搞不懂设计模式?

看这个就会了

---

1. css 高度塌陷，怎么去解决

```
子元素设置浮动以后，子元素会完全脱离文档流，导致父元素高度塌陷

解决方案
1. 给父元素一个固定的值 （只是表面好了）
2. overflow设置为hidden
3. 在高度塌陷的父元素最后，设置一个空白的div，对其进行清除浮动
4. 通过after伪类，添加一个空白的块元素，然后清除浮动
```

2. 继承

什么是对象

> 对象是对单个事物的抽象，对象是一个容器，封装了属性（状态）和方法（行为）

原型

> 在javascript中，函数可以有属性，每个函数都有一个特殊的属性：原型（prototype），属性值是一个对象

> 在prototype中，默认存在一个constructor属性，属性值就是当前函数自身

> 原型对象的所有属性和方法都能被实例共享

原型链

> 在javascript中规定，所有对象都有自己的原型对象，原型对象也是对象，所以他也有自己的原型，就形成了原型链

构造函数

> 用来专门生成实例对象的函数，它是对象的模板，描述对象的基本结构

构造函数的特点

- 函数体内使用this关键字，代表所要生成的实例对象
- 生成对象的时候，必须使用new关键字
- 构造函数的首字母大写

new命令

> 执行构造函数，返回一个实例对象

new命令的原理

- 创建一个空对象，作为将要返回的对象实例
- 将这个空对象的原型，指向构造函数的prototype属性
- 将这个空对象的赋值给函数内部关键字this
- 执行构造函数内部代码

Es5继承

- 原型链继承

> 让新实例的原型对象等于父类的实例

优点：
子类可以继承父元素所有的属性和方法
缺点：
子类无法向父类传参
父类原型链上的属性被多个实例共享，造成一个实例修改了原型，其他的也会变

- 使用构造函数继承

> 用.call和.apply将父类的构造函数引入子类函数

优点：
原型链引用值独立，不再被所有的实例共享
子类可以向父类传参
缺点：
构造函数只能继承父类实例的属性和方法，不能继承父类的原型

- 组合继承

> 结合二种模式的优点，传参和复用

- 原型式继承
- 寄生式继承
- 寄生组合式继承

es6继承

> 主要使用extends关键字实现继承，利用class配合extentds和super

```
class A{
    constructor(){

    }
}

class B extends A{
    constructor(){
        super()
    }
}
```

3. 基本数据类型和内置对象

- js数据类型分为基本数据类型和引用数据类型
- 基本数据类型分为 undefined null boolean number string
- 引用数据类型为object
- js内置对象，包含boolean string number array function date math object regexp error global

4. webpack中loader和plugin区别、

loader
loader 让 webpack 能够去处理那些非 **JavaScript** 文件（webpack 自身只理解 **JavaScript**）。

loader 可以将所有类型的文件转换为 webpack 能够处理的有效模块,然后你就可以利用 webpack 的打包能力,对它们进行处理。

本质上,webpack loader 将所有类型的文件,转换为应用程序的依赖图（和最终的 **bundle**）**可以直接引用的模块。**

plugin

loader 被用于转换某些类型的模块,而插件则可以用于执行范围更广的任务。

插件的范围包括,从打包优化和压缩,一直到重新定义环境中的变量。插件接口功能极其强大,可以用来处理各种各样的任务。

常用的loader

- babel-loader 把ES6转为ES5
- eslint-loader 检查代码格式
- style-loader css-loader less-loader
- file-loader url-loader

常用的plugin

- html-webpack-plugin 根据模板自动生成html代码，并且自动引用css、js
- clear-webpack-plugin 打包之前将指定 文件夹清空
- uglifyjs-webpack-plugin：压缩js代码

5. 怎么使用chunk做代码分割

```
optimization: {
  splitChunks: {
    chunks: "all"
  }
},
```

6. vue $set

Q: 当生成vue实例，再次给数据赋值，有时候数据并没有更新视图，是因为受到es5的限制，vue不能检测到对象属性的添加或者删除，vue在初始化实例的时候将属性转换为getter/setter，

使用$set，让其有getter/setter

Vue.set()是将set函数绑定在Vue的构造函数上，this.$set是将set函数绑定在Vue原型上

7. node中怎么处理文件 fs模块
8. node高并发

- 只有一个主线程执行程序代码
- 主线程之外，维护了一个事件队列
- 主线程代码执行完毕，通过event loop，也就是事件循环机制。开始从event queue的开头去除第一个事件，从线程池分配一个线程去执行这个事件，然后去取第二个。主线程不断的检查事件队列中是否有未执行的事件，直到事件队列都执行完毕。
- 不断重复第三步

9. 白鹭引擎
10. http几种请求方式

```
1. get
get请求指定的页面信息，返回实体主体
2.head
类似get请求，返回的响应没有具体的内容，用户获取报头
3.post
数据包含在请求体中
4.put
从客户端向服务器传送的数据取带指定的文档的内容
5.delete
请求服务器删除指定的页面
6.connect
7.options
准许客户端查看服务器的性能
8.trace
回显服务器收到的请求，主要用于测试或诊断
```

11. restful特点

- 每一个url代表1中资源
- 客户端使用get、post、put、delete4个镖师操作方式的动词对服务器资源进行操作，get用来获取资源，post用来新建资源（也可以是更新）、put用来更新资源，delete用来删除资源
- 用过操作资源的表现形式来操作资源
- 资源的表现形式是xml或者html
- 客户端与服务器端之前的交互在请求之间是无状态的，从客户端到服务器端的每个请求都必须包含理解请求所必须的信息

12. 有哪些块元素行内元素，有什么区别

- 块元素

```
h标签1-6
p标签
ul
ol
dl
table
form
div
 1.总是从新的一行开始
 2.高度、宽度都是可控的
 3.宽度没有设置时，默认为100%
 4.块级元素中可以包含块级元素和行内元素
```

- 行内元素

span
a
br
b
strong
i
select
  1.和其他元素都在一行
  2.高度、宽度以及内边距都是不可控的
  3.宽高就是内容的高度，不可以改变
  4.行内元素只能行内元素，不能包含块级元素

13. 讲一下Promise、async/await、Generator

Promise的写法只是回调函数的改进，用then()方法免去了嵌套，更为直观。

最优秀的解决方案是什么呢?
就是**async/await**
讲**async**前我们先讲讲协程与Generator

协程(coroutine)，意思是多个线程相互协作，完成异步任务。

协程遇到**yield**命令就会暂停，把执行权交给其他协程，等到执行权返回继续往后执行。最大的优点就是代码写法和同步操作几乎没有差别，只是多了**yield**命令。

Generator是协程在ES6的实现，最大的特点就是可以交出函数的执行权，懂得退让。

从回调函数，到Promise对象，再到Generator函数，JavaScript异步编程解决方案历程可谓辛酸，终于到了Async/**await**。很多人认为它是异步操作的最终解决方案(谢天谢地，这下不用再学新的解决方案了吧)

其实**async**函数就是Generator函数的语法糖，

async函数的优点
（1）内置执行器
Generator 函数的执行必须靠执行器，所以才有了 co 函数库，而 **async** 函数自带执行器。也就是说，**async** 函数的执行，与普通函数一模一样，只要一行。

（2） 语义化更好
async 和 **await**，比起星号和 **yield**，语义更清楚了。**async** 是“异步”的简写，而 **await** 可以认为是 **async** wait 的简写。所以应该很好理解 **async** 用于申明一个 function 是异步的，而 **await** 用于等待一个异步方法执行完成。

（3）更广的适用性
yield 命令后面只能是 Thunk 函数或 Promise 对象，而 **async** 函数的 **await** 命令后面，可以跟 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。

14. 浏览器的事件循环机制

基本概念

> 1 同步任务 2异步任务 3宏任务 4微任务 5任务队列

事件循环可以简单的描述为 主代码块-微任务-宏任务-微任务（主代码块也是一个宏任务）

详细解释

- 主代码块入栈
- 顺序执行同步任务，遇到异步任务交浏览器内核的模块处理，处理完将异步任务回调函数加入到任务队列中
- 函数执行栈为空时，如果任务队列中微任务加入到任务栈，函数执行栈为空，在从任务队列取宏任务入栈
- 执行2-3，知道所有的任务执行完

15. 手写防抖节流

- 函数防抖

> 再次触发会清除掉旧的定制器，重新计时

```
function debounce (fn, time) {
    let timer
    return funtion (){
        if (timer) {
            clearTimeout(timer)
        }
        timer = setTimeout(fn, time)
    }
}
```

- 函数节流

> 函数在执行一次以后，在这次执行完毕之前，不会再次触发

```
 let flag = true
function throttle (fn, time) {
    return function (){
        if (!flag) {
            return false
        }
        flag = false
        setTimeout(()=>{
            fn()
            flag = true
        }, time)

    }
}
```

16. jsonp解决跨域为什么用script而不用image

> 新图像元素只要设置了src属性就会开始下载,优点是很明显的：兼容性非常好，缺点就是：只能发生GET请求，而且无法获取响应文本。

> JSONP的优势在于：可以能够直接访问响应文本，支持在浏览器和服务器之间的双向通信

17. 如何判断一个属性是自身属性还是原型属性

```
hasOwnProperty用于检查给定的属性是否存在当前实例对象中，而不是原型中
function (obj,pro) {
    return !obj.hasOwnProperty(pro) && pro in object
}
```

18. csrf是什么?

> 跨站请求攻击，【攻击者盗了用户的身份。发送恶意请求】

- 如何防御?

【1】 提交验证码 【2】 refer check 【3】 token验证

19. 手写个事件订阅

```
class EventEmitter{
    constructor(){
        this.subs = Object.create(null)
    }
    // 注册
    $on(eventType,handler){
        this.subs[eventType] = this.subs[eventType]|| []
        this.subs[eventType].push(handler)
    }
    // 触发
    $emit(eventType,handler){
        if (this.subs[eventType){
            this.subs[eventType].foreach(handler=>{
                handler()
            })
        }
    }
}
```

20. webpack优化

- 优化loader，使用include缩小文件搜索范围
- 使用DLLPlugin 【动态链接库】

> 大量复用模块的动态链接库只需编译一次

- 使用happyPack

```
loader对文件转换操作分配给多个进程去并行处理
```

每日一遍，多学多现

21. node模块查找规则

- require('./find')

```
先找同名js文件，再找同名文件夹
如果找到文件夹，再找文件夹下的index.js
如果没有index.js,那就在find文件夹下的package.js查找main的入口文件
如果指定的入口文件不存在或者没有指定入口文件，就会报错
```

- require('find')

```
假设是系统模块
会在node_modules下找
xxx 【和上面的保持一致】
```

22. async/defer/preload

> defer/sync都告诉浏览器，可以在后台加载脚本的同时解析html，并在脚本加载完以后执行，这样，脚本下载就不用阻碍dom的构建和渲染，用户在所有脚本加载完成之前可以看到页面

> defer比async要先引入浏览器，他的执行在解析完成以后才开始执行，它处在DOMContentLoaded事件之前，

> async脚本在他们下载完成以后就开始执行，有可能会阻断dom的构建，通常设置了async的文件优先级较低

> preload有较高的优先级，告诉浏览器尽快的加载他们

23. 手写一个instanceof

```
fucntion _instanceof(A, B){
    if (!A || !B) {
        return false
    }

    let O = B.prototype
    A = A.__proto__
    while(1){
        if (A === null){
            return false
        } else if (A === O){
            return true
        } else {
            // 接着往上找
            A = A.__proto__
        }
    }
}
```

24. promise的错误是怎么捕获的

> return promise

25. promise.all参数是什么，返回的是什么

> 参数是一个数组，返回的也是一个数组【传入的数组是多个promise实例】

> 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。

26. 如果promise.all传入的不是promise，会报错吗

> 调用Promise.resolve方法，将参数转为 Promise 实例，再进一步处理

27. 哪些操作会引起重排

- 页面初始化
- 浏览器窗口改变尺寸
- 元素位置、尺寸变化
- 添加或者删除元素

28. 数组都有哪些方法，foreach和map的区别

```
concat() 合并数组
join() 使用分隔符，将数组变成字符串返回
pop() 删除最后一位，并返回删除的数据
shift() 删除第一位，返回删除的数据
push()
reverse() 反转数组
slice() 截取指定位置的数组，并返回
sort() 排序
splice() 删除指定位置并替换，返回删除的数据
foreach() 参数为回调函数，接受3个参数，value、index、self，没有返回值
map() 回调函数返回数据，组成新数组由map返回
```

29. es6新特性

```
1. let const
2. 模板字符串
3. 解构赋值
4. for...of  / for..in
5. 展开运算符
const a= [1,2,3,4,5]
...a
6. 箭头函数
7. super和extends
```

30. 浏览器缓存

- 强缓存
- 协商缓存

深入理解浏览器缓存

31. json-server和mock怎么选择

> JsonServer 主要是搭建本地的数据接口，创建json文件，便于调试调用

> mock主要是随机生成js数据

32. 逻辑题：写出一个函数，判断字符串是否是由2个子字符串merge而成

> 如‘a12bc3d4’是由'abcd' '1234'合并而成

```
function isMerge(a,b,c){
    if (a.length !== b.length+c.length){
        return false
    }
    var bIndex = 0
    var cIndex = 0
    for (var x = 0; x< a.length;x++){
        if (a[x] == b[bIndex]){
            bIndex++
        } else if (a[x] == c[cIndex]){
            cIndex++
        } else {
            break
        }
    }
    return (b.length == bIndex) && (c.length == cIndex)
}
```

33. 介绍一下this

> this的指向在函数定义的时候是确定不了的，只有在函数执行的时候才能确定this到底指向谁，实际上this的最终指向就是调用它的对象

- 例1

```
function a(){
    var user = 张三;
    console.log(this.user); //undefined
    console.log(this); //Window
}
a(); // 等同于window.a
```

- 例2

```
var o = {
    user:"张三",
    fn:function(){
        console.log(this.user);  //张三，this指向的就o
    }
}
o.fn();
```

- 例3

```
var o = {
    a:10,
    b:{
        a:12,
        fn:function(){
            console.log(this.a); //undefined
            console.log(this); //window
        }
    }
}
var j = o.b.fn;
j();// this指向的是window
```

- 例4

```
function fn()
{
    this.user = '张三';
    return {};
}
var a = new fn;
console.log(a.user); //undefined

---

function fn()
{
    this.user = '张三';
    return function(){};
}
var a = new fn;
console.log(a.user); //undefined

---

function fn()
{
    this.user = '张三';
    return 1;
}
var a = new fn;
console.log(a.user); //张三

---

function fn()
{
    this.user = '张三';
    return undefined;
}
var a = new fn;
console.log(a.user); //张三

// 如果函数内部return的是一个对象，this就指向返回的那个对象，如果返回值不是一个对象那就指向函数的实例
```

34. 高阶组件

> 一个函数接受一个组件为参数，返回一个包装后的组件

35. 手写一个观察者

```
class Dep{
    constructor(){
        this.subs = []
    }

    addSub(sub){
        if (sub && sub.update){
            this.subs.push(sub)
        }
    }

    notify(){
        this.subs.forEach((sub)=>{
            sub.update()
        })
    }
}

class Watcher(){
    constructor(){

    }

    update(){

    }
}

let dep = new Dep()
let watcher = new Wathcher()
dep.addSub(watcher)
dep.notify()
```

36. 描述一下策略模式

> 对象有某个行为，但是在不同的场景下，该行为有不同的实现算法

37. JavaScript基本设计原则和常见的设计模式

设计原则

- 单一职责【只做一件事】
- 最少知识职责【尽可能的避免和其他实体的交互】
- 开放-封闭原则【可以扩展，不可修改】

设计模式

- 单例模式

> 确保只有一个实例，并提供全局访问

- 策略模式

> 将算法的使用和算法的实现分离

- 代理模式

> 提供一个替身对象来控制对某个对象的访问

- 迭代器模式
- 发布-订阅模式

> 也称作观察者模式，定义了对象间的一种一对多的依赖关系，当一个对象的状态发 生改变时，所有依赖于它的对象都将得到通知

- 命令模式
- 组合模式
- 模板方法模式
- 享元模式
- 职责链模式
- 中介者模式
- 装饰者模式
- 状态模式
- 适配器模式
- 外观模式

38. 数组去重

es6方法去重

```
let a = [1,2,3,4,4,4,4,5,5,5]
a = new Set(a)
```

es5,for循环，splice去重

```
    function upique(arr){
        for (var x =0;x<arr.length;x++) {
            for (var j = x + 1; j<arr.length;j++){
                if (arr[x] == arr[j]){
                    arr.splice(j,1)
                    j--
                }
            }
        }
        return arr
    }
    console.log(upique([1,2,2,2,3,3,3,4]))
```

使用filter去重

```
    function upique1(arr){
        return arr.filter((item,index,arr)=>{
            return arr.indexOf(item, 0) === index
        })
    }

    console.log(upique1([1,2,2,2,3,3,3,4]))
```

39. 手写一个promise.all

```
Promise.prototype.all = function(iterator){
    let promises = Array.from(iterator)
    let len = promises.length
    let conunt = 0
    let resultList = []
    return new Promise((resolve,reject)=>{
        promises.forEach((p,index)=>{
            Promise.resolve(p).then((result)=>{
                conunt++
                resultList[index]=result
                if (conunt == len) {
                    resolve(resultList)
                }
            })
        })
    }).catch(e=>{
        reject(e)
    })
}
```

40. 多种方法实现数组排序

sort

```
let arr = [1,7,4,2,9]
arr.sort(function(a,b){
    return a-b
})
console.warn(arr)
```

冒泡排序

```
let arr = [1,7,4,2,9]
function maopao(arr){
    for(var x= 1; x< arr.length;x++){
        for (var y= 0;y<arr.length-x;y++){
            if (arr[y] > arr[y+1]){
                var temp = arr[y]
                arr[y] = arr[y+1]
                arr[y+1] = temp
            }
        }
    }
    return arr
}
console.warn(maopao(arr))
```

选择排序

```
function xuanze(arr){
    for (var x =0;x<arr.length;x++){
        let minIndex = x
        for (var y = x +1; y<arr.length; y++){
            if (arr[minIndex]> arr[y]){
                minIndex = y
            }
        }
        let temp = arr[x]
        arr[x] = arr[minIndex]
        arr[minIndex] = temp
    }
    return arr
}
    console.warn(xuanze(arr))
```

41. ajax、axios、fetch的区别

ajax

> 传统 Ajax 指的是 XMLHttpRequest（XHR）， 最早出现的发送后端请求技术，隶属于原始js中，核心使用XMLHttpRequest对象，多个请求之间如果有先后关系的话，就会出现回调地狱。

axios

> axios 是一个基于Promise 用于浏览器和 nodejs 的 HTTP 客户端，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，它本身具有以下特征：

```
1.从浏览器中创建 XMLHttpRequest
2.支持 Promise API
3.客户端支持防止CSRF
4.提供了一些并发请求的接口（重要，方便了很多的操作）
5.从 node.js 创建 http 请求
6.拦截请求和响应
7.转换请求和响应数据
8.取消请求
9.自动转换JSON数据
```

fetch

> fetch号称是AJAX的替代品，是在ES6出现的，使用了ES6中的promise对象。Fetch是基于promise设计的。Fetch的代码结构比起ajax简单多了，参数有点像jQuery ajax。但是，一定记住fetch不是ajax的进一步封装，而是原生js，没有使用XMLHttpRequest对象。

42. webpack devServer实现原理，热更新是怎么实现的?

devServer

> 原理其实就是启动一个express服务器，调用app.static方法。

热更新

> 通过建立websocket实现服务端和客户端的双向通讯，当我们的服务端发生变化时可以通知客户端进行页面的刷新。 实现的方式主要有两种iframe mode和inline mode。

43. 高阶组件和mixin的区别

> 场景：多个页面需要增加权限控制功能

44. 前端性能优化有哪些?

原则

1. 多使用内存、缓存
2. 减少cpu的计算，减少网络请求
3. 减少IO操作（硬盘读写）

加载资源优化

1. 静态资源的合并和压缩
2. 静态资源缓存
3. 使用cdn让资源加载更快

```
Q: 为什么使用cdn可以让资源加载更快?

A:  CDN指的是内容分发网络。可以理解为就近原则
```

渲染优化

1. css放在head中，js放在body后
2. 图片懒加载
3. 减少dom操作
4. 事件节流
5. 非核心代码异步加载

```
defer/async/
```

45. webpack构建原理

1. 初始化参数

> 从配置文件和shell语句中读取与合并参数，得到最终的参数

2. 开始编译

> 从上一个得到的参数初始化Compiler参数，加载所有配置的插件，执行对象的run方法开始编译

3. 确定入口

> 根据配置中的entry找出所有的入口文件

4. 编译模块

> 从入口文件出发，调用所有配置的loader对模块进行编译，再找出改模块依赖的模块，再递归本步骤知道所有入口依赖的文件都处理完毕

5. 完成模块编译

> 使用loader编译完所有的模块后，得到了每个模块被编译后的最终内容以及他们的依赖关系

6. 输出资源

> 根据入口和模块之间的依赖关系，组装成一个个包含多个模块的chunk，再把每一个chunk转换成一个单独的文件加入到输出列表，

7. 输出完毕

> 在确定好输出内容以后，根据配置确定好输出的路径和文件名，把文件内容写入到文件系统

46. babel原理

> babel的转译过程也分为三个阶段：parsing、transforming、generating，

例: ES6-ES5

> ES6代码输入 ==》 babylon进行解析 》 得到AST》 plugin用babel-traverse对AST树进行遍历转译 》 得到新的AST树》 用babel-generator通过AST树生成ES5代码

47. 观察者模式VS发布订阅模式

> 所谓观察者模式，其实就是为了实现松耦合(loosely coupled)。

> 发布订阅模式里，发布者和订阅者，不是松耦合，而是完全解耦的。

从表面看

- 观察者模式里，只有两个角色 —— 观察者 + 被观察者
- 而发布订阅模式里，却不仅仅只有发布者和订阅者两个角色，还有一个经常被我们忽略的 —— 经纪人Broker

更深层次

- 观察者和被观察者，是松耦合的关系
- 发布者和订阅者，则完全不存在耦合

从使用上来看

- 观察者模式，多用于单个应用内部
- 发布订阅模式，则更多的是一种跨应用的模式(cross-application pattern)，比如我们常用的消息中间件

48. Javascript面向对象

基本特征

1. 封装

> 把客观的事物封装成抽象的类，并且这些类可以吧自己的数据和方法只让可信的类或者对象操作，

2. 继承

> 通过继承创建的新类称为"子类"或"派生类"。继承的过程，就是从一般到特殊的过程

3. 多态

> 对象的多功能，多方法， 一个方法多种表现形式。同一个方法,面对不同的对象有不同的表现形式就叫做多态。

49. 高阶函数、纯函数、函数式编程

- 高阶函数

> 把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

- 纯函数

> 简单来说，一个函数的返回结果只依赖于它的参数，并且在执行过程里面没有副作用，我们就把这个函数叫做纯函数

- 函数式编程

> 函数式编程（通常缩写为FP）是通过编写纯函数，避免共享状态、可变数据、副作用 来构建软件的过程

50. 老生常谈：在地址栏输入网址敲回车发生了什么

1.我们在浏览器中输入域名，通过DNS解析出ip地址

> dns是因特网的一项核心服务，它作为可以将域名和ip地址相互映射的一个分布式数据库

具体解析流程

> 用户发起请求->操作系统把域名发送给本地区的域名服务器->没有->到Root Server的域名服务器请求解析->返回一个主域名（.com）的服务器地址地址->本地的域名服务器再向主域名服务器发起请求->返回Name Server域名服务器地址（jianshu.com）->接下来的解析就由域名提供商的服务器来解析->Name Server域名服务器查询存储的域名和ip的映射关系表->返回ip地址和一个过期时间，根据这个时间缓存到本地，解析结束。

2.通过解析出来的ip与服务器进行连接（三次握手）

> 第一次：客户端向服务器端发送一个连接请求等待服务器确认（第一次握手由浏览器发起，告诉服务器我要发送请求） 第二次：服务器端收到请求并确认在回复一个指令（第二次握手由服务器发起，告诉浏览器我准备接收了，你发送吧；） 第三次：客户端收到服务器的回复并确认返回（第三次握手由浏览器发起，告诉服务器，我马上发送，准备接收；） 通过三次握手建立了客户端与服务器端之间的连接，现在可以请求和发送数据的请求了

3.发送http请求

4.服务器返回一个http请求浏览器接受响应

5.浏览器拿到响应文本后开始渲染

1. 根据 HTML 解析出 DOM 树；

根据 HTML 的内容，将标签按照结构解析成为 DOM 树 DOM
树解析的过程是一个深度优先遍历。即先构建当前节点的所有子节点，再构建下一个兄弟节点。 在读取 HTML 文档，构建 DOM
树的过程中，若遇到 script 标签，则 DOM 树的构建会暂停，直至脚本执行完毕。*

2. 根据 CSS 解析生成 CSS 规则树；

解析 CSS 规则树时 **js** 执行将暂停，直至 CSS 规则树就绪。 浏览器在 CSS 规则树生成之前不会进行渲染。*

3. 结合 DOM 树和 CSS 规则树，生成渲染树

DOM 树和 CSS 规则树全部准备好了以后，浏览器才会开始构建渲染树。 精简 CSS 并可以加快 CSS
规则树的构建，从而加快页面相应速度。*

4. 根据渲染树计算每一个节点的信息

布局：通过渲染树中渲染对象的信息，计算出每一个渲染对象的位置和尺寸
回流：在布局完成后，发现了某个部分发生了变化影响了布局，那就需要倒回去重新渲染。*

5. 根据计算好的信息绘制页面

绘制阶段，系统会遍历呈现树，并调用呈现器的“paint”方法，将呈现器的内容显示在屏幕上。
重绘：某个元素的背景颜色，文字颜色等，不影响元素周围或内部布局的属性，将只会引起浏览器的重绘。
回流：某个元素的尺寸发生了变化，则需重新计算渲染树，重新渲染。*

6.数据传输完毕断开连接（四次挥手）

> 第一次挥手：由浏览器发起，发送给服务器，我请求报文发送完了，你准备关闭吧； 第二次挥手：由服务器发起，告诉浏览器，我接收完请求报文，我准备关闭，你也准备吧； 第三次挥手：由服务器发起，告诉浏览器，我响应报文发送完毕，你准备关闭吧； 第四次挥手：由浏览器发起，告诉服务器，我响应报文接收完毕，我准备关闭，你也准备吧；