浏览器性能优化-渲染性能

2017-10-08

| 分类于 前端， 浏览器 | | 阅读次数

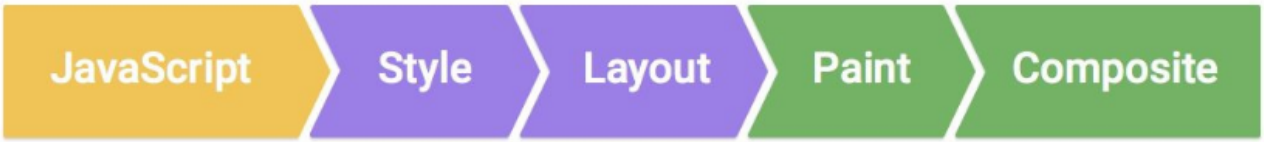
在浏览器渲染过程与性能优化一文中（建议先去看一下这篇文章再来阅读本文），我们了解与认识了浏览器的关键渲染路径以及如何优化页面的加载速度。在本文中，我们主要关注的是如何提高浏览器的渲染性能（浏览器进行布局计算、绘制像素等操作）与效率。

很多网页都使用了看起来效果非常酷炫的动画与用户进行交互，这些动画效果显著提高了用户的体验，但如果因为性能原因导致动画的每秒帧数太低，反而会让用户体验变得更差（如果一个酷炫的动画效果运行起来总是经常卡顿或者看起来反应很慢，这些都会让用户感觉糟透了）。

一个流畅的动画需要保持在每秒60帧，换算成毫秒浏览器需要在10毫秒左右完成渲染任务（每秒有1000毫秒，1000/60 约等于 16毫秒一帧，但浏览器还有其他工作需要占用时间，所以估算为10毫秒），如果能够理解浏览器的渲染过程并发现性能瓶颈对其优化，可以使你的项目变得具有交互性且动画效果如飘柔般顺滑。

像素管道

所谓像素管道其实就是浏览器将渲染树绘制成像素的流程。管道的每个区域都有可能产生卡顿，即管道中的某一区域如果发生变化，浏览器将会进行自动重排，然后重新绘制受影响的区域。



- JavaScript：**该区域其实指的是实现动画效果的方法**，一般使用JavaScript来实现动画，例如JQuery的animate函数、对一个数据集进行排序或动态添加一些DOM节点等。当然，也可以使用其他的方法来实现动画效果，像CSS的Animation、Transition和Transform。

- Style：**该区域为样式计算阶段，浏览器会根据选择器（就是css选择器，如.td）计算出哪些节点应用哪些css规则，然后计算出每个节点的最终样式并应用到节点上。**

- Layout：**该区域为布局计算阶段，浏览器会在该过程中根据节点的样式规则来计算它要占据的空间大小以及在屏幕中的位置。**

- Paint：**该区域为绘制阶段，浏览器会先创建绘图调用的列表，然后填充像素。**绘制阶段会涉及到文本、颜色、图像、边框和阴影，基本上包括了每个可视部分。绘制一般是在多个图层（用过Photoshop等图片编辑软件的童鞋一定很眼熟图层这个词，这里的图层的含义其实是差不多的）上完成的。

- Composite：**该区域为合成阶段，浏览器将多个图层按照正确顺序绘制到屏幕上。**

假设我们修改了一个几何属性（例如宽度、高度等影响布局的属性），这时Layout阶段受到了影响，浏览器必须检查所有其他区域的元素，然后自动重排页面，任何受到影响的部分都需要重新绘制，并且最终绘制的元素还需要重新进行合成（简单地说就是整个像素管道都要重新执行一遍）。

如果我们只修改了不会影响页面布局的属性，例如背景图片、文字颜色等，那么浏览器会跳过布局阶段，但仍需要重新绘制。

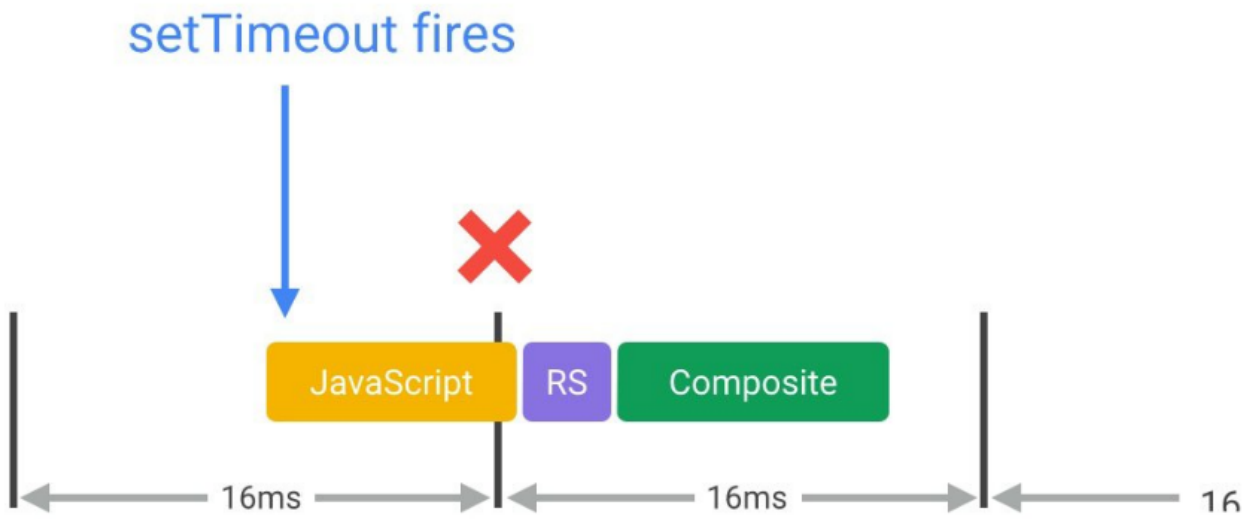
又或者，我们只修改了一个不影响布局也不影响绘制的属性，那么浏览器将跳过布局与绘制阶段，显然这种改动是性能开销最小的。

如果想要知道每个css属性将会对哪个阶段产生怎样的影响，请去CSS Triggers，该网站详细地说明了每个css属性会影响到哪个阶段。

使用RequestAnimationFrame函数实现动画

我们经常使用JavaScript来实现动画效果，然而时机不当或长时间运行的JavaScript可能就是导致你性能下降的原因。

避免使用setTimeout()或者setInterval()函数来实现动画效果，这种做法的主要问题是回调将会在帧中的某个时间点运行，这可能会刚好在末尾（会丢失帧导致发生卡顿）。



有些第三方库仍在使用setTimeout()&setInterval()函数来实现动画效果，这会产生很多不必要的性能下降，例如老版本的JQuery，如果你使用的是JQuery3，那么不必为此担心，JQuery3已经全面改写了动画模块，采用了requestAnimationFrame()函数来实现动画效果。但如果你使用的是之前版本的JQuery，那么就需要jquery-requestAnimationFrame来将setTimeout()替换为requestAnimationFrame()函数。

读到这里，想必一定会对requestAnimationFrame()产生好奇。要想得到一个流畅的动画，我们希望让视觉变化发生在每一帧的开头，而保证JavaScript在帧开始时运行的方式则是使用requestAnimationFrame()函数，本质上它与setTimeout()没有什么区别，都是在递归调用同一个回调函数来不断更新画面以达到动画的效果，requestAnimationFrame()的使用方法如下：

```
function updateScreen(time) {
}
requestAnimationFrame(updateScreen);
```

并不是所有浏览器都支持requestAnimationFrame()函数，如IE9（又是万恶的IE），但基本上现代浏览器都会支持这个功能的，如果你需要兼容老旧版本的浏览器，可以使用以下函数。

```
(function() {
    var lastTime = 0;
    var vendors = ['ms', 'moz', 'webkit', 'o'];
    for(var x = 0; x < vendors.length && !window.requestAnimationFrame; ++x) {
        window.requestAnimationFrame = window[vendors[x]+'RequestAnimationFrame'];
        window.cancelAnimationFrame = window[vendors[x]+'CancelAnimationFrame']
                                   || window[vendors[x]+'CancelRequestAnimationFrame'];
    }
    if (!window.requestAnimationFrame)
        window.requestAnimationFrame = function(callback, element) {
            var currTime = new Date().getTime();
```

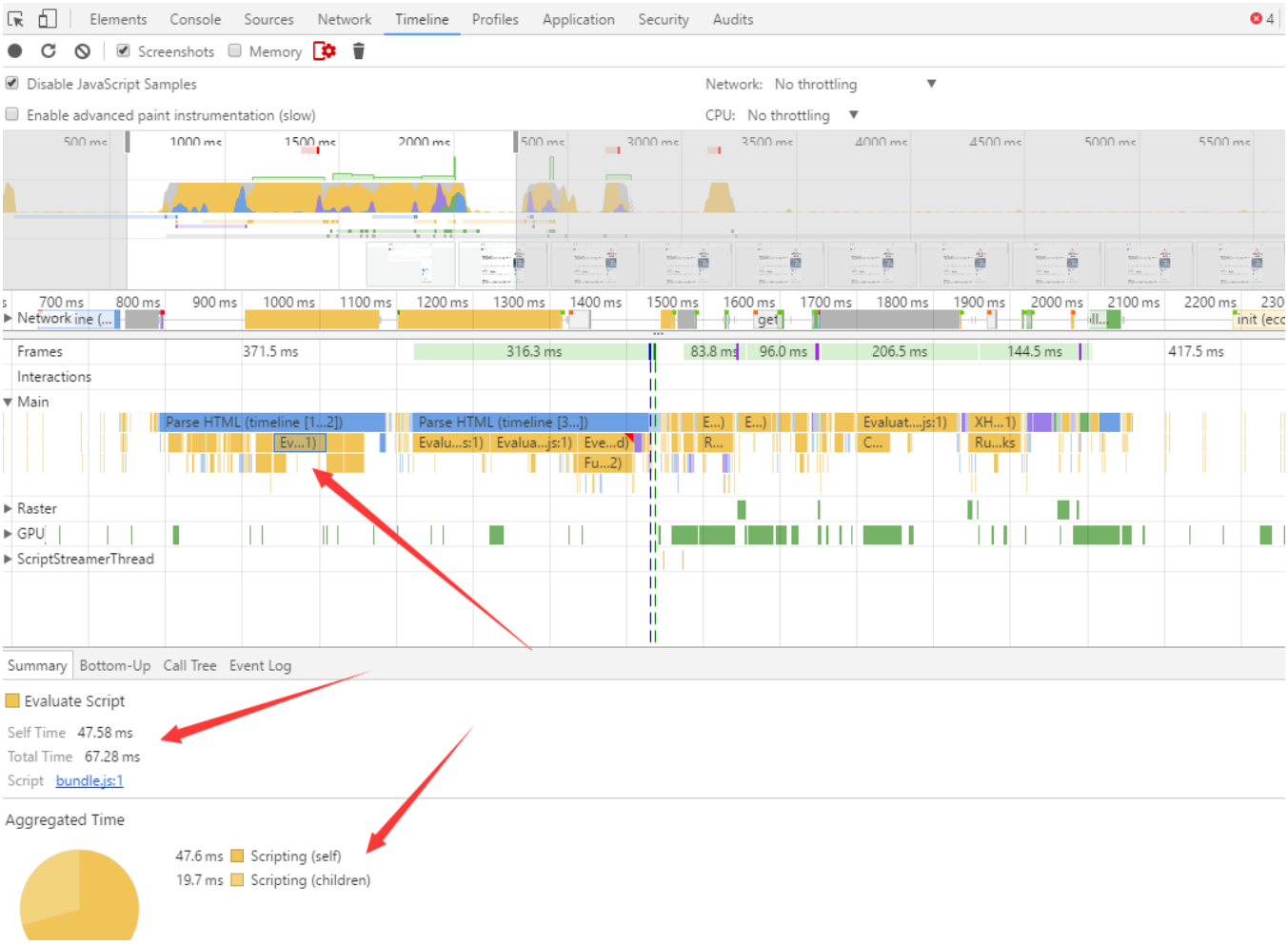
```
            var timeToCall = Math.max(0, 16 - (currTime - lastTime));
            var id = window.setTimeout(function() { callback(currTime + timeToCall); },
              timeToCall);
            lastTime = currTime + timeToCall;
            return id;
        };
    if (!window.cancelAnimationFrame)
        window.cancelAnimationFrame = function(id) {
            clearTimeout(id);
        };
}());


```

Web Workers

我们知道JavaScript是单线程的，但浏览器可不是单线程的。JavaScript在浏览器的主线程上运行，这恰好与样式计算、布局等许多其他情况下的渲染操作一起运行，如果JavaScript的运行时间过长，就会阻塞这些后续工作，导致帧丢失。

使用Chrome开发者工具的Timeline功能可以帮助我们查看每个JavaScript脚本的运行时间（包括子脚本），帮助我们发现并突破性能瓶颈。



在找到影响性能的JavaScript脚本后，我们可以通过Web Workers进行优化。Web Workers是HTML5提出的一个标准，它可以让**JavaScript**脚本运行在后台线程（类似于创建一个子线程），而后台线程不会影响到主线程中的页面。不过，使用**Web Workers**创建的线程是不能操作**DOM**树的（这也是Web Workers没有颠覆JavaScript是单线程的原因，JavaScript之所以一直是单线程设计主要也是因为为了避免多个脚本操作DOM树的同步问题，这会提高很多复杂性），所以它只适合于做一些纯计算的工作（数据的排序、遍历等）。

如果你的JavaScript必须要在主线程中执行，那么只能选择另一种方法。将一个大任务分割为多个小任务（每个占用时间不超过几毫秒），并且在每帧的requestAnimationFrame()函数中运行：

```
1
2
3
4
5
var taskList = breakBigTaskIntoMicroTasks(monsterTaskList);
```

```
requestAnimationFrame(processTaskList);
function processTaskList(taskStartTime) {
  var taskFinishTime;
  do {
    var nextTask = taskList.pop();
    processTask(nextTask);
    taskFinishTime = window.performance.now();
  } while (taskFinishTime - taskStartTime < 3);
  if (taskList.length > 0)
    requestAnimationFrame(processTaskList);
}
```

创建一个Web Workers对象很简单，只需要调用Worker()构造器，然后传入指定脚本的URI。现代主流浏览器均支持Web Workers，除了Internet Explorer（又是万恶的IE），所以我们在下面的示例代码中还需要检测浏览器是否兼容。

```
var myWorker;
if (typeof(Worker) !== "undefined") {
        myWorker = new Worker("worker.js");
} else {
}
```

Web Workers与主线程之间通过postMessage()函数来发送信息，使用onmessage()事件处理函数来响应消息（主线程与子线程之间并没有共享数据，只是通过复制数据来交互）。

```
main.js:
myWorker.postMessage("Hello,World");
```

```
console.log('Message posted to worker');
worker.js:
onmessage = function(data) {
    console.log("Message received from main script.");
        console.log("Posting message back to main script.");
        postMessage("Hello~");
}
main.js:
myWorker.onmessage = function(data) {
    console.log("Received message: " + data);
}
```

如果你需要从主线程中立刻终止一个运行中的worker，可以调用worker的`terminate()`函数：

```
myWorker.terminate();
```

myWorker会被立即杀死，不会有任何机会让它继续完成剩下的工作。而在worker线程中也可以调用`close()`函数进行关闭：

```
close();
```

有关更多的`Web Workers`使用方法，请参考Using Web Workers - Web APIs | MDN。

降低样式计算的复杂度

每次修改DOM和CSS都会导致浏览器重新计算样式，在很多情况下还会对页面或页面的一部分重新进行布局计算。

计算样式的第一部分是创建一组匹配选择器（用于计算哪些节点应用哪些样式），第二部分涉及从匹配选择器中获取所有样式规则，并计算出节点的最终样式。

通过降低选择器的复杂性可以提升样式计算的速度。

下面是一个复杂的CSS选择器：

```
.box:nth-last-child(-n+1) .title {

}
```

浏览器如果想要找到应用该样式的节点，需要先找到有.title类的节点，然后其父节点正好是负n个子元素+1个带.box类的节点。浏览器计算此结果可能需要大量的时间，但我们可以把选择器的预期行为更改为一个类：

```
.final-box-title {

}
```

我们只是将CSS的命名模块化（降低选择器的复杂性），然后只让浏览器简单地将选择器与节点进行匹配，这样浏览器计算样式的效率会提升许多。

BEM是一种模块化的CSS命名规范，使用这种方法组织CSS不仅结构上十分清晰，也对浏览器的样式查找提供了帮助。

BEM其实就是Block,Element,Modifier，它是一种基于组件的开发方式，其背后的思想就是将用户界面划分为独立的块。这样即使是使用复杂的UI也可以轻松快速地开发，并且模块化的方式可以提高代码的复用性。

Block是一个功能独立的页面组件（可以被重用），Block的命名方式就像写Class名一样。如下面的.button就是代表<button>的Block。

```
.button {
    background-color: red;
}
<button class="button">I'm a button</button>
```

Element是一个不能单独使用的Block的复合部分。可以认为Element是Block的子节点。

```
<!-- `search-form`是一个block -->
```

```
<form class="search-form">
    <!-- 'search-form__input'是'search-form' block中的一个element -->
    <input class="search-form__input">
    <!-- 'search-form__button'是'search-form' block中的一个element  -->
    <button class="search-form__button">Search</button>
</form>

```

Modifier是用于定义Block或Element的外观、状态或行为的实体。假设，我们有了一个新的需求，对button的背景颜色使用绿色，那么我们可以使用Modifier对.button进行一次扩展：

```

.button {
    background-color: red;
}
.button--secondary {
    background-color: green;
}

```

第一次接触BEM的童鞋可能会对这种命名方式感到奇怪，但BEM重要的是模块化与可维护性的思想，至于命名完全可以按照你所能接受的方式修改。限于篇幅，本文就不再继续探讨BEM了，感兴趣的童鞋可以去看BEM的官方文档。

避免强制同步布局和布局抖动

浏览器每次进行布局计算时几乎总是会作用到整个DOM，如果有大量元素，那么将会需要很长时间才能计算出所有元素的位置与尺寸。

所以我们应当尽量避免在运行时动态地修改几何属性（宽度、高度等），因为这些改动都会导致浏览器重新进行布局计算。如果无法避免，那么要优先使用**Flexbox**，它会尽量减少布局所需的开销。

强制同步布局就是使用JavaScript强制浏览器提前执行布局。需要先明白一点，**在JavaScript运行时，来自上一帧的所有旧布局值都是已知的。**

以下代码为例，它在每一帧的开头输出了元素的高度：

```
requestAnimationFrame(logBoxHeight);
function logBoxHeight() {
  console.log(box.offsetHeight);
}
```

但如果在请求高度之前，修改了其样式，就会出现问题，浏览器必须先应用样式，然后进行布局计算，之后才能返回正确的高度。这是不必要的且会产生非常大的开销。

```
function logBoxHeight() {
  box.classList.add('super-big');
  console.log(box.offsetHeight);
}
```

正确的做法，应该利用浏览器可以使用上一帧布局值的特性，然后再执行任何写操作：

```
function logBoxHeight() {
  console.log(box.offsetHeight);
  box.classList.add('super-big');
}
```

如果接二连三地发生强制同步布局，那么就会产生布局抖动。以下代码循环处理一组段落，并设置每个段落的宽度以匹配一个名为“box”的元素的宽度。

```
function resizeAllParagraphsToMatchBlockWidth() {
  for (var i = 0; i < paragraphs.length; i++) {
    paragraphs[i].style.width = box.offsetWidth + 'px';
  }
}
```

这段代码的问题在于每次迭代都会读取`box.offsetWidth`，然后立即使用此

值来更新段落的宽度。在循环的下次迭代中，浏览器必须考虑样式更新这一事实（box.offsetWidth是在上一次迭代中请求的），因此它必须应用样式更改，然后执行布局。这会导致每次迭代都会产生强制同步布局，正确的做法应该先读取值，然后再写入值。

```

var width = box.offsetWidth;

function resizeAllParagraphsToMatchBlockWidth() {
  for (var i = 0; i < paragraphs.length; i++) {
    paragraphs[i].style.width = width + 'px';
  }
}

```

要想轻松地解决这个问题，可以使用FastDOM进行批量读取与写入，它可以防止强制布局同步与布局抖动。

使用不会触发布局与绘制的属性来实现动画

在像素管道一节中，我们发现有种属性修改后会跳过布局与绘制阶段，这显然会减少不少性能开销。目前只有两种属性符合这个条件：transform和opacity 。

需要注意的是，使用transform和opacity时，更改这些属性所在的元素应处于其自身的图层，所以我们需要将设置动画的元素单独新建一个图层（这样做的好处是该图层上的重绘可以在不影响其他图层上元素的情况下进行处理。如果你用过Photoshop，想必能够理解多图层工作的方便之处）。

创建新图层的最佳方式是使用will-change属性，该属性告知浏览器该元素会有哪些变化，这样浏览器可以在元素属性真正发生变化之前提前做好对应的优化准备工作。

```

.moving-element {
  will-change: transform;
}
// 对于不支持 will-change 但受益于层创建的浏览器，需要使用（滥用）3D 变形来强制创建一个新层
```

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

```

但不要认为**`will-change`**可以提高性能就随便滥用，使用**`will-change`**进行预优化与创建图层都需要额外的内存和管理开销，随便滥用只会得不偿失。

参考文献

- Web | Google Developers
- Using Web Workers - Web APIs | MDN
- will-change - CSS | MDN
- Quick start / Methodology / BEM