

37 | 浏览器原理（一）：浏览器为什么要用多进程模型？

2022-10-26 LMOS 来自北京

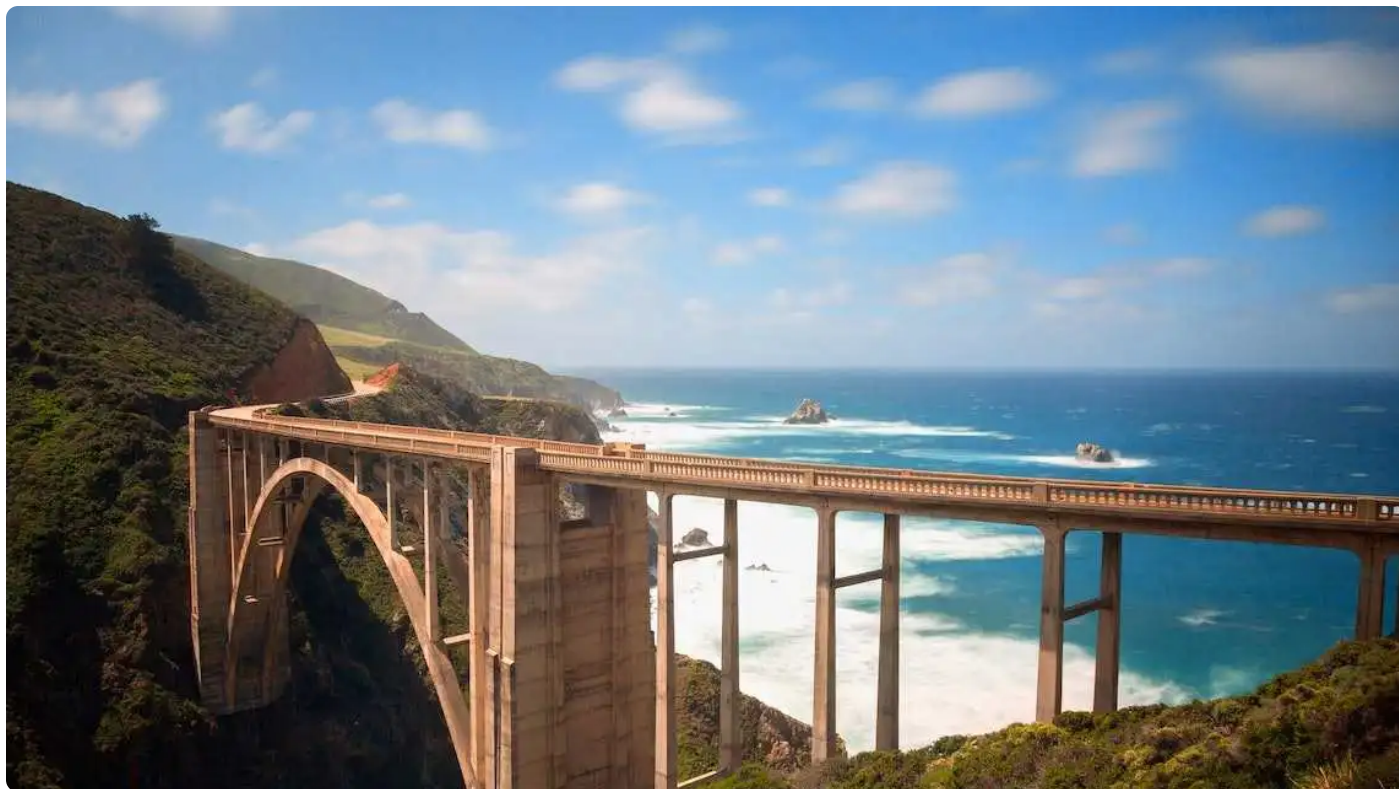


天下无鱼

<https://shikey.com/>

《计算机基础实战课》

[课程介绍 >](#)



讲述：陈晨

时长 12:08 大小 11.08M



你好，我是 LMOS。

前面我们学过了很多基础理论，你可能已经迫不及待，想把这些知识运用到应用层开发里了。所以从这应用篇开始，我们会学以致用，分析一些开发工作中的实际问题，挑战几个典型的综合应用场景。

这节课我会从浏览器开始讲起，浏览器是目前使用范围最广、使用人数最多的终端应用程序之一。作为互联网中最重要的端口，浏览器伴随着互联网的高速发展，发展也是日新月异。通过接下来的两节课，我希望带你看看巨型软件应用优秀的架构设计，同时也带你了解一下平时用到的浏览器里，有哪些技术原理比较关键。

浏览器原本是很简单的东西，只能渲染简单的页面，后来才逐步迈进百花齐放的阶段。

浏览器内核的演变史

我先带你梳理一下浏览器的发展过程。了解了这段历史，你就会重新理解 WebKit 内核和 Chrome 浏览器的地位，知道它们是怎么演变而来的。

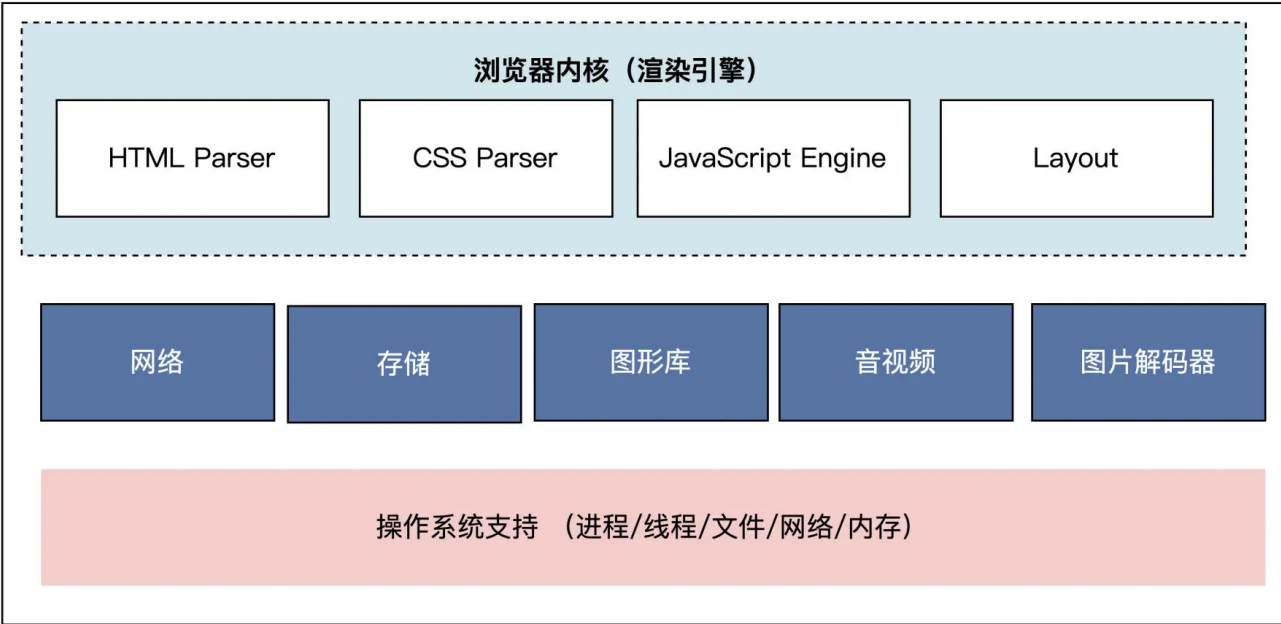
说起浏览器，我们就不得不提到 1994 年诞生的网景浏览器（从 Mosaic 浏览器衍生而来）。虽然网景浏览器只能展示最简单的 HTML 静态页面，不支持动态的脚本（JavaScript）和样式（CSS），但是它仍然获得了很大的成功。

在操作系统中，内核是最基本的功能，随着浏览器的发展，在浏览器中，现在同样也存在内核的概念。浏览器内核的作用相对更加简化，浏览器内核的英文名为 Rendering Engine，你可以把它理解成一个渲染引擎，用途就是把文件资源转化成可视化的图像结果。

浏览器常见的浏览器内核有：Blink、WebKit、Gecko、Trident 等，目前 WebKit 内核占据了非常大的市场，包括 Chrome、Safari、安卓浏览器等市面上的主流浏览器，都使用了 WebKit 内核。

从 WebKit 看浏览器内核架构

既然 WebKit 这么经典，我们就以它为例来看一下浏览器内核的架构。浏览器内核主要包含：HTML Parser，CSS Parser，Layout，JavaScript Engine 几部分，如下图所示：



我们简单看一下，上图中的几个关键部分承担了什么工作：

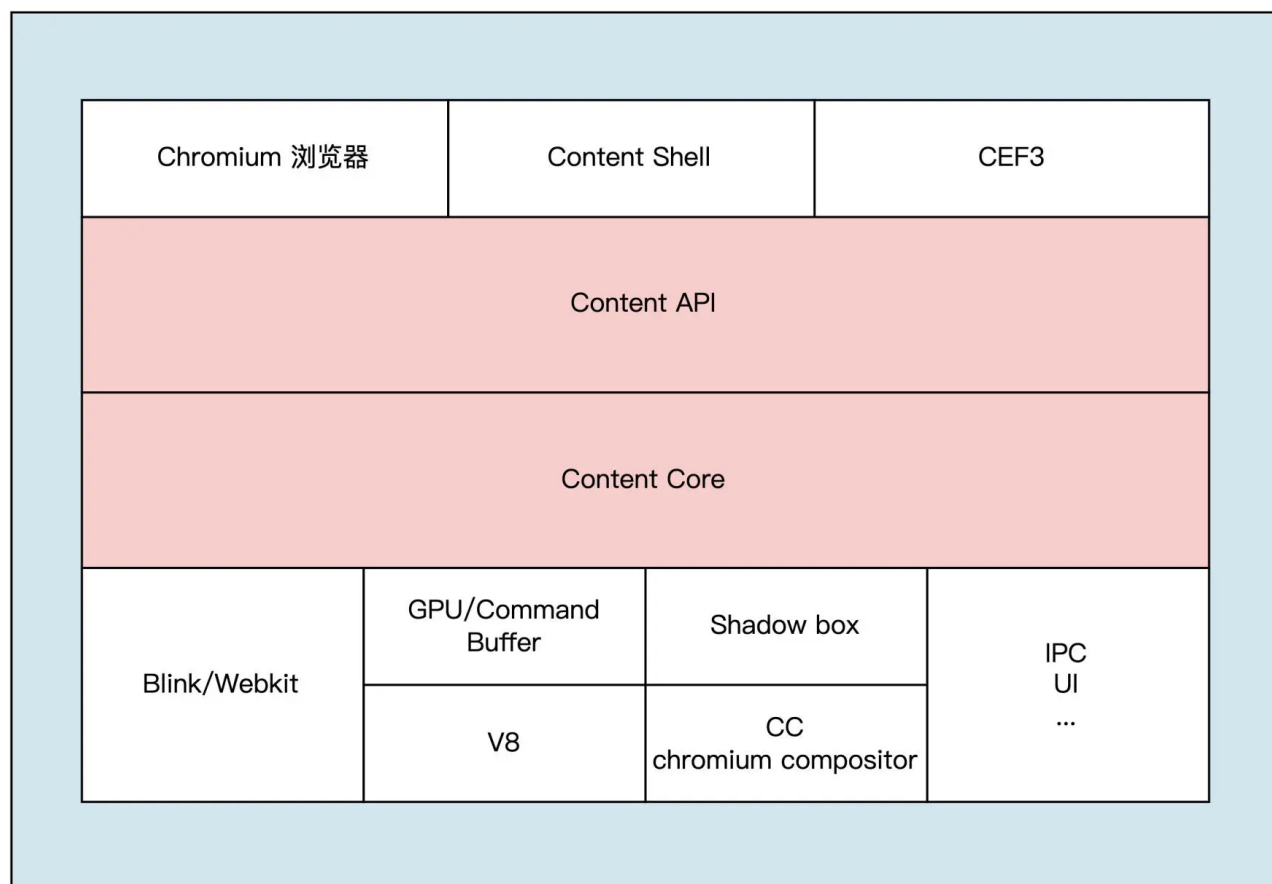
- **HTML Parser:** HTML 解析器，负责 HTML 文本的解析，将 HTML 解析为可编程结构 —— DOM（文档对象模型）树；
- **CSS Parser:** CSS 解析器是层叠样式的解析器，用来计算布局所需要的节点样式信息 —— CSSOM（样式）树；
- **Layout:** 布局，在得到 DOM 树和 CSSOM 树后，需要计算出 DOM 树中可见元素的几何位置，生成布局树 —— Layout Tree；
- **JavaScript Engine:** JavaScript 语言的解析引擎，执行页面的动态逻辑，并可以访问 DOM 和 CSSOM 数据接口；
- 操作系统支持 —— 移植：WebKit 代码中，因为其天生具有跨平台性质，所以部分平台相关的能力需要做跨平台兼容的移植。

上面是一个简略的浏览器内核的功能，不过它仅仅是完成了核心的渲染过程，实际上浏览器则要复杂得多。

在 2013 年，Chromium 发布了 Blink 项目。这个项目是从 WebKit 项目独立出来的，它抽离出了一套新的编程接口和进程模型接口，同时浏览器内核屏蔽了 Chromium 底层的进程模型实现。

Chromium 浏览器架构解读

我们会以 Chromium 浏览器为例，来分析浏览器及其内核。后面是 Chromium 的简易架构图，为方便分析，我删去了部分细节。



在上图中，比较重要的是 **Content** 模块以及 **Content** 接口。

你可以这样理解，**Content** 模块和接口是浏览器对渲染过程的抽象，它们将浏览器的渲染、插件、沙箱等功能，进行了包装和抽象，提供一个接口层，方便上层的应用调用。**Chromium** 中我们可以看到的浏览器可视化界面，它构建在 **Content** 接口之上，用于接收用户交互和展示界面，**content shell** 是一个简易版的浏览器，通常被第三方浏览器软件进行二次开发，它在 **Android** 系统上也应用广泛。

浏览器下的多进程与多线程模型

Chromium 和 **Blink** 最大的一个特性就是采用了新的进程模型和线程模型。在前面进程篇的课程中我们了解到，进程是应用程序运行时操作系统进行资源分配的最小容器，这些资源包括指令集、独立内存空间、IO、PCB 等等。

不过，进程虽然能帮助我们更方便地分配资源，也会引发一些问题：

1. 进程切换上下文的开销比较大。由于虚拟内存的存在，我们需要从硬盘中频繁读写；

2. 多进程应用通讯复杂度高。由于操作系统的保护策略，系统资源跨进程是无法共享的。如果需要跨进程共享资源就要采用 IPC 通讯，但是成本相对高。

线程则是 CPU 调度的基本单位。线程的优点显而易见，切换成本很低，只有少量 CPU 寄存器、堆栈等内容，线程的创建、销毁本身也有性能成本，但这个成本相对较低，而且通常可以通过线程池优化。

不过我们也要关注到不足之处，多线程应用编码复杂度高，这会带来后面这几个问题：

1. 线程可以共享进程内的所有资源，但需要考虑资源竞态问题；
2. 线程间的指令时序不可预测，无法保证代码按照预期的顺序执行；
3. 单个进程崩溃可能会影响其他线程。

那么在目前常见的 **Chrome** 浏览器里，采用的是多进程还是单进程多线程模型呢？在我们的电脑中，我打开进程管理器，就可以看到浏览器的后台进程占用情况。通常后台都存在多个进程：

[illegible]

其实在一些旧的浏览器中，采用的是单进程多线程的模型，如 IE 浏览器；但是以 Chromium 浏览器为例的现代浏览器，采用的都是多进程架构。

那么为什么现代浏览器采用的是多进程架构呢？我们需要先分析一下，浏览器如果是单进程多线程会引发哪些问题。

首先是**稳定性的问题**。因为一个浏览器程序，是可以同时启动多个 Tab 的，浏览器多进程化的最大的好处就是，单个 Tab 的卡死、崩溃不会影响其它 Tab。

我们在浏览器程序内新建一个 Tab 时，就会启动一个新的渲染进程，Chrome 支持四种不同的进程模型模型：

- **Process-per-site-instance**。这种进程模型会为每一个同一个域的实例都会创建一个 **Renderer** 进程。
- **Process-per-site**。这种进程模型会为不同一个域创建独立的进程，同一域的不同实例共享同一个进程。
- **Process-per-tab**。这种进程模型会为每个标签页创建一个 **Renderer** 进程。
- **Single process**。这种进程模型不为页面创建任何独立的进程，所有渲染工作都在 **browser** 进程中（这种模式是实验性质的，不推荐使用）。

Chromium 默认采用 **Process-per-site-instance** 方式，不过我们可以在浏览器启动时传递一个命令行开关，用来指定浏览器的进程模型。

其次，是**加载速度的问题**。由于整个程序只存在一个进程，浏览器的 JS 代码和插件逻辑和页面渲染是运行在同一个进程中的，如果存在一些计算量很大的操作，这些计算量大的线程会抢占大量资源，从而导致其他的渲染逻辑无法正常执行。这会严重影响页面的加载速度，甚至造成崩溃。在多进程架构下，将插件提取为单独的进程，不会存在插件卡顿和崩溃影响整个浏览器的情况。

最后还有**安全性的考虑**。由于 JS 脚本和插件的存在，很容易利用浏览器的系统漏洞，进而获得整个计算机的权限，从而造成安全问题。而多进程架构很容易就可以实现沙箱控制。

虽然采用多进程模型就可以解决上述几个问题，但是多进程模型也不是银弹，它同样也会引发一些问题。

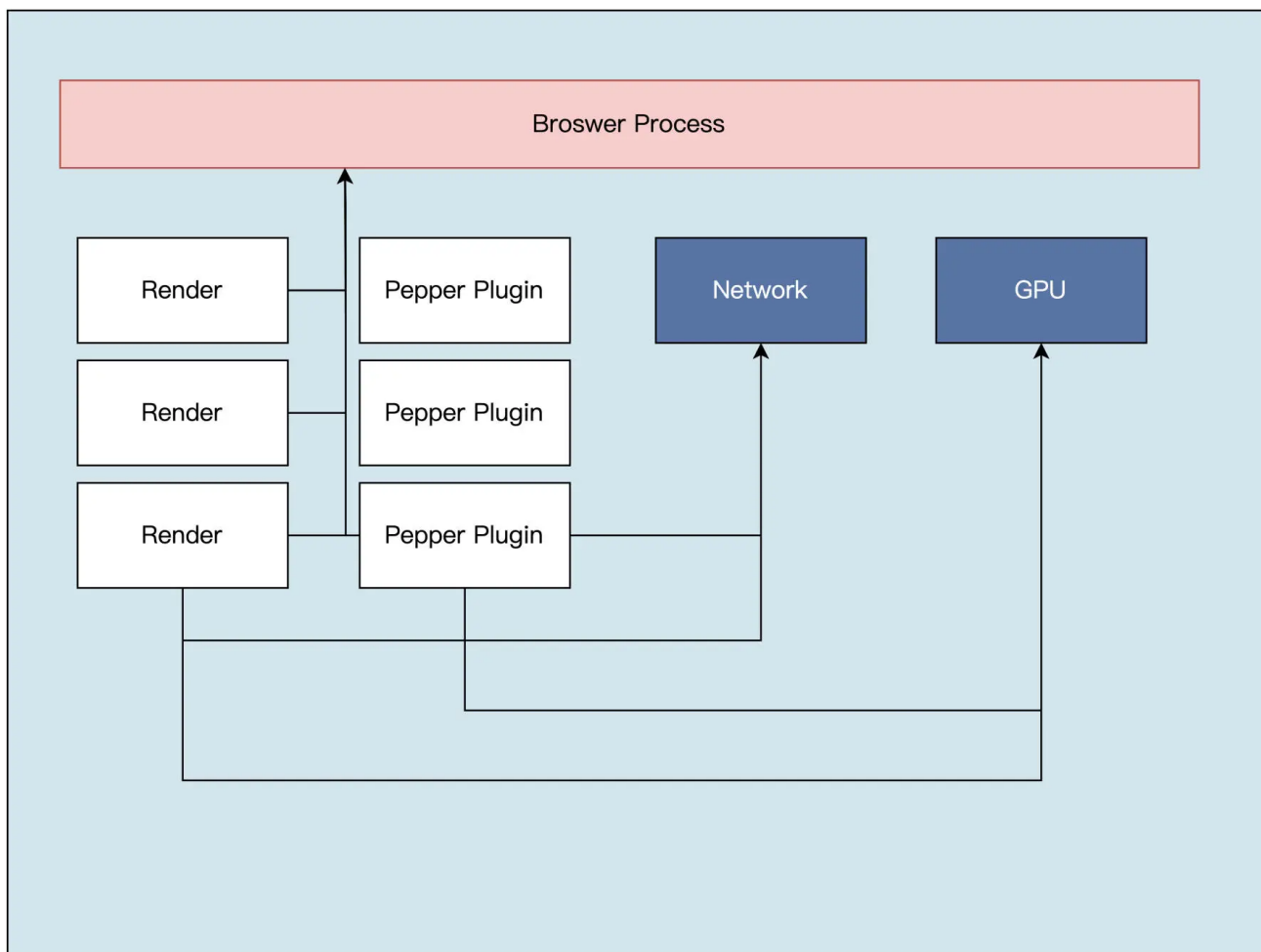
比如性能的问题，由于基础的指令无法共享，多进程会带来很大程度的资源浪费，这也是我们很多同学吐槽的“Chrome 非常吃内存”这个问题。由于每个 Tab 和插件都是一个独立的进程，

所以在打开多个 Tab 或者插件的情况下，我们会看到系统的内存会疯狂飙升。

这其实是设计上存在的问题，随着时间的发展，硬件性能瓶颈的突破也会推动软件架构发展。现在计算机内存越来越大，没有过去那么宝贵了，所以不同的选择，都是有利有弊。

Chrome 有哪些进程

现在，我们知道了浏览器采用的是多进程模型，那么具体有哪些常见进程呢？我们结合后面这张图来看看。



图中的方框表示具体的进程，连线表示进程间进行的通信，没有连线则表示不会发生进程通信。比如多个渲染进程之间不会进行通信，它们会通过主进程交互，NPAPI 插件也不会和 GPU 进行交互，因为它太过于古老，所以没有 GPU 的接口实现。

图中我们可以看到以下几种进程：

- 浏览器进程：主要负责用户交互、子进程管理和文件储存等功能；
- 网络进程：浏览器主进程和渲染进程通过他来向操作系统申请端口以及与操作系统的协议栈进程通信；
- 渲染进程：主要职责是把从网络下载的 HTML、JavaScript、CSS、图片等资源解析为可以显示和交互的页面；
- 插件进程：主要负责单个插件功能的运行；
- GPU 进程：主要负责 3D 效果的实现以及 UI 的绘制。

这里我们着重了解一下渲染进程（renderer）内的主要线程。渲染进程内部是多线程的，在内核控制下各线程相互配合以保持同步。那么渲染进程内部具体有哪些线程呢？一共有八类：

- GUI 线程：负责渲染浏览器中的页面，并解析 HTML，CSS；
- JS 线程：负责处理 JavaScript 脚本程序；
- 事件触发线程：归属于浏览器而不是 JS 引擎，用来控制事件循环；
- 定时触发器线程：浏览器的定时任务，如 `setInterval` 与 `setTimeout` 事件，也包括浏览器内部的一些定时任务。
- IO 线程：用来和其他进程进行 IPC 通信，接受发送消息；
- 异步 http 请求线程：处理所有的异步请求，如果有回调函数，就放入异步事件队列，由事件触发线程处理；
- WebWorker 线程：每声明一个 WebWorker 就会新建一个 WebWorker 线程处理；
- 合成线程：在 GUI 渲染后执行，将 GUI 渲染线程生成的产物转换为位图。

重点回顾

这节课告一段落，我给你做个总结吧。

今天我们先简单分析了 WebKit 和 Chromium 的架构，它们俩在浏览器里是非常经典的设计。

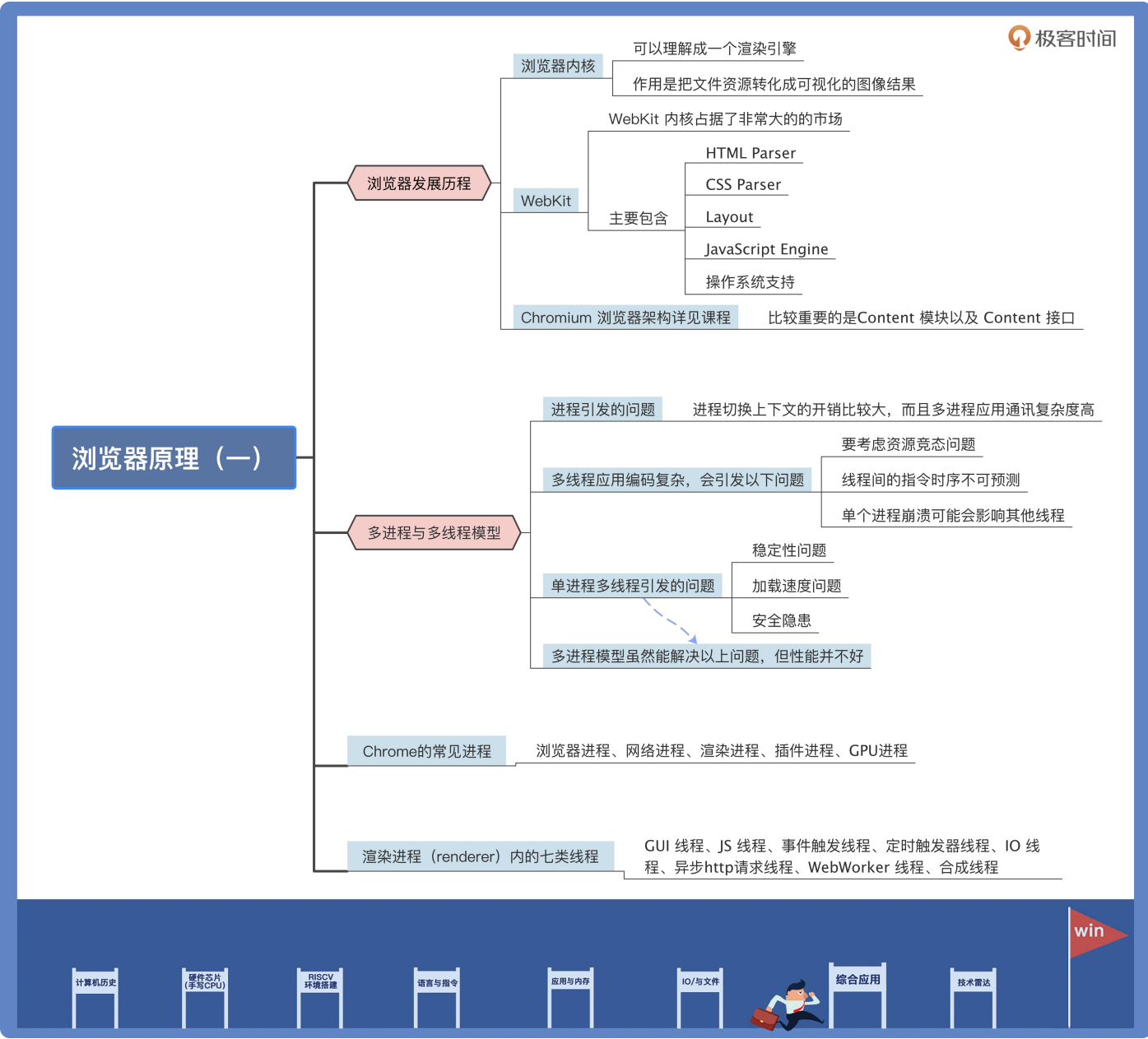
然后，我们讨论了 Chrome 浏览器里，采用的模型是多进程，还是单进程多线程。Chrome 浏览器使用了多进程模型解决了一些历史问题，它内部有负责统领全局的浏览器主进程、负责网

络交互的网络进程、负责页面渲染的渲染进程、负责插件运行的独立插件进程以及负责 3D 效果的 GPU 进程。

这里重点说说渲染进程，它主要负责单个 Tab 内的页面渲染逻辑，在渲染进程内又是多线程的，在浏览器内核控制下各线程相互配合以保持同步，高效协作。

在各个进程和线程的配合下，浏览器会进行一系列的动作完成渲染，我们称之为**渲染流水线**，就是从接受网络请求开始，到将其处理成可展示的图形和用户进行交互的过程。那这个过程具体如何运转呢？我们下节课再继续讨论。

这节课导图如下所示，供你复习回顾：



思考题

浏览器的多进程模型下，进程之间是如何通信的呢？

期待你在留言区说说自己的思考。也推荐你把这节课分享给更多同事、朋友。

分享给需要的人，Ta购买本课程，你将得 20 元

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 36 | Linux文件系统（二）：Linux如何存放文件？

下一篇 38 | 浏览器原理（二）：浏览器进程通信与网络渲染详解

精选留言 (5)

写留言



苏流郁宓

2022-10-26 来自湖北

chrome内核部分代码网上有人说是kernel？

浏览器中有cache的存在，网址寻址信息也是先找本地资源，找不到再到网上去找！

那么浏览器进程通信应该与操作系统进程通信原理差不多，唯一的区别的浏览器的内存地址分配需要操作系统协助，包括但不限于管道通信，消息队列通信，套接字通信等！（不喜欢用ipc一笔带过）

作者回复: 对



4



peter

2022-10-26 来自湖北

对于思考题，关于chrome进程间通信：我感觉应该是消息队列。

另外，有几个问题：

Q1：但进程崩溃影响的线程是其他进程的线程吗？还是本进程的线程？

Q2: 进程模型中的域是指什么？

Q3: chrome线程间同步机制是什么？用synchronized吗？

作者回复: 影响



功夫熊猫

2022-11-16 来自湖北

跟操作系统一个道理呗，ipc通信。

作者回复: 嗯嗯



ryohei

2022-10-26 来自湖北

IPC通信啊，具体没有说。

作者回复: 嗯嗯



苏流郁宓

2022-10-26 来自湖北

其实可以模块化管理，庞大的浏览器在模块化上可以理解为有缺陷的微型操作系统（无独立硬件内存分页的权利），一个个不同网页可以理解为操作系统上不同的app，那么进程间通信就好理解了，唯一的区别是浏览器兼顾网络数据，对安全这块要求甚高，需要套接字等一系列机制来保障安全，高效的通信的！

作者回复: 6666 你理解能力强

