**call和apply**

1、都是首先让当前操作的这个函数执行，然后把函数中的this改变成第一个参数传递的那个值

[function].call([this],parameter1,parameter2,parameter3...)

[function].apply([this],[parameter1,parameter2,parameter3...])

2、区别只在于传递参数的时候，call的参数可以有无数个，从第二个参数会依次传给fn做参；apply最多只有两个参数。第二个参数是一个集合（数组），它相当于fn的arguments。但是只是写法不一样，效果还是给形参依次赋值

1)在非严格模式下，如果执行的时候一个参数都没有写，默认this是window，例如：fn.call() fn中的this-->window

obj.fn.call(); this-->window

2)在非严格模式下，第一个参数写null或者undefined，this都是window，在严格模式下，你写啥this就是啥，严格模式下如果不写，默认this是undefined

call和apply是定义在Function这个基类的原型上的，而所有的函数都是Function这个类的一个实例，所以所有的函数都可以使用call和apply两个方法

函数：所有的内置类都是函数,所有的function的都是函数

首先明确的是：是call在执行，call的执行的时候使fn也执行了，并且让fn执行的时候this指向call的第一个参数。call和fn里的this不一样

不管写多少个call,都是让最后一个call执行，call方法里面有这样的两步操作：让执行call这个方法的主体this执行，然后改变this这个方法中的this

换句话说，也就是让call方法前面的”.”的前面的东西执行，这里相当于让fn1.call执行,把fn1.call这个函数体拿到,fn1.call其实拿的也是Function.prototype中的call的定义，让它执行，改变里面的this是fn2，也就是让fn2执行

fn1.call(fn2);//执行fn1，让fn1中的this变成fn2

**fn1.call.call(fn2);//两个及以上的call，其实就是让fn2执行**

Function.prototype.call(fn1);//让Function的原型对应的Empty函数执行，里面的this变成fn1，没有任何的意义

Function.prototype.call.call(fn1);//两个及以上的call，fn1执行

arguments里面存储了传进来的参数，它是一个类数组(不是数组)，所以不能直接的使用数组的方法，如果想使用这些方法，首先把它转化为数组

arguments借用数组的slice方法，实现将类数组转换为数组：arguments是类数组，不能直接用数组的方法，但是和数组一样，可以使用方法中的代码，只需要让方法中的this变成arguments，就可以实现了

var ary = Array.prototype.slice.call(arguments, 0);

var ary = [].slice.call(arguments, 0); //[].slice==Array.prototype.slice

DOM集合借用数组类上的方法的原因就是：

1、它也是以数字为属性（索引）来读取里面的值的 eles[i]。

2、有length属性，边界。

凡是具备以上两点的，都可以借用Array类上的方法。

也把具备以上两点的对象称之“类数组”

**try、catch捕获异常信息，并且不影响下面代码的执行**

try {

//我们执行的JS代码

} catch (e) {

//如果JS代码执行的时候报错了，这里可以获取报错信息

//e就是存储错误信息的对象

//e.code 错误码

//e.message 报错信息

} finally{

//通常不加：不管try中的代码是否报错，finally中的代码都会执行

}

在JS中，不做任何处理的情况下，当前代码报错，下面的所有代码都不再执行了。而用try、catch捕获异常后，即使报错，下面的代码也可以正常的执行

JS中可以手动的抛出异常错误

**throw new Error("错误信息");**

try {

console.log(a);

} catch (e) {

throw new Error("浏览器繁忙，请稍后再试！");

}

console.log(a + 100);

通常也用try、catch检测JS兼容问题：如果不兼容的一个方法执行会报错，我们可以监听是否报错，报错的话就是不兼容，我们想其他的办法解决

**JSON只是一种数据格式，只是一种约定的形式：把普通对象的属性名用""包起来，这个对象是JSON格式的对象**

JS中基本上都是用JSON来存储想要的数据

一个普通的对象

var obj = {name: "李晓慧",age: 18};

JSON格式的对象

var obj = {"name": "李晓慧","age": 18};

操作JSON格式的数据我们通常按照下面的来操作：

1、将json格式的对象转换为json格式的字符串:**JSON.stringify();**

console.log(JSON.stringify(jsonAry));

2、将json格式的字符串转化为json格式的对象：**JSON.parse()**

在IE6、7下这个方法是不存在的：eval("("+jsonStr+")") 千万不要忘记手动加一个小括号

console.log(eval("(" + jsonStr + ")"));

console.log(JSON.parse(jsonStr));

**DOM集合的特点：**

第一、它是活的有生命的，随着文档结构的变化而变化

第二点、每次运行相同的表达式得到的集合是相同的DOM结构“影射”

第三点、DOM树的变化会（DOM结构）会自动影响DOM集合（和第一个是相同的意思），但DOM集合不能被修改

在DOM中，开始有3个li，后面添加了10个li(不管用动态创建还是innerHTML的方式)，添加后都不需要重新的获取，就可以得到最新的13个li，这就是DOM的映射机制

**DOM中的回流、重绘**

**回流：**只要DOM结构发生改变(增加、删除、修改、位置调整...)，浏览器都要重新渲染结构

**重绘：**样式发生改变(颜色、字体...)

**数据绑定的四种方法：**

1、innerHTML拼接字符串：首先把原有页面中的内容当做字符串拿出来，然后和最新的字符串进行拼接，最后把拼接完成的从新的放到页面中，浏览器开始重新解析有效的html标签

弊端：之前给元素绑定的事件等内容，重新添加一次都消失了，需要重新的获取绑定

优势：把所有数据拼接好后，整体添加到页面中，只重新渲染1次

2、动态创建的方式：不改变原有的html元素，依次用DOM方法一个个的添加到我们的页面中,原来的元素不产生任何的影响，而且DOM本身存在映射的机制，不需要重新的获取，之前获取的自动会跟着改变

优势：原有的html元素不需要改变

弊端：添加一次就要重新渲染一次，非常的耗性能

3、document.createDocumentFragment创建文档碎片：我们用DOM动态创建的方式创建10个li，但是不是每一次创建一个就添加到页面一个，而是创建一个就放在文档碎片中，都创建完，把文档碎片统一次添加到页面中,这样只重新渲染一次

4、如果用innerHTML，如果想不修改原有的三个，我不往ul1中添加即可，创建一个ul元素，然后把内容添加到这个ul元素中，再用appendChild把这个元素添加到ul1中

**表格中常用的属性方法：**

var oTab = document.getElementById("tab");//获取一个table标签

var tHead = oTab.tHead;//获取table中的唯一一个tHead

var tBody = oTab.tBodies[0];//获取table中所有的tBody，它是一个类数组集合(HTMLCollection)，如果只想用第一个tBody，需要加[0]

var oRows = tBody.rows; 获取tBody中的所有的行，是一个类数组集合（HTMLCollection）

var oRow=oRows[0] 第一行

var oCells=oRow.cells; 获取第一行中的所有列，是一个类数组集合（HTMLCollection）

oCells[0] 第一列

oCells[0].innerHTML 第一列中的内容