渲染

浏览器运行机制

从Chorme源码看浏览器如何加载资源

- 14种资源 分为5类
  - enum Type : uint8_t {
    - kMainResource,
    - kImage,
    - kCSSStyleSheet,
    - kScript,
    - kFont,
    - kRaw,
    - kSVGDocument,
    - kXSLStyleSheet,
    - kLinkPrefetch,
    - kTextTrack,
    - kImportResource,
    - kMedia, // Audio or video file requested by a HTML5 media element
    - kManifest,
    - kMock // Only for testing
    - };
- 完美解释了预处理prefetch preload 以及 defer/async等是怎么来控制加载的
  - ResourceLoadPriority TypeToPriority(Resource::Type type) {
    - switch (type) {
      - case Resource::kMainResource:
      - case Resource::kCSSStyleSheet:
      - case Resource::kFont:
        - // Also parser-blocking scripts (set explicitly in loadPriority)
        - return kResourceLoadPriorityVeryHigh;
      - case Resource::kXSLStyleSheet:
        - DCHECK(RuntimeEnabledFeatures::XSLTEnabled());
      - case Resource::kRaw:
      - case Resource::kImportResource:
      - case Resource::kScript:
        - // Also visible resources/images (set explicitly in loadPriority)
        - return kResourceLoadPriorityHigh;
      - case Resource::kManifest:
      - case Resource::kMock:
        - // Also late-body scripts discovered by the preload scanner (set
        - // explicitly in loadPriority)
        - return kResourceLoadPriorityMedium;
      - case Resource::kImage:
      - case Resource::kTextTrack:
      - case Resource::kMedia:
      - case Resource::kSVGDocument:
        - // Also async scripts (set explicitly in loadPriority)
        - return kResourceLoadPriorityLow;
      - case Resource::kLinkPrefetch:
        - return kResourceLoadPriorityVeryLow;
    - }
    - return kResourceLoadPriorityUnresolved;
    - }

浏览器内核分为：渲染引擎和JS引擎

- 1.HTML 解释器：将 HTML 文档经过词法分析输出 DOM 树。
- 2. • CSS 解释器：解析 CSS 文档，生成样式规则。
- 3.图层布局计算模块：布局计算每个对象的精确位置和大小
- 4.视图绘制模块：进行具体节点的图像绘制，将像素渲染到屏幕上。
- 5.JavaScript 引擎：编译执行 Javascript 代码

浏览器渲染过程解析

- 1.parseHtml，解析html成DOM树
- 2.style计算样式 解析css创建CSSOM树(和dom解析并行)
- 3.layout计算图层布局
- 4.paint绘制图层
- 5.composit最浏览器会合并合各个图层，将数据由 CPU 输出给 GPU 最终绘制在屏幕上
- 总结：解析成DOM tree 解析CSSOM tree(并行) ——CSSOM和DOM结合成 render tree——从根节点递归调用计算大小位置的布局渲染树 Layout of the render tree——遍历渲染树绘制渲染树Painting the render tree

基于渲染流程的CSS优化建议

- CSS 选择符是从右到左进行匹配的
- 1.避免通配符
- 2.关注可以通过继承实现的属性，避免重复匹配重复定义
- 3.少用标签选择器
- 4.减少嵌套(3层)后代选择器的开销是最高的

加载顺序优化，告别阻塞

- html css js都有做有阻塞渲染的特性，为了缩短首次渲染时间，应该尽早尽快下载css
  - 1.尽快 将 CSS 放在 head 标签里
  - 2.尽早 启用 CDN 实现静态资源加载速度的优化
- 将加载到JS，JS 引擎抢走了渲染引擎的控制权，从而阻塞页面渲染
  - script正常加载→当代js加载执行完毕在做其他事
  - async模式 不会阻塞浏览器，加载时异步的，加载结束后立即执行JS
  - defer模式：加载时异步的，职业也会被推迟，当整个DOM解析完成，DOMContentLoaded出发后才会执行

重绘重排

- 哪些操作会导致重绘重排
  - 添加/删除DOM
  - 元素内容
  - 页面渲染初始化
  - 浏览器窗口大小改变
  - 改变DOM的集合属性，和他相关的节点都要被重新计算
    - width height padding margin left top border
  - 重绘：border-raduis box-shadow color
  - 最容易被忽略的操作：获取一些特定属性的值
    - 这些值需要即时计算LoffsetTop、offsetLeft、offsetWidth、offsetHeight、scrollTop、scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight
- 优化：最小化重绘和重排
  - 变量缓存DOM或布局信息，再操作(尤其是在一些循环中)
  - 独立的层：重排重绘 不会影响父层
    - 根元素 position transform 半透明 canvas video overflow 滤镜...
  - 批量修改DOM
    - 1让元素脱离文档流2修改 3 带回
    - Display:none 只触发一次
    - 将 DOM “离线”
  - 将原始元素拷贝到一个独立的节点中，操作这个节点，再覆盖原始元素
    - let old = document.querySelector('#mylist');
      let clone = old.cloneNode(true);
      appendNode(clone, data);
      old.parentNode.replaceChild(clone, old);
  - GPU加速：虽然生成独立的层，但是还是cpu一直在计算重绘重排，再传到GPU 进行合成
    - 开启GPU加速后 就省去CPU重排重绘的过程
    - GPU(图形处理器)擅长图形处理
      - 节约了
        - 1. CPU进行layout paint的时间
        - 2. CPU上传位图的时间
      - GPU 能对 快速对Texture 进行 偏移 旋转，缩放，修改透明度等
      - 浏览器渲染 会把 ele 放在GPU上 ele 的 transform 属性 会 开启GPU加速
      - 开启GPU加速的方法
        - 3d transform
        - 设置 opacity 属性
        - 设置 position 属性 fixed
        - 设置 filter 属性
        - 设置 will-change 属性
        - 设置 backface-visibility 属性 hidden
        - 设置 flex 属性
        - 设置 transition 属性
        - 设置 transform 属性
        - 设置 canvas 属性
        - 设置 video 属性
      - 开启GPU加速的坏处
        - 1. 内存 GPU 内存 是 有限的 开启 GPU 加速 会 使用 GPU 内存 来 存储 图层 信息 ，如果 开启 GPU 加速 的 图层 过多 ，会 导致 内存 不足 ，反而 会 导致 页面 卡顿 ，甚至 会 导致 浏览器 崩溃 ，所以 要 谨慎 开启 GPU 加速 ，并 在 合适 的 时机 开启 GPU 加速 ，在 不 需要 GPU 加速 的 时候 ，要 及时 关闭 GPU 加速 ，以 释放 内存 资源 。
        - 2. 文字 模糊 ，开启 GPU 加速 后 ，文字 可 能 会 出现 模糊 ，这 是 因 为 GPU 对 文字 的 渲染 方式 与 CPU 不 同 ，CPU 是 逐 像素 渲染 ，而 GPU 是 通过 纹理 来 渲染 ，所 以 开启 GPU 加速 后 ，文字 可 能 会 出现 模糊 ，这 是 正常 现象 ，可 以 通过 调整 字体 大小 ，或 使用 抗锯齿 技术 来 减轻 模糊 。
        - 3. 兼容性 ，并 不 是 所有 的 浏览器 都 支持 GPU 加速 ，特别 是 一些 老 的 浏览器 ，可 能 会 出现 兼容性 问题 ，需 要 进行 测试 和 调试 。
  - 减少重排
    - 1.使用 transform 替代 top
    - 2.使用 visibility 替换 display: none ，因 前 者 只 引起 重绘 ，后 者 会 引发 重排
    - 3.不 把 节点 的 属性 逐一 加入 样式 ，而 是 通过 重新 定义 样式 的 方式 来 一次性 更 改 属性
    - 4.不 把 节点 的 属性 值 放 在 一个 循环 里 当 成 循环 变量 来 用 ，这 会 导致 这 个 属性 的 每 一次 改变 都 会 引起 重排 ，建议 将 属性 值 缓存 入 变量 ，最 后 赋值 给 该 属性 ，这 样 只 会 引起 一次 重排 。
    - 5.不 使用 table 布局 ，一 个 小 的 改动 可 能 会 使 整个 table 重新 计算 布局 ，这 会 导致 很 大 的 计算 量 ，建议 使用 其 他 元素 模拟 表格 的 布局 ，比 如 给 div 设置 display:table 属性 ，让 其 具 有 table 的 特性 ，但 又 不 会 引起 table 的 重排 问题 。
    - 6.将 动画 脱离 文档 流 ，这 样 动画 的 重绘 不 会 影响 其 他 元素 ，浏览 器 会 为 其 单 独 开 一 个 图层 来 绘制 动画 ，这 样 可 以 减少 重排 的 范围 ，提高 性能 。
    - 7.将 频繁 重排 的 元素 设置 为 绝对 定位 ，这 样 其 重排 不 会 影响 其 他 元素 ，因 为 绝对 定位 的 元素 脱离 了 文档 流 ，其 重排 只 会 影响 自 身 及 子 元素 ，不 会 影响 其 他 元素 。
  - 减少重绘
    - 1.将 频繁 重绘 的 元素 设置 为 绝对 定位 ，这 样 其 重绘 不 会 影响 其 他 元素 ，因 为 绝对 定位 的 元素 脱离 了 文档 流 ，其 重绘 只 会 影响 自 身 及 子 元素 ，不 会 影响 其 他 元素 。
    - 2.将 频繁 重绘 的 元素 设置 为 固定 定位 ，这 样 其 重绘 不 会 影响 其 他 元素 ，因 为 固定 定位 的 元素 脱离 了 文档 流 ，其 重绘 只 会 影响 自 身 及 子 元素 ，不 会 影响 其 他 元素 。
    - 3.将 频繁 重绘 的 元素 设置 为 opacity 属性 ，这 样 其 重绘 不 会 影响 其 他 元素 ，因 为 opacity 属性 的 元素 脱离 了 文档 流 ，其 重绘 只 会 影响 自 身 及 子 元素 ，不 会 影响 其 他 元素 。
  - 重排一定会重绘，重绘不一定重排，重排的代价也比重绘高
  - what
    - 重排(回流)
      - 当我们对DOM的修改引起DOM的集合属性的变化的时候(宽高 显隐)等，浏览器需要重新计算元素的集合属性然后再将结果绘制
    - 重绘
      - 当我们对DOM修改值导致了样式的变化(颜色背景色)，浏览器需要重新绘制元素
  - 逐帧动画抖动解决方案：https://aotu.io/notes/2017/08/14/fix-sprite-anim/
  - Flush队列
    - 浏览器对重排有一定的优化，合并多次重排
      - 下例中其实只会触发一次重排
        - let container = document.getElementById('container')
        container.style.width = '100px'
        container.style.height = '200px'
        container.style.border = '10px solid red'
        container.style.color = 'red'
    - 浏览器缓存了一个 flush 队列，把我们触发的回流与重绘任务都塞进去，待到队列里的任务多起来，或者达到了一定的时间间隔，或者“不得已”的时候，再将这些任务一口气出队。
    - 但是有些操作会导致浏览器跳过自身的优化，而提前将 flush 队列的任务出队
      - 读取 offsetTop scrollTop width clientWidth getComputedStyle() getBoundingClientRect() 及时间的属性
    - 并不是所有的浏览器都会有

Event Loop和异步更新策略

- event loop
  - 事件循环中的异步队列有两种：macro（宏任务）队列和 micro（微任务）队列。
    - macro
      - setTimeout、setInterval、MessageChannel setImmediate、script（整体代码）、I/O 操作、UI 渲染等
    - micro
      - process.nextTick、Promise、MutationObserver（提供了监控DOM数所做更改的能力）
  - Event Loop 过程解析
    - 1.初始阶段: micro队列为空，macro队列只有一个script脚本
    - 2.全局上下文script被推入调用栈,执行同步代码，遇到一些接口调用等情况，产生新的macro-task,micro-task，分别推入到队列中，执行完代码，script出栈(macro-task 的执行和出队的过程) 需要注意的是，假如遇到的macro里面再遇到micro,也是要等到此次宏任务完成之后再去处理下一个宏文物
    - 3.macro-task出队，是指此次宏任务里产生的所有微任务，整队出队清空队列。
    - 4. 执行渲染操作，更新界面(重点：所以我们应该把异步更新策略放到微任务)
    - 5. 检查是否有 web worker任务，有则处理
    - 6. 循环以上过程，知道两个队列清空
- 渲染时机分析
  - 当我们需要在异步任务中实现 DOM 修改时，应该把它包装成 micro 任务
  - 剖析：
    - 上文中已经说明：event loop 的过程：macro ——micro ——render
    - 假如时机放在macro
      - 本次操作会推入到macro中，然后，执行script中剩余同步代码(macro)，再执行本次所有micro,之后再render,但是我们需要修改的操作，在后面的macro(可能还不止一个)，本次的render是一次无效的render
    - 假如时机放在micro
      - 本次操作推入到micro-task中，——执行本次macro task其他代码，执行本次产生的所以micro（包括本次修改）——render，good
- 异步更新策略
  - 当我们使用 Vue 或 React 提供的接口去更新数据时，这个更新并不会立即生效，而是会推入到一个队列里，待到适当的时机，队列中的更新任务会被批量触发，避免过度渲染。
    - 优点：只看结果，不看过程
  - 假如我们对同一状态多次修改是，传统的同步更新策略就会操作3次DOM，假如我们把这三次任务放到异步更新队列中，它是会在JS的层面上被批量执行完毕，当流程走到render是，他只会有意义的计算结果进行操作一次DOM
  - vue状态更新手法：nextTick »
    - Vue 每次想要更新一个状态的时候，会先把它这个更新操作给包装成一个异步操作派发出去
    - Vue 的异步任务默认情况下都是用 Promise 来包装的
    - 源码中：macroTimeFunc »
      - setImmediate——MessageChannel——setTimeout(func,0)
    - 源码中：microTimeFunc »
      - 派发 macro 任务/ micro 任务 派发的任务对象是 flushCallbacks
    - 源码中：flushCallbacks »
      - Vue 中每产生一个状态更新任务，它就会被塞进一个叫 callbacks 的数组（此处是任务队列的实现形式）中。这个任务队列在被丢进 micro 或 macro 队列之前，会先去检查当前是否有异步更新任务正在执行（即检查 pending 锁）。如果确认 pending 锁是开着的（false），就把它设置为锁上（true），然后对当前 callbacks 数组的任务进行派发（丢进 micro 或 macro 队列）和执行。设置 pending 锁的意义在于保证状态更新任务的有序进行，避免发生混乱。

服务端渲染?

- 运行机制和区别
  - SSR
    - 当用户第一次请求页面时，由服务器把需要的组件或页面渲染成 HTML 字符串，然后把它返回给客户端。客户端拿到手的，是可以直接渲染然后呈现给用户的 HTML 内容
  - 客户端渲染
    - 访问html文件 –> 初始化路由 –> 找到对应跟路由 –> 找到对应组件 –> 发请求 发给后端再渲染 导致白屏
    - 服务端会把渲染需要的静态文件发送给客户端，客户端加载过来之后，自己在浏览器里跑一遍 JS，根据 JS 的运行结果，生成相应的 DOM
- 解决了什么问题
  - 首屏加载速度过慢
  - 处于效益的考虑：SEO
- 实践SPA+SSR(用户访问到网站直接吐html)
  - 不管是vue还是react
    - 1.renderToString() 方法将vdom转成字符串
    - 二是把转化结果“塞”进模板里
- 你可能不需要SSR
  - 服务端渲染本质上是本该浏览器做的事情，分担给服务器去做 浪费性能
  - SPA的SEO和首屏渲染优化有很多替代方案
    - 预渲染Prerender：prerender-spa-plugin
    - 骨架屏预渲染

DOM优化原理与基本事件

- 为什么慢
  - 操作DOM是昂贵的：S 引擎和渲染引擎，“跨界沟通”要收过路费
  - 代码层面的优化：可参考《高性能jacascript》
- 优化
  - 较少dom操作
    - DOM Fragment等
    - 重绘重排话题(下面单独讲)
      - 重绘不一定导致回流，回流一定会导致重绘
    - 批量操作的异步更新策略(micro更新 另一个话题单独讲)

其他

- 应用级的：lazyload等（后面说原理）
  - chrome黑科技：loading 属性：auto/lazy/eager
    - Lazy一张图片被分为了两次去请求！第一次的状态码是 206（只请求头部信息），第二次的状态码才是 200
  - 文章：https://juejin.im/post/5a76bb95518825fa79247cbd
    https://juejin.im/post/583b10640ce463006ba2a71a
    https://mp.weixin.qq.com/s/HedCmFQOSHLZuKqtDB3uA
  - 依靠JS实现
    - 传统思路
    - IntersectionObserver API
  - 锦上添花：更好的方式的preview 的效果
    - 根据图片生成css渐变色背景
    - https://calendar.perfplanet.com/2018/gradient-image-placeholders/ 这篇文章中推荐的工具 GIP 进行转换
      在线转换的地址 https://tools.w3clubs.com/qip/
- 黑科技waifu2x(二次元福利)
  - https://waifu2x.booru.pics/
  - 利用深度卷积神经网络（其实就是深度学习）得到的一种对于图片的放大算法，并对图片进行降噪处理（因为放大的时候会使原有的噪点变得更加明显或者产生更多噪点），使得放大后的图片显得更加自然，甚至你会以为这就是原图
- pre-cache:img-2自动预缓存图像（技术点：IntersectionObserver Custom Elements and Shadow DOM）
  - 1.仅渲染用户可见的初始图像懒加载图像
  - 2.使用Web Worker预先缓存主线程上的所有其他图像
  - 3.进入视口时，从缓存中拿lazyload的图
  - 4.在用户等待初始图像时显示模糊的预览图像
  - 还可预先显示一个低分辨率模糊预览图
  - react+ts中声明img2
    - declare namespace JSX {
      - interface IntrinsicElements {
        - "img-2": React.Detailed<HTMLProps<React.HTMLAttributes<HTMLElement>>
        - }
      - }
    - 亲测后：width/height必填且只能是px,不适合移动端
- 预处理 preload prefetch等

图片优化

- 响应式图片
  - 动态内容
    - 小于 1024px 的时候显示 img-center.jpg，而在大于等于 1024px 的时候显示 img-full.jpg
      - <picture>
        - <source
          - media="(min-width: 1024px)"
          - srcset="img-full.jpg"
        - >
        - <img
          - src="img-center.jpg"
        - >
      - </picture>
  - 动态格式
    - 在支持 WebP 的浏览器上使用 img.webp，在不支持 WebP 的浏览器上使用 img.jpg
      - <picture>
        - <source
          - srcset="img.webp"
          - type="image/webp"
        - >
        - <img
          - src="img.jpg"
        - >
      - </picture>
  - 动态尺寸
    - 1.有哪些尺寸的图片
      2.图片显示的时候是什么尺寸
    - srcset 属性指定有哪些图片，分别是图片名和图片的尺寸，这里注意单位不用 px 而是 w，用于表示图片的固有宽度
    - sizes 属性告诉浏览器，这个图片在不同的条件下会是什么样的宽度。这个属性用于给到浏览器提示，并不会真正的指定 img 的宽度，我们还是需要另外使用 CSS 来指定
    - 这样 给定一个视口宽度，浏览器可以得知图片需要的宽度，然后根据 DPI 情况，在所有可选图片中选择最合适的一个
      - <img
        - src="img-400.jpg"
        - sizes="(min-width: 640px) 60vw, 100vw"
        - srcset="img-200.jpg 200w,
          - img-400.jpg 400w,
          - img-800.jpg 800w,
          - img-1200.jpg 1200w"
        - >
    - 动态尺寸的一种简化情况，根据DIP返回不同分辨率
    - 背景图片情况：image-set()function (postcss-preset-env)
      - background-image: image-set(
        - "foo.png" 1x,
        - "foo-2x.png" 2x,
        - "foo-print.png" 600dpi
        -)
      - less .bg-image(@url) {
        - background-image:~"url('@{url}@2x.png')";
        - @media (-webkit-min-device-pixel-ratio: 3), (min-device-pixel-ratio:3) {
          - background-image:~"url('@{url}@3x.png')";
        - }
        - }
    - 预处理器
      - <img
        - srcset="img-200.jpg,
          - img-300.jpg 1.5x,
          - img-400.jpg 2x"
        - src="img-400.jpg"
        - >
    - 适合：*
  - 动态尺寸
    - 关键：picture标签 picture 标签会从 source 中选择最合适的一个，然后将它的 URL 赋值给 img。对于不认识 picture 的旧浏览器，他们会忽略 picture，只渲染 img，标签内最后一个一定是img标签
    - 概念：根据客户端的情况返回最适合客户端的图片(不同图片内容/不同图片格式/不同图片尺寸/动态分辨率)
    - 图片的优化是在质量和性能的权衡，牺牲一部分成像质量为代价，换取更小的体积
  - 压缩
    - https://zh.recompressor.com/
    - https://tinypng.com/
    - https://squoosh.app
    - 命令行小工具oimg
    - 构建时压缩image-webpack-loader
    - imagemin-webpack-plugin
      - progressive: true 选项，这可以将图片转换为渐进式图片
    - 文章：从图片优化说起：https://citing.me/2019/07/29/image-optimization/ 该谈Web应用中的图片优化技巧及反思https://segmentfault.com/a/1190000020047949
      https://developers.google.com/web/fundamentals/design-and-ux/responsive/images
- 图片选型方案
  - JPEG/JPG
    - 有损压缩，体积小，加载快，不支持透明
    - 把图片体积压缩至原有体积的 50% 以下时，JPG 仍然可以保持住 60% 的品质
    - 缺陷：1处理矢量图和logo等线条感强颜色对比强烈的图，认为压缩会导致模糊 2.不支持透明
    - Baseline JPEG（标准型）VS Progressive JPEG（渐进式）
      - 渐进式更好，但是吃用户的CPU和内存
      - 本质是压缩算法不同：小波算法 vs 离散余弦变换算法
      - 基于DCT的渐进模式（Progressive DCT-based mode of operation）
        - 当图像传输的时间较长时，可将图像分数次处理，以从模糊到清晰的方式来传送图像（效果类似GIF在网络上的传输）。
      - 基于DCT的连续模式（Sequential DCT-based mode of operation）
        - 一次将图像由左到右、由上到下顺序处理。
      - 在photoshop中有“存储为web所用格式”，打开后选择“连续”就是渐进式JPEG。
    - JPG 格式以 24 位存储单个图，可以呈现多达 1600 万种颜色
    - 使用场景：
      - JPG 适用于呈现色彩丰富的图片，在我们日常开发中，JPG 图片经常作为大的背景图、轮播图或 Banner 图出现。
  - PNG(png8 png24)
    - 无损压缩、质量高、体积大、支持透明
    - 当你追求最佳的显示效果、并且不在意文件体积大小时，是推荐使用 PNG-24 的。
    - 8 位的 PNG 最多支持 256 种颜色，而 24 位的可以呈现约 1600 万种颜色
    - 缺陷：大
    - 先按照这两种格式分别输出，看 PNG-8 输出的结果是否会带来肉眼可见的质量损耗
    - 使用场景
      - PNG 在处理线条和颜色对比度方面的优势，我们主要用它来呈现小的 Logo、颜色简单且对比强烈的图片或背景纹理
  - SVG（可缩放矢量图形）
    - 基于 XML 语法的图像格式。SVG 对图像的处理不是基于像素点，而是是基于对图像的形状描述
    - 形式
      - 标签写入html
      - 作为单独的svg文件引入html <img src="文件名.svg" alt="">
    - 文本文件 体积小 兼容性好 可无限放大而不失真
  - 雪碧图CSS Sprites（小图标解决方案）
    - 将小图标和背景图像合并到一张图片上，然后利用 CSS 的背景定位来显示其中的每一部分的技术
    - 减少HTTP 请求，对内存和带宽更加友好
    - postcss-sprites
  - Apng
    - 为了替代老旧的 GIF 格式，但它目前并没有获得 PNG 组织官方的认可
  - Base64(小图标解决方案)
    - 文本文件 依赖编码 小图标解决方案 它并非一种图片格式，而是编码方式 (Base64 是作为雪碧图的补充而存在的)
    - base64是一种用于产出8Bit字节码的编码技术，对图片编码直接将编码结果插入到 html css中减少http请求
    - webpack 的 url-loader
    - 缺陷 »
      - 1.Base64 编码后，图片大小会膨胀为原文件的 4/3，太多则会让css文件体积失控渲染阻塞资源
      - 2.让浏览器的资源缓存策略功亏一篑：本来浏览器可以直接从本地缓存取出的图片，会相城多个页面重复下载
      - 3. IE8下兼容
      - 4. 不利于开发者工具调试与查看
    - 场景：只适合于小图片，大图片推不宜矣
  - WebP
    - 全能:WebP 像 JPEG 一样对细节丰富的图片信手拈来，像 PNG 一样支持透明，像 GIF 一样可以显示动态图片
    - 官方描述：与 PNG 相比，WebP 无损图像的尺寸缩小了 26%。在等效的 SSIM 质量指数下，WebP 有损图像比同类 JPEG 图像小 25–34%。无损 WebP 支持透明度（也称为 alpha 通道），仅需 22% 的额外字节。对于可以接受有损RGB压缩的情况，有损WebP还支持透明性，通常提供比PNG小3倍的文件大小。
    - 缺陷
      - 兼容性
      - 增加服务器的负担——和编码 JPG 文件相比，编码同样质量的 WebP 文件会占用更多的计算资源。
    - JPG/PNG 转换成 WebP:imagemin-webp
    - 业务场景：都可用唯一就是降级处理
      - 淘宝首页：//img.alicdn.com/imgextra/i2/45827046/O1CN01STNrOq2Ln7AbITOSY_!!0-saturn_solar.jpg_220x220.jpg_.webp 支持2种方式，支持webp的对图片字符串进行进行拼接
      - 在浏览器环境支持 WebP 的情况下，优先使用 WebP 格式，否则就把图片降级为 JPG 格式
      - https://www.jianshu.com/p/73ca9e8b986a
        - .webpbg(@url) {
          - background-image: url(@url);
          - .webpa & {
            - background-image: url('@{url}.webp');
          - }
        - }_html根节点设置 webp
      - 最好方案：把判断工作交给后端，由服务器根据 HTTP 请求头部的 Accept 字段来决定该回什么格式的图片，当 Accept 字段包含 image/webp 时，就返回 WebP 格式的图片，否则返回原图。