JavaScript 创建对象的几种方式

- 工厂模式
- 构造函数模式
- 原型模式
- 混合构造函数和原型模式
- 动态原型模式*
- 寄生构造函数模式*
- 稳妥构造函数模式 *

1.工厂模式

抽象出创建具体对象的过程，通过函数入参来创建对象

```
<script>

    function createObject(name, age){
        var o = new Object();
        o.name = name;
        o.age = age;
        o.sayName = function(){
            alert(this.name);
        };
        return o;
    }

    var o1 = createObject('zhang', '21');
    var o2 = createObject('li', '22');

    //缺点：调用的还是不同的方法，没有解决对象识别的问题
    //优点：解决了前面的代码重复的问题，解决了创建多个相似对象的问题
    alert(o1.sayName===o2.sayName);//false

</script>
```

2.构造函数模式

ECMAScript 中的构造函数来创建对象，可以将它的实例标识为一种特定的类型，是优于工厂模式的点；

使用 new 来创建对象会经历一下 4 步：
1.创建一个新对象

2.将构造函数的作用域赋给新对象，this 就指向了这个新对象

3.执行构造函数中的代码，为新对象添加属性和方法

4.隐式返回新对象

```
<script>

    function Person(name){
        this.name = name;
        this.sayName = function(){
            alert(this.name);
        };
    }

    var p1 = new Person('zhang');
    var p2 = new Person('li');

    p1.sayName();
    p2.sayName();

    //constructor 属性可以用来标识对象类型，但是 instanceof 更加可靠一些
    //var arr = new Array(); arr.constructor == Array;//ture
    //Object.prototype.toString.call(arr);//[object:Array]

    alert(p1.constructor === p2.constructor);//true
    alert(p1.constructor === Person);//true
    alert(p2.constructor === Person);//true

    //所有对象均继承自 Object 对象
    alert(typeof(p1));//object
    alert(p1 instanceof Object); //true
    alert(p2 instanceof Object); //true

    alert(p1 instanceof Person); //true
    alert(p2 instanceof Person); //true

    // sayName 函数是独立存在于每个对象的，所以...
    alert(p1.sayName===p2.sayName);//false

</script>
```

与工厂模式的不同在于：

1.没有显示的创建对象

2.直接将对象和方法赋予了 this 对象

3.没有 return

3.原型模式

每个方法都有一个 prototype 的原型属性，每个原型都有一个 constructor 构造器，构造器指向这个方法

不必在构造函数中定义对象实例的信息，而是可以将这些信息直接添加到原型对象中

如果往新建的对象中加入属性，那么这个属性是放在对象中，如果存在与原型同名的属性，也不会改变原型的值。但是访问这个属性，拿到的是对象的值。

访问的顺序：对象本身>构造函数的 prototype

如果对象中没有该属性，则去访问 prototype，如果 prototype 中没有，继续访问父类，直到 Object，如果都没有找到，返回 undefined

```
<script>

    function Animal() { }//构造函数

    Animal.prototype.name = 'animal';
    Animal.prototype.sayName = function () { alert(this.name); };

    var a1 = new Animal();
    var a2 = new Animal();

    a1.sayName();
    //通过原型创建对象，把属性和方法绑定到 prototype，方法是共享的，每个对象调用的都是同一个方法
    alert(a1.sayName === a2.sayName);//true
    alert(Animal.prototype.constructor);//function Animal(){}
    alert(Animal.prototype.constructor==Animal);//true

    //通过 isPrototypeOf()函数来确定对象之间的关系
    //由于 a1 和 a2 内部都有一个指向 Animal.prototype 的指针，因此都返回 true
    alert(Animal.prototype.isPrototypeOf(a1));//true
    alert(Animal.prototype.isPrototypeOf(a2));//true

    //ECAMScript5 中新增 getPrototypeOf()函数，来判断对象的原型
    alert(Object.getPrototypeOf(a1) == Animal.prototype);//true,确定了返回的对象实际上就是这个对象的原型
    alert(Object.getPrototypeOf(a1).name);//animal
```

```
</script>
```

多个对象实例共享原型所保存的属性和方法的基本原理：
当我们调用 a1.sayName()的时候，解析器会进行 2 次搜索。
首先解析器会问：“a1 有 sayName 属性吗？”回答是：“没有”。
继续搜索，“a1 的原型有 sayName 属性吗？”回答是：“有”。
当我们调用 a2.sayName()的时候，将出现相同的搜索过程，得到相同的结果。

虽然可以通过对象实例访问来保存在原型中的值，但却不能通过对象实例重写原型中的值，
如果我们为对象实例添加了一个于原型的同名属性值，那么就在实例中创建了该属性，该属性会屏蔽实例原型中的属性。

```
<script>

    function Person() {}
    Person.prototype.name = "nicholas";
    Person.prototype.age = 29;
    Person.prototype.job = "programmer";
    Person.prototype.sayName = function() { alert(this.name); }

    var person1 = new Person();
    var person2 = new Person();

    alert(person1.hasOwnProperty(name));// false
    alert(person2.hasOwnProperty(name));// false

    person1.name = "greo";
    alert(person1.name); // greo 来源于对象实例
    alert(person1.hasOwnProperty(name));// true
    alert(person2.name); // nicholas 来源于实例原型

    delete person1.name; // 删除后，恢复对原型中 name 属性的连接
    alert(preson1.name); // nicholas
    alert(person1.hasOwnProperty(name));// false

</script>
```

如何确定该属性存在于对象还是原型中？？

```
function hasPrototypeProperty(object, name) {
```

```
    return object.hasOwnProperty(name) && (name in object);
}
hasOwnProperty()---判断属性在对象中
in ---你要通过对象能够访问到就返回 true
如果 hasOwnProperty()返回 false，in 返回 true，可以确定属性位于原型中。

alert(hasPrototypeProperty(person1, name)); //false,因为 person1 拥有自己的 name 属性
alert(hasPrototypeProperty(person2, name)); // true
```

```
原型模式的缺点： 很少有人单独使用原型模式
1.忽略了为构造函数传递初始化参数的环节，结果所有实例默认都将取得相同的属性值
2.原型中属性为多个实例对象共享的方式，对函数来说非常合适；但是对于包含引用类型值得属性来说就存在很大问题

<script>

    function Person() {}
    Person.prototype = {
        name : "nicholas",
        age : 29,
        job : "programmer",
        friends: ["shelby", "court"],
        sayName = function() { alert(this.name); }
    }

    var person1 = new Person();
    var person2 = new Person();

    person1.friends.push("van");

    alert(person1.friends); // shelby, court, van
    alert(person2.friends); // shelby, court, van
    alert(person1.friends == person2.friends);// true

</script>
```

4.混合构造函数和原型模式

```
创建自定义类型的最常见方式，构造函数模式用于定义实例属性，而原型模式用于定义方法和共享属性。
每个实例都会有自己的一份实例属性的副本，同时又共享着对方法的引用，集两种模式之长。
```

```
<script>

    function Person(name, age, job) {
        this.name = name;
        this.age = age;
        this.job = job;
        this.friend = ["shelby", "court"];
    }

    Person.prototype = {
        constructor: Person,
        sayName: function() { alert(this.name); }
    }

    var person1 = new Person("nicholas", 29, "student");
    var person2 = new Person("gero", 22, "techer");

    person1.friends.push("van");

    alert(person1.friends); // shelby, court, van
    alert(person2.friends); // shelby, court
    alert(person1.friends == person2.friends); // false
    alert(person1.sayName == person2.sayName); // true

</script>
```

5.动态原型模式

通过检查某个应该存在的方法是否有效，来决定是否需要初始化原型。

```
<script>

    function Person(name, age, job) {
        //属性
        this.name = name;
        this.age = age;
        this.job = job;
        //方法
        if(typeof this.sayName != 'function') {
            Person.prototype.sayName = function() {
                alert(this.name);
            }
        }
    }
```

```
    var friend = new Person("nicholas", 22, "engineer");
    friend.sayName();

</script>
```

6.{} 、new Object()、 字面量创建方式

缺点： 代码冗余，对象中的方法不能共享，每个对象中的方法都是独立的；

```
1.{}  如果对象不需要重复创建，这种方式很方便
var obj = {};
obj.name = "nicholas";
obj.age = 22;
obj.sayName = function() {alert(this.name);}
```

```
2.new Object()
var obj = new Object();
obj.name = "nicholas";
obj.age = 22;
obj.sayName = function() {alert(this.name);}
```

```
3.字面量创建方式
var person = {name: "nicholas", age: 22, sayName: function() {alert("haha");}}
```