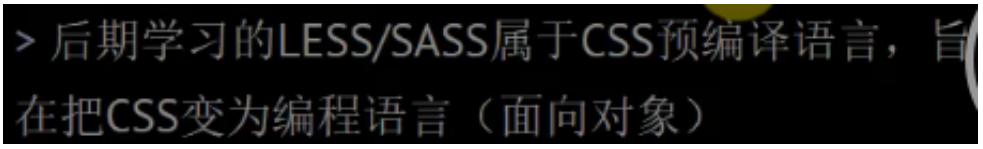
js二

为什么要学习面向对象？

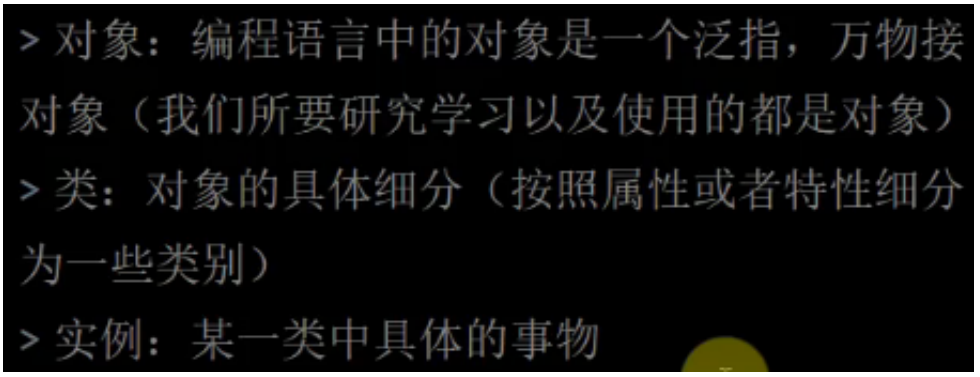
- 能够对JS内置类有更清晰了解
- 学习封装代码的方式，各种模式之间的区别是什么？（学会在合适的场景下选择对应的设计模式）

面向对象（OOP）是什么意思？

- 全称 object oriented programming 以对象数据类型为导向的编程
- 单例模式 工厂模式 构造函数模式 原型模式



对象，类，实例



对象 万物皆对象 地球（抽象）

- 类 对象中细分的类型 例如:地球上所有的事物给分类了（抽象）
- 动物类：
  - 人类： 程序猿类 专家类 科学家类
  - 鱼类
  - 鸟类
- 植物类：
- 微生物类
- 实例：某类中实实在在的事物（具体的事物）一个人，一个鱼

es5中类叫构造函数 es6 class 才叫类

- JS中的内置类：
  Number,String,Boolean,Null,Undefined,Object,Array,RegExp,Date,Function…
  - Number,String,Boolean 包装类
    var num = 10;//实例 属于Number类 字面量的写法
    //var num = new Number(10); num = num.valueOf(); //10
    //var str = “abc”;//是String类实例

```
//var ary = [];//字面量  Array类的实例
var ary = new Array(10);//参数是数值，数组中有10项
var ary = new Array("abc");//数组中有一项是"abc";
var ary = new Array(10,20);//[10,20]
console.log(ary);
//自己写个类（Person），自定义类
```

构造函数

- 为了跟普通函数区别开来，构造函数首字母要大写

```
//自定义一个Person类
function Person(name,age){
//1.创建对象 （系统自己创建对象）
//2.给对象添加属性 （对象用this表示）
        var name="rose";
        this.name =name;
        this.age = age;
//3.返回对象  （系统自己返回对象）
    }
Person.prototype.writeCss = function(){//放在原型上是所有实例功能的属性
       console.log('writeCss');
    };
//Person();//这个Person就是普通函数，函数中的this是window
var p1 = new Person("lily",20);//通过new去运行的函数才称为构造函数(类)  P
erson构造函数或Person类  p1指的是返回的对象，也就是实例
var p2 = new Person("lucy",18);
//console.log(p1.writeCss == p2.writeCss);
console.log(p1);
```

构造函数的细节知识点

- 1.new 运行函数时，若不需要传参，小括号可省略
- 2.构造函数中this指的是实例，实例只跟this.xxx有关,跟私有变量无关

- 3.构造函数定义的属性都是私有属性，所有实例都有的功能放在原型上
- 4.return不用写，若非写，return 后面跟基本类型数据对实例没影响，若return 后面跟引用类型的数据会把系统返回的实例覆盖

单例模式

- 其实就是对象
- 概念 将同一事物的属性和方法封装在一个命名空间(对象名)里
- 解决问题：防止代码冲突和覆盖
- 不同模块之间获取属性：模块名.xxx
- 同一模块之间获取属性：this.xxx
- 升级单例模式后，正常情况下，内容是模块私有的，只有这个模块内部才能调用，若要公有的，得处理下，在暴露给外界的内容放在return 后的对象中返回

```
//login模块
var login = { //login命名空间中
   username:"lily",
   password:"123456",
   email:function(){
   console.log("email")
            },
fn1:function(){
   this.email(); //调用同一模块中的方法 this.方法名
   console.log("fn1")
            }
        };
login.fn1();
var register = {
    username:"lucy",
    password:"123456",
    fn2:function(){
//验证邮箱
login.email();//不同模块之间调用 模块名.方法名
                console.log(this.username);
            }
        }
    register.fn2();

btn.onclick = function(){
            console.log(login.username);
            console.log(login.password);
        }
```

高级单例模式，解决私有和共享问题

```
var login = (function(){
    var username = "lily";
    var password = "123456";
    function email(){
       console.log("email");
        }
    function check(){
       console.log("email");
        }
       return {
         email:email
        }
    })();

var register = {
    fn:function(){
       login.check();
       console.log("fn");
        }
    };
register.fn();
```

工厂模式

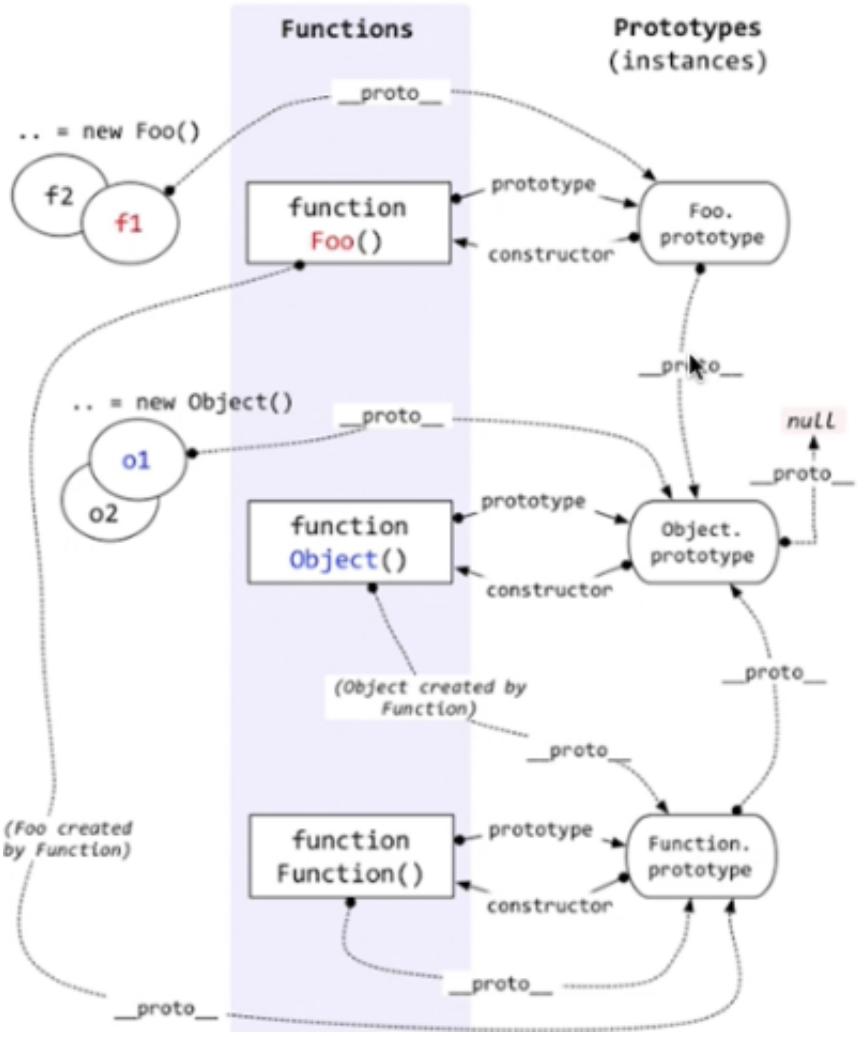
- 单例模式 手工作坊式
- “批量生产” ->工厂模式

```
function shirt(){
//1.创建一个对象
    var obj = {};
//2.给对象添加属性
    obj.size = "XXL";
    obj.material ="棉";
    obj.fn = function(){console.log("fn")};
//3.返回对象
    return obj;
    }
for(var i = 0;i<1000;i++){
    console.log(shirt());
    }
```

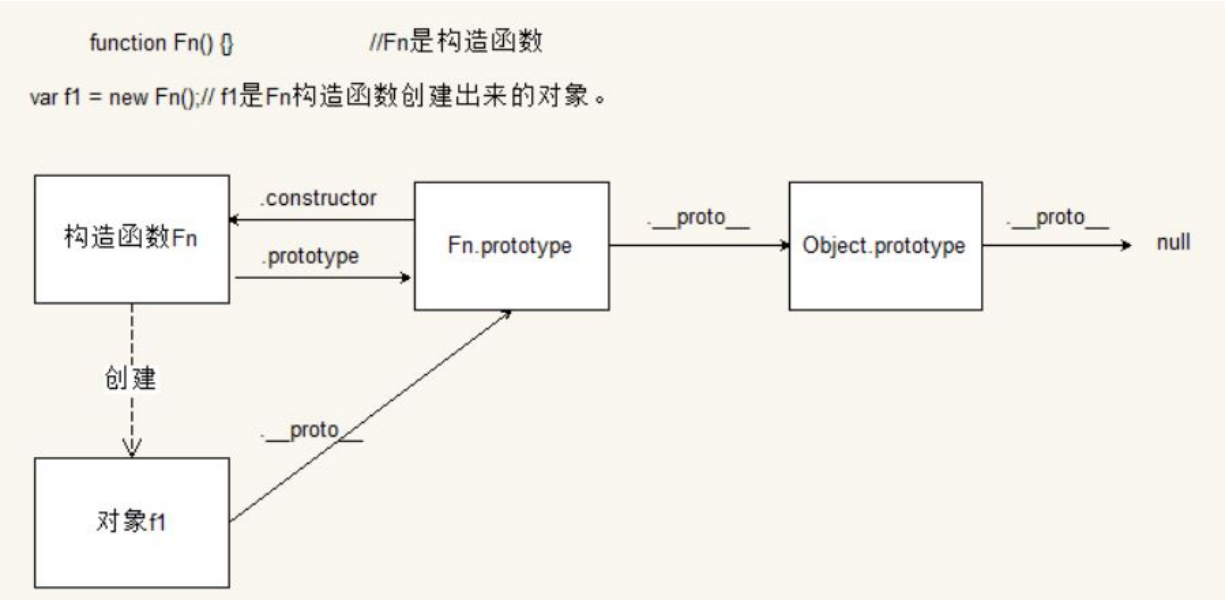
原型上记住三句话：

- 1.所有的函数有一个属性叫prototype,prototype指向默认的原型对象
- 2.默认的原型对象上有个属性叫constructor,它指向于构造函数
- 3.所有的对象上有一个属性叫**proto**,它指向于所属类的原型

原型图



例子



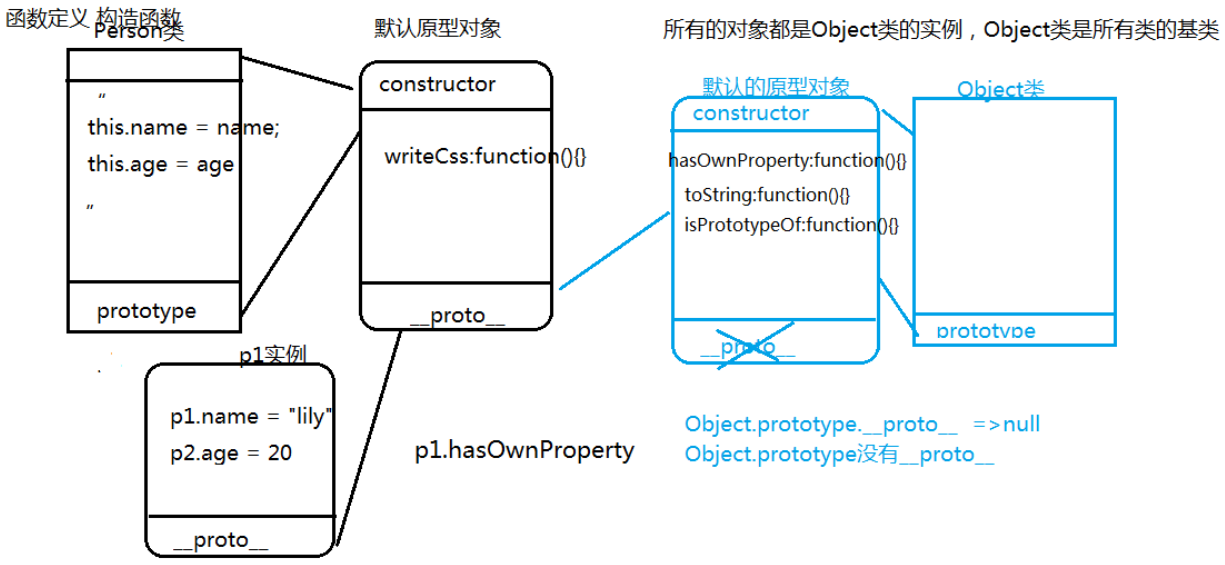
所以对象f1的原型链：f1.**proto**—>Fn.prototype.**proto**—>Object.prototype.**prototype**—>null

查找属性的顺序

1.若实例上有，则不会继续查找，获取的就是实例上私有的属性
2.若实例上没有，则通过**proto**去所属类的原型上查找，若还没找到，则继续通过**proto**查找，一直找到Object类的原型，若还没找到，则是undefined
注意，通过**proto**查找所属类的原型，一级级往父类的原型查找，就形成了原型链 **proto**实现继承的关键

->Object类原型上的属性

- hasOwnProperty 检测是否是私有的属性
- isPrototypeOf 检测一个对象是否在另一个对象的原型链上
- propertyIsEnumerable 属性是否是可枚举 自己查
- toString 转换成字符串



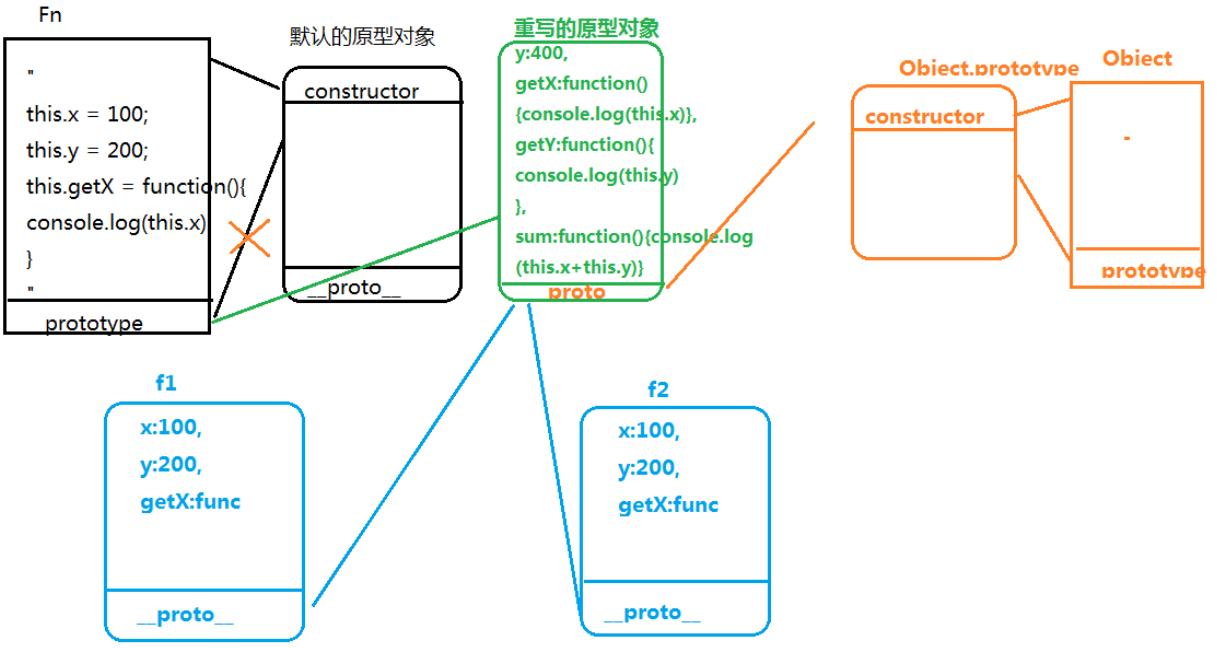
批量的设置原型上的属性

```
Person.prototype = { //重写原型对象
        constructor:Person, //记得添加constructor
        writeCss:function(){
            console.log("writeCss");
        },
        writeJs:function(){
            console.log("writeJs");
        },
        speakEnglish:function(){
            console.log("speakEnglish");
        }
    };
var p1 = new Person("lily",20);
    p1.writeCss();
console.log(p1.constructor);
```

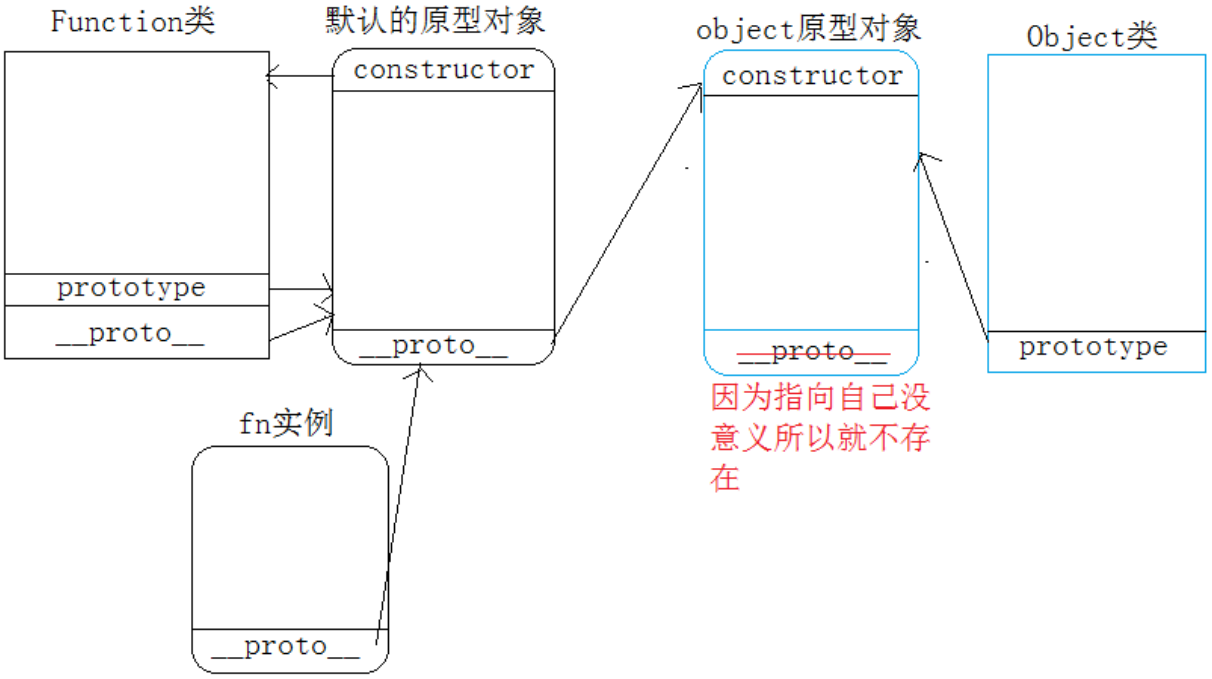
例题

```
function Fn() {
        this.x = 100;
        this.y = 200;
        this.getX = function () {
        console.log(this.x);
        }
    }
Fn.prototype = {
   y: 400,
   getX: function () {
```

```
            console.log(this.x);
        },
    getY: function () {
            console.log(this.y);
        },
    sum: function () {
        console.log(this.x + this.y);
        }
    };
var f1 = new Fn;
var f2 = new Fn;
console.log(f1.getX === f2.getX);//false
console.log(f1.getY === f2.getY);//true
console.log(f1.__proto__.getY === Fn.prototype.getY);//true
console.log(f1.__proto__.getX === f2.getX);//false
console.log(f1.getX === Fn.prototype.getX);//false
    console.log(f1.constructor);//Object类
    console.log(Fn.prototype.__proto__.constructor);//Object类
f1.getX();//100
f1.__proto__.getX();//getX中的this指原型对象  undefined
f2.getY();//this指f2 ->f2.y->200
Fn.prototype.getY();//this指原型对象（Fn.prototype）->Fn.prototype.y ->400
f1.sum();//this指f1 f1.x+f1.y =>300
Fn.prototype.sum();//this指Fn.prototype Fn.prototype.x+Fn.prototype.y =>undefined+400=>NaN
```



深入学习函数



函数三种角色

- 1.普通的函数
  - 1.私有作用域
  - 2.形参赋值
  - 3.变量提升
  - 4.代码从上而下执行
- 2.构造函数（类）
  - 类，实例，原型
- 3.对象
  - 增删改查
- instanceof 一个对象是否在另一个对象的原型链上

call apply bind（原型上的方法）

- Function类原型上方法 实例去调用（fn.call）
- **call apply bind**方法执行的作用：用来改变点前面方法的this关键字
- **call**方法的参数：第一个参数用来改变点前面方法里的this关键字，从第二个参数开始会当成实参传给点前面的方法（传参方式：散列式）
- **apply**方法的参数;第一个参数用来改变点前面的this关键字，第二个参数得是数组或者类数组然后一起传给点前面的方式 (传参方式：打包式)
- **bind**方法 柯里化函数（预处理思想）
  bind方法的第一个参数也是用来改变点前面方法的this关键字,传参方式跟call方法传参一样都是散列式

```
function fn(a,b,c){
```

```
    console.log(this,a,b,c);
}
var res = fn.bind({},10,20);//把所有的参数传给点前面方法后返回一个需要执行的小函数，想要执行时自己手动执行
    res(30);
    res(50);
Function.prototype.bind = function(context){
//把从第二个参数开始所有实参拿到
//实例（this）保存下来
    return function(){
//把小函数的所有参数跟上面参数合并
//实例.apply(context,所有的参数)
   }
}
```

扩展题

- 求数组的最大值和最小值，有几种办法

```
var ary = [10, 5, 25, 23, 3, 2, 17];
//1.假设法
var max = ary[0];
var min = ary[0];
for (var i = 1; i < ary.length; i++) {
   var cur = ary[i];
   /* if(cur>=max){
      max = cur;
     }
    if(cur<=min){
      min = cur;
}*/
cur >= max ? max = cur : cur <= min ? min = cur : null;
    }
console.log(max, min);

//2.Math.max()和Math.min()
var ary = [10, 5, 25, 23, 3, 2, 17];
eval(ary.join()) ;//"10, 5, 25, 23, 3, 2, 17"
//括号运算符 (10, 5, 25, 23, 3, 2, 17)取最后一项 得出17
    //console.log(Math.max(eval(ary.join())));//17
console.log(eval("Math.max(" + ary.join() + ")"));
//eval("Math.max(10, 5, 25, 23, 3, 2, 17)")
console.log(eval("Math.min(" + ary.join() + ")"));

//3.apply传参特点，传的时候是打包式，传过时会一个一个拿出来
console.log(Math.max.apply(null, ary));
console.log(Math.min.apply(null, ary));
```

```
//4.es6扩展运算符
console.log(Math.max(...ary));
console.log(Math.min(...ary))
```

- 去掉一个最大值，去掉最小值，求平均数

```
function average() {
//数组的方法 sort
//1.把arguments转换成数组
var ary = [];
for (var i = 0; i < arguments.length; i++) {
ary.push(arguments[i]);
}
ary.sort(function (a, b) {
return a - b;
});
ary.shift();
ary.pop();
return eval(ary.join("+")) / ary.length.toFixed(2)
}

console.log(average(10, 23, 35, 15, 23, 5));
```

类数组转数组

- 1.对象的数字属性名是从0开始依次递增
- 2.有length属性 这样对象称为类数组 arguments和元素集合
- 类数组和数组克隆的逻辑一样，仅仅是操作的主体不一样，一个是arguments,一个是数组，想要实现类数组间接调用数组的方法，只需改变执行主体，把this改成arguments
- `ary.slice(arguments)`

把类数组转换成数组

- 若原有数组发生改变，类数组间接调用数组方法时，改变是类数组
- var res = Array.prototype.slice.call(arguments,0);
- var res = Array.prototype.push.call(arguments,11,”b”);
- var res = Array.prototype.pop.call(arguments);
- [].sort.call(arguments,function(a,b){return a-b;})

```
function average(){
//console.log(arguments instanceof Array);//false 检测arguments对象是否属于Array类
```

```
//先把类数组转换成数组
var ary = [];
for(var i = 0;i<arguments.length;i++){
    ary.push(arguments[i])
        }
console.log(ary);

//把类数组的最后一项删除，并把删除的内容保存下来
var res = arguments[arguments.length-1];
delete arguments[arguments.length-1];
console.log(arguments,res);

//思考：截取类数组，从索引1~3,返回截取的内容
//由此推算出：类数组若想实现跟数组一样的操作，例如，排序，截取，删除，增加等等
必须得把这些功能全部实现一遍
//想的解决办法:类数组间接的去调用数组的方法

Array.prototype.slice = function(){
//实现克隆数组
//实例用this表示
var arr = [];
for(var i = 0;i<this.length;i++){
   arr.push(this[i]) ;
   }
   return arr;

//由此可以看出，类数组和数组克隆的逻辑一样，仅仅是操作的主体不一样，一个是argum
ents,一个是数组，想要实现类数组间接调用数组的方法，只需改变执行主体，把this改成
arguments
       };
var ary = [12,5,7];
var res = ary.slice();
}
average(10,20,50,2,5,18,9);
```

调试

JS中一旦遇到错误，下面的代码都不再执行

- 遇到错误后需要捕获浏览器异常
- 使用方法：try{异常代码}catch(e){进行另外一种处理方式}

```
try{
   var ary = [].slice.call(oLis);
```

```
}catch(e){//一旦JS代码错误会被catch给捕获，而不会再在浏览器中抛出异常
console.log(e.message);//将错误信息打印出来
//在catch语句中可以对错误信息进行另外一种方式处理
var ary = [];
for(var i = 0;i<oLis.length;i++){
   ary[ary.length] = oLis[i];
  }
}
```

js调试页面

调试代码时 需要手动让浏览器抛出异常，阻止代码运行

```
throw new Error("变量a不存在");
throw new ReferenceError("引用类型错误");
throw new RangeError("范围错误");
throw new TypeError("类型错误");
```

json格式数据

- 他是一种数据格式，不是数据类型
- 作用：方便前后端数据的交互，方便不同语言之间数据传递，相当于一种运输工具
- json格式的数据是在原有的数据类型，制定了一些规范
  - 对于基本数据：字符串必须是双引号
  - 引用数据：可以是数组或对象，不可以是函数，正则，Date等
  - 对象或数组最后一项不能有逗号
  - 对象或数组中字符串都的用双引号抱起来，对象中的属性名用双引号包起来

普通的对象

`var obj = {name:"lily",age:18,gender:'女',hobby:"sleep"};`

json格式的对象

`var obj = {"name":"lily","age":18,"gender":"女","hobby":"sleep"};`

json格式的字符串

`var str = '{"name":"lily","age":18,"gender":"女","hobby":"sleep"}';`

多维数组

```
var ary = [
 {"name":"lily","age":18,"gender":"女","hobby":[1,2,{"a":1}]},
        {"name":"lily","age":18,"gender":"女","hobby":"sleep"},
        {"name":"lily","age":18,"gender":"女","hobby":"sleep"},
        {"name":"lily","age":18,"gender":"女","hobby":"sleep"}
```

```
];
```

window下提供了一个对象，这个对象下有两个方法，对JSON格式的字符串和JSON格式的对象进行转换

JSON.parse()

> 将JSON格式的字符串转换成对象

```
//=>JSON格式的对象和字符串之间的相互转换
// console.log(window.JSON);//=>它是一个对象,里面提供了两个方法:
 parse、stringify
//1、JSON.parse:把字符串(一般都是JSON格式的字符串)转换为JSON格式的对象
var str = '[{"name":"张三","age":28},{"name":"李四","age":25}]';
var jsonAry = JSON.parse(str);
```

```
    //=>把类数组转换为数组(兼容所有的浏览器)
    function toArray(classAry) {...}

    //=>把JSON格式的字符串转换为JSON格式的对象
    function toJSON(str) {
        return "JSON" in window ? JSON.parse(str) : eval('(' +
 str + ')');
    }

    return {
        toArray: toArray,
        toJSON:toJSON
    }
})();
```

JSON.stringify()

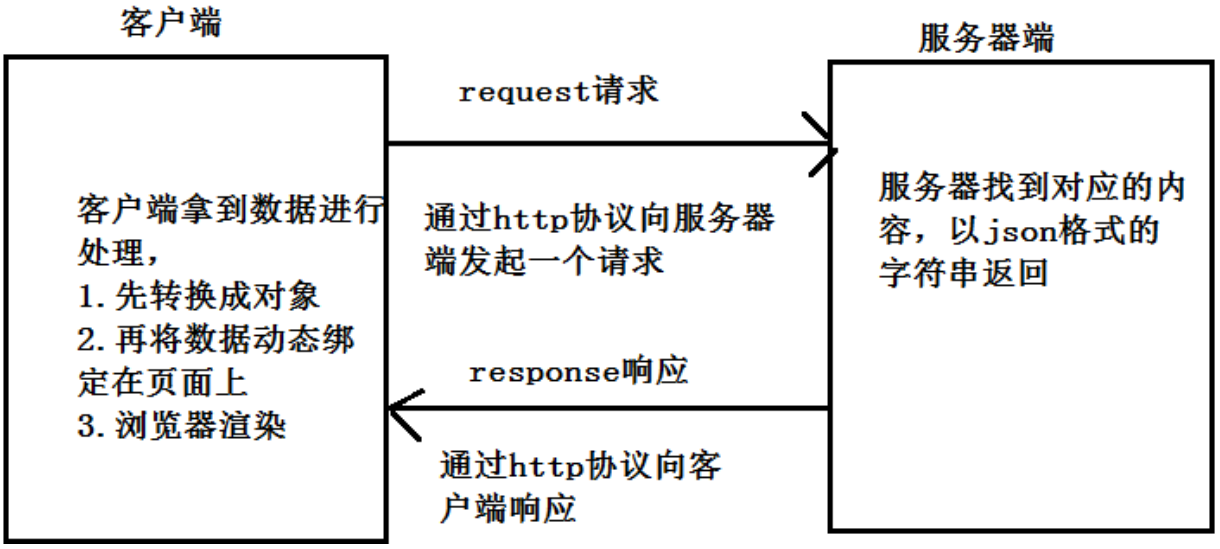
> 将对象转换成JSON格式的字符串

```
//2、JSON.stringify:把对象转化为JSON格式的字符串
// var ary = [{name: '珠峰培训', age: 8}];
// console.log(JSON.stringify(ary));//=>'[{"name":"珠峰培训
 ","age":8}]'

var ary = [{name: '珠峰培训', age: 8, friend: ['阮一峰', '廖雪峰',
'大漠']}];
```

思考题：对多维数组进行深度拷贝（递归调用）

ajax的作用：




```
var xhr = new XMLHttpRequest();
//=>打开请求的URL
//[HTTP METHOD]: HTTP请求方式 GET POST PUT DELETE HEAD...
//[URL]: 请求数据的地址
//[ASYNC]: 设置同步或者异步请求，默认是TRUE异步，我们暂时写 FALSE同步
xhr.open('GET', 'data.txt', false);
//=>监听状态改变,完成数据的获取
xhr.onreadystatechange = function () {
    if (xhr.readyState === 4 && xhr.status === 200) {
        var result = xhr.responseText;
        console.log(result);
    }
};
//=>发送AJAX请求
xhr.send(null);
```

在不刷新页面的基础上，实现数据的局部更新

false同步，true异步

ajax四步骤：

1、创建一个ajax对象

```
var xhr = new XMLHt tpRequest()
```

2、打开一个链接地址（请求一个借口地址）

//第一个参数表示请求的方式 第二个参数表示请求的接口地址 第三参数表示请求是同步还是异步 true异步 false 同步

```
xhr.open("get","./data.json",false);
```

3、监听请求的状态 请求状态码(xhr.readyState)和网络状态码 (xhr.status)

```
xhr.onreadystatechange = function(){
if (xhr.readystate == 4 && /^2\d{2}&/.test(xhr.status)){
```

`//响应成功`
`JSON.parse(xhr.responseText);`
`}`
`}`

4、发送请求

`xhr.send(null)`

动态绑定数据

```
**获取数据**
var xhr = new XMLHttpRequest();
xhr.open("get","./data.json",false);
var data = null;
xhr.onreadystatechange = function(){
      if(xhr.readyState==4 && /^2\d{2}$/.test(xhr.status)){
            data = JSON.parse(xhr.responseText);
        }
}
xhr.send(null);

//绑定数据方法一：字符串拼接
//有几条数据就应该有几个li,每个li的内容是每条数据里title字段的值
/*var str ="";//用来累加li字符串
for(var i = 0;i<data.length;i++){
    var cur = data[i];//每条数据，cur是数组中每个对象
//str+="<li>"+cur.title+"</li>"; //字符串拼接
str+=`<li>${cur.title}</li>`; //es6的模板字符串  变量放在${}里
}
var oUl = document.getElementById("list");
oUl.innerHTML = str;*/

//绑定数据方法二：DOM的方式
var oUl = document.getElementById("list");
var frg = document.createDocumentFragment();//文档碎片
for(var i = 0;i<data.length;i++){
var cur = data[i];
var oLi = document.createElement("li");
oLi.innerHTML = cur.title;
frg.appendChild(oLi);
}
oUl.appendChild(frg);//只会渲染一次
```

`回流` 页面布局发生改变时，页面所有的内容都会重新渲染
`重绘` 某个元素的外观发生变化，只会对这个元素重新渲染

li按照分数从小到大排列

```
<ul id ="list">
<li>98</li>
<li>89</li>
<li>72</li>
<li>86</li>
</ul>
var oUl = document.getElementById("list");
var oLis = oUl.getElementsByTagName("li");
//1.类数组转换成数组
var aLis = [].slice.call(oLis);
//2.排序 sort
aLis.sort(function(a,b){  //让li按照内容进行排序
 return  a.innerHTML - b.innerHTML
});

//3.把排好的数组的每一项重新添加到页面
for(var i = 0;i<aLis.length;i++){
  var cur = aLis[i];
  oUl.appendChild(cur); //若没有则表示添加，若有呢则表示移动
}
//为什么能发现从数组中添加的li是原来页面上li?->DOM映射
```

剩余运算符 reset参数

…arr ：接受剩余参数放在一个数组中返回
function fn (a,b,…ary){};

扩展运算符

[…arguments]

捕获错误 try cashe

- js中一旦遇到错误，下面的代码不会再执行
- 遇到错误后需要捕获浏览器异常
  try{
  var ay = [].
  错误代码包在这里

}cashe(e){//规定必须要传e
一旦js代码错误会被cashe捕获，而不会在浏览器抛出异常.
用e.message将错误打印出来。
在cashe语句中可以对错误信息进行另外一种形式处理
}

throw new Error

调试代码时 需要手动让浏览器抛出异常，阻止代码运行

- throw new Error (“变量不存在”)
- throw new ReferenceError (“引用类型错误”)
- throw new RangeError (“范围错误”)

正则