面向对象汇总

> OOP：(Object Oriented Programming)面向对象编程
>
> 类的继承、封装、多态

封装

> 把实现一个功能的JS代码封装为一个函数，以后想实现这个功能，执行函数即可 =>”低耦合、高内聚”

多态

> 类具备多种形态：重载、重写

```
1. //->重载
2. //=>后台语言中的重载：方法名相同，参数不同(参数的类型或者个数不一样)，这样相当于两个不同的方法，通过传参不一样，执行对应方法
3. public void sum(int num1,int num2){
4.     //->code
5. }
6. public void sum(int num1){
7.     //->code
8. }
9. sum(10,20); //->执行第一个sum
10. sum(10); //->执行第二个sum
11.
12. //=>JS中不存在类似于后台这样的重载，JS中如果方法名相同，其中一个会把其它相同的函数名都给覆盖掉，最后只保留一个，不管传递几个参数，都只执行这一个；如果一定说有重载，那么JS中的重载就是，通过给一个方法传递不同的实参，在方法中我们判断传递的实参，来处理不同的事情(arguments)
13. function sum(num1,num2){
14.
```

```
        console.log(1);
}
function sum(num1){
    console.log(2);
    if(arguments.length>1){
        num1+=100;
    }else{
        num1-=100;
    }
}
sum(10,20); //->2
sum(10); //->2
```

继承

> 子类继承父类的属性和方法；在继承后，子类还可以把父类的属性和方法进行修改，这就是多态中的重写；

1、call继承 (****)

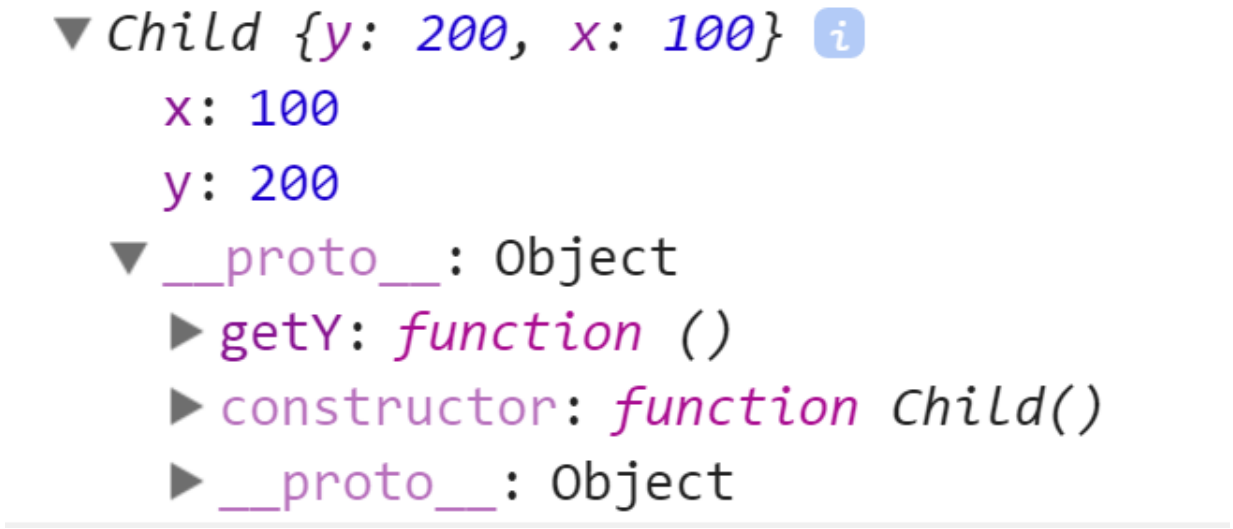
只能让子类的实例，继承父类`私有的属性和方法`

原理：在创建子类实例的时候，把父类当做普通函数执行，让函数中的this变为当前子类的实例(使用call修改的this)，此时在父类函数体中写的this.xxx=xxx这些私有的属性和方法都被子类的实例所占有了

弊端：只能继承父类私有的

```
function Parent() {
    this.x = 100;
}
Parent.prototype.getX = function () {
    console.log(++this.x);
};

function Child() {
    //->this:c
    this.y = 200;
    //Parent();//->this:window
    Parent.call(this);//->this:c  c.x=100
}
Child.prototype.getY = function () {
    console.log(--this.y);
};
var c = new Child();
```


Child {y: 200, x: 100}
x: 100
y: 200
__proto__: Object
getY: function ()
constructor: function Child()
__proto__: Object

2、原型继承 (*****)

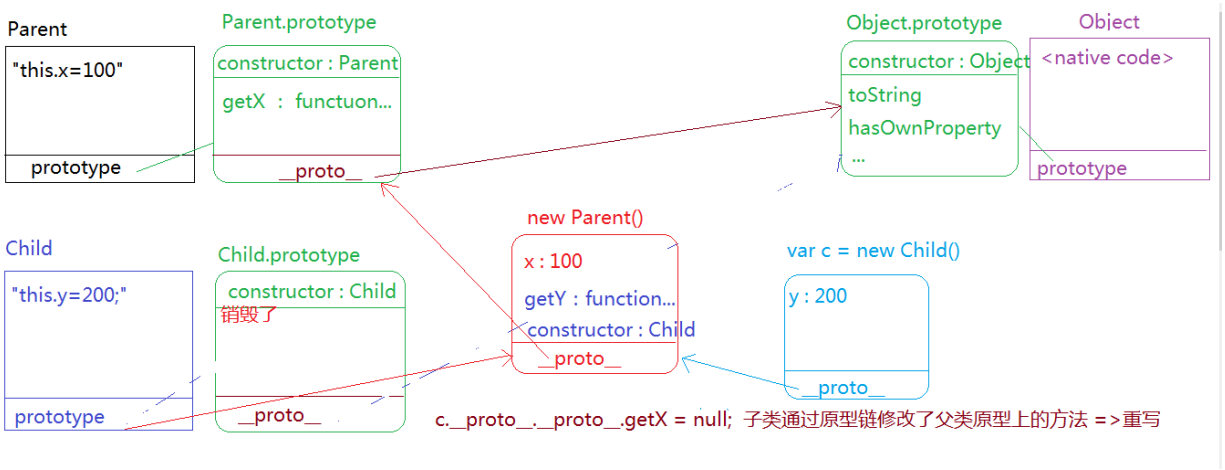
父类私有的属性和公有的属性方法都被子类继承了，而且都变成子类实例公有的属性和方法

原理：让子类的原型等于父亲类的一个实例(父类的实例能够拥有父类私有和公有的属性方法)，这样子类的实例也同时拥有父类私有和公有的；但是原型继承和遗传不太一样，遗传是把父母的基因克隆一份到自己的身上(call继承就是遗传)，而原型继承仅仅是是让子类和父类之间建立了原型链的链接通道，子类实例所使用的父类的公有方法，依然在父类的原型上，使用的时候只是通过原型链查找找到的

弊端：不管父类私有的还是公有的，都是子类公有的了

```
function Parent() {
    this.x = 100;
}
Parent.prototype.getX = function () {
    console.log(++this.x);
};

function Child() {
    this.y = 200;
}
Child.prototype = new Parent(); //->写在第一步,后续再向子类的原型上增加一些属于自己的属性和方法(防止覆盖原有的属性方法)
Child.prototype.constructor = Child;//->防止constructor改变,我们手动增加
Child.prototype.getY = function () {
    console.log(--this.y);
};
var c = new Child();
console.log(c);
```


Parent
"this.x=100"
prototype
Parent.prototype
constructor : Parent
getX ： functuon...
__proto__
Object.prototype
constructor : Object
toString
hasOwnProperty
...
Object
<native code>
prototype
Child
"this.y=200;"
prototype
Child.prototype
constructor : Child
销毁了
__proto__
new Parent()
x : 100
getY ： function...
constructor : Child
__proto__
var c = new Child()
y : 200
__proto__
c.__proto__.__proto__.getX = null; 子类通过原型链修改了父类原型上的方法 =>重写

思考题:

```
//->有三个类
function A(){
    this.a=100;
}
A.prototype.getA=function(){
    console.log(this.a);
}

function B(){
    this.b=200;
}
B.prototype.getB=function(){
    console.log(this.b);
}

function C(){
    this.c=300;
}
C.prototype.getC=function(){
    console.log(this.c);
}

//->让C同时具备A和B的私有属性以及公有属性?
```

3、冒充对象继承 (*)

> 把父类的实例当做一个普通的对象，遍历循环，把父类的私有或者公有的属性和方法，可以放在子类的任意位置，随你喜好即可 => 一般不用

```
function Parent() {
    this.x = 100;
}
Parent.prototype.getX = function () {
    console.log(++this.x);
};

function Child() {
    //->this:c
    this.y = 200;
    var obj = new Parent();
    // obj
    //    x:100
    //    __proto__:
    //     getX:function...
    //     constructor:Parent
    //     __proto__:Object

    //=>for in 循环既可以遍历一个对象私有的属性和方法，也可以遍历部分它原型链上的属性和方法(所有可枚举的都可以遍历,不可枚举的不能遍历) =>一般内置的属性和方法是不能枚举出来的

```

```
        for (var key in obj) {
            //this[key] = obj[key];//->把父类私有的和公有的都变为子类私有的属性和方法(遗传式继承:把父类东西克隆一份过来,继承后子类和父类没啥关系)
            //Child.prototype[key] = obj[key];//->都变为子类公有的

            //->私有的是私有的,公有的是公有的
            if (obj.hasOwnProperty(key)) {
                this[key] = obj[key];
            } else {
                Child.prototype[key] = obj[key];
            }
        }
    }
    Child.prototype.getY = function () {
        console.log(--this.y);
    };

```

```
var c = new Child();
```

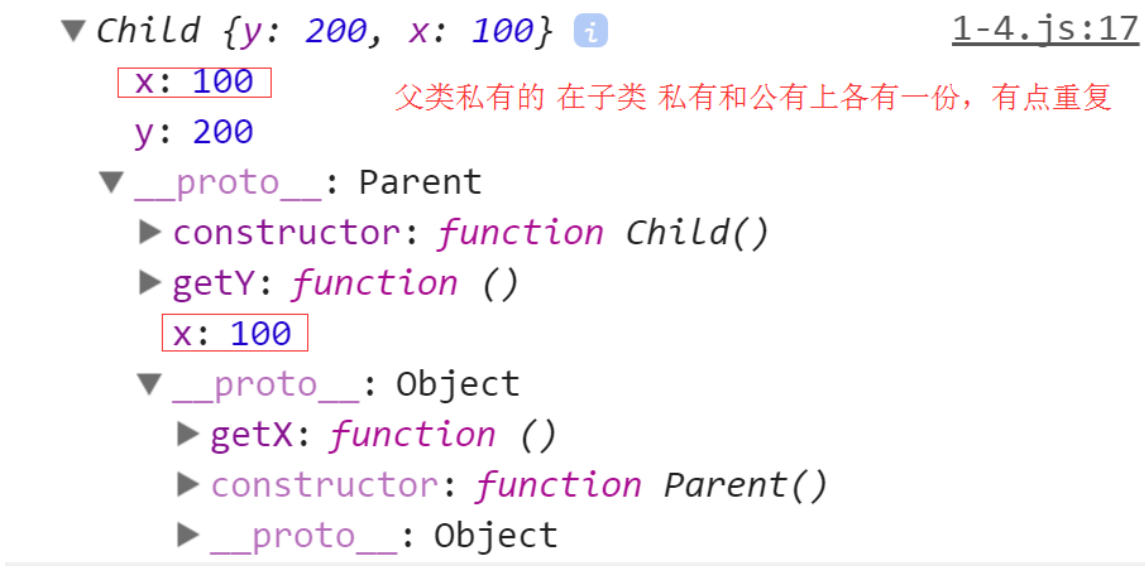
```
//->以后使用for in循环遍历对象的时候，为了防止遍历那些公有的属性和方法，所以我们写这个循环都这样写
for(var key in obj){
    if(obj.hasOwnProperty(key)){
        //...
    }
}

//=>for循环不能遍历公有的属性和方法
```

4、组合继承 (*****)

> call继承+原型继承

```
function Parent() {
    this.x = 100;
}
Parent.prototype.getX = function () {
    console.log(++this.x);
};
function Child() {
    this.y = 200;
    Parent.call(this);//->call继承
}
Child.prototype = new Parent();//->原型继承
Child.prototype.constructor = Child;
Child.prototype.getY = function () {
    console.log(--this.y);
};
var c = new Child();
console.log(c);
```



5、寄生组合式继承 (******)

> 它是把传统组合式继承中，父类私有的在子类私有和公有上各有一份，这个瑕疵点完善了一下
> 父类私有的 => 子类私有的 (遗传式：把私有克隆一份过来的)
> 父类公有的 => 子类公有的 (非遗传式：让原型链之间建立连接的通道)

`Object.create([obj])`

> 创建一个新的空对象
> 让新创建的这个对象的`__proto__`指向`[obj]`

```
> var obj={name:'zxt'};
< undefined
> Object.create(obj)
< ▼ Object {} i
    ▼ __proto__: Object
        name: "zxt"
      ▶ __proto__: Object
```

```
function Parent() {
    this.x = 100;
}
Parent.prototype.getX = function () {
    console.log(++this.x);
};
function Child() {
    this.y = 200;
    Parent.call(this);//->call 继承
}
Child.prototype = Object.create(Parent.prototype);
// new Parent()
//   x : 100
//   __proto__:Parent.prototype

// Object.create(Parent.prototype)
//
//     __proto__:Parent.prototype

```

```
20. Child.prototype.constructor =
    Child;
21. Child.prototype.getY = functio
    n () {
22.     console.log(--this.y);
23. };
24. var c = new Child();
25. console.log(c);
```

6、ES6中的继承

> 采用的原理就是`寄生组合式继承`

```
class Parent {
    constructor() {
        this.x = 100;
    }

    getX() {
        console.log(++this.x);
    }
}

class Child extends Parent {
    constructor() {
        super();//->CALL继承
        this.y = 200;
    }

    getY() {
        console.log(--this.y);
    }
}

var c = new Child();
console.log(c);
```

7、周氏继承法 (中间类继承法)

> 不兼容IE，因为使用了 `__proto__`

```
function sum(){
    //var ary=[].slice.call(arguments);

    //->arguments.__proto__之前指向的是Object.prototype，不能使用数组中的方法

    arguments.__proto__ = Array.prototype;

    //->arguments.__proto__指向Array.prototype，现在就可以使用数组中的方法了
    //->Array.prototype.__proto__指向的是Object.prototype

    //->排序、去头尾、求和、求平均...
    arguments.sort(function(a,b){return a-b;})
    arguments.pop();
    arguments.shift();
    ...
```

```
16.  }
17.  sum(12,23,34,13,24,25);
```