

目录：

## web前端常见性能优化及vue性能优化

- 一、前言
- 二、性能指标
  - 2.1 常用性能指标
  - 2.2 三大核心性能指标
  - 2.3 测量指标工具
- 三、web性能指标优化
- 四、Vue性能优化
  - 4.1 vue代码层面的优化
  - 4.2 Webpack 层面的优化
- 五、总结

# web前端常见性能优化及vue性能优化

## 一、前言

在web前端开发中，编写界面好看，交互简单的页面固然重要，但页面性能也是非常重要的一方面，抛开用户网速差异的限制，据统计一个网站如果超过5s还不能显示出来，那么糟糕的体验，会使用户将不会再打开这个页面，因此web前端开发中性能优化是十分重要的一环。本文结合工作中遇到的页面性能问题，从性能指标出发，分析页面性能的瓶颈所在，并致力在开发过程中做出相应优化，争取在页面开发过程中加快页面加载速度，尽可能地流畅易用。

## 二、性能指标

性能指标其实就是用户体验指标，是直接影响用户体验的直观因素。下文从指标本身的作用，如何测量，推荐时间和指标优化等作出分析。

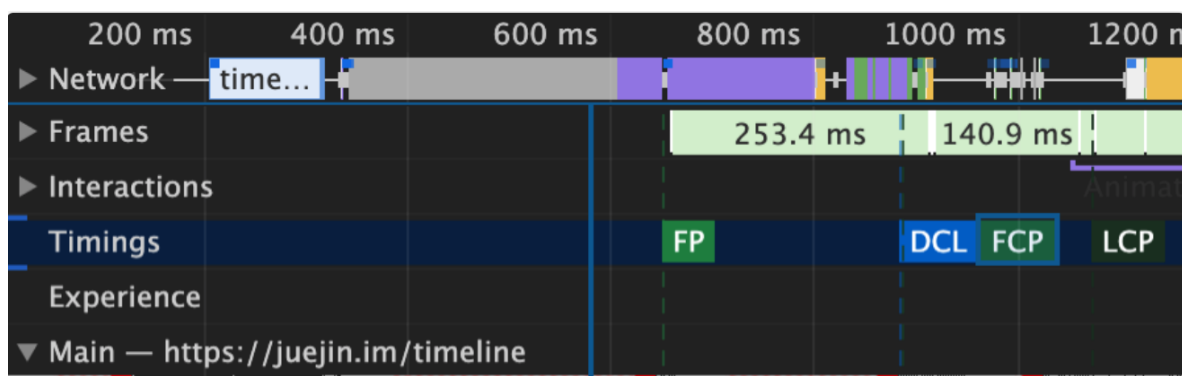
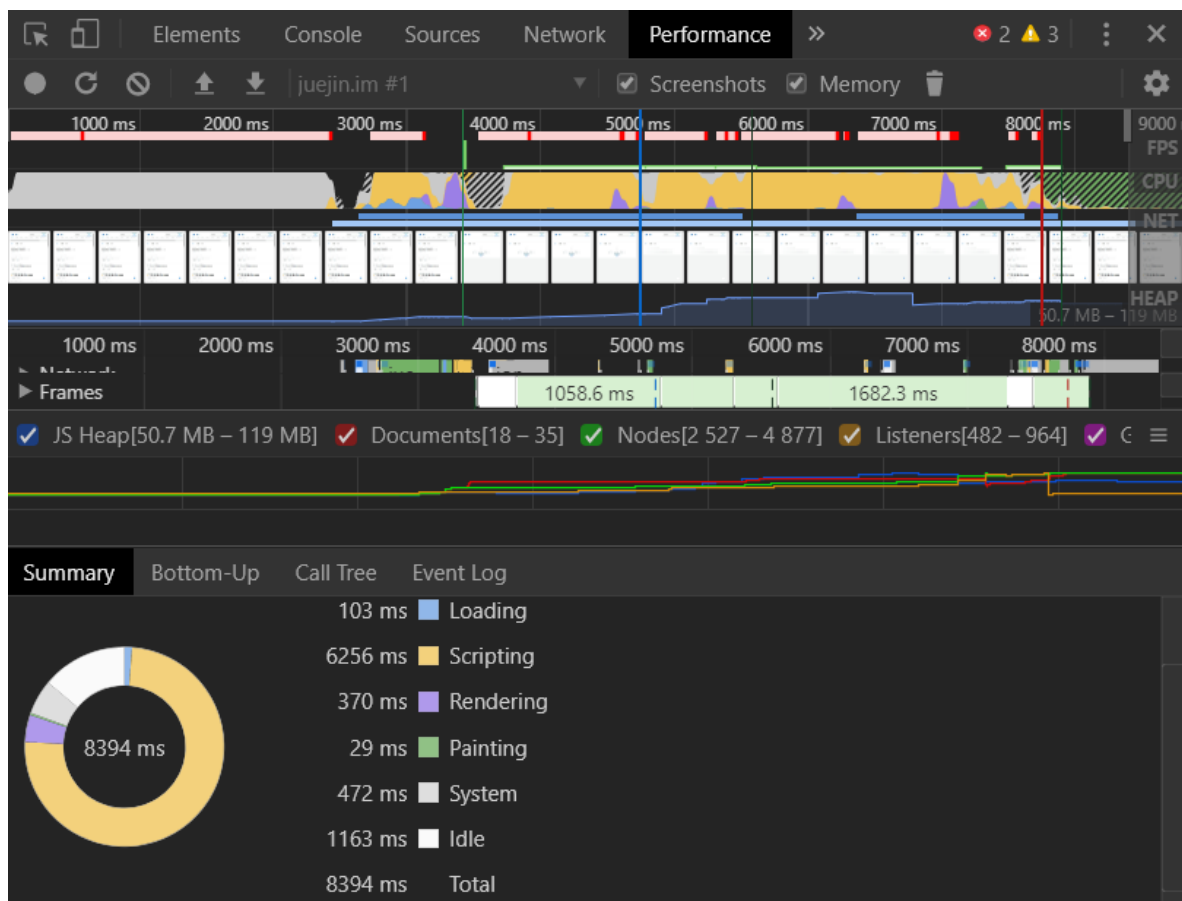
### 2.1 常用性能指标

#### 1. FP&FCP

FP (First Paint) 首次绘制，用于记录页面 第一次绘制像素的时间。

FCP (First Contentful Paint) 首次内容绘制，用于记录页面首次绘制文本、图片、非空白Canvas或者SVG的时间。

FP所用的时间必然大于等于FCP的时间，访问掘金网站为例：



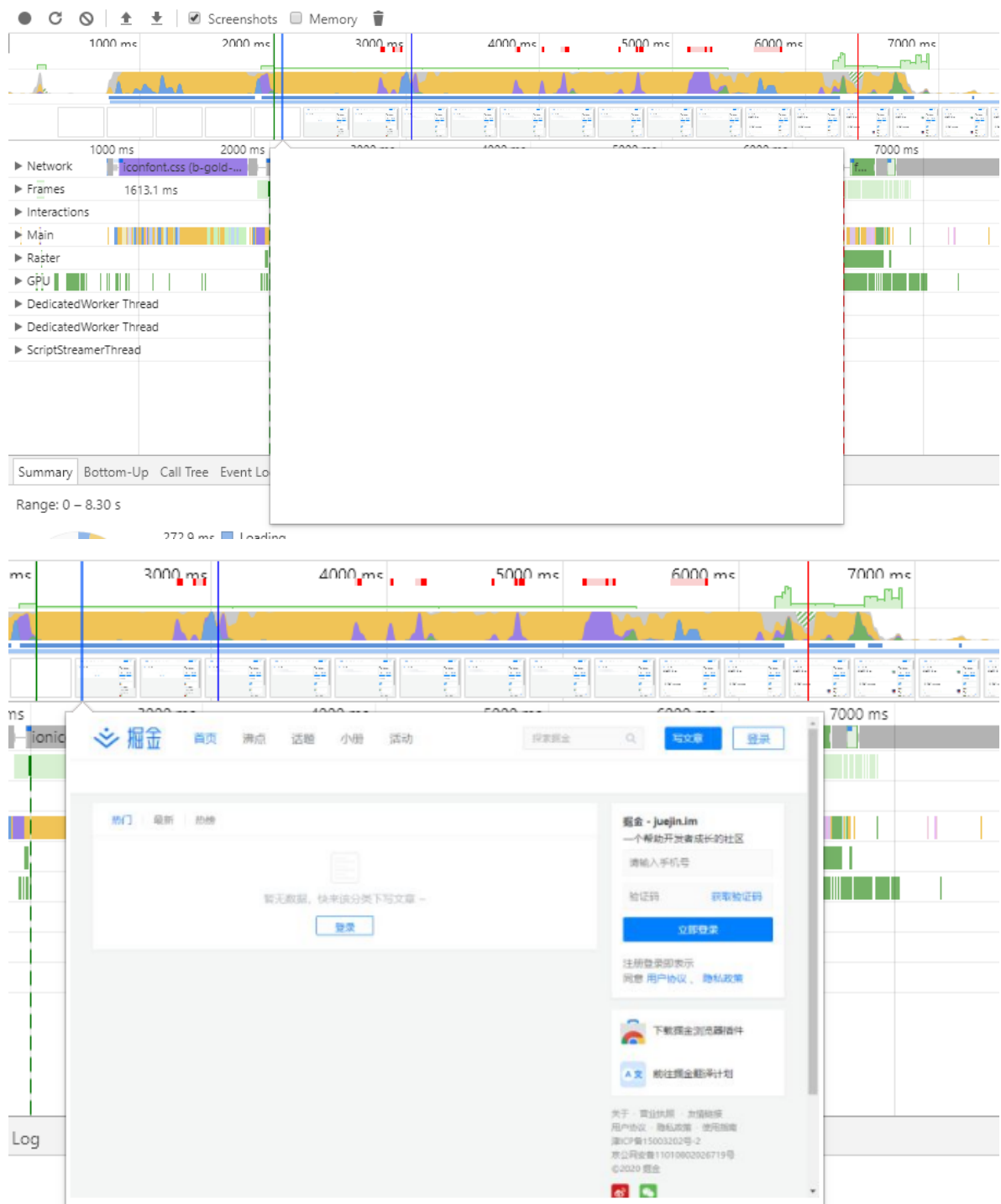
FP 指的是绘制像素，比如说页面的背景色是灰色的，那么在显示灰色背景时就记录下了 FP 指标。但是此时 DOM 内容还没开始绘制，可能需要文件下载、解析等过程，只有当 DOM 内容发生变化才会触发，比如说渲染出了一段文字，此时就会记录下 FCP 指标。因此说我们可以把这两个指标认为是和白屏时间相关的指标，所以肯定是最快越好。

谷歌官方推荐的时间是最好在2s以内m，此时页面体验是最佳的。

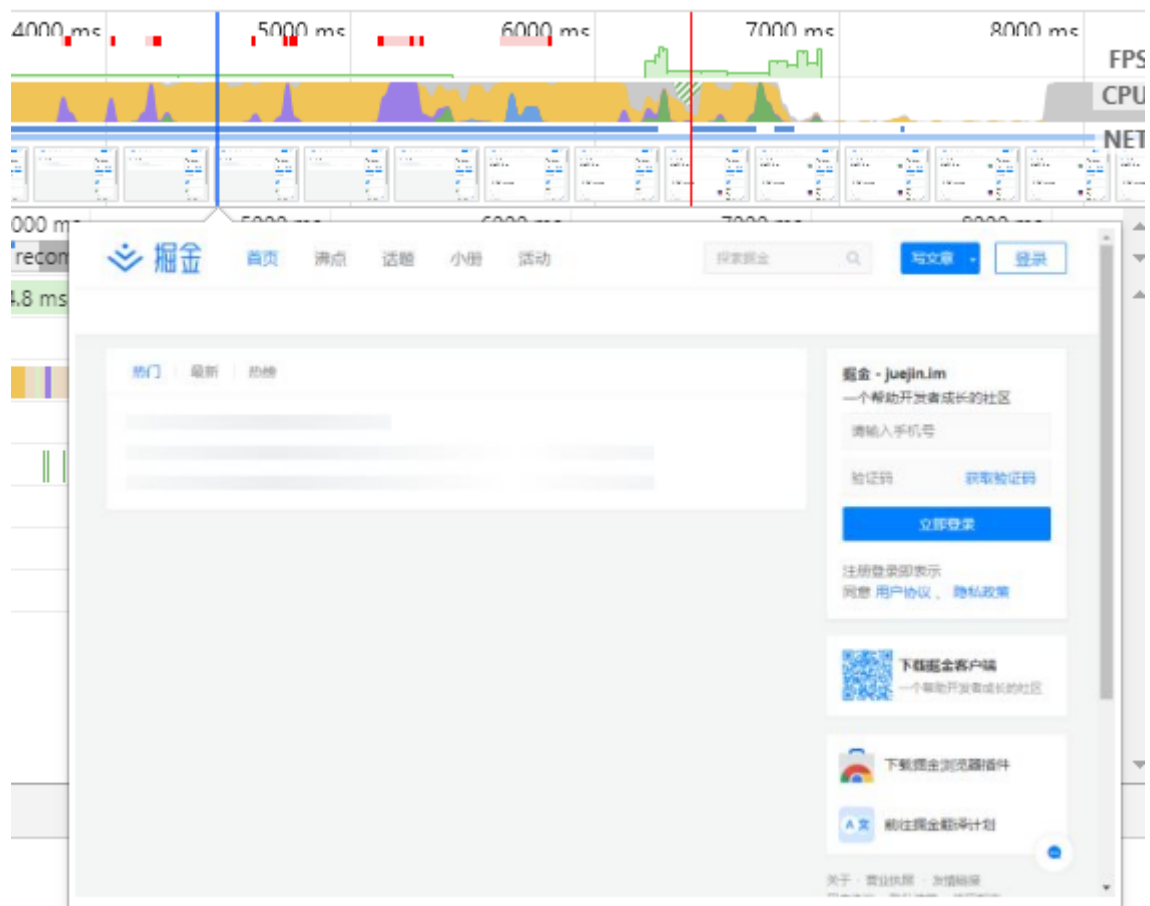
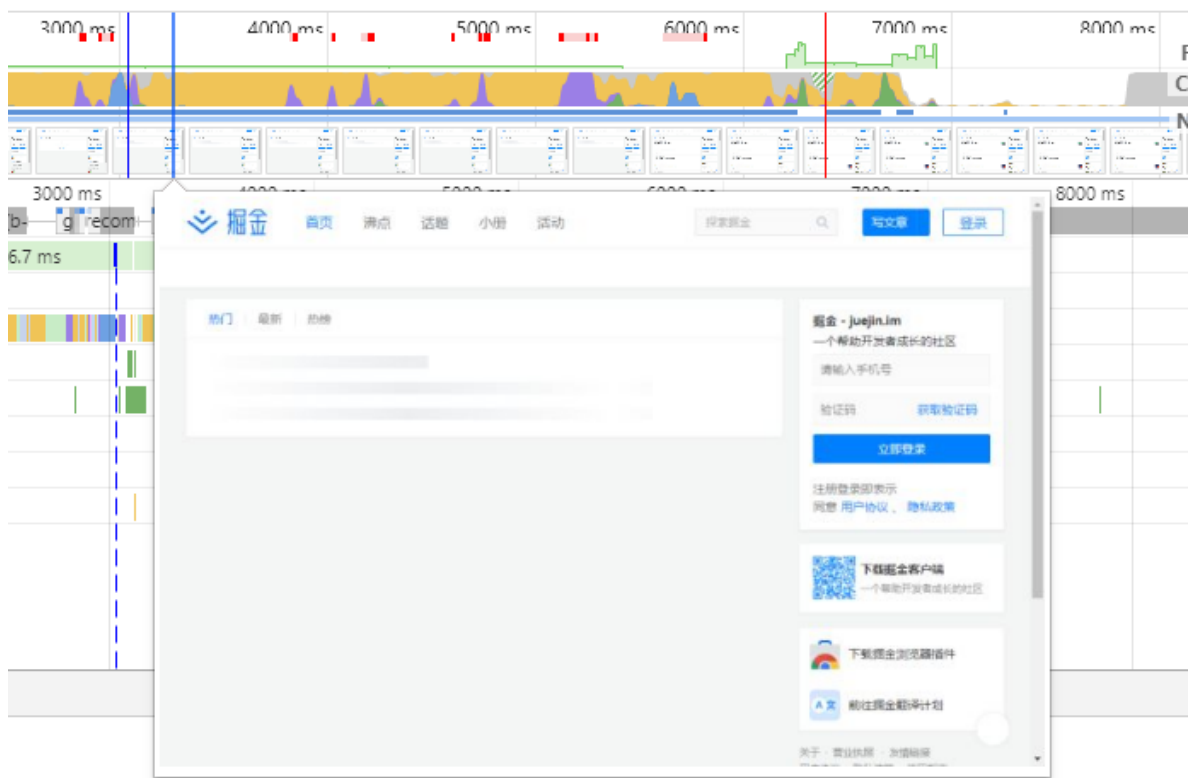
FCP time (in seconds)	Color-coding	FCP score (HTTP Archive percentile)
0-2	Green (fast)	75-100
2-4	Orange (moderate)	50-74
Over 4	Red (slow)	0-49

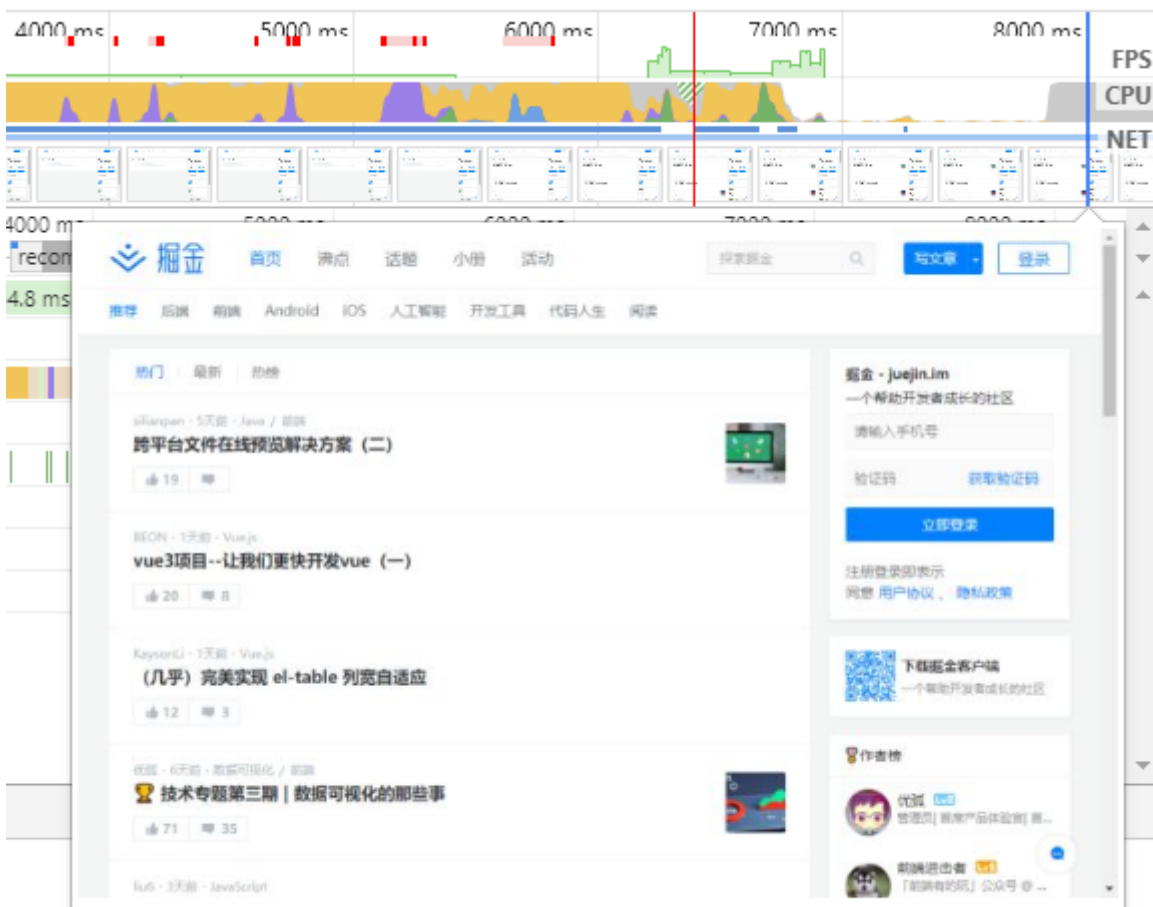
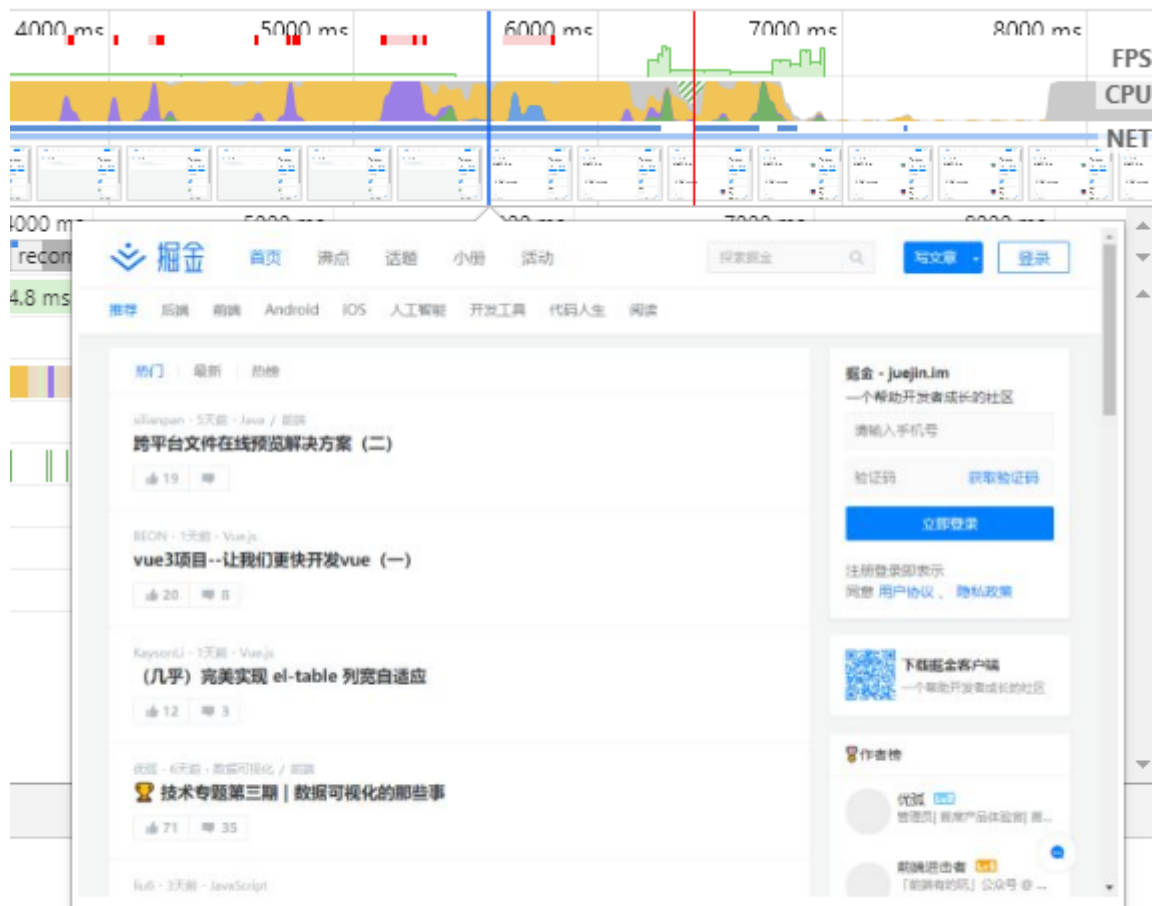
## 2. LCP

LCP (Largest Contentful Paint) 最大内容绘制，用于记录视窗内最大的元素绘制的时间，该时间会随着页面渲染变化而变化，因为页面中的最大元素在渲染过程中可能会发生改变，该指标也会在用户第一次交互后停止记录。指标变化如下图：



ading  
ripting





LCP 其实能比前两个指标更能体现一个页面的性能好坏程度，因为这个指标会在页面加载过程中持续更新。例如：当页面出现骨架屏或者 Loading 动画时 FCP 其实已经被记录下来了，但是此时用户希望看到的内容其实并未呈现，我们更想知道的是页面主要的内容是何时呈现出来的。

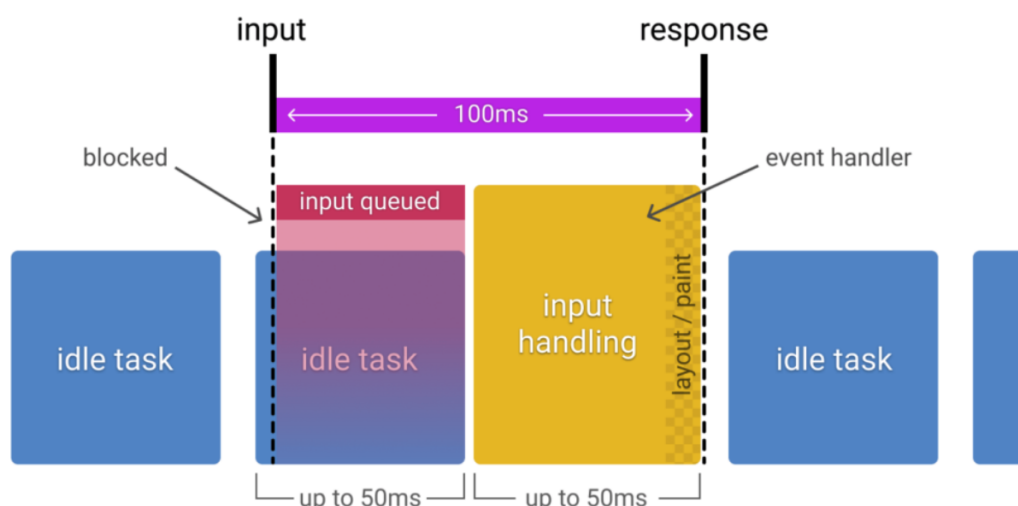
谷歌官方推荐的时间区间是在2.5s以内较为优秀。



### 3. TTI

TTI (Time to Interactive) 首次可交互时间，这个指标计算过程略微复杂，它需要满足以下几个条件：

- 从 FCP 指标后开始计算
  - 持续 5 秒内无长任务（执行时间超过 50 ms）且无两个以上正在进行中的 GET 请求
  - 往前回溯至 5 秒前的最后一个长任务结束的时间
- Google 提出了一个 RAIL 模型：对于用户交互（比如点击事件），推荐的响应时间是 100ms 以内。那么为了达成这个目标，推荐在空闲时间里执行任务不超过 50ms（W3C 的标准规定），这样能在用户无感知的情况下响应用户的交互，否则就会造成延迟感。



How idle tasks affect input response budget.

这是一个很重要的用户体验指标，代表着页面何时真正进入可用的状态。光内容渲染的快是不够的，还要能迅速响应用户的交互。想必大家应该体验过某些网站，虽然内容渲染出来了，但是响应交互很卡顿，只能过一会才能流畅交互的情况。

### 4. FID

FID (First Input Delay) 首次输入延迟，记录在 FCP 和 TTI 之间用户首次与页面交互时响应的延迟。这个指标其实就是看用户交互事件触发到页面响应中间耗时多少，如果其中有长任务发生的话那么势必会造成响应时间变长。

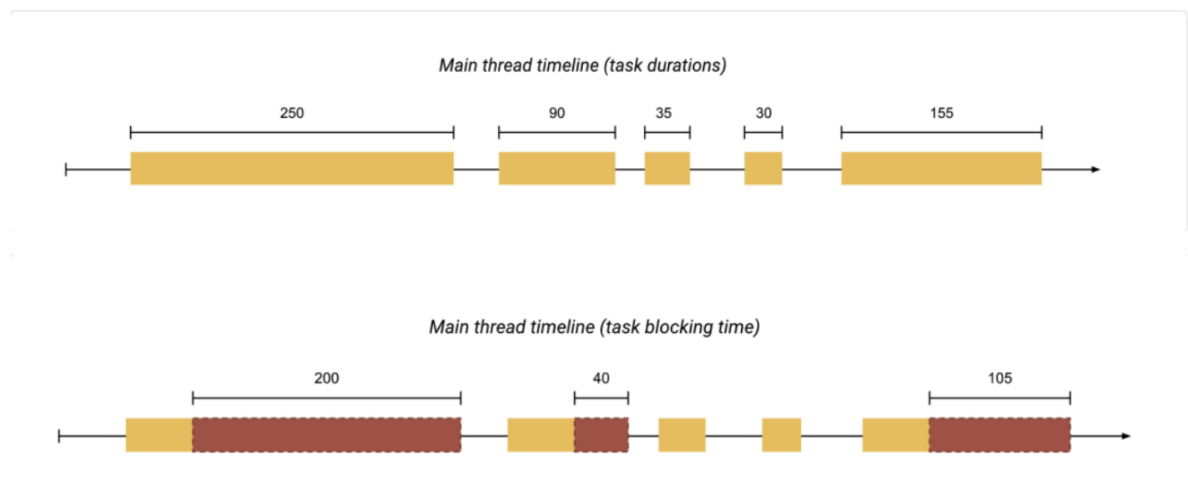
谷歌官方推荐的响应交互时间在100ms以内为最佳。



## 5. TBT

TBT (Total Blocking Time) 阻塞总时间，记录在 FCP 到 TTI 之间所有长任务的阻塞时间总和。

假如说在 FCP 到 TTI 之间页面总共执行了以下长任务（执行时间大于 50ms）及短任务（执行时间低于 50ms），那么每个长任务的阻塞时间就等于它所执行的总时间减去 50ms。



所以对于上图的情况来说，TBT 总共等于 345ms。

这个指标的高低其实也影响了 TTI 的高低，或者说和长任务相关的几个指标都有关联性。

## 6. CLS

CLS (Cumulative Layout Shift) 累计位移偏移，记录了页面上非预期的位移波动。页面渲染过程中突然插入一张巨大的图片或者说点击了某个按钮突然动态插入了一块内容等等相当影响用户体验的网站。这个指标就是为这种情况而生的，计算方式为：位移影响的面积 位移距离。





silianpan Lv3

2020年08月24日 阅读 950

关注

## 跨平台文件在线预览解决方案（二）

### 一、前言

上一篇文章《跨平台（uni-app）文件在线预览解决方案》，文中着重介绍了采用pdf.js在各个平台（H5和APP）进行PDF文件在线预览，关于Office文件（Word、PPT、Excel）文件预览是搭建OpenOffice服务，利用服务转换PDF文件，然后再进行预览。

### 二、需求不断扩大

随着公司业务的发展，产品对Office文件的处理需求不断扩大，包括文档的转换、预览、编辑、加工等，以上的文件在线预览方案已经无法满足产品的需求，体现在如下几个方面：



silianpan Lv3

2020年08月24日 阅读 952

关注



## 跨平台文件在线预览解决方案（二）



以上图为例，由于插入图片，文字移动了 25% 的屏幕高度距离（位移距离），位移前后影响了 75% 的屏幕高度面积（位移影响的面积），那么 CLS 为  $0.25 \times 0.75 = 0.1875$ 。





CLS 推荐值为低于 0.1，越低说明页面跳来跳去的情况就越少，用户体验越好。毕竟很少有人喜欢阅读或者交互过程中网页突然动态插入 DOM 的情况，比如说插入广告~

## 2.2 三大核心性能指标

- LCP
- FID
- CLS

LCP 代表了页面的速度指标，虽然还存在其他的一些体现速度的指标，但是LCP 能体现的东西更多一些。一是指标实时更新，数据更精确，二是代表着页面最大元素的渲染时间，通常来说页面中最大元素的快速载入能让用户感觉性能还挺好。

FID 代表了页面的交互体验指标，毕竟没有一个用户希望触发交互以后页面的反馈很迟缓，交互响应的快会让用户觉得网页挺流畅。

CLS 代表了页面的稳定指标，尤其在手机上这个指标更为重要。因为手机屏幕挺小，CLS 值一大的话会让用户觉得页面体验做的很差。

## 2.3 测量指标工具

1. Lighthouse 通过安装 Lighthouse 插件来获取FCP，LCP等指标
2. web-vitals-extension 通过安装 web-vitals-extension 插件来获取三大核心指标
3. web-vitals 库 通过安装 web-vitals 包来获取CLS，FID，LCP，FCP，TTFB等指标，如下使用：

```
1 import {getCLS, getFID, getLCP} from 'web-vitals';
2 getCLS(console.log);
3 getFID(console.log);
4 getLCP(console.log);
```

4. Chrome DevTools 打开 Performance 即可快速获取如各种指标

## 三、web性能指标优化

1. 页面内容优化（优化 FP、FCP、LCP 指标）

Web 前端 80% 的响应时间花在图片、样式、脚本等资源下载上。浏览器对每个域名的连接数是有限制的，减少请求次数是缩短响应时间的关键。

- 压缩前端文件、使用 Tree-shaking 删除无用代码
- 服务端配置 Gzip 进一步再压缩文件体积
- 资源按需加载，懒加载，预加载等（非首屏所使用的数据，样式，脚本和图片等，用户交互才会出现的内容等都可以懒加载，根据用户行为预判用户去向，预加载相关资源，页面即将上线新版前预先加载新版内容，提前加载一些新版的资源缓存到客户端，以便新版正式上线后更快的载入可以预加载）
- 通过 Chrome DevTools 分析首屏不需要使用的 CSS 文件，以此来精简 CSS
- 合并内联关键的 CSS 代码，合并 JavaScript、CSS 等文件
- 图片优化，包括：用 CSS 代替图标图片、裁剪适配屏幕的图片大小、小图使用 base64 或者 PNG 格式、支持 WebP 图片就尽量使用 WebP 图片、渐进式加载图片，懒加载图片，内嵌 SVG，用 CSS Sprite，Icon Font 和 SVG Sprite 取代图片等
- 减少 http 请求数量
- 内容分片，将请求划分到不同的域名上
- 缓存文件，对首屏数据做离线缓存，添加 Expires 或 Cache-Control 响应头，配置 Etag

## 2. 减少 DNS 查询

输入 URL 以后，浏览器首先要查询域名 (hostname) 对应服务器的 IP 地址，一般需要耗费 20-120 毫秒时间。DNS 查询完成之前，浏览器无法从服务器下载任何数据，浏览器一般会限制每个域的并行线程（一般为 6 个，甚至更少），使用不同的域名可以最大化下载线程，但注意保持在 2-4 个域名内，以避免 DNS 查询损耗。

- IE 缓存 30 分钟，可以通过注册表中 DnsCacheTimeout 项设置；
- Firefox 缓存 1 分钟，通过 network.dnsCacheExpiration 配置；
- 首次访问、没有相应的 DNS 缓存时，域名越多，查询时间越长。所以应尽量减少域名数量。但基于并行下载考虑，把资源分布到 2 个域名上（最多不超过 4 个）。这是减少 DNS 查询同时保证并行下载的折衷方案。
- 使用 CDN 加载资源及 dns-prefetch 预解析 DNS 的 IP 地址
- 对资源使用 preconnect，以便预先进行 IP 解析、TCP 握手、TLS 握手

## 3. 网络优化（优化 FP、FCP、LCP 指标）

网络优化可以利用后端或运维去做，例如优化 sql 查询减少请求时间，升级至最新的网络协议通常能让你网站加载的更快，使用 HTTP2.0 协议、TLS 1.3 协议或者直接拥抱 QUIC 协议等

## 4. 优化耗时任务（优化 TTI、FID、TBT 指标）

使用 Web Worker 将耗时任务丢到子线程中，这样能让主线程在不卡顿的情况下处理 JS 任务 调度任务 + 时间切片，这块技术在 React 16 中有使用到。简单来说就是给不同的任务分配优先级，然后将一段长任务切片，这样能尽量保证任务只在浏览器的空闲时间中执行而不卡顿主线程

## 5. 禁止动态插入内容（优化 CLS 指标）

使用骨架屏给用户一个预期的内容框架，突兀的显示内容体验不会很好 图片切勿不设置长宽，而是使用占位图给用户一个图片位置的预期 不要在现有的内容中间插入内容，起码给出一个预留位置用来占位

## 6. 避免重定向

HTTP 重定向通过 301/302 状态码实现。

客户端收到服务器的重定向响应后，会根据响应头中 Location 的地址再次发送请求。重定向会影响用户体验，尤其是多次重定向时，用户在一段时间内看不到任何内容，只看到浏览器进度条一直在刷新。

有时重定向无法避免，在糟糕也比抛出 404 好。虽然通过 HTML meta refresh 和 JavaScript 也能实现，但首选 HTTP 3xx 跳转，以保证浏览器「后退」功能正常工作（也利于 SEO）。

- 最浪费的重定向经常发生、而且很容易被忽略：URL 末尾应该添加 / 但未添加。比如，访问 <http://astrology.yahoo.com/astrology> 将被 301 重定向到 <http://astrology.yahoo.com/astrology/>（注意末尾的 /）。如果使用 Apache，可以通过 Alias 或 mod\_rewrite 或 DirectorySlash 解决这个问题。
- 网站域名变更：CNAME 结合 Alias 或 mod\_rewrite 或者其他服务器类似功能实现跳转。

## 7. 较少DOM数量

复杂的页面不仅下载的字节更多，JavaScript DOM 操作也更慢。例如，同是添加一个事件处理器，500 个元素和 5000 个元素的页面速度上会有很大区别

以下几个角度考虑移除不必要的标记：

- 是否还在使用表格布局？
- 塞进去更多的  
仅为了处理布局问题？也许有更好、更语义化的标记。
- 能通过伪元素实现的功能，就没必要添加额外元素，如清除浮动。

## 8. 减少iframe的使用

使用 iframe 可以在页面中嵌入 HTML 文档，但有利有弊。

`<iframe>` 的优点：

- 可以用来加载速度较慢的第三方资源，如广告、徽章；
- 可用作安全沙箱；
- 可以并行下载脚本。

`<iframe>` 的缺点：

- 加载代价昂贵，即使是空的页面；

- 阻塞页面 load 事件触发；Iframe 完全加载以后，父页面才会触发 load 事件。Safari、Chrome 中通过 JavaScript 动态设置 iframe src 可以避免这个问题。
- 缺乏语义。

## 9. Ajax 请求使用 GET 方法

浏览器执行 XMLHttpRequest POST 请求时分成两步，先发送 Header，再发送数据。而 GET 只使用一个 TCP 数据包发送数据，所以首选 GET 方法。

根据 HTTP 规范，GET 用于获取数据，POST 则用于向服务器发送数据，所以 Ajax 请求数据时使用 GET 更符合规范（GET 和 POST 对比）。

IE 中最大 URL 长度为 2K，如果超出 2K，则需要考虑使用 POST 方法。

## 10. 避免图片 src 为空

图片src 属性为空字符串，但浏览器仍然会向服务器发起一个 HTTP 请求：

- IE 向页面所在的目录发送请求；
- Safari、Chrome、Firefox 向页面本身发送请求；
- Opera 不执行任何操作。

空 src 产生请求的后果不容小觑：

- 给服务器造成意外的流量负担，尤其时日 PV 较大时；
- 浪费服务器计算资源；
- 可能产生报错。

## 11. 减少 Cookie 大小

Cookie 被用于身份认证、个性化设置等诸多用途。Cookie 通过 HTTP 头在服务器和浏览器间来回传送，减少 Cookie 大小可以降低其对响应速度的影响。

- 去除不必要的 Cookie；
- 尽量压缩 Cookie 大小；
- 注意设置 Cookie 的 domain 级别，如无必要，不要影响到 sub-domain；
- 设置合适的过期时间。

# 四、Vue性能优化

Vue 框架通过数据双向绑定和虚拟 DOM 技术，帮我们处理了前端开发中最脏最累的 DOM 操作部分，我们不再需要去考虑如何操作 DOM 以及如何最高效地操作 DOM；但 Vue 项目中仍然存在项目首屏优化、Webpack 编译配置优化等问题

## 4.1 vue代码层面的优化

### 1. v-if 和 v-show 区分使用场景

**v-if** 是 **真正** 的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建；也是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

**v-show** 就简单得多，不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 的 display 属性进行切换。

所以，v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景

## 2. computed 和 watch 区分使用场景

**computed**：是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值；

**watch**：更多的是「观察」的作用，类似于某些数据的监听回调，每当监听的数据变化时都会执行回调进行后续操作；

**运用场景：**

- 当我们需要进行数值计算，并且依赖于其它数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算；
- 当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，使用 watch 选项允许我们执行异步操作（访问一个 API），限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的

## 3. v-for 遍历必须为 item 添加 key，且避免同时使用 v-if

**(1) v-for 遍历必须为 item 添加 key**在列表数据进行遍历渲染时，需要为每一项 item 设置唯一 key 值，方便 Vue.js 内部机制精准找到该条列表数据。当 state 更新时，新的状态值和旧的状态值对比，较快地定位到 diff。

**(2) v-for 遍历避免同时使用 v-if**，v-for 比 v-if 优先级高，如果每一次都需要遍历整个数组，将会影响速度，尤其是当之需要渲染很小一部分的时候，必要情况下应该替换成 computed 属性。**推荐**

```

<template>
  <ul>
    <li v-for="user in activeUsers" :key="user.id"> {{ user.name }} </li>
  </ul>
</template>
<script>
export default {
  data() {
    return {
      users: [
        { id: "1", name: "前端公虾米", isActive: true },
        { id: "2", name: "干货多", isActive: true },
        { id: "3", name: "有料", isActive: false }
      ]
    };
  },
  computed: {
    activeUsers: function() {
      return this.users.filter(function(user) {
        return user.isActive;
      });
    }
  }
};
</script>

```

不推荐


```

<template>
  <ul>
    <li v-if="user.isActive" v-for="user in users" :key="user.id"> {{ user.name }} </li>
  </ul>
</template>
<script>
export default {
  data() {
    return {
      users: [
        { id: "1", name: "前端公虾米", isActive: true },
        { id: "2", name: "前端", isActive: true },
        { id: "3", name: "JAVA", isActive: false }
      ]
    };
  }
};
</script>

```

#### 4. 长列表性能优化

Vue2.x 会通过 `Object.defineProperty` 对数据进行劫持，来实现视图响应数据的变化，然而有些时候我们的组件就是纯粹的数据展示，不会有任何改变，我们就不需要 Vue 来劫持我们的数据，在大量数据展示的情况下，这能够很明显的减少组件初始化的时间，那如何禁止 Vue 劫持我们的数据呢？可以通过 `Object.freeze` 方法来冻结一个对象，一旦被冻结的对象就再也不能被修改了。



```

<script>
export default {
  data() {
    return {
      users: {}
    };
  },
  async created() {
    const users = await axios.get(/api/users);
    this.users = Object.freeze(users);
  }
};
</script>

```

## 5. 事件的销毁

Vue 组件销毁时，会自动清理它与其它实例的连接，解绑它的全部指令及事件监听器，但是仅限于组件本身的事件。如果在 js 内使用 `addEventListener` 等方式是不会自动销毁的，我们需要在组件销毁时手动移除这些事件的监听，以免造成内存泄露，如：



```

<script>
export default {
  data() {
    return {
      users: {}
    };
  },
  methods: {
    click() {
      console.log("前端公虾米");
    }
  },
  created() {
    addEventListener("click", this.click, false);
  },
  beforeDestroy() {
    removeEventListener("click", this.click, false);
  }
};
</script>

```

## 6. 图片资源懒加载



对于图片过多的页面，为了加速页面加载速度，所以很多时候我们需要将页面内未出现在可视区域内的图片先不做加载，等到滚动到可视区域后再去加载。这样对于页面加载性能上会有很大的提升，也提高了用户体验。我们在项目中使用 Vue 的 vue-lazyload 插件

#### (1) 安装插件

```
1 | import VueLazyload from 'vue-lazyload'
```

#### (2) 在入口文件 main.js 中引入并使用

```
1 | import VueLazyload from 'vue-lazyload'
```

然后再 vue 中直接使用

```
1 | vue.use(VueLazyload)
```

或者添加自定义选项

```
1 | vue.use(VueLazyload, {preLoad: 1.3,error:  
  | 'dist/error.png',loading: 'dist/loading.gif',attempt: 1})
```

(3) 在 vue 文件中将 img 标签的 src 属性直接改为 v-lazy，从而将图片显示方式更改为懒加载显示：

```
1 | <img v-lazy=/static/img/1.png>
```

以上为 vue-lazyload 插件的简单使用

## 7. 路由懒加载

Vue 是单页面应用，可能会有很多的路由引入，这样使用 webpack 打包后的文件很大，当进入首页时，加载的资源过多，页面会出现白屏的情况，不利于用户体验。

如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应的组件，这样就更加高效了。这样会大大提高首屏显示的速度，但是可能其他的页面的速度就会降下来。**路由懒加载：**

```
1 | const Foo = () => import('./Foo.vue')  
2 | const router = new VueRouter({routes: [ { path: '/foo',  
  | component: Foo } ] })
```

## 8. 第三方插件的按需引入

在项目中经常会需要引入第三方插件，如果我们直接引入整个插件，会导致项目的体积太大，我们可以借助 `babel-plugin-component`，然后可以只引入需要的组件，以达到减小项目体积的目的。

(1) 首先, 安装 `babel-plugin-component` :

```
1 | npm install babel-plugin-component -D
```

(2) 然后, 将 `.babelrc` 修改为:

```
1 | {presets: [[es2015, { modules: false }]],plugins: [
  |   [component, {libraryName: element-ui,styleLibraryName:
  |     theme-chalk } ] ]}
```

(3) 在 `main.js` 中引入部分组件:

```
1 | import Vue from 'vue';import { Button, Select } from
  | 'element-ui'; Vue.use(Button) Vue.use(Select)
```

## 4.2 Webpack 层面的优化

### 1. Webpack 对图片进行压缩

在 `vue` 项目中除了可以在 `webpack.base.conf.js` 中 `url-loader` 中设置 `limit` 大小来对图片处理, 对小于 `limit` 的图片转化为 `base64` 格式, 其余的不做操作。所以对有些较大的图片资源, 在请求资源的时候, 加载会很慢, 我们可以用 `image-webpack-loader` 来压缩图片

### 2. 减少 ES6 转为 ES5 的冗余代码

Babel 插件会在将ES6代码转换成 ES5 代码时会注入一些辅助函数, 例如下面的 ES6 代码:

```
1 | class Hellowebpack extends Component{...}
```

这段代码再被转换成能正常运行的 ES5 代码时需要以下两个辅助函数:

```
1 | babel-runtime/helpers/createClass // 用于实现 class 语法babel-
  | runtime/helpers/inherits // 用于实现 extends 语法
```

在默认情况下, Babel 会在每个输出文件中内嵌这些依赖的辅助函数代码, 如果多个源代码文件都依赖这些辅助函数, 那么这些辅助函数的代码将会出现很多次, 造成代码冗余。为了不让这些辅助函数的代码重复出现, 可以在依赖它们时通过 `require('babel-runtime/helpers/createClass')` 的方式导入, 这样就能做到只让它们出现一次。 `babel-plugin-transform-runtime` 插件就是用来实现这个作用的, 将相关辅助函数进行替换成导入语句, 从而减小 babel 编译出来的代码的文件大小。

### 3. 提取公共代码

如果项目中没有去将每个页面的第三方库和公共模块提取出来，则项目会存在以下问题：

- 相同的资源被重复加载，浪费用户的流量和服务器的成本。
- 每个页面需要加载的资源太大，导致网页首屏加载缓慢，影响用户体验。

所以我们需要将多个页面的公共代码抽离成单独的文件，来优化以上问题。

Webpack 内置了专门用于提取多个Chunk 中的公共部分的插件

CommonsChunkPlugin

#### 4. 提取组件的 CSS

当使用单文件组件时，组件内的 CSS 会以 style 标签的方式通过 JavaScript 动态注入。这有一些小小的运行时开销，如果你使用服务端渲染，这会导致一段“无样式内容闪烁 (fouc)”。将所有组件的 CSS 提取到同一个文件可以避免这个问题，也会让 CSS 更好地进行压缩和缓存。查阅这个构建工具各自的文档来了解更多：

- webpack + vue-loader ( vue-cli 的 webpack 模板已经预先配置好)
- Browserify + vueify
- Rollup + rollup-plugin-vue

#### 5. 优化 SourceMap

我们在项目进行打包后，会将开发中的多个文件代码打包到一个文件中，并且经过压缩、去掉多余的空格、babel编译后，最终将编译得到的代码会用于线上环境，那么这样处理后的代码和源代码会有很大的差别，当有 bug 的时候，我们只能定位到压缩处理后的代码位置，无法定位到开发环境中的代码，对于开发来说不好调试定位问题，因此 sourceMap 出现了，它就是为了解决不好调试代码问题的。

**开发环境推荐：cheap-module-eval-source-map 生产环境推荐：cheap-module-source-map**

原因如下：

- **cheap**：源代码中的列信息是没有任何作用，因此我们打包后的文件不希望包含列相关信息，只有行信息能建立打包前后的依赖关系。因此不管是开发环境或生产环境，我们都希望添加 cheap 的基本类型来忽略打包前后的列信息；
- **module**：不管是开发环境还是正式环境，我们都希望能定位到bug的源代码具体的位置，比如说某个 Vue 文件报错了，我们希望能定位到具体的 Vue 文件，因此我们也需要 module 配置；
- **source-map**：source-map 会为每一个打包后的模块生成独立的 sourcemap 文件，因此我们需要增加source-map 属性；
- **eval-source-map**：eval 打包代码的速度非常快，因为它不生成 map 文件，但是可以对 eval 组合使用 eval-source-map 使用会将 map 文件以 DataURL 的形式存在打包后的 js 文件中。在正式环境中不要使用 eval-source-map, 因为它会增加文件的大小，但是在开发环境中，可以试用下，因为他们打包的速度很快。

## 五、总结

---

本文结合前端性能指标来关注前端开发的用户体验问题，从性能指标出发，深入浅出地描述了前端性能瓶颈所在，并结合实际详解了性能优化的方向，极大地避免在web开发中遇到阻塞性能的问题。

本文结合实际开发工作，参看书籍和网上相关资料写成，行文多有欠妥之处。如有错误，还望及时指正，谢谢查看。