创建对象、原型、属性的查询与设置

对象

1.对象类型

1.1 内部对象

js 中的内部对象包括 Array、Boolean、Date、Function、Math、Number、Object、RegExp、String 以及各种错误类对象，包括 Error、EvalError、RangeError、ReferenceError、SyntaxError 和 TypeError。
其中 Math 这两个对象又被称为“内置对象”，这两个对象在脚本程序初始化时被创建，不必实例化这个对象。

1.2 宿主对象，又称为浏览器对象

宿主对象就是执行 JS 脚本的环境提供的对象。对于嵌入到网页中的 JS 来说，其宿主对象就是浏览器提供的对象，所以又称为浏览器对象，如 IE、Firefox 等浏览器提供的对象。不同的浏览器提供的宿主对象可能不同，即使提供的对象相同，其实现方式也大相径庭！这会带来浏览器兼容问题，增加开发难度。
浏览器对象有很多，如 window 和 document，Screen 对象，History 对象，Location 对象，Navigator 对象等等。
参考链接：https://www.runoob.com/jsref/jsref-tutorial.html

```
var  Img = new Image();
Img.src='...';
```

```
Img.onload=function(){
   //图片加载完成
   }
```

FormData 对象用以将数据编译成键值对，以便用 XMLHttpRequest 来发送数据。其主要用于发送表单数据

```
var formData = new FormData();
formData.append("username", "Groucho");
formData.append("accountnum", 123456);
```

1.3 自定义对象

顾名思义，就是开发人员自己定义的对象。JS 允许使用自定义对象，使 JS 应用及功能得到扩充

```
var obj1 = { name:’ song’ };
```

1.4 函数对象

```
var fun = function() {};
fun.a = 1;
console.log(fun.a) // 打印 “1”
```

1.5 函数对象和普通对象的区别？

- 函数对象和内部对象有 prototype 属性，普通对象没有
- __proto__是每个对象都是的属性

2. 属性的读取和设置

属性的读取有两种方法。可以通过"."和"[]"来读取。

中括号法可以用变量作为属性名，而点方法不可以；

中括号法可以用数字作为属性名，而点语法不可以。

读取：

```
var onePerson = {
age:’20’,
sleep : function () {
      console.log("xiaoming 在睡觉")
   }

}
onePerson.weight      //undefined
onePerson.age
onePerson[‘age’]

var test = ‘age’;
onePerson[test]

访问对象的方法
onePerson.sleep()
```

设置：

```
onePerson.weight = ‘40 公斤’;

onePerson[‘weight’]=’40 公斤’

Object.defineProperty(onePerson,’name’,{
          value:’laney’,
          writable:true,
          enumerable:true,
          configurable:true
})
```

继承：

如果要查询对象 o 的属性 x，如果 o 没有属性 x，则会到 o 的原型去查，一级一级往上，直到 Object.prototype。

```
var o2 = {};
o2.x = 1;
var p = Object.create(o2);
p.x = 2;                              //改变了继承的属性 x
```

```
Console.log(p.x);       //输出 2，p.x 继承自 o2.x
var q = Object.create(o2);
Console.log(q.x);       //输出 1，p 改变的是自己 x,改变不了原型的 x
```

使用 Object.create 创建的对象

使用 Object.create 的 propertyObject 参数

```
var o;

// 创建一个原型为 null 的空对象
o = Object.create(null);
o = {};
// 以字面量方式创建的空对象就相当于:
o = Object.create(Object.prototype);

o = Object.create(Object.prototype, {
// foo 会成为所创建对象的数据属性
foo: {
          writable:true,
          configurable:true,
          value: "hello"
},
// bar 会成为所创建对象的访问器属性
bar: {
          configurable: false,
          get: function() { return 10 },
          set: function(value) {
          console.log("Setting o.bar to", value);
  }
}
});

const obj = Object.create({foo: 1}, { // foo 是个继承属性。
   bar: {
      value: 2  // bar 是个不可枚举属性。
   },
   baz: {
      value: 3,
      enumerable: true  // baz 是个自身可枚举属性。
   }
```

});

const copy = Object.assign({}, obj);
console.log(copy); // { baz: 3 }

3.属性的分类 ：

实例属性和继承属性

Object.create(prototype, [descriptors])

作用: 以指定对象为原型创建新的对象
为新的对象指定新的属性, 并对属性进行描述
value : 指定值
writable : 标识当前属性值是否是可修改的, 默认为 false
configurable: 标识当前属性是否可以被删除 默认为 false
enumerable： 标识当前属性是否能用 for in 枚举 默认为 false

```
var onePerson = {
age:'20',
weight:50
}
var otherPerson = Object.create(onePerson);
otherPerson.name='laney' ;
```

otherPerson 实例属性(自有属性): laney
otherPerson 继承属性：age,weight

4.属性的删除：

delete()方法只能够删除自有属性，不能够删除继承属性。delete()只是断开属性和宿主对象的联系，而不会去操作属性中的属性。

```
var otherperson = Object.create(onePerson);
otherperson .color = 'red';
```

delete otherperson.weight; //继承属性，不能删除

delete otherperson.color ; //自有属性，能删除

5. 属性查找

属性的查找路径： 假设读取某个对象的属性 x，首先会在此对象的实例属性中查找。若没有找到，将会在此对象的原型对象中查找属性 x。若原型对象也没有属性 x，将继续查找这个原型对象的原型(假设原型对象含有原型)，一直到找到属性 x 或者无原型为止。

6 .属性检测。

```
var  flower= {
name:’玫瑰’,
color:’red’
}
flower.__proto__.size="big"
```

in: 检测某对象是否含有某个属性 ,包含实例属性和继承属性
‘name’ in flower //true
‘size’ in flower //true
flower.name //如果没有就是 undefined

hasOwnProperty: 要判断一个属性是否是对象的自身拥有的，而不是继承得到的

flower.hasOwnProperty（‘name’） //true
flower.hasOwnProperty（‘size’） //false

propertyIsEnumerable：
//判断属性是否为可枚举性的,true 时可以被 for in 循环到，false 就不能被循环到
flower.propertyIsEnumerable(‘name’).

6.属性遍历。

for / in 语句块、Object.keys(obj) 、Object.getOwnPropertyNames(obj)

如果有多个属性，如何遍历呢?

1、使用 for..in..遍历

遍历对象可枚举的实例属性和继承属性

2、使用 Object.keys()遍历 ， Object.values()

返回一个数组，包含对象可枚举的实例属性名称

3、使用 Object.getOwnPropertyNames(obj)遍历

返回一个数组，包含对象的所有实例属性名称。包括可枚举和不可枚举的

```
var  flower= {
          name:’玫瑰’,
          color:’red’
}
flower.__proto__.size="big"

var keys = Object.keys(flower); //可枚举的实例属性
var values = Object.values(flower); //可枚举的实例属性
var ownproper =  Object.getOwnPropertyNames(flower); //自有属性
```

方式一：

```
var keysObj = Object.keys(flower);
console.log(keysObj);    //["name", "color"]
keysObj.forEach(function(key,index){
       console.log(oneStudent[key])
});
```

方式二：

```
 for(var key in flower) {
      if(flower.hasOwnProperty(key)){
          //只遍历自有属性
          console.log(flower[key]);
       }
  }

var values = Object.values(flower) ;
```

方式三：

```
Object.getOwnPropertyNames(oneStudent).forEach(function(key){

    console.log(oneStudent[key]);

})
```

7. 序列化对象

JSON.stringify()和 JSON.parse()用来序列化和还原 JavaScript 对象。
JSON.stringify(value [, replacer] [, space]）的原始方法中含有三个参数，第一个参数是必选的，其中第二个参数和第三个参数是可选的
第一个参数：必选参数，就是输入的对象
第二个参数：为过滤参数，数组或者方法
第三个参数：可以控制字符串之间的间距，如果是一个数字，则在字符串化时每一个级别会比上一个级别多缩进 c 个字符

```
var man = { name: "张三", age: 24,color:’red’ };

var person1 = JSON.stringify(man )；  //序列化
var person2 =  JSON.stringify(man,["name","age"] );
var person3 =  JSON.stringify(man,[“name”,"age"],8 );
var person3 =  JSON.stringify(man,[“name”,”age”],’\t' );
var person3 =  JSON.stringify(man,[“name”,”age”],’OK' );
```

8.属性的特性,描述符

属性分为数据属性和访问器属性；

数据属性

value、writable、enumerable、configurable。

value：设置属性的值
writeable：是否可修改属性的值。
enumerable: 是否可枚举属性；true：可枚举，可通过 for/in 语句枚举属性；false：不可枚举。
configurable: 是否可配置。
是否可以通过 delete 删除属性，能否修改属性的特性，能否把属性修改为访问器属性，默认 false；
true：可修改属性的特性

false：不可修改属性的特性

属性描述符用来描述属性特性的(只有在内部才能用的特性),配置属性是否可写，是否可枚举，值是多少，是否可修改.

```
var obj={};
Object.defineProperty(obj,’name’,{
        value:’laney’,
        writable:true,
        enumerable:true,
        configurable:true
})

Object.defineProperty(obj,'age',{
        configurable:false,  //不能删除
        writable:true,   //能否修改
        enumerable:true, //是否可枚举
        value:22
});
delete obj.age   // false 不能删除
obj.age=30  //true 能修改
```

defineProperties

或者用下面的一次性多次添加属性

直接在一个对象上定义一个或多个新的属性或修改现有属性，并返回该对象。

语法: Object.defineProperties(obj, props)

obj: 将要被添加属性或修改属性的对象

props: 该对象的一个或多个键值对定义了将要为对象添加或修改的属性的具体配置

```
Object.defineProperties(obj,{
        name:{
                value:'song',
                writable:true,
```

```
            enumerable:true,
            configurable:true
        },
        age:{
            value:'20',
            writable:true,
            enumerable:true,
            configurable:true
        }
})
```

注明：

(1) 在使用 Object.defineProperty、Object.defineProperties 或 Object.create 函数的情况下添加数据属性，writable、enumerable 和 configurable 默认值为 false。

(2) 使用对象直接量创建的属性，writable、enumerable 和 configurable 特性默认为 true。

- 对象直接量；属性特性默认为 true

```
var o1 = {
   name: 'tom'
};
console.log(Object.getOwnPropertyDescriptor(o1, ‘name'));
```

- 通过 Object.create 创建，属性特性默认为 false

```
var o2 = Object.create(null, {
   name: {value:'tom'}
})
console.log(Object.getOwnPropertyDescriptor(o2, ‘name'));
```

访问器属性

访问器属性不能直接定义，必须使用 Object.defineProperty()来定义

访问器属性 ： set、get、enumerable、configurable。

configurable：true/false，同上；

enumerable：true/false，同上；

set：function，修改属性值时调用的函数；

get：function，读取属性值时调用的函数。

备注：访问器属性可以起到很好的保护作用，当只有 get 方法时，就实现只读不能写；反之，只有 set 时，便是只能写入而不能读取

```
var emp = {
   _name:’laney’,
   _age:’20’
};
Object.defineProperty(emp,’ otherName’ ,{
          enumerable:true,
          configuration:true,
          get:function(){ return this._name};
});
console.log(emp.otherName);//输出 laney，由 get 方法返回_name 的值
emp.otherName =  ‘jery';
console.log(emp.otherName);//输出 laney，没有 set 方法，修改不了_name 的值


//defineProperties 方式一：
      var obj2 = {firstName: 'kobe', lastName: 'bryant'};
      Object.defineProperties(obj2, {
         fullName: {
            //获取扩展属性的值，在获取扩展属性值的时候 get 方法自动调用
            get: function(){
               console.log('get()');
               return this.firstName + " " + this.lastName;
            },
            //监听扩展属性，当扩展属性发生变化的时候，会自动调用，自动调用后会将变化的值作为实参注入到 set 函数
            set: function(data){
               console.log('set()', data);
               var names = data.split(' ');
               this.firstName = names[0];
               this.lastName = names[1];
            }
         }
```

```
    })
    console.log(obj2);
    console.log(obj2.fullName);
    obj2.fullName = 'tim duncan';
    console.log(obj2.fullName);

//字面量定义
var obj3 = {
        firstName: 'stephen',
        lastName: 'curry',
        get fullName(){
            return this.firstName + " " + this.lastName;
        },
        set fullName(data){
            var names = data.split(' ');
            this.firstName = names[0];
            this.lastName = names[1];
        }
 };
 console.log(obj3);
 obj3.fullName = 'kobe bryant';
 console.log(obj3.fullName);
```

访问器属性和数据属性的转化：

```
var testObj = {}

 var testObj 2= Object.create(testObj ,{
        color:{
            value:'red',
            enumerable:false, //不能被 for in 循环到
            writable:true,
            configurable:false
        },
        country:{
            value:'中国',
            enumerable:true,
            writable:true,
            configurable:true
        }
    });


 Object.defineProperty(testObj ,'country',{
```

```
        enumerable:true,
        configuration:true,
        get:function(){ return this.age}
}); //不报错

 Object.defineProperty(testObj ,'country',{
        enumerable:true,
        configuration:true,
        get:function(){ return this.age}
}); //报错
```

9：JavaScript 中的可枚举属性与不可枚举属性

所以可枚举与否都是开发者自己定义的，可以通过 Object.defineProperty()方法

设置可枚举属性

可枚举的属性就是可以列举出来的属性， 换句话说就是可以用 for in 循环的出来的

其实在上面的例子中已经使用到了设置 enumerable 的方法：Object.defineProperty()

```
var person = {
   name:'xiao',
   age: '18',
   sex: 'boy'
}

Object.defineProperty(person,'age',{
   enumerable:true,//可以被枚举
});
Object.defineProperty(person,'sex',{
   enumerable:false,//不可以被枚举
})

Object.defineProperty(obj, 'grade', {
   configurable: false,
   writable: true,
   enumerable: true,
   value: '一年级'
```

```
})

for(var k in person){
   console.log(person[k]),可以被枚举
}
```

总结：可枚举属性能够通过 **for-in** 循环或 **Object.keys()**返回属性

属性 (对象属性)

属性就是 类中包含的变量;每一个对象实例有若干个属性. 为了正确的继承，属性应该被定义在类的原型属性 (函数)中。

在下面的示例中，我们为定义 Person 类定义了一个属性 firstName 并在实例化时赋初值

```
function Person(firstName) {
  this.firstName = firstName;
  alert('Person instantiated');
}

var person1 = new Person('Alice');
var person2 = new Person('Bob');
```

方法（对象属性）

调用方法很像存取一个属性，不同的是在方法名后面很可能带着参数. 为定义一个方法, 需要将一个函数赋值给类的 prototype 属性; 这个赋值给函数的名称就是用来给对象在外部调用它使用的。

在下面的示例中，我们给 Person 类定义了方法 sayHello()，并调用了它.

```
function Person(firstName) {
  this.firstName = firstName;
}

Person.prototype.sayHello = function() {
```

```
  alert("Hello, I'm " + this.firstName);
};

var person1 = new Person("Alice");
var person2 = new Person("Bob");
```

思考下面示例中的代码

```
function Person(firstName) {
  this.firstName = firstName;
}

Person.prototype.sayHello = function() {
  alert("Hello, I'm " + this.firstName);
};

var person1 = new Person("Alice");
var person2 = new Person("Bob");
var helloFunction = person1.sayHello;

person1.sayHello();                    // alerts "Hello, I'm Alice"
person2.sayHello();                    // alerts "Hello, I'm Bob"
helloFunction();
```

Array 扩展

① Array.prototype.indexOf(value) : 得到值在数组中的第一个下标
② Array.prototype.lastIndexOf(value) : 得到值在数组中的最后一个下标
③ Array.prototype.forEach(function(item, index){}) : 遍历数组
④ Array.prototype.map(function(item, index){}) : 遍历数组返回一个新的数组，返回加工之后的值
⑤ Array.prototype.filter(function(item, index){}) : 遍历过滤出一个新的子数组， 返回条件为 true 的值

```
/*
  需求:
  1. 输出第一个 6 的下标
```

```
 2. 输出最后一个 6 的下标
 3. 输出所有元素的值和下标
 4. 根据 arr 产生一个新数组,要求每个元素都比原来大 10
 5. 根据 arr 产生一个新数组, 返回的每个元素要大于 4
 */
var arr = [2,4,3,1,2,6,5,4];
console.log(arr.indexOf(6));
console.log(arr.lastIndexOf(6));
arr.forEach(function(item, index){
  console.log(item, index);
})
var arr1 = arr.map(function(item, index) {
  return item + 10;
});
console.log(arr1);
var arr2 = arr.filter(function(item, index){
  return item > 3;
});
console.log(arr, arr2);
```