前端必会！四步带你吃透浏览器渲染基本原理

CSDN 2019-07-09

以下文章来源于前端下午茶 ，作者SHERlocked93

作者 | SHERlocked93

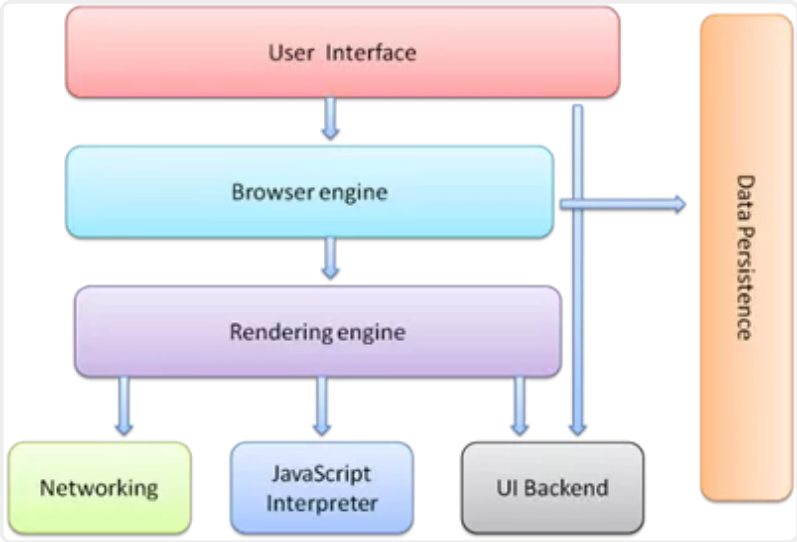
责编 | 胡巍巍

大多数设备的刷新频率是60Hz，也就说是浏览器对每一帧画面的渲染工作要在16ms内完成,超出这个时间，页面的渲染就会出现卡顿现象，影响用户体验。前端的用户体验给了前端直观的印象，因此对B/S架构的开发人员来说，熟悉浏览器的内部执行原理显得尤为重要。

浏览器主要组成与浏览器线程

1.1 浏览器组件

浏览器大体上由以下几个组件组成，各个浏览器可能有一点不同。



- 界面控件 – 包括地址栏，前进后退，书签菜单等窗口上除了网页显示区域以外的部分
- 浏览器引擎 – 查询与操作渲染引擎的接口
- 渲染引擎 – 负责显示请求的内容。比如请求到HTML, 它会负责解析HTML、CSS并将结果显示到窗口中
- 网络 – 用于网络请求, 如HTTP请求。它包括平台无关的接口和各平台独立的实现
- UI后端 – 绘制基础元件，如组合框与窗口。它提供平台无关的接口，内部使用操作系统的相应实现
- JS解释器 - 用于解析执行JavaScript代码
- 数据存储持久层 - 浏览器需要把所有数据存到硬盘上，如cookies。新的HTML5规范规定了一个完整（虽然轻量级）的浏览器中的数据库 web database

注意：chrome浏览器与其他浏览器不同，chrome使用多个渲染引擎实例，每个Tab页一个，即每个Tab都是一个独立进程。

1.2 浏览器中的进程与线程

Chrome浏览器使用多个进程来隔离不同的网页，在Chrome中打开一个网页相当于起了一个进程，每个tab网页都有由其独立的渲染引擎实例。因为如果非多进程的话，如果浏览器中的一个tab网页崩溃，将会导致其他被打开的网页应用。另外相对于线程，进程之间是不共享资源和地址空间的，所以不会存在太多的安全问题，而由于多个线程共享着相同的地址空间和资源，所以会存在线程之间有可能会恶意修改或者获取非授权数据等复杂的安全问题。

在内核控制下各线程相互配合以保持同步，一个浏览器通常由以下常驻线程组成：

1. GUI 渲染线程

GUI渲染线程负责渲染浏览器界面HTML元素,当界面需要重绘(Repaint)或由于某种操作引发回流(reflow)时，该线程就会执行。在Javascript引擎运行脚本期间,GUI渲染线程都是处于挂起状态的，也就是说被冻结了.

2. JavaScript引擎线程

JS为处理页面中用户的交互，以及操作DOM树、CSS样式树来给用户呈现一份动态而丰富的交互体验和服务器逻辑的交互处理。如果JS是多线程的方式来操作这些UI DOM，则可能出现UI操作的冲突；如果JS是多线程的话，在多线程的交互下，处于UI中的DOM节点就可能成为一个临界资源，假设存在两个线程同时操作一个DOM，一个负责修改一个负责删除，那么这个时候就需要浏览器来裁决如何生效哪个线程的执行结果，当然我们可以通过锁来解决上面的问题。但为了避免因为引入了锁而带来更大的复杂性，JS在最初就选择了单线程执行。

GUI渲染线程与JS引擎线程互斥的，是由于JavaScript是可操纵DOM的，如果在修改这些元素属性同时渲染界面（即JavaScript线程和UI线程同时运行），那么渲染线程前后获得的元素数据就可能不一致。当JavaScript引擎执行时GUI线程会被挂起，GUI更新会被保存在一个队列中等到引擎线程空闲时立即被执行。由于GUI渲染线程与JS执行线程是互斥的关系，当浏览器在执行JS程序的时候，GUI渲染线程会被保存在一个队列中，直到JS程序执行完成，才会接着执行。因此如果JS执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞的感觉。

3. 定时触发器线程

浏览器定时计数器并不是由JS引擎计数的, 因为JS引擎是单线程的, 如果处于阻塞线程状态就会影响记计时的准确, 因此通过单独线程来计时并触发定时是更为合理的方案。

4. 事件触发线程

当一个事件被触发时该线程会把事件添加到待处理队列的队尾，等待JS引擎的处理。这些事件可以是当前执行的代码块如定时任务、也可来自浏览器内核的其他线程如鼠标点击、AJAX异步请求等，但由于JS的单线程关系所有这些事件都得排队等待JS引擎处理。

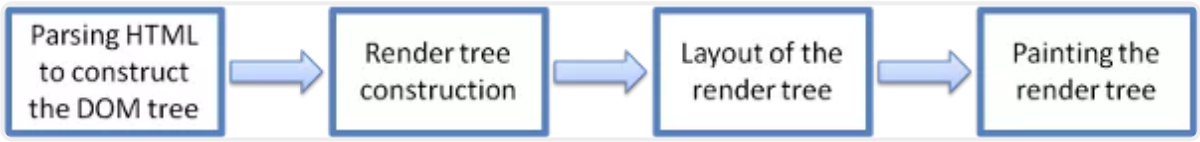
5. 异步http请求线程

在XMLHttpRequest在连接后是通过浏览器新开一个线程请求，将检测到状态变更时，如果设置有回调函数，异步线程就产生状态变更事件放到JS引擎的处理队列中等待处理。

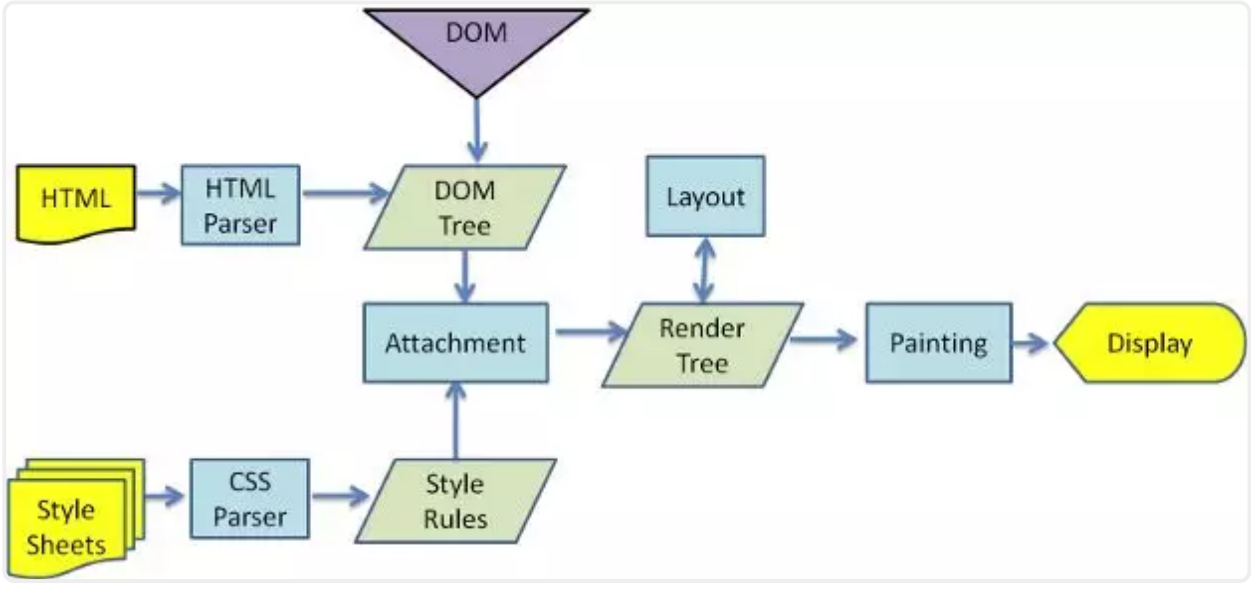
渲染过程

2.1 渲染流程

用户请求的HTML文本(text/html)通过浏览器的网络层到达渲染引擎后，渲染工作开始。每次通常渲染不会超过8K的数据块，其中基础的渲染流程图：



webkit引擎渲染的详细流程，其他引擎渲染流程稍有不同：



渲染流程有四个主要步骤：

1. 解析HTML生成DOM树 - 渲染引擎首先解析HTML文档，生成DOM树
2. 构建Render树 - 接下来不管是内联式，外联式还是嵌入式引入的CSS样式会被解析生成CSSOM树，根据DOM树与CSSOM树生成另外一棵用于渲染的树-渲染树(Render tree)，
3. 布局Render树 - 然后对渲染树的每个节点进行布局处理，确定其在屏幕上的显示位置
4. 绘制Render树 - 最后遍历渲染树并用UI后端层将每一个节点绘制出来

以上步骤是一个渐进的过程，为了提高用户体验，渲染引擎试图尽可能快的把结果显示给最终用户。它不会等到所有HTML都被解析完才创建并布局渲染树。它会在从网络层获取文档内容的同时把已经接收到的局部内容先展示出来。

2.2 渲染细节

1. 生成DOM树

DOM树的构建过程是一个深度遍历过程：当前节点的所有子节点都构建好后才会去构建当前节点的下一个兄弟节点。DOM树的根节点就是document对象。

DOM树的生成过程中可能会被CSS和JS的加载执行阻塞，具体可以参见下一章。当HTML文档解析过程完毕后，浏览器继续进行标记为deferred模式的脚本加载，然后就是整个解析过程的实际结束触发DOMContentLoaded事件，并在async文档文档执行完之后触发load事件。

2. 生成Render树

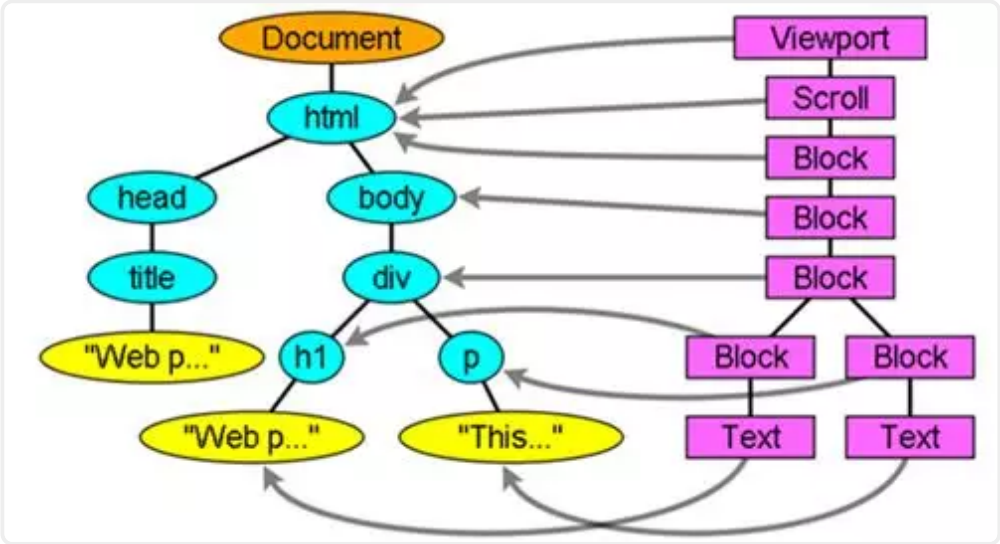
生成DOM树的同时会生成样式结构体CSSOM（CSS Object Model）Tree，再根据CSSOM和DOM树构造渲染树Render Tree，渲染树包含带有颜色，尺寸等显示属性的矩形，这些矩形的顺序与显示顺序基本一致。从MVC的角度来说，可以将Render树看成是V，DOM树与CSSOM树看成是M，C则是具体的调度者，比HTMLDocumentParser等。

可以这么说，没有DOM树就没有Render树，但是它们之间不是简单的一对一的关系。Render树是用于显示，那不可见的元素当然不会在这棵树中出现了，譬如 <head>。除此之外，display等于none的也不会被显示在这棵树里头，但是visibility等于hidden的元素是会显示在这棵树里头的。

3. DOM树与Render树

DOM对象类型很丰富，什么head、title、div，而Render树相对来说就比较单一了，毕竟它的职责就是为了以后的显示渲染用嘛。Render树的每一个节点我们叫它渲染器renderer。

一棵Render树大概是酱紫，左边是DOM树，右边是Render树：



从上图我们可以看出，renderer与DOM元素是相对应的，但并不是一一对应，有些DOM元素没有对应的renderer，而有些DOM元素却对应了好几个renderer，对应多个renderer的情况是普遍存在的，就是为了解决一个renderer描述不清楚如何显示出来的问题，譬如有下拉列表的select元素，我们就需要三个renderer：一个用于显示区域，一个用于下拉列表框，还有一个用于按钮。

另外，renderer与DOM元素的位置也可能是不一样的。那些添加了 float 或者position:absolute的元素，因为它们脱离了正常的文档流，构造Render树的时候会针对它们实际的位置进行构造。

4. 布局与绘制

上面确定了renderer的样式规则后，然后就是重要的显示元素布局了。当renderer构造出来并添加到Render树上之后，它并没有位置跟大小信息，为它确定这些信息的过程，接下来是布局(layout)。

浏览器进行页面布局基本过程是以浏览器可见区域为画布，左上角为 (0,0)基础坐标，从左到右，从上到下从DOM的根节点开始画，首先确定显示元素的大小跟位置，此过程是通过浏览器计算出来的，用户CSS中定义的量未必就是浏览器实际采用的量。如果显示元素有子元素得先去确定子元素的显示信息。

布局阶段输出的结果称为box盒模型（width,height,margin,padding,border,left,top,...），盒模型精确表示了每一个元素的位置和大小，并且所有相对度量单位此时都转化为了绝对单位。

在绘制(painting)阶段，渲染引擎会遍历Render树，并调用renderer的 paint() 方法，将renderer的内容显示在屏幕上。绘制工作是使用UI后端组件完成的。

5. 回流与重绘

回流(reflow)：当浏览器发现某个部分发生了点变化影响了布局，需要倒回去重新渲染。reflow 会从 <html>这个 root frame 开始递归往下，依次计算所有的结点几何尺寸和位置。reflow 几乎是无法避免的。现在界面上流行的一些效果，比如树状目录的折叠、展开（实质上是元素的显示与隐藏）等，都将引起浏览器的 reflow。鼠标滑过、点击……只要这些行为引起了页面上某些元素的占位面积、定位方式、边距等属性的变化，都会引起它内部、周围甚至整个页面的重新渲染。通常我们都无法预估浏览器到底会 reflow 哪一部分的代码，它们都彼此相互影响着。

重绘(repaint)：改变某个元素的背景色、文字颜色、边框颜色等等不影响它周围或内部布局的属性时，屏幕的一部分要重画，但是元素的几何尺寸没有变。

每次Reflow，Repaint后浏览器还需要合并渲染层并输出到屏幕上。所有的这些都会是动画卡顿的原因。Reflow 的成本比 Repaint 的成本高得多的多。一个结点的 Reflow 很有可能导致子结点，甚至父点以及同级结点的 Reflow 。在一些高性能的电脑上也许还没什么，但是如果 Reflow 发生在手机上，那么这个过程是延慢加载和耗电的。可以在csstrigger上查找某个css属性会触发什么事件。

reflow与repaint的时机：

1. display:none 会触发 reflow，而 visibility:hidden 只会触发 repaint，因为没有发生位置变化。
2. 有些情况下，比如修改了元素的样式，浏览器并不会立刻 reflow 或 repaint 一次，而是会把这样的操作积攒一批，然后做一次 reflow，这又叫异步 reflow 或增量异步 reflow。
3. 有些情况下，比如 resize 窗口，改变了页面默认的字体等。对于这些操作，浏览器会马上进行 reflow。

关键渲染路径与阻塞渲染

在浏览器拿到HTML、CSS、JS等外部资源到渲染出页面的过程，有一个重要的概念关键渲染路径（Critical Rendering Path）。例如为了保障首屏内容的最快速显示，通常会提到一个渐进式页面渲染，但是为了渐进式页面渲染，就需要做资源的拆分，那么以什么粒度拆分、要不要拆分，不同页面、不同场景策略不同。具体方案的确定既要考虑体验问题，也要考虑工程问题。了解原理可以让我们更好的优化关键渲染路径，从而获得更好的用户体验。

现代浏览器总是并行加载资源，例如，当 HTML 解析器（HTML Parser）被脚本阻塞时，解析器虽然会停止构建 DOM，但仍会识别该脚本后面的资源，并进行预加载。

同时，由于下面两点：

1. CSS 被视为渲染 阻塞资源 (包括JS)，这意味着浏览器将不会渲染任何已处理的内容，直至 CSSOM 构建完毕，才会进行下一阶段。
2. JavaScript 被认为是解释器阻塞资源，HTML解析会被JS阻塞，它不仅可以读取和修改 DOM 属性，还可以读取和修改 CSSOM 属性。

存在阻塞的 CSS 资源时，浏览器会延迟 JavaScript 的执行和 DOM 构建。另外：

1. 当浏览器遇到一个 script 标记时，DOM 构建将暂停，直至脚本完成执行。
2. JavaScript 可以查询和修改 DOM 与 CSSOM。
3. CSSOM 构建时，JavaScript 执行将暂停，直至 CSSOM 就绪。

所以，script 标签的位置很重要。实际使用时，可以遵循下面两个原则：

1. CSS 优先：引入顺序上，CSS 资源先于 JavaScript 资源。
2. JavaScript 应尽量少影响 DOM 的构建。

下面来看看 CSS 与 JavaScript 是具体如何阻塞资源的。

3.1 CSS

```
<style> p { color: red; }</style>
<link rel="stylesheet" href="index.css">
```

这样的 link 标签（无论是否 inline）会被视为阻塞渲染的资源，浏览器会优先处理这些 CSS 资源，直至 CSSOM 构建完毕。

渲染树（Render-Tree）的关键渲染路径中，要求同时具有 DOM 和 CSSOM，之后才会构建渲染树。即，HTML 和 CSS 都是阻塞渲染的资源。HTML 显然是必需的，因为包括我们希望显示的文本在内的内容，都在 DOM 中存放，那么可以从 CSS 上想办法。

最容易想到的当然是精简 CSS 并尽快提供它。除此之外，还可以用媒体类型（media type）和媒体查询（media query）来解除对渲染的阻塞。

```
<link href="index.css" rel="stylesheet">
<link href="print.css" rel="stylesheet" media="print">
<link href="other.css" rel="stylesheet" media="(min-width: 30em) and (orient
```

第一个资源会加载并阻塞。第二个资源设置了媒体类型，会加载但不会阻塞，print 声明只在打印网页时使用。第三个资源提供了媒体查询，会在符合条件时阻塞渲染。

关于CSS加载的阻塞情况：

1. css加载不会阻塞DOM树的解析
2. css加载会阻塞DOM树的渲染
3. css加载会阻塞后面js语句的执行

没有js的理想情况下，html与css会并行解析，分别生成DOM与CSSOM，然后合并成Render Tree，进入Rendering Pipeline；但如果有js，css加载会阻塞后面js语句的执行，而（同步）js脚本执行会阻塞其后的DOM解析（所以通常会把css放在头部，js放在body尾）

3.2 JavaScript

JavaScript 的情况比 CSS 要更复杂一些。如果没有 defer 或 async，浏览器会立即加载并执行指定的脚本，“立即”指的是在渲染该 script 标签之下的HTML元素之前，也就是说不等待后续载入的HTML元素，读到就加载并执行。观察下面的代码：

```
<p>Do not go gentle into that good night,</p>
<script>console.log("inline1")</script>
<p>Old age should burn and rave at close of day;</p>
<script src="app.js"></script>
<p>Rage, rage against the dying of the light.</p>
<script src="app.js"></script>
<p>Old age should burn and rave at close of day;</p>
<script>console.log("inline2")</script>
<p>Rage, rage against the dying of the light.</p>
```

这里的 script 标签会阻塞 HTML 解析，无论是不是 inline-script。上面的 P 标签会从上到下解析，这个过程会被两段 JavaScript 分别打断一次（加载、执行）。

解析过程中无论遇到的JavaScript是内联还是外链，只要浏览器遇到 script 标记，唤醒JavaScript解析器，就会进行暂停 (blocked)浏览器解析HTML，并等到 CSSOM 构建完毕，才去执行js脚本。因为脚本中可能会操作DOM元素，而如果在加载执行脚本的时候DOM元素并没有被解析，脚本就会因为DOM元素没有生成取不到响应元素，所以实际工程中，我们常常将资源放到文档底部。

3.3 改变脚本加载次序defer与async

defer 与 async 可以改变之前的那些阻塞情形，这两个属性都会使 script 异步加载，然而执行的时机是不一样的。注意 async 与 defer 属性对于 inline-script 都是无效的，所以下面这个示例中三个 script 标签的代码会从上到下依次执行。

```
<script async>console.log("1")</script>
<script defer>console.log("2")</script>
<script>console.log("3")</script>
```

上面脚本会按需输出 1 2 3，故，下面两节讨论的内容都是针对设置了 src 属性的 script 标签。

先放个熟悉的图~

蓝色线代表网络读取，红色线代表执行时间，这俩都是针对脚本的；绿色线代表 HTML 解析。

defer:

```
<script src="app1.js" defer></script>
<script src="app2.js" defer></script>
<script src="app3.js" defer></script>
```

defer 属性表示延迟执行引入 JavaScript，即 JavaScript 加载时 HTML 并未停止解析，这两个过程是并行的。整个 document 解析完毕且 defer-script 也加载完成之后（这两件事情的顺序无关），会执行所有由 defer-script 加载的 JavaScript 代码，再触发

DOMContentLoaded(初始的 HTML 文档被完全加载和解析完成之后触发，无需等待样式表图像和子框架的完成加载) 事件 。

defer 不会改变 script 中代码的执行顺序，示例代码会按照 1、2、3 的顺序执行。所以，defer 与相比普通 script，有两点区别：载入 JavaScript 文件时不阻塞 HTML 的解析，执行阶段被放到 HTML 标签解析完成之后。

async:

async 属性表示异步执行引入的 JavaScript，与 defer 的区别在于，如果已经加载好，就会开始执行，无论此刻是 HTML 解析阶段还是 DOMContentLoaded 触发(HTML解析完成事件)之后。需要注意的是，这种方式加载的 JavaScript 依然会阻塞 load 事件。换句话说，async-script 可能在 DOMContentLoaded 触发之前或之后执行，但一定在 load 触发之前执行。

从上一段也能推出，多个 async-script 的执行顺序是不确定的，谁先加载完谁执行。值得注意的是，向 document 动态添加 script 标签时，async 属性默认是 true。

document.createElement:

使用 document.createElement 创建的 script 默认是异步的，示例如下。

```
console.log(document.createElement("script").async); // true
```

所以，通过动态添加 script 标签引入 JavaScript 文件默认是不会阻塞页面的。如果想同步执行，需要将 async 属性人为设置为 false。

如果使用 document.createElement 创建 link 标签会怎样呢?

```
const style = document.createElement("link");
style.rel = "stylesheet";
style.href = "index.css";
document.head.appendChild(style); // 阻塞?
```

其实这只能通过试验确定，已知的是，Chrome 中已经不会阻塞渲染，Firefox、IE 在以前是阻塞的，现在会怎样目前不太清楚。

优化渲染性能

结合渲染流程，可以针对性的优化渲染性能：

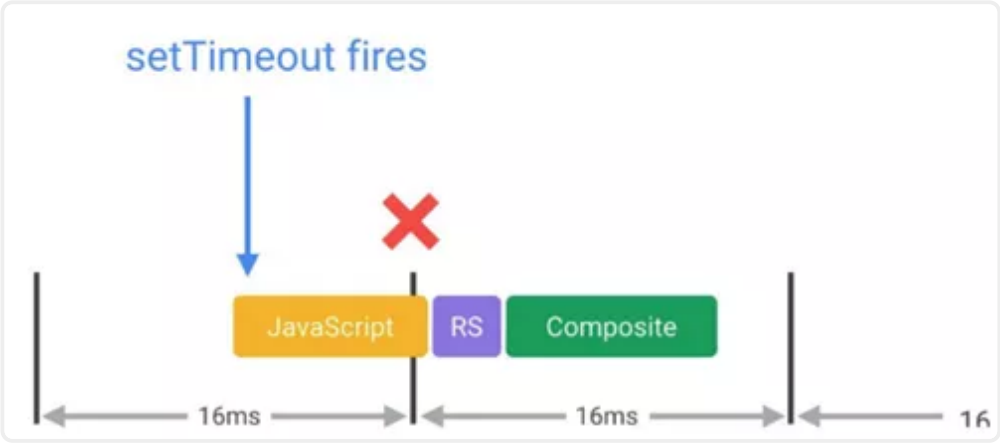
1. 优化JS的执行效率
2. 降低样式计算的范围和复杂度
3. 避免大规模、复杂的布局
4. 简化绘制的复杂度、减少绘制区域
5. 优先使用渲染层合并属性、控制层数量
6. 对用户输入事件的处理函数去抖动（移动设备）

这里主要参考Google的浏览器渲染性能的基础讲座，想看更详细内容可以去瞅瞅~

4.1 优化JS的执行效率

1. 动画实现使用requestAnimationFrame

setTimeout(callback)和setInterval(callback)无法保证callback函数的执行时机，很可能在帧结束的时候执行，从而导致丢帧，如下图：



requestAnimationFrame(callback)可以保证callback函数在每帧动画开始的时候执行。注意：jQuery3.0.0以前版本的animate函数就是用setTimeout来实现动画，可以通过jquery-requestAnimationFrame这个补丁来用requestAnimationFrame替代setTimeout

2. 长耗时的JS代码放到Web Workers中执行

JS代码运行在浏览器的主线程上，与此同时，浏览器的主线程还负责样式计算、布局、绘制的工作，如果JavaScript代码运行时间过长，就会阻塞其他渲染工作，很可能会导致丢帧。前面提到每帧的渲染应该在16ms内完成，但在动画过程中，由于已经被占用了不少时间，所以JavaScript代码运行耗时应该控制在3-4毫秒。如果真的有特别耗时且不操作DOM元素的纯计算工作，可以考虑放到Web Workers中执行。

```
var dataSortWorker = new Worker("sort-worker.js");
dataSortWorker.postMesssage(dataToSort);

// 主线程不受Web Workers线程干扰
dataSortWorker.addEventListener('message', function(evt) {
var sortedData = e.data;
// Web Workers线程执行结束
// ...
});
```

3. 拆分操作DOM元素的任务，分别在多个frame完成

由于Web Workers不能操作DOM元素的限制，所以只能做一些纯计算的工作，对于很多需要操作DOM元素的逻辑，可以考虑分步处理，把任务分为若干个小任务，每个任务都放到requestAnimationFrame中回调执行。

```
var taskList = breakBigTaskIntoMicroTasks(monsterTaskList);
requestAnimationFrame(processTaskList);

function processTaskList(taskStartTime) {
var nextTask = taskList.pop();
// 执行小任务
    processTask(nextTask);
if (taskList.length > 0) {
        requestAnimationFrame(processTaskList);
     }
}
```

4. 使用Chrome DevTools的Timeline来分析JavaScript的性能

打开 ChromeDevTools>Timeline>JSProfile，录制一次动作，然后分析得到的细节信息，从而发现问题并修复问题。

4.2 降低样式计算的范围和复杂度

添加或移除一个DOM元素、修改元素属性和样式类、应用动画效果等操作，都会引起DOM结构的改变，从而导致浏览器要repaint或者reflow。那么这里可以采取一些措施。

1. 降低样式选择器的复杂度

尽量保持class的简短，或者使用Web Components框架。

```
.box:nth-last-child(-n+1) .title {}
// 改善后
.final-box-title {}
```

2. 减少需要执行样式计算的元素个数

由于浏览器的优化，现代浏览器的样式计算直接对目标元素执行，而不是对整个页面执行，所以我们应该尽可能减少需要执行样式计算的元素的个数。

4.3 避免大规模、复杂的布局

布局就是计算DOM元素的大小和位置的过程，如果你的页面中包含很多元素，那么计算这些元素的位置将耗费很长时间。布局的主要消耗在于：1. 需要布局的DOM元素的数量；2. 布局过程的复杂程度

1. 尽可能避免触发布局

当你修改了元素的属性之后，浏览器将会检查为了使这个修改生效是否需要重新计算布局以及更新渲染树，对于DOM元素的几何属性修改，比如width/height/left/top等，都需要重新计算布局。对于不能避免的布局，可以使用Chrome DevTools工具的Timeline查看布局的耗时，以及受影响的DOM元素数量。

2. 使用flexbox替代老的布局模型

老的布局模型以相对/绝对/浮动的方式将元素定位到屏幕上，而Floxbox布局模型用流式布局的方式将元素定位到屏幕上。通过一个小实验可以看出两种布局模型的性能差距，同样对1300个元素布局，浮动布局耗时14.3ms，Flexbox布局耗时3.5ms。IE10+支持。

3. 避免强制同步布局事件的发生

根据渲染流程，JS脚本是在layout之前执行，但是我们可以强制浏览器在执行JS脚本之前先执行布局过程，这就是所谓的强制同步布局。

```
requestAnimationFrame(logBoxHeight);
// 先写后读，触发强制布局
```

```
function logBoxHeight() {
// 更新box样式
    box.classList.add('super-big');

// 为了返回box的offersetHeight值
// 浏览器必须先应用属性修改，接着执行布局过程
    console.log(box.offsetHeight);
}

// 先读后写，避免强制布局
function logBoxHeight() {
// 获取box.offsetHeight
    console.log(box.offsetHeight);
// 更新box样式
    box.classList.add('super-big');
}
```

在JS脚本运行的时候，它能获取到的元素样式属性值都是上一帧画面的，都是旧的值。因此，如果你在当前帧获取属性之前又对元素节点有改动，那就会导致浏览器必须先应用属性修改，结果执行布局过程，最后再执行JS逻辑。

4. 避免连续的强制同步布局发生

如果连续快速的多次触发强制同步布局，那么结果更糟糕。比如下面的例子，获取box的属性，设置到paragraphs上，由于每次设置paragraphs都会触发样式计算和布局过程，而下一次获取box的属性必须等到上一步设置结束之后才能触发。

```
function resizeWidth() {
// 会让浏览器陷入'读写读写'循环
for (var i = 0; i < paragraphs.length; i++)
        paragraphs[i].style.width = box.offsetWidth + 'px';
}
}

// 改善后方案
var width = box.offsetWidth;
function resizeWidth() {
for (var i = 0; i < paragraphs.length; i++)
        paragraphs[i].style.width = width +px';
     }
}
```

注意：可以使用FastDOM来确保读写操作的安全，从而帮你自动完成读写操作的批处理，还能避免意外地触发强制同步布局或快速连续布局，消除大量操作DOM的时候的布局抖动。

4.4 简化绘制的复杂度、减少绘制区域

Paint就是填充像素的过程，通常这个过程是整个渲染流程中耗时最长的一环，因此也是最需要避免发生的一环。如果Layout被触发，那么接下来元素的Paint一定会被触发。当然纯粹改变元素的非几何属性，也可能会触发Paint，比如背景、文字颜色、阴影效果等。

1. 提升移动或渐变元素的绘制层

绘制并非总是在内存中的单层画面里完成的，实际上，浏览器在必要时会将一帧画面绘制成多层画面，然后将这若干层画面合并成一张图片显示到屏幕上。这种绘制方式的好处是，使用transform来实现移动效果的元素将会被正常绘制，同时不会触发其他元素的绘制。

2. 减少绘制区域，简化绘制的复杂度

浏览器会把相邻区域的渲染任务合并在一起进行，所以需要对动画效果进行精密设计，以保证各自的绘制区域不会有太多重叠。另外可以实现同样效果的不同方式，应该采用性能更好的那种。

3. 通过Chrome DevTools来分析绘制复杂度和时间消耗，尽可能降低这些指标

打开DevTools，在弹出的面板中，选中 MoreTools>Rendering选项卡下的Paint flashing，这样每当页面发生绘制的时候，屏幕就会闪现绿色的方框。通过该工具可以检查Paint发生的区域和时机是不是可以被优化。通过Chrome DevTools中的 Timeline>Paint选项可以查看更细节的Paint信息

4.5 优先使用渲染层合并属性、控制层数量

1. 使用transform/opacity实现动画效果

使用transform/opacity实现动画效果，会跳过渲染流程的布局和绘制环节，只做渲染层的合并。

Type	Func
Position	transform: translate(-px,-px)
Scale	transform: scale(-)

Type	Func
Rotation	transform: rotate(-deg)
Skew	transform: skew(X/Y)(-deg)
Matrix	transform: matrix(3d)(..)
Opacity	opacity: 0-1

使用transform/opacity的元素必须独占一个渲染层，所以必须提升该元素到单独的渲染层。

2. 提升动画效果中的元素

应用动画效果的元素应该被提升到其自有的渲染层，但不要滥用。在页面中创建一个新的渲染层最好的方式就是使用CSS属性will-change，对于目前还不支持will-change属性、但支持创建渲染层的浏览器，可以通过3D transform属性来强制浏览器创建一个新的渲染层。需要注意的是，不要创建过多的渲染层，这意味着新的内存分配和更复杂的层管理。注意，IE11，Edge17都不支持这一属性。

```
.moving-element {
    will-change: transform;
    transform: translateZ(0);
}
```

3. 管理渲染层、避免过多数量的层

尽管提升渲染层看起来很诱人，但不能滥用，因为更多的渲染层意味着更多的额外的内存和管理资源，所以当且仅当需要的时候才为元素创建渲染层。

```
* {
  will-change: transform;
  transform: translateZ(0);
}
```

4. 使用Chrome DevTools来了解页面的渲染层情况

开启 Timeline>Paint选项，然后录制一段时间的操作，选择单独的帧，看到每个帧的渲染细节，在ESC弹出框有个Layers选项，可以看到渲染层的细节，有多少渲染层，为何被创建?

4.6 对用户输入事件的处理函数去抖动（移动设备）

用户输入事件处理函数会在运行时阻塞帧的渲染，并且会导致额外的布局发生。

1. 避免使用运行时间过长的输入事件处理函数

理想情况下，当用户和页面交互，页面的渲染层合并线程将接收到这个事件并移动元素。这个响应过程是不需要主线程参与，不会导致JavaScript、布局和绘制过程发生。但是如果被触摸的元素绑定了输入事件处理函数，比如touchstart/touchmove/touchend，那么渲染层合并线程必须等待这些被绑定的处理函数执行完毕才能执行，也就是用户的滚动页面操作被阻塞了，表现出的行为就是滚动出现延迟或者卡顿。

简而言之就是你必须确保用户输入事件绑定的任何处理函数都能够快速的执行完毕，以便腾出时间来让渲染层合并线程完成他的工作。

2. 避免在输入事件处理函数中修改样式属性

输入事件处理函数，比如scroll/touch事件的处理，都会在requestAnimationFrame之前被调用执行。因此，如果你在上述输入事件的处理函数中做了修改样式属性的操作，那么这些操作就会被浏览器暂存起来，然后在调用requestAnimationFrame的时候，如果你在一开始就做了读取样式属性的操作，那么将会触发浏览器的强制同步布局操作。

3. 对滚动事件处理函数去抖动

通过requestAnimationFrame可以对样式修改操作去抖动，同时也可以使你的事件处理函数变得更轻。

```
function onScroll(evt) {
// Store the scroll value for laterz.
    lastScrollY = window.scrollY;
// Prevent multiple rAF callbacks.
if (scheduledAnimationFrame) {
return;
}
    scheduledAnimationFrame = true;
    requestAnimationFrame(readAndUpdatePage);

}
window.addEventListener('scroll', onScroll);
```