

性能优化琐碎事

注意：该知识点属于性能优化领域。

总的来说性能优化这个领域的很多内容都很碎片化，这一章节我们将学习这些碎片化的内容。

图片优化

计算图片大小

对于一张 $100 * 100$ 像素的图片来说，图像上有 10000 个像素点，如果每个像素的值是 **RGBA** 存储的话，那么也就是说每个像素有 4 个通道，每个通道 1 个字节（8 位 = 1 个字节），所以该图片大小大概为 39KB（ $10000 * 1 * 4 / 1024$ ）。

但是在实际项目中，一张图片可能并不需要使用那么多颜色去显示，我们可以通过减少每个像素的调色板来相应缩小图片的大小。

了解了如何计算图片大小的知识，那么对于如何优化图片，想必大家已经有 2 个思路了：

- 减少像素点
- 减少每个像素点能够显示的颜色

图片加载优化

1. 不用图片。很多时候会使用到很多修饰类图片，其实这类修饰图片完全可以用 CSS 去代替。
2. 对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 CDN 加载，可以计算出适配屏幕

的宽度，然后去请求相应裁剪好的图片。

3. 小图使用 base64 格式

4. 将多个图标文件整合到一张图片中（雪碧图）

5. 选择正确的图片格式：

- 对于能够显示 WebP 格式的浏览器尽量使用 WebP 格式。因为 WebP 格式具有更好的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好
- 小图使用 PNG，其实对于大部分图标这类图片，完全可以使用 SVG 代替
- 照片使用 JPEG

DNS 预解析

DNS 解析也是需要时间的，可以通过预解析的方式来预先获得域名所对应的 IP。

```
<link rel="dns-prefetch" href="//yuchengkai.cn">
```

节流

考虑一个场景，滚动事件中会发起网络请求，但是我们并不希望用户在滚动过程中一直发起请求，而是隔一段时间发起一次，对于这种情况我们就可以使用节流。

理解了节流的用途，我们就来实现下这个函数

```
// func是用户传入需要防抖的函数
// wait是等待时间
const throttle = (func, wait = 50) => {
  // 上一次执行该函数的时间
  let lastTime = 0
  return function(...args) {
    // 当前时间
    let now = +new Date()
    // 将当前时间和上一次执行函数时间对比
    // 如果差值大于设置的等待时间就执行函数
    if (now - lastTime > wait) {
      lastTime = now
      func.apply(this, args)
    }
  }
}

setInterval(
  throttle(() => {
    console.log(1)
  }, 500),
  1
)
```

防抖

考虑一个场景，有一个按钮点击会触发网络请求，但是我们并不希望每次点击都发起网络请求，而是当用户点击按钮一段时间后没有再次点击的情况才去发起网络请求，对于这种情况我们就可以使用防抖。

理解了防抖的用途，我们就来实现下这个函数

```
// func是用户传入需要防抖的函数
// wait是等待时间
const debounce = (func, wait = 50) => {
  // 缓存一个定时器id
  let timer = 0
  // 这里返回的函数是每次用户实际调用的防抖函数
  // 如果已经设定过定时器了就清空上一次的定时器
  // 开始一个新的定时器，延迟执行用户传入的方法
  return function(...args) {
    if (timer) clearTimeout(timer)
    timer = setTimeout(() => {
      func.apply(this, args)
    }, wait)
  }
}
```

预加载

在开发中，可能会遇到这样的情况。有些资源不需要马上用到，但是希望尽早获取，这时候就可以使用预加载。

预加载其实是声明式的 `fetch`，强制浏览器请求资源，并且不会阻塞 `onload` 事件，可以使用以下代码开启预加载

```
<link rel="preload" href="http://example.com">
```

预加载可以一定程度上降低首屏的加载时间，因为可以将一些不影响首屏但重要的文件延后加载，唯一缺点就是兼容性不好。

预渲染

可以通过预渲染将下载的文件预先在后台渲染，可以使用以下代码开启预渲染

```
<link rel="prerender" href="http://example.com">
```

预渲染虽然可以提高页面的加载速度，但是要确保该页面大概率会被用户在之后打开，否则就是白白浪费资源去渲染。

懒执行

懒执行就是将某些逻辑延迟到使用时再计算。该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行。懒执行需要唤醒，一般可以通过定时器或者事件的调用来唤醒。

懒加载

懒加载就是将不关键的资源延后加载。

懒加载的原理就是只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域）内需要加载的东西。对于图片来说，先设置图片标签的 `src` 属性为一张占位图，将真实的图片资源放入一个自定义属性中，当进入自定义区域时，就将自定义属性替换为 `src` 属性，这样图片就会去下载资源，实现了图片懒加载。

懒加载不仅可以用于图片，也可以使用在别的资源上。比如进入可视区域才开始播放视频等等。

CDN

CDN 的原理是尽可能的在各个地方分布机房缓存数据，这样即使我们的根服务器远在国外，在国内的用户也可以通过国内的机房迅速加载资源。

因此，我们可以将静态资源尽量使用 CDN 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 CDN 域名。并且对于 CDN 加载静态资源需要注意 CDN 域名要与主站不同，否则每次请

求都会带上主站的 Cookie，平白消耗流量。

小结

这些碎片化的性能优化点看似很短，但是却能在出现性能问题时简单高效的提高性能，并且好几个点都是面试高频考点，比如节流、防抖。如果你还没有在项目中使用过这些技术，可以尝试着用到项目中，体验下功效。