性能的计算方式

确认自己需要关注的指标

常见的指标有：

1. 页面总加载时间 load
2. 首屏时间
3. 白屏时间

- 代码 尝试用一个指令, 挂载在重要元素上, 当此元素inserted就上报

各个属性所代表的含义

1. connectStart, connectEnd
   分别代表TCP建立连接和连接成功的时间节点。如果浏览器没有进行TCP连接（比如使用持久化连接webscoket），则两者都等于domainLookupEnd；
2. domComplete
   Html文档完全解析完毕的时间节点；
3. domContentLoadedEventEnd
   代表DOMContentLoaded事件触发的时间节点：页面文档完全加载并解析完毕之后,会触发DOMContentLoaded事件，HTML文档不会等待样式文件,图片文件,子框架页面的加载(load事件可以用来检测HTML页面是否完全加载完毕(fully-loaded))。
4. domContentLoadedEventStart
   代表DOMContentLoaded事件完成的时间节点，此刻用户可以对页面进行操作，也就是jQuery中的domready时间；
5. domInteractive
   代表浏览器解析html文档的状态为interactive时的时间节点。domInteractive并非DOMReady，它早于DOMReady触发，代表html文档解析完毕（即dom tree创建完成）但是内嵌资源（比如外链css、js等）还未加载的时间点；
6. domLoading
   代表浏览器开始解析html文档的时间节点。我们知道IE浏览器下的document有readyState属性，domLoading的值就等于readyState改变为loading的时间节点；
7. domainLookupStart domainLookupEnd
   分别代表DNS查询的开始和结束时间节点。如果浏览器没有进行DNS查询（比如使用了cache），则两者的值都等于fetchStart；
8. fetchStart
   是指在浏览器发起任何请求之前的时间值。在fetchStart和domainLookupStart之间，浏览器会检查当前文档的缓存；

9. loadEventStart, loadEventEnd
   分别代表onload事件触发和结束的时间节点
10. navigationStart
11. redirectStart, redirectEnd
    如果页面是由redirect而来，则redirectStart和redirectEnd分别代表redirect开始和结束的时间节点；
12. requestStart
    代表浏览器发起请求的时间节点，请求的方式可以是请求服务器、缓存、本地资源等；
13. responseStart, responseEnd
    分别代表浏览器收到从服务器端（或缓存、本地资源）响应回的第一个字节和最后一个字节数据的时刻；
14. ecureConnectionStart
    可选。如果页面使用HTTPS，它的值是安全连接握手之前的时刻。如果该属性不可用，则返回undefined。如果该属性可用，但没有使用HTTPS，则返回0；
15. unloadEventStart, unloadEventEnd
    如果前一个文档和请求的文档是同一个域的，则unloadEventStart和unloadEventEnd分别代表浏览器unload前一个文档的开始和结束时间节点。否则两者都等于0；

performance具体计算

代码：performance.ts

性能优化

1. 加速或减少HTTP请求损耗：使用CDN加载公用库，使用强缓存和协商缓存，小图片使用Base64代替，页面内跳转其他域名或请求其他域名的资源时使用浏览器prefetch预解析等；
2. 延迟加载：非重要的库、非首屏图片延迟加载，SPA的组件懒加载等；
3. 减少请求内容的体积：开启服务器Gzip压缩，JS、CSS文件压缩合并，减少cookies大小，SSR直接输出渲染后的HTML等；
4. 浏览器渲染原理：优化关键渲染路径，尽可能减少阻塞渲染的JS、CSS；
5. 优化用户等待体验：白屏使用加载进度条、菊花图、骨架屏代替等；

本地如何查看页面性能?

chrome performance

chrome-extension的请求混杂在页面请求中，难以分析
filter中填写 **-scheme:chrome-extension** 即可过滤掉插件请求

- webpack打包分析

webpack-bundle-analyzer插件，代码：vue.config.js + package.json

如何监控性能变化?

- 日志上报 阿里云演示
- pagespeed https://developers.google.com/speed/pagespeed/insights/?hl=zh-cn
- 数据分析 90分位 50分位 TP50、TP90和TP99等指标常用于系统性能监控场景，指高于50%、90%、99%等百分线的情况。

具体的优化方式

浏览器渲染原理

和渲染息息相关的主要有

1. GUI渲染线程

GUI渲染线程负责渲染浏览器界面HTML元素,当界面需要重绘(Repaint)或由于某种操作引发回流(reflow)时，该线程就会执行。
在Javascript引擎运行脚本期间,GUI渲染线程都是处于挂起状态的，也就是说被冻结了.

2. Js引擎线程

JS为处理页面中用户的交互，以及操作DOM树、CSS样式树来给用户呈现一份动态而丰富的交互体验和服务器逻辑的交互处理。

GUI渲染线程与JS引擎线程互斥的，是由于JavaScript是可操纵DOM的，如果在修改这些元素属性同时渲染界面（即JavaScript线程和UI线程同时运行），那么渲染线程前后获得的元素数据就可能不一致。
当JavaScript引擎执行时GUI线程会被挂起，GUI更新会被保存在一个队列中等到引擎线程空闲时立即被执行。
由于GUI渲染线程与JS执行线程是互斥的关系，当浏览器在执行JS程序的时候，GUI渲染线程会被保存在一个队列中，直到JS程序执行完成，才会接着执行。

因此如果JS执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞的感觉。

3. tips:
   回流(reflow)：当浏览器发现某个部分发生了点变化影响了布局，需要倒回去重新渲染。reflow 会从这个 root frame 开始递归往下，依次计算所有的结点几何尺寸和位置。reflow 几乎是无法避免的。现在界面上流行的一些效果，比如树状目录的折叠、展开（实质上是元素的显示与隐藏）等，都将引起浏览器的 reflow。鼠标滑过、点击……只要这些行为引起了页面上某些元素的占位面积、定位方式、边距等属性的变化，都会引起它内部、周围甚至整个页面的重新渲染。

重绘(repaint)：改变某个元素的背景色、文字颜色、边框颜色等等不影响它周围或内部布局的属性时，屏幕的一部分要重画，但是元素的几何尺寸没有变。

渲染流程主要分为以下4步：

1. 解析HTML生成DOM树 - 渲染引擎首先解析HTML文档，生成DOM树
2. 构建Render树 - 接下来不管是内联式，外联式还是嵌入式引入的CSS样式会被解析生成CSSOM树，根据DOM树与CSSOM树生成另外一棵用于渲染的树-渲染树(Render tree)，
3. 布局Render树 - 然后对渲染树的每个节点进行布局处理，确定其在屏幕上的显示位置
4. 绘制Render树 - 最后遍历渲染树并用UI后端层将每一个节点绘制出来

现代浏览器总是并行加载资源，例如，当 HTML 解析器（HTML Parser）被脚本阻塞时，解析器虽然会停止构建 DOM，但仍会识别该脚本后面的资源，并进行预加载。

同时，由于下面两点：

1. CSS 被视为渲染阻塞资源(包括JS)，这意味着浏览器将不会渲染任何已处理的内容，直至 CSSOM 构建完毕，才会进行下一阶段。
2. JavaScript 被认为是解释器阻塞资源，HTML解析会被JS阻塞，它不仅可以读取和修改 DOM 属性，还可以读取和修改 CSSOM 属性。

所以存在阻塞的 CSS 资源时，浏览器会延迟 JavaScript 的执行和 DOM 构建。另外：

1. 当浏览器遇到一个 script 标记时，DOM 构建将暂停，直至脚本完成执行。
2. JavaScript 可以查询和修改 DOM 与 CSSOM。
3. CSSOM 构建时，JavaScript 执行将暂停，直至 CSSOM 就绪。

css资源和js资源的放置位置

所以，script 标签的位置很重要。实际使用时，可以遵循下面两个原则：

1. CSS 优先：引入顺序上，CSS 资源先于 JavaScript 资源。
2. JavaScript 应尽量少影响 DOM 的构建。

js的异步执行

defer async

都是异步加载js资源, 但是区别是async加载完资源后会立即开始执行, 而defer会在整个document解析完成后执行

其他

一、图片资源优化

1. 懒加载

- 图片懒加载

tips: 如何判断元素是否在可视区域内?

原理：默认给图片设置一个兜底图, 监听页面滚动, 判断图片进入可视区域内后, 则给图片设置真实的

src

代码：懒加载.html

2. 预加载图片等资源

```
const img = new Image();
img.src= 'xxxxxx';
```

```
<img src="http://pic26.nipic.com/20121213/6168183 0044449030002.jpg" style="display:none"/>
```

3. webp, 渐进式加载

https://help.aliyun.com/document_detail/171050.html?
spm=5176.10695662.1996646101.searchclickresult.2b9b75d6NiTHLQ
https://help.aliyun.com/document_detail/44704.html?
spm=a2c4g.11186623.6.1428.4acb1ecfHEUVE6

二、其他静态资源加载优化

1. quickLink

https://github.com/GoogleChromeLabs/quicklink

2. prefetch

```
link(rel="dns-prefetch", href="//www.baidu.com")
link(rel="preconnect", href="//www.baidu.com")
link(rel="preload", as="script", href=xxxxx)
link(rel="preload", as="image", href=xxxxxx)
```

3. 静态资源压缩

gzip, Brotli https://help.aliyun.com/document_detail/27127.html?
spm=a2c4g.11186623.6.645.70471769PeSCe6

4. webpack

打包优化 https://webpack.docschina.org/configuration/optimization/

5. 骨架图

代码：index.html

5. 服务端渲染

同构渲染 - next.js nuxt.js
pug模板渲染

6. 动态polyfill

为了兼容低版本浏览器, 我们一般会引入polyfill。但是@babel/polyfill这个包非常大, gzip之后都有27.7k, 非常影响体积.

https://polyfill.io/v3/api/

阿里云cdn: https://polyfill.alicdn.com/polyfill.min.js?features=es6,es7,es2017,es2018&flags=gated

```
<script src="https://polyfill.io/v3/polyfill.js?features=es5,es6,es7,es2017,es2018&flags=gated&callback=in



function invokeMain() {
    var scriptElement = document.createElement('script');
    scriptElement.type = 'text/javascript';
    scriptElement.async = true;
    scriptElement.src = 'xxxxxxxxx';
    document.body.appendChild(scriptElement);
}
```

7. 不重要的资源延迟加载

可以放到window.addEventListener('load', () => {})

8. 节流 防抖

代码 - 节流 防抖

节流函数：当持续触发事件时，保证一定时间段内只调用一次事件处理函数。
防抖函数：当持续触发事件时，一定时间段内没有再触发事件，事件处理函数才会执行一次，如果设定的时间到来之前，又一次触发了事件，就重新开始延时。

函数节流不管事件触发有多频繁，都会保证在规定时间内一定会执行一次真正的事件处理函数
函数防抖只是在最后一次事件后才触发一次函数。

比如在页面的无限加载场景下，我们需要用户在滚动页面时，每隔一段时间发一次 Ajax 请求，而不是在用户停下滚动页面操作时才去请求数据。这样的场景，就适合用节流技术来实现。
而比如搜索框, 输入结束后才发起请求, 就适合用防抖