Java 基于对象基础

基于对象和面向对象的区别

JavaScript 设计者想把 JavaScript 语言设计成基于对象(object-based)的语言，他想把这个与面向对象(object-oriented)语言区分开来。但是实际上，可以将基于对象看作是面向对象。

原型对象和类的区别

在 JavaScript 中没有类这一可以使用关键字，当然保留字不算。所以它有自己对类这种封装结构的实现，这和 Java，c++还是有很大区别的。但是我们还是能将原型对象(prototype object)看做是类，它们的用法也很相似。

类（原型对象）和对象的区别和联系

1、类是抽象的概念，代表一类事物

2、对象是具体的，代表一类实体

3、对象是以类（原型对象）为模板创建起来的

案例：

[html] view plain copy

```
<html>
  <head>
    <script language="javascript">
      // 人类
      function Person(){
      }
      // 函数用法
      // Person();
      // 对象用法
      var p = new Person();
      p.name = "张三";
      p.age = 20;
      // 从上面可以看出
      // 1.js 中的对象的属性可以动态的添加
      // 2.属性没有限制
      window.alert(p.name + " " + p.age);
      // 类也是特殊的对象
      window.alert(Person.constructor);
```

```
            // 判断对象是不是类型
            if(p instanceof Person){
                window.alert('true');
            }
            if(p.constructor == Person){
                window.alert('true');
            }
            // 访问对象属性方式
            window.alert(p.name);
            window.alert(p["age"]);
            // 拼接
            var age = "a" + "ge";
            window.alert(p[age]);
        </script>
    </head>
</html>
```

PS：对象公共属性的访问方式 1：对象名.属性名; 2：对象名[‘属性名’];

JS 中创建对象的 7 种方式

1、使用 new Object 创建对象并添加相关属性

2、工厂方式

3、使用字面值创建对象

4、使用构造函数来定义类（原型对象）

5、使用 prototype

6、构造函数及 prototype 混合方式

7、动态创建对象模式

1、使用 new Object 创建对象并添加相关属性

[javascript] view plain copy

```
var obj=new Object();
obj.属性=” aa” ;
obj.show=function (){
};
```

2、工厂方式

[javascript] view plain copy

```
function createPerson(name, age){
    var o = new Object();
    o.name = name;
    o.age = age;
    o.getName() = function(){
        return this.name;
    };
    reuturn o;
}
createPerson( “zs” , 20).getName();
```

3、使用字面值创建对象

[javascript] view plain copy

```
var person = {
    属性;
};
person.属性;
```

4、通过构造函数来定义类

[javascript] view plain copy

```
function 类名/函数名(){
    属性;
}
var p=new 类名/函数名();
```

5、通过 prototype 定义

[javascript] view plain copy

```
function Person(){
}
Person.prototype = {
    属性;
};
var person = new Person();
person.属性;
```

6、构造函数及原型混用（使用最多）

[javascript] view plain copy

```
function Person(){
    属性;
}
var p=new 类名/函数名();
Person.prototype.属性名 = function{
    属性;
};
Person.prototype.属性名 = 属性;
var person = new Person();
person.属性;
```

7、动态原型方法

[javascript] view plain copy

```
function Person(name,age,job)
{
    //属性
    this.name=name;
    this.age=age;
    this.job=job;
    this.friends=["Jams","Martin"];
    //方法
    if(typeof this.sayName !="function")
    {
        Person.prototype.sayName=function()
        {
            alert(this.name);
        };
        Person.prototype.sayFriends=function()
        {
            alert(this.friends);
        };
    }
}
```

```
var person = new Person("kevin",31,"SE");
person.sayName();
person.sayFriends();
```

特别说明：JS 中一切都是对象

实例：

[javascript] view plain copy

```
function Person(){}
window.alert(Person.constructor); //function Function(){ [native code ]}
var a=new Person();
window.alert(a.constructor);//对象实例的构造函数 function Person(){}
window.alert(typeof a);//a 的类型是 object
var b=123;
window.alert(b.constructor); // function Number() { [native code ] }
```

如何判断一个对象实例是不是某个类型

方法 1：

[javascript] view plain copy

```
if(a instanceof Person){
    window.alert("a 是 Person");
}
```

方法 2：

[javascript] view plain copy

```
if(a.constructor==Person){
    window.alert("a 是 Person");
}
```

补充说明：带 var 和不带 var 的区别

[javascript] view plain copy

```
var abc=89;//全局变量
function test(){
    abc=900; // var abc=900;则输出 89
}
test();
window.alert(abc);   //输出 900,
```

PS：这是因为使用在函数中使用 var 之后就将那变量看做局部变量了，JS 的作用域和 Java

以块作用域(Block Scope)不同，它是函数作用域(Function Scope)。

对象的引用问题说明：

[javascript] view plain copy

```
function Person(){}
var a=new Person();
a.age=10;
a.name="小明";
var b=a;
b.name="小白";
window.alert(b.age+"名字"+b.name+"名字"+a.name);
// 输出：10 名字小白名字小白
```

PS :对象是以引用的方式指向堆空间的，所以改变引用对象的值，其堆空间的值也被改变了，那么其他对象在引用，其值是修改后的。

JS 对象回收机制

PS：JavaScript 的对象回收机制是垃圾收集(Garbage Collection)机制，在这点上和 Java 很像，都是当对象的地址被引用的次数变成 0 的时候，GC 就认为这对象是垃圾，就会回收它。但是不一定是变成 0 之后立马回收，GC 会按照一个固定的模式进行回收操作。

除此之外，JS 还提供了一种主动销毁对象属性的方法

基本语法：

[javascript] view plain copy

```
delete 对象名.属性名; // delete 不同作用于对象
```

this 关键字

PS：JavaScript 中 this 关键字，用于指明当前是哪个对象调用函数或者使用属性，this 也可用于区分原型对象(类)中的公开或者私有属性，还可以在传参的时候指定所传入的对象。

案例：

[javascript] view plain copy

```
function Person(){
   var name="abc"; //私有的，只能在内部使用
   var age=900; //私有的，只能在内部使用
   //this 说明 show 属性是公开. 该属性是指向一个函数地址属性.
   //则通过 show 去调用该函数.
   this.show=function(){
      window.alert(name+" "+age);
```

```
    }
}
var p1=new Person();
//window.alert(p1.name+" "+p1.age);//错误的，因为 name,age 是私有的
p1.show();
```

案例：

[html] view plain copy

```
<html>
    <head>
    <script type="text/javascript">
        function test1(){
            alert(this.v);
        }
        var v=190;
        test1(); // <==> window.test1();
        window.test1(); // 输出 190
    </script>
    </head>
</html>
```

this 只能在类定义的内部使用

[javascript] view plain copy

```
//说明 this 只能在 类定义的内部使用
function Dog(){
    this.name="小明";
}
var dog1=new Dog();
window.alert(this.name); //报空, 因为这样使用，相当于去访问 window 对象的 name 属性，但是你没有写.
```

PS：在原型对象（类）内部除了属性之外，还能有函数，函数的创建方式可以参考我的另一篇博客 JavaScript 入门，而函数的添加到原型对象(类)的方法，可以参考上面写到的创建对象的 7 种方式

案例 1：

[html] view plain copy

```
<html>
    <head>
        <script language="javascript">
            function Person(){
                // 公共属性
                this.name = "abc";
                this.age = 20;
                // 私有属性
                var name2 = "xyz";
                var age2 = 30;
                // 公共方法
                this.show = function(){
                    window.alert(name2 + " " + age2);
                    show2();
                }
                // 私有方法
                function show2(){
                    window.alert("show2:" + name2 + " " + age2);
                }
            }
            var p1 = new Person();
            var p2 = new Person();
            //window.alert(p1.name + " " + p2.name);
            //p2.name = "cba";
            //window.alert(p1.name + " " + p2.name);
            p1.show();
            // 不能使用
            //p1.show2();
            // JavaScript 支持这种属性名，属性值的定义方式，这和 CSS 很像
            var dog = {name: '小狗', age: 5, fun1 : function(){ window.alert('hello world'); },
                    fun2 : function(){ window.alert('hello js'); }};
            window.alert(dog.name + " " + dog.age);
            dog.fun1();
```

```
            dog.fun2();
            for(var key in history){
                document.writeln(key + ":" + history[key] + "<br/>");
            }
        </script>
    </head>
</html>
```

案例 2：

[html] view plain copy

```
<html>
    <head>
        <script language="javascript">
            function Person(name, age){
                // 传入参数，进行初始化
                this.name = name;
                this.age = age;
                this.show = function(){
                    document.write("名字是" + this.name);
                }
                 // 1 + 2 + ... + n
                this.plus = function(n){
                    var res = 0;
                    for(var i = 1; i <= n; i++){
                        res += i;
                    }
                    return res;
                }
            }
            var p1 = new Person("张三", 20); `
            p1.show();
            document.write("<br/>" + p1.plus(10));
        </script>
    </head>
```

</html>

综合案例：

JS7.css

[css] view plain copy

```
/* 游戏 */
.gamediv {
    width: 500px;
    height: 400px;
    background-color: silver;
    border: 1px solid red;
}
/* 表格样式 */
.controlcenter{
    width: 200px;
    height: 100px;
    border: 1px solid red;
}
/* 图片样式 */
.mario{
    width: 80;
    position: relative;
}
```

JS7.html

[html] view plain copy

```
<html>
    <head>
        <!--引入 CSS-->
        <link href="JS7.css" type="text/css" rel="stylesheet">
        <script languege="javascript" type="text/javascript">
            // Mario 类
            function Mario(){
                // 初始化坐标
                this.x = 0;
```

```
this.y = 0;
// 移动方式 0 上，1 右，2 下，3 左
this.move = function(direct){
    switch(direct){
        case 0:
            // window.alert("向上移动");
            // 获取 img 元素
            var mymario = document.getElementById("mymario");
            // 通过这样的获取方式,top 和 left 必须直接在 HTML 里面定义
            var top = mymario.style.top;
            top = parseInt(top.substr(0, top.length - 2));
            // 边界情况
            if((top - 10) <= 0){
                mymario.style.top = 0 + "px";
            } else {
                mymario.style.top = (top - 10) + "px";
            }
            break;
        case 1:
            // window.alert("向右移动");
            // 获取 img 元素
            var mymario = document.getElementById("mymario");
            var left = mymario.style.left;
            left = parseInt(left.substr(0, left.length - 2));
            // 边界情况
            if((left + 10) >= 420){
                mymario.style.left = 420 + "px";
            } else {
                mymario.style.left = (left + 10) + "px";
            }
            break;
        case 2:
            // window.alert("向下移动");
```

```
            // 获取 img 元素
            var mymario = document.getElementById("mymario");
            var top = mymario.style.top;
            top = parseInt(top.substr(0, top.length - 2));
            // 边界情况
            if((top + 10) >= 340){
                mymario.style.top = 340 + "px";
            } else {
                mymario.style.top = (top + 10) + "px";
            }
            break;
        case 3:
            // window.alert("向左移动");
            // 获取 img 元素
            var mymario = document.getElementById("mymario");
            var left = mymario.style.left;
            left = parseInt(left.substr(0, left.length - 2));
            // 边界情况
            if((left - 10) <= 0){
                mymario.style.left = 0 + "px";
            } else {
                mymario.style.left = (left - 10) + "px";
            }
            break;
    }
  }
}
// 创建 Mario 对象
var mario = new Mario();
// 全局函数
function marioMove(direct){
  switch(direct){
        case 0:
```

```
                mario.move(0);
                break;
            case 1:
                mario.move(1);
                break;
            case 2:
                mario.move(2);
                break;
            case 3:
                mario.move(3);
                break;
        }
    }
</script>
</head>
<body>
<div class="gamediv">
    <img class="mario" id="mymario" src="cat1.jpg" style="left: 0px; top: 0px; "/>
</div>
<!-- 控制中心-->
<table border="1px" class="controlcenter">
    <tr>
        <td colspan="3">游戏键盘</td>
    </tr>
    <tr>
        <td>**</td>
        <td><input type="button" value="↑ ↑" onclick="mario.move(0)"></td>
        <td>**</td>
    </tr>
    <tr>
        <td><input type="button" value="←←" onclick="mario.move(3)"></td>
        <td>**</td>
        <td><input type="button" value="→→" onclick="mario.move(1)"></td>
```

```
        </tr>
        <tr>
          <td>**</td>
          <td><input type="button" value="↓ ↓" onclick="mario.move(2)"></td>
          <td>**</td>
        </tr>
      </table>
  </body>
</html>
```

JS 闭包

PS:在说明闭包之前，要先弄清楚 js 的变量作用域，JS 的变量作用域是函数作用域(Function Scope)，这也就是说在如果是在嵌套层内定义的变量会覆盖嵌套层之外的变量。可以这样理解，每当进入一个函数的时候，它会想检查这个函数中的各个变量的声明，如果有和函数外相同的，那么就被覆盖掉了，这个将声明提前到函数头而定义位置不变的语法是 JS 的一个特色。

除了了解函数作用域之外，还要知道一点作用域链的知识，在 JS 实现时，它将函数的变量通过链表的形式进行组织的，当执行到嵌套层内部时，嵌套层内部的定义将在链表最前面，之后是上一层，如果还有上一层的话，又会被挂在之后，这也就是为什么嵌套层数越多，全局变量访问越慢的原因了，当然，我们可以通过在嵌套层中获取全局变量的副本，再使用，这样优化，在大型程序中，还是能节省很多时间的。

代码:

[html] view plain copy

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8"/>
<script type="text/javascript">
  a=20;
  function test1(){
    var a=700;
    function test2(){
      //如果没 var ,则会到父函数去，找 a,找到就使用父函数的 a,否则创建
      // 这里有个需要注意的地方是，当 a 前面没有任何符号时，JS 会默认它是公开的，
```

也就是说会自动在前面加一个 this.

```
            a=890;
        }
        return test2;
    }
    var var1=test1();
    var1();
    document.write(a);
</script>
</head>
</html>
```

[javascript] view plain copy

```
function test1(){
    var n=90;
    //test1 函数的内部函数,可以访问 var n
    funciton test2(){
        window.alert(n++);
    }
    //把内部函数 test2 返回外部调用者
    return test2;
}
var res=test1();//调用 test1 ,返回 test2 函数
res(); //这时 res 就是 test1 内部函数 test2, 输出 90
```

闭包的主要用处

1、把局部变量保存在内存中，不让垃圾回收(GC)机制将其回收.

2、让外部去访问内部函数的局部变量.

PS：当你 return 的是内部 function 时，就是一个闭包。内部 function 会 close-over 外部 function 的变量直到内部 function 结束