目录

1：什么是闭包 1

2：闭包的特点 2

3：闭包的原理 2

4：闭包的应用场景 3

5：使用闭包注意点 3

应用： 3

6：构造函数继承 3

7：call/apply继承 5

8：prototype的概念 6

9：原型链的继承 8

10：混合继承 9

应用： 10

综合应用： 10

## 1：什么是闭包

闭包就是能够读取其他函数内部变量的函数。在JS中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解为”定义在一个函数内部的函数”。

## 2：闭包的特点

1：可以读取函数内部的变量。

2：让这些变量的值始终保存在内存中。

## 3：闭包的原理

理解闭包，首先必须理解JS变量的作用域。变量的作用域无非就是两种：全局变量和局部变量。

JS语言的特殊之处，就在于函数内部可以直接读取全局变量。另一方面，函数外部自然无法读取函数内的局部变量。

**注意：**

1：函数内部声明变量的时候，一定要使用var声明。如果不用的话，你实际上声明了一个全局变量。

2：局部变量的作用域，在函数定义的时候就已经确定下来了。

出于各种原因，我们有时候需要得到函数内部的局部变量。但是正常情况下这是办不到的。只有变通一下才能实现，那就是在函数内部再定义一个函数。外部变量不能访问内部变量，内部变量却能访问外部变量，这正是因为JS特有的”链式作用域”结构(chain scope)，子对象会一级一级地向上寻找所有父对象的变量。所以父对象的所有变量，对子对象都是可见的，反之则不成立。我们只需要把子函数返回出来，我们就可以在外部读取内部变量了。

## 4：闭包的应用场景

1：函数作为返回值。

2：函数作为参数被传递。

## 5：使用闭包注意点

1：由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包。否则会造成网页性能问题，在IE中可能导致内存泄漏。解决方法就是在函数退出之前，将不使用的局部变量删除。

2：闭包会在父函数外部，改变父函数内部变量的值。所以不要随便改变父函数内部变量的值。

闭包（closure）是Javascript语言的一个难点，也是它的特色，很多高级应用都要依靠闭包实现。

要理解闭包，首先必须理解变量作用域。前面提到，JavaScript有两种作用域：全局作用域和函数作用域。函数内部可以直接读取全局变量。

var n = 999;

function f1() {

console.log(n);}

f1() // 999

上面代码中，函数f1可以读取全局变量n。

但是，在函数外部无法读取函数内部声明的变量。

function f1() {

var n = 999;

}

console.log(n)// Uncaught ReferenceError: n is not defined(

上面代码中，函数f1内部声明的变量n，函数外是无法读取的。

如果出于种种原因，需要得到函数内的局部变量。正常情况下，这是办不到的，只有通过变通方法才能实现。那就是在函数的内部，再定义一个函数。

function f1() {

var n = 999;

function f2() {

console.log(n); // 999

}

}

上面代码中，函数f2就在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。但是反过来就不行，f2内部的局部变量，对f1就是不可见的。这就是JavaScript语言特有的”链式作用域”结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然f2可以读取f1的局部变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗！

function f1() {

var n = 999;

function f2() {

console.log(n);

}

return f2;}

var result =f1();

result(); // 999

上面代码中，函数f1的返回值就是函数f2，由于f2可以读取f1的内部变量，所以就可以在外部获得f1的内部变量了。

闭包就是函数f2，即能够读取其他函数内部变量的函数。由于在JavaScript语言中，只有函数内部的子函数才能读取内部变量，因此可以把闭包简单理解成“定义在一个函数内部的函数”。闭包最大的特点，就是它可以“记住”诞生的环境，比如f2记住了它诞生的环境f1，所以从f2可以得到f1的内部变量。在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

闭包的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量始终保持在内存中，即闭包可以使得它诞生环境一直存在。请看下面的例子，闭包使得内部变量记住上一次调用时的运算结果。

function createIncrementor(start) {

return function () {

return start++;

};}

var inc = createIncrementor(5);

inc() // 5

inc() // 6

inc() // 7

上面代码中，start是函数createIncrementor的内部变量。通过闭包，start的状态被保留了，每一次调用都是在上一次调用的基础上进行计算。从中可以看到，闭包inc使得函数createIncrementor的内部环境，一直存在。所以，闭包可以看作是函数内部作用域的一个接口。

为什么会这样呢？原因就在于inc始终在内存中，而inc的存在依赖于createIncrementor，因此也始终在内存中，不会在调用结束后，被垃圾回收机制回收。

闭包的另一个用处，是封装对象的私有属性和私有方法。

function Person(name) {

var \_age;

function setAge(n) {

\_age = n;

}

function getAge() {

return \_age;

}

return {

name: name,

getAge: getAge,

setAge: setAge

};}

var p1 = Person('张三');

p1.setAge(25);

p1.getAge() // 25

上面代码中，函数Person的内部变量\_age，通过闭包getAge和setAge，变成了返回对象p1的私有变量。

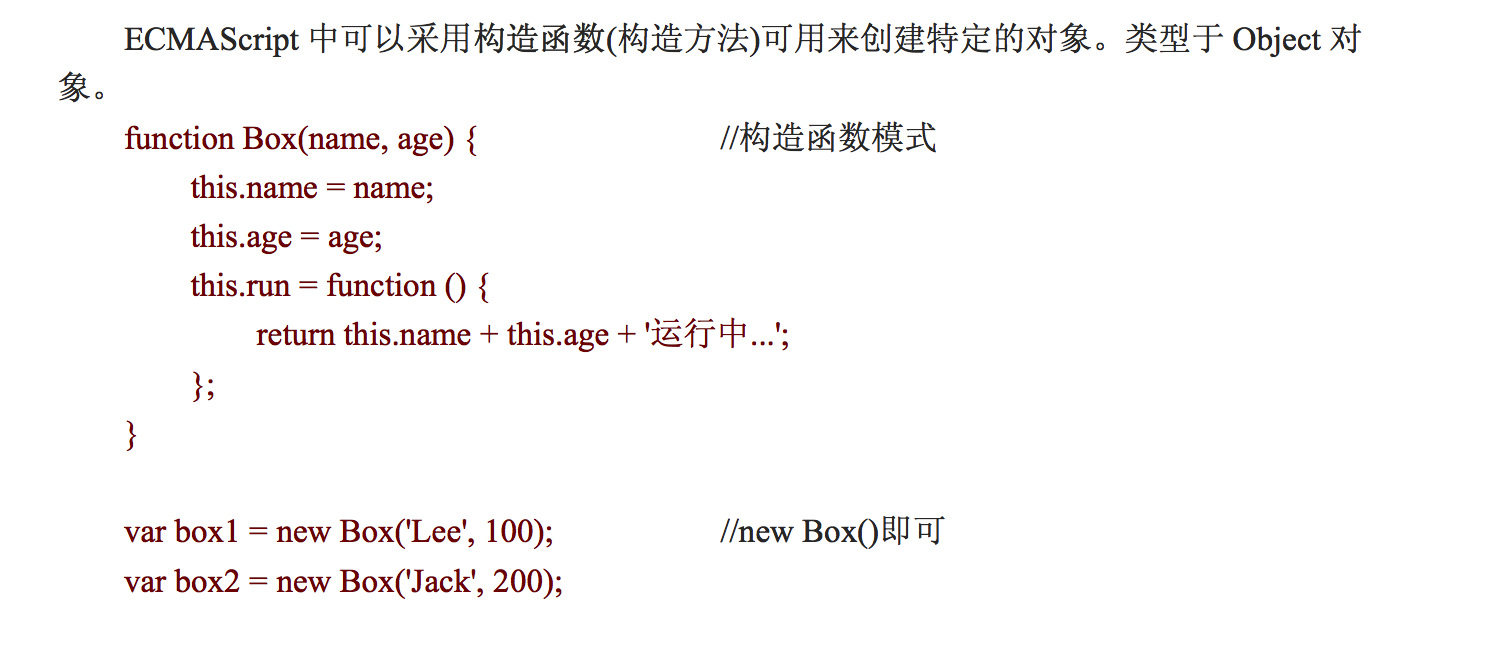
注意，外层函数每次运行，都会生成一个新的闭包，而这个闭包又会保留外层函数的内部变量，所以内存消耗很大。因此不能滥用闭包，否则会造成网页的性能问题。

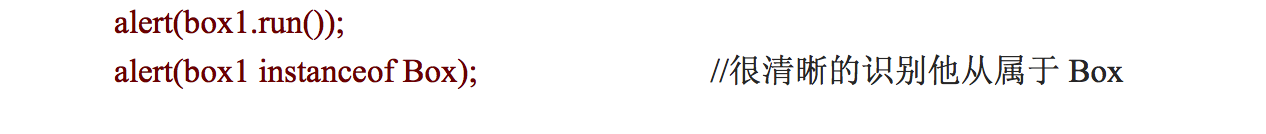
## 应用：

1：掌握闭包的特点和原理。

2：掌握闭包的应用场景。

## 6：构造函数继承







## 7：call/apply继承

call和apply都是为了改变某个函数运行时的context即上下文而存在的，换句话说，就是为了改变函数内部this的指向。

二者作用完全一样，只是接受参数的方式不太一样。

Fn.call(obj, arg1, arg2 [, argN]);

fn,.apply(obj, [arg1, arg2,…, argN]);

apply以数组的形式传参，call是以列表的形式。

call方法是Function类中的方法

call方法的第一个参数的值赋值给类(即方法)中出现的this

call方法的第二个参数开始依次赋值给类(即方法)所接受的参数

apply方法接受2个参数，

A、第一个参数与call方法的第一个参数一样，即赋值给类(即方法)中出现的this

B、第二个参数为数组类型，这个数组中的每个元素依次赋值给类(即方法)所接受的参数

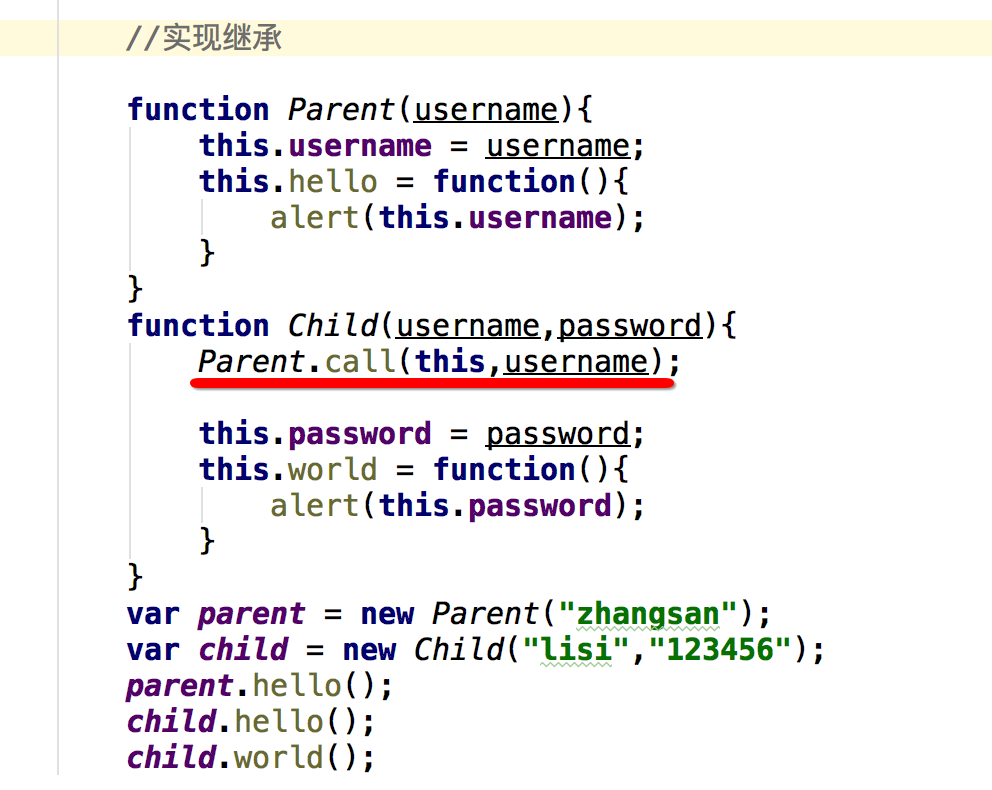
JavaScript不提供找出数组最大元素的函数。结合使用apply方法和Math.max方法，就可以返回数组的最大元素。

var a = [10, 2, 4, 15, 9];

Math.max.apply(null, a)// 15

实现继承：

将父对象的构造函数绑定在子对象上，即在子对象构造函数中加一行：





## 8：prototype的概念

我们创建的每个函数都有一个 prototype(原型)属性，这个属性是一个对象，它的用途是

包含可以由特定类型的所有实例共享的属性和方法。逻辑上可以这么理解: prototype 通过 调用构造函数而创建的那个对象的原型对象 。

function Box() {} //声明一个构造函数

Box.prototype.name = 'Lee'; //在原型里添加属性

Box.prototype.age = 100;

Box.prototype.run = function () { //在原型里添加方法

return this.name + this.age + '运行中...';

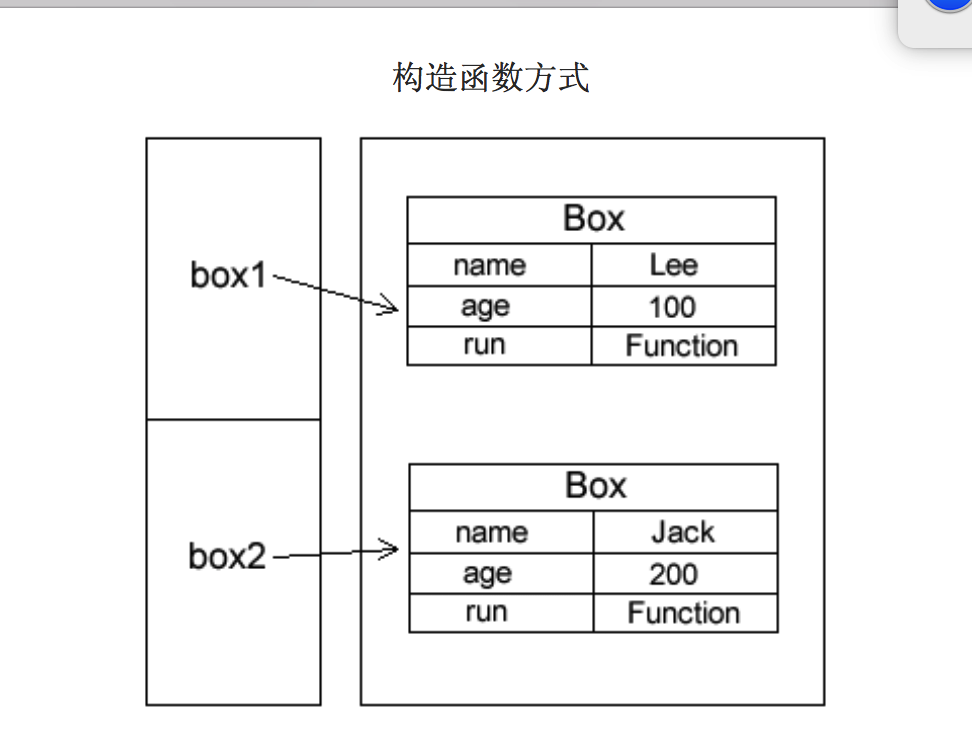
};

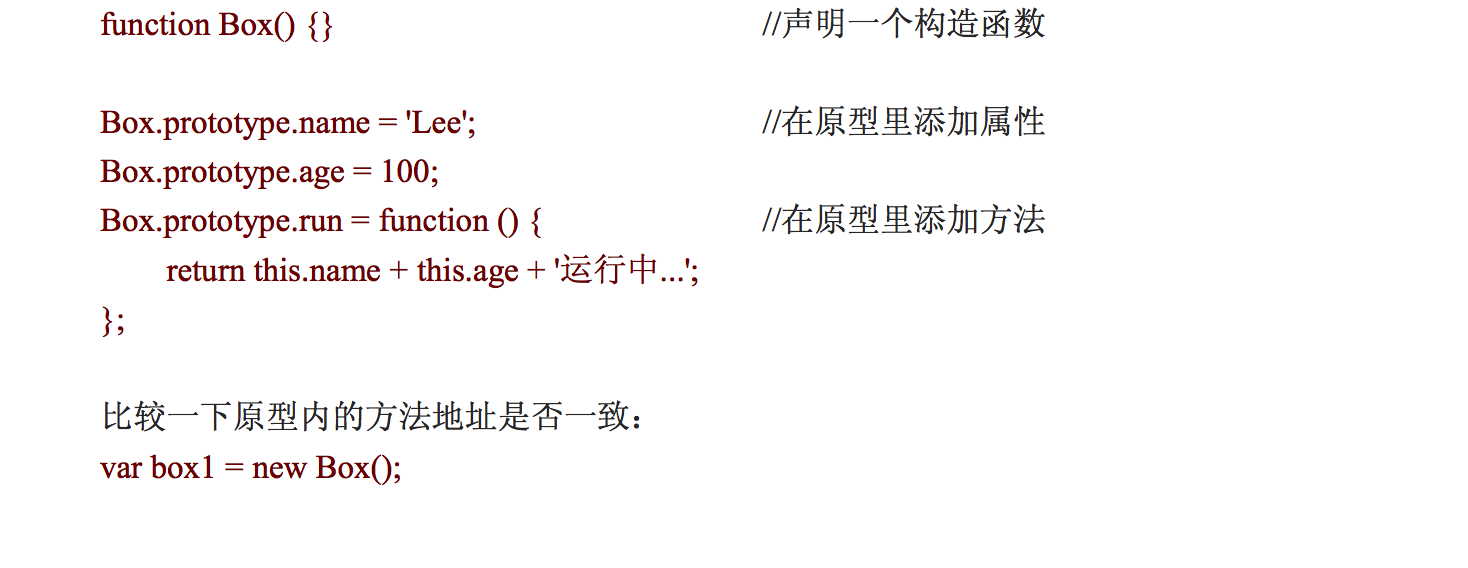
Var box1 = new Box();

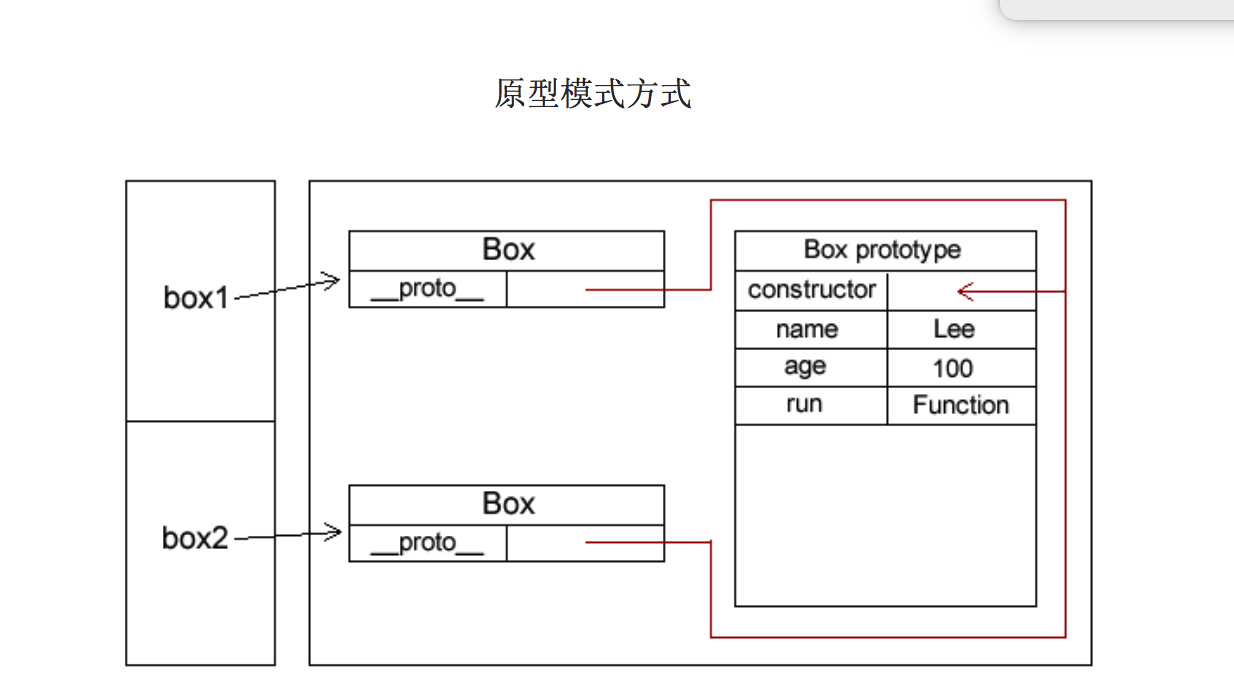
Var box2 = new Box();

Alert(box1.name);

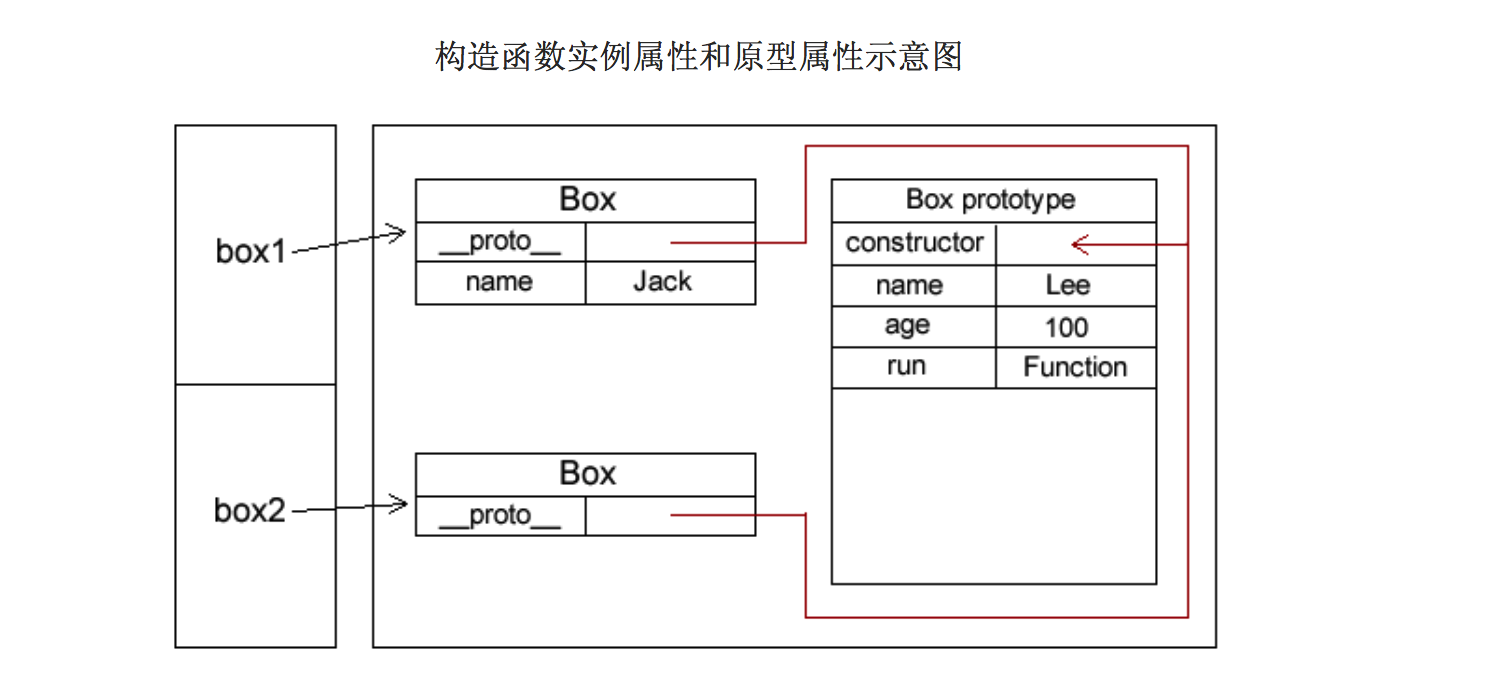
Alert(box2.name);//两个对象的name属性是一样的







在原型模式声明中，多了两个属性，这两个属性都是创建对象时自动生成的 。\_\_proto\_\_ 属性是实例指向原型对象的一个指针，它的作用就是指向构造函数的原型属性， constructor-》构造函数 。 通过这两个属性，就可以访问到原型里的属性和方法了。



如何判断属性是在构造函数的实例里，还是在原型里? 可以使用 hasOwnProperty()函数 来验证:

alert(box.hasOwnProperty('name')); //实例里有返回 true，否则返回 false

in 操作符会在通过对象能够访问给定属性时返回 true，无论该属性存在于实例中还是原 型中。

alert('name' in box); //true，存在实例中或原型中

原型链：

原型模式的执行流程: 1.先查找构造函数实例里的属性或方法，如果有，立刻返回; 2.如果构造函数实例里没有，则去它的原型对象里找，如果有，就返回;

JS在创建对象的时候，都有一个叫做\_\_proto\_\_的内置属性，用于指向创建它的函数对象的原型对象prototype。  
同样，box1.prototype对象也有\_\_proto\_\_属性，它指向创建它的函数对象（Object）的prototype。Object.prototype对象也有\_\_proto\_\_属性，但它比较特殊，为null。我们把这个由\_\_proto\_\_串起来的直到Object.prototype.\_\_proto\_\_为null的链叫做原型链

## 9：原型链的继承

将一个构造函数的原型指向另一个构造函数的实例对象来实现继承。

function Box() { //Box构造

this.name = 'Lee';

}

function Desk() { //Desk构造

this.age = 100;

}

Desk.prototype = new Box(); //Desc继承了Box，通过原型，形成链条

var desk = new Desk();

alert(desk.age);

alert(desk.name); //得到被继承的属性

function Table() { //Table构造

this.level = 'AAAAA';

}

Table.prototype = new Desk(); //继续原型链继承

var table = new Table();

alert(table.name); //继承了Box和Desk

## 10：混合继承

组合继承就是结合使用原型链与构造函数的优点，组合而成的一个模式。实现也很简单，既然是结合，那当然结合了两方的优点，

**即原型链继承方法，而在构造函数继承属性**。

function Box(age) {

this.name = ['Lee', 'Jack', 'Hello']

this.age = age;

}

Box.prototype.run = function () {

return this.name + this.age;

};

function Desk(age) {

Box.call(this, age); //对象冒充 继承了 name 和age 属性

}

Desk.prototype = new Box(); //原型链继承 run方法

var desk = new Desk(100);

alert(desk.run());

利用数组对象的slice方法，可以将一个类似数组的对象（比如arguments对象）转为真正的数组。

Array.prototype.slice.apply({0:1,length:1})// [1]

Array.prototype.slice.apply({0:1})// []

Array.prototype.slice.apply({0:1,length:2})// [1, undefined]

Array.prototype.slice.apply({length:1})// [undefined]

上面代码的apply方法的参数都是对象，但是返回结果都是数组，这就起到了将对象转成数组的目的。从上面代码可以看到，这个方法起作用的前提是，被处理的对象必须有length属性，以及相对应的数字键。

Bind方法（功能和call，apply类似）

bind方法用于将函数体内的this绑定到某个对象，然后返回一个新函数。

var d = new Date();d.getTime() // 1481869925657

var print = d.getTime;print() // Uncaught TypeError: this is not a Date object.

上面代码中，我们将d.getTime方法赋给变量print，然后调用print就报错了。这是因为getTime方法内部的this，绑定Date对象的实例，赋给变量print以后，内部的this已经不指向Date对象的实例了。

bind方法可以解决这个问题，让log方法绑定console对象。

var print = d.getTime.bind(d);print() // 1481869925657

上面代码中，bind方法将getTime方法内部的this绑定到d对象，这时就可以安全地将这个方法赋值给其他变量了。

下面是一个更清晰的例子。

var counter = {

count: 0,

inc: function () {

this.count++;

}};

counter.count // 0counter.inc()counter.count // 1

上面代码中，counter.inc内部的this，默认指向counter对象。如果将这个方法赋值给另一个变量，就会出错。

var counter = {

count: 0,

inc: function () {

this.count++;

}};

var func = counter.inc;func();counter.count // 0count // NaN

上面代码中，函数func是在全局环境中运行的，这时inc内部的this指向顶层对象window，所以counter.count是不会变的，反而创建了一个全局变量count。因为window.count原来等于undefined，进行递增运算后undefined++就等于NaN。

为了解决这个问题，可以使用bind方法，将inc内部的this绑定到counter对象。

var func = counter.inc.bind(counter);func();counter.count // 1

上面代码中，bind方法将inc方法绑定到counter以后，再运行func就会得到正确结果。

this绑定到其他对象也是可以的。

var obj = {

count: 100};var func = counter.inc.bind(obj);func();obj.count // 101

上面代码中，bind方法将inc方法内部的this，绑定到obj对象。结果调用func函数以后，递增的就是obj内部的count属性。

bind比call方法和apply方法更进一步的是，除了绑定this以外，还可以绑定原函数的参数。

var add = function (x, y) {

return x \* this.m + y \* this.n;}

var obj = {

m: 2,

n: 2};

var newAdd = add.bind(obj, 5);

newAdd(5)// 20

上面代码中，bind方法除了绑定this对象，还将add函数的第一个参数x绑定成5，然后返回一个新函数newAdd，这个函数只要再接受一个参数y就能运行了。

如果bind方法的第一个参数是null或undefined，等于将this绑定到全局对象，函数运行时this指向顶层对象（在浏览器中为window）。

function add(x, y) {

return x + y;}

var plus5 = add.bind(null, 5);plus5(10) // 15

上面代码中，函数add内部并没有this，使用bind方法的主要目的是绑定参数x，以后每次运行新函数plus5，就只需要提供另一个参数y就够了。而且因为add内部没有this，所以bind的第一个参数是null，不过这里如果是其他对象，也没有影响

## 应用：

1：掌握继承的原理。

## 综合应用：

1：分析微信飞机大战游戏的继承关系。