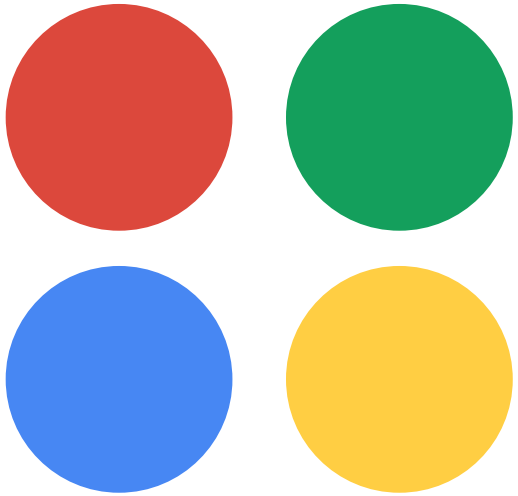


撩一下浏览器



Chrome 是如何工作的？

2019-07-02　zhoulianchi

目录

1 基本架构

2 导航过程

3 渲染进程

4 事件处理

1

基本架构

No. 1 基本架构

“如何开发一个浏览器？”

No. 1 基本架构

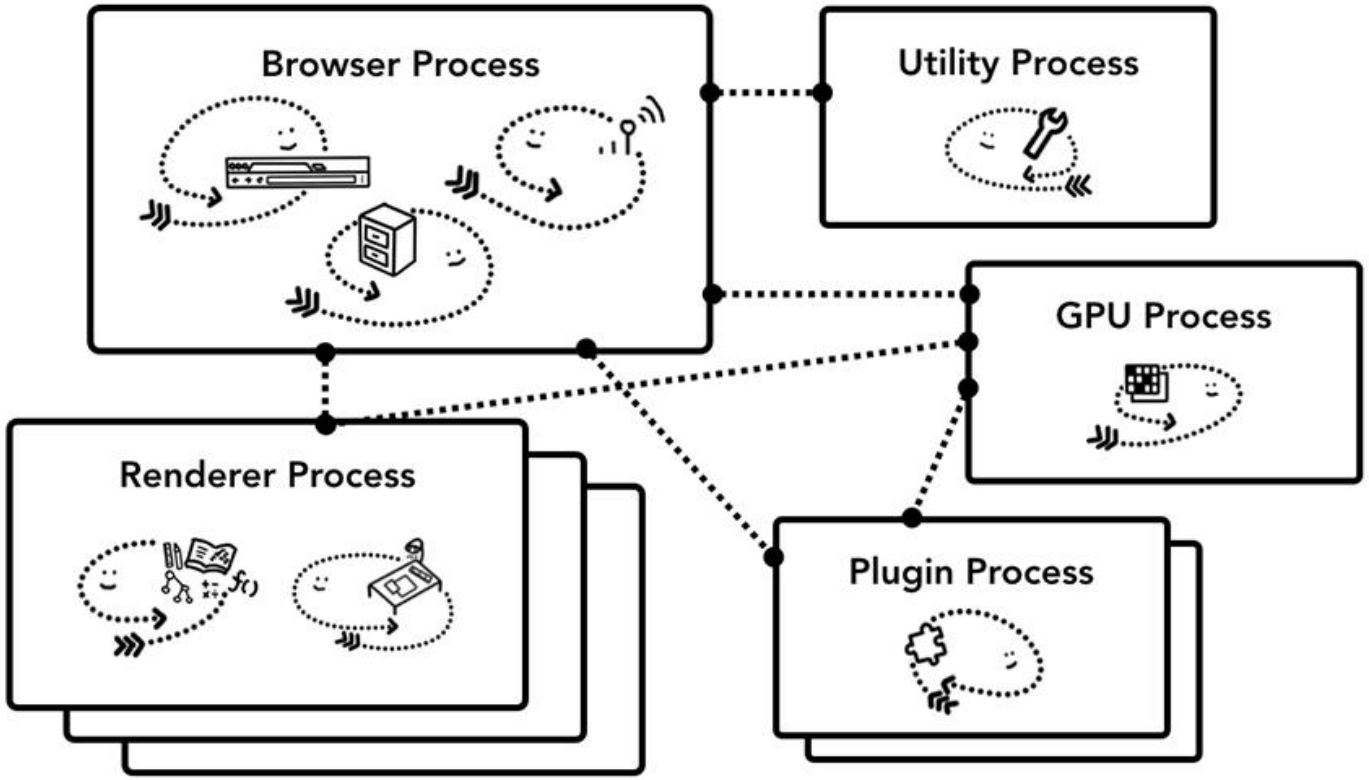
浏览器的多进程架构

- 以 Chrome 为例，它由多个进程组成，每个进程都有自己核心的职责，它们相互配合完成浏览器的整体功能。

- 每个进程中又包含多个线程，一个进程内的多个线程也会协同工作，配合完成所在进程的职责。

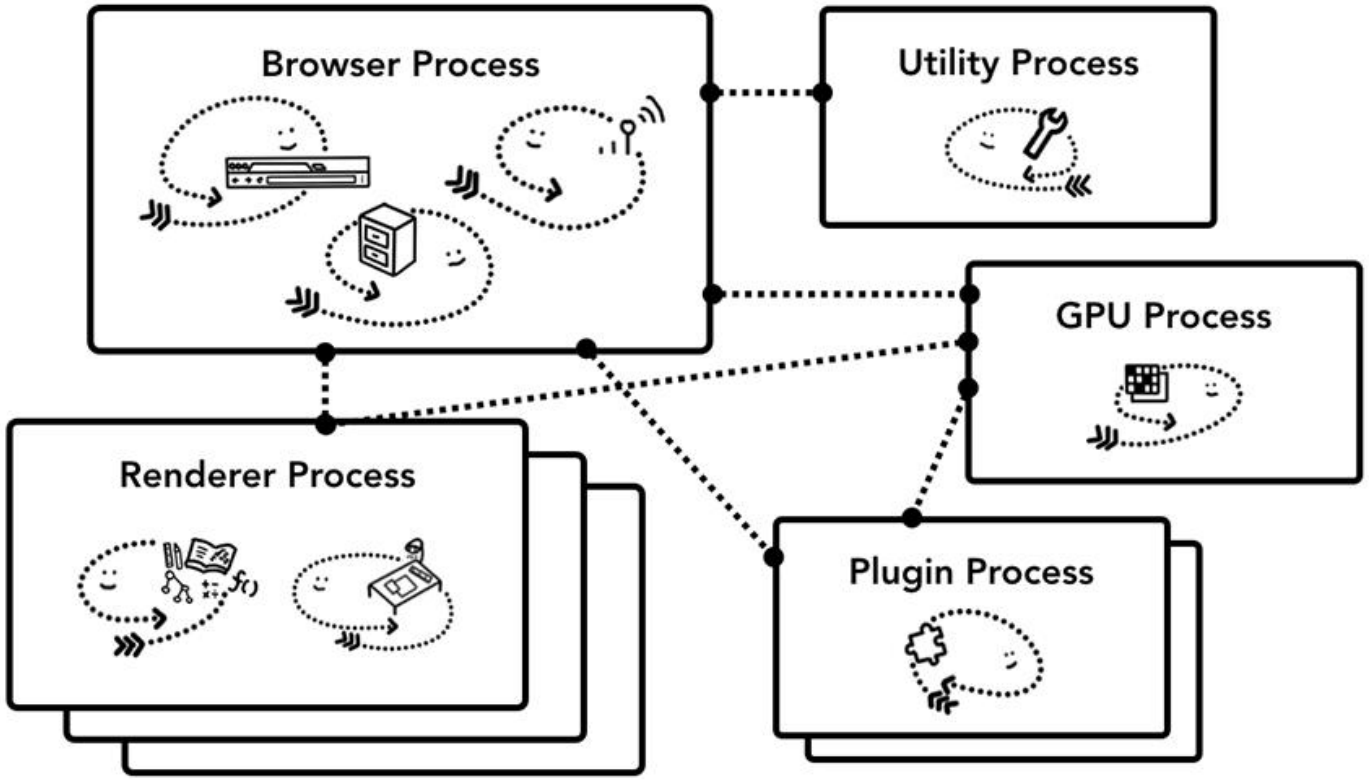
No. 1 基本架构

浏览器的多进程架构



No. 1 基本架构

浏览器的多进程架构



Browser Process：

- 负责包括地址栏，书签栏，前进后退等部分的工作；
- 负责处理浏览器的一些不可见的底层操作，比如网络请求和文件访问按钮。

Renderer Process：

- 负责一个 tab 内关于网页呈现的所有事情。

Plugin Process：

- 负责控制一个网页用到的所有插件，如 flash 。

GPU Process：

- 负责处理 GPU 相关的任务。

浏览器的多进程架构

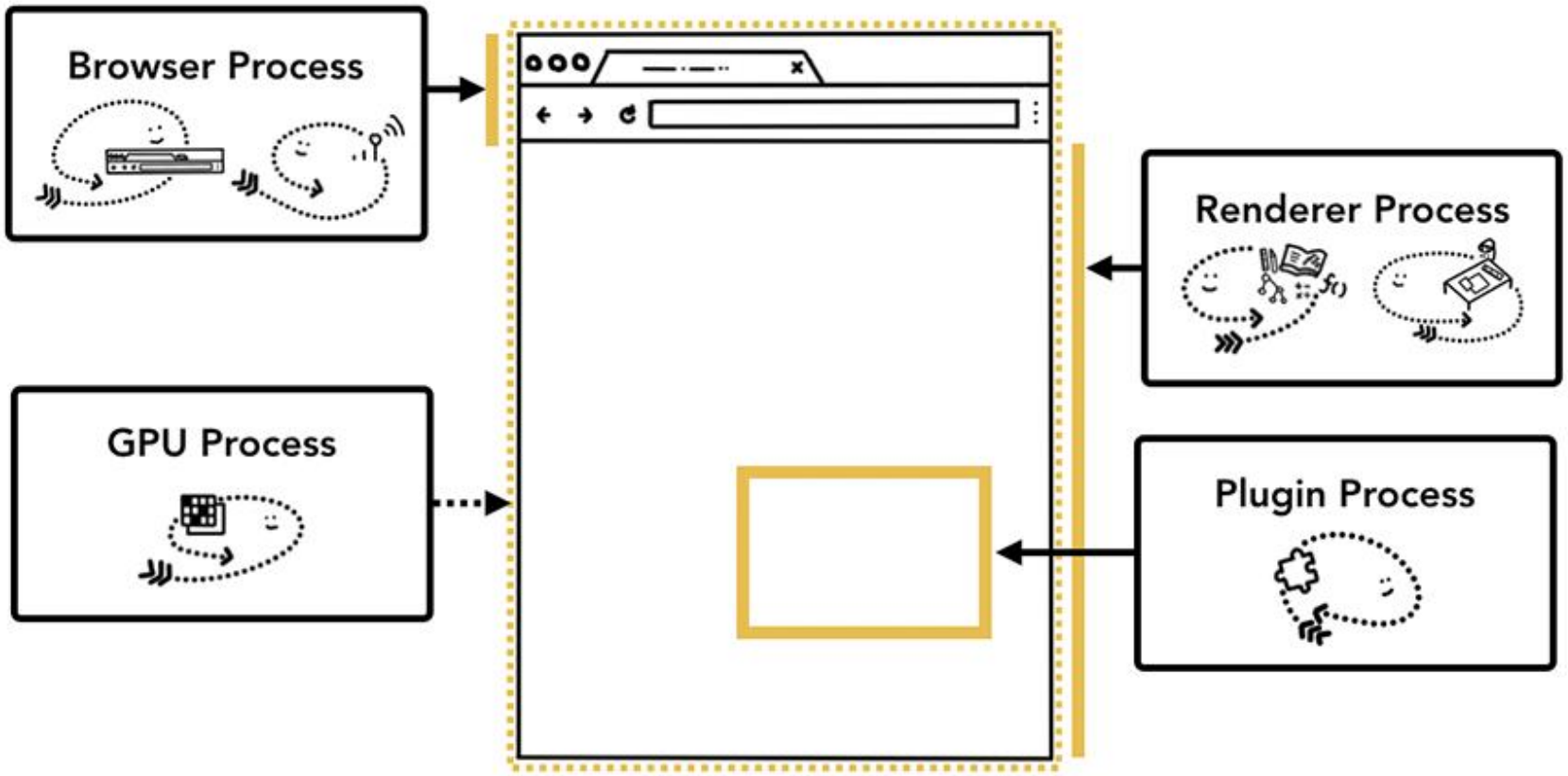


图 - 不同进程负责的浏览器区域

Chrome 多进程架构的优缺点

优点

- 某一渲染进程出问题不会影响其他进程
- 更为安全，在系统层面上限定了不同进程的权限

缺点

- 由于不同进程间的内存不共享，不同进程的内存常常需要包含相同的内容。

Chrome 的一些内存优化



- 为了节省内存，Chrome 限制了最大进程数，最大进程数量由设备的内存和 CPU 能力决定，当达到这一限制时，新打开的 Tab 会共用之前同一个站点的渲染进程。

- 以 Broswer Process 为例，如果 Chrome 运行在强大的硬件上，它会分割不同的服务到不同的进程，这样 Chrome 整体的运行会更加稳定，但是如果 Chrome 运行在资源贫瘠的设备上，这些服务又会合并到同一个进程中运行。

2

导航过程

No. 2 导航过程

Browser Process 的线程划分

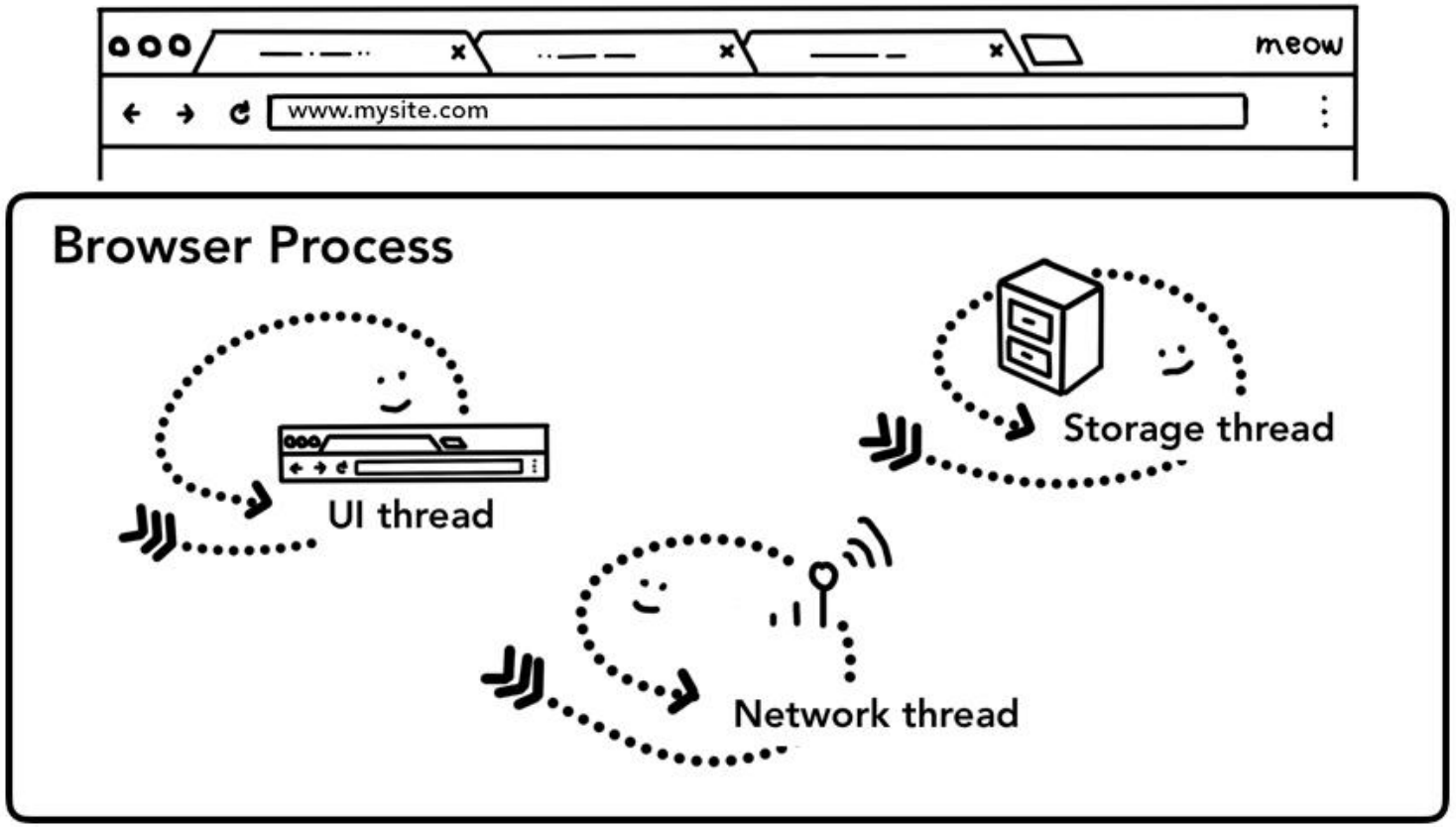


图 - 浏览器主进程中的不同线程

- UI thread： 控制浏览器上的按钮及输入框。
- Network thread：处理网络请求，从网上获取数据。
- Storage thread：控制文件等的访问。

从输入到呈现

步骤 1：处理输入

- UI thread 需要判断用户输入的是 URL 还是 query。

步骤 2：开始导航

- 当用户点击回车键，UI thread 通知 Network thread 获取网页内容，并控制 tab 上的 spinner 展现，表示正在加载中。
- Network thread 会执行 DNS 查询，随后为请求建立 TLS 连接。
- 如果 Network thread 接收到了重定向状态码如 301，Network thread 会通知 UI thread 服务器要求重定向，之后，另外一个 URL 请求会被触发。

从输入到呈现

步骤 3：读取响应

- 当响应返回时，Network thread 会依据 Content-Type 及 MIME Type sniffing 判断响应内容的格式。

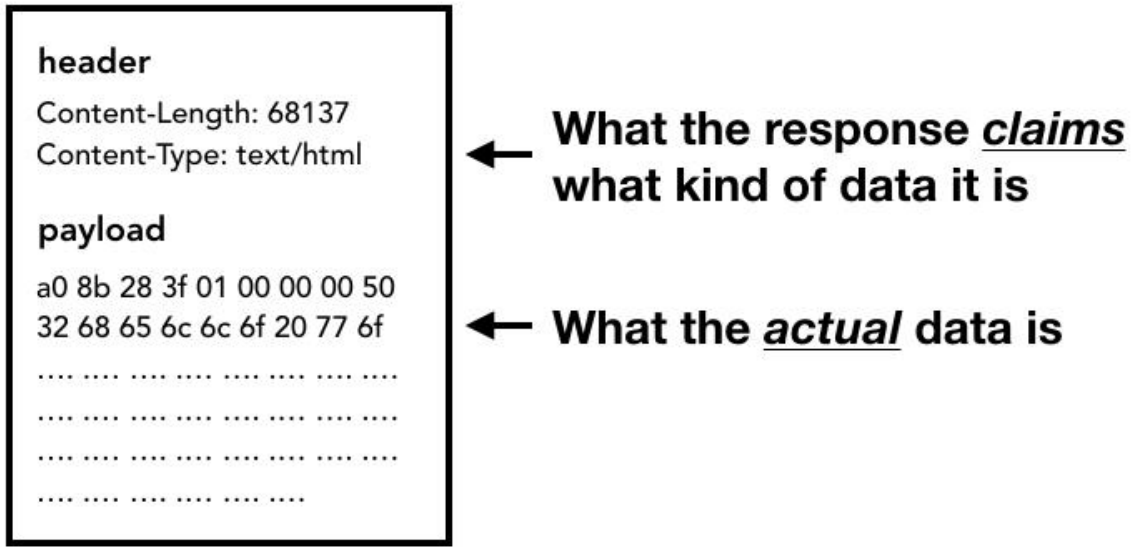


图 - 判断响应内容的格式

从输入到呈现

步骤 3：读取响应

- 如果响应内容的格式是 HTML ，下一步将会把这些数据传递给 Renderer process，如果是 zip 文件或者其它文件，会把相关数据传输给下载管理器。
- Safe Browsing 检查也会在此时触发，如果域名或者请求内容匹配到已知的恶意站点，Network thread 会展示一个警告页。
- 此外 CORB 检测也会触发确保敏感数据不会被传递给渲染进程。

从输入到呈现

步骤 4：查找渲染进程

- 当上述所有检查完成，Network thread 确信浏览器可以导航到请求网页，Network thread 会通知 UI thread 数据已经准备好，UI thread 会查找到一个 Renderer process 进行网页渲染。
- 加速方案 —— 当 UI thread 发送 URL 请求给 Network thread 时，浏览器其实已经知道了将要导航到那个站点。UI thread 会并行的预先查找和启动一个渲染进程，如果一切正常，当 Network thread 接收到数据时，渲染进程已经准备就绪了。

从输入到呈现

步骤 5：确认导航

- 经过了上述过程，数据以及渲染进程都已可用， Browser Process 会给 Renderer process 发送 IPC 消息来确认导航，一旦 Browser Process 收到后者的渲染确认消息，导航过程结束，页面加载过程开始。
- 此时，地址栏会更新，展示出新页面的网页信息。history tab 会更新，可通过返回键返回导航来的页面，为了让关闭 tab 或者窗口后便于恢复，这些信息会存放在硬盘中。

从输入到呈现

步骤 6：完成渲染

- 一旦导航被确认，Renderer process 会使用相关资源渲染页面。当 Renderer process 渲染结束（意味着该页面包括所有 iframe 都触发了 onload），会发送 IPC 信号到 Browser process， UI thread 会停止展示 tab 中的 spinner 。

- 注意：上面的流程只是网页首帧渲染完成，在此之后，浏览器依旧可下载额外的资源渲染出新的视图。

从输入到呈现

Service Worker

- 除了上述流程，有些页面还拥有 Service Worker （服务工作线程），Service Worker 让开发者对本地缓存及判断何时从网络上获取信息有了更多的控制权，如果 Service Worker 被设置为从本地 cache 中加载数据，那么就没有必要从网上获取更多数据了。

- 注意： Service Worker 也是运行在渲染进程中的 JS 代码。

3

渲染进程

No. 3 渲染进程

渲染进程的核心目的：转换 HTML、CSS、JS 为用户可交互的 Web 页面。

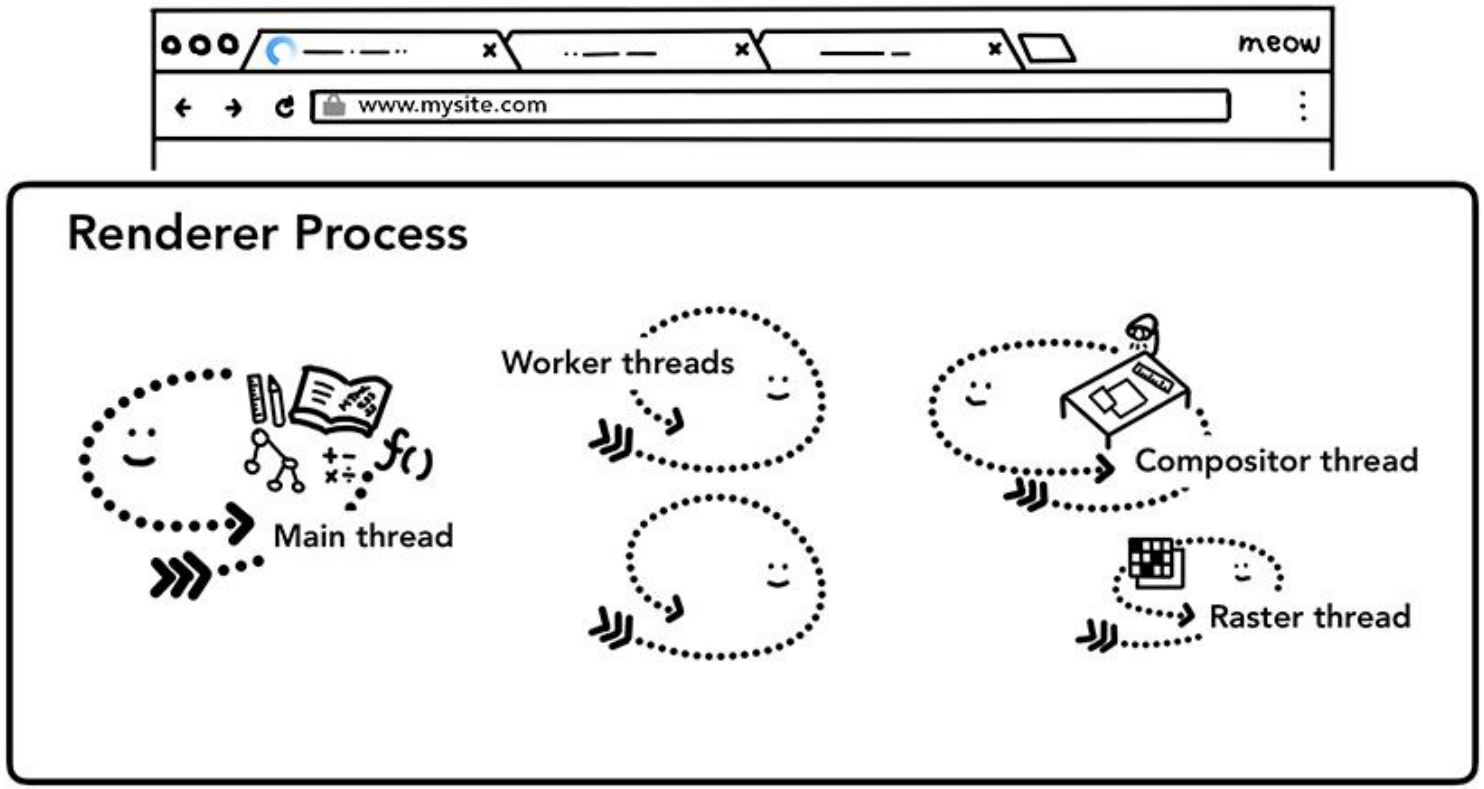


图 – 渲染进程包含的线程

No. 3 渲染进程

渲染的流程

步骤 1：构建 DOM

- 当渲染进程接收到导航的确认信息，开始接受 HTML 数据时，主线程会解析文本字符串为 DOM。
- 渲染 HTML 为 DOM 的方法由 HTML Standard 定义。

步骤 2：加载次级资源

- 对于图片，CSS，JS 等额外的资源，主进程在构建 DOM 的过程中会逐一请求它们，为了加速，Preload Scanner 会同时运行，如果在 html 中存在 <img> <link> 等标签，Preload Scanner 会把这些请求传递给 Browser process 中的 Network thread 进行相关资源的下载。

渲染的流程

步骤 3：JS 的下载与执行

- 当遇到 <script> 标签时，渲染进程会停止解析 HTML，而去加载、解析和执行 JS 代码，停止解析 html 的原因在于 JS 可能会改变 DOM 的结构（使用诸如 document.write() 等 API）。
- 其他处理：在 <script> 标签上添加了 async 或 defer 等属性，浏览器会异步的加载和执行 JS 代码，而不会阻塞渲染。

渲染的流程

步骤 4：样式计算

- 除了解析 DOM，主线程还会基于 CSS 选择器解析 CSS 获取每一个节点的最终的计算样式值 。

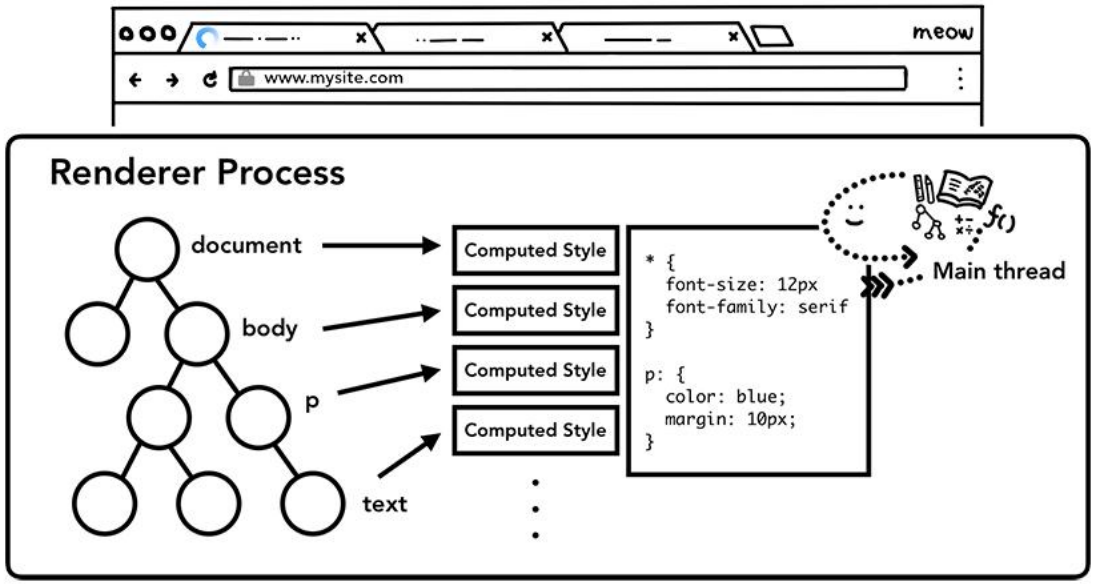


图 – 渲染进程主线程计算每一个元素节点的最终样式值

渲染的流程

步骤 5：获取布局

- 通过遍历 DOM 及相关元素的计算样式，主线程会构建出包含每个元素的坐标信息及盒子大小的布局树。

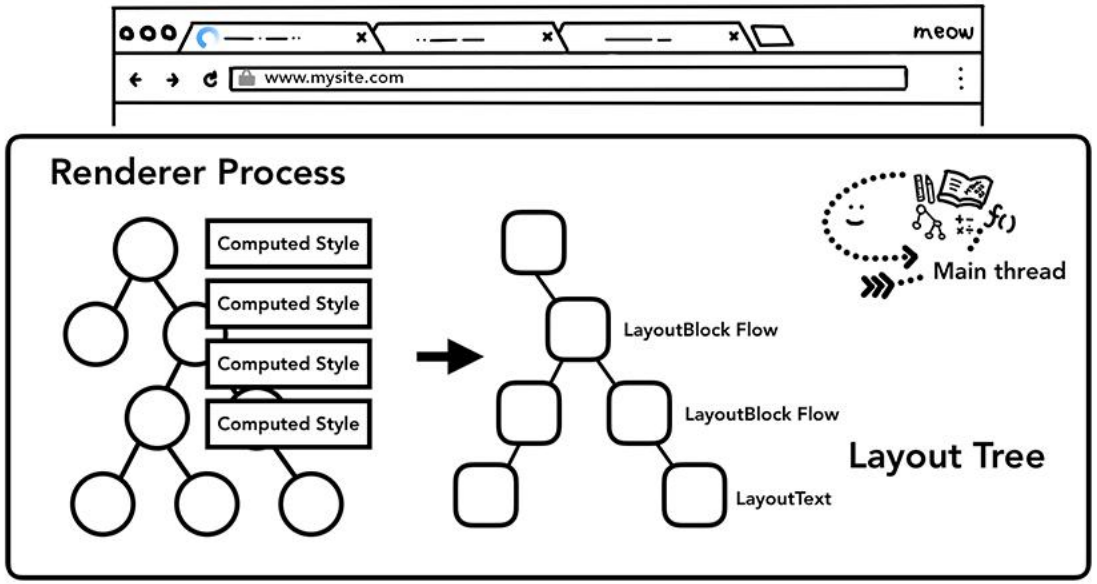


图 – 布局树

渲染的流程

步骤 6：绘制各元素

- 在绘制阶段，主线程会遍历布局树以创建绘制记录。绘制记录可以看做是记录各元素绘制先后顺序的笔记 。

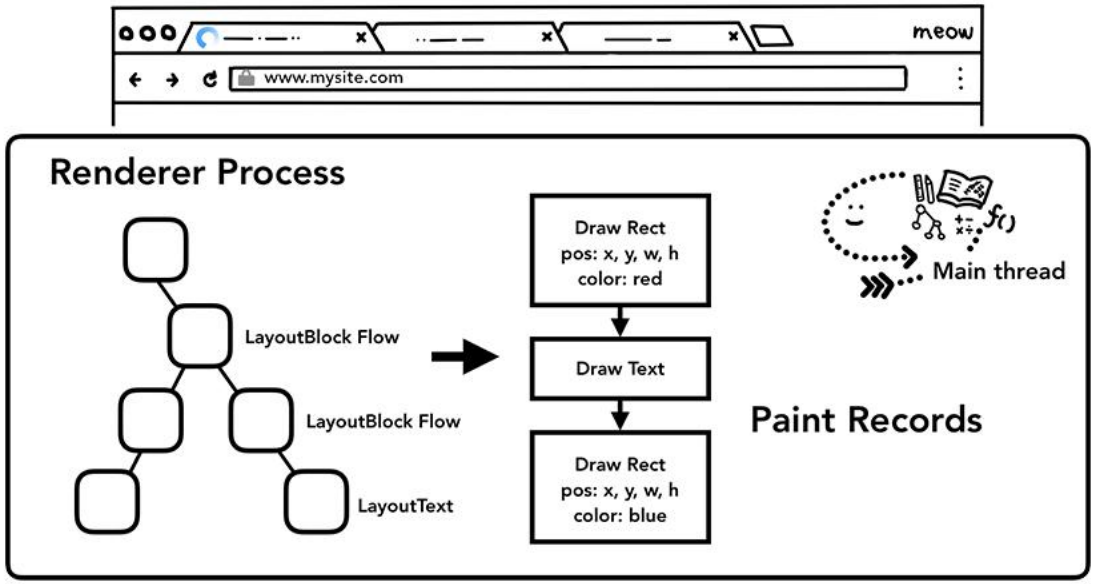


图 – 主线程依据布局树构建绘制记录

渲染的流程

步骤 7：合成帧

- 复合是一种分割页面为不同的层，并单独栅格化，随后组合为帧的技术。不同层的组合由 Compositor 线程完成。
- 主线程会遍历布局树来创建层树（layer tree）。

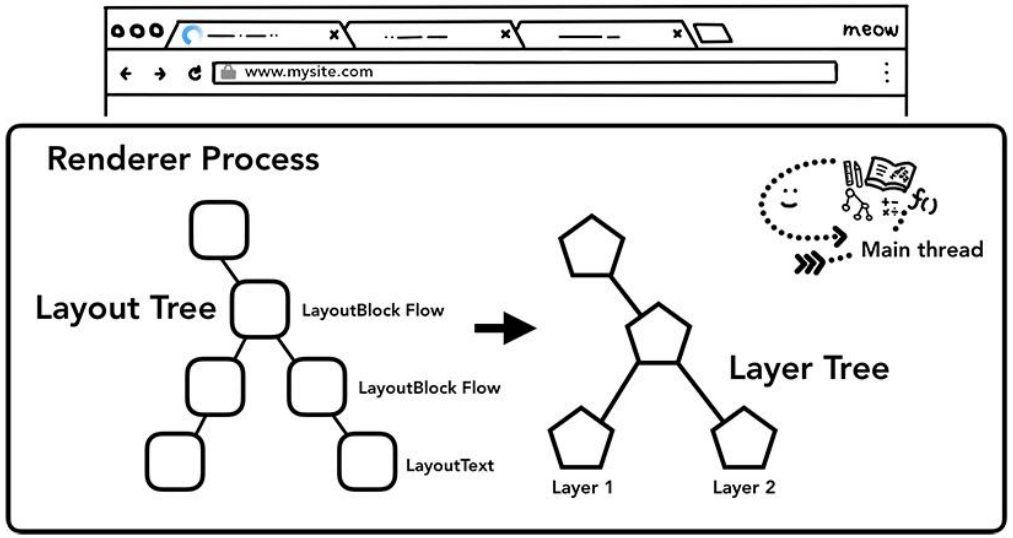


图 – 主线程遍历布局树生成层树

渲染的流程

步骤 7：合成帧

- 一旦层树被创建，渲染顺序被确定，主线程会把这些信息通知给合成器线程，合成器线程会栅格化每一层。
- 有的层的可以达到整个页面的大小，因此，合成器线程将它们分成多个磁贴，并将每个磁贴发送到栅格线程，栅格线程会栅格化每一个磁贴并存储在 GPU 显存中。

渲染的流程

步骤 7：合成帧

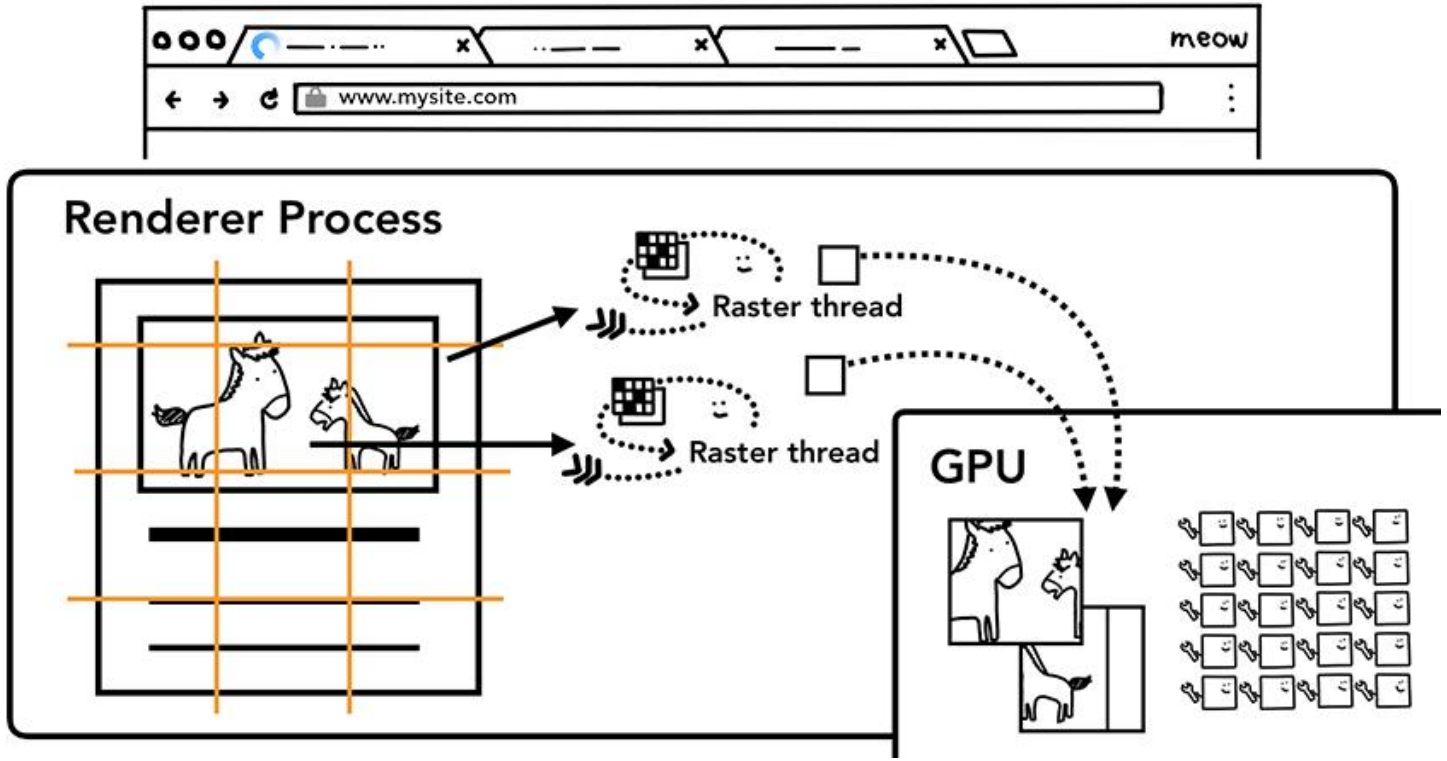


图 – 栅格线程会栅格化每一个磁贴并存储在 GPU 显存中

渲染的流程

步骤 7：合成帧

- 合成器线程的优点在于，其工作无关主线程，合成器线程不需要等待样式计算或者 JS 执行，这就是为什么合成器相关的动画最流畅，如果某个动画涉及到布局或者绘制的调整，就会涉及到主线程的重新计算，自然会慢很多。

4

事件处理

No. 4 事件处理

在浏览器眼里，事件是什么？

- 在浏览器的看来，输入、鼠标滚动、悬置、点击等等都是事件。
- 当用户在屏幕上触发诸如 touch 等手势时，首先收到手势信息的是 Browser process， 不过 Browser process 只会感知到在哪里发生了手势，对 tab 内内容的处理是还是由渲染进程控制的。
- 事件发生时，浏览器进程会发送事件类型及相应坐标给渲染进程，渲染进程随后找到事件对象并执行所有绑定在其上的相关事件处理函数。

查找到事件对象

- 当渲染进程中的合成器线程发送输入事件给主线程时，主线程首先会进行命中测试（hit test）来查找对应的事件目标，命中测试会基于渲染过程中生成的绘制记录（ paint records ）查找事件发生坐标下面存在的元素。

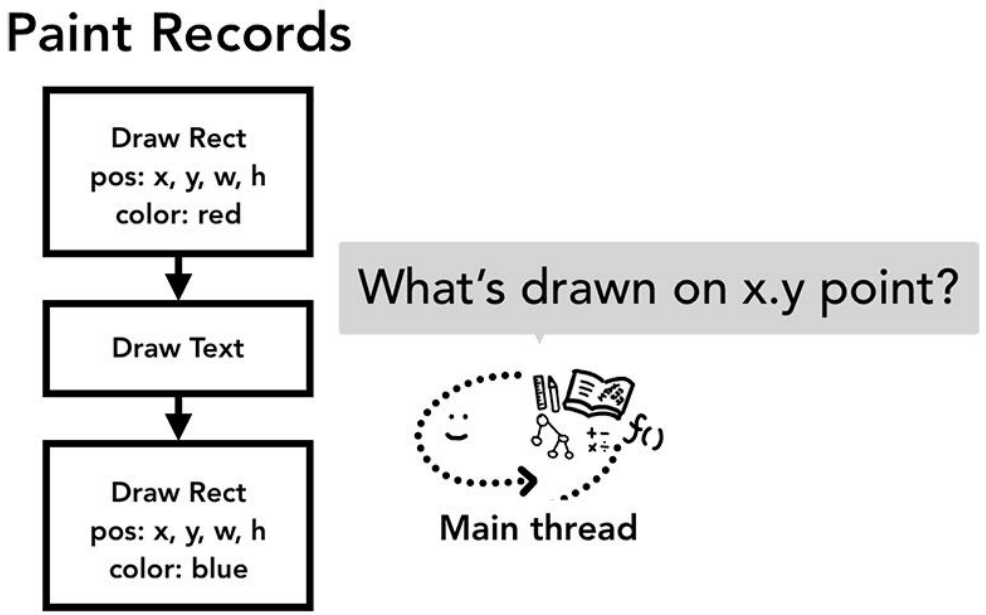


图 – 主线程依据绘制记录查找事件相关元素

异步与事件

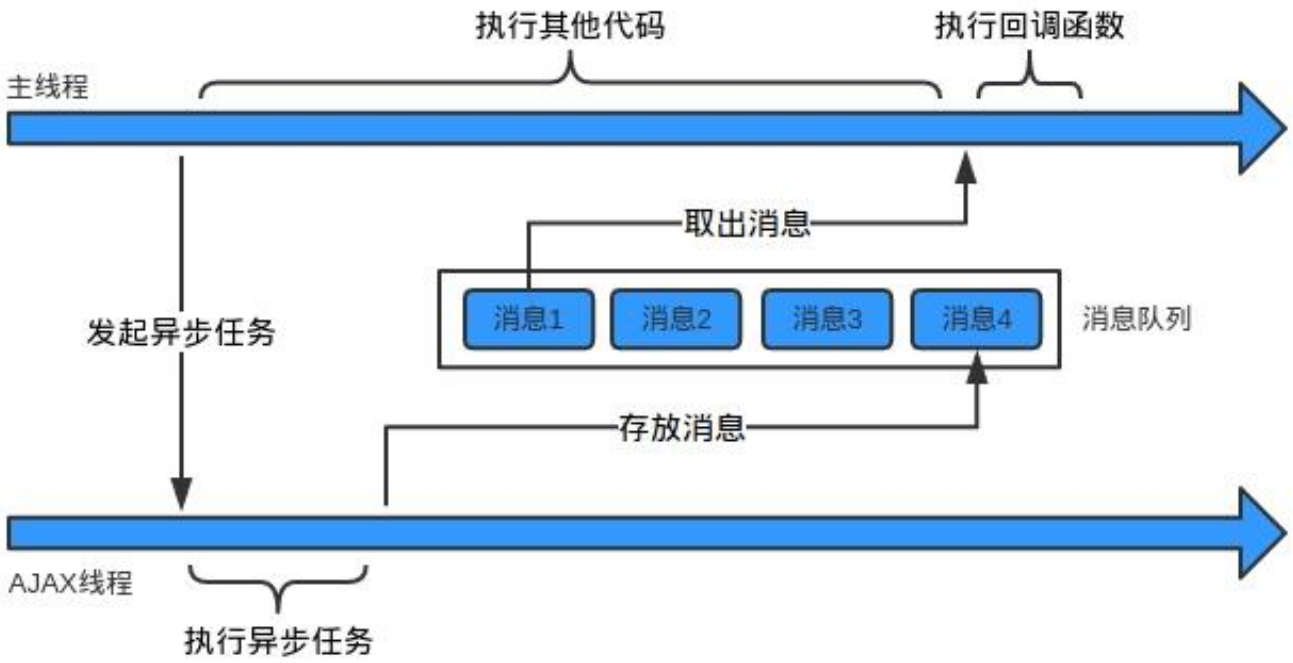


图 – 消息队列与事件循环

No. 4 事件处理

异步与事件

为什么是「事件」循环？ —— 消息队列中的每条消息实际上都对应着一个事件。

```
const button = document.getElementById('#btn')
button.addEventListener('click', function(e) {
  console.log(e)
})
```

- 从事件的角度来看，上述代码表示：在按钮上添加了一个鼠标单击事件的事件监听器；当用户点击按钮时，鼠标单击事件触发，事件监听器函数被调用。
- 从异步过程的角度看，addEventListener 函数就是异步过程的发起函数，function(e) 就是异步过程的回调函数。事件触发时，表示异步任务完成，会将 function(e) 封装成一条消息放到消息队列中，等待主线程执行。
- 事件的概念实际上并不是必须的，事件机制实际上就是异步过程的通知机制。它的存在是为了编程接口对开发者更友好。

感谢聆听