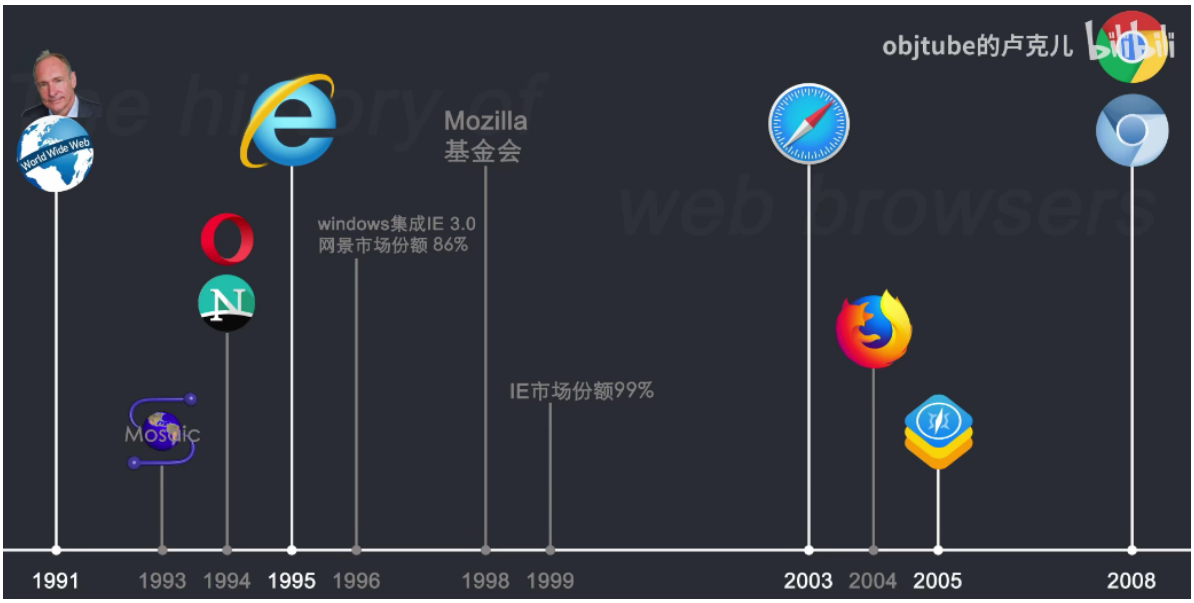
浏览器的工作原理

1. 为什么我们要去了解浏览器的工作原理?

- 写出更好的代码
- 提供更好的用户体验

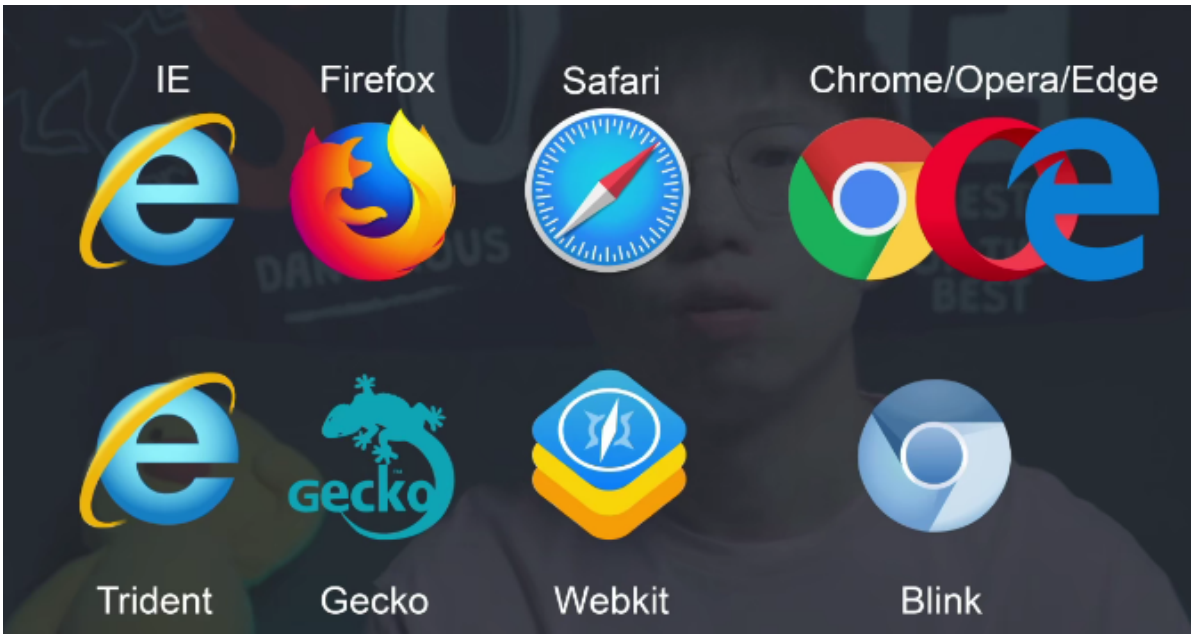
2. 浏览器发展历史



3. 浏览器结构

1. 用户页面：用于展示除标签页窗口之外的其他用户界面内容。
2. 浏览器引擎：用于在用户界面和渲染引擎之间传递数据。
3. **渲染引擎（浏览器内核）**：负责渲染用户请求的页面内容，是浏览器的核心与灵魂。
    - 网络模块：负责网络请求。
    - js解释器：用于解析和执行js。
    - 数据存储持久层：帮助浏览器存储各种数据，比如cookie等。

4. 浏览器内核



5. 进程和线程

浏览器是运行在操作系统上的一个**应用程序**，每个应用程序必须至少启动一个**进程**来执行其功能，每个程序往往需要运行很多任务，此时进程就会创建一些**线程**来帮助它去执行这些小的任务。

5.1 进程

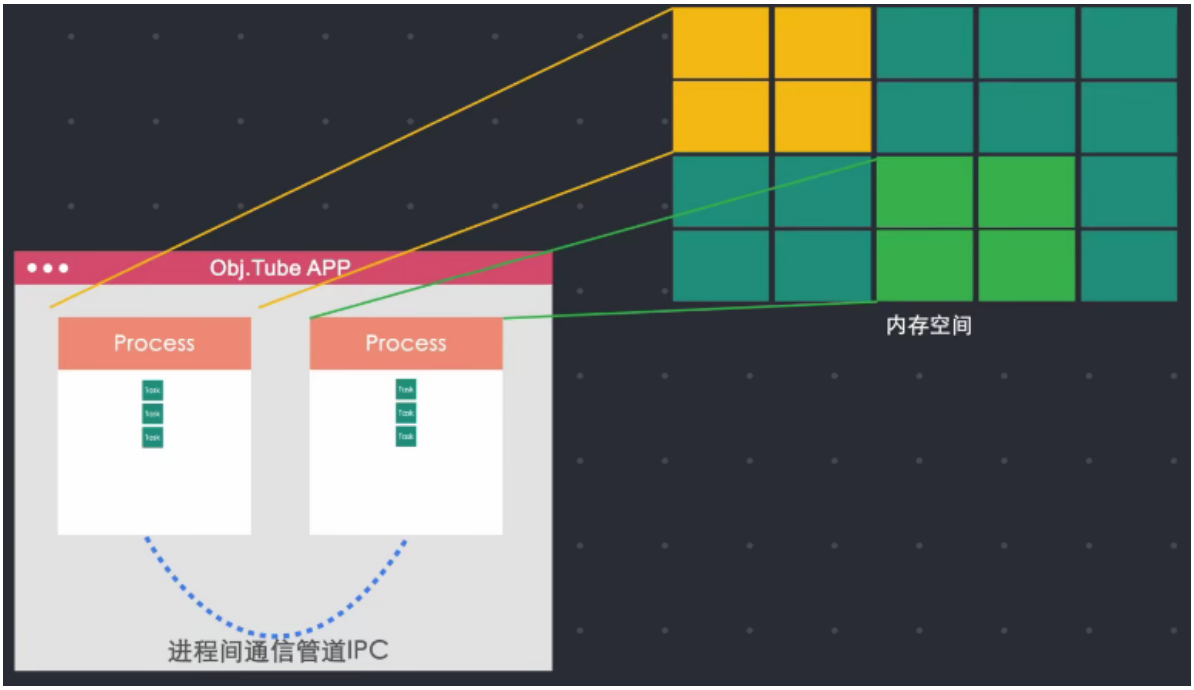
进程是操作系统进行**资源分配和调度**的基本单元，可以**申请**和**拥有**计算机资源，进程是**程序的基本执行实体**。

5.2 线程

线程是操作系统能够进行**运算调度**的最小单位，一个进程阔以并发多个线程，每条线程执行不同的任务。

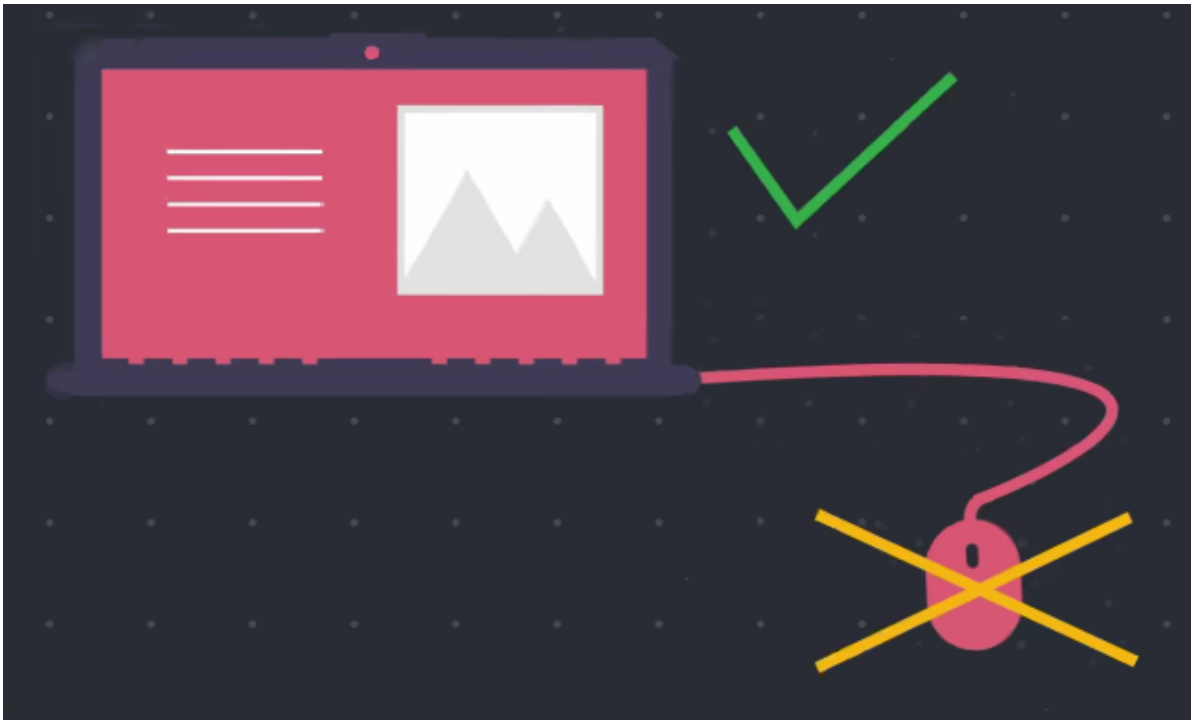
5.3 进程与线程理解

进程和线程理解：当我们启动某个程序时，就会创建一个进程来执行任务代码，同时会为该进程分配内存空间，该应用程序的状态都保存在该内存空间里，当应用关闭时，该内存空间就会被回收，进程可以启动更多的进程来执行任务，由于每个进程分配的内存空间是独立的，如果两个进程见需要传递某些数据则需要通过进程间通信管道 IPC 来传递。

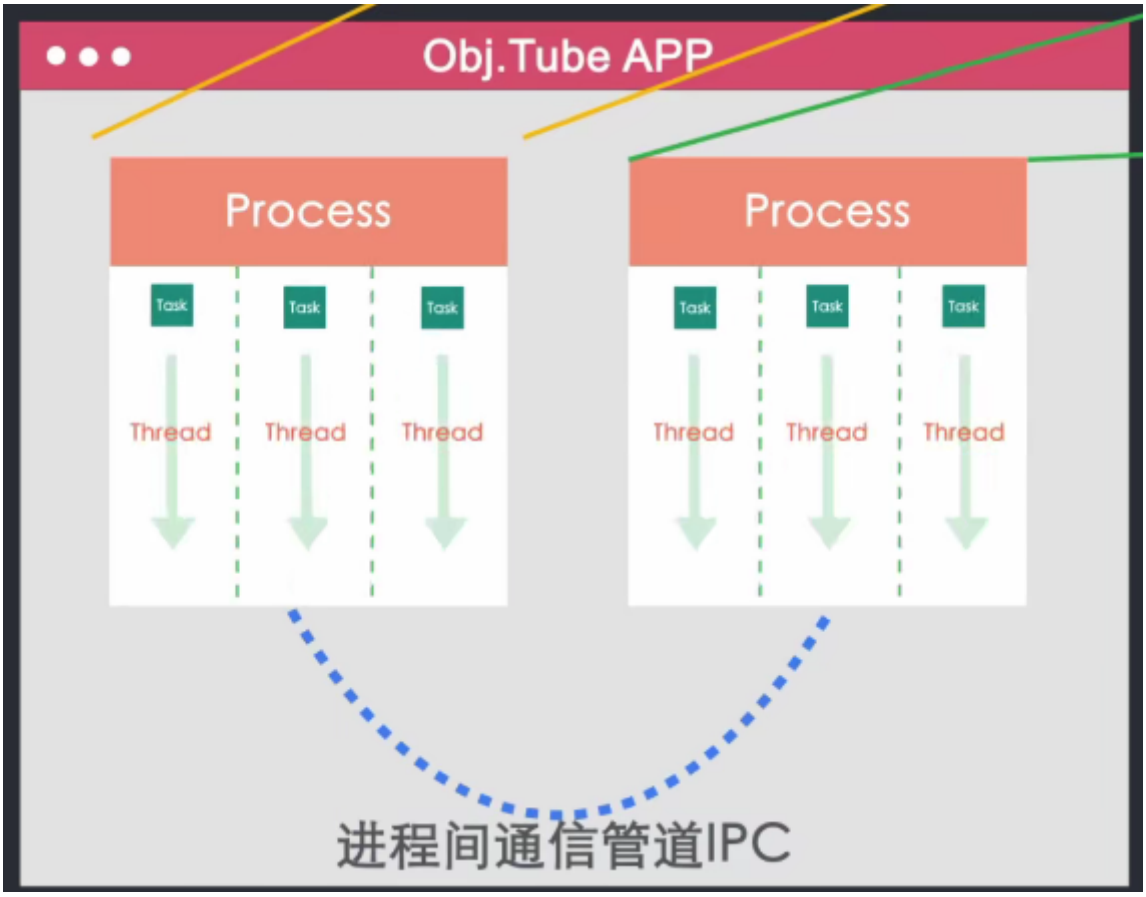


很多应用程序都是多进程结构，这样是为了避免某一个进程卡死，由于进程间相互独立，这样不会影响到整个应用程序。

举个栗子：把笔记本电脑想象成一个应用程序，外接鼠标是该应用程序的的一个进程，如果外接鼠标出了问题，并不会影响你继续使用笔记本电脑。

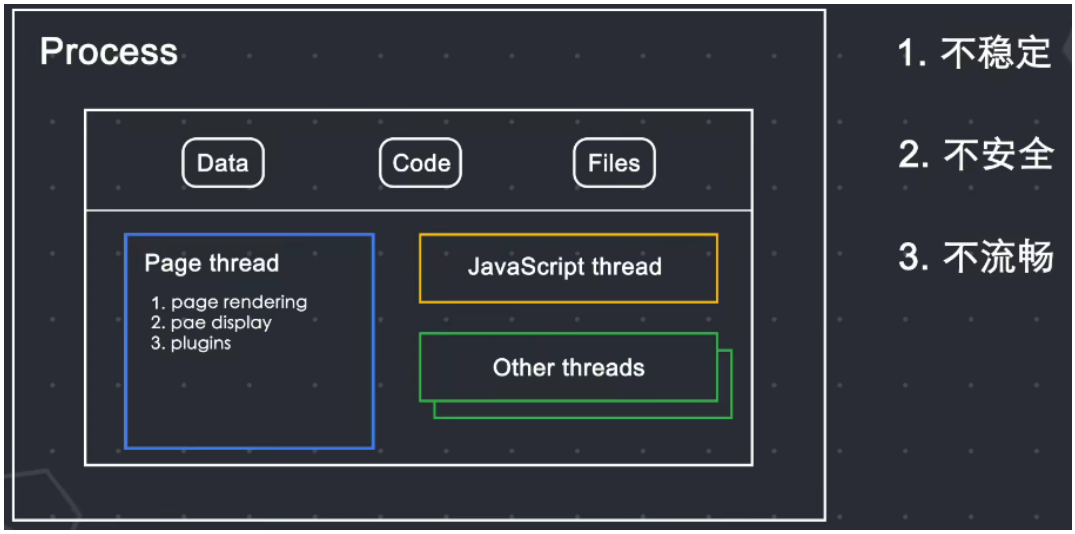


进程可以将任务分成更多细小的任务，然后通过创建多个线程并行执行不同的任务，同一进程下的线程之间是可以直接通信共享数据的



5.4 浏览器中的进程和线程

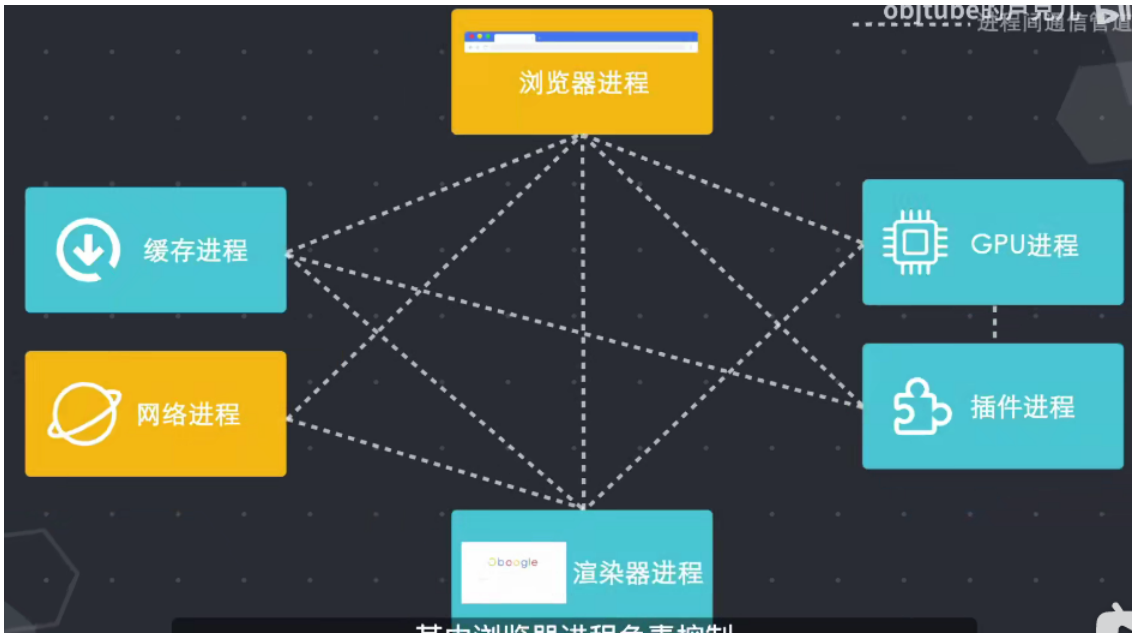
浏览器也是一个多进程结构（早期的是单进程浏览器）

1. 单进程浏览器，包括以下部分：
    - 页面线程负责页面渲染和展示等
    - JS线程执行JS代码
    - 其他线程
    - 

单进程浏览器缺点：

- 不稳定。其中一个线程的卡死可能导致整个进程出问题，比如其中一个标签卡死，可能导致整个浏览器无法正常运行。
- 不安全。浏览器之间是可以共享数据的，这样JS线程就可以随意访问浏览器进程内的所有数据。
- 不流畅。一个进程需要负责太多事情，会导致运行效率问题

2. 多进程浏览器结构（为了解决单线程浏览器的缺点）



  - 浏览器进程：负责控制浏览器除标签页外的用户界面，包括地址栏、书签、后退和前进按钮，以及负责与浏览器的其他进程协调工作。
  - 缓存进程：
  - 网络进程：负责发起接受网络请求。
  - GPU进程：负责整个浏览器界面的渲染。
  - 插件进程：控制网站使用的所有插件，例如：flash。这里的插件并不是指的安装的扩展。
  - 渲染器进程：用来控制显示tab标签内的所有内容。

浏览器在默认情况下会为每个标签页都创建一个进程

6. 浏览器获取网页数据过程

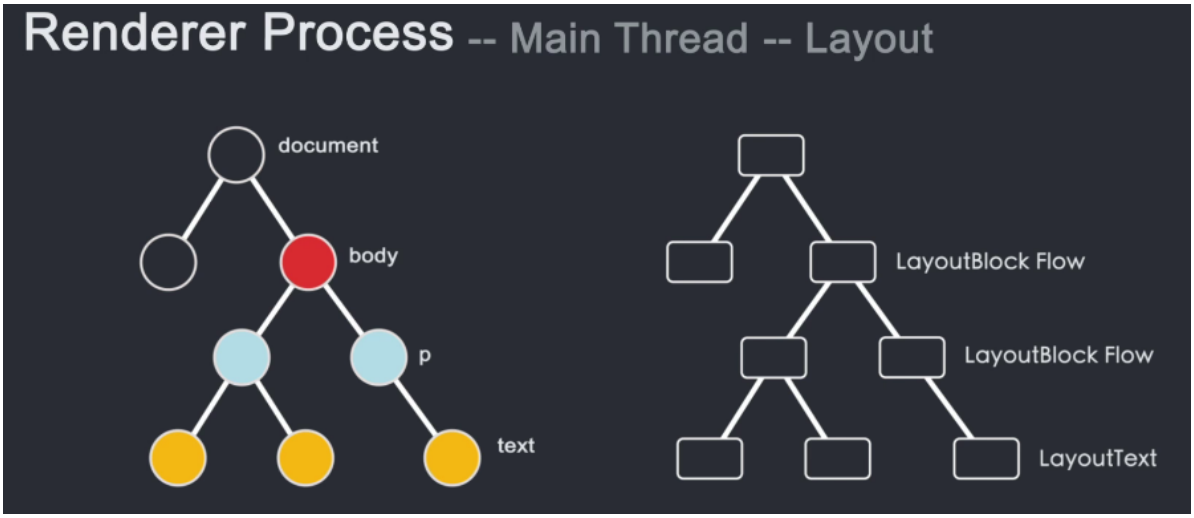
当你在浏览器地址栏里输入内容时，浏览器进程的UI线程会捕捉你的输入内容，

如果访问的是网址，则UI线程会启动一个网络线程来请求DCS进行域名解析，接着开始连接服务器获取数据。

如果你的输入不是网址而是一串关键词，浏览器就知道你是要搜索，于是就会使用默认配置的搜索引擎来查询。

6.1 网络线程获取数据之后发生的事

1. 当网络线程获取到数据后，会通过SafeBrowsing来检查站点是否是恶意站点，如果是则告诉你这个站点有安全问题，浏览器阻止你的访问（也可以强行继续访问），SafeBrowsing是谷歌内部的一套站点安全系统，通过检测该站点的数据来判断是否安全。比如通过查看该站点的IP是否在谷歌的黑名单内。
2. 当返回数据准备完毕，并且安全校验通过时，网络线程会通知UI线程我就要准备好了，该你了，然后UI线程会创建一个渲染器进程（Renderer Thread）来渲染页面。
3. 浏览器进程通过IPC管道将数据传递给渲染器进程，正式进入渲染流程，渲染器进程接收到的数据也就是HTML。渲染器进程的核心任务就是把html、css、js、image等资源渲染成用户可以交互的web页面。
4. 渲染器进程的主线程将html进行解析，构造DOM数据结构（文档对象模型，是浏览器对页面在其内部的表示形式，是web开发程序员可以通过JS与之交互的数据结构和API。
5. html首先通过tokeniser标记化，通过词法分析将输入的HTML内容解析成多个标记，根据识别后的标记进行DOM树构造，在DOM树构造过程中会创建document对象，然后以document为根节点的DOM树不断进行修改，向其中添加各种元素。HTML代码中往往会引入一些额外的资源，比如图片、CSS、JS脚本等，这些资源需要从网络下载或缓存中直接加载，这些资源不会阻塞HTML的解析（因为他们不会影响DOM的生成）。
6. 当HTML解析过程中遇到script标签，就会停止HMTL解析流程，转而去加载并且执行JS（因为JS有可能会改变DOM结构）
7. HTML解析完成后，我们会获得一个DOM树，但我们还不知道DOM树上的每个节点应该长什么样子， 主线程需要解析CSS样式并确定每个DOM节点的计算样式，即使你没有提供自定义的CSS样式，也会有默认的样式表
8. 接下来需要知道每个节点应该放在页面的哪个位置，也就是节点的坐标以及该节点需要占用多大的区域，这个阶段被称为layout布局。主线程通过遍历DOM和计算好的样式来生成Layout Tree



9. Layout Tree上的每个节点都记录了x，y坐标和边框尺寸，注意DOM Tree和Layout Tree并不是一一对应的，设置了display：none的节点不会出现在Layout Tree上，而在before伪类中添加了content值得元素，content里的内容会出现在Layout Tree上，不会出现在DOM树里，
10. 接下来需要知道以什么样的顺序绘制（paint）这个节点举例来说z-index属性会影响节点绘制得层级关系，如果按照DOM得层级结构来绘制页面则会导致错误得渲染。为了保证在屏幕上展示正确的层级，主线程遍历Layout Tree创建一个绘制记录表（Paint Record），代表记录了会绘制得顺序，这个阶段被称为绘制（paint）。
11. 知道了文档的绘制顺序，接着该把这些信息转化成像素点显示到屏幕上，这种行为被称为栅格化（Rastering），栅格化方案：合成（Compsoting），合成是一种将页面的各个部分分成多个图层，分别对其进行栅格化，并在合成器线程（Compositor Thread）单独进行合成页面的技术，简单来说就是页面所有的元素按照某种规则进行分图层，并把图层都栅格化好了，然后只需要把可是取得内容组合成一帧展示给用户即可。
12. 主线程遍历Layout Tree生产Layer（图层） Tree，当Layout Tree生成完毕和绘制顺序确定后，主线程将这些信息传递给合成器线程，合成器线程将每个图层栅格化。由于一层可能像页面得整个长度一样大，因此合成器线程将他们切分为许多图块（tiles），然后将每个图块发送给栅格化线程（Raster Thread），栅格线程栅格化每个图块，并将它们存储在GPU内存中。当图片栅格化完成后，合成器线程将收集称为“draw quads”得图块信息，这些信息里记录了图块在内存中的位置和在页面得哪个位置绘制图块得信息。根据这些信息合成器线程生成了一个合成器帧（Compositor Frame），然后这个合成器帧Frame(帧）通过IPC传送给浏览器进程，接着浏览器进程将合成器帧传送给GPU，然后GPU渲染展示到屏幕上
13. 当你的页面发生变化，比如滚动，都会生成一个新的合成器帧，新的帧再传给GPU，然后再次渲染到屏幕上。

浏览器解析过程总结：

- 浏览器进程中得网络进程请求获取到HTML数据后，通过IPC将数据传给渲染器进程的主线程主线程将HTML解析构造DOM树，
- 然后进行样式计算，根据DOM树和生成好得样式生成Layout Tree，通过遍历Layout Tree生成绘制顺序表，
- 接着遍历Layout Tree生成Layer Tree，然后主线程将Layer Tree和绘制信息一起传给合成器线程，
- 合成器线程按规则进行分图层，并把图层分为更小的图块（tiles）传给栅格线程进行栅格化，
- 栅格化完成后合成器线程会获得栅格线程传过来的“draw quads”图块信息，根据这些信息合成器线上合成了一个合成器帧，
- 然后将该合成器帧通过IPC传回给浏览器进程，浏览器进程再传给GPU机型渲染，之后就展示到了屏幕上。
- 当我们改变一个元素的尺寸位置属性时，会重新进行样式计算（Computed Style），布局（Layout）绘制（Paint）以及后面的所有流程，这种行为我们成为重排。
- 当我们改变某个元素的颜色属性时，不会触发布局，但还是会触发样式计算和绘制，这个就是重绘。

注意：重排和重绘都会占用主线程，还有另外一个东西也运行在主线程上，那就是JS，既然他们都是在主线程运行，就会出现抢占执行时间的问题。如果你写了一个不断导致重排重绘的动画，浏览器则需要在每一帧都运行样式计算布局和绘制的操作。

问题：我们知道当页面以每秒60帧的刷新 率时（也就是每帧16ms），才不会让用户感觉到页面卡顿，如果你在运行动画时还有大量的JS任务需要执行，因为布局、绘制和JS执行都是在主线程运行的，当在一帧的时间内布局和重绘结束后，还有剩余时间JS就会拿到主线程的使用权，如果JS的执行时间过长，就会导致在下一帧开始时JS没有及时归还主线程导致下一帧没有按时渲染，就会出现页面动画的卡顿。

优化手段：

1. **requestAnimationFrame()**，该方法会在每一帧被调用，然后我们可以把JS运行任务分成一些更小的任务块（分到每一帧），在每一帧时间用完前暂停JS执行归还主线程。这样，主线程就可以按时执行布局和绘制。React最新的渲染引擎React Fiber就是用到了这个API来做了很多优化。
2. 栅格化的整个流程是不占用主线程的，只在合成器线程和栅格线程中运行，这就意味着他无需和JS抢夺主线程，我们刚才提到如果反复进行重绘和重排可能会导致掉帧，这是因为有可能JS执行阻塞了主线程，而CSS中有个动画属性叫**transform**，通过该属性实现的动画不会经过布局和绘制，而是直接运行在合成器线程和栅格线程中，所以不会受到主线程中JS执行的影响。更重要的时，通过transform实现的动画由于不需要经过布局、绘制、样式计算等操作所以节省了很多运算时间（方便实现负责的动画）

这些就是为什么需要尽量避免重绘和重排的原因。