);
//再次调用lilei的intr()方法, 请lilei做自我介绍
lilei.intr();
```

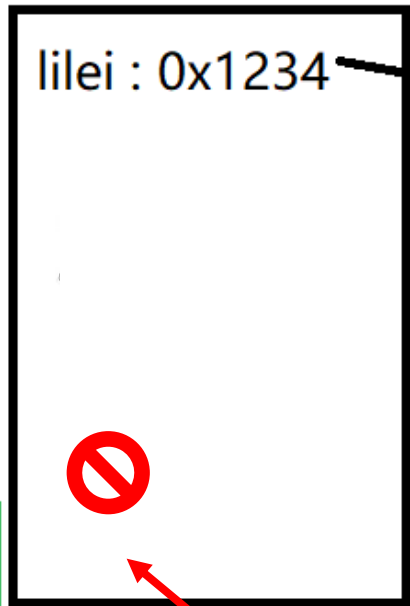
调用lilei.intr()时

封装

继承

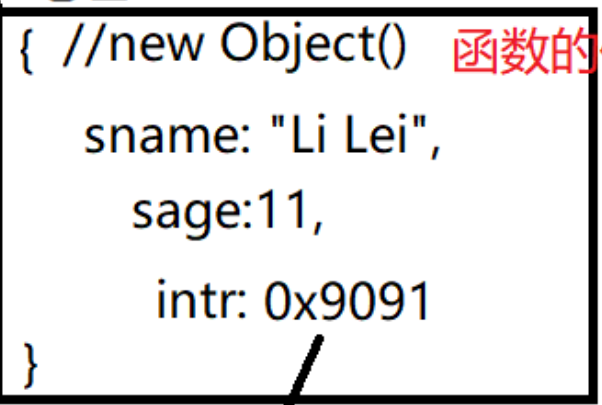
多态

window



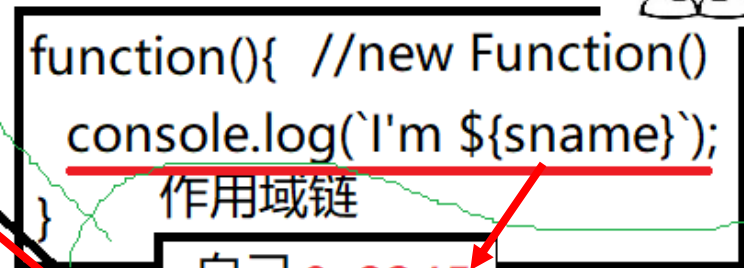
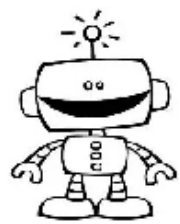
地址: 0x1234

不是作用域, 进不了



函数的作用域链

地址: 0x9091



作用域链

自己 0x2345

window

做菜三部曲:

2. 按菜谱做菜

临时创建

intr的函数作用域对象

地址: 0x2345



lilei.intr();

报错: sname is not defined

定义对象时 写死对象名

window

lilei : 0x1234

地址: 0x1234

```
{ //new Object()
```

sname: "Li Lei",

sage:11,

intr: 0x9091

```
}
```

不是作用域，进不了

函数的作用域链

地址: 0x9091

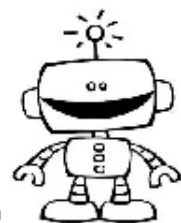
```
function(){ //new Function()
```

```
console.log(`I'm ${ lilei. sname}`);
```

作用域链

自己 0x2345

window



做菜三部曲: lilei.intr();

2. 按菜谱做菜

临时创建

intr的函数作用域对象

地址: 0x2345

空

封装

继承

多态



调用lilei.intr()时 对象名

window

lilei : 0x1234

地址: 0x1234

{ //new Object()

sname: "Li Lei",

sage:11,

intr: 0x9091

}

不是作用域，进不了

函数的作用域链

地址: 0x9091

function(){ //new Function()

console.log(`I'm \${ lilei. sname}`);

作用域链

自己 0x2345

window

做菜三部曲: lilei.intr();

2. 按菜谱做菜

临时创建

intr的函数作用域对象

地址: 0x2345

空



封装

继承

多态



window

lilei : 0x1234



地址: 0x1234

```
{ //new Object()
  sname: "Li Lei",
  sage: 11,
  intr: 0x9091
}
```

不是作用域，进不了

函数的作用域链

地址: 0x9091

```
function(){ //new Function()
  console.log(`I'm ${ lilei. sname }`);
}
```

做菜三部曲: lilei.intr();

2. 按菜谱做菜

临时创建

intr的函数作用域对象

作用域链

自己 0x2345

window

地址: 0x2345

空





调用lilei.intr()时 对象名

封装

继承

多态

window

lilei : 0x1234

地址: 0x1234

不是作用域, 进不了

{ //new Object()

函数的作用域链

sname: "Li Lei",  
sage:11,

intr: 0x9091

地址: 0x9091

function(){ //new Function()

console.log(`I'm \${ lilei. sname}`);

临时创建

作用域链

intr的函数作用域对象

自己 0x2345

地址: 0x2345

window

空



调用lilei.intr()时 对象名

window

封装

继承

多态



lilei : 0x1234

地址: 0x1234

不是作用域, 进不了

函数的作用域链

```
{ //new Object()
  sname: "Li Lei",
  sage:11,
  intr: 0x9091
}
```



地址: 0x9091

做菜三部曲: lilei.intr(); "Li Lei",

2. 按菜谱做菜

```
function(){ //new Function()
```

```
  console.log(`I'm ${ lilei. sname}`);
```

临时创建

作用域链

intr的函数作用域对象

自己 0x2345

地址: 0x2345

window



空



封装

继承

多态

## 问题： 一旦对象名发生变化，

而忘记修改对象内写死的对象名，  
都会报错！ “xxx is not defined!”

——紧耦合，是不好的解决方法



封装

继承

多态

## 解决2: 用this



# 什么是this

封装

继承

多态

- 每个函数内自带的
- 专门指向正在调用当前函数的**.前的对象**
- 的关键字



# 什么是this

封装

继承

多态

The image shows a web browser's developer tools interface. The top bar includes tabs for '元素' (Elements), '控制台' (Console), '来源' (Sources), '网络' (Network), '性能' (Performance), '内存' (Memory), '应用' (Application), '安全' (Security), and 'Lighthouse'. The '来源' (Sources) tab is active, showing a file named '1\_this.html'. The code in the editor is as follows:

```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1">
8   <title>Document</title>
9 </head>
10
11 <body>
12   <script>
13     var lilei = {
14       sname: "Li Lei", //姓名
15       sage: 11, //年龄
16       intr: function () {
17         debugger;
18         console.log(`I'm ${lilei.sname}`);
19       }
20     }
21     //输出lilei的年龄
22     console.log(lilei.sage);
23     //调用lilei的intr()方法, 请lilei做自我介绍
24     lilei.intr();
```

The 'Scope' panel on the right shows the current execution context. It is set to '本地' (Local) and shows a variable 'this' of type 'Object'. A red arrow points to this object, which contains the following properties:

- `intr`: `f ()`
- `sage`: `11`
- `sname`: `"Li Lei"`
- `[[Prototype]]`: `Object`

The '调用堆栈' (Call Stack) panel at the bottom shows the current call stack. It includes the function `intr` at line 17 of `1_this.html` and an anonymous function at line 24 of `1_this.html`.

# 什么是this

封装

继承

多态



已在调试程序...

元素 控制台 来源 网络 性能 内存 应用 安全 Lighthouse

网页 >> 1\_this.html x

```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" co
7   <meta name="viewport" content="width=
8   <title>Document</title>
9 </head>
10
11 <body>
12   <script>
13     var lilei = {
14       sname: "Li Lei", //姓名
15       sage: 11, //年龄
16       intr: function () {
17         debugger;
18         console.log(`I'm ${lilei.sname}`);
19       }
20     }
21     //输出lilei的年龄
22     console.log(lilei.sage);
23     //调用lilei的intr()方法, 请lilei做自
24     lilei.intr();
```

在遇到异常时暂停

调试程序已暂停

监视

断点

无断点

作用域

本地

this: Object

- intr: f ()
- sage: 11
- sname: "Li Lei"
- [[Prototype]]: Object

全局 Window

调用堆栈

- intr 1\_this.html:17
- (匿名) 1\_this.html:24
- XHR/提取断点

调用lilei.intr()时

对象名

封装

继承

多态

window

lilei : 0x1234

地址: 0x1234

不是作用域, 进不了

```
{ //new Object()
```

函数的作用域链

sname: "Li Lei",

sage:11,

intr: 0x9091

```
}
```

地址: 0x9091

```
function(){ //new Function()
```

```
console.log(`I'm ${ lilei. sname}`);
```

临时创建

作用域链

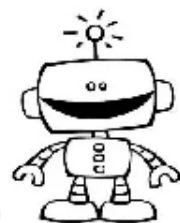
intr的函数作用域对象

自己 0x2345

window

地址: 0x2345

空



做菜三部曲: lilei.intr(); "Li Lei",

1. 备料

调用lilei.intr() 用this

封装

继承

多态



window

lilei : 0x1234

地址: 0x1234

不是作用域, 进不了

```
{ //new Object()
```

函数的作用域链

```
  sname: "Li Lei",  
  sage:11,
```

```
  intr: 0x9091  
}
```

地址: 0x9091

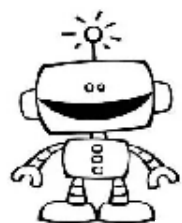
```
function(){ //new Function()
```

```
  console.log(`I'm ${ this. sname}`);
```

作用域链

自己 0x2345

window



做菜三部曲: lilei.intr();

1. 备料

临时创建

intr的函数作用域对象

地址: 0x2345

空

调用lilei.intr()

用this

封装

继承

多态

window

lilei : 0x1234

地址: 0x1234

```
{ //new Object()
```

sname: "Li Lei",

sage:11,

intr: 0x9091

}

不是作用域，进不了

函数的作用域链

地址: 0x9091

```
function(){ //new Function()
```

console.log(`I'm \${ this.sname}`);

作用域链

自己 0x2345

window

做菜三部曲: lilei.intr();

1. 备料

临时创建

intr的函数作用域对象

地址: 0x2345

空  
this



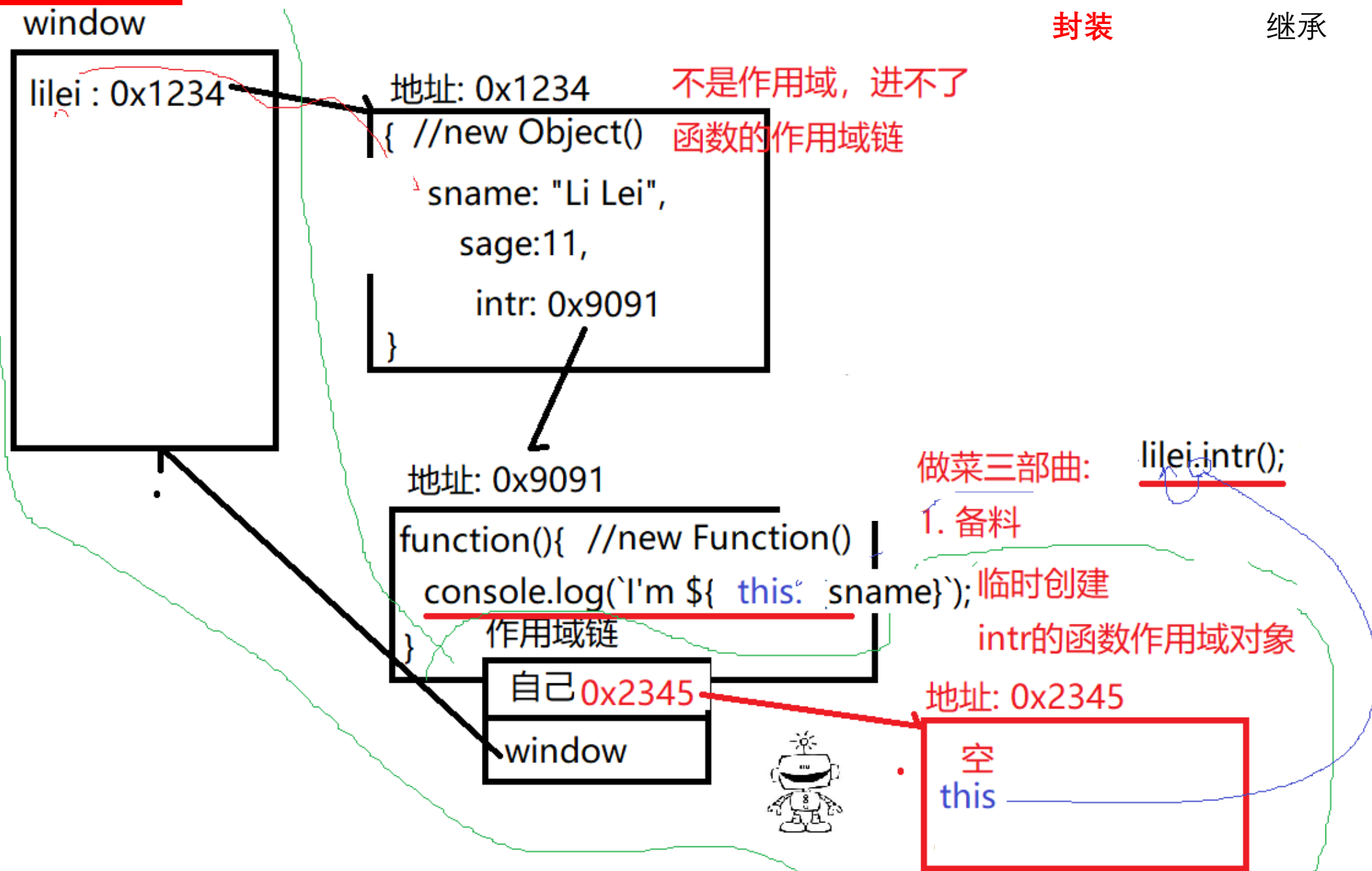


调用lilei.intr() 用this

封装

继承

多态

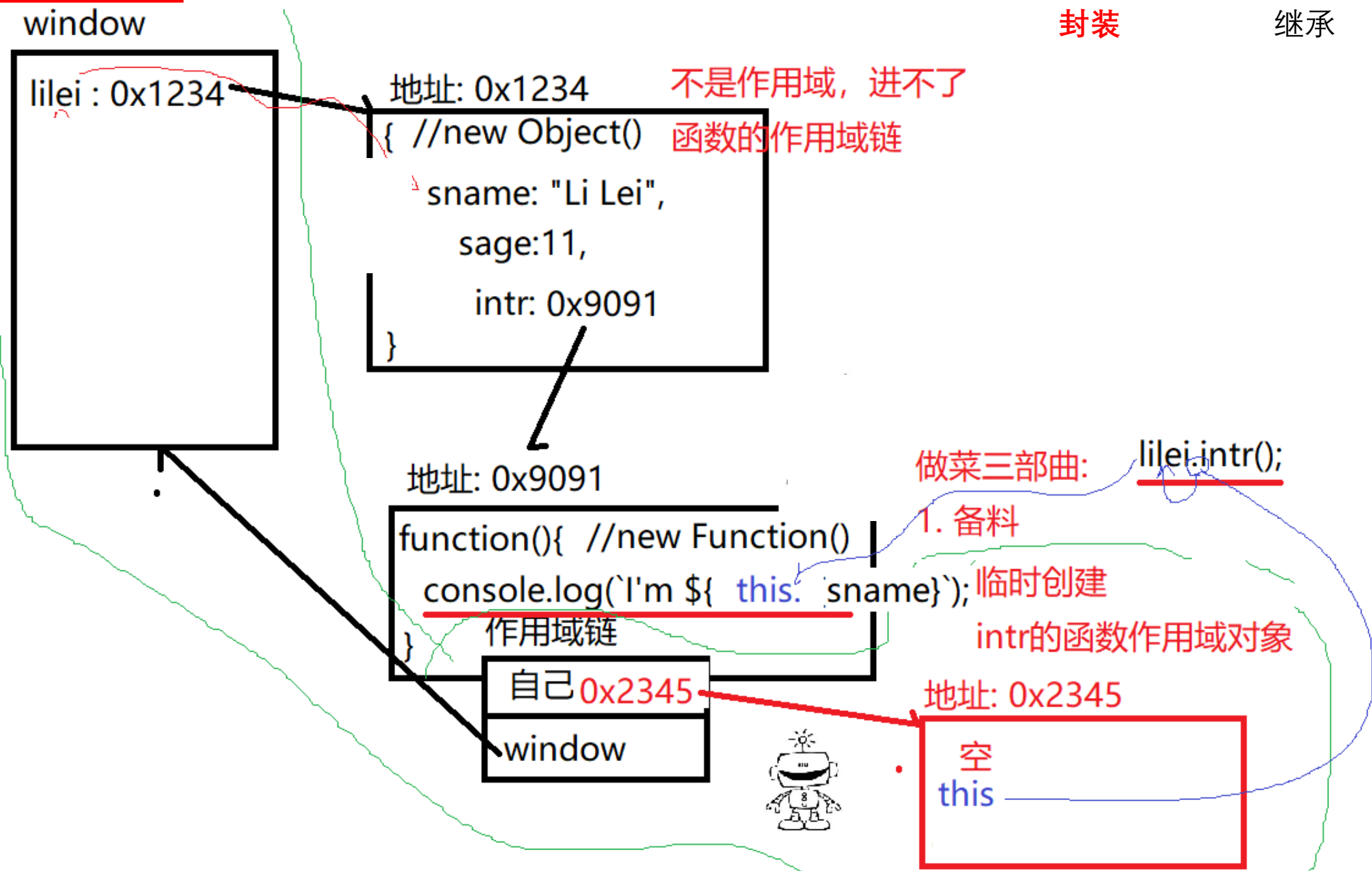


调用lilei.intr() 用this

封装

继承

多态

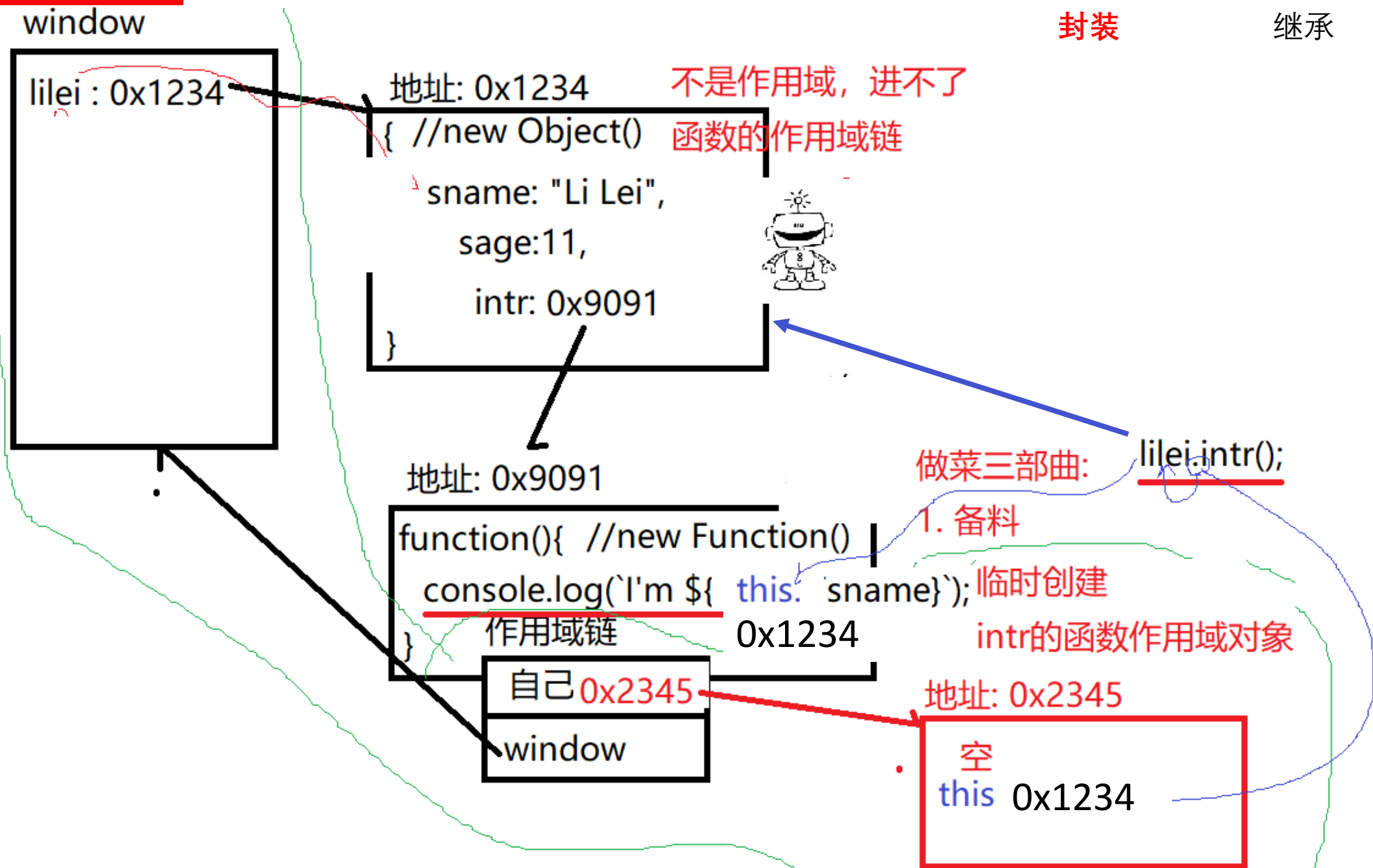


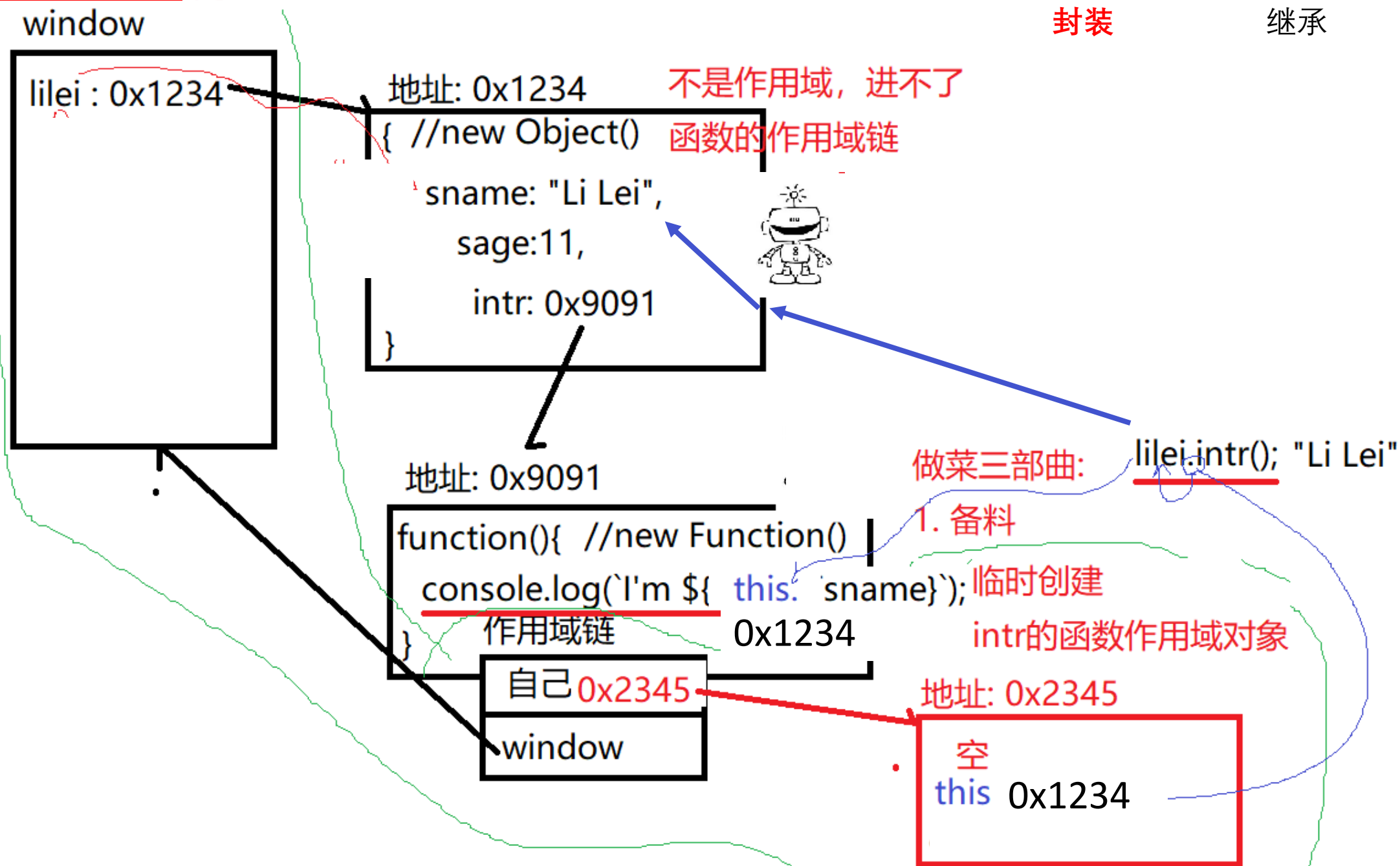
调用lilei.intr() 用this

封装

继承

多态





封装

继承

多态

## 总结:

今后，对象中的方法中，只要想使用当前对象自己的属性或其他方法时，都要加this





封装

继承

多态

第二种方式：  
用new：  
new Object()

+  
○ •

封装

继承

多态

## 如何: 2步

```
var 对象名=new Object(); //先创建空对象{}  
//强行给空对象中添加新属性和新方法  
对象名.新属性=属性值;  
对象名.新方法=function(){ ... this.属性名 ... }
```

**揭示了：**  
**js语言底层最核心的原理：**

封装

继承

多态



封装

继承

多态

其实

js中所有对象底层都是  
关联数组





封装

继承

多态

证据





## 关联数组

---

```
▼ [sname: "Han Meimei", sage: 11]  
  sage: 11  
  sname: "Han Meimei"  
  length: 0  
  ► [[Prototype]]: Array(0)
```

---

## 对象

---

```
▼ {sname: "Li Lei", sage: 11}  
  sage: 11  
  sname: "Li Lei"  
  ► [[Prototype]]: Object
```

---

i. 存储结构：都是名值对儿的组合

封装

继承

多态

## ii. 访问成员时:

- 标准写法都是: 对象名/数组名["成员名"]
  - 简写都是: 对象名/数组名.成员名
- 比如: `lilei["sname"]`和`lilei.sname`都行



### iii. 强行给 不存在的位置赋值

- 不但不会报错，而且还会自动添加该属性。
- 所以，js中，给对象添加一个新属性或新方法，都可通过强行赋值的方式。

## iv. 强行访问 不存在的位置 的值

- 都不会报错，而是返回undefined。
- 所以，可以使用  
if(对象.成员名!==undefined)  
判断一个对象中是否包含某个成员

# v. 都可以用for in循环遍历!

```
for(var 属性名 in 对象名/数组名){
```

//in 会自动依次取出对象或数组中每个:前的属性名, 保存到in前的变量中

//如果想进一步获得属性值:

对象名/数组名[属性名]

```
}
```



## 强调：访问对象属性/数组元素，其实有3种方式

- 标准：如果下标名后数组名时已知的，对象或数组[“已知的属性名或下标名”]
- 简写：如果下标名后数组名时已知的
  - 如果下标名是数字，可简写为[已知的数字下标]
  - 如果下标名是非数字的字符串，可简写为.已知的属性名
- 如果属性名或下标不是写死的！来自于一个变量！只能用[变量]，不能用.

封装

继承

多态

# 第三种方式： 用构造函数：



# 问题

- 用{}一次只能创建一个对象。
- 如果想创建多个相同结构的对象时，
- 代码就会很多重复！
- ——极其不便于将来的维护。

```
var lilei = {  
  sname: "Li Lei", //姓名  
  sage: 11, //年龄  
  intr: function () {  
    console.log(  
      `I'm ${lilei.sname}... ..`  
    )  
  }  
}  
  
var hmm = {  
  sname: "Han Meimei", //姓名  
  sage: 12, //年龄  
  intr: function () {  
    console.log(  
      `I'm ${lilei.sname}... ..`  
    )  
  }  
}
```

封装

继承

多态

# 解决:今后,

只要想反复创建多个相同结构,  
只是内容不同的对象时,  
都用构造函数。



封装

继承

多态

# 什么是构造函数

描述同一类型的所有对象的统一结构的函数。





# 为什么使用构造函数

代码重用!

封装

继承

多态



# 如何使用构造函数:

## 2步

封装

继承

多态



## i. 定义构造函数:

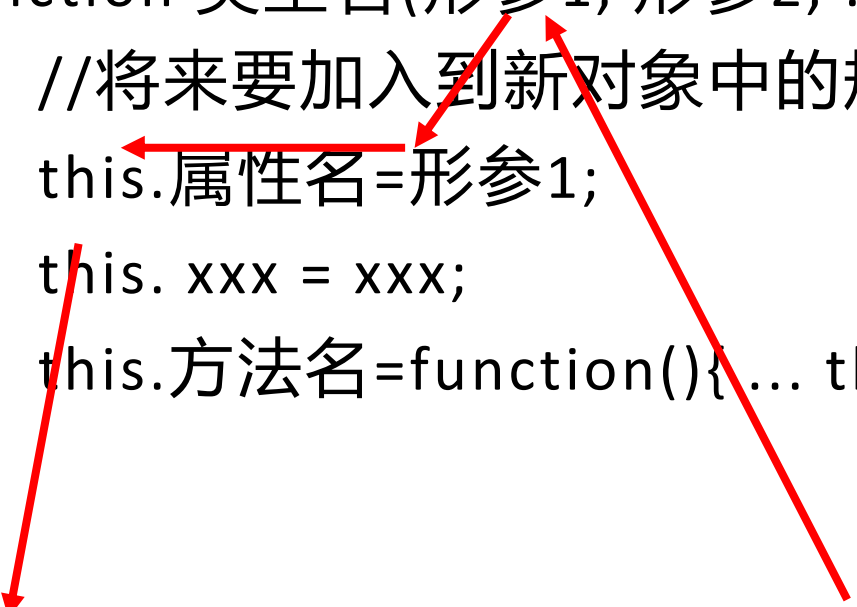
```
function 类型名(形参1, 形参2, ...){  
    //将来要加入到新对象中的规定的属性  
    this.属性名=形参1;  
    this.xxx = xxx;  
    this.方法名=function(){ ... this.属性名 ... }  
}
```

## ii. 使用构造函数 反复创建多个 相同结构的对象

- var 对象名=new 类型名(实参值1, 实参值2, ...)
- 至少做2件事儿:
  - 创建指定类型的一个新对象
  - 同时把实参值传给构造函数的形参变量。

### iii. 原理 (表面)

```
function 类型名(形参1, 形参2, ...){  
    //将来要加入到新对象中的规定的属性  
    this.属性名=形参1;  
    this.xxx = xxx;  
    this.方法名=function(){... this.属性名 ... }  
}  
  
var 对象名=new 类型名(实参值1, 实参值2, ...)
```



封装

继承

多态

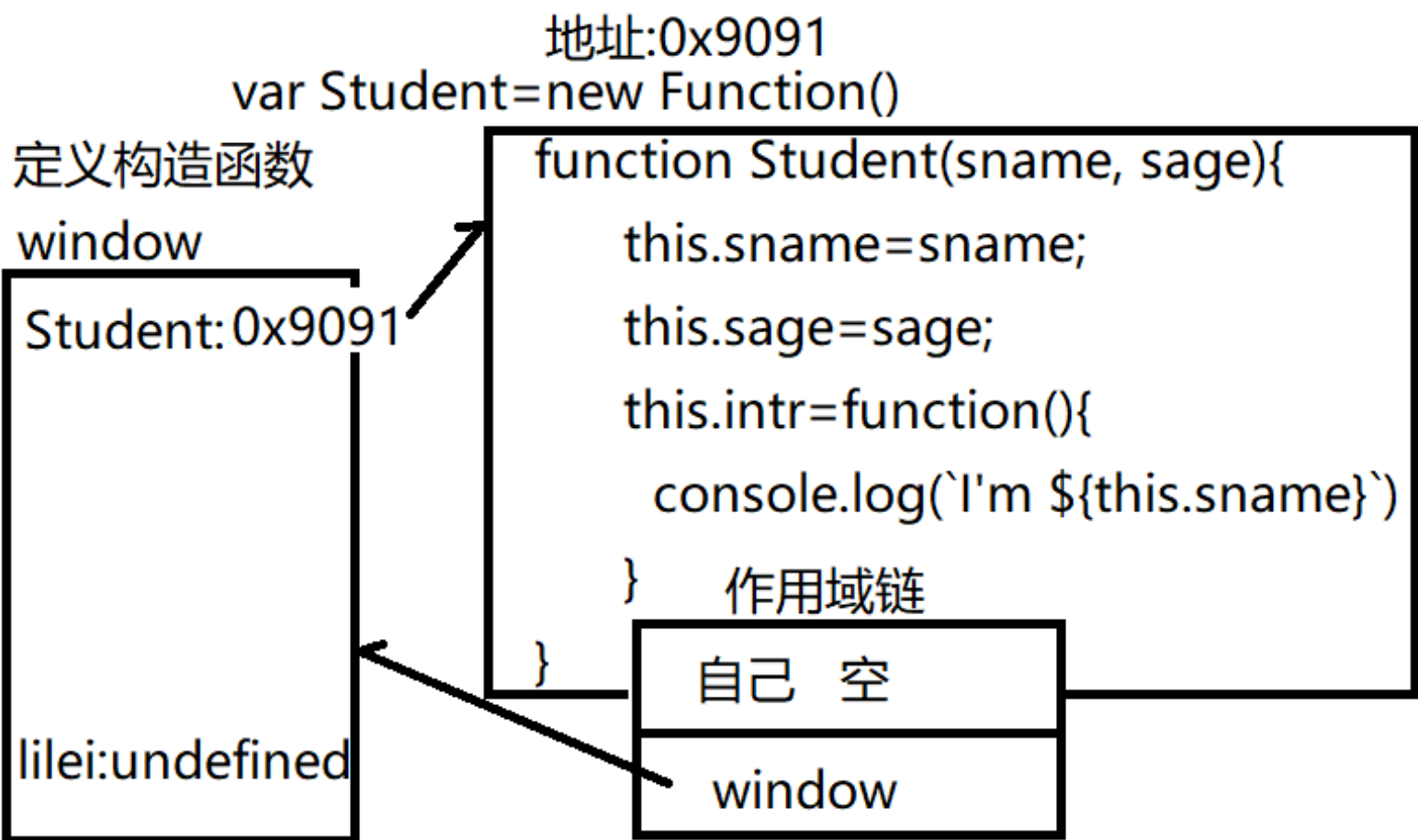
### iii. 原理（底层）： 定义构造函数时



开局

window

```
function Student(sname, sage){  
    this.sname=sname;  
    this.sage=sage;  
    this.intr=function(){  
        console.log(`I'm ${this.sname}`)  
    }  
}  
var lilei=new Student("Li Lei",11);
```



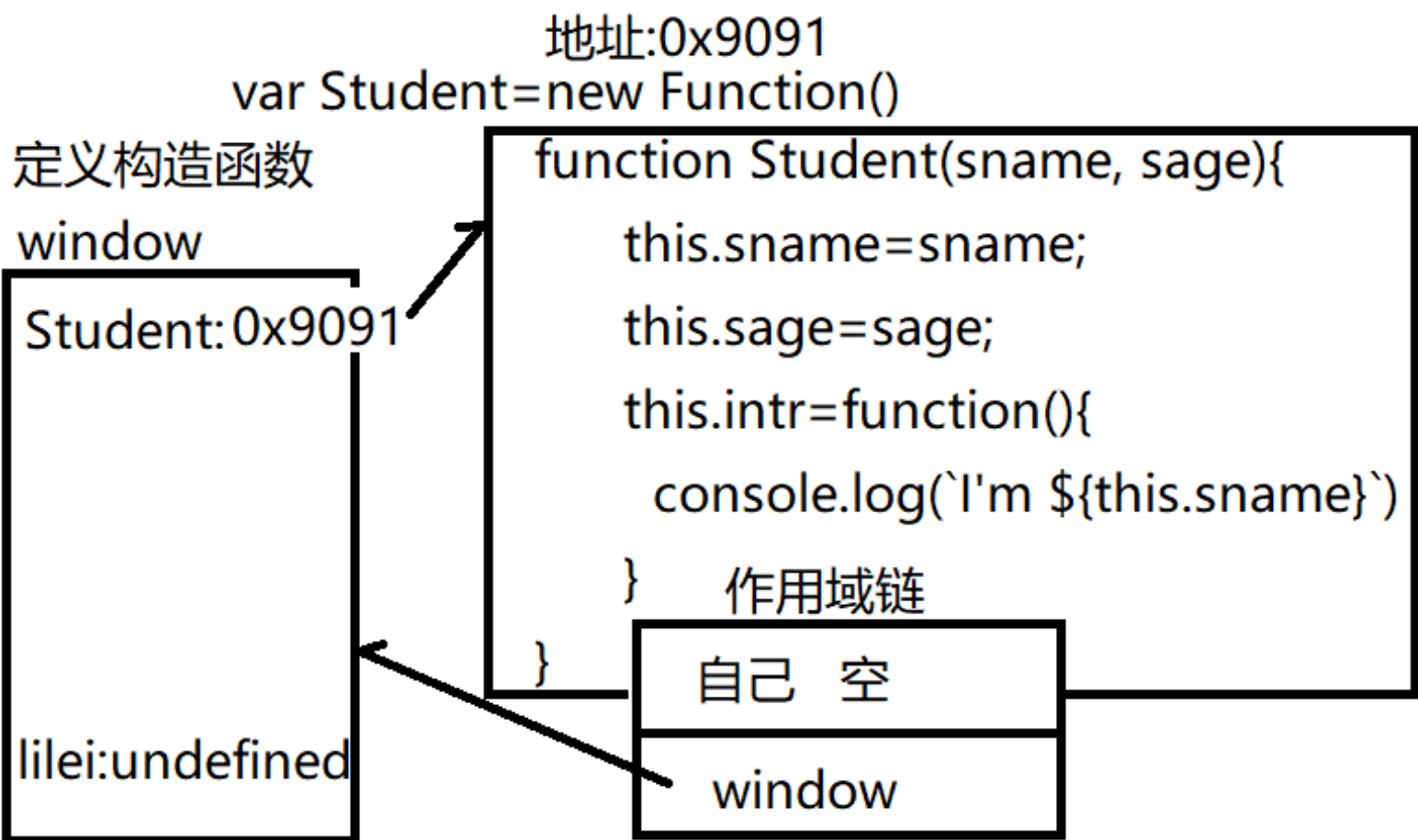
```
var lilei=new Student("Li Lei",11);
```

封装

继承

多态

# 调用构造函数时



```
var lilei=new Student("Li Lei",11);
```

封装

继承

多态

new

③

Student("Li Lei",11);

var lilei=

地址:0x9091

var Student=new Function()

定义构造函数

window

Student:0x9091

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
  this.intr=function(){  
    console.log(`I'm ${this.sname}`)  
  }  
}
```

作用域链

window

lilei:undefined



封装

继承

多态



# new做了几件事?

封装

继承

多态

**new做了4件事:**

封装

继承

多态

Student("Li Lei",11);

var lilei=

- i. 创建一个新的空对象等待

地址:0x9091

var Student=new Function()

定义构造函数

window

Student:0x9091

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
  this.intr=function(){  
    console.log(`I'm ${this.sname}`)  
  }  
}
```

作用域链

window

lilei:undefined

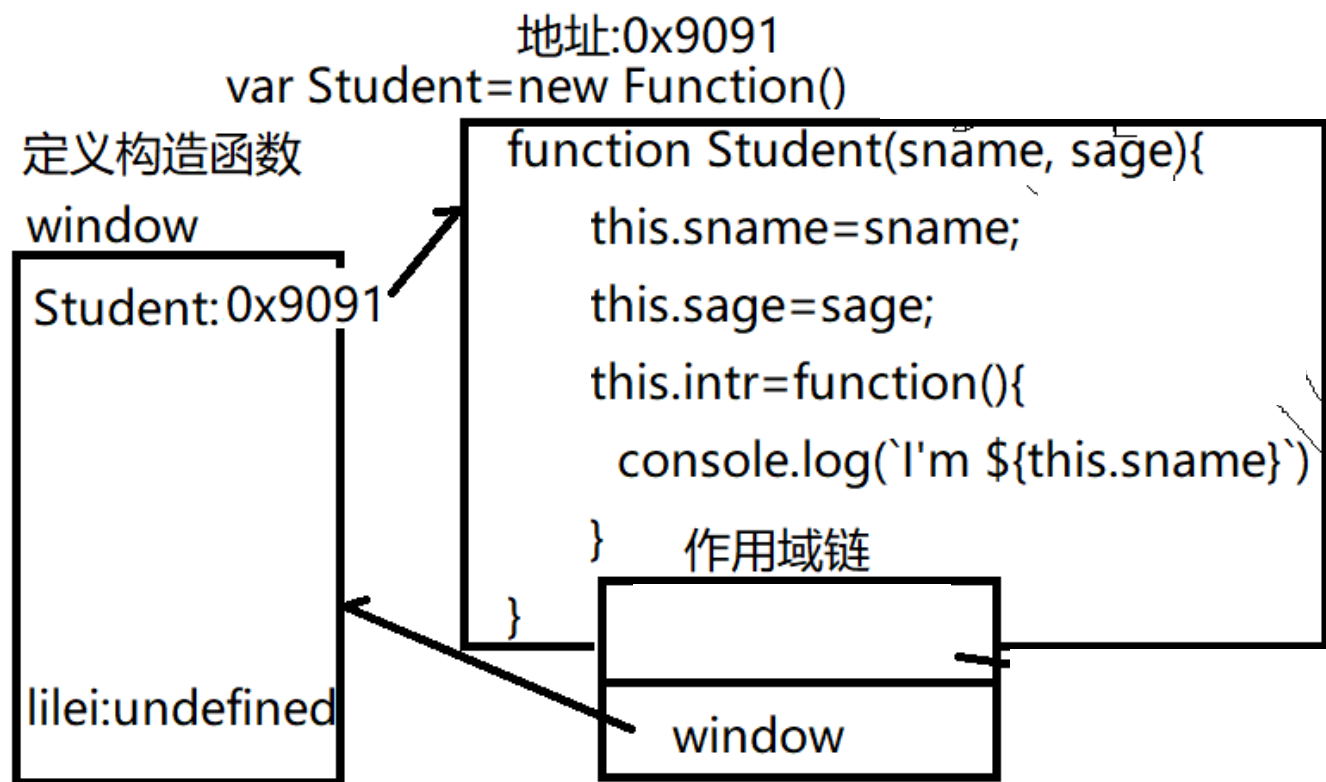
封装

继承

多态



① 封装 ③ 继承  
var lilei = Student("Li Lei", 11);



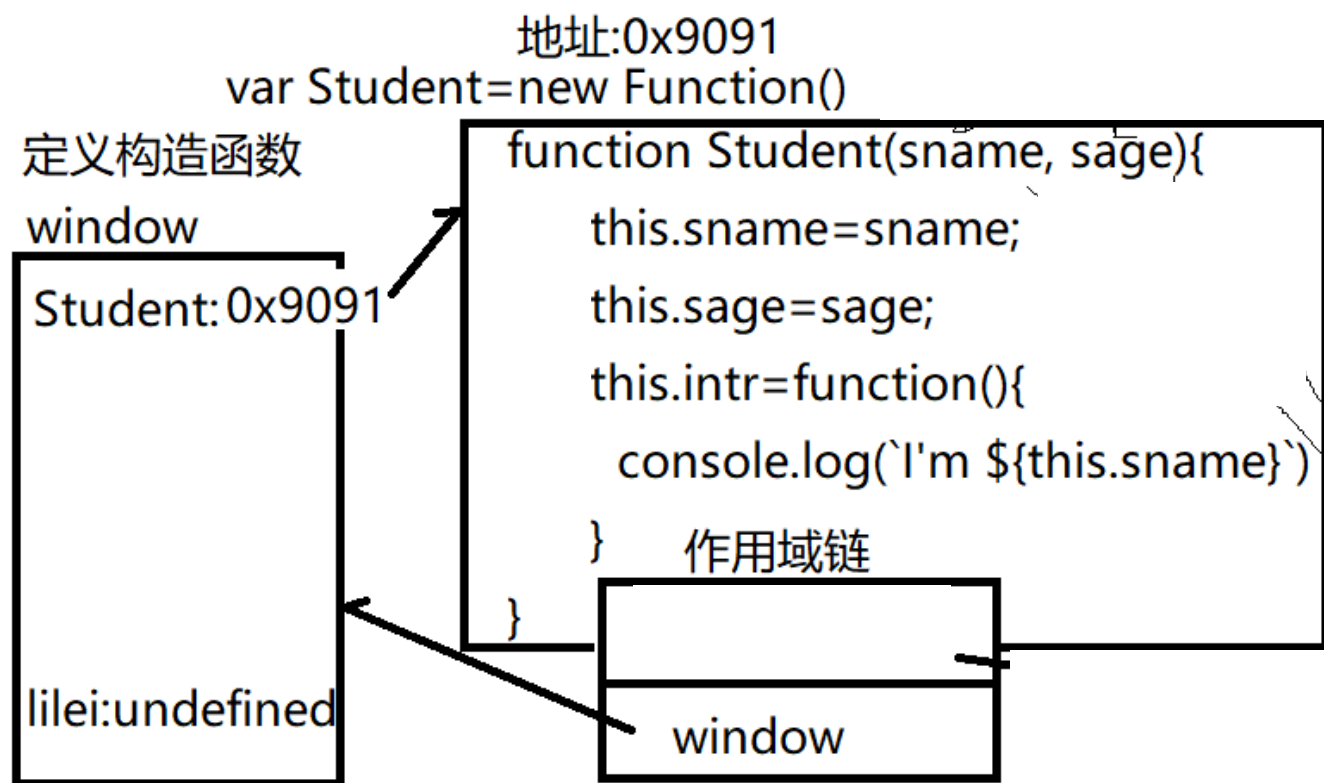
① 地址: 0x1234  
new

新对象

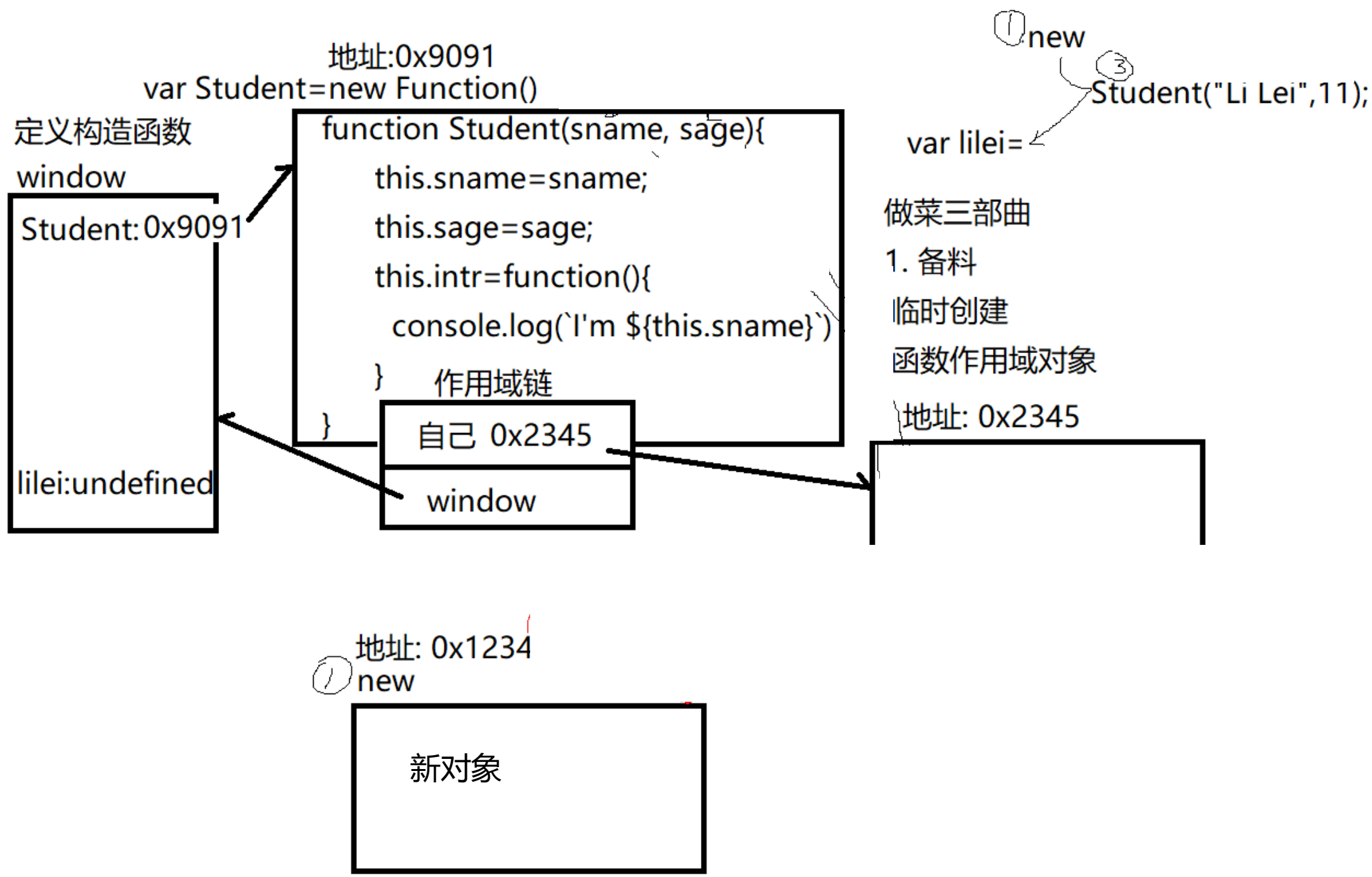
## new做了4件事:

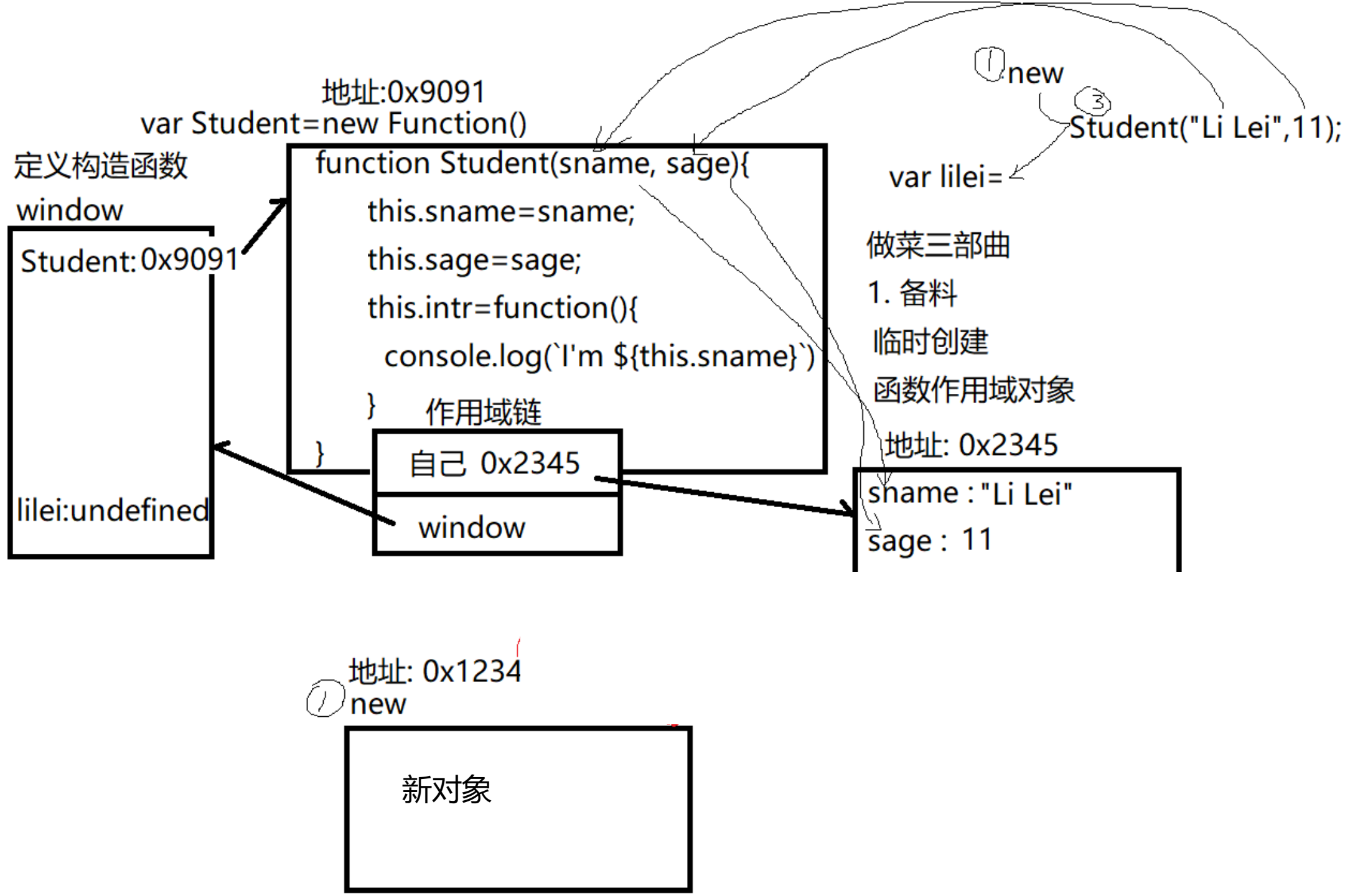
- i. 创建一个新的空对象等待
- ii. ?
- iii. 调用构造函数:
  - 将构造函数中的this->new刚创建的新对象
  - 在构造函数内通过强行赋值方式, 为新对象添加规定的属性和方法

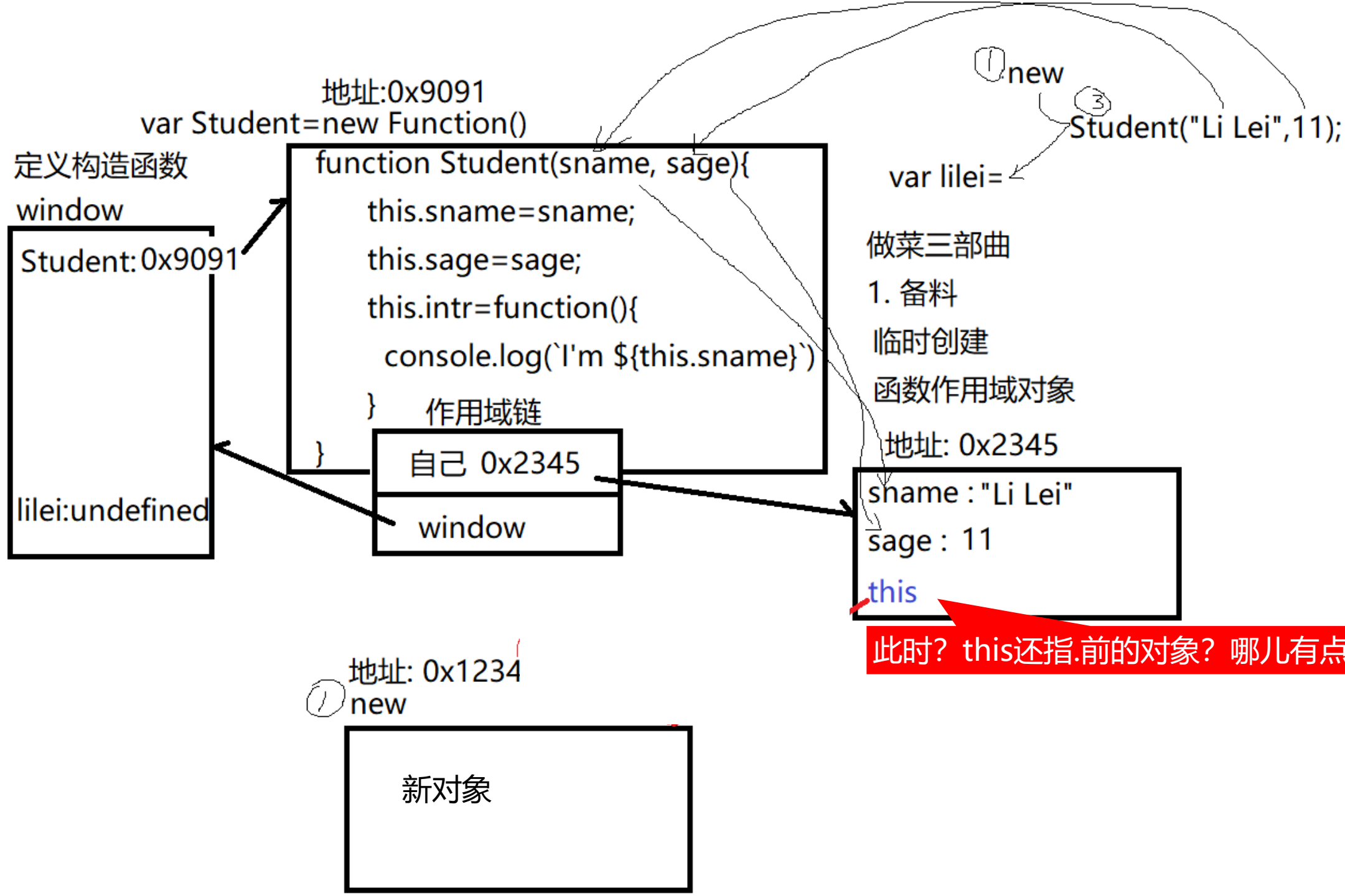


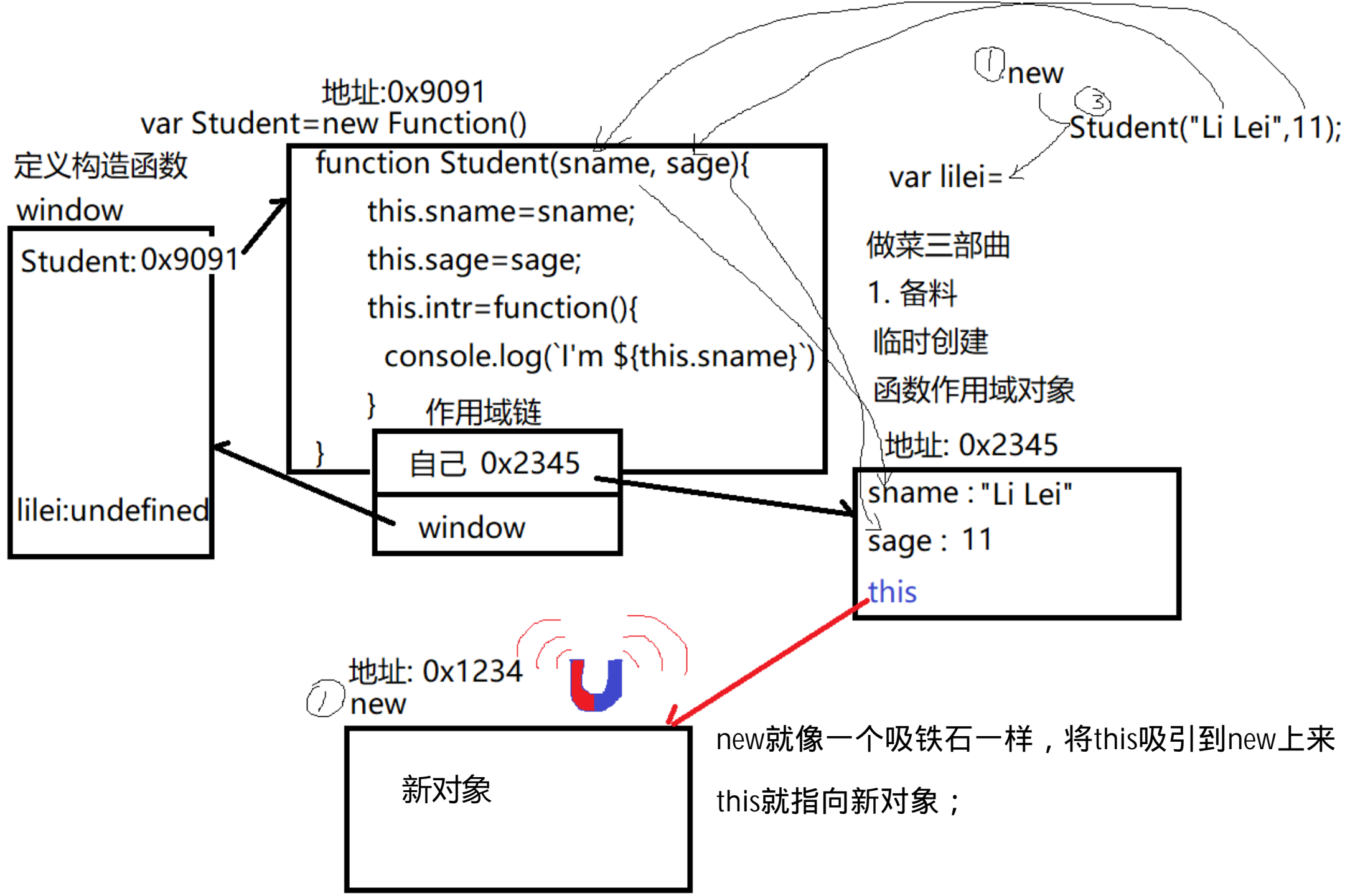


① new  
③ Student("Li Lei",11);  
var lilei=

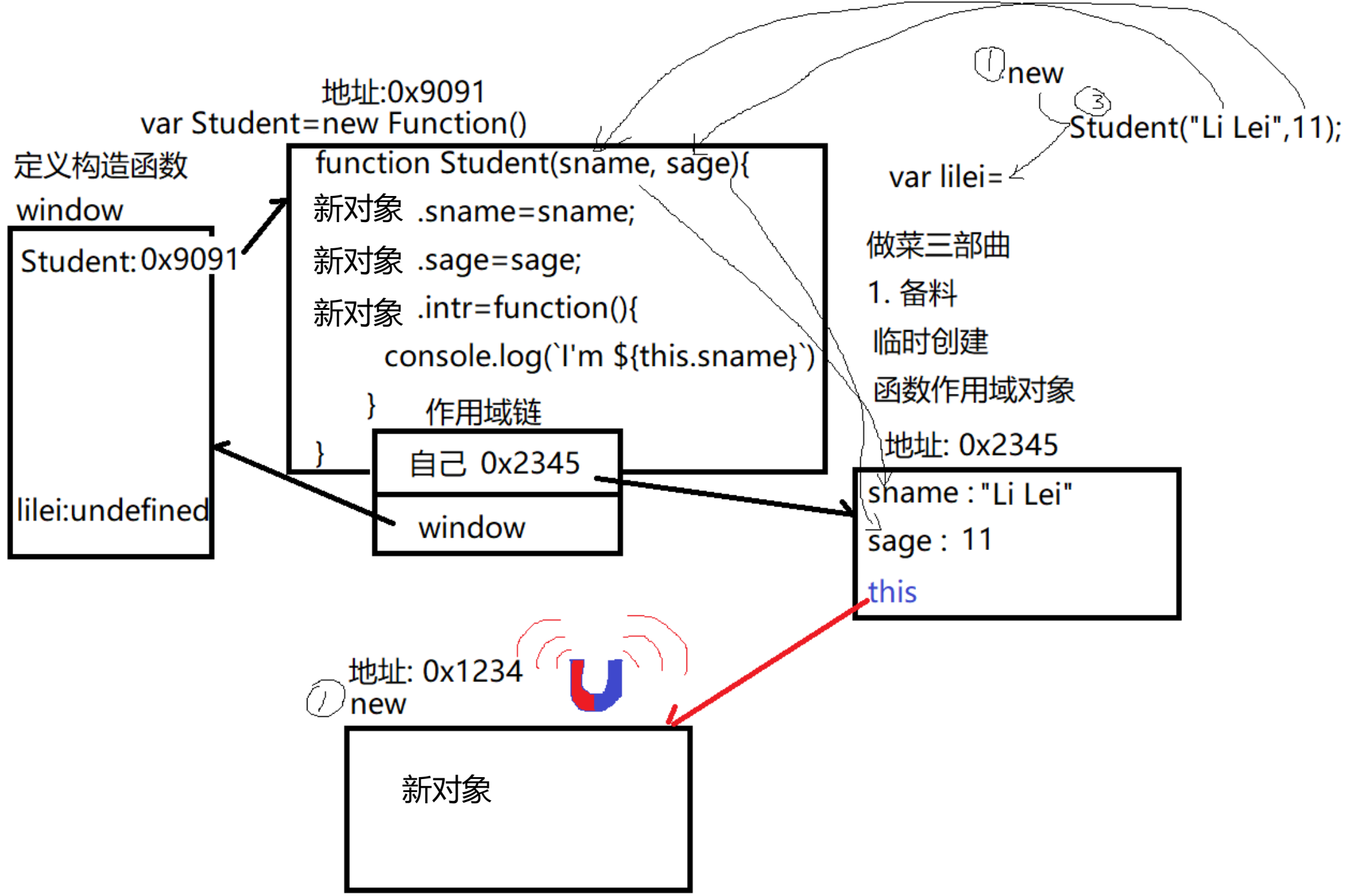


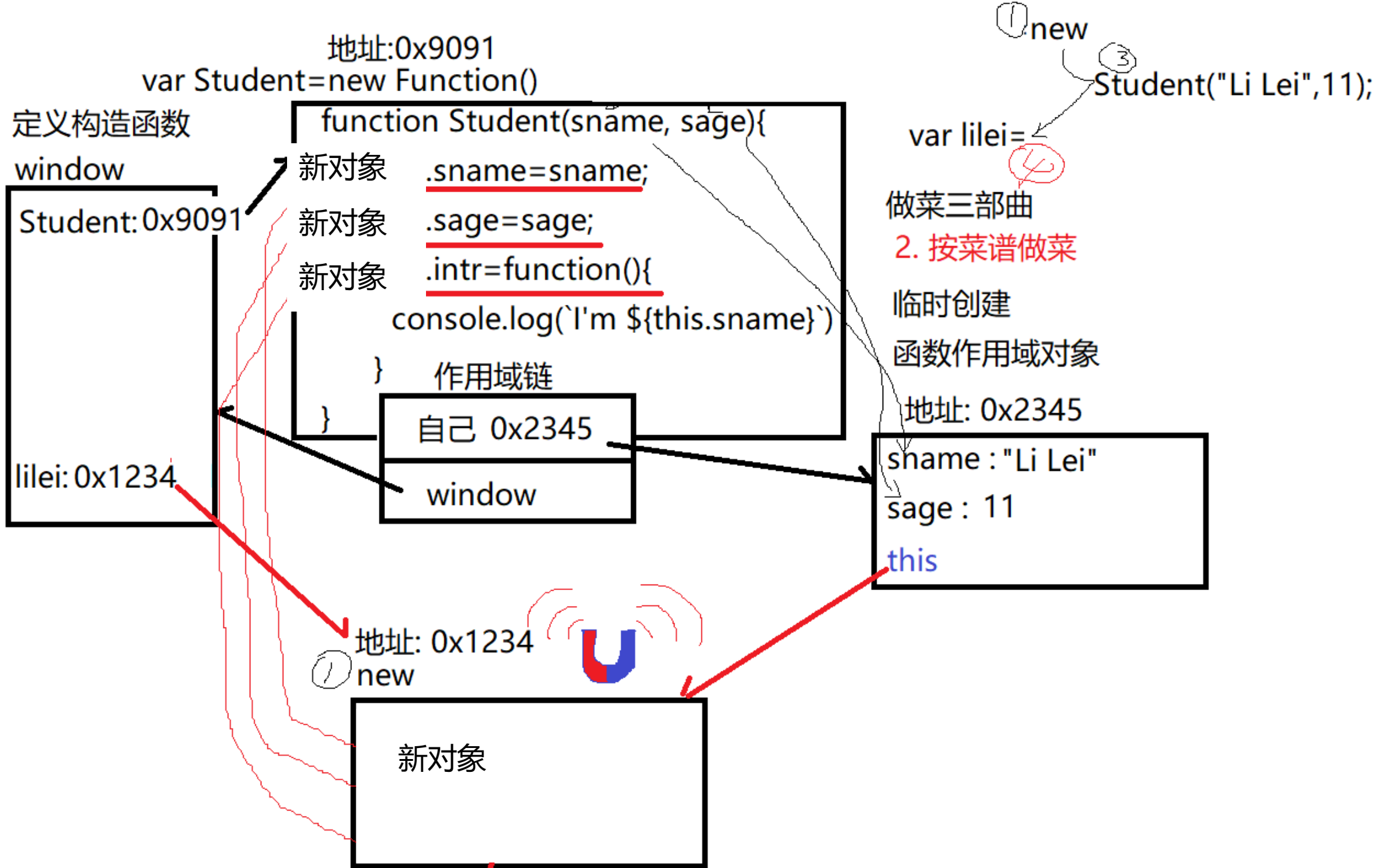


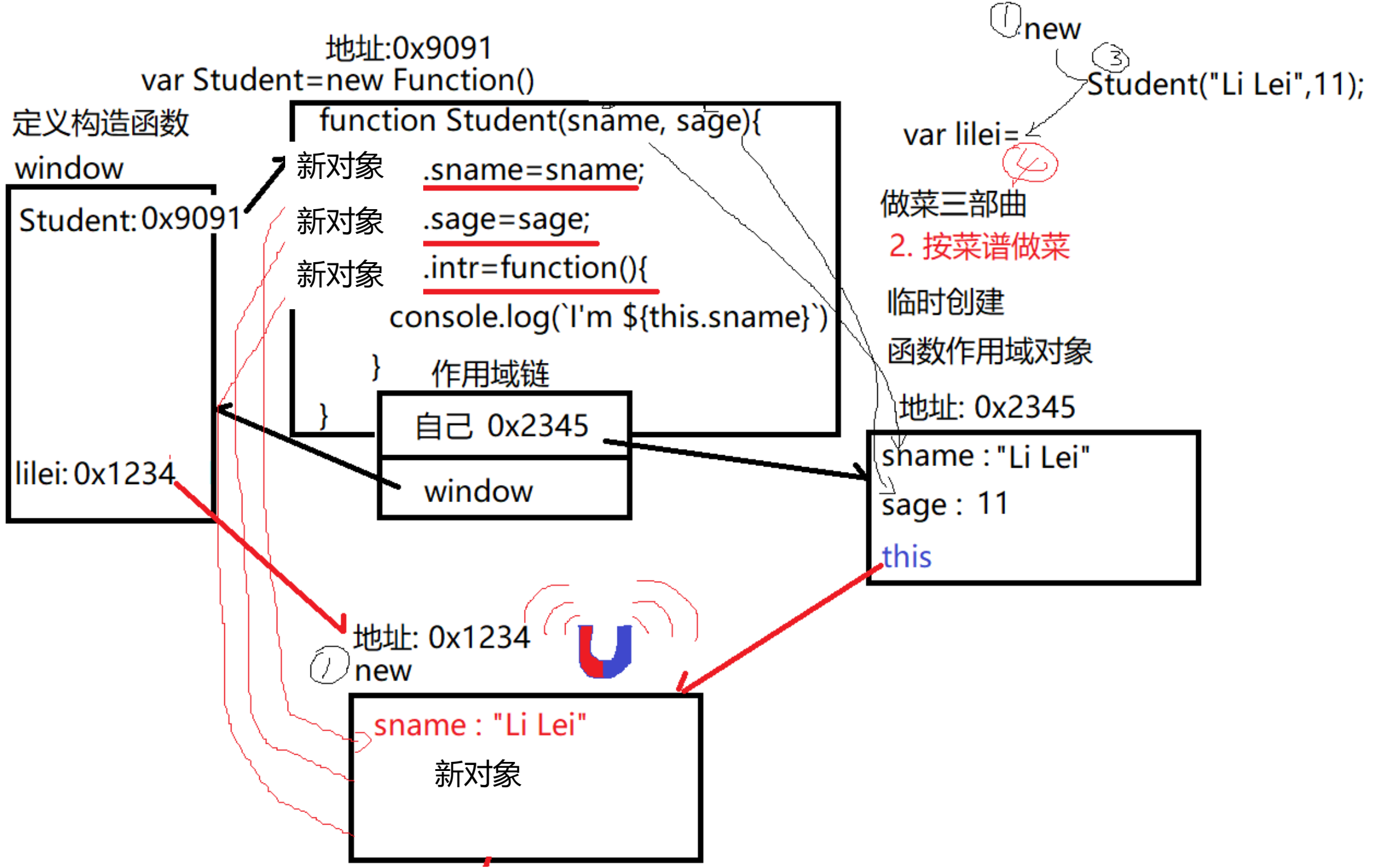


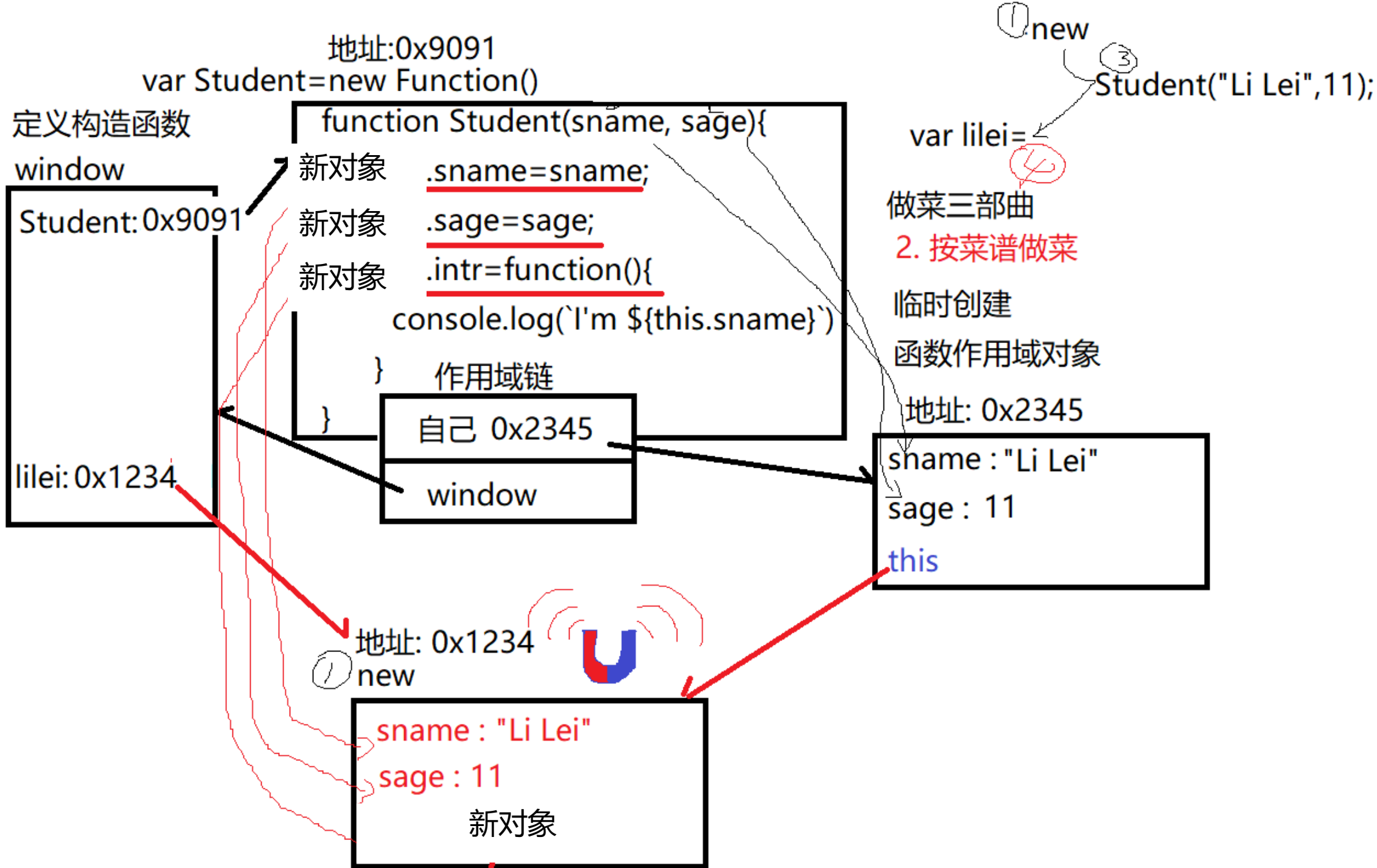


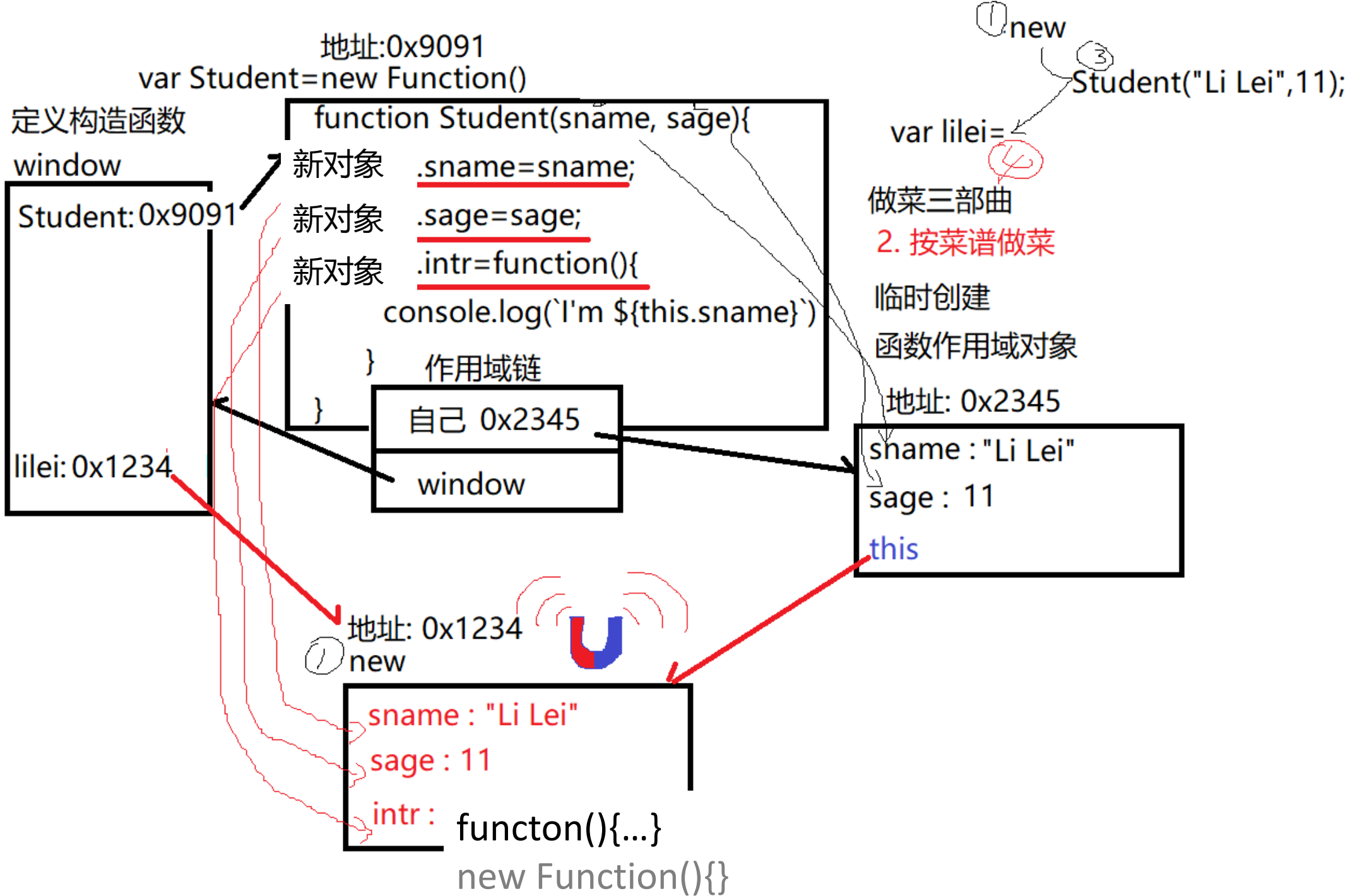


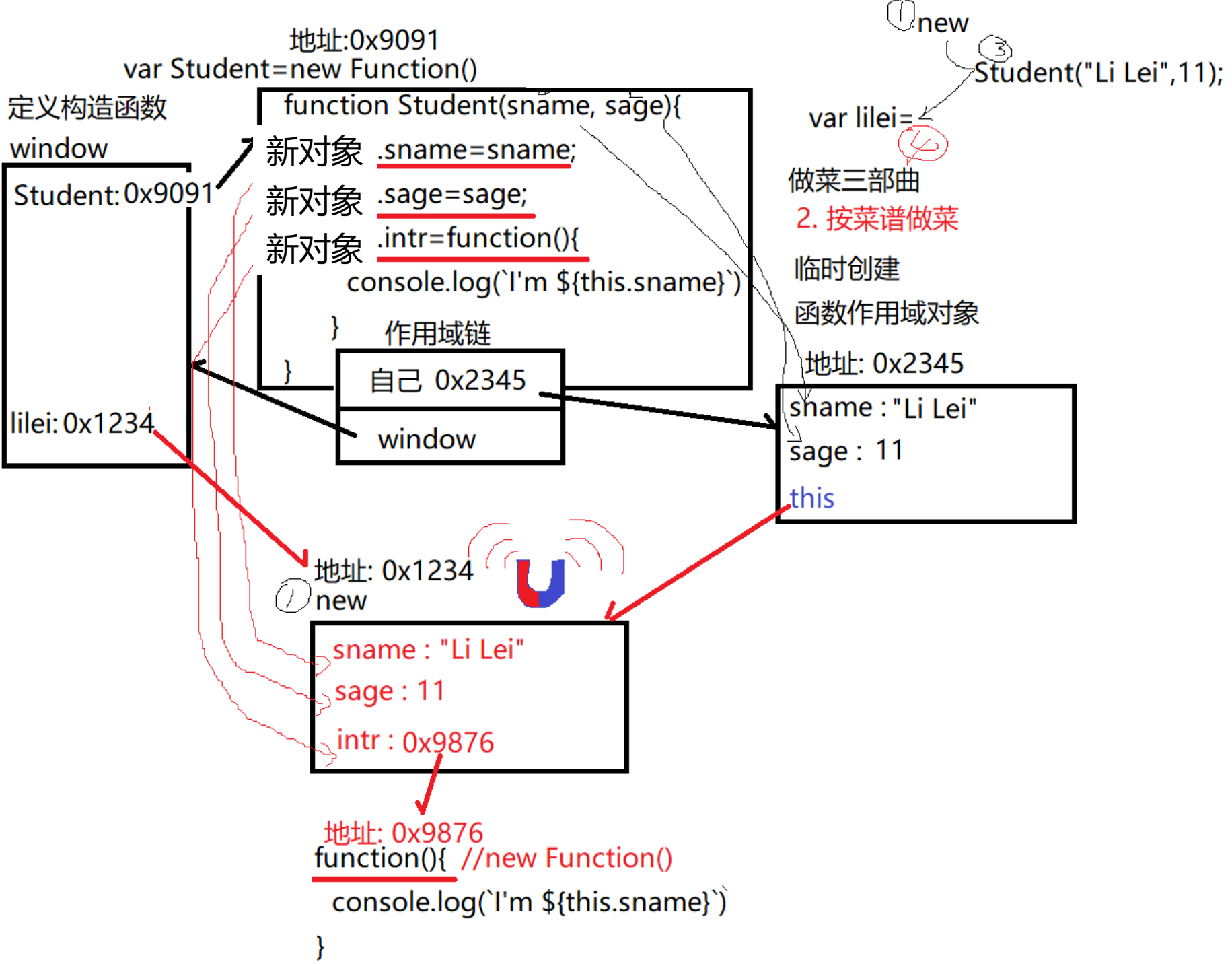








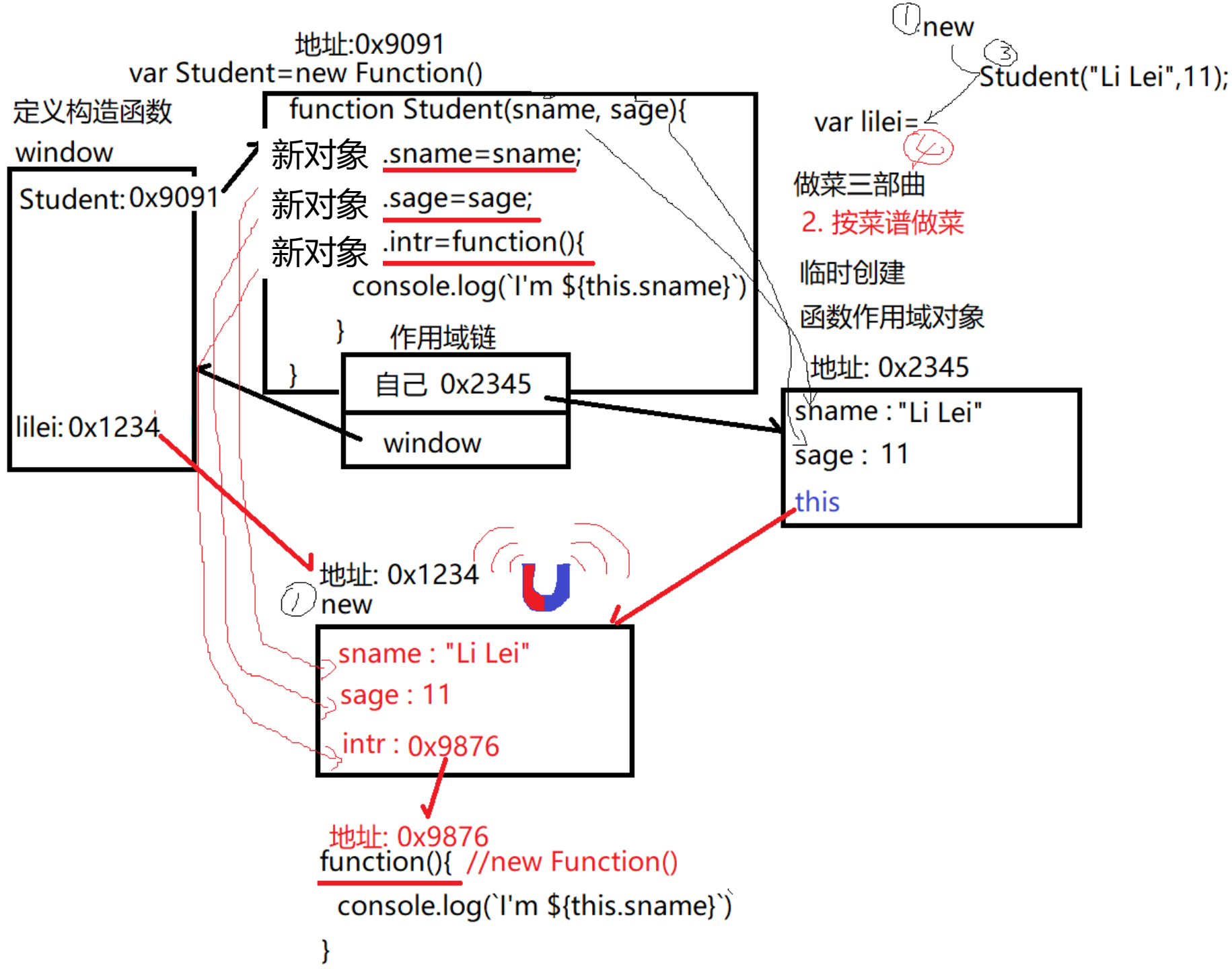






# new做了4件事:

- i. 创建一个新的空对象等待
- ii. ?
- iii. 调用构造函数:
  - 将构造函数中的this->new刚创建的新对象
  - 在构造函数内通过强行赋值方式, 为新对象添加规定的属性和方法
- iv. 返回新对象的地址, 保存到=左边的变量里。



地址:0x9091

var Student=new Function()

定义构造函数

window

Student:0x9091

lilei:0x1234

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
  this.intr=function(){  
    console.log(`I'm ${this.sname}`)  
  }  
}
```

作用域链

自己 空

window

做菜三部曲

3. 收拾厨房

地址: 0x1234

sname : "Li Lei"

sage : 11

intr : 0x9876

地址: 0x9876

```
function(){ //new Function()  
  console.log(`I'm ${this.sname}`)  
}
```

封装

继承

多态



封装

继承

多态

**new做的  
第二件事儿?  
是啥?  
(后续讲)**

# 总结: this 2种情况

- obj.fun() fun中的this->.前的obj对象
- new Fun() Fun中的this->new创建的新对象

封装

继承

多态



封装

继承

多态

# 继承





# 问题:

- 只要将方法定义放在构造函数中,
- 那么, 每次new时都会执行function,
- 就会反复创建相同函数的多个副本!
- ——浪费内存



妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
  this.intr= function(){  
    自我介绍  
  }  
}
```

封装

继承

多态

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
  this.intr= function(){  
    自我介绍  
  }  
}
```

孩子

```
var lilei=  
  new Student("Li Lei",11);
```

```
sname : "Li Lei",  
sage : 11
```

封装

继承

多态

妈妈

```
function Student(sname, sage){  
    this.sname=sname;  
    this.sage=sage;  
    this.intr= function(){  
        自我介绍  
    }  
}
```

孩子

```
var lilei=  
    new Student("Li Lei",11);
```

```
sname : "Li Lei",  
sage : 11
```

intr: 0x1234

地址: 0x1234

function(){ 自我介绍 }

封装

继承

多态

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
  this.intr= function(){  
    自我介绍  
  }  
}
```

孩子

```
var lilei=  
  new Student("Li Lei",11);
```

```
sname : "Li Lei",  
sage : 11
```

intr: 0x1234

地址: 0x1234

function(){ 自我介绍 }

封装

继承

多态

```
var hmm=  
  new Student("Han Meimei",12)
```

```
sname : "Han Meimei",  
sage : 12
```

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
  this.intr= function(){  
    自我介绍  
  }  
}
```

孩子

```
var lilei=  
  new Student("Li Lei",11);
```

```
sname : "Li Lei",  
sage : 11
```

intr: 0x1234

地址: 0x1234  
function(){ 自我介绍 }

封装

继承

多态

```
var hmm=  
  new Student("Han Meimei",12)
```

```
sname : "Han Meimei",  
sage : 12
```

intr: 0x2345

地址: 0x2345  
function(){ 自我介绍 }



封装

继承

多态

# 解决:

- 如果将来发现多个子对象都要使用相同的功能和属性值时,
- 都可以用继承来解决

封装

继承

多态

# 什么是继承:

- 父对象中的成员,
- 子对象无需重复创建, 就可直接使用!
- 就像使用自己的成员一样!
- `this.属性名/方法名()`

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
  this.intr=function(){  
    自我介绍  
  }  
}
```



孩子

```
var lilei=  
  new Student("Li Lei",11);
```

```
sname : "Li Lei",  
sage : 11
```

intr: 0x1234

地址: 0x1234  
function(){ 自我介绍 }

封装

继承

多态

```
var hmm=  
  new Student("Han Meimei",12)
```

```
sname : "Han Meimei",  
sage : 12
```

intr: 0x2345

地址: 0x2345  
function(){ 自我介绍 }

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
  this.intr=function(){  
    自我介绍  
  }  
}
```

孩子

```
var lilei=  
  new Student("Li Lei",11);
```

```
sname : "Li Lei",  
sage : 11
```

intr: 0x1234

地址: 0x1234  
function(){ 自我介绍 }



爸爸

封装

继承

多态

```
intr=function(){ ... }  
  new Function()
```

孩子

```
var hmm=  
  new Student("Han Meimei",12)
```

```
sname : "Han Meimei",  
sage : 12
```

intr: 0x2345

地址: 0x2345  
function(){ 自我介绍 }



妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

爸爸

封装

继承

多态

```
intr=function(){ ... }  
new Function()
```



孩子

```
var lilei=  
new Student("Li Lei",11);
```

孩子

```
var hmm=  
new Student("Han Meimei",12)
```

```
sname : "Li Lei",  
sage : 11
```

lilei.intr()

```
sname : "Han Meimei",  
sage : 12
```

hmm.intr()

封装

继承

多态

# js中继承都是通过原型对象实现



封装

继承

多态

## 什么是 原型对象：

- 替所有子对象集中保存共有属性值和方法的父对象.

## 何时使用 原型对象

- 今后，只要发现多个子对象都需要使用相同的功能和属性值时，
- 都可将相同的功能和属性值集中定义在原型对象中。

# 如何创建 原型对象

- 不用自己创建。
- 而是在定义构造函数时，
- 程序自动附赠我们一个空的原型对象

妈妈

爸爸

封装

继承

多态

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```



```
intr=function(){ ... }  
new Function()
```

孩子  
var lilei=  
new Student("Li Lei",11);

孩子  
var hmm=  
new Student("Han Meimei",12)

```
sname : "Li Lei",  
sage : 11
```

lilei.intr()

```
sname : "Han Meimei",  
sage : 12
```

hmm.intr()

妈妈

赠

自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



孩子  
var lilei=

new Student("Li Lei",11);

sname : "Li Lei",  
sage : 11

孩子  
var hmm=

new Student("Han Meimei",12)

sname : "Han Meimei",  
sage : 12

# 如何找到原型对象

- 构造函数中都有一个自带的属性prototype, 指向自己配对的原型对象。
- 构造函数.prototype

妈妈

赠

自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



孩子  
var lilei=  
new Student("Li Lei",11);

sname : "Li Lei",  
sage : 11

孩子  
var hmm=  
new Student("Han Meimei",12)

sname : "Han Meimei",  
sage : 12



# 何时 如何 继承

- 不用自己设置继承关系！
- **new的第二步**自动让新创建的子对象，继承构造函数的原型对象。
- **new的第二步**自动设置子对象的`__proto__`属性，指向构造函数的原型对象。

妈妈

赠

自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



孩子  
var lilei=

new Student("Li Lei",11);

sname : "Li Lei",  
sage : 11

孩子  
var hmm=

new Student("Han Meimei",12)

sname : "Han Meimei",  
sage : 12

妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



孩子  
var lilei=

new Student("Li Lei",11);

2. 让新子对象继承构造函数  
的原型对象

sname : "Li Lei",  
sage : 11

继承

\_\_proto\_\_

孩子  
var hmm=

new Student("Han Meimei",12)

2. 让新子对象继承构造函数的原型对象

继承

\_\_proto\_\_

sname : "Han Meimei",  
sage : 12

# 总结:new 4步

- 1. 创建一个新的空对象等待
- 2. 让子对象继承构造函数的原型对象
- 3. 调用构造函数，将this替换为新对象，通过强行赋值方式为新对象添加规定的属性
- 4. 返回新对象地址

# 如何向原型对象中添加共有属性

- 只能强行赋值:
- 构造函数.prototype.属性名=属性值
- 构造函数.prototype.方法名=function(){... ...}

妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



孩子  
var lilei=

```
new Student("Li Lei",11);  
2. 让新子对象继承构造函数  
的原型对象
```

```
sname : "Li Lei",  
sage : 11
```

--proto--

继承

孩子  
var hmm=

```
new Student("Han Meimei",12)  
2. 让新子对象继承构造函数的原型对象
```

```
sname : "Han Meimei",  
sage : 12
```

--proto--

继承

妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype

Student.prototype.intr=function(){ ... }



孩子  
var lilei=

new Student("Li Lei",11);  
2. 让新子对象继承构造函数

sname : "Li Lei",  
sage : 11

继承

--proto--

孩子  
var hmm=

new Student("Han Meimei",12)  
2. 让新子对象继承构造函数的原型对象

继承

--proto--

sname : "Han Meimei",  
sage : 12



妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



Student.prototype.intr=function(){ ... }

intr=function(){ ... }  
new Function()

孩子  
var lilei=

new Student("Li Lei",11);

2. 让新子对象继承构造函数

sname : "Li Lei",  
sage : 11

继承

--\_proto\_--

孩子

var hmm=

new Student("Han Meimei",12)

2. 让新子对象继承构造函数的原型对象

继承

--\_proto\_--

sname : "Han Meimei",  
sage : 12

妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



Student.prototype.intr=function(){ ... }

intr=function(){ ... }  
new Function()

孩子  
var lilei=

new Student("Li Lei",11);

2. 让新子对象继承构造函数

sname : "Li Lei",  
sage : 11

lilei.intr()

继承

\_\_proto\_\_

孩子

var hmm=

new Student("Han Meimei",12)

2. 让新子对象继承构造函数的原型对象

继承

\_\_proto\_\_

sname : "Han Meimei",  
sage : 12

hmm.intr()

# 结果:

- 今后，用子对象.访问任何成员时，
- js引擎先在子对象内部查找自有的属性。
- 如果子对象没有，则js引擎会自动延\_\_proto\_\_属性去父对象查找。
- 如果在父对象中找到了想要的属性或方法，则和访问子对象的方法一样调用。

# 所以

---

- 今后，构造函数中一定不要包含方法的定义。
- ——浪费内存
- 所有公用的方法都应该集中定义到原型对象中一份。
- 所有子对象共用。

封装

继承

多态

## 为什么父对象叫原型对象



封装

继承

多态

原型

prototype 嘘！





封装

继承

多态

原型  
prototype 噓！





封装

继承

多态

原型  
prototype 噓！



封装

继承

多态

原型  
prototype 噓！



**this: 第三种情况:  
原型对象中的this->?**

---

# 错误说法：

- ~~this指当前函数所在的对象~~ 看哪里，以何种方式调用！！！！
- 比如：
  - 原型对象中的共有方法，虽然保存在原型对象中，
  - 但是却被子对象调用的。
  - 所以，this不指原型对象，指将来的某个子对象。

妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



Student.prototype.intr=function(){ ... }

intr=function(){ ... }  
new Function()

孩子  
var lilei=

new Student("Li Lei",11);

2. 让新子对象继承构造函数

\_\_proto\_\_

sname : "Li Lei",  
sage : 11

lilei.intr()

继承

孩子  
var hmm=

new Student("Han Meimei",12)

2. 让新子对象继承构造函数的原型对象

\_\_proto\_\_

sname : "Han Meimei",  
sage : 12

hmm.intr()

继承

妈妈



自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



`Student.prototype.intr=function(){ ... }`

`intr=function(){ ... } this.sname, this.sage  
new Function()`

孩子

`var lilei=`

`new Student("Li Lei",11);`

2. 让新子对象继承构造函数

`--_proto_--`

`sname : "Li Lei",  
sage : 11`

`lilei.intr()`

继承

孩子

`var hmm=`

`new Student("Han Meimei",12)`

2. 让新子对象继承构造函数的原型对象

`--_proto_--`

`sname : "Han Meimei",  
sage : 12`

`hmm.intr()`

继承



妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



Student.prototype.intr=function(){ ... }

intr=function(){ ... } this.sname, this.sage  
new Function()

一定不要看  
定义在哪儿

孩子

var lilei=

new Student("Li Lei",11);

2. 让新子对象继承构造函数

\_\_proto\_\_

sname : "Li Lei",  
sage : 11

lilei.intr()

继承

孩子

var hmm=

new Student("Han Meimei",12)

2. 让新子对象继承构造函数的原型对象

\_\_proto\_\_

sname : "Han Meimei",  
sage : 12

hmm.intr()

继承



妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



Student.prototype.intr=function(){ ... }

intr=function(){ ... } this.sname, this.sage  
new Function()

一定不要看  
定义在哪儿

孩子

var lilei=

new Student("Li Lei",11);

2. 让新子对象继承构造函数

\_\_proto\_\_

sname : "Li Lei",  
sage : 11

lilei.intr()

只看在哪里  
如何调用

孩子

var hmm=

new Student("Han Meimei",12)

2. 让新子对象继承构造函数的原型对象

继承

\_\_proto\_\_

sname : "Han Meimei",  
sage : 12

hmm.intr()

妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



Student.prototype.intr=function(){ ... }

intr=function(){ ... } this.sname, this.sage  
new Function()

一定不要看  
定义在哪儿

孩子

var lilei=

new Student("Li Lei",11);

2. 让新子对象继承构造函数

\_\_proto\_\_

sname : "Li Lei",  
sage : 11

lilei.intr()

以及调用时  
.前是谁?

孩子

var hmm=

new Student("Han Meimei",12)

2. 让新子对象继承构造函数的原型对象

继承

\_\_proto\_\_

sname : "Han Meimei",  
sage : 12

hmm.intr()

妈妈

赠 自动创建 爸爸 原型对象

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



Student.prototype.intr=function(){ ... }

intr=function(){ ... } this.sname, this.sage  
new Function()

一定不要看  
定义在哪儿

孩子

var lilei=

new Student("Li Lei",11);

2. 让新子对象继承构造函数的原型对象

\_\_proto\_\_

sname : "Li Lei",  
sage : 11

this-> lilei.intr()

以及调用时  
.前是谁?

孩子

var hmm=

new Student("Han Meimei",12)

2. 让新子对象继承构造函数的原型对象

继承

\_\_proto\_\_


sname : "Han Meimei",  
sage : 12

this-> hmm.intr()

## this: 第三种情况：原型对象中的this

- 判断this，只看在哪里如何调用！不要看定义在哪
- 因为原型对象中的共有方法将来使用“子对象.方法名()”调用。
- 原型对象中的this->将来调用这个公共函数的.前的那个子对象。
- 简单记: 谁调用指谁

# 总结: this 3种:不看定义, 只看调用

- 
- obj.fun() this->.前的obj
  - new Fun() this->new创建的新对象
  - 原型对象中共有方法里的this->将来调用这个共有方法的.前的那个子对象
- 总结: 谁调用就指谁

封装

继承

多态

内置类型的原型对象:



# a. 什么是内置类型:

---

ES标准中规定的，浏览器已经实现，我们可以直接使用的类型。



## b. 包括: 11种:

---

- String, Number, Boolean
- Array, Date, RegExp,
- Math(不是类型, 已经是一个{}对象)
- Error
- Function Object
- global(全局作用域对象, 在浏览器中被window代替)

## c. 什么是类型:

凡事能new的，都叫做类型

- 每种类型一定有2部分组成:
- 1). 构造函数: 负责创建该类型的子对象
- 2). 原型对象: 负责为该类型所有子对象集中保存共有的属性值和方法定义。

## d. 11种内置类型中

---

- 有九种类型也都由构造函数和原型对象组成。
- 也都可以new创建子对象。

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype



自动创建 爸爸 原型对象

Student.prototype.intr=function(){ ... }

intr=function(){ ... }

new Function()

className="初一2班"

Student.prototype.className="初二2班"

孩子

var lilei=

new Student("Li Lei",11);

2. 让新子对象继承构造函数  
的原型对象

sname : "Li Lei",  
sage : 11

lilei.intr()

lilei.className

继承

--\_proto\_--

孩子

var hmm=

new Student("Han Meimei",12)

2. 让新子对象继承构造函数的原型对象

sname : "Han Meimei",  
sage : 12

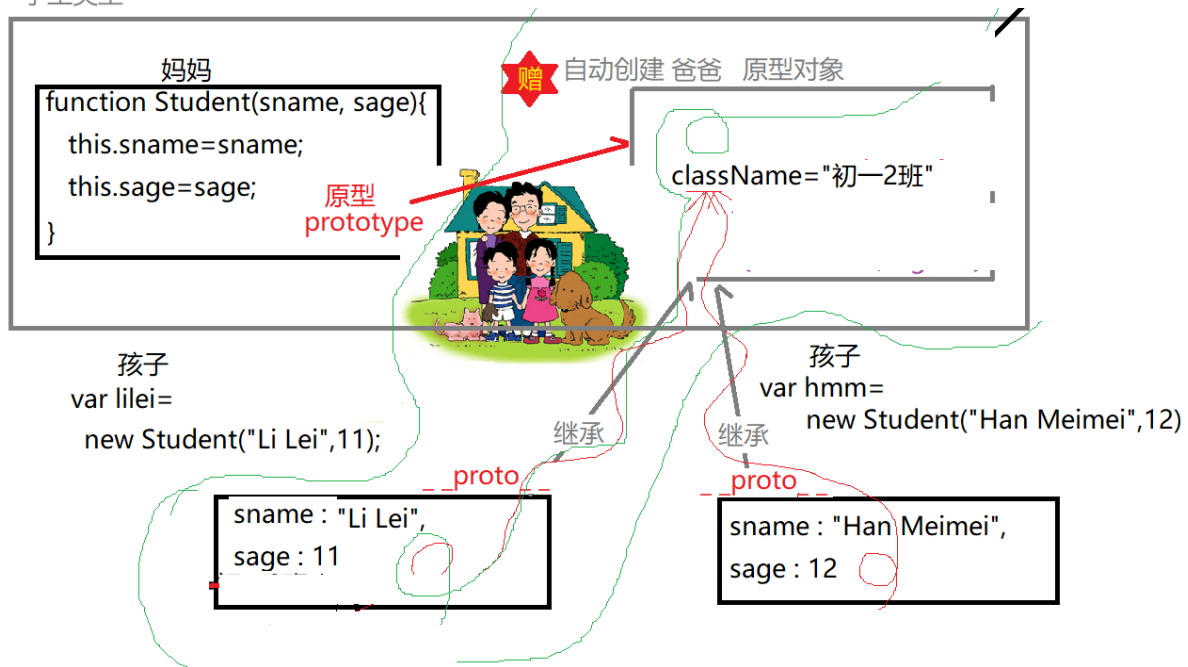
hmm.intr()

hmm.className

继承

--\_proto\_--

学生类型



学生类型

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype

自动创建 爸爸 原型对象

className="初一2班"



继承

继承

孩子

```
var lilei=  
new Student("Li Lei",11);
```

孩子

```
var hmm=  
new Student("Han Meimei",12)
```

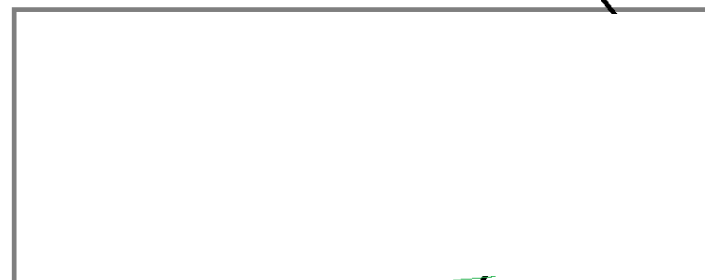
\_\_proto\_\_

\_\_proto\_\_

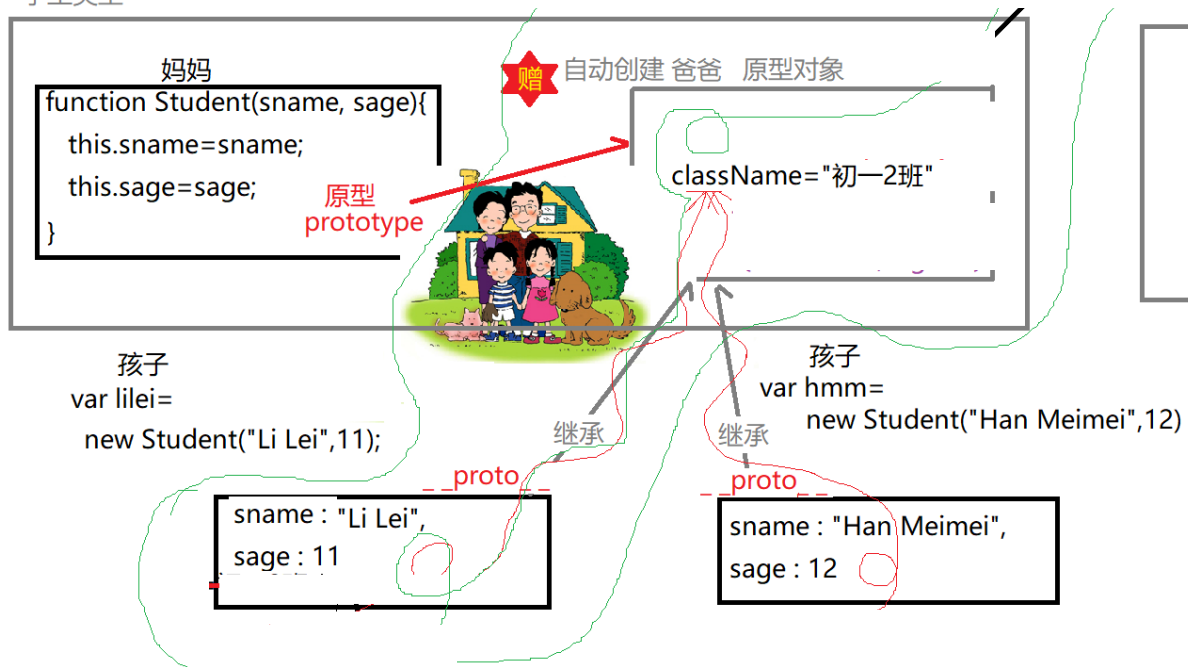
```
sname : "Li Lei",  
sage : 11
```

```
sname : "Han Meimei",  
sage : 12
```

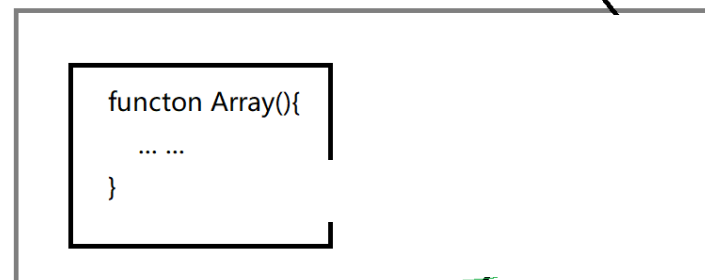
Array类型(家)



学生类型

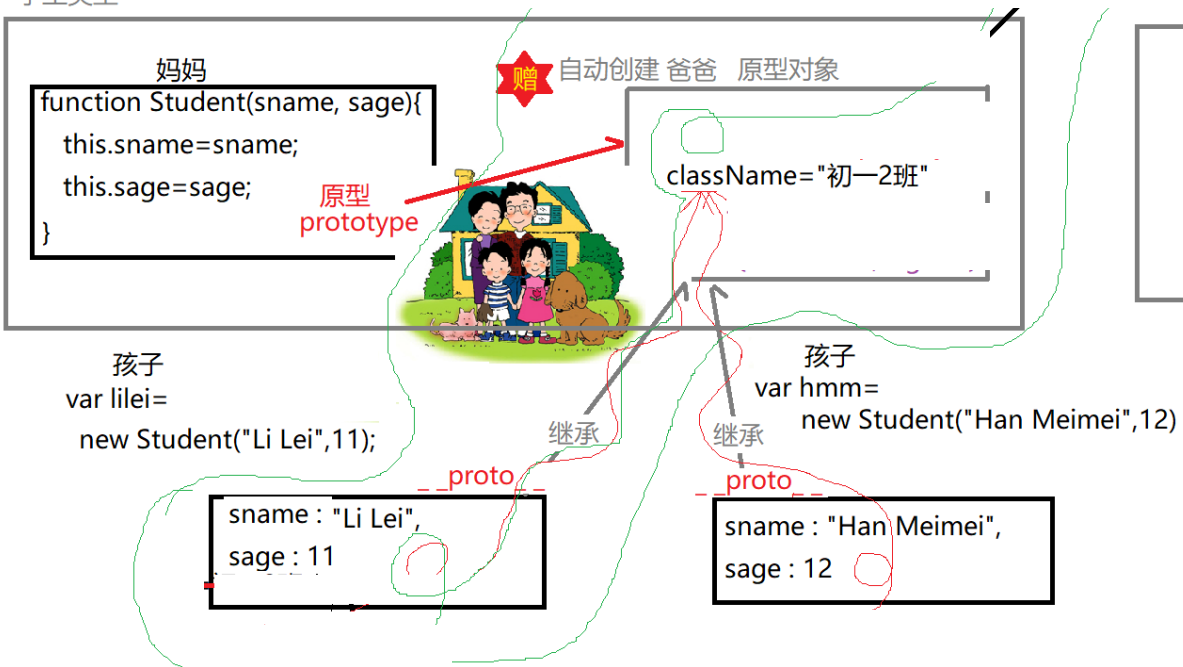


Array类型(家)

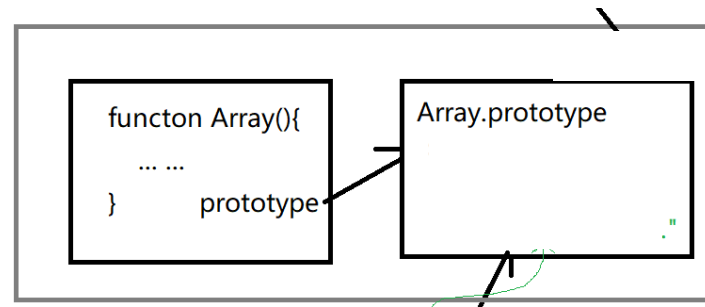




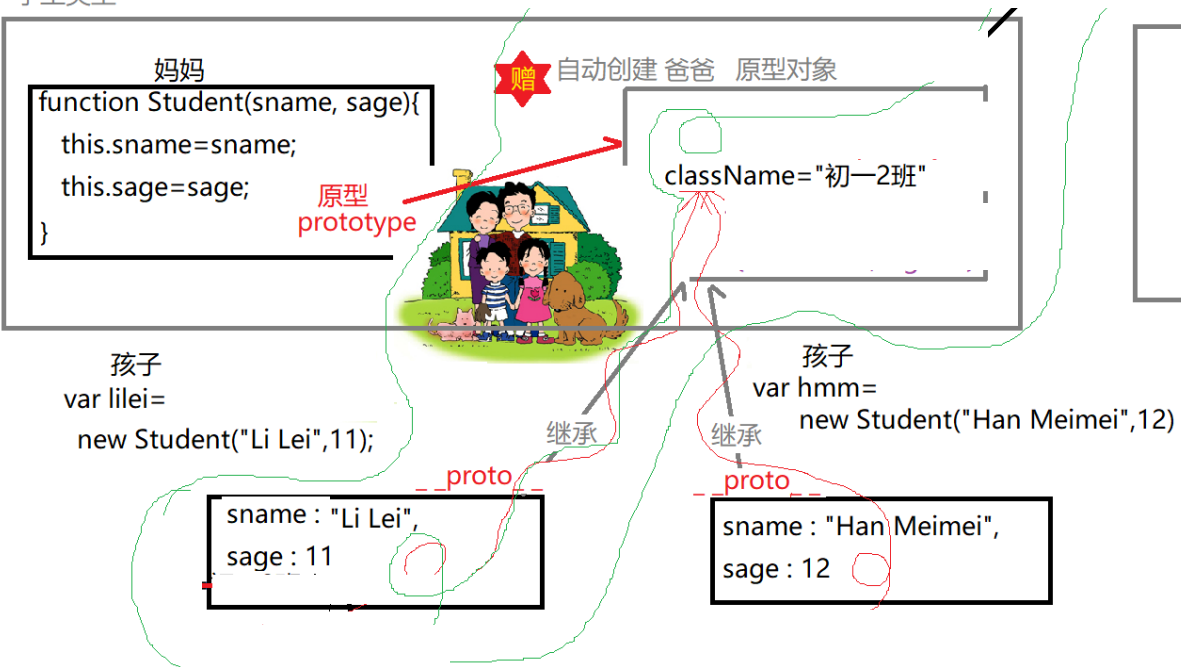
学生类型



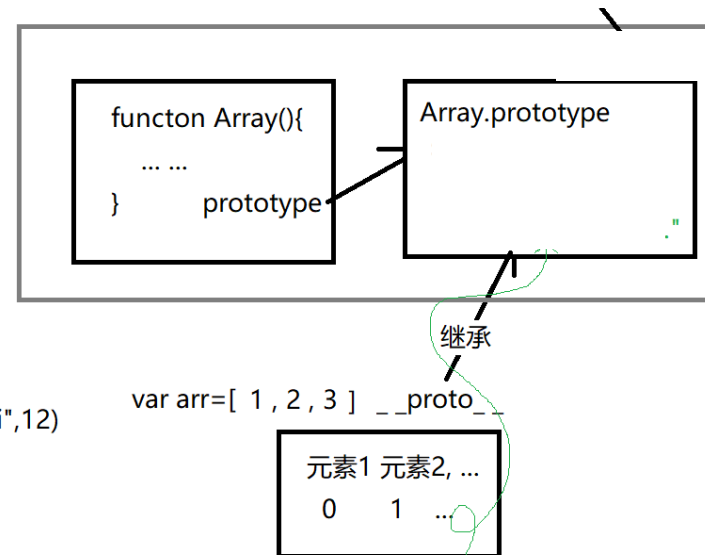
Array类型(家)



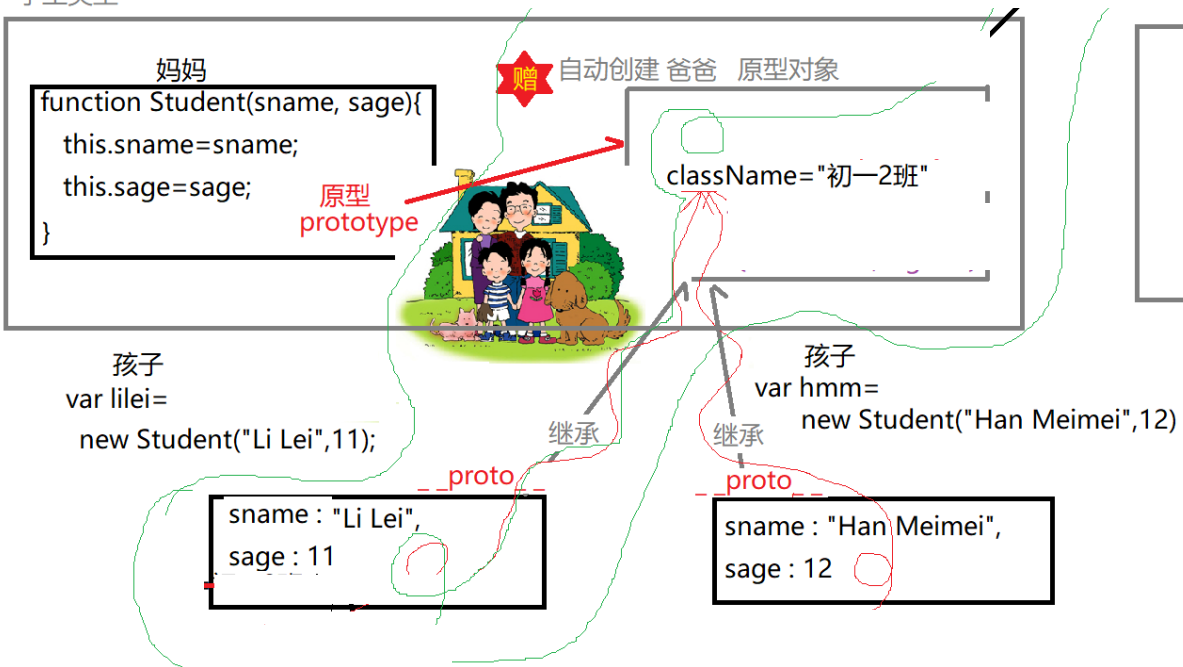
学生类型



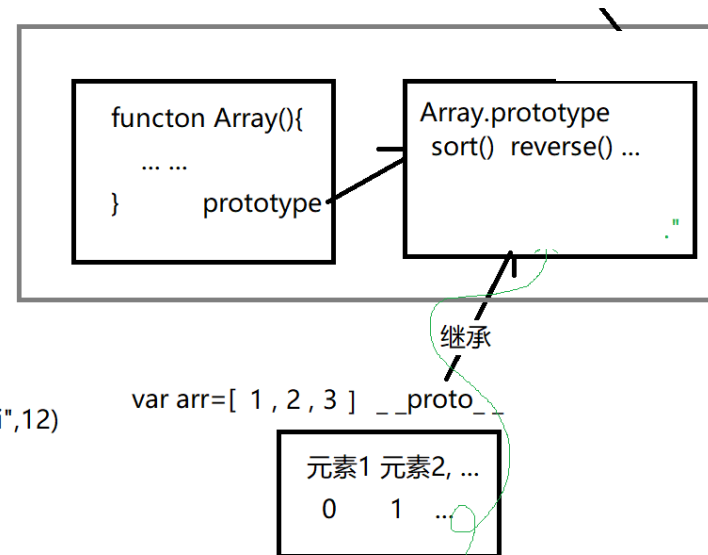
Array类型(家)



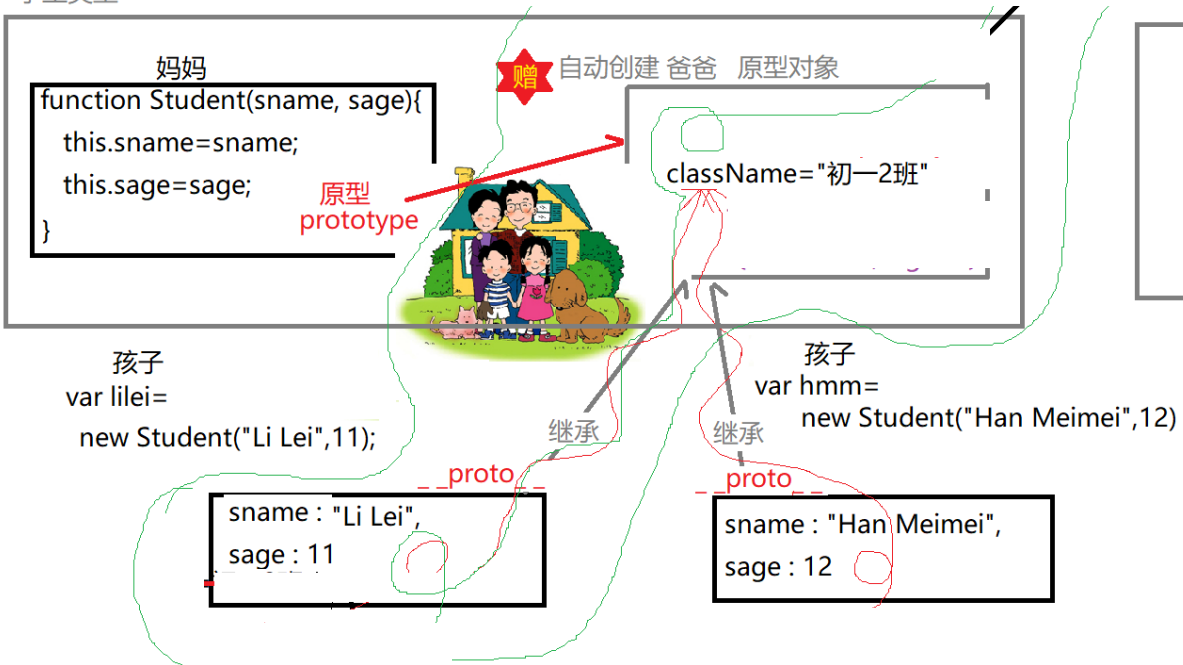
学生类型



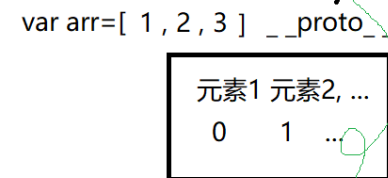
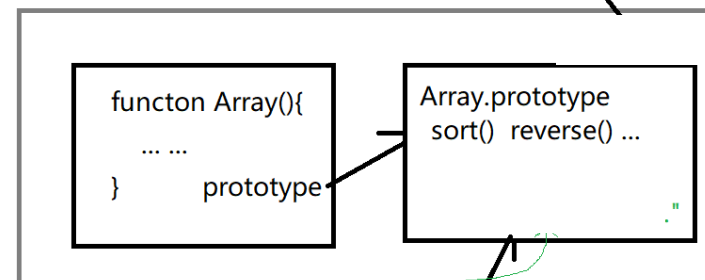
Array类型(家)



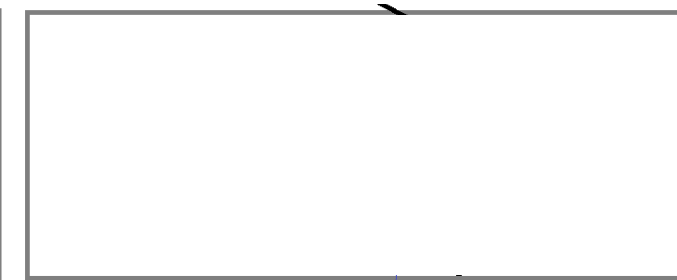
学生类型



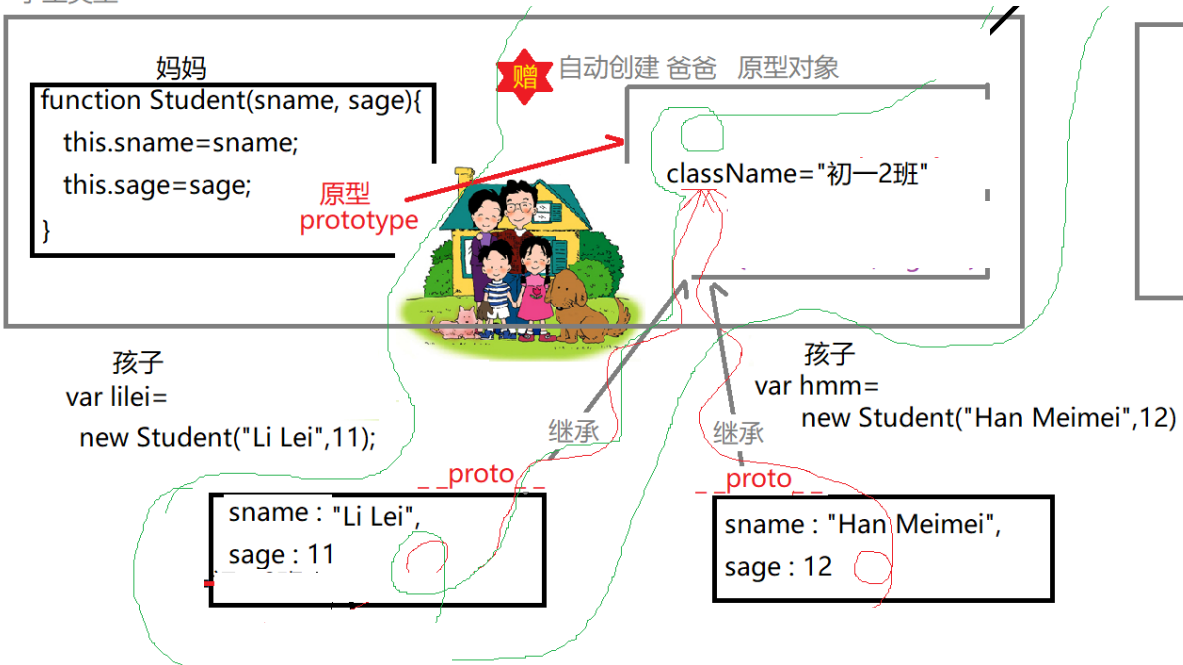
Array类型(家)



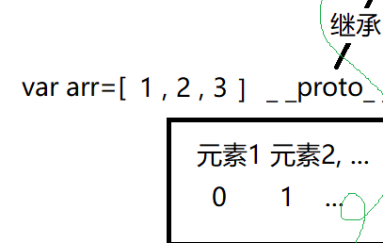
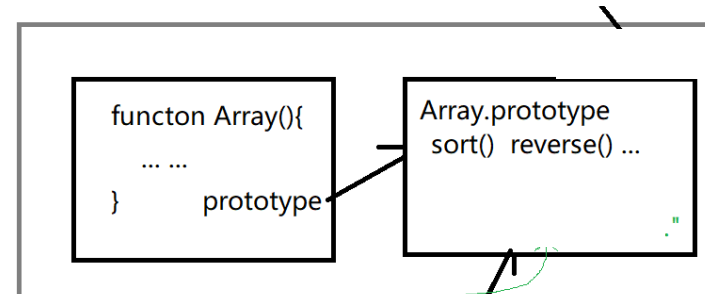
Date类型(家)



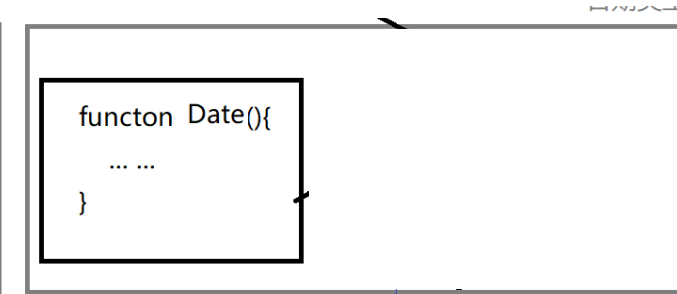
学生类型



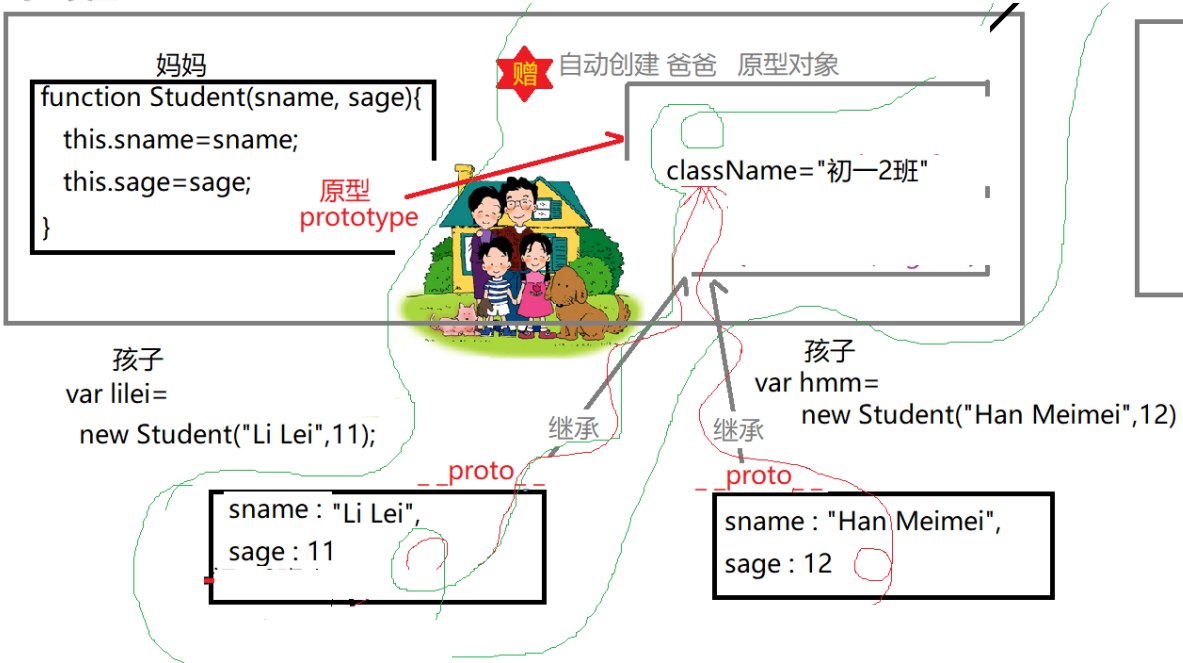
Array类型(家)



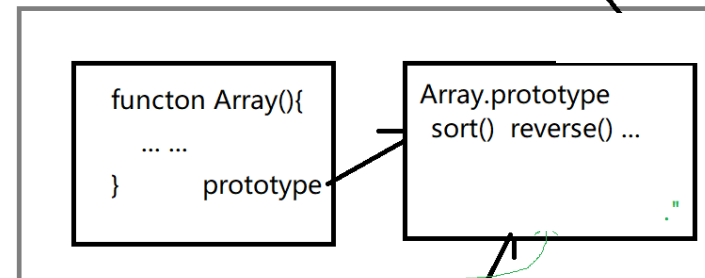
Date类型(家)



学生类型



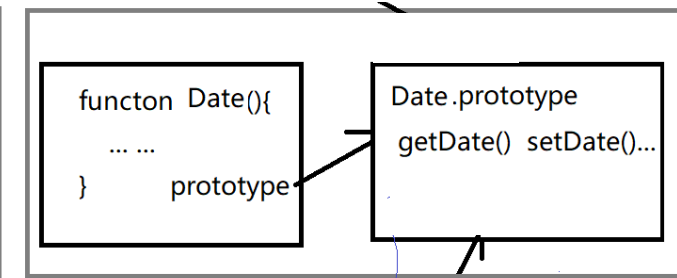
Array类型(家)



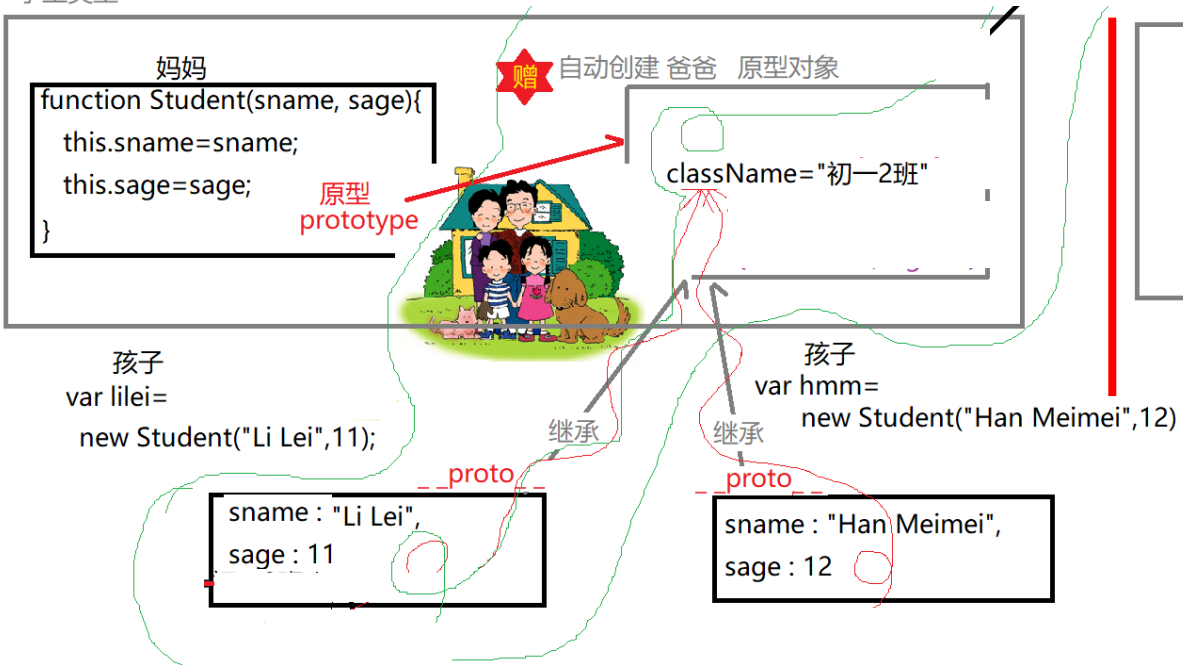
var arr=[ 1 , 2 , 3 ] \_\_proto\_\_

元素1 元素2, ...  
0 1 ...

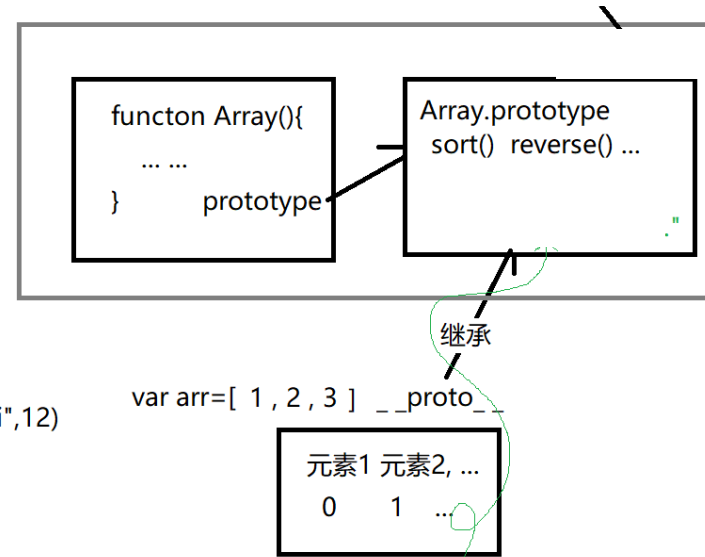
Date类型(家)



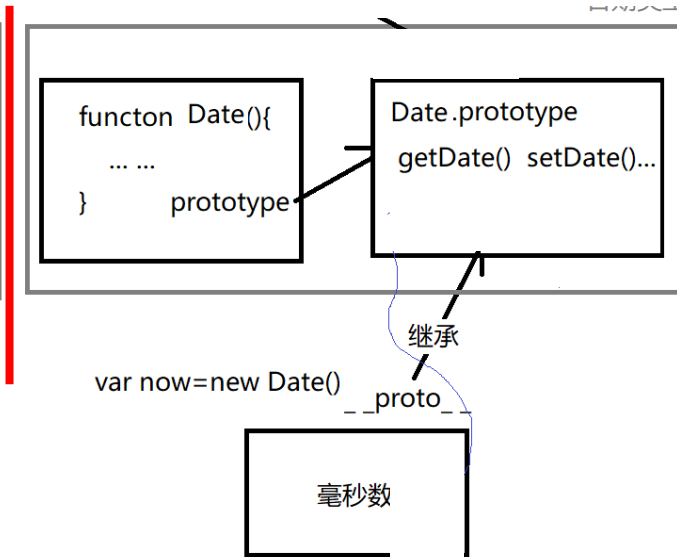
学生类型



Array类型(家)



Date类型(家)





# 内置类型的原型对象

---

- 今后，只要想知道新标准的ES中新增了哪些函数，都可看这个类型的原型对象。
- 比如: `Array.prototype`, `String.prototype`, `Date.prototype`, ... ..

封装

继承

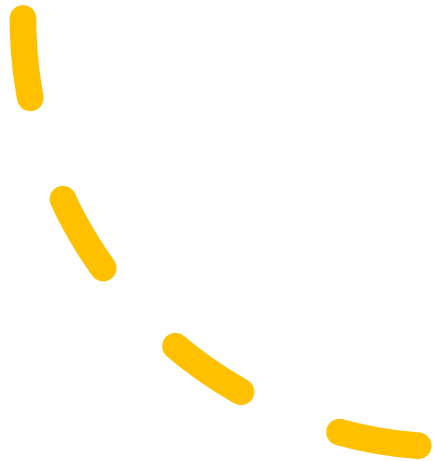
多态

# 问题:

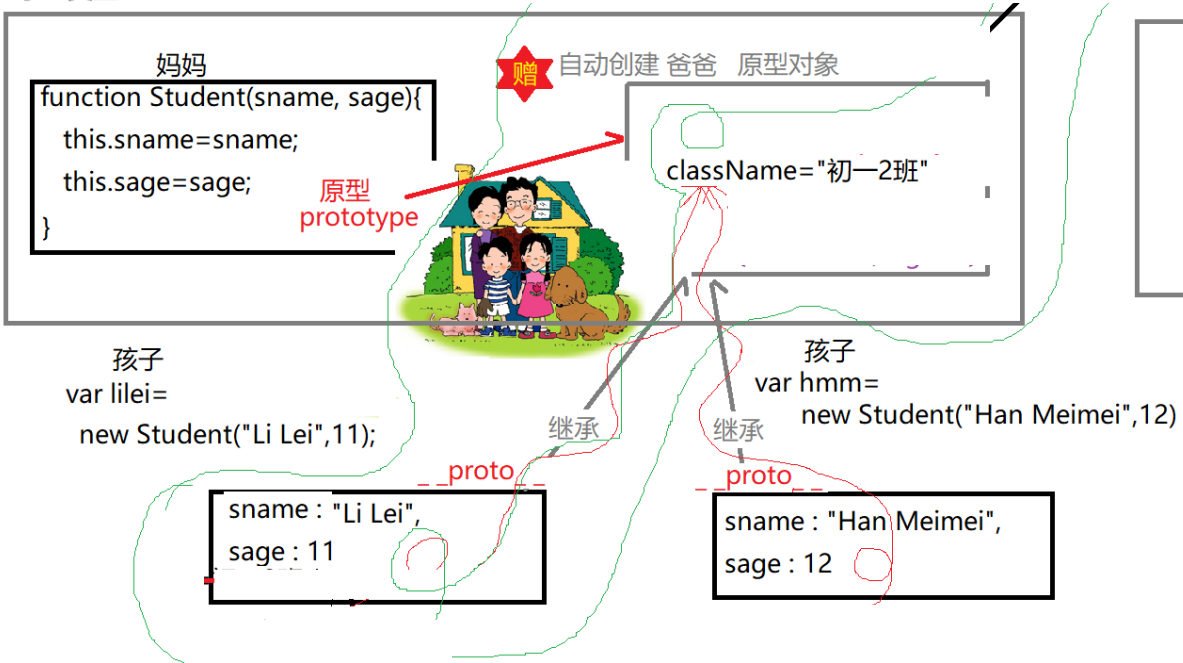
如果经常使用的一个功能，但是原型  
对象中没有提供！

# 解决:自定义一个函数

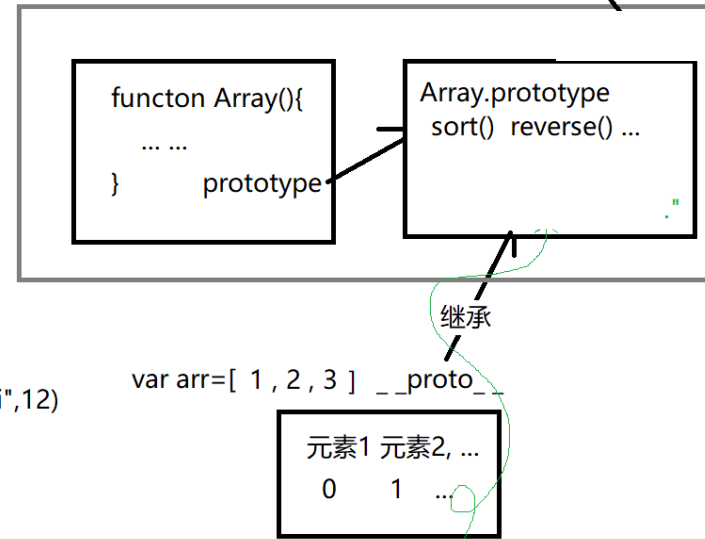
- 保存到原型对象中:  
构造函数.prototype.新方法=function(){...}
- 结果: 所有该类型的子对象, 都可通过:  
子对象.新方法() ——来调用自定义共有方法



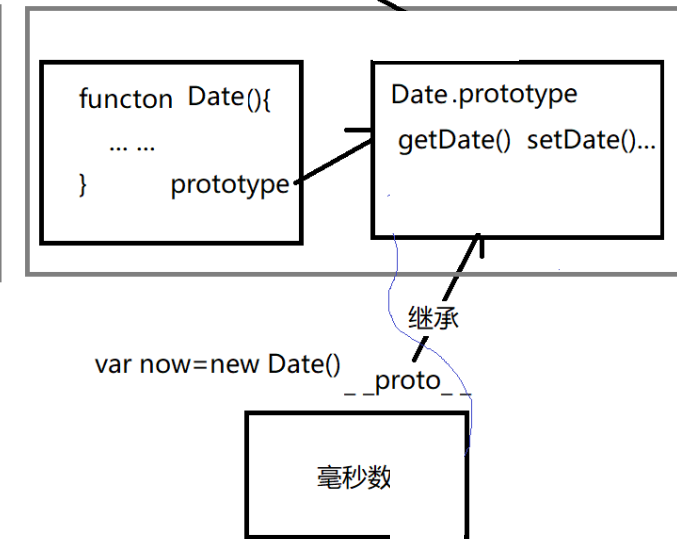
学生类型



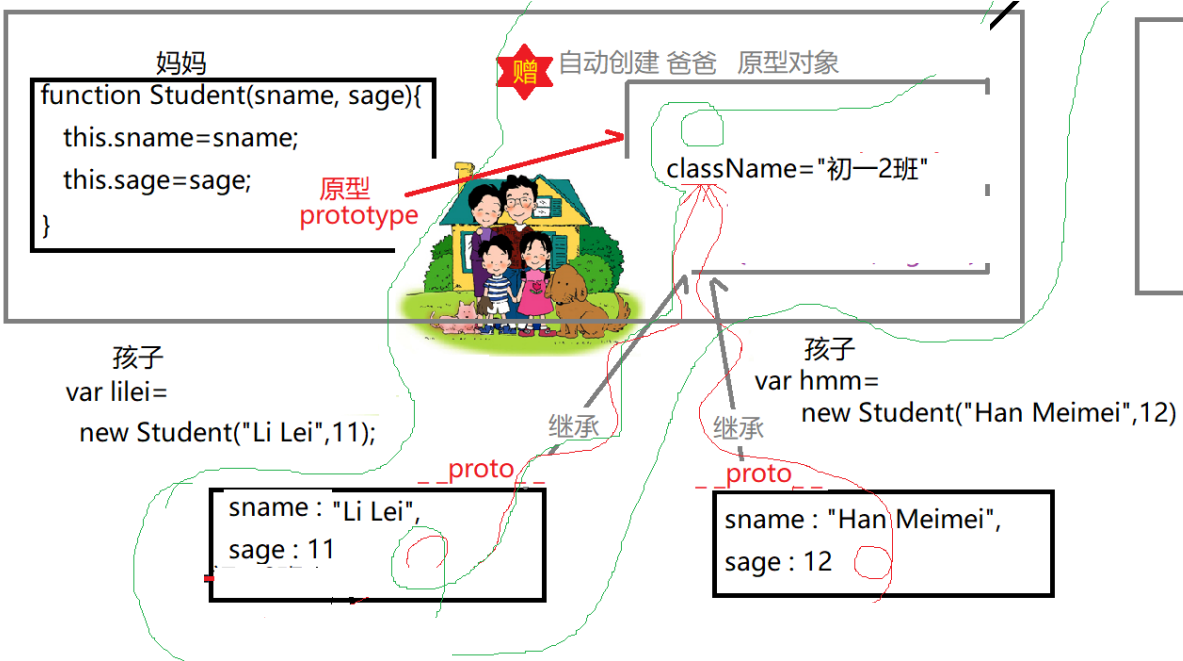
Array类型(家)



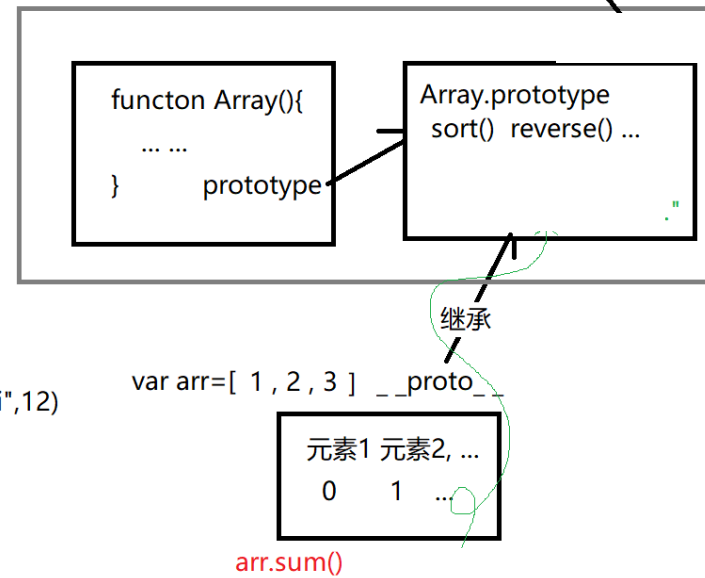
Date类型(家)



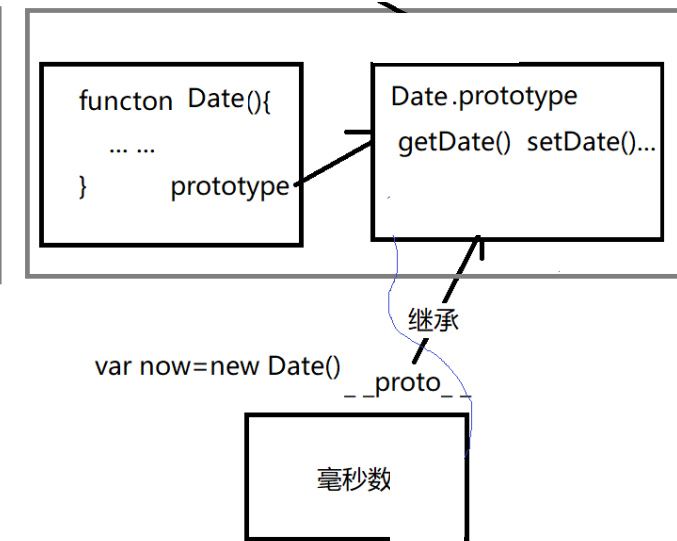
学生类型



Array类型(家)

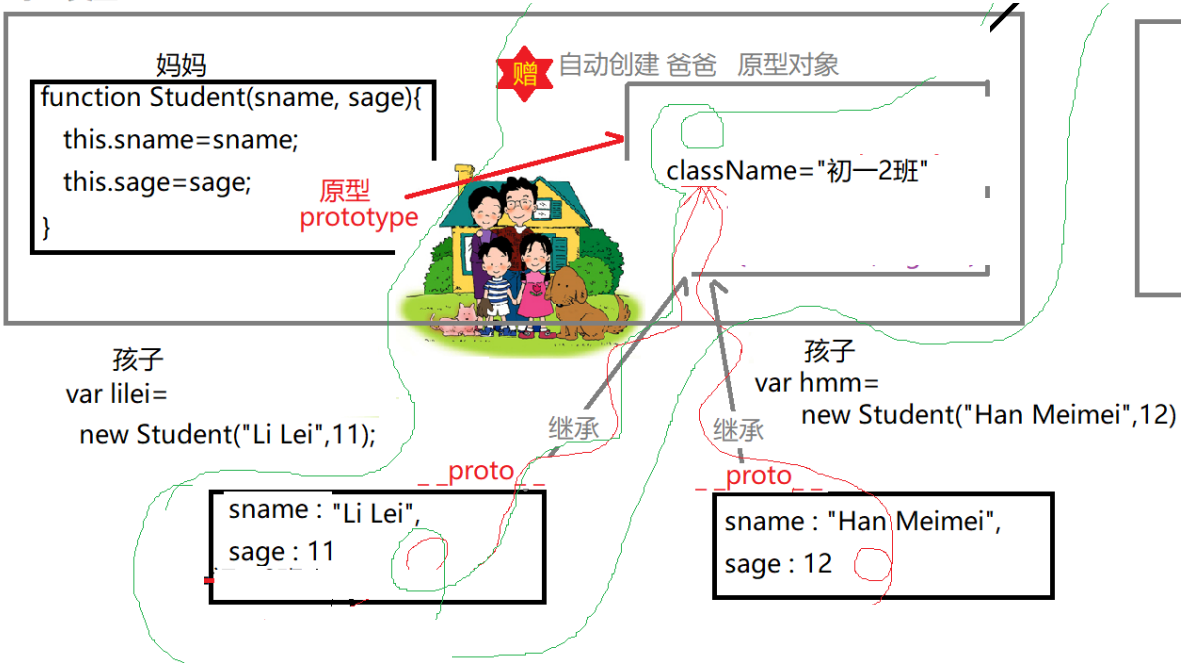


Date类型(家)

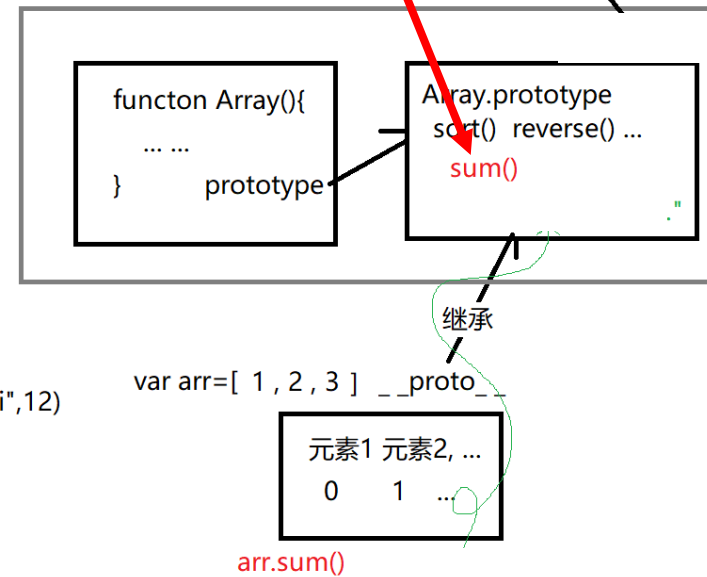


Array.prototype.sum=function(){ ... }

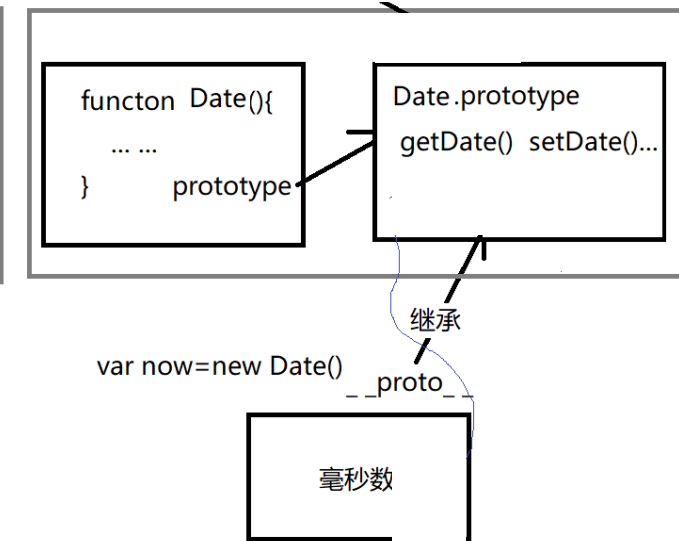
学生类型



Array类型(家)



Date类型(家)



封装

继承

多态

## (8). 原型链:





封装

继承

多态

# 什么是原型链：

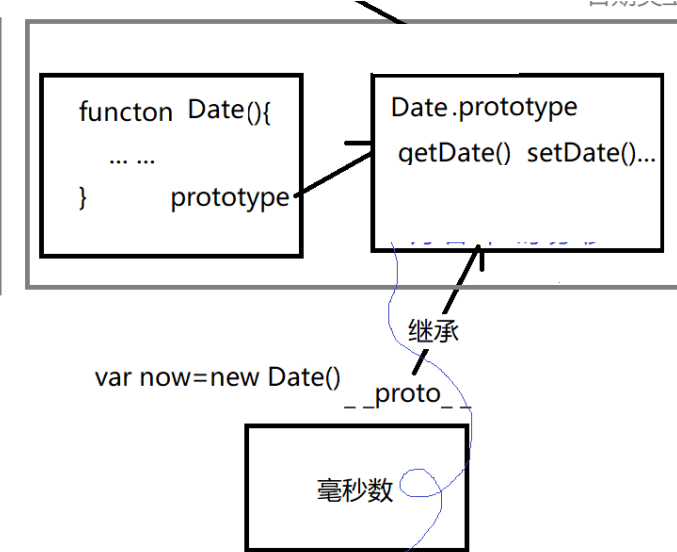
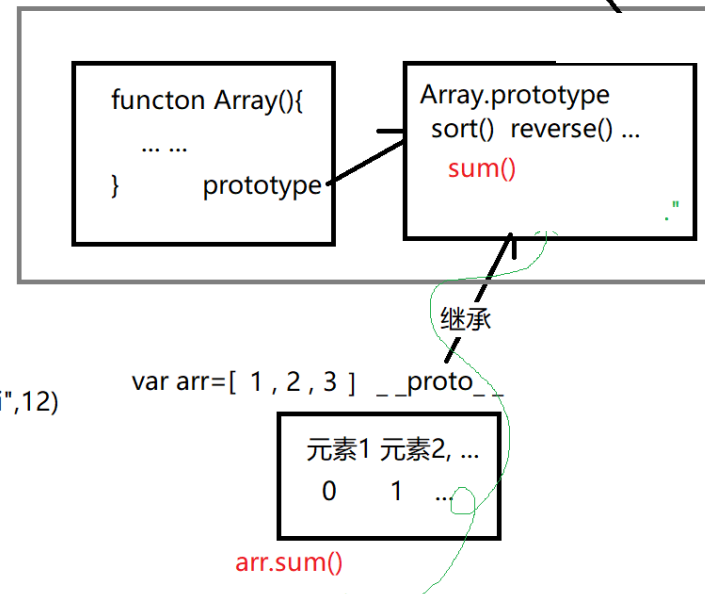
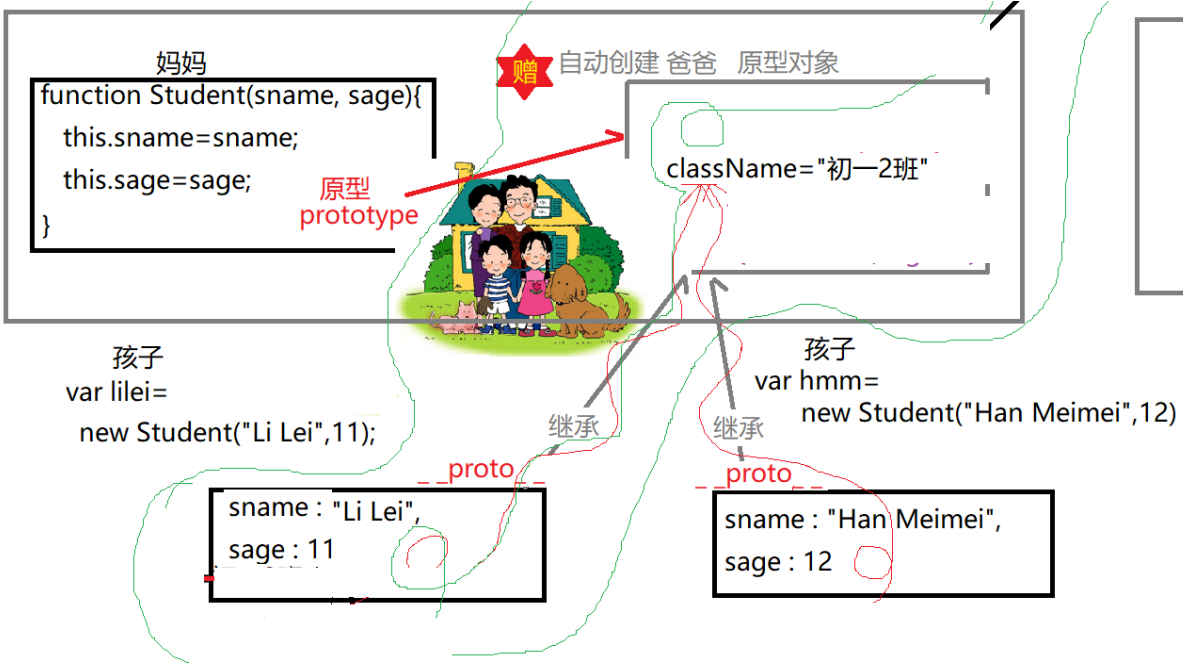
由多级父对象逐级继承形成的链式结构

[[ prototype ]] 这里的双中括号叫内部属性  
真正使用的话：

还是：对象.\_\_proto\_\_的方式调用属性或者方法

```
> function Student(){  
  }  
  Student.className="初一2班";  
  var s=new Student();  
< "初一2班"  
> console.log(s)  
Student {}  
  [[Prototype]]: Object  
    fun: f ()  
    constructor: f Student()  
    [[Prototype]]: Object  
< undefined  
> s.__proto__.fun=function(){}  
< f (){}  
>
```

学生类型



顶级父类型

```
function Object(){  
  prototype  
}
```

Object的原型对象  
toString() // 将对象转为字符串  
"[object 类型名]"

`__proto__ = null`

Object : 又称顶级父类型 ;

继承

继承

继承

学生类型

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype

自动创建 爸爸 原型对象

`__proto__`

className="初一2班"

孩子

```
var lilei=  
new Student("Li Lei",11);
```

`__proto__`

```
sname : "Li Lei",  
sage : 11
```

继承

孩子

```
var hmm=  
new Student("Han Meimei",12)
```

`__proto__`

```
sname : "Han Meimei",  
sage : 12
```

继承

数组类型

```
function Array(){  
  ... ..  
}
```

prototype

```
Array.prototype  
sort() reverse() ...  
sum()
```

继承

```
var arr=[ 1, 2, 3 ]
```

```
元素1 元素2, ...  
0 1 ...
```

`arr.sum()`

日期类型

```
function Date(){  
  ... ..  
}
```

prototype

```
Date.prototype  
getDate() setDate()...
```

继承

```
var now=new Date()
```

毫秒数

`__proto__`

封装

继承

多态

顶级父类型

```
function Object(){  
  prototype  
}
```

Object的原型对象

```
toString() // 将对象转为字符串  
"[object 类型名]"  
__proto__ = null
```

\_\_proto\_\_ = null

继承

继承

继承

学生类型

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype

赠

自动创建 爸爸 原型对象

\_\_proto\_\_

className="初一2班"

孩子

```
var lilei=  
new Student("Li Lei",11);
```

\_\_proto\_\_

```
sname : "Li Lei",  
sage : 11
```

lilei.toString()

孩子

```
var hmm=  
new Student("Han Meimei",12)
```

\_\_proto\_\_

```
sname : "Han Meimei",  
sage : 12
```

Hmm.toString()

数组类型

```
function Array(){  
  ... ..  
  prototype  
}
```

Array.prototype

```
sort() reverse() ...  
sum()
```

继承

```
var arr=[ 1, 2, 3 ] __proto__
```

```
元素1 元素2, ...  
0 1 ...
```

arr.sum()

arr.toString()

日期类型

```
function Date(){  
  ... ..  
  prototype  
}
```

Date.prototype

```
getDate() setDate()...
```

继承

```
var now=new Date() __proto__
```

毫秒数

now.toString()

## a. 什么是原型链:

- 由多级父对象逐级继承形成的链式结构
  - 保存着: 一个对象可用的所有属性和方法
  - 控制着: 属性和方法的使用顺序:  
就近原则: 先子级后父级

# 但是:

- 所有对象调用toString结果都一样吗?

```
function Student(sname,sage){  
    this.sname=sname;  
    this.sage=sage;  
}
```

```
var lilei=new Student("Li Lei",11);
```

```
console.log(lilei);
```

```
► Student {sname: 'Li Lei', sage: 11}
```

```
var arr=[1,2,3];
```

```
var now=new Date();
```

```
console.log(lilei.toString());
```

```
[object Object]
```

Why? 往下看-多态

```
console.log(arr.toString());
```

```
1,2,3
```

```
console.log(now.toString());
```

```
Mon Jan 24 2022 14:50:45 GMT+0800 (中国标准时间)
```

封装

继承

多态

## 2. 多态:





封装

继承

多态

# 什么是多态:

同一个函数  
在不同情况下表现出不同的状态

## (2). 包括: 2种:

- a. 重载overload: 同一个函数, 输入不同的参数, 执行不同的逻辑
- b. 重写override: (推翻、遮挡)



## b. 重写override: (推翻、遮挡)

- 1). 什么是: 在子对象中定义一个和父对象中的成员同名的自有成员。
- 2). 何时: 从父对象继承来的个别成员不好用时, 就可以在子对象中定义同名成员, 来覆盖父对象中的同名成员。



封装

继承

多态

**所有对象调用  
toString结果都  
竟然不一样？**

封装

继承

多态

# 因为

- 数组家的原型对象爸爸觉得Object爷爷家的toString不好用!
- 于是就自己重写了一个好用的同名的toString()方法




封装

继承

多态

## 结果:

- 所有数组家孩子调用toString(), 调用的都是数组原型对象爸爸定义的好用的toString,
  - 不再用Object爷爷的不好用的toString()
  - Date家也是如此!
- 

封装

继承

多态

顶级父类型

```
function Object(){  
  prototype  
}
```

Object的原型对象

```
toString() // 将对象转为字符串  
"[object 类型名]"  
__proto__ = null
```

继承

继承

继承

学生类型

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype

赠

自动创建 爸爸 原型对象

className="初一2班"

孩子

```
var lilei=  
new Student("Li Lei",11);
```

孩子

```
var hmm=  
new Student("Han Meimei",12)
```

```
sname : "Li Lei",  
sage : 11
```

lilei.toString()

```
sname : "Han Meimei",  
sage : 12
```

Hmm.toString()

数组类型

```
function Array(){  
  ... ..  
}
```

prototype

```
Array.prototype  
sort() reverse() ...  
sum()
```

继承

```
var arr=[ 1, 2, 3 ]
```

```
元素1 元素2, ...  
0 1 ...
```

arr.sum()

arr.toString()

日期类型

```
function Date(){  
  ... ..  
}
```

prototype

```
Date.prototype  
getDate() setDate()...
```

继承

```
var now=new Date()
```

毫秒数

now.toString()



顶级父类型

```
function Object(){  
  prototype  
}
```

Object的原型对象

```
toString() // 将对象转为字符串  
"[object 类型名]"
```

`__proto__ = null`

继承

继承

继承

学生类型

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype

赠

自动创建 爸爸 原型对象

`__proto__`

className="初一2班"

孩子

```
var lilei=  
new Student("Li Lei",11);
```

`__proto__`

```
sname : "Li Lei",  
sage : 11
```

lilei.toString()

孩子

```
var hmm=  
new Student("Han Meimei",12)
```

`__proto__`

```
sname : "Han Meimei",  
sage : 12
```

Hmm.toString()

数组类型

```
function Array(){  
  ... ..  
}
```

prototype

Array.prototype

```
sort() reverse() ...  
sum() toString()  
"元素,元素,..."
```

继承

```
var arr=[ 1, 2, 3 ] __proto__
```

```
元素1 元素2, ...  
0 1 ...
```

arr.sum()

arr.toString() "1,2,3"

日期类型

```
function Date(){  
  ... ..  
}
```

prototype

Date.prototype

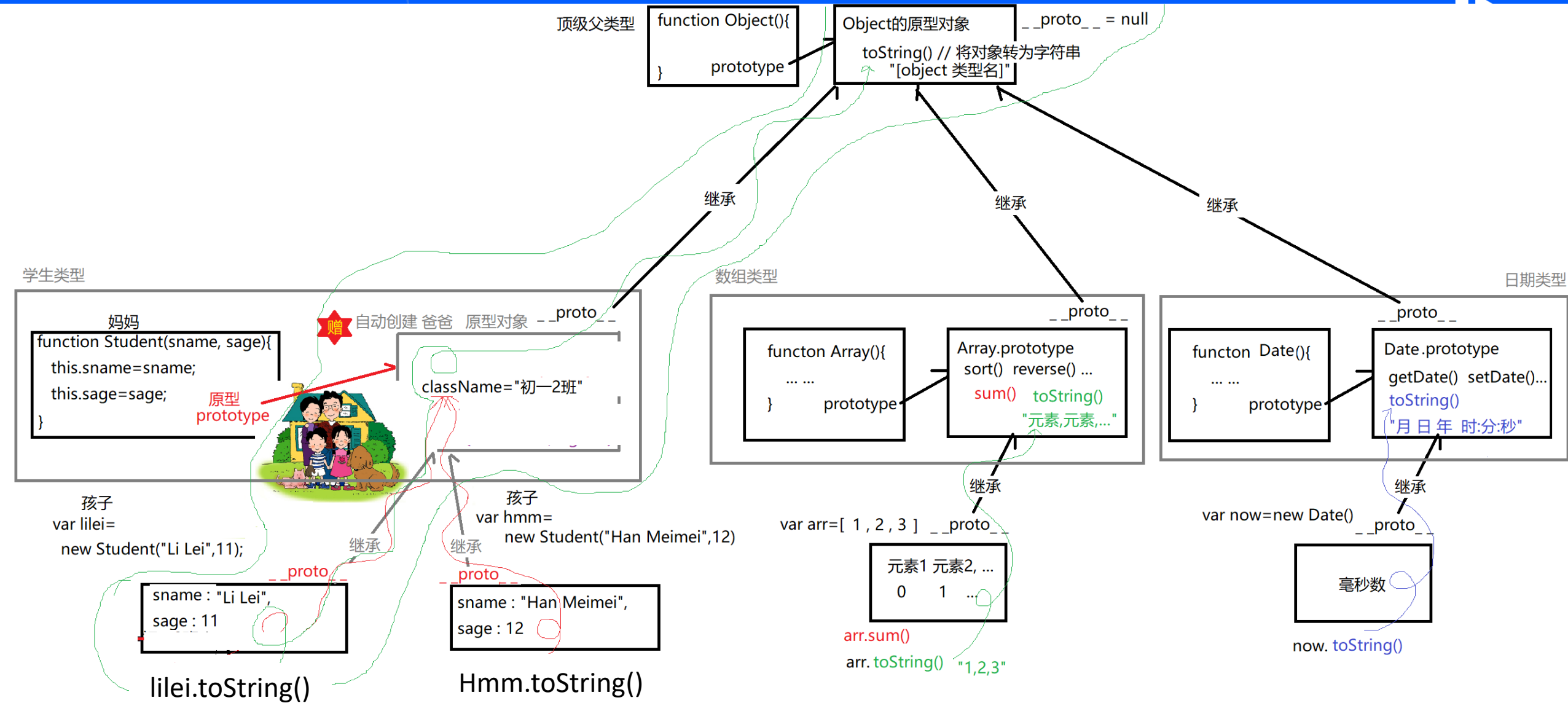
```
getDate() setDate()...
```

继承

```
var now=new Date() __proto__
```

毫秒数

now.toString()



封装

继承

多态

# 问题:Student 家怎么办?

Student家的孩子也想用好用的  
toString()

# 解决：重写

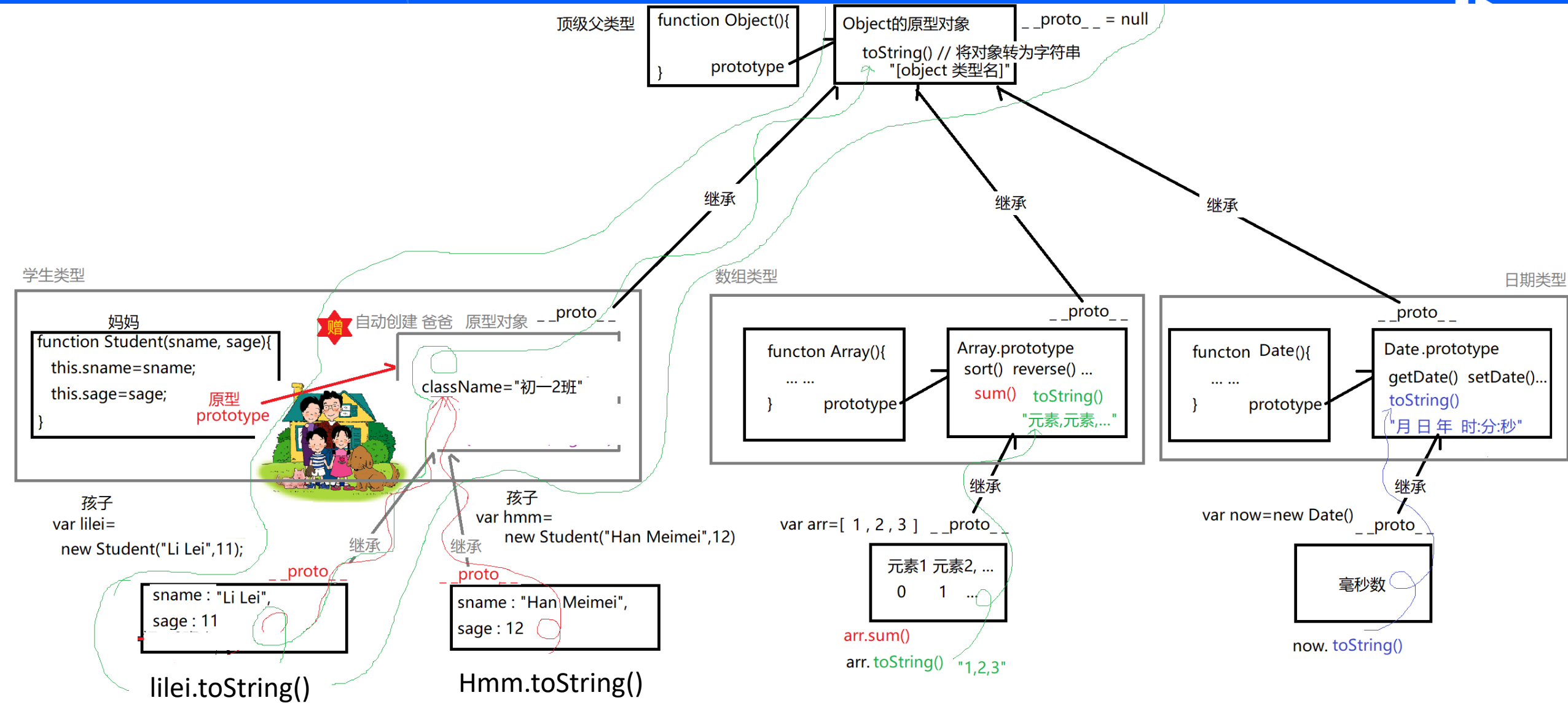
封装

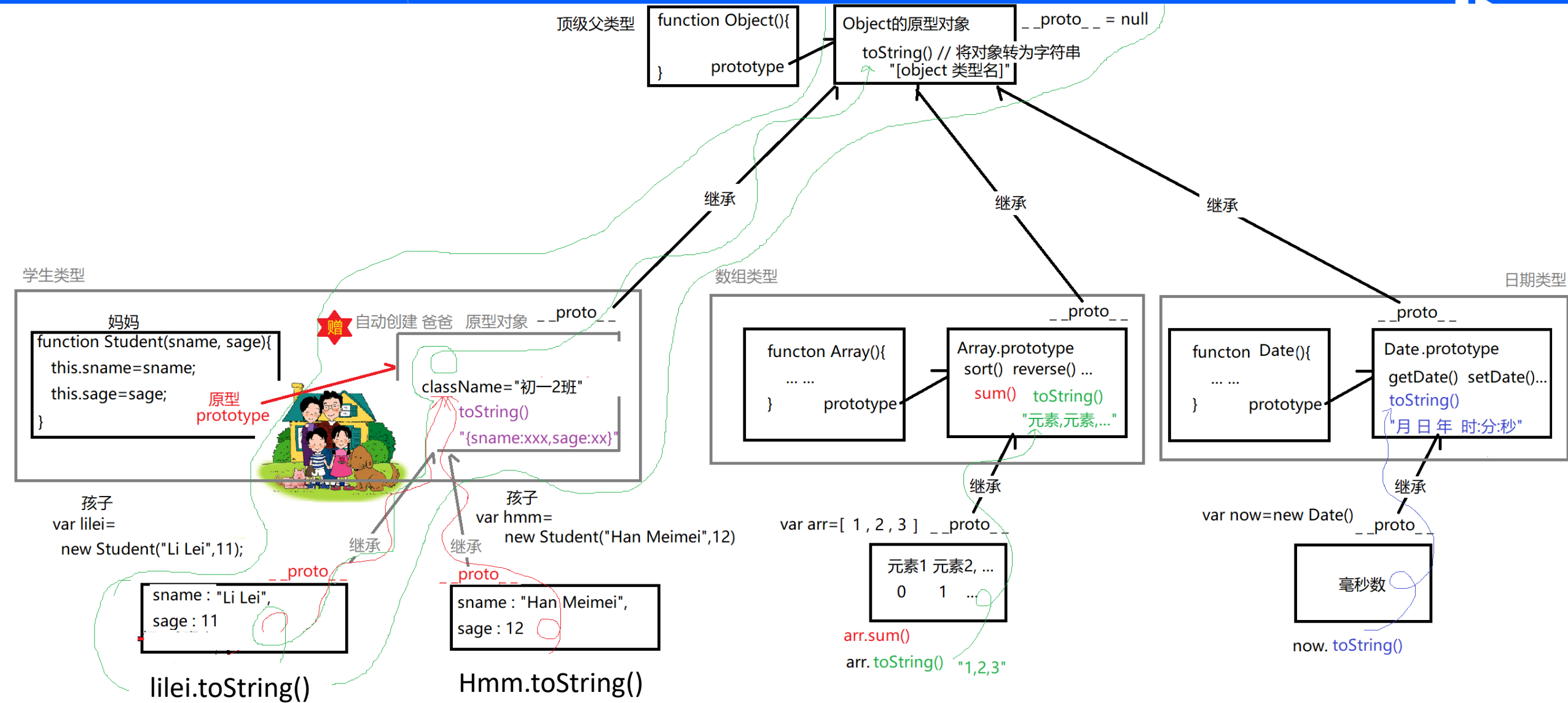
继承

多态

- Student家的原型对象爸爸，也重写一个新的好用的toString()方法，给Student家的子对象孩子们使用

```
function Student(sname,sage){
  this.sname=sname;
  this.sage=sage;
}
//向Student的原型对象中添加一个好用的toString方法
Student.prototype.toString=function(){
  return `{ sname:${this.sname}, sage:${this.sage} }`
}
var lilei=new Student("Li Lei",11);
console.log(lilei);
var arr=[1,2,3];
var now=new Date();
console.log(lilei.toString()); { sname:Li Lei, sage:11 }
console.log(arr.toString());
console.log(now.toString());
```





顶级父类型

```
function Object(){  
  prototype  
}
```

Object的原型对象

```
toString() // 将对象转为字符串  
"[object 类型名]"
```

`__proto__ = null`

继承

继承

继承

学生类型

妈妈

```
function Student(sname, sage){  
  this.sname=sname;  
  this.sage=sage;  
}
```

原型  
prototype

自动创建 爸爸 原型对象



```
className="初一2班"  
toString()  
"{sname:xxx,sage:xx}"
```

孩子

```
var lilei=  
new Student("Li Lei",11);
```

`__proto__`

```
sname : "Li Lei",  
sage : 11
```

`lilei.toString()`

继承

孩子

```
var hmm=  
new Student("Han Meimei",12)
```

`__proto__`

```
sname : "Han Meimei",  
sage : 12
```

`hmm.toString()`

继承

数组类型

```
function Array(){  
  ...  
}
```

prototype

```
Array.prototype  
sort() reverse() ...  
sum() toString()  
"元素,元素,..."
```

继承

```
var arr=[ 1, 2, 3 ]
```

```
元素1 元素2, ...  
0 1 ...
```

`arr.sum()``arr.toString()` "1,2,3"

日期类型

```
function Date(){  
  ...  
}
```

prototype

```
Date.prototype  
getDate() setDate()...  
toString()  
"月 日 年 时:分:秒"
```

继承

```
var now=new Date()
```

毫秒数

`now.toString()`



# 面向对象 小总结: 封装, 继承, 多态

①封装: 创建对象, 2种:

如果只创建一个对象: {}

如果反复创建多个相同结构的对象: 构造函数

②继承: 所有子对象共用的属性值和方法, 都要放在构造函数的原型对象中

③多态: 重写: 只要觉得从父对象继承来的成员  
不要用, 都在子对象中重写同名成员



Q2: this

——this共有几种情况

# 判断this

**一定不要看在哪里定义。一定只看将来在哪里，如何被调用。**

1. obj.fun() this->.前的obj对象
2. new构造函数() this->new正在创建的新对象
3. 构造函数.prototype.fun=function(){}  
因为将来原型对象中的方法，都是“子对象.fun()”方式调用。  
所以，this->将来调用这个fun函数的.前的某个子对象

# 判断this

**一定不要看在哪里定义。一定只看将来在哪里，如何被调用。**

4. fun()、匿名函数自调和回调函数中的 this->window  
严格模式(usestrict)下, this->undefined  
因为这类函数调用时, 前边即没有., 也没有new!

比如:

```
(function(){ ... this ...})();
```

和

```
arr.forEach(  
  function(){ ... this ... }  
);
```

## 判断this

**一定不要看在哪里定义。一定只看将来在哪里，如何被调用。**

5.button.onclick=function(){}

或

button.addEventListener("click",function(){...})

DOM事件处理函数里的this->当前正在出发事件的.前的DOM元素对象。

强调:

这里不能改成箭头函数!

一旦改为箭头函数, this指外层的window。

功能就会出错。

# 判断this

一定不要看在哪里定义。一定只看将来在哪里，如何被调用。



## 6. Vue中this默认都指当前vue对象

```
<button@click="fun">
```

```
export default{
```

```
  methods:{
```

```
    //vue中methods中的方法中的this默认都指当前vue组件对象
```

```
    fun(e){ e.target }
```

```
    //如果想获得当前出发事件的DOM元素对象，必须用$event关键字和e.target联合使用
```

```
  }
```

```
}
```

## 判断this

**一定不要看在哪里定义。一定只看将来在哪里，如何被调用。**

7. 箭头函数中的this->当前函数之外最近的作用域中的this
- 几乎所有匿名函数都可用箭头函数简化
  - 箭头函数是对大多数匿名函数的简写





**问题：**  
**所有function都  
可改为箭头函数吗？**

```
var lilei={  
}
```



```
var lilei={  
    sname:"Li Lei",  
    friends:["涛涛","楠楠","东东"],  
}
```



```
var lilei={  
    sname:"Li Lei",  
    friends:["涛涛","楠楠","东东"],  
    intr:function(){  
  
    }  
}
```



```
var lilei={  
    sname:"Li Lei",  
    friends:["涛涛","楠楠","东东"],  
    intr:function(){  
        //Li Lei认识涛涛  
        //Li Lei认识楠楠  
        //Li Lei认识东东  
    }  
}
```



```
var lilei={  
    sname:"Li Lei",  
    friends:["涛涛","楠楠","东东"],  
    intr:function(){  
        this.friends.forEach(  
  
        )  
    }  
}
```





```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }
    )
  }
}
```





```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }
    )
  }
}
lilei.intr();
```

# 结果，是你想要的吗？

—

- undefined 认识 涛涛
- undefined 认识 楠楠
- undefined 认识 东东



原因:



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }
    )
  }
}
lilei.intr();
```



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }
    )
  }
}
lilei.intr();
```

intr的作用域范围



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛", "楠楠", "东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n}` )
      }
    )
  }
}
lilei.intr();
```

intr的作用域范围

回调函数的作用域范围



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n}` )
      }
    )
  }
}
lilei.intr();
```

intr的作用域范围

回调函数的作用域范围





```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```

intr的作用域范围

```
  intr:function(){
```

```
    this.friends.forEach(  
      function(元素值n){
```

回调函数的作用域范围

```
        console.log(` ${this.sname} 认识 ${元素值n}`)  
      }  
    )  
  }
```

```
}  
lilei.intr();
```



```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```

intr的作用域范围

```
  intr:function(){
```

```
    this.friends.forEach(  
      function(元素值n){
```

回调函数的作用域范围

```
        console.log(` ${this.sname} 认识 ${元素值n}` )
```

```
      }  
    )  
  }
```

```
}
```

```
lilei.intr();
```



```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```

intr的作用域范围

```
  intr:function(){  
    this.friends.forEach(  
      lilei
```

```
      function(元素值n){  
        console.log(` ${this.sname} 认识 ${元素值n}` )  
      }  
    )  
  }
```

回调函数的作用域范围

认识

```
}  
lilei.intr();
```



```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```

intr的作用域范围

```
  intr:function(){  
    this.friends.forEach(  
      lilei.friends
```

```
    function(元素值n){  
      console.log(` ${this.sname} 认识 ${元素值n}` )  
    }  
  )  
}
```

回调函数的作用域范围

认识

```
lilei.intr();
```



```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```

intr的作用域范围

```
  intr:function(){  
    this.friends.forEach(  
      lilei.friends v
```

```
    function(元素值n){  
      console.log(` ${this.sname} 认识 ${元素值n}` )  
    }  
  )  
}
```

回调函数的作用域范围

认识

```
lilei.intr();
```



```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```

intr的作用域范围

```
  intr:function(){  
    this.friends.forEach(  
      lilei.friends v
```

```
      function(元素值n){  
        console.log(` ${this.sname} 认识 ${元素值n}` )  
      }  
    )  
  }  
}
```

回调函数的作用域范围

认识

```
lilei.intr();
```



```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```

intr的作用域范围

```
  intr:function(){  
    this.friends.forEach(  
      lilei.friends v
```

回调函数的作用域范围

```
function(元素值n){  
  console.log(` ${this.sname} 认识 ${元素值n}` )  
}
```

```
)  
}  
}  
lilei.intr();
```



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
```



intr的作用域范围

```
  intr:function(){
    this.friends.forEach(
      lilei.friends v
```

回调函数的作用域范围

```
function(元素值n){
  console.log(` ${this.sname} 认识 ${元素值n}` )
}
```

window

```
)
}
}
lilei.intr();
```



```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```

intr的作用域范围

```
  intr:function(){  
    this.friends.forEach(  
      lilei.friends v
```

回调函数的作用域范围

```
    function(元素值n){  
      console.log(` ${this.sname} 认识 ${元素值n} `)  
    }  
    window.sname
```

```
  )  
}  
}
```

```
lilei.intr();
```



```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```

intr的作用域范围

```
  intr:function(){  
    this.friends.forEach(  
      lilei.friends v
```

回调函数的作用域范围

```
function(元素值n){  
  console.log(` ${this.sname} 认识 ${元素值n}` )  
}
```

window.sname ×

undefined

```
}  
lilei.intr();
```



**比喻：函数作用域就  
像实体墙**

**隔绝了内部的this与外部的this，  
使内外this不一样**



**解决：回调函数改为  
箭头函数**

# 箭头函数 重要特征

- 箭头函数可让函数内的this与函数外的this保持一致！



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛", "楠楠", "东东"],
  intr:function(){
    this.friends.forEach(
      lilei.friends √
      function(元素值n){//把墙推倒
        console.log(` ${this.sname} 认识 ${元素值n}` )
      }
    )
  }
}
lilei.intr();
```

intr的作用域范围



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      lilei.friends √
      元素值n=>{ //把墙推倒
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }
    )
  }
}
lilei.intr();
```

intr的作用域范围



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
```



intr的作用域范围

```
  intr:function(){
    this.friends.forEach(
      lilei.friends v
      元素值n=>{ //把墙推倒
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }
    )
  }
}
```

```
}
lilei.intr();
```

```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
```



intr的作用域范围

```
intr:function(){
```

```
  this.friends.forEach(
```

~~lilei.friends v~~

元素值n=>{~~//把墙推倒~~

console.log(` \${this.sname} 认识 \${元素值n} `)

```
  }
```

```
)
```

```
}
```

```
}
```

```
lilei.intr();
```

```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
```



intr的作用域范围

```
intr:function(){
```

```
  this.friends.forEach(
```

lilei.friends v

元素值n=>{ //把墙推倒

```
    console.log(` ${this.sname} 认识 ${元素值n} `)
```

lilei

```
  }
```

```
)
```

```
}
```

```
}
```

```
lilei.intr();
```

```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],
```



intr的作用域范围

```
  intr:function(){
```

```
    this.friends.forEach(  
      lilei.friends v  
      元素值n=>{ //把墙推倒
```

```
        console.log(` ${this.sname} 认识 ${元素值n} ` )  
        lilei.sname
```

```
      }  
    )  
  }
```

```
}
```

```
}
```

```
lilei.intr();
```

```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
```



intr的作用域范围

```
intr:function(){
```

```
  this.friends.forEach(
    lilei.friends v
```

元素值n=>{ //把墙推倒

```
    console.log(` ${this.sname} 认识 ${元素值n} `)
```

```
  }
```

lilei.sname

"Li Lei" v

```
  )
```

```
}
```

```
}
```

```
lilei.intr();
```



# 对象的intr方法上 还有个function

能不能改成箭头函数？



```
var lilei={
    sname:"Li Lei",
    friends:["涛涛","楠楠","东东"],
    intr:function(){
        this.friends.forEach(

            元素值n=>{ //把墙推倒
                console.log(` ${this.sname} 认识 ${元素值n} `)
            }
        )
    }
}
lilei.intr();
```



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:()=>{
    this.friends.forEach(
      元素值n=>{ //把墙推倒
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }
    )
  }
}
lilei.intr();
```

不能访问undefined的forEach属性



但是, intr()外的对象lilei={}, 只是new Object的简写  
不是作用域, 不包含this。

所以, intr()中的this会超出lilei={ }大括号, 直接指向全局window中的this

```
var lilei={  
  sname:"Li Lei",  
  friends:["涛涛","楠楠","东东"],  
  intr:()=>{
```

因为, 箭头函数内部this指向外部this  
所以, intr()中的this指向intr()外的this

```
    this.friends.forEach(  
      元素值n=>{//把墙推倒  
        console.log(` ${this.sname} 认识 ${元素值n}` )  
      }  
    )  
  }  
}  
lilei.intr();
```

## window 全局作用域



```
var lilei={ //不是作用域, 只是new Object()的简写
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:()=>{
    this.friends.forEach(
```

元素值n=>{ //把墙推倒

console.log(` \${this.sname} 认识 \${元素值n} ` )

}

)

}

}

lilei.intr();

## window 全局作用域



var lilei={//不是作用域, 只是new Object()的简写

sname:"Li Lei",

friends:["涛涛","楠楠","东东"],

intr:()=>{

this.friends.forEach(

window

元素值n=>{//把墙推倒

console.log(`\${this.sname} 认识 \${元素值n}`)

}

)

}

}

lilei.intr();

## window 全局作用域



var lilei={ //不是作用域, 只是new Object()的简写

  sname:"Li Lei",

  friends:["涛涛","楠楠","东东"],

  intr:()=>{

    this.friends.forEach(

  window.friends

    元素值n=>{//把墙推倒

      console.log(` \${this.sname} 认识 \${元素值n} `)

    }

  )

}

}

lilei.intr();

## window 全局作用域



```
var lilei={ //不是作用域, 只是new Object()的简写
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:()=>{
    this.friends.forEach(
```

× window.friends

元素值n=>{ //把墙推倒

console.log(` \${this.sname} 认识 \${元素值n} ` )

}

)

}

}

lilei.intr();

# 所以：

1). 今后，

- 如果函数中就不包含this，
- 或刚好希望函数内的this与外部this保持一致时，
- 就可以改为箭头函数

2). 今后，

- 如果反而不希望函数内的this与函数外的this保持一致时，
- 都不能改为箭头函数。

**比如：对象的方法就  
不能改为箭头函数**

---

## 解决:

- ES6中为对象方法提供了一种专门的不带function的简写。

```
var 对象名={  
    属性名: 属性值,  
    方法名:function(){ ... },  
    方法名(){ ... this.属性名 ... }  
}
```

既不带:function, 又不要加=>

- 好处:
  - 既省略了function,
  - 但是又不等同于箭头函数, 不会影响内部的this!



## window 全局作用域



var lilei={//不是作用域, 只是new Object()的简写

sname:"Li Lei",

friends:["涛涛","楠楠","东东"],

intr:function(){

intr的作用域范围

this.friends.forEach(

元素值n=>{//把墙推倒

console.log(`\${this.sname} 认识 \${元素值n}`)

}

)

}

}

lilei.intr();

## window 全局作用域



var lilei={//不是作用域, 只是new Object()的简写

sname:"Li Lei",

friends:["涛涛","楠楠","东东"],

intr(){

intr的作用域范围

this.friends.forEach(

元素值n=>{//把墙推倒

console.log(` \${this.sname} 认识 \${元素值n} `)

}

)

}

}

lilei.intr();

## window 全局作用域



var lilei={//不是作用域, 只是new Object()的简写

  sname:"Li Lei",

  friends:["涛涛","楠楠","东东"],

  //ES6中对象方法的专用简写, 纯简写, 不改变this intr的作用域范围

  intr(){

    this.friends.forEach(

      元素值n=>{//把墙推倒

        console.log(` \${this.sname} 认识 \${元素值n} `)

      }

    )

  }

}

lilei.intr();

## window 全局作用域



var lilei={//不是作用域, 只是new Object()的简写

sname:"Li Lei",

friends:["涛涛","楠楠","东东"],

//ES6中对象方法的专用简写, 纯简写, 不改变this intr的作用域范围

intr(){

this.friends.forEach(

元素值n=>{//把墙推倒

console.log(` \${this.sname} 认识 \${元素值n} `)

}

)

}

}  
lilei.intr();

## window 全局作用域



var lilei={//不是作用域, 只是new Object()的简写

sname:"Li Lei",

friends:["涛涛","楠楠","东东"],

//ES6中对象方法的专用简写, 纯简写, 不改变this intr的作用域范围

intr(){

this.friends.forEach(

lilei.friends ✓

元素值n=>{//把墙推倒

console.log(` \${this.sname} 认识 \${元素值n} `)

}

)

}

}  
lilei.intr();

# window 全局作用域



var lilei={//不是作用域, 只是new Object()的简写

sname:"Li Lei",

friends:["涛涛","楠楠","东东"],

//ES6中对象方法的专用简写, 纯简写, 不改变this intr的作用域范围

intr(){

this.friends.forEach(

lilei.friends.v

元素值n=>{//把墙推倒

console.log(` \${this.sname} 认识 \${元素值n} `)

}

)

}

}  
lilei.intr();

# window 全局作用域



var lilei={//不是作用域, 只是new Object()的简写

sname:"Li Lei",

friends:["涛涛","楠楠","东东"],

//ES6中对象方法的专用简写, 纯简写, 不改变this intr的作用域范围

intr(){

this.friends.forEach(

lilei.friends.v

元素值n=>{//把墙推倒

console.log(`\${this.sname} 认识 \${元素值n}`)

}

v lilei.sname

"Li Lei"

)

}

}  
lilei.intr();

# “箭头函数没有/不是作用域”

---

- 错误!
- 箭头函数只让this，指向外部作用域的this
- 而箭头函数内的局部变量，依然只能在箭头函数内使用。出了箭头函数不能用!
- 所以，箭头函数依然是作用域！只不过影响this而已！不影响局部变量



# 箭头函数 底层原理

- 发现，改bind效果和箭头函数效果一样



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }
    )
  }
}
lilei.intr();
```



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }.bind(this)
    )
  }
}
lilei.intr();
```



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }.bind(this)
    )
  }
}
lilei.intr();
```



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }.bind(this)
    )
  }
}
lilei.intr();
```



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
      //lilei
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }.bind(this)
    )
  }
}
lilei.intr();
```



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
//lilei
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
      }.bind(this)
//lilei
    )
  }
}
lilei.intr();
```



```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
//lilei
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
//lilei
      }.bind(this)
//lilei
    )
  }
}
```





```
var lilei={
  sname:"Li Lei",
  friends:["涛涛","楠楠","东东"],
  intr:function(){
    this.friends.forEach(
//lilei
      function(元素值n){
        console.log(` ${this.sname} 认识 ${元素值n} `)
//lilei ✓
      }.bind(this)
//lilei
    )
  }
}
```

# 结论：箭头函数底层 相当于.bind()

永久绑定外部this！

---

**所以: call无法替换  
箭头函数中this**

---

# 判断this


**一定不要看在哪里定义。一定只看将来在哪里，如何被调用。**

- 8. 其实，对于this，我们不一定逆来顺受
  - - 可用call或apply，临时替换一次函数中的this
  - - 可用bind，永久替换函数中的this

# 替换this: 3种情况

---

## (1)一次调用函数时, 临时替换一次this



要调用的函数.call(替换this的对象, 实参值1,...)  
调用

call做3件事儿:

- 1). 立刻调用一次点前的函数
- 2). 自动将.前的函数中的this替换为指定的新对象
- 3). 还能向要调用的函数中传实参值

//有一个公共的计算器函数，计算每个人的总工资

//                    底薪      奖金1      奖金2

```
function jisuan(base, bonus1, bonus2){  
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )  
}
```



//有一个公共的计算器函数，计算每个人的总工资

//                    底薪      奖金1      奖金2

```
function jisuan(base, bonus1, bonus2){  
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )  
}
```

//两个员工:

```
var lilei={ename:"Li Lei"};
```

```
var hmm={ename:"Han Meimei"};
```





**lilei想调用jisuan函数**



**错误做法:**





```
//有一个公共的计算器函数，计算每个人的总工资
//          底薪    奖金1    奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

//两个员工:
var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};

//lilei想用jisuan()计算自己这个月的薪资
//错误:
//  jisuan(10000,1000,2000);
//          this->window
```



```
//有一个公共的计算器函数，计算每个人的总工资
//          底薪    奖金1    奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

//两个员工：
var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};

//lilei想用jisuan()计算自己这个月的薪资
//错误：
// lilei.jisuan(10000,1000,2000);
//lilei自己没有计算，lilei的爹Object也没有计算函数，
//所以报错：不是一个function
```



**正确做法:**





```
//有一个公共的计算器函数，计算每个人的总工资
//          底薪    奖金1    奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

//两个员工：
var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};

//要调用      替换this  还能
//的函数  调用  的对象  传实参
  jisuan.call(lilei,      10000, 1000, 2000);
//          |           ↓       ↓       ↓
//相当于jisuan( ↓       base,bonus1,bonus2)
//          this.ename
//          lilei.ename
//          "Li Lei"
```



```
//有一个公共的计算器函数，计算每个人的总工资
//          底薪    奖金1    奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

//两个员工:
var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};

//要调用      替换this 还能
//的函数 调用 的对象 传实参

  jisuan.call(  hmm,5000, 6000, 3000)
//          |    ↓    ↓    ↓
//相当于jisuan(  ↓ base,bonus1,bonus2 )
//          this.ename
//          hmm.ename
//          "Han Meimei"
```

## (2).特殊:

---

- 如果多个实参值是放在一个数组中给的。
- 需要既替换this，又要拆散数组再传参



**错误做法:**





```
//有一个公共的计算器函数，计算每个人的总工资
//      底薪      奖金1      奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

//两个员工：
var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};

//lilei想用jisuan()计算自己这个月的薪资
var arr=[10000, 1000, 2000];

//错误：
// jisuan.call(lilei,arr);
// call不能打散数组，
// arr只能传给第一个形参base，后两个形参接到的都是undefined。
// 无法正常计算薪资
```

## (2).特殊:

---

- 如果多个实参值是放在一个数组中给的。
- 需要既替换this, 又要拆散数组再传参
- 要调用的函数.apply( 替换this的对象, 包含实参值的数组 )
- 应用
- 也做3件事儿:
  - 1). 调用.前的函数
  - 2). 替换.前的函数中的this为指定对象
  - 3). 先拆散数组为多个元素值,  
再分别传给函数的形参变量



```
//有一个公共的计算器函数，计算每个人的总工资
//          底薪    奖金1    奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

//两个员工：
var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};
//lilei想用jisuan()计算自己这个月的薪资
var arr=[10000, 1000, 2000];
jisuan.apply(lilei, arr );
//          |          拆散数组—apply的特异功能
//          |          10000, 1000, 2000 —多个值
//相当于jisuan( ↓      base,bonus1,bonus2)
//          this.ename
//          lilei.ename
//          "Li Lei"
```



# 问题:

---

- call和apply只能临时替换一次this
- 如果反复调用函数，每次都要用call和apply临时替换this，太麻烦！
- 比如：
  - jisuan.call(lilei, 10000,1000,2000)
  - jisuan.call(lilei, 10000,2000,2000)
  - jisuan.call(lilei, 10000,1000,1000)

### **(3). 创建函数副本 并永久绑定this**

- a. `var 新函数=原函数.bind(替换this的对象)`
- b. 2件事:
  - 1). 创建一个和原函数一模一样的新函数副本
  - 2). 永久替换新函数中的this为指定对象



```
//有一个公共的计算器函数，计算每个人的总工资
//          底薪    奖金1    奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};

//lilei每个月都要call，很麻烦
//于是买一个一模一样的自己专属的jisuan()函数。
var jisuan2=jisuan.bind(lilei);
//jisuan2=function(basic, bonus1, bonus2){
//  ...` ${lilei.ename}的总工资是:${base+bonus1+bonus2}` ...
//}
```

## c. 如何使用新函数:

- 新函数(实参值,...)
- 强调:
  - 因为bind()已提前将指定对象替换新函数中this,
  - 所以后续每次调用新函数副本时,
  - 不需要再替换this了!





```
//有一个公共的计算器函数，计算每个人的总工资
//          底薪    奖金1    奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};

//lilei每个月都要call，很麻烦
//于是买一个一模一样的自己专属的jisuan()函数。
var jisuan2=jisuan.bind(lilei);
//jisuan2=function(basic, bonus1, bonus2){
//  ...` ${lilei.ename}的总工资是:${base+bonus1+bonus2}` ...
//}
jisuan2(10000, 1000,2000);
```

## d. 其实,

- bind()不但可以提前永久绑定this,
- 而且还能提前永久绑定部分实参值
- var 新函数=原函数.bind(替换this的对象, 不变的实参值)
- 也做3件事儿:
  - 1). 创建一模一样的新函数副本
  - 2). 永久替换this为指定对象
  - 3). 永久替换部分形参变量为固定的实参值!

```
//有一个公共的计算器函数，计算每个人的总工资
//          底薪    奖金1    奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};

//lilei每个月都要call，很麻烦
//于是lilei决定买一个一模一样的自己专属的jisuan()函数。
var jisuan2=jisuan.bind(lilei,10000);
//jisuan2=function(10000, bonus1, bonus2){
// console.log(
//   ` ${lilei.ename}的总工资是:${10000+bonus1+bonus2}` )
//}
```



## g. 如何调用:

- 只需要传剩余的实参值即可!
- 新函数(剩余实参值,...,...)



```
//有一个公共的计算器函数，计算每个人的总工资
//          底薪    奖金1    奖金2
function jisuan(base, bonus1, bonus2){
  console.log(` ${this.ename}的总工资是:${base+bonus1+bonus2}` )
}

var lilei={ename:"Li Lei"};
var hmm={ename:"Han Meimei"};

//lilei每个月都要call，很麻烦
//于是lilei决定买一个一模一样的自己专属的jisuan()函数。
var jisuan2=jisuan.bind(lilei,10000);
//jisuan2=function(10000, bonus1, bonus2){
//  ...` ${lilei.ename}的总工资是:${10000+bonus1+bonus2}` ...
//}
jisuan2(1000,2000);
```

## h. 强调:

- 被bind()永久绑定的this, 即使用call, 也无法再替换为其它对象了。
- ——箭头函数的底层原理

## (4). 总结:

- a. 只在一次调用函数时, 临时替换一次this: call
- b. 既要替换一次this, 又要拆散数组再传参: apply
- c. 创建新函数副本, 并永久绑定this: bind



# Q3: 创建对象

——Js中创建对象共有几种方式





# JS中创建对象10种方式总结

- 1. new Object() 缺点: 步骤多
- 2. 字面量: var 对象名={} 缺点: 如果反复创建多个对象, 代码会很冗余
- 3. 工厂函数方式:

```
function createPerson(name, age) {  
    var o = new Object();  
    o.name = name;  
    o.age = age;  
    o.say = function() { alert(this.name); }  
    return o;  
}  
var p1 = createPerson("lilei", 11);
```

缺点: 本质还是Object(), 将来无法根据对象的原型对象准确判断对象的类型



# JS中创建对象10种方式总结

- 4. 构造函数方式：2步：
  - 先用构造函数定义一类对象的统一属性结构和方法
  - 再用new调用构造函数，反复创建相同属性结构，不同属性值的多个对象
  - 缺点：如果构造函数中包含方法，则重复创建，浪费内存

```
function Student(sname, sage){  
    this.sname=sname;  
    this.sage=sage;  
    this.intr=function(){ ... }  
}  
var lilei=new Student("Li Lei",11)
```



# JS中创建对象10种方式总结

- 5. 原型对象方式：先创建完全相同的对象，再给子对象添加个性化属性。

```
function Person() {  
}  
Person.prototype.name = "主人很懒";  
Person.prototype.age = 11;  
Person.prototype.say = function() {  
  console.log(this.name);  
};
```

var p1 = new Person(); //创建一个实例p1

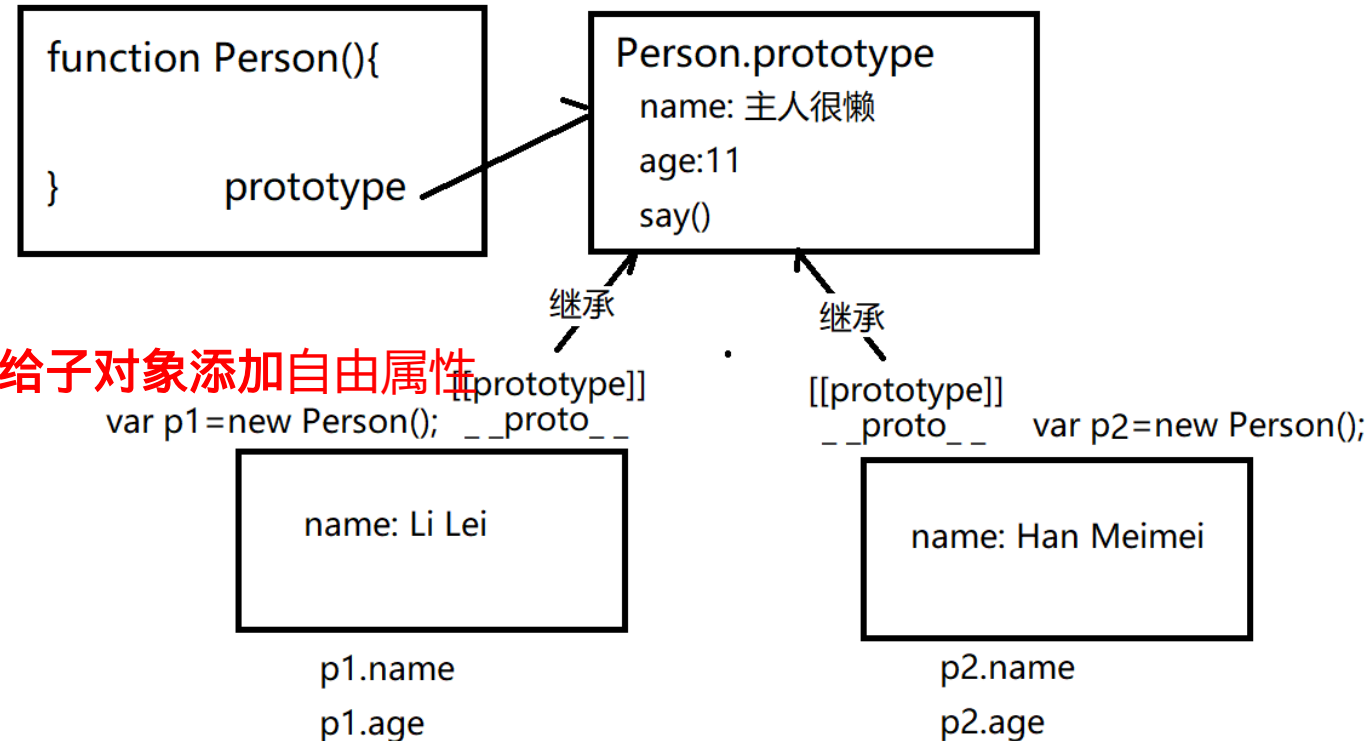
p1.name="Li Lei" //禁止修改共有属性，而是给子对象添加自由属性

var p2 = new Person(); //创建实例p2

p2.name = "Han Meimei"; //同上

console.log(lilei);

console.log(hmm);



- 缺点：步骤繁琐！



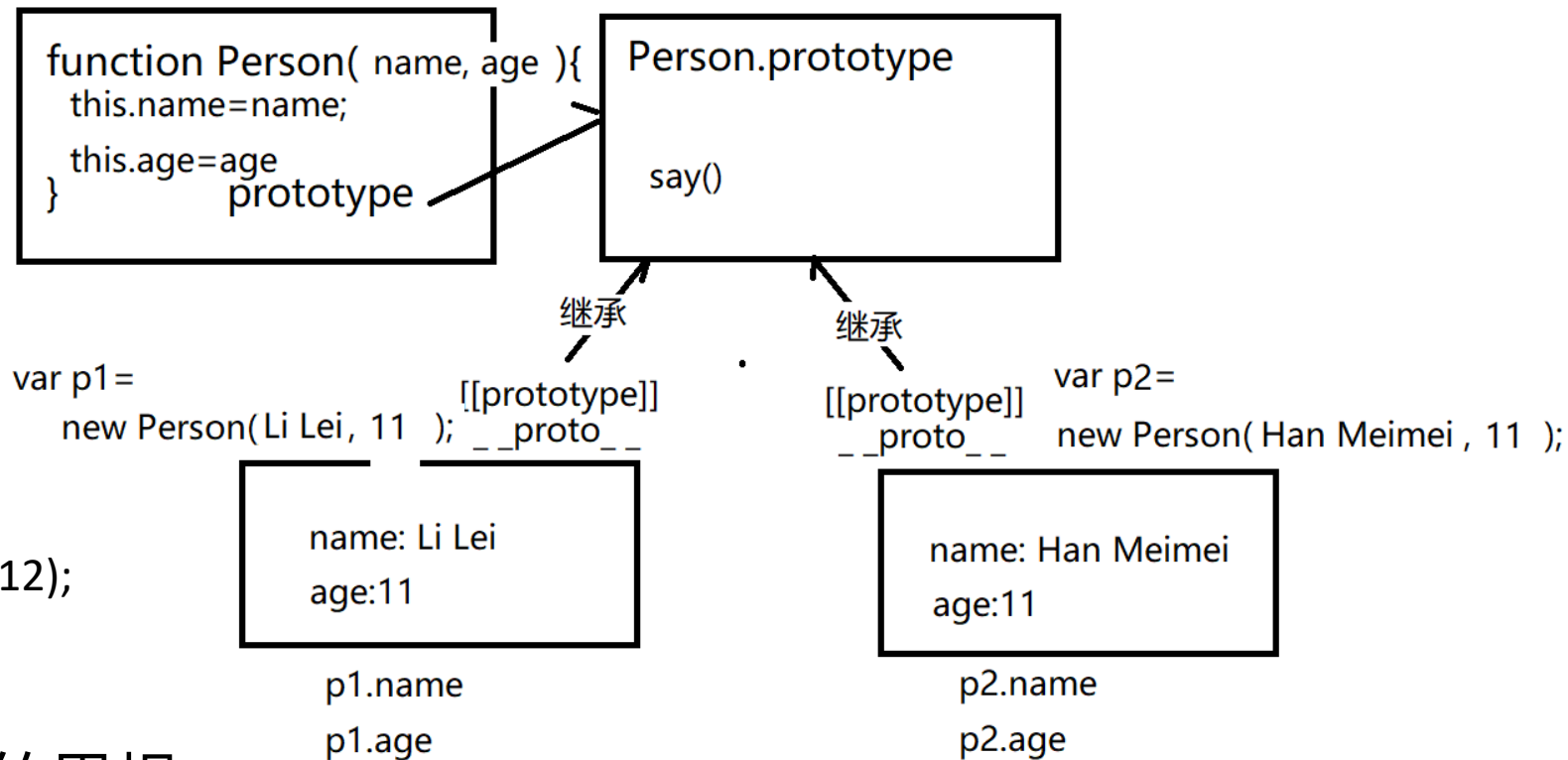
# JS中创建对象10种方式总结

- 6. 混合模式：先创建完全相同的对象，再给子对象添加个性化属性。

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

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

```
var p1 = new Person("Li Lei",11);  
var p2 = new Person("Han Meimei",12);  
console.log(lilei);  
console.log(hmm);
```



- 缺点: 不符合面向对象封装的思想

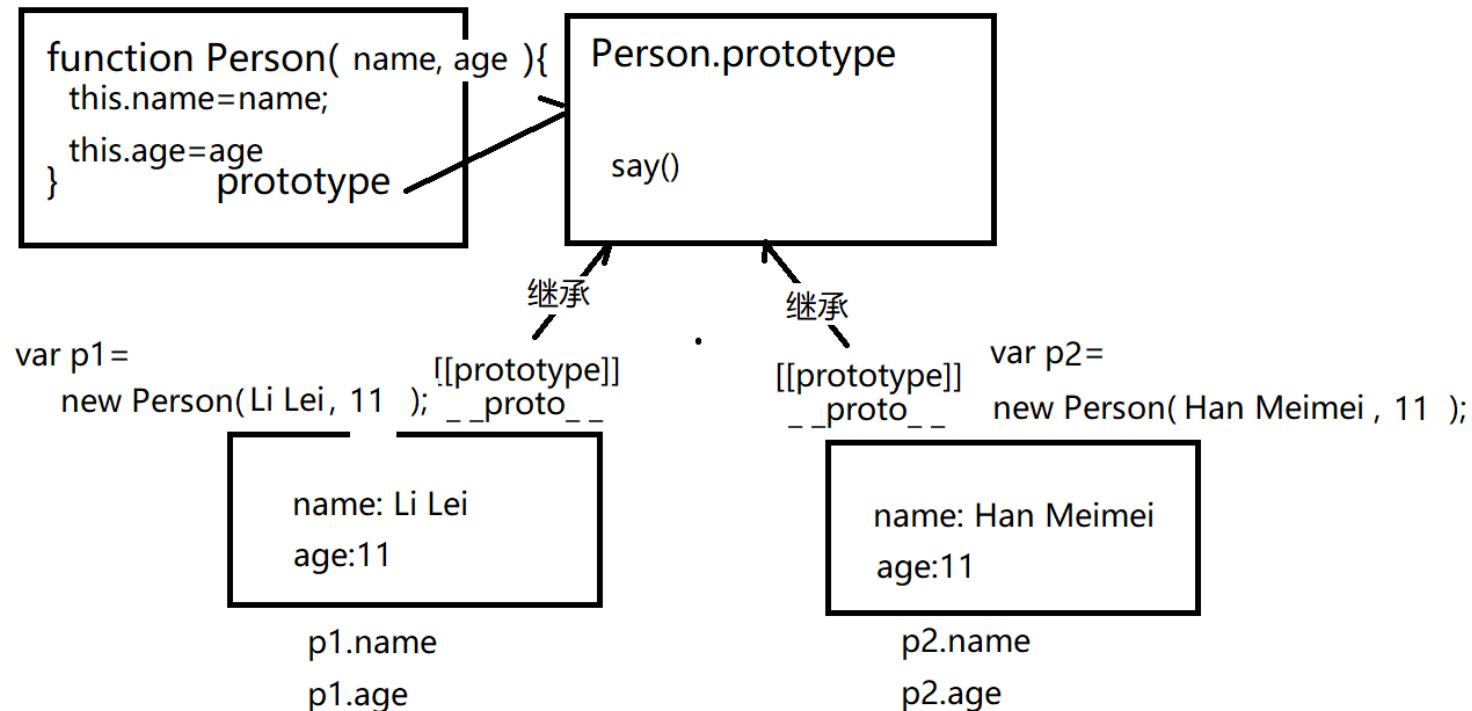


# JS中创建对象10种方式总结

- 7. 动态混合：先创建完全相同的对象，再给子对象添加个性化属性。
- 缺点：语义不符，其实if只在创建第一个对象时有意义。

```
function Person(name, age) {  
  this.name=name;  
  this.age=age;  
  if(Person.prototype.say===“undefined”){  
    Person.prototype.say = function(){  
      console.log(this.name);  
    };  
  }  
}
```

```
var p1 = new Person(“Li Lei”,11);  
var p2 = new Person(“Han Meimei”,12);  
console.log(lilei);  
console.log(hmm);
```





# JS中创建对象10种方式总结

- 8. 寄生构造函数：构造函数里调用其他的构造函数

```
function Person(name, age) {  
  this.name=name;  
  this.age=age;  
  if(Person.prototype.say===“undefined”){  
    Person.prototype.say = function(){  
      console.log(this.name);  
    };  
  }  
}
```

```
function Student(name, age, className) {  
  var p=new Person(name,age); //借鸡生蛋——橘子  
  p.className=className  
  return p;  
}
```

```
var p1 = new Student(“Li Lei”,11,“初一2班”);  
var p2 = new Student(“Han Meimei”,12,“初二2班”);  
console.log(lilei);  
console.log(hmm);
```

- 缺点: 可读性差。



# JS中创建对象10种方式总结

- 9. ES6 Class:
- 什么是Class:
  - 程序中专门集中保存
  - 一种类型的所有子对象
  - 的统一属性结构和方法定义
  - 的程序结构。

# 如何定义class: 3句话: 混合模式基础上

- i. 用class{}包裹原构造函数+原型对象方法



//想定义学生类型，描述所有学生的统一结构和功能

```
function Student(sname, sage){  
    this.sname=sname;  
    this.sage=sage;  
}
```

```
Student.prototype.intr=function(){  
    console.log(  
        `I'm ${this.sname}, I'm ${this.sage}` );  
}
```

//创建一个学生类型的对象

```
var lilei=new Student("Li Lei",11);  
console.log(lilei);  
lilei.intr();
```



//想定义学生类型，描述所有学生的统一结构和功能

```
class {  
  function Student(sname, sage){  
    this.sname=sname;  
    this.sage=sage;  
  }  
  Student.prototype.intr=function(){  
    console.log(  
      `I'm ${this.sname}, I'm ${this.sage}` );  
  }  
}
```

//创建一个学生类型的对象

```
var lilei=new Student("Li Lei",11);  
console.log(lilei);  
lilei.intr();
```



# 如何定义class: 3句话:

- i. 用class{}包裹原构造函数+原型对象方法
- ii. 原构造函数名升级为整个class的名字, 所有构造函数统一更名为"constructor"

//想定义学生类型，描述所有学生的统一结构和功能

```
class {  
  function Student(sname, sage){  
    this.sname=sname;  
    this.sage=sage;  
  }  
  Student.prototype.intr=function(){  
    console.log(  
      `I'm ${this.sname}, I'm ${this.sage}` );  
  }  
}
```

//创建一个学生类型的对象

```
var lilei=new Student("Li Lei",11);  
console.log(lilei);  
lilei.intr();
```



//想定义学生类型，描述所有学生的统一结构和功能

```
class Student{  
  function (sname, sage){  
    this.sname=sname;  
    this.sage=sage;  
  }  
  Student.prototype.intr=function(){  
    console.log(  
      `I'm ${this.sname}, I'm ${this.sage}` );  
  }  
}
```

//创建一个学生类型的对象

```
var lilei=new Student("Li Lei",11);  
console.log(lilei);  
lilei.intr();
```





//想定义学生类型，描述所有学生的统一结构和功能

```
class Student{
  constructor(sname, sage){
    this.sname=sname;
    this.sage=sage;
  }
  Student.prototype.intr=function(){
    console.log(
      `I'm ${this.sname}, I'm ${this.sage}` );
  }
}
```

//创建一个学生类型的对象

```
var lilei=new Student("Li Lei",11);
console.log(lilei);
lilei.intr();
```

# 如何定义class: 3句话:

- i. 用class{}包裹原构造函数+原型对象方法
- ii. 原构造函数名升级为整个class的名字, 所有构造函数统一更名为"constructor"
- iii. 原型对象中的方法, 不用再加prototype前缀, 也不用=function, 直接简写为: **方法名(){ ... ...}**

直接放在class{}内的方法定义, 其实还是保存在原型对象中的。



//想定义学生类型，描述所有学生的统一结构和功能

```
class Student{
  constructor(sname, sage){
    this.sname=sname;
    this.sage=sage;
  }
  Student.prototype.intr=function(){
    console.log(
      `I'm ${this.sname}, I'm ${this.sage}` );
  }
}
```

//创建一个学生类型的对象

```
var lilei=new Student("Li Lei",11);
console.log(lilei);
lilei.intr();
```



//想定义学生类型，描述所有学生的统一结构和功能

```
class Student{
  constructor(sname, sage){
    this.sname=sname;
    this.sage=sage;
  }
  intr=function(){
    console.log(
      `I'm ${this.sname}, I'm ${this.sage}` );
  }
}
```

//创建一个学生类型的对象

```
var lilei=new Student("Li Lei",11);
console.log(lilei);
lilei.intr();
```



//想定义学生类型，描述所有学生的统一结构和功能

```
class Student{  
  constructor(sname, sage){  
    this.sname=sname;  
    this.sage=sage;  
  }  
  intr(){  
    console.log(  
      `I'm ${this.sname}, I'm ${this.sage}` );  
  }  
}
```

直接放在class{}内的方法定义，其实还是保存在原型对象中的。

//创建一个学生类型的对象

```
var lilei=new Student("Li Lei",11);  
console.log(lilei);  
lilei.intr();
```



# 如何使用class:

- 和使用旧的构造函数完全一样:
- `var 对象名=new class名(属性值,...);`

▼ *Student* {*sname*: "Li Lei", *sage*: 11}

*sage*: 11

子对象

*sname*: "Li Lei"

▼ *[[Prototype]]*: Object 父对象原型对象

▶ *constructor*: *class Student*

▶ *intr*: *f intr()*

▶ *[[Prototype]]*: Object

**本质:新瓶装旧酒, 换汤不换药:**



# JS中创建对象10种方式总结

- 10. 稳妥构造函数：闭包，不用this，不用new！安全，可靠。

```
function Person(name, age) {  
  var p={};  
  p.getName=function(){ return name };  
  p.setName=function(value){ name=value };  
  p.getAge=function(){ return age }  
  return p;  
}
```

```
var p1 = Student("Li Lei",11);  
var p2 = Student("Han Meimei",12);  
console.log(p1.getName(), p1.getAge());  
console.log(lilei);  
console.log(hmm);
```

- 缺点：使用了闭包，容易造成内存泄漏。



# Q4: 继承

——Js中实现继承共有几种方式



# JS中7种实现继承方式:

- 1. 原型链式继承: 将父类的实例作为子类的原型

// 定义一个父类型

```
function Animal (name) {  
  this.name = name;  
  this.say = function() { console.log(I'm this.name); }  
}
```

// 原型对象方法

```
Animal.prototype.eat = function(food) {  
  console.log(this.name + '正在吃: ' + food);  
};
```

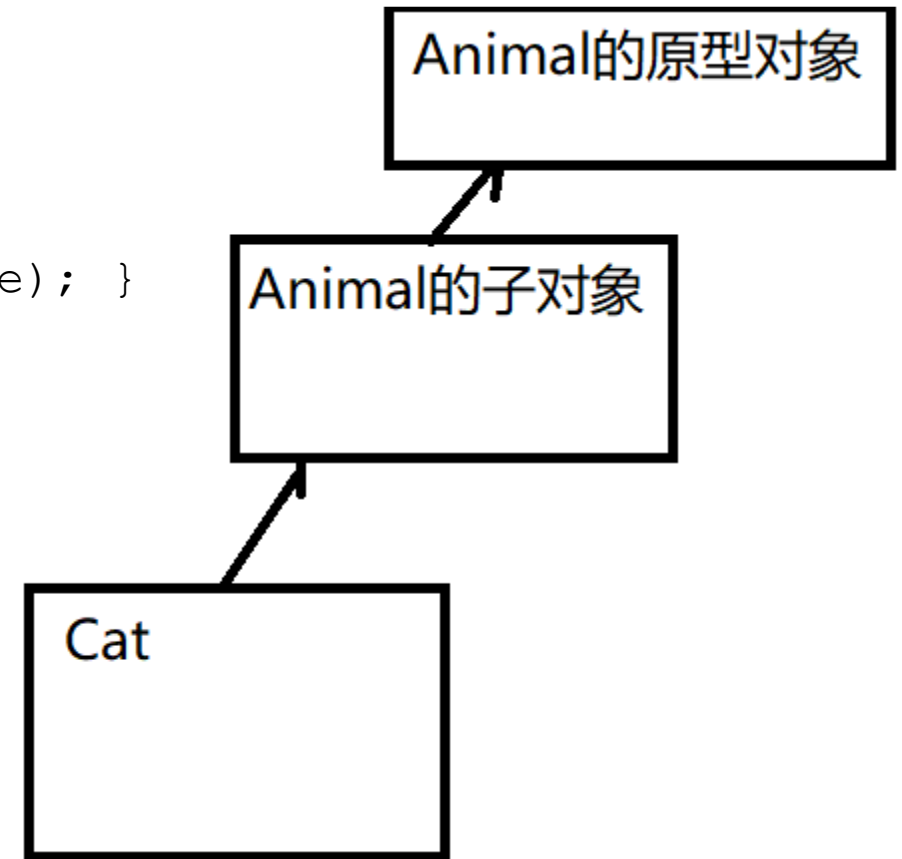
```
function Cat() { }
```

```
Cat.prototype = new Animal();
```

```
Cat.prototype.name = 'cat';
```

```
var cat = new Cat();
```

- 缺点: 创建子类实例时, 无法向父类构造函数传参





# JS中7种实现继承方式:

- 2. 构造函数继承:

// 定义一个父类型

```
function Animal (name) {  
  this.name = name;  
  this.say = function() { console.log(I'm this.name); }  
}
```

// 原型对象方法

```
Animal.prototype.eat = function(food) {  
  console.log(this.name + '吃: ' + food);  
};
```

```
function Cat(name,age){  
  Animal.call(this,name);  
  this.age = age;  
}
```

```
var cat = new Cat();
```





# JS中7种实现继承方式:

- 3. 实例继承:

// 定义一个父类型

```
function Animal (name) {  
  this.name = name;  
  this.say = function() { console.log(I'm this.name); }  
}
```

// 原型对象方法

```
Animal.prototype.eat = function(food) {  
  console.log(this.name + '吃: ' + food);  
};
```

```
function Cat(name,age){  
  var o = new Animal(name); //先创建子类型实例  
  o.age = age;  
  return o;                借鸡生蛋法  
}  
var cat = new Cat();
```



# JS中7种实现继承方式:

- 4. 拷贝继承: 无法获取父类不可for in遍历的方法

// 定义一个父类型

```
function Animal (name) {  
  this.name = name;  
  this.say = function() { console.log(I'm this.name); }  
}
```

// 原型对象方法

```
Animal.prototype.eat = function(food) {  
  console.log(this.name + '吃: ' + food);  
};
```

```
function Cat(name,age) {  
  var animal = new Animal(name);  
  for(var p in animal) {  
    Cat.prototype[p] = animal[p];  
  }  
  this.age = age  
}  
var cat = new Cat();
```



# JS中7种实现继承方式:

- 5. 组合继承:

// 定义一个父类型

```
function Animal (name) {  
  this.name = name;  
  this.say = function() { console.log(I'm this.name); }  
}
```

// 原型对象方法

```
Animal.prototype.eat = function(food) {  
  console.log(this.name + '吃: ' + food);  
};
```

```
function Cat(name, age) {  
  Animal.call(this, name);  
  this.age = age  
}
```

```
Cat.prototype = new Animal();  
Cat.prototype.constructor = Cat;  
var cat = new Cat();
```



# JS中7种实现继承方式:

- 6. 寄生组合继承

// 定义一个父类型

```
function Animal (name) {  
  this.name = name;  
  this.say = function() { console.log(I'm this.name); }  
}
```

// 原型对象方法

```
Animal.prototype.eat = function(food) {  
  console.log(this.name + '吃: ' + food);  
};
```

```
function Cat(name, age) { Animal.call(this, name); this.age = age }  
(function() { // 创建一个没有实例方法的类  
  var Super = function() {};  
  Super.prototype = Animal.prototype; //将实例作为子类的原型  
  Cat.prototype = new Super();  
}) ();  
var cat = new Cat();
```



# JS中7种实现继承方式：

- 7. ES6 class extends继承

```
Class 父类型{  
    constructor() {  
  
    }  
    ...  
}
```

```
Class 子类型 extends 父类型{  
    constructor() {  
        super();  
    }  
}
```



**Q5：深克隆**

**——实现深克隆共有几种方式**

封装

继承

多态

定义函数，克隆一个对象



错误做法:

```
var lilei:0x9091
```

封装

继承

多态

地址:0x9091

{...}



错误做法:

```
var lilei:0x9091
```

```
var lilei2=lilei
```

封装

继承

多态

地址:0x9091

{...}

错误做法:

封装

继承

多态

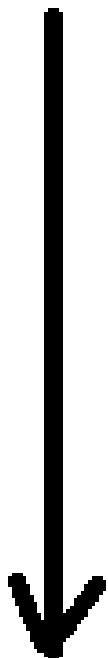
```
var lilei: 0x9091
```

地址: 0x9091

{...}

```
var lilei2 = lilei
```

0x9091 0x9091



```
var lilei2: 0x9091
```

错误做法:

封装

继承

多态

```
var lilei: 0x9091
```

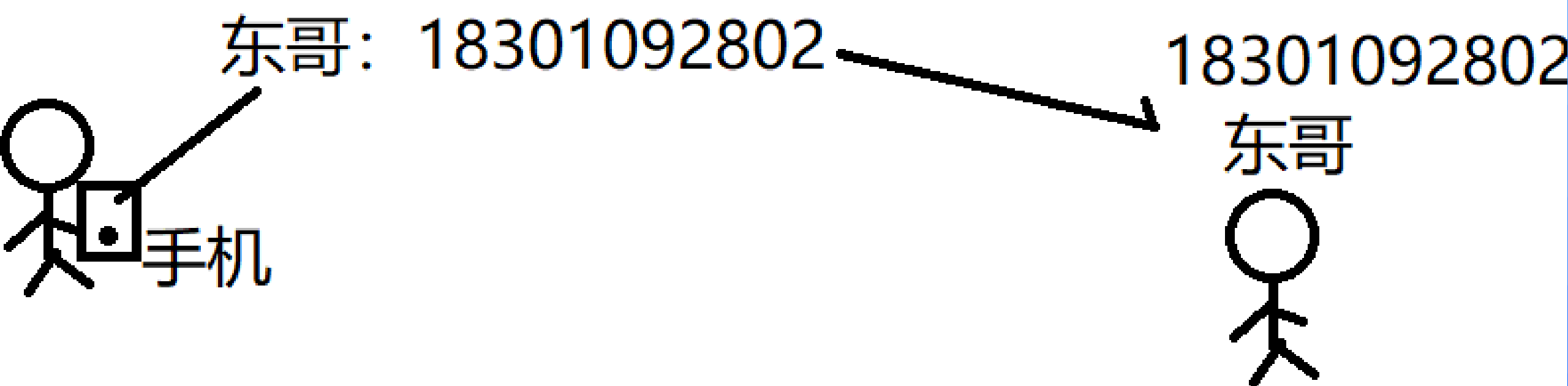
地址: 0x9091

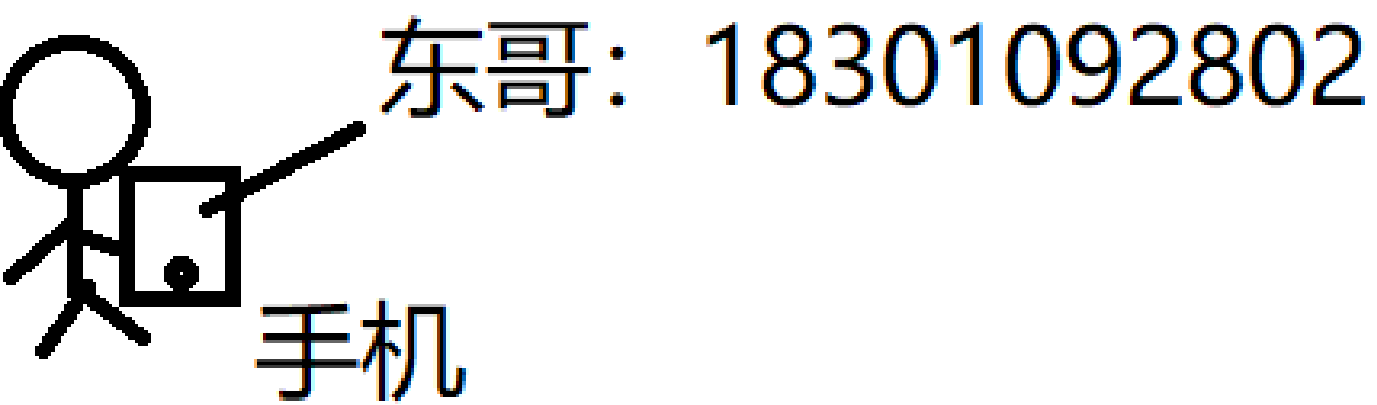
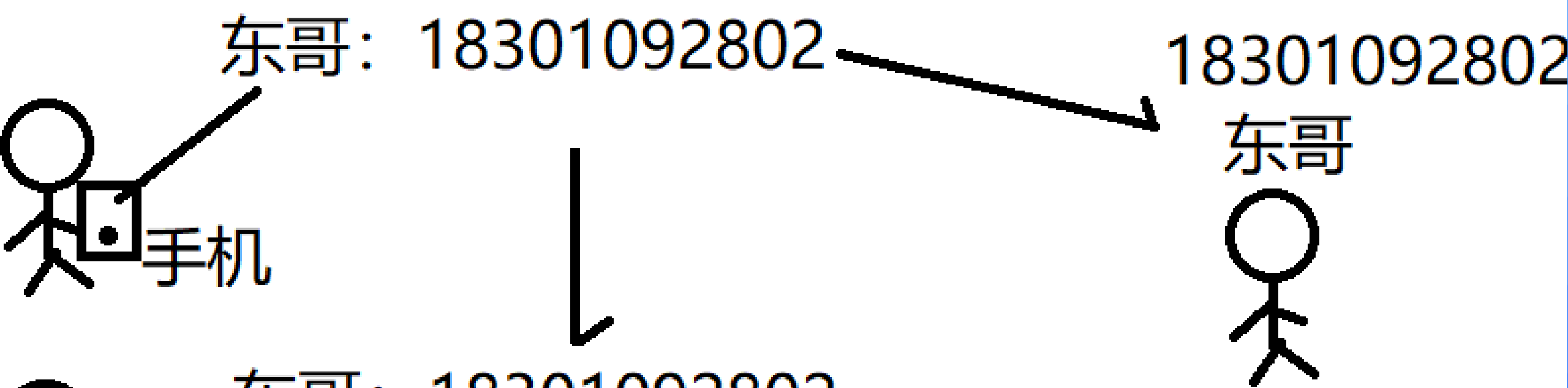
{...}

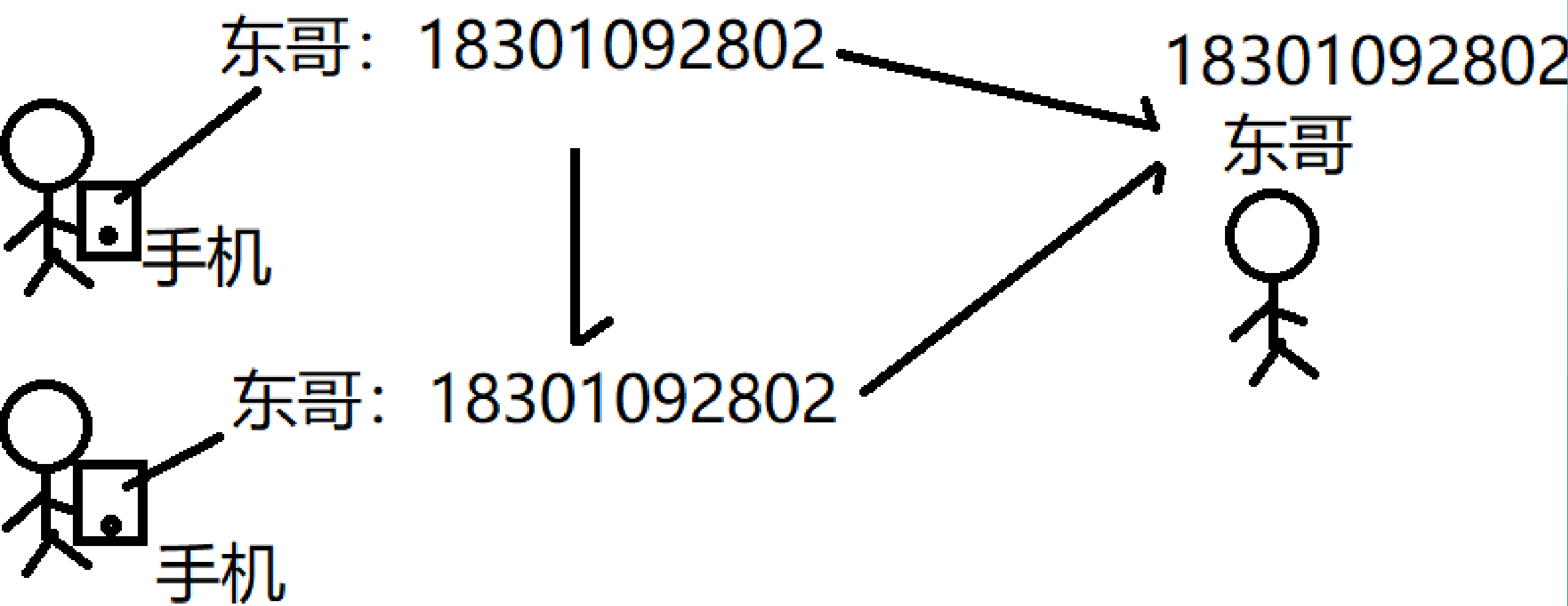
```
var lilei2 = lilei  
0x9091 0x9091
```

```
var lilei2: 0x9091
```

```
graph TD; A["var lilei: 0x9091"] --> B["地址: 0x9091<br/>{...}"]; A --> C["var lilei2: 0x9091"]; D["var lilei2 = lilei<br/>0x9091 0x9091"] --> A;
```







正确做法:

封装

继承

多态

1. 创建一个空对象等着

原对象



# 正确做法:

封装

继承

多态

## 1. 创建一个空对象等着

### 原对象



地址:0x1234





正确做法:

封装

继承

多态

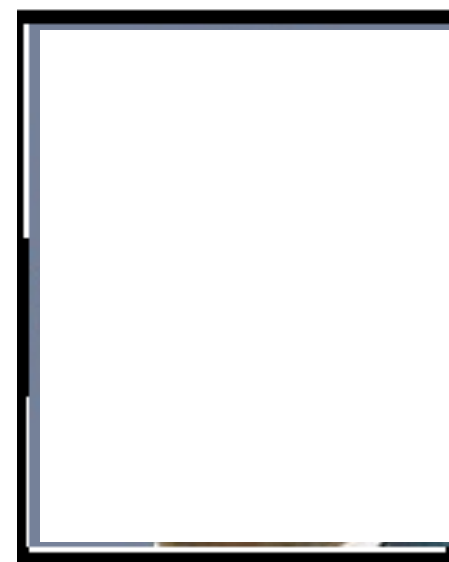
1. 创建一个空对象等着

2. 遍历原对象中所有属性

原对象



地址:0x1234



# 正确做法:

封装

继承

多态

1. 创建一个空对象等着

2. 遍历原对象中所有属性

3. 每遍历一个属性

就为新对象添加同名的新  
属性, 属性值与原属性也  
相同

地址: 0x1234



原对象



# 正确做法:

封装

继承

多态

1. 创建一个空对象等着

2. 遍历原对象中所有属性

3. 每遍历一个属性

就为新对象添加同名的新

属性 属性值与原属性也

**新对象的头发等于原对象的头发**

地址:0x1234

原对象



# 正确做法:

封装

继承

多态

1. 创建一个空对象等着

2. 遍历原对象中所有属性

3. 每遍历一个属性

就为新对象添加同名的新  
属性, 属性值与原属性也  
相同

**新对象的眼睛等于原对象的眼睛**

地址:0x1234

原对象



# 正确做法:

封装

继承

多态

1. 创建一个空对象等着

2. 遍历原对象中所有属性

3. 每遍历一个属性

就为新对象添加同名的新  
属性, 属性值与原属性也  
相同

地址: 0x1234

原对象



**新对象的鼻子等于原对象的鼻子**

# 正确做法:

封装

继承

多态

1. 创建一个空对象等着

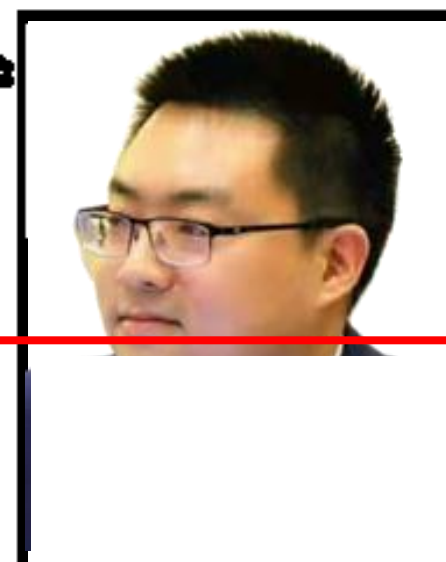
2. 遍历原对象中所有属性

3. 每遍历一个属性

就为新对象添加同名的新  
属性, 属性值与原属性也  
相同

地址: 0x1234

原对象



新对象的嘴巴等于原对象的嘴巴



# 正确做法:

封装

继承

多态

1. 创建一个空对象等着

2. 遍历原对象中所有属性

3. 每遍历一个属性

就为新对象添加同名的新  
属性, 属性值与原属性也  
相同

地址:0x1234

原对象



新对象的下巴等于原对象的下巴

# 正确做法:

封装

继承

多态

1. 创建一个空对象等着

2. 遍历原对象中所有属性

3. 每遍历一个属性

就为新对象添加同名的新  
属性, 属性值与原属性也  
相同

地址: 0x1234

原对象



4. 返回克隆好的新对象





```
function clone(原对象){  
    var 新对象={}  
    for(var 原属性名 in 原对象){  
  
        var 原属性值=原对象[原属性名];  
  
        新对象[原属性名]=原属性值;  
  
    }  
    return 新对象;  
}
```

原对象

0x9091

```
var dong={  
    头发: 浓密,  
    眼睛: 戴眼镜,  
    鼻子: 大鼻头,  
    嘴巴: 嘴角上翘,  
    下巴: 圆润  
}
```



```
clone(dong)
原对象
0x9091
var dong={
  头发: 浓密,
  眼睛: 戴眼镜,
  鼻子: 大鼻头,
  嘴巴: 嘴角上翘,
  下巴: 圆润
}
```

```
function clone(原对象){
  var 新对象={}
  for(var 原属性名 in 原对象){

    var 原属性值=原对象[原属性名];

    新对象[原属性名]=原属性值;

  }
  return 新对象;
}
```



0x9091

```
function clone(原对象){  
    var 新对象={}  
    for(var 原属性名 in 原对象){
```

```
        var 原属性值=原对象[原属性名];
```

```
        新对象[原属性名]=原属性值;
```

```
    }  
    return 新对象;
```

```
}
```

clone(dong)

0x9091

```
var dong={  
    头发: 浓密,  
    眼睛: 戴眼镜,  
    鼻子: 大鼻头,  
    嘴巴: 嘴角上翘,  
    下巴: 圆润  
}
```



0x9091

```
function clone(原对象){
```

```
  var 新对象={}
```

```
  for(var 原属性名 in 原对象){
```

```
    var 原属性值=原对象[原属性名];
```

```
    新对象[原属性名]=原属性值;
```

```
  }
```

```
  return 新对象;
```

```
}
```

clone(dong)

0x9091

```
var dong={  
  头发: 浓密,  
  眼睛: 戴眼镜,  
  鼻子: 大鼻头,  
  嘴巴: 嘴角上翘,  
  下巴: 圆润  
}
```



0x9091

```
function clone(原对象){  
  var 新对象={}  
  for(var 原属性名 in 原对象){
```

clone(dong)

```
    var 原属性值=原对象[原属性名];
```

```
    新对象[原属性名]=原属性值;
```

```
  }  
  return 新对象;  
}
```

0x9091

```
var dong={  
  头发: 浓密,  
  眼睛: 戴眼镜,  
  鼻子: 大鼻头,  
  嘴巴: 嘴角上翘,  
  下巴: 圆润  
}
```

0x1234

{

}



0x9091

```
function clone(原对象){  
  var 新对象={} 0x1234  
  for(var 原属性名 in 原对象){
```

clone(dong)

```
    var 原属性值=原对象[原属性名];
```

0x1234

```
    新对象[原属性名]=原属性值;
```

```
  }  
  return 新对象;  
}
```

0x9091

```
var dong={  
  头发: 浓密,  
  眼睛: 戴眼镜,  
  鼻子: 大鼻头,  
  嘴巴: 嘴角上翘,  
  下巴: 圆润  
}
```

{

}



0x9091

```
function clone(原对象){
```

```
  var 新对象={} 0x1234
```

```
  for(var 原属性名 in 原对象){
```

0x9091

```
    var 原属性值=原对象[原属性名];
```

```
    新对象[原属性名]=原属性值;
```

```
  }
```

```
  return 新对象;
```

```
}
```

clone(dong)

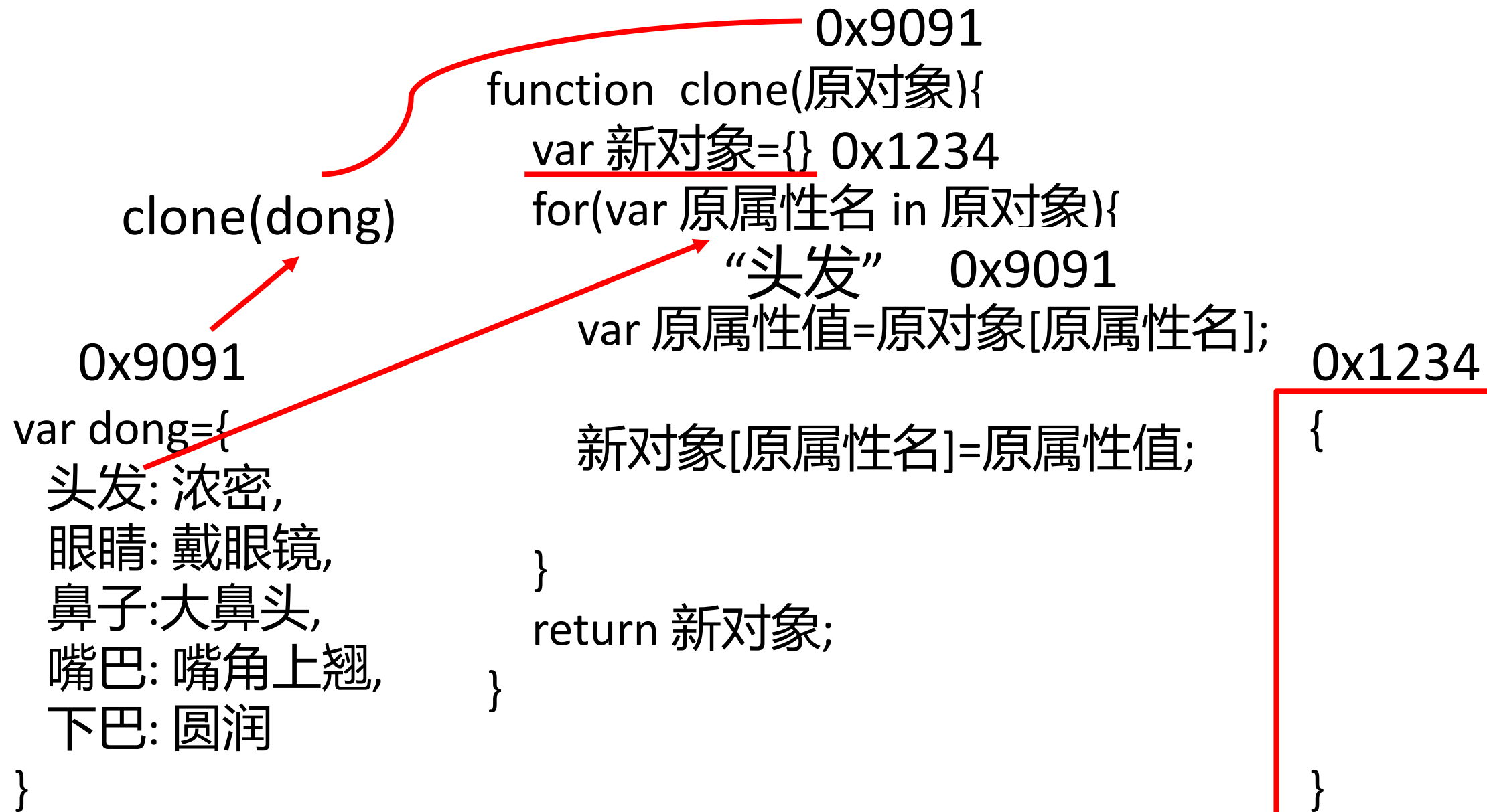
0x9091

```
var dong={  
  头发: 浓密,  
  眼睛: 戴眼镜,  
  鼻子: 大鼻头,  
  嘴巴: 嘴角上翘,  
  下巴: 圆润  
}
```

0x1234

```
{
```

```
}
```







0x9091

```
function clone(原对象){
```

var 新对象={} 0x1234

```
for(var 原属性名 in 原对象){
```

“头发” 0x9091

```
var 原属性值=原对象["头发"];
```

## 新对象[原属性名]=原属性值;

}

```
return 新对象;
```

}

# clone(dong)

0x9091

```
var dong={  
    头发: 浓密,  
    眼睛: 戴眼镜,  
    鼻子: 大鼻头,  
    嘴巴: 嘴角上翘,  
    下巴: 圆润  
}
```

0x1234

 $\{$ 

}



0x9091

```
function clone(原对象){
```

var 新对象={} 0x1234

```
for(var 原属性名 in 原对象){
```

“头发” 0x9091

```
var 原属性值=原对象["头发"];
```

# “浓密”

## 新对象[原属性名]=原属性值;

}

```
return 新对象;
```

}

# clone(dong)

0x9091

```
var dong={
    头发: 浓密,
    眼睛: 戴眼镜,
    鼻子: 大鼻头,
    嘴巴: 嘴角上翘,
    下巴: 圆润
}
```

0x1234

 $\{$ 

}



0x9091

```
function clone(原对象){
```

var 新对象={} 0x1234

```
for(var 原属性名 in 原对象){
```

“头发” 0x9091

```
var 原属性值=原对象["头发"];
```

“浓密” “浓密”

## 新对象[原属性名]=原属性值;

}

```
return 新对象;
```

}

# clone(dong)

0x9091

```
var dong={
    头发: 浓密,
    眼睛: 戴眼镜,
    鼻子: 大鼻头,
    嘴巴: 嘴角上翘,
    下巴: 圆润
}
```

0x1234

 $\{$ 

}



0x9091

```
function clone(原对象){
```

var 新对象={} 0x1234

```
for(var 原属性名 in 原对象){
```

“头发” 0x9091

```
var 原属性值=原对象["头发"];
```

“浓密” “浓密”

新对象[原属性名]=原属性值;

# “浓密”

}

```
return 新对象;
```

}

# clone(dong)

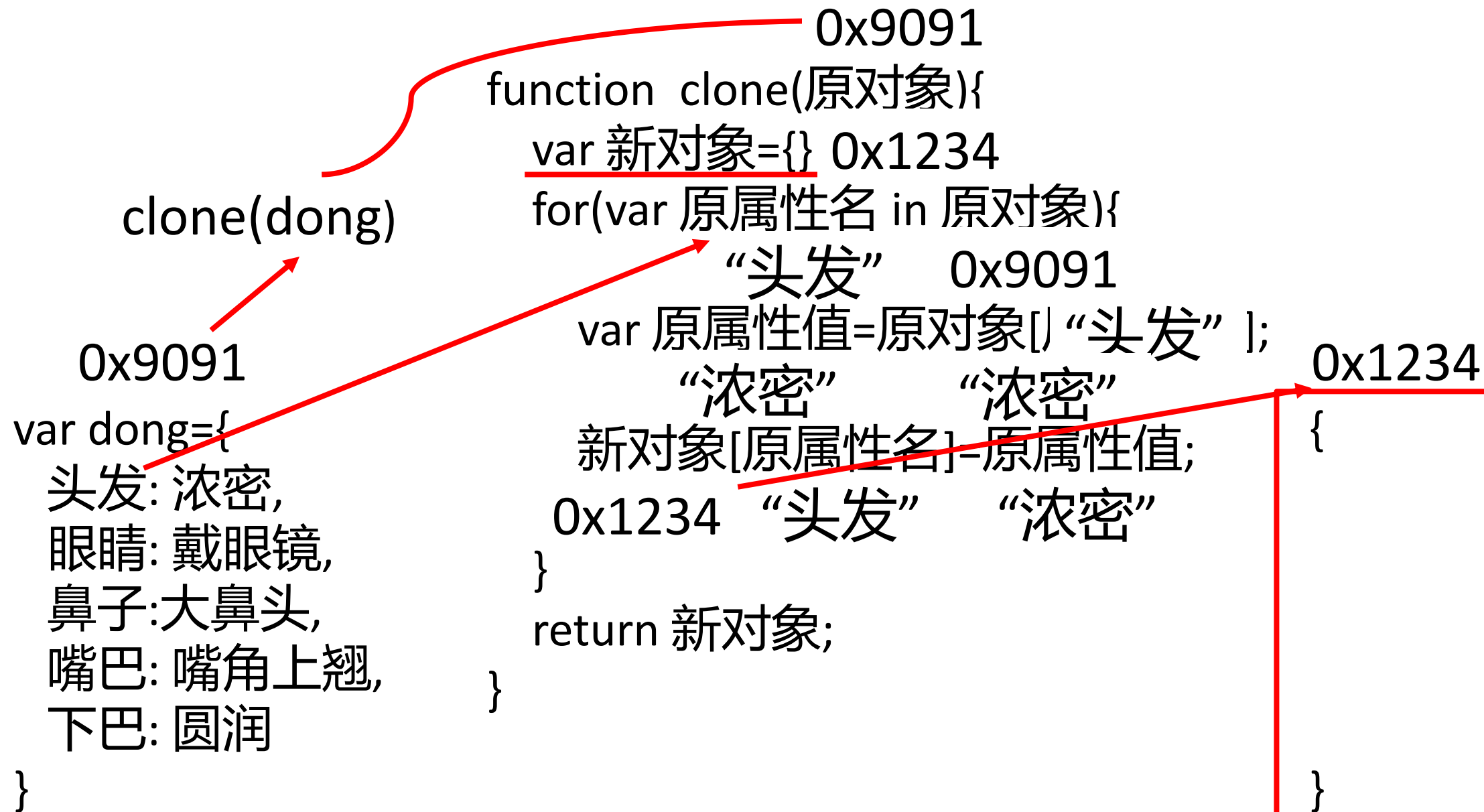
0x9091

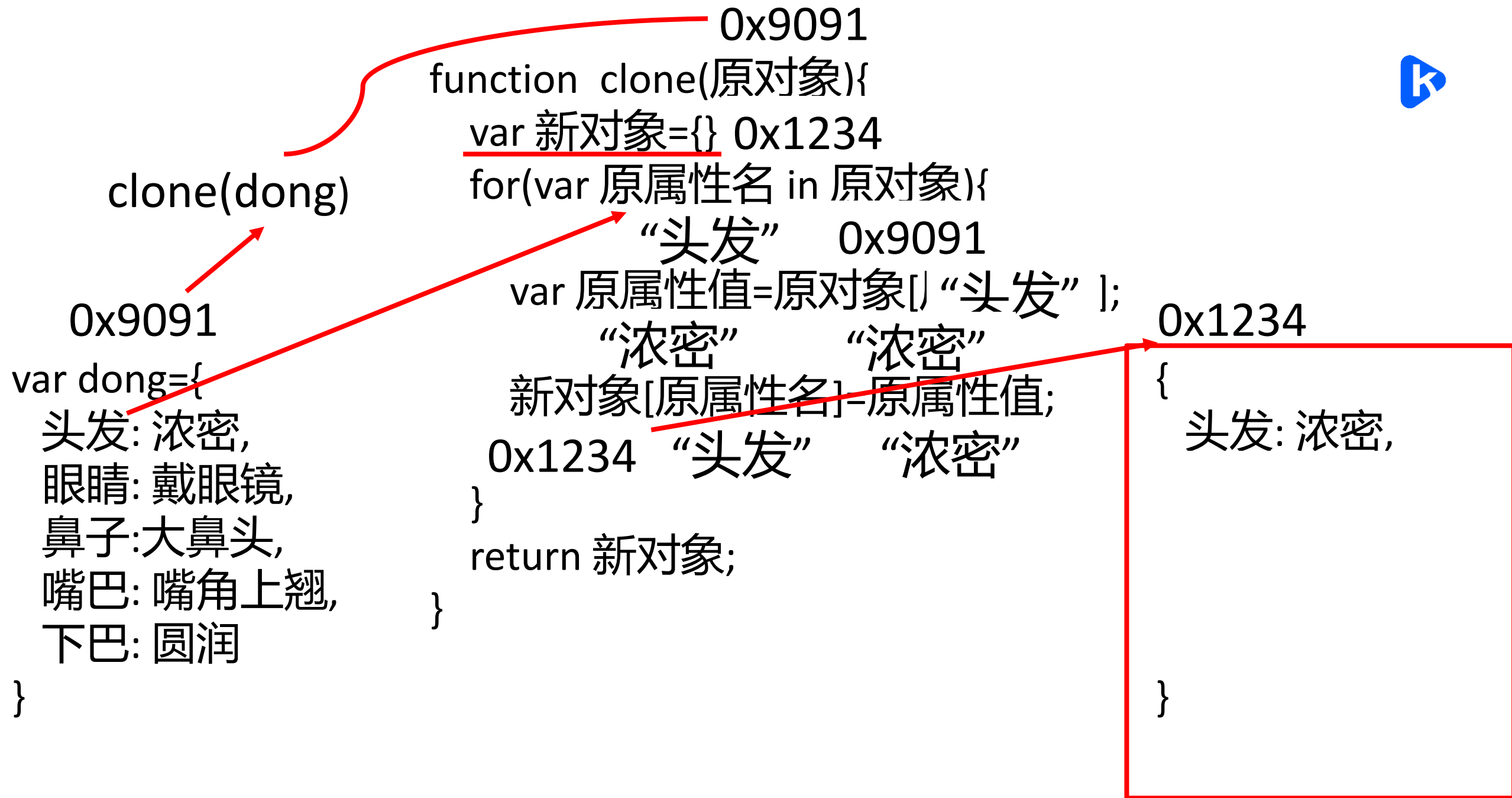
```
var dong={
    头发: 浓密,
    眼睛: 戴眼镜,
    鼻子: 大鼻头,
    嘴巴: 嘴角上翘,
    下巴: 圆润
}
```

0x1234

 $\{$ 

}







0x9091

```
function clone(原对象){
```

```
  var 新对象={} 0x1234
```

```
  for(var 原属性名 in 原对象){
```

```
    “眼睛” 0x9091
```

```
    var 原属性值=原对象[原属性名];
```

```
    新对象[原属性名]=原属性值;
```

```
  }
```

```
  return 新对象;
```

```
}
```

clone(dong)

0x9091

```
var dong={  
  头发: 浓密,  
  眼睛: 戴眼镜,  
  鼻子: 大鼻头,  
  嘴巴: 嘴角上翘,  
  下巴: 圆润  
}
```

0x1234

```
{  
  头发: 浓密,  
  
  
  
  
  
  
  
  
}
```



0x9091

```
function clone(原对象){
```

var 新对象={} 0x1234

```
for(var 原属性名 in 原对象){
```

“眼睛” 0x9091

```
var 原属性值=原对象["眼睛"];
```

## 新对象[原属性名]=原属性值;

}

```
return 新对象;
```

}

# clone(dong)

0x9091

```
var dong={
    头发: 浓密,
    眼睛: 戴眼镜,
    鼻子: 大鼻头,
    嘴巴: 嘴角上翘,
    下巴: 圆润
}
```

0x1234

{  
头发: 浓密,  
}





```
function clone(原对象){  
    var 新对象={} 0x1234  
    for(var 原属性名 in 原对象){  
        “眼睛” 0x9091  
        var 原属性值=原对象[“眼睛”];  
        新对象[原属性名]=原属性值;  
    }  
    return 新对象;  
}
```

clone(dong)

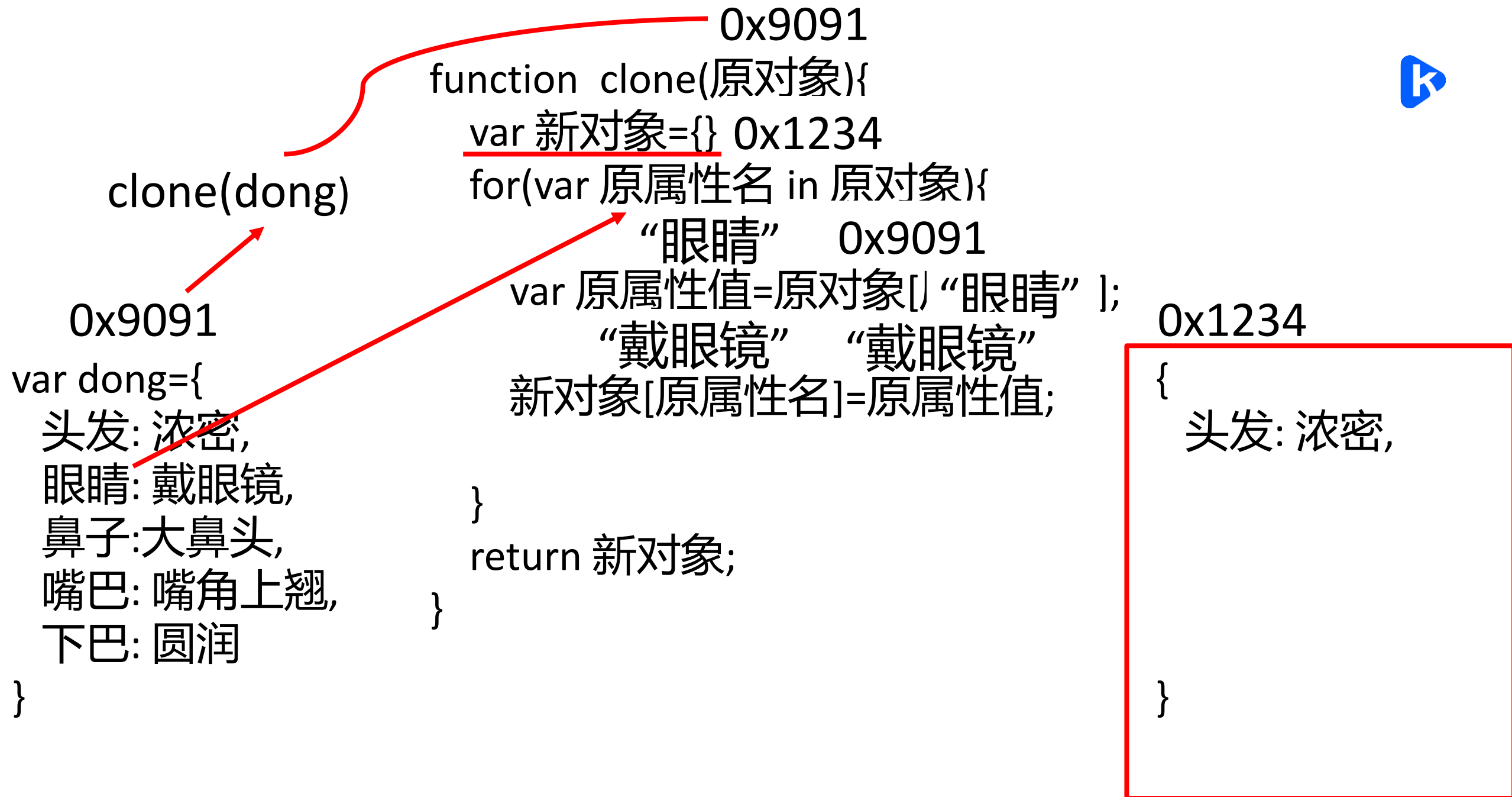
0x9091

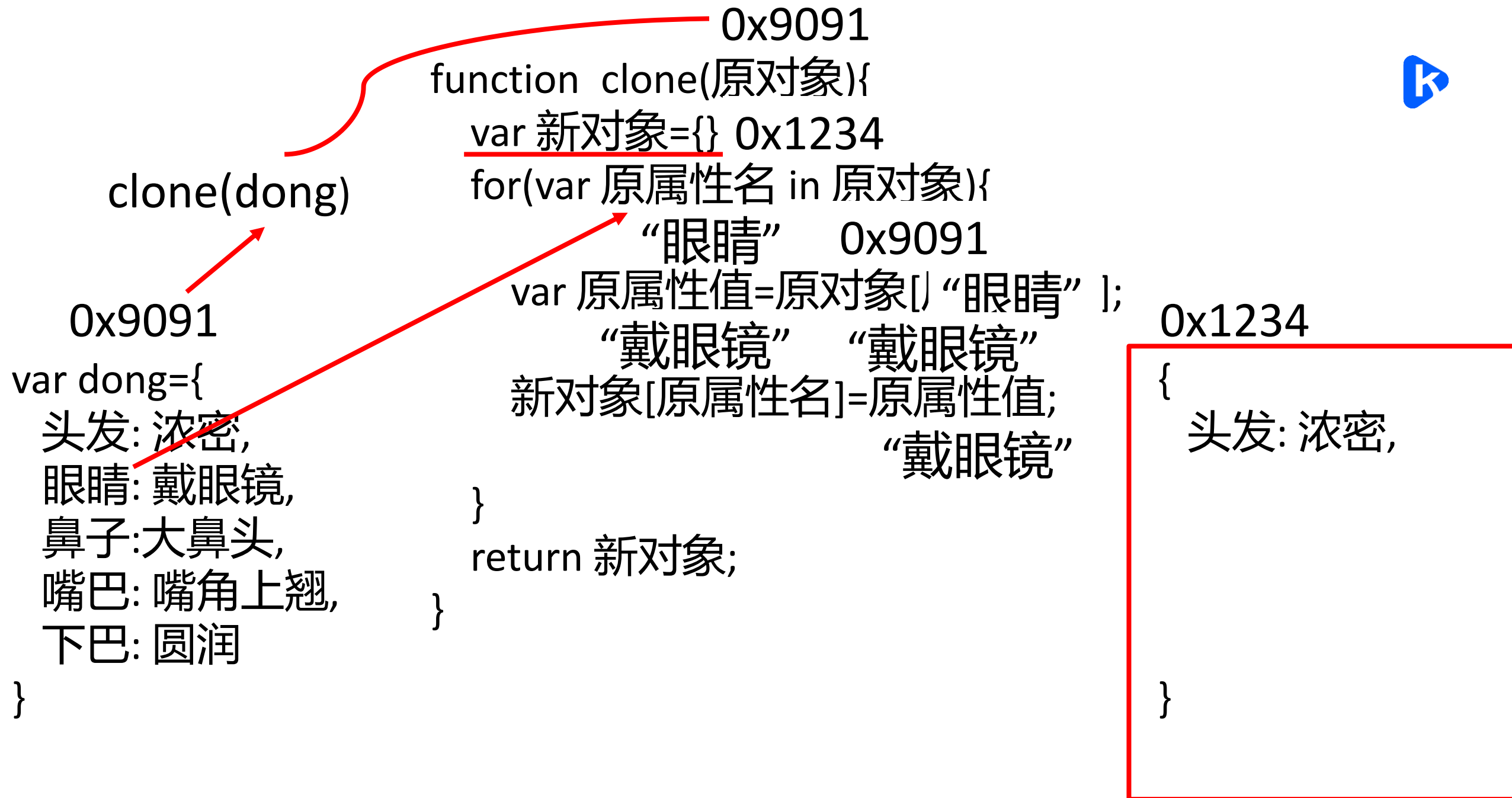
```
var dong={  
    头发: 浓密,  
    眼睛: 戴眼镜,  
    鼻子: 大鼻头,  
    嘴巴: 嘴角上翘,  
    下巴: 圆润  
}
```

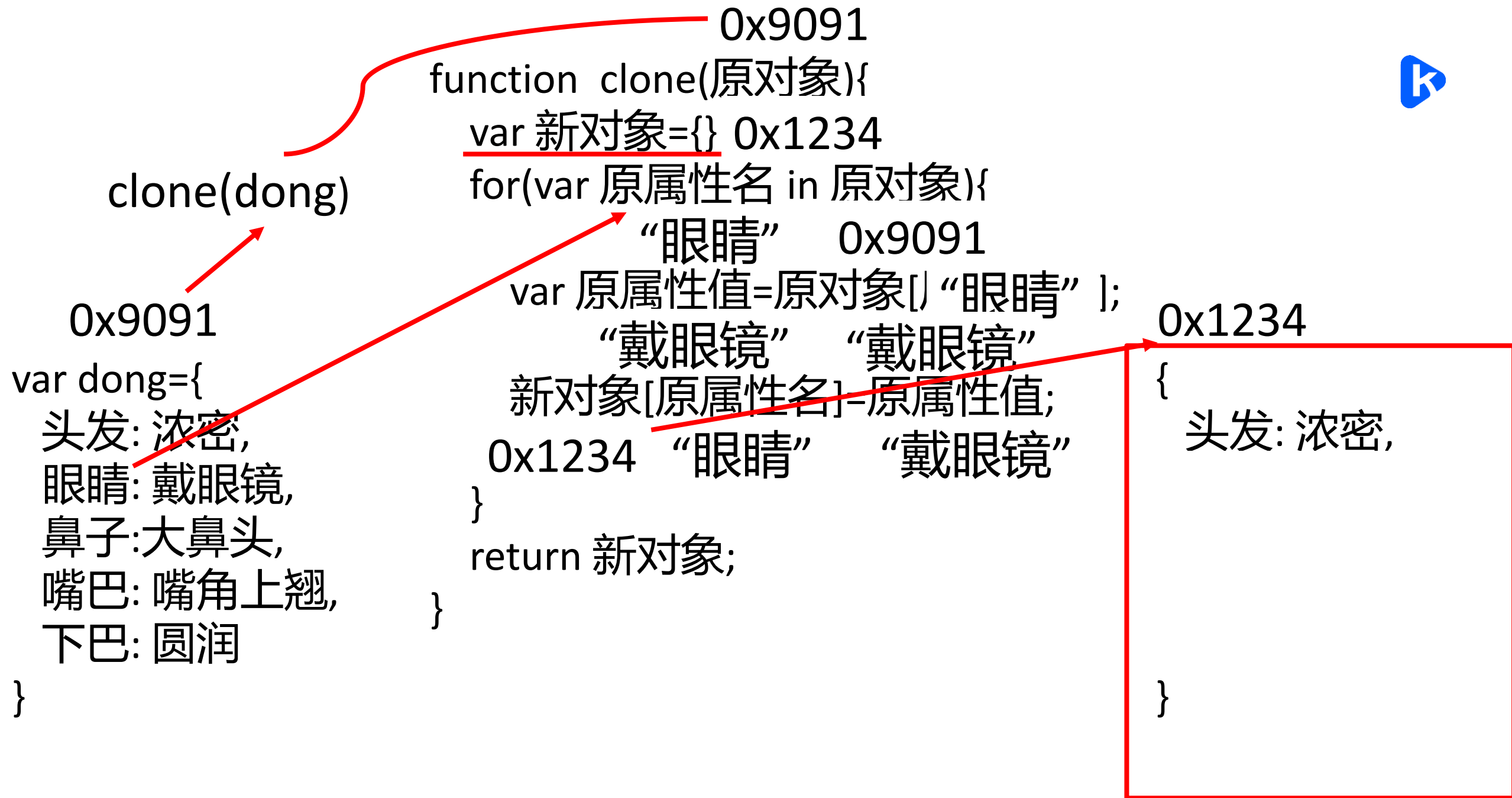
0x9091

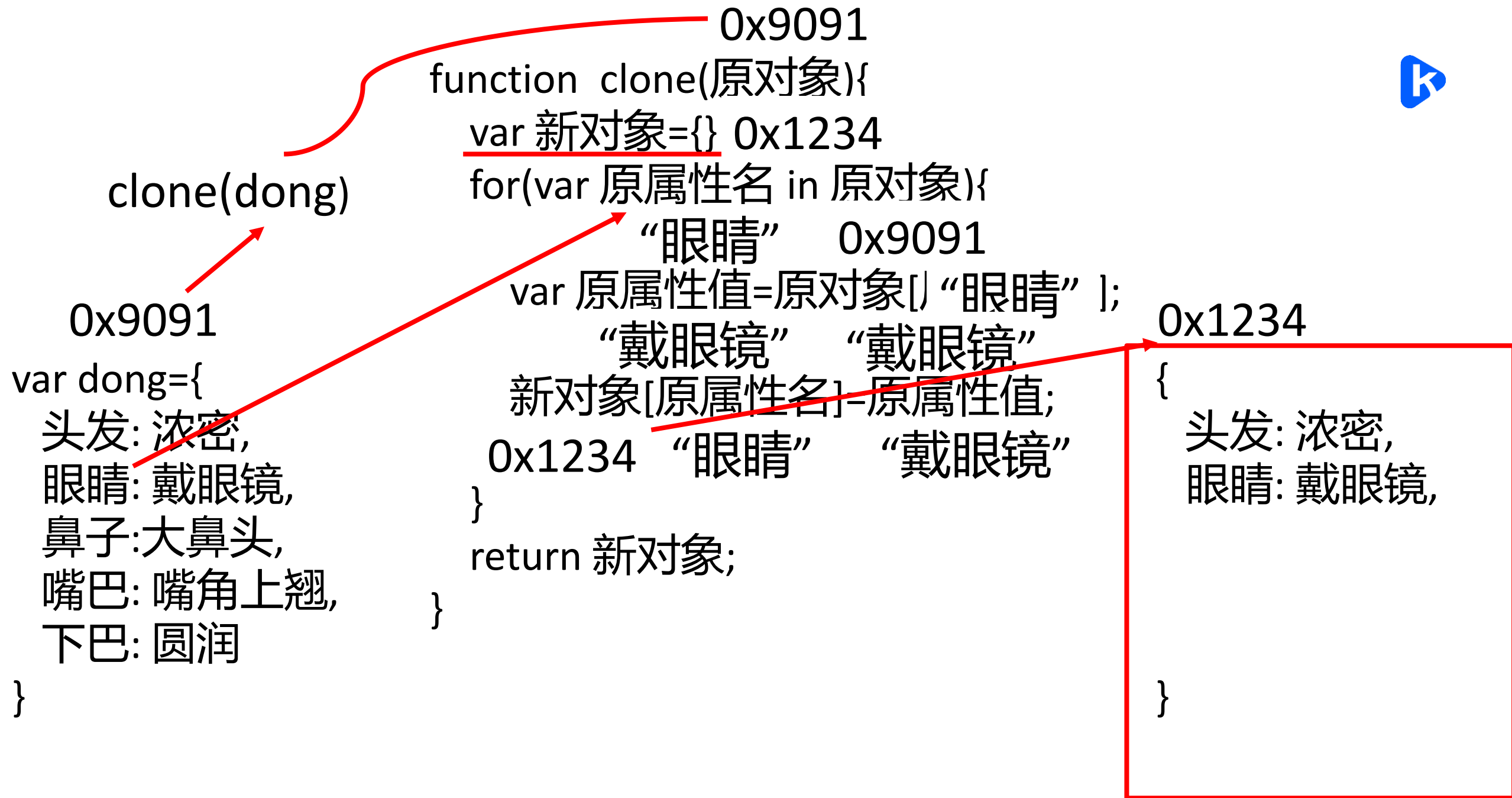
0x1234

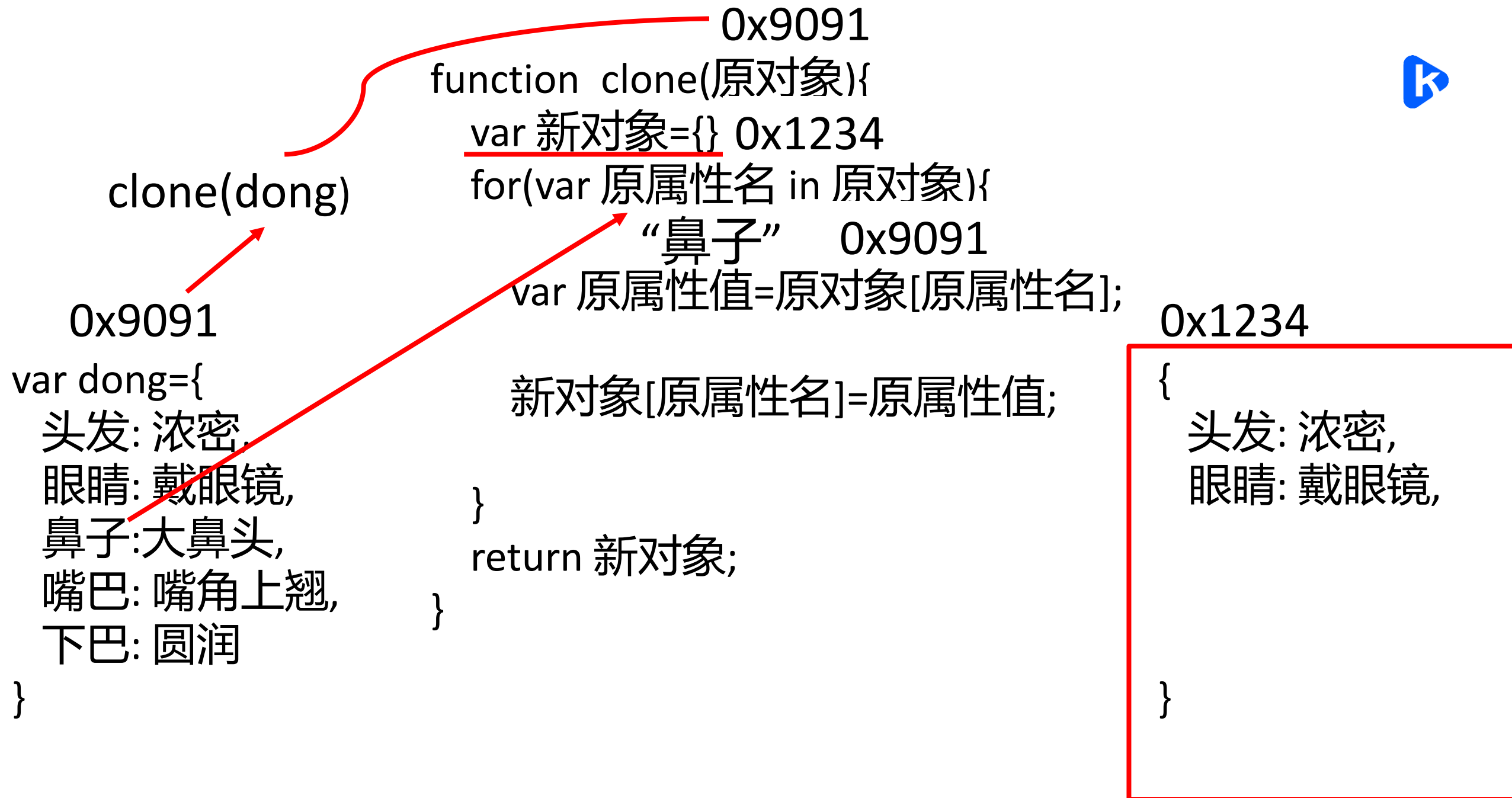
```
{  
    头发: 浓密,  
    ...  
}
```

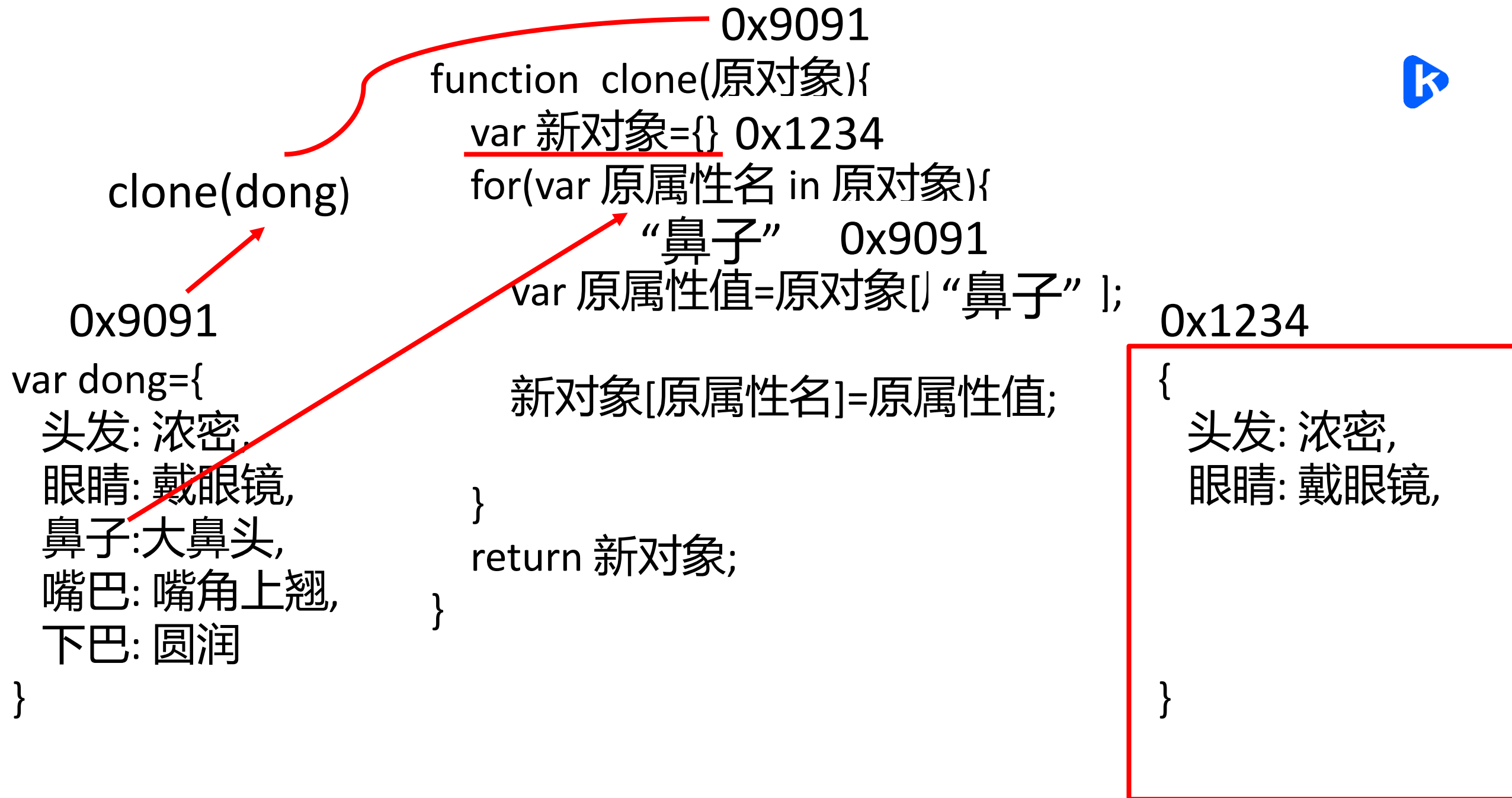


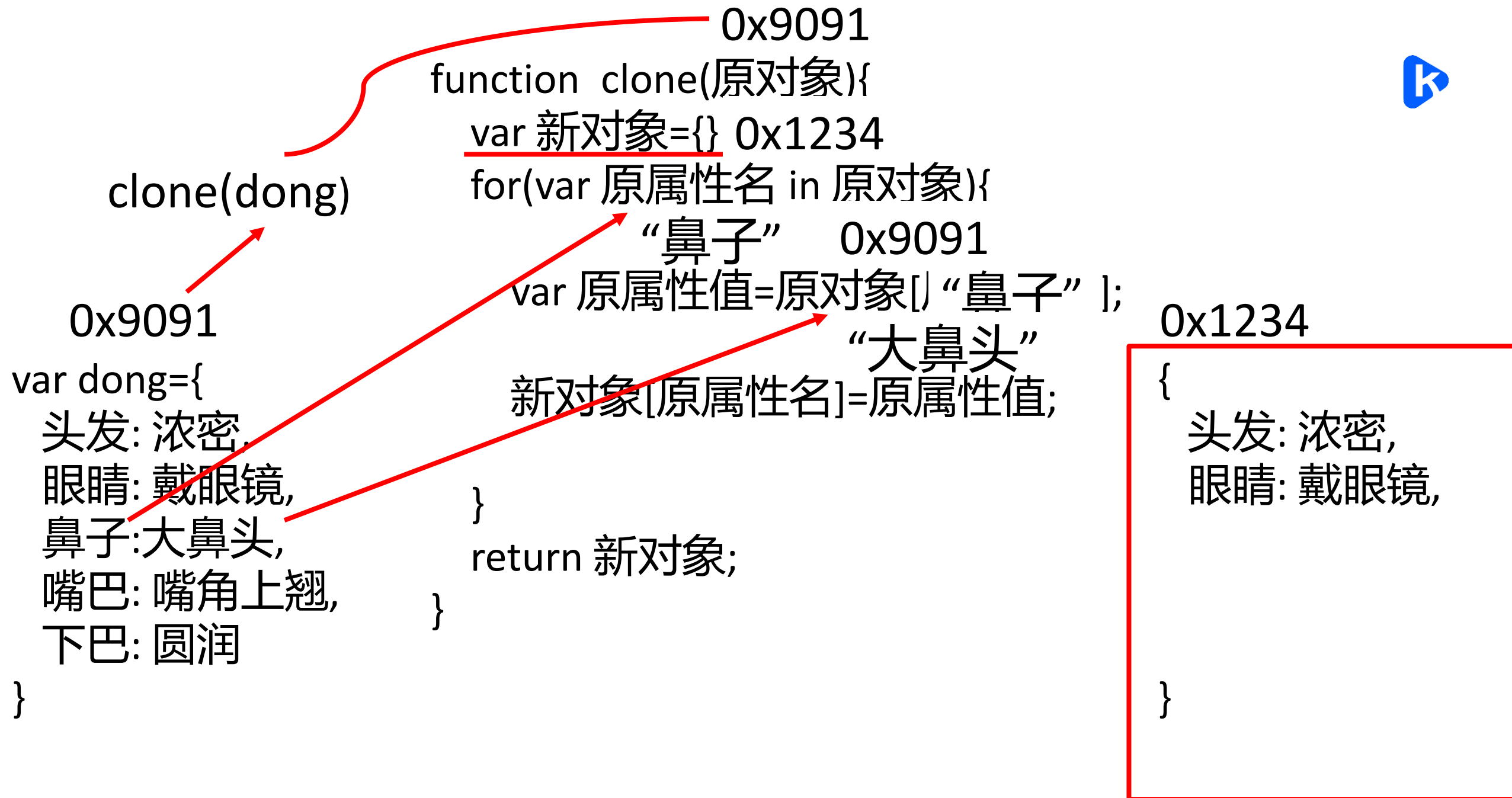




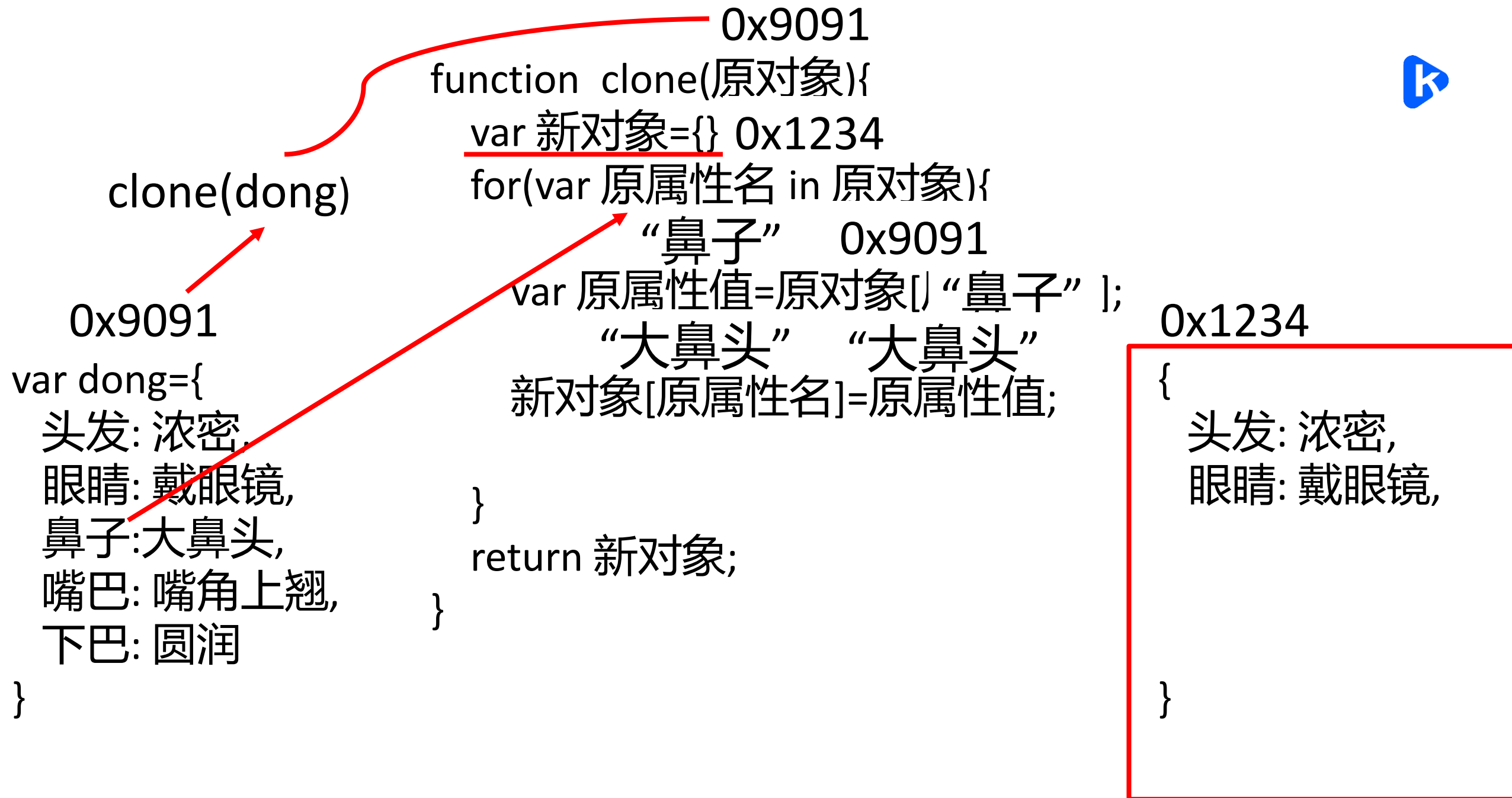


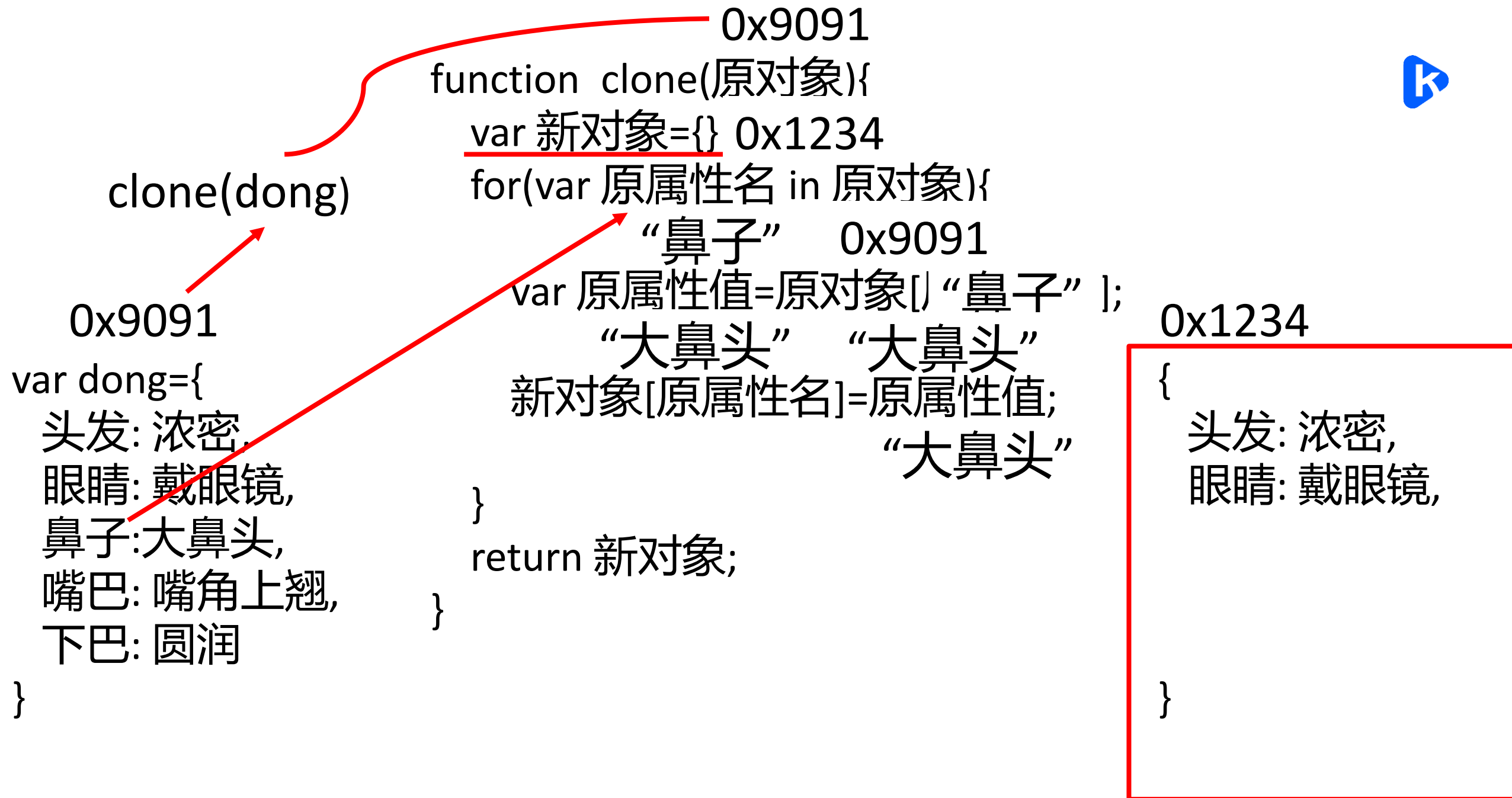


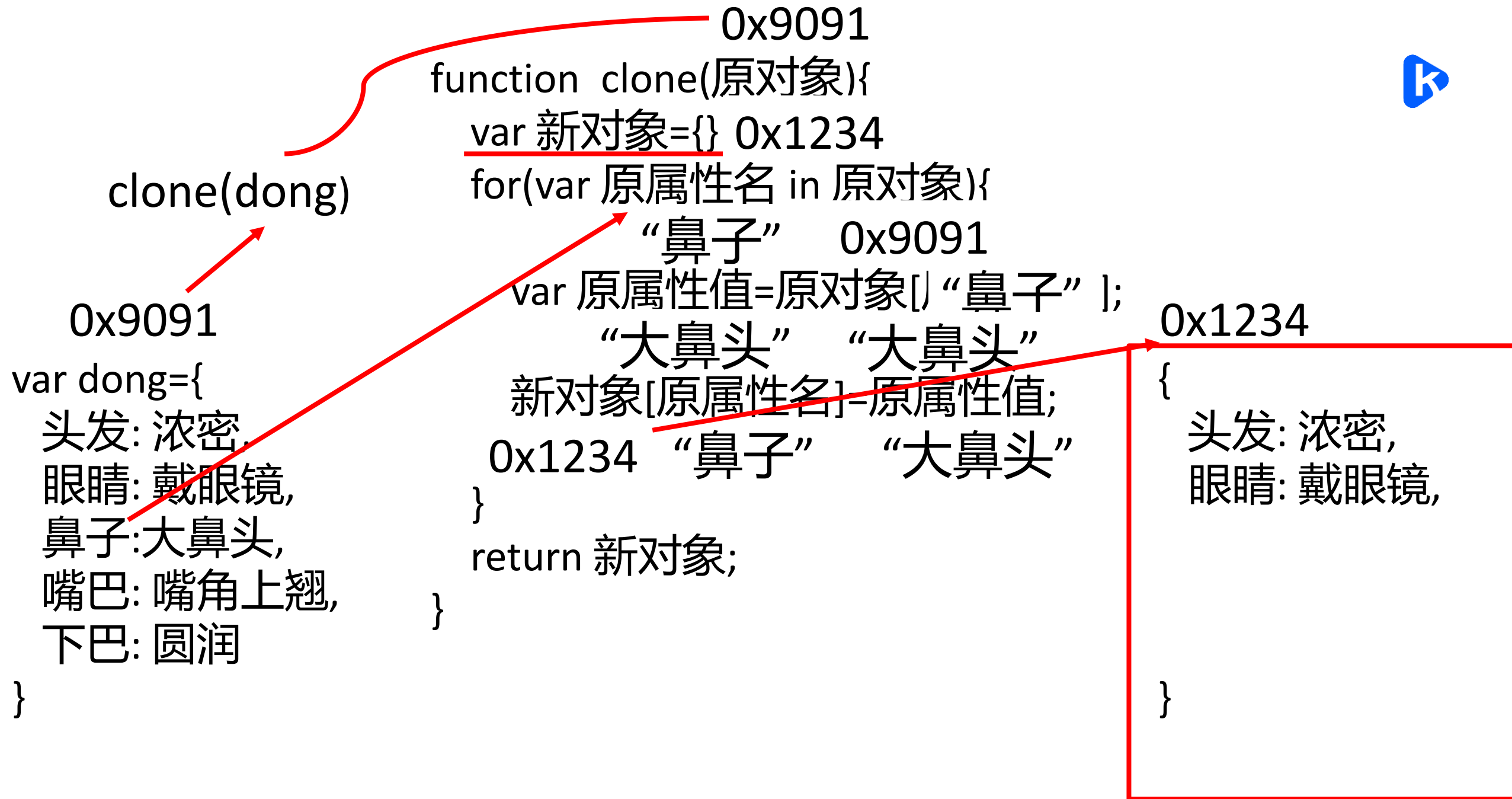


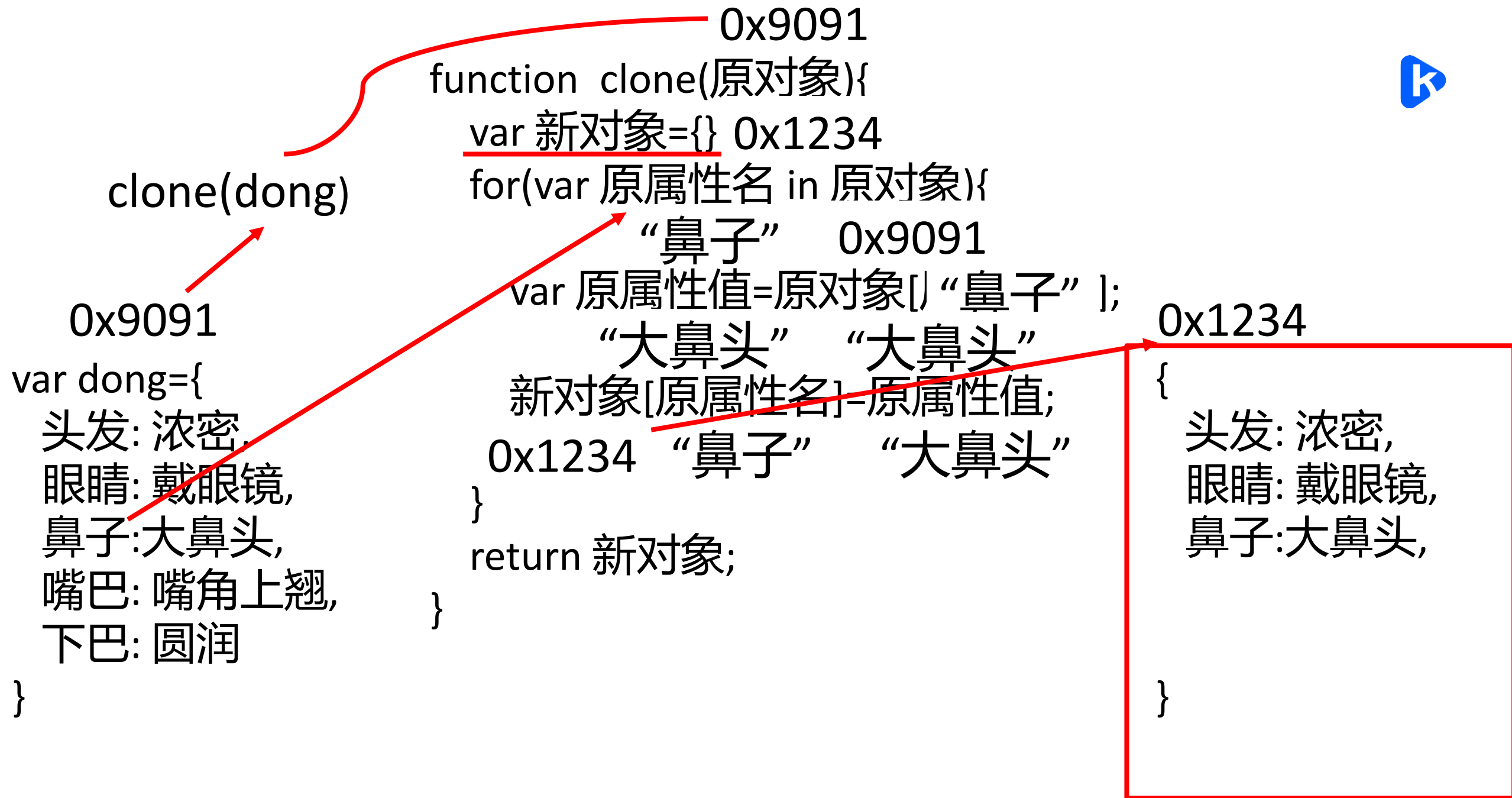


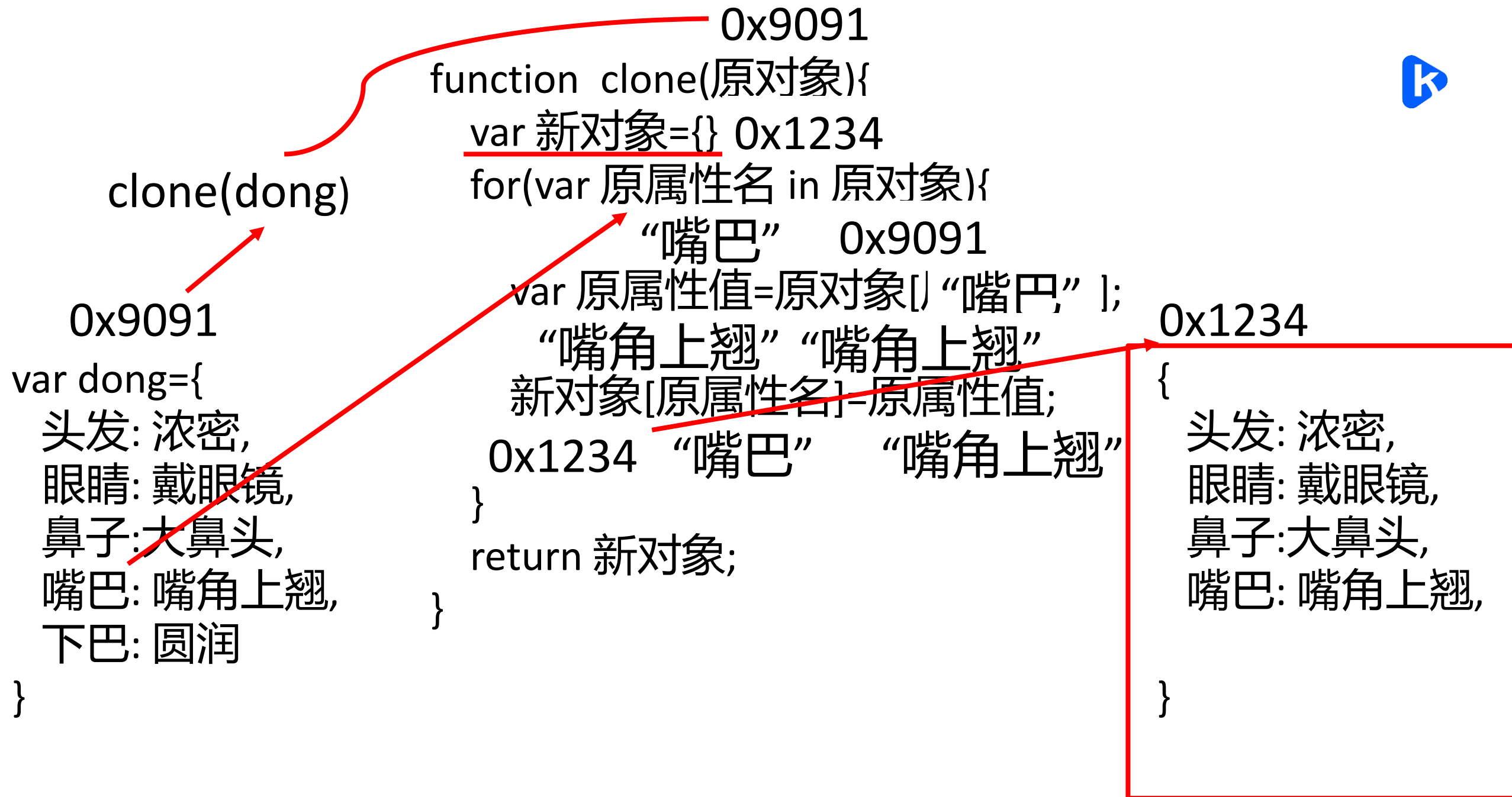


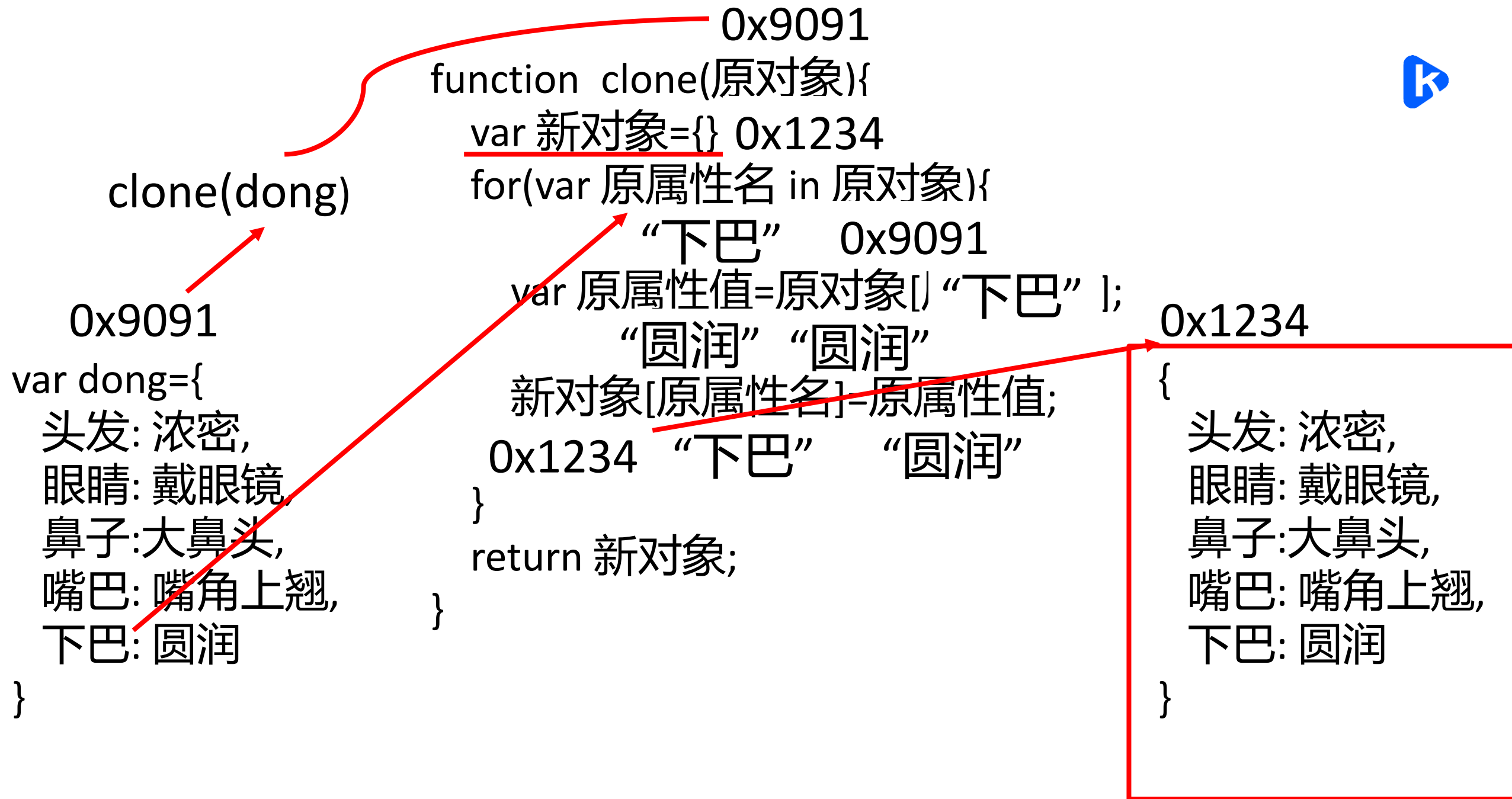












**但是**

**这种只是浅克隆  
如何深克隆？**



# 什么是：

- 浅克隆：只复制对象的第一级属性值。如果对象的第一级属性中又包含引用类型，则只复制地址
- 浅克隆的问题：如果对象中又包含引用类型的属性值，则导致克隆后，新旧对象依然共用同一个引用类型的对象属性值。
- 结果：任意一方修改了引用类型的对象内容，都会导致另一方同时受影响。
- 深克隆：不但复制对象的第一级属性值，而且，即使对象中又包含引用类型的属性值，深克隆也会继续复制内嵌类型的属性值。
- 结果：克隆后，两个对象彻底再无瓜葛。





# 实现克隆对象共有3种方法:

- 1. JSON.stringify()以及JSON.parse(), 无法深克隆undefined值和内嵌函数

```
var obj1 = { x: 1, y: 2, z: 3 }  
var str = JSON.stringify(obj1);  
var obj2 = JSON.parse(str);
```



# 实现克隆对象共有3种方法:

- 2. Object.assign(target, source)

```
var obj1 = {  
  x: 1,  
  y: 2,  
  z: 3  
}  
var obj2 = Object.assign({}, obj1);
```



# 实现克隆对象共有3种方法:

- 3. 自定义递归克隆函数:

```
function deepClone(target) {  
  let newObj; // 定义一个变量, 准备接新副本对象  
  // 如果当前需要深拷贝的是一个引用类型对象  
  if (typeof target === 'object') {  
    if (Array.isArray(target)) { // 如果是一个数组  
      newObj = []; // 将newObj赋值为一个数组, 并遍历  
      for (let i in target) { // 递归克隆数组中的每一项  
        newObj.push(deepClone(target[i]))  
      }  
      // 判断如果当前的值是null; 直接赋值为null  
    } else if (target === null) {  
      newObj = null;  
      // 判断如果当前的值是一个正则表达式对象, 直接赋值  
    } else if (target.constructor === RegExp) {  
      newObj = target;  
    } else {  
      // 否则是普通对象, 直接for in循环递归遍历复制对象中每个属性值  
      newObj = {};  
      for (let i in target) {  
        newObj[i] = deepClone(target[i]);  
      }  
    }  
    // 如果不是对象而是原始数据类型, 那么直接赋值  
  } else { newObj = target; }  
  // 返回最终结果 return newObj;  
}
```



# Q6: 其他面向对象笔试题

——作业: 读程序、判断输出结果、画图



**我们的口号是：  
早学早面试卷别人  
晚学晚面试被人卷**

**paikaba**  
开课吧