**浏览器工作原理**

**一、进程和线程**

进程是资源分配的最小单位，线程是CPU调度的最小单位

**（一）进程**

一个进程就是一个程序的运行实例，  
每启动一个应用程序，操作系统都会为此程序创建一块内存，用来存放代码、数据数据、一个执行任务的主线程，我们把这样的一个运行环境叫进程。

一个进程关闭，操作系统则会回收为该进程分配的内存空间

**（二）线程**

线程是依附于进程的，而进程中使用多线程并行处理能提升运算效率。

进程与线程的之间的关系： （进程是火车，线程是每节车厢）

1. 进程中的某一线程执行出错，都会导致整个进程的崩溃
2. 线程之间共享进程中的公共数据。
3. 当一个进程关闭之后，操作系统会回收进程所占用的内存。
4. 进程之间的内容相互隔离

**二、浏览器渲染流程**

**第一步，解析：主线程开始解析HTML**

1. 浏览器收到HTML，HTML解析器开始解析HTML，生成DOM Tree，并保存在浏览器内存中  
   -- 同时开启一个预解析线程，用来分析 HTML 文件中包含的Javascript、 CSS 、Img等资源，通知网络进程提前加载这些资源
2. 解析遇到CSS(style、行内、link)，CSS解析器开始对CSS进行解析，生成CSSOM（ 即styleSheets）

* 样式计算：（css样式的继承、层叠等规则）
* 转换样式中的属性值，如color: red; => color: rgb(255, 0, 0)
* 计算出DOM每个节点的具体样式

1. 遇到 <script> ，渲染线程停止解析剩余的 HTML 文档，等待Javascript 资源加载，Javascript引擎执行脚本完成后，HTML再继续解析

**JavaScript 脚本是依赖样式表的，会先等CSS文件加载并解析完成再执行，因此Javascript对元素的样式是最终生效的**

**javascript 会阻塞HTML解析和页面渲染**

**css解析和HTML解析并行，不会阻塞HTML解析,但是会阻塞页面渲染（但是Javascript执行，会导致CSS的解析增加HTML解析的时间）**

**第二步，生成Layout Tree(布局树）**

根据DOM和styleSheets生成LayoutTree布局树（渲染树），所有不可见的元素会被忽略，如head标签 , display:none的元素，script标签等

**第三步，布局计算**

* 渲染引擎计算出布局树中各元素的几何位置，并将计算结果保存在布局树中，
* 布局阶段的输出就是我们常说的盒子模型，它会精确地捕获每个元素在屏幕内的确切位置与大小

**第四步，分层，生成图层树**

渲染引擎根据布局树生成图层树，

**第五步， 绘制**

* 主线程根据图层树生成绘制列表，交给合成线程
* 合成线程对图层进行分割，生成大小固定的图块
* 合成线程按照视口附近的图块来优先交给GPU进程

**第六步，光栅化，生成位图**

GPU进程根据不同图块生成位图，还给合成线程

**第七步，合成**

* 合成线程收到各图块位图之后，发出合成命令，交给浏览器主进程

**第八步，显示界面**

* 浏览器主进程然后进行界面显示

渲染流程中的特殊情况：

**1. 重排（回流）：**

指修改了元素几何属性，如位置、尺寸、内容、结构等变化，引发元素几何位置变化，浏览器需要重新计算样式、构建布局树，开始之后的一系列子阶段，这个过程就叫重排。

**重排需要更新完整的渲染流水线，所以开销也是最大的。**

触发重排的情况：（Javascript操作DOM，引发不同渲染流水线重新工作）

* 添加或删除可见的DOM元素
* 元素位置改变
* 元素尺寸改变
* 元素内容改变
* 改变字体大小会引发回流
* 页面渲染器初始化
* 浏览器窗口大小发生改变
* 当获取一些属性时，浏览器为了获得正确的值也会触发回流，这样使得浏览器优化无效，包括  
  (1) offset(Top/Left/Width/Height)  
  (2) scroll(Top/Left/Width/Height)  
  (3) cilent(Top/Left/Width/Height)  
  (4) width,height  
  (5) 调用了getComputedStyle()或者IE的currentStyle

**2. 重绘：**

指修改了元素的外观样式，不会引起几何位置变化，直接入绘制阶段，生成绘制列表，然后执行之后的一系列子阶段，这个过程就叫重绘。如背景颜色、边框颜色，文字颜色等

**重绘省去了布局和分层阶段，所以执行效率会比重排操作要高一些。重排必然带来重绘，但是重绘未必带来重排**

**3. 直接合成：**

指更改一个既不要布局也不要绘制的属性，直接分割图块阶段，然后交给浏览器主进程并不线上显示，这个过程叫做直接合成。  
如 transform:translate(100px, 100px)

相对于重绘和重排，直接合成能大大提升效率

减少重排（回流）、重绘, 方法：

* 多次dom 操作合成一次，批量操作，例如 createDocumentFragment，vue框架虚拟DOM和diff算法
* 使用 class 操作样式，而不是频繁操作 style
* 处理动画时，使用will-change和transform 做优化  
  在css中使用will-change，渲染引擎会将该元素单独生成一个图层

**三、JavaScript执行机制**

**（一）JavaScript代码执行流程**

**第一步，代码编译：JavaScript 引擎对代码进行编译，并保存在内存中**

编译结果为两部分：执行上下文、可执行代码

showName()；//函数showName被执行

console.log(myname)；//undefined

var myname = '小白'

function showName() {

console.log('我是小白');

}

编译时的执行上下文如下：（变量环境部分）

{

showName: xxx, //showName 函数在堆内存的引用地址

myname: undefined

}

可执行上下文如下：

showName()；

console.log(myname)；//undefined

myname = '小白'

* 执行上下文：是 JavaScript 执行一段代码时的运行环境  
  每个执行上下文包含以下几个部分：
  1. 变量环境
  2. 词法环境
  3. 外部环境，即当前执行上下文中变量的外部引用，用来指向外部的执行上下文，也称为 outer
  4. this，this的指向在于当前函数的调用方式  
     -直接调用指向全局对象window （严格模式下则是undefined）  
     -通过对象调用，this指向该对象  
     -通过apply、call、bind等方法调用则指向第一个参数对象  
     -箭头函数中的this指向外层函数的this（解析箭头函数不会创建执行上下文）

let userInfo = {

userName: "小白",

age: 18,

sayHello: function () {

setTimeout(function () {

console.log(`${this.userName}，你好`) //undefined

}, 100)

}

}

userInfo.sayHello()

修改一个函数this指向的方法：

* 缓存外部的this, 如 var \_this = this;
* 使用箭头函数
* 使用app、call、bind改变this指向

**第二步，执行可执行代码**

问题：

1. var变量提升  
   编译时变量声明提升，并初始化值为undefind，
2. 函数声明提升

* 同时声明了多个相同名字的函数，后声明的会覆盖前面声明的函数
* 函数声明的优先级高于变量提升，变量名和函数声明的名字相同时，采用函数名

解决： 引入let、const、块级作用域

**（二）函数执行（调用）过程**

1. 执行上下文栈：  
   用来管理执行上下文，后进先出

* 全局执行上下文：执行全局代码生成一个全局执行上下文，仅有一个，伴随页面的整个生存周期
* 函数执行上下文：执行每个函数会生成一个函数执行上下文，可以有多个， 当函数执行结束，该函数的执行上下文会被销毁

**（三）作用域、作用域链、闭包**

**1. 作用域：是指变量和函数可以被访问的范围**

* 全局作用域：代码中任何地方都能被访问，即全局执行上下文中的变量和函数能在任何地方被访问，生命周期伴随着页面的生命周期。
* 函数作用域：函数内部定义的变量或函数只能在函数内部被访问，函数执行结束之后，函数内部定义的变量会随着函数执行上下文一起销毁（闭包除外）
* 块级作用域 { }

var 、 let、const的区别：

1. var：  
   -- 在javascript解析时， 声明和初始化提升，声明之前访问不报错，值为undefined；  
   -- 存放在执行上下文中的变量环境中  
   -- 可以多次声明同一个变量，后一个值会覆盖之前的值；  
   -- 不支持块级作用域
2. let ：  
   -- 用来声明一个变量，在解析时，声明会提升，但是初始化不会提升，声明之前访问报错；  
   -- 存放在执行上下中的词法环境中  
   -- 同一作用域内不能多次声明；  
   -- 支持块级作用域
3. const ：  
   -- 用来声明一个常量，不能再次修改  
   --声明会提升，但是初始化不会提升，声明之前访问报错；  
   -- 存放在执行上下中的词法环境中  
   -- 同一作用域内不能多次声明；  
   -- 支持块级作用域

function foo(){

var a = 1

let b = 2

{

let b = 3

var c = 4

let d = 5

console.log(a); //1

console.log(b); //3

}

console.log(b) ;//2

console.log(c); //4

console.log(d); //报错：d is not defined

}

foo()

**2. 作用域链：变量查找沿着各作用域一层层向外部引用指向的执行上下文查找，形成一个链条，即作用域链条**

函数的作用域由词法作用域决定  
词法作用域：是指作用域是函数声明的位置来决定的，和函数怎么调用无关

**3. 闭包：**

当函数执行完毕时，函数体内的定义的变量会随着函数执行上下文立即销毁，但是当外部函数包含内部函数，且内部函数使用了外部函数中定义的变量，这些变量就不会销毁，仍然保存在内存，这些变量和内部函数就形成了闭包

闭包的形成条件：

1. 外部函数里有内部函数
2. 内部函数中使用了外部函数中定义的变量

function foo() {

var myName = "小白";

var age = 18;

function sayHello(){

console.log (`你好，我的名字是：${myName}，今年${age}`)

}

return sayHello;

}

let hello = foo();

hello()

// myName和age就是foo函数的闭包

* 闭包形成原因：  
  Javascript在代码编译阶段，遇到内部函数 时，JavaScript 引擎会对内部函数做一次快速的词法扫描，  
  发现该内部函数引用了外部函数定义的变量，于是在堆空间创建换一个“closure”的对象，用来保存内部函数使用的变量，这个closure对象就是闭包
* 闭包何时回收？
  1. 引用闭包的函数是全局变量时，闭包则会一直保存在内存中，直到页面关闭
  2. 引用闭包的内部函是局部变量时，内部函数执行结束后，内部函数就会立即销毁，下次JavaScript 引擎的执行垃圾回收时，判断不再使用，则销毁闭包，回收内存

问题：内存泄露（ 该回收的内存未被及时回收 ）

**（四）Javascrip的垃圾回收机制**

**1. Javascript的内存机制**

* 栈内存： 存储基本类型数据（调用栈，执行上下文栈）  
  **变量是引用类型时，存储的是引用类型的引用地址（编号）**
* 堆内存：存储引用类型数据
* 代码空间：存储可执行代码

**2. Javascript的垃圾回收机制**

数据被使用之后，不再需要了，就称为垃圾数据，垃圾数据要及时销毁，释放内存空间，否则会内存泄漏。

* 手动回收，如设置变量为null
* 自动回收

（1）栈内存回收

当Javascript代码执行时，记录当前执行状态的指针（称为 ESP），指向当前执行上下文的指针，当前函数代码之前完毕，指针下移指向下一个要执行的函数执行上下文，当前执行上下文弹出调用栈进行销毁，这个过程就是该函数栈内存回收的过程

function foo(){

var a = 1

var b = {name:"极客邦"}

function showName(){

var c = 2

var d = {name:"极客时间"}

}

showName()

}

foo()

（2）堆内存回收  
垃圾回收器：

* 主垃圾回收器： 负责回收生存时间长的垃圾数据（老生代垃圾数据）
* 副垃圾回收器：负责回收生存时间短的垃圾数据（新生代垃圾数据）

第一步，标记堆内存中活动对象和非活动对象

* 活动对象：还在使用的数据
* 非活动对象：垃圾数据

第二步，回收非活动数据所占据的内存  
在所有的标记完成之后，统一清理内存中所有被标记为可回收的对象

第三步，做内存整理

**（五）浏览器的事件循环机制**

每个渲染进程都有一个**主线程**，处理以下事件：

* 渲染事件（如解析 DOM、计算布局、绘制）
* 用户交互事件（如鼠标点击、滚动页面、放大缩小等）
* JavaScript 脚本执行事件
* 网络请求完成、文件读写完成事件

**消息队列和循环机制保证了页面有条不紊地运行**

**四、浏览器中的页面**

页面的生命周期：

* 加载阶段
* 更新阶段（交互阶段）
* 销毁阶段

**（一）页面优化：**

从页面的生命周期方向思考：

**1. 加载阶段：如何让页面渲染快？**

关键资源（核心资源）：阻塞页面首次渲染的资源称为页面的关键资源，HTML、CSS、Javascript

* 减少关键资源个数，减少请求次数
* 减小关键资源大小，提高资源加载速度
* 传输关键资源需要多少个 RTT（Round Trip Time）  
  --TCP协议传输资源时，是将资源分成一个个数据包（一般为14KB 左右），来回多次进行传输  
  --RTT ，是指客户端开始发送数据开始，到收到服务器端接收确认信息所经历的时间

**具体优化方法**：

（1）压缩HTML文件，移除 不必要注释  
（2）合并并压缩CSS 、JavaScript等文件 ，script 标签加上 async 或 defer属性  
（3）避免使用table布局  
（4）缓存（第二次请求命中缓存则直接读取缓存）

**2. 更新阶段（交互阶段）：通过Javascript操作DOM时，页面再次渲染速度如何更快？**

目标是减少页面渲染过程的重排、重绘

**具体优化方法**：

（1）减少DOM操作，将多次操作DOM合并为一次，如插入元素节点  
（2）减少逐项更改样式，最好一次性更改style，或者将样式定义为class并一次性更新  
（3）前端框架Vue、React（虚拟DOM和Diff算法等）  
（3）避免多次读取offset等属性，使用变量做缓存  
（4）防抖、节流  
（5）做动画效果时，使用will-change和transform 做优化

**（二）虚拟DOM及算法**

1. 多次

**1. 页面加载阶段：**

* 首次加载时，先创建虚拟DOM树，
* 再根据虚拟DOM树创建真实的DOM树，然后继续一系列渲染流水线工作

**2. 页面加载阶段：**

* 如果数据发生了改变，再创建一棵新的虚拟DOM树
* 两棵虚拟DOM树对比，计算出最少变化
* 把所有变化记录一次性更新到真实DOM树上，然后继续一系列渲染流水线工作

引入虚拟DOM树执行流程.png

**五、浏览器中安全**

**同源策略：协议、域名、端口三者都相同则称为同源**

**1. XSS 攻击：跨站脚本攻击（Cross Site Scripting）**

XSS 攻击是指黑客往 HTML 文件中或者 DOM 中注入恶意 JavaScript 脚本，在用户浏览页面用户实施攻击的一种手段

**（1）风险：**

* **窃取用户Cookie信息**  
  -- 通过document.cookie获取用户Cookie 信息，发送到恶意服务器  
  -- 恶意服务器拿到用户的 Cookie 信息之后，就可以模拟用户的登录，进行转账等操作
* **监听用户行为**  
  -- 通过addEventListener来监听键盘事件，获取用户账号、密码、信用卡等信息， 发送到恶意服务器  
  -- 恶意服务器拿拿到这些信息，又可以做很多违法的事情
* **生成广告等影响用户体验**

**（2）解决方法：**

1. 对输入脚本进行过滤或转码

如：<script> -->&lt;script&gt;

1. 响应头Set-Cookie加使用限制  
   -- httpOnly，通知浏览器此 Cookie 只能通过浏览器 HTTP 协议传输，浏览器的 JS 引擎就会禁用 document.cookie；  
   -- SameSite=Strict，限制此Cookie不能随着跳转链接跨站发送

**2. CSRF攻击，跨站请求伪造（Cross Site Request Forgery）**

**目的是利用服务器的漏洞和用户的登录状态来实施攻击**

发起CSRF攻击的方式：

* 通过<img src="恶意网站">，自动跳转到恶意网站
* 通过诱导用户点击隐藏链接，指向恶意网站

**解决方法：**  
-- SameSite=Strict，限制此Cookie不能随着跳转链接跨站发送  
-- 验证请求来源站点  
-- 使用Token验证  
服务器第一次返回时生成一个Token  
再次请求客户端带着对应的Token，进行验证