# 图解浏览器的工作原理（史上最全）

zhangwang

作者丨zhangwang

编辑丨覃云

本文经作者授权转载，原文链接：

https://zhuanlan.zhihu.com/p/47407398

可能每一个前端工程师都想要理解浏览器的工作原理。

我们希望知道从在浏览器地址栏中输入 url 到页面展现的短短几秒内浏览器究竟做了什么；

我们希望了解平时常常听说的各种代码优化方案是究竟为什么能起到优化的作用；

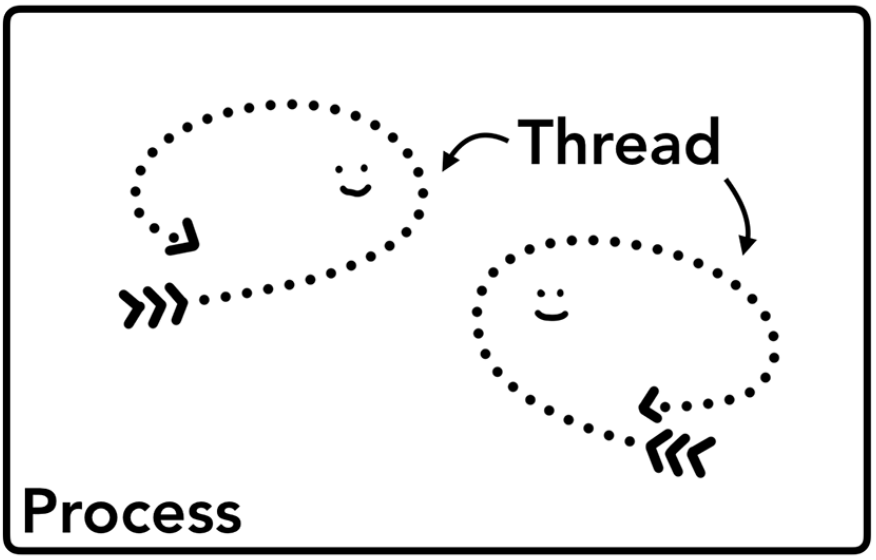
我们希望更细致地了解浏览器的渲染流程。

## 浏览器的多进程架构

一个好的程序常常被划分为几个相互独立又彼此配合的模块，浏览器也是如此，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。

对一些前端开发同学来说，进程和线程的概念可能会有些模糊，为了更好的理解浏览器的多进程架构，这里我们简单讨论一下进程和线程。

### 进程（process）和线程（thread）




进程就像是一个有边界的生产厂间，而线程就像是厂间内的一个个员工，可以自己做自己的事情，也可以相互配合做同一件事情。

当我们启动一个应用，计算机会创建一个进程，操作系统会为进程分配一部分内存，应用的所有状态都会保存在这块内存中，应用也许还会创建多个线程来辅助工作，这些线程可以共享这部分内存中的数据。如果应用关闭，进程会被终结，操作系统会释放相关内存。更生动的示意图如下：

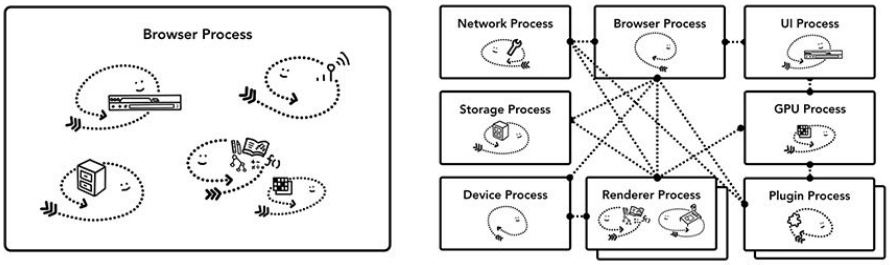
一个进程还可以要求操作系统生成另一个进程来执行不同的任务，系统会为新的进程分配独立的内存，两个进程之间可以使用 IPC （Inter Process Communication）进行通信。很多应用都会采用这样的设计，如果一个工作进程反应迟钝，重启这个进程不会影响应用其它进程的工作。

如果对进程及线程的理解还存在疑惑，可以参考下述文章：

http://www.ruanyifeng.com/blog/2013/04/processes_and_threads.html

## 浏览器的架构

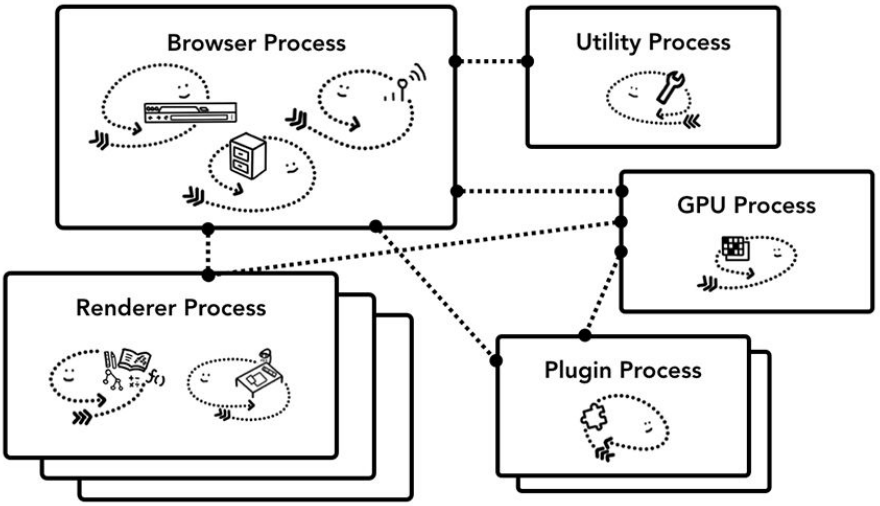
有了上面的知识做铺垫，我们可以更合理的讨论浏览器的架构了，其实如果要开发一个浏览器，它可以是单进程多线程的应用，也可以是使用 IPC 通信的多进程应用。




不同浏览器的架构模型

不同浏览器采用了不同的架构模式，这里并不存在标准，本文以 Chrome 为例进行说明 ：

Chrome 采用多进程架构，其顶层存在一个 Browser process 用以协调浏览器的其它进程。




Chrome 的不同进程

具体说来，Chrome 的主要进程及其职责如下：

Browser Process：

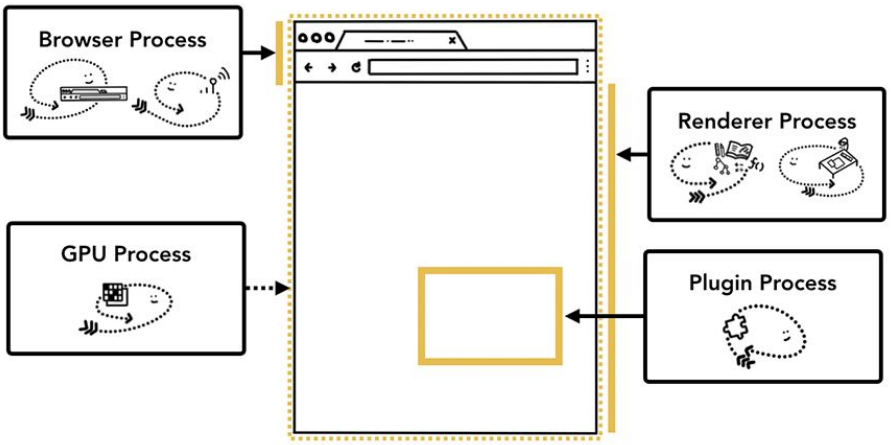
- 负责包括地址栏，书签栏，前进后退按钮等部分的工作；
- 负责处理浏览器的一些不可见的底层操作，比如网络请求和文件访问；

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 flashGPU Process
- 负责处理 GPU 相关的任务




不同进程负责的浏览器区域示意图

Chrome 还为我们提供了「任务管理器」，供我们方便的查看当前浏览器中运行的所有进程及每个进程占用的系统资源，右键单击还可以查看更多类别信息。

通过「页面右上角的三个点点点 --- 更多工具 --- 任务管理器」即可打开相关面板。

## 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。

为了节省内存，Chrome 限制了最多的进程数，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。

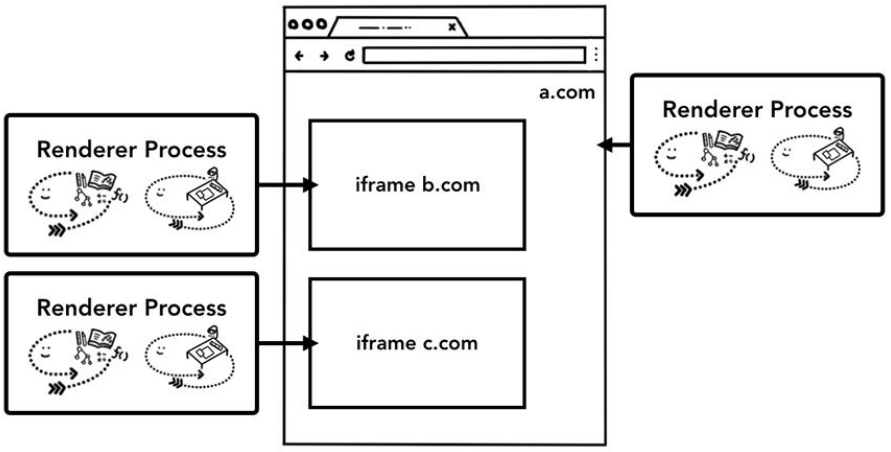
测试了一下在 Chrome 中打开不断打开知乎首页，在 Mac i5 8g 上可以启动四十多个渲染进程，之后新打开 tab 会合并到已有的渲染进程中。

Chrome 把浏览器不同程序的功能看做服务，这些服务可以方便的分割为不同的进程或者合并为一个进程。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，这样可以节省内存。

### iframe 的渲染 -- Site Isolation

在上面的进程图中我们还可以看到一些进程下还存在着 Subframe，这就是 Site Isolation 机制作用的结果。

Site Isolation 机制从 Chrome 67 开始默认启用。这种机制允许在同一个 Tab 下的跨站 iframe 使用单独的进程来渲染，这样会更为安全。




iframe 会采用不同的渲染进程

Site Isolation 被大家看做里程碑式的功能， 其成功实现是多年工程努力的结果。Site Isolation 不是简单的叠加多个进程。这种机制在底层改变了 iframe 之间通信的方法，Chrome 的其它功能都需要做对应的调整，比如说 devtools 需要相应的支持，甚至 Ctrl + F 也需要支持。关于 Site Isolation 的更多内容可参考下述链接：

https://developers.google.com/web/updates/2018/07/site-isolation

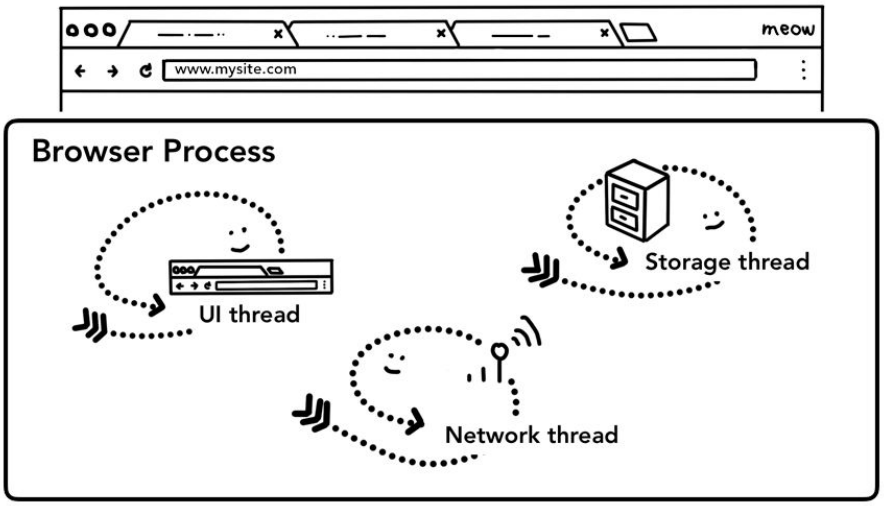
介绍完了浏览器的基本架构模式，接下来我们看看一个常见的导航过程对浏览器来说究竟发生了什么。

## 导航过程发生了什么

也许大多数人使用 Chrome 最多的场景就是在地址栏输入关键字进行搜索或者输入地址导航到某个网站，我们来看看浏览器是怎么看待这个过程的。

我们知道浏览器 Tab 外的工作主要由 Browser Process 掌控，Browser Process 又对这些工作进一步划分，使用不同线程进行处理：

- 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；




浏览器主进程中的不同线程

回到我们的问题，当我们在浏览器地址栏中输入文字，并点击回车获得页面内容的过程在浏览器看来可以分为以下几步：

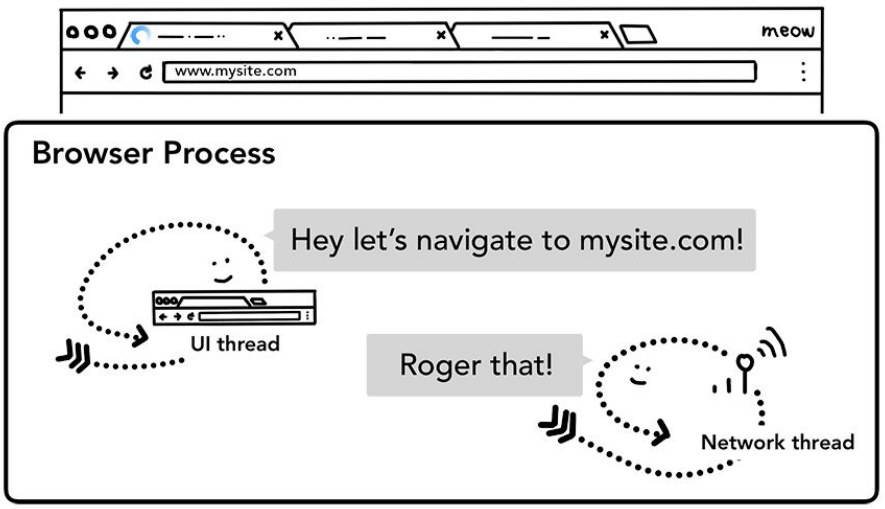
**1. 处理输入**

UI thread 需要判断用户输入的是 URL 还是 query；

**1. 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。

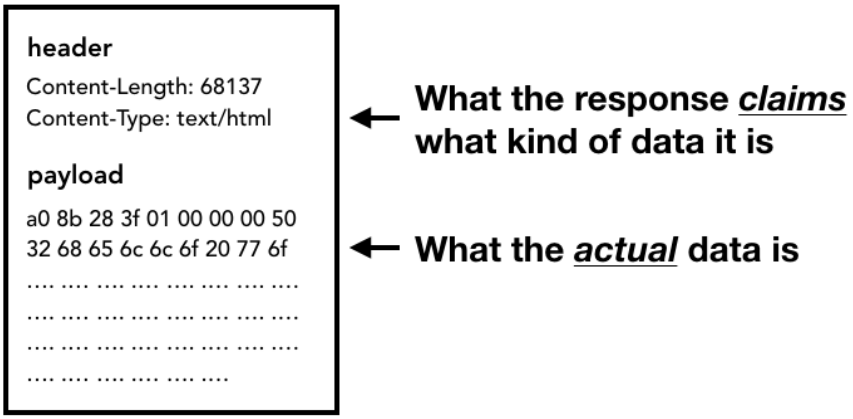
network thread 会执行 DNS 查询，随后为请求建立 TLS 连接。



如果 network thread 接收到了重定向请求头如 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。

**1. 读取响应**

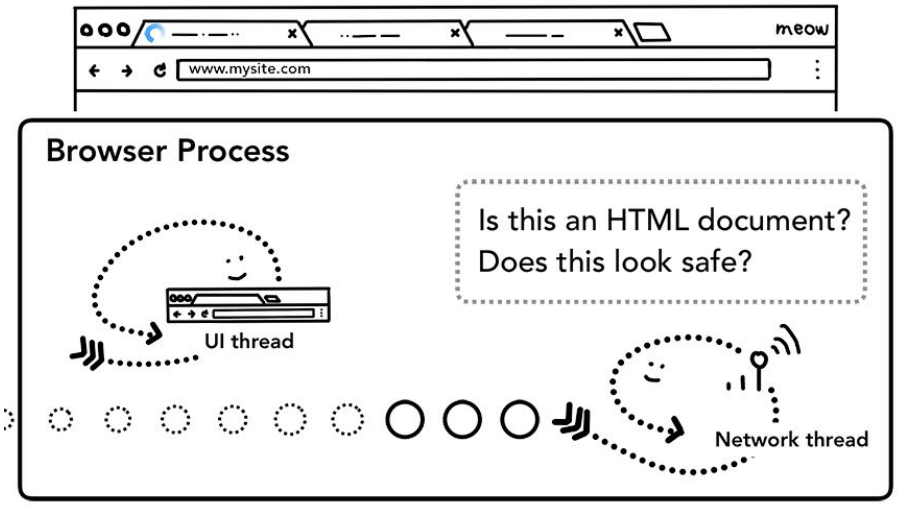
当请求响应返回的时候，network thread 会依据 Content-Type 及 MIME Type sniffing 判断响应内容的格式。




判断响应内容的格式

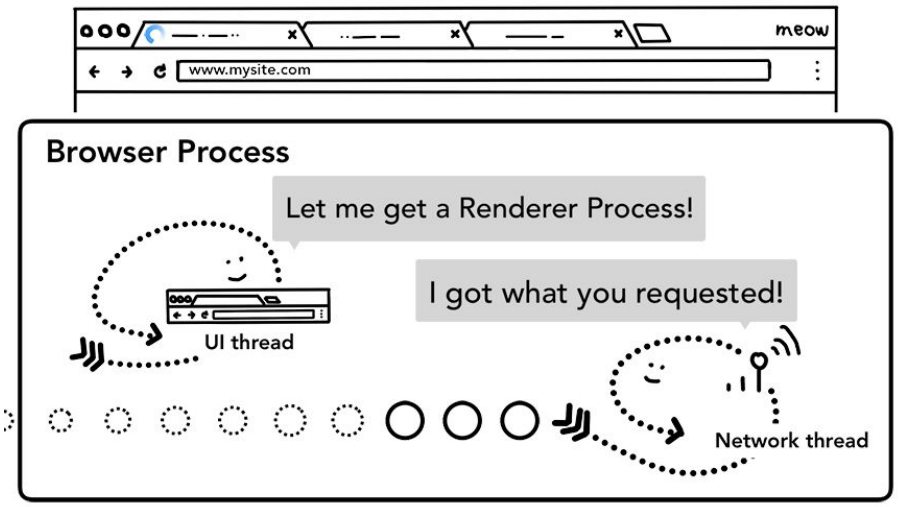
如果响应内容的格式是 HTML ，下一步将会把这些数据传递给 renderer process，如果是 zip 文件或者其它文件，会把相关数据传输给下载管理器。

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。




**1. 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 进行网页的渲染。

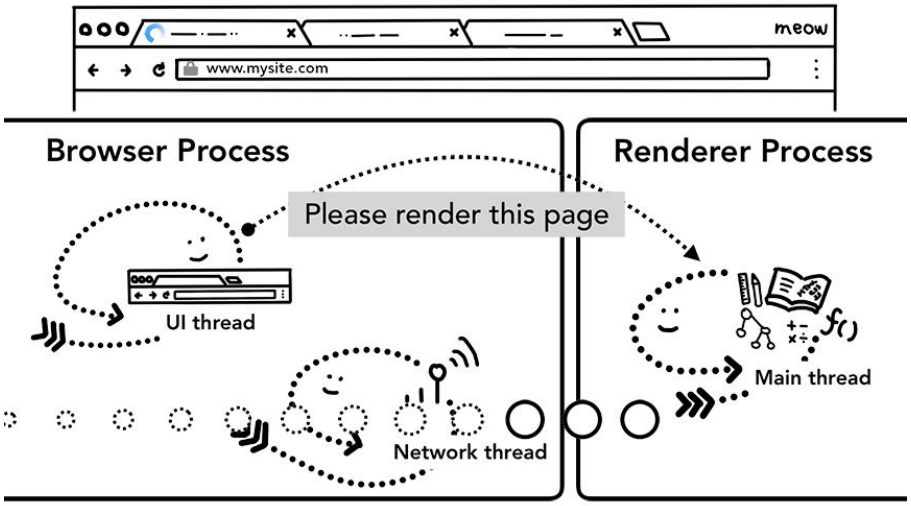



收到 Network thread 返回的数据后，UI thread 查找相关的渲染进程由于网络请求获取响应需要时间，这里其实还存在着一个加速方案。当 UI thread 发送 URL 请求给 network thread 时，浏览器其实已经知道了将要导航到那个站点。UI thread 会并行的预先查找和启动一个渲染进程，如果一切正常，当 network thread 接收到数据时，渲染进程已经准备就绪了，但是如果遇到重定向，准备好的渲染进程也许就不可用了，这时候就需要重启一个新的渲染进程。

**1. 确认导航**

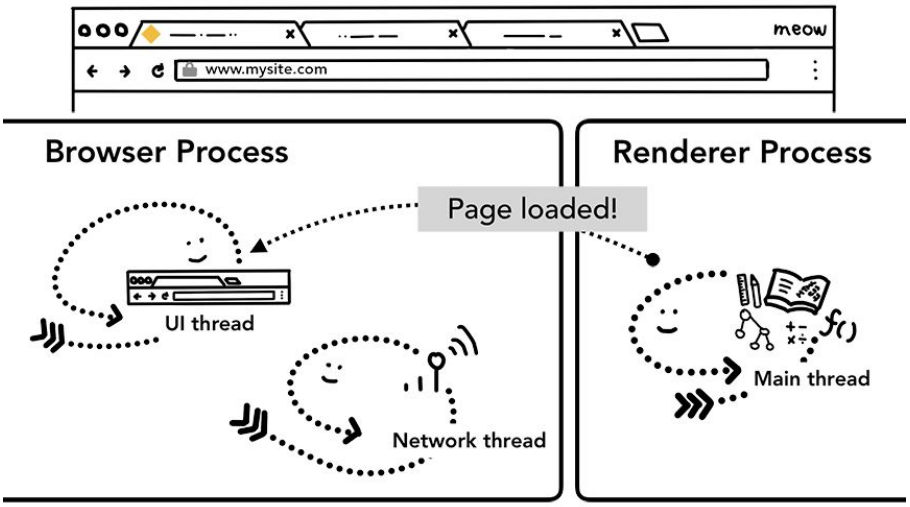
进过了上述过程，数据以及渲染进程都可用了， Browser Process 会给 renderer process 发送 IPC 消息来确认导航，一旦 Browser Process 收到 renderer process 的渲染确认消息，导航过程结束，页面加载过程开始。

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。




**1. 额外的步骤**

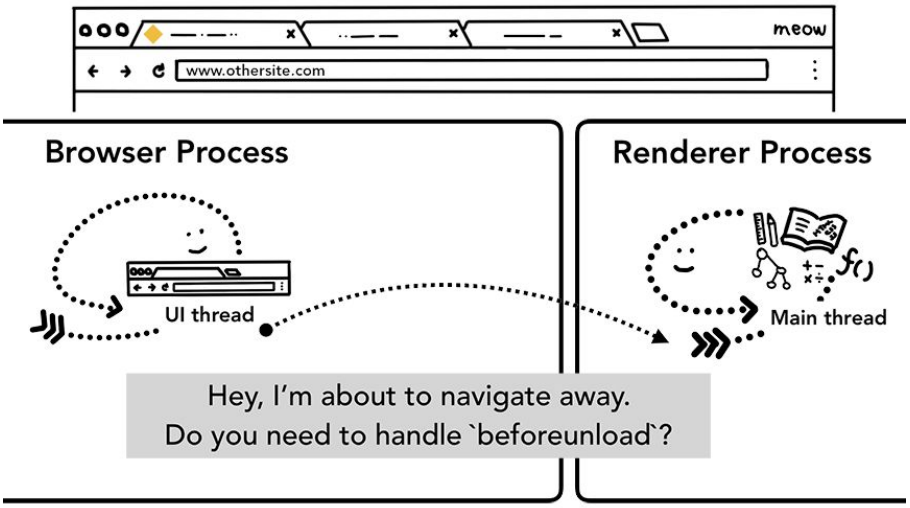
一旦导航被确认，renderer process 会使用相关的资源渲染页面，下文中我们将重点介绍渲染流程。当 renderer process 渲染结束（渲染结束意味着该页面内的所有的页面，包括所有 iframe 都触发了 onload 时），会发送 IPC 信号到 Browser process， UI thread 会停止展示 tab 中的 spinner。




Renderer Process 发送 IPC 消息通知 browser process 页面已经加载完成

当然上面的流程只是网页首帧渲染完成，在此之后，客户端依旧可下载额外的资源渲染出新的视图。

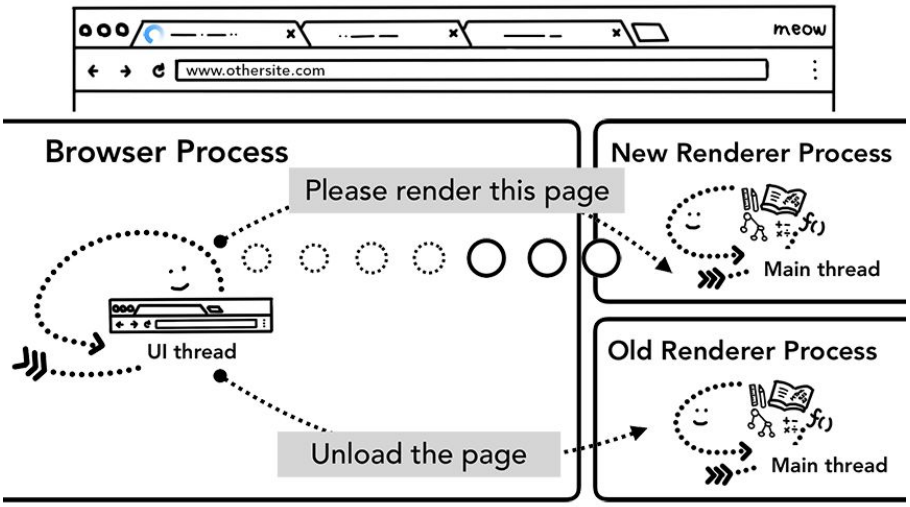
在这里我们可以明确一点，所有的 JS 代码其实都由 renderer Process 控制的，所以在你浏览网页内容的过程大部分时候不会涉及到其它的进程。不过也许你也曾经监听过 beforeunload 事件，这个事件再次涉及到 Browser Process 和 renderer Process 的交互，当当前页面关闭时（关闭 Tab ，刷新等等），Browser Process 需要通知 renderer Process 进行相关的检查，对相关事件进行处理。




浏览器进程发送 IPC 消息给渲染进程，通知要离开当前网站了如果导航由 renderer process 触发（比如在用户点击某链接，或者 JS 执行 window.location = "http://newsite.com" ） renderer process 会首先检查是否有 beforeunload 事件处理器，导航请求由 renderer process 传递给 Browser process。

如果导航到新的网站，会启用一个新的 render process 来处理新页面的渲染，老的进程会留下来处理类似 unload 等事件。

关于页面的生命周期，更多内容可参考 Page Lifecycle API 。

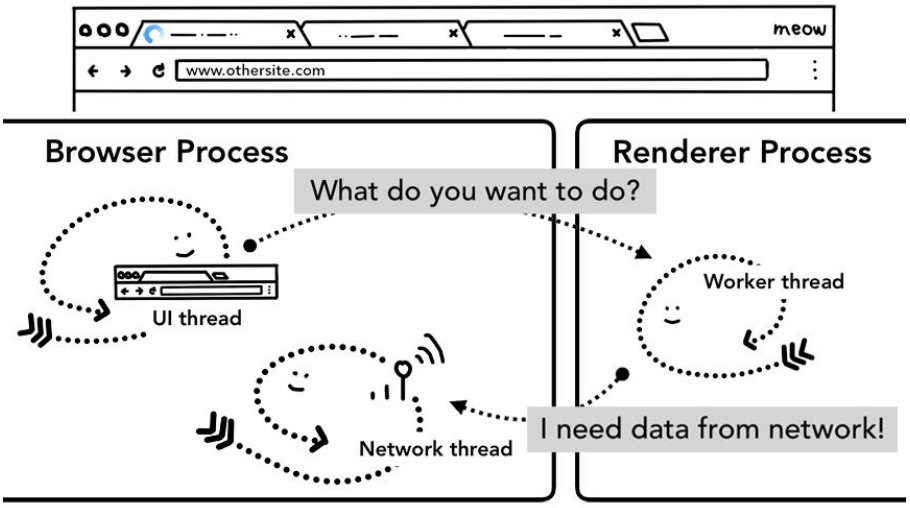



浏览器进程发送 IPC 消息到新的渲染进程通知渲染新的页面，同时通知旧的渲染进程卸载

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。

值得注意的是 service worker 也是运行在渲染进程中的 JS 代码，因此对于拥有 Service Worker 的页面，上述流程有些许的不同。

当有 Service Worker 被注册时，其作用域会被保存，当有导航时，network thread 会在注册过的 Service Worker 的作用域中检查相关域名，如果存在对应的 Service worker，UI thread 会找到一个 renderer process 来处理相关代码，Service Worker 可能会从 cache 中加载数据，从而终止对网络的请求，也可能从网上请求新的数据。




Service Worker 依据具体情形做处理

关于 Service Worker 的更多内容可参考：

https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle

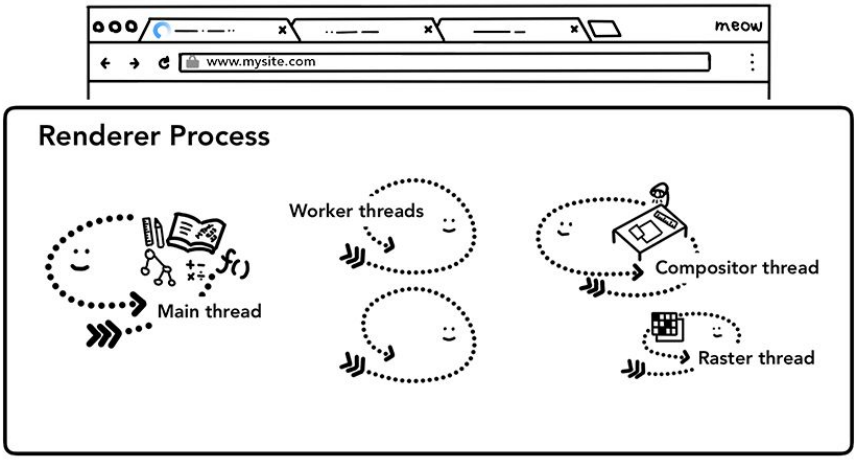
如果 Service Worker 最终决定通过网上获取数据，Browser 进程 和 renderer 进程的交互其实会延后数据的请求时间 。Navigation Preload 是一种与 Service Worker 并行的加速加载资源的机制，服务端通过请求头可以识别这类请求，而做出相应的处理。

更多内容可参考：

https://developers.google.com/web/updates/2017/02/navigation-preload

## 渲染进程是如何工作的

渲染进程几乎负责 Tab 内的所有事情，渲染进程的核心目的在于转换 HTML CSS JS 为用户可交互的 web 页面。渲染进程中主要包含以下线程：




渲染进程包含的线程

1. 主线程 Main thread
2. 工作线程 Worker thread
3. 排版线程 Compositor thread
4. 光栅线程 Raster thread

后文我们将逐步介绍不同线程的职责，在此之前我们先看看渲染的流程。

**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。

**1. 加载次级的资源**

网页中常常包含诸如图片，CSS，JS 等额外的资源，这些资源需要从网络上或者 cache 中获取。主进程可以在构建 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。

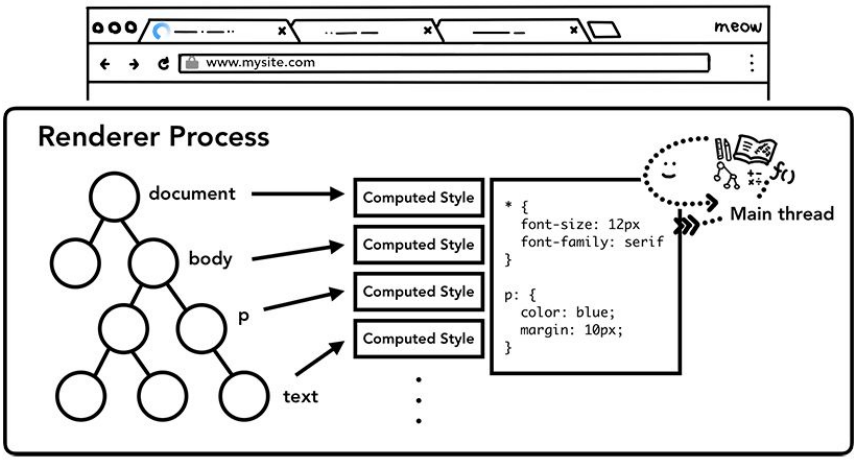
3. JS 的下载与执行

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 documwnt.write()等 API）。

不过开发者其实也有多种方式来告知浏览器应对如何应对某个资源，比如说如果在<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。更多的方法可参考 Resource Prioritization - Getting the Browser to Help You。

**1. 样式计算**

仅仅渲染 DOM 还不足以获知页面的具体样式，主进程还会基于 CSS 选择器解析 CSS 获取每一个节点的最终的计算样式值。即使不提供任何 CSS，浏览器对每个元素也会有一个默认的样式。

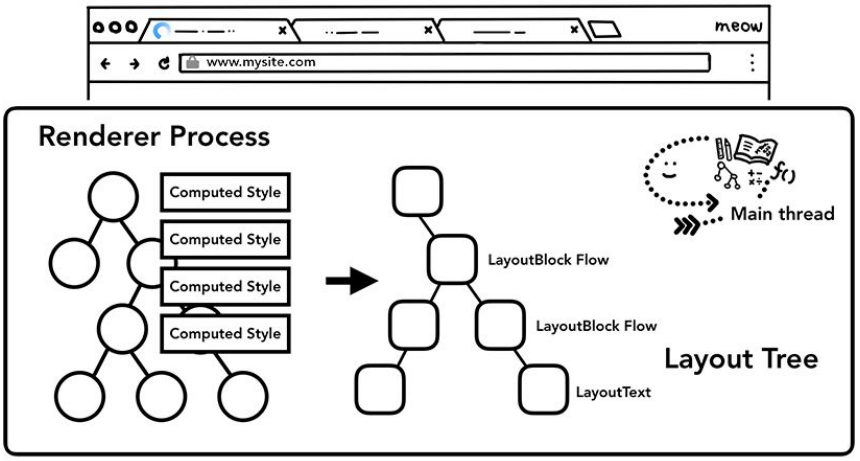



渲染进程主线程计算每一个元素节点的最终样式值

**1. 获取布局**

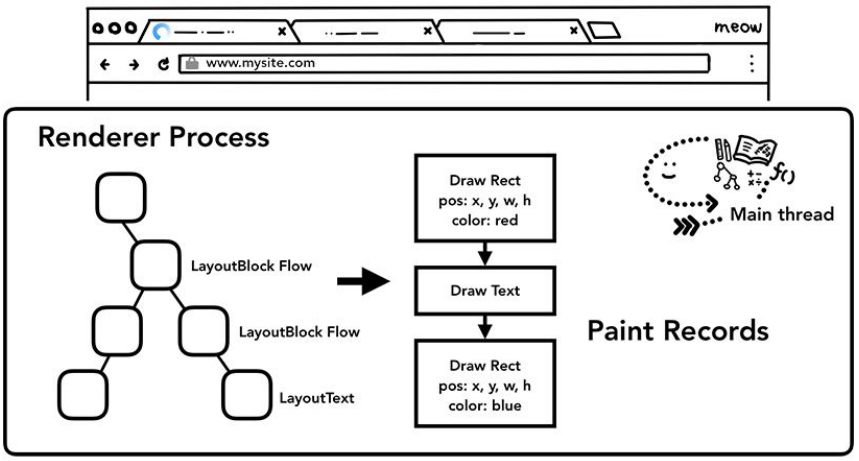
想要渲染一个完整的页面，除了获知每个节点的具体样式，还需要获知每一个节点在页面上的位置，布局其实是找到所有元素的几何关系的过程。其具体过程如下：

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。布局树和 DOM 树类似，但是其中只包含页面可见的元素，如果一个元素设置了 display:none ，这个元素不会出现在布局树上，伪元素虽然在 DOM 树上不可见，但是在布局树上是可见的。




**1. 绘制各元素**

即使知道了不同元素的位置及样式信息，我们还需要知道不同元素的绘制先后顺序才能正确绘制出整个页面。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。



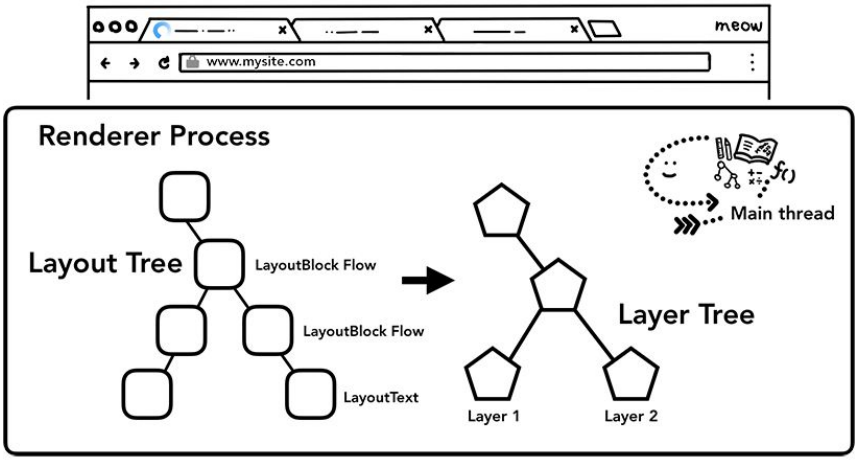

主线程依据布局树构建绘制记录

**1. 合成帧**

熟悉 PS 等绘图软件的童鞋肯定对图层这一概念不陌生，现代 Chrome 其实利用了这一概念来组合不同的层。

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（合成器线程）完成。

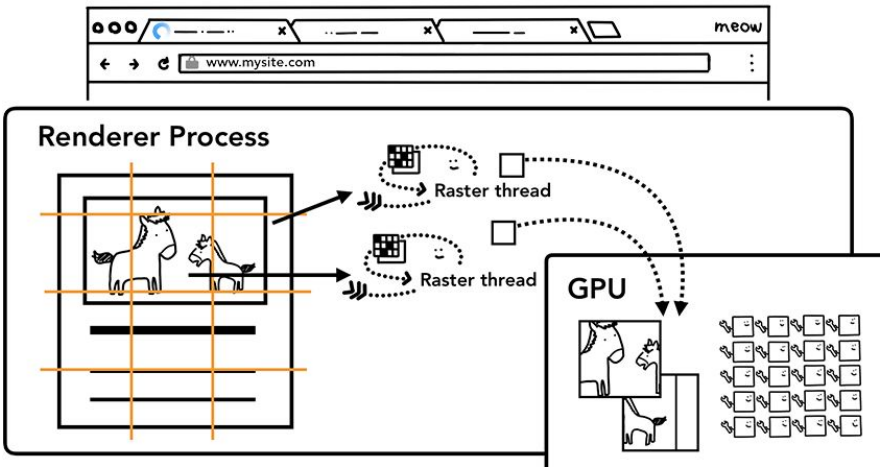
主线程会遍历布局树来创建层树（layer tree），添加了 will-change CSS 属性的元素，会被看做单独的一层。




主线程遍历布局树生成层树

你可能会想给每一个元素都添加上 will-change，不过组合过多的层也许会比在每一帧都栅格化页面中的某些小部分更慢。为了更合理的使用层，可参考 坚持仅合成器的属性和管理层计数 。

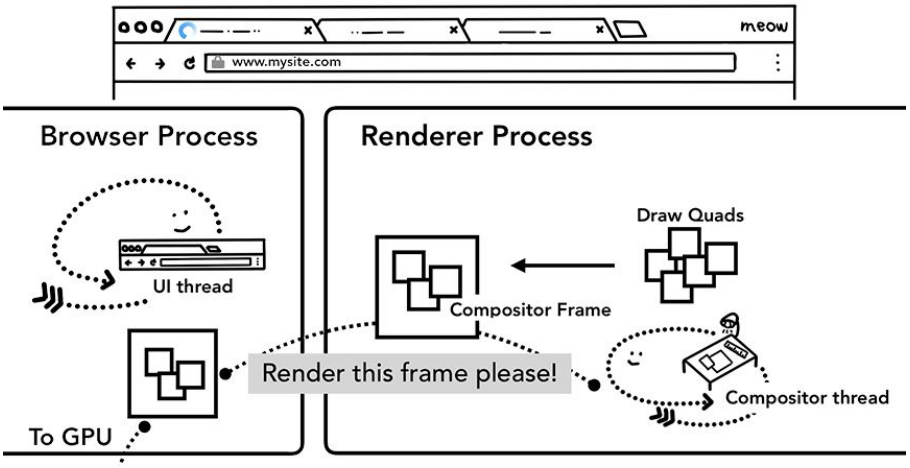
一旦层树被创建，渲染顺序被确定，主线程会把这些信息通知给合成器线程，合成器线程会栅格化每一层。有的层的可以达到整个页面的大小，因此，合成器线程将它们分成多个磁贴，并将每个磁贴发送到栅格线程，栅格线程会栅格化每一个磁贴并存储在 GPU 显存中。




栅格线程会栅格化每一个磁贴并存储在 GPU 显存中

一旦磁贴被光栅化，合成器线程会收集称为绘制四边形的磁贴信息以创建合成帧。

合成帧随后会通过 IPC 消息传递给浏览器进程，由于浏览器的 UI 改变或者其它拓展的渲染进程也可以添加合成帧，这些合成帧会被传递给 GPU 用以展示在屏幕上，如果滚动发生，合成器线程会创建另一个合成帧发送给 GPU。




合成器线程会发送合成帧给 GPU 渲染

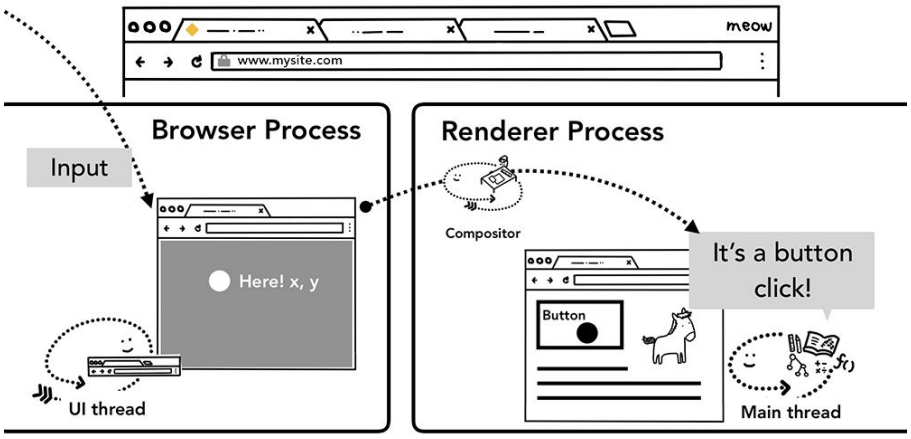
合成器的优点在于，其工作无关主线程，合成器线程不需要等待样式计算或者 JS 执行，这就是为什么合成器相关的动画 最流畅，如果某个动画涉及到布局或者绘制的调整，就会涉及到主线程的重新计算，自然会慢很多。

## 浏览器对事件的处理

浏览器通过对不同事件的处理来满足各种交互需求，这一部分我们一起看看从浏览器的视角，事件是什么，在此我们先主要考虑鼠标事件。

在浏览器的看来，用户的所有手势都是输入，鼠标滚动，悬置，点击等等都是。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。

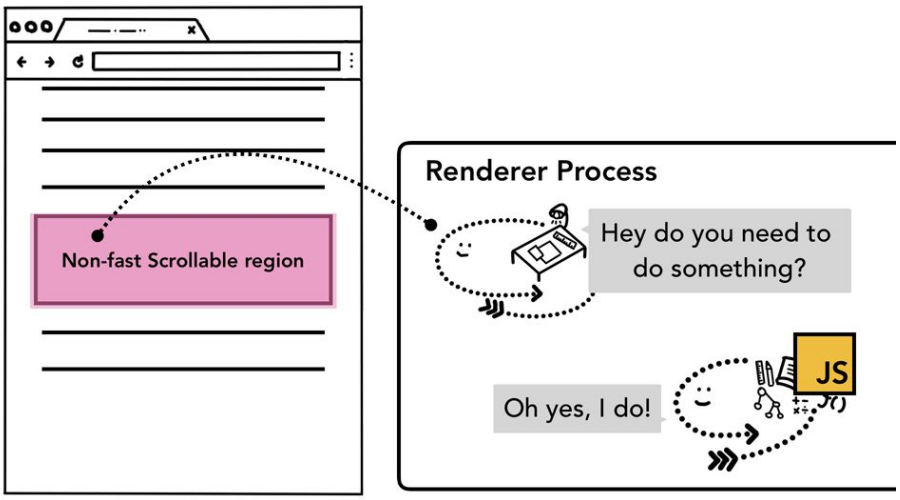
事件发生时，浏览器进程会发送事件类型及相应的坐标给渲染进程，渲染进程随后找到事件对象并执行所有绑定在其上的相关事件处理函数。




事件从浏览器进程传送给渲染进程

前文中，我们提到过合成器可以独立于主线程之外通过合成栅格化层平滑的处理滚动。如果页面中没有绑定相关事件，组合器线程可以独立于主线程创建组合帧。如果页面绑定了相关事件处理器，主线程就不得不出来工作了。这时候合成器线程会怎么处理呢？

这里涉及到一个专业名词「理解非快速滚动区域（non-fast scrollable region）」由于执行 JS 是主线程的工作，当页面合成时，合成器线程会标记页面中绑定有事件处理器的区域为 non-fast scrollable region ，如果存在这个标注，合成器线程会把发生在此处的事件发送给主线程，如果事件不是发生在这些区域，合成器线程则会直接合成新的帧而不用等到主线程的响应。



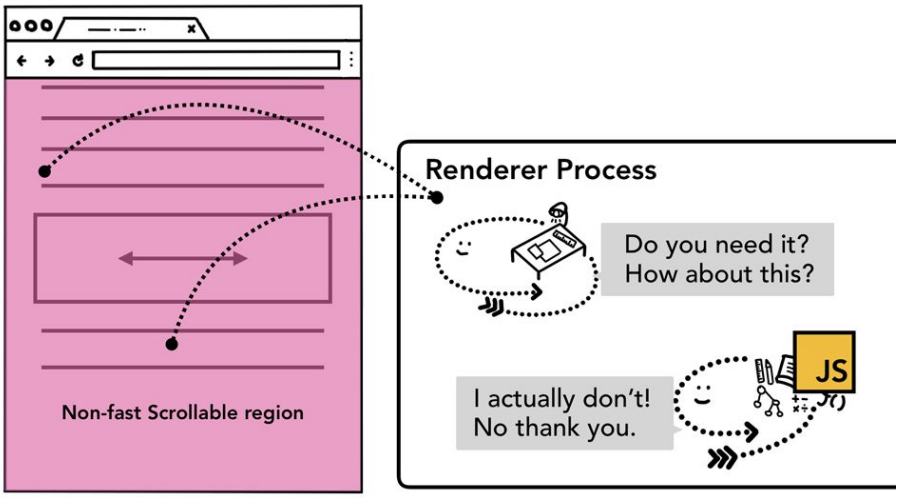

涉及 non-fast scrollable region 的事件，合成器线程会通知主线程进行相关处理

web 开发中常用的事件处理模式是事件委托，基于事件冒泡，我们常常在最顶层绑定事件：

```
document.body.addEventListener('touchstart', event => {
      if (event.target === area) {
            event.preventDefault();
      }
});
```

上述做法很常见，但是如果从浏览器的角度看，整个页面都成了 non-fast scrollable region 了。

这意味着即使操作的是页面无绑定事件处理器的区域，每次输入时，合成器线程也需要和主线程通信并等待反馈，流畅的合成器独立处理合成帧的模式就失效了。




由于事件绑定在最顶部，整个页面都成为了 non-fast scrollable region

为了防止这种情况，我们可以为事件处理器传递 passive: true 做为参数，这样写就能让浏览器即监听相关事件，又让组合器线程在等等主线程响应前构建新的组合帧。

```
document.body.addEventListener('touchstart', event => {
      if (event.target === area) {
            event.preventDefault()
      }
 }, {passive: true});
```

不过上述写法可能又会带来另外一个问题，假设某个区域你只想要水平滚动，使用 passive: true 可以实现平滑滚动，但是垂直方向的滚动可能会先于event.preventDefault()发生，此时可以通过 event.cancelable 来防止这种情况。

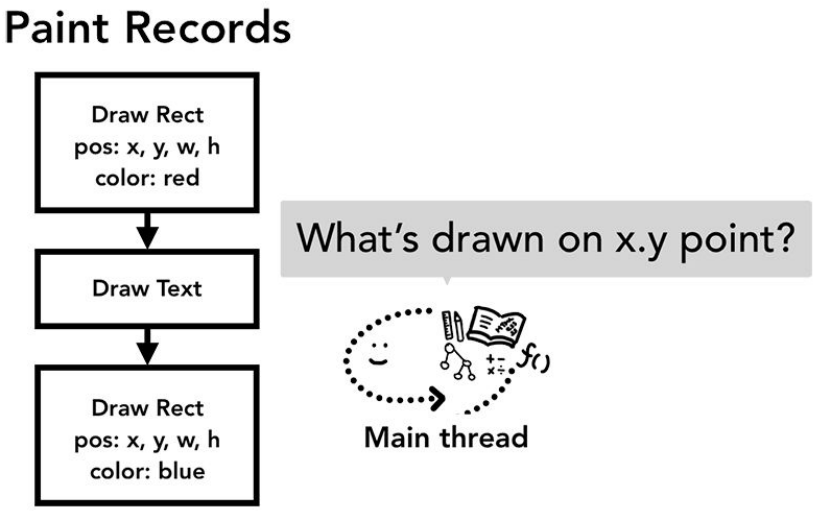
```
document.body.addEventListener('pointermove', event => {
      if (event.cancelable) {
            event.preventDefault(); // block the native scroll
            /*
            *   do what you want the application to do here
            */
      }
}, {passive: true});
```

也可以使用 css 属性 touch-action 来完全消除事件处理器的影响，如：

```
#area {
   touch-action: pan-x;
}
```

### 查找到事件对象

当组合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存在的元素。

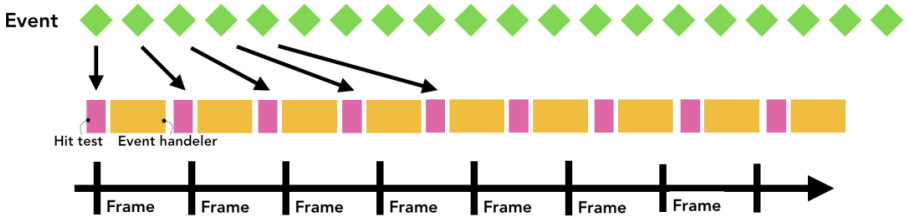



主线程依据绘制记录查找事件相关元素

### 事件的优化

一般我们屏幕的刷新速率为 60fps，但是某些事件的触发量会不止这个值，出于优化的目的，Chrome 会合并连续的事件 （如 wheel, mousewheel, mousemove, pointermove, touchmove ），并延迟到下一帧渲染时候执行。

而如 keydown, keyup, mouseup, mousedown, touchstart, 和 touchend 等非连续性事件则会立即被触发。

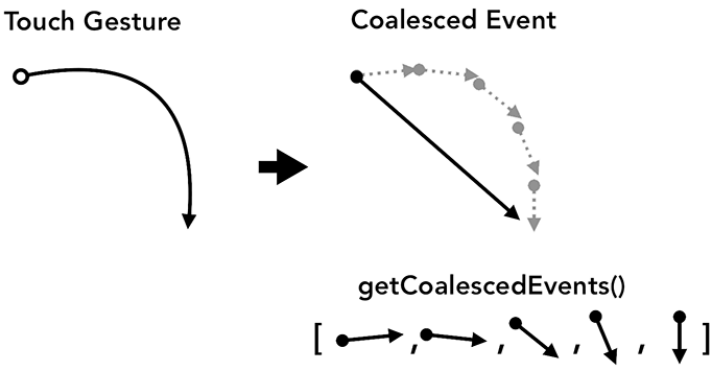



Chrome 会合并连续事件到下一帧触发

合并事件虽然能提示性能，但是如果你的应用是绘画等，则很难绘制一条平滑的曲线了，此时可以使用 getCoalescedEvents API 来获取组合的事件。示例代码如下：

```
window.addEventListener('pointermove', event => {
      const events = event.getCoalescedEvents();
```

```
    for (let event of events) {
        const x = event.pageX;
        const y = event.pageY;
        // draw a line using x and y coordinates.
    }
});
```




花了好久来整理上面的内容，整理的过程收获还挺大的，也希望这篇笔记能对你有所启发，如果有任何疑问，欢迎一起来讨论。

参考链接


- https://developers.google.com/web/updates/2018/09/inside-browser-part1
- https://developers.google.com/web/updates/2018/09/inside-browser-part2
- https://developers.google.com/web/updates/2018/09/inside-browser-part3
- https://developers.google.com/web/updates/2018/09/inside-browser-part4
- https://www.html5rocks.com/zh/tutorials/internals/howbrowserswork/#Layered_representation


活动推荐

[阅读原文]