目录

1.1 什么是规范？

规范，名词意义上：即明文规定或约定俗成的标准，如：道德规范、技术规范等。动词意义上：是指按照既定标准、规范的要求进行操作，使某一行为或活动达到或超越规定的标准，如：规范管理、规范操作。

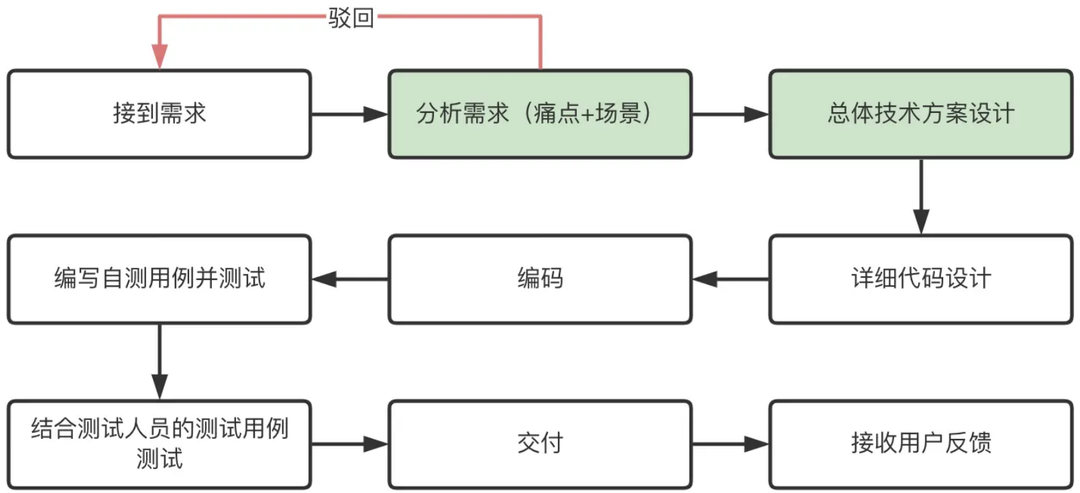
1.2 为什么需要规范？

- 降低新成员融入团队的成本，同时也一定程度避免挖坑
- 提高开发效率、团队协作效率，降低沟通成本
- 实现高度统一的代码风格，方便 review，另外一方面可以提高项目的可维护性
- 规范是实现自动化的基础
- 规范是一个团队知识沉淀的直接输出

1.3 前端规范都包含哪些?

1. 工作流程规范
2. 技术栈规范
3. 编码规范
4. git规范
5. 文档规范
6. IDE规范
7. 前后端协作规范

1.3.1 工作流程规范



在接收到需求后应第一时间去了解这个需求的背景是什么？这么做到底有没有解决用户的痛点？或者说用户更深层次的需求是什么？如果团队的产品经理经验不丰富，往往可以在这个阶段砍掉很多不合理的需求（这一点真的很重要）。对于复杂大功能往往还需要进行技术方案调研和技术方案设计，并输出详细的设计文档。涉及到细节上，则需要将数据流走向、组件设计等通过脑图的形式呈现出来。

1.3.2 技术栈规范

- 到底是用 TypeScript 还是 JavaScript ？
- 到底是用 Vue 还是 React ？
- 到底是用 Less 还是 Sass ？
- 到底是用 Webpack 还是 Vite ？
- 到底是用 Koa 还是 Express ？
- 到底是用element UI 、 ant design、还是tdesign?
- ……

这些都需要提前约定好

1.3.3 编码规范

无规矩不成方圆，有了规范才有好的团队

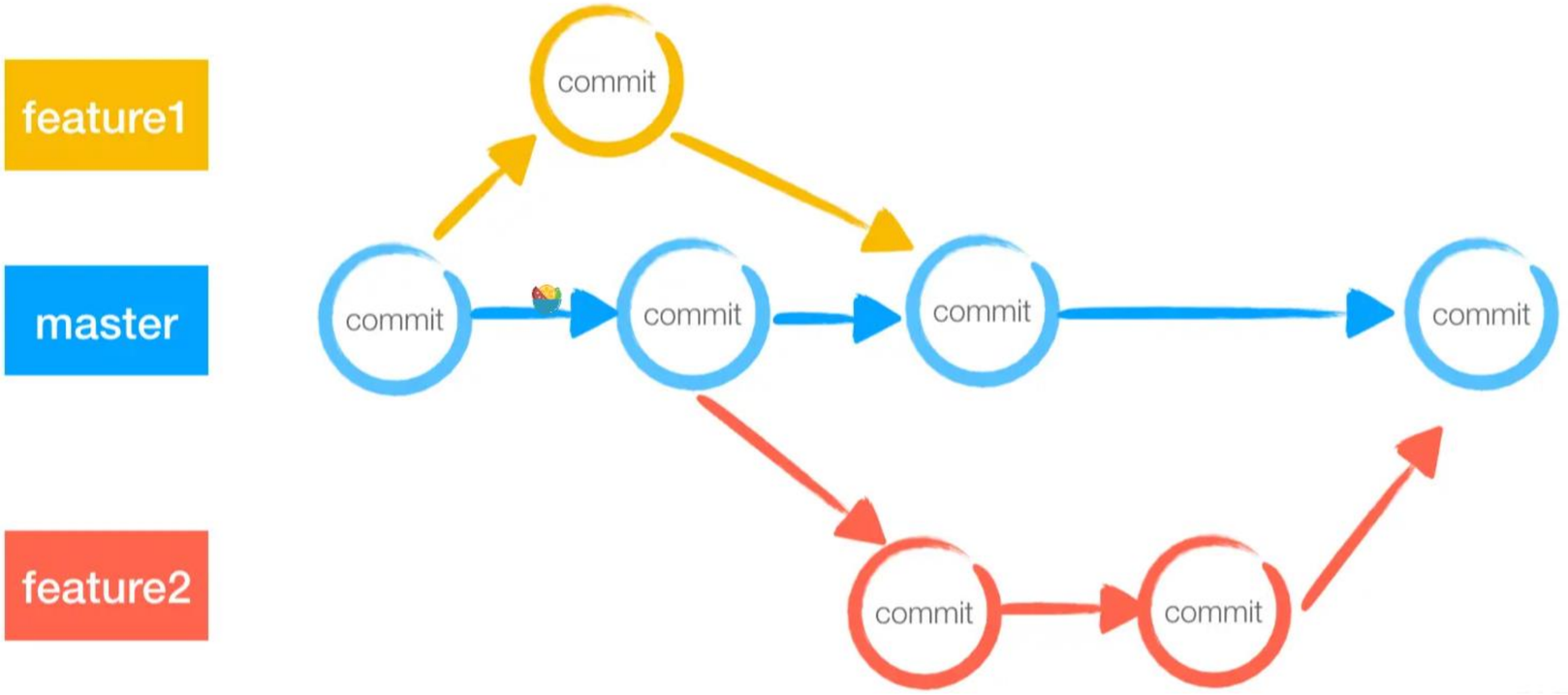
可以借鉴腾讯前端代码规范

- html规范
- js规范（书写、换行、缩进、标点）
- css规范
- 图片规范（格式、大小、质量、引入方式）
- 命名规范（目录、文件名、className、JS变量）
- 注释规范
- 模块化规范（CJS/ESM）
- 成功和错误提示规范
- 框架代码规范（VUE\REACT文档中的规范）

1.3.4 git规范

这里主要说一下较为常用的 功能分支流；

功能分支工作流是以集中式工作流为基础的。它提倡为各个新功能分配一个专门的分支来开发， 当功能分支稳定，或者说经过完备的测试之后才合并到 master 分支。



1.3.4 git规范

Git Commit 规范 主要可以帮助开发人员在 code review 期间更容易理解提交的内容，现在大部分主流 commit 规范都是基于Angular 团队的规范而衍生出来的，它的 message 格式如下：

```
<type>: <subject>
// 注意冒号 : 后有空格
// 如 feat: 增加用户中心的 xx 功能
```

type 必填，表示提交类型，值一般有以下几种：

feat：新功能 feature
bug：测试反馈 bug 列表中的 bug 号
fix： 修复 bug
ui：更新UI；
docs： 文档注释变更
style： 代码格式(不影响代码运行的变动)；
refactor： 重构、优化(既不增加新功能，也不是修复bug)；
perf： 性能优化；
release：发布；
deploy：部署；
test： 增加测试
chore： 构建过程或辅助工具的变动
revert： 回退
build： 打包

subject 用于对 commit 进行简短的描述；

1.3.5 文档规范

相信大家都吃过没文档的亏，文档对于团队的发展是至关重要的

它无论是对项目的开发和维护，还是对旧与新的交替，亦或者是团队的建设轨迹，都有着无可代替的作用；

文档有什么作用：

- 对新人友好，快速融入团队；
- 规范化编码；
- 有效的控制团队的原型以及代码的版本；
- 重大决策与讨论的记录；
- 完善内部技术讨论交流；
- 团队建设；

1.3.6 IDE规范

IDE 编辑器规范的意义

- 统一配置，方便开发
- 规定团队编码风格
- 规范对应插件及配置

A 同学研发一般喜欢使用 Webstorm 开发;
某日 A 同学在研发一些 jQuery 老旧项目时:
每次 save 会自动格式化一次代码;
但是由于 Webstorm 编辑器内置的格式化插件的差异;
就导致了与其它同学的代码出现了格式风格混乱的情况

建议统一使用vs code

Auto Rename 、Tag auto-close-tag 、Highlight Matching Tag 、ESLint 、Prettier 、Git History 、volar 、Vue 3 Snippets

1.3.7 前后端协作规范

随着前后端分离开发模式的流行，前端和后端已经在各自领域上渐行渐远；我们把前后端共同研发的一个需求所产生的关联称之为联调；

如何去把控好这个联调的品质就是我们值得关注的点了

稍不注意就很可能产生不必要的问题。

因此，咱们就很有必要制定前后端协作规范来解决这些问题了 ~

1.3.7 前后端协作规范

接口规范

接口风格使用 RESTful 风格；

请求方法规范：

GET：获取对应的信息；
POST：用于创建或者某些资源的提交；
UPDATE：更新某些资源；
DELETE：删除某个资源；
OPTIONS：对请求的校验，与 POST 配合；

其它规范：

URI 结尾不应包含（/）；
正斜杠分隔符（/）必须用来指示层级关系；
应使用连字符（-）来提高URI的可读性
不得在URI中使用下划线（_）；
URI路径中全都使用小写字母
具体详见：RESTful 架构详解

接口文档规范：

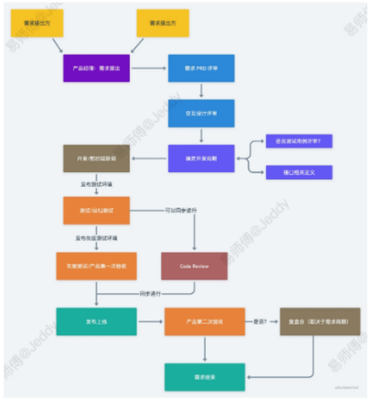
版本号；

接口注释与字段的描述；

具体接口定义：

方法名称或者 URI
方法描述
请求参数及其描述，必须说明类型(数据类型、是否可选等)
响应参数及其描述，必须说明类型(数据类型、是否可选等)
可能的异常情况、错误代码、以及描述
请求示例，可选

1.4 如何落地前端规范？



- 前端规范
  - 1.工作流程规范
  - 2.技术栈规范
    - 前端
      - react
      - vue/vue next?
    - 后端
      - egg.js
      - nest.js
    - ssr
      - nuxt.js
      - next.js
  - 3.编码规范
    - 代码风格
      - HTML、JavaScript、typescript、css、vue、react等
    - 工具自动化
      - eslint
      - lslint
      - stylelint
      - volar
      - prettier
  - 4.git规范
    - huskey
    - commitlint
  - 5.文档规范
    - markdown（lint-md）
      - 新人文档
      - 技术文档
      - 业务文档
      - 技术分享
    - 在线文档 （腾讯文档）
      - 项目文档
      - 计划文档
  - 6.IDE规范
    - vscode
    - webstorm
    - Hbuilder
  - 7.前后端协作规范
    - BFF
      - 接口数据返回统一原则
      - 微服务接口聚合原则
      - 业务下沉原则

2. 前端性能分析工具——Lighthouse

在前端开发中，自己开发的app或者web page性能的好坏，一直是让前端开发人员很在意的话题。因为影响用户浏览网页速度的因素主要有：服务端数据返回、网络传输、页面渲染等等，这些方面做的不够好，都会影响客户体验。所以我们除了在开发的过程中注意代码的质量，同时还需要专业的网站测试工具辅助，让我们知道自己的网页还有哪些需要更为优化的方面。

当下流行的前端性能分析工具有很多，比如 Lighthouse、Pingdom、SpeedTracker、WebPageTest、Sitespeed.io等等。

这里主要介绍我自己常用的一款工具：Lighthouse，感觉还不错，上手容易，操作简单。

Lighthouse是一个开源的自动化工具，用于帮助改进网络应用的质量。可将其作为一个Chrome扩展程序运行，或从命令行运行。Lighthouse分析web应用程序和web页面，收集关于开发人员最佳实践的现代性能指标和见解，让开发人员根据生成的评估页面，来进行网站优化和完善，提高用户体验。

2. 前端性能分析工具——Lighthouse

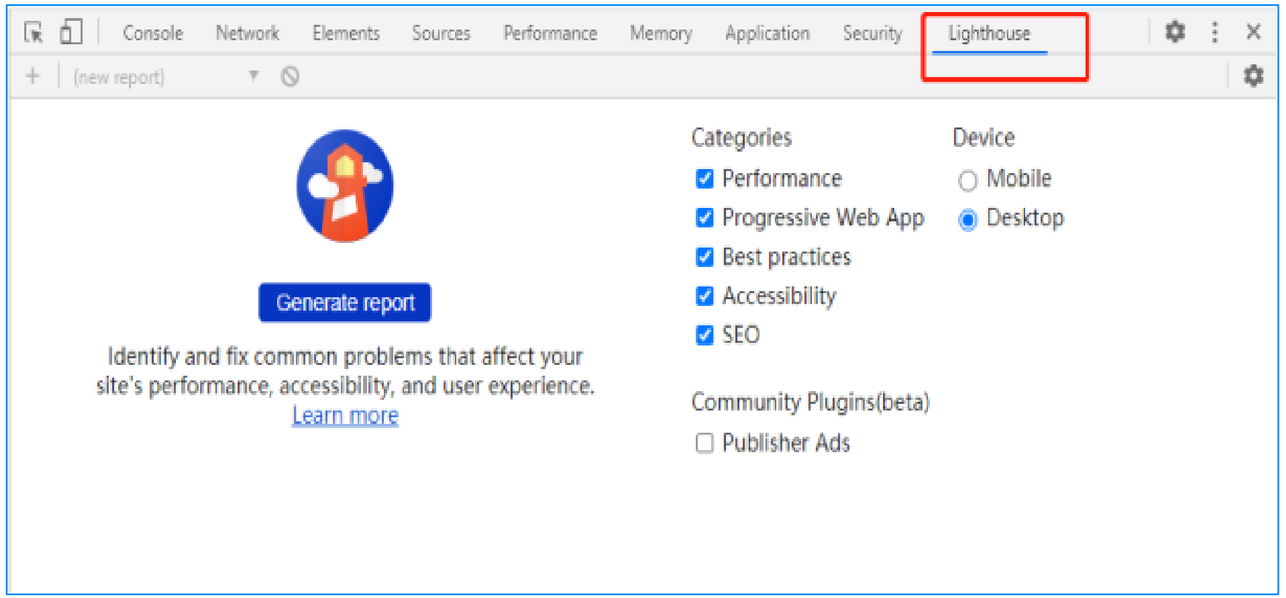
安装使用

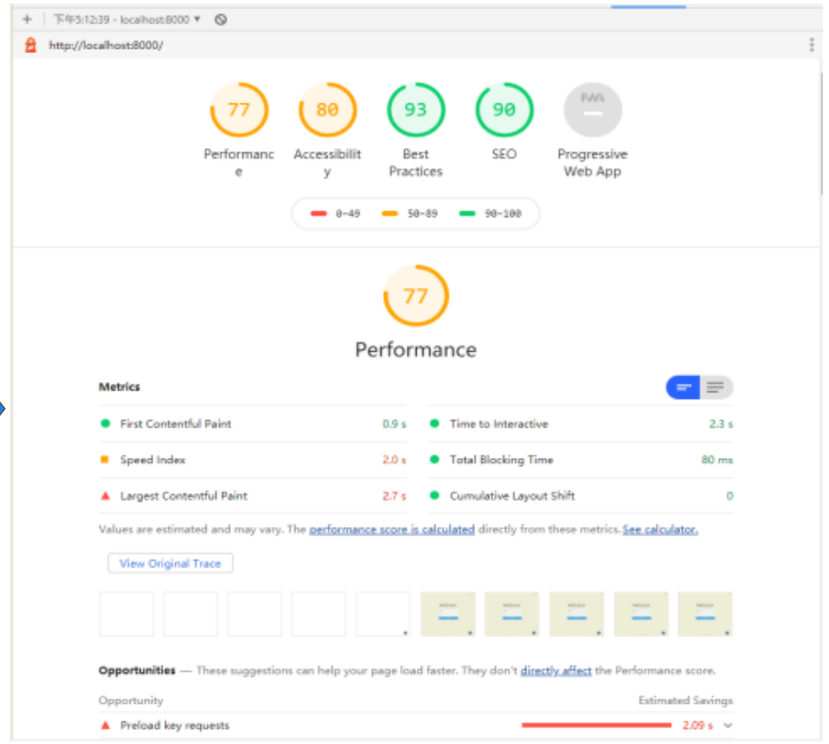
安装方式一：

1. 全局安装：npm install -g lighthouse
2. 创建一个文件夹（用于存放生成的报告文件），然后进入到文件夹，运行命令：lighthouse http://xxxxxxx
3. 然后静态运行，生成报告，跑完之后，会生成一个.html的报告

安装方式二（推荐）：

通过chrome按F12打开调试模式，会在浏览器控制台上方出现Lighthouse图标，如图所示，点击Generate report按钮，即可以看到最终的性能检测报告。





2.1 前端性能分析工具——Lighthouse

指标详解

1.Performance（性能）

First Contentful Paint(FCP)：衡量页面开始加载到页面中第一个元素被渲染之间的时间。元素包含文本、图片、canvas等。

Speed Index：代表页面内容渲染所消耗的时间，该值越低越好。

Largest Contentful Paint(LCP)：衡量标准视口内可见的最大内容元素的渲染时间。元素包括img、video、div及其他块级元素。

Time to Interactive(TTL)：测量页面所有资源加载成功并能够可靠地快速响应用户输入的时间，即互动时间。

Total Blocking Time(TBT)：这是 FCP 与 TTL 之间的所有时间段的总和。

Cumulative Layout Shift (CLS)：衡量视觉稳定性，为了提供良好的用户体验，页面的CLS应保持小于 0.1。

2.Accessibility（可访问性）

无障碍功能：

当我们说某个网站具有无障碍功能时，我们的意思是网站的内容可用，其功能可由任何人操作。所有用户都能看见和使用键盘、鼠标或触摸屏，并且与网页内容的交互方式也清晰明了。这会让使用者获得良好的体验。我们在探讨无障碍功能时往往是围绕身体有缺陷的用户

2.1 前端性能分析工具——Lighthouse

指标详解

3.Best Practices（最佳实践）

检查网页总体代码运行状况。通过的审核显示了针对最佳实践还验证了其他内容：

Uses HTTPS：使用HTTPS

Avoids requesting geolocation permission on page load：避免在页面加载时请求地理位置许可

Avoids front-end JavaScript libraries with known security vulnerabilities：避免具有已知安全漏洞的前端JavaScript库

Allows users to paste into password fields：允许用户粘贴到密码字段中

4.SEO（搜索引擎优化）

搜索引擎优化计分卡，它检查页面是否针对搜索引擎结果排名进行了优化。

这些是Lighthouse使用的标准：

没有阻止页面建立索引、链接具有描述性文字、页面具有成功的HTTP状态代码、具有宽度或初始比例的<meta name=”viewport”>标签(适合移动设备)、文档具有<title>元素、文档具有元描述等等。

5.Progressive Web App（PWA）

Is installable：可安装

Works in any browser：在任何浏览器中均可使用

Starts fast and stays fast：快速启动并保持快速

Is fully accessible：完全可访问

Responsive to any screen size：响应任何屏幕尺寸

Provides a custom offline page：提供自定义的离线页面

Can be discovered through search：可以通过搜索发现

Works with any input type, such as a mouse, a keyboard, a stylus, or touch：适用于任何输入类型，例如鼠标，键盘，手写笔或触摸

Provides context for permission requests：提供权限请求的上下文

Follows best practices for healthy code：遵循最佳实践以获取健康代码

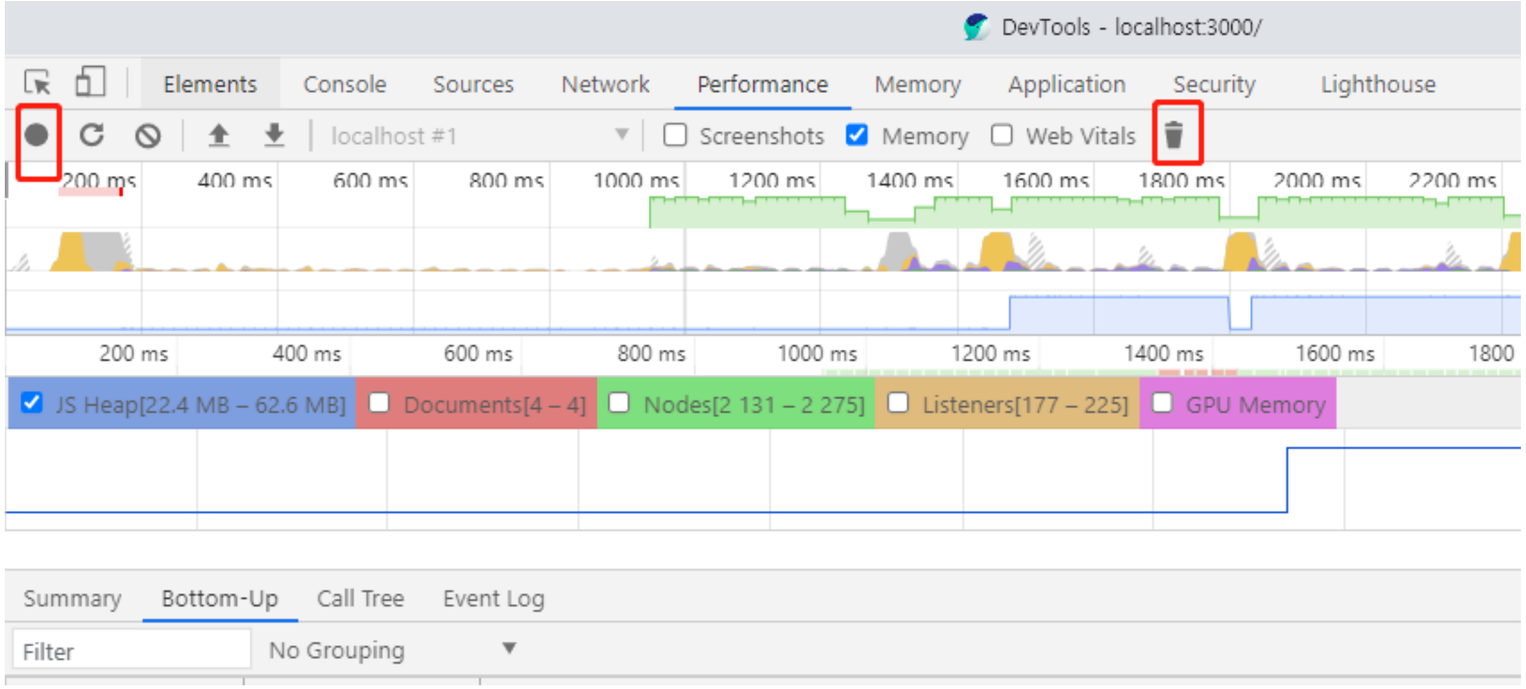
2. 前端性能之内存分析工具——Performance

```
1  const testGc = () =>{
2    const testGcValue = new Array(10000000);
3    console.log("测试垃圾回收", testGcValue);
4  };
5
6  return (<>
7          <Button onClick={testGc}>测试垃圾回收</Button>
8    </>)
```

左边的代码就是一个按钮，点击之后创建一个数组，执行一些计算。

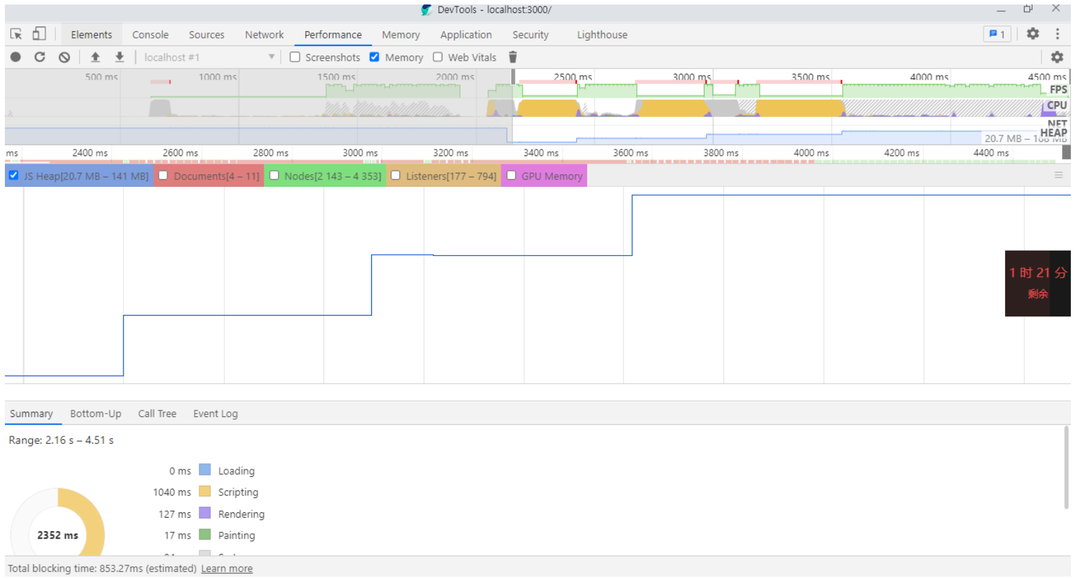
我们最后加了一个 console.log 打印了下这个数组。

先点击GC，勾选memory,再点击录制按钮



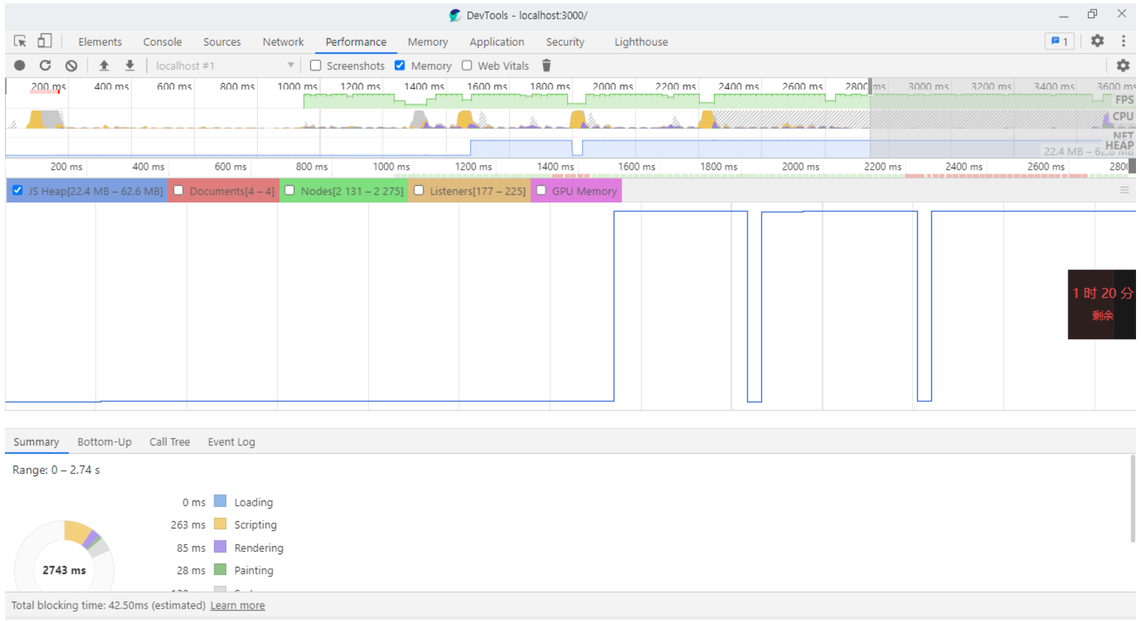
2.2 前端性能之内存分析工具——Performance

然后点击三次‘测试垃圾回收按钮’，你会发现内存是这样的：



内存占用有三次增长，因为我们点击三次按钮的时候会创建 3 次大数组。
但是最后我们手动 GC 之后并没有回落下去，也就是这个大数组没有被回收。
按理来说，代码执行完，那用的内存就要被释放，然后再执行别的代码，结果这段代码执行完之后大数组依然占据着内存，这样别的代码再执行的时候可用内存就少了。这就是发生了内存泄漏，也就是代码执行完了不释放内存的流氓行为。

接下来我们注释了console.log的部分，然后重复上述操作。
现在的内存分配情况是这样的



分配了三次内存，但是 GC 后又会落下去了。
这才是没有内存泄漏的好代码。
那为啥 console.log 会导致内存泄漏呢？

2.3 前端性能之内存分析工具——Performance

结论

因为控制台打印的对象，你是不是有可能展开看？那如果这个对象在内存中没有了，是不是就看不到了？

所以有这个引用在，浏览器不会把你打印的对象的内存释放掉。

当然，也不只是 console.log 会导致内存泄漏，还有别的 4 种情况：

1. 定时器用完了没有清除，那每次执行都会多一个定时器的内存占用，这就是内存泄漏
2. DOM元素的事件监听，对同一个事件重复监听，但是忘记移除，会导致内存泄露。
3. 元素从 dom 移除了，但是还有一个变量引用着他，这样的游离的 dom 元素也不会被回收。每执行一次代码，就会多出游离的 dom 元素的内存，这也是内存泄漏
4. 闭包引用了某个变量，这个变量不会被回收，如果这样的闭包比较多，那每次执行都会多出这些被引用变量的内存占用。这样引用大对象的闭包多了之后，也会导致内存问题
5. 全局变量，这个本来就不会被 GC，要注意全局变量的使用

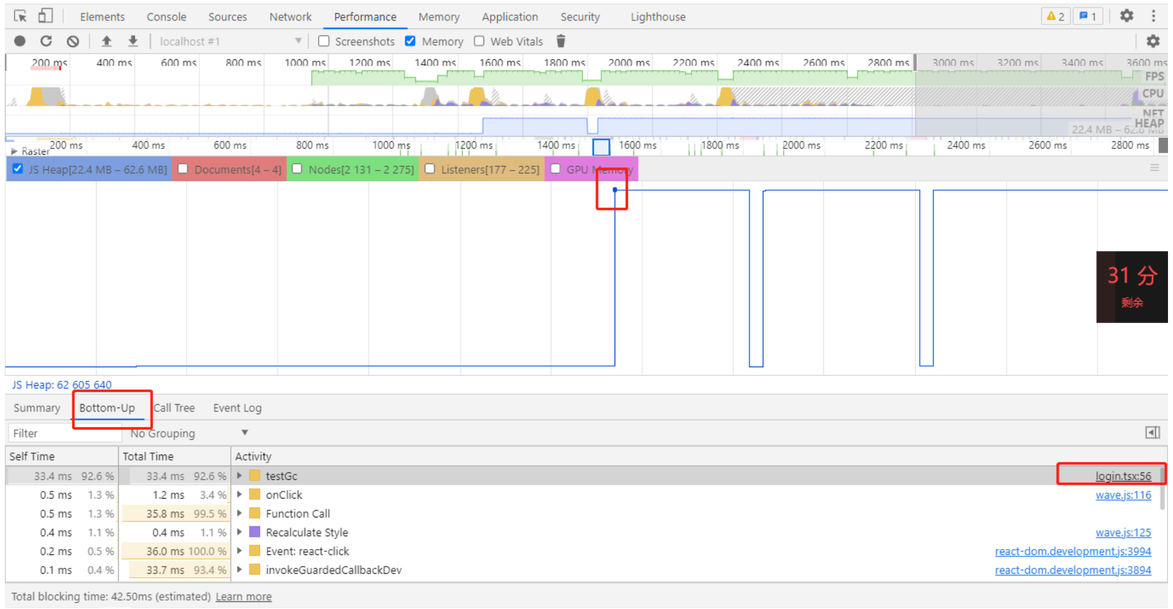
总之，全局变量、闭包引用的变量、被移除的 dom 依然被引用、定时器用完了没清除、console.log 都会发生代码执行完了，但是还占用着一部分内存的流氓行为，也就是内存泄漏。

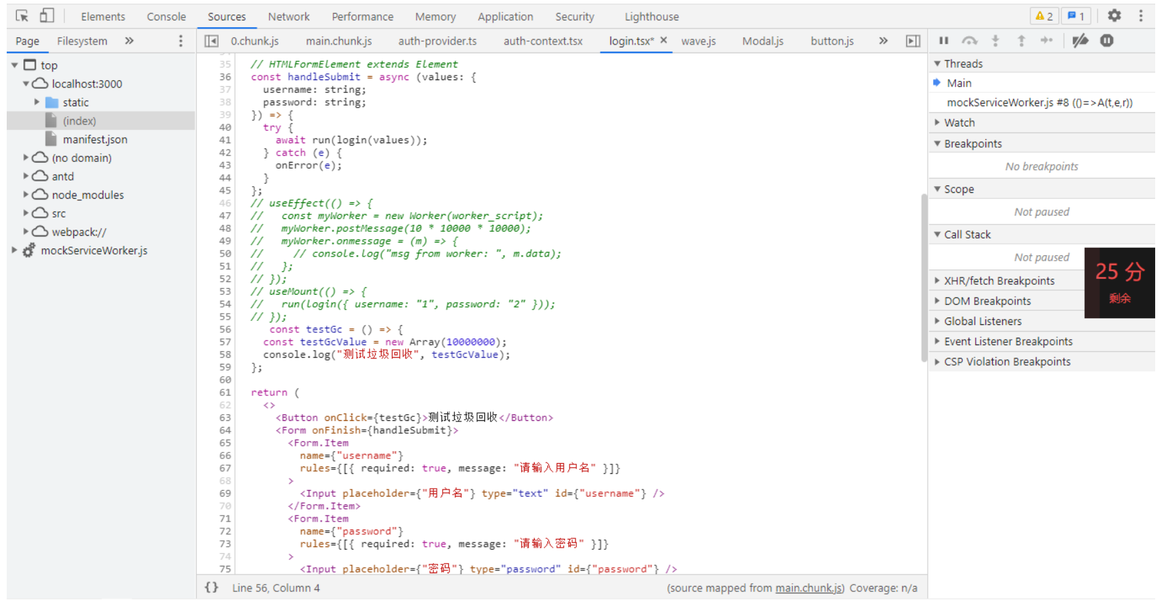
注意，这里指的是使用完毕后没有回收，在使用期间的内存增长是正常的。

2.4 前端性能之内存分析工具——Performance

如何使用Performance排查是哪里代码造成的?

点击内存分配的点，跳转到对应的目录





这样就定位到了分配内存的代码，分析一下哪里会有问题即可。
当然，前提还是要执行先 GC，再做一些操作，再 GC 的这个流程。
这是从代码角度来分析内存泄漏

3. 前端项目打包优化策略

优化方案：

路由懒加载（代码分割）

Vue 是单页面应用，可能会有很多的路由引入 ，这样使用 webpcak/vite 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。

第三方插件按需加载

我们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

常用插件库、静态资源使用CDN加速

在我们的项目中会使用到很多的第三方库，静态资源，这些往往都是不会作更改的，所以我们可以选择将这些资源使用CDN引入的方式，而不将这些库或资源打包到我们的项目目录中

gzip压缩

gzip 是 GNUzip 的缩写，最早用于 UNIX 系统的文件压缩。HTTP 协议上的 gzip 编码是一种用来改进 web 应用程序性能的技术，web 服务器和客户端（浏览器）必须共同支持 gzip。目前主流的浏览器，Chrome，firefox，IE等都支持该协议。常见的服务器如 Apache，Nginx，IIS 同样支持，gzip 压缩效率非常高，通常可以达到 70% 的压缩率，也就是说，如果你的网页有 30K，压缩之后就变成了 9K 左右，这样就可以大大提高网络传输效率，减少资源请求耗时。

打包不生成map文件

map文件的作用在于：项目打包后，代码都是经过压缩加密的，如果运行时报错，输出的错误信息无法准确得知是哪里的代码报错。有了map就可以像未加密的代码一样，准确的输出是哪一行哪一列有错。

4. 小程序如何设计及优化

用户对小程序速度的第一感知就是首屏加载速度，所以首屏加载要快，让用户‘误以为’小程序加载很快。

优化方案：

加快首屏加载，我们做了两件事：缓存和预加载，避免白屏。

利用 storage API，对变动频率比较低的异步数据进行缓存，二次启动时，先利用缓存数据进行初始化渲染，然后后台进行异步数据的更新，这不仅优化了性能，在无网环境下，用户也能很顺畅的使用到关键服务；

预加载是预加载页面框架结构，这里分几步：

第一步，对于小程序里相对固定（稳定）的页面结构框架（如首页一般是底部 tab，运营位，列表这样的结构），先预留位置。小程序打开后整个框架是直接出来的，再极快地用 storage 里的数据填上，让用户感觉不到有网络请求。

第二步，对网络动态数据的预加载。接口请求应返回图片宽高，让小程序可以预加载好框架，还可以先加载一张模糊的极小的预览图，等大图加载完毕后再渲染大图。

第三步，对用户即将去到的页面数据预加载。比如首页加载完毕后预请求后面几个 tab 的首屏数据；或者对于列表页和详情页中都要显示的公共数据（如标题、描述），从列表页传到详情页，这样用户就有了秒开详情页的感觉。

首屏加载完成后，再将首屏下面的数据异步加载渲染出来，这种预加载首屏数据 + 异步加载其他数据的渲染方式，给用户一种页面加载很快的感觉。