高级2

问题1： apply、call 有什么作用，什么区别

Javascript的每个Function对象中有一个apply方法：

```
function.apply([thisObj[,argArray]])
```

还有一个类似功能的call方法：

```
function.call([thisObj[,arg1[, arg2[, [,.argN]]]]])
```

- 它们各自的定义：
  apply：应用某一对象的一个方法，用另一个对象替换当前对象。
  call：调用一个对象的一个方法，以另一个对象替换当前对象。
- 它们的共同之处：
  都“可以用来代替另一个对象调用一个方法，将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。
- 它们的不同之处：
  apply：
  最多只能有两个参数——新this对象和一个数组 argArray。如果给该方法传递多个参数，则把参数都写进这个数组里面，当然，即使只有一个参数，也要写进数组里面。如果 argArray 不是一个有效的数组或者不是 arguments 对象，那么将导致一个 TypeError。如果没有提供 argArray 和 thisObj 任何一个参数，那么 Global 对象将被用作 thisObj，并且无法被传递任何参数。
  call：
  则是直接的参数列表，主要用在js对象各方法互相调用的时候，使当前this实例指针保持一致,或在特殊情况下需要改变this指针。如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。

更简单地说，apply和call功能一样，只是传入的参数列表形式不同：如 func.call(func1,var1,var2,var3) 对应的apply写法为：func.apply(func1,[var1,var2,var3]) 也就是说：call调用的为单个，apply调用的参数为数组

```
function sum(a,b){
  console.log(this === window);//true
  console.log(a + b);
}
sum(1,2);
sum.call(null,1,2);
sum.apply(null,[1,2]);
```

作用

调用函数

```
var info = 'tom';
```

```
function foo(){
  //this指向window
  var info = 'jerry';
  console.log(this.info);   //tom
  console.log(this===window)  //true
}
foo();
foo.call();
foo.apply();
```

call和apply可以改变函数中this的指向

```
var obj = {
      info:'spike'
}
foo.call(obj);    //这里foo函数里面的this就指向了obj
foo.apply(obj);
```

借用别的对象的方法
eg:求数组中的最大值

```
var arr = [123,34,5,23,3434,23];
//方法一
var arr1 = arr.sort(function(a,b){
  return b-a;
});
console.log(arr1[0]);
//方法二
var max = Math.max.apply(null,arr)   //借用别的对象的方法
console.log(max);
```

问题2： 以下代码输出什么?

```
var john = {
  firstName: "John"
}
function func() {
  alert(this.firstName + ": hi!")
}
john.sayHi = func
john.sayHi()  // John: hi!
//这种调用方式会使this指向调用sayHi()的john对象
```

问题3： 下面代码输出什么，为什么

```
func()
function func() {
  alert(this) //window
  //在这里this指向undefined。在浏览器中，这里undefined默认为window
```

```
}
```

问题4：下面代码输出什么

```
    document.addEventListener('click', function(e){
    console.log(this);  //document
                        //在事件处理程序中this代表事件源DOM对象
    setTimeout(function(){
        console.log(this);  //window
                            //setTimeout、setInterval这两个方法执行的函数this指向
全局对象
    }, 200);
}, false);
```

问题5：下面代码输出什么，why？

```
var john = {
  firstName: "John"
}

function func() {
  alert( this.firstName )
}
func.call(john)  //John
               //call(john)传入了john这个执行上下文，this指向john这个对象
```

问题6： 以下代码有什么问题，如何修改？

```
var module= {
  bind: function(){
    $btn.on('click', function(){
      console.log(this) //$btn
      this.showMsg();
    })
  },

  showMsg: function(){
    console.log('饥人谷');
  }
}
////////////////////////////
var module= {
  bind: function(){
      var cur = this;  //将module存到变量cur
      $btn.on('click', function(){
      console.log(this) //$btn
      cur.showMsg(); //饥人谷
    })
```

```
  },

  showMsg: function(){
    console.log('饥人谷');
  }
}
```

原型链相关问题

问题7：有如下代码，解释`Person、 prototype、__proto__、p、constructor`间的关联。

```
function Person(name){
    this.name = name;
}
Person.prototype.sayName = function(){
    console.log('My name is :' + this.name);
}
var p = new Person("若愚")
p.sayName();
```

1. `Person`是构造函数，也是一个对象，这个对象里面存在一个`prototype`属性，而构造函数内部定义了实例的属性和方法，这些属性和方法是属于该类的所有实例的特征；

2. `p`是通过构造函数`Person`构造出来的实例，也是拥有`__proto__`属性。所以`p.__proto__ === Person.prototype;`

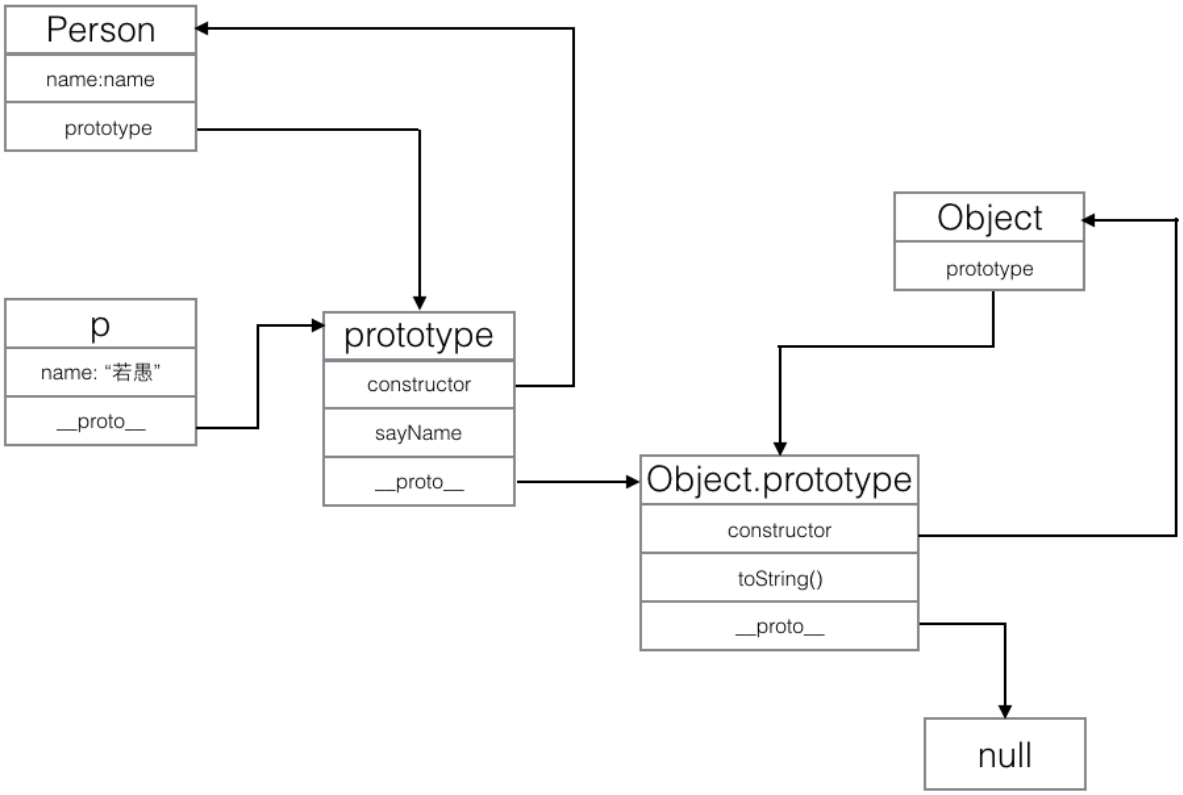
3. `prototype`是构造函数内部的原型对象，所以拥有`contructor`和`__proto__`属性，其中`contructor`属性指向构造函数Person，`__proto__`指向该对象的原型，即

```
Person.prototype.__proto__ === Object.prototype;
Person.prototype.constructor == Person
```

问题8： 上例中，对对象 p可以这样调用`p.toString()`。`toString`是哪里来的? 画出原型图?并解释什么是原型链。

`p.toString()`方法是继承构造函数`Object`的原型对象里定义的`toString`方法，首先p会找自己的`toString`方法，如果没有找到，会沿着`__proto__`属性继续到构造函数`Person`的`prototype`里找`toString`方法，如果还未找到，再继续往`Person.prototype`的`__proto__`即`Object.prototype`找`toString`方法，最后找到`toString()`方法。

原型链：由于原型对象本身也是对象，而每个`javascript`对象都有一个原型对象，每个对象都有一个隐藏的`__proto__`属性，原型对象也有自己的原型，而它自己的原型对象又可以有自己的原型，这样就组成了一条链，这个就是原型链。在访问对象的属性时，如果在对象本身中没有找到，则会去原型链中查找，如果找到，直接返回值，如果整个链都遍历且没有找到属性，则返回`undefined`。原型链一般实现为一个链表，这样就可以按照一定的顺序来查找。



问题9：对String做扩展，实现如下方式获取字符串中频率最高的字符

```
String.prototype.getMostOften = function(){
    var all = {};
    for(var i=0,word;i<this.length;i++){

        word = this[i];
        if(all[word]){
            all[word]+=1;
console.log(word);
console.log(all[word]);
console.log(all);
        }else{
            all[word] = 1
        }
    }

    var max = 0,who;
    for(var k in all){
        if(all[k]>max){
```

```
            max = all[k];
            who = k;
        }
     }

    return who;
}
var str = 'ahbbccdeddddfg';
var ch = str.getMostOften();
console.log(ch);   //d
```

问题10： `instanceOf`有什么作用？内部逻辑是如何实现的？

1.instanceof用来检查一个对象是不是另一个构造对象的实例。

2.其内部逻辑是测试一个对象在其原型链中是否存在一个构造函数的 prototype 属性。

3.所以如果表达式obj instanceof Foo 返回true，则并不意味着该表达式会永远返回true，因为Foo.prototype属性的值有可能会改变，改变之后的值很有可能不存在于obj的原型链上，这时原表达式的值就会成为false。

```
function instanceOf(obj,fn){
  var oldpro = obj.__proto__;
  while(oldpro){
    if(oldpro === fn.prototype){
        return true;
    }else{
        oldpro = oldpro.__proto__;
    }
  }
  return false;
}
```

（额外）11.isPrototypeOf()

这个方法用来判断，某个proptotype对象和某个实例之间的关系。

```
alert(Cat.prototype.isPrototypeOf(cat1)); //true
alert(Cat.prototype.isPrototypeOf(cat2)); //true
```

（额外）12.in运算符

in运算符可以用来判断，某个实例是否含有某个属性，不管是不是本地属性。

```
alert("name" in cat1); // true
alert("type" in cat1); // true
```

in运算符还可以用来遍历某个对象的所有属性。

```
for(var prop in cat1) {
alert("cat1["+prop+"]="+cat1[prop]); }
```

继承相关问题

问题11：继承有什么作用?

1.可以使子类共享父类的属性和方法;

2.可以覆盖和扩展父类的属性和方法。

问题12： 下面两种写法有什么区别?

```
//方法1
function People(name, sex){
    this.name = name;
    this.sex = sex;
    this.printName = function(){
        console.log(this.name);
    }
}
var p1 = new People('饥人谷', 2)

//方法2
function Person(name, sex){
    this.name = name;
    this.sex = sex;
}

Person.prototype.printName = function(){
    console.log(this.name);
}
var p1 = new Person('若愚', 27);
```

前者会把方法重新定义在每个构建的对象上，如果构建的对象比较多，代码就会很多，对性能造成负面影响；

后者会把个方法放到构建函数的protoType这个公共容器里，往后新构建的函数将会通过原型链获取到这个方法，节约了内存。这种方法比较先进。

问题13： Object.create 有什么作用？兼容性如何？

`Object.create()` 方法创建一个拥有指定原型和若干个指定属性的对象。Object.create是在ES5中规定的，IE9以下无效。

问题14： hasOwnProperty有什么作用？ 如何使用？

`hasOwnPerperty`是`Object.prototype`的一个方法，可以判断一个对象是否包含自定义属性而不是原型链上的属性，`hasOwnProperty`是`JavaScript`中唯一一个处理属性但是不查找原型链的函数。

```
m.hasOwnProperty('name'); // true
m.hasOwnProperty('printName'); // false
Male.prototype.hasOwnProperty('printAge'); // true
```

问题15：如下代码中call的作用是什么?

```
function Person(name, sex){
    this.name = name;
    this.sex = sex;
}
function Male(name, sex, age){
    Person.call(this, name, sex);    //调用Person函数，使Male函数能够执行Person上的初始化代码，实现构造函数继承
    this.age = age;
}
```

问题16： 补全代码，实现继承

```
function Person(name, sex){
    this.name = name;
    this.sex = sex;
}

Person.prototype.getName = function(){
    console.log(this.name);
};

function Male(name, sex, age){
    Person.call(this,name,sex);
    this.age = age;
}

Male.prototype = Object.create(Person.prototype);
Male.prototype.constructor = Male;
Male.prototype.getAge = function(){
console.log(this.name);
};

var ruoyu = new Male('若愚', '男', 27);
ruoyu.printName();
```