[同步和异步 3](#_Toc14848)

[日期 3](#_Toc24877)

[随机数 、数组的随机排序 4](#_Toc23454)

[JS面向对象——封装、继承（原型链继承、借用构造函数、组合继承）、多态 5](#_Toc20496)

[从输入URL到页面加载发生了什么（包含了浏览器渲染页面的流程） 9](#_Toc27740)

[Promise 13](#_Toc31679)

[浏览器内核的理解 15](#_Toc27949)

[三列布局（圣杯布局、双飞翼布局、使用绝对定位、分别左浮动和右浮动） 15](#_Toc5328)

[事件循环机制（event loop）——一些结论，涉及promise、setTimeout 18](#_Toc23059)

[URI 和 URL 区别 19](#_Toc1519)

[JS浅拷贝和深拷贝 19](#_Toc16092)

[浏览器缓存（HTTP缓存机制） 20](#_Toc4164)

[一、新鲜度（过期机制） 20](#_Toc21107)

[HTTP状态码、301和302区别、http长连接、无状态 22](#_Toc10210)

[Cookie 和 Session区别——登录网址后如何区别你是你？ 24](#_Toc22029)

[Cookie、localStorage和sessionStorage区别 25](#_Toc30936)

[arguments.callee和function.caller 26](#_Toc7808)

[作用域链与闭包 26](#_Toc412)

[排序 27](#_Toc8574)

[CSS reset 样式重置，清除浏览器默认样式；以及更好的Normalize.css，保留一些浏览器默认样式 29](#_Toc21984)

[跨域携带cookie的问题 31](#_Toc28053)

[设计模式在javascript中的应用 32](#_Toc29977)

###### 同步和异步

**两者区别：**

1.同步会阻塞代码执行，而异步不会。

2.比如alert是同步，setTimeout是异步。

**前端使用异步的场景：（这些场景的特点都是需要等待）**

1.定时任务：setTimeout,setInterval

2.网络请求：ajax请求，动态<img>加载

3.事件绑定

**例子**：

console.log(1);

console.log(setTimeout(function(){

console.log(2);

},1000));

console.log(3);

结果：

1

2850

3

2

解释：setTimeout执行后会立马返回一个数值id，但setTimeout中定义的函数会先暂存起来，继续执行后续代码，最后再执行暂存起来的函数。

###### 日期

Date.now(); //获取当前时间毫秒数(距 1970 年 1 月 1 日)

var dt = new Date(); // 如Wed Sep 06 2017 15:39:58 GMT+0800 (中国标准时间) 一般用不到

dt.getTime(); //获取毫秒数

dt.getFullYear(); //年 ，四位数

dt.getMonth(); //月（0-11），真实的月份要加上1

dt.getDate(); //日（0-31）

dt.getHours(); //小时（0-23）

dt.getMinutes(); //分钟（0-59）

dt.getSeconds(); //秒（0-59）

dt.getDay(); //星期几，返回值是 0（周日） 到 6（周六） 之间的一个整数。

**获取2017-06-10格式的日期**

function formatDate(dt){

var year = dt.getFullYear();

var month = dt.getMonth() + 1;

var date = dt.getDate();

if(month < 10){

month = '0' + month;

}

if(date < 10){

date = '0' + date;

}

return year + '-' + month + '-' + date;

}

var dt = new Date();

var formatDate = formatDate(dt);

console.log(formatDate);

###### 随机数 、数组的随机排序

Math.random() 产生一个[0，1)之间的随机数，每次执行，小数点后的位数并不是一致的

**获取随机数，要求是长度一致的字符串格式**

var random = Math.random();

var random = random + "0000000000";

var random = random.slice(0,10); //要求长度为10

console.log(random);

**数组随机排序方法：**

**法一：**

var arr = [1,2,3,4,5,6,7];

function randomSort(arr){

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

var index = parseInt(Math.random() \*　arr.length);

var temp = arr[index];

arr[index] = arr[i],

arr[i] = temp;

}

return arr;

}

console.log(randomSort(arr));

**法二：**

var arr = [1,2,3,4,5,6,7];

function randomSort(arr){

var newArr = [];

while(arr.length > 0){

var index = parseInt(Math.random() \*　arr.length);

newArr.push(arr[index]);

arr.splice(index,1);

}

return newArr;

}

console.log(randomSort(arr));

**法三：**

var arr = [1,2,3,4,5,6,7];

function randomSort(arr){

arr.sort(function(){

return Math.random() - 0.5;

});

return arr;

}

console.log(randomSort(arr));

###### JS面向对象——封装、继承（原型链继承、借用构造函数、组合继承）、多态

1. **封装**：

我们将一类实体共同具有的属性和行为提取出来抽象为一个类，将这些属性和行为都放在这个类里。就好像人类，可以具有name,sex,age等属性，同时也具有eat(),sleep()等行为，这就可以把他们封装成一个Person类。JS这种弱类型语言不像强类型语言中通过class关键字实现类的封装（ES6另谈），而是通过声明一个函数，在函数内部通过为this变量添加属性或者方法来实现对类添加属性和方法。

如：（Book例子）

function Book(id,name){

this.id = id;

this.name = name

}

**也可以在类的原型上添加属性和方法：**

Book.prototype.display = function(){};

//或者

Book.prototype = {

display : function(){}

};

通过this添加属性和在protoytpe中添加属性的**区别**：通过类创建一个新对象时，this指向的属性和方法都会得到相应的创建，而通过prototype继承的属性或方法是每个对象通过prototype访问到，所以每次创建一个新对象时这些属性和方法不会再次创建。

1. **继承——父类的特性传给子类**

**1.借用构造函数继承**

function Parent0(){

this.name = "parent0";

this.colors = ["red","blue","yellow"];

}

function Child0(){

**Parent0.call( this ); // 或apply**

this.type = "child0";

}

第6行，在子类（Child0）中执行父类（Parent0）的构造函数，通过这种调用，把父类构造函数的this指向为子类实例化对象引用，从而导致父类执行的时候父类里面的属性都会被挂载到子类的实例上去。

new Child0().name; // Parent0

new Child0().colors; // (3) ["red", "blue", "yellow"]

但是通过这种方式，父类原型上的东西是没法继承的

Parent0.prototype.sex = "男";

Parent0.prototype.say = function() {

console.log(" Oh,My God! ");

}

**new Child0().sex; // undefined**

**// Uncaught TypeError: (intermediate value).say is not a function**

**new Child0().say();**

缺点：Child1无法继承Parent1的原型对象，并没有真正的实现继承（部分继承）

**2.原型链继承（借用原型链实现继承）**

function Parent1(){

this.name = "parent1";

this.colors = ["red","blue","yellow"];

}

function Child1(){

this.name = "child1";

}

**Child1.prototype = new Parent1();**

这种方式能否解决借用构造函数继承的缺点呢？来看下面代码，我们依然为父类的原型添加sex属性和say方法：

Parent1.prototype.sex = "男";

Parent1.prototype.say = function() {

console.log(" Oh,My God! ");

}

**new Child1().sex; // 男**

**new Child1().say(); // Oh,My God!**

这种方式确实解决了上面借用构造函数继承方式的缺点。

但是，这种方式仍有缺点，我们来看如下代码：

var s1 = new Child1();

s1.colors.push("black");

var s2 = new Child1();

**s1.colors; // (4) ["red", "blue", "yellow", "black"]**

**s2.colors; // (4) ["red", "blue", "yellow", "black"]**

我们实例化了两个Child1，在实例s1中为父类的colors属性push了一个颜色，但是s2也被跟着改变了。造成这种现象的原因就是原型链上中的原型对象它俩是共用的。

这不是我们想要的，s1和s2这个两个对象应该是隔离的，这是这种继承方式的缺点。

**3.组合式继承**

这里所谓的组合是指组合借用构造函数和原型链继承两种方式。

function Parent2(){

this.name = "parent2";

this.colors = ["red","blue","yellow"];

}

function Child2(){

**Parent2.call(this);**

this.type = "child2";

}

**Child2.prototype = new Parent2()**

注意第6，9行，这种方式结合了借用构造函数继承和原型链继承的有点，能否解决上述两个实例对象没有被隔离的问题呢？

var s1 = new Child2();

s1.colors.push("black");

var s2 = new Child2();

s1.colors; // (4) ["red", "blue", "yellow", "balck"]

s2.colors; // (3) ["red", "blue", "yellow"]

可以看到，s2和s1两个实例对象已经被隔离了。

但这种方式仍有缺点。父类的构造函数被执行了两次，第一次是Child2.prototype = new Parent2()，第二次是在实例化的时候，这是没有必要的。

**3.1组合式继承优化1**

直接把父类的原型对象赋给子类的原型对象

function Parent3(){

this.name = "parent3";

this.colors = ["red","blue","yellow"];

}

Parent3.prototype.sex = "男";

Parent3.prototype.say = function(){console.log("Oh, My God！")}

function Child3(){

**Parent3.call(this);**

this.type = "child3";

}

**Child3.prototype = Parent3.prototype;**

var s1 = new Child3();

var s2 = new Child3();

console.log(s1, s2);

但是，我们来看如下代码：

console.log(s1 instanceof Child3); // true

console.log(s1 instanceof Parent3); // true

可以看到，我们无法区分实例对象s1到底是由Child3直接实例化的还是Parent3直接实例化的。用instanceof关键字来判断是否是某个对象的实例就基本无效了。

我们还可以用.constructor来观察对象是不是某个类的实例：

console.log(s1.constructor.name); // Parent3

从这里可以看到，s1的构造函数居然是父类，而不是子类Child3，这显然不是我们想要的。

**3.2组合式继承优化2**

这是继承的最完美方式

function Parent4(){

this.name = "parent4";

this.colors = ["red","blue","yellow"];

}

Parent4.prototype.sex = "男";

Parent4.prototype.say = function(){console.log("Oh, My God！")}

function Child4(){

**Parent4.call(this);**

this.type = "child4";

}

**Child4.prototype = Object.create(Parent4.prototype)；**

**Child4.prototype.constructor = Child4;**

Object.create是一种创建对象的方式，它会创建一个中间对象

var p = {name: "p"}

var obj = Object.create(p)

// Object.create({ name: "p" })

通过这种方式创建对象，新创建的对象obj的原型就是p，同时obj也拥有了属性name，这个新创建的中间对象的原型对象就是它的参数。

这种方式解决了上面的所有问题，是继承的最完美实现方式。

**4.ES6中继承**

Class 可以通过**extends**关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。

class Parent {

}

class Child1 **extends** Parent {

**constructor**(x, y, colors) {

**super**(x, y); // 调用父类的constructor(x, y)

this.colors = colors;

}

toString() {

return this.colors + ' ' + super.toString(); // 调用父类的toString()

}

}

上面代码中，constructor方法和toString方法之中，都出现了super关键字，它在这里表示父类的构造函数，用来新建父类的this对象。

子类必须在constructor方法中调用super方法，否则新建实例时会报错。如果子类没有定义constructor方法，这个方法会被默认添加，不管有没有显式定义，任何一个子类都有constructor方法。

ES5 的继承，实质是先创造子类的实例对象this，然后再将父类的方法添加到this上面（Parent.apply(this)）。ES6 的继承机制完全不同，实质是先创造父类的实例对象this（所以必须先调用super方法），然后再用子类的构造函数修改this。

1. **多态（多种调用方式）**

摘自设计模式：就是同一个方法多种调用方式。比如在js中对传入的参数做判断以实现多种调用方式，如定义一个add方法

function add(){

var arg = arguments,

len = arg.length;

switch(len){

case 0 : return 10;

case 1 : return + arg[0];

case 2 :return arg[0] + arg[1];

}

}

console.log(add()); //10

console.log(add(5)); //15

console.log(add(6,7)); //13

转化为类的形式：

function Add(){

function zero(){

return 10

}

function one(num){

return 10 + num;

}

function two(num1,num2){

return num1 + num2;

}

function add(){

var arg = arguments,

len = arg.length;

switch(len){

case 0 : return zero();

case 1 : return one(arg[0]);

case 2 : return two(arg[0],arg[1]);

}

}

var A = new Add();

console.log(A.add());

console.log(A.add(5));

console.log(A.add(6,7));

###### 从输入URL到页面加载发生了什么（包含了浏览器渲染页面的流程）

总体来说该过程分为以下几步:

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

**1.DNS解析**

**DNS指将字符串域名映射为对应的IP地址。**请求方发起请求的前提是知道自己的请求该发向哪，而在地址栏里输入的网址其实就是对方的地址，比如<www.baidu.com>、<www.taobao.com>等，这种写法很方便用户记忆，而无需输入一长串无记忆点的IP地址，但在网络中机器之间只能互相识别IP地址，所以要将域名映射为IP地址。

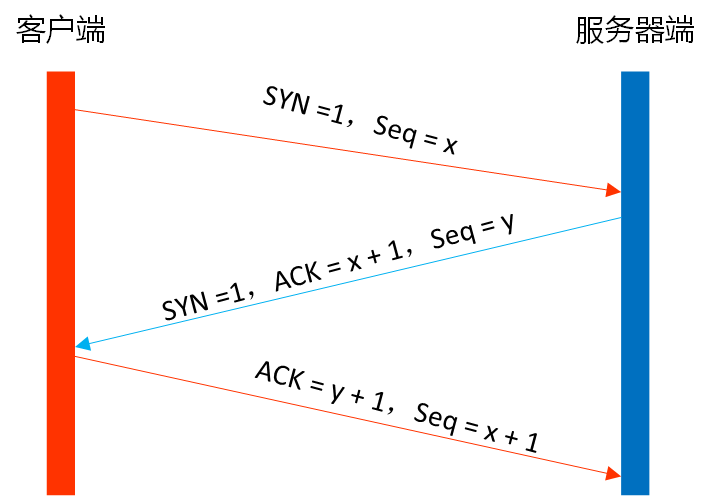
DNS是应用层协议，域名解析运行在UDP协议之上。**为什么选择UDP而非TCP？**原因如下：UDP无需连接，时效性更好，进行一次查询只需要两个DNS包。而TCP需要先用3个包建立连接，再用2个DNS包进行查询，最后用4个包断开连接，连接成本远大于查询本身，容易让DNS服务器不堪重负。

**DNS查询**

* 先检查本地hosts文件是否有这个网址映射关系，如果有就调用这个IP地址映射，完成域名解析。
* 否则，查找本地DNS解析器缓存，如果查找到则返回。
* 否则，查找本地DNS服务器，如果查找到则返回。
* 否则，按根域名服务器 -->顶级域名服务器-->次级域名服务器 -->主机名的IP地址的顺序依次查找下去，直到查找到IP地址。以输入<www.baidu.com>为例，默认所有网址尾部是有个.的，这里即<www.baidu.com>.，这个.对应的就是根域名服务器，所以网址真正的解析过程为:. -> .com-> google.com.->www.baidu.com.。

**2.TCP连接**

HTTP协议是使用TCP作为其传输层协议的,在HTTP请求发送前还要通过**三次握手**建立连接。



1. 第一次握手：建立连接。客户端发送连接请求报文段，将SYN位置为1，Sequence Number为x；然后，客户端进入SYN\_SEND状态，等待服务器的确认；
2. 第二次握手：服务器收到SYN报文段。服务器收到客户端的SYN报文段，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x+1(Sequence Number+1)；同时，自己自己还要发送SYN请求信息，将SYN位置为1，Sequence Number为y；服务器端将上述所有信息放到一个报文段（即SYN+ACK报文段）中，一并发送给客户端，此时服务器进入SYN\_RECV状态；
3. 第三次握手：客户端收到服务器的SYN+ACK报文段。然后将Acknowledgment Number设置为y+1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入ESTABLISHED状态，完成TCP三次握手。

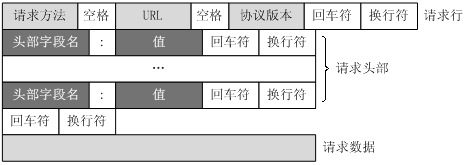
**问：为什么需要进行三次握手，而不是两次握手？**

原因是两次握手不可靠，防止已经失效的连接请求报文段突然又传送到了服务器端而产生错误。设想只有两次握手的场景，假如客户端某次发出的请求由于网络滞留原因最终到达了服务器端，但此时该请求已失效，而服务器端却会认为这是客户端发起的一次新请求，就会向客户端发出应答报文。而客户端认为自己没有发出请求，就不会对此做出响应，此时另一端的服务器通过两次握手以为建立了连接，一直等待客户端发来数据，白白浪费了资源。而三次握手的情况下，服务器端只有明确收到了客户端的应答报文，才可以确定客户端是想要建立连接的。

**3.发送HTTP请求**

其实这部分又可以称为前端工程师眼中的HTTP，它主要发生在客户端。发送HTTP请求的过程就是构建HTTP请求报文并通过TCP协议发送到服务器指定端口**(HTTP协议默认端口80, HTTPS协议默认端口443)**。

更相关的内容在[你想知道的AJAX都在这里](http://www.wuxuann-n.cn/2017/08/20/%E4%BD%A0%E6%83%B3%E7%9F%A5%E9%81%93%E7%9A%84AJAX%E9%83%BD%E5%9C%A8%E8%BF%99%E9%87%8C/)一文中已有介绍，可参考。这里就放出HTTP请求报文的直观图，并补充上文没讲的一点内容。



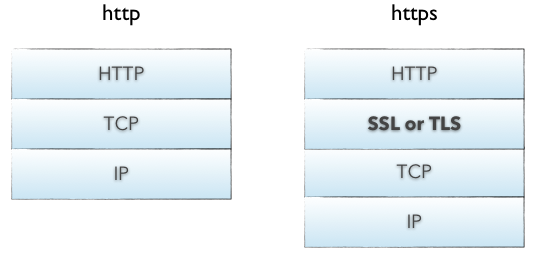
**补充：http请求方法**

HTTP/1.1 定义的请求方法有8种：GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS、TRACE。

* OPTIONS：返回服务器针对特定资源所支持的HTTP请求方法。也可以利用向Web服务器发送'\*'的请求来测试服务器的功能性。
* HEAD：向服务器索要与GET请求相一致的响应，只不过响应体将不会被返回。这一方法可以在不必传输整个响应内容的情况下，就可以获取包含在响应消息头中的元信息。
* GET：向特定的资源发出请求。
* POST：向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的创建和/或已有资源的修改。
* PUT：向指定资源位置上传其最新内容。
* DELETE：请求服务器删除Request-URI所标识的资源。
* TRACE：回显服务器收到的请求，主要用于测试或诊断。
* CONNECT：HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

**补充：HTTPS**

现在https用的也是越来越多，所以这边穿插一点https的知识！



HTTP报文是包裹在TCP报文中发送的，服务器端收到TCP报文时会解包提取出HTTP报文，网络请求需要很多路由器的转发，HTTP报文是明文的，所以容易在中间节点处被篡改信息。那么在HTTP报文进入TCP报文之前，先使用SSL/TLS对HTTP报文进行加密就可以解决这个问题了。从网络的层级结构看SSL/TLS位于HTTP协议与TCP协议之间。

**为什么HTTPS安全**

HTTPS更安全，是因为它利用 了SSL/TLS协议传输，SSL/TLS协议的基本思路是采用[公钥加密法](http://en.wikipedia.org/wiki/Public-key_cryptography)，也就是说，客户端先向服务器端索要公钥，然后用公钥加密信息，服务器收到密文后，用自己的私钥解密。虽然HTTPS相比于HTTP更安全，但是势必会带来一些时间上的损耗，如握手和加密等过程，是否使用HTTPS需要根据具体情况在安全和性能方面做出权衡。具体过程请参考经典的阮一峰先生的博客[SSL/TLS协议运行机制的概述](http://www.ruanyifeng.com/blog/2014/09/illustration-ssl.html)。

**4.服务器处理请求并返回HTTP报文**

这里只放出HTTP响应报文的图：



服务器返回给浏览器的文本信息，通常HTML, CSS, JS, 图片等文件就放在响应正文这一部分。

**5.浏览器解析渲染页面**

接下来的工作就是浏览器接收HTML,CSS,JS文件后，如何将页面呈现在我们眼前？这就是浏览器解析渲染页面的过程了！

**总体流程简单来说：**

* 根据HTML结构生成DOM Tree
* 根据CSS生成CSSOM
* 将DOM和CSSOM整合形成Render Tree
* 根据Render Tree开始渲染和展示
* 遇到<script>时，会执行并阻塞渲染

**整个过程解释：**

浏览器接收到html文档，会被浏览器的HTML PARSER(HTML 解析器)解析， 通过词法分析将tag分析为相应的token，对HTML文档从上往下依次去解析token，这个过程的特点一是从上往下的，二是在此法分析的过程可以解析出link、script这样的标签，这些标签里面对应的外部资源会进一步由浏览器向网络发起请求（请求CSS和js等资源 ）。css相关的资源请求回来后会由浏览器生成相应的CSS树。将dom树和css树进行合并生成render tree渲染树，之后再进行布局layout和绘制paint，最后成功渲染出页面。

**问1：为什么要把css放在head中？**

将css样式放在head中引入，会先加载css，之后浏览器就知道这个css规则了，然后他在渲染DOM节点的时候就按照已知的css规则来渲染。

反例：如果把css内容放在DOM节点之后引入，会首先按照默认样式渲染dom内容（比如字体16px），之后加载完css发现样式的宽啊高啊都得改变，又得重新去渲染一遍，这时可能会出现屏幕的跳变，这个变化可能快可能慢，和你的电脑配、网络情况、页面规模等有关，用户体验差，所以得出结论，css要放在head中，而不要放在body中。

**问2：为什么要把js放在body最下面？**

script引入的js会阻塞页面渲染，因为script的js代码有权改变dom结构和dom内容。

遇到script会先加载和执行相应的js，该过程完毕后才会继续加载解析后续的页面。把js放在body最下面，一方面这样就不会阻塞页面的渲染了，能让页面更快地出来；另一方面放在下面，加载到它时，它已经能拿到所有标签，而放到中间，就有可能拿不到它后面的标签了，对后面标签所做的操作就失效了。

**结束连接**



**1.第一次挥手：** 主动方（可以使客户端，也可以是服务器端），设置Sequence Number，向被动方发送一个FIN报文段，告诉另一方我打算断开连接，后面不会再发送数据了，但是此时主动关闭方还是可以接收数据的。

**2.第二次挥手：**被动方收到了主动方发送的FIN报文段，向主动方回一个ACK报文段（Acknowledgment Number为Sequence Number + 1），告诉主动方，我“同意”你的关闭请求；

**3.第三次挥手：**被动方收到断开连接信息时，可能还有数据没有传完，所以等待数据全部传输结束后，再发送一条 FIN 为 1 的信息，告诉对方也做了断开连接的准备，但没有断开。

**4.第四次挥手：** 主动方收到被动方发送的FIN报文段，向被动方发送ACK报文段，被动方收到主动方的ACK报文段以后，就关闭连接；此时，主动方等待2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，主动方也可以关闭连接了。

**问：为什么需要进行四次挥手？**

第一次挥手是在主动方没有数据需要再发送过去的情况下发起的，第二挥手是被动方发送ACK表示同意，当然，如果在第二次挥手时被动方也向主动方发送FIN包请求断开连接似乎也行，但考虑到它自己有可能还有数据没有发送完，所以当它真的没有数据可发了，才会进行第三次挥手发送FIN包告诉主动方，我也不再发送数据了，之后主动发再送ACK以表确认，即第四次挥手。

简而言之，一端断开连接需要两次挥手（请求和回应），两端断开连接就需要四次挥手了。

###### Promise

**1.** Promise对象代表一个异步操作，有**三种状态**：pending: 进行中，表示还没有得到结果；

resolved(Fulfilled):已成功；rejected: 已失败。**一旦状态改变，就不会再变，也不可逆，任何时候都可以得到这个结果。**

**2．then（）方法**

then方法是定义在原型对象Promise.prototype上的。它的作用是为 Promise 实例添加状态改变时的回调函数。第一个参数是resolved状态的回调函数，第二个参数（可选）是rejected状态的回调函数。

then方法返回的是一个**新的Promise实例**（注意，**不是原来那个Promise实例**）。因此可以采用链式写法，即then方法后面再调用另一个then方法。

**3．catch()方法**

Promise.prototype.catch方法是.then(null, rejection)的别名，用于指定发生错误时的回调函数。

一般来说，不要在then方法里面定义Reject状态的回调函数（即then的第二个参数），总是使用catch方法。

**4.Promise.all([p1,p2,p3])方法**

Promise.all方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例，如果不是，就会先调用Promise.resolve方法，将参数转为 Promise 实例，再进一步处理。（Promise.all方法的参数可以不是数组，但必须具有 Iterator 接口，且返回的每个成员都是 Promise 实例。

p的状态由p1、p2、p3决定，分成两种情况。

（1）只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

（2）只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

**5.Promise.race([p1,p2,p3])方法：**

只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

1. **利用promise封装一个ajax请求**

function getJSON(url){

var promise = new Promise(fuction(resolve,reject){

var xhr = new XMLHttpRequest();

xhr.open("GET",url,true);

xhr.send();

xhr.onreadystatechange = function(){

if(xhr.readyState == 4){

if(xhr.status == 200){

try{

var responese = JSON.parse(xhr.responseText);

resolve(response);

}catch(e){

reject(e);

}

}else{

reject(new Error(xhr.statusText));

}

}

}

});

return promise;

}

getJSON(url).then(resp => console.log(resp));

###### 浏览器内核的理解

浏览器内核又可以分成两部分：**渲染引擎**(layout engineer 或者 Rendering Engine)和 **JS 引擎**。

**渲染引擎**负责取得网页的内容（HTML、XML、图像等等）、整理讯息（例如加入 CSS 等），以及计算网页的显示方式，然后会输出至显示器或打印机。浏览器的内核的不同对于网页的语法解释会有不同，所以渲染的效果也不相同。所有网页浏览器、电子邮件客户端以及其它需要编辑、显示网络内容的应用程序都需要内核。

**JS 引擎**则是解析和执行 javascript 语言来实现网页的动态效果。

最开始渲染引擎和 JS 引擎并没有区分的很明确，后来 JS 引擎越来越独立，**内核就倾向于只指渲染引擎**。

常见的浏览器内核可以分这四种：

Trident内核：IE,MaxThon,TT,The World,360,搜狗浏览器等。[又称MSHTML]

Gecko内核：Netscape6及以上版本，FF,MozillaSuite/SeaMonkey等

Presto内核：Opera7及以上。 [Opera内核原为：Presto，现为：Blink;]

Webkit内核：Safari,Chrome等。 [ Chrome的：Blink（WebKit的分支）]

###### 三列布局（圣杯布局、双飞翼布局、使用绝对定位、分别左浮动和右浮动）

1. **圣杯布局**

<style>

body{

min-width: 730px; /\* 2\*left + right \*/

}

.container{

**padding-left: 210px;**

**padding-right: 310px;**

}

.column{

float: left;

height: 300px;

}

.middle{

width: 100%;

background-color: rgba(255, 0, 0, .5);

}

.left{

**position: relative;**

**left: -210px;**

width: 200px;

**margin-left: -100%;**

background-color: rgba(0, 255, 0, .5);

}

.right{

**position: relative;**

**right: -310px;**

width: 300px;

**margin-left: -300px;**

background-color: rgba(0, 0, 255, .5);

}

</style>

<body>

<div class="container">

<div class="column middle"></div>

<div class="column left"></div>

<div class="column right"></div>

</div>

**2.双飞翼布局（比圣杯好理解些）**

<style>

.middle-wrapper {

float: left;

width: 100%;

}

.middle {

height: 300px;

**margin-left: 210px;**

**margin-right: 310px;**

background-color: rgba(255, 0, 0, .5);

}

.left {

float: left;

width: 200px;

height: 300px;

**margin-left: -100%;**

background-color: rgba(0, 255, 0, .5);

}

.right {

float: left;

width: 300px;

height: 300px;

**margin-left: -300px;**

background-color: rgba(0, 0, 255, .5);

}

</style>

<body>

<div class="middle-wrapper">

<div class="middle"></div>

</div>

<div class="left"></div>

<div class="right"></div>

</body>

**圣杯布局、双飞翼布局异同：**

**相同之处：**都是让三列浮动，然后通过负外边距形成三列布局，都是把主列放在文档流最前面，使主列优先加载。

**区别：**在于如何处理中间主列的位置，圣杯布局是利用父容器的左、右内边距定位；双飞翼布局是把主列嵌套在div后利用主列的左、右外边距定位。

1. **使用绝对定位实现三列布局（用的多）：中间列通过左右外边距定位，左右列用absolute定位。**

<style>

.container{

**position: relative;**

}

.middle{

margin-left: 210px;

margin-right: 310px;

height: 300px;

background-color: red;

}

.left{

position: absolute;

top: 0;

left: 0;

width: 200px;

height: 300px;

background-color: blue;

}

.right{

position: absolute;

top: 0;

right: 0;

width:300px;

height: 300px;

background-color: green;

}

</style>

<body>

<div class="container">

<div class="middle"></div>

<div class="left"></div>

<div class="right"></div>

</div>

</body>

1. **使用左浮动和右浮动：先左列左浮动，右列右浮动，然后中间列通过设置左右外边距，注意先后顺序。**

<style>

.container{

position: relative;

}

.left{

**float: left;**

width: 200px;

height: 300px;

background-color: blue;

}

.right{

**float: right;**

width:300px;

height: 300px;

background-color: green;

}

.middle{

**margin-left: 210px;**

**margin-right: 310px;**

height: 300px;

background-color: red;

}

</style>

<body>

<div class="container">

**<!-- middle写在最后 -->**

<div class="left"></div>

<div class="right"></div>

<div class="middle"></div>

</div>

</body>

###### 事件循环机制（event loop）——一些结论，涉及promise、setTimeout

1. JavaScript的一大特点就是单线程，而这个线程中拥有唯一的一个事件循环。
2. JavaScript代码的执行过程中，除了依靠**函数调用栈**来搞定函数的执行顺序外，还依靠**任务队列(task queue**)来搞定另外一些代码的执行。

3. 一个线程中，事件循环是唯一的，但是任务队列可以拥有多个。

4. 任务队列又分为**macro-task（宏任务）与micro-task（微任务）**，在最新标准中，它们被分别称为task与jobs。

5. **macro-task**大概包括：script(整体代码), setTimeout, setInterval, setImmediate, I/O, UI rendering。

6. **micro-task**大概包括: process.nextTick, Promise, Object.observe(已废弃), MutationObserver(html5新特性)

1. setTimeout/Promise等我们称之为任务源。而进入任务队列的是他们指定的具体执行任务。

来自不同任务源的任务会进入到不同的任务队列。其中setTimeout与setInterval是同源的。

8. **事件循环的顺序**，决定了JavaScript代码的执行顺序。它从script(整体代码)开始第一次循环。之后全局上下文进入函数调用栈。直到调用栈清空(只剩全局)，然后执行所有的micro-task。当所有可执行的micro-task执行完毕之后。循环再次从macro-task开始，找到其中一个任务队列执行完毕，然后再执行所有的micro-task，这样一直循环下去。

9. 其中每一个任务的执行，无论是macro-task还是micro-task，都是借助函数调用栈来完成。

URI 和 URL 区别

**1. URI**  
Web上可用的每种资源如HTML文档、图像、视频片段、程序等都是一个来URI来定位的；  
URI一般由三部组成

①访问资源的命名机制

②存放资源的主机名

③资源自身的名称，由路径表示，着重强调于资源。

**2. URL**  
URL是Internet上用来描述信息资源的字符串，主要用在各种WWW客户程序和服务器程序上，特别是著名的Mosaic。  
采用URL可以用一种统一的格式来描述各种信息资源，包括文件、服务器的地址和目录等。  
URL一般由三部组成

①协议(或称为服务方式)

②存有该资源的主机IP地址(有时也包括端口号)

③主机资源的具体地址。如目录和文件名等

**3、URN：Uniform Resource Name，**统一资源名称。

**4.区别：**URI，是统一资源标识符，用来唯一的标识一个资源。而URL是统一资源定位器，它是一种具体的URI，即URL可以用来标识一个资源，而且还指明了如何locate这个资源。而URN，统一资源命名，是通过名字来标识资源，比如mailto:java-net@java.sun.com。也就是说，URI是以一种抽象的，高层次概念定义统一资源标识，而URL和URN则是具体的资源标识的方式。**通俗地说，URL和URN是URI的子集，URI属于URL更高层次的抽象，一种字符串文本标准。**

JS浅拷贝和深拷贝

JavaScript存储对象都是存地址的，所以浅拷贝会导致 obj1 和obj2 指向同一块内存地址。改变了其中一方的内容，都是在原来的内存上做修改会导致拷贝对象和源对象都发生改变，而深拷贝是开辟一块新的内存地址，将原对象的各个属性逐个复制进去。对拷贝对象和源对象各自的操作互不影响**。**

1. **（对于数组）Array的slice和concat方法**

如果数组元素不是引用数据类型，使用slice和concat可以达到深拷贝的效果；否则，也只是浅拷贝。

**二、（对于数组，对象）JSON对象的parse方法（JS对象序列化成JSON字符串）和stringify方法（JSON字符串解析成JS对象）**

可以满足基本的深拷贝需求，拷贝对象必须遵从JSON的格式，且对于正则表达式类型、函数类型等无法进行深拷贝。

**例子：**

//例1 **正常实现了深拷贝**

var source = { name:"source", child:{ name:"child" } }

**var target = JSON.parse(JSON.stringify(source));**

target.name = "target"; //改变target的name属性

console.log(source.name); //source

console.log(target.name); //target

target.child.name = "target child"; //改变target的child

console.log(source.child.name); //child

console.log(target.child.name); //target child

//例2 **对象中的属性包含函数或正则**

var source = { name:function(){console.log(1);}, child:new RegExp("e") }

var target = JSON.parse(JSON.stringify(source));

console.log(target.name); /**/undefined**

console.log(target.child); //**Object {}**

**三、（对于数组、对象）递归调用“浅拷贝”，实现深拷贝**

function deepCopy(obj){

var newObj = obj.constructor === "Array" ? [] : {};

for(var i in obj){

newObj[i] = typeof obj[i] === "object" ? deepCopy(obj[i]) : obj[i];

}

return newObj;

}

浏览器缓存（HTTP缓存机制）

一、新鲜度（过期机制）

Expires是HTTP1.0中的响应头，Cache-Control是HTTP1.1中新增的响应头。

**1、Cache-Control：**

1. max-age（单位为s）：指定设置缓存最大的有效时间，定义的是时间长短。当浏览器向服务器发送请求后，在max-age这段时间里浏览器就不会再向服务器发送请求了。
2. **s-maxage**（单位为s）同max-age，只用于共享缓存（比如CDN缓存）。
3. **public** 指定响应可以在代理缓存中被缓存，于是可以被多用户共享。如果没有明确指定private，则默认为public。
4. **private**响应只能在私有缓存中被缓存，不能放在代理缓存上。对一些用户信息敏感的资源，通常需要设置为private。
5. **no-store**绝对禁止缓存任何资源，也就是说每次用户请求资源时，都会向服务器发送一个请求，每次都会下载完整的资源。通常用于机密性资源。
6. **no-cache** 表示必须先与服务器确认资源是否被更改过（依靠If-None-Match和Etag），然后再决定是否使用本地缓存。

**2.Expires：**

缓存过期时间，用来指定资源到期的时间，是**服务器端的具体的时间点**（形如Last-Modified：Fri,19 Dec 2014 02:50:47 GMT）。由于Cache-Control使用的是相对时间，而Expires可能存在客户端与服务器端时间不一样的问题（准确时**Expires =max-age +   “每次下载时的当前请求时间”**），所以我们更倾向于选择Cache-Control。

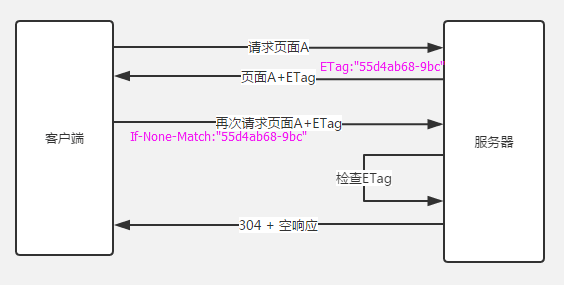
**二、服务器再验证（协商缓存机制/验证机制）**

浏览器或代理缓存中缓存的资源过期了，并不意味着它和原始服务器上的资源有实际的差异，仅仅意味着到了要进行核对的时间了。这种情况被称为服务器再验证。

* 如果资源发生变化，则需要取得新的资源，并在缓存中替换旧资源。
* 如果资源没有发生变化，缓存只需要获取新的响应头，和一个新的过期时间，对缓存中的资源过期时间进行更新即可。HTTP1.1推荐使用的验证方式是If-None-Match/Etag，在HTTP1.0中则使用If-Modified-Since/Last-Modified。

**1. Etag与If-None-Match**

ETag是根据**实体内容生成一段hash字符串**，标识资源的状态，由服务端产生。浏览器会将这串字符串传回服务器，验证资源是否已经修改，如果没有修改，过程如下



**2. Last-Modified/If-Modified-Since**

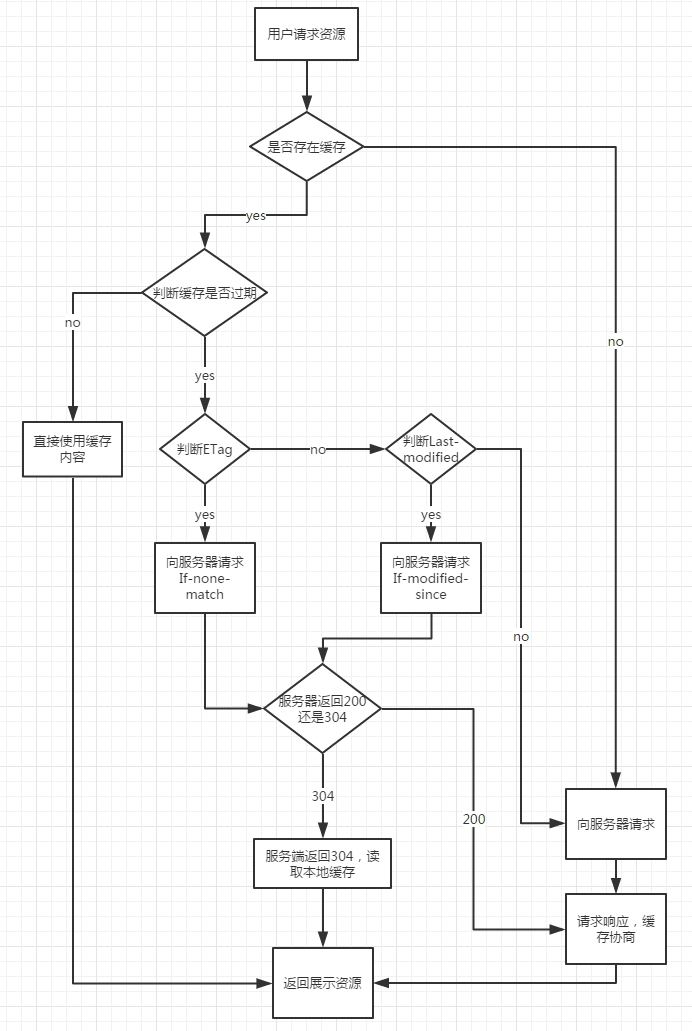
  服务器端文件的**最后修改时间**,是检查服务器端资源是否更新的一种方式。当浏览器再次进行请求时，会向服务器传送If-Modified-Since报头，询问Last-Modified时间点之后资源是否被修改过。如果没有修改，则返回码为304，使用缓存；如果修改过，则再次去服务器请求资源，返回码和首次请求相同为200，资源为服务器最新资源。这两个头部都是日期（形如Last-Modified：Fri,19 Dec 2014 02:50:47 GMT），验证过程与Etag类似。

**使用ETag可以解决Last-modified存在的一些问题：**

a、某些服务器不能精确得到资源的最后修改时间，这样就无法通过最后修改时间判断资源是否更新

b、如果资源修改非常频繁，在秒以下的时间内进行修改，而Last-modified只能精确到秒

c、一些资源的最后修改时间改变了，但是内容没改变(比如只修改了注释)，使用ETag就认为资源还是没有修改的。



**注意，我们讨论的所有关于缓存资源的问题，都仅仅针对**GET**请求。而对于**POST**,**DELETE**,**PUT**这类行为性操作通常不做任何缓存。**

###### HTTP状态码、301和302区别、http长连接、无状态

1. **状态码**

100 Continue 继续。客户端应继续其请求

101 Switching Protocols 切换协议。服务器根据客户端的请求切换协议。只能切换到更高级的协议，例如，切换到HTTP的新版本协议

200 OK 请求成功。一般用于GET与POST请求

201 Created 已创建。成功请求并创建了新的资源

202 Accepted 已接受。已经接受请求，但未处理完成

203 Non-Authoritative Information 非授权信息。请求成功。但返回的meta信息不在原始的服务器，而是一个副本

204 No Content 无内容。服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档

205 Reset Content 重置内容。服务器处理成功，用户终端（例如：浏览器）应重置文档视图。可通过此返回码清除浏览器的表单域

206 Partial Content 部分内容。服务器成功处理了部分GET请求

300 Multiple Choices 多种选择。请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户终端（例如：浏览器）选择

301 Moved Permanently 永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替

302 Found 临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有URI

303 See Other 查看其它地址。与301类似。使用GET和POST请求查看

304 Not Modified 未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源

305 Use Proxy 使用代理。所请求的资源必须通过代理访问

306 Unused 已经被废弃的HTTP状态码

307 Temporary Redirect 临时重定向。与302类似。使用GET请求重定向

400 Bad Request 客户端请求的语法错误，服务器无法理解

401 Unauthorized 请求要求用户的身份认证

402 Payment Required 保留，将来使用

403 Forbidden 服务器理解请求客户端的请求，但是拒绝执行此请求

404 Not Found 服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面

405 Method Not Allowed 客户端请求中的方法被禁止

406 Not Acceptable 服务器无法根据客户端请求的内容特性完成请求

500 Internal Server Error 服务器内部错误，无法完成请求

501 Not Implemented 服务器不支持请求的功能，无法完成请求

502 Bad Gateway 充当网关或代理的服务器，从远端服务器接收到了一个无效的请求

503 Service Unavailable 由于超载或系统维护，服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的Retry-After头信息中

504 Gateway Time-out 充当网关或代理的服务器，未及时从远端服务器获取请求

505 HTTP Version not supported 服务器不支持请求的HTTP协议的版本，无法完成处理

**二、301和302**

1. **共同点：**301和302状态码都表示重定向，就是说浏览器在拿到服务器返回的这个状态码后会自动跳转到一个新的URL地址，这个地址可以从响应的**Location**首部中获取（用户看到的效果就是他输入的地址A瞬间变成了另一个地址B）
2. **区别：**301表示旧地址A的资源已经被永久地移除了（这个资源不可访问了），**搜索引擎在抓取新内容的同时也将旧的网址交换为重定向之后的网址**；302表示旧地址A的资源还在（仍然可以访问），这个重定向只是临时地从旧地址A跳转到地址B，**搜索引擎会抓取新的内容而保存旧的网址。在生产环境，建议使用301永久重定向，不使用302临时跳转，因为对于搜索引擎来说，会自动将原来页面的收录和权重转移到新的页面，有利于SEO.**
3. **为什么进行重定向？**

比如1）网站调整（如改变网页目录结构）；2）网页被移到一个新地址；3）网页扩展名改变(如应用需要把.php改成.Html或.shtml)。

这些情况下，如果不做重定向，则用户收藏夹或搜索引擎数据库中旧地址只能让访问客户得到一**个404页面错误信息，访问流量白白丧失**；再者某些注册了多个域名的网站，也需要通过重定向让访问这些域名的用户自动跳转到主站点等。

**4.301跳转的场景**

1）域名到期不想续费（或者发现了更适合网站的域名），想换个域名。

2）在搜索引擎的搜索结果中出现了不带www的域名，而带www的域名却没有收录，这个时候可以用301重定向来告诉搜索引擎我们目标的域名是哪一个。

3）空间服务器不稳定，换空间的时候。

1. **HTTP长连接、无状态**

**1.长连接**

HTTP1.1规定了默认保持长连接（持久连接），数据传输完成后**保持TCP连接不断开**（不发RST包、不四次握手），等待在同域名下继续用这个通道传输数据；相反的就是短连接。HTTP首部的Connection: Keep-alive，默认是长连接，如不使用长连接，则Connection: close.

客户端的长连接不可能无限期，会有一个超时时间，服务器有时候会告诉客户端超时时间，譬如：

Connection: keep-alive

Keep-Alive: timeout=20

Keep-Alive: timeout=20，表示这个TCP通道可以保持20秒。另外还可能有max=XXX，表示这个长连接最多接收XXX次请求就断开。对于客户端来说，如果服务器没有告诉客户端超时时间也没关系，服务端可能主动发起四次握手断开TCP连接，客户端能够知道该TCP连接已经无效；另外TCP还有心跳包来检测当前连接是否还活着，方法很多，避免浪费资源。

**2.无状态**

　 意味着HTTP 每个请求都是独立的，我们给服务器发送 HTTP 请求之后，服务器根据请求，会给我们发送数据过来，但是，发送完，不会记录任何。

　 缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大，而在服务器不需要先前信息时它的应答就较快。

优点在于解放了服务器，每一次请求“点到为止”不会造成不必要连接占用，缺点在于每次请求会传输大量重复的内容信息。

于是，两种用于保持 HTTP 连接状态的技术就应运而生了，一个是 Cookie，而另一个则是[Session](http://www.chinabyte.com/keyword/session/" \t "http://network.chinabyte.com/240/_blank)。

###### Cookie 和 Session区别——登录网址后如何区别你是你？

1. cookie存放在浏览器端，session数据放在服务器端。
2. cookie不安全，本地的cookie容易被别人获取，进行cookie欺骗。
3. Session会在一定时间保存在服务器上（默认时间20分钟），当访问增多，会很占用服务器的性能。考虑到减轻服务器的性能，使用cookie。
4. 建议：将登陆信息等重要信息存放在session，其他信息如果需要保留，可以放在cookie中。

**例子：登陆网站后如何区别你是你？**

第一次登陆后，将用户名、密码发送给服务器，服务器校验身份认证成功后，为该用户生成token（唯一的），将密码清空，然后session中存放token加上用户的一些基本信息（用户名、密码空、手机、邮箱等），然后把token发送给浏览器作为cookie值。之后请求另一个页面时，浏览器把此cookie发送给服务器，服务器只需拿到cookie值就可以做身份验证，如没有匹配到cookie值，就需要用户重新登录了。

**补充：Cookie：<https://segmentfault.com/a/1190000004556040>**

cookie选项包括：expires、domain、path、secure、HttpOnly。

**1.服务器端设置cookie，可以设置cookie 的所有选项：**

response header中有一项叫set-cookie，是服务端专门用来设置cookie，有多少个cookie就多少个set-cookie字段。

1. **客户端设置cookie，**可以设置expires、domain、path、secure（有条件：只有在https协议的网页中，客户端设置secure类型的 cookie 才能成功），但无法设置HttpOnly选项：

形如：document.cookie="age=12; expires=Thu, 26 Feb 2116 11:50:25 GMT; domain=sankuai.com; path=/";

添加多个cookie需要重复执行：

document.cookie = "name=Jonh";

document.cookie = "age=12";

document.cookie = "class=111";

1. **修改cookie过期时间**

服务器端Response.Cookies("user\_name").Expires=。。

客户端直接对document.cookie赋值：document.cookie = "name=Jonh; expires=" + XX;

###### Cookie、localStorage和sessionStorage区别

1. **用途不一样**。Cookie是网站为了标示用户身份而存储在本地终端的数据，在同源的http请求中携带，会在浏览器和服务器端来回传递。而后两个不会把数据发送给客户端，仅作为本地存储。（在html5的localStorage和sessionStorage出现之前，都只是借用了cookie的本地存储功能）
2. **存储大小**。cookie数据大小不超过4kB，localStorage和sessionStorage最大能达到5M。
3. **有效期**。cookie默认是浏览器关闭后就没了，但也可以设置有效期，在该有效期之前都有用，即使窗口或浏览器关闭；localStorage持久存储，除非手动删除；sessionStorage数据在当前浏览器窗口关闭后自动删除。

**4.API易用性**。cookie的API需要封装使用，而且也比较麻烦；后两个有现成的API可使用，setItem(), getItem();

###### arguments.callee和function.caller

arguments.callee返回当前正在执行的函数，function.caller返回函数的调用体所在函数。

arguments.callee 在哪一个函数中运行，它就代表哪一个函数。由于匿名函数无名可调，这时就可以用arguments.callee来代替匿名的函数，比如自身调用自身函数会用到。

**另外 ：arguments.length表示实参长度；** **arguments.callee.length表示形参长度。**

此外在ES5的严格模式下禁止调用caller、callee、arguments变量，会报错。

###### 作用域链与闭包

简单举例：全局作用域中有个A函数，A函数中包含了一个B函数。在全局作用域中想访问A或者B中的局部变量是不行的，而在B中是可以一层一层向外部搜索访问到A直至全局作用域中的变量的，这是由于Javascript语言特有的"链式作用域"结构，子对象会一级一级地向上寻找所有父对象的变量。

现在如何实现在全局中访问到A中的局部变量呢？正常情况下式办不到的，变通下方法，使用闭包可以办到。因为既然B可以读取A中的局部变量，那么只要把B作为返回值，我们就可以在外部读取A的内部变量了。

****作用域链（即自由变量的查找），是由一系列变量对象组成（当前环境与上层环境的一系列变量对象组成），它保证了当前执行环境对符合访问权限的变量和函数的有序访问。****

**一、闭包的特性**

**函数内再嵌套一个函数：**

**函数内部可以引用外层的变量和参数：**

**参数和变量不会被垃圾回收机制回收：**JS拥有自动的垃圾回收机制，关于垃圾回收机制，有一个重要的行为，那就是，当一个值，在内存中失去引用时，垃圾回收机制会根据特殊的算法找到它，并将其回收，释放内存。而我们知道，函数的执行上下文，在执行完毕之后，生命周期结束，那么该函数的执行上下文就会失去引用。其占用的内存空间很快就会被垃圾回收器释放。可是闭包的存在，会阻止这一过程。

**二、使用闭包的注意点**

1）由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

2）闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

**三、闭包的使用场景**：

1. 函数作为返回值：

function F1(){

var a = 100;

//返回一个函数（函数作为返回值）

return function(){

console.log(a);

}

}

var f1 = F1();

var a =200;

f1(); //100

1. 函数作为参数传递

function F1(){

var a = 100;

return function(){

console.log(a);

}

}

var f1 = F1();

function F2(fn){

var a = 200;

fn();

}

F2(f1); //100

###### 排序

**一、冒泡** 复杂度: O(N^2) 稳定

function bubbleSort(arr){

var len = arr.length;

for(var i = 0;i < len;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;

}

bubbleSort([5,4,3,2,1]);

1. **快排** 复杂度: O(nlogn) 不稳定

/\* "快速排序"的思想很简单，整个排序过程只需要三步：

　　（1）在数据集之中，选择一个元素作为"基准"（pivot）。(基准值可以任意选择，但是选择中间的值比较容易理解。)

　　（2）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。

　　（3）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){

//不需要把arr的长度缓存出来，即var len = arr.length; 因为下面splice会改变该len。

if(arr.length <= 1 ){ return arr;}

var index = Math.floor(arr.length / 2),

valueOfIndex = arr.splice(index,1)[0];

var left =[],right=[];

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

if(arr[i] < valueOfIndex){

left.push(arr[i]);

}else{

right.push(arr[i]);

}

}

return quickSort(left).concat(valueOfIndex,quickSort(right));

}

quickSort([5,4,3,2,1]);

1. **直接插入** 复杂度: O(n^2) 稳定

function insertionSort(arr){

var len = arr.length;

var curValue,preIndex;

for(var i = 1;i < len;i++){

curValue = arr[i];

preIndex = i-1;

while(preIndex >= 0 && arr[preIndex] > curValue){

arr[preIndex + 1] =arr[preIndex];

preIndex--;

}

arr[preIndex + 1] = curValue;

}

return arr;

}

insertionSort([5,4,3,2,1]);

**四、选择排序** 复杂度: O(n^2) 稳定

function selectionSort(arr){

var len = arr.length;

var minIndex,temp;

for(var i = 0; i < len -1;i++){

minIndex = i;

for(var j = i; j < len; j++){

if(arr[j] < arr[minIndex]){

minIndex = j;

}

}

temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

return arr;

}

selectionSort([5,4,3,2,1]);

###### CSS reset 样式重置，清除浏览器默认样式；以及更好的Normalize.css，保留一些浏览器默认样式

1. reset的作用

现在所使用的主流浏览器对一些标签的默认属性上并没有做到统一，所以我们偶尔会发现，某个页面在chrome浏览器上很正常，到了firefox上面却有意想不到的偏差。当然编程人员不喜欢这样的兼容性问题的，而reset.css就是解决默认样式不兼容问题的办法之一。

不是每一个项目都需要重置怎么多的标签，所以最好不要想也不想，把全部的reset复制粘贴到自己的style目录下，应该仔仔细细选择一翻，留下有用的，没用的和不必要的就不要让它们白白增加css的大小了。

1. reset举例。

html, body, div, span, applet, object, iframe,

h1, h2, h3, h4, h5, h6, p,

blockquote, pre, a, abbr, acronym, address, big,

cite, code, del, dfn, em, font, img,

ins, kbd, q, s, samp, small, strike,

strong, sub, sup, tt, var, dl, dt, dd, ol, ul, li,

fieldset, form, label, legend,

table, caption, tbody, tfoot, thead, tr, th, td,

center, u, b, i {

margin: 0;

padding: 0;

border: 0;

outline: 0;

font-weight: normal;

font-style: normal;

font-size: 100%;

font-family: inherit;

vertical-align: baseline

}

body {

line-height: 1

}

:focus {

**outline: 0**

}

ol, ul {

**list-style: none**

}

table {

**border-collapse: collapse;** //设置表格的边框是否被合并为一个单一的边框

border-spacing: 0 //border-spacing 属性设置相邻单元格的边框间的距离（仅用于“边框分离”模式）。

}

blockquote:before, blockquote:after, q:before, q:after {

content: “”

}

blockquote, q {

quotes: “” “”

}

input, textarea {

margin: 0;

padding: 0

}

hr {

margin: 0;

padding: 0;

border: 0;

color: #000;

background-color: #000;

height: 1px

}}

1. **Normalize.css** 是一个可以定制的CSS文件，它让不同的浏览器在渲染网页元素的时候形式更统一。它能：

1.保留了有用的默认值，这个区别于其他的CSS resets

2.修复了浏览器的bug，比如针对html中出现的input类型search，实现代码：

input[type="search"] {

-webkit-appearance: textfield; /\* 1 \*/

-moz-box-sizing: content-box;

-webkit-box-sizing: content-box; /\* 2 \*/

box-sizing: content-box;

}

/\*\*

\* Removes inner padding and search cancel button in S5, Chrome on OS X

\*/

input[type="search"]::-webkit-search-decoration,

input[type="search"]::-webkit-search-cancel-button {

-webkit-appearance: none;

}

###### 跨域携带cookie的问题

**一、CORS跨域发送cookie**

在 Web 页面中可以随意地载入跨域的图片、视频、样式等资源， 但 AJAX 请求通常会被浏览器应用同源安全策略，禁止获取跨域数据，以及限制发送跨域请求。 虽然有多种方法利用资源标签进行跨域，但能够进行的数据交互非常有限。 在 2014 年 W3C 发布了 CORS Recommendation 来允许更方便的跨域资源共享。 **默认情况下浏览器对跨域请求不会携带 Cookie**，但鉴于 Cookie 在身份验证等方面的重要性， CORS 推荐使用额外的响应头字段来允许跨域发送 Cookie（前后端都要做处理）

**1.客户端代码：**

var xhr = new XMLHttpRequest();

xhr.open('GET', url);

xhr.withCredentials = true;

xhr.send();

//如果你在使用 jQuery，可以通过xhrFields来设置：

$.ajax({

url: a\_cross\_domain\_url,

xhrFields: {

withCredentials: true

}

});

**2.服务器端：**

只设置客户端当然是没用的，还需要目标服务器接受你跨域发送的 Cookie，否则会被浏览器的同源策略挡住。服务器同时设置[Access-Control-Allow-Credentials](https://www.w3.org/TR/cors/" \l "access-control-allow-credentials-response-header)响应头为"true"， 即可允许跨域请求携带 Cookie。

除了Access-Control-Allow-Credentials之外，跨域发送 Cookie 还要求 **Access-Control-Allow-Origin不允许使用通配符**，事实上不仅不允许通配符，而且只能指定单一域名，否则，浏览器还是会挡住跨域请求：

###### 设计模式在javascript中的应用

1. **单例模式**