常用的 DOM 优化

随着用户体验的日益重视，前端性能对用户体验的影响备受关注，但由于引起性能问题的原因相对复杂，我们很难但从某一方面或某几个方面来全面解决它，接下来用一系列文章来深层次探讨与梳理有关 Javascript 性能的方方面面，以填补并夯实大家的知识结构。

接下来我们来聊一聊关于 DOM 操作相关的性能优化。前端工程师被退出到现在，一直说的的一句话：操作 DOM 的成本很高，不要轻易去操作 DOM。尤其是 React、vue 等 MV*框架的出现，数据驱动视图的模式越发深入人心，jQuery 时代提供的强大便利地操作 DOM 的 API 在前端工程里用的越来越少。刨根问底，这里说的成本，到底高在哪儿呢？

DOM 操作成本到底高在哪儿？

什么是 DOM？可能很多人第一反应就是 div、p、span 等 html 标签（至少我是），但要知道，DOM 是 Model，是 Object Model，对象模型，是为 HTML（and XML）提供的 API。HTML(Hyper Text Markup Language)是一种标记语言，HTML 在 DOM 的模型标准中被视为对象，DOM 只提供编程接口，却无法实际操作 HTML 里面的内容。但在浏览器端，前端们可以用脚本语言（JavaScript）通过 DOM 去操作 HTML 内容。

实质上还存在 CSSOM：CSS Object Model，浏览器将 CSS 代码解析成树形的数据结构，与 DOM 是两个独立的数据结构。

接下来说一说浏览器渲染。过程。

讨论 DOM 操作成本，肯定要先了解该成本的来源，那么就离不开浏览器渲染。

1. 解析 HTML，构建 DOM 树（这里遇到外链，此时会发起请求）
2. 解析 CSS，生成 CSS 规则树
3. 合并 DOM 树和 CSS 规则，生成 render 树
4. 布局 render 树（Layout/reflow），负责各元素尺寸、位置的计算
5. 绘制 render 树（paint），绘制页面像素信息
6. 浏览器会将各层的信息发送给 GPU，GPU 将各层合成（composite），显示在屏幕上

1.构建 DOM 树

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
```

```
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```

无论是 DOM 还是 CSSOM，都是要经过 Bytes → characters → tokens → nodes → object model 这个过程。

DOM 树构建过程：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。属于深度优先遍历过程。

2.构建 CSSOM 树

上述也提到了 CSSOM 的构建过程，也是树的结构，在最终计算各个节点的样式时，浏览器都会先从该节点的普遍属性（比如 body 里设置的全局样式）开始，再去应用该节点的具体属性。还有要注意的是，每个浏览器都有自己默认的样式表，因此很多时候这棵 CSSOM 树只是对这张默认样式表的部分替换。

3.生成 render 树

简单描述这个过程：

DOM 树从根节点开始遍历可见节点，这里之所以强调了“可见”，是因为如果遇到设置了类似 display: none;的不可见节点，在 render 过程中是会被跳过的（但 visibility: hidden; opacity: 0 这种仍旧占据空间的节点不会被跳过 render），保存各个节点的样式信息及其余节点的从属关系。

4.Layout 布局

有了各个节点的样式信息和属性，但不知道各个节点的确切位置和大小，所以要通过布局将样式信息和属性转换为实际可视窗口的相对大小和位置。

5.Paint 绘制

万事俱备，最后只要将确定好位置大小的各节点，通过 GPU 渲染到屏幕的实际像素。

Tips

- 在上述渲染过程中，前 3 点可能要多次执行，比如 js 脚本去操作 dom、更改 css 样式时，浏览器又要重新构建 DOM、CSSOM 树，重新 render，重新 layout、paint；
- Layout 在 Paint 之前，因此每次 Layout 重新布局（reflow 回流）后都要重新出发 Paint 渲染，这时又要去消耗 GPU；
- Paint 不一定会触发 Layout，比如改个颜色改个背景；（repaint 重绘）
- 图片下载完也会重新出发 Layout 和 Paint；

何时触发 reflow 和 repaint

reflow(回流): 根据 Render Tree 布局(几何属性)，意味着元素的内容、结构、位置或尺寸发生了变化，需要重新计算样式和渲染树；
repaint(重绘): 意味着元素发生的改变只影响了节点的一些样式（背景色，边框颜色，文字颜色等），只需要应用新样式绘制这个元素就可以了；
reflow 回流的成本开销要高于 repaint 重绘，一个节点的回流往往回导致子节点以及同级节点的回流；

引起 reflow 回流

现代浏览器会对回流做优化，它会等到足够数量的变化发生，再做一次批处理回流。

1. 页面第一次渲染（初始化）
2. DOM 树变化（如：增删节点）
3. Render 树变化（如：padding 改变）
4. 浏览器窗口 resize
5. 获取元素的某些属性： 浏览器为了获得正确的值也会提前触发回流，这样就使得浏览器的优化失效了，这些属性包括 offsetLeft、offsetTop、offsetWidth、offsetHeight、 scrollTop/Left/Width/Height、clientTop/Left/Width/Height、调用了 getComputedStyle()或者 IE 的 currentStyle

引起 repaint 重绘

1. reflow 回流必定引起 repaint 重绘，重绘可以单独触发
2. 背景色、颜色、字体改变（注意：字体大小发生变化时，会触发回流）

优化 reflow、repaint 触发次数

- 避免逐个修改节点样式，尽量一次性修改
- 使用 DocumentFragment 将需要多次修改的 DOM 元素缓存，最后一次性 append 到真实 DOM 中渲染
- 可以将需要多次修改的 DOM 元素设置 display: none，操作完再显示。（因为隐藏元素不在 render 树内，因此修改隐藏元素不会触发回流重绘）
- 避免多次读取某些属性（见上）
- 将复杂的节点元素脱离文档流，降低回流成本

操作 DOM 具体的成本，说到底是造成浏览器回流 reflow 和重绘 reflow，从而消耗 GPU 资源。

既然 DOM 操作是很耗性能的，我们该怎么做尽量的减少性能的损耗呢？

DOM 优化常用方法

- 优化节点修改。
  - 使用 cloneNode 在外部更新节点然后再通过 replace 与原始节点互换。

```
;

var orig = document.getElementById('container');
var clone = orig.cloneNode(true);
var list = ['foo', 'bar', 'baz'];
var content;
for (var i = 0; i < list.length; i++) {
   content = document.createTextNode(list[i]);
   clone.appendChild(content);
}
orig.parentNode.replaceChild(clone, orig)
```

- 优化节点添加

  多个节点插入操作，即使在外面设置节点的元素和风格再插入，由于多个节点还是会引发多次 reflow。

  - 优化的方法是创建 DocumentFragment，在其中插入节点后再添加到页面。
    - 如 JQuery 中所有的添加节点的操作如 append，都是最终调用 DocumentFragment 来实现的。

```
createSafeFragment(document) {
     var list = nodeNames.split( "|" ),
         safeFrag = document.createDocumentFragment();
```

```
    if (safeFrag.createElement) {
        while (list.length) {
            safeFrag.createElement(
                list.pop();
            );
        };
    };
    return safeFrag;
};
```

优化 CSS 样式转换。

如果需要动态更改 CSS 样式，尽量采用触发 reflow 次数较少的方式。

- 如以下代码逐条更改元素的几何属性，理论上会触发多次 reflow。
- element.style.fontWeight = 'bold';
- element.style.marginLeft= '30px';
  element.style.marginRight = '30px';

  可以通过直接设置元素的 className 直接设置，只会触发一次 reflow。

  element.className = 'selectedAnchor';

- 减少 DOM 元素数量
  - 在 console 中执行命令查看 DOM 元素数量。
  - `document.getElementsByTagName('*').length`
  - 正常页面的 DOM 元素数量一般不应该超过 1000。
  - DOM 元素过多会使 DOM 元素查询效率，样式表匹配效率降低，是页面性能最主要的瓶颈之一。
- DOM 操作优化。
  - DOM 操作性能问题主要有以下原因。
    - DOM 元素过多导致元素定位缓慢。
    - 大量的 DOM 接口调用。
      - JAVASCRIPT 和 DOM 之间的交互需要通过函数 API 接口来完成，造成延时，尤其是在循环语句中。
    - DOM 操作触发频繁的 reflow(layout)和 repaint。
    - layout 发生在 repaint 之前，所以 layout 相对来说会造成更多性能损耗。
      - reflow(layout)就是计算页面元素的几何信息。
      - repaint 就是绘制页面元素。
    - 对 DOM 进行操作会导致浏览器执行回流 reflow。

    - 解决方案。
        - 纯 JAVASCRIPT 执行时间是很短的。
        - 最小化 DOM 访问次数，尽可能在 js 端执行。
        - 如果需要多次访问某个 DOM 节点，请使用局部变量存储对它的引用。
        - 谨慎处理 HTML 集合（HTML 集合实时连系底层文档），把集合的长度缓存到一个变量中，并在迭代中使用它，如果需要经常操作集合，建议把它拷贝到一个数组中。
        - 如果可能的话，使用速度更快的 API，比如 querySelectorAll 和 firstElementChild。
        - 要留意重绘和重排。
        - 批量修改样式时，离线操作 DOM 树。
        - 使用缓存，并减少访问布局的次数。
        - 动画中使用绝对定位，使用拖放代理。
        - 使用事件委托来减少事件处理器的数量。
- 优化 DOM 交互 >在 JAVASCRIPT 中，DOM 操作和交互要消耗大量时间，因为它们往往需要重新渲染整个页面或者某一个部分。
    - 最小化现场更新。
        - 当需要访问的 DOM 部分已经已经被渲染为页面中的一部分，那么 DOM 操作和交互的过程就是再进行一次现场更新。
            - 现场更新是需要针对现场（相关显示页面的部分结构）立即进行更新，每一个更改（不管是插入单个字符还是移除整个片段），都有一个性能损耗。
            - 现场更新进行的越多，代码完成执行所花的时间也越长。
    - 多使用 innerHTML。
        - 有两种在页面上创建 DOM 节点的方法：
            - 使用诸如 createElement()和 appendChild()之类的 DOM 方法。
            - 使用 innerHTML。
                - 当使用 innerHTML 设置为某个值时，后台会创建一个 HTML 解释器，然后使用内部的 DOM 调用来创建 DOM 结构，而非基于 JAVASCRIPT 的 DOM 调用。由于内部方法是编译好的而非解释执行，故执行的更快。 >对于小的 DOM 更改，两者效率差不多，但对于大的 DOM 更改，innerHTML 要比标准的 DOM 方法创建同样的 DOM 结构快得多。
- 回流 reflow。
    - 发生场景。
        - 改变窗体大小。
        - 更改字体。
        - 添加移除 stylesheet 块。
        - 内容改变哪怕是输入框输入文字。

- CSS 虚类被触发如 :hover。
- 更改元素的 className。
- 当对 DOM 节点执行新增或者删除操作或内容更改时。
- 动态设置一个 style 样式时（比如 element.style.width="10px"）。
- 当获取一个必须经过计算的尺寸值时，比如访问 offsetWidth、clientHeight 或者其他需要经过计算的 CSS 值。

- 解决问题的关键，就是限制通过 DOM 操作所引发回流的次数。
  - 在对当前 DOM 进行操作之前，尽可能多的做一些准备工作，保证 N 次创建，1 次写入。
  - 在对 DOM 操作之前，把要操作的元素，先从当前 DOM 结构中删除：
    - 通过 removeChild()或者 replaceChild()实现真正意义上的删除。
    - 设置该元素的 display 样式为“none”。
  - 每次修改元素的 style 属性都会触发回流操作。
  - element.style.backgroundColor = "blue";
    - 使用更改 className 的方式替换 style.xxx=xxx 的方式。
    - 使用 style.cssText = '';一次写入样式。
    - 避免设置过多的行内样式。
    - 添加的结构外元素尽量设置它们的位置为 fixed 或 absolute。
    - 避免使用表格来布局。
    - 避免在 CSS 中使用 JavaScript expressions(IE only)。
  - 将获取的 DOM 数据缓存起来。这种方法，对获取那些会触发回流操作的属性（比如 offsetWidth 等）尤为重要。
  - 当对 HTMLCollection 对象进行操作时，应该将访问的次数尽可能的降至最低，最简单的，你可以将 length 属性缓存在一个本地变量中，这样就能大幅度的提高循环的效率。