**有关于浏览器缓存：**

强缓存：规定时间内不询问服务器强制使用浏览器缓存。使用响应头的expires（绝对时间：客户端服务器时间不同步会出现用不了缓存现象）字段和cache-control（相对时间：能在有效时间里使用强缓存）字段

协商缓存：if-Modified-Since字段询问服务器是否使用缓存，服务器根据传来的if-Modified-Since时间来判断是否需要使用缓存。，返回304告诉浏览器从缓存中加载资源；

常见的前端缓存：cookie。session。localstorage和sessionstorage

**CDN--解析域名获得相应的CNAME域名-对CNAME域名解析获得缓存服务器的IP地址。--向IP发送请求**

**跨域：jsonp cors机制**

1.cors(cross-origin resource sharing跨域资源共享)：使用自定义的头部让浏览器和服务器进行沟通从而决定请求或响应应该成功还是失败。（Access-Control-Allow-Origin）

缺点：Firefox Chrome 安卓平台的webkit都通过XHR对象实现了对CORS的原生支持，要请求位于另一个域的资源，只需要使用标准的XHR对象，并在open方法传入绝对URL即可。缺点：不能发送和接收cookie，不能使用setRequestHeader()设置自定义头 调用getAllResponseHeaders()总返回空字符串。而CORS又通过preflight requests的机制支持开发者使用自定义头，get/post之外的方法和不同类型的主题内容。而设置withCredentials:true可以允许跨域请求带凭据（cookie http认证等）

2.jsonp:json with padding:前端< script>标签发送跨域请求同时约定好回调函数，后端返回一个字符串模拟js调函数完成跨域。支持浏览器与服务器间的双向通信，简单易用。但是只能处理get请求；存在安全隐患：jsonp是从其他域中加载代码执行，如果其他域不安全，可能会在响应中夹带恶意代码。

3.图像ping:一个网页可以从任何网页加载图像，不存在跨域问题。特点：简单，单向跨域通信。缺点：只能发送get请求，无法访问服务器响应文本。常用在跟踪用户点击页面或动态广告曝光次数。

**window.name传值：打开的浏览器内不同页面间加载后始终存在的一个window对象的属性**

**React是什么？**

react是一个用于视图的js框架；是Facebook打造的，致力于更快的dom渲染和更易于维护的dom层，用于视图的用户交互；很多人认为react是MVC的View

**React Element**

virtual DOM和React Element分别对应是浏览器DOM和HTML元素；virtualDOM不是像HTML一样放在浏览器中而是在内存中的；DOM是由一个个的HTML element组成的，而virtual DOM是由一个个的React element组成的。React element多了之后层层嵌套就形成了DOM树，即virtual DOM。而ReactDOM.render将虚拟DOM转化成真是的DOM。

**对vue生命周期的理解？**

组件创建前/后：

beforeCreate/created

beforeCreate阶段vue实例的挂载元素$el（页面已经存在的一个dom元素）和数据对象data都为undefined，未初始化；

**created阶段**vue实例的data对象有了即完成数据观测，但是$el还不可用；

组件渲染前后 beforeMount/mounted 。

beforeMount阶段，vue的$el和data都已经初始化。vue实例已经完成编译模板把data中的数据和模板生成HTML，但是HTML还没渲染到页面上；

**mounted阶段**vue实例挂载完成，html成功渲染。此过程可以进行ajax交互

数据更新前后 beforeUpdate/updated

销毁前后**beforeDestory**/destoryed 执行destory方法后对data的改变不会再触发周期函数，说明vue实例已经解除了事件监听以及和dom的绑定，但是dom结构依然存在。

beforeDestory:主要用于解绑一些使用addEventListener监听的事件等。

1.什么是vue生命周期？  
答： Vue 实例从创建到销毁的过程，就是生命周期。从开始创建、初始化数据、编译模板、挂载Dom→渲染、更新→渲染、销毁等一系列过程，称之为 Vue 的生命周期。

2.vue生命周期的作用是什么？  
答：它的生命周期中有多个事件钩子，让我们在控制整个Vue实例的过程时更容易形成好的逻辑。

3.vue生命周期总共有几个阶段？  
答：它可以总共分为8个阶段：创建前/后, 载入前/后,更新前/后,销毁前/销毁后。

4.第一次页面加载会触发哪几个钩子？  
答：会触发 下面这几个beforeCreate, created, beforeMount, mounted 。

5.DOM 渲染在 哪个周期中就已经完成？  
答：DOM 渲染在 mounted 中就已经完成了。

Bootstrap常见的面试题：

1.为什么使用bootstrap

bootstrap具有移动设备优先，浏览器支持良好，容易上手和响应式设计的特点。

2.为什么bootstrap栅格系统采用12列？  
原先是16列，但是设计者发现人们更喜欢奇数分割屏幕。12是2.3.4的倍数

3.对于各类尺寸的设备：bootstrap内置了一套响应式、移动设备优先的流式栅格系统随着屏幕设备或者视口viewport尺寸增加，系统自动分为最多12列；

|  |  |  |  |
| --- | --- | --- | --- |
| 手机<768px | 平板≥768px | 台式电脑≥992px | 大型台式电脑≥1200px |
| col-xs- | col-sm- | col-md- | col-lg- |

4.

|  |  |
| --- | --- |
| 设置响应式表格 | class="table-responsive" |
| 创建垂直表单 | 父form添加role="form"  标签和控件放在一个带有class="form-group"的div中  所有表单元素<input><textarea><select>添加class="form-control" |
| 创建水平表单 | 父form添加role="form-horizontal"  标签和控件放在一个带有class="form-group"的div中  所有表单元素<input><textarea><select>添加class="control-label" |
| 元素浮动 | class="pull left" class="pull right" |
| 清除浮动 | class="clearfix" |
| 除了屏幕阅读器外，其他设备隐藏 | class="sr-only" |
| 分页 | 分页：class="pagination"  翻页：class="pager" |
| 超大屏幕 | class="jumbotron".该组件可以增加标题的大小并增加更多的外边距 |

5.为了让bootstrap开发的网站对移动设备友好，确保适当的绘制和触屏缩放，需要在网页的head中添加viewport meta标签（<meta> 标签永远位于 head 元素内部。提供了有关页面的元信息）

<meta name="viewport" content="width=device-width,initial-scale=1">

width控制设备宽度。设置为device-width确保网页正确呈现在不同设备上

initial-scale=1确保网页加载时以1:1的比例呈现不会有任何缩放

通常情况下，maximum-scale=1.0 与 user-scalable=no 一起使用。这样禁用缩放功能后，用户只能滚动屏幕，就能让您的网站看上去更像原生应用的感觉，所以要加上以下代码

补充：

**css3的animation**

1.关键帧：@keyframes:定义动画名以及关键帧（动画在不同阶段的状态）

2.animation:名+时长+何种函数方式播放动画(linear全程一个速度，ease默认，ease-in低速开始...)+播放次数（n,infinite）

**canvas与SVG区别：**

canvas和SVG都允许在浏览器创建图形；

|  |  |
| --- | --- |
| SVG：使用xml描述2D图形的语言  基于xml，则svg DOM中的每个元素都是可用的。可以为某个元素添加js事件处理器  每个被绘制的图形都被视作一个对象，如果svg对象的属性发生变化，浏览器自动重现图形 | canvas通过js绘制2D图形  逐像素渲染  上屏像素化，如果位置发生变化，则需要擦掉重画 |
| * 不依赖分辨率 * 支持事件处理器 * 复杂度高会减慢渲染速度（然和过度使用dom的应用都不快） * 适合带有大型渲染区域的应用程序 * 不适合游戏应用 | * 依赖分辨率 * 不支持事件处理器 * 弱的文本渲染能力 * 能够以png jpg格式保存结果图像 * 适合图像密集型游戏，许多图像会被频繁重绘 |

**浏览器reflow和repaint**

|  |  |
| --- | --- |
| reflow | repaint |
| render Tree的元素因为改变了自身的宽高，布局，显示或隐藏或者元素内部文字结构变化导致需要重新构建页面 | 只是改变了元素的外观风格时如background-color |
| 回流必定触发重绘，重绘不一定触发回流 | |
| 盒子模型相关属性触发重布局 width height padding margin display  定位属性及浮动触发重布局 top left right bottom float clear  改变节点内部文字结构触发重布局 text-align font-size font-weight vertical-align | boder-radius  box-shadow  color  visibility |

注意：渲染树不包含隐藏的节点如display:none的节点，因为其不用于呈现；visibility:hidden隐藏的元素还是包含在渲染树中，因为其会影响布局，会占有空间。

避免回流和重绘：

1.不要使用js对dom设置多条样式，用className代替el.addClass('changeStyle'); .changeStyle{width: 200px;height: 200px;}

2.如果确实需要使用js对dom设置多条样式，可以先将dom display：none，修改--再display:block

3.cssText合并所有修改的样式一次性写入el.style.cssText += "; left: " + left + "px; top: " + top + "px;";

4当前文档之外创建一个子树，修改后再插回去：

var fragment = document.createDocumentFragment();  
var list = document.getElementById("list");

list.appendChild(fragment);

5.transform代替top left margin-left等位移属性

原因：CSS的最终表现分为以下四步：Recalculate Style -> Layout -> Paint Setup and Paint -> Composite Layers（组合层）

 由于transform是位于Composite Layers层，而width、left、margin等则是位于Layout（排布）层，在Layout层发生的改变必定导致Paint Setup and Paint -> Composite Layers，所以相对而言使用transform实现的动画效果肯定比使用改变位置(margin-left等)这些更加流畅。

6. 不要在循环内获取dom 的样式例如：offsetWidth, offsetHeight, clientWidth, clientHeight... 这些浏览器不会马上操作而是缓存在一个栈中，栈满了会一次性触发所有样式的更改且刷新这个栈，如果多次获取就会让browser不断刷新这个栈导致回流增多

|  |  |  |
| --- | --- | --- |
| clientWidth | offsetWidth | scrollWidth |
| width+padding | width+padding+border+（滚动条） | 获取指定标签内容层的真实宽度（可视区域宽度+被隐藏区域宽度） |

有关opacity待定；

**ECMA**：European Computer Manufacture's Association欧洲计算机制造商协会。ECMAScript是一个重要的标准，一个完整的JavaScript实现由核心ECMAScript文档对象模型DOM和浏览器对象模型BOM。ECMAScript可以为不同种类的宿主环境提供核心的脚本编程能力，web浏览器就是一种宿主环境。它仅仅是一个描述，定义了脚本语言的所有属性方法和对象。其他的语言可以实现ECMAScript来作为功能的基准，JavaScript就是这样。

**闭包：**

**前言：**执行环境与作用域：执行环境定义了变量和函数有权访问的其他数据，决定了他们各自的行为。全局执行环境是最外围的执行环境，在web浏览器中全局执行环境被认为是window对象。**某个执行环境中的代码全部执行完毕后，环境被销毁，保存在其中的变量和函数定义也随之销毁，全局执行环境直到应用程序退出--例如browser关闭时才被销毁。**

作用域链查找：当我们想知道某个变量的值，首先在当前层即当前执行环境查找，没有则父级查找一直到全局执行环境，仍没有返回undefined。

作用域链的前端总是当前执行代码所在环境的变量对象，下一个变量对象来自包含（外部）环境，...最后一个对象是全局执行环境的变量对象。作用域链实际上是一个指向变量对象的指针列表，它只引用但不实际包含变量对象。

块级作用域：js没有。如C或Java中定义if(true){var color="blue"}if执行完后color就会被销毁，而js不存在这种块级作用域。同样，for语句初始化变量的表达式所定义的变量，在for循环执行完毕后也依旧存在于循环外部的执行环境中。

模仿块级作用域：匿名函数

|  |  |
| --- | --- |
| (function(){  //块级作用域  })(); | var someF=function(){  //块级作用域  };  someF(); |

闭包就是可以访问另一个函数作用域中的变量的函数

闭包特点：1.读取另一个函数作用域中的变量；2.外部函数和闭包中的整个变量对象会一直保存在内存中即使函数执行完毕。

优点：变量长期储存在内存中；避免全局变量的污染；私有成员的存在

缺点：闭包会携带外部（包含它的）函数的作用域，因此比其他函数占用更多内存。长时间不清除会引起内存泄露。

解决弊端：不使用的变量及时清除；

注意：闭包只能取得包含函数中任何变量的最后一个值（详见for循环）

**ES5为数组定义的迭代方法(数组项的值，索引，数组对象本身)**

|  |  |
| --- | --- |
| arr.filter() | 给数组每一项运行给定函数。返回该函数会返回true的项组成的数组 |
| arr.forEach() | 给数组每一项运行给定函数。没返回值 |
| arr.map() | 给数组每一项运行给定函数。返回每次函数调用的结果组成的数组 |
| arr.every() arr.some() | 给数组每一项运行给定函数。如果该函数对**任**一项都返回true则返回true  给数组每一项运行给定函数。如果该函数对**某**一项都返回true则返回true |

**缩小数组方法：**

arr.reduce()从数组第一项开始逐个遍历到最后 arr.reduceRight()从数组最后一项开始逐个遍历到第一项

**arr.sort()**

默认情况下，sort()按升序排列数组项，比较的永远是字符串，因为其会调用每个数组项的toString ()方法比较得到的字符串。因而有时得到的结果和数字排序并不一样。因此它接受一个函数作为参数：每次从数组中（倒序）挑选两个数，如果返回0，1则两个数不交换位置 返回小于0的数则交换位置。

**HTTP请求方法合集**

|  |  |  |  |
| --- | --- | --- | --- |
| http1.0 | get | 获取资源 | 请求数据放在url中，限制在2k左右 |
| head | 获取报文首部 | 和get相同，只是返回时不返回主体只返回响应的头 |
| post | 传输实体文本 | 传输数据放在请求体中，数量不限 |
| http1.1 | put | 传输文件，然后保存在服务器指定位置 |  |
| delete | 让服务器删除某个资源 |  |
| trace | 追踪路径，回显服务器收到的请求 | 主要用于测试判断 |
| options | 询问服务器可以提交哪些请求方法 |  |
| connection | 要求用隧道协议进行TCP通信 | 主要使用SSL和TLS（传输层安全）协议把通信协议加密后经网络隧道传输 |

**http消息报头:普通报头实体报头请求报头响应报头**

cache-control/accept/host/authorizaiton/content-type/content-encoding/expires/user-agent/server/location/www-authenticate

**js观察者模式**

观察者模式又叫发布订阅模式，（publish/subscribe），它让多个观察者对象同时监听某一个主题对象，这个主题对象状态发生变化时就会通知所有的观察者对象，使他们自动更新自己。适用于：当一个对象改变需要同时改变其他对象，并且他不知道具体多少对象需要改变时

使用观察者模式的好处：

1. 支持简单的广播通信，自动通知所有已经订阅过的对象。
2. 页面载入后目标对象很容易与观察者存在一种动态关联，增加了灵活性。
3. 目标对象与观察者之间的抽象耦合关系能够单独扩展以及重用。

**http与https：**

http缺点：

1.通信使用不加密明文，内容可能被窃听

2.无法验证通信方身份，可能遭遇伪装

3.无法证明接收报文的完整性，可能已经遭遇篡改。

针对1：

通信的加密，http没有加密机制但可以通过和SSL（secure socket layer）或TLS(transport layer security)的组合使用加密http通信内容；与SSL组合使用的HTTP就叫HTTPS（http secure）超文本传输安全协议。（SSL建立安全通信线路后就可以在这条线路上进行http通信）

针对2：

SSL不仅提供加密处理还使用了第三方颁发的证书用于确定通信方身份

针对3：

SSL提供认证和加密处理以及摘要功能保证完整性。

https常用场景：登录页面；购物车清算。

|  |  |
| --- | --- |
| http | https |
| 直接和TCP通信 | http-ssl-tcp |
|  | 通信慢，消耗CPU和内存资源导致处理速度变慢（必须进行加密处理，服务端和客户端都需要进行加密解密的运算处理。） |
|  | 需要购买证书，小网站并不必要。 |

将网站从http--https

申请CA证书--配置服务器配置文件，开放443端口--项目里引用的js和css相对路径从原先的http转换为https--(配置CDN)

**有关node**

node的全局对象：global（node所处的全局环境，类似于浏览器的window对象），process（node所处的当前进程，允许开发者与该进程互动）console（node内置的console模块，提供命令行环境的标准输入标准输出）。

全局函数：定时器函数： setTimeout clearTimeout setInterval clearInterval

require:加载模块；

Buffer()：操作二进制数据

全局变量:\_\_filename 当前运行脚本文件名 \_\_dirname所在的目录

**node特点：**高并发，所有IO请求都是异步的。不擅长CPU密集型的操作。

**node如何实现高并发？**

传统的网络服务技术是每新增一个请求就生成一个新线程，这样多个线程会占用系统内存，而node是单线程，使用非阻塞的异步IO调用，所有连接都由该线程处理，在libuv的加分下，可以将数万个连接挂到该线程的事件循环中

**JS运行机制**

**js事件循环机制分为浏览器和node事件循环机制。**（以下是浏览器）

同步任务

异步任务

同步任务在主线程上执行形成执行栈--异步任务有了结果就向任务队列添加一个事件---执行栈中同步任务执行完后读取任务队列--任务队列进入执行栈--重复上面一步

线程中：事件循环唯一，但是任务队列有多个

除了广义的同步任务和异步任务还有更精细的定义：

宏任务：整体代码，setTimeout setInterval setImmediate I/O

微任务：process,nextTick promise

js执行优先级：同步代码（promise构造函数立即执行属于同步代码）--任务队列所有的微任务（process.nextTick>promise.then）--其中的一个任务队列--所有微任务--其中一个任务队列

注意：js运行机制：在第一次事件循环中：遇到宏任务先执行宏任务将它放入任务队列，执行回调函数时从微任务里拿回调函数再从（已经是下次循环了）宏任务开始找到其中一个任务队列执行完毕再执行微任务以此类推

**flex：**

flex: *flex-grow* *flex-shrink* *flex-basis*|auto|initial|inherit;

相对于其他flex项目进行扩展的量；相对于..收缩的量；项目长度

flex:1的子元素代表让所有弹性盒模型对象的子元素都有相同的长度，且忽略它们内部的内容

**promise的特点：**

1、对象的状态不受外界影响。

Promise对象有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。

只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。

这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

2、一旦状态改变，就不会再变，任何时候都可以得到这个结果。

Promise对象的状态改变，只有两种可能：

从pending变为fulfilled和从pending变为rejected。

只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）。

**问题一：axios、fetch与ajax有什么区别？**

主要区别是 axios、fetch请求后都支持Promise对象API，ajax只能用回调函数

Axios本质上也是对原生XHR的封装，只不过它是Promise的实现版本，符合最新的ES规范，它有以下几条特性：

* 从 node.js 创建 http 请求
* 支持 Promise API
* 客户端支持防止CSRF
* 提供了一些并发请求的接口（重要，方便了很多的操作）

**有关vue-cli下的跨域问题：**

通过node.js代理服务器来实现跨域：在vue-cli项目中的config文件夹下的index.js文件里配置dev对象的proxyTable对象

**axios特点**

1、Axios 是一个基于 promise 的 HTTP 库，支持promise所有的API  
2、它可以拦截请求和响应  
3、它可以转换请求数据和响应数据，并对响应回来的内容自动转换成 JSON类型的数据  
4、安全性更高，客户端支持防御 XSRF

vue 的spa

**单页Web应用（single page web application，SPA）：**

就是只有一张Web页面的应用。单页应用程序 (SPA) 是加载单个HTML 页面并在用户与应用程序交互时动态更新该页面的Web应用程序。浏览器一开始会加载必需的HTML、CSS和JavaScript，所有的操作都在这张页面上完成，都由JavaScript来控制。因此，对单页应用来说模块化的开发和设计显得相当重要。

**单页Web应用的优点：**

1、提供了更加吸引人的用户体验：具有桌面应用的即时性、网站的可移植性和可访问性。

2、单页应用的内容的改变不需要重新加载整个页面，web应用更具响应性和更令人着迷。

3、单页应用没有页面之间的切换，就不会出现“白屏现象”,也不会出现假死并有“闪烁”现象

4、单页应用相对服务器压力小，服务器只用出数据就可以，不用管展示逻辑和页面合成，吞吐能力会提高几倍。

5、良好的前后端分离。后端不再负责模板渲染、输出页面工作，后端API通用化，即同一套后端程序代码，不用修改就可以用于Web界面、手机、平板等多种客户端。

**单页Web应用的缺点：**

1、首次加载耗时比较多。

2、SEO问题，不利于百度，360等搜索引擎收录。

3、容易造成Css命名冲突。

4、前进、后退、地址栏、书签等，都需要程序进行管理，页面的复杂度很高，需要一定的技能水平和开发成本高。

**为什么要按需加载？**

在于一次性的加载所有功能对应的代码，但其实用户每一阶段只可能使用其中一部分功能。

**如何按需加载**

* 把整个网站划分成一个个小功能，再按照每个功能的相关程度把它们分成几类。
* 把每一类合并为一个 Chunk，按需加载对应的 Chunk。
* 对于用户首次打开你的网站时需要看到的画面所对应的功能，不要对它们做按需加载，而是放到执行入口所在的 Chunk 中，以降低用户能感知的网页加载时间。
* 对于个别依赖大量代码的功能点，例如依赖 Chart.js 去画图表、依赖 flv.js 去播放视频的功能点，可再对其进行按需加载。

import 返回一个 Promise，当文件加载成功时可以在 Promise 的  then  方法中获取到  show.js  导出的内容

 /\* webpackChunkName: "show" \*/  的含义是为动态生成的 Chunk 赋予一个名称，以方便我们追踪和调试代码。 如果不指定动态生成的 Chunk 的名称，默认名称将会是  [id].js。 /\* webpackChunkName: "show" \*/  是在 Webpack3 中引入的新特性，在 Webpack3 之前是无法为动态生成的 Chunk 赋予名称的。

分离css:引入-实例化，同时在下面的plugins数组里添加-rules里面配置把刚刚处理css的用plugin.extract()方法包起来-

**展示型组件和容器型组件**

展示组件

- 只关心它们的样子。

- 可能同时包含子级容器组件和展示组件，一般含DOM标签和自定的样式。

- 通常用this.props.children来包含其他组件

- 不依赖app其它组件，比如flux的actions和stores

- 不会定义数据如何读取，如何改变

- 只通过this.props接受数据和回调函数

- 很少有自己的状态变量，即使有，也是UI的状态变量，比如toggleMenuOpen,InputFocus

- 一般是函数级组件，除非它们需要状态，lifecycle hooks，优化处理。

容器组件

- 只关心它们的运作方式。

- 可能同时包含子级容器组件和展示组件，但大都不含DOM标签，而含他们自己所用的wrapping div，从不用自己的样式。

- 为展示组件或其他组件提供数据和方法。

- 调用Flux的actions，并且将其作为展示组件的回调函数。

- 维持许多状态变量，通常充当一个数据源。

- 通常由高阶组件生成，比如Redux里的connect()，Relay里的createContainer()，Flux Utils里的Container.create()，而非手工写出（译者：可能在meteor中数据是例外吧）

- 例子有UserPage, FollowersSidebar, StoryContainer, FollowedUserList。

# 如何提高页面加载速度

1.减少http请求（图片地图 css雪碧图，特别是图片比较多的网站如果能用css sprites降低图片数量带来的将是速度的提升）

2.使用CDN

3.添加expires头

4.压缩组件

5.将样式表放在头部，希望页面尽快显示内容，为用户提供可视化的反馈

6.将脚本放在底部。因为js的下载和执行会阻塞DOM树的构建（严谨说是中断了dom树的更新）

7.使用外部的js和CSS，当脚本或样式是从外部引入的文件，浏览器有可能缓存它们，从而以后加载的时候能够直接使用缓存；

8.减少DNS查找

9.精简js和css，删除重复脚本

10.避免重定向

重定向：用于用户从一个URL重新路由到另一个URL

常用重定向的类型：

301：永久重定向（主要用于网站的域名发生变更后）

302：临时重定向（主要实现post请求后告知浏览器转移到新的URL）

304：not modified（主要用于当浏览器在其缓存中保留了组件的一个副本，同时组件已经过期了，浏览器就会生成一个get请求，如果服务器的组件没有被修改过就会返回304状态码，同时不携带主体，告知浏览器可以重用这个副本）

损伤性能：页面重定向后，延迟整个HTML文档的传输，在HTML文档到达之前页面不会显示任何东西，也没有任何组件被下载；

重定向的应用场景：跟踪内部流量：当拥有一个门户主页时，同时想对用户离开主页后的流量进行跟踪，这时可以使用重定向；跟踪出站流量（?）

11.在进行ajax请求的时候选择尽量使用get方法，这样可以使用客户端的缓存，提高请求速度。（因为post的请求是不可以在客户端缓存的，每次请求都需要发送给服务器进行处理）

# 前端优化方案

优化的目的：

用户--页面加载更快

服务商角度--减少页面请求数或者减少请求所占的带宽能够节省可观的资源

一页面级别的优化：

1.减少HTTP请求数（1）设计实现层面简化页面（2）合理设置http缓存比如：原则很简单，能缓存越多越好，能缓存越久越好。例如，很少变化的图片资源可以直接通过 HTTP Header中的Expires设置一个很长的过期头 ;变化不频繁而又可能会变的资源可以使用 Last-Modifed来做请求验证。尽可能的让资源能够在缓存中待得更久（3）资源合并与压缩（4）css sprites

2.外部脚本置底：让脚本内容在页面信息内容加载后再加载

3.css放在head中：放在body中时有可能还没开始解析CSS就开始渲染页面了，用户体验差

4.减少不必要的http跳转

5.避免重复的资源请求。

二代码级优化

1.减少DOM操作，避免回流和重绘，减少作用域链查找（如果在循环中需要访问非本作用域下的变量时请在遍历之前用局部变量缓存该变量，并在遍历结束后再重写那个变量，这一点对全局变量尤其重要，因为全局变量处于作用域链的最顶端，访问时的查找次数是最多的。）

2.css选择器

3.图片压缩

# 如何提高页面加载速度

1.减少http请求（图片地图 css雪碧图，特别是图片比较多的网站如果能用css sprites降低图片数量带来的将是速度的提升）

2.使用CDN

3.添加expires头（对于很少变化的图片资源）

4.压缩组件

5.将样式表放在头部，希望页面尽快显示内容，为用户提供可视化的反馈

6.将脚本放在底部。因为js的下载和执行会阻塞DOM树的构建（严谨说是中断了dom树的更新）

7.使用外部的js和CSS，当脚本或样式是从外部引入的文件，浏览器有可能缓存它们，从而以后加载的时候能够直接使用缓存；

8.减少DNS查找

9.精简js和css，删除重复脚本

10.避免重定向

重定向：用于用户从一个URL重新路由到另一个URL

常用重定向的类型：

301：永久重定向（主要用于网站的域名发生变更后）

302：临时重定向（主要实现post请求后告知浏览器转移到新的URL）

304：not modified（主要用于当浏览器在其缓存中保留了组件的一个副本，同时组件已经过期了，浏览器就会生成一个get请求，如果服务器的组件没有被修改过就会返回304状态码，同时不携带主体，告知浏览器可以重用这个副本）

损伤性能：页面重定向后，延迟整个HTML文档的传输，在HTML文档到达之前页面不会显示任何东西，也没有任何组件被下载；

重定向的应用场景：跟踪内部流量：当拥有一个门户主页时，同时想对用户离开主页后的流量进行跟踪，这时可以使用重定向；跟踪出站流量（?）

11.在进行ajax请求的时候选择尽量使用get方法，这样可以使用客户端的缓存，提高请求速度。（因为post的请求是不可以在客户端缓存的，每次请求都需要发送给服务器进行处理）

# JS创建对象的方法

|  |  |  |
| --- | --- | --- |
| 工厂模式factory | 解决了创建多个相似对象的问题 | 无法识别对象即无法知道一个对象的类型 |
| 构造函数模式constructor | 可以将构造函数的实例标识成某种特定的类型 | 构造函数中的每个方法都要在实例上重新创建一遍。但是这个方法最后又不是同一个函数的实例，因为在ECMAScript中，每定义一个函数就是实例化了一个对象。 |
| 原型模式（prototype） | 解决了构造函数中每个方法都要在实例上重新创建一遍的问题 | 所有实例共享原型上的属性 |
| 组合使用构造函数模式和原型模式 | 可以向构造函数传递参数，并且每个实例都有自己的一份实例属性的副本是用来定义引用类型的默认模式 | 暂无 |
| 动态原型（dynamic prototype）模式 | 动态定义原型中的方法，创建的对象可以确定它的类型 | 不能使用对象字面量重写原型，不然会切断现有实例和新原型的联系 |
| 寄生构造函数模式（parasitic） | 仅仅封装构造对象的代码返回新的构建对象，类似构造函数。（和工厂模式代码一样只是调用的时候用new调用）该模式下构造函数返回的对象与构造函数外部创建的对象没区别 | 无法确定创建的对象的类型 |
| 稳妥构建函数模式（durable constructor） | 没有公共属性，新创建对象的实例方法不引用this  不使用new操作符调用构造函数  安全性得到保障 | 只能调用其定义的方法，没有其他方法访问其数据成员 |

注：对象的constructor属性可以用来标识对象类型

# JS实现继承：

|  |  |  |
| --- | --- | --- |
| 构造函数继承 | call apply | 无法继承父类原型上的属性和方法； |
| 原型链继承 | sub.prototype=new parent(); | 引用类型的原型属性会被共享，一个实例向该引用类型里添加数据 。另一个实例的这个属性也会跟着变化。 |
| 组合继承 | 构造函数实现对实例属性的继承  原型链实现对原型属性和方法的继承 | 两次调用父类的构造函数 |
| 原型式继承 | Object.create() | 引用类型的属性被所有实例共享  原型式继承是浅拷贝，所以引用类型的数据共享在不同实例中 |
| 寄生式继承 | 思路和构造函数和工厂模式类似。创建一个封装继承过程的函数。函数内部以某种方式增强对象 | 每次创建对象都会创建一遍方法 |
| 寄生组合式继承 | 不必为了指定子类型的原型而调用父类的构造函数。  让子类型的原型对象等于父类型原型对象的副本 | 寄生式继承和组合式继承混合；规避组合式继承的调用两次构造函数的问题，但是比较繁琐 |