网站性能数据采集

网站性能数据采集分为**FP**、**LCP**、**TTFB**、**FID**、**CLS**，他们的具体含义如下所示：

- `FP` 首次绘制，即浏览器首次将像素渲染到屏幕上的时间点
- `LCP` 最大内容渲染，即页面中最大元素（图片、视频等） 在是口内完全渲染的时间点
- `TTFB` 首字节时间，即从浏览器发出请求到接收到第一个字节响应的时间。
- `FID` 首次输入延迟，即用户首次与页面交互（点击链接、按钮等）到浏览器响应该事件的延迟时间
- `CLS` 累积布局位移，即页面中所有元素在加载过程中发生意外移动的总和

通过监测这些指标，我们可以了解到当前网站在各个方面的性能表现，排查出需要改进的地方。

网站个性化指标

`Long task` 是一种网站性能指标，用于度量在主线程上执行的长时间任务的数量和持续时间。当浏览器执行长时间任务时，用户可能会遇到页面冻结或卡顿等问题，从而对用户体验造成负面影响。因此，通过监测长任务，可以帮助开发人员识别并优化导致页面性能下降的瓶颈，从而提高网站的响应速度和用户体验。

数据上报

在网站中插入一个像素大小为1*1的透明图片，将图片的url设置成包含需要上报的数据参数，来完成数据上报的目的。当浏览器加载该图片时，会向服务器发送一个HTTP请求，该请求的URL中携带了需要上报的数据，因此服务器可以接收到这些数据并进行相应的处理。由于该方法不会对页面的渲染和用户体验造成影响，因此被广泛用于网站的数据统计、监控和分析等方面。

除了这种方式外，还可以使用fetch请求进行上报，采用这种方式的话坏处就是如果上报的域名跟当前页面不是同域的，会出现跨域问题，需要在服务端设置响应头，优点就是数据的大小不用受浏览器URL的长度限制。

无sourcemap，如何在性能监控平台做到代码还原

1. 根据前端错误日志定位到出错的代码文件和行数；
2. 通过版本管理工具（如Git）找到对应版本的代码，并下载相应的代码文件；
3. 使用第三方工具（如Chrome DevTools或Firefox Developer Edition）进行代码调试，以还原出代码执行时的运行环境和调用堆栈信息；
4. 分析代码运行过程中的性能数据（如CPU占用率、内存占用量等），结合代码逻辑和运行情况，找出可能影响性能的问题。

按照 5W1H 法则来分析前端异常，需要知道以下信息

- What，发生了什么错误：JS错误、异步错误、资源加载、接口错误等
- When，出现的时间段，如时间戳
- Who，影响了多少用户，包括报错事件数、IP
- Where，出现的页面是哪些，包括页面、对应的设备信息
- Why，错误的原因是为什么，包括错误堆栈、行列、SourceMap、异常录屏
- How，如何定位还原问题，如何异常报警，避免类似的错误发生

性能数据采集

以Spa页面来说，页面的加载过程大致是这样的：
dns查询、建立tcp连接、发送http请求、返回html文档、html文档解析等阶段

最初，可以通过 window.performance.timing 来获取加载过程模型中各个阶段的耗时数据。

通过 PerformanceObserver 来获取。旧的 api，返回的是一个 UNIX 类型的绝对时间，和用户的系统时间相关，分析的时候需要再次计算。而新的 api，返回的是一个相对时间，可以直接用来分析。

现在 chrome 开发团队提供了 web-vitals 库，方便来计算各性能数据（注意：web-vitals 不支持safari浏览器）

用户行为数据采集

用户行为包括：页面路由变化、鼠标点击、资源加载、接口调用、代码报错等行为

获取页面内存信息

通过 `performance.memory` 可以显示此刻内存占用情况，它是一个动态值，其中：

- jsHeapSizeLimit 该属性代表的含义是：内存大小的限制。
- totalJSHeapSize 表示总内存的大小。
- usedJSHeapSize 表示可使用的内存的大小。
  通常，usedJSHeapSize 不能大于 totalJSHeapSize，如果大于，有可能出现了内存泄漏。

首屏加载时间

首屏加载时间和首页加载时间不一样，首屏指的是屏幕内的dom渲染完成的时间。

计算首屏加载时间流程

- 利用MutationObserver监听document对象，每当dom变化时触发该事件
- 判断监听的dom是否在首屏内，如果在首屏内，将该dom放到指定的数组中，记录下当前dom变化的时间点
- 在MutationObserver的callback函数中，通过防抖函数，监听document.readyState状态的变化
- 当document.readyState === 'complete'，停止定时器和 取消对document的监听
- 遍历存放dom的数组，找出最后变化节点的时间，用该时间点减去performance.timing.navigationStart 得出首屏的加载时间

用户行为收集

用一个栈来存储用户行为，当长度超过限制时，最早的一条数据会被覆盖掉，在上报错误时，对应的用户行为会添加到该错误信息中。

何时上报录屏数据

- window上设置 hasError、recordScreenId 变量，hasError用来判断某段时间代码是否报错；recordScreenId 用来记录此次录屏的id
- 当页面发生错误需要上报时，先判断是否开启了录屏，如果开启了，将 hasError 设为 true，同时将 window 上的 recordScreenId 存储到此次上报信息的 data 中
- rrweb 设置 10s/次 录制快照的频率，每次重置录屏时，判断 hasError 是否为 true（即这段时间内是否发生报错），如果有发生错误，将这次的录屏信息上报，并重置录屏信息和 recordScreenId，作为下次录屏使用
- 后台报错列表，从本次报错报的data中取出 recordScreenId 来播放录屏

source-map 的还原流程

- 从服务器获取指定.map 的文件内容
- new 一个 SourceMapConsumer 的实例，表示一个已解析的源映射，给它一个文件位置来查询有关原始文件位置的信息
- 输入报错发生的行和列，可以得到源码对应原始文件名、行和列信息
- 从源文件的 sourcesContent 字段中，获取对应的源码信息

性能分析与优化

好比去医院看病一样，得了什么病，通过检测化验后才知道。网站也是一样，需要借助性能分析工具来检测。

Lighthouse工具

Lighthouse是 Chrome 自带的性能分析工具，它能够生成一个有关页面性能的报告
通过报告我们可以知道需要采取哪些措施，来改进应用的性能和体验。

并且 Lighthouse 可以对页面多方面的效果指标进行评测，并给出最佳实践的建议，以帮助开发者改进网站的质量。

Lighthouse拿到页面的“优化”报告

通过 Lighthouse 拿到网站的整体分析报告，通过报告来诊断“病情”，重点关注Performance性能评分

性能评分的分值区间是 0 到 100，如果出现 0 分，通常是在运行 Lighthouse 时发生了错误，满分 100 分代表了网站已经达到了 98 分位值的数据，而 50 分则对应 75 分位值的数据。

Lighthouse 给出 Opportunities 优化建议

Lighthouse 会针对当前网站，给出一些Opportunities优化建议
Opportunities 指的是优化机会，它提供了详细的建议和文档，来解释低分的原因，帮助我们具体进行实现和改进

Web-vitals 官方标准

web-vitals是 Google 给出的定义是 一个良好网站的基本指标
过去要衡量一个网站的好坏，需要使用的指标太多了，现在我们可以将重点聚焦于 Web Vitals 指标的表现即可

我们将 Lighthouse 中 Performance 列出的指标表现，与官方指标标准做对比，可以发现页面哪些指标超出了范围。

Performance 工具

通过 Lighthouse 我们知道了页面整体的性能得分，但是页面打开慢或者卡顿的瓶颈在哪里?
具体是加载资源慢、dom渲染慢、还是js执行慢呢?
chrome 浏览器提供的performance是常用来查看网页性能的工具，通过该工具，我们可以知道页面在浏览器运行时的性能表现

Performance 各区域功能介绍
1）FPS
FPS(Frames Per Second)，表示每秒传输帧数，是用来分析页面是否卡顿的一个主要性能指标
如下图所示，绿色的长条越高，说明FPS越高，用户体验越好
如果发现了一个红色的长条，那么就说明这些帧存在严重问题，可能会造成页面卡顿

2）NET
NET 记录资源的等待、下载、执行时间，每条彩色横杠表示一种资源
横杠越长，检索资源所需的时间越长。每个横杠的浅色部分表示等待时间（从请求资源到第一个字节下载完成的时间）
Network 的颜色说明：白色表示等待的颜色、浅黄色表示请求的时间、深黄色表示下载的时间
在这里，我们可以看到所有资源的加载过程，有两个地方重点关注：
1）资源等待的时间是否过长（标准 ≤100ms）
2）资源文件体积是否过大，造成加载很慢（就要考虑如何拆分该资源）

Performance Main 性能瓶颈的突破口

Main 表示主线程，主要负责

- Javascript 的计算与执行
- CSS 样式计算
- Layout 布局计算
- 将页面元素绘制成位图（paint），也就是光栅化（Raster）

展开 Main,可以发现很多红色三角（long task），这些执行时间超过 50ms就属于长任务，会造成页面卡顿，严重时会造成页面卡死

展开其中一个红色三角，Devtools 在Summary面板里展示了更多关于这个事件的信息，在在 summary 面板里点击app.js链接，Devtools 可以跳转到需要优化的代码处

性能监控

项目发布生产后，用户使用时的性能如何，页面整体的打开速度是多少、白屏时间多少，FP、FCP、LCP、FID、CLS 等指标，要设置多大的阀值呢，才能满足TP50、TP90、TP99的要求呢?

性能指标的计算

方式一：通过 web-vitals 官方库进行计算

方式二：通过performance api进行计算

- 打开任意网页，在控制台中输入 performance 回车，可以看到一系列的参数，
- performance.timing记录了页面各个关键时间点

计算各资源的 TTFB

要计算TTFB（Time To First Byte，即从客户端发出请求到服务器返回第一个字节的时间），可以使用 PerformanceResourceTiming 对象中的两个属性：requestStart 和 responseStart。

- requestStart 属性表示浏览器向服务器发送请求的时间戳，单位是毫秒。
- responseStart 属性表示浏览器接收到服务器响应的第一个字节的时间戳，单位也是毫秒。

因此，TTFB 时间可以通过计算这两个属性的差值得出。

计算各资源的加载时长

要计算资源的加载时间，可以使用 PerformanceResourceTiming 对象的几个属性：

- startTime：资源开始加载的时间戳
- responseEnd：资源最后一个字节完成下载的时间戳
- duration：资源加载完成所需的时间（毫秒）

PerformanceResourceTiming对象中第一个资源的 fetchStart 为什么不为0，代表含义是什么?

fetchStart属性指的是浏览器开始请求资源的时间戳，通常情况下这个时间戳不为0是因为浏览器在发送请求之前会进行一些处理，例如：DNS解析、TCP连接等。这些处理过程会消耗一些时间，因此导致fetchStart不为0

具体而言，浏览器在发送网络请求之前需要完成以下步骤：

- DNS解析：将域名解析成IP地址，浏览器需要向DNS服务器查询解析结果，这个过程通常需要数十毫秒甚至数百毫秒的时间
- 建立TCP连接：浏览器需要向服务器发送一个SYN包，等待服务器返回一个ACK包，这个过程通常需要数十毫秒的时间
- 发送HTTP请求：浏览器需要构建HTTP请求报文，包括请求头、请求体等内容，这个过程通常很快，但也需要一定的时间

因此，fetchStart属性不为0是正常的，它反映了浏览器开始处理资源请求的时间点，而非真正开始传输数据的时间点。

白屏时间 FP

白屏时间 FP（First Paint）指的是从用户输入 url 的时刻开始计算，一直到页面有内容展示出来的时间节点，标准≤2s
这个过程包括 dns 查询、建立 tcp 连接、发送 http 请求、返回 html 文档、html 文档解析

首次内容绘制时间 FCP

FCP(First Contentful Paint) 表示页面任一部分渲染完成的时间，标准≤2s

最大内容绘制时间 LCP

LCP(Largest Contentful Paint)表示最大内容绘制时间，标准≤2.5 秒

累积布局偏移值 CLS

CLS(Cumulative Layout Shift) 表示累积布局偏移，标准≤0.1

首字节时间 TTFB

平常所说的TTFB，默认指导航请求的TTFB
导航请求：在浏览器切换页面时创建，从导航开始到该请求返回 HTML

首次输入延迟 FID

FID（first input delay）首次输入延迟，标准是用户触发后，浏览器的响应时间， 标准≤100ms

计算资源的缓存命中率

缓存命中率：从缓存中得到数据的请求数与所有请求数的比率，理想状态是缓存命中率越高越好，缓存命中率越高说明网站的缓存策略越有效，用户打开页面的速度也会相应提高。

如何判断该资源是否命中缓存?

- 通过performance.getEntries()找到所有资源的信息
- 在这些资源对象中有一个transferSize 字段，它表示获取资源的大小，包括响应头字段和响应数据的大小
- 如果这个值为 0，说明是从缓存中直接读取的（强制缓存）
- 如果这个值不为 0，但是encodedBodySize 字段为 0，说明它走的是协商缓存（encodedBodySize 表示请求响应数据 body 的大小）

将所有命中缓存的数据 / 总数据 就能得出缓存命中率

常见的前端项目性能优化手段

分析打包后的文件

- 可以使用webpack-bundle-analyzer插件生成资源分析图
- vue 项目可以在 build 命令上添加--report 指令，"build": "vue-cli-service build --report"，打包时会生成 report.html 页面，即资源分析图

我们要清楚的知道项目中使用了哪些三方依赖，以及依赖的作用。特别对于体积大的依赖，分析是否能优化。

合理处理公共资源

如果项目支持 CDN，可以配置externals，将Vue、Vue-router、Vuex、echarts等公共资源，通过 CDN 的方式引入，不打到项目里边
如果项目不支持 CDN，可以使用DllPlugin动态链接库，将业务代码和公共资源代码相分离，公共资源单独打包，给这些公共资源设置强缓存（公共资源基本不会变），这样以后可以只打包业务代码，提升打包速度

首屏必要资源 preload 预加载 和 DNS 预解析

首屏不必要资源延迟加载

方式一： defer 或 async
使用 script 标签的defer或async属性，这两种方式都是异步加载 js，不会阻塞 DOM 的渲染。

async 是无顺序的加载，而 defer 是有顺序的加载

- 使用 defer 可以用来控制 js 文件的加载顺序
- 如果你的脚本并不关心页面中的 DOM 元素（文档是否解析完毕），并且也不会产生其他脚本需要的数据，可以使用 async，如添加统计、埋点等资源

方式二：依赖动态引入
项目依赖的资源，推荐在各自的页面中动态引入，不要全部都放到 index.html 中
比如echart.js，只有 A 页面使用，可以在 A 页面的钩子函数中动态加载，在onload事件中进行 echart 初始化

方式三：import()
使用import() 动态加载路由和组件，对资源进行拆分，只有使用的时候才进行动态加载

合理利用缓存

html 资源设置协商缓存，其他 js、css、图片等资源设置强缓存
当用户再次打开页面时，html 先和服务器校验，如果该资源未变化，服务器返回 304，直接使用缓存的文件；若返回 200，则返回最新的 html 资源

网络方面的优化

- 开启服务器 Gzip 压缩，减少请求内容的体积，对文本类能压缩 60%以上
- 使用 HTTP2，接口解析速度快、多路复用、首部压缩等
- 减少 HTTP 请求，使用 url-loader，limit 限制图片大小，小图片转 base64

代码层面的优化

- 前端长列表渲染优化，分页 + 虚拟列表，长列表渲染的性能效率与用户体验成正比
- 图片的懒加载、图片的动态裁剪

> 特别是是手机端项目，图片几乎不需要原图，使用七牛或阿里云的动态裁剪功能，可以将原本几M的大小裁剪成几k

- 动画的优化，动画可以使用绝对定位，让其脱离文档流，修改动画不造成主界面的影响

> 使用 GPU 硬件加速包括：transform 不为none、opacity、filter、will-change

- 函数的节流和防抖，减少接口的请求次数
- 使用骨架屏优化用户等待体验，可以根据不同路由配置不同的骨架

> vue 项目推荐使用vue-skeleton-webpack-plugin，骨架屏原理将
>
> 中的内容替换掉

- 大数据的渲染，如果数据不会变化，vue 项目可以使用Object.freeze()

> Object.freeze()方法可以冻结一个对象，Vue 正常情况下，会将 data 中定义的是对象变成响应式，但如果判断对象的自身属性不可修改，就直接返回改对象，省去了递归遍历对象的时间与内存消耗

- 定时器和绑定的事件，在页面销毁时卸载

性能分析总结

- 先用 Lighthouse 得到当前页面的性能得分，了解页面的整体情况，重点关注 Opportunities 优化建议和 Diagnostics 诊断问题列表
- 通过 Performance 工具了解页面加载的整个过程，分析到底是资源加载慢、dom 渲染慢、还是 js 执行慢，找到具体的性能瓶颈在哪里，重点关注长任务（long task）
- 利用 Memory 工具，了解页面整体的内存使用情况，通过 JS 堆动态分配时间线，找到内存最高的时刻。结合具体的代码，去解决或优化内存变大的问题

设计一个不能操作DOM和调接口的环境

- 利用 iframe 创建沙箱，取出其中的原生浏览器全局对象作为沙箱的全局对象
- 设置一个黑名单，若访问黑名单中的变量，则直接报错，实现阻止\隔离的效果
- 在黑名单中添加 document 字段，来实现禁止开发者操作 DOM
- 在黑名单中添加 XMLHttpRequest、fetch、WebSocket 字段，实现禁用原生的方式调用接口
- 若访问当前全局对象中不存在的变量，则直接报错，实现禁用三方库调接口
- 最后还要拦截对 window 对象的访问，防止通过 window.document 来操作 DOM，避免沙箱逃逸

Webpack 打包流程

- webpack 从项目的entry入口文件开始递归分析，调用所有配置的 loader对模块进行编译
  因为 webpack 默认只能识别 js 代码，所以如 css 文件、.vue 结尾的文件，必须要通过对应的 loader 解析成 js 代码后，webpack 才能识别
- 利用babel(babylon)将 js 代码转化为ast抽象语法树，然后通过babel-traverse对 ast 进行遍历

- 遍历的目的找到文件的import引用节点
  因为现在我们引入文件都是通过 import 的方式引入，所以找到了 import 节点，就找到了文件的依赖关系
- 同时每个模块生成一个唯一的 id，并将解析过的模块缓存起来，如果其他地方也引入该模块，就无需重新解析，最后根据依赖关系生成依赖图谱
- 递归遍历所有依赖图谱的模块，组装成一个个包含多个模块的 Chunk(块)
- 最后将生成的文件输出到 output 的目录中

热更新原理

主要是通过websocket实现，建立本地服务和浏览器的双向通信。当代码变化，重新编译后，通知浏览器请求更新的模块，替换原有的模块

- 通过`webpack-dev-server`开启server服务，本地 server 启动之后，再去启动 websocket 服务，建立本地服务和浏览器的双向通信
- webpack 每次编译后，会生成一个Hash值，Hash 代表每一次编译的标识。本次输出的 Hash 值会编译新生成的文件标识，被作为下次热更新的标识
- webpack监听文件变化（主要是通过文件的生成时间判断是否有变化），当文件变化后，重新编译
- 编译结束后，通知浏览器请求变化的资源，同时将新生成的 hash 值传给浏览器，用于下次热更新使用
- 浏览器拿到更新后的模块后，用新模块替换掉旧的模块，从而实现了局部刷新

plugin系统

它用于扩展webpack的功能，webpack 通过内部的事件流机制保证了插件的有序性，底层是利用发布订阅模式，webpack 在运行过程中会广播事件，插件只需要监听它所关心的事件，在特定的时机对资源做处理。

它的组成部分：

- Plugin 的本质是一个 node 模块，这个模块导出一个 JavaScript 类
- 它的原型上需要定义一个apply 的方法
- 通过compiler获取 webpack 内部的钩子，获取 webpack 打包过程中的各个阶段
  钩子分为同步和异步的钩子，异步钩子必须执行对应的回调
- 通过compilation操作 webpack 内部实例特定数据
- 功能完成后，执行 webpack 提供的 cb 回调

常用的 Plugin 插件

插件名称	作用
html-webpack-plugin	生成 html 文件,引入公共的 js 和 css 资源
webpack-bundle-analyzer	对打包后的文件进行分析，生成资源分析图
terser-webpack-plugin	代码压缩，移除 console.log 打印等
HappyPack Plugin	开启多线程打包，提升打包速度
Dllplugin	动态链接库，将项目中依赖的三方模块抽离出来，单独打包
DllReferencePlugin	配合 Dllplugin，通过 manifest.json 映射到相关的依赖上去
clean-webpack-plugin	清理上一次项目生成的文件
vue-skeleton-webpack-plugin	vue 项目实现骨架屏

loader系统

webpack只能直接处理js格式的资源，任何非js文件都必须被对应的loader处理转换为js代码。

它的组成部分：

- 本质是一个node模块
- 该模块导出一个函数，函数接收source（源文件），返回处理后的source

它的执行顺序

- 相同优先级的loader链，执行顺序为：从左到右，从下到上
- 例如，`use: ['loader1', 'loader2', 'loader3']`，执行顺序为`loader3 → loader2 → loader1`

编写一个loader

- 声明一个函数，在内部实现功能后将其返回，最后使用`module.exports`将其导出

在vue中使用loader

- 在vue.config.js中引入编写好的loader
- 在configureWebpack中添加配置

常用的loader如下所示

Webpack5 模块联邦

webpack5 模块联邦(Module Federation) 使 JavaScript 应用得以从另一个 JavaScript 应用中动态的加载代码，实现共享依赖，用于前端的微服务化
比如，项目A、项目B需要共用项目C组件，使用模块联邦后，可以在远程模块的webpack配置中将C组件暴露出去，项目A和项目B就可以远程进行依赖引用。当C组件发生变化后，A和B无需重新引用。

模块联邦利用 webpack5 内置的ModuleFederationPlugin插件，实现了项目中间相互引用的按需热插拔。

使用方法如下所示：

Vite的原理

- Vite 利用浏览器支持原生的es module模块，开发时跳过打包的过程，提升编译效率
- 当通过 import 加载资源时，浏览器会发出 HTTP 请求对应的文件，Vite拦截到该请求，返回对应的模块文件

vite热更新速度

Vite 热更新的速度不会随着模块增多而变慢。它与webpack相比区别如下：

- Webpack 的热更新原理：一旦某个依赖（比如上面的 a.js）改变，就将这个依赖所处的 整个module 更新，并将新的 module 发送给浏览器重新执行
  试想如果依赖越来越多，就算只修改一个文件，热更新的速度会越来越慢
- Vite 的热更新原理：如果 a.js 发生了改变，只会重新编译这个文件 a，而其余文件都无需重新编译
  所以理论上 Vite 热更新的速度不会随着文件增加而变慢

vite的实现流程

- 通过koa开启一个服务，获取请求的静态文件内容
- 通过es-module-lexer 解析 ast 拿到 import 的内容
- 判断 import 导入模块是否为三方模块，是的话，返回node_module下的模块， 如 import vue 返回 import './@modules/vue'
- 如果是.vue文件，vite 拦截对应的请求，读取.vue 文件内容进行编译，通过compileTemplate 编译模板，将template转化为render函数
- 通过 babel parse 对 js 进行编译，最终返回编译后的 js 文件

AST抽象语法树

AST是源代码的抽象语法结构的树状表现形式，是babel的核心，在 js 世界中，可以认为抽象语法树(AST)是最底层。

Babel的基本原理与作用

Babel 是一个 JS 编译器，把我们的代码转成浏览器可以运行的代码。
它主要用于将新版本的代码转换为向后兼容的 js 语法(Polyfill 方式)，以便能够运行在各版本的浏览器或其他环境中
基本原理

- 核心就是 AST (抽象语法树)
- 首先，将源码转成抽象语法树
- 然后，对语法树进行处理生成新的语法树
- 最后，将新语法树生成新的 JS 代码

Babel的流程

3 个阶段： parsing (解析)、transforming (转换)、generating (生成)

- 通过babylon将 js 转化成 ast (抽象语法树)
- 通过babel-traverse是一个对 ast 进行遍历，使用 babel 插件转化成新的 ast
- 通过babel-generator将 ast 生成新的 js 代码

开发一个babel插件

Babel 插件担负着编译过程中的核心任务：转换 AST，它的基本格式如下所示：

- 一个函数，参数是 babel，然后就是返回一个对象，key是visitor，然后里面的对象是一个箭头函数
- 函数有两个参数，path表示路径，state表示状态
- CallExpression就是我们要访问的节点，path 参数表示当前节点的位置，包含的主要是当前节点（node）内容以及父节点（parent）内容

Gulp的使用

gulp 是基于 node 流 实现的前端自动化开发的工具

适用场景
在前端开发工作中有很多“重复工作”，比如批量将Scss文件编译为CSS文件。

通过Gulp给elementUI增加一键换肤功能

总体流程为：

- 使用css var()定义颜色变量
- 创建主题theme.css文件，存储所有的颜色变量
- 使用gulp将theme.css合并到base.css中，解决按需引入的情况
- 使用gulp将index.css与base.css合并，解决全局引入的情况

三种常见模式的解析