Javascript

## 浏览器

内核：

主流的四大浏览内核：webkit、trident、gecko、presto

苹果：webkit(基于KHTML) chrome:Blink(原webkit) firefox:Gecko(基于Netscape)

微软：Trident opera:Presto

Webkit:微量，快速 {苹果，安卓默认浏览器}

Presto:标准化、微量（甚至可以运行在功能机中）{opera}

Trident:市场份额高（早先），但标准型过差 {ie}

Gecko:体积较大，占用内存高，功能多，速度快 {firefox}

浏览器内核参考资料：<https://www.jianshu.com/p/133a7f65c622>

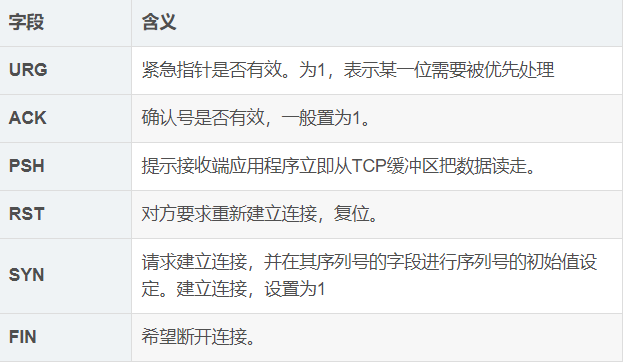
TCP理解：  
序列号seq：4字节，用来标记数据段顺序，报文段中的第一个字节的数据标号

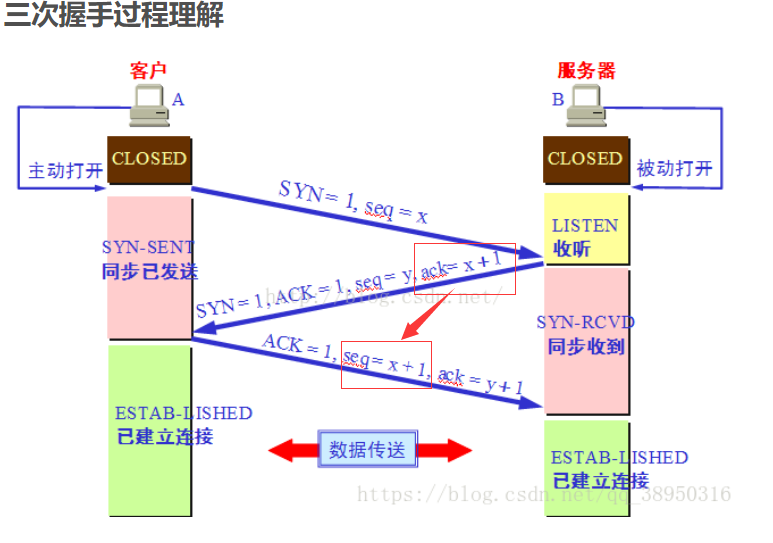
确认号ack：4字节，期待收到对方下一个报文段的第一个数据字节序号，即当前ack = seq(最后一字节) + 1

确认ACK：1字节，当ACK=1时，确认号字段才有效

同步SYN：连接建立时用于同步序列号。SYN只有在TCP建立连接时才被置为1，握手完成后SYN被置为0（第一次握手客户端SYN：建立请求，第二次握手服务器SYN：同意请求，此时服务器已同意了双方交流）

终止FIN：用来释放一个连接。FIN=1：此报文段的发送方数据已发送完毕，并要求释放运输连接



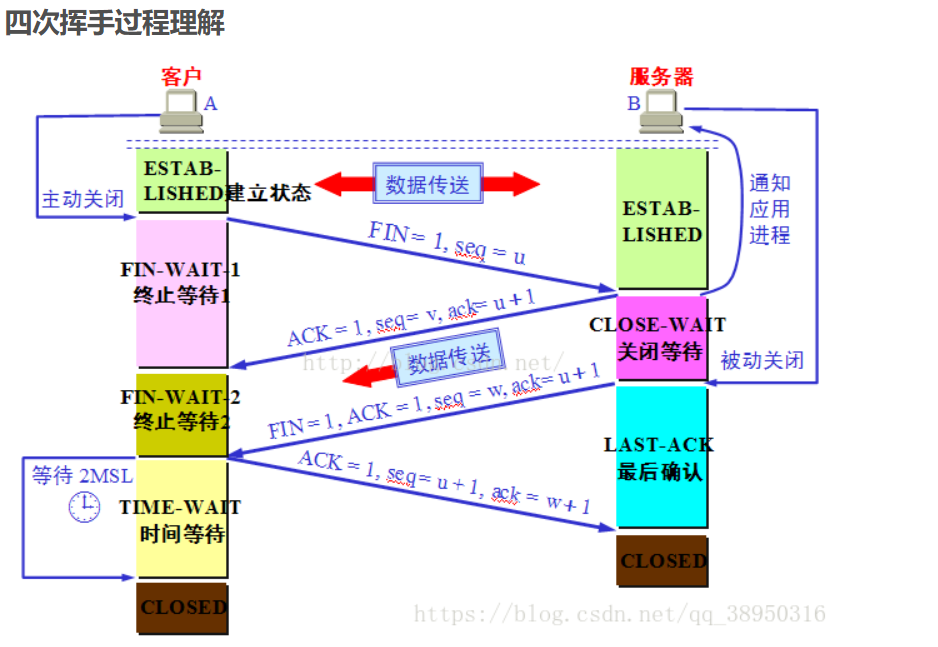


第一次握手（客户端）：发送SYN=1请求建立连接，并发送一个随机seq（假如为1343）

第二次握手（服务器端）：发送SYN=1同意建立连接，也发送一个随机seq（假如为3456），同时发送一个ack（1344）

第三次握手（客户端）：发送ACK=1确认服务器的ack有效，同时发送一个符合服务器期待的seq(ack)与下一个ack

此时成功建立连接，客户端再发送TCP段，成功后，服务端返回一个ack(接收客户端data的ack)



第一次挥手（客户端）：发送FIN=1请求断开连接，同时发送了seq=a（上一次的ack+1）

第二次挥手（服务器段）：发送ACK=1确认报文，同时发送ack=a+1,seq=c(服务器的数据序列号)

服务器发送数据..：seq = n

第三次挥手（服务器段）：发送FIN=1,ACK=1,seq = d(上一次数据序列号+1:(n+1))，ack=a+1,进入最后确认状态

第四次挥手（客户端）：发送ACK=1(确认报文),seq=a+1(序列号),ack=d+1(确认号)。服务器接收到此消息后立即关闭，客户段发送此报文后需等待2MSL（最长报文段寿命）（避免最后一个ACK包服务器没接收到）后再关闭

【

第二次挥手后ack不变化的原因：此时只存在服务器端向（单向）客户端发送数据，客户端只接收数据，所以ack不产生变化

】

TCP参考资料：<https://blog.csdn.net/qq_38950316/article/details/81087809>

## IE8

获取ie浏览器版本：

//获取版本信息

var ua = navigator.userAgent.toLowerCase();

//判断是否ie浏览器【ie: msie】

var isIE = ua.indexOf("msie")>-1;

//获取ie版本

ua.match(/msie ([\*d*.]+)/)[1]

1、无法使用console.log()

//解决 IE8、IE9 不支持 console 问题

window.console = window.console || (function () {

var c = {}; c.log = c.warn = c.debug = c.info = c.error = c.time = c.dir = c.profile

= c.clear = c.exception = c.trace = c.assert = function () { };

return c;

})();

2、数组最后一个需去掉”，”，否则视为一个空数组

3、对display的兼容性很差，如flex,inlin-flex等

4、无法使用trim()去除空格，但是有替代方法：

//去左空格;

function ltrim(s){

 return s.replace(/(^/s\*)/g, "");

}

//去右空格;

function rtrim(s){

  return s.replace(/(/s\*$)/g, "");

}

//去左右空格;

function trim(s){

    //s.replace(/(^/s\*)|(/s\*$)/g, "");

 return rtrim(ltrim(s));

}

5、不支持rgba设置透明度， 解决方法： opacity或alpha（ie独有）

【区别： opacity设置整体透明度（继承性），alpha设置单独元素透明度】

6、不支持text-shadow属性(ie10支持)

7、不支持border-radius

8、不支持transform

## ECMAScript(ES6)

高版本兼容低版本浏览器方法：

1. 在线编译

添加script文件如：babel.js(browser.js)【可以自动将es6降级转化es5】

其中需要在script标签中添加编译名 <script type=’text/babel’></script>

1. 提前编译

ES6：

### 变量

let 变量 const 常量  
 新增块级作用域 || 不可重复声明

### 函数

箭头函数【

() =>{}

let aa=b=>{return b\*2} 如果变量只有一个，()可以省略

let aa=b=>return b\*2 如果return只有一个，{}可以省略

】

### 数组

“…”

收集数据

function show(a,b,…c){} show(1,2,3,4,5,6) …c=3,4,5,6

数组展开

a=[1,2,3] === …a

默认参数

function show(a,b=10,c=20){} show(1,2,3) – a=1,b=2,c=3

show(100) -a=100,b=10,c=20

解构赋值

结构必须对称

let [a,b,c]=[1,2,3] let [a,b]=[{1,2,3},[4,5,6]]

声明和赋值必须放在一起

let [a,b] = [1,2]

**map(映射)**

let list = [1,2,3] var result=list.map(item=>item\*2);

**reduce(汇总)**

let result = list.reduce(function(tmp, item, index){

return tmp + item; //6

})

**filter(过滤器)**

let arr=[1,2,3,4,5,6]

let result=arr.filter(function(item){

return item%3==0;

})

**forEach(循环|迭代)**

let arr=[1,2,3,4,5]

arr.forEach((item,index)=>{return item\*index})

**Object.assign(复制对象) 复制一个或多个对象到目标对象，并返回目标对象**

let obj = {user:123},

let arr = [1,2]

Object.assign({},obj, arr) //{1:1, 2:2, user:123}

### 字符串

startsWidth() 开头字符

let str=’absdgfdg’;

if(str.startsWidth(‘abs’){alert(str)})

endsWidth() 字符串结尾

字符串模板

直接把东西塞到字符串里面

可以换行

let a = 123;

let b = 321;

let b = `aaa${a + b}` //b: aaa444

### 面向对象

类、构造器

**class** Student{

**constructor**(name, age){

this.name = name;

this.age = age;

}

showName(){

console.log(this.name);

}

showAge(){

console.log(this.age);

}

}

继承

class Score **extends** Student{

constructor(name,age,score){

**super(name, age)**;

this.score = score;

}

showScore(){

console.log(this.score);

}

}

let report = new Score(“张三”, 20, 80);

report.showName();

report.showAge();

report.showScore();

json

标准json: {“aaa”: 123, “bbb”: “bbb”}

stringify() 对象转json || parse() json转对象

let list1 = {a:123, b:”飞洒地方”}

let list2 = ‘{“a“:123, “b”:”飞洒地方”}’

let show1 = encodeURIComponent(JSON.stringify(list1))

let show2 = JSON.parse(list2);

es6-json 简写

1、当键名与键值名相同可只写一个

let [a,b]= [1,2]

let json = {a, b, c : 12} // {a: 1, b: 2, c: 12}

2、方法function可简写

let json = {

a: 123,

//show: function()

show(){

console.log(this.a)

}

}

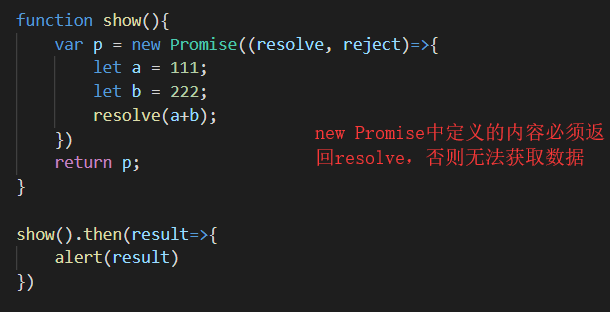
### Promise

作用：将异步操作以同步方式写出来（高版本Jquery自带promise）

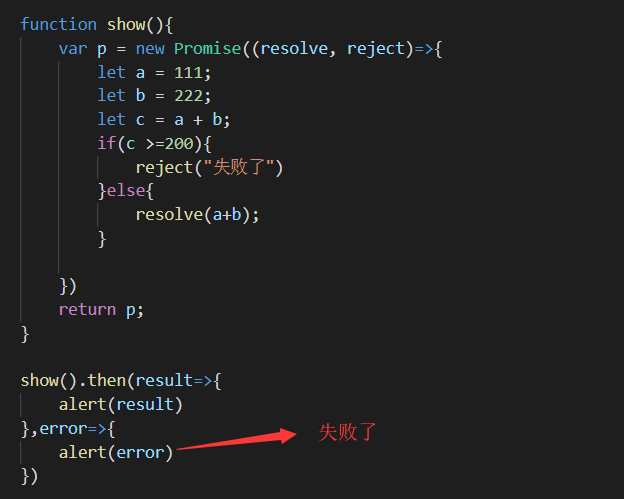
适用于读取一堆数据，不做逻辑分析

Promise对象无法转化为js对象

**resolve:**



**reject:**



**catch:**



运行的方法中return 的内容自动添加进resolve|reject对象

//谁执行的慢，以谁为准执行回调（并发）

Promise.all([$.ajax(),$.ajax()]).then(

success=>{

let [a,b] = success;

console.log(“全都成功了” + a + “\n” + b)

},

error=>{

console.log(“至少有一个失败” + error);

}

)

//谁执行的快，以谁为准执行回调，当一旦有方法执行完毕后马上执行then内容，其余未完成的方法会继续执行，

Promise.race([ 竞速

$.ajax(),$.ajax()

].then(data=>{console.log(data)}))

### generat

适用于读取数据中存在逻辑分析的情况

generrator -生成器

generat – 生成

普通函数 – 一路执行到底

generator函数 – 中间能停止

function **\*** show(){

console.log(“aaa”);

**let data = yield $.ajax() || function(){…};**

console.log(“bbb”)

console.log(data);

}

let geneObj = show();

geneObj.next(); aaa

geneObj.next(); bbb

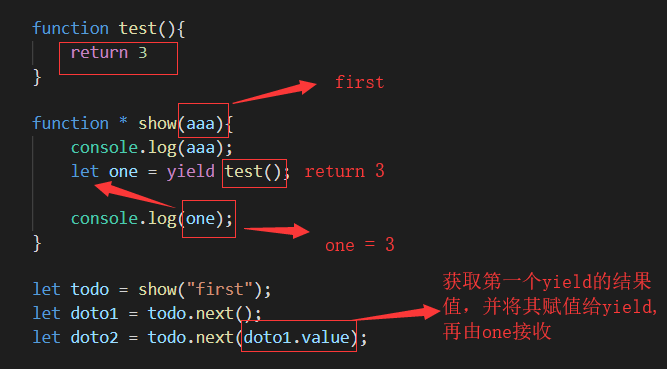
yield(传参、返回)可多个

传参：

geneObj.next() 第一个yield无法传参，想传参靠show(data)…

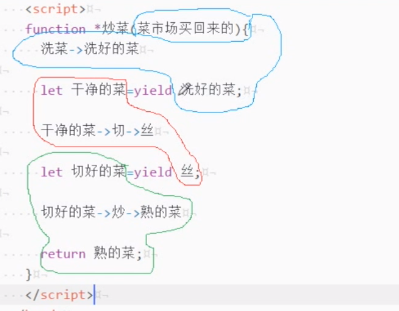
geneObj.next(555) 后面可以传参

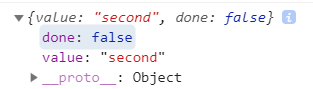
返回：



let data1 = geneObj.next(); 第一部分获取的值（yield之前）

let data2 = geneObj.next(); 整个方法的返回值（return ..）





Promise与generat结合使用产生异步效果



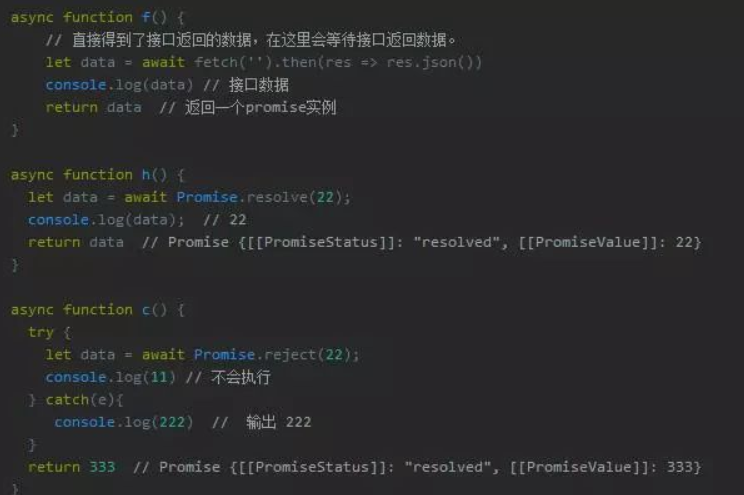
### 模块化

## ECMAScript(ES7)

### async/await

await:接收一个Promise实例

async：返回一个Promise对象



## 正则表达式

### 基础使用

var regBox = /aa/i

var str = ‘a + a = aa’

regBox.test(str) //true

【

regBox.test(str)：判断是否满足表达式

str.match(regBox)：获取满足条件的内容，并存储为数组

】

### 语法

“^”：匹配行或者字符串的起始位置

“$”：匹配行或字符串的结尾

regBox=/^ab$/

“\b”：匹配单词边界，若匹配的字符为单独单词，true：

regBox=/\baa\b/ regBox.test(“this is aa”) true

regBox=/\baa\b/ regBox.test(“this is sdfaa” | “this is aaaa”) false

“\d”：匹配数字

regBox=/123\d/ --匹配是否含有123

regBox.test(‘123435345fdc’) true

“\w”：匹配字母，数字，下划线

“\s”：匹配空格

“/g“：在字符串中查找所有可能的匹配，返回的结果可能是多个

“/i“：匹配不区分大小写

“/m”：多行匹配，匹配换行符两端的潜在匹配

“.”：匹配除了换行符以外的任何字符

“[^abc]”：字符组，匹配包含括号内元素的字符

regBox=/[^a]d$/ --匹配以a开头，以d结尾

regBox.test(‘adfgadgwdfd’) true

反义【将匹配改成大写，意思与原来相反】

“\W”：匹配任意不是字母，数字，下划线的字符

“\D”：匹配任意非数字的字符

“\S”：匹配任意不是空白符的字符

“\B”：匹配不是单词开头或结束的位置

“[^abc]”：匹配除了abc以外的任意字符

量词

“\*”：贪婪量词，匹配全部，选定尽可能多的内容，如果失败，回退一个字符，再尝试匹配，回退，此过程叫回溯，每次回退一个字符，知道找到匹配的内容或者没有字符可以回退，消耗最大

“?”：懒惰量词，从目标起始位置开始匹配，每次检查一个字符，查找匹配内容，循环到字符结尾

“+”：占有量词，覆盖整个目标字符串，尝试寻找匹配内容，只尝试一次，不会回溯

应用： --

“\*”：重复0次或更多

“+”：重复一次或更多次

“?”：重复零次或一次

“{n}”：重复n次

“{n,m}”：重复n到m次

“{n,}“：重复n次或更多次

### 实例

1. 查找字符串中的hi

var str=”Hi,my name is hi,hI,HI”;

var regBox=/\bhi\b/gi --\b：查找单词 g：返回全部结果（可多个） i：不区分大小写

str.match(regBox) //[“Hi”,”hi”,”hI”,”HI”]

1. qq邮箱验证

var str = [2826315773@qq.com](mailto:2826315773@qq.com)

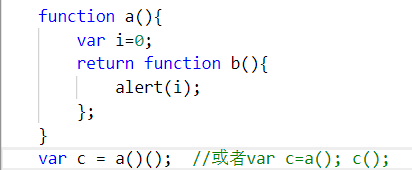
var regBox =/ (\d{10})@([q]{2})\.(com|cn)\b/gi | /[0-9a-zA-Z]\*@qq(\.)(com|cn)\b/gi

## 闭包

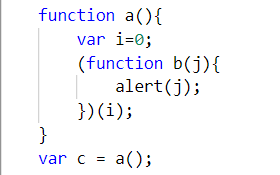
js特性，局部变量由内向外权限递减，方法内变量为私有变量，只有方法内的方法拥有访问权限

闭包作用：根据js此特性，在外部通过调用内部方法获取到方法内的私有数据，通常用来保存某特殊数值或状态。如：

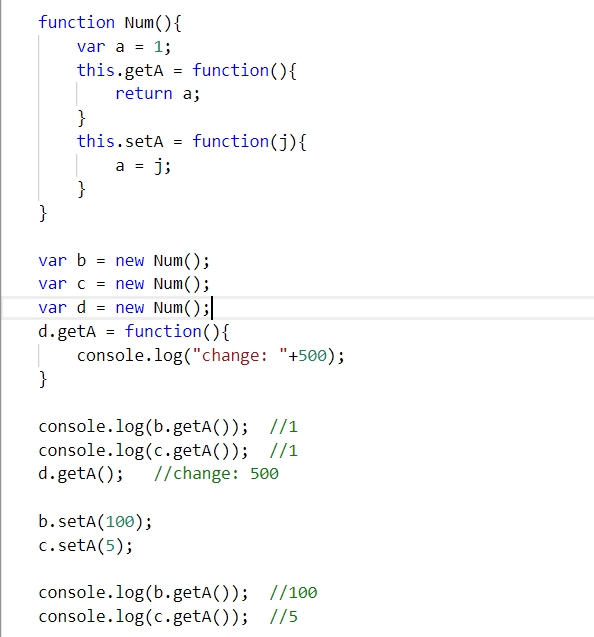
手动调用内部函数b: (c = 0)



自动调用内部函数b: (c = 0)



闭包的使用（定义私有方法、函数等）：

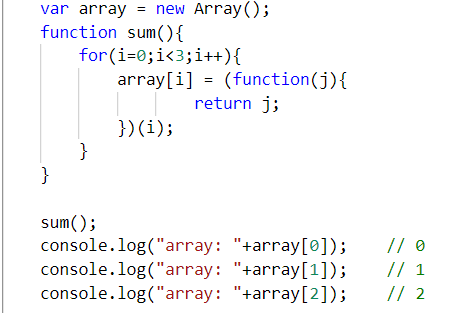


闭包由于不进入垃圾回收机制（方法一直存在被引用状态），并且每个对象都会产生独立的方法，即使一模一样，所以导致大量浪费内存，需谨慎使用。

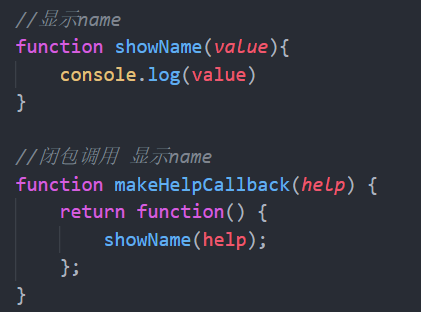
闭包在循环中使用时产生的数值只保留最后值的原因与解决方案：

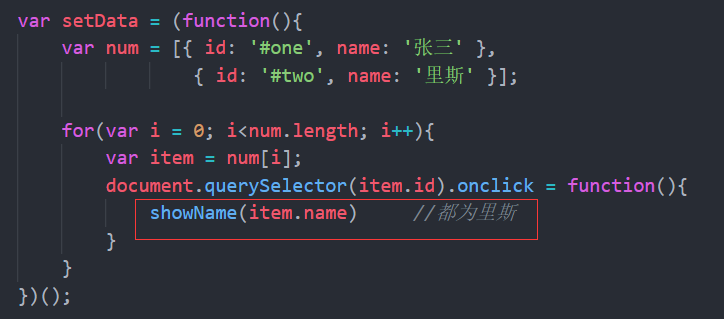
原因：其中的方法中的变量记录的是变量的索引地址，因此当其数值改变时，其中的值同时变化。

解决方法：将闭包中的方法改写为自动执行，并使用局部变量记录数值，达到复制的效果。



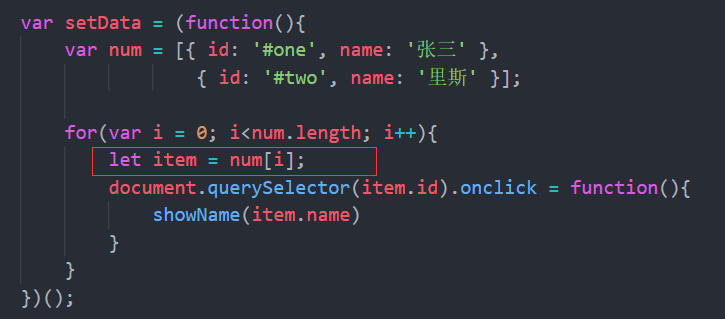
赋值后都为最后一个（i作用域）：



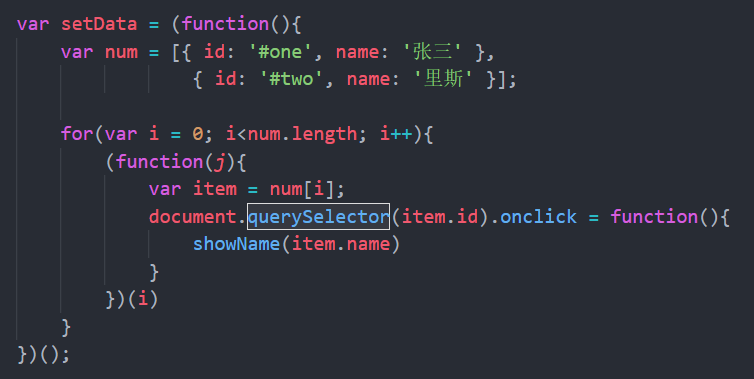


解决：

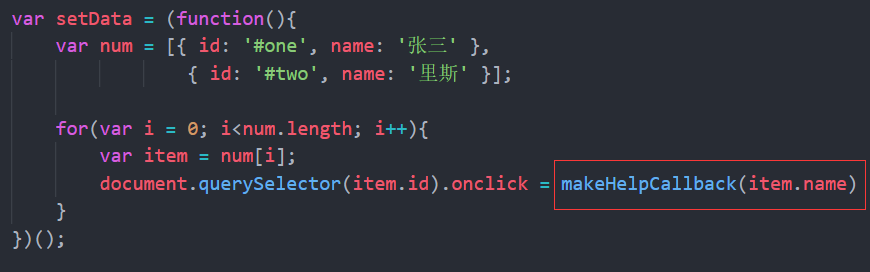
1. 使用let，使每个item单独作用



1. 自运行中绑定作用域



1. 闭包



作用：将变量私有化、单独作用域

参考资料：<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Closures>

## 防抖函数

效果：在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时。

*// 防抖函数*

function debounce(*func*, *wait*) {

    let timeout;

    return function () {

        let context = this;

        let args = arguments;

        if (timeout) clearTimeout(timeout);

        timeout = setTimeout(() => {

            func.apply(context, args)

        }, wait);

    }

}

## 节流函数

效果：规定在一个单位时间内，只能触发一次函数。如果这个单位时间内触发多次函数，只有一次生效。

*// 节流函数*

function throttle(*func*, *wait*) {

    let timeout;

    return function() {

        let context = this;

        let args = arguments;

        if (!timeout) {

            timeout = setTimeout(() => {

                timeout = null;

                func.apply(context, args)

            }, wait)

        }

    }

}

## 派发事件

1. var evt = new Event(‘name’);

evt.data = {price: 123}

1. var evt = document.createEvent('Event');

evt.initEvent('btnClick',true,true);

evt.data = {price: 123}

3、docuemt.dispatchEvent(evt)

4、document.addEventListener(‘name’, function(e){console.log(e.data)})

1和2都为创建自定义事件，name为自定义事件名称,随之给事件添加属性（此处为data,用于存储数据）

3的作用是派发事件（触发evt事件）

4在全局监听‘name’事件，若此事件被派发出来，那么就会执行后续的function,值得注意的是，若监听处于事件派发之后，那么也不能监听到事件派发

作用：一处创建事件，可多处派发而获取到事件中的数据，类似继承中的父类

参考资料：<https://blog.csdn.net/weixin_44272533/article/details/85719828>

## 保留小数点位数

【四舍五入】xx.toFixed(num) //自动根据设置的保留位数计算

【不四舍五入】Math.floor(xxx) Math.floor(31.415926 \* 100)/100 //31.41

## 节点

### 元素节点（Dom）

直接获取节点：

Document.querySelector(“xxx”); //返回查找到的第一个节点

Document.querySelectorAll(“xxx”); //返回查找到的第所有节点 – 数组

childNodes获取节点：

document.querySelector(“xxx”).childNodes; //返回数组（包括空格等）

children获取节点：

document.querySelector(“xxx”).children[0] //返回数组

firstElementChild获取第一个子节点：

document.querySeletor(“xxx”).firstElementChild

//返回父元素的第一个子元素节点（无空格）

LastChild获取最后一个子节点：

Document.querySelector(“xxx”).lastChild //获取最后一个节点 （含空格）

Document.querySelector(“xxx”).lastElementChild //获取最后一个元素节点（无空格）

parentNode、parentElement(ie标准，效果一样) 获取父节点：

document.querySelector(“xxx”).parentNode

offsetParent获取所有父节点：

document.querySelector(“xxx”).offsetParent //body下的所有节点信息

previousElementSibling、previousSibling(匹配字符，包括换行、空格)获取兄弟节点：

document.querySeletor(“xxx”).previousElementSibling

nextElementSibling、nextSibling（匹配字符）获取下一个兄弟节点：

document.querySelector(“xxx”).nextElementSibling

创建新元素节点： createElement(element)

创建一个包含文本的新文本节点：createTextNode()

将节点插入另外元素节点之前：insertBefore()

将节点插入到节点给定节点之内：appendChild()

从一个元素中删除节点：removeChild()

将父元素中的节点替换为另外节点：replaceChild()

切换内容

1. 切换节点内内容:

Input: Document.querySelector(“xxx”).value = xxx

Text: document.querySelector(“xxx”).innerText = xxx

Html: document.querySelector(“xxx”).innerHTML = xxx

【注意：替换html时，此中方法替换的html中绑定的事件，如onclick等，原生js方法可运行，若加入其他方法- vue @click，文档认其为字符，事件失效，在vue中想切换html可使用组件模式，使用一个组件切换其中内容，组件中事件正常运行】

### 文档节点（Range与Selection）

Randge对象：页面上一段连续区域

创建对象：var range = document.createRange();

Selection对象：每个浏览器窗口都有一个Selection对象，代表用户鼠标在页面中所选区域

创建对象：var selection = window.getSelection(); or document.getSelection();

FireFox可通过ctrl选取多个区域，即，一个selection中拥有多个range

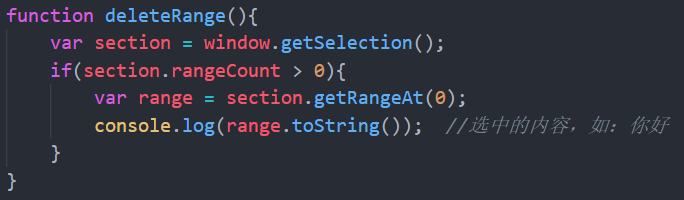
Chrome、Safari中只能选取一段区域，即一个selection中只有一个range

获取selection中得range对象：

var range = document.getSelection().getRangeAt(index) 从0开始，firefox可能多个



获取选中的range：



Range对象属性：

collapsed: (boolean) 判断所选区域起始点与终点是否相同

commonAncestorContainer: (node) 返回Range对象区域所处的节点

endContainer: (node) 返回Range对象区域终点所处节点

endOffset: (int) 返回Range对象区域的终点与包含该终点的节点的起始点之间的距离

startContainer: (node) 返回Range对象区域起点所处节点

starOffset: (int) 返回Range对象区域的起点与包含该起点的节点的起始点之间的距离

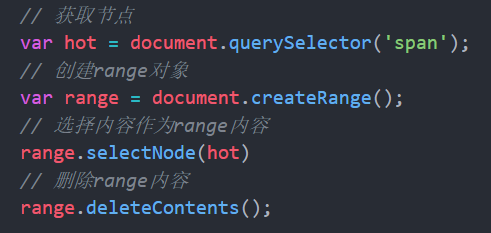
Range对象的方法：

selectNode: 选取节点内容（包含节点本身）

selectNodeContents：选取节点内容（不包含节点本身）

deleteContents：删除range内容

例：



setStart：将节点起点作为range起点位置

setEnd：将节点终点作为range终点位置

当选取内容为节点时，将setStart – index - dom作为range起点

当选取内容为文本时，将setStart – index – txt 作为range起点（如删除第一个字）

例：



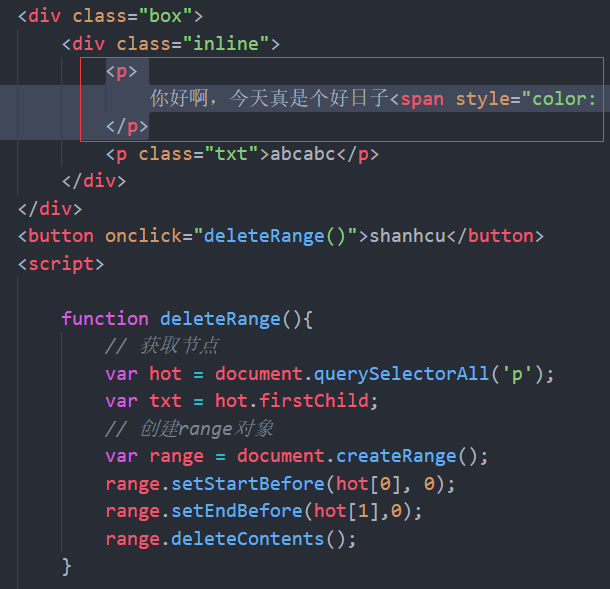
setStartBefore：将节点起始位置作为range的起点位置

setStartAfter：将节点终点位置作为range起点位置

setEndBefore：将节点起始位置作为range终点位置

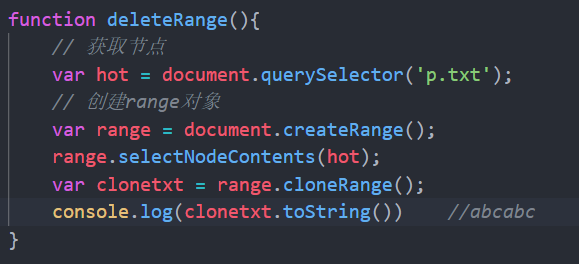
setEndAfter：将节点终点位置作为range终点位置

例：



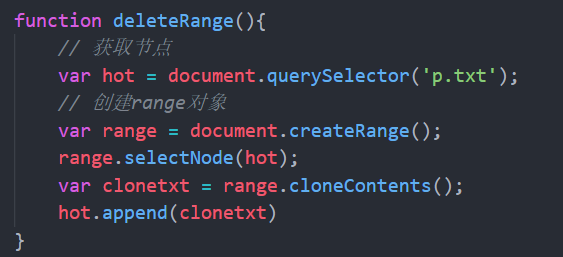
cloneRange：对当前range对象进行复制，返回复制的range对象

例：



cloneContents：在页面上追加一段html代码

例：



extraContents：将range对象区域的html代码克隆到DocumentFragment中，再将此

html从页面中删除

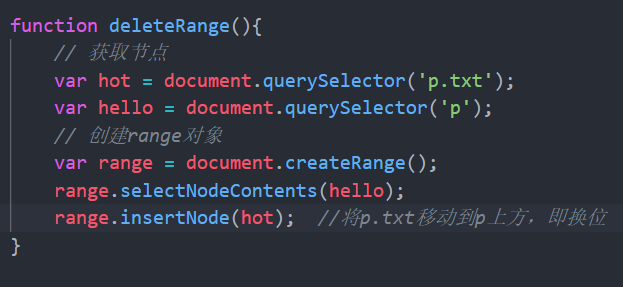
例：



insertNode：将指定节点插入到range对象区域，插入位置为range对象区域的起点位

置，若该节点已存在于页面中，那么此节点被移动到range区域起始点

例：

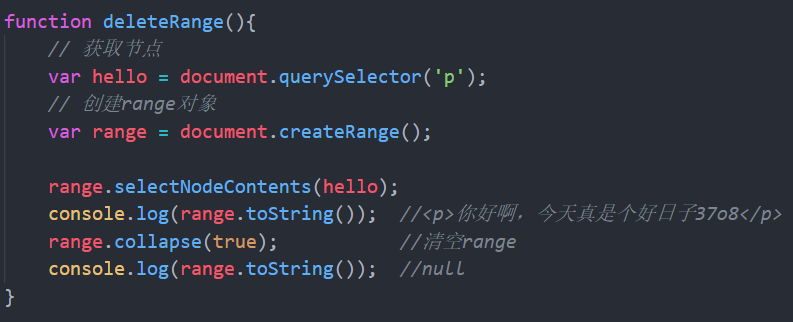


collapse：将range区域的终点移动到该区域起点处，或将range区域起点移动到终点

处，使range区域内不包含内容（清空range内容），collapse(boolean)：

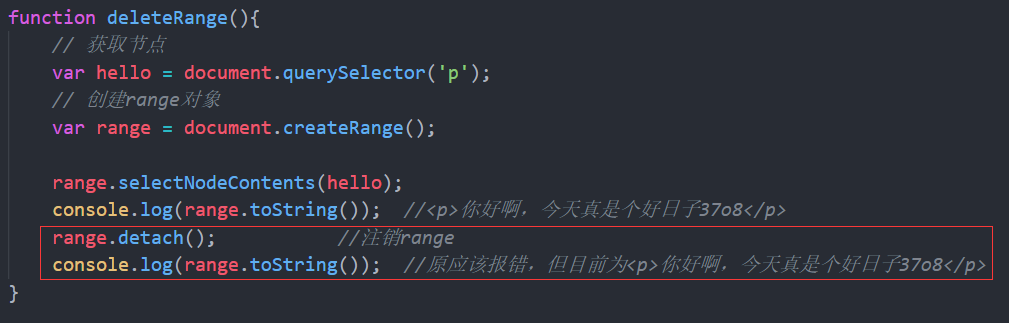
true:终点移至起点， false:起点移至终点

例：



detach：从浏览器中释放range，后不能再访问range对象

例：（暂无效果，未找到原因）



## 动态操作css样式

1. 对元素添加内联样式：

【

var intentNode = document.querySeletor(“intentNode”);

intentNode.style.height = 100 + ‘px’ //只对元素内直接样式起效

intentNode.setAttribute(“style”,”margin:100px,…”)

】

1. 修改元素内部样式：

【

var intentNode = document.styleSheets[x].cssRule[x]. selectorText == intentNode

(查找元素位置)

IntentNode. document.styleSheets[x].cssRule[x].cssText = ‘your style’

(对目标元素设置样式)

】

## 点击获取当前列表下标

Target: 返回当前点击的元素节点

currentTarget: 返回绑定事件的元素节点，如下：对父元素one绑定了点击事件，currentTarget时全都返回one元素

<body>

<div class="one">

<div class="two"></div>

<div class="three"></div>

</div>

</body>

<script>

var target = document.querySelector(".one");

target.onclick = function(e){

console.log("target: " + e.target.getAttribute('class'));

【//two / three】

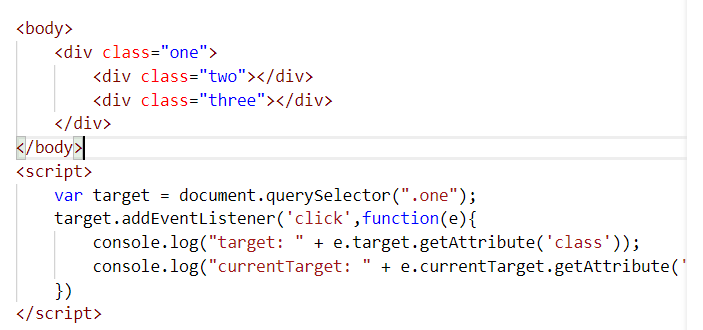
console.log("currentTarget: " +

e.currentTarget.getAttribute('class'));

【//one】

}

</script>



## 作用域链

作用域链优先级：由内而外，优先级从高到低（本身最高，window最低），

查找方法：从内而外，层层查找

## this指向

### window对象

* var aaa = 10; //window.aaa = 10
* function fc(){console.log(10)} //window.fc() === fc() 10
* setTimeout(console.log(this.aaa),100) //setTimeout内this 永远指向window 10
* 嵌套function时，只要父级未绑定this指向，默认指向window,其内尽皆随之指向window

### 绑定this

```

var value = 111;

function show() {

var value = 999;

this.person = function () {

var value = 888;

console.log('person: ' + this.value) //obj 222

}

this.person1 = function () {

console.log('person1-1: ' + value); //未绑定this遵循作用域链 999

console.log('person-this: ' + this); //obj

console.log('person1: ' + this.value) //obj 222

function person2() {

console.log('person2 -this: ' + this) //window

console.log('person2: ' + this.value); //window 111

var value = 777;

this.person3 = function () {

console.log('person-3: ' + value); //作用域链 777

console.log('person3: ' + this.value); //window 111

var value = 666;

this.person4 = function () {

console.log('person-4: ' + value); //作用域链 666

console.log('person4: ' + this.value); //window 111

}

}

}

person2(); *//window*

person3(); *//window*

person4(); //window

}

console.log('show -this: ' + this) //obj

console.log('show: ' + this.value) //obj 222

this.person(); //obj 222

this.person1(); //obj 222

}

var obj = {

value: 222,

show: show

}

var objj = {

value: 333,

obj: obj,

show: show

}

obj.show(); == objj.obj.show(); //show()指向obj

//obj.show.call(objj) //show()指向objj

new绑定：

```

function toStudent(age){

this.age = age //指向student

}

var student = new toStudent(20);

console.log(student.age) //student 20

//通过new方式给对象绑定属性

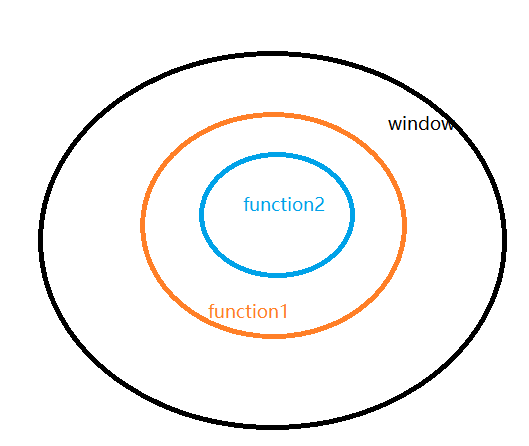
```

**总结：this只指向直接父级，不能跨级访问**

**es6箭头函数：只遵循外层（函数或全局）作用域，类似上下文关系**

**this优先级：默认绑定🡪隐式绑定 🡪显示绑定 🡪 new绑定 从弱到强**

1. **默认绑定：window**
2. **隐式绑定：上下文关系**
3. **显示绑定：call() | bind() | apply()**
4. **new绑定：new obj()**



### call()、apply()与bind()

相同点：均可以改变this指向

不同点：1、apply接收一个函数并产生一个新函数

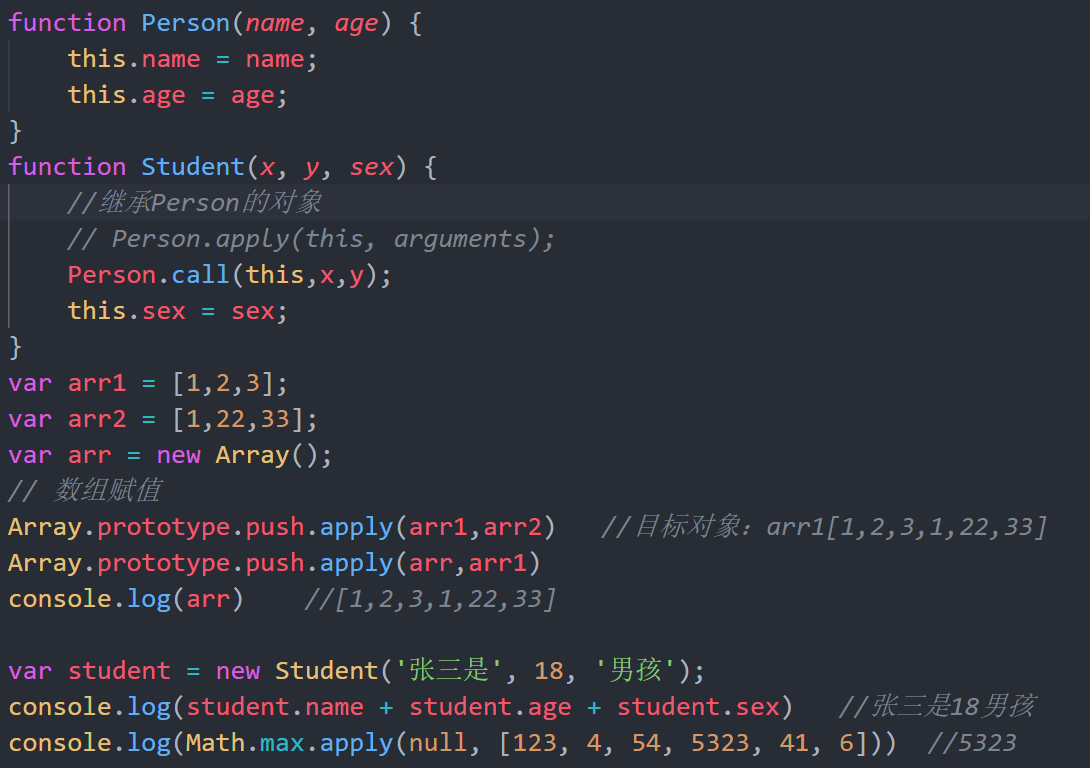
2、call、bind接收多个参数绑定到函数，参数单一传入，apply接收数组

运用：function add(){conosle.log(this)}

add.bind(obj, 数值) //绑定null或不绑定为全局

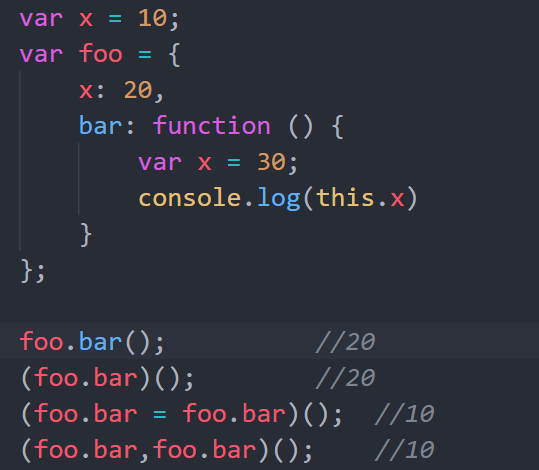
add.call(obj, 数值)

var newAdd = add.apply(obj, [1,2,3..]) //返回一个新函数newAdd()



this经典试题：

```



解析：

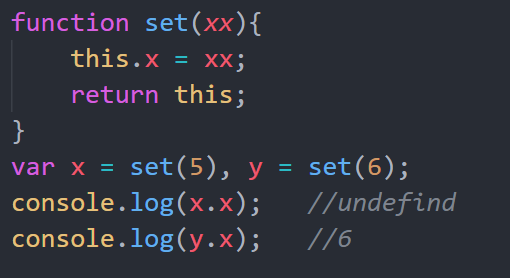
1. 常规-运行bar函数，因绑定的foo,this指向foo,this.x = 20
2. 自动运行函数，在此与第一类似，指向foo
3. 自动运行内部赋值语句，因其内为赋值运算，执行右边部分，左边为全局变量, foo.bar等同于window.foo.bar,即变为引用变量，所以指向window，this.x = 10
4. 自动运行内部添加逗号运算符的函数，

猜测：自动运行函数间使用逗号运算符后，只运行返回最后（最右）一个函数 ，逗号运算符中涉及的变量操作可正常执行，与正常的js右前向后运行规则一致，因为此处处于自动运行函数中，所以函数只运行最后，去除自动运行后依序运行

```

this经典试题

```



解析：

1、x = set(5), set中this指向window，即window.x = 5, return window, x = window

x.x = window.

2、y = set(6), set中this指向window，即window.x = 6, return window, y = window

y.x = window.x

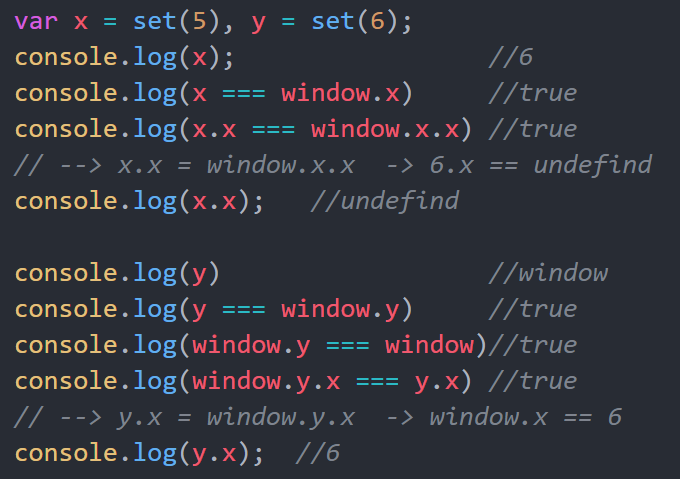
坑：若按照此逻辑运行，x = window, y = window,那么后面x.x、y.x = window.x = 6才

对，为什么x.x应该也是6，怎么会undefind呢，这也就是一个坑了

【默认this指向window !!!】

3、console.log(x.x), 此时x = 6 -> 6.x = undefind

4、console.log(y.x), 此时 y = window -> window.x = 6



```

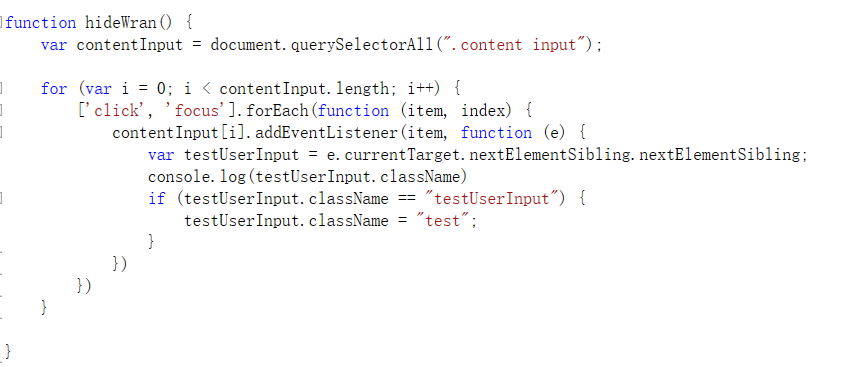
## 鼠标事件

### 点击

onclick 、 click 【js方法： dom.click() || dom.onclick()】

dom. addEventListener(‘click’, function(){…})

addEventListener绑定多事件:



oncontextmenu：鼠标右键事件

onmousedown：鼠标按下事件

onselectstart：鼠标选中（不包括input、textarea）

onselect：鼠标选中（适用Input、textarea）

ondragstart：拖放事件

oncopy：复制事件

onpaste：粘贴事件

onfocus： 当元素获取到焦点的时候触发【odiv.onfocus = function(){}】

onblur： 当元素失去焦点的时候

obj.focus() : 给指定的元素设置焦点

obj.blur() ：取消指定元素的焦点

obj.select()：全选当前的文字

鼠标点击参考资料： <https://blog.csdn.net/weixin_41342585/article/details/80659736>

W3C参考资料：<http://www.w3school.com.cn/jsref/dom_obj_event.asp>

## 操作数据

Object. defineProperty(obj, propName, descriptor) – 对象名，属性名，描述

【当使用set和get后，不可使用writable，否则报错】

descriptor：{

writable：是否可被重新赋值

enumerable：是否可枚举（遍历）

configurable：是否可被删除或修改

set:function(){}：设置属性值

get:function(){}：获取属性值

}

实例：

let user = {}, value = “张三”;

Object.defineProperty(user, “name”,{

enumerabel: true,

configurabel: true,

get: function(){

return value;

},

set:function(newValue){

value = newValue;

}

})

console.log(user.name) //张三

user.name = “李四”;

console.log(user.name); //李四

console.log(value); //李四

### 数组

Var array = new Array();

Array.splice(起始下标，替换/删除数量)

#### 增删查改

##### 删除元素

arr.splice(startIndex, num)

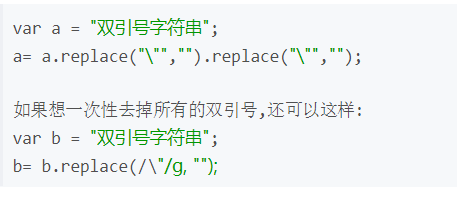
例子：a = [1,2,3,4,5]

a.splice(0,2) 🡪 [1,2]

a = [3,4,5]

##### 修改、删除元素

**Replace()**



**push()和unshift()**

向数组的 尾部/头部 **添加若干元素**，并返回 数组的 **新长度**；【arr.push(12,21)】

**pop()和shift()**

从数组的 尾部/头部 **删除1个元素(删且只删除1个)**，并返回 **被删除的元素**；空数组是继续删除，不报错，但返回undefined；【arr.pop()】

##### 添加属性

var arr = [];

无对象存储空间的情况：

arr.push({“name”: “123”}) -- 确保数组有对象存储空间，

如此时：

a[{name:”123”}]

否则会出现a[0] undefind导致出错

有对象存储空间的情况：

var arr = [{"name”: 123}]

arr[0].age = “bbb” -- {name: 123, age: "bbb"}

##### 删除属性

a[0] = {“one”: 123, “two”: 234, “three”：666 }

delete arr[0][“three”]

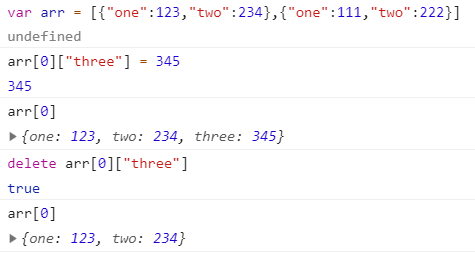
##### 修改属性值

arr[0][“one”] = 555

##### 查询元素

var b = [1,2,3,4,5,6]

b.indexOf(2) -- 1



数组深度拷贝：

var arr = [1,2,3]

var list = JSON.parse(JSON.stringify(arr))

#### 合并数组

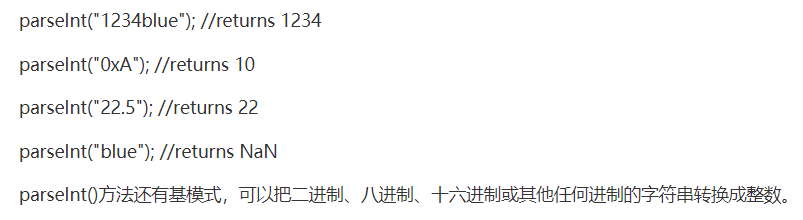
##### concat()

let newArr = array1.concat(array2)

### 字符

#### 字符串转化数字

parseInt():



### 对象

#### 判断对象类型

Object.prototype.toString.call(value)

=== '[object Object]' –对象

=== '[object Array]' –数组

=== '[object String]' –字符

=== '[object Boolean]' –布尔值

#### key

1. 获取key【

Object.keys(value) -- return arr

】

#### value

1. 获取value【

Object.values(value) -- return arr

】

扩展：

Object. entries(value) --对象转化为键值对形式用数组存储

Object.entries(obj).forEach(([*key*, *value*]) => {

console.log(`${key} ${value}`); *// "a 5", "b 7", "c 9"*

});

Object. fromEntries(value) --数组键值对转化为对象形式

const arr = [ ['0', 'a'], ['1', 'b'], ['2', 'c'] ];

const obj = Object.fromEntries(arr);

console.log(obj); *// { 0: "a", 1: "b", 2: "c" }*

### JSON

JSON.stringify() --对象转json

JSON.parse() --json转对象

### uncode编码

escape()、encodeURI ()、encodeURIComponent() --编码

unescape()、decodeURI ()、decodeURIComponent() --解码

## 页面

js 跳转页面：

当前页面刷新为链接页面：【a -> href=””】

Window.location.href = “https:www.baidu.com”

新开页面跳转到链接页面：【a -> href=”” target=” view\_frame”】

Window.open(“https:www.baidu.com”)

离开页面：

onunload --用户退出页面

onbeforeunload --即将离开页面或刷新时启动

window.onunload/onbeforeunload = function(){…}

### 获取链接地址

window.location.href --获取完整的url字符串

如：<https://blog.csdn.net/shenjun1992722/article/details/50076277>

window.location.protocol --获取url的协议部分

如：https:

window.location.host --获取url的主机部分

如：blog.csdn.net

window.location.port --获取url的端口部分



window.location.pathname --获取url的路径部分（文件地址）

如：shenjun1992722/article/details/50076277

/fisker/post/0703/window.location.html

window.location.search //查询（参数）部分

如：?ver=1.0&id=6

window.location.hash --获取瞄点



### 获取url参数

//获取url中参数 存储为：username:123形式

function GetRequest() {

var url = decodeURIComponent(location.search); //获取url中"?"符后的字串

var theRequest = new Object();

if (url.indexOf("?") != -1) {

var str = url.substr(1);

strs = str.split("&");

for (var i = 0; i < strs.length; i++) {

theRequest[strs[i].split("=")[0]] = unescape(strs[i].split("=")[1]);

}

}

return theRequest;

}

【编码：encodeURIComponent () – 将内容转化为unicode

如：

http%3A%2F%2Fwww.baidu.com%2F%E6%98%A5%E8%8A%82

http://www.baidu.com/春节

解码：decodeURIComponent() 】

【编码：encodeURI 解码：decodeURI】

【编码：escape 解码： unescape 不推荐】

### Http与Https

https：http + ssl+tcp

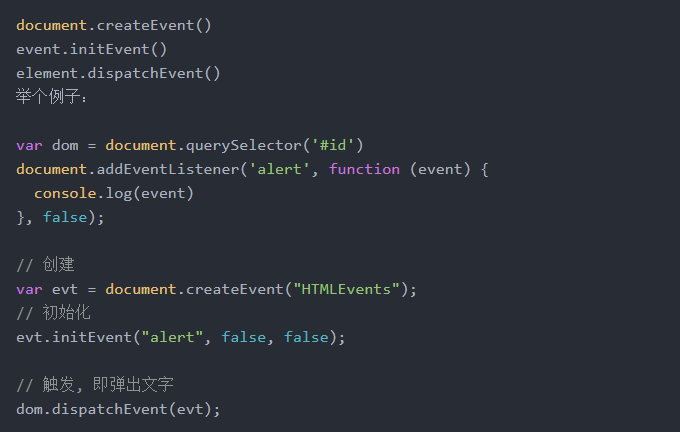
## 方法

### sort()



### dispatchEvent(eventObject)

监听事件对象，使用前应创建和初始化：









### 页面状态（window）

#### visibilityState

查看页面状态：正在浏览(visible)、正查看其他窗口页面|最小化等(hidden)

document.visibilityState -- “visible” | “hidden”

类似方法：

页面是否被隐藏

document.hidden -- true | false

#### readyState

查看页面加载情况：

1、uninitialized --还未开始载入

2、loading --载入中

3、interactive --已加载，文档与用户可以开始交互

4、complete --载入完成

document.readyState “complete”

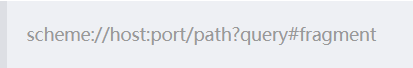
页面刷新：

window.location.href = “url” --改变url地址

window.location.replace(“url”) --将地址替换成新的url，通过指定url替换当前缓存在历史里的项目，导致不能通过“前进”、“后退”来访问已被替换的url

window.location.reload() --强制刷新页面，从服务器重新请求

完整url组成：





#### history(历史记录)

window.history.back(n？) --返回上一步操作，如返回点击之前状态，不重置页面

window.history.length --返回记录的操作数

window.history.go(n) --前进/后退 n步操作且重置页面（表单数据丢失）

window.location.forward --前进，与back相反

window.location.reload() --刷新

window.location.replace(url) --以当前history栈替换url地址

window.document. referrer --获取上一页面路径

window.location.href = url --跳转到url地址 或 获取当前url

history.pushStatus(state, title, url)

【state： 与指定网址相关的状态对象，popstate事件触发时，该对象传入回调函数】

【title： 新页面标题，但浏览器几乎忽略，可为null】

【url： 新网址，必须与当前页面处在同一域】

例：window.history.pushState(null, null, ‘articleList?page=123’)

history.replaceState()与history.pushState()使用方式一致，但history.pushState()是在history栈中添加一新条目，history.replaceState()是修改当前条目数据为replaceState内容

使用方式：获取 – history.state

### 滚动条

**onscroll**：元素滚动时执行

<div onscroll="test()">

1. 获取页面总高度（总滚动条高度）

document.body. **scrollHeight**

1. 获取当前滚动条位置

window.scrollY == window.**pageYOffset**  //true

1. 可视视图高度

document.documentElement.clientHeight || document.body.clientHeight;

1. 滚动条距上部的距离

document.documentElement.scrollTop || document.body.scrollTop

1. 设置滚动条位置

window.scrollTo(x, y) --直接跳到指定位置，类似锚点跳转效果

window.scrollTo({top: x, left: y, behavior: (‘smooth’)})

### 滚轮

除firl fox中使用DOMMouseScroll添加滚轮，其他浏览器均可使用onmousewheel添加滚轮：

Chrome:

window. onmousewheel = document. onmousewheel = function(e){console.log(e.wheelDelta)}

Firefox:

document.addEventListener(‘DOMMouseScroll’,function(e){console.log(e.detail)},

false) --false 终止标准滚轮事件处理，防止它上下滑动网页

### 简单数组查询( select() )

*//只查询简单数组数值： [1,2,{name:'123',age:12}]*

      select(*array*, *select*){

        let result = [];

        if(array != undefined){

          Object.keys(array).forEach((*key*, *index*) =>{

*//是否为对象*

            if(Object.prototype.toString.call(array[key]) === '[object Object]'){

              if(Object.values(array[key]).indexOf(select) != -1){

                result.push(array[key]);

              }

            } *//是否为数组*

            else if(Object.prototype.toString.call(array[key]) === '[object Array]'){

              console.log('数组：' + select)

            }else{

              if(array[key] === select){

                result.push(array[key]);

              }

            }

          })

        }

        return result;

      },

### 图片预览

change(*src*){ *//src - input按钮*

var img = document.querySelector(src);

var file = document.querySelector("input").files;

img.src = URL.createObjectURL(file[0]);

}

## 数据传递

### 跨域

1. jsonp跨域

前台之间数据交流

A：动态生成script标签，script中含有一个获取数据的方法，访问B的js文件

B：script中有一个js方法向本域后台获取数据，并此方法名与A中方法一致

当A页面加载成功时，其中的function调用B的同名function,得到数据

参考资料：<https://blog.csdn.net/hansexploration/article/details/80314948>

### ajax

jquery:

$.ajax({

url: url,

data:{},

success(){},

error(){}

……

})

asnyc: false --同步执行（等待ajax执行完成后再执行后面程序）

success --ajax成功执行并获取返回值

error --ajax出错：

url 404、数据格式不对（dataType）、未开启服务器。。。

XMLHttpRequest.readyState: 状态码

0 － （未初始化）还没有调用send()方法

1 － （载入）已调用send()方法，正在发送请求

2 － （载入完成）send()方法执行完成，已经接收到全部响应内容

3 － （交互）正在解析响应内容

4 － （完成）响应内容解析完成，可以在客户端调用了

### 动画

1. setTimeout 定时器-（定时后执行一次）
2. setInterval 定时器-（根据设定时间一直执行,若执行时间不足，可能会产生掉帧现象）
3. requestAnimationframe 帧定时器-（根据浏览器刷新频率执行一次，通常16.7ms，关闭页面或tab页面会自动停止，节约性能）

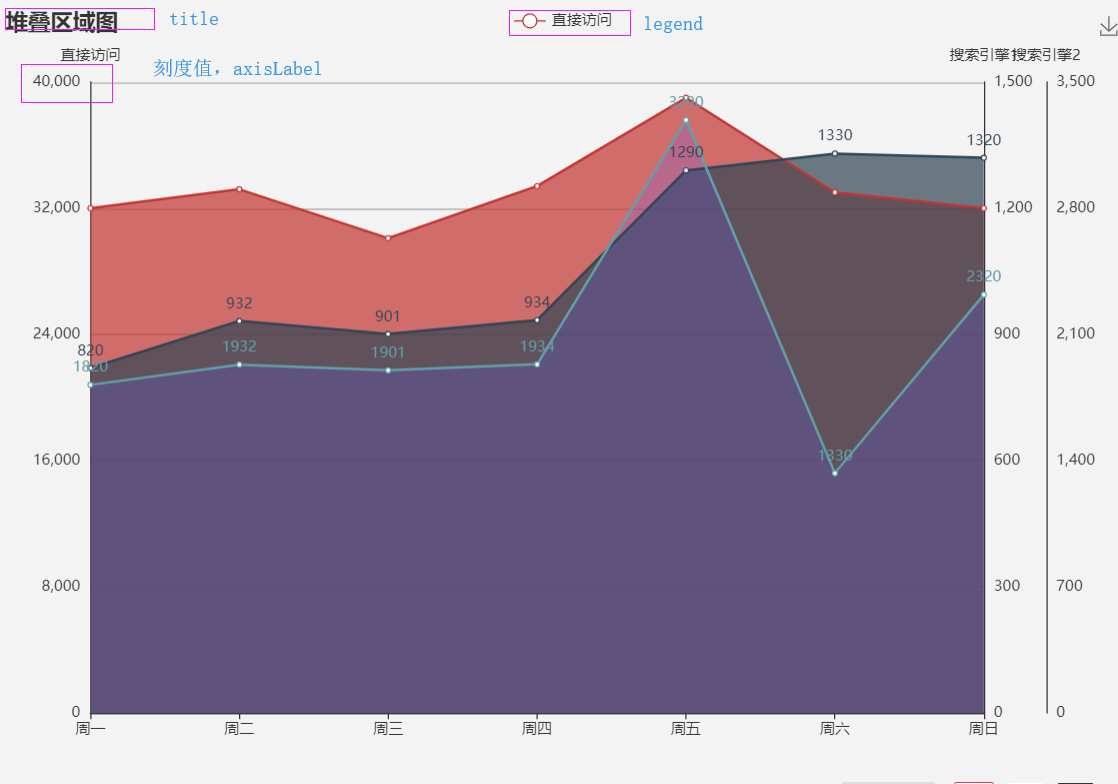
## eCharts（图表）

图表随屏幕宽度改变而改变：

单图表：window.onresize = myChart.resize()

多图表：window. addEventListener(“resize”,()=>{myChart.resize()})

常用属性：



1. title 标题组件
2. legend 图例
3. series 图表数据样式..

：hoverOffset – 悬浮时扩展的距离

：startAngle – 初始旋转角度，初始一般为12点方向

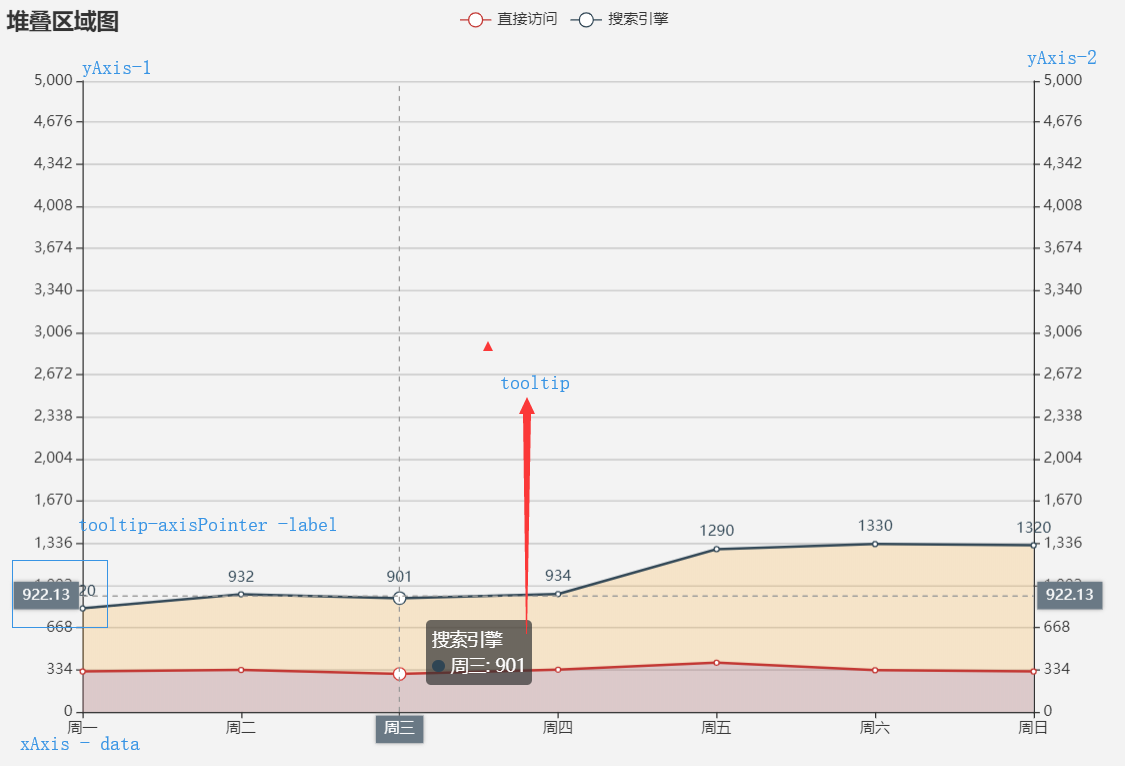
：silent – 是否不响应鼠标事件-悬浮、点击..

：stack: string 多轴时此值相同导致图表数值与刻度校对不准

：yAxisIndex: number – 多y轴时设置后才能显示刻度等，默认0,

让此series对应轴线

4、grid 图表距离外部的距离 - containLabel: true(自动收缩)



5、tooltip 提示框

trigger: ‘item’:数据项图形触发，散点图、饼图等无类目图表

trigger: ‘axis’:坐标轴触发，柱状图、折线图等会使用类目轴图表

axisPointer: 坐标轴指示器

鼠标辅助标线：axisPointer – type{‘line’:纵标线，

’cross’:十字标线，shadow:阴影区块}

颜色渐变：



yAxis：

axisLabel: {show:true}, - 显示y轴刻度数值及相关设置

splitLine: {show:false， lineStyle:{}}, - 显示y轴对应的数值线 – 分割线样式

axisLine: {show:false}, - 不显示y轴

– y轴刻度文字白色

axisTick:{ show: false }, - 不显示刻度标线

scale: true – 取消0刻度，

position: ‘left/right’ – 位置

xAxis：

boundaryGap：false – 坐标轴两边不留白

折线图：

symbol:'none'/’circle’, //这句就是去掉点的

smooth:true, //这句就是让曲线变平滑的

symbolSize: 12, - 节点大小

min：0 – 最小值

max：1000 – 最大值

interval: Math.ceil(1000(最大值)/5) – 将区域平分5份

offset: 20 – 偏移量（平移值），

areaStyle:{normal:{}} -折叠区域

textStyle:{normal:{}} - 节点、连线

柱状图：

横向排列：

xAxis: ‘value’ yAxis: ‘category’

纵向排列：

xAxis: ‘category’ yAxis: ‘value’

折线图：

位置(确定圆心位置)：

series – center(‘内圆半径’，‘外圆半径’)

参考资料：<https://www.cnblogs.com/benmumu/p/8316652.html>

## canvas

**moveTo(x,y) : 把路径移动到画布中的指定位置**

**rotate() : 旋转画布**

**lineTo(x,y) : 添加一个新点**

**lineWidth : 线条宽度**

**setLineDash([]) : 线条虚线间隔**

**stroke() : 绘制线条，默认黑色**

**strokeStyle : 指定线条的颜色**

**fill() : 颜色填充**

**fillStyle : 指定填充的颜色 （同一canvas设置不同颜色时，需要每次设置前清除画笔记录，ctx.beginPath()）**

**translate(width,height) : 以上一中心为基点移动旋转点位置，**

**默认为(0,0)**

**beginPath() : 清空子路径开始一个新路径（清空之前绘制记录），**

**清除之前的设置，如颜色、宽度等，避免被后续设置覆**

**盖。**

**closePath() : 将笔点返回到当前子路径起始点的方法（相当于在起始点设置一个lineTo(),但是当线条宽度不同时，两者表现形式略有不同）**

### 绘制矩形



// 绘制空心矩形

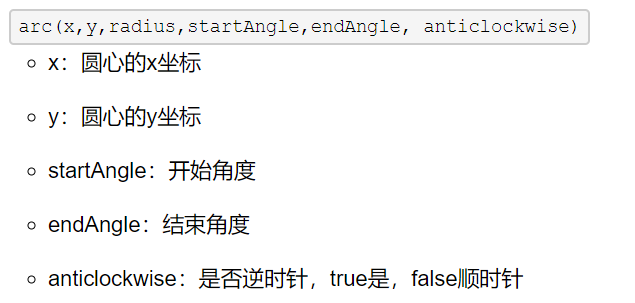
context.beginPath();

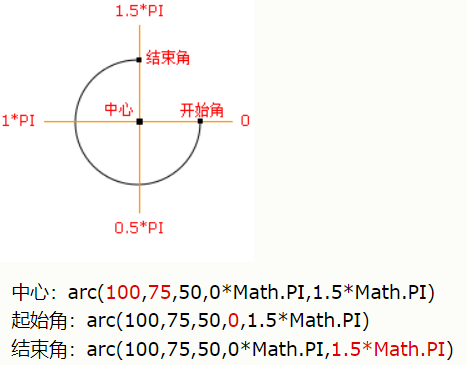
context.strokeStyle = "grey";

context.strokeRect(130, 150, 40, 50);

// context.rect(130, 150, 40, 50);

### 绘制圆形





（**绘制的图形**：由起始点沿圆弧至终点形成的封闭区域）

// 绘制红色圆形

context.beginPath();

context.fillStyle = "red";

context.arc(160, 180, 6, 0, 2 \* Math.PI);

context.fill();

### 线性渐变

**createLinearGradient(xStart,yStart,xEnd,yEnd)**

* xStart：渐变开始点x坐标
* yStart：渐变开始点y坐标
* xEnd：渐变结束点x坐标
* yEnd:渐变结束点y坐标
* addColorStop(‘offset’,’color’)
  + offset:设定的颜色离渐变结束点的偏移量(0.0~1.0),根据create中x,y长宽比例
  + color:绘制时要使用的颜色

### 径向渐变

**createRadialGradient(xStart,yStart,radiusStart,xEnd,yEnd,radiusEnd)**

* xStart：发散开始圆心x坐标
* yStart：发散开始圆心y坐标
* radiusStart：发散开始圆的半径
* xEnd：发散结束圆心的x坐标
* yEnd：发散结束圆心的y坐标
* radiusEnd：发散结束圆的半径

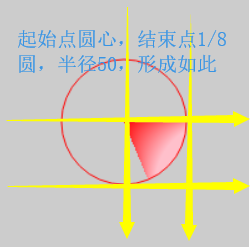
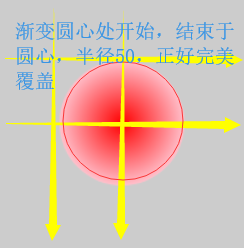
createRadialGradient(100, 100, 0, 175, 150, 50)

createRadialGradient(100, 100, 0, 100, 100, 50)

个人总结：**createRadialGradient中类似定义了两个圆，两个圆心位置为：**

**xStart,yStart，r**

**xEnd,yEnd， r，设置渐变后取两圆相交部分**



径向渐变参考资料：<https://www.bbsmax.com/A/MAzA6P3yJ9/>

### 阴影

**shadowOffsetX : 阴影在x方向上的偏移量，默认为0**

**shadowOffsetY : 阴影在y方向上的偏移量，默认为0**

**shadowColor : 阴影颜色，默认为#000000**

**shadowBlur : 阴影模糊度，默认为0**

**阴影出现的条件：offsetx、y和blur三者不全为0**

**设置后此canvas内都默认为此阴影效果，可单独修改**

ctx.fillStyle = "#393550";

ctx.fillRect(0, 0, 300, 300);

ctx.strokeStyle = "#41339c";

ctx.shadowColor = '#81f2f4';

ctx.shadowBlur = 20;

ctx.strokeRect(10, 10, 280, 280);

### 绘制图像

**drawImage(img, x, y) : 在画布上定位图像**

**drawImage(img, x, y,width,height) : 在画布上定位图像，并规定图像高度**

**drawImage(img,sx,sy,swidth,sheight,x,y,width,height) : 剪切图像，并在画布上**

**定位被剪切的部分**

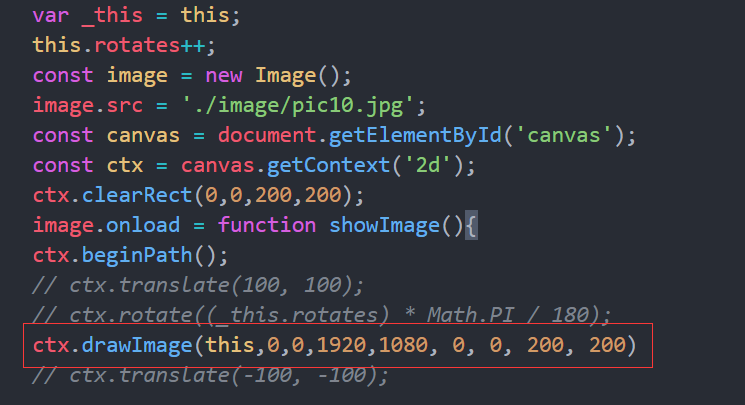
* img：规定要使用的图像、画布或视频
* sx：可选。开始剪切的图片上的 x 坐标位置
* sy：可选。开始剪切的图片上的 y 坐标位置
* swidth：可选。被剪切图像的宽度
* sheight：可选。被剪切图像的高度
* x：在画布上放置图像的 x 坐标位置
* y：在画布上放置图像的 y 坐标位置
* width：可选。要使用的图像的宽度
* height：可选。要使用的图像的高度

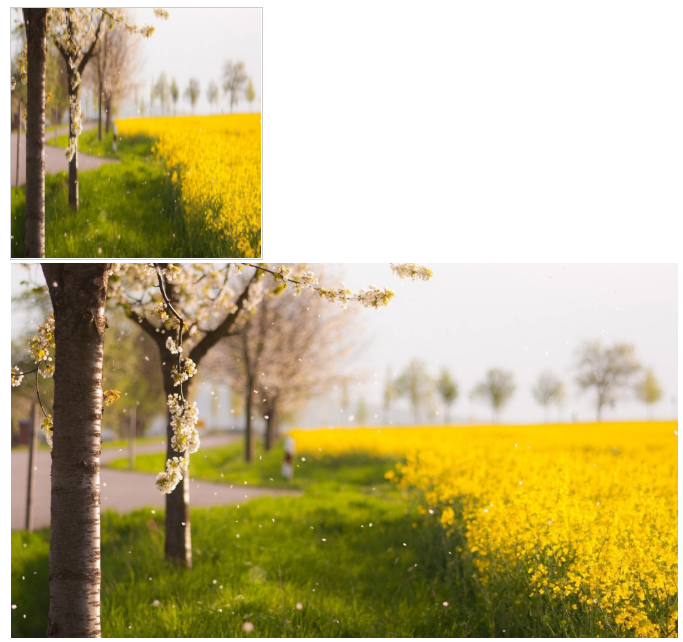
截图功能：sx、sy（截图图片的起始位置，左上角）

swidth、sheight（截图图片的宽高，此截图的宽高为实际图片的真实大小）

x、y（截图结果图片展示的左上角位置）

width、height（截图结果图片的宽高）





### 图形变形

scale(x,y)：缩放

* + x ：x坐标轴按 x 比例缩放
  + y ：x坐标轴按 y 比例缩放

translate(x,y)：平移

* + x ：坐标原点向x轴方向平移
  + y ：坐标原点向y轴方向平移

rotate(angle)：旋转

* + angle ：坐标轴旋转x角度（角度变化模型和画圆的模型一样）

var img = new Image();

img.src = "./images/2.png";

img.onload = function () {

ctx.beginPath(); ctx.scale(0.5, 0.5);

ctx.translate(200, 150);

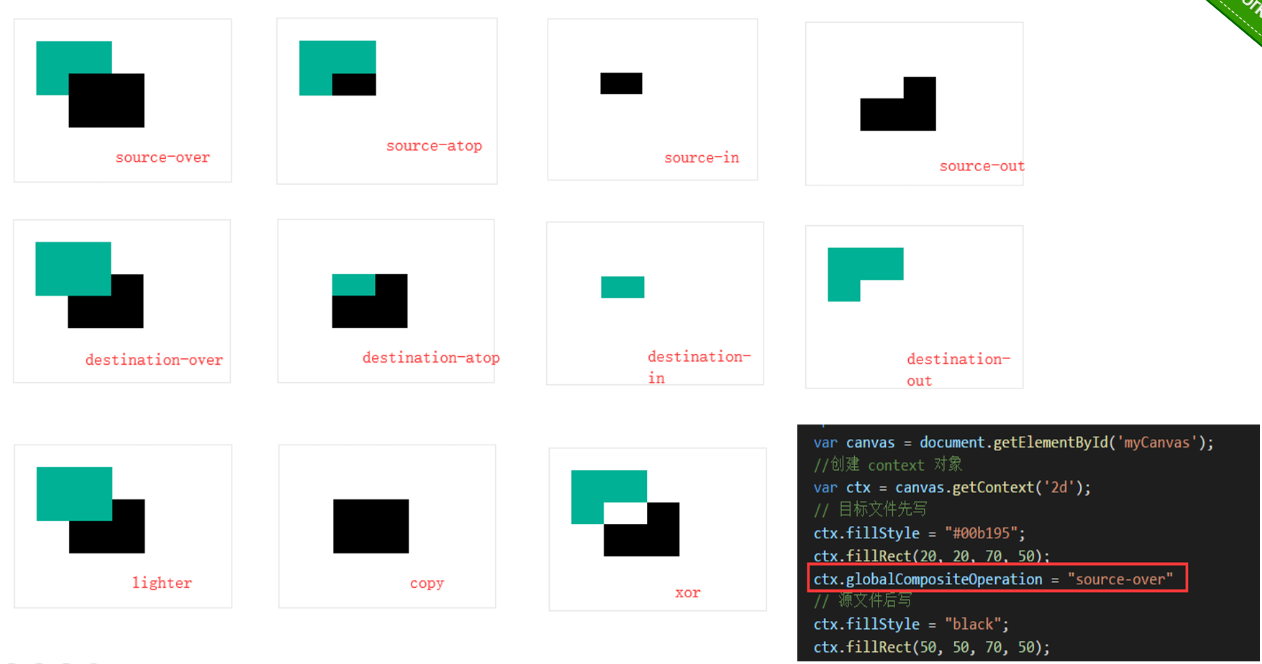
ctx.rotate(30 \* Math.PI / 180);

ctx.drawImage(this, 10, 10, 280, 280)

}

### 图形组合

**globalCompositeOperation : 设置或返回如何将一个源（新的）图像绘制到目标（已有）的图像上**



### 图形平铺

createPattern（img, type）: 在指定的方向内重复指定的元素（repeat）

* img：图片、视频, 或者其他
* type：repeat：默认。该模式在水平和垂直方向重复
* repeat-x：该模式只在水平方向重复
* repeat-y：该模式只在垂直方向重复
* no-repeat：该模式只显示一次（不重复）

// 获取页面上已有的图片

var img = document.getElementById('img');

var g1 = ctx.createPattern(img, "repeat-x");

ctx.fillStyle = g1;

ctx.rect(0, 0, 275, 200);

ctx.fill()

### 图像剪切

**clip（）：从原始画布中剪切任意形状和尺寸，一旦剪切了某个区域，则所有之后的绘图都会被限制在剪切的区域内（类似覆盖 + overflow:hidden）**

ctx.arc(100, 100, 50, 0, 2 \* Math.PI);

ctx.fillStyle = "#00b195";

ctx.fill(); ctx.clip();

ctx.fillStyle = "black";

ctx.fillRect(100, 100, 50, 50);



**在canvas中可以通过 save() 方法保存裁切区之前的状态，完成裁切后再使用 restore() 方法进行状态读取。**

ctx.arc(100, 100, 50, 0, 2 \* Math.PI);

ctx.fillStyle = "#00b195";

ctx.fill();

// 保存裁切前的状态

ctx.save();

ctx.clip();

ctx.fillStyle = "black";

ctx.fillRect(100, 100, 50, 50);

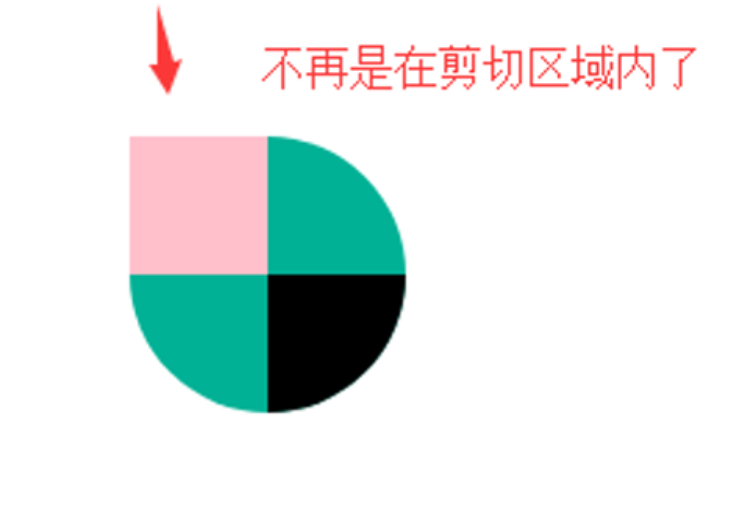
// 释放裁切前的状态

ctx.restore();

ctx.fillStyle = "pink";

ctx.fillRect(50, 50, 50, 50);

//恢复的canvas相当于重新渲染，然后在第二层上绘制矩形



### 绘制文字

font：设置或返回文本字体属性，如font-style，font-weight，font-size等

textAlign：对齐方式设置，取值：start， end，left，right，center，默认值为start

textBaseline：文本基线设置，取值：top，middle，alphabetic，ideographic，

bottom，默认值：alphabetic

fillText(text,x,y,[maxWidth])：绘制实心文字

strokeText(text,x,y,[maxWidth])：绘制实心文字

* + text：显示文本
  + x：文本开始的x坐标
  + y：文本开始的y坐标
  + maxWidth：可选，文本显示的最大宽度

ctx.font = "40px 隶书";

ctx.strokeStyle = "#00b195";

ctx.strokeText('黑玛鱼', 250, 40);

// 加了maxWidth

ctx.strokeText('黑玛鱼', 250, 100, 240);

// 加了textAlign

ctx.textAlign = "right";

ctx.fillStyle = "#409eff";

ctx.fillText('黑玛鱼', 250, 150, 240);

// 加了textBaseline

ctx.textAlign = "left";

ctx.textBaseline = "middle";

ctx.fillText('黑玛鱼', 250, 150, 240);

### 塞尔曲线

**二次贝塞尔曲线：**

**quadraticCurveTo(cpx,cpy,x,y) - cpx，cpy表示控制点的坐标, x，y表示终点坐标**

//绘制起始点、控制点、终点

context.beginPath();

context.moveTo(20,170);

context.lineTo(130,40);

context.lineTo(180,150);

context.stroke();

//绘制2次贝塞尔曲线

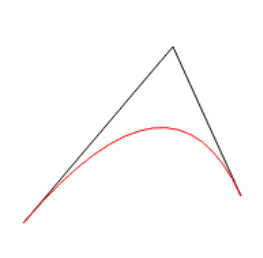
context.beginPath();

context.moveTo(20,170);

context.**quadraticCurveTo**(130,40,180,150);

context.strokeStyle = "red";

context.stroke();



**三次贝塞尔曲线：**

**bezierCurveTo(cp1x,cp1y,cp2x,cp2y,x,y)**

**cp1x，cp1y表示第一个控制点的坐标,**

**cp2x，cp2y表示第二个控制点的坐标,**

**x，y表示终点的坐标；**

//绘制起始点、控制点、终点

context.beginPath();

context.moveTo(25,175);

context.lineTo(60,80);

context.lineTo(150,30);

context.lineTo(170,150);

context.stroke();

//绘制3次贝塞尔曲线

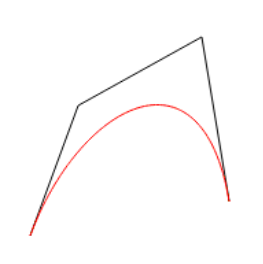
context.beginPath();

context.moveTo(25,175);

context.**bezierCurveTo**(60,80,150,30,170,150);

context.strokeStyle = "red";

context.stroke();



#### 画笔绘制 – 直线

let canvas = this.$refs.canvas; //找目标元素canvas

let context = canvas.getContext('2d');//选择二元绘制

context.beginPath();//清空笔迹（以前的路径）

context.moveTo(20,100);//设置下笔位置

context.lineTo(50,150);//从上一点直线移动到此点位置

context.lineTo(100,120);

context.stroke(); //绘制线条，否则线条因没绘制不显示

参考资料：<https://www.cnblogs.com/webhmy/p/9556121.html>