浏览器原理与PWA

这节课我们就Chrome浏览器来讲解浏览器原理。

正式开始之前, 咱们先来复习一下大学学的cpu.gpu.内存.进程.线程的概念, 会有助于这节课的听讲。

1. CPU

中央处理器, 解释计算机指令以及处理计算机软件中的数据, 它可以串行地一件接着一件处理交给它的任务

现代电脑上cpu通常会有多个核心, 比如经常听到的8核处理器, 4核处理器等等。

CPU的核心数是指物理上，也就是硬件上存在着几个核心。比如，双核就是包括2个相对独立的CPU核心单元组，四核就包含4个相对独立的CPU核心单元组

2. GPU

图形处理器, 单个GPU核心只能处理一些简单的任务，不过它胜在数量多，单片GPU上会有很多很多的核心可以同时工作，也就是说它的并行计算能力是非常强的

3. 进程 & 线程

进程 - 可以看成正在被执行的应用程序（executing program）。
线程 - 是跑在进程里面的，一个进程里面可能有一个或者多个线程，这些线程可以执行任何一部分应用程序的代码

当你启动一个应用程序的时候，操作系统会为这个程序创建一个进程同时还为这个进程分配一片私有的内存空间，这片空间会被用来存储所有程序相关的数据和状态。
当你关闭这个程序的时候，这个程序对应的进程也会随之消失，进程对应的内存空间也会被操作系统释放掉。

很多应用程序都会采取多进程的方式来工作，因为进程和进程之间是互相独立的，它们互不影响。

换句话来说，如果其中一个工作进程挂掉了，其他进程不会受到影响，而且挂掉的进程还可以重启

一、浏览器架构

这里看下图 单进程和多进程浏览器的区别.png

咱们要讲的Chrome浏览器, 是多进程架构, 先来看一下都包含哪些进程。

Browser(一个) - 浏览器进程, 只有一个浏览器进程，负责浏览器的主体部分，包括导航栏，书签，前进和后退按钮, 提供存储等功能

Network(一个) - 网络进程, 主要负责页面的网络资源加载，之前是作为一个模块运行在浏览器进程里面的，直至最近才独立出来，成为一个单独的进程

GPU(一个) - 图像渲染进程, 负责独立于其它进程的GPU任务。它之所以被独立为一个进程是因为它要处理来自于不同tab的渲染请求并把它在同一个界面上画出来。

Renderer(多个) - 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 JavaScript 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。

Plugin(多个) - 插件进程

Extensions(多个) - 扩展程序进程

其他进程 - 工具进程，辅助框架等等

可以在chrome浏览器 更多工具 - 任务管理器 查看当前浏览器所开启的进程, 以及内存和cpu消耗

多进程架构的好处

1. 容错性

Chrome会为每个tab单独分配一个属于它们的渲染进程（render process）。举个例子，假如你有三个tab，你就会有三个独立的渲染进程。

当其中一个tab的崩溃时，你可以随时关闭这个tab并且其他tab不受到影响。可是如果所有的tab都跑在同一个进程的话，它们就会有连带关系，一个挂全部挂。

2. 安全性和沙盒性

因为操作系统可以提供方法让你限制每个进程拥有的能力，所以浏览器可以让某些进程不具备某些特定的功能。例如，由于tab渲染进程可能会处理来自用户的随机输入，所以Chrome限制了它们对系统文件随机读写的能力。

3. 每个进程可以拥有更多内存

因为每个进程都会分配一块独立的内存空间, 所以理所当然的, 每个进程都会有更多的内存。

多进程架构的坏处

其实上面已经提到了, 每个进程都会拥有自己独立的内存空间, 他们并不能像同一个进程中的线程一样共享内存空间。

而一些基础的东西比如V8 Javascript引擎, 会在不同进程的内存空间中同时存在, 所以就消耗了不必要的内存。

多进程架构内存的优化

- 那么Chrome是怎么优化这种情况的呢?

答案就是：限制启动的进程数目，当进程数目达到一定界限后, Chrome会将访问同一个网站的tab都放在一个进程里面跑。

- Chrome的服务化, 节省更多的内存

这里可以看一下图 assets/Chrome的服务化.png

同样的优化方法也可以被使用在浏览器进程（browser process）上面。

Chrome浏览器的架构正在发生一些改变，目的是将和浏览器本身（Chrome）相关的部分拆分为一个个不同的服务，服务化之后，这些功能既可以放在不同的进程里面运行也可以合并为一个单独的进程运行。

这样做的主要原因是让Chrome在不同性能的硬件上有不同的表现。当Chrome运行在一些性能比较好的硬件时，浏览器进程相关的服务会被放在不同的进程运行以提高系统的稳定性。相反如果硬件性能不好，这些服务就会被放在同一个进程里面执行来减少内存的占用。

这样，原来的各种模块会被重构成独立的服务（Service），每个服务（Service）都可以在独立的进程中运行，访问服务（Service）必须使用定义好的接口，通过 IPC 来通信，从而构建一个更内聚、松耦合、易于维护和扩展的系统，更好实现 Chrome 简单、稳定、高速、安全的目标。

网站隔离 （Site Isolation）

网站隔离会为网站内不同站点的iframe分配一个独立的渲染进程。

之前说过Chrome会为每个tab分配一个单独的渲染进程，可是如果一个tab只有一个进程的话不同站点的iframe都会跑在这个进程里面，这也意味着它们会共享内存，这就有可能会破坏同源策略。

同源策略是浏览器最核心的安全模型，它可以禁止网站在未经同意的情况下去获取另外一个站点的数据，因此绕过同源策略是很多安全攻击的主要目的。

而进程隔离（proces isolation）是隔离网站最好最有效的办法了。因此在Chrome 67版本之后，桌面版的Chrome会默认开启网站隔离功能，这样每一个跨站点的iframe都会拥有一个独立的渲染进程。

二、一个经典问题, 导航时都发生了什么?

上面提到过，浏览器中tab外面发生的一切都是由浏览器进程（browser process）控制的。

浏览器进程有很多负责不同工作的线程（worker thread），其中包括：

1. UI线程（UI thread）：绘制浏览器顶部按钮和导航栏输入框等组件，当你在导航栏里面输入一个URL的时候，其实就是UI线程在处理你的输入。
2. 存储线程（storage thread）： 控制文件读写。

所以，当你在导航栏上输入一串内容的时候，Chrome到底为我们做了哪些工作?

处理输入

当用户开始在导航栏上面输入内容的时候，UI线程（UI thread）做的第一件事就是询问：“你输入的字符串是一些搜索的关键词（search query）还是一个URL地址呢？”。

因为对于Chrome浏览器来说，导航栏的输入既可能是一个可以直接请求的域名，也可能是用户想在搜索引擎里面搜索的关键词信息，
所以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来判定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。

开始导航

当用户按下回车键的时候，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。

这时候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。

- tips: 这时如果网络进程收到服务器的HTTP 301重定向响应，它就会告知UI线程进行重定向然后它会再次发起一个新的网络请求。

读取响应

1. 响应类型判断

网络进程在收到HTTP响应的主体时，在必要的情况下它会先检查一下流的前几个字节以确定响应主体的具体媒体类型（MIME Type）。

响应主体的媒体类型一般可以通过HTTP头部的Content-Type来确定，不过Content-Type有时候会缺失或者是错误的，这种情况下浏览器就要进行MIME类型嗅探来确定响应类型了。

- 这里可以打开一个窗口, 看一下Content-type响应头

2. 不同响应类型的处理

如果响应的主体是一个HTML文件，浏览器会将获取的响应数据交给渲染进程（renderer process）来进行下一步的工作。
如果拿到的响应数据是一个压缩文件（zip file）或者其他类型的文件，响应数据就会交给下载管理器（download manager）来处理。

3. 安全检查

网络进程在把内容交给渲染进程之前还会对内容做SafeBrowsing检查。

如果请求的域名或者响应的内容和某个已知的病毒网站相匹配，网络进程会给用户展示一个警告的页面。除此之外，网络进程还会做CORB（Cross Origin Read Blocking）检查来确定那些敏感的跨

站数据不会被发送至渲染进程。

寻找一个渲染进程来绘制页面

在网络进程做完所有的检查后并且能够确定浏览器应该导航到该请求的站点，它就会告诉UI线程所有的数据都已经被准备好了。

UI线程在收到网络进程的确认后会为这个网站寻找一个渲染进程（renderer process）来渲染界面。

- tips. 这里chrome有个小优化

因为网络请求的耗时可能会很长, 所以第二步中当UI线程发送URL链接给网络进程后，它其实已经知晓它们要被导航到哪个站点了。

所以在网络进程干活的时候，UI线程会主动地为这个网络请求启动一个渲染线程。如果一切顺利的话（没有重定向之类的东西出现），网络进程准备好数据后页面的渲染进程已经就准备好了，这就节省了新建渲染进程的时间。

不过如果发生诸如网站被重定向到不同站点的情况，刚刚那个渲染进程就不能被使用了，它会被摒弃，一个新的渲染进程会被启动。

提交导航

到这一步的时候，数据和渲染进程都已经准备好了，浏览器进程（browser process）会通过IPC告诉渲染进程去提交本次导航（commit navigation）。

除此之外浏览器进程还会将刚刚接收到的响应数据流传递给对应的渲染进程让它继续接收到来的HTML数据。

一旦浏览器进程收到渲染线程的回复说导航已经被提交了（commit），导航这个过程就结束了，文档的加载阶段（document loading phase）会正式开始。

到了这个时候，导航栏会被更新，安全指示符和站点设置会展示新页面相关的站点信息。
当前tab的会话历史（session history）也会被更新，这样当你点击浏览器的前进和后退按钮也可以导航到刚刚导航完的页面。为了方便你在关闭了tab或窗口（window）的时候还可以恢复当前tab和会话（session）内容，当前的会话历史会被保存在磁盘上面。

加载完成

当导航提交完成后，渲染进程开始着手加载资源以及渲染页面。

一旦渲染进程完成渲染（load），它会通过IPC告知浏览器进程，然后UI线程就会停止导航栏上旋转的圈圈。

三、导航到不同的站点

上面讲述了一个导航的过程, 那么这时候如果我们想去浏览另一个网页, 浏览器会怎么做呢?

能够想到的是, 浏览器必然会重复一遍导航的步骤, 但是在这之前, 浏览器还有一些收尾工作要做!

浏览器进程会对渲染进程说, 我准备重新发起导航了, 你那边是否需要处理beforeunload事件?

beforeunload可以在用户重新导航或者关闭当前tab时给用户展示一个“你确定要离开当前页面吗？”的二次确认弹框。

浏览器进程之所以要在重新导航的时候和当前渲染进程确认的原因是，当前页面发生的一切（包括页面的JavaScript执行）是不受它控制而是受渲染进程控制，它不知道里面的具体情况。

- tips：所以不要随便给页面添加beforeunload事件监听，你定义的监听函数会在页面被重新导航的时候执行，因此这会增加重导航的时延。
  beforeunload事件监听函数只有在十分必要的时候才能被添加，例如用户在页面上输入了数据，并且这些数据会随着页面消失而消失。

1. 如果第二次导航是在页面内发起的, 比如页面内Js执行了location.href=xxxx, 这时候浏览器是怎么做的?

渲染进程会自己先检查一个它有没有注册beforeunload事件的监听函数，如果有的话就执行，执行完后发生的事情就和之前的情况没什么区别了，唯一的不同就是这次的导航请求是由渲染进程给浏览器进程发起的。

2. 如果第二次导航是到不同的站点呢?

会有另外一个渲染进程被启动来完成这次重导航，而当前的渲染进程会继续处理现在页面的一些收尾工作，例如unload事件的监听函数执行。

- 这里可以看一下图 重新导航不同站点.png

四、Service Worker场景下的导航

如果开发者在service worker里设置了当前的页面内容从缓存里面获取，当前页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。

五、导航预加载

在上面的例子中，你应该可以感受到如果启动的service worker最后还是决定发送网络请求的话，浏览器进程和渲染进程这一来一回的通信包括service worker启动的时间其实增加了页面导航的时延。

导航预加载就是一种通过在service worker启动的时候并行加载对应资源的方式来加快整个导航过程效率的技术。预加载资源的请求头会有一些特殊的标志来让服务器决定是发送全新的内容给客户端还是只发送更新了的数据给客户端。

六、渲染进程中具体做了什么

渲染进程负责标签（tab）内发生的所有事情。

渲染进程的主要任务是将HTML，CSS，以及JavaScript转变为我们可以进程交互的网页内容。

渲染进程里面有：一个主线程（main thread），几个工作线程（worker threads），一个合成线程（compositor thread）以及一个光栅线程（raster thread）

在渲染进程里面，主线程（main thread）处理了绝大多数你发送给用户的代码。如果你使用了web worker或者service worker，相关的代码将会由工作线程（worker thread）处理。合成（compositor）以及光栅（raster）线程运行在渲染进程里面用来高效流畅地渲染出页面内容。

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页面的内部表示，也是Web开发人员通过JavaScript与网页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（

）），这个页面还是会被当做为有效的HTML来处理；

```
Hi! <b>I'm <i>Chrome</b>!</i>
```

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循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页需要用到的资源。

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页面的外貌，因为我们通常会为页面的元素设置一些样式。

主线程会解析页面的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。

即使你的页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，页面上的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页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页面的样子.

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页面内容的，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页面上的x，y坐标以及盒子大小（bounding box sizes）的具体信息。布局树长得和先前构建的DOM树差不多，不同的是这颗树只有那些可见的（visible）节点信息。

举个例子，如果一个节点被设置为了display:none，这个节点就是不可见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页面来的。

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）。

如果页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页面的正确性。

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

如果你的页面元素有动画效果（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页面的元素。

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，页面看起来就会很“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. 如何绘制一个页面

浏览器已经知道了关于页面以下的信息：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页面来的呢？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, 来展示整个页面

2. 什么是合成

合成是一种将页面分成若干层，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）里面合并成一个页面的技术。当用户滚动页面时，由于页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。页面的动画效果实现也是类似，将页面上的层进行移动并构建出一个新的帧即可。