25-页面性能：如何系统地优化页面？

在前面几篇文章中，我们分析了页面加载和DOM生成，讨论了JavaScript和CSS是如何影响到DOM生成的，还结合渲染流水线来讲解了分层和合成机制，同时在这些文章里面，我们还穿插说明了很多优化页面性能的最佳实践策略。通过这些知识点的学习，相信你已经知道渲染引擎是怎么绘制出帧的，不过之前我们介绍的内容比较零碎、比较散，那么今天我们就来将这些内容系统性地串起来。

那么怎么才能把这些知识点串起来呢？我的思路是从如何系统优化页面速度的角度来切入。

这里我们所谈论的页面优化，其实就是要让页面更快地显示和响应。由于一个页面在它不同的阶段，所侧重的关注点是不一样的，所以如果我们要讨论页面优化，就要分析一个页面生存周期的不同阶段。

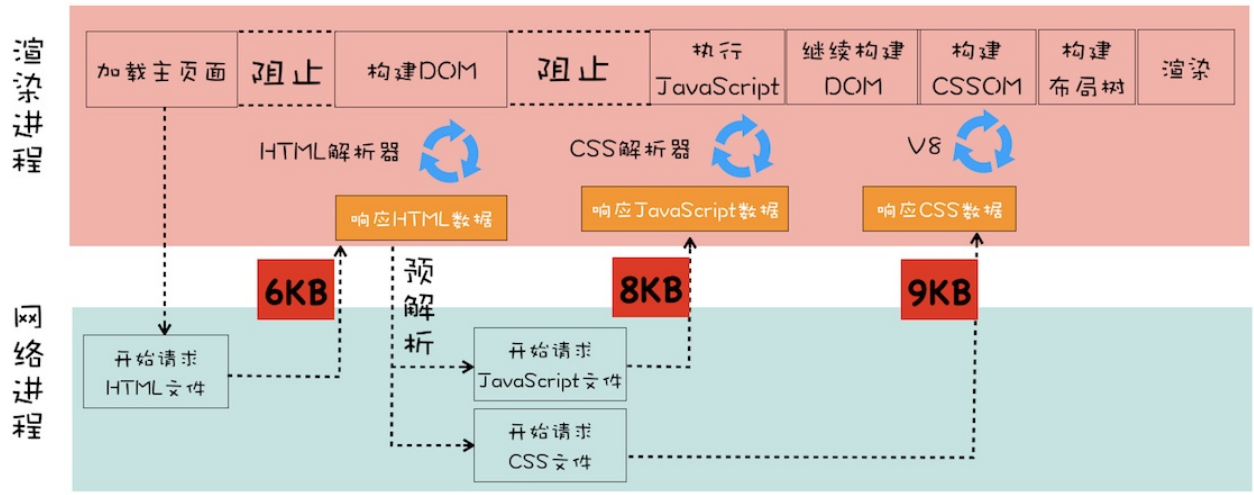
通常一个页面有三个阶段：**加载阶段、交互阶段和关闭阶段**。

- 加载阶段，是指从发出请求到渲染出完整页面的过程，影响到这个阶段的主要因素有网络和JavaScript脚本。
- 交互阶段，主要是从页面加载完成到用户交互的整合过程，影响到这个阶段的主要因素是JavaScript脚本。
- 关闭阶段，主要是用户发出关闭指令后页面所做的一些清理操作。

这里我们需要**重点关注加载阶段和交互阶段**，因为影响到我们体验的因素主要都在这两个阶段，下面我们就来逐个详细分析下。

加载阶段

我们先来分析如何系统优化加载阶段中的页面，还是先看一个典型的渲染流水线，如下图所示：



加载阶段渲染流水线

观察上面这个渲染流水线，你能分析出来有哪些因素影响了页面加载速度吗？下面我们就先来分析下这个问题。

通过前面文章的讲解，你应该已经知道了并非所有的资源都会阻塞页面的首次绘制，比如图片、音频、视频等文件就不会阻塞页面的首次渲染；而JavaScript、首次请求的HTML资源文件、CSS文件是会阻塞首次渲

染的，因为在构建DOM的过程中需要HTML和JavaScript文件，在构造渲染树的过程中需要用到CSS文件。

我们把**这些能阻塞网页首次渲染的资源称为关键资源**。基于关键资源，我们可以继续细化出来三个影响页面首次渲染的核心因素。

第一个是关键资源个数。关键资源个数越多，首次页面的加载时间就会越长。比如上图中的关键资源个数就是3个，1个HTML文件、1个JavaScript和1个CSS文件。

第二个是关键资源大小。通常情况下，所有关键资源的内容越小，其整个资源的下载时间也就越短，那么阻塞渲染的时间也就越短。上图中关键资源的大小分别是6KB、8KB和9KB，那么整个关键资源大小就是23KB。

第三个是请求关键资源需要多少个RTT（Round Trip Time）。那什么是RTT呢？ 在《02 | TCP协议：如何保证页面文件能被完整送达浏览器？》这篇文章中我们分析过，当使用TCP协议传输一个文件时，比如这个文件大小是0.1M，由于TCP的特性，这个数据并不是一次传输到服务端的，而是需要拆分成一个个数据包来回多次进行传输的。**RTT就是这里的往返时延。它是网络中一个重要的性能指标，表示从发送端发送数据开始，到发送端收到来自接收端的确认，总共经历的时延**。通常1个HTTP的数据包在14KB左右，所以1个0.1M的页面就需要拆分成8个包来传输了，也就是说需要8个RTT。

我们可以结合上图来看看它的关键资源请求需要多少个RTT。首先是请求HTML资源，大小是6KB，小于14KB，所以1个RTT就可以解决了。至于JavaScript和CSS文件，这里需要注意一点，由于渲染引擎有一个预解析的线程，在接收到HTML数据之后，预解析线程会快速扫描HTML数据中的关键资源，一旦扫描到了，会立马发起请求，你可以认为JavaScript和CSS是同时发起请求的，所以它们的请求是重叠的，那么计算它们的RTT时，只需要计算体积最大的那个数据就可以了。这里最大的是CSS文件（9KB），所以我们就按照9KB来计算，同样由于9KB小于14KB，所以JavaScript和CSS资源也就可以算成1个RTT。也就是说，上图中关键资源请求共花费了2个RTT。

了解了影响加载过程中的几个核心因素之后，接下来我们就可以系统性地考虑优化方案了。**总的优化原则就是减少关键资源个数，降低关键资源大小，降低关键资源的RTT次数**。

- 如何减少关键资源的个数？一种方式是可以将JavaScript和CSS改成内联的形式，比如上图的JavaScript和CSS，若都改成内联模式，那么关键资源的个数就由3个减少到了1个。另一种方式，如果JavaScript代码没有DOM或者CSSOM的操作，则可以改成sync或者defer属性；同样对于CSS，如果不是在构建页面之前加载的，则可以添加媒体取消阻止显现的标志。当JavaScript标签加上了sync或者defer、CSSlink 属性之前加上了取消阻止显现的标志后，它们就变成了非关键资源了。
- 如何减少关键资源的大小？可以压缩CSS和JavaScript资源，移除HTML、CSS、JavaScript文件中一些注释内容，也可以通过前面讲的取消CSS或者JavaScript中关键资源的方式。
- 如何减少关键资源RTT的次数？可以通过减少关键资源的个数和减少关键资源的大小搭配来实现。除此之外，还可以使用CDN来减少每次RTT时长。

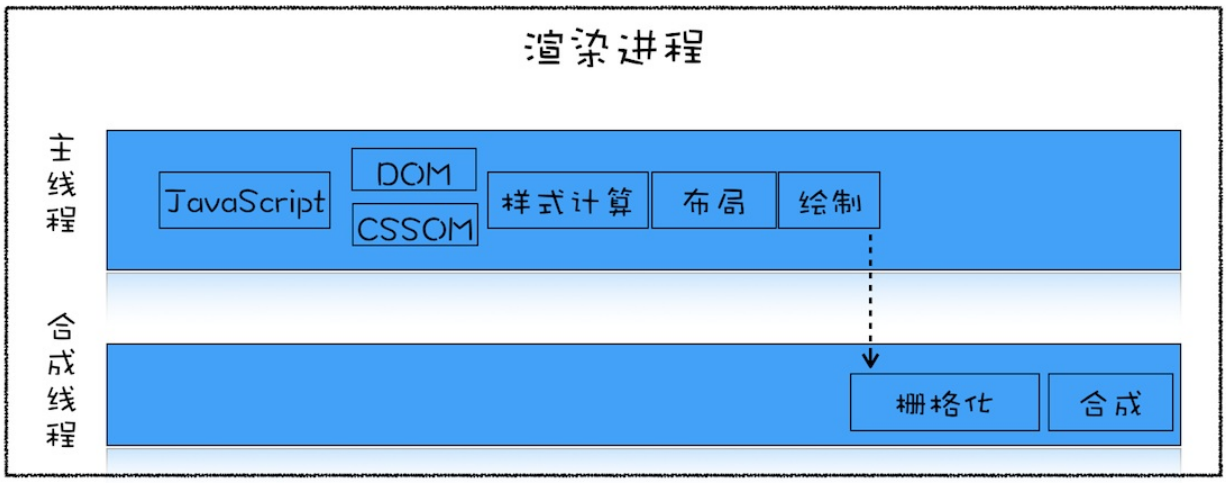
在优化实际的页面加载速度时，你可以先画出优化之前关键资源的图表，然后按照上面优化关键资源的原则去优化，优化完成之后再画出优化之后的关键资源图表。

交互阶段

接下来我们再来聊聊页面加载完成之后的交互阶段以及应该如何去优化。谈交互阶段的优化，其实就是在谈

渲染进程渲染帧的速度，因为在交互阶段，帧的渲染速度决定了交互的流畅度。因此讨论页面优化实际上就是讨论渲染引擎是如何渲染帧的，否则就无法优化帧率。

我们先来看看交互阶段的渲染流水线（如下图）。和加载阶段的渲染流水线有一些不同的地方是，在交互阶段没有了加载关键资源和构建DOM、CSSOM流程，通常是由JavaScript触发交互动画的。



交互阶段渲染流水线

结合上图，我们来一起回顾下交互阶段是如何生成一个帧的。大部分情况下，生成一个新的帧都是由JavaScript通过修改DOM或者CSSOM来触发的。还有另外一部分帧是由CSS来触发的。

如果在计算样式阶段发现有布局信息的修改，那么就会触发**重排**操作，然后触发后续渲染流水线的一系列操作，这个代价是非常大的。

同样如果在计算样式阶段没有发现有布局信息的修改，只是修改了颜色一类的信息，那么就不会涉及到布局相关的调整，所以可以跳过布局阶段，直接进入绘制阶段，这个过程叫**重绘**。不过重绘阶段的代价也是不小的。

还有另外一种情况，通过CSS实现一些变形、渐变、动画等特效，这是由CSS触发的，并且是在合成线程上执行的，这个过程称为合成。因为它不会触发重排或者重绘，而且合成操作本身的速度就非常快，所以执行合成是效率最高的方式。

回顾了在交互过程中的帧是如何生成的，那接下来我们就可以讨论优化方案了。**一个大的原则就是让单个帧的生成速度变快**。所以，下面我们就来分析下在交互阶段渲染流水线中有哪些因素影响了帧的生成速度以及如何去优化。

1. 减少JavaScript脚本执行时间

有时JavaScript函数的一次执行时间可能有几百毫秒，这就严重霸占了主线程执行其他渲染任务的时间。针对这种情况我们可以采用以下两种策略：

- 一种是将一次执行的函数分解为多个任务，使得每次的执行时间不要过久。
- 另一种是采用Web Workers。你可以把Web Workers当作主线程之外的一个线程，在Web Workers中是可以执行JavaScript脚本的，不过Web Workers中没有DOM、CSSOM环境，这意味着在Web Workers中是

无法通过JavaScript来访问DOM的，所以我们可以把一些和DOM操作无关且耗时的任务放到Web Workers中去执行。

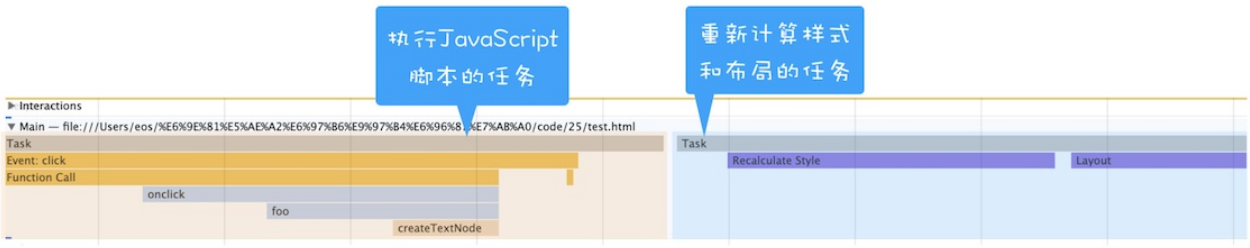
总之，在交互阶段，对JavaScript脚本总的原则就是不要一次霸占太久主线程。

2. 避免强制同步布局

在介绍强制同步布局之前，我们先来聊聊正常情况下的布局操作。通过DOM接口执行添加元素或者删除元素等操作后，是需要重新计算样式和布局的，不过正常情况下这些操作都是在另外的任务中异步完成的，这样做是为了避免当前的任务占用太长的主线程时间。为了直观理解，你可以参考下面的代码：

```
<html>
<body>
    <div id="mian_div">
        <li id="time_li">time</li>
        <li>geekbang</li>
    </div>

    <p id="demo">强制布局demo</p>
    <button onclick="foo()">添加新元素</button>

    <script>
        function foo() {
            let main_div = document.getElementById("mian_div")
            let new_node = document.createElement("li")
            let textnode = document.createTextNode("time.geekbang")
            new_node.appendChild(textnode);
            document.getElementById("mian_div").appendChild(new_node);
        }
    </script>
</body>
</html>
```

对于上面这段代码，我们可以使用Performance工具来记录添加元素的过程，如下图所示：



Performance记录添加元素的执行过程

从图中可以看出来，执行JavaScript添加元素是在一个任务中执行的，重新计算样式布局是在另外一个任务中执行，这就是正常情况下的布局操作。

理解了正常情况下的布局操作，接下来我们就可以聊什么是强制同步布局了。

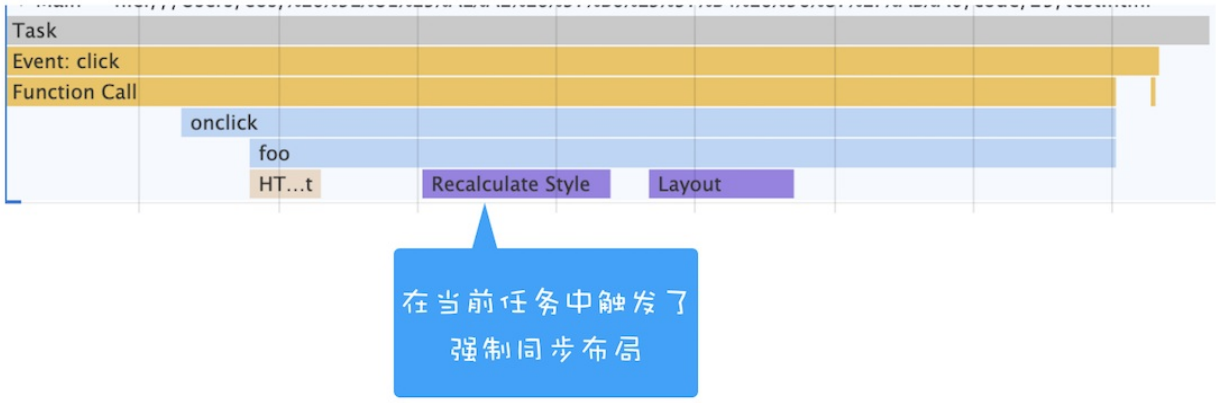
所谓强制同步布局，是指JavaScript强制将计算样式和布局操作提前到当前的任务中。为了直观理解，这

里我们对上面的代码做了一点修改，让它变成强制同步布局，修改后的代码如下所示：

```
function foo() {
    let main_div = document.getElementById("mian_div")
    let new_node = document.createElement("li")
    let textnode = document.createTextNode("time.geekbang")
    new_node.appendChild(textnode);
    document.getElementById("mian_div").appendChild(new_node);
    //由于要获取到offsetHeight，
    //但是此时的offsetHeight还是老的数据，
    //所以需要立即执行布局操作
    console.log(main_div.offsetHeight)
}
```

将新的元素添加到DOM之后，我们又调用了`main_div.offsetHeight`来获取新main_div的高度信息。如果要获取到main_div的高度，就需要重新布局，所以这里在获取到main_div的高度之前，JavaScript还需要强制让渲染引擎默认执行一次布局操作。我们把这个操作称为强制同步布局。

同样，你可以看下面通过Performance记录的任务状态：



触发强制同步布局Performance图

从上图可以看出来，计算样式和布局都是在当前脚本执行过程中触发的，这就是强制同步布局。

为了避免强制同步布局，我们可以调整策略，在修改DOM之前查询相关值。代码如下所示：

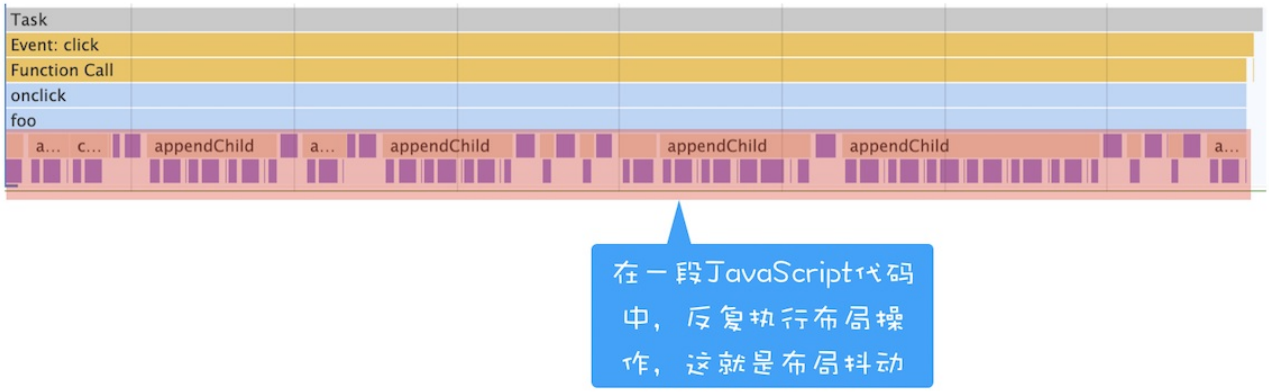
```
function foo() {
    let main_div = document.getElementById("mian_div")
    //为了避免强制同步布局，在修改DOM之前查询相关值
    console.log(main_div.offsetHeight)
    let new_node = document.createElement("li")
    let textnode = document.createTextNode("time.geekbang")
    new_node.appendChild(textnode);
    document.getElementById("mian_div").appendChild(new_node);

}
```

3. 避免布局抖动

还有一种比强制同步布局更坏的情况，那就是布局抖动。所谓布局抖动，是指在一次JavaScript执行过程中，多次执行强制布局和抖动操作。为了直观理解，你可以看下面的代码：

```
function foo() {
    let time_li = document.getElementById("time_li")
    for (let i = 0; i < 100; i++) {
        let main_div = document.getElementById("mian_div")
        let new_node = document.createElement("li")
        let textnode = document.createTextNode("time.geekbang")
        new_node.appendChild(textnode);
        new_node.offsetHeight = time_li.offsetHeight;
        document.getElementById("mian_div").appendChild(new_node);
    }
}
```

我们在一个for循环语句里面不断读取属性值，每次读取属性值之前都要进行计算样式和布局。执行代码之后，使用Performance记录的状态如下所示：



Performance中关于布局抖动的表现

从上图可以看出，在foo函数内部重复执行计算样式和布局，这会大大影响当前函数的执行效率。这种情况的避免方式和强制同步布局一样，都是尽量不要在修改DOM结构时再去查询一些相关值。

4. 合理利用CSS合成动画

合成动画是直接在合成线程上执行的，这和在主线程上执行的布局、绘制等操作不同，如果主线程被JavaScript或者一些布局任务占用，CSS动画依然能继续执行。所以要尽量利用好CSS合成动画，如果能让CSS处理动画，就尽量交给CSS来操作。

另外，如果能提前知道对某个元素执行动画操作，那就最好将其标记为will-change，这是告诉渲染引擎需要将该元素单独生成一个图层。

5. 避免频繁的垃圾回收

我们知道JavaScript使用了自动垃圾回收机制，如果在一些函数中频繁创建临时对象，那么垃圾回收器也会

频繁地去执行垃圾回收策略。这样当垃圾回收操作发生时，就会占用主线程，从而影响到其他任务的执行，严重的话还会让用户产生掉帧、不流畅的感觉。

所以要尽量避免产生那些临时垃圾数据。那该怎么做呢？可以尽可能优化储存结构，尽可能避免小颗粒对象的产生。

总结

好了，今天就介绍到这里，下面我来总结下本文的主要内容。

我们主要讲解了如何系统优化加载阶段和交互阶段的页面。

在加载阶段，核心的优化原则是：优化关键资源的加载速度，减少关键资源的个数，降低关键资源的RTT次数。

在交互阶段，核心的优化原则是：尽量减少一帧的生成时间。可以通过减少单次JavaScript的执行时间、避免强制同步布局、避免布局抖动、尽量采用CSS的合成动画、避免频繁的垃圾回收等方式来减少一帧生成的时长。

思考时间

那你来分析下新浪官网（https://www.sina.com.cn/ ）在加载阶段和交互阶段所存在的一些性能问题。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言：

- HB 2019-10-01 11:57:16
  为老师点赞，假期不断更 [3赞]

作者回复2019-10-01 17:49:23
争取快点把主干课程写完，然后还要整理几百条留言，来系统地答疑

- 阿哲 2019-10-03 09:53:10
加载阶段渲染流水线的配图中，css解析器和v8引擎是不是写反了？ [1赞]

- 柴柴 2019-10-02 15:32:08
老师辛苦！ [1赞]

- 石川 2019-10-05 18:32:29
如何减少关键资源个数那一段，应该是加上 async 或 defer 属性。两者区别是，async 可以在解析 HTML 时并行下载 JS 文件，下载完成之后，暂停 HTML 解析，执行完 JS 再接着解析；而 defer 会并行下载 JS ，等 HTML 解析完之后再执行 JS

- This 2019-10-05 11:35:35
老师的课程是我遇到干货最满的，希望老师陆续出其他课程

- mfist 2019-10-03 22:03:18
加载阶段：
通过分析network中关键资源（html文件 js文件 css文件）的大小，个数，只找到一个可能性能问题：html文件是128kb比较大，网站本身已经开启gzip http2 多个静态资源域名、开启缓存等多个优化手段
交互阶段
新浪首页页面加载完成后，滚动页面查看次屏页面，没有太多的交互，查看performance没有发现太明显的性能问题
今日总结：
一个页面从生命周期的维度主要分为三个阶段：加载阶段、交互阶段、关闭阶段。
1. 加载阶段影响网页首次渲染的关键资源几个指标：个数、大小、RTT（round trip time）。通常一个HTTP的数据包在14kb左右。
2. 交互阶段的优化主要是指渲染进程渲染帧速度。如何让单个帧生成的速度变快呢?
* 减少JavaScript脚本执行时间
* 避免强制同步布局，添加 删除dom后计算样式布局是在另外一个任务中执行的，这时候获取样式信息，会将其变成同步任务。
* 避免布局抖动
* 合理利用CSS合成动画（标识 will-change 单独生成一个图层）
* 避免频繁的垃圾回收。（尽量避免临时垃圾数据，优化存储结构，避免小颗粒对象产生）

- nickbing 2019-10-02 21:54:08
老师我想问下，内联CSS和JavaScript不是也让HTML文件加载时间变长了吗？最后结果不还是一样？