面向对象的javascript

概念：JS没有提供传统面向对象语言中的类式继承，而是通过原型委托的方式来实现对象与对象之间的继承，同时也没有在语言层面提供对抽象类和接口的支持.

设计模式原则：

1. 单一职责原则（SRP）：这里的职责是指“引起变化的原因”；单一职责原则体现为：一个对象（方法）只做一件事，但并不是一成不变的，许多时候还是要在方便性和稳定性之间做一些取舍，比如jquery的attr方法既负责取值又负责赋值.
2. 最少知识原则（LKP）：一个软件实体应当尽可能少地与其他实体发生相互作用。这里的软件实体是一个广义的概念，不仅包括对象，也包括系统、类、模块、函数、变量等。
3. 开放-封闭原则（OCP）：软件实体（类、模块、函数）等可以扩展，但不可以被修改。即：当需要变动 一个程序的功能或者给这个程序增加新功能的时候，可以通过增加代码的方式，但是不允许更改程序的源代码

面向接口继承：

这里的接口体现的更为抽象，指的是对象能响应的请求的集合。对于javascript而言，因为javascript是一门动态的语言，类型本身就是一个模糊的概念，不需要利用抽象类或者interface给对象进行“向上转型”，或者说，是天生可以向上转型的。即在动态类型语言中，对象的多态性是与生俱来的。动态类型语言中广泛应用“鸭子类型”

```
var markSound = function(animal){
  if(animal instanceof Duck){
    console.log('嘎嘎嘎')
  }else if(animal instanceof Chicken){
    console.log('咯咯咯')
  }
}
var Duck = function(){}
var Chicken = function(){}
markSound(new Duck())
markSound(new Chicken())
```

```
var markSound = function(animal) {
  animal.sound()
}

var Duck = function(){
}
Duck.prototype.sound = function(){
  console.log('嘎嘎嘎')
}
var Chicken = function(){}
Chicken.prototype.sound = function(){
  console.log('咯咯咯')
```

```
}
markSound(new Duck())
markSound(new Chicken())
```

一. 原型模式

原型编程的基本规则：

1.大部分数据都是对象（undefined）

```
var obj1 = new Object()
var obj2 = {}
console.log(Object.getPrototypeOf(obj1) === Object.prototype)  // true
console.log(Object.getPrototypeOf(obj2) === Object.prototype)  // true
// 从上面可以看出JS的根对象就是Object.prototype,我们遇到的每个对象都是从它克隆过来的,
Object.prototype就是他们的原型
```

2.要得到一个对象，不是通过实例化类，而是找到一个对象作为原型并克隆它

```
function Person(name){
  this.name = name
}
Person.prototype.getName = function(){
  return this.name
}
var a = new Person('sundong')  // 这个地方的Person函数是一个构造器,当使用new运算符
来创建对象时，实际上也是先克隆Object.prototype对象，然后js做一些额外的处理.
console.log(a.name) // sundong
console.log(a.getName()) // sundong
console.log(Object.getPrototypeOf(a) === Person.prototype)  // true
```

3.对象会记住它的原型 (**proto**):

```
var a = new Object()
console.log(a._proto_ === Object.prototype)  // true
```

4.如果对象无法响应某个请求，它会把这个事情委托给它自己的原型

```
var obj = {name: 'sundong'}
var A = function (){}
A.prototype = obj
var newObj = new A()
console.log(newObj.name) // sundong

// 1.尝试遍历对象newObj中的所有属性，没有找到name属性
// 2.查找name属性的这个请求被委托给对象newObj的构造器的原型上，它被newObj._proto_记录着并且指向A.prototype,而A.prototype被设置为对象obj
// 3.最后在obj中找到了name属性，并返回它的值
```

当期望得到一个“类”继承另一个“类”时

```
var A = function () {}
A.prototype = {name: 'sundong'}
var B = function () {}
B.prototype = new A()
var b = new B()
console.log(b.name)  // sundong
// 1.首先遍历b中的所有对象，没有找到name属性
// 2.查找name属性的请求被委托给对象b的构造器的原型，它被b._proto_记录着并指向B.prototype，而B.prototype被指向通过new A()创建出来的对象
// 3.在该对象中依然没有找到name属性，于是请求被继承委托给这个对象构造器的原型A.prototype
// 4.在A.prototype中找到了name属性，并返回它的值
```

注意：

1. **原型链并不是无限长的，如果最后Object.Prototype上面没有找到想要的属性,后面没有其他的节点了，所以会返回undefined**
2. **对象都会有一个原型，但通过Object.create(null)可以创建出没有原型的对象.（效率并不高，慢.）**

二. this指向

js的this总是指向一个对象，而具体指向哪个对象是在运行时基于函数的执行环境动态绑定的，而非函数被声明时的环境

this指向

1.作为对象的方法调用

```
var obj = {
  name: 'sundong',
  getName:function(){
    console.log(this === obj)  // true
    console.log(this.name) // sundong
  }
}
obj.getName()  // 当函数作为对象的方法被调用时，this指向该对象
```

2.作为普通函数调用

```
window.name = 'sundong'
var obj = {
  name: 'WADE',
  getName:function(){
    return this.name
  }
}
var getName = obj.getName
console.log(getName())  // sundong
// 当函数不作为对象的属性被调用时，也就是我们常说的普通函数方式，此时的this指向全局对象，在浏览器的JS里面这个全局对象就是window对象
```

3.构造器调用： javascript中没有类，但是可以从构造器中创建对象，同时提供了new运算符.

```
// 当用new运算符调用函数时，该函数总是会返回一个对象，通常情况下，构造器里面的this就指向这个对象
var Myclass = function(){
  this.name = 'sundong'
}
var obj = new Myclass()
console.log(obj.name)  // sundong
```

4.Function.prototype.call或Function.prototype.apply调用

作用：可以动态的传入this

区别：传入参数形式的不同

```
var func = function(a,b,c){
  console.log([a,b,c])  // [1,2,3]
}
func.apply(null, [1,2,3])
func.call(null, 1,2,3)
// 第一个参数为null，函数体的this会指向默认的宿主对象，在浏览器中则是window
// ps:以上情况在严格模式下，this还是指向null而不是window
```

```
var obj1 = {
  name: 'sundong'
}
var obj2 = {
  name: 'WADE'
}
window.name = 'window'
var getName = function(name){
  console.log('this', this.name)
}
getName()  // window
getName.call(obj1) // sundong
getName.call(obj2) // WADE
```

```
// 模拟实现Function.prototype.bind
Function.prototype.bind = function() {
    var self = this,
        context = [].shift.call(arguments),
        args = [].slice.call(arguments);
    return function() {
        return self.apply(context, [].concat.call(args,
[].slice.call(arguments)));
    };
};

var obj = {
    name: "sundong"
};
var func = function(a, b, c) {
    console.log(this.name);  // sundong
    console.log([a, b, c])   // [1, 2, 3]
}.bind(obj, 1, 2);

func(3)
```

丢失的this

```
var obj = {
  myName: 'sundong',
  getName:function(){
    return this.myName
  }
}
console.log(obj.getName()) // sundong
var getName2 = obj.getName
console.log(getName2.getName())  // undefined  属于普通函数调用，this指向window
```

三. 闭包

定义：是函数和声明该函数的词法环境的组合

```
function init () {
  var name = 'sundong'  // name是init的局部变量
  function getName () { // getName()是内部函数，一个闭包
    console.log(name)  // sundong  使用了父函数中声明的变量
  }
  return getName
}
var myFunc = init()
myFunc()
```

缓存机制

```
var mult = (function() {
    var cache = {};
    return function() {
        var args = Array.prototype.join.call(arguments, ",");
        if(args in cache) {
            return cache[args];
        }
        var a = 1;
        for(var i = 0, l = arguments.length; i < l; i++) {
            a = a * arguments[i];
        }
        return cache[args] = a;
    }
})();

console.log(mult(1, 2, 3));
console.log(mult(1, 2, 3));     // 再次计算，直接从缓存中取
```

注意：闭包容易导致循环引用，从而导致内存泄漏。可以通过把这些变量设置为null，回收这些变量

四.高阶函数

定义：函数作为参数传递，函数作为返回值输出.

1.回调函数（作为函数传递）

```
function fun (callback) {
  console.log(111)
  if(typeof callback === 'function'){
    setTimeout(function(){
      callback()
    })
  }
}
var fn = function(){
  console.log(2)
}
fun(fn)
```

2.判断数据类型（作为返回值输出）

```
var isType = function (type) {
  return function (obj) {
    return Object.prototype.toString.call(obj) === '[Object' + type +']'
  }
}
var isArray = isType('Array')
var isNumber = isType('Number')
isArray([])
isNumber("123")
```

3.单例（既把函数当做参数传递，又让函数执行后返回了另外一个函数）

```
var getSingle = function(fn) {
    var result;
    return function() {
        return result || (result = fn.apply(this, arguments));
    };
};

function testSingle(){}
getSingle(testSingle)() === getSingle(testSingle)();    // true
```