浏览器渲染机制

CRP: 关键渲染路径(critical render path)

围绕渲染的机制和步骤, 去详细的进行每一步的优化, 依次来提高页面的渲染速度和运行性能

步骤:

1. 从服务器基于HTTP网路请求回来的数据
    - 16进制的文件流
    - 浏览器把它解析为字符串(HTML字符串)
    - 按照W3C规则识别为一个个的节点
    - 生成XXX树
2. 访问页面, 首先请求回来的是HTML的文档, 浏览器开始自上而下渲染

    浏览器是可以开辟多个进程/线程的
    - GUI渲染线程: 渲染页面
    - JS引擎线程: 渲染JS代码
    - HTTP网络线程, 可以开辟N多个: 从服务器获取资源和数据
    - 定时器监听线程
    - DOM监听线程

- ....

3. 页面渲染过程

(CSS)

1. 遇到style内嵌样式,GUI直接渲染
    - CSS代码量较少的时候, 可以直接内嵌, 拉取HTML的时候,同时CSS的也拉取完毕, 渲染的时候同时渲染了.
    - 但如果CSS代码量多, 一方面会影响HTML的拉取速度, 同时也不利于代码维护, 此时用外链的方式更好
2. 遇到link, 浏览器开辟一个HTTP线程去请求资源文件信息, 同时GUI继续向下渲染(**异步**)
    - 浏览器同时能够发送的HTTP请求有限
        - 谷歌 (5-7个)
    - 超过最大并发限制的HTTP请求需要排队等待(即 HTTP请求一定是越少越好)
3. 遇到@import. 浏览器同样开辟HTTP请求资源, 但是GUI线程会被阻塞, 等资源请求完毕, GUI才能继续渲染(**同步**)
    - **真实项目中应该避免使用@import**

真实项目当中, 我们一般把link放在页面的头部, 是为了在没渲染DOM的时候, 就通知HTTP去请求CSS, 这样DOM渲染完, CSS也差不多回来, 更有效的利用时间, 提高页面渲染速度

(JavaScript)

1. 遇到<script> 会阻碍GUI的渲染
    - async:

      请求js资源是异步的, GUI继续渲染, 一旦JS资源请求回来, 会立即暂停GUI处理, 去渲染JavaScript代码...(这样可能导致代码还没有加载出对应的DOM)

      多个script标签, 加了async标签, 谁先加载完谁执行, 之间的依赖关系是无效的.
    - defer:

请求资源, 带到全部DOM加载完毕, 才会执行JavaScript代码(与LINK相似)

加了defer属性, 可以建立依赖关系, 是排队顺序执行.

一般把JS代码放在页面的底部, 防止其阻碍GUI的渲染, 如果不放在底部, 最好设置上async, defer属性.

浏览器渲染流程

DOM Tree (DomContentLoaded 事件触发) ⇒ 执行JS ❓ ⇒ CSSOM Tree ⇒ RENDER Tree渲染树(浏览器未来是按照这个树来绘制页面的) ⇒ Layout布局计算(回流/重排) ⇒ Painting绘制(重绘){ 分层绘制 }

- 页面第一次渲染,必然会引发一次回流和重绘
  - **重绘:** 元素的样式改变(大小, 位置, 宽高 等不变**)**

    如: outline, visibility, color, background-color
  - **回流:** 元素的大小或者位置发生了变化(当页面布局和几何信息发生变化的时候), 触发了重新布局, 导致渲染树重新计算布局和渲染

    注意: 回流一定会触发重绘, 而重绘不一定会回流

- 改变页面元素的位置和大小, 浏览器需要重新计算元素在视口的位置和大小信息, 重新计算的过程是回流/重绘, 一旦发生回流操作, 一定会触发重绘(**DOM操作消耗性能, 90%说的都是这个操作**)
- 若果只是普通样式的改变, 位置大小不改变, 只需要重回即可

性能优化:

CSS优化方案:

1. 标签语义化和避免深层次嵌套
2. CSS选择器渲染 从右到左

3. 尽快尽快地把CSS下载到客户端(充分利用HTTP多请求并发机制)

避免DOM的回流:

1. 放弃传统操作DOM的时代, 基于vue/react开始数据影响视图模式
2. 分离读写操作

样式集中改变

- **渲染队列机制**(新版浏览器的机制):

  如果遇到获取样式语法, 则会**刷新**浏览器渲染队列, 所以要注意**读写分离**

> 轮播图利用渲染队列机制解决循环问题

```
let container = document.querySelector('.container'),
    wrapper = container.querySelector('.wrapper'),
    step = 0,
    timer;

timer = setInterval(function () {
    step++;
    if (step >= 5) {
        // 立即回到第一张
        wrapper.style.transition = 'left 0s';
        wrapper.style.left = `0px`;
        // 运动到第二张
        step = 1;
        // 刷新渲染队列
        wrapper.offsetLeft;
    }
    wrapper.style.transition = 'left .3s';
    wrapper.style.left = `-${step*800}px`;
}, 2000);
```

缓存布局信息

元素批量修改

❌: 使用for循环批量增加元素

文档碎片:

```
// 将元素加到文档碎片, 再统一加到DOM树
let box = document.querySelector('#box')
  frag = documet.createDocumentFragment()
for(let i = 0;i < 10; i++) {
  let span = document.createElement('span')
  span.innerHTML = i + 1
  frag.appendChild(spam)
}
box.appendChild(frag)
```

动画元素脱离文档流

动画效果应用到position属性为absolute或者fixed的元素上

CSS3硬件加速(GPU加速)

牺牲平滑度换取速度

避免table布局和使用css的JavaScript表达式