浏览器原理

# CPU 中央处理器：

解释计算机指令处理软件中的数据，串行执行任务。4核8核处理器。cpu的核心数是指物理上，也就是硬件上存在几个核心。

# GPU图形处理器

单个GPU只能处理简单任务，但是数量多，并行计算能力强。

# 进程和线程

3.1进程可以看做是正在执行的应用程序。

3.2线程是运行在进程里面的，一个进程可能包含一个或者多个线程。

3.3启动应用程序的时候，操作系统会为这个程序创建一个进程，同时为这个进程分配一块私有的内存空间，用来存储所有程序相关的数据和状态。

# 4、浏览器架构

4.1、多进程浏览器的组成

Browser：浏览器进程，只有一个，负责浏览器的主体部分，包括导航栏，书签，前进，后退按钮，提供存储等功能。

Network:网络进程，只有一个，负责页面的网络资源加载。

GPU：图像渲染进程，负责独立于其他进程的GPU任务。

Renderer:渲染进程，负责tab内和网页展示相关的所有工作，比如将HTML,CSS和js转换为用户可以与之交互的网页，默认情况下每个tab都有一个独立的渲染进程。

plugins:-插件进程（多个）

Extensions:扩展程序进程（多个）

其他进程：工具进程，辅助框架等。

**4.2、多进程的好处**

1）兼容性：为每个tab分配独立的渲染进程。

2）安全性和沙盒性：操作系统提供方法限制每个进程的能力，所以浏览器可以让某些进程不具备某些特定的功能。提高了安全性。

1）独立进程，每个进程都可以拥有更多内存。

**4.3、多进程的坏处**

每个进程都有自己独立的内存，不能像每个进程中的线程一样，共享内存。所以耗用更多的内存。

浏览器如何优化？

限制开启的进程数：当进程数达到一定的界限后，Chrome会将访问同一个网站的tab放到同一个进程里跑。

chrome服务化，浏览器本身相关的部分拆分为一个个的服务，这些服务可以在一个进程里跑，也可以在一个进程中跑，根据硬件的好坏。

**4.4、网站隔离**

每个iframe会有一个独立的渲染进程，哪怕一个tab页中有两个甚至多个iframe

# 导航时发生了什么

**5.1、浏览器进程分为：**UI线程，绘制浏览器顶部按钮和导航输入框等组件。

存储线程：控制文件读写。

**5.2、处理输入：**

当用户在导航栏输入信息的时候，UI线程要进行一系列的解析是将用户输入发送给搜索引擎还是直接请求你输入的站点资源。

**5.3、开始导航**

当客户点击enter键之后，UI线程会通知网络进程初始化一个网络请求来获取站点的内容。这时tab上的icon会展示一个提示资源正在加载中的旋转圈圈，而且网络进程会进行一系列诸如DNS寻址以及为请求建立TLS连接的操作。

注意：如果这个时候网络进程收到服务器的HTTP301重定向请求，它会通知浏览器ui线程进行重定向，然后它会重新发起一个网络请求。

**5.4、读取响应**

5.4.1、响应类型判断：网络进程在收到HTTP响应主体时，确定响应主体的媒体类型值。

5.4.2、不同响应类型的处理

1）如果是HTML文件，浏览器将获取的响应数据交给渲染进程。

2）如果拿到的是压缩文件，浏览器会将数据交给下载管理器来处理。

5.4.3、安全检查：

将内容交给渲染进程之前进行SafeBrowsing检查。

**5.5、寻找一个渲染进程来绘制页面。**

UI线程在收到网络进程的确认后（数据准备好），寻找一个渲染进程来渲染界面。其实这个渲染进程在网络进程干活的时候已经准备好了。

**5.6、提交导航**

一旦浏览器进程收到渲染进程的回复说导航已经被提交了，导航过程就结束了，文档的加载阶段就开始了。

**5.7、加载完成**

导航提交完成后，渲染进程就开始着手加载资源以及渲染界面。

# 6、Service Worker场景下的导航

如果开发者在service worker⾥设置了当前的⻚⾯内容从缓存⾥⾯获取，当前⻚⾯的渲染就不需要重 新发送⽹络请求了，这就⼤⼤加快了整个导航的过程。 这⾥要重点留意的是service worker其实只是⼀些跑在渲染进程⾥⾯的JavaScript代码。 那么问题来了，当导航开始的时候，浏览器进程是如何判断要导航的站点存不存在对应的service worker并启动⼀个渲染进程去执⾏它的呢？ 其实service worker在注册的时候，它的作⽤范围（scope）会被记录下来。 在导航开始的时候，⽹络进程会根据请求的域名在已经注册的service worker作⽤范围⾥⾯寻找有没 有对应的service worker。如果有命中该URL的service worker，UI线程就会为这个service worker启 动⼀个渲染进程（renderer process）来执⾏它的代码。Service worker既可能使⽤之前缓存的数据 也可能发起新的⽹络请求。

# 7、渲染进程中具体做了什么

**7.1、渲染进程负责tab标签内发生的所有事情**。将HTML，CSS,以及js转变成我们可以进行交互的网页。

渲染进程里有：一个主线程，几个工作线程，一个合成线程以及一个光栅线程。

**7.2、解析**

**1. 构建DOM**

上⾯提到过，渲染进程在导航结束的时候会收到来⾃浏览器进程提交导航的消息，在这之后渲染进 程就会开始接收HTML数据，同时主线程也会开始解析接收到的⽂本数据，并把它转化为⼀个 DOM（Document Object Model）对象。 DOM对象既是浏览器对当前⻚⾯的内部表⽰，也是Web开发⼈员通过JavaScript与⽹⻚进⾏交互的 数据结构以及API。 如何将HTML⽂档解析为DOM对象是在HTML标准中定义的。 不过在你的web开发⽣涯中，你可能从来没有遇到过浏览器在解析HTML的时候发⽣错误的情景。 这是因为浏览器对HTML的错误容忍度很⼤。举些例⼦：如果⼀个段落缺失了闭合p标签（ ），这个⻚⾯还是会被当做为有效的HTML来处理； Hi! **I'm *Chrome****!* 虽然有语法错误，不过浏览器会把它处理为 Hi! **I'm *Chrome****!*。

**2. ⼦资源加载**

除了HTML⽂件，⽹站通常还会使⽤到⼀些诸如图⽚，CSS样式以及JavaScript脚本等⼦资源，这些 ⽂件会从缓存或者⽹络上获取。 主线程会按照在构建DOM树时遇到各个资源的顺序⼀个接着⼀个地发起⽹络请求，为了提升效率， 浏览器会同时运⾏“预加载扫描”程序。 如果在HTML⽂档⾥⾯存在诸如或者这样的标签，预加载扫描程序会在HTML解析器⾥⾯找到对应要 获取的资源，并把这些要获取的资源告诉浏览器进程⾥⾯的⽹络线程。 看⼀下图⽚ 6.⼦资源加载.png 3. JavaScript会阻塞HTML的解析过程 当HTML解析器碰到script标签的时候，它会停⽌HTML⽂档的解析从⽽转向JavaScript代码的加载， 解析以及执⾏。 为什么要这样做呢？因为script标签中的JavaScript可能会使⽤诸如document.write()这样的代码改变 ⽂档流（document）的形状，从⽽使整个DOM树的结构发⽣根本性的改变。因为这个原因，HTML 解析器不得不等JavaScript执⾏完成之后才能继续对HTML⽂档流的解析⼯作。 给浏览器⼀点如何加载资源的提⽰ Web开发者可以通过很多⽅式告诉浏览器如何才能更加优雅地加载⽹⻚需要⽤到的资源。 你可以为script标签添加⼀个async或者defer属性来使JavaScript脚本进⾏异步加载。 资源预加载可以⽤来告诉浏览器这个资源在当前的导航肯定会被⽤到，你想要尽快加载这个资源。 样式计算 CSS 拥有了DOM树我们还不⾜以知道⻚⾯的外貌，因为我们通常会为⻚⾯的元素设置⼀些样式。 主线程会解析⻚⾯的CSS从⽽确定每个DOM节点的计算样式（computed style）。计算样式是主线 程根据CSS样式选择器（CSS selectors）计算出的每个DOM元素应该具备的具体样式，你可以打开 devtools来查看每个DOM节点对应的计算样式。 即使你的⻚⾯没有设置任何⾃定义的样式，每个DOM节点还是会有⼀个计算样式属性，这是因为每 个浏览器都有⾃⼰的默认样式表。 因为这个样式表的存在，⻚⾯上的h1标签⼀定会⽐h2标签⼤，⽽且不同的标签会有不同的magin和 padding。 看图⽚. 7.样式计算.png 布局 Layout 前⾯这些步骤完成之后，渲染进程就已经知道⻚⾯的具体⽂档结构以及每个节点拥有的样式信息 了，可是这些信息还是不能最终确定⻚⾯的样⼦. 只知道⽹站的⽂档流以及每个节点的样式是远远不⾜以渲染出⻚⾯内容的，还需要通过布局 （layout）来计算出每个节点的⼏何信息。 布局的具体过程是： 1. 主线程会遍历刚刚构建的DOM树，根据DOM节点的计算样式计算出⼀个布局树（layout tree）。 2. 布局树上每个节点会有它在⻚⾯上的x，y坐标以及盒⼦⼤⼩（bounding box sizes）的具体信息。 布局树⻓得和先前构建的DOM树差不多，不同的是这颗树只有那些可⻅的（visible）节点信息。 举个例⼦，如果⼀个节点被设置为了display:none，这个节点就是不可⻅的就不会出现在布局树上⾯ （visibility:hidden的节点会出现在布局树上⾯）。同样的，如果⼀个伪元素（pseudo class）节点有 诸如p::before{content:"Hi!"}这样的内容，它会出现在布局上，⽽不存在于DOM树上。 绘画 - Paint 知道了DOM节点以及它的样式和布局其实还是不⾜以渲染出⻚⾯来的。 为什么呢？举个例⼦，假如你现在想对着⼀幅画画⼀幅⼀样的画，你已经知道了画布上每个元素的 ⼤⼩，形状以及位置，你还是得思考⼀下每个元素的绘画顺序，因为画布上的元素是会互相遮挡的 （z-index）。 如果⻚⾯上的某些元素设置了z-index属性，绘制元素的顺序就会影响到⻚⾯的正确性。 ⾼成本的渲染流⽔线（rendering pipeline）更新 关于渲染流⽔线有⼀个⼗分重要的点就是流⽔线的每⼀步都要使⽤到前⼀步的结果来⽣成新的数 据，这就意味着如果某⼀步的内容发⽣了改变的话，这⼀步后⾯所有的步骤都要被重新执⾏以⽣成 新的记录。举个例⼦，如果布局树有些东西被改变了，⽂档上那些被影响到的部分的绘画顺序是要 重新⽣成的。 看图 8.渲染流⽔线.png 如果你的⻚⾯元素有动画效果（animating），浏览器就不得不在每个渲染帧的间隔中通过渲染流⽔ 线来更新⻚⾯的元素。 我们⼤多数显⽰器的刷新频率是⼀秒钟60次（60fps），如果你在每个渲染帧的间隔都能通过流⽔线 移动元素，⼈眼就会看到流畅的动画效果。可是如果流⽔线更新时间⽐较久，动画存在丢帧的状况 的话，⻚⾯看起来就会很“卡顿”。 即使你的渲染流⽔线更新是和屏幕的刷新频率保持⼀致的，这些更新是运⾏在主线程上⾯的，这就 意味着它可能被同样运⾏在主线程上⾯的JavaScript代码阻塞。 看图 9.js阻塞渲染流⽔线.png 对于这种情况，你可以将要被执⾏的JavaScript操作拆分为更⼩的块然后通过 requestAnimationFrame这个API把他们放在每个动画帧中执⾏。想知道更多关于这⽅⾯的信息的 话，可以参考Optimize JavaScript Execution。当然你还可以将JavaScript代码放在WebWorkers中 执⾏来避免它们阻塞主线程。 看图 10.rAF优化 合成 1. 如何绘制⼀个⻚⾯ 浏览器已经知道了关于⻚⾯以下的信息：⽂档结构，元素的样式，元素的⼏何信息以及它们的绘画 顺序。那么浏览器是如何利⽤这些信息来绘制出⻚⾯来的呢？将以上这些信息转化为显⽰器的像素 的过程叫做光栅化（rasterizing）。 现代浏览器采⽤合成的⽅式, 来展⽰整个⻚⾯ 2. 什么是合成 合成是⼀种将⻚⾯分成若⼲层，然后分别对它们进⾏光栅化，最后在⼀个单独的线程 - 合成线程 （compositor thread）⾥⾯合并成⼀个⻚⾯的技术。当⽤⼾滚动⻚⾯时，由于⻚⾯各个层都已经被 光栅化了，浏览器需要做的只是合成⼀个新的帧来展⽰滚动后的效果罢了。⻚⾯的动画效果实现也 是类似，将⻚⾯上的层进⾏移动并构建出⼀个新的帧即可。

**4.2、什么操作会触发浏览器的回流**

1）删除或者增加dom元素。

2）浏览器的resize事件。

3）改变dom元素的位置，尺寸。

4）获取offsetTop,scrollTop,getComputedStyle();

为什么？浏览器保证每次返回给用户的值都是最新的值。当获取上面的信息的时候，会强制清空重排操作的队列。

5）页面的初始渲染，布局的时候。

注：由于每次重排/回流的时候都会增加额外的消耗，所以浏览器会通过队列化来批量执行优化重排过程（即需要重排的操作放在队列中，当操作达到一定的数值后，清空重排队列）