由于本人学艺不精，有很多知识点，是从各位大神中的博客中摘取出来的，如果构成侵权，请告知，本文只做分享使用，希望对看到你有所帮助，如有错误或者更好的答案，欢迎指正。

js篇

==js基本数据类型==

5中基本数据类型：null、undefined、string、number、boolean

==setTimeout和setInterval==

《JavaScript高级程序设计》这本书里面，介绍了很多关于setTimeout函数的神奇使用，今天来介绍下第一个——使用setTimeout代替setInterval进行间歇调用。

“在开发环境下，很少使用间歇调用（setInterval），原因是后一个间歇调用很可能在前一个间歇调用结束前启动”。

这话怎么理解呢？

首先我们来看一下一般情况下的setInterval函数的使用，以及如何使用setTimeout代替setInterval

```
var executeTimes = 0;
var intervalTime = 500;
var intervalId = null;

// 放开下面的注释运行setInterval的Demo
intervalId = setInterval(intervalFun,intervalTime);
// 放开下面的注释运行setTimeout的Demo
// setTimeout(timeOutFun,intervalTime);

function intervalFun(){
    executeTimes++;
    console.log("doIntervalFun——"+executeTimes);
    if(executeTimes==5){
        clearInterval(intervalId);
    }
}

function timeOutFun(){
    executeTimes++;
    console.log("doTimeOutFun——"+executeTimes);
    if(executeTimes<5){
        setTimeout(arguments.callee,intervalTime);
    }
}
```

代码比较简单，我们只是在setTimeout的方法里面又调用了一次setTimeout，就可以达到间歇调用的目的。

重点来了，为什么作者建议我们使用setTimeout代替setInterval呢？setTimeout式的间歇调用和传统的setInterval间歇调用有什么区别呢？

- 区别在于，setInterval间歇调用，是在前一个方法执行前，就开始计时，比如间歇时间是500ms，那么不管那时候前一个方法是否已经执行完毕，都会把后一个方法放入执行的序列中。这时候就会发生一个问题，假如前一个方法的执行时间超过500ms，加入是1000ms，那么就意味着，前一个方法执行结束后，后一个方法马上就会执行，因为此时间歇时间已经超过500ms了。

```
var executeTimes = 0;
var intervalTime = 500;
var intervalId = null;
var oriTime = new Date().getTime();

// 放开下面的注释运行setInterval的Demo
// intervalId = setInterval(intervalFun,intervalTime);
// 放开下面的注释运行setTimeout的Demo
setTimeout(timeOutFun,intervalTime);

function intervalFun(){
    executeTimes++;
    var nowExecuteTimes = executeTimes;
    var timeDiff = new Date().getTime() - oriTime;
    console.log("doIntervalFun——"+nowExecuteTimes+", after " + timeDiff + "ms");
    var delayParam = 0;
    sleep(1000);
    console.log("doIntervalFun——"+nowExecuteTimes+" finish !");
    if(executeTimes==5){
        clearInterval(intervalId);
    }
}

function timeOutFun(){
    executeTimes++;
    var nowExecuteTimes = executeTimes;
    var timeDiff = new Date().getTime() - oriTime;
    console.log("doTimeOutFun——"+nowExecuteTimes+", after " + timeDiff + "ms");
    var delayParam = 0;
    sleep(1000);
    console.log("doTimeOutFun——"+nowExecuteTimes+" finish !");
    if(executeTimes<5){
        setTimeout(arguments.callee,intervalTime);
    }
}

function sleep(sleepTime){
    var start=new Date().getTime();
    while(true){
        if(new Date().getTime()-start>sleepTime){
            break;
        }
    }
```

```
}
```

（这里使用大牛提供的sleep函数来模拟函数运行的时间）

执行setInterval的Demo方法，看控制台

```
doIntervalFun——1, after 500ms
VM2854:19 doIntervalFun——1 finish !
VM2854:16 doIntervalFun——2, after 1503ms
VM2854:19 doIntervalFun——2 finish !
VM2854:16 doIntervalFun——3, after 2507ms
VM2854:19 doIntervalFun——3 finish !
VM2854:16 doIntervalFun——4, after 3510ms
VM2854:19 doIntervalFun——4 finish !
VM2854:16 doIntervalFun——5, after 4512ms
VM2854:19 doIntervalFun——5 finish !
```

可以发现，fun2和fun1开始的间歇接近1000ms，刚好就是fun1的执行时间，也就意味着fun1执行完后fun2马上就执行了，和我们间歇调用的初衷背道而驰。

我们注释掉setInterval的Demo方法，放开setTimeout的Demo方法，运行，查看控制台

```
doTimeOutFun——1, after 500ms
VM2621:32 doTimeOutFun——1 finish !
VM2621:29 doTimeOutFun——2, after 2001ms
VM2621:32 doTimeOutFun——2 finish !
VM2621:29 doTimeOutFun——3, after 3503ms
VM2621:32 doTimeOutFun——3 finish !
VM2621:29 doTimeOutFun——4, after 5004ms
VM2621:32 doTimeOutFun——4 finish !
VM2621:29 doTimeOutFun——5, after 6505ms
VM2621:32 doTimeOutFun——5 finish !
```

这下终于正常了，fun1和fun2相差了1500ms = 1000 + 500，fun2在fun1执行完的500ms后执行。

==闭包==

闭包的定义：

在js高级教程中的定义是：有权访问另一个函数作用域中的变量的函数。通俗地讲，如果一个函数执行完以后，这个函数中还存在一部分在内存当中，没有被垃圾回收机制回收，这个函数就称为闭包。

闭包的作用

1.实现私有变量

如果我们写一个函数，里面有一个name值，我们可以允许任何人访问这个name属性，但是只有少部分人，可以修改这个name属性，我们就可以使用闭包，可以在setName值中，写哪些人具有修改的权限。

```
var person = function(){
    //变量作用域为函数内部，外部无法访问，不会与外部变量发生重名冲突
    var name = "FE";
    return {
      //管理私有变量
       getName : function(){
           return name;
       },
       setName : function(newName){
           name = newName;
       }
    }
};
```

2.数据缓存

假如说我们执行一个计算量很大函数，返回一个值，而这个值在其他函数中还有应用，这种情况下使用闭包，可以将该数据保存在内存中，供其他的函数使用（这是在其他博客中看到的，具体不是很清楚，如果有兴趣，可以自己查阅相关文献）。

缺点：

造成内存消耗过大，如果处理不当，会造成内存泄漏

==事件防抖和事件节流==

- 事件防抖：debounce的作用是在让在用户动作停止后延迟x ms再执行回调。
- 事件节流：throttle的作用是在用户动作时没隔一定时间（如200ms）执行一次回调。

使用原因：

浏览器的一些事件，如：resize,scroll,keydown,keyup,keypress,mousemove等。这些事件触发频率太过频繁，绑定在这些事件上的回调函数会不停的被调用。这样浏览器的目的是为了保证信息的一致性，而对于我们来说就是一种资源的浪费了。

代码

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>test</title>

</head>
<style>
    #mydiv{
        border-bottom: 1px solid;
        width: 100%;
        height: 300px;
        background-color: #999999;
```

```
    }
    .mouseMoveDisplay{
        border-top: 1px solid rosybrown;
        width: 100%;
        height: 200px;
        background-color: #bbbbbb;

    }
</style>
<body>
<div id="mydiv"></div>
<table class="mouseMoveDisplay">
    <tr>
        <th>
            一共移动了<span class="mouseMove all">0</span>次
        </th>

    </tr>

    <tr>
        <th>
            防抖移动了<span class="mouseMove debounce">0</span>次
        </th>

    </tr>

    <tr>
        <th>
            节流移动了<span class="mouseMove throttle">0</span>次
        </th>
    </tr>




</table>

<script>
    //防抖模式
    function debounceFunction(fn,delay){

        var delay=delay||200;
        var timer;
        return function(){
            var th=this;
            var args=arguments;
            if (timer) {
                clearTimeout(timer);
            }
            timer=setTimeout(function () {
                timer=null;
                fn.apply(th,args);

            }, delay);
```

```
        };
    }


    //事件节流
    function throttleFunction(fn,interval){
        var last;
        var timer;
        var interval=interval||200;
        return function(){
            var th=this;
            var args=arguments;
            var now=+new Date();
            if(last&&now-last<interval){
                clearTimeout(timer);
                timer=setTimeout(function(){
                    last=now;
                    fn.apply(th,args);
                },interval);
            }else{
                last=now;
                fn.apply(th,args);
            }
        }
    }

    var mydiv=document.getElementById("mydiv");

    //所有的移动次数
    var allCount = 0;//所有的次数
    var all = document.getElementsByClassName('all')[0];
    mydiv.addEventListener("mousemove",function () {
        allCount++;
        all.innerHTML = allCount;
    })


    //防抖的移动次数
    var debounceCount = 0;//所有的次数
    var debounce = document.getElementsByClassName('debounce')[0];
    mydiv.addEventListener("mousemove",debounceFunction(function () {
        debounceCount++;
        debounce.innerHTML = debounceCount;
    }))


    //节流的移动次数
    var throttleCount = 0;//所有的次数
    var throttle = document.getElementsByClassName('throttle')[0];
    mydiv.addEventListener("mousemove",throttleFunction(function () {
        throttleCount++;
        throttle.innerHTML = throttleCount;

    }))
```

```
120
121
122
123
124
125 </script>
126 </body>
127
128 </html>
```

==数组中的forEach和map的区别==

大多数情况下，我们都要对数组进行遍历，然后经常用到的两个方法就是forEach和map方法。 先来说说它们的共同点

相同点

- 都是循环遍历数组中的每一项
- forEach和map方法里每次执行匿名函数都支持3个参数，参数分别是item（当前每一项），index（索引值），arr（原数组）
- 匿名函数中的this都是指向window
- 只能遍历数组
- 都不会改变原数组

区别

==map方法==

1.map方法返回一个新的数组，数组中的元素为原始数组调用函数处理后的值。 2.map方法不会对空数组进行检测，map方法不会改变原始数组。 3.浏览器支持：chrome、Safari1.5+、opera都支持，IE9+,

```
1 array.map(function(item,index,arr){},thisValue)
2
3 var arr = [0,2,4,6,8];
4 var str = arr.map(function(item,index,arr){
5     console.log(this); //window
6     console.log("原数组arr:",arr); //注意这里执行5次
7     return item/2;
8 },this);
9 console.log(str);//[0,1,2,3,4]
```

若arr为空数组，则map方法返回的也是一个空数组。

==forEach方法==

1.forEach方法用来调用数组的每个元素，将元素传给回调函数 2.forEach对于空数组是不会调用回调函数的。

```
Array.forEach(function(item,index,arr){},this)
var arr = [0,2,4,6,8];
var sum = 0;
var str = arr.forEach(function(item,index,arr){
    sum += item;
    console.log("sum的值为：",sum); //0 2 6 12 20
    console.log(this); //window
},this)
console.log(sum);//20
console.log(str); //undefined
```

无论arr是不是空数组，forEach返回的都是undefined。这个方法只是将数组中的每一项作为callback的参数执行一次。

==for in和for of的区别==

遍历数组通常使用for循环，ES5的话也可以使用forEach，ES5具有遍历数组功能的还有map、filter、some、every、reduce、reduceRight等，只不过他们的返回结果不一样。但是使用foreach遍历数组的话，使用break不能中断循环，使用return也不能返回到外层函数。

```
Array.prototype.method=function(){
    console.log(this.length);
}
var myArray=[1,2,4,5,6,7]
myArray.name="数组"
for (var index in myArray) {
  console.log(myArray[index]);
}
```

使用for in 也可以遍历数组，但是会存在以下问题：

1.index索引为字符串型数字，不能直接进行几何运算

2.遍历顺序有可能不是按照实际数组的内部顺序

3.使用for in会遍历数组所有的可枚举属性，包括原型。例如上栗的原型方法method和name属性

所以for in更适合遍历对象，不要使用for in遍历数组。

那么除了使用for循环，如何更简单的正确的遍历数组达到我们的期望呢（即不遍历method和name），ES6中的for of更胜一筹.

```
Array.prototype.method=function(){
    console.log(this.length);
}
var myArray=[1,2,4,5,6,7]
myArray.name="数组";
for (var value of myArray) {
  console.log(value);
}
```

记住，for in遍历的是数组的索引（即键名），而for of遍历的是数组元素值。

for of遍历的只是数组内的元素，而不包括数组的原型属性method和索引name

遍历对象 通常用for in来遍历对象的键名

```
Object.prototype.method=function(){
    console.log(this);
}
var myObject={
    a:1,
    b:2,
    c:3
}
for (var key in myObject) {
  console.log(key);
}
```

for in 可以遍历到myObject的原型方法method,如果不想遍历原型方法和属性的话，可以在循环内部判断一下,==hasOwnPropery==方法可以判断某属性是否是该对象的实例属性

```
for (var key in myObject) {
    if ( myObject.hasOwnProperty(key)){
        console.log(key);
    }
}
```

同样可以通过ES5的Object.keys(myObject)获取对象的实例属性组成的数组，不包括原型方法和属性。

```
Object.prototype.method=function(){
    console.log(this);
}
var myObject={
    a:1,
    b:2,
    c:3
}
Object.keys(myObject).forEach(function(key,index){
    console.log(key,myObject[key])

})
```

==实现EventEmitter方法==

```
class EventEmitter {
    constructor() {
        this.events = {};
    }

```

```
    on(eventName, fn) {
        let fnList = this.events[eventName] || [];
        fnList.push(fn)
        if (eventName) {
            this.events[eventName] = fnList;
        }
    }

    emit(eventName, ...agr) {
        let funcs = this.events[eventName];
        if (funcs && funcs.length) {
            for (let j = 0; j < funcs.length; j++) {
                funcs[j](...agr);
            }
        }
    }
    off(eventName, fn) {
        let funcs = this.events[eventName];
        if (fn) {
            this.events[eventName].splice(fn, 1);
        } else {
            delete this.events[eventName]
        }
    }
}
```

==let、var、const区别==

var

第一个就是作用域的问题，var不是针对一个块级作用域，而是针对一个函数作用域。举个例子：

```
function runTowerExperiment(tower, startTime) {
  var t = startTime;

  tower.on("tick", function () {
    ... code that uses t ...
  });
  ... more code ...
}
```

这样是没什么问题的，因为回调函数中可以访问到变量t，但是如果我们在回调函数中再次命名了变量t呢？

```
function runTowerExperiment(tower, startTime) {
  var t = startTime;

  tower.on("tick", function () {
    ... code that uses t ...
    if (bowlingBall.altitude() <= 0) {
      var t = readTachymeter();
      ...
    }
  });
  ... more code ...
}
```

后者就会将前者覆盖。

第二个就是循环的问题。 看下面例子：

```
var messages = ["Meow!", "I'm a talking cat!", "Callbacks are fun!"];

for (var i = 0; i < messages.length; i++) {
    setTimeout(function () {
        document.write(messages[i]);
    },i*1500);
}
```

输出结果是：undefined 因为for循环后，i置为3，所以访问不到其值。

let

为了解决这些问题，ES6提出了let语法。let可以在{},if,for里声明，其用法同var，但是作用域限定在块级。但是javascript中不是没有块级作用域吗？这个我们等会讲。还有一点很重要的就是let定义的变量==不存在变量提升==。

变量提升 这里简单提一下什么叫做变量提升。

```
var v='Hello World';
(function(){
    alert(v);
    var v='I love you';
})()
```

上面的代码输出结果为：undefined。

为什么会这样呢？这就是因为变量提升，变量提升就是把变量的声明提升到函数顶部，比如：

```
(function(){
    var a='One';
    var b='Two';
    var c='Three';

})()
```

实际上就是：

```
(function(){
    var a,b,c;
    a='One';
    b='Two';
    c='Three';
})()
```

所以我们刚才的例子实际上是：

```
var v='Hello World';
(function(){
    var v;
    alert(v);
    v='I love you';
})()
```

所以就会返回undefined啦。

这也是var的一个问题，而我们使用let就不会出现这个问题。因为它会报语法错误:

```
{
    console.log( a );   // undefined
    console.log( b );   // ReferenceError!
    var a;
    let b;
}
```

再来看看let的块级作用域。

```
function getVal(boo) {
    if (boo) {
        var val = 'red'
        // ...
        return val
    } else {
        // 这里可以访问 val
        return null
    }
    // 这里也可以访问 val
}

```

而使用let后：

```
function getVal(boo) {
    if (boo) {
        let val = 'red'
        // ...
        return val
    } else {
        // 这里访问不到 val
        return null
    }
    // 这里也访问不到 val
}
```

同样的在for循环中：

```
function func(arr) {
    for (var i = 0; i < arr.length; i++) {
        // i ...
    }
    // 这里访问得到i
}

```

使用let后：

```
function func(arr) {
    for (let i = 0; i < arr.length; i++) {
        // i ...
    }
    // 这里访问不到i
}
```

也就是说，==let只能在花括号内部起作用==。

const

再来说说const，==const代表一个值的常量索引==。

```
const aa = 11;
alert(aa) //11
aa = 22;
alert(aa) //11
```

但是常量的值在垃圾回收前永远不能改变，所以需要谨慎使用。

还有一条需要注意的就是和其他语言一样，==常量的声明必须赋予初值==。即使我们想要一个undefined的常量，也需要声明：

```
const a = undefined;
```

块级作用域 最后提一下刚才说到的块级作用域。

在之前，javascript是没有块级作用域的，我们都是通过()来模拟块级作用域。

```
(function(){
 //这里是块级作用域
})();
```

但是在ES6中，{}就可以直接代码块级作用域。所以{}内的内容是不可以在{}外访问得到的。

我们可以看看如下代码：

```
if (true) {
    function foo() {
        document.write( "1" );
    }
}
else {
    function foo() {
        document.write( "2" );
    }
}

foo();      // 2
```

在我们所认识的javascript里，这段代码的输出结果为2。这个叫做函数声明提升，不仅仅提升了函数名，也提升了函数的定义。如果你基础不扎实的话，可以看看这篇文章：深入理解javascript之IIFE

但是在ES6里，这段代码或抛出ReferenceErroe错误。因为{}的块级作用域，导致外面访问不到foo()，也就是说函数声明和let定义变量一样，都被限制在块级作用域中了。

==事件循环==

从promise、process.nextTick、setTimeout出发，谈谈Event Loop中的Job queue

简要介绍：谈谈promise.resove,setTimeout,setImmediate,process.nextTick在EvenLoop队列中的执行顺序

1.问题的引出

event loop都不陌生，是指主线程从“==任务队列==”中循环读取任务，比如

例1：

```
setTimeout(function(){console.log(1)},0);

console.log(2)

//输出2,1
```

在上述的例子中，我们明白首先执行主线程中的同步任务，当主线程任务执行完毕后，再从event loop中读取任务，因此先输出2，再输出1。

event loop读取任务的先后顺序，取决于任务队列（Job queue）中对于不同任务读取规则的限定。比如下面一个例子：

例2：

```
setTimeout(function () {
  console.log(3);
}, 0);

Promise.resolve().then(function () {
  console.log(2);
});
console.log(1);
//输出为  1  2 3
```

先输出1，没有问题，因为是同步任务在主线程中优先执行，这里的问题是setTimeout和Promise.then任务的执行优先级是如何定义的。

2 . Job queue中的执行顺序

在Job queue中的队列分为两种类型：==macro-task和microTask==。我们举例来看执行顺序的规定，我们设

macro-task队列包含任务: a1, a2 , a3 micro-task队列包含任务: b1, b2 , b3

执行顺序为，首先执行marco-task队列开头的任务，也就是 a1 任务，执行完毕后，在执行micro-task队列里的所有任务，也就是依次执行b1, b2 , b3，执行完后清空micro-task中的任务，接着执行marco-task中的第二个任务，依次循环。

了解完了macro-task和micro-task两种队列的执行顺序之后，我们接着来看，真实场景下这两种类型的队列里真正包含的任务（我们以node V8引擎为例），在node V8中，这两种类型的真实任务顺序如下所示：

macro-task（宏任务）队列真实包含任务：

script(主程序代码),setTimeout, setInterval, setImmediate, I/O, UI rendering

micro-task（微任务）队列真实包含任务：

process.nextTick, Promises, Object.observe, MutationObserver

由此我们得到的执行顺序应该为：

==script(主程序代码)—>process.nextTick—>Promises...———>setTimeout———>setInterval———>setImmediate———> I/O———>UI rendering==

在ES6中macro-task队列又称为ScriptJobs，而micro-task又称PromiseJobs

3 . 真实环境中执行顺序的举例

（1）setTimeout和promise

例3:

```
setTimeout(function () {
  console.log(3);
}, 0);

Promise.resolve().then(function () {
  console.log(2);
});

console.log(1);
```

我们先以第1小节的例子为例，这里遵循的顺序为：

```
script(主程序代码)—>promise—>setTimeout
对应的输出依次为：1 —>2———>3
```

（2）process.nextTick和promise、setTimeout

例子4：

```
setTimeout(function(){console.log(1)},0);

new Promise(function(resolve,reject){
   console.log(2);
   resolve();
}).then(function(){console.log(3)
}).then(function(){console.log(4)});

process.nextTick(function(){console.log(5)});

console.log(6);
//输出2,6,5,3,4,1
```

这个例子就比较复杂了，这里要注意的一点在定义promise的时候，promise构造部分是同步执行的，这样问题就迎刃而解了。

首先分析Job queue的执行顺序：

==script(主程序代码)———>process.nextTick———>promise———>setTimeout==

I) 主体部分： 定义promise的构造部分是同步的， 因此先输出2 ，主体部分再输出6（同步情况下，就是严格按照定义的先后顺序）

II)process.nextTick: 输出5

III）promise： 这里的promise部分，严格的说其实是promise.then部分，输出的是3,4

IV) setTimeout ： 最后输出1

综合的执行顺序就是： 2———>6———>5———>3———>4———>1

（3）更复杂的例子

```
setTimeout(function(){console.log(1)},0);

new Promise(function(resolve,reject){
   console.log(2);
   setTimeout(function(){resolve()},0)
}).then(function(){console.log(3)
}).then(function(){console.log(4)});

process.nextTick(function(){console.log(5)});

console.log(6);

//输出的是  2 6 5 1 3 4
```

种情况跟我们（2）中的例子，区别在于promise的构造中，没有同步的resolve，因此promise.then在当前的执行队列中是不存在的，只有promise从pending转移到resolve，才会有then方法，而这个resolve是在一个setTimout时间中完成的，因此3,4最后输出。

==浏览器内核==

1. IE浏览器内核：Trident内核，也是俗称的IE内核；
2. Chrome浏览器内核：统称为Chromium内核或Chrome内核，以前是Webkit内核，现在是Blink内核；
3. Firefox浏览器内核：Gecko内核，俗称Firefox内核；
4. Safari浏览器内核：Webkit内核；
5. Opera浏览器内核：最初是自己的Presto内核，后来是Webkit，现在是Blink内核；
6. 360浏览器、猎豹浏览器内核：IE+Chrome双内核；
7. 搜狗、遨游、QQ浏览器内核：Trident（兼容模式）+Webkit（高速模式）；
8. 百度浏览器、世界之窗内核：IE内核；
9. 2345浏览器内核：以前是IE内核，现在也是IE+Chrome双内核；

==懒加载的原理==

- 原理：先将img标签中的src链接设为同一张图片（空白图片），将其真正的图片地址存储再img标签的自定义属性中（比如data-src）。当js监听到该图片元素进入可视窗口时，即将自定义属性中的地址存储到src属性中，达到懒加载的效果。
- 这样做能防止页面一次性向服务器响应大量请求导致服务器响应慢，页面卡顿或崩溃等问题。
  代码实现

既然懒加载的原理是基于判断元素是否出现在窗口可视范围内，首先我们写一个函数判断元素是否出现在可视范围内：

```
function isVisible($node){
    var winH = $(window).height(),
        scrollTop = $(window).scrollTop(),
        offSetTop = $(window).offSet().top;
    if (offSetTop < winH + scrollTop) {
        return true;
    } else {
        return false;
    }
}
```

再添加上浏览器的事件监听函数，让浏览器每次滚动就检查元素是否出现在窗口可视范围内：

```
$(window).on("scroll", function{
    if (isVisible($node)){
        console.log(true);
    }
})
```

我们已经很接近了，现在我们要做的是，让元素只在第一次被检查到时打印true，之后就不再打印了

```
var hasShowed = false;
$(window).on("sroll",function{
    if (hasShowed) {
        return;
    } else {
        if (isVisible($node)) {
            hasShowed = !hasShowed;
            console.log(true);
        }
    }
})
```

咦，我们好像已经实现了懒加载。

懒加载实例

- 1.简单懒加载:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        img{
            display: block;
            max-height: 300px;
        }
    </style>
```

```
</head>
<body>
    <div class="container">
        <h1>懒加载页面</h1>
        <img src="1.png" data-src='1.jpg' alt="">
        <img src="1.png" data-src='2.jpg' alt="">
        <img src="1.png" data-src='3.jpg' alt="">
        <img src="1.png" data-src='4.jpg' alt="">
        <img src="1.png" data-src='5.jpg' alt="">
        <img src="1.png" data-src='6.jpg' alt="">
        <img src="1.png" data-src='7.jpg' alt="">
        <img src="1.png" data-src='8.jpg' alt="">
        <img src="1.png" data-src='9.jpg' alt="">
    </div>
</body>
</html>
<script>
    var scrollTop = window.scrollY;
    var imgs = Array.from(document.querySelectorAll('img'));

    lazyLoad();

    window.onscroll = () => {
        scrollTop = window.scrollY;
        lazyLoad();
    }
    function lazyLoad(){
        imgs.forEach((item,index)=>{
            if( item.offsetTop < window.innerHeight + scrollTop ){
                console.log(item.offsetTop)
                item.setAttribute('src',item.dataset.src)
            }
        })
    }
</script>
```

这里有坑请注意!!! 而且这个问题不好百度 - . - 如果复制上面的代码,首次加载进页面发现所有图片均已经加载完毕,没有实现懒加载的效果 因为函数调用时img.onload没有完成,img元素没有高度!!! 解决办法是在外层套一个window.onload

```
window.onload = function(){
        lazyLoad();
}
```

- 2.函数节流throttle懒加载
  推荐!!!高频滚动模式下, 每隔一段时间才会实现渲染~~
  实现原理是 加入一个开关变量, 控制每隔固定的一段时间,函数才可能被触发~

```
window.onload = function(){
    var scrollTop = window.scrollY;
    var imgs = Array.from(document.querySelectorAll('img'));
```

```
    lazyLoad();

    //函数节流模式
    var canRun = true;
    window.onscroll = () => {
        if( !canRun ){
            return
        }
        canRun = false;
        setTimeout(()=>{
            scrollTop = window.scrollY;
            lazyLoad();
            canRun = true;
        },1000)

    }

    function lazyLoad(){
        imgs.forEach((item,index)=>{
            if( item.offsetTop < window.innerHeight + scrollTop ){
                console.log(item.offsetTop)
                item.setAttribute('src',item.dataset.src)
            }
        })
    }
}
```

为了逻辑清晰，打包成函数调用:

```
window.onload = function(){

    var scrollTop = window.scrollY;
    var imgs = Array.from(document.querySelectorAll('img'));
    lazyLoad();
    let canRun = true;//开关变量用于函数节流
    window.addEventListener('scroll',throttle(lazyLoad,500));



    //定义懒加载函数 ， 从上到下懒加载 ， 从下到上也是懒加载
    function lazyLoad(){
        imgs.forEach((item,index)=>{
            if( scrollTop===0 && item.dataset.src !== '' && item.offsetTop < window.innerHeight + scrollTop ){
                alert()
                item.setAttribute('src',item.dataset.src)
                item.setAttribute('data-src','')
            }else if( item.dataset.src !== '' && item.offsetTop < window.innerHeight + scrollTop && item.offsetTop > scrollTop ){
                item.setAttribute('src',item.dataset.src)

                item.setAttribute('data-src','')
```

```
            }
        })
    }


    //定义函数节流函数
    function throttle(fun,delay){
        return function(){
            // fun();
            if( !canRun ){
                return
            }
            console.log('!!!')
            canRun = false;
            setTimeout(()=>{
                scrollTop = window.scrollY;
                fun(imgs);
                canRun = true
            },delay)
        }
    }

}
```

- 3.函数防抖debounce
  原理是设置clearTimeout和setTimeout,的dalayTime控制一个事件如果频繁触发,将只会执行最近的一次... 可以用在用户注册时候的手机号码验证和邮箱验证。只有等用户输入完毕后，前端才需要检查格式是否正确，如果不正确，再弹出提示语。以下还是以页面元素滚动监听的例子

效果:一直滑动的时候,将不会有图片加载, 停下后300ms会加载

```
window.onload = function(){
    var scrollTop = window.scrollY;
    var imgs = Array.from(document.querySelectorAll('img'));

    lazyLoad();

    //函数防抖模式
    var timer = null;
    window.onscroll = () => {
        clearTimeout(timer);
        timer = setTimeout(()=>{
            scrollTop = window.scrollY;
            lazyLoad();
        },300)
    }

    function lazyLoad(){
        imgs.forEach((item,index)=>{
            if( item.offsetTop < window.innerHeight + scrollTop ){
                console.log(item.offsetTop)

                item.setAttribute('src',item.dataset.src)
```

```
            }
        })
    }
}
```

- 4.最终版 throttle + debounce
  完美懒加载
  注意点: 在滚动条下拉状态下刷新页面, 页面实现更新渲染之后会立马触发滚动条事件,回到上一次页面的停留点,但是并不是从scrollTop为0的位置出发~

```
window.onload = function(){

    var scrollTop = window.scrollY;
    var imgs = Array.from(document.querySelectorAll('img'));
    lazyLoad();
    // 采用了节流函数
    window.addEventListener('scroll',throttle(lazyLoad,500,1000));

    function throttle(fun, delay, time) {
        var timeout,
            startTime = new Date();
        return function() {

            var context = this,
                args = arguments,
                curTime = new Date();
            clearTimeout(timeout);
            // 如果达到了规定的触发时间间隔，触发 handler
            console.log(curTime - startTime)
            if (curTime - startTime >= time) {
                fun();
                startTime = curTime;
                // 没达到触发间隔，重新设定定时器
            } else {
                timeout = setTimeout(fun, delay);
            }
        };
    };
    // 实际想绑定在 scroll 事件上的 handler
    // 需要访问到imgs ,  scroll
    function lazyLoad(){
        scrollTop = window.scrollY;
        imgs.forEach((item,index)=>{
            if( scrollTop===0 && item.dataset.src !== '' && item.offsetTop < window.innerHeight + scrollTop ){
                // alert()
                item.setAttribute('src',item.dataset.src)
                item.setAttribute('data-src','')
            }else if( item.dataset.src !== '' && item.offsetTop < window.innerHeight + scrollTop && item.offsetTop > scrollTop ){
                item.setAttribute('src',item.dataset.src)

                item.setAttribute('data-src','')
```

```
            }
        })
    }

}
```

==typeof和instanceof==

ECMAScript是松散类型的，一次需要一种手段来检测给定变量的数据类型，typeof操作符（注意不是函数哈！）就是负责提供这方面信息的，

typeof 可以用于检测基本数据类型和引用数据类型。

语法格式如下：

```
typeof variable
```

返回6种String类型的结果：

- "undefined" - 如果这个值未定义
- "boolean" - 如果这个值是布尔值
- "string" - 如果这个值是字符串
- "number" - 如果这个值是数值
- "object" - 如果这个值是对象或null
- "function" - 如果这个值是函数
  示例：

```
console.log(typeof 'hello'); // "string"
console.log(typeof null); // "object"
console.log(typeof (new Object())); // "object"
console.log(typeof(function(){})); // "function"
```

typeof主要用于检测基本数据类型：数值、字符串、布尔值、undefined， 因为typeof用于检测引用类型值时，==对于任何Object对象实例（包括null），typeof都返回"object"值，没办法区分是那种对象，对实际编码用处不大。==

instanceof 用于判断一个变量是否某个对象的实例

在检测基本数据类型时typeof是非常得力的助手，但在检测引用类型的值时，这个操作符的用处不大，通常，我们并不是想知道某个值是对象，而是想知道它是什么类型的对象。此时我们可以使用ECMAScript提供的instanceof操作符。

语法格式如下：

```
result = variable instanceof constructor
```

返回布尔类型值：

- true - 如果变量（variable）是给定引用类型的实例，那么instanceof操作符会返回true
- false - 如果变量（variable）不是给定引用类型的实例，那么instanceof操作符会返回false

示例：

```
    function Person(){}
    function Animal(){}
    var person1 = new Person();
    var animal1 = new Animal();
    console.log(person1 instanceof Person); // true
    console.log(animal1 instanceof Person); // false
    console.log(animal1 instanceof Object); // true
    console.log(1 instanceof Person);   //false


    var oStr =  new String("hello world");
    console.log(typeof(oStr));      // object
    console.log(oStr instanceof String);
    console.log(oStr instanceof Object);

    // 判断 foo 是否是 Foo 类的实例

    function Foo(){}
    var foo = new Foo();
    console.log(foo instanceof Foo);

    // instanceof 在继承中关系中的用法
    console.log('instanceof 在继承中关系中的用法');


    function Aoo(){}
    function Foo(){}

    Foo.prototype = new Aoo();
    var fo = new Foo();

    console.log(fo instanceof Foo);
    console.log(fo instanceof Aoo)
```

根据规定，所有引用类型的值都是Object的实例。因此，在检测一个引用类型值和Object构造函数时，instanceof操作符会始终返回true。如果使用instanceof 操作符检测基本类型值时，该操作符会始终返回false，因为基本类型不是对象。

```
console.log(Object.prototype.toString.call(null));
// [object Null]
undefined
console.log(Object.prototype.toString.call([1,2,3]));
//[object Array]
undefined
console.log(Object.prototype.toString.call({}));
// [object Object]
```

==原型和原型链==

参考js高级教程

==数组的使用方法==

push、pop、shift、unshift、splice、join、reverse、sort、slice、map every some fliter forEach、reduce.....作为最常用的类型，JavaScript中的数组还是和其他语言中有很大的区别的。 主要体现在两点：

- 数组中的每一项都可以保存任何类型的数据
- 数组的大小可以动态调整

首先来介绍创建数组的两种方法

1. 第一种方式

```
var arr1 = new Array();

var arr2 = new Array(3);

var arr3 = new Array('jerry');
```

可以看到这种方式建立数组，arr1是一个空数组，arr2是一个长度为3的数组，arr3是一个包含'**jerry**'一个元素的数组。同时通过这种方式创建的数组，new操作符可以省略。

2. 第二种方式称为数组字面量表示法。

```
var a = [];
var arr = ['tom','jack']
```

数组的长度是可动态调整，导致我们直接就可以设置它的长度

```
var a = [123,423];
a.length = 10;
a[9]='123';
console.log(a[8])//undefined

a[10] = '123'
console.log(a.length)//10
```

从上面的代码中我们可以看出：

- 如果我们设置的长度大于原来的数组的长度的时候， 数组后面的元素自动设置为undefined。
- 如果我们对大于当前数组长度的位置赋值的时候，那么就会导致数组的长度自动变为你所赋值位置+1.

2016103149463arr_function.png

改变数组的方法

栈方法

pop和push很简单，也很容易理解。pop就是从数组的末尾删除一个元素并返回。push是在数组的末尾添加一个元素。

```
var arr = [1,3,4];
arr.pop();
console.log(arr);//[1,3]

arr.push(5);
console.log(arr);//[1,3,5]
```

队列方法

shift和unshift是和栈方法是相对的，它俩是从数组的头部进行操作。shift是从头部删除一个元素，unshift是从同步加入一个元素。

```
var arr = [1,3,4];
arr.shift();
console.log(arr);//[3,4]

arr.unshift(5);
console.log(arr);//[5,3,4]
```

重排序方法

reverse是对数组进行翻转。

```
var arr = [1,3,4];
arr.reverse();
console.log(arr);//[4,3,1]
```

sort是对数组进行排序。

```
var arr = [1,3,5,4];
arr.sort();
console.log(arr);//[1,3,4,5];
```

sort默认的对数组进行升序排序。sort可以接收一个自定义的比较函数，自定义排序规则。

sort方法会调用每个元素的toString()方法，从而通过字符串进行比较大小。即使是数值，依然要变换成字符串，从而就会带来一些问题。比如

```
var arr = [1,3,15,4];
arr.sort()
console.log(arr);//[1,15,3,4];
```

转换为字符串之后，‘15’是排在‘3’，‘4’的前面的。这就带来了问题，所以在进行数值数组的排序，必须进行自定义排序规则。

```
var arr = [1,3,15,4];
function compare(v1,v2){
    if(v1 > v2)
        return 1;
    if(v1 < v2)
        return -1;
    return 0;
}
arr.sort(compare)
console.log(arr);//[1,3,4,15]
```

splice方法

splice方法可以说是数组中功能最强大的方法，集多项功能于一身。主要的用途就是用来向数组的中部插入元素。

splice方法主要有三种用法。

splice的返回值为删除的元素组成的数组。如果删除的元素为空，返回空数组。

- 删除元素

splice（index,count）,index表示删除的位置，count表示删除的项数。

```
var arr = [1,3,4];
console.log(arr.splice(2,1));//[4]
//删除元素
console.log(arr);[1,3];
```

- 插入元素

splice(index,0,element,....) index 表示要插入的位置，0代表删除0个元素，element要插入的元素,如果要插入多个元素，可以继续添加。

```
var arr = [1,3,4];
console.log(arr.splice(2,0,'tom'));//[ ]

console.log(arr);//[1,3,'tom',4]
```

如果index的值大于数组本身的长度，那么就在最后位置添加。且数组的长度只会加1.

```
var arr = [1,3,4];
console.log(arr.splice(5,0,'tom'));//[ ]

console.log(arr);//[1,3,4,'tom']
console.log(arr.length);//4
```

如果index的值为负数,那么就从（arr.length+index）位置开始插入，如果（arr.length+index）的值小于0，那么就从数组的开始位置进行插入。

```
var arr = [1,3,4,4,7,6];
```

```
console.log(arr.splice(-1,0,'tom'));//[ ]

console.log(arr);//[1,3,4,4,7,'tom',6]
console.log(arr.length);//7

console.log(arr.splice(-7,0,'tom'));//[ ]

console.log(arr);//['tom',1,3,4,4,7,'tom',6]
console.log(arr.length);//8

console.log(arr.splice(-10,0,'jack'));//[ ]

console.log(arr);//['jack','tom',1,3,4,4,7,'tom',6]
console.log(arr.length);//9
```

- 替换元素

splice（index,count,element,....）.index代表替换开始的位置，count > 0,element表示要替换成的元素。其实替换过程包含两个过程:1.删除. 2插入.也就是上面的两个过程的融合。

```
var arr = [1,3,4];
console.log(arr.splice(1,1,'tom'));//[3]

console.log(arr);//[1,'tom',4]
```

如果index大于数组的长度，或者小于0，处理的结果同上面插入元素处理的方式一样。

不改变数组的方法

转换方法

join方法主要是用来将数组的元素通过规定的方式连接成字符串。

```
var arr = [1,3,4,5];
console.log(arr.join(','))//1,3,4,5
console.log(arr.join('+'))//1+3+4+5
console.log(arr.join('?'))//1?3?4?5
console.log(arr)//[1,3,4,5]
```

操作方法

slice和concat方法。 slice方法主要用来返回指定位置的数组的子数组。slice(start,end)。end省略，返回的是从开始位置到数组的末尾。end不省略，返回的是从start到end之间的子数组，包括start位置但不包括end位置的数组。

```
var arr = [1,3,4,5];

console.log(arr.slice(1));//[3,4,5]

console.log(arr.slice(1,2));//[3]
```

如果slice方法的参数中有一个负数，则用数组长度加上该数来确定相应的位置。例如在一个长度为5的数组上调用slice(-2,-1)与调用slice(3,4)得到的结果相同。如果结束位置小于起始位置，则返回空数组。

concat 方法，主要是连接多个数组。

```
var arr = [1,3,4,5];
var testArr = [1,23,4];
console.log(arr.concat(testArr));//[1,3,4,5,1,23,4]
console.log(arr.concat('tom'));//[1,3,4,5,'tom']
```

迭代方法

ES5新增加的迭代方法主要包括如下几种

> map every some fliter forEach

这几个方法有一下共同点，都接收两个参数，一个是要在数组上每一项运行的函数，一个是运行该函数作用域的对象，改变this的指向（可选）。其中函数需要传入三个参数，一个是每个元素的值，每个元素的index，数组本身。

```
function(value,index,array)
{
}
```

下面一个一个的来介绍

- map

map返回数组中每一个数组元素经过传入的函数处理后组成的新数组

```
var arr = [1,3,4];
var newArr = arr.map(function(value,index,array){
    return value*2;
})
console.log(newArr);//[2,6,8]
console.log(arr);//[1,3,4]
```

- some和every

some和every比较相像。some是对每一个数组中的元素运行传入的函数，如果有一个返回true，那么就返回true；every是对每一个数组中的元素运行传入的函数，如果所有的都返回true，那么就返回true。

```
var arr = [1,3,4];
var result1 = arr.some(function(value,index,array){
    return value > 2;
})

var result2 = arr.every(function(value,index,array){
    return value > 2;
})
console.log(result1);// true
console.log(result2);// false
```

- filter

从名字可以看出，这是一个过滤的方法，返回的一个数组，这个数组是满足传入的参数的函数的元素所组成的。

```
var arr = [1,3,4];
var result = arr.filter(function(value,index,array){
    return value > 2;
})
console.log(result);// [3,4]
```

- forEach

forEach主要用来遍历，遍历数组中每一个元素，对其进行操作。该方法没有返回值。

```
var arr = [1,3,4];
arr.forEach(function(value,index,array){
    console.log('arr['+index+']='+value);
})
// 结果
arr[0]=1
arr[1]=3
arr[2]=4
```

缩小方法

reduce和reduceRight.这两个方法接收两个参数，一个是每项都运行的函数，一个是缩小基础的初始值（可选）。reduce和reduceRight返回的是一个值。其中每项都运行的函数包含四个参数，

```
funciton(prev,cur,index,array){
}
```

下面通过一个例子就可以说明这个函数是干嘛的。

```
var arr = [1,3,4];
var result = arr.reduce(function(prev,cur,index,array){
    return prev+cur;
},10);
console.log(result)//18
var result1 = arr.reduce(function(prev,cur,index,array){
    return prev+cur;
});
console.log(result1)//8
```

```
array.reduce(function(total, currentValue, currentIndex, arr), initialValue)
```

参数	描述
function(total,currentValue, index,arr)	必需。用于执行每个数组元素的函数。
total	必需。初始值，或者计算结束后的返回值。
currentValue	必需。当前元素
currentIndex	可选。当前元素的索引
arr	可选。当前元素所属的数组对象。
initialValue	可选。传递给函数的初始值

```
var arr = [11,22,33,44,55,66];
    var arr1 = arr.reduce(function(total, currentValue, currentIndex, arr){
        return total+"-"+currentValue;
    },"00");
    console.log(arr1);//00-11-22-33-44-55-66
```

```
var arr = [11,22,33,44,55,66];
    var arr1 = arr.reduceRight(function(total, currentValue, currentIndex, arr){
        return total+"-"+currentValue;
    },"00");
    console.log(arr1);//00-66-55-44-33-22-11
```

reduceRight和reduce一样，无非他开始的位置是从数组的后面。

其他方法

- indexOf()
- lastIndexOf()

这两个主要是用来判断元素在数组中的位置,未找到返回-1，接收两个参数，indexOf(searchElement[, fromIndex]),lastIndexOf(searchElement[, fromIndex])。fromIndex可选。其中formIndex也可以指定字符串。

```
var arr = [1,3,4,4,1,5,1];
```

```
var value = arr.indexOf(1)
console.log(value)//0
value = arr.indexOf(1,4)
console.log(value)//4
value = arr.indexOf(1,5)
console.log(value)//6

value = arr.lastIndexOf(1)
console.log(value)//6



value = arr.lastIndexOf(1,3)
console.log(value)//0
```

- toString()
- toLocalString()
- valueOf()

这三个方法是所有对象都具有的方法。

toString()返回的是一个字符串，toLocaleString同它类似。valueOf()返回的是一个数组

```
var arr= [1,3,4]
console.log(arr.toString());//1,3,4
console.log(arr.valueOf());//[1,3,4]
console.log(arr.toLocaleString());//1,3,4
```

可以复写toString(),toLocaleString()返回不同的结果。

==callee和caller==

1 ：caller 返回一个调用当前函数的引用 如果是由顶层调用的话 则返回null

（ 举个栗子哈 ==caller给你打电话的人 == 谁给你打电话了 谁调用了你 很显然是下面a函数的执行 只有在打电话的时候你才能知道打电话的人是谁 所以对于函数来说 只有caller在函数执行的时候才存在 ）

```
var callerTest = function() {

          console.log(callerTest.caller) ;

    } ;

     function a() {

          callerTest() ;

    }

    a() ;//输出function a() {callerTest();}

```

```
15      callerTest() ;//输出null
```

2 ：callee 返回一个正在被执行函数的引用 （这里常用来递归匿名函数本身 但是在严格模式下不可行）

callee是arguments对象的一个成员 表示对函数对象本身的引用 它有==个length属性（代表形参的长度）==

```
1
2       var c = function(x,y) {
3
4               console.log(arguments.length,arguments.callee.length,arguments.callee)
5
6        } ;
7
8
9
10      c(1,2,3) ;//输出3 2 function(x,y)
{console.log(arguments.length,arguments.callee.length,arguments.callee)}
```

==new的执行原理==

1、创建一个新对象；[var o = new Object();]

2、将构造函数的作用域赋给新对象（因此this指向了这个新对象）；

3、执行构造函数中的代码(为这个新对象添加属性)；

4、返回新对象。

==get和post的区别==

get请求传参长度的误区

```
1   误区：我们经常说get请求参数的大小存在限制，而post请求的参数大小是无限制的。
```

实际上HTTP 协议从未规定 GET/POST 的请求长度限制是多少。对get请求参数的限制是来源与浏览器或web服务器，浏览器或web服务器限制了url的长度。为了明确这个概念，我们必须再次强调下面几点:

HTTP 协议 未规定 GET 和POST的长度限制 GET的最大长度显示是因为 浏览器和 web服务器限制了 URI的长度 不同的浏览器和WEB服务器，限制的最大长度不一样 要支持IE，则最大长度为2083byte，若只支持Chrome，则最大长度 8182byte

补充get和post请求在缓存方面的区别

补充补充一个get和post在缓存方面的区别：

- get请求类似于查找的过程，用户获取数据，可以不用每次都与数据库连接，所以可以使用缓存。

- post不同，post做的一般是修改和删除的工作，所以必须与数据库交互，所以不能使用缓存。因此get请求适合于请求缓存。

HTTP 协议中GET和POST到底有哪些区别

HTTP 定义了与服务器交互的不同方法，最常用的有4种，Get、Post、Put、Delete,如果我换一下顺序就好记了，Put（增）,Delete（删），Post（改）,Get（查），即增删改查，下面简单叙述一下：

- 1）Get， 它用于获取信息，注意，他只是获取、查询数据，也就是说它不会修改服务器上的数据，从这点来讲，它是数据安全的，而稍后会提到的Post它是可以修改数据的，所以这也是两者差别之一了。
- 2） Post，它是可以向服务器发送修改请求，从而修改服务器的，比方说，我们要在论坛上回贴、在博客上评论，这就要用到Post了，当然它也是可以仅仅获取数据的。
- 3）Delete 删除数据。可以通过Get/Post来实现。用的不多，暂不多写，以后扩充。
- 4）Put，增加、放置数据，可以通过Get/Post来实现。用的不多，暂不多写，以后扩充。

下面简述一下Get和Post区别：

1） GET请求的数据是放在HTTP包头中的，也就是URL之后，通常是像下面这样定义格式的，（而Post是把提交的数据放在HTTP正文中的）。

```
login.action?name=hyddd&password=idontknow&verify=%E4%BD%E5%A5%BD
```

- a，以 ？ 来分隔URL和数据；
- b，以& 来分隔参数；
- c，如果数据是英文或数字，原样发送；
- d，如果数据是中文或其它字符，则进行BASE64编码。

2）GET提交的数据比较少，最多1024B，因为GET数据是附在URL之后的，而URL则会受到不同环境的限制的，比如说IE对其限制为2K+35，而POST可以传送更多的数据（理论上是没有限制的，但一般也会受不同的环境，如浏览器、操作系统、服务器处理能力等限制，IIS4可支持80KB，IIS5可支持100KB）。

3）Post的安全性要比Get高，因为Get时，参数数据是明文传输的，而且使用GET的话，还可能造成Cross-site request forgery攻击。而POST数据则可以加密的，但GET的速度可能会快些。

所以综上几点，总结成下表：

操作方式	数据位置	明文密文	数据安全	长度限制	应用场景
GET	http包头	明文	不安全	长度较小	查询数据
POST	http正文	可明可密	安全	支持较大的数据传输	修改数据

==点击li能够实现弹出当前li索引==

经典的js问题 实现点击li能够弹出当前li索引与innerHTML的函数

按照我们平常的想法，代码应该是这样写的：

```
var myul = document.getElementsByTagName("ul")[0];
    var list = myul.getElementsByTagName("li");

    function foo(){
        for(var i = 0, len = list.length; i < len; i++){
            list[i].onclick = function(){
                alert(i + "----" + this.innerHTML);
            }
        }
    }
    foo();
```

但是不巧的是产生的结果是这样的：

索引index为什么总是4呢，这是js中没有块级作用域导致的。这里有三种解决思路

1. 使用闭包

```
<script type="text/javascript">

    var myul = document.getElementsByTagName("ul")[0];
    var list = myul.getElementsByTagName("li");

    function foo(){
        for(var i = 0, len = list.length; i < len; i++){
            var that = list[i];
            list[i].onclick = (function(k){
                var info = that.innerHTML;
                return function(){
                    alert(k + "----" + info);
                };
            })(i);
        }
    }
    foo();
    </script>
```

2.使用ES6中的新特性let来声明变量

用let来声明的变量将具有块级作用域，很明显可以达到要求，不过需要注意的是得加个'use strict'（使用严格模式）才会生效

```
<script type="text/javascript">

var myul = document.getElementsByTagName("ul")[0];

var list = myul.getElementsByTagName("li");
```

```
function foo(){'use strict'
    for(let i = 0, len = list.length; i < len; i++){
        list[i].onclick = function(){
            alert(i + "----" + this.innerHTML);
        }
    }
}
foo();

</script>
```

3.事件委托

```
<script type="text/javascript">

var myul = document.querySelector('ul');
var list = document.querySelectorAll('ul li');

myul.addEventListener('click', function(ev){
    var ev = ev || window.event;
    var target = ev.target || ev.srcElemnt;

    for(var i = 0, len = list.length; i < len; i++){
        if(list[i] == target){
            alert(i + "----" + target.innerHTML);
        }
    }
});

</script>
```

4.引入jquery,使用其中的on或delegate进行事件绑定（它们都有事件代理的特性）

```
<script type="text/javascript">

$("ul").delegate("li", "click", function(){
    var index = $(this).index();
```

var info = $(this).html(); alert(index + "----" + info); });

```
</script>

<script type="text/javascript">

$("ul").on("click", "li", function(){
    var index = $(this).index();
    var info = $(this).html();

        alert(index + "----" + info);

    });
</script>
```

==js添加事件-兼容各种环境==

JS中添加事件 兼容各种环境

```
var EventUtil = {
    //添加
    addHandler : function (element , type, handler {
        if ( element.addEventListener){
            element.addEventListener(type, handler, false);
        }else if ( element.attachEvent) {
            element.attachEvent("on"+type,handler);
        }else {
            element["on" + type] = handler;
        }
    },
    //移除
    removeHandler : function (element , type , handler){
        if(element.removeEventListener){
            element.removeEventListener(type , handler , false);
        }else if(element.detachEvent){
            element.detachEvent("on" + type , handler);
        }else{
            element["on" + type] = handler;
        }
    }
}
```

==this指向的问题==

首先必须要说的是，this的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定this到底指向谁，实际上this的最终指向的是==那个调用它的对象==（这句话有些问题，后面会解释为什么会有问题，虽然网上大部分的文章都是这样说的，虽然在很多情况下那样去理解不会出什么问题，但是实际上那样理解是不准确的，所以在你理解this的时候会有种琢磨不透的感觉），那么接下来我会深入的探讨这个问题。

例子1：

```
function a(){
    var user = "追梦子";
    console.log(this.user); //undefined
    console.log(this); //Window
}
a();
```

按照我们上面说的this最终指向的是调用它的对象，这里的函数a实际是被Window对象所点出来的，下面的代码就可以证明。

```
function a(){
    var user = "追梦子";
    console.log(this.user); //undefined
    console.log(this);    //Window
}
window.a();
```

和上面代码一样吧，其实alert也是window的一个属性，也是window点出来的。

例子2：

```
var o = {
    user:"追梦子",
    fn:function(){
        console.log(this.user);  //追梦子
    }
}
o.fn();
```

这里的this指向的是对象o，因为你调用这个fn是通过o.fn()执行的，那自然指向就是对象o，这里再次强调一点，this的指向在函数创建的时候是决定不了的，在调用的时候才能决定，谁调用的就指向谁，一定要搞清楚这个。

其实例子1和例子2说的并不够准确，下面这个例子就可以推翻上面的理论。

如果要彻底的搞懂this必须看接下来的几个例子

例子3：

```
var o = {
    user:"追梦子",
    fn:function(){
        console.log(this.user); //追梦子
    }
}
window.o.fn();
```

这段代码和上面的那段代码几乎是一样的，但是这里的this为什么不是指向window，如果按照上面的理论，最终this指向的是调用它的对象，这里先说个而外话，window是js中的全局对象，我们创建的变量实际上是给window添加属性，所以这里可以用window点o对象。

这里先不解释为什么上面的那段代码this为什么没有指向window，我们再来看一段代码。

```
var o = {
    a:10,
    b:{
        a:12,
        fn:function(){
            console.log(this.a); //12
        }
    }
}
o.b.fn();
```

这里同样也是对象o点出来的，但是同样this并没有执行它，那你肯定会说我一开始说的那些不就都是错误的吗？其实也不是，只是一开始说的不准确，接下来我将补充一句话，我相信你就可以彻底的理解this的指向的问题。

- 情况1：如果一个函数中有this，但是它没有被上一级的对象所调用，那么this指向的就是window，这里需要说明的是在js的严格版中this指向的不是window，但是我们这里不探讨严格版的问题，你想了解可以自行上网查找。
- 情况2：如果一个函数中有this，这个函数有被上一级的对象所调用，那么this指向的就是上一级的对象。
- 情况3：如果一个函数中有this，这个函数中包含多个对象，尽管这个函数是被最外层的对象所调用，this指向的也只是它上一级的对象，例子3可以证明，如果不相信，那么接下来我们继续看几个例子。

```
var o = {
    a:10,
    b:{
        // a:12,
        fn:function(){
            console.log(this.a); //undefined
        }
    }
}
o.b.fn();
```

尽管对象b中没有属性a，这个this指向的也是对象b，因为this只会指向它的上一级对象，不管这个对象中有没有this要的东西。

还有一种比较特殊的情况，例子4：

```
var o = {
    a:10,
    b:{
        a:12,
        fn:function(){
            console.log(this.a); //undefined
            console.log(this); //window
        }
    }
}
var j = o.b.fn;
j();
```

这里this指向的是window，是不是有些蒙了？其实是因为你没有理解一句话，这句话同样至关重要。

this永远指向的是最后调用它的对象，也就是看它执行的时候是谁调用的，例子4中虽然函数fn是被对象b所引用，但是在将fn赋值给变量j的时候并没有执行所以最终指向的是window，这和例子3是不一样的，例子3是直接执行了fn。

this讲来讲去其实就是那么一回事，只不过在不同的情况下指向的会有些不同，上面的总结每个地方都有些小错误，也不能说是错误，而是在不同环境下情况就会有不同，所以我也没有办法一次解释清楚，只能你慢慢地的去体会。

构造函数版this：

```
function Fn(){
    this.user = "追梦子";
}
var a = new Fn();
console.log(a.user); //追梦子
```

这里之所以对象a可以点出函数Fn里面的user是因为new关键字可以改变this的指向，将这个this指向对象a，为什么我说a是对象，因为用了new关键字就是创建一个对象实例，理解这句话可以想想我们的例子3，我们这里用变量a创建了一个Fn的实例（相当于复制了一份Fn到对象a里面），此时仅仅只是创建，并没有执行，而调用这个函数Fn的是对象a，那么this指向的自然是对象a，那么为什么对象a中会有user，因为你已经复制了一份Fn函数到对象a中，用了new关键字就等同于复制了一份。

除了上面的这些以外，我们还可以自行改变this的指向，关于自行改变this的指向请看JavaScript中call,apply,bind方法的总结这篇文章，详细的说明了我们如何手动更改this的指向。

更新一个小问题当this碰到return时

```
function fn()
{
    this.user = '追梦子';
    return {};
}
var a = new fn;

console.log(a.user); //undefined
```

再看一个

```
function fn()
{
    this.user = '追梦子';
    return function(){};
}
var a = new fn;
console.log(a.user); //undefined
```

再来

```
function fn()
{
    this.user = '追梦子';
    return 1;
}
var a = new fn;
console.log(a.user); //追梦子
```

```
function fn()
{
    this.user = '追梦子';
    return undefined;
}
var a = new fn;
console.log(a.user); //追梦子
```

什么意思呢？

==如果返回值是一个对象，那么this指向的就是那个返回的对象，如果返回值不是一个对象那么this还是指向函数的实例。==

```
function fn()
{
    this.user = '追梦子';
    return undefined;
}
var a = new fn;
console.log(a); //fn {user: "追梦子"}
```

还有一点就是虽然null也是对象，但是在这里this还是指向那个函数的实例，因为null比较特殊。

```
function fn()
{
    this.user = '追梦子';
    return null;
}
var a = new fn;
console.log(a.user); //追梦子
```

知识点补充：

1.在严格版中的默认的this不再是window，而是undefined。

2.new操作符会改变函数this的指向问题，虽然我们上面讲解过了，但是并没有深入的讨论这个问题，网上也很少说，所以在这里有必要说一下。

```
function fn(){
    this.num = 1;
}
var a = new fn();
console.log(a.num); //1
```

为什么this会指向a？首先new关键字会创建一个空的对象，然后会自动调用一个函数apply方法，将this指向这个空对象，这样的话函数内部的this就会被这个空的对象替代。

2017-09-15 11:49:14

注意: 当你new一个空对象的时候,js内部的实现并不一定是用的apply方法来改变this指向的,这里我只是打个比方而已.

if (this === 动态的\可改变的) return true;

==tcp三次握手==

第一次

第一次握手：建立连接时，客户端发送syn包（syn=j）到服务器，并进入SYN_SENT状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

第二次

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态；

第三次

第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1），此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。

==数组去重==

* 数组去重【Array原型扩展一个方法实现数组去重】，利用hash对象

1. 遍历，数组下标去重法

```
时间复杂度：O(n^2),indexOf本身也消耗了O(n)的复杂度，
空间复杂度：O(n)
IE8以下不支持indexOf
```

```
Array.prototype.removeRepeat1 = function() {
    var res =[this[0]];
    for(var i=1; i<this.length;i++){ //从第二项开始遍历
        if(this.indexOf(this[i])==i){
            res.push(this[i]);
        }
    }
    return res;
};
```

2. 遍历，比较备用数组去重法

```
Array.prototype.removeRepeat2 = function() {
    var res =[];
    for(var i=0; i<this.length;i++){
        if(res.indexOf(this[i])==-1){
            res.push(this[i]);
         }
    }
    return res;
};
```

3. 遍历，hash去重法

```
类似于，利用对象的属性不能相同的特点进行去重
时间复杂度：O(n)，空间复杂度：O(n)
```

```
Array.prototype.removeRepeat3 = function() {
    var h= {};  //哈希表
    var res = [];
    for(var i=0; i<this.length;i++){
        if(!h[this[i]]){    //如果hash表中没有当前项
            h[this[i]]=true; //存入hash表
            res.push(this[i]);
        }
    }
    return res;
};
```

4. 遍历，Set去重法（ES6的 Set）

```
时间复杂度：O(n)，
空间复杂度：O(n)
Set兼容性不好，IE11以下不支持
```

```
Array.prototype.removeRepeat4 = function(){
    var result = new Set();
    for(var i=0; i<this.length; i++){
        result.add(this[i]);
    }
    return result;
}


//Set的方法二：
Array.from(array)把Set转化为数组
    Array.prototype.removeRepeat41 = function(){
        return Array.from(new Set(this));;
}
```

5. 排序后相邻去重法

```
Array.prototype.removeRepeat5 = function() {
    this.sort();
    var res=[this[0]];
    for(var i = 1; i< this.length; i++){
        if(this[i]!=this[i-1]){
            res.push(this[i]);
        }
    }
    return res;
}
```

==apply、call、bind区别==

call apply bind

第一个传的参数都是对象，不能传入构造函数，构造函数的typeof是function

传`null`或`undefined`时，将是JS执行环境的全局变量。浏览器中是window，其它环境（如node）则是global

call方法

语法：call(thisObj，Object)

定义：调用一个对象的一个方法，以另一个对象替换当前对象。

说明：call 方法可以用来代替另一个对象调用一个方法。call 方法可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。 如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。

apply方法

语法：apply(thisObj，[argArray])

定义：应用某一对象的一个方法，用另一个对象替换当前对象。

说明：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。

- 相同点

> 调用函数时，改变当前传入的对象为函数中this指针的引用

> 当第一个参数thisObj传入null/undefined的时候将执行js全局对象浏览器中是window，其他环境是global。

- 不同点：

> call, apply方法区别是,从第二个参数起, call方法参数将依次传递给借用的方法作参数, 而apply直接将这些参数放到一个数组中再传递, 最后借用方法的参数列表是一样的.

bind相同点和call apply相同

而bind是返回一个新函数，这个函数的上下文，为传入的对象。需要再次调用才能时候用.

apply、call、bind比较

那么 apply、call、bind 三者相比较，之间又有什么异同呢？何时使用 apply、call，何时使用 bind 呢。简单的一个栗子：

```
ar obj = {

x: 81,

};


var foo = {

getX: function() {

return this.x;

}

}


console.log(foo.getX.bind(obj)()); //81

console.log(foo.getX.call(obj)); //81

console.log(foo.getX.apply(obj)); //81
```

三个输出的都是81，但是注意看使用 bind() 方法的，他后面多了对括号。

也就是说，区别是，当你希望改变上下文环境之后并非立即执行，而是回调执行的时候，使用 bind() 方法。而 apply/call 则会立即执行函数。

==再总结一下==：

- apply 、 call 、bind 三者都是用来改变函数的this对象的指向的；
- apply 、 call 、bind 三者第一个参数都是this要指向的对象，也就是想指定的上下文；
- apply 、 call 、bind 三者都可以利用后续参数传参；
- bind是返回对应函数，便于稍后调用；apply、call则是立即调用 。

==深拷贝和浅拷贝==

JavaScript有两种数据类型，基础数据类型和引用数据类型。基础数据类型都是按值访问的，我们可以直接操作保存在变量中的实际的值。而引用类型如Array，我们不能直接操作对象的堆内存空间。引用类型的值都是按引用访问的，即保存在变量对象中的一个地址，该地址与堆内存的实际值相关联。

一、深拷贝和浅拷贝的区别

- 浅拷贝（shallow copy）：只复制指向某个对象的指针，而不复制对象本身，新旧对象共享一块内存；
- 深拷贝（deep copy）：复制并创建一个一摸一样的对象，不共享内存，修改新对象，旧对象保持不变。

```
var a = 25;
var b = a;
b = 10;
console.log(a);//25
console.log(b);//10
```

//浅拷贝

```
var obj1 = { a: 10, b: 20, c: 30 };
var obj2 = obj1;
obj2.b = 40;
console.log(obj1);// { a: 10, b: 40, c: 30 }
console.log(obj2);// { a: 10, b: 40, c: 30 }
```

//深拷贝

```
var obj1 = { a: 10, b: 20, c: 30 };
var obj2 = { a: obj1.a, b: obj1.b, c: obj1.c };
obj2.b = 40;
console.log(obj1);// { a: 10, b: 20, c: 30 }
console.log(obj2);// { a: 10, b: 40, c: 30 }
```

二、浅拷贝的实现

```
var json1 = {"a":"name","arr1":[1,2,3]}
function copy(obj1) {
        var obj2 = {};
        for (var i in obj1) {
            obj2[i] = obj1[i];
        }
        return obj2;
}
var json2 = copy(json1);
json1.arr1.push(4);
alert(json1.arr1);  //1234
alert(json2.arr1)  //1234
```

三、深拷贝的实现

- 1、Object.assign()

```
let foo = {
    a: 1,
    b: 2,
    c: {
        d: 1,
    }
}
let bar = {};
Object.assign(bar, foo);
foo.a++;
foo.a === 2 //true
bar.a === 1 //true
foo.c.d++;
foo.c.d === 2 //true
bar.c.d === 1 //false
bar.c.d === 2 //true
```

Object.assign()是一种可以对==非嵌套对象==进行深拷贝的方法，如果对象中出现嵌套情况，那么其对被嵌套对象的行为就成了普通的浅拷贝。

- 2、转成JSON
  用JSON.stringify把对象转成字符串，再用JSON.parse把字符串转成新的对象。

```
var obj1 = { body: { a: 10 } };
var obj2 = JSON.parse(JSON.stringify(obj1));
obj2.body.a = 20;
console.log(obj1);   // { body: { a: 10 } }
console.log(obj2);   // { body: { a: 20 } }
console.log(obj1 === obj2);   // false
console.log(obj1.body === obj2.body);   // false
```

但这种方法的缺陷是会==破坏原型链==，并且无法拷贝属性值为function的属性

- 3、递归

采用递归的方法去复制拷贝对象

```
var json1={
    "name":"shauna",
    "age":18,
    "arr1":[1,2,3,4,5],
    "string":'got7',
    "arr2":[1,2,3,4,5],
    "arr3":[{"name1":"shauna"},{"job":"web"}]
};


var json2={};
function copy(obj1,obj2){
  var obj2=obj2||{};
  for(var name in obj1){
    if(typeof obj1[name] === "object"){
      obj2[name]= (obj1[name].constructor===Array)?[]:{};
      copy(obj1[name],obj2[name]);
    }else{
      obj2[name]=obj1[name];
    }
  }
  return obj2;
}
json2=copy(json1,json2)
json1.arr1.push(6);
alert(json1.arr1);  //123456
alert(json2.arr1);  //12345
```

==实现一个bind方法==

```
Function.prototype.mybind = function(context) {
    var self = this;
    var args = [];//保存bind函数调用时传递的参数
    for(var i = 1, len = arguments.length; i< len;i ++) {
        args.push(arguments[i]);
    }

    //bind()方法返回值是一个函数
    return function() {
        //哇，新创建的函数传进来的参数可以在这里拿到哎！！
        var bindArgs = Array.prototype.slice.call(arguments);
        self.apply(context, args.concat(bindArgs))
    }
}
```

==ajax==

代码

```
var xmlhttp;
if (window.XMLHttpRequest){
    //  IE7+, Firefox, Chrome, Opera, Safari 浏览器执行代码
    xmlhttp=new XMLHttpRequest();
}
else{
    // IE6, IE5 浏览器执行代码
    xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}

xmlhttp.open("GET","/try/ajax/ajax_info.txt",true);
xmlhttp.send();
xmlhttp.onreadystatechange=function()
{
    if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
        document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
    }
}
```

优缺点：

- 优点
  - 通过异步模式，提升用户体验
  - 优化了浏览器和服务器之间的传输，减少了不必要的数据往返，减少了带宽的使用
  - Ajax在客户端运行，承担了一部分本来由服务器承担的工作，减少了大量用户下的服务器负载
  - Ajax是通过异步通信实现的局部刷新
- 缺点
  - ajax不支持浏览器的back按钮
  - 安全问题，Ajax暴露了与服务器交互的细节
  - 对搜索引擎的支持比较弱
  - 破坏了程序的异常机制
  - 不容易调试

==跨域==

一 跨域的原因

很多朋友不知道为什么要跨域 其实跨域请求存在的原因：由于浏览器的同源策略，即属于不同域的页面之间不能相互访问各自的页面内容。

那么什么是==同源策略==呢？

简单说来就是同协议，同域名，同端口。

二 跨域的场景

1.域名不同 www.yangwei.com 和www.wuyu.com 即为不同的域名）

2.二级域名相同，子域名不同（ www.wuhan.yangwei.com www.shenzheng.yangwei.com 为子域不同）

3.端口不同，协议不同（ http://www.yangwei.com 和https://www.yangwei.com属于跨域www.yangwei.con:8888和www.yangwei.con:8080)

三 跨域的方式

1.前端的方式: possMessage，window.name,document.domain,image.src(得不到数据返回)，jsonP(script.src后台不配合得不到数据返回)，style.href（得不到数据返回)

一.==imge.src==,==script.src==,==style.href== 不受同源策略的影响可以加载其他域的资源，可以用这个特性，向服务器发送数据。最常用的就是使用image.src 向服务器发送前端的错误信息。image.src 和style.href 是无法获取服务器的数据返回的，script.src 服务器端配合可以得到数据返回。

二possMessage,window.name,document.domain 是两个窗口直接相互传递数据。

（1）possMessage 是HTML5中新增的，使用限制是 必须获得窗口的window 引用。IE8+支持，firefox，chrome,safair,opera支持

（2）window.name ，在一个页面中打开另一个页面时，window.name 是共享的，所以可以通过window.name来传递数据，window.name的限制大小是2M，这个所有浏览器都支持,且没有什么限制。

3）document.domain 将两个页面的document.domain 设置成相同，document.domain 只能设置成父级域名，既可以访问，使用限制：这顶级域名必须相同，document.domain + iframe跨域 此方案仅限主域相同，子域不同的跨域应用场景。

- 实现原理：两个页面都通过js强制设置document.domain为基础主域，就实现了同域。

1.）父窗口：(www.domain.com/a.html)

```
<iframe id="iframe" src="http://child.domain.com/b.html"></iframe>
<script>
    document.domain = 'domain.com';
    var user = 'admin';
</script>
```

2.）子窗口：(child.domain.com/b.html)

```
<script>
    document.domain = 'domain.com';
    // 获取父窗口中变量
    alert('get js data from parent ---> ' + window.parent.user);
</script>
```

2.纯后端方式: CORS，服务器代理

CORS 是w3c标准的方式，通过在web服务器端设置：响应头Access-Control-Alow-Origin 来指定哪些域可以访问本域的数据，ie8&9(XDomainRequest),10+,chrom4 ，firefox3.5,safair4，opera12支持这种方式。

服务器代理，同源策略只存在浏览器端，通过服务器转发请求可以达到跨域请求的目的，劣势：增加服务器的负担，且访问速度慢。
image

前端代码--script.html：

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Insert title here</title>
    <script>
        // var getvalue = function (data) {
        //     alert(JSON.stringify(data));
        // };
        // var url = "http://127.0.0.1:3000/cors?callback=getvalue";
        // var script = document.createElement('script');
        // script.setAttribute('src', url);
        // document.getElementsByTagName('head')[0].appendChild(script);

        var xhr = new XMLHttpRequest(); // IE8/9需用window.XDomainRequest兼容

        // 前端设置是否带cookie
        xhr.withCredentials = false;

        xhr.open('get', 'http://127.0.0.1:3000/cors.js?callback=getvalue', true);
        xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
        xhr.send();

        xhr.onreadystatechange = function() {
            if (xhr.readyState == 4 && xhr.status == 200) {
                alert(xhr.responseText);
            }
        };


    </script>
    <!--<script src="http://127.0.0.1:3000"></script>-->
</head>
<body>
ffffffffffffffffffffffffff
</body>
</html>
```

后端代码--demo2.js：

```javascript
var express = require('express');
var app = express();
var http = require('http');
var fs = require('fs');
var url = require('url');

app.get('/', function (req, res) {
    res.sendfile(__dirname+"/index.html");
})



```

```
app.get('/cors.js', function(req, res) {
    var pathname = url.parse(req.url).pathname;
    console.log("req for " + pathname + " received.");
    fs.readFile(pathname.substr(1), function (err, data) {
        if (err) {
            console.log(err);
            // HTTP 状态码: 404 : NOT FOUND
            // Content Type: text/plain
            res.writeHead(404, {'Content-Type': 'text/html'});
        }else{
            res.header("Access-Control-Allow-Origin", "*"); //设置请求来源不受限制
            res.header("Access-Control-Allow-Headers", "X-Requested-With");
            res.header("Access-Control-Allow-Methods", "PUT,POST,GET,DELETE,OPTIONS"); //请求方式
            res.header("X-Powered-By", ' 3.2.1')
            res.header("Content-Type", "application/json;charset=utf-8");
            var data = {
                name:  ' - server 3001 cors process',
                id: ' - server 3001 cors process'
            }
            console.log(data);
            // "getvalue(data)"
            res.send("getvalue({ name: '5'})");
        }
        //  发送响应数据
        res.end();
    });


     })


var server = app.listen(3000, function () {
    var host = server.address().address
    var port = server.address().port

    console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
// 控制台会输出以下信息
console.log('Server running at http://127.0.0.1:3000/');

```

3.前后端结合:JsonP

script.src 不受同源策略的限制，所以可以动态的创建script标签，将要请求数据的域写在src 中参数中附带回调的方法，服务器端返回回调函数的字符串，并带参数。

如 script.src="http://www.yangwei.com/?id=001&callback=getInfoCallback,服务器端返回 getInfoCallBack("name:yangwei;age:18") 这段代码会直接执行，在前面定义好getInfoCallBack函数，既可以获得数据并解析。 这种是最长见的方式。 image

前端代码--script.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Insert title here</title>
    <script>
        var getvalue = function (data) {
            alert(JSON.stringify(data));
        };
        var url = "http://127.0.0.1:3000/person.js?callback=getvalue";
        var script = document.createElement('script');
        script.setAttribute('src', url);
        document.getElementsByTagName('head')[0].appendChild(script);

    </script>
    <!--<script src="http://127.0.0.1:3000"></script>-->
</head>
<body>
fffffffffffffffffffffffff
</body>
</html>
```

后端代码--demo2.js

```
var express = require('express');
var app = express();
var http = require('http');
var fs = require('fs');
var url = require('url');

app.get('/', function (req, res) {
    res.sendfile(__dirname+"/index.html");
})

app.get('/person.js',function (req,res) {
    var pathname = url.parse(req.url).pathname;

    // 输出请求的文件名
    console.log("req for " + pathname + " received.");
    // fs.readFile(pathname.substr(1), function (err, data) {
    //     if (err) {
    //         console.log(err);
    //         // HTTP 状态码: 404 : NOT FOUND
    //         // Content Type: text/plain
    //         res.writeHead(404, {'Content-Type': 'text/html'});
    //     }else{
    //         // HTTP 状态码: 200 : OK
    //         // Content Type: text/plain
    //         res.writeHead(200, {'Content-Type': 'text/html'});
    //
    //         // 响应文件内容
    //         res.write(data.toString());
```

```
    //     }
    //     //  发送响应数据
    //     res.end();
    // });
    res.send("getvalue({ name: '5'})")


})

app.get('/message.js',function (req,res) {
    var pathname = url.parse(req.url).pathname;

    // 输出请求的文件名
    console.log("req for " + pathname + " received.");
    fs.readFile(pathname.substr(1), function (err, data) {
        if (err) {
            console.log(err);
            // HTTP 状态码: 404 : NOT FOUND
            // Content Type: text/plain
            res.writeHead(404, {'Content-Type': 'text/html'});
        }else{
            // HTTP 状态码: 200 : OK
            // Content Type: text/plain
            res.writeHead(200, {'Content-Type': 'text/html'});

            // 响应文件内容
            res.write(data.toString());
        }
        //  发送响应数据
        res.end();
    });



})

app.get('/cors.js', function(req, res) {
    var pathname = url.parse(req.url).pathname;
    console.log("req for " + pathname + " received.");
    fs.readFile(pathname.substr(1), function (err, data) {
        if (err) {
            console.log(err);
            // HTTP 状态码: 404 : NOT FOUND
            // Content Type: text/plain
            res.writeHead(404, {'Content-Type': 'text/html'});
        }else{
            res.header("Access-Control-Allow-Origin", "*"); //设置请求来源不受限制
            res.header("Access-Control-Allow-Headers", "X-Requested-With");
            res.header("Access-Control-Allow-Methods", "PUT,POST,GET,DELETE,OPTIONS"); //请求方式
            res.header("X-Powered-By", ' 3.2.1')
            res.header("Content-Type", "application/json;charset=utf-8");

            var data = {
```

```
                name:  ' - server 3001 cors process',
                id: ' - server 3001 cors process'
            }
            console.log(data);
            // "getvalue(data)"
            res.send("getvalue({ name: '5'})");
        }
        //  发送响应数据
        res.end();
    });


     })


var server = app.listen(3000, function () {
    var host = server.address().address
    var port = server.address().port

    console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
// 控制台会输出以下信息
console.log('Server running at http://127.0.0.1:3000/');

```

==继承的几种方法==

字面量创建

- 代码

```
person = {
    name:'FE',
    age:23
}
```

- 优缺点
  使用同一个接口创建很多对象，会产生大量重复的代码

工厂模式

- 代码

```
function creatPerson(name,age,job){
    var o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;

    o.sayName=function(){
```

```
        alert(this.name);
    }

    return o;
}


var person1 = creatPerson('FE',20,'teacher');
```

- 优缺点
  虽然解决了创建相似对象的问题，但是没有解决对象识别的问题（即怎样知道一个对象的类型）

构造函数模式

- 代码

```
function Person(name,age,job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName=function(){
        alert(this.name);
    }

}


var person1 = Person('FE',20,'teacher');
```

- 优缺点
  创建自定义函数意味着将来可以将它的实例标识为一种特定的数据类型。但是每个方法都要在实例上重新创建一遍。

原型模式

- 代码

```
function Person(){};
Person.prototype.name = "FE";
Person.prototype.age = 20;
Person.prototype.sayName = function(){
    alert(this.name);
};

var person1 = new Person();
person1.sayName();            //'FE'
```

- 优缺点
  可以让所有的实例共享它所包含的属性和方法。原型中的属性和方法是共享的，但是实例一般要有单独的属性和方法，一般很少单独使用原型模式。

混合模式

- 代码

```
function Person(name,age,job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.friends=['aa','ss','dd'];
}

Person.prototype.sayName = function(){
    alert(this.name);
}

var person1 = new Person('FE',20,'teacher');
```

- 优缺点
  构造函数模式定义实例的属性，原型模式定义公共的属性和方法

==创建字面量的几种方法==

原型链继承

- 定义
  利用原型让一个引用类型继承另外一个引用类型的属性和方法
- 代码

```
function SuperType(){
    this.property = 'true';
}

SuperType.prototype.getSuperValue = function(){
    return this.property;
}

function SubType(){
    this.subProperty = 'false';
}

SubType.prototype = new SuperType();
SubType.prototype.getSubValue = function(){
    return this.subProperty;
}

var instance = new SubType();
alert(instance.getSuperValue());//true
```

- 优点
  简单明了，容易实现，在父类新增原型属性和方法，子类都能访问到。

- 缺点
  包含引用类型值的函数，所有的实例都指向同一个引用地址，修改一个，其他都会改变。不能像超类型的构造函数传递参数

构造函数继承

- 定义
  在子类型构造函数的内部调用超类型的构造函数
- 代码

```
function SuperType(){
    this.colors = ['red','yellow'];
}

function SubType(){
    SuperType.call(this);
}

var instance1 = new SubType();
instance1.colors.push('black');

var instance2 = new SubType();
instance2.colors.push('white');

alert(instance1.colors);//'red','yellow','black'

alert(instance2.colors);//'red','yellow','white'

```

- 优点
  简单明了，直接继承了超类型构造函数的属性和方法
- 缺点
  方法都在构造函数中定义，因此函数复用就无从谈起了，而且超类型中的原型的属性和方法，对子类型也是不可见的，结果所有的类型只能使用构造函数模式。

组合继承

- 定义
  使用原型链实现多原型属性和方法的继承，使用构造函数实现实例的继承
- 代码

```
function SuperType(name){
    this.name = name;
    this.colors = ['red','black'];
}

SuperType.prototype.sayName = function()
{
   alert(this.name);

}
```

```
10
11
12  function SubType(name,age){
13      SuperType.call(this,name);
14      this.age = age;
15  }
16
17  SubType.protptype = new SuperType();
18  SubType.protptype.sayAge = function(){
19      alert(this.age);
20
21  }
22
```

- 优点
  解决了构造函数和原型继承中的两个问题
- 缺点
  无论什么时候，都会调用两次超类型的构造函数

还有其他实现继承的方法，这里就不再赘述了，可参考js高级教程

==promise==

ES6 Promise 用法讲解 Promise是一个构造函数，自己身上有==all、reject、resolve==这几个眼熟的方法，原型上有==then、catch==等同样很眼熟的方法。

那就new一个

```
1  var p = new Promise(function(resolve, reject){
2      //做一些异步操作
3      setTimeout(function(){
4          console.log('执行完成');
5          resolve('随便什么数据');
6      }, 2000);
7  });
```

Promise的构造函数接收一个参数，是函数，并且传入两个参数：resolve，reject，分别表示异步操作执行成功后的回调函数和异步操作执行失败后的回调函数。其实这里用“成功”和“失败”来描述并不准确，按照标准来讲，resolve是将Promise的状态置为fulfilled，reject是将Promise的状态置为rejected。不过在我们开始阶段可以先这么理解，后面再细究概念。

在上面的代码中，我们执行了一个异步操作，也就是setTimeout，2秒后，输出“执行完成”，并且调用resolve方法。

运行代码，会在2秒后输出“执行完成”。注意！我只是new了一个对象，并没有调用它，我们传进去的函数就已经执行了，这是需要注意的一个细节。所以我们用Promise的时候一般是包在一个函数中，在需要的时候去运行这个函数，如：

```
function runAsync(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            console.log('执行完成');
            resolve('随便什么数据');
        }, 2000);
    });
    return p;
}
runAsync()
```

这时候你应该有两个疑问：1.包装这么一个函数有毛线用？2.resolve('随便什么数据');这是干毛的？

我们继续来讲。在我们包装好的函数最后，会return出Promise对象，也就是说，执行这个函数我们得到了一个Promise对象。还记得Promise对象上有then、catch方法吧？这就是强大之处了，看下面的代码：

```
runAsync().then(function(data){
    console.log(data);
    //后面可以用传过来的数据做些其他操作
    //......
});
```

在runAsync()的返回上直接调用then方法，then接收一个参数，是函数，并且会拿到我们在runAsync中调用resolve时传的的参数。运行这段代码，会在2秒后输出“执行完成”，紧接着输出“随便什么数据”。

这时候你应该有所领悟了，原来then里面的函数就跟我们平时的回调函数一个意思，能够在runAsync这个异步任务执行完成之后被执行。这就是Promise的作用了，简单来讲，就是能把原来的回调写法分离出来，在异步操作执行完后，用链式调用的方式执行回调函数。

你可能会不屑一顾，那么牛逼轰轰的Promise就这点能耐？我把回调函数封装一下，给runAsync传进去不也一样吗，就像这样：

```
function runAsync(callback){
    setTimeout(function(){
        console.log('执行完成');
        callback('随便什么数据');
    }, 2000);
}

runAsync(function(data){
    console.log(data);
});
```

效果也是一样的，还费劲用Promise干嘛。那么问题来了，有多层回调该怎么办？如果callback也是一个异步操作，而且执行完后也需要有相应的回调函数，该怎么办呢？总不能再定义一个callback2，然后给callback传进去吧。而Promise的优势在于，可以在then方法中继续写Promise对象并返回，然后继续调用then来进行回调操作。

链式操作的用法 所以，从表面上看，Promise只是能够简化层层回调的写法，而实质上，Promise的精髓是“状态”，用维护状态、传递状态的方式来使得回调函数能够及时调用，它比传递callback函数要简单、灵活的多。所以使用Promise的正确场景是这样的：

```
runAsync1()
.then(function(data){
    console.log(data);
    return runAsync2();
})
.then(function(data){
    console.log(data);
    return runAsync3();
})
.then(function(data){
    console.log(data);
});
```

这样能够按顺序，每隔两秒输出每个异步回调中的内容，在runAsync2中传给resolve的数据，能在接下来的then方法中拿到。运行结果如下：

猜猜runAsync1、runAsync2、runAsync3这三个函数都是如何定义的？没错，就是下面这样

```
function runAsync1(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            console.log('异步任务1执行完成');
            resolve('随便什么数据1');
        }, 1000);
    });
    return p;
}


function runAsync2(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            console.log('异步任务2执行完成');
            resolve('随便什么数据2');
        }, 2000);
    });
    return p;
}
function runAsync3(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            console.log('异步任务3执行完成');
            resolve('随便什么数据3');
```

```
        }, 2000);
    });
    return p;
}
```

在then方法中，你也可以直接return数据而不是Promise对象，在后面的then中就可以接收到数据了，比如我们把上面的代码修改成这样：

```
runAsync1()
.then(function(data){
    console.log(data);
    return runAsync2();
})
.then(function(data){
    console.log(data);
    return '直接返回数据';  //这里直接返回数据
})
.then(function(data){
    console.log(data);
});
```

那么输出就变成了这样：

reject的用法 到这里，你应该对“Promise是什么玩意”有了最基本的了解。那么我们接着来看看ES6的Promise还有哪些功能。我们光用了resolve，还没用reject呢，它是做什么的呢？事实上，我们前面的例子都是只有“执行成功”的回调，还没有“失败”的情况，reject的作用就是把Promise的状态置为rejected，这样我们在then中就能捕捉到，然后执行“失败”情况的回调。看下面的代码。

```
function getNumber(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            var num = Math.ceil(Math.random()*10); //生成1-10的随机数
            if(num<=5){
                resolve(num);
            }
            else{
                reject('数字太大了');
            }
        }, 2000);
    });
    return p;
}

getNumber()
.then(
    function(data){
        console.log('resolved');
```

```
        console.log(data);
    },
    function(reason, data){
        console.log('rejected');
        console.log(reason);
    }
);
```

getNumber函数用来异步获取一个数字，2秒后执行完成，如果数字小于等于5，我们认为是“成功”了，调用resolve修改Promise的状态。否则我们认为是“失败”了，调用reject并传递一个参数，作为失败的原因。

运行getNumber并且在then中传了两个参数，then方法可以接受两个参数，第一个对应resolve的回调，第二个对应reject的回调。所以我们能够分别拿到他们传过来的数据。多次运行这段代码，你会随机得到下面两种结果： 或者 catch的用法 我们知道Promise对象除了then方法，还有一个catch方法，它是做什么用的呢？其实它和then的第二个参数一样，用来指定reject的回调，用法是这样：

```
getNumber()
.then(function(data){
    console.log('resolved');
    console.log(data);
})
.catch(function(reason){
    console.log('rejected');
    console.log(reason);
});
```

效果和写在then的第二个参数里面一样。不过它还有另外一个作用：在执行resolve的回调（也就是上面then中的第一个参数）时，如果抛出异常了（代码出错了），那么并不会报错卡死js，而是会进到这个catch方法中。请看下面的代码：

```
getNumber()
.then(function(data){
    console.log('resolved');
    console.log(data);
    console.log(somedata); //此处的somedata未定义
})
.catch(function(reason){
    console.log('rejected');
    console.log(reason);
});
```

在resolve的回调中，我们console.log(somedata);而somedata这个变量是没有被定义的。如果我们不用Promise，代码运行到这里就直接在控制台报错了，不往下运行了。但是在这里，会得到这样的结果：

也就是说进到catch方法里面去了，而且把错误原因传到了reason参数中。即便是有错误的代码也不会报错了，这与我们的try/catch语句有相同的功能。 all的用法 Promise的all方法提供了并行执行异步操作的能力，并且在所有异步操作执行完后才执行回调。我们仍旧使用上面定义好的runAsync1、runAsync2、runAsync3这三个函数，看下面的例子：

```
Promise
.all([runAsync1(), runAsync2(), runAsync3()])
.then(function(results){
    console.log(results);
});
```

用Promise.all来执行，all接收一个数组参数，里面的值最终都算返回Promise对象。这样，三个异步操作的并行执行的，等到它们都执行完后才会进到then里面。那么，三个异步操作返回的数据哪里去了呢？都在then里面呢，all会把所有异步操作的结果放进一个数组中传给then，就是上面的results。所以上面代码的输出结果就是：

有了all，你就可以并行执行多个异步操作，并且在一个回调中处理所有的返回数据，是不是很酷？有一个场景是很适合用这个的，一些游戏类的素材比较多的应用，打开网页时，预先加载需要用到的各种资源如图片、flash以及各种静态文件。所有的都加载完后，我们再进行页面的初始化。

race的用法 all方法的效果实际上是「谁跑的慢，以谁为准执行回调」，那么相对的就有另一个方法「谁跑的快，以谁为准执行回调」，这就是race方法，这个词本来就是赛跑的意思。race的用法与all一样，我们把上面runAsync1的延时改为1秒来看一下：

```
Promise
.race([runAsync1(), runAsync2(), runAsync3()])
.then(function(results){
    console.log(results);
});
```

这三个异步操作同样是并行执行的。结果你应该可以猜到，1秒后runAsync1已经执行完了，此时then里面的就执行了。结果是这样的：

你猜对了吗？不完全，是吧。在then里面的回调开始执行时，runAsync2()和runAsync3()并没有停止，仍旧再执行。于是再过1秒后，输出了他们结束的标志。

这个race有什么用呢？使用场景还是很多的，比如我们可以用race给某个异步请求设置超时时间，并且在超时后执行相应的操作，代码如下：

```
//请求某个图片资源
function requestImg(){
    var p = new Promise(function(resolve, reject){
        var img = new Image();
        img.onload = function(){
            resolve(img);
        }
        img.src = 'xxxxxx';
    });
    return p;
}

//延时函数，用于给请求计时
function timeout(){
    var p = new Promise(function(resolve, reject){
        setTimeout(function(){
            reject('图片请求超时');
        }, 5000);
```

```
    });
    return p;
}

Promise
.race([requestImg(), timeout()])
.then(function(results){
    console.log(results);
})
.catch(function(reason){
    console.log(reason);
});
```

requestImg函数会异步请求一张图片，我把地址写为"xxxxxx"，所以肯定是无法成功请求到的。timeout函数是一个延时5秒的异步操作。我们把这两个返回Promise对象的函数放进race，于是他俩就会赛跑，如果5秒之内图片请求成功了，那么遍进入then方法，执行正常的流程。如果5秒钟图片还未成功返回，那么timeout就跑赢了，则进入catch，报出“图片请求超时”的信息。运行结果如下：
image

==几种常见的http状态码==

2XX----成功

- 200 ok 成功
- 204 No Content 请求处理成功，但是没有资源返回
- 206 Partial Content 对资源某一部分的请求

3XX----重定向

- 301 永久重定向
- 302 临时重定向
- 304 资源已找到，但是没有符合条件的请求

4XX----客户端错误

- 400 请求报文中存在语法错误
- 401 需要http认证
- 403 对请求资源的访问被服务器给拒绝了
- 404 页面找不到了

5XX----服务器错误

- 500 内部资源出故障了
- 503 服务器正在超负载的工作或者停机维护

http方法

- get----获取资源
- post----修改数据
- put----传输文件

- head----获取报文的头部
- delete----删除文件
- options----询问支持的方法
- trace----追踪路径

html&css篇

==语义化的html==

一、什么是语义化的HTML？

语义化的HTML就是正确的标签做正确的事情，能够便于开发者阅读和写出更优雅的代码的同时让网络爬虫很好地解析。

二、为什么要做到语义化？

1、有利于SEO，有利于搜索引擎爬虫更好的理解我们的网页，从而获取更多的有效信息，提升网页的权重。

2、在没有CSS的时候能够清晰的看出网页的结构，增强可读性。

3、便于团队开发和维护，语义化的HTML可以让开发者更容易的看明白，从而提高团队的效率和协调能力。

4、支持多终端设备的浏览器渲染。

三、语义化HTML该怎么做呢？

在做前端开发的时候要记住：HTML 告诉我们一块内容是什么（或其意义），而不是它长的什么样子，它的样子应该由CSS来决定。（结构与样式分离！）

写语义化的 HTML 结构其实很简单，首先掌握 HTML 中各个标签的语义，在看到内容的时候想想用什么标签能更好的描述它，是什么就用什么标签。

```
<h1>~<h6> ，作为标题使用，并且依据重要性递减，<h1> 是最高的等级。

<p>段落标记，知道了 <p>
作为段落，你就不会再使用 <br />
来换行了，而且不需要 <br />
来区分段落与段落。<p>
中的文字会自动换行，而且换行的效果优于 <br />。
段落与段落之间的空隙也可以利用 CSS 来控制，很容易而且清晰的区分出段落与段落。

<ul>、<ol>、<li>，<ul> 无序列表，这个被大家广泛的使用，<ol>
有序列表也挺常用。在 web 标准化过程中，<ul>
还被更多的用于导航条，本来导航条就是个列表，这样做是完全正确的，
而且当你的浏览器不支持 CSS 的时候，导航链接仍然很好使，只是美观方面差了一点而已。

<dl>、<dt>、<dd>，<dl> 就是“定义列表”。比如说词典里面的词的解释、定义就可以用这种列表。

<em>、<strong>，<em> 是用作强调，<strong> 是用作重点强调。

<q>也不仅仅只是为文字增加双引号，而是代表这些文字是引用来的。

```

```
21 <table>、<td>、<th>、<caption>， (X)HTML中的表格不再是用来布局。
```

补充：网络爬虫，SEO等概念

SEO：Search Engine Optimization

——搜索引擎优化，这是一种利用搜索引擎的搜索规则，采取优化策略或程序，提高网站在搜索结果中的排名。

网络爬虫：

又称网络蜘蛛、网络机器人，是一种搜索引擎用于自动抓取网页资源的程序或者说叫机器人。从某一个网址为起点，去访问，然后把网页存回到数据库中，如此不断循环，一般认为搜索引擎爬虫都是靠链接爬行的，所以管他叫爬虫。这个只有开发搜索引擎才会用到。对于网站而言，只要有链接指向我们的网页，爬虫就会自动提取我们的网页。

h5新增标签

![image](

新增元素	说明
video	表示一段视频并提供播放的用户界面
audio	表示音频
canvas	表示位图区域
source	为video和audio提供数据源
track	为video和audio指定字母
svg	定义矢量图
code	代码段
figure	和文档有关的图例
figcaption	图例的说明
main	http://www.w3school.com.cn/tags/tag_main.asp
time	日期和时间值
mark	高亮的引用文字
datalist	提供给其他控件的预定义选项
keygen	秘钥对生成器控件
output	计算值
progress	进度条
menu	菜单
embed	嵌入的外部资源
menuitem	用户可点击的菜单项
menu	菜单
section	
nav	<nav> 标签定义导航链接的部分。
aside	： <aside> 的内容可用作文章的侧栏。
article	标签规定独立的自包含内容。 一篇文章应有其自身的意义，应该有可能独立于站点的其余部分对其进行分发。
footer	

==position==

1.position中 relative和absolute，fix的区别

- fixed 属性会固定不动，不会随着屏幕的滚动滚动
- absolute：
  温馨提示的《CSS彻底研究》对绝对定位描述如下：1、使用绝对定位的盒子以它的“最近”的一个“已定位”（position属性被设置，并且被设置为不是static的任意一种）的“祖先元素”为基准进行偏移。如果没有已经定位的祖先元素，那么会以浏览器窗口为基准进行定位。2、绝对定位的框从标准流中脱离，这意味着它们对其后的兄弟盒子的定位没有影响，其他的盒子就好像这个盒子不存在一样。

3.生成绝对定位的元素，相对于 static 定位以外的第一个父元素进行定位。元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。4.子元素：position：absolute也可以==相对于父元素==position：absolute

- relative:这个是相对于，他不加定位之前的位置，定位之后，他原来的位置不清空，其他元素不能占用，会影响其他盒子的位置。

==css垂直水平居中（不知宽高）==

方法一

- 思路：显示设置父元素为：table，子元素为：cell-table，这样就可以使用vertical-align: center，实现水平居中
- 优点：父元素（parent）可以动态的改变高度（table元素的特性）
- 缺点：IE8以下不支持

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>未知宽高元素水平垂直居中</title>
</head>
<style>

.parent1{
    display: table;
    height:300px;
    width: 300px;
    background-color: #FD0C70;
}
.parent1 .child{
    display: table-cell;
    vertical-align: middle;
    text-align: center;
    color: #fff;
    font-size: 16px;
}

</style>
<body>
    <div class="parent1">
        <div class="child">hello world-1</div>
    </div>
</body>
</html>
```

方法二：

- 思路：使用一个空标签span设置他的vertical-align基准线为中间，并且让他为inline-block，宽度为0
- 缺点：多了一个没用的空标签，display:inline-blockIE 6 7是不支持的(添加上：_zoom1;*display:inline)。
- 当然也可以使用伪元素来代替span标签，不过IE支持也不好，但这是后话了

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>未知宽高元素水平垂直居中</title>
</head>
<style>
.parent2{
    height:300px;
    width: 300px;
    text-align: center;
    background: #FD0C70;
}
.parent2 span{
    display: inline-block;;
    width: 0;
    height: 100%;
    vertical-align: middle;
    zoom: 1;/*BFC*/
    *display: inline;
}
.parent2 .child{
    display: inline-block;
    color: #fff;
    zoom: 1;/*BFC*/
    *display: inline;
}

</style>
<body>
    <div class="parent1">
        <div class="child">hello world-1</div>
    </div>

    <div class="parent2">
        <span></span>
        <div class="child">hello world-2</div>
    </div>
</body>
</html>
```

方法三

- 思路：子元素绝对定位，距离顶部 50%，左边50%，然后使用css3 transform:translate(-50%; -50%)
- 优点：高大上,可以在webkit内核的浏览器中使用
- 缺点：不支持IE9以下不支持transform属性

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>未知宽高元素水平垂直居中</title>
</head>
<style>
.parent3{
    position: relative;
    height:300px;
    width: 300px;
    background: #FD0C70;
}
.parent3 .child{
    position: absolute;
    top: 50%;
    left: 50%;
    color: #fff;
    transform: translate(-50%, -50%);
}
</style>
<body>
<div class="parent3">
        <div class="child">hello world-3</div>
    </div>
</body>
</html>
```

方法四：

- 思路：使用css3 flex布局
- 优点：简单 快捷
- 缺点：兼容不好吧，详情见：http://caniuse.com/#search=flex

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>未知宽高元素水平垂直居中</title>
</head>
<style>
.parent4{
    display: flex;
    justify-content: center;
    align-items: center;
    width: 300px;
    height:300px;
    background: #FD0C70;
}
.parent4 .child{
    color:#fff;
```

```
}
</style>
<body>div> <div class="parent4">
        <div class="child">hello world-4</div>
    </div>
</body>
</html>
```

==margin折叠==

margin 折叠--margin值合并

代码

```
<!DOCTYPE html>
 <html>
 <head lang="en">
       <meta charset="UTF-8">
      <title>测试</title>
    </head>
 <style>
        *{
            margin: 0;
            padding: 0;
        }
        #father{
            width: 2000px;
            height: 2000px;
            background: #0016d9;
            overflow: hidden;
                   }
        #first-child{
            margin-top: 20px;
            background: chocolate;
            width: 60px;
            height: 60px;
        }
        #second-child{
            background: chartreuse;
            width: 60px;
            height: 60px;
            margin-bottom: 20px;
        }
        #three-child{
            margin-top:40px;
            background: fuchsia;
            width: 60px;
            height: 60px;
            display: inline-block;
        }


     </style>
```

```
<body>
    <div id="father">
        <div id="first-child">box1</div>
        <div id="second-child">box2</div>
        <div id="three-child">box3</div>
</div>
 </body>
 </html>
```

1. 子元素的父元素的间距问题：

- 在父层div加上：overflow:hidden；
- 把margin-top外边距改成padding-top内边距;
- 父元素产生边距重叠的边有不为 0 的 padding 或宽度不为 0 且 style 不为 none 的 border
  父层div加： padding-top: 1px,或者 border-top:1px ;
- 设置父元素或子元素浮动（left/right）
- 设置父元素dispaly:inline-block或者display:table-cell;
- 给父元素加上绝对定位

2. 相邻元素之间的margin

- 给最后面的元素加上浮动（left/right）
- 给最后一个元素加上display:inline-block;但是IE6和IE7下不完全支持display:inline-block，所以要加上*display:inline;zoom:1即可解决IE6、7的bug;

==清除浮动==

1、父级div定义伪类：after和zoom

复制代码

```
<style type="text/css">
   .div1{background:#000080;border:1px solid red;}

   .div2{background:#800080;border:1px solid red;height:100px;margin-top:10px}

   .left{float:left;width:20%;height:200px;background:#DDD}

   .right{float:right;width:30%;height:80px;background:#DDD}

   /*清除浮动代码*/
   .clearfloat:after{display:block;clear:both;content:"";visibility:hidden;height:0}

   .clearfloat{zoom:1}
 </style>


<div class="div1 clearfloat">
    <div class="left">Left</div>
    <div class="right">Right</div>
</div>

```

```
<div class="div2">
   div2
</div>
```

- 原理：IE8以上和非IE浏览器才支持:after，原理和方法2有点类似，zoom(IE转有属性)可解决ie6,ie7浮动问题
- 优点：浏览器支持好，不容易出现怪问题（目前：大型网站都有使用，如：腾迅，网易，新浪等等）
- 缺点：代码多，不少初学者不理解原理，要两句代码结合使用，才能让主流浏览器都支持
- 建议：推荐使用，建议定义公共类，以减少CSS代码
- 评分：★★★★☆

2.在结尾处添加空div标签clear:both

```
<style type="text/css">
   .div1{background:#000080;border:1px solid red}
   .div2{background:#800080;border:1px solid red;height:100px;margin-top:10px}

   .left{float:left;width:20%;height:200px;background:#DDD}
   .right{float:right;width:30%;height:80px;background:#DDD}

   /*清除浮动代码*/
   .clearfloat{clear:both}
   </style>
<div class="div1">
<div class="left">Left</div>
<div class="right">Right</div>
<div class="clearfloat"></div>
</div>
<div class="div2">
   div2
   </div>
```

- 原理：添加一个空div，利用css提高的clear:both清除浮动，让父级div能自动获取到高度
- 优点：简单，代码少，浏览器支持好，不容易出现怪问题
- 缺点：不少初学者不理解原理；如果页面浮动布局多，就要增加很多空div，让人感觉很不爽
- 建议：不推荐使用，但此方法是以前主要使用的一种清除浮动方法
- 评分：★★★☆☆

3.父级div定义height

```
<style type="text/css">
   .div1{background:#000080;border:1px solid red;/*解决代码*/height:200px;}
   .div2{background:#800080;border:1px solid red;height:100px;margin-top:10px}

   .left{float:left;width:20%;height:200px;background:#DDD}
   .right{float:right;width:30%;height:80px;background:#DDD}
   </style>
<div class="div1">
<div class="left">Left</div>
<div class="right">Right</div>
</div>
<div class="div2">
```

```
   div2
   </div>
```

- 原理：父级div手动定义height，就解决了父级div无法自动获取到高度的问题
- 优点：简单，代码少，容易掌握
- 缺点：只适合高度固定的布局，要给出精确的高度，如果高度和父级div不一样时，会产生问题
- 建议：不推荐使用，只建议高度固定的布局时使用
- 评分：★★☆☆☆

4.父级div定义overflow:hidden

```
<style type="text/css">
   .div1{background:#000080;border:1px solid red;/*解决代码*/width:98%;overflow:hidden}
   .div2{background:#800080;border:1px solid red;height:100px;margin-top:10px;width:98%}

   .left{float:left;width:20%;height:200px;background:#DDD}
   .right{float:right;width:30%;height:80px;background:#DDD}
   </style>
<div class="div1">
<div class="left">Left</div>
<div class="right">Right</div>
</div>
<div class="div2">
   div2
   </div>
```

- 原理：必须定义width或zoom:1，同时不能定义height，使用overflow:hidden时，浏览器会自动检查浮动区域的高度
- 优点：简单，代码少，浏览器支持好
- 缺点：不能和position配合使用，因为超出的尺寸的会被隐藏
- 建议：只推荐没有使用position或对overflow:hidden理解比较深的朋友使用
- 评分：★★★☆☆

5.父级div定义overflow:auto

```
<style type="text/css">
   .div1{background:#000080;border:1px solid red;/*解决代码*/width:98%;overflow:auto}
   .div2{background:#800080;border:1px solid red;height:100px;margin-top:10px;width:98%}

   .left{float:left;width:20%;height:200px;background:#DDD}
   .right{float:right;width:30%;height:80px;background:#DDD}
   </style>
<div class="div1">
<div class="left">Left</div>
<div class="right">Right</div>
</div>
<div class="div2">
   div2
   </div>
```

- 原理：必须定义width或zoom:1，同时不能定义height，使用overflow:auto时，浏览器会自动检查浮动区域的高度
- 优点：简单，代码少，浏览器支持好
- 缺点：内部宽高超过父级div时，会出现滚动条。
- 建议：不推荐使用，如果你需要出现滚动条或者确保你的代码不会出现滚动条就使用吧。
- 评分：★★☆☆☆

6.父级div也一起浮动

```
<style type="text/css">
   .div1{background:#000080;border:1px solid red;/*解决代码*/width:98%;margin-bottom:10px;float:left}
   .div2{background:#800080;border:1px solid red;height:100px;width:98%;/*解决代码*/clear:both}

   .left{float:left;width:20%;height:200px;background:#DDD}
   .right{float:right;width:30%;height:80px;background:#DDD}
   </style>
<div class="div1">
<div class="left">Left</div>
<div class="right">Right</div>
</div>
<div class="div2">
   div2
   </div>
```

- 原理：所有代码一起浮动，就变成了一个整体
- 优点：没有优点
- 缺点：会产生新的浮动问题。
- 建议：不推荐使用，只作了解。
- 评分：★☆☆☆☆

7.父级div定义display:table

```
<style type="text/css">
   .div1{background:#000080;border:1px solid red;/*解决代码*/width:98%;display:table;margin-bottom:10px;}
   .div2{background:#800080;border:1px solid red;height:100px;width:98%;}

   .left{float:left;width:20%;height:200px;background:#DDD}
   .right{float:right;width:30%;height:80px;background:#DDD}
   </style>
<div class="div1">
<div class="left">Left</div>
<div class="right">Right</div>
</div>
<div class="div2">
   div2
   </div>
```

- 原理：将div属性变成表格
- 优点：没有优点
- 缺点：会产生新的未知问题
- 建议：不推荐使用，只作了解
- 评分：★☆☆☆☆

8、结尾处加br标签clear:both

```
<style type="text/css">
   .div1{background:#000080;border:1px solid red;margin-bottom:10px;zoom:1}
   .div2{background:#800080;border:1px solid red;height:100px}

   .left{float:left;width:20%;height:200px;background:#DDD}
   .right{float:right;width:30%;height:80px;background:#DDD}

   .clearfloat{clear:both}
   </style>
<div class="div1">
<div class="left">Left</div>
<div class="right">Right</div>
<br class="clearfloat" />
</div>
<div class="div2">
   div2
   </div>
```

- 原理：父级div定义zoom:1来解决IE浮动问题，结尾处加br标签clear:both
- 建议：不推荐使用，只作了解
- 评分：★☆☆☆☆

==box-sizing==

box-sizing:content-box

box-sizing:content-box情况下，元素的宽度=width+pading+border； 解释：box-sizing:content-box，相当于你从网上买东西，content-box为你买的实际要用的东西，假设为A。 快递员配送快递的时候，实际上你收到的快递是带有包装的A。

类比一下，content-box是A，box-sizing是收快递的你，赋值是快递员配送， 最后你手里收到的东西就是A+包装盒，也就是content+border+padding；

/* width 和 height 属性包括内容，内边距和边框，但不包括外边距 */

box-sizing:border-box

情况下，元素的宽度=width，pading,border都包含在width里面 解释：box-sizing:border-box;相当于你从网上买东西，border-box是带有包装的你买的东西，假设为B。 快递员配送快递的时候，实际上你收到的快递就是B。

类比一下，border-box是B，box-sizing是收快递的你，赋值是快递员配送， 最后你手里收到的东西就是B；

==列举一些常见的块级元素和行内元素==

- 行内元素有：title lable span br a em b i strong
- 块级元素有：body form select textarea h1-h6 table button hr p ol ul dl div
- 行内块元素常见的有： img input td

==CSS的优先级【class，id，内联，!important】==

```
!important > 行内样式>ID选择器 > 类选择器 > 标签 > 通配符 > 继承 > 浏览器默认属性

类选择器和属性选择器优先级相同，谁在后面谁的优先级较高

注意：通用选择器（*），子选择器（>）和相邻同胞选择器（+），他们的权值是0，所以两个通配符选择器和一个通配符选择器的权值是相同的

```

1. 内联样式表的权值为 1000，就是在元素内写style
2. ID 选择器的权值为 100
3. Class 类选择器的权值为 10
4. HTML 标签选择器的权值为 1

==对inline元素设置padding、margin有效吗？==

- inline元素设置width和height无效
- 设置margin-left、margin-right、padding-left、padding-right有效
- 设置margin-top、margin-bottom、padding-top、padding-bottom无效

==line-heghit==

line-height的继承【带单位和不带单位】

line-height是可以继承的，所以子元素就可以不用重复定义line-height了。我们一般也会在后面带上单位(如:line-height:22px; 或是line-height:1.4em;)，但line-height会给人带来麻烦的地方也就是这个继承和后面加的单位。

【Css高级应用】line-height带单位与不带单位的区别

有的时候，我们为了实现单行文字的垂直居中，会给line-height一个和height相同的固定的值；有的时候，我们为了调整特定的某段文字 的行间距，通常会考虑使用百分比或者相对尺寸的em。 或许是习惯，于是我们都习惯了line-height是要有单位的。 这些情况下，我们都不需要考虑 line-height的继承,也不会发现任何问题，当然后我们在使用到line-height继承的时候，就会发现问题的所在。 例如下面的代码：

1、样式

```
<style type="text/css">
 .test{line-height:1.4em; font-size:12px;}
 span{font-size:30px; font-weight:bold;}
 </style>
```

2、HTML结构

```
<div class="test">
    <span> 白培铭先生于1960年出生于中国台湾。<br/>
            毕业于中国台湾省清华大学核物理系，<br/>
    </span>
    之后留学于美国加州大学伯克利分校和密西根大学，获得双硕士学位。<br/>
    在工作之后，凭着对营销领域的浓厚兴趣，他又考入密执安大学深造<br/>
</div>
```

看过例子后，你会发现，只要有单位的line-height继承，都发生了重叠的现象。那到底这是什么原因导致的呢？

- 如果line-height属性值有单位，那么继承的值则是换算后的一个具体的px级别的值；
- 而如果属性值没有单位，则浏览器会直接继承这个“因子（数值）”，而非计算后的具体值，此时它的line-height会根据本==身的font-size==值重新计算得到新的line-height 值。

==rem和em的区别？==

css中单位em和rem的区别 在css中单位长度用的最多的是px、em、rem，这三个的区别是：

- px是固定的像素，一旦设置了就无法因为适应页面大小而改变。
- em和rem相对于px更具有灵活性，他们是相对长度单位，意思是长度不是定死了的，更适用于响应式布局。
- 对于em和rem的区别一句话概括：em相对于父元素，rem相对于根元素。
- rem中的r意思是root（根源），这也就不难理解了。

em

- 子元素字体大小的em是相对于父元素字体大小
- 元素的width/height/padding/margin用em的话是相对于该元素的font-size

rem

- rem是全部的长度都相对于根元素，根元素是谁？元素。通常做法是给html元素设置一个字体大小，然后其他元素的长度单位就为rem。

==元素position有哪几种【static、relative、absolute、fixed】==

- static：没有定位，在正常的流中
- relative：相对于正常位置
- absolute：相当于第一个父元素进行定位
- fixed：相对于浏览器的窗口进行定位

==BFC==

块级格式上下文Block Formatting Context（简称BFC ），这是Web页面的一种布局方式，通俗点说，也是指页面上一个渲染区域，里面的元素按文档流中的顺序垂直排列，并且发生垂直方向上的margin折叠，同时这个区域内的元素布局不会对外面的元素有任何影响，反过来，也是一样。

当元素满足一下任何一个条件是都会产生一个BFC:

- float属性取值不是“none”
- overflow属性取值不是“visible”

- display的值为 “table-cell”, “table-caption”, or “inline-block”中的任何一个
- position的值不为 “static” 或 “relative”中的任何一个

相关文献

https://www.qdfuns.com/article/51117/a08e225f766e9f31c4787f58e1b5b484.html