深入了解现代网络浏览器（第 3 部分）

Mariko Kosaka

渲染程序进程的内部工作方式

这是 4 篇博客系列文章的第 3 部分，我们将介绍浏览器的工作原理。之前，我们介绍了多进程架构和导航流程。在这篇博文中，我们将了解渲染器进程内部会发生什么

渲染程序进程涉及 Web 性能的许多方面。由于渲染程序进程内部发生了很多事情，因此这篇博文仅作简要概述。如果您想深入了解，请参阅“网站开发基础”的“性能”部分，其中包含更多资源。

渲染程序进程处理 Web 内容

渲染程序进程负责在标签页中发生的一切。在渲染程序进程中，主线程会处理您发送给用户的大部分代码。如果您使用 Web Worker 或 Service Worker，则有时 JavaScript 的部分内容由工作器线程处理。合成器线程和光栅线程也在渲染器进程内运行，可高效且流畅地渲染网页。

渲染程序进程的核心任务是将 HTML、CSS 和 JavaScript 转换为用户可以与之互动的网页。

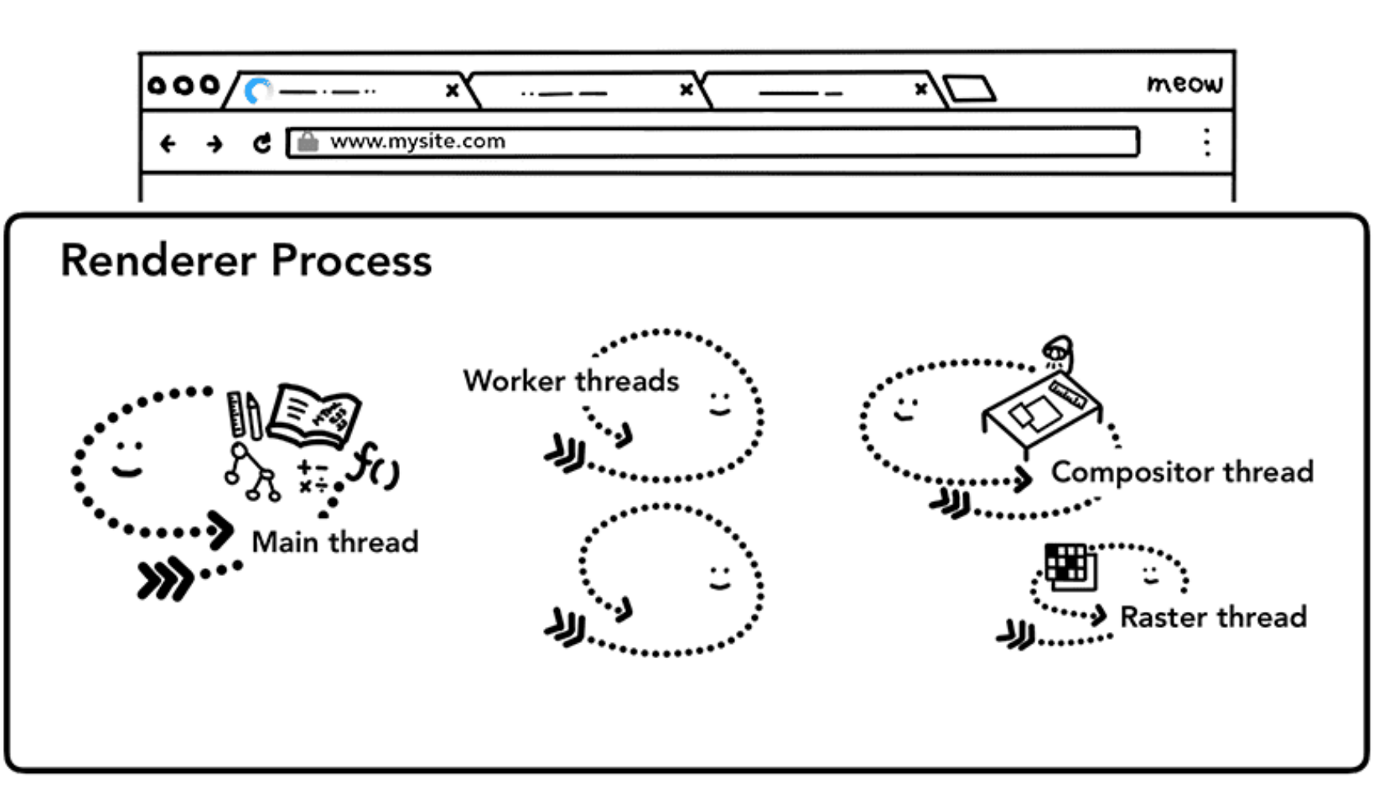


图 1：包含主线程、工作器线程、合成器线程和光栅线程的渲染程序进程

解析

DOM 构建

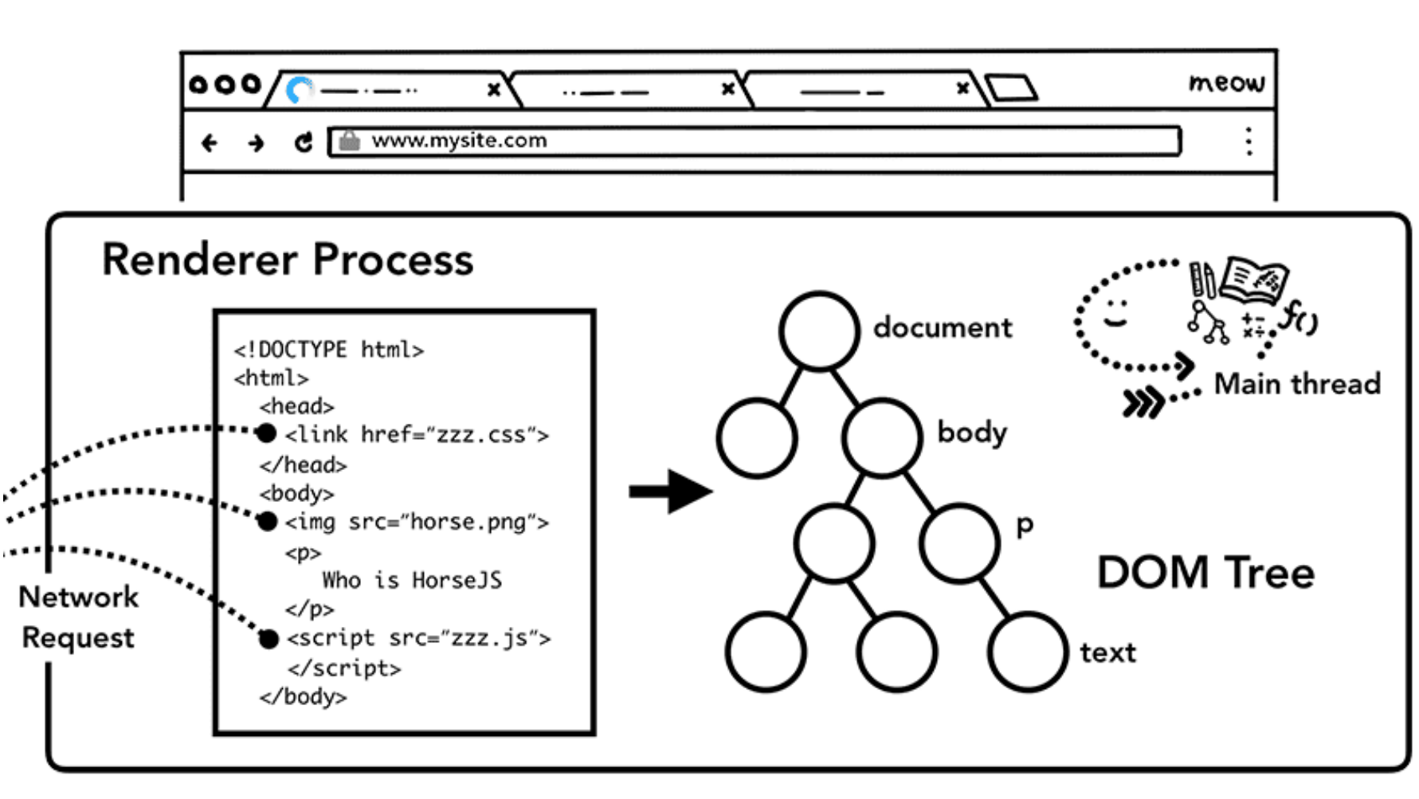
当渲染程序进程收到针对导航的提交消息并开始接收 HTML 数据时，主线程会开始解析文本字符串 (HTML)，并将其转变为 **D**ocument **O** 对象 **M**odel (**DOM**)。

DOM 是浏览器对页面的内部表示形式，也是 Web 开发者可通过 JavaScript 与之交互的数据结构和 API。

将 HTML 文档解析为 DOM 的操作由 HTML 标准定义。您可能已经注意到，将 HTML 提供给浏览器绝不会抛出错误。例如，缺少结束标记 `</p>` 就是有效的 HTML。诸如 `Hi! <b>I'm <i>Chrome</b>!</i>` 之类的错误标记（b 标记在 i 标记之前关闭）将被视为您编写了 `Hi! <b>I'm <i>Chrome</i></b><i>!</i>`。这是因为 HTML 规范旨在妥善处理这些错误。如果您想知道如何完成这些任务，请参阅 HTML 规范的“错误处理和解析器中的奇怪情况简介”部分。

子资源加载

网站通常会使用图片、CSS 和 JavaScript 等外部资源。这些文件需要从网络或缓存加载。主线程在解析以构建 DOM 时可以在找到它们时逐个请求，但为了加快速度，会并发运行“预加载扫描器”。如果 HTML 文档中有 `<img>` 或 `<link>` 之类的内容，预加载扫描程序会查看 HTML 解析器生成的令牌，并将请求发送到浏览器进程中的网络线程。



```
<!DOCTYPE html>
<html>
  <head>
  <link href="zzz.css">
  </head>
  <body>
  <img src="horse.png">
    <p>
      Who is HorseJS
    </p>
  <script src="zzz.js">
    </script>
  </body>
```

图 2：解析 HTML 并构建 DOM 树的主线程

JavaScript 可能会阻止解析

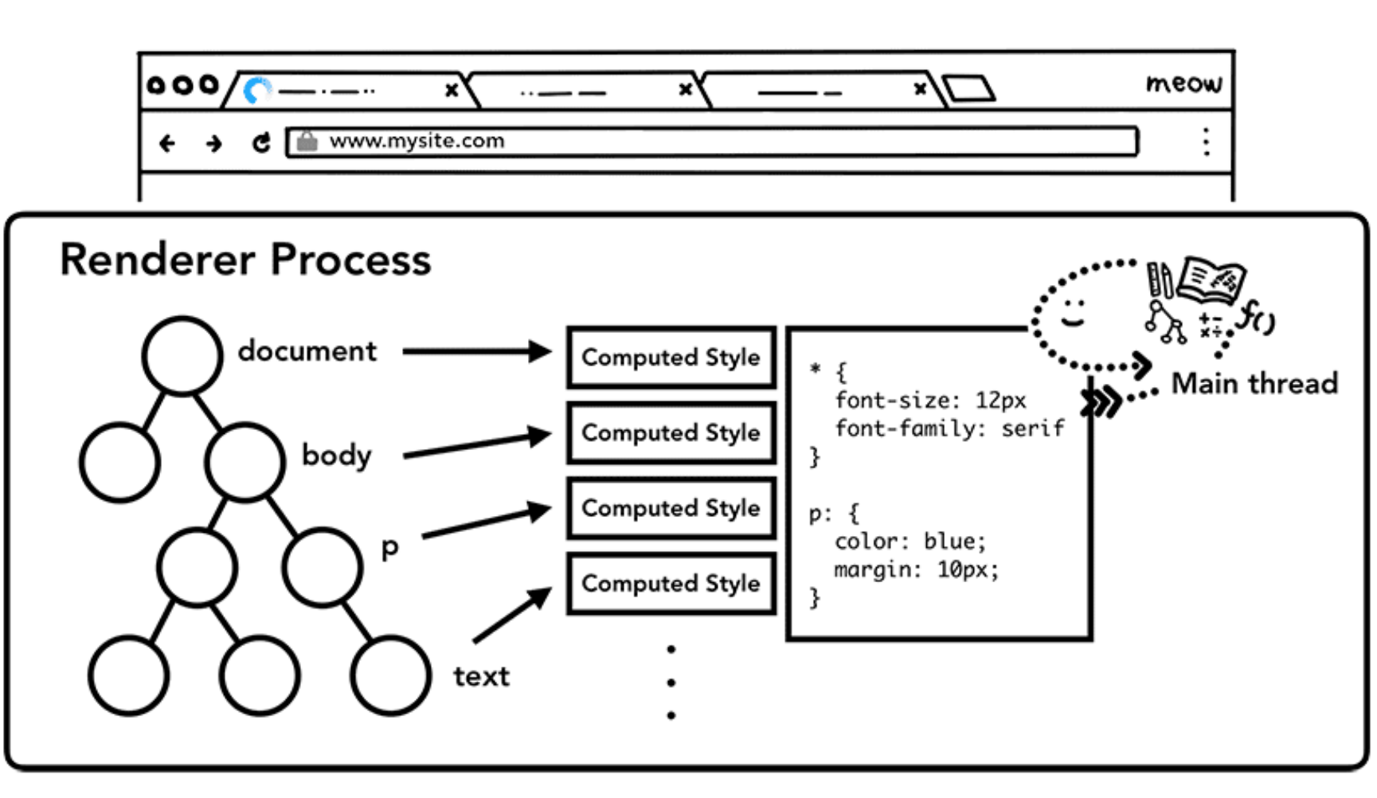
当 HTML 解析器找到 `<script>` 标记时，它会暂停 HTML 文档的解析，并且必须加载、解析和执行 JavaScript 代码。原因是什么？因为 JavaScript 可以使用 `document.write()` 之类的内容更改整个 DOM 结构之类的内容来更改文档形状（HTML 规范中的解析模型概览中有很好的示意图）。因此，HTML 解析器必须先等待 JavaScript 运行，然后才能继续解析 HTML 文档。如果您想知道在 JavaScript 执行过程中会发生什么，V8 团队会就此展开讨论和博文。

提示浏览器您希望如何加载资源

为了顺利加载资源，网络开发者可以通过多种方式向浏览器发送提示。 如果您的 JavaScript 不使用 `document.write()`，您可以向 `<script>` 标记添加 `async` 或 `defer` 属性。然后，浏览器会异步加载并运行 JavaScript 代码，而不会阻止解析。您也可以使用 JavaScript 模块（如果适用）。`<link rel="preload">` 用于告知浏览器当前导航肯定需要该资源，您希望尽快下载该资源。如需了解详情，请参阅资源优先级 – 让浏览器为您提供帮助。

样式计算

拥有 DOM 并不足以知道页面的显示效果，因为我们可以在 CSS 中设置页面元素的样式。主线程解析 CSS 并确定每个 DOM 节点的计算样式。这是关于根据 CSS 选择器将哪种样式应用于每个元素的信息。您可以在开发者工具的 `computed` 部分查看此信息。



```
* {
  font-size: 12px
  font-family: serif
}

p: {
  color: blue;
  margin: 10px;
}
```

图 3：解析 CSS 以添加计算样式的主线程

即使您未提供任何 CSS，每个 DOM 节点都有一个计算样式。`<h1>` 标记的显示区域大于 `<h2>` 标记，并为每个元素定义了外边距。这是因为浏览器具有默认的样式表。如果您想了解 Chrome 的默认 CSS 是什么，请点击此处查看源代码。

布局

现在，渲染程序进程知道文档的结构以及每个节点的样式，但这不足以渲染页面。假设您正尝试通过电话向朋友描述一幅画。“有一个大的红色圆圈和一个小的蓝色方块”是不够的信息，不足以让您的朋友知道这幅画究竟是什么样子。

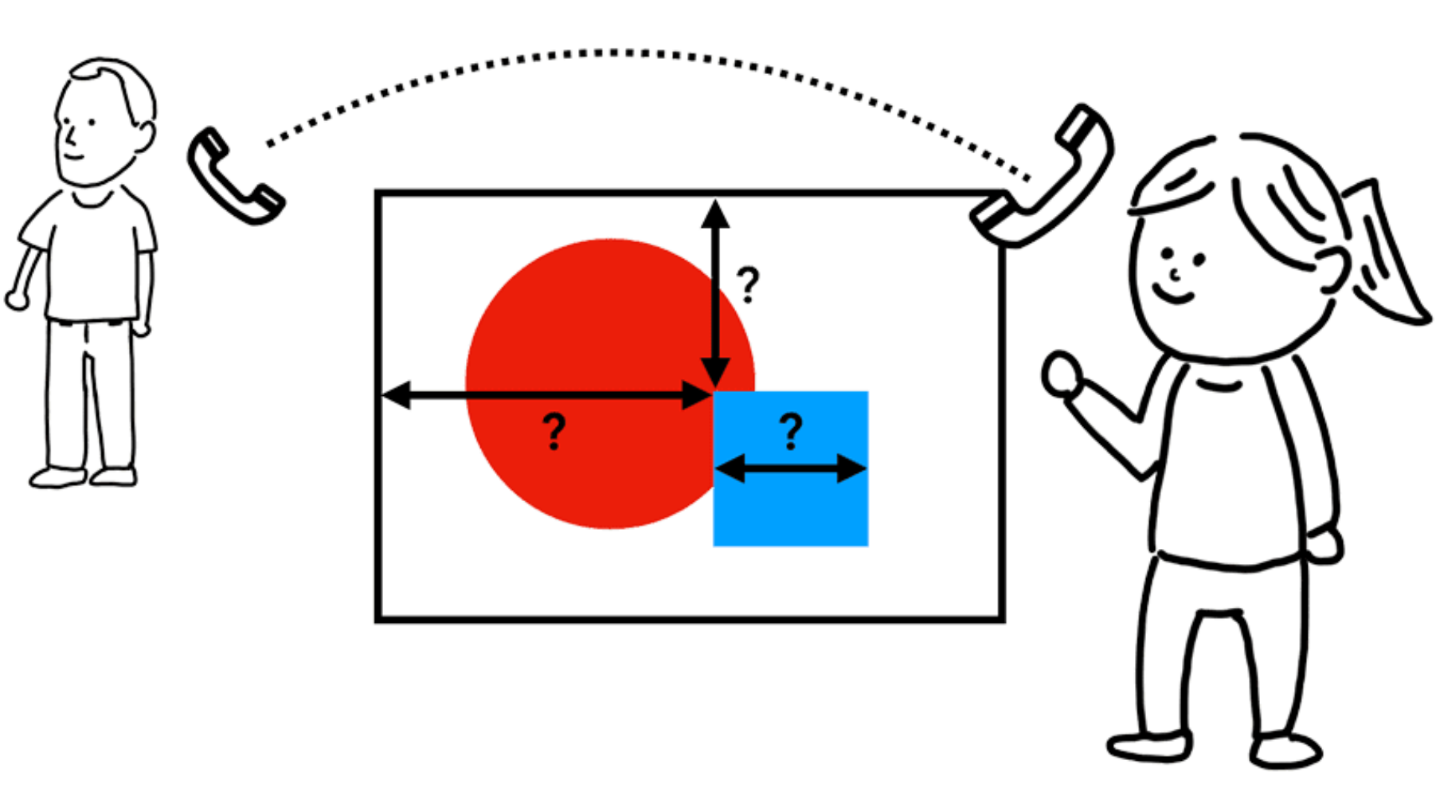


图 4：一个人站在一幅画前，电话线路连接到另一人

布局是查找元素几何图形的过程。主线程会遍历 DOM 和计算出的样式，并创建布局树，其中包含 x y 坐标和边界框大小等信息。布局树的结构可能与 DOM 树类似，但它仅包含与页面上可见内容相关的信息。如果应用了 `display: none`，则该元素不属于布局树（然而，具有 `visibility: hidden` 的元素在布局树中）。同样，如果应用包含类似 `p::before{content:"Hi!"}` 的伪元素，它就会包含在布局树中，即使它不在 DOM 中也是如此。

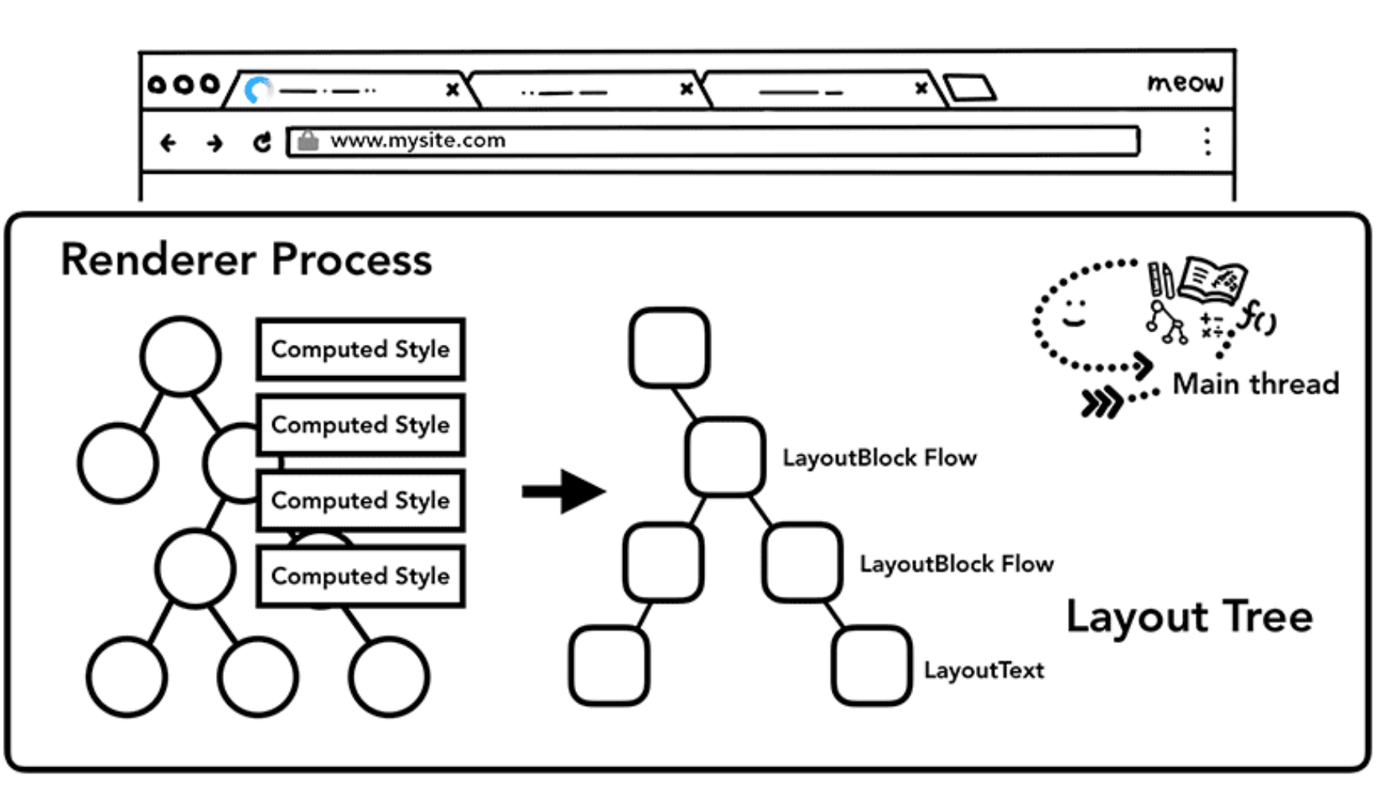


图 5：主线程遍历经过计算的样式并生成布局树的 DOM 树

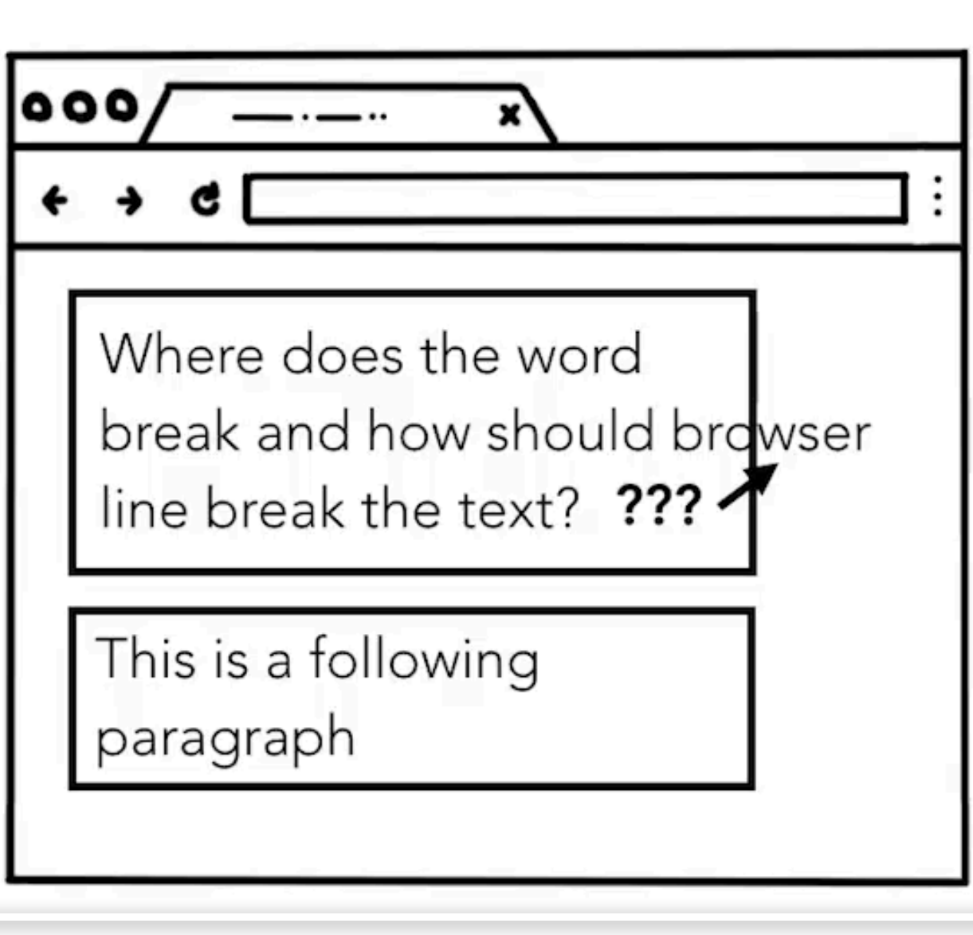


图 6：因换行变化而移动段落的 Box 布局

确定网页的布局是一项具有挑战性的任务。即使是最简单的页面布局（例如从上到下的块流），也必须考虑字体大小和换行位置，因为它们会影响段落的大小和形状；而后又会影响下一段落的位置。

CSS 可以使元素悬浮到一侧、遮盖溢出内容以及更改书写方向。可以想象，这个布局阶段有一项繁重的任务。在 Chrome 中，有整个工程师团队负责开发布局。如果您想详细了解他们的工作，可以观看 BlinkOn Conference 的一些演讲，并且非常有意思。

颜料

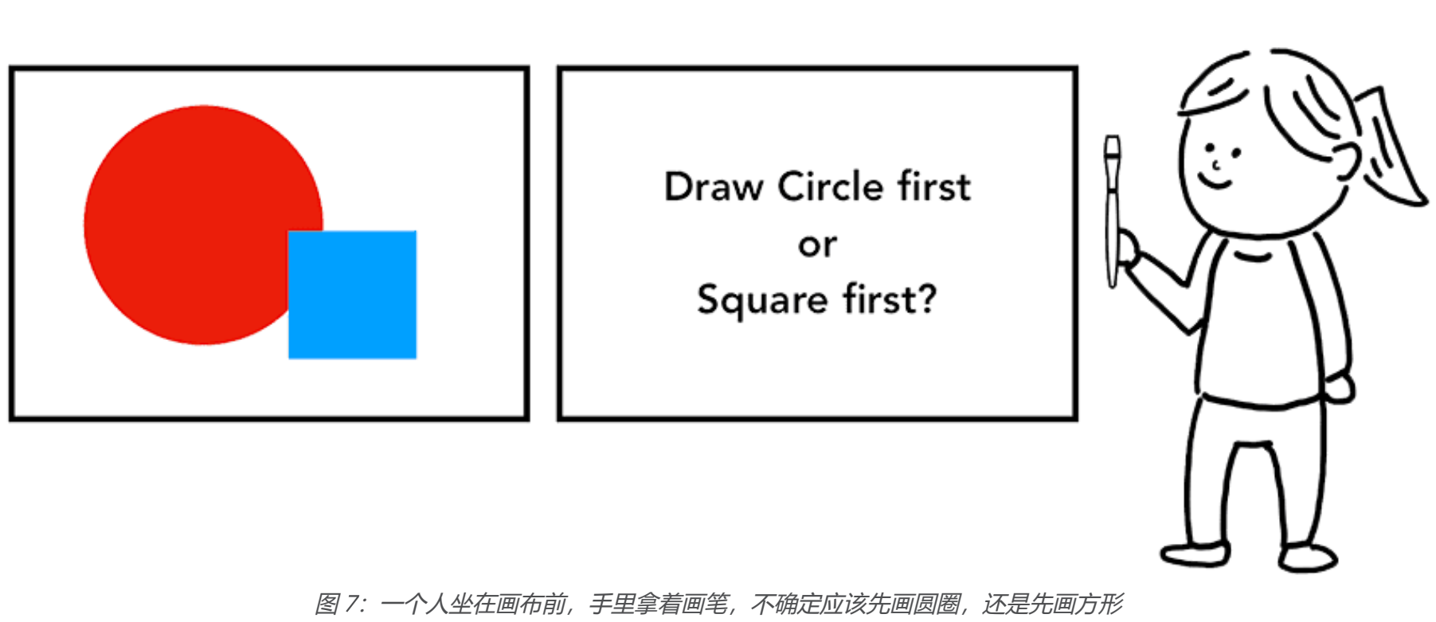
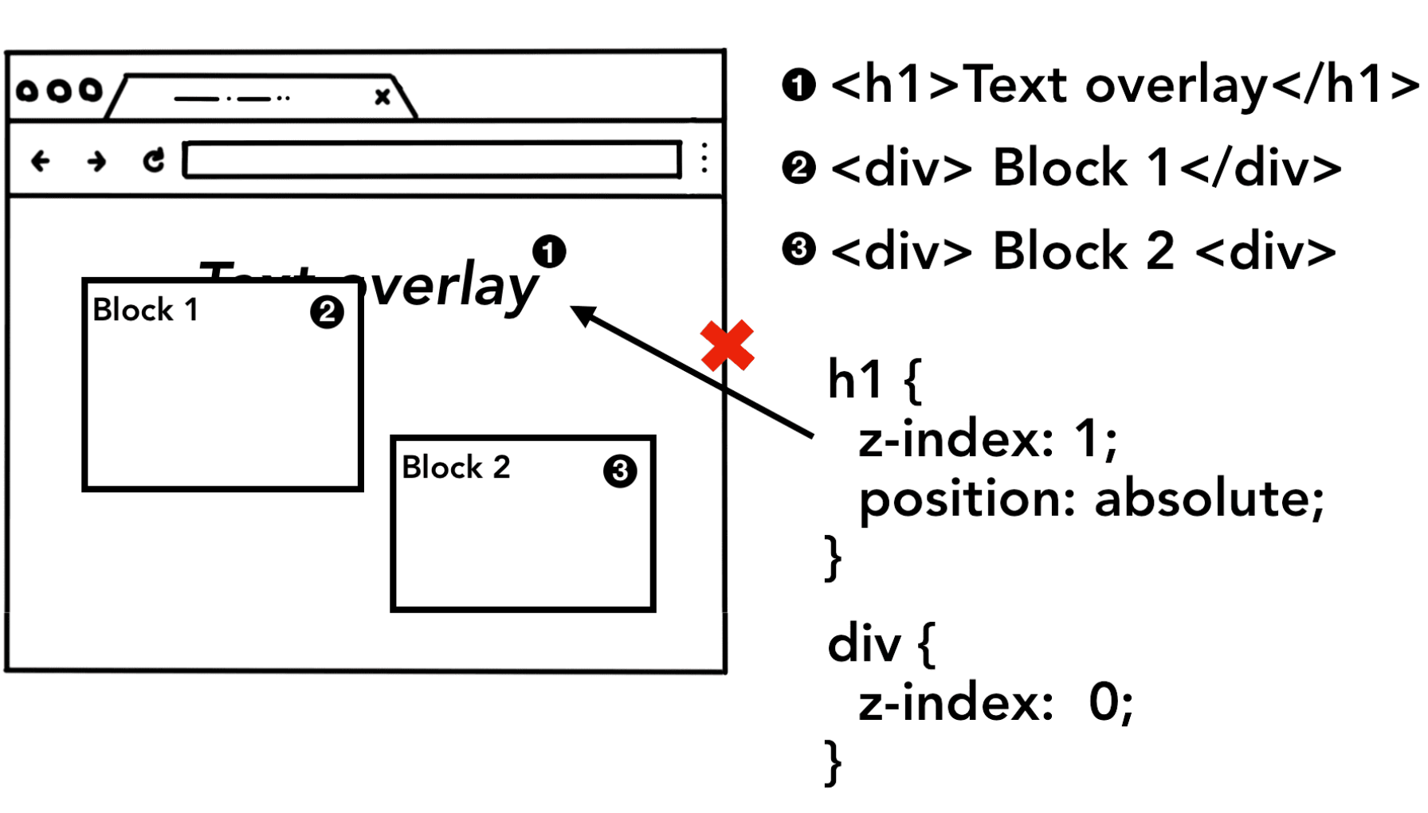


图 7：一个人坐在画布前，手里拿着画笔，不确定应该先画圆圈，还是先画方形

拥有 DOM、样式和布局仍然不足以渲染页面。假设您正尝试复制一幅画。您已经知道元素的大小、形状和位置，但仍需判断它们的绘制顺序。

例如，系统可能会为某些元素设置 `z-index`，在这种情况下，按 HTML 中编写的元素的顺序进行绘制会导致呈现错误。



```
❶ <h1>Text overlay</h1>
❷ <div> Block 1</div>
❸ <div> Block 2 <div>

h1 {
  z-index: 1;
  position: absolute;
}
div {
  z-index:  0;
}
```

图 8：页面元素按 HTML 标记顺序显示，由于未考虑 Z-index 值，导致呈现的图片有误

在此绘制步骤中，主线程会遍历布局树来创建绘制记录。绘制记录是绘制过程的备注，例如“先提供背景，然后是文本，最后是矩形”。如果您使用 JavaScript 在 `<canvas>` 元素上绘制了内容，那么您可能已经熟悉此过程。

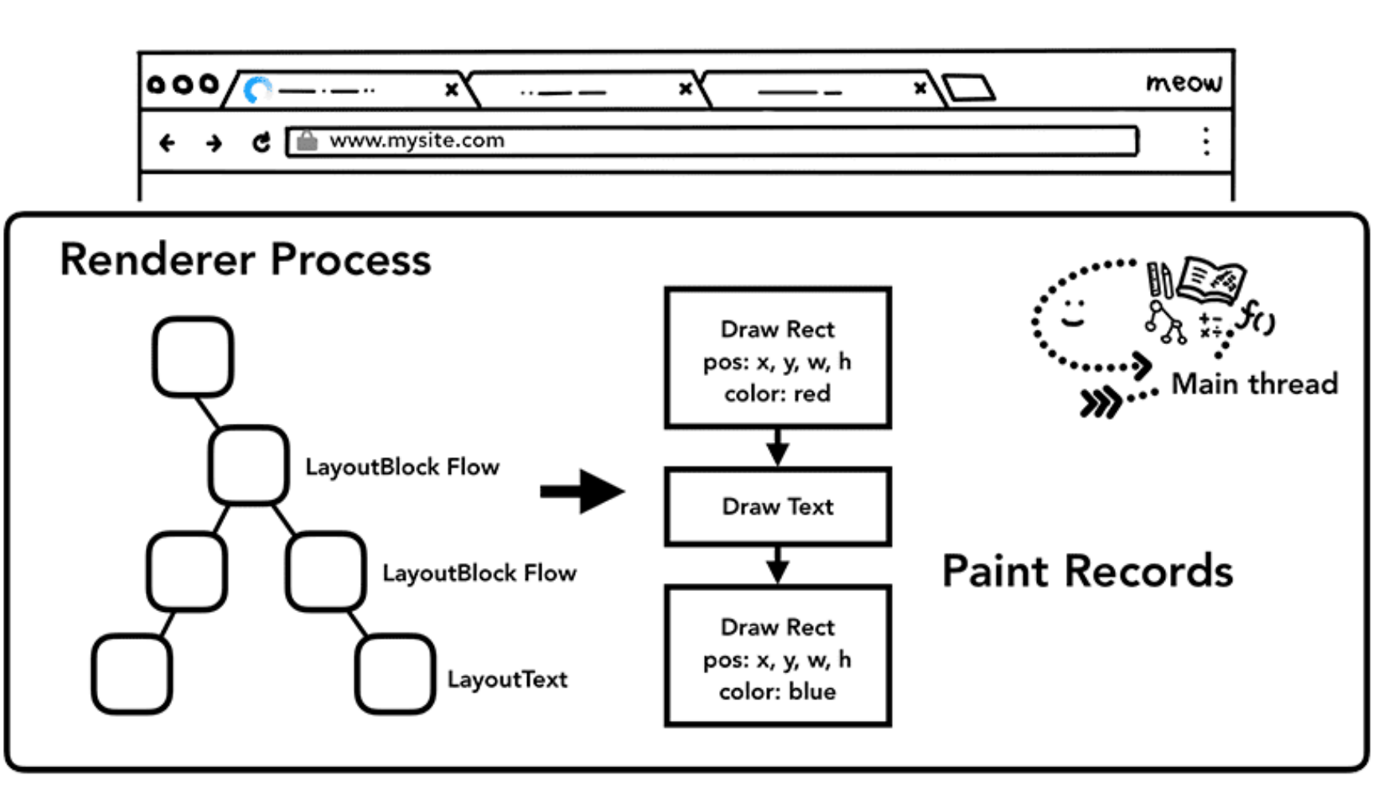


图 9：主线程遍历布局树并生成绘制记录

更新渲染流水线的成本很高

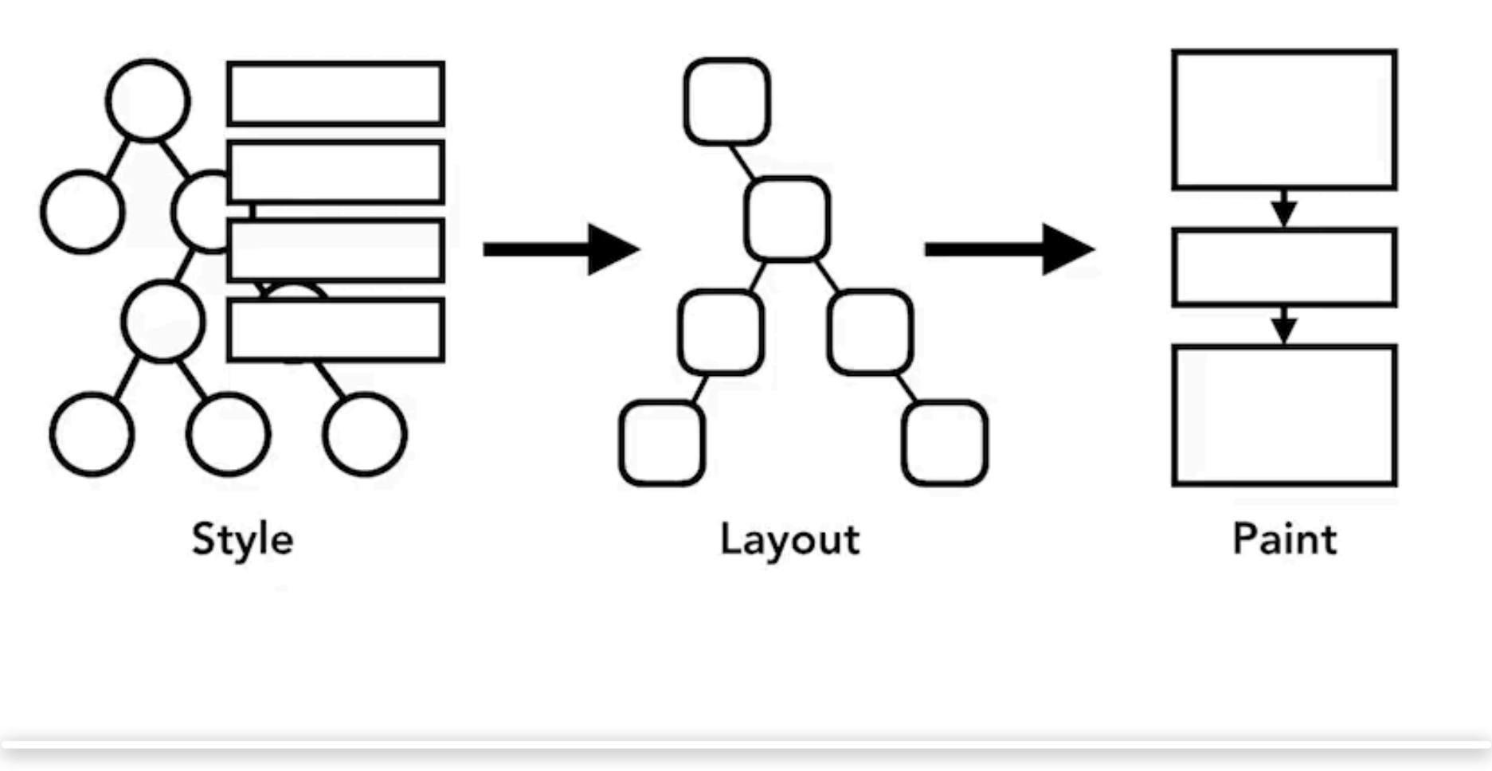


图 10：DOM+样式、布局和绘制树（按顺序排列）

在渲染流水线中，最重要的一点是，在每个步骤中，系统都会使用前一操作的结果创建新数据。例如，如果布局树中有一些变化，则需要为文档中受影响的部分重新生成绘制顺序。

如果您要为元素添加动画效果，浏览器必须在每一帧之间执行这些操作。我们的大多数显示屏每秒都会刷新屏幕 60 次 (60 fps)；当您在每一帧屏幕上在屏幕上移动内容时，动画将对用户来说非常流畅。但是，如果动画缺少其中帧，则页面将显得“卡顿”。

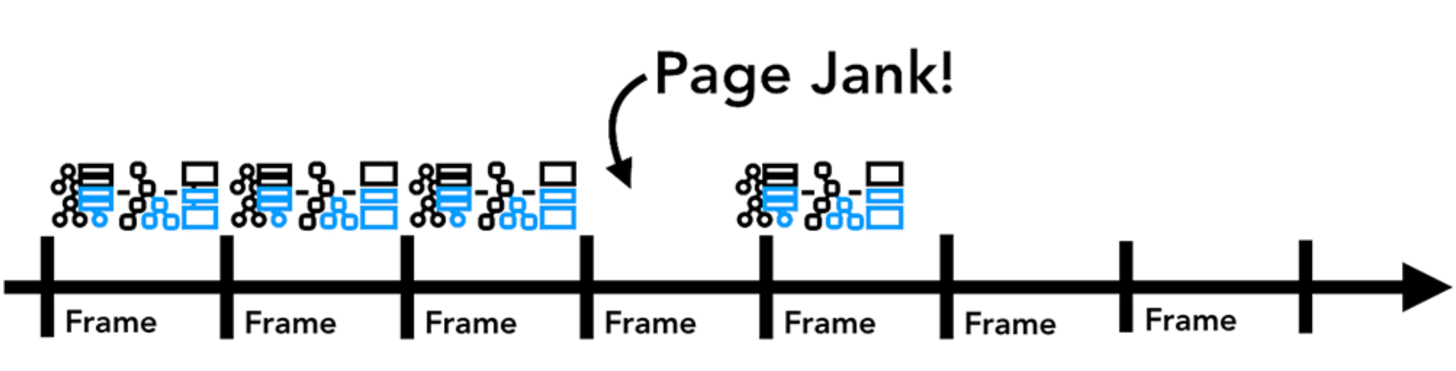


图 11：时间轴上的动画帧

即使您的渲染操作能够与屏幕刷新保持同步，这些计算也会在主线程上运行，这意味着当应用运行 JavaScript 时，这些计算可能会被阻塞。

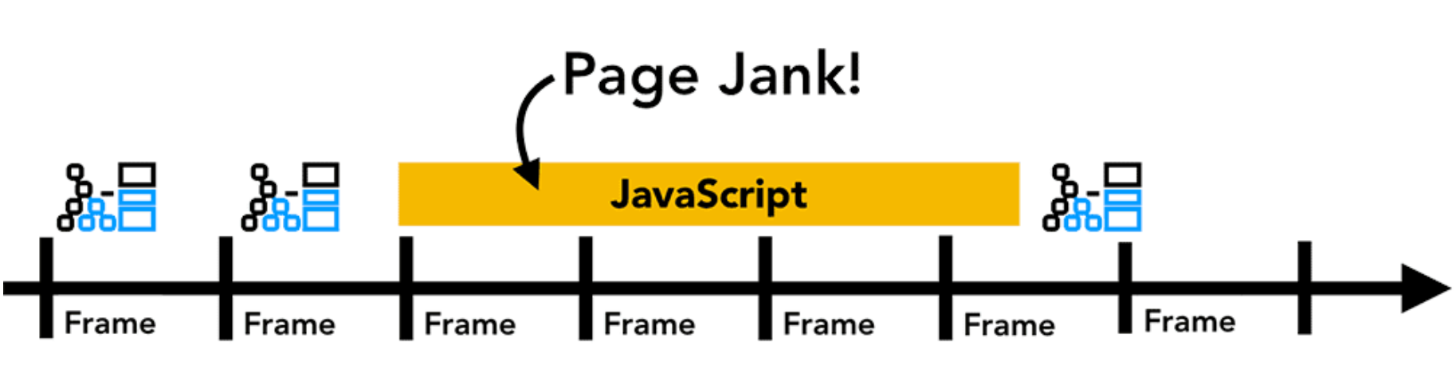


图 12：时间轴上的动画帧，但有一帧被 JavaScript 屏蔽

您可以将 JavaScript 操作分成几小块，并使用 `requestAnimationFrame()` 安排在每一帧运行。如需详细了解此主题，请参阅优化 JavaScript 执行。您还可以在 Web 工作器中运行 JavaScript，以避免阻塞主线程。

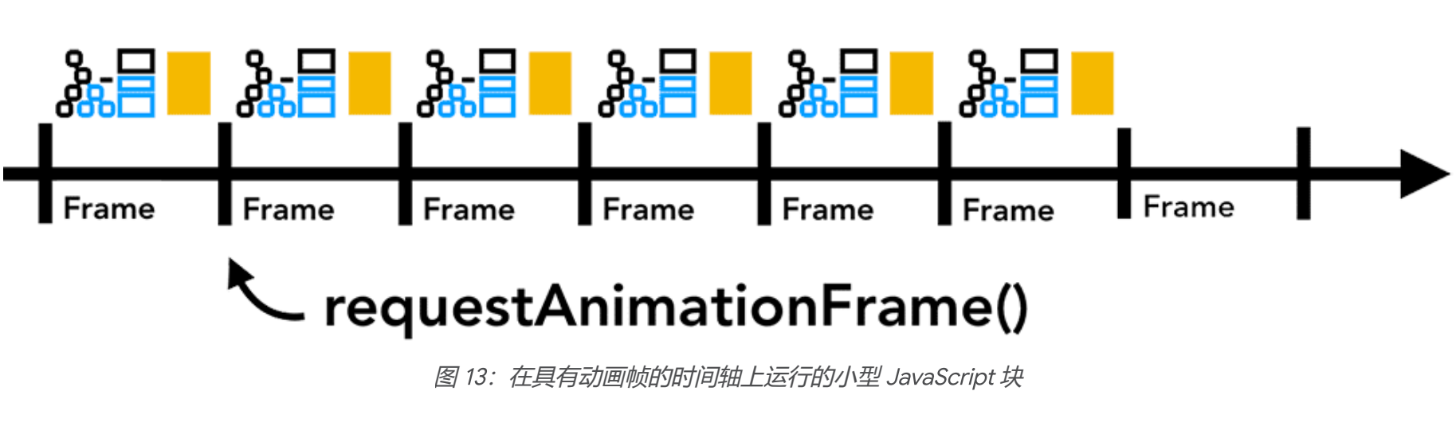


图 13：在具有动画帧的时间轴上运行的小型 JavaScript 块

合成

如何绘制页面？

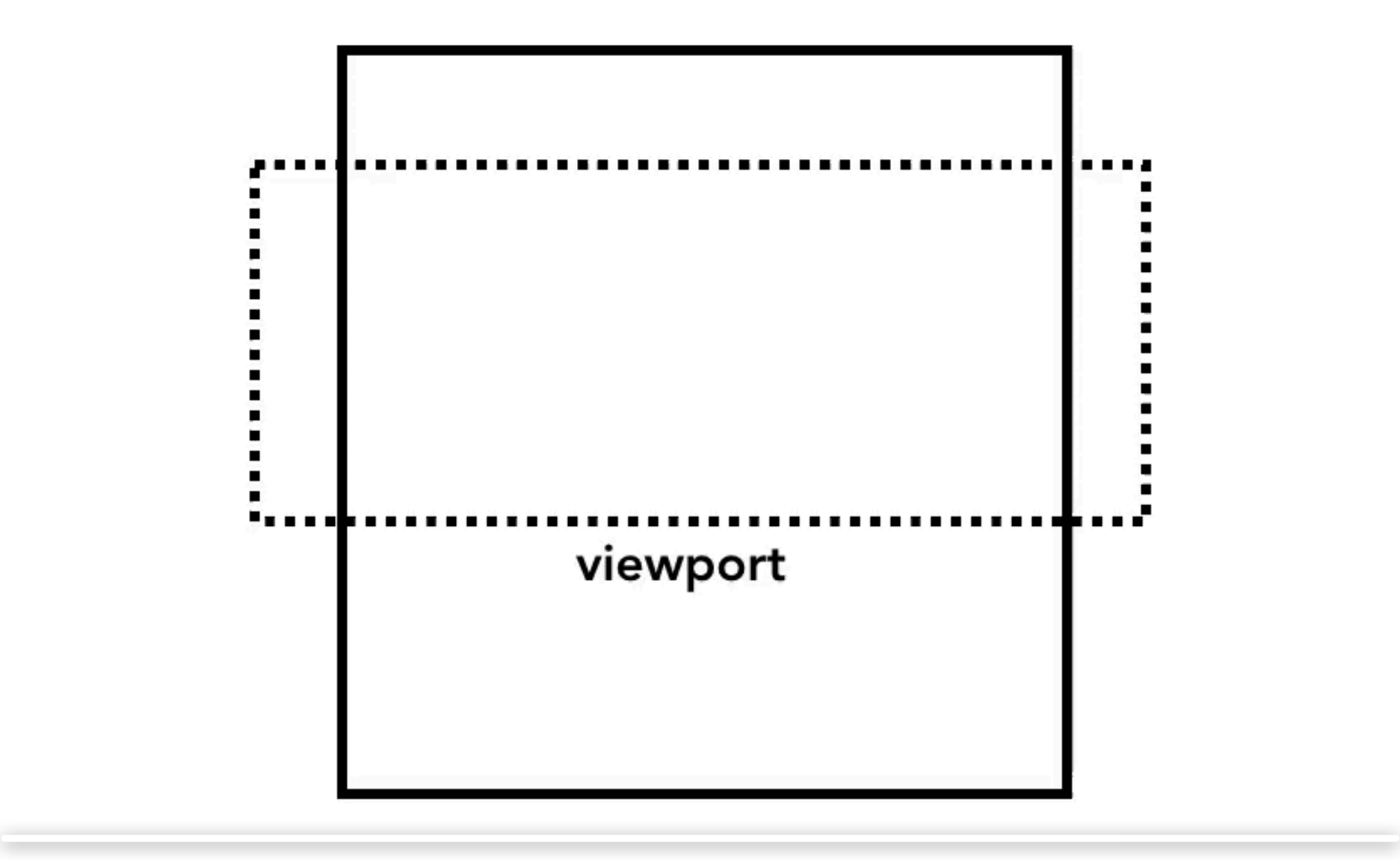


图 14：简单光栅过程的动画

现在，浏览器已经了解了文档的结构、每个元素的样式、页面的几何图形以及绘制顺序，接下来该如何绘制页面呢？将这些信息转换为屏幕上的像素称为光栅化。

处理此问题的一种简单方法是光栅视口内的部分。如果用户滚动页面，则移动光栅框架，然后通过光栅化更多内容来填补缺失的部分。这就是 Chrome 在首次发布时对光栅化的处理方式。不过，现代浏览器会运行一个名为“合成”的更复杂的过程。

什么是合成

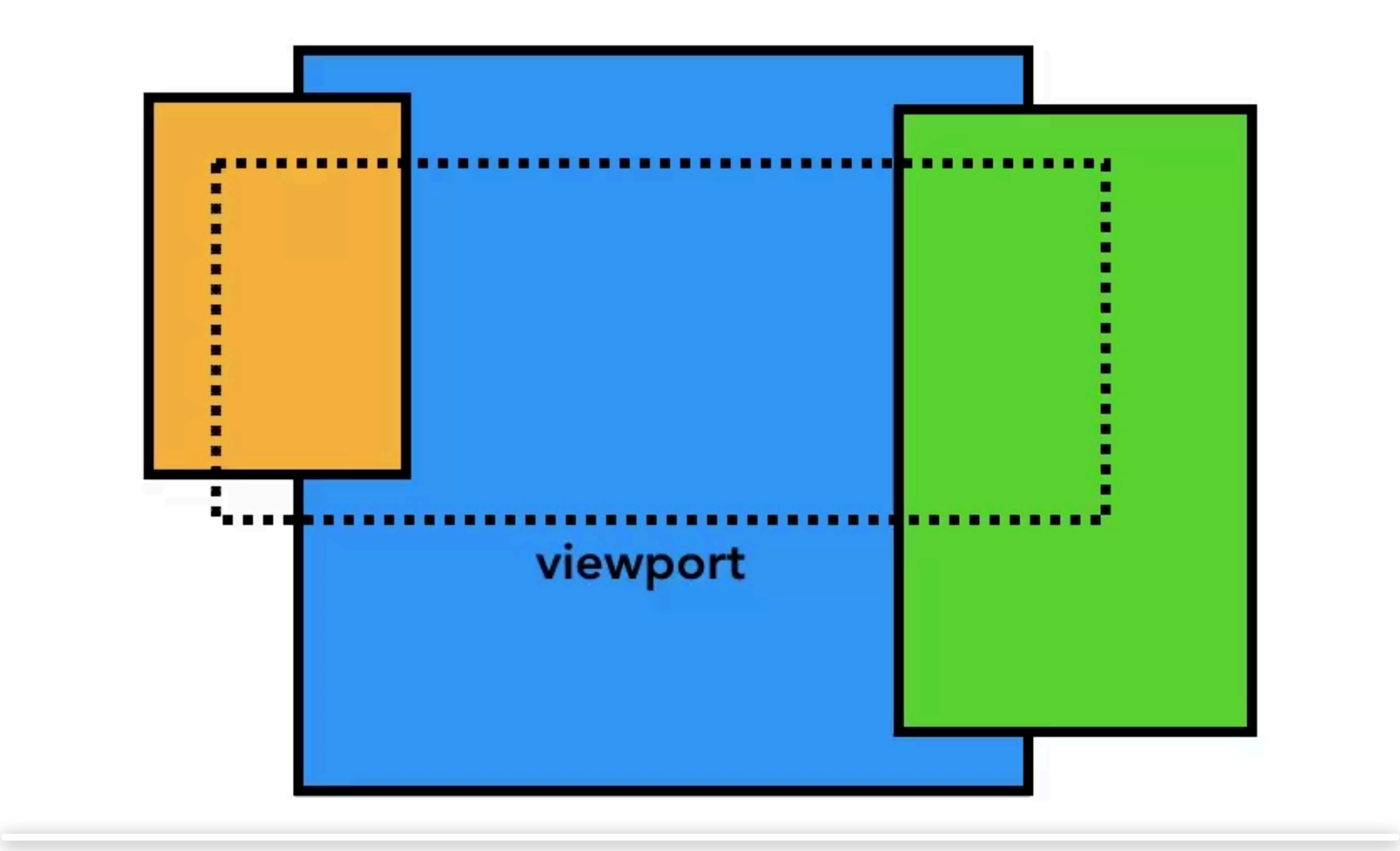


图 15：合成过程的动画

合成是一种技术，可将网页的各个部分分离成图层，分别将它们光栅化，然后在单独的线程（称为“合成器线程”）中合成为网页。如果发生滚动，由于图层已光栅化，因此只需合成新帧即可。同样，可以通过移动层和合成新帧来实现动画效果。

您可以在开发者工具中使用“Layers”面板查看网站是如何划分为多个图层的。

划分为层

为了确定哪些元素需要位于哪些层，主线程会遍历布局树来创建层树（此部分在开发者工具性能面板中称为“Update Layer Tree”）。如果网页中本应属于单独图层的某些部分（例如滑入式侧边菜单）没有出现，您可以使用 CSS 中的 `will-change` 属性来提示浏览器。

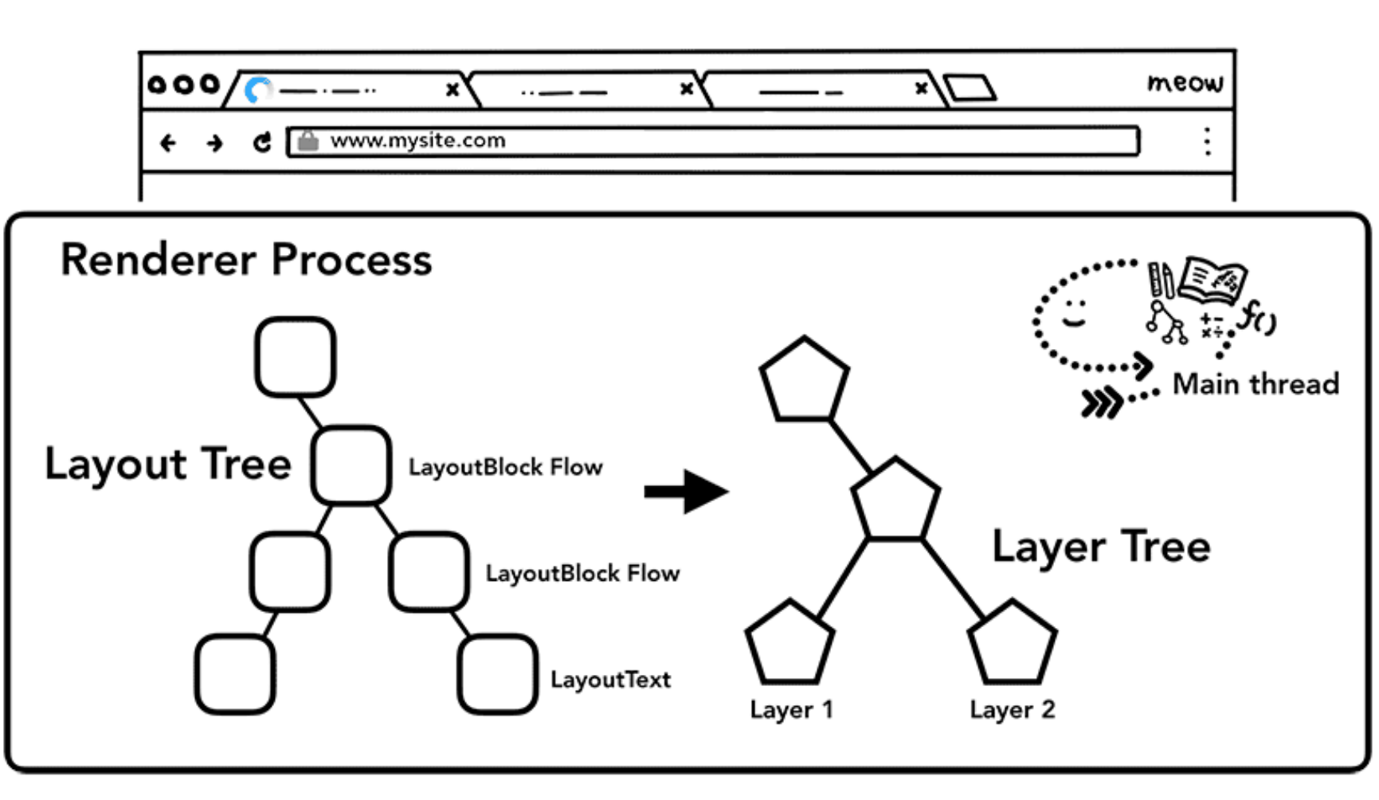


图 16：遍历布局树生成层树的主线程

您可能很想为每个元素都添加层，但与每帧将页面的一小部分光栅化相比，跨过多层进行合成可能会导致操作速度缓慢，因此请务必衡量应用的渲染性能。如需详细了解此主题，请参阅坚持仅合成器的属性和管理层计数。

主线程外的光栅和合成

创建层树并确定绘制顺序后，主线程会将该信息提交到合成器线程。然后，合成器线程会光栅化每个图层。图层的大小可能相当于页面的全部长度，因此合成器线程会将它们分成多块图块，并将每个图块发送到光栅线程。光栅线程会光栅化每个图块并将它们存储在 GPU 内存中。

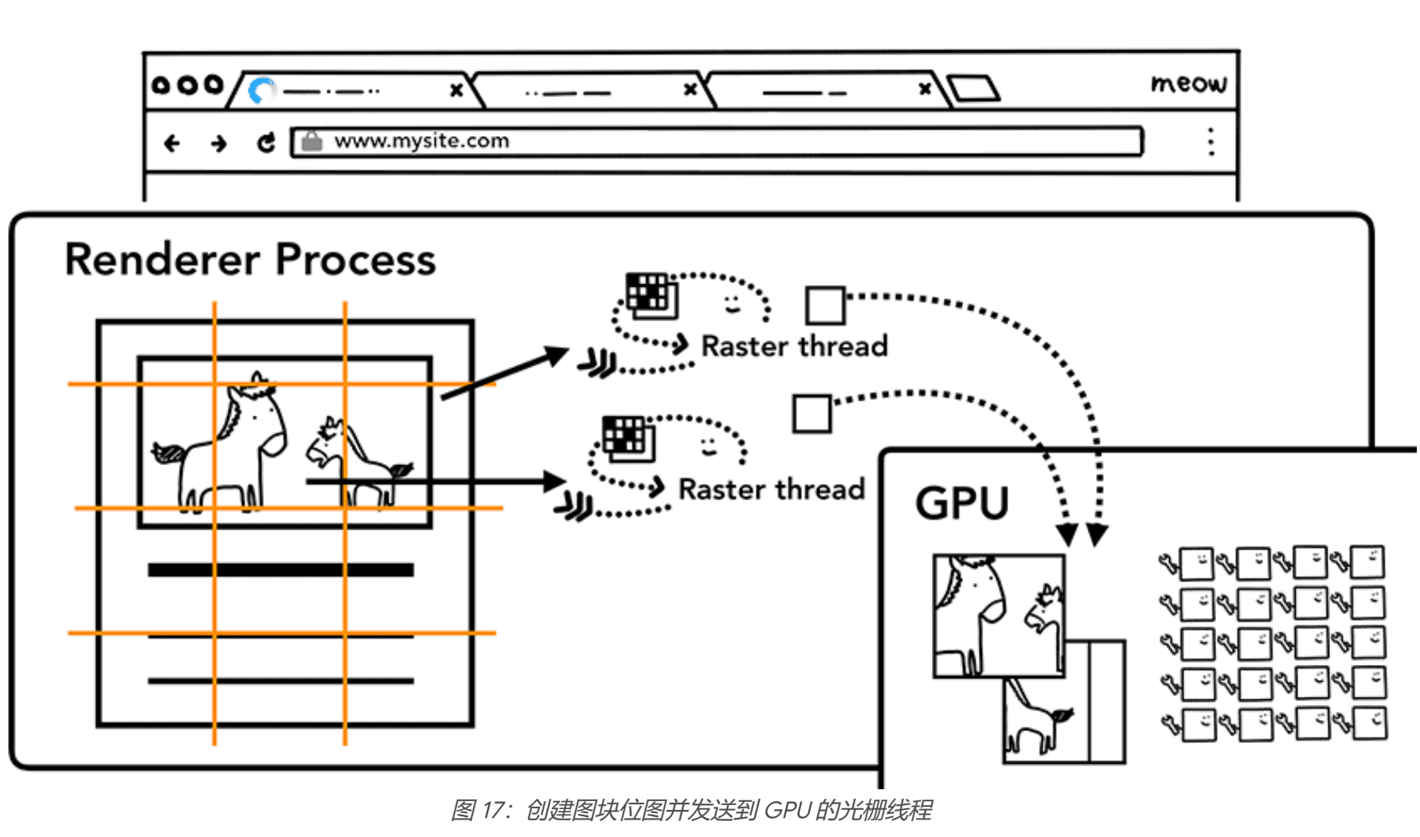


图 17：创建图块位图并发送到 GPU 的光栅线程

合成器线程可以优先处理不同的光栅线程，以便先对视口内（或附近）的内容进行光栅化。图层还具有多个针对不同分辨率的平铺处理，以处理放大操作等操作。

图块光栅化后，合成器线程会收集图块信息（称为“绘制四边形”）来创建**合成器框架**。

绘制四边形	包含功能块在内存中的位置，以及要在页面中的什么位置绘制功能块（考虑页面合成）等信息。
合成器框架	表示页面帧的绘制四边形集合。

然后，通过 IPC 将合成器帧提交到浏览器进程。此时，可以从界面线程（针对浏览器界面更改）或针对扩展程序的其他渲染程序进程添加另一个合成器帧。系统会将这些合成器帧发送到 GPU，以便在屏幕上显示。如果发生滚动事件，合成器线程会再创建一个合成器帧以发送到 GPU。

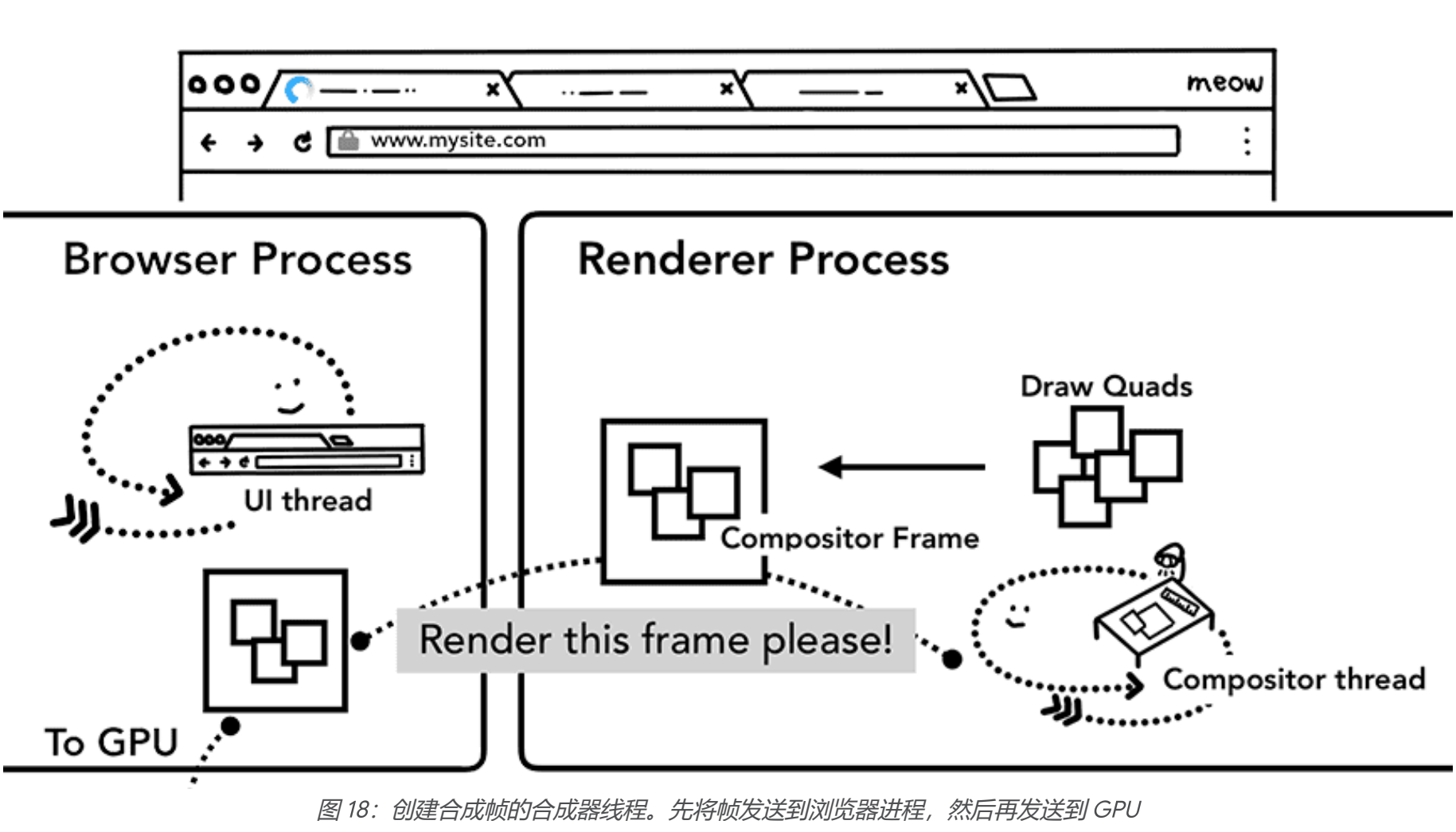


图 18：创建合成帧的合成器线程。先将帧发送到浏览器进程，然后再发送到 GPU

合成的好处是，在不涉及主线程的情况下完成。合成器线程不需要等待样式计算或 JavaScript 执行。因此，仅合成动画被认为是实现流畅性能的最佳选择。如果需要再次计算布局或绘制，则必须涉及主线程。

小结

在这篇博文中，我们了解了渲染流水线从解析到合成。希望您现在能够详细了解网站性能优化。

在本系列的下一篇博文和最后一篇博文中，我们将更详细地介绍合成器线程，并了解输入 `mouse move` 和 `click` 等用户输入时会发生什么情况。

您喜欢这个帖子吗？如果您对以后的帖子有任何疑问或建议，欢迎在下方的评论部分或在 Twitter 上 @kosamari 与我们分享您的想法。

下一步：输入已到达合成器