面试题总结

JavaScript （前端玩家必备技能）

1. ele.getAttribute(‘propName’) 和 ele.propName区别

两种方式都是获取元素对象的自定义属性的,上两种方式是两个系统，互不影响。

- ele.propName是基于对象键值对的方式，直接获取存储在元素对象堆内存空间的对应的属性值。
- ele.getAttribute(‘propName’)是获取通过setAttribute给元素对象添加的自定义属性。这种方式添加属性是通过修改页面中HTML标签的结构来完成的，操作DOM会消耗性能。
  以

`e.getAttribute()`

> 使用的DOM的API setAttribute和getAttribute方法操作的`属性元素节点`
> e.getAttribute()，是标准DOM操作文档元素属性的方法，具有通用性可在任意文档上使用，返回元素在源文件中设置的属性
>
> 返回值是源文件中设置的值，类型是字符串或者null
>
> 大部分attribute与property是一一对应关系，修改其中一个会影响另一个，如id，title等属性

`e.propName`

> document.getElementById(el)[name] = value，这是转化成`JS对象的一个属性`。
>
> e.propName通常是在HTML文档中访问特定元素的特性，浏览器解析元素后生成对应对象（如a标签生成HTMLAnchorElement），这些对象的特性会根据特定规则结合属性设置得到，对于没有对应特性的属性，只能使用getAttribute进行访问
>
> 返回值可能是字符串、布尔值、对象、undefined等

2. mouseover和mouseenter的区别

1. over属性滑过 (覆盖)事件，从父元素进入到子元素 ,属于离开了父元素,会触发父元素的out ,触发了元素的over
   enter属于进入 ,从父元素进入子元素，并不算离开父元素 ,不会触发父元素的leave ,触发子元素的enter
2. enter 和leave阻止了事件的冒泡传播 ,而over 和out 还存在冒泡传播的

> 所以对于父元素嵌套子元素的这种情况 . 使用over 会发生很多不愿意操作的事情,此时我们使用enter会更加简单，操作方便，所以真实项目中enter的使用会比over 多

3. 什么是事件代理

> 利用事件的冒泡传播机制,如果一个容器的后代元素当中,很多元素的(`相同的行为`)点击行为(其他行为也是)都要做一些处理,此时我们不需要在像以前一样一个个获取一个个绑定了.我们只需要给容器的clcik绑定方法即可, 这样不管点击的是哪一个后代元素，都会根据冒泡传播的传递机制 ,把容器的click行为触发，把对应的方法执行 ,根据事件源,我们可以知道点击的是谁，从未做不同的事情即可
>
> 基于事件委托绑定轮播图的左右箭头和检点事件
> 事件委托的用处：
>
> - 1.容器中很多后代元素的某个行为要进行操作,委托给容器处理是不错的选择
> - 2.元素是动态绑定的

> - 3.需求是除了某某某,剩下的操作都是干同样的事情 (此时把点击行为的操作委托body,)事件源是某某做什么，不是统一做什么

4. localStorage和cookie的区别，cookie和session的关系！

[概念]

- localStorage：用于浏览器端缓存数据,仅在客户端做保存，不参与服务器端的通信，而且它是永久性保存，除非自己手动清除，否则永久的保存，他存储的大小比cookie要大
- cookie：一般由服务器生成，可以设置失效的时间，如果在浏览器生成，默认是关闭浏览器之后失效，每次都会携带在HTTP头中，如果使用 cookie 保存过多数据会带来性能问题，一般由服务器端生成，用于标识用户身份，cokkie存放大小只有4KB
- session：Session 是存放在服务器端的，类似于Session结构来存放用户数据，当浏览器 第一次发送请求时，服务器自动生成了一个Session和一个Session ID用来唯一标识这个Session，并将其通过响应发送到浏览器.当浏览器第二次发送请求，会将前一次服务器响应中的Session ID放在请求中一并发送到服务器上，服务器从请求中提取出Session ID，并和保存的所有Session ID进行对比，找到这个用户对应的Session。

`cookie和session的比较：`

- 相同点：都存储在客户端（浏览器端）
- 不同点：
  - `存储大小`
    cookie：数据大小不能超过4k。
    sessionStorage和localStorage：虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大。
  - `有效时间`
    cookie：cookie是存在有效期限的，过期后会被删除，过期之前即使窗口或浏览器关闭也一直存在。
    localStorage：永久存储，没有时间限制，除非自己删除。
    sessionStorage：针对会话session数据存储的对象。生命周期是当前窗口。前进后退刷新，数据都存在；关闭窗口，数据被删除。
  - `作用域不同`
    sessionStorage在不同的浏览器窗口中不是共享的，即使是同一个页面；
    localStorage 和cookie在所有同源窗口中都是共享的。
  - `数据与服务器之间的交互方式`
    cookie：数据会自动的传递到服务器，服务器端也可以写cookie到客户端；
    sessionStorage和localStorage：不会自动把数据发给服务器，仅在本地保存。
    另外，Web Storage 支持事件通知机制，可以将数据更新的通知发送给监听者，API接口使用更方便。

`Cookie、session和localStorage的区别`

cookie的内容主要包括：名字、值、过期时间、路径和域。路径与域一起构成cookie的作用范围。若不设置时间，则表示这个cookie的生命期为浏览器会话期间，关闭浏览器窗口，cookie就会消失。这种生命期为浏览器会话期的cookie被称为会话cookie。

会话cookie一般不存储在硬盘而是保存在内存里，当然这个行为并不是规范规定的。若设置了过期时间，浏览器就会把cookie保存到硬盘上，关闭后再打开浏览器这些cookie仍然有效直到超过设定的过期时间。对于保存在内存里的cookie，不同的浏览器有不同的处理方式session机制。
当程序需要为某个客户端的请求创建一个session时，服务器首先检查这个客户端的请求里是否已包含了一个session标识（称为session id），如果已包含则说明以前已经为此客户端创建过session，服务器就按照session id把这个session检索出来使用（检索不到，会新建一个），如果客户端请求不包含session id，则为客户端创建一个session并且生成一个与此session相关联的session id，session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个session id将被在本次响应中返回给客户端保存。保存这个session id的方式可以采用cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发送给服务器。

5. 什么是闭包，你在项目中哪一块用到了闭包！

```
闭包是JS中一个非常重要的机制，我们很多编程思想、业务逻辑、设计模式都是基于闭包完成的，先说一下我对闭包的理解：闭包就是函数执行产生一个私有的作用域（不销毁），在这个作用域中的私有变量和外界互不干扰，而且作用域（栈）不销毁，这些私有变量存储的值也都保存下来了，所有整体来说闭包就是为了保护和保存变量的
实际项目开发中，很多地方使用到了闭包，例如：
1.循环事件绑定，由于事件绑定是异步编程的，我们此时在循环的时候把索引存储起来（可以基于自定义属性存储，也可以基于闭包存储），后期需要使用的时候，向上级作用域查找使用即可
2.平时做业务逻辑的时候，我一般都是基于单例模式来管理代码的，这种单例的构建就应用到了闭包
let xxxRender=(function(){
    return {
        init:function(){

        }
    }
})();
3.我之前在学习资料上了解了柯理化函数思想，它其实也是基于闭包完成的
Function.prototype.bind = function bind(context, ...arg) {
    return () => {
        fn.call(context, ...arg);
    }
};
document.onclick=fn.bind(obj, 10, 20);

还有很多地方也应用了闭包，但是闭包比较占内存，我会尽量减少对它的使用，但是有些需求必须要用
```

6. js中定义函数的方式有哪些，区别是什么！

1. 函数声明
2. 函数表达式
3. 自执行函数
4. new Function()

主要区别：
函数声明 包括申明加定义，函数表达式在变量提升阶段不存在定义的情况，自执行函数主要是创建

一个新的私有作用域，大多是以匿名函数方式存在，且立即自动执行。在这个作用域内声明的变量都是私有变量，避免与全局变量污染。new Function()\是创建一个Function类的实例。

7. 说出你掌握的继承方式及优缺点，并加以改进！

1. 原型继承
   继承方式：

```
function A() {
    this.A = "A";
}
function B() {
    this.B = "B";
}
B.prototype = new A();
```

举例：
B.prototype=new A() A的实例本身具备A的私有属性和公有方法，子类B原型只向他，那么字类B的实例就可以找到这些属性方法了

他和传统后台语言的继承不一样，子类继承父类，并不是把父类的属性方法克隆一份给子类的（这样处理子类和父类就没有直接的关系了）js中的原型继承是让子类和父类建立原型链接的机制，子类的实例调取父类原型上的方法，都是基于原型链的查找机制完成的

存在的问题：子类可以重写父类原型上的方法，子类和父类还有关系的，修改了之后，其他的实例也会受到影响的
如果子类B的原型上之前有属性方法，重写执行A的实例后，之前的方法也都没有了

2. call继承 `fn.call(f)`

```
function A() {
    this.A = "A";
}
function B() {
    //this: B的实例
    A.call(this);
    this.B = "B";
}
```

把父类A作为普通函数执行，让A中的this变为B的实例，相当于给B的实例增加一些属性和方法

存在的问题：把父类A当作普通函数执行，和父类原型没啥关系了，仅仅是把A中的私有属性，变为子类B的私有属性而已，A原型上的公有属性方法和B及他的实例没啥关系

3. 寄生组合继承 `Object.create`
   好处在于我们创建一个没有任何私有属性的空对象，指向A的原型，这样B的共有属性就不会存在A的私有属性了

```
//寄生组合继承（构造函数加原型的方式）
S.prototype=Object.create(F.prototype)//IE6~8不支持
//解决了子类继承父类原型上的属性是,父类的私有属性也会在子类原型上的问题
```

存在的问题：这种方式可以轻易修改父类原型上的东西，(重写太方便了)，这样会导致A的其他实例也会收到影响

```
function A() {
    this.A = "A";
}
function B() {
    A.call(this);
    this.B = "B";
}
B.prototype = Object.create(A.prototype);
```

4. ES6中class类实现继承

```
class A {
    constructor(m) {
        this.x = m;
    }
    getX() {
        console.log(this.x);
    }
}
class B extends A {//=>extends类似于实现了原型继承
    constructor() {
        super('i am a');
        this.y = 200;
    }
    getY() {
        console.log(this.y);
    }
}
let B1 = new B();
```

他是ES6中子新增的方法，类似于call继承，但是又不是，他的这种方法继承，解决了上面的弊端，所以工作中，我们会使用class类继承

8. 说出ES6和ES5的区别！

- let / const

> 和ES5中的VAR的区别
> 1)let不存在变量提升机制（变量不允许在声明之前使用）

2)let不允许重复声明
3)在全局作用域中基于let声明的变量不是window的一个属性，和他没关系
4)typeof 未被声明的变量 =>不是undefined而是报错（暂时性死区）
5)let会形成块级作用域（类似于私有作用域，大部分大括号都会形成块作用域）

- 解构赋值
- “…” 拓展、剩余、展开运算符
- ES6中的模板字符串
- 箭头函数
  和普通函数的区别
  1) 没有arguments，但是可以基于…arg获取实参集合（结果是一个数组）
  2) 没有自己的this，箭头函数中的this是上下文中的this
- Promise（async/await）
- class（ES6中创建类的）
- interator（for of 循环）
- Map / Set
- …..

9. 阐述JS中的同步编程和异步编程，以及你在项目中是如何来使用异步操作的！

同步 ： 在一个线程上（主栈/主任务队列）同一个事件只能做一件事情，当前事情完成才能进行下一个事情（先把一个任务进栈执行 ，执行完成，在把下一个任务进栈，上一个任务出栈），

异步 ：在主栈中执行一个任务 ，但是发现这个任务是一个异步的操作 ，我们会把它移除主栈，放到等待任务队列中（ `此时浏览器会分配其它线程监听异步任务是否到达指定的执行条件（时间）` ），如果主栈执行完成，监听者会把到达条件（时间）的异步任务重新放到主栈中执行。。
[宏任务 ： macro task]

- 定时器（设置的最小等待时间）
- 事件绑定
- ajax
- 回调函数
- node 中 fs 可以进行异步 I/O操作

[微任务 ： micro task]

- process.nextTick
- Promise (async / await)

真实环境的执行顺序

```
sync =>  micro  => macro
```

所以JS中的异步编程仅仅是根据某些机制来管控任务的执行顺序，不存在同时执行两个任务这一说法

==>Pormise不是完成的同步，当在Excutor中执行resolve或者reject的时候，此时是异步的，会先执行then/catch等，当主栈完成后，才会再调用resolve/reject把存放的方法执行

10. 实现一个Promise

```
class Promise {
    constructor(excutorCallBack){
        this.status='pending';
        this.value='undefined';
        //=>实例上挂载两个空容器(计划表)
        this.fulfilledAry=[];
        this.rejectedAry=[];
        //=>执行excutor
        let resolveFn=(result)=> {
            let timer=setTimeout(()=>{//=>用定时器包起来把他们推到异步队列里.等同步完成了在执行下面的代码
                clearTimeout(timer);
                if (this.status !== 'pending') return;
                this.status="fulfilled";
                this.value=result;
                this.fulfilledAry.forEach(item=>{
                    item(this.value);
                });
            });
        };
        let rejectFn=(reason)=>{
            let timer=setTimeout(()=>{
                clearTimeout(timer);
                if(this.status!=='pending') return;
                this.status="rejected";
                this.value=reason;
                this.rejectedAry.forEach(item=>{
                    item(this.value);
                });
            });
        };

        //=>执行excutorCallBack(异常捕获)
        try{
            excutorCallBack(resolveFn,rejectFn);
        }catch (err) {
            //=>有异常信息按照reject状态处理
            rejectFn(err);
        }
    }
    then(fulfilledCallBack,rejectedCallBack){
        //=>处理不传递的状况
        typeof fulfilledCallBack !=='function'?
fulfilledCallBack=result=>result:null;
```

```
        typeof rejectedCallBack !=='function'?rejectedCallBack=reason=>{
            throw new Error(reason instanceof Error? reason.message:reaso
n);
        }:null;

        //=>返回一个新的promise的实例
        return new Promise((resolve,reject)=>{
            this.fulfilledAry.push(()=>{
                try{
                    let x= fulfilledCallBack(this.value);
                    x instanceof Promise? x.then(resolve,reject):resolve
(x);
                }catch(err){
                    reject(err);
                }
            });
            this.rejectedAry.push(()=>{
                try{
                    let x= rejectedCallBack(this.value);
                    x instanceof Promise? x.then(resolve,reject):resolve
(x);
                }catch(err){
                    reject(err);
                }
            });
        });
    }
    catch(rejectedCallBack){
        return this.then(null,rejectedCallBack);
    }

    static all(promiseAry=[]){

        return new Promise((resolve,reject)=>{
            let index=0, //=>记录成功的数量, result :记录成功的结果
                result=[];
            for (let i = 0; i <promiseAry.length; i++) {
               // promiseAry[i] 每个需要处理的promise的实例
                promiseAry[i].then(val=>{
                    console.log(val);
                    index++;
                    //=>索引需要和promiseAry对应上,保证结果的顺序和数组顺序一致
                    result[i]=val;
                    if(index===promiseAry.length){
                        resolve(result);
                    }
                },reject);
            }
        });
    }
}
```

HTTP && AJAX && 跨域 （18+玩家必备技能，初级玩家需要了解一些的）

1. 写出项目中经常用到的性能优化方案

//=>详细七周笔记
content方面

1. 减少HTTP请求：合并文件、CSS精灵、inline Image
2. 减少DNS查询：DNS查询完成之前浏览器不能从这个主机下载任何任何文件。方法：DNS缓存、将资源分布到恰当数量的主机名，平衡并行下载和DNS查询
3. 避免重定向：多余的中间访问
4. 使Ajax可缓存
5. 非必须组件延迟加载
6. 未来所需组件预加载
7. 减少DOM元素数量
8. 将资源放到不同的域下：浏览器同时从一个域下载资源的数目有限，增加域可以提高并行下载量
9. 减少iframe数量
10. 不要404

Server方面

1. 使用CDN
2. 添加Expires或者Cache-Control响应头
   HTTP/1.1 200 OK
   Date: Tue, 08 Jul 2014 05:28:43 GMT
   Server: Apache/2
   Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
   ETag: “40d7-3e3073913b100”
   Accept-Ranges: bytes
   Content-Length: 16599
   Cache-Control: max-age=21600
   Expires: Tue, 08 Jul 2014 11:28:43 GMT
   P3P: policyref=”http://www.w3.org/2001/05/P3P/p3p.xml”
   Content-Type: text/html; charset=iso-8859-1
   {“name”: “qiu”, “age”: 25}
3. 对组件使用Gzip压缩
4. 配置ETag
5. Flush Buffer Early
6. Ajax使用GET进行请求
7. 避免空src的img标签
   Cookie方面
8. 减小cookie大小
9. 引入资源的域名不要包含cookie

css方面

1. 将样式表放到页面顶部
2. 不使用CSS表达式
3. 使用不使用@import
4. 不使用IE的Filter

Javascript方面

1. 将脚本放到页面底部
2. 将javascript和css从外部引入
3. 压缩javascript和css
4. 删除不需要的脚本
5. 减少DOM访问
6. 合理设计事件监听器

图片方面

1. 优化图片：根据实际颜色需要选择色深、压缩
2. 优化css精灵
3. 不要在HTML中拉伸图片
4. 保证favicon.ico小并且可缓存

移动方面

1. 保证组件小于25k
2. Pack Components into a Multipart Document

2. 从浏览器地址栏输入URL到显示页面，中间都经历了什么（尽可能写详细，最好回答出TCP的三次握手和四次挥手，以及浏览器加载页面的细节）

1. 在浏览器地址栏输入URL
2. 浏览器查看缓存，如果请求资源在缓存中并且新鲜，跳转到转码步骤
    1. 如果资源未缓存，发起新请求
    2. 如果已缓存，检验是否足够新鲜，足够新鲜直接提供给客户端，否则与服务器进行验证。
    3. 检验新鲜通常有两个HTTP头进行控制 Expires 和 Cache-Control ：HTTP1.0提供Expires，值为一个绝对时间表示缓存新鲜日期
    HTTP1.1增加了Cache-Control: max-age=,值为以秒为单位的最大新鲜时间
3. 浏览器解析URL获取协议，主机，端口，path
4. 浏览器组装一个HTTP（GET）请求报文
5. 浏览器获取主机ip地址，过程如下：
    1. 浏览器缓存
    2. 本机缓存
    3. hosts文件

    4. 路由器缓存
    5. ISP DNS缓存
    6. DNS递归查询（可能存在负载均衡导致每次IP不一样）

6. 打开一个socket与目标IP地址，端口建立TCP链接，三次握手如下：
    1. 客户端发送一个TCP的SYN=1，Seq=X的包到服务器端口
    2. 服务器发回SYN=1， ACK=X+1， Seq=Y的响应包
    3. 客户端发送ACK=Y+1， Seq=Z

7. TCP链接建立后发送HTTP请求
8. 服务器接受请求并解析，将请求转发到服务程序，如虚拟主机使用HTTP Host头部判断请求的服务程序
9. 服务器检查HTTP请求头是否包含缓存验证信息如果验证缓存新鲜，返回304等对应状态码
10. 处理程序读取完整请求并准备HTTP响应，可能需要查询数据库等操作
11. 服务器将响应报文通过TCP连接发送回浏览器
12. 浏览器接收HTTP响应，然后根据情况选择关闭TCP连接或者保留重用，关闭TCP连接的四次握手如下：
    1. 主动方发送Fin=1， Ack=Z， Seq= X报文
    2. 被动方发送ACK=X+1， Seq=Z报文
    3. 被动方发送Fin=1， ACK=X， Seq=Y报文
    4. 主动方发送ACK=Y， Seq=X报文

13. 浏览器检查响应状态吗：是否为1XX，3XX， 4XX， 5XX，这些情况处理与2XX不同
14. 如果资源可缓存，进行缓存
15. 对响应进行解码（例如gzip压缩）
16. 根据资源类型决定如何处理（假设资源为HTML文档）
17. 解析HTML文档，构件DOM树，下载资源，构造CSSOM树，执行js脚本，这些操作没有严格的先后顺序，以下分别解释
18. 构建DOM树：
    1. Tokenizing：根据HTML规范将字符流解析为标记
    2. Lexing：词法分析将标记转换为对象并定义属性和规则
    3. DOM construction：根据HTML标记关系将对象组成DOM树

19. 解析过程中遇到图片、样式表、js文件，启动下载
20. 构建CSSOM树：
    1. Tokenizing：字符流转换为标记流
    2. Node：根据标记创建节点
    3. CSSOM：节点创建CSSOM树

21. 根据DOM树和CSSOM树构建渲染树:
    1. 从DOM树的根节点遍历所有可见节点，不可见节点包括：
       1） script , meta 这样本身不可见的标签。2)被css隐藏的节点，如display: none
    2. 对每一个可见节点，找到恰当的CSSOM规则并应用
    3. 发布可视节点的内容和计算样式

22. js解析如下：

1. 浏览器创建Document对象并解析HTML，将解析到的元素和文本节点添加到文档中，此时document.readystate为loading
2. HTML解析器遇到没有async和defer的script时，将他们添加到文档中，然后执行行内或外部脚本。这些脚本会同步执行，并且在脚本下载和执行时解析器会暂停。这样就可以用document.write()把文本插入到输入流中。同步脚本经常简单定义函数和注册事件处理程序，他们可以遍历和操作script和他们之前的文档内容
3. 当解析器遇到设置了async属性的script时，开始下载脚本并继续解析文档。脚本会在它下载完成后尽快执行，但是解析器不会停下来等它下载。异步脚本禁止使用document.write()，它们可以访问自己script和之前的文档元素
4. 当文档完成解析，document.readState变成interactive
5. 所有defer脚本会按照在文档出现的顺序执行，延迟脚本能访问完整文档树，禁止使用document.write()
6. 浏览器在Document对象上触发DOMContentLoaded事件
7. 此时文档完全解析完成，浏览器可能还在等待如图片等内容加载，等这些内容完成载入并且所有异步脚本完成载入和执行，document.readState变为complete,window触发load事件

23. 显示页面（HTML解析过程中会逐步显示页面）

三次挥手

服务端的TCP进程先创建传输控制块TCB，准备接受客户端进程的连接请求，然后服务端进程处于LISTEN状态，等待客户端的连接请求，如有，则作出响应。

1、客户端的TCP进程也首先创建传输控制模块TCB，然后向服务端发出连接请求报文段，该报文段首部中的SYN=1，ACK=0，同时选择一个初始序号 seq=i。TCP规定，SYN=1的报文段不能携带数据，但要消耗掉一个序号。这时，TCP客户进程进入SYN—SENT（同步已发送）状态，这是TCP连接的第一次握手。

2、服务端收到客户端发来的请求报文后，如果同意建立连接，则向客户端发送确认。确认报文中的SYN=1，ACK=1，确认号ack=i+1，同时为自己 选择一个初始序号seq=j。同样该报文段也是SYN=1的报文段，不能携带数据，但同样要消耗掉一个序号。这时，TCP服务端进入SYN—RCVD（同 步收到）状态，这是TCP连接的第二次握手。

3、TCP客户端进程收到服务端进程的确认后，还要向服务端给出确认。确认报文段的ACK=1，确认号ack=j+1，而自己的序号为seq=i+1。 TCP的标准规定，ACK报文段可以携带数据，但如果不携带数据则不消耗序号，因此，如果不携带数据，则下一个报文段的序号仍为seq=i+1。这 时，TCP连接已经建立，客户端进入ESTABLISHED（已建立连接）状态。这是TCP连接的第三次握手，可以看出第三次握手客户端已经可以发送携带 数据的报文段了。

当服务端收到确认后，也进入ESTABLISHED（已建立连接）状态。

3. 说出你所熟知的HTTP状态码！GET和POST有啥区别！

`status`：HTTP网络状态码
根据状态能够清楚的反应出当前交互的结果及原因 (每次http事务都靠他分辨bug所在)
常用状态码

200 ok 成功 (只能证明服务器成功返回信息了,但是信息不一定是你业务需要的)
301 Moved Permanently 永久转移(永久重定向)//=>www.360buy.com
=>域名更改 , 访问原始域名重定向到新的域名
302 临时转移 (临时重定向 => 307)
=>网站现在是基于https协议运作的 , 如果访问的是http协议 ,会基于307重定向https协议上
=>302一般用作服务器负载均衡 : 当一台服务器达到最大并发数的时候 ,会把后续访问的用户临时转移到其他的服务器机组上处理
=>偶尔真实项目中会把所有的图片放到单独的服务器上"图片处理服务器" , 这样减少主服务器压力,当用户向主服务器访问图片的时候,主服务器都把它转移到图片服务器上处理 (可以防止盗图)
304 Not Modefied 设置缓存
=>对于不经常更新的资源文件 , 例如:css/JS/HTML/IMG等,服务器会结合客户端设置304缓存,第一次加载过这些资源就缓存到客户端了, 下次在获取的时候,是从缓存中获取; 如果资源更新了, 服务器端会通过最后修改时间来强制让客户端从服务器重新拉取 ; 基于ctrl +F5 强制刷新页面 , 304做的缓存就没有用了

400 Bad Request 请求参数错误
401 无权限访问
404 Not Found 要访问地址不存在
413 Requset Entity Too Large 和服务器交互的 内容资源 超过服务器最大限制

500 Internal Server Error 未知的服务器错误 (服务器宕机断电等服务器出问题)
503 Service Unavailable 服务器超负荷 ()

GET和POST有啥区别

区别一: **[传递给服务器信息的方式不一样]**
get是基于url地址 问号传参 的方式把信息传递给服务器 ,post是基于 请求主体 把信息传递给服务器

```
[get]
xhr.open('get','/temp/list?xxx=xxx&xxx=xxx')

[post]
xhr.send('xxx=xxx & xxx=xxx');
//=>实例
xhr.send(JSON.stringify({id:1000,lx:2000}));//=>请求主体中传递给服务器的是JSON格式的字符,但是真实项目中常用的是urlEncode  格式的字符串 "id=1000&lx=2000"
```

//=>get一般应用于拿 (给服务器的少) ,而post给服务器的很多,如果post是基于问号传参放来来搞会出现一些问题 : URL会拼很长,浏览器对于url的长度有最大限度 (谷歌8kb火狐7KB IE2KB...) , 超过的部分浏览就把它截掉了 =>所以get请求可以基于url传参,而post都是使用请求主体传递 (请求主体理论上是没有限制的 , 但是真实项目中我们会自己做大小限制,防止上传过大的信息 , 导致请求迟迟完不成)

区别二: **[get不安全 , post相对安全]**
因为get是基于"问号传参"把信息传递给服务器的, 容易被骇客进行url劫持,post是基于请求主体传递的 , 相对来说不好被劫持 : 所以登录 , 注册等设计安全性的交互操作, 我们都应该用post请求

区别三: `get会产生不可控制的缓存 ,post不会`
不可控 : 不是想要就要 , 想不要就不要的 ,这是浏览器自主记忆的缓存 , 我们无法基于JS控制 , 真实项目中我们都会把这个缓存干掉 (是根本就不会产生)
get 请求产生缓存是因为 : 连续多次向相同的地址 (并且传递的参数信息也是相同的) ,发送请求,浏览器会把之前获取的数据从缓存中拿到返回 , 导致无法获取服务器最新的数据 (post不会)

4. 什么是HTTP报文，你熟知的报文都有哪些！

在客户端向服务器发送请求，以及服务器把内容响应给客户端时，中间相互传递了很多内容，我们把传递的内容统称为“HTTP报文”。
报文包括：
1）起始行：请求起始行、响应起始行
2）头部：通用头、请求头、响应头
3）主体：请求主体、响应主体

5. 能说下304具体怎样实现吗？

304表示：该资源在上次请求之后没有任何修改，可直接使用浏览器中的缓存版本。
缓存分为强缓存和协商缓存，在做项目时一般用协商缓存，实现304可遵循以下步骤：
1）对于不经常更新的资源文件，例如：css、js、HTML、IMG等，服务器会结合客户端设置304缓存，第一次加载过这些资源就缓存到客户端了。
2）下次再获取时，会首先检查本地缓存，如果本地缓存存在，会获取 获取缓存中文档的最后修改时间（If-Modified-Since），发送请求给Web服务器。
3）web服务器接收到请求，会将服务器的文档修改时间（Last-Modified）: 跟request header 中的If-Modified-Since相比较，如果时间是一样的，说明缓存还是最新的， Web服务器将发送304 Not Modified给浏览器客户端，告诉客户端直接使用缓存里的版本。
4）如果资源更新了，服务器端会通过最后修改时间来强制让客户端从服务器重新拉取。

200 from memory cache 不访问服务器，直接读缓存，从内存中读取缓存。此时的数据时缓存到内存中的，当kill进程后，数据将不存在

200 from disk cache 不访问服务器，直接读缓存，从磁盘中读取缓存，当kill进程时，数据还是存在。

304 Not Modified 访问服务器，发现数据没有更新，服务器返回此状态码。然后从缓存中读取数据。

6. 跨域是什么？http协议中如何判断跨域？如何解决跨域问题？

跨域: 简单的解释就是相同域名,端口相同,协议相同

7. HTTP2具体内容？SDPY了解么？

当HTTPbis小组决定开始制定http2的时候，SPDY已经充分证实了它是一个非常好用的方案。当时已经有人在互联网上成功部署SPDY，并且也有一些文章讨论他的性能。因此，http2便基于SPDY/3草案进行一些修改之后发布了http2的draft-00。
SPDY是由Google牵头开发的协议。他们将其开源，使得每个人都可以参与开发。但很明显，他们通过控制浏览器的实现和享用着优质服务的大量用户来获益。PDY为speedy（单词原意：快速的）

的缩写，读音也就是speedy。SPDY协议已发布过4个草案，分别为版本1、2、3、3.1。目前版本4已在试验阶段，但未发布，Chromium里已有一些针对版本4的代码。

扩展阅读:

```
SPDY对比HTTP的优势：

复用连接，可在一个TCP连接上传送多个资源。应对了TCP慢启动的特性。
请求分优先级，重要的资源优先传送。
HTTP头部数据也被压缩，省流量。
服务器端可主动连接客户端来推送资源（Server Push）。
缺点：

单连接会因TCP线头阻塞（head-of-line blocking）的特性而传输速度受限。加上存在可能丢包的情况，其负面影响已超过压缩头部和优先级控制带来的好处。
由于这些缺点，SPDY在小网站（资源文件数量较少）的效果不明显，有可能比多并发连接更慢。
（由此催生了QUIC）


HTTP/2由标准化组织来制定，是基于SPDY的，差别是：

增加了HTTP/1.1 Upgrade的机制，可在TCP上直接使用HTTP/2，不像SPDY那样必须在TLS上。
HTTPS连接时使用NPN的规范版ALPN（Applcation Layer Protocol Negociation）。
更完善的协议商讨和确认流程。
更完善的Server Push流程。
增加控制帧的种类，并对帧格式考虑得更细致。
有新算法HPACK专门压缩SPDY header block。
HTTP/2文档带有一些示例和详细说明，这是SPDY没有的。
```

8. HTTPS如何实现？tsl/ssl是什么？对称加密、非对称加密在什么时候、对什么数据加密？

`https`

> 在http(超文本传输协议)基础上提出的一种安全的http协议，因此可以称为安全的超文本传输协议。http协议直接放置在TCP协议之上，而https提出在http和TCP中间加上一层加密层。从发送端看，这一层负责把http的内容加密后送到下层的TCP，从接收方看，这一层负责将TCP送来的数据解密还原成http的内容。

`SSL`

> （Secure Socket Layer，安全套接字层），位于可靠的面向连接的网络层协议和应用层协议之间的一种协议层。SSL通过互相认证、使用数字签名确保完整性、使用加密确保私密性，以实现客户端和服务器之间的安全通讯。该协议由两层组成：SSL记录协议和SSL握手协议。

`TLS`

> (Transport Layer Security，传输层安全协议)，用于两个应用程序之间提供保密性和数据完整性。该协议由两层组成：TLS记录协议和TLS握手协议。

`非对称加密`

服务器建立公钥： 每一次启动 `sshd` 服务时，该服务会主动去找 `/etc/ssh/ssh_host*` 的文件，若系统刚刚安装完成时，由于没有这些公钥，因此 `sshd` 会主动去计算出这些需要的公钥，同时也会计算出服务器自己需要的私钥

客户端主动联机请求： 若客户端想要联机到 `ssh` 服务器，则需要使用适当的客户端程序来联机，包括 `ssh`，`putty` 等客户端程序连接

服务器传送公钥给客户端： 接收到客户端的要求后，服务器便将第一个步骤取得的公钥传送给客户端使用 (此时应是明码传送，反正公钥本来就是给大家使用的)

客户端记录并比对服务器的公钥数据及随机计算自己的公私钥： 若客户端第一次连接到此服务器，则会将服务器的公钥记录到客户端的用户家目录内的 `~/.ssh/known_hosts` 。若是已经记录过该服务器的公钥，则客户端会去比对此次接收到的与之前的记录是否有差异。若接受此公钥， 则开始计算客户端自己的公私钥

回传客户端的公钥到服务器端： 用户将自己的公钥传送给服务器。此时服务器：具有服务器的私钥与客户端的公钥，而客户端则是： 具有服务器的公钥以及客户端自己的私钥，你会看到，在此次联机的服务器与客户端的密钥系统 (公钥+私钥) 并不一样，所以才称为非对称加密系统

开始双向加解密： (1)服务器到客户端：服务器传送数据时，拿用户的公钥加密后送出。客户端接收后，用自己的私钥解密

9. DNS劫持是什么？

DNS劫持又称域名劫持，是指在劫持的网络范围内拦截域名解析的请求，分析请求的域名，把审查范围以外的请求放行，否则返回假的IP地址或者什么都不做使请求失去响应，其效果就是对特定的网络不能访问或访问的是假网址

10. 封装一个AJAX库！

> 手写axios的封装

```
~function anonymous(window) {
    //=>设置支持的默认配置项
    let _default = {
        url: '',
        baseURL: '',
        method: 'GET',
        headers: {},
```

```
        dataType: 'JSON',
        data: null,//=>post 系列请求基于请求主体传递给服务器的内容
        params: null,//=>get系列请求基于问号传参传递给服务器的内容
        cache: true
    };


    //=>基于promise设计模式管理ajax请求
    let myAxios = function myAxios(options) {
        //=>options必须融合了: 默认配置信息,用于基于defaluts修改的信息,用户执行get/post方法传递的配置信息 , 越靠后的优先级越高
        let {url, baseURL, method, headers, dataType, data, params, cache} = options;

        //=>把传递的参数进一步处理
        if (/^(GET|DELETE|HEAD|OPTIONS)$/.test(method)){
            //=>get系列
            if(params){
                url+=`${myAxios.check(url)}${myAxios.formatData(params)}`;
            }
            if(cache===false){
                url+=`${myAxios.check(url)}_=${+(new Date())}`
            }
            data=null;//=>get系列就是请求主体什么都不放的
        }else {
            //=>post系列
            if (data){
                data=myAxios.formatData(data);
            }
        }

        //=>基于promise发送ajax
        return new Promise((resolve, reject) => {
            let xhr = new XMLHttpRequest;
            xhr.open(method, `${baseURL}${url}`);
            //=>如果headers存在,我们还需要设置请求头
            if(headers !==null && typeof headers ==='object'){
                for (let attr in headers) {
                    if (headers.hasOwnProperty(attr)) {
                        let value=headers[attr];
                        if(/[\u4e00-\u9fa5]/.test(value)){
                            //=>请求头中不能有中文,我们把他进行编码
                            //encodeURLComponent/decodeURLComponent
                            value=encodeURLComponent(value);
                        }
                        xhr.setRequestHeader(attr,headers[attr]);
                    };
                }
            }
            xhr.onreadystatechange = () => {
                if (xhr.readyState === 4) {
```

```
                    if (/^(2|3)\d{2}$/.test(xhr.status)) {
                        let result = xhr.responseText;
                        dataType = dataType.toUpperCase();
                        result = dataType === 'JSON' ? JSON.parse(result)
 : (dataType === 'XML' ? xhr.responseXML : result);
                        resolve(result,xhr);
                    }
                    console.log(xhr.statusText);
                    reject(xhr.statusText, xhr);
                }
            };
            console.log(data);
            xhr.send(data);
        })

    };

    //=>get  这个处理很重要

    ['get','delete','head','options'].forEach(item=>{
        myAxios[item]=function (url,options={}){
            options={
                ..._default,//=>默认值或者基于defaults修改的值
                ...options,//=>用户调取方法传递配置项
                url:url,//=>请求的url地址 (第一个参数 :默认的配置项和传递的配置项
都不会出现url,只能这样获取)
                method:item.toUpperCase()//=>以后执行肯定是myAxios.head执行,
不会设置method这哥配置项,我们自己需要配置才可以
            };
            return myAxios(options)
        }
    });

    //=>post系列

    ['post','put','patch'].forEach(item=>{
        myAxios[item]=function (url,data={},options={}){
            options={
                ..._default,//=>默认值或者基于defaults修改的值
                ...options,//=>用户调取方法传递配置项
                url:url,//=>请求的url地址 (第一个参数 :默认的配置项和传递的配置项
都不会出现url,只能这样获取)
                method:item.toUpperCase(),//=>以后执行肯定是myAxios.head执
行,不会设置method这哥配置项,我们自己需要配置才可以
                data:data
            };
            return myAxios(options)
        }
    });
    // myAxios.get = function (url, options) {
    //
    // };
```

```
    // myAxios.post = function (url, data, options) {
    //
    // };
    //=>把默认设置暴露出去,后期用户在使用的时候可以自己设置一些基础的默认值 (发送ajax请求按照配置的信息进行处理)
    myAxios.defaults = _default;

    //=>把对象变为urlencoded格式的字符串
    myAxios.formatData=function formatData (obj){
        let str=``;
        for (let attr in obj) {
            if (obj.hasOwnProperty(attr)){
                str+=`${attr}=${obj[attr]}&`;
            }
        }
        return str.substring(0,str.length-1);
    };
    myAxios.check=function check(url){
        return url.indexOf('?')>=-1?'&':'?';
    };
    window.myAxios = myAxios;
}(window);
```