GET与POST

- GET在浏览器回退时是无害的，而POST会再次提交请求。
- GET产生的URL地址可以被Bookmark，而POST不可以。
- GET请求会被浏览器主动cache，而POST不会，除非手动设置。
- GET请求只能进行url编码，而POST支持多种编码方式。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
- GET请求在URL中传送的参数是有长度限制的，而POST么有。
- 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
- GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
- GET参数通过URL传递，POST放在Request body中。

重大区别：GET产生一个TCP数据包；POST产生两个TCP数据包。
对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

遍历方法(不会改变自身)

基于ES6的方法一共有12个，分别为forEach、every、some、filter、map、reduce、reduceRight 以及ES6新增的方法entries、find、findIndex、keys、values。

forEach()　遍历数组，指定数组的每项元素都执行一次传入的函数，返回值为undefined。
every()　测试返回布尔值，只要其中有一个函数返回值为 false，那么该方法的结果为 false；如果全部返回 true，那么该方法的结果才为 true。
some()　测试返回布尔值，与every() 方法相反，只要有一个函数返回值为 true，则该方法返回 true，若全部返回 false，则该方法返回 false。
filter()　过滤返回新数组，并返回所有通过测试的元素组成的新数组。它就好比一个过滤器，筛掉不符合条件的元素。
map()　操作返回新数组，使用传入函数处理每个元素，并返回函数的返回值组成的新数组。
reduce()　累加，接收一个方法作为累加器，数组中的每个值(从左至右) 开始合并，最终为一个值。
find()　返回数组中第一个满足条件的元素（如果有的话），如果没有，则返回undefined。
findIndex()　它返回数组中第一个满足条件的元素的索引（如果有的话）否则返回-1。

```
array.map(function(value,index,arr), thisValue)
```

```
var array = [1, 3, 5, 7, 8, 9, 10];
function f(value, index, array){
  return value%2==0; // 返回偶数
}
function f2(value, index, array){
  return value > 20; // 返回大于20的数
}
console.log(array.find(f)); // 8
console.log(array.find(f2)); // undefined
console.log(array.findIndex(f)); // 4
console.log(array.findIndex(f2)); // -1
```

entries()　返回一个数组迭代器对象，该对象包含数组中每个索引的键值对。

```
var array = ["a", "b", "c"];
var iterator = array.entries();
console.log(iterator.next().value); // [0, "a"]
console.log(iterator.next().value); // [1, "b"]
console.log(iterator.next().value); // [2, "c"]
console.log(iterator.next().value); // undefined, 迭代器处于数组末尾时，再迭代就会返回
undefined
```

keys()　返回一个数组索引的迭代器。（浏览器实际实现可能会有调整）

```
var array = ["abc", "xyz"];
var iterator = array.keys();
console.log(iterator.next()); // Object {value: 0, done: false}
console.log(iterator.next()); // Object {value: 1, done: false}
console.log(iterator.next()); // Object {value: undefined, done: true}
```

Symbol.iterator 为每一个对象定义了默认的迭代器。该迭代器可以被 `for...of` 循环使用。

```
let obj = {
          0: 'a',
          1: 'b',
          2: 'c',
          length: 3,
          [Symbol.iterator]: function () {
              // index用来记遍历圈数
              let index = 0;
              let next = () => {
                  return {
                      value: this[index],
                      done: this.length == ++index
                  }
              }
              return {
                  next
              }
          }
      }

      // console.log(obj.length)
      console.log([...obj]);//(2) ["a", "b"]

      for(let p of obj){
          console.log(p) //"a"  "b"
      }
```

```
//ios端，使用input时，弹出键盘后，整个页面会被键盘顶上去
$("input").blur(function() {
            window.scrollTo(0, 0);//重点  =======当键盘收起的时候让页面回到原始位置(这
里的top可以根据你们个                                    //人的需求改变，并不一定要回到页
面顶部)
});
```

h5与app交互

```
  if (ua.match(/iPhone|iPod|iPad/i) != null) {
    try {
      window.webkit.messageHandlers.showPicDialog.postMessage(paramStr);
    } catch (e) {
      Toast(utils.toastconfig(e));
    }
  } else if (ua.match(/Android/i) != null) {
    try {
      android.showPicDialog(paramStr);
    } catch (e) {
      Toast(utils.toastconfig(e));
    }
  }
```

```
            get: function (index) {
                var gettedDoms = [];
                if (arguments.length > 0) {
                    return this[index];
                }

                this.each(function () {
                    gettedDoms.push(this);
                });
                return gettedDoms;
            }
        });

        $.ajax = function (config) {
            var url = config.url;
            var type = config.type || 'POST';
            var headers = config.headers || [];
            var contentType = config.contentType ||
'application/json;charset=utf-8';
            var data = config.data;
            var dataType = config.dataType || 'json';
            var fnError = config.error;
            var fnSuccess = config.success;
            var fnComplete = config.complete;
            var async = config.async !== false;
            var xmlhttp;

            if (window.XMLHttpRequest) {
                xmlhttp = new XMLHttpRequest();
            }
            else {
                xmlhttp = new ActiveXObject('Microsoft.XMLHTTP');
            }

            xmlhttp.onreadystatechange = function () {
                if (xmlhttp.readyState === 4) {
                    if (xmlhttp.status === 200) {
                        var rspData = xmlhttp.responseText ||
xmlhttp.responseXML;
                        if (dataType === 'json') {
```

```
                            rspData = eval('(' + rspData + ')');
                        }

                        callFunction(fnSuccess, [rspData, xmlhttp.statusText,
xmlhttp]);
                    }
                    else {
                        callFunction(fnError, [xmlhttp, xmlhttp.statusText]);
                    }
                    if (fnComplete) {
                        callFunction(fnComplete, [xmlhttp, xmlhttp.statusText]);
                    }
                }

            };
            xmlhttp.open(type, url, async);
            if (async !== false) {
                xmlhttp.timeout = config.timeout || 0;
            }

            for (var i = 0; i < headers.length; ++i) {
                xmlhttp.setRequestHeader(headers[i].name, headers[i].value);
            }
            xmlhttp.setRequestHeader('Content-Type', contentType);
            if (typeof data === 'object' && contentType === 'application/x-www-
form-urlencoded') {
                var s = '&';
                for (var attr in data) {
                    if (data.hasOwnProperty(attr)) {
                        s += ('&' + attr + '=' + data[attr]);
                    }

                }
                s = s.substring(1);
                xmlhttp.send(s);
            }
            else {
                xmlhttp.send(data);
            }
        };
```

hash和history

hash
原理：`onhashchange`事件,可以在window对象上监听这个事件

```
//通过改变hash来改变页面字体颜色
window.onhashchange = function(event){
     console.log(event.oldURL, event.newURL);
     let hash = location.hash.slice(1);
     document.body.style.color = hash;
}
```

因为hash发生变化的url都会被浏览器记录下来，从而你会发现浏览器的前进后退都可以用了，同时点击后退时，页面字体颜色也会发生变化。这样一来，尽管浏览器没有请求服务器，但是页面状态和url一一关联起来，后来人们给它起了一个霸气的名字叫前端路由，成为了单页应用标配。

history
前面的hashchange，你只能改变#后面的url片段，而history api则给了前端完全的自由
history api可以分为两大部分，切换和修改:
切换历史状态包括back、forward、go，对应浏览器的前进，后退，跳转操作

```
history.go(-2);//后退两次
history.go(2);//前进两次
history.back(); //后退
hsitory.forward(); //前进
```

修改历史状态包括了`pushState`,`replaceState`,接收三个参数:`stateObj`,`title`,`url`

```
//通过pushstate把页面的状态保存在state对象中，当页面的url再变回这个url时，可以通过
event.state取到这个state对象，从而可以对页面状态进行还原，这里的页面状态就是页面字体颜色，其
实滚动条的位置，阅读进度，组件的开关的这些页面状态都可以存储到state的里面
history.pushState({color:'red'}, 'red', 'red')
history.back();
setTimeout(function(){
    history.forward();
 },0)
window.onpopstate = function(event){
     console.log(event.state)
     if(event.state && event.state.color === 'red'){
           document.body.style.color = 'red';
      }
}
```

history模式的问题

通过history api，我们丢掉了丑陋的#，但是它也有个问题：不怕前进，不怕后退，就怕**刷新**，**f5**，（如果后端没有准备的话）,因为刷新是实实在在地去请求服务器的。
在hash模式下，前端路由修改的是#中的信息，而浏览器请求时是不带它玩的，所以没有问题.但是在history下，你可以自由的修改path，当刷新时，如果服务器中没有相应的响应或者资源，会分分钟刷出一个404来。

```
// 用一个函数域包起来
// 在这里边 var 定义的变量，属于这个函数域内的局部变量，避免污染全局
// 外部变量通过函数参数引入进来
(function(window, undefined) {
  // jQuery 代码
})(window);
```

jQuery 无 new 构造

```
//实例化一个 jQuery 对象的方法
// 无 new 构造
$('#test').text('Test');

// 当然也可以使用 new
var test = new $('#test');
test.text('Test');
```

```
(function(window, undefined) {
    var
    // ...
    jQuery = function(selector, context) {
        // The jQuery object is actually just the init constructor 'enhanced'
        // 看这里，实例化方法 jQuery() 实际上是调用了其拓展的原型方法 jQuery.fn.init
        return new jQuery.fn.init(selector, context, rootjQuery);
    },

    // jQuery.prototype 即是 jQuery 的原型，挂载在上面的方法，即可让所有生成的 jQuery 对象使用
    jQuery.fn = jQuery.prototype = {
        // 实例化化方法，这个方法可以称作 jQuery 对象构造器
        init: function(selector, context, rootjQuery) {
            // ...
        }
    }
    // 这一句很关键
    // jQuery 没有使用 new 运算符将 jQuery 实例化，而是直接调用其函数
    // 要实现这样,那么 jQuery 就要看成一个类，且返回一个正确的实例
    // 且实例还要能正确访问 jQuery 类原型上的属性与方法
    // jQuery 的方式是通过原型传递解决问题，把 jQuery 的原型传递给 jQuery.prototype.init.prototype
    // 所以通过这个方法生成的实例 this 所指向的仍然是 jQuery.fn，所以能正确访问 jQuery 类原型上的属性与方法
    jQuery.fn.init.prototype = jQuery.fn;

})(window);
```

jQuery.fn.init.prototype = jQuery.fn ：

1）首先要明确，使用 $('xxx') 这种实例化方式，其内部调用的是 return new jQuery.fn.init(selector, context, rootjQuery) 这一句话，也就是构造实例是交给了 jQuery.fn.init() 方法去完成。

2）将 jQuery.fn.init 的 prototype 属性设置为 jQuery.fn，那么使用 new jQuery.fn.init() 生成的对象的原型对象就是 jQuery.fn ，所以挂载到 jQuery.fn 上面的函数就相当于挂载到 jQuery.fn.init() 生成的 jQuery 对象上，所有使用 new jQuery.fn.init() 生成的对象也能够访问到 jQuery.fn 上的所有原型方法。

3）也就是实例化方法存在这么一个关系链

- jQuery.fn.init.prototype = jQuery.fn = jQuery.prototype ;
- new jQuery.fn.init() 相当于 new jQuery() ;
- jQuery() 返回的是 new jQuery.fn.init()，而 var obj = new jQuery()，所以这 2 者是相当的，所以我们可以无 new 实例化 jQuery 对象。

jQuery.fn.extend 与 jQuery.extend

1）**jQuery.extend(object)** 为扩展 jQuery 类本身，为类添加新的静态方法；

2）**jQuery.fn.extend(object)** 给 jQuery 对象添加实例方法，也就是通过这个 extend 添加的新方法，实例化的 jQuery 对象都能使用，因为它是挂载在 jQuery.fn 上的方法（上文有提到，jQuery.fn = jQuery.prototype ）。

它们的官方解释是：

1）jQuery.extend(): 把两个或者更多的对象合并到第一个当中，

2）jQuery.fn.extend()：把对象挂载到 jQuery 的 prototype 属性，来扩展一个新的 jQuery 实例方法。

也就是说，使用 jQuery.extend() 拓展的静态方法，我们可以直接使用 $.xxx 进行调用（xxx是拓展的方法名），

而使用 jQuery.fn.extend() 拓展的实例方法，需要使用 $().xxx 调用。

jQuery 的链式调用及回溯

链式调用，这一点的实现其实很简单，只需要在要实现链式调用的方法的返回结果里，返回 this ，就能够实现链式调用了。
当然，除了链式调用，jQuery 甚至还允许回溯

```
// 通过 end() 方法终止在当前链的最新过滤操作，返回上一个对象集合
$('div').eq(0).show().end().eq(1).hide();
```

小程序的生命周期函数

- onLoad() 页面加载时触发，只会调用一次，可获取当前页面路径中的参数。
- onShow() 页面显示/切入前台时触发，一般用来发送数据请求；
- onReady() 页面初次渲染完成时触发, 只会调用一次，代表页面已可和视图层进行交互。
- onHide() 页面隐藏/切入后台时触发, 如底部 tab 切换到其他页面或小程序切入后台等。
- onUnload() 页面卸载时触发，如redirectTo或navigateBack到其他页面时。

微信小程序原理

- 小程序本质就是一个单页面应用，所有的页面渲染和事件处理，都在一个页面内进行，但又可以通过微信客户端调用原生的各种接口；
- 它的架构，是数据驱动的架构模式，它的UI和数据是分离的，所有的页面更新，都需要通过对数据的更改来实现；
- 它从技术讲和现有的前端开发差不多，采用JavaScript、WXML、WXSS三种技术进行开发；
- 功能可分为webview和appService两个部分；
- webview用来展现UI，appService有来处理业务逻辑、数据及接口调用；
- 两个部分在两个进程中运行，通过系统层JSBridge实现通信，实现UI的渲染、事件的处理等。

wx.switchTab(Object object)　　跳转到 tabBar 页面，并关闭其他所有非 tabBar 页面
wx.reLaunch(Object object)　　关闭所有页面，打开到应用内的某个页面
wx.redirectTo(Object object)　　关闭当前页面，跳转到应用内的某个页面。但是不允许跳转到 tabbar 页面。wx.navigateTo(Object object)　　保留当前页面，跳转到应用内的某个页面。但是不能跳到 tabbar 页面。使用 wx.navigateBack 可以返回到原页面。小程序中页面栈最多十层。
wx.navigateBack(Object object)　　关闭当前页面，返回上一页面或多级页面。可通过 getCurrentPages 获取当前的页面栈，决定需要返回几层。

排序

```
//冒泡排序
```

```
//依次比较相邻的两个元素，如果后一个小于前一个，则交换，这样从头到尾一次，就将最大的放到了末尾。
function bubbleSort(arr) {
  var len = arr.length;
  for (var i = 0; i < len - 1; i++) {
    for (var j = 0; j < len - 1 - i; j++) {
      if (arr[j] > arr[j+1]) {      // 相邻元素两两对比
        var temp = arr[j+1];      // 元素交换
        arr[j+1] = arr[j];
        arr[j] = temp;
      }
    }
  }
  return arr;
}
```

```
//选择排序
//每次都找一个最大或者最小的排在开始即可。
//首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置
//再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
//重复第二步，直到所有元素均排序完毕。
function selectionSort(arr) {
  var len = arr.length;
  var minIndex, temp;
  for (var i = 0; i < len - 1; i++) {
    minIndex = i;
    for (var j = i + 1; j < len; j++) {
      if (arr[j] < arr[minIndex]) {    // 寻找最小的数
        minIndex = j;          // 将最小数的索引保存
      }
    }
    temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
  return arr;
}
```

```
//插入排序
//双层循环，外循环控制未排序的元素，内循环控制已排序的元素，将未排序元素设为标杆，与已排序的元素进行比较，小于则交换位置，大于则位置不动
//[5] 6 3 1 8 7 2 4  //第一个元素被认为已经被排序
//[5,6]  3 1 8 7 2 4 //6与5比较，放在5的右边
//[3，5，6]  1 8 7 2 4 //3与6和5比较，都小，则放入数组头部
function insertSort(arr){
    var tmp;
    for(var i=1;i<arr.length;i++){
        tmp  = arr[i];
        for(var j=i;j>=0;j--){
            if(arr[j-1]>tmp){
                arr[j]=arr[j-1];
            }else{
                arr[j]=tmp;
                break;
            }
        }
    }
```

```
    returan arr
}
```

```
//快速排序
//在数据集之中，选择一个元素作为”基准”（pivot）。
//所有小于”基准”的元素，都移到”基准”的左边；所有大于”基准”的元素，都移到”基准”的右边。这个操作称为分区 //(partition)操作，分区操作结束后，基准元素所处的位置就是最终排序后它的位置。
//对”基准”左边和右边的两个子集，不断重复第一步和第二步，直到所有子集只剩下一个元素为止。
function quickSort(arr){
    if(arr.length<=1) return arr;
    var partitionIndex=Math.floor(arr.length/2);
    var tmp=arr[partitionIndex];
    var left=[];
    var right=[];
    for(var i=0;i<arr.length;i++){
        if(arr[i]<tmp){
            left.push(arr[i])
        }else{
            right.push(arr[i])
        }
    }
    return quickSort(left).concat([tmp],quickSort(right))
}
```

```
//归并排序
//将数组一直等分，然后合并
//5 6 3 1 8 7 2 4
//[5,6] [3,1] [8,7] [2,4]
//[5,6] [1,3] [7,8] [2,4]
//[5,6,1,3] [7,8,2,4]
//[1,3,5,6] [2,4,7,8]
//[1,2,3,4,5,6,7,8]
function merge(left,right){
    var tmp=[];
    while(left.length && right.length){
        if(left[0]<right[0]){
            tmp.push(left.shift());
        }else{
            tmp.push(right.shift());
        }
    }
    return tmp.concat(left,right)
}
function mergeSort(arr){
    if(arr.length==1) return arr;
    var mid=Math.floor(a.length/2),
        left=arr.slice(0,mid);
        right=arr.slice(mid);
    return merge(mergeSort(left),mergeSort(right))
}
```

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

Date 对象方法

方法	描述
Date()	返回当日的日期和时间。
getDate()	从 Date 对象返回一个月中的某一天 (1 ~ 31)。
getDay()	从 Date 对象返回一周中的某一天 (0 ~ 6)。
getMonth()	从 Date 对象返回月份 (0 ~ 11)。
getFullYear()	从 Date 对象以四位数字返回年份。
getYear()	请使用 getFullYear() 方法代替。
getHours()	返回 Date 对象的小时 (0 ~ 23)。
getMinutes()	返回 Date 对象的分钟 (0 ~ 59)。
getSeconds()	返回 Date 对象的秒数 (0 ~ 59)。
getMilliseconds()	返回 Date 对象的毫秒(0 ~ 999)。
getTime()	返回 1970 年 1 月 1 日至今的毫秒数。
getTimezoneOffset()	返回本地时间与格林威治标准时间 (GMT) 的分钟差。
getUTCDate()	根据世界时从 Date 对象返回月中的一天 (1 ~ 31)。
getUTCDay()	根据世界时从 Date 对象返回周中的一天 (0 ~ 6)。
getUTCMonth()	根据世界时从 Date 对象返回月份 (0 ~ 11)。
getUTCFullYear()	根据世界时从 Date 对象返回四位数的年份。
getUTCHours()	根据世界时返回 Date 对象的小时 (0 ~ 23)。
getUTCMinutes()	根据世界时返回 Date 对象的分钟 (0 ~ 59)。
getUTCSeconds()	根据世界时返回 Date 对象的秒钟 (0 ~ 59)。
getUTCMilliseconds()	根据世界时返回 Date 对象的毫秒(0 ~ 999)。
parse()	返回1970年1月1日午夜到指定日期（字符串）的毫秒数。
setDate()	设置 Date 对象中月的某一天 (1 ~ 31)。
setMonth()	设置 Date 对象中月份 (0 ~ 11)。
setFullYear()	设置 Date 对象中的年份（四位数字）。
setYear()	请使用 setFullYear() 方法代替。
setHours()	设置 Date 对象中的小时 (0 ~ 23)。
setMinutes()	设置 Date 对象中的分钟 (0 ~ 59)。
setSeconds()	设置 Date 对象中的秒钟 (0 ~ 59)。
setMilliseconds()	设置 Date 对象中的毫秒 (0 ~ 999)。
setTime()	以毫秒设置 Date 对象。

方法	描述
setUTCDate()	根据世界时设置 Date 对象中月份的一天 (1 ~ 31)。
setUTCMonth()	根据世界时设置 Date 对象中的月份 (0 ~ 11)。
setUTCFullYear()	根据世界时设置 Date 对象中的年份（四位数字）。
setUTCHours()	根据世界时设置 Date 对象中的小时 (0 ~ 23)。
setUTCMinutes()	根据世界时设置 Date 对象中的分钟 (0 ~ 59)。
setUTCSeconds()	根据世界时设置 Date 对象中的秒钟 (0 ~ 59)。
setUTCMilliseconds()	根据世界时设置 Date 对象中的毫秒 (0 ~ 999)。
toSource()	返回该对象的源代码。
toString()	把 Date 对象转换为字符串。
toTimeString()	把 Date 对象的时间部分转换为字符串。
toDateString()	把 Date 对象的日期部分转换为字符串。
toGMTString()	请使用 toUTCString() 方法代替。
toUTCString()	根据世界时，把 Date 对象转换为字符串。
toLocaleString()	根据本地时间格式，把 Date 对象转换为字符串。
toLocaleTimeString()	根据本地时间格式，把 Date 对象的时间部分转换为字符串。
toLocaleDateString()	根据本地时间格式，把 Date 对象的日期部分转换为字符串。
UTC()	根据世界时返回 1970 年 1 月 1 日 到指定日期的毫秒数。
valueOf()	返回 Date 对象的原始值。

`tail -f log-file` 使用 `-f` 选项进行实时查看，这个命令执行后会等待，如果有新行添加到文件尾部，它会继续输出新的行，查看日志

- chmod 用于改变文件和目录的权限。
- 给指定文件的属主和属组所有权限(包括读、写、执行)：`chmod ug+rwx file.txt` 。
- 删除指定文件的属组的所有权限：`chmod g-rwx file.txt` 。
- 修改目录的权限，以及递归修改目录下面所有文件和子目录的权限：`chmod -R ug+rwx file.txt` 。