2024年前端最新场景题⾯试攻略

1. 前端如何实现截图？

前端实现截图需要使⽤ HTML5 的 Canvas 和相关 API ，具体步骤如下：

1. ⾸先在⻚⾯中创建⼀个 Canvas元素 ，并设置其宽⾼和样式。

2. 使⽤ Canvas API在 Canvas上绘制需要截图的内容 ，⽐如⻚⾯的某个区域、某个元素、 图⽚等。

3. 调⽤ Canvas API 中的

|  |
| --- |
| toDataURL |

()

⽅法将 Canvas转化为 base64 编码的图⽚数据。

4. 将 base64 编码的图⽚数据传递给后端进⾏处理或者直接在前端进⾏显⽰。 以下是⼀个简单的例⼦ ，实现了对整个⻚⾯的截图：

|  |
| --- |
| 1 htmlCopy code <html<head<meta charset="UTF-8"<title截图⽰例 </title<style  2  {position: fixed;left: 0;top: 0;z-index: 9999;}  3 </style</head<body<h1截图⽰例</h1<p这是⼀个简单的截图⽰例。 </p<button id="btn"截  图 </button<canvas id="canvas"</canvas<script  4 const canvas = document.getElementById('canvas');const ctx =  canvas.getContext('2d');const btn = document.getElementById('btn');  5 canvas.width = window.innerWidth;  6 canvas.height = window.innerHeight;  7 btn.addEventListener('click', () => {  8 ctx.clearRect(0, 0, canvas.width, canvas.height);  9 ctx.drawImage(document.documentElement, 0, 0);const imgData =  canvas.toDataURL();console.log(imgData);  10 });</script</body</html |

这个例⼦中 ，在⻚⾯中创建了⼀个

|  |
| --- |
| canvas |

元素 ，并设置其宽⾼和样式 ，将其放在⻚⾯最上⽅ 。在

点击“ 截图”按钮时 ，通过

|  |
| --- |
| toDataURL |

()

⽅法将整个⻚⾯的截图转换为 base64 编码的图⽚数据，

并打印到控制台上。

2. 当QPS达到峰值时, 该如何处理？ 当QPS达到峰值时 ，可以从以下⼏个⽅⾯来进⾏优化：

1. 数据库优化：数据库的优化包括优化SQL语句、使⽤索引、避免全表扫描、分表分库等措施 ，以提 ⾼数据库的读写性能。

2. 缓存优化：缓存可以降低对数据库的访问频率 ，提⾼响应速度。可以使⽤Redis、 Memcached等缓 存技术 ，减轻服务器负载。

3. 代码优化：优化代码可以提⾼代码的执⾏效率 ，减少不必要的开销。可以通过⼀些优化⼿段 ，如减 少不必要的代码执⾏ 、避免循环嵌套、避免不必要的递归调⽤等来提⾼代码的性能。

4. 负载均衡：负载均衡可以将请求分发到多个服务器上 ，减少单个服务器的负载 ，提⾼整个系统的性 能和可⽤性。

5. 异步处理：将⼀些计算量⼤ 、耗时⻓的操作异步处理 ，减少对主线程的阻塞 ，提⾼响应速度。

6. CDN加速：使⽤CDN技术可以将静态资源缓存到CDN节点上 ，提⾼资源的加载速度 ，减少服务器的 负载。

7. 硬件升级：可以通过升级服务器硬件 ，增加带宽等⽅式来提⾼系统的处理能⼒。 以上是⼀些常⻅的优化⼿段 ，需要根据具体情况进⾏选择和实施。

3. js 超过 Number 最⼤值的数怎么处理？ js 超过 Number 最⼤值的数怎么处理

在 JavaScript 中 ，超过

|  |
| --- |
| Number .MAX\_VALUE |

的数值被认为是

|  |
| --- |
| Infinity |

（正⽆穷⼤） 。如果要

处理超过

|  |
| --- |
| Number .MAX\_VALUE |

的数值 ，可以使⽤第三⽅的 JavaScript 库 ，如

|  |
| --- |
| big . s |

j

或

, 这些库可以处理任意精度的数值。

bignumber .js

例如 ，使⽤

|  |
| --- |
| big . s |

j

库可以将两个超过

|  |
| --- |
| Number .MAX\_VALUE |

的数相加：

|  |
| --- |
| 1  big = require('big.js'); 2  3  x =  big('9007199254740993'); 4  y =  big('100000000000000000'); 5  6  result = x.plus(y); 7  8 console.log(result.toString()); *// 输出：100009007194925474093* |

这⾥创建了两个

x

对象

|  |
| --- |
| big . s |

j

和 y ，分别存储超过

|  |
| --- |
| Number .MAX\_VALUE |

的数值。通过

plus

⽅法将它们相加 ，得到了正确的结果。最后 ，通过

|  |
| --- |
| toString |

⽅法将结果转换为字符串。

如果不依赖外部库 ，咋处理

JavaScript 中 ，数值超过了 Number 最⼤值时 ，可以使⽤ BigInt 类型来处理 ，它可以表⽰任意精度的 整数。

使⽤ BigInt 类型时 ，需要在数值后⾯添加⼀个

n

后缀来表⽰ BigInt 类型。例如：

|  |
| --- |
| 1  bigNum = 9007199254740993n; *// 注意：数字后⾯添加了* *'n' 后缀* |

注意 ，BigInt 类型是 ECMAScript 2020 新增的特性 ，因此在某些浏览器中可能不被⽀持。如果需要在 不⽀持 BigInt 的环境中使⽤ BigInt ，可以使⽤ polyfill 或者第三⽅库来实现。

4. 使⽤同⼀个链接 ， 如何实现 PC 打开是 web 应⽤ 、⼿机打 开是⼀个 H5 应⽤？

可以通过根据请求来源（ User-Agent）来判断访问设备的类型 ，然后在服务器端进⾏适配。例如 ，可 以在服务器端使⽤ Node.js 的 Express框架 ，在路由中对不同的 User-Agent 进⾏判断 ，返回不同的⻚ ⾯或数据。具体实现可以参考以下步骤：

1. 根据 User-Agent 判断访问设备的类型 ，例如判断是否为移动设备。可以使⽤第三⽅库如 ua- parser-js 进⾏ User-Agent 的解析。

2. 如果是移动设备 ，可以返回⼀个 H5 ⻚⾯或接⼝数据。

3. 如果是 PC 设备 ，可以返回⼀个 web 应⽤⻚⾯或接⼝数据。

具体实现⽅式还取决于应⽤的具体场景和需求 ，以上只是⼀个⼤致的思路。

5. 如何保证⽤⼾的使⽤体验

【如何保证⽤⼾的使⽤体验】 这个也是⼀个较为复杂的话题 ， 这个也不是问题了 ， 这个算是话题吧； 主要从以下⼏个⽅⾯思考问题：

1. 性能⽅向的思考

2. ⽤⼾线上问题反馈 ，线上 on call 的思考

3. ⽤⼾使⽤体验的思考 ， 交互体验使⽤⽅向

4. 提升⽤⼾能效⽅向思考

6. 如何解决⻚⾯请求接⼝⼤规模并发问题

如何解决⻚⾯请求接⼝⼤规模并发问题 ， 不仅仅是包含了接⼝并发 ， 还有前端资源下载的请求并发。 应该说这是⼀个话题讨论了；

个⼈认为可以从以下⼏个⽅⾯来考虑如何解决这个并发问题:

1. 后端优化：可以对接⼝进⾏优化 ，采⽤缓存技术 ，对数据进⾏预处理 ，减少数据库操作等。使⽤集 群技术 ，将请求分散到不同的服务器上 ，提⾼并发量。另外可以使⽤反向代理、负载均衡等技术 ， 分担服务器压⼒。

2. 做 BFF 聚合：把所有⾸屏需要依赖的接⼝ ， 利⽤服务中间层给聚合为⼀个接⼝ 。

3. CDN加速：使⽤CDN缓存技术可以有效减少服务器请求压⼒ ，提⾼⽹站访问速度。CDN缓存可以将 接⼝的数据存储在缓存服务器中 ，减少对原始服务器的访问 ，加速数据传输速度。

4. 使⽤ WebSocket：使⽤ WebSocket 可以建⽴⼀个持久的连接 ，避免反复连接请求。WebSocket 可以实现双向通信 ，⼤幅降低服务器响应时间。

5. 使⽤ HTTP2 及其以上版本 ， 使⽤多路复⽤ 。

6. 使⽤浏览器缓存技术：强缓存、协商缓存、离线缓存、Service Worker 缓存 等⽅向。

7. 聚合⼀定量的静态资源： ⽐如提取⻚⾯公⽤复⽤部分代码打包到⼀个⽂件⾥⾯ 、对图⽚进⾏雪碧图 处理 ， 多个图⽚只下载⼀个图⽚。

8. 采⽤微前端⼯程架构： 只是对当前访问⻚⾯的静态资源进⾏下载 ， ⽽不是下载整站静态资源。

9. 使⽤服务端渲染技术： 从服务端把⻚⾯⾸屏直接渲染好返回 ， 就可以避免掉⾸屏需要的数据再做 额外加载和执⾏。

7. 设计⼀套全站请求耗时统计⼯具

⾸先我们要知道有哪些⽅式可以统计前端请求耗时 从代码层⾯上统计全站所有请求的耗时⽅式主要有以下⼏种：

1. Performance API： Performance API 是浏览器提供的⼀组 API ，可以⽤于测量⽹⻚性能。通过 Performance API ，可以获取⻚⾯各个阶段的时间、资源加载时间等。其中 ，Performance

Timing API可以获取到每个资源的加载时间 ，从⽽计算出所有请求的耗时。

2. XMLHttpRequest 的 load 事件：在发送 XMLHttpRequest 请求时 ，可以为其添加 load 事件 ，在请 求完成时执⾏回调函数 ，从⽽记录请求的耗时。

3. fetch 的 Performance API：类似 XMLHttpRequest ，fetch 也提供了 Performance API ，可以通过 Performance API 获取请求耗时。

4. ⾃定义封装的请求函数：可以⾃⼰封装⼀个请求函数 ，在请求开始和结束时记录时间 ，从⽽计算请 求耗时。

设计⼀套前端全站请求耗时统计⼯具 可以遵循以下步骤：

1. 实现⼀个性能监控模块 ，⽤于记录每个请求的开始时间和结束时间 ，并计算耗时。

2. 在应⽤⼊⼝处引⼊该模块 ，将每个请求的开始时间记录下来。

3. 在每个请求的响应拦截器中 ，记录响应结束时间 ，并计算请求耗时。

4. 将每个请求的耗时信息发送到服务端 ，以便进⾏进⼀步的统计和分析。

5. 在服务端实现数据存储和展⽰ ，可以使⽤图表等⽅式展⽰请求耗时情况。

6. 对于请求耗时较⻓的接⼝ ，可以进⾏优化和分析 ，如使⽤缓存、使⽤异步加载、优化查询语句等。

7. 在前端应⽤中可以提供开关 ，允许⽤⼾⾃主开启和关闭全站请求耗时统计功能。 以下是⼀个简单的实现⽰例：

|  |
| --- |
| 1 *// performance.js* *performance = {*  2 timings: {},  3 config: {  4 reportUrl: '/report',  5 },  6 init() {*// 监听所有请求的开始时间window.addEventListener('fetchStart', (event) => {* *.timings[event.detail.id] = {*  7 startTime: Date.now(),  8 };  9 });  10 *// 监听所有请求的结束时间，并计算请求耗时window.addEventListener('fetchEnd',*  *(event) => {* *id = event.detail.id;* *(* *.timings[id]) {* *timing =*  *.timings[id];*  11 timing.endTime = Date.now();  12 timing.duration = timing.endTime - timing.startTime;  13 *// 将耗时信息发送到服务端* *reportData = {*  14 url: event.detail.url,  15 method: event.detail.method,  16 duration: timing.duration,  17 }; .report(reportData);  18 }  19 });  20 },  21 report(data) {*// 将耗时信息发送到服务端* *xhr =*  *XMLHttpRequest();*  22 xhr.open('POST',  .config.reportUrl);  23 xhr.setRequestHeader('Content-Type', 'application/json');  24 xhr.send(JSON.stringify(data));  25 },  26 };  27  28   performance; |

在应⽤⼊⼝处引⼊该模块：

|  |
| --- |
| 1 *// main.js* *performance*  *'./performance';*  2 performance.init(); |

在每个请求的响应拦截器中触发

|  |
| --- |
| fetchEnd |

事件：

|  |
| --- |
| 1 *// fetch.js* *EventBus*  *'./EventBus';* 2  3  fetch = (url, options) => { id =  Math.random().toString(36).slice(2); fetchStartEvent =  CustomEvent('fetchStart', {  4 detail: {  5 id,  6 url,  7 method: options.method || 'GET',  8 },  9 });  10 EventBus.dispatchEvent(fetchStartEvent);  11  window.fetch(url, options)  12 .then((response) => { fetchEndEvent =  CustomEvent('fetchEnd', {  13 detail: {  14 id,  15 url,  16 method: options.method || 'GET',  17 },  18 });  19 EventBus.dispatchEvent(fetchEndEvent);  20  response; 21 });  22 }; 23  24   fetch; |

在服务端实现数据存储和展⽰ ，可以使⽤图表等⽅式展⽰请求耗

8. ⼤⽂件上传了解多少 ⼤⽂件分⽚上传

如果太⼤的⽂件 ，⽐如⼀个视频1g 2g那么⼤ ，直接采⽤上⾯的栗⼦中的⽅法上传可能会出链接现超时 的情况 ，⽽且也会超过服务端允许上传⽂件的⼤⼩限制 ，所以解决这个问题我们可以将⽂件进⾏分⽚ 上传 ，每次只上传很⼩的⼀部分 ⽐如2M。

Blob 它表⽰原始数据, 也就是⼆进制数据 ，同时提供了对数据截取的⽅法 slice ,⽽ File 继承

了 Blob 的功能 ，所以可以直接使⽤此⽅法对数据进⾏分段截图。 过程如下：

• 把⼤⽂件进⾏分段 ⽐如2M ，发送到服务器携带⼀个标志 ，暂时⽤当前的时间戳 ，⽤于标识⼀个完 整的⽂件

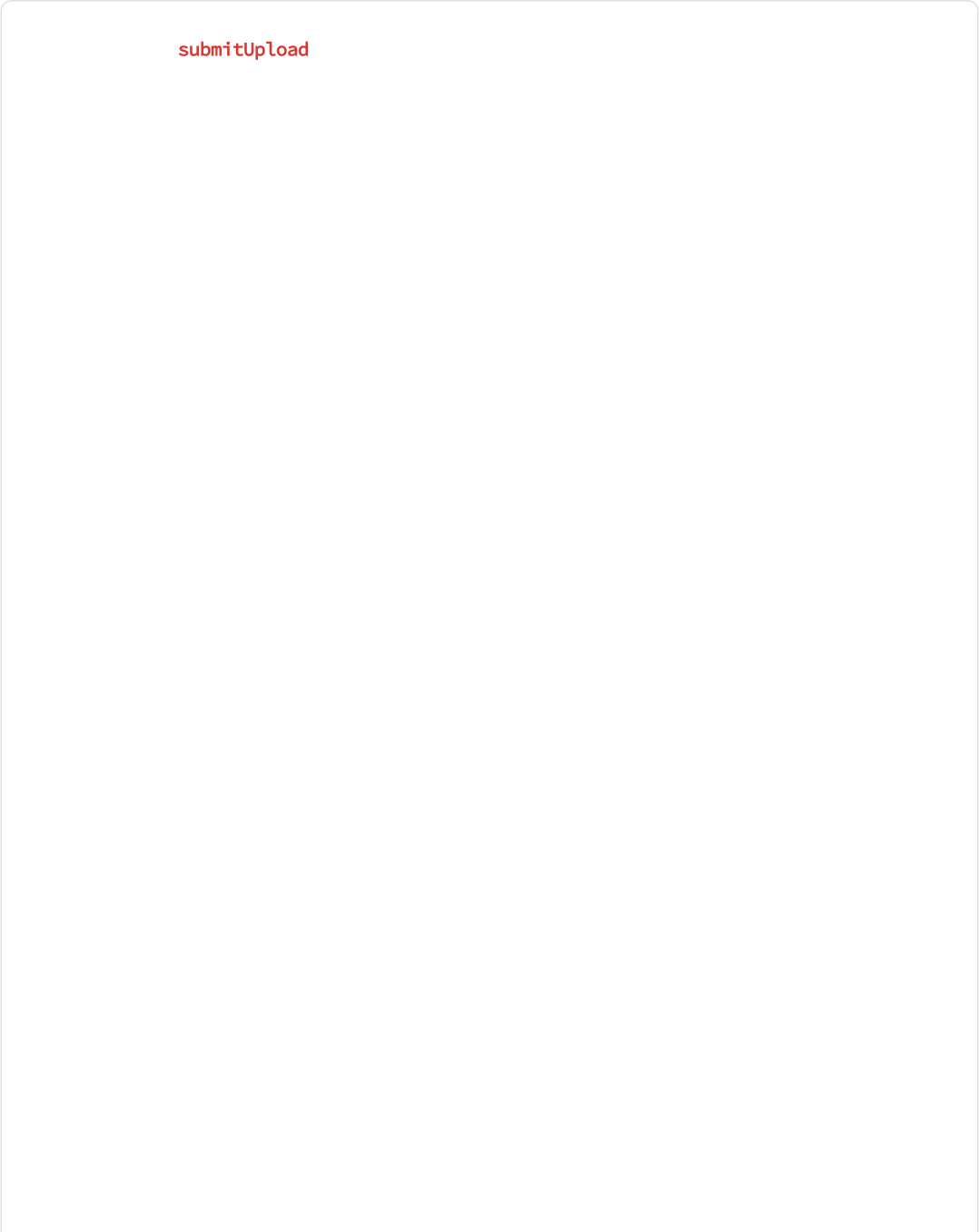
• 服务端保存各段⽂件

• 浏览器端所有分⽚上传完成 ，发送给服务端⼀个合并⽂件的请求

• 服务端根据⽂件标识、类型、各分⽚顺序进⾏⽂件合并

• 删除分⽚⽂件

客⼾端 JS 代码实现如下



1  () { chunkSize = 210241024;*//分⽚⼤⼩* *2M* *file =*

*document.getElementById('f1').files[0];* *chunks = [], //保存分⽚数据*

2 token = (+ Date()),*//时间戳*

3 name = file.name, chunkCount = 0, sendChunkCount = 0;

4 *//拆分⽂件* *像操作字符串⼀样* *(file.size > chunkSize) {//拆分⽂件* *start = 0, end = 0;* *(true) {*

5 end += chunkSize; blob = file.slice(start, end);

6 start += chunkSize;

7 *//截取的数据为空* *则结束* *(!blob.size) {//拆分结束**;* 8 }

9 chunks.push(blob);*//保存分段数据*

10 }

11 }  {

12 chunks.push(file.slice(0));

13 }

14 chunkCount = chunks.length;*//分⽚的个数//没有做并发限制，较⼤⽂件导致并发过多，tcp 链接被占光* *，需要做下并发控制，⽐如只有4个在请求在发送* *(* *i = 0; i < chunkCount; i++) {* *fd =*  *FormData(); //构造FormData对象*

15 fd.append('token', token);

16 fd.append('f1', chunks[i]);

17 fd.append('index', i);

18 xhrSend(fd,  () {

19 sendChunkCount += 1; (sendChunkCount === chunkCount) {*//上传完成，发送合并*

*请求console.log('上传完成，发送合并请求');* *formD =*  *FormData();*

20 formD.append('type', 'merge');

21 formD.append('token', token);

22 formD.append('chunkCount', chunkCount);

23 formD.append('filename', name);

24 xhrSend(formD);

25 }

26 });

27 }

28 }

29

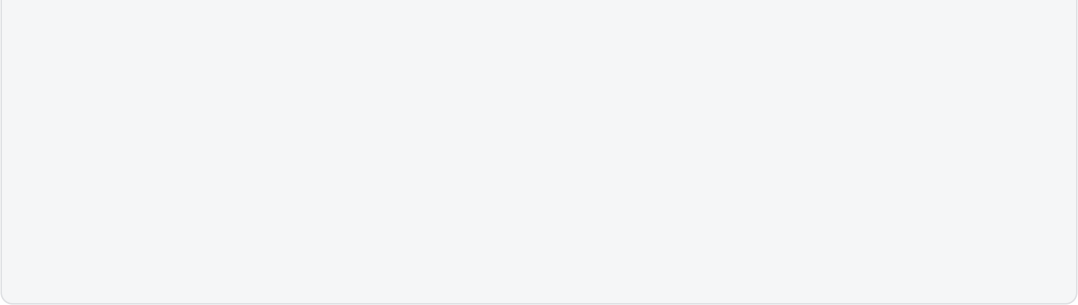
30  (fd, cb) {

31  xhr =  XMLHttpRequest(); *//创建对象*

32 xhr.open('POST', 'http://localhost:8100/', true);

33 xhr.onreadystatechange =  () {console.log('state change',

xhr.readyState); (xhr.readyState == 4) {console.log(xhr.responseText);



34 cb && cb();

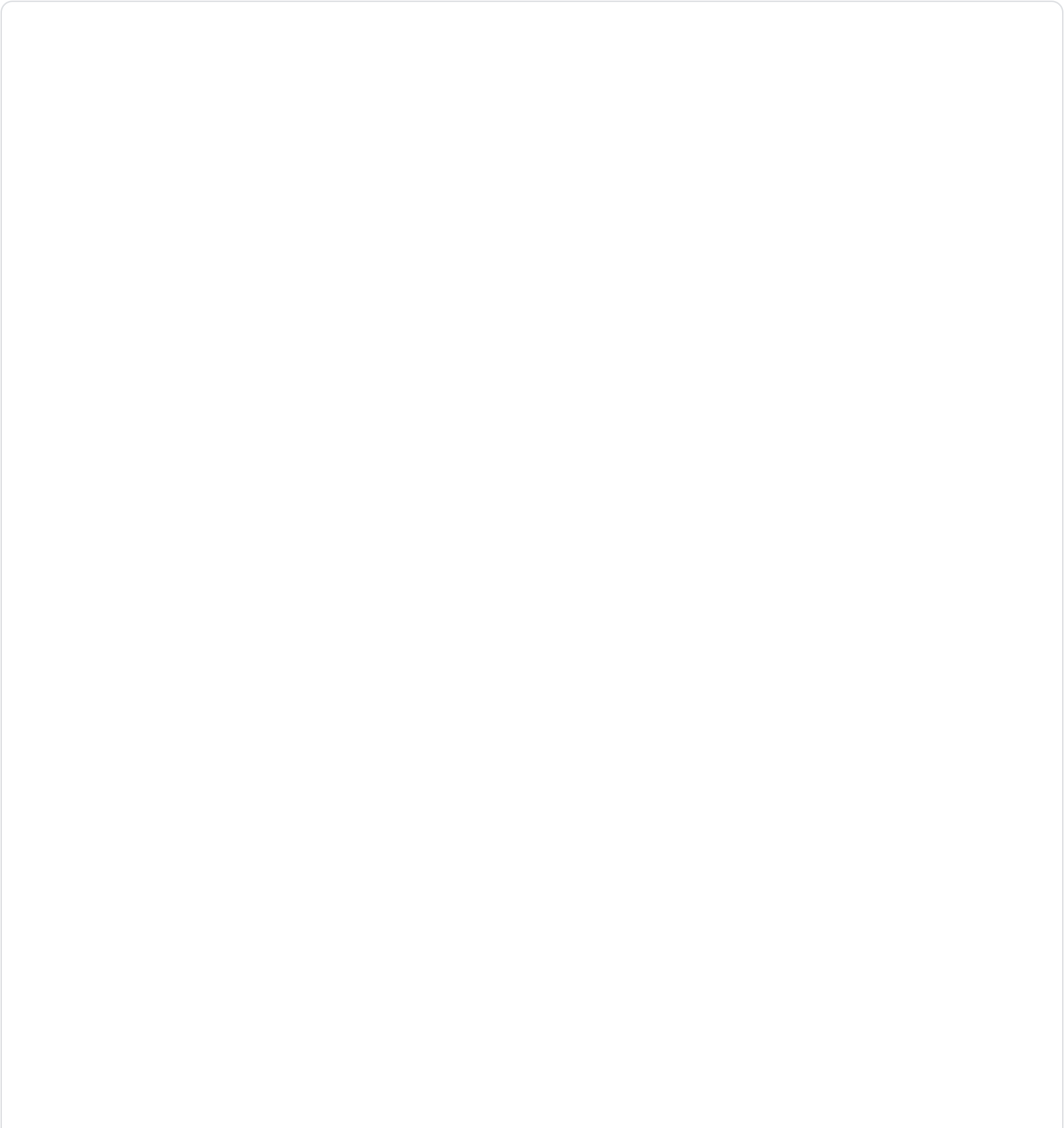
35 }

36 }

37 xhr.send(fd);*//发送* 38 }

39

40 *//绑定提交事件document.getElementById('btn-submit').addEventListener('click', submitUpload);*

服务端 node 实现代码如下： 合并⽂件这⾥使⽤ stream pipe实现 ，这样更节省内存 ，边读边写⼊， 占⽤内存更⼩ ，效率更⾼ ，代码⻅fnMergeFile⽅法。

1 *//⼆次处理⽂件，修改名称*

2 app.use((ctx) => { body = ctx.request.body; files = ctx.request.files ?

ctx.request.files.f1 : [];*//得到上传⽂件的数组* *result = [];* *fileToken =*

*ctx.request.body.token;// ⽂件标识* *fileIndex = ctx.request.body.index;//⽂件顺* *序* *(files && !Array.isArray(files)){//单⽂件上传容错*

3 files = [files];

4 }

5 files && files.forEach(item => { path = item.path; fname = item.name;*// 原⽂件名称* *nextPath = path.slice(0, path.lastIndexOf('/') + 1) + fileIndex + '-'fileToken;* *(item.size > 0 && path) {//得到扩展名* *extArr =*

*fname.split('. ');* *ext = extArr[extArr.length - 1];//var nextPath = path + '. ' + ext;//重命名⽂件*

6 fs.renameSync(path, nextPath);

7 result.push(uploadHost + nextPath.slice(nextPath.lastIndexOf('/') + 1));

8 }

9 });

10  (body.type === 'merge') {*//合并分⽚⽂件* *filename = body.filename,*

11 chunkCount = body.chunkCount,

12 folder = path.resolve(\_\_dirname, '../static/uploads') + '/';

13  writeStream = fs.createWriteStream(

14 ${folder}${filename}

15 );

16  cindex = 0;

17 *//合并⽂件* *(){* *fname =*

*${folder}${cindex}-${fileToken};* *readStream = fs.createReadStream(fname);*

18 readStream.pipe(writeStream, { end: false });

19 readStream.on("end",  () {

20 fs.unlink(fname,  (err) { (err) { err;

21 }

22 }); (cindex + 1 < chunkCount) {

23 cindex += 1;

24 fnMergeFile();

25 }

26 });

27 }

28 fnMergeFile();

29 ctx.body = 'merge ok 200';

30 }

31 });

⼤⽂件上传断点续传

在上⾯我们实现了⽂件分⽚上传和最终的合并 ，现在要做的就是如何检测这些分⽚ ，不再重新上传即

可。 这⾥我们可以在本地进⾏保存已上传成功的分⽚ ，重新上传的时候使⽤

|  |
| --- |
| spark-md5 |

来⽣成⽂件

hash ，区分此⽂件是否已上传。

• 为每个分段⽣成 hash 值 ，使⽤

|  |
| --- |
| spark-md5 |

库

• 将上传成功的分段信息保存到本地

• 重新上传时 ，进⾏和本地分段 hash 值的对⽐ ，如果相同的话则跳过 ，继续下⼀个分段的上传

⽅案⼀： 保存在本地

|  |
| --- |
| indexDB localStorage |

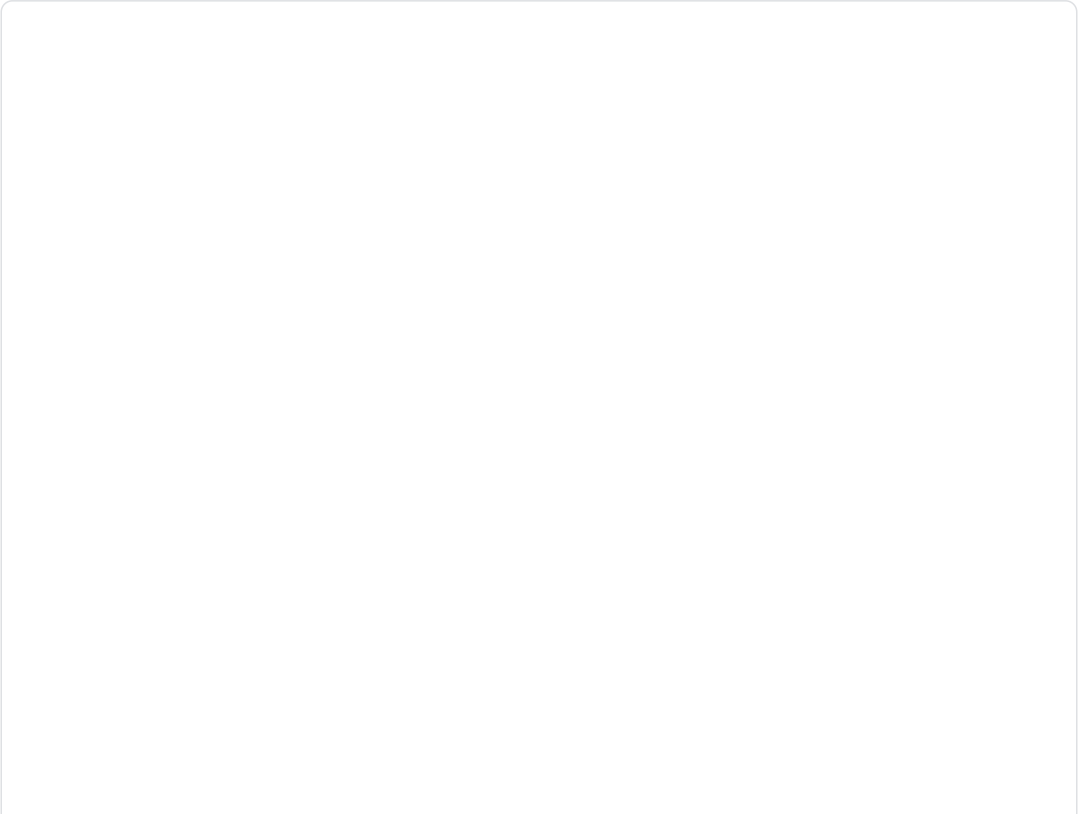
/

等地⽅ ， 推荐使⽤

|  |
| --- |
| localForage |

这个库

|  |
| --- |
| npm install localforage |

客⼾端 JS 代码：

1 *//获得本地缓存的数据* *(){* *JSON.parse(localforage.getItem(saveChunkKey) || "{}");*

2 } 3

4 *//写⼊缓存* *(index) {* *obj = getUploadedFromStorage();*

5 obj[index] = true;

6 localforage.setItem(saveChunkKey, JSON.stringify(obj));

7 }

8

9 *//分段对⽐* *uploadedInfo = getUploadedFromStorage();//获得已上传的分段信息*

*(* *i = 0; i < chunkCount; i++) {console.log('index', i, uploadedInfo[i] ? '已* *上传过' : '未上传');*

10  (uploadedInfo[i]) {*//对⽐分段*

11 sendChunkCount = i + 1;*//记录已上传的索引**;//如果已上传则跳过*

12 } fd =  FormData(); *//构造FormData对象*

13 fd.append('token', token);

14 fd.append('f1', chunks[i]);

15 fd.append('index', i);

16 (  (index) {

17 xhrSend(fd,  () {

18 sendChunkCount += 1;*//将成功信息保存到本地*

19 setUploadedToStorage(index); (sendChunkCount === chunkCount)

{console.log('上传完成 ，发送合并请求'); formD =  FormData();

20 formD.append('type', 'merge');

21 formD.append('token', token);

22 formD.append('chunkCount', chunkCount);

23 formD.append('filename', name);

24 xhrSend(formD);

25 }

26 });

27 })(i);

28 }

⽅案2：服务端⽤于保存分⽚坐标信息 ， 返回给前端

需要服务端添加⼀个接⼝只是服务端需要增加⼀个接⼝ 。 基于上⾯⼀个栗⼦进⾏改进 ，服务端已保存 了部分⽚段 ，客⼾端上传前需要从服务端获取已上传的分⽚信息（上⾯是保存在了本地浏览器），本 地对⽐每个分⽚的 hash 值 ，跳过已上传的部分 ，只传未上传的分⽚。

⽅法1是从本地获取分⽚信息,这⾥只需要将此⽅法的能⼒改为从服务端获取分⽚信息就⾏了。

9. H5 如何解决移动端适配问题

移动端适配问题是指如何让⽹⻚在不同的移动设备上显⽰效果相同。下⾯是⼀些常⻅的 H5 移动端适配 ⽅案：

1. 使⽤ viewport 标签

通过设置 viewport 标签的 meta 属性 ，来控制⻚⾯的缩放⽐例和宽度 ，以适配不同的设备。例如：

|  |
| --- |
| 1 <meta name="viewport" content="width=device-width, initial-scale=1.0" |

其中

|  |
| --- |
| width=device-width |

表⽰设置 viewport 的宽度为设备宽度，

|  |
| --- |
| initial-scale=1 .0 |

表

⽰初始缩放⽐例为 1。

1. 使⽤ CSS3 的媒体查询

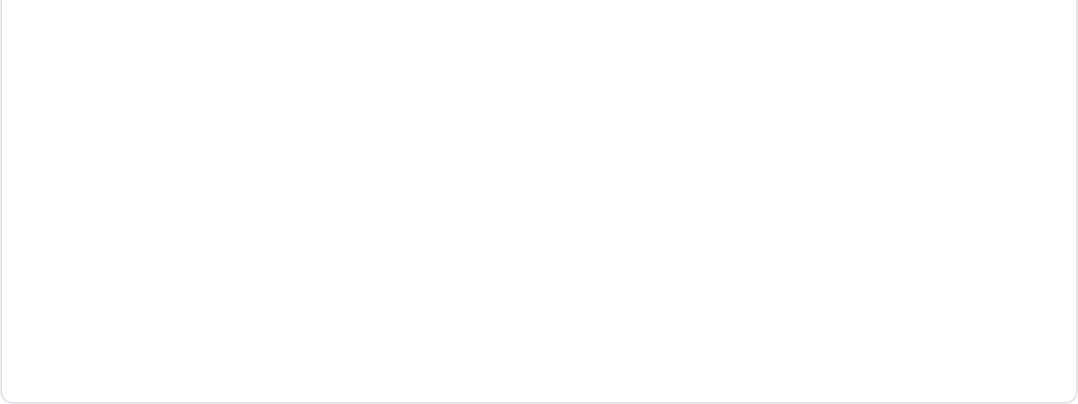
通过 CSS3 的媒体查询 ，根据不同的设备宽度设置不同的样式 ，以适配不同的设备。例如：

|  |
| --- |
| 1 arduinoCopy code@media screen and (max-width: 640px) {*/\* 样式* *\*/* 2 } |

其中

|  |
| --- |
| max-width |

表⽰最⼤宽度 ， 当屏幕宽度⼩于等于 640px 时 ，应⽤这些样式。

1. 使⽤ rem 单位

通过将 px转化为 rem 单位 ，根据不同的设备字体⼤⼩设置不同的样式 ，以适配不同的设备。例如：

|  |
| --- |
| 1 html {font-size: 16px;  2 }@ screen and (max-width: 640px) {html {font-size: 14px;}  3 }  4  5 div {width: 10rem;  6 } |

其中

|  |
| --- |
| font-size : 16px |

表⽰将⽹⻚的基准字体⼤⼩设置为 16px，

|  |
| --- |
| font-size : 14px |

表⽰在

屏幕宽度⼩于等于 640px 时将基准字体⼤⼩设置为 14px，

div

元素的

|  |
| --- |
| width : 10rem |

表⽰该元

素的宽度为 10 个基准字体⼤⼩。

1. 使⽤ flexible 布局⽅案

通过使⽤ flexible 布局⽅案 ，将 px转化为 rem 单位 ，并且动态计算根节点的字体⼤⼩ ，以适配不同的 设备。例如使⽤ lib-flexible 库：

|  |
| --- |
| 1 arduinoCopy code// index.html  2 <script src="https://cdn.bootcdn.net/ajax/libs/lib-flexible/0.3.4/flexible.js"  </script  3 // index.js  4 import 'lib-flexible/flexible.js' |

其中

flexible .js

会在⻚⾯加载时动态计算根节点的字体⼤⼩ ，并将 px转化为 rem 单位。在样

式中可以直接使⽤ px 单位 ，例如：

|  |
| --- |
| 1 div {width: 100px;height: 100px;  2 } |

这个 div 元素的⼤⼩会根据设备屏幕的宽度进⾏适配。

10. 站点⼀键换肤的实现⽅式有哪些？ ⽹站⼀键换肤实现⽅式有以下⼏种

1. 使⽤ CSS 变量：通过定义⼀些变量来控制颜⾊ 、字体等 ，然后在切换主题时动态修改这些变量的 值。

2. 使⽤ class 切换：在 HTML 的根元素上添加不同的 class 名称 ，每个 class 名称对应不同的主题样 式 ，在切换主题时切换根元素的 class 名称即可。

3. 使⽤ JavaScript 切换：使⽤ JavaScript 动态修改⻚⾯的样式 ，如修改元素的背景颜⾊ 、字体颜⾊ 等。

4. 使⽤ Less/Sass等 CSS 预处理器：通过预处理器提供的变量、 函数等功能来实现主题切换。

需要注意的是 ，⽆论采⽤哪种⽅式实现 ，都需要在设计⻚⾯样式时尽量遵循⼀些规范 ，如不使⽤绝对 的像素值 ，使⽤相对单位等 ，以便更好地适应不同的屏幕⼤⼩和分辨率。

以 less 举例 ， 详细讲述⼀下具体操作流程

通过 Less 实现⽹⻚换肤可以使⽤ CSS 变量和 Less 变量。CSS 变量的语法如下：

|  |
| --- |
| 1 :root {--primary-color: #007bff;  2 }.btn {background-color: var(--primary-color);  3 } |

⽽ Less 变量则是通过 Less 预编译器提供的变量语法来实现的 ，如下所⽰：

|  |
| --- |
| 1 lessCopy code@primary-color: #007bff; 2  3 .btn {background-color: @primary-color;  4 } |

通过 Less变量来实现⽹⻚换肤的⽅式可以在运⾏时使⽤ JavaScript 来修改 Less 变量的值 ，从⽽实现 换肤效果。具体步骤如下：

1. 使⽤ Less 预编译器来编译 Less⽂件为 CSS ⽂件。

2. 在 HTML ⽂件中引⼊编译后的 CSS ⽂件。

3. 在 JavaScript 中动态修改 Less 变量的值。

4. 使⽤ JavaScript 将新的 Less 变量值注⼊到编译后的 CSS ⽂件中。

5. 将注⼊后的 CSS 样式应⽤到⻚⾯上。

以下是⼀段实现通过 Less 变量来实现⽹⻚换肤的⽰例代码：

|  |
| --- |
| 1 *// base.less ⽂件*@primary-color: #007bff;  2  3 .btn {background-color: @primary-color;  4 }  5  6 *// dark.less ⽂件*@primary-color: #343a40; |

|  |
| --- |
| 1 *<!-- index.html ⽂件* *-->*<html<head<meta charset="UTF-8"<title⽹⻚ 换肤⽰例</title<link rel="stylesheet/less" type="text/css" href="base.less"<link  rel="stylesheet/less" type="text/css" href="dark.less"</head<body<button class="btn"按钮</button<script src="less.min.js"</script<script  2 function changeSkin() {// 修改 Less 变量的值  3 less.modifyVars({'@primary-color': '#28a745' 4 }).(() => {console.log('换肤成功');  5 }).  (() => {console.error('换肤失败');  6 });  7 }</script</body</html |

在上⾯的⽰例代码中 ，我们引⼊了两个 Less ⽂件 ，⼀个是

|  |
| --- |
| base .less |

, ⼀个是

|  |
| --- |
| dark .less |

。其

中

|  |
| --- |
| base .less |

定义了⼀些基础的样式 ，⽽

|  |
| --- |
| dark .less |

则是定义了⼀个暗⿊⾊的主题样式。在

JavaScript 中 ，我们使⽤

|  |
| --- |
| less .modifyVars |

⽅法来修改 Less 变量的值 ，从⽽实现了换肤的效

果。 当然 ，这只是⼀个简单的⽰例代码 ，实际的换肤功能还需要根据实际需求来进⾏设计和实现。

11. 如何实现⽹⻚加载进度条？ 监听静态资源加载情况

可以通过

|  |
| --- |
| window .performance |

对象来监听⻚⾯资源加载进度。该对象提供了各种⽅法来获取资

源加载的详细信息。

可以使⽤

|  |
| --- |
| performance .getEntries |

()

⽅法获取⻚⾯上所有的资源加载信息。可以使⽤该⽅法

来监测每个资源的加载状态 ，计算加载时间 ，并据此来实现⼀个资源加载进度条。 下⾯是⼀个简单的实现⽅式：

|  |
| --- |
| 1  resources = window.performance.getEntriesByType('resource'); 2  totalResources = resources.length;  3  loadedResources = 0;  4 resources.forEach((resource) => { (resource.initiatorType !== 'xmlhttprequest') {*// 排除* *AJAX 请求*  5 resource.onload = () => {  6 loadedResources++; progress = Math.round((loadedResources /  totalResources) \* 100);  7 updateProgress(progress);  8 };  9 }  10 });  11  12 (progress) {*// 更新进度条* 13 } |



该代码会遍历所有资源 ，并注册⼀个

|  |
| --- |
| onload |

事件处理函数。 当每个资源加载完成后 ，会更新

|  |
| --- |
| loadedResources |

变量 ，并计算当前的进度百分⽐ ，然后调⽤

updateProgress()

函数来更

新进度条。需要注意的是 ，这⾥排除了 AJAX 请求 ，因为它们不属于⻚⾯资源。 当所有资源加载完成后 ，⻚⾯就会完全加载。

实现进度条

⽹⻚加载进度条可以通过前端技术实现 ，⼀般的实现思路是通过监听浏览器的⻚⾯加载事件和资源加 载事件 ，来实时更新进度条的状态。下⾯介绍两种实现⽅式。

1. 使⽤原⽣进度条

在 HTML5 中提供了

|  |
| --- |
| progress |

元素 ，可以通过它来实现⼀个原⽣的进度条。

|  |
| --- |
| 1 <progress id="progressBar" value="0" max="100"</progress |

然后在 JavaScript 中 ，监听⻚⾯加载事件和资源加载事件 ，实时更新

|  |
| --- |
| progress |

元素的

|  |
| --- |
| value |

属性。

|  |
| --- |
| 1  progressBar = document.getElementById('progressBar'); 2  3 window.addEventListener('load', () => {  4 progressBar.value = 100;  5 }); 6  7 document.addEventListener('readystatechange', () => { progress =  Math.floor((document.readyState / 4) \* 100);  8 progressBar.value = progress;  9 }); |

2. 使⽤第三⽅库

使⽤第三⽅库可以更加⽅便地实现⽹⻚加载进度条 ，下⾯以

|  |
| --- |
| nprogress |

库为例：

1. 安装

|  |
| --- |
| nprogress |

库

|  |
| --- |
| 1 bashCopy codenpm install nprogress --save |

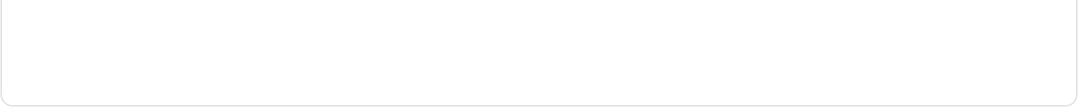
1. 在⻚⾯中引⼊

和

|  |
| --- |
| nprogress .css |

|  |
| --- |
| nprogress . s |

j



1 <link rel="stylesheet" href="/node\_modules/nprogress/nprogress.css"<script

src="/node\_modules/nprogress/nprogress.js"</script

1. 在 JavaScript 中初始化

|  |
| --- |
| nprogress |

并监听⻚⾯加载事件和资源加载事件

|  |
| --- |
| 1 *// 初始化* *nprogress*  2 NProgress.configure({ showSpinner: false }); 3  4 *// 监听⻚⾯加载事件window.addEventListener('load', () => {*  5 NProgress.done();  6 });  7  8 *// 监听资源加载事件document.addEventListener('readystatechange', () => {* *(document.readyState === 'interactive') {*  9 NProgress.start();  10 }  (document.readyState === 'complete') {  11 NProgress.done();  12 }  13 }); |

使⽤

|  |
| --- |
| nprogress |

可以⾃定义进度条的样式 ，同时也提供了更多的 API 供我们使⽤ ，⽐如说⼿动控制

进度条的显⽰和隐藏 ，以及⽀持 Promise 和 Ajax 请求的进度条等等。

12. 常⻅图⽚懒加载⽅式有哪些？【热度: 1,001】

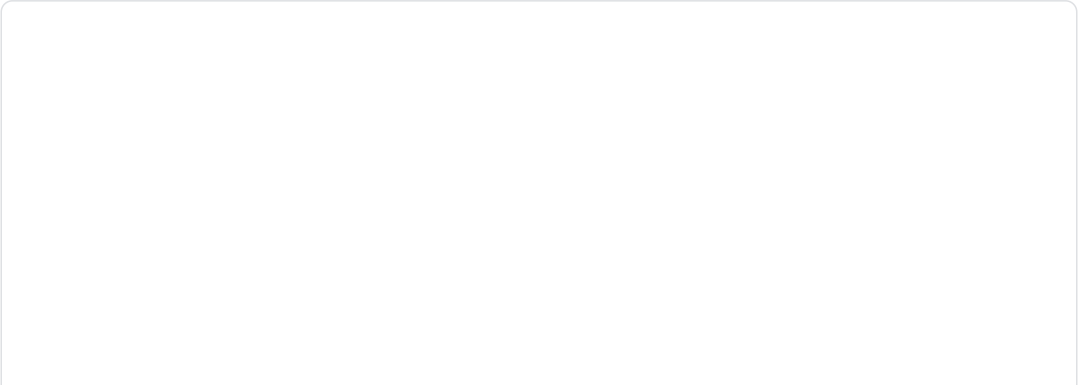
图⽚懒加载可以延迟图⽚的加载 ，只有当图⽚即将进⼊视⼝范围时才进⾏加载。这可以⼤⼤减轻⻚⾯ 的加载时间 ，并降低带宽消耗 ，提⾼了⽤⼾的体验。 以下是⼀些常⻅的实现⽅法：

1. Intersection Observer API

是⼀种⽤于异步检查⽂档中元素与视⼝叠加程度的API。可以将

|  |
| --- |
| Intersection Observer API |

其⽤于检测图⽚是否已经进⼊视⼝ ，并根据需要进⾏相应的处理。



1  observer =  IntersectionObserver(  (entries) {

2 entries.forEach(  (entry) { (entry.isIntersecting) { lazyImage

= entry.target;

3 lazyImage.src = lazyImage.dataset.src;

4 observer.unobserve(lazyImage);

5 }

6 });

7 });

8

9  lazyImages = [...document.querySelectorAll(".lazy")];

10 lazyImages.forEach(  (image) {

11 observer.observe(image);

12 });

1. ⾃定义监听器

或者 ，可以通过⾃定义监听器来实现懒加载。其中 ，应该避免在滚动事件处理程序中频繁进⾏图⽚加 载 ，因为这可能会影响性能。相反 ，使⽤⾃定义监听器只会在滚动停⽌时进⾏图⽚加载。

|  |
| --- |
| 1  () { images = document.querySelectorAll(".lazy"); scrollTop = window.pageYOffset;  2 images.forEach((img) => { (img.offsetTop < window.innerHeight + scrollTop)  {  3 img.src = img.dataset.src;  4 img.classList.remove("lazy");  5 }  6 });  7 }  8  9  lazyLoadThrottleTimeout;  10 document.addEventListener("scroll",  () { (lazyLoadThrottleTimeout) {  11 clearTimeout(lazyLoadThrottleTimeout);  12 }  13 lazyLoadThrottleTimeout = setTimeout(lazyLoad, 20);  14 }); |

在这个例⼦中 ，我们使⽤了

setTimeout()

函数来延迟图⽚的加载 ，以避免在滚动事件的频繁触发

中对性能的影响。

⽆论使⽤哪种⽅法 ，都需要为需要懒加载的图⽚设置占位符 ，并将未加载的图⽚路径保存在 data 属 性中 ，以便在需要时进⾏加载。这些占位符可以是简单的 div 或样式类 ，⽤于预留图⽚的空间 ，避免⻚ ⾯布局的混乱。

|  |
| --- |
| 1 *<!-- 占位符⽰例* *-->*<div class="lazy-placeholder" style="background-color: #ddd;height: 500px;"</div*<!-- 图⽚⽰例* *-->*<img class="lazy" data-  src="path/to/image.jpg" alt="预览图 " /> |

总体来说 ，图⽚懒加载是⼀种这很简单 ，但⾮常实⽤的优化技术 ，能够显著提⾼⽹⻚的性能和⽤⼾体 验。

13. cookie 构成部分有哪些 【热度: 598】

在 HTTP 协议中 ，cookie 是⼀种包含在请求和响应报⽂头中的数据 ，⽤于在客⼾端存储和读取信息。 cookie 是由服务器发送的 ，客⼾端可以使⽤浏览器 API将 cookie 存储在本地进⾏后续使⽤ 。

⼀个 cookie 通常由以下⼏个部分组成：

1. 名称：cookie 的名称（键） ，通常是⼀个字符串。

2. 值：cookie 的值 ，通常也是⼀个字符串。

3. 失效时间：cookie 失效的时间 ，过期时间通常存储在⼀个

|  |
| --- |
| expires |

属性中 ，以便浏览器⾃动清

除失效的 cookie。

4. 作⽤路径：cookie 的作⽤路径 ，只有在指定路径下的请求才会携带该 cookie。

5. 作⽤域：cookie 的作⽤域 ，指定了该 cookie 绑定的域名 ，可以使⽤ domain  属性来设置。

例如 ，以下是⼀个设置了名称为 "user"、值为 "john"、失效时间为2022 年 1 ⽉ 1 ⽇ ，并且作⽤于全 站的 cookie：

|  |
| --- |
| 1 Set-Cookie: user=john; expires=Sat, 01 Jan 2022 00:00:00 GMT; path=/; domain=example.com |

其中，

|  |
| --- |
| Set-Cookie |

是响应报⽂头 ，⽤于设置 cookie。在该响应报⽂中 ，将 cookie 数据设置为

"user=john" ，失效时间为 "2022年1⽉1⽇ " ，作⽤路径为全站 ，作⽤域为 "[example.com](http://example.com/)" 的域名。这 个 cookie 就会被存储在客⼾端 ，以便在以后的请求中发送给服务器。

14. 扫码登录实现⽅式 【热度: 734】

扫码登录的实现原理核⼼是基于⼀个中转站 ，该中转站通常由应⽤提供商提供 ，⽤于维护⼿机和PC之 间的会话状态。

整个扫码登录的流程如下：

1. ⽤⼾在PC端访问应⽤ ，并选择使⽤扫码登录⽅式。此时 ，应⽤⽣成⼀个随机的认证码 ，并将该认证 码通过⼆维码的形式显⽰在PC端的⻚⾯上。

2. ⽤⼾打开⼿机上的应⽤ ，并选择使⽤扫码登录⽅式。此时 ，应⽤会打开⼿机端的相机 ，⽤⼾可以对 着PC端的⼆维码进⾏扫描。

3. ⼀旦⽤⼾扫描了⼆维码 ，⼿机上的应⽤会向应⽤提供商的中转站发送⼀个请求 ，请求包含之前⽣成 的随机认证码和⼿机端的⼀个会话ID。

4. 中转站验证认证码和会话ID是否匹配 ，如果匹配成功 ，则该中转站将⽤⼾的⾝份信息发送给应⽤ ， 并创建⼀个PC端和⼿机端之间的会话状态。

5. 应⽤使⽤收到的⾝份信息对⽤⼾进⾏认证 ，并创建⼀个与该⽤⼾关联的会话状态。 同时 ，应⽤返回 ⼀个通过认证的响应给中转站。

6. 中转站将该响应返回给⼿机端的应⽤ ，并携带⼀个⽤于表⽰该会话的令牌 ，此时⼿机和PC之间的认 证流程就完成了。

7. 当⽤⼾在PC端进⾏其他操作时 ，应⽤将会话令牌附加在请求中 ，并通过中转站向⼿机端的应⽤发起 请求。⼿机端的应⽤使⽤会话令牌（也就是之前⽣成的令牌）来识别并验证会话状态 ，从⽽允许⽤ ⼾在PC端进⾏需要登录的操作。

15. DNS 协议了解多少 【热度: 712】

DNS 基本概念

DNS（ Domain Name System ，域名系统）是因特⽹上⽤于将主机名转换为 IP 地址的协议。 它是⼀个 分布式数据库系统 ，通过将主机名映射到 IP 地址来实现主机名解析 ，并使⽤⼾能够通过更容易识别的 主机名来访问互联⽹上的资源。

在使⽤ DNS 协议进⾏主机名解析时 ，系统⾸先查询本地 DNS 缓存。如果缓存中不存在结果 ，系统将 向本地 DNS 服务器发出请求 ，并逐级向上查找 ，直到找到权威 DNS 服务器并获得解析结果。在域名 解析的过程中 ，DNS 协议采⽤了分级命名空间的结构 ，不同的域名可以通过点分隔符分为多个级别，

例如

|  |
| --- |
| [www.example.com](http://www.example.com/) |

可以分为三个级别： www 、

|  |
| --- |
| example |

和 com 。

除了将域名映射到 IP 地址之外 ，DNS 协议还⽀持多种其他功能：

1. 逆向映射：将 IP 地址解析为域名。

2. 邮件服务器设置：⽀持邮件服务器的⾃动发现和设置。

3. 负载均衡： DNS 还可以实现简单的负载均衡 ，通过将相同 IP 地址的主机名映射到不同的IP 地址来 分散负载。

4. 安全： DNSSEC（ DNS Security Extensions ，DNS 安全扩展） 可以提供对域名解析的认证和完整 性。

如何加快 DNS 的解析？

有以下⼏种⽅法可以加快 DNS 的解析：

1. 使⽤⾼速 DNS 服务器：默认情况下 ，⽹络服务提供商（ ISP） 为其⽤⼾提供 DNS 服务器。但是， 这些服务器不⼀定是最快的 ，有时会出现瓶颈。如果您想加快 DNS 解析 ，请尝试使⽤其他⾼速 DNS 服务器 ，例如 Google 的公共 DNS 服务器或 OpenDNS。

2. 缓存 DNS 记录：在本地计算机上缓存 DNS 记录可以⼤⼤加快应⽤程序的响应。 当您访问特定的⽹ 站时 ，计算机会⾃动缓存该⽹站的 DNS 记录。如果您再次访问该⽹站 ，则计算机将使⽤缓存的

DNS 记录。

3. 减少 DNS 查找： 当您访问⼀个⽹站时 ，您的计算机将会查找该域名的 IP 地址。如果⽹站有很多域 名 ，则查找过程可能会变得⾮常缓慢。 因此 ，尽可能使⽤较少的域名可以减少 DNS 查找的数量，

并提⾼响应速度。

4. 使⽤ CDN：CDN（ 内容分发⽹络）是⼀种将内容存储在全球多个位置的系统。这些位置通常都有专 ⽤的 DNS 服务器 ，可以⼤⼤加快站点的加载速度。

5. 使⽤ DNS 缓存⼯具：⼀些辅助⼯具可以帮助您优化与 DNS 相关的设置 ，例如免费的 DNS Jumper 软件和 Namebench ⼯具 ，它们可以测试您的 DNS 响应时间并为您推荐最佳配置。

通过使⽤⾼速 DNS 服务器、缓存 DNS 记录、减少 DNS 查找、使⽤ CDN 和 DNS 缓存⼯具等⽅法 ，可 以显著提⾼ DNS 解析速度 ，从⽽加快应⽤程序响应时间。

16. 函数式编程了解多少？ 【热度: 1,789】

函数式编程的核⼼概念

函数式编程是⼀种编程范式 ，它将程序看做是⼀系列函数的组合 ，函数是应⽤的基础单位。 函数式编 程主要有以下核⼼概念：

1. 纯函数： 函数的输出只取决于输⼊ ，没有任何副作⽤ ，不会修改外部变量或状态 ，所以对于同样的 输⼊ ，永远返回同样的输出值。 因此 ，纯函数可以有效地避免副作⽤和竞态条件等问题 ，使得代码 更加可靠、易于调试和测试。

2. 不可变性：在函数式编程中 ，数据通常是不可变的 ，即不允许在内部进⾏修改。这样可以避免副作 ⽤的发⽣ ，提⾼代码可靠性。

3. 函数组合： 函数可以组合成复杂的函数 ，从⽽减少重复代码的产⽣。

4. ⾼阶函数：⾼阶函数是指可以接收其他函数作为参数 ，也可以返回函数的函数。例如 ，函数柯⾥化 和函数的组合就是⾼阶函数的应⽤场景。

5. 惰性计算：指在必要的时候才计算（执⾏） 函数 ，⽽不是在每个可能的执⾏路径上都执⾏ ，从⽽提 ⾼性能。

函数式编程的核⼼概念是将函数作为基本构建块来组合构建程序 ，通过纯函数、不可变性、 函数组 合、⾼阶函数和惰性计算等概念来实现代码的简洁性、可读性和可维护性 ，以及⾼效的性能运⾏。

函数式编程的优势 函数式编程有以下优势：

1. 易于理解和维护： 函数式编程强调数据不变性和纯函数概念 ，可以提⾼代码的可读性和可维护性， 因为它避免了按照顺序对变量进⾏修改 ，并强调函数⾏为的确定性。

2. 更少的 bug： 由于函数式编程强调纯函数的概念 ，它可以消除由于副作⽤引起的bug。 因为纯函数 不会修改外部状态或数据结构 ，只是将输⼊转换为输出。这么做有助于保持代码更加可靠。

3. 更好的可测试性： 由于纯函数不具有副作⽤ ，它更容易测试 ，因为测试数据是预测性的。

4. 更少的重构： 函数式编程使⽤函数组合和柯⾥化等⽅法来简化代码。 它将⼤型问题分解为微⼩问 题 ，从⽽减少了代码重构的需要。

5. 避免并发问题： 由于函数式编程强调不变性和纯函数的概念 ，这使得并发问题变得更简单。纯函数 允许并⾏运⾏ ，因此 ， 当程序在不同的线程上执⾏时 ，它更容易保持同步。

6. 代码复⽤： 由于函数是基本构建块 ，并且可以组合成更⾼级别的功能块 ，使⽤函数式编程可以更⼤ 程度上推崇代码复⽤ ，减少代码冗余。

函数式编程通过强调纯函数、不可变数据结构和函数组合等概念 ，可以提⾼代码可读性和可维护性，

降低程序bug出现的⻛险 ，更容易测试 ，并且更容易将问题分解为更容易处理的⼩部分 ，更好地应对并 发和可扩展性。

17. 前端⽔印了解多少？【热度: 641】

明⽔印和暗⽔印的区别

前端⽔印可以分为明⽔印和暗⽔印两种类型。 它们的区别如下：

1. 明⽔印：明⽔印是通过在⽂本或图像上覆盖另⼀层图像或⽂字来实现的。这种⽔印会明显地出现在 ⻚⾯上 ，可以⽤来显⽰版权信息或其他相关信息。

2. 暗⽔印：暗⽔印是指在⽂本或图像中隐藏相关信息的⼀种技术。这种⽔印不会直接出现在⻚⾯上 ， 只有在特殊的程序或⼯具下才能被检测到。暗⽔印通常⽤于保护敏感信息以及追踪⽹⻚内容的来源 和版本。

添加明⽔印⼿段有哪些

可以参考这个⽂档： [https://zhuanlan.zhihu.com/p/374734095](https://gitee.com/link?target=https%3A%2F%2Fzhuanlan.zhihu.com%2Fp%2F374734095) 总计⼀下：

1. 重复的dom元素覆盖实现： 在⻚⾯上覆盖⼀个position:fixed的div盒⼦ ，盒⼦透明度设置较低 ，设 置pointer-events: none;样式实现点击穿透 ，在这个盒⼦内通过js循环⽣成⼩的⽔印div ，每个⽔印 div内展⽰⼀个要显⽰的⽔印内容

2. canvas输出背景图： 绘制出⼀个⽔印区域 ，将这个⽔印通过toDataURL⽅法输出为⼀个图⽚ ，将 这个图⽚设置为盒⼦的背景图 ，通过backgroud-repeat： repeat；样式实现填满整个屏幕的效果。

3. svg实现背景图： 与canvas⽣成背景图的⽅法类似 ，只不过是⽣成背景图的⽅法换成了通过svg⽣ 成

4. 图⽚加⽔印

css 添加⽔印的⽅式 ， 如何防⽌⽤⼾删除对应的css ， 从⽽达到去除⽔印的⽬的 使⽤ CSS 添加⽔印的⽅式本⾝并不能完全防⽌⽤⼾删除对应的 CSS 样式 ，从⽽删除⽔印。但是 ，可以 采取⼀些措施来增加删除难度 ，提⾼⽔印的防伪能⼒ 。以下是⼀些常⻅的⽅法：

1. 调⽤外部CSS⽂件：将⽔印样式单独设置在⼀个CSS⽂件内 ，并通过外链的⽅式在⽹站中调⽤ ，可 以避免⽤⼾通过编辑⻚⾯HTML⽂件或内嵌样式表的⽅式删除⽔印。

2. 设置样式为 !important：在CSS样式中使⽤ !important 标记可以避免被覆盖。但是 ，这种⽅式会 影响⽹⻚的可读性 ，需慎重考虑。

3. 添加⾃定义类名：通过在CSS样式中加⼊⾃定义的class类名 ，可以防⽌⽤⼾直接删掉该类名 ，进⽽ 删除⽔印。但是 ，⽤⼾也可以通过重新定义该类名样式来替换⽔印。

4. 将⽔印样式应⽤到多个元素上：将⽔印样式应⽤到多个元素上 ，可以使得⽤⼾删除⽔印较为困难。 例如 ，在⽹站的多个位置都加上"Power by XXX"的⽔印样式。

5. 使⽤JavaScript动态⽣成CSS样式：可以监听挂载⽔印样式的dom 节点 ， 如果⽤⼾改变了该dom , 重新⽣成 对应的⽔印挂载上去即可。 这种⽅法可通过JS动态⽣成CSS样式 ，从⽽避免⽤⼾直接在 ⽹⻚源⽂件中删除CSS代码。但需要注意的是 ，这种⽅案会稍稍加重⽹⻚的加载速度 ，需要合理权 衡。

6. 混淆CSS代码：通过多次重复使⽤同⼀样式 ，或者采⽤CSS压缩等混淆⼿段 ，可以使CSS样式表变得 复杂难懂 ，增加⽔印被删除的难度。

7. 采⽤图⽚⽔印的⽅式：将⽔印转化为⼀个透明的PNG图⽚ ，然后将其作为⽹⻚的背景图⽚ ，可以更 有效地防⽌⽔印被删除。

8. 使⽤SVG图形：可以将⽔印作为SVG图形嵌⼊到⽹⻚中进⾏展⽰ 。由于SVG的⽮量性质 ，这种⽅式 可以保证⽔印在缩放或旋转后的清晰度 ，同时也增加了删除难度。

暗⽔印是如何把⽔印信息隐藏起来的

暗⽔印的基本原理是在原始数据（如⽂本、 图像等） 中嵌⼊信息 ，从⽽实现版权保护和溯源追踪等功 能。暗⽔印把信息隐藏在源数据中 ，使得⼈眼难以察觉 ，同时对源数据的影响尽可能⼩ ，保持其⾃⾝ 的特征。

⼀般来说 ，暗⽔印算法主要包括以下⼏个步骤：

1. ⽔印信息处理：将待嵌⼊的信息经过处理和加密后 ，转化为⼆进制数据。

2. 源数据处理：遍历源数据中的像素或⼆进制数据 ，根据特定规则对其进⾏调整 ，以此腾出空间插⼊ ⽔印⼆进制数据。

3. 嵌⼊⽔印：将⽔印⼆进制数据插⼊到源数据中的指定位置 ，以某种⽅式嵌⼊到源数据之中。

4. 提取⽔印：在使⽤暗⽔印的过程中 ，需要从带⽔印的数据中提取出隐藏的⽔印信息。提取⽔印需要 使⽤特定的解密算法和提取密钥。

暗⽔印的⼀个关键问题是在嵌⼊⽔印的过程中 ，要保证⽔印对源数据的伤害尽可能的⼩ ，同时嵌⼊⽔ 印后数据的分布、统计性质等不应发⽣明显变化 ，以更好地保持数据的质量和可视效果。

18. 什么是领域模型 【热度: 1,092】 什么是领域模型

领域模型是软件开发中⽤于描述领域（业务）概念和规则的⼀种建模技术。 它通过定义实体、值对 象、关联关系、⾏为等元素 ，抽象出领域的核⼼概念和业务规则 ，帮助开发⼈员理解和设计软件系 统。

以下是领域模型中常⻅的⼀些元素：

1. 实体（ Entity） ：实体是领域模型中具有唯⼀标识的对象 ，通常代表领域中的具体事物或业务对 象。实体具有属性和⾏为 ，并且可以通过其标识进⾏唯⼀标识和识别。

2. 值对象（Value Object） ：值对象是没有唯⼀标识的对象 ，通常⽤于表⽰没有明确⽣命周期的属性 集合。值对象的相等性通常基于其属性值 ，⽽不是标识。例如 ， ⽇期、 时间、货币等都可以作为值 对象。

3. 关联关系（Association） ：关联关系描述了不同实体之间的关系和连接。关联关系可以是⼀对

⼀ 、⼀对多、多对多等不同类型。关联关系可以带有⽅向和导航属性 ，⽤于表⽰实体之间的关联和 导航。

4. 聚合（Aggregation） ：聚合是⼀种特殊的关联关系 ，表⽰包含关系 ，即⼀个实体包含其他实体。 聚合关系是⼀种强关联 ，被包含实体的⽣命周期受到包含实体的控制。

5. 领域事件（ Domain Event） ：领域事件表⽰领域中发⽣的具体事件或状态变化。 它可以作为触发 业务逻辑的信号 ，通常⽤于解耦和处理领域中的复杂业务流程。

6. 聚合根（Aggregate Root） ：聚合根是聚合中的根实体 ，它代表整个聚合的⼀致性边界。通过聚合 根 ，可以对整个聚合进⾏操作和维护。

7. 领域服务（ Domain Service） ：领域服务是⼀种封装了领域逻辑的服务 ，⽤于处理领域中的复杂业 务操作或跨实体的操作。 它通常与具体实体⽆关 ，提供⼀些⽆状态的操作。

通过建⽴领域模型 ，开发⼈员可以更好地理解和表达领域的业务需求和规则 ，从⽽指导软件系统的设 计和实现。领域模型可以作为开发团队之间沟通的⼯具 ，也可以⽤于⽣成代码、进⾏⾃动化测试等。

前端系统应该如何划分领域模型

在前端系统中划分领域模型的⽅式可以根据具体业务需求和系统复杂性进⾏灵活调整。 以下是⼀些常 ⻅的划分领域模型的⽅式：

1. 模块划分：将前端系统按照模块进⾏划分 ，每个模块对应⼀个领域模型。模块可以根据功能、业务 领域或者⻚⾯进⾏划分。每个模块可以有⾃⼰的实体、值对象、关联关系和业务逻辑。

2. ⻚⾯划分：将前端系统按照⻚⾯进⾏划分 ，每个⻚⾯对应⼀个领域模型。每个⻚⾯可以有⾃⼰的实 体、值对象和关联关系 ，以及与⻚⾯相关的业务逻辑。

3. 组件划分：将前端系统按照组件进⾏划分 ，每个组件对应⼀个领域模型。每个组件可以有⾃⼰的实 体、值对象和关联关系 ，以及与组件相关的业务逻辑。组件可以是⻚⾯级别的 ，也可以是更细粒度 的功能组件。

4. 功能划分：将前端系统按照功能进⾏划分 ，每个功能对应⼀个领域模型。功能可以是⽤⼾操作的具 体功能模块 ，例如登录、注册、购物⻋等。每个功能可以有⾃⼰的实体、值对象和关联关系 ，以及 与功能相关的业务逻辑。

在划分领域模型时 ，需要根据具体业务的复杂性和团队的组织⽅式进⾏调整。重要的是识别系统中的 核⼼业务概念和规则 ，并将其抽象成适当的实体和值对象。 同时 ，要保持领域模型的聚合性和⼀致

性 ，避免出现过于庞⼤和紧耦合的领域模型。划分的领域模型应该易于理解、扩展和维护 ，以⽀持前 端系统的开发和演进。

19. ⼀ 直在 window 上⾯挂东西是否有什么⻛险

在前端开发中 ，将内容或应⽤程序运⾏在浏览器的全局window对象上可能会带来⼀些潜在的⻛险。 以下是⼀些需要注意的⻛险：

1. 命名冲突： window对象是浏览器的全局对象 ，它包含许多内置属性和⽅法。如果您在全局命名 空间中定义的变量或函数与现有的全局对象属性或⽅法发⽣冲突 ，可能会导致意外⾏为或错误。

2. 安全漏洞：在全局window对象上挂载的代码可以访问和修改全局的数据和功能。这可能导致安



全漏洞 ，特别是当这些操作被恶意利⽤时。攻击者可能通过篡改全局对象来窃取⽤⼾敏感信息或执 ⾏恶意代码。

3. 代码维护性：过多地依赖全局window对象可能导致代码的维护困难。全局状态的过度共享可能

导致代码变得难以理解和调试 ，尤其在⼤型应⽤程序中。 为了减轻这些⻛险 ，建议采⽤以下最佳实践：

1. 使⽤模块化开发：将代码模块化 ，避免对全局window对象的直接依赖。使⽤模块加载器（如ES

Modules、CommonJS、AMD）来管理模块之间的依赖关系 ，以减少全局命名冲突和代码冗余。

2. 使⽤严格模式：在JavaScript代码中使⽤严格模式 (

|  |
| --- |
| "use strict" |

) , 以启⽤更严格的语法检

查和错误处理。严格模式可以帮助捕获潜在的错误和不安全的⾏为。

3. 显式访问全局对象：如果确实需要访问全局

|  |
| --- |
| window |

对象的属性或⽅法 ，请使⽤显式访问⽅式，

如

、

|  |
| --- |
| window .localStorage |

|  |
| --- |
| window .setTimeout |

()

等。避免直接引⽤全局属性 ，以减少

冲突和误⽤的⻛险。

4. 谨慎处理第三⽅代码：在使⽤第三⽅库或框架时 ，注意审查其对全局window对象的使⽤⽅式。 确保库或框架的操作不会产⽣潜在的安全⻛险或全局命名冲突。

20. 深度 SEO 优化的⽅式有哪些 ， 从技术层⾯来说 深度 SEO 优化涉及到⼀些技术层⾯的优化策略 ，以下是⼀些常⻅的⽅式：

1. ⽹站结构优化：优化⽹站的结构 ，确保每个⻚⾯都可以被搜索引擎爬取和索引。使⽤合适的 HTML 标签和语义化的内容结构 ，使搜索引擎能够更好地理解⻚⾯的内容。

2. ⽹站速度优化：提升⽹站的加载速度对 SEO 很重要。通过压缩和合并 CSS 和 JavaScript ⽂件、优 化图像、使⽤浏览器缓存、使⽤ CDN（ 内容分发⽹络）等技术⼿段来减少⻚⾯加载时间。

3. ⻚⾯渲染优化：确保搜索引擎可以正常渲染和索引使⽤ JavaScript 技术构建的单⻚⾯应⽤ （SPA） 或动态⽣成的内容。使⽤服务端渲染（SSR） 或预渲染技术 ，确保搜索引擎能够获取到完整的⻚⾯ 内容。

4. URL 优化：使⽤短、描述性的 URL ，并使⽤关键词来优化 URL 结构。避免使⽤动态参数或过⻓的 URL。

5. 链接优化： 内部链接和外部链接都对 SEO 有影响。在⽹站内部设置相关性强的链接 ，使⻚⾯之间相 互连接。外部链接是获取更多外部⽹站链接指向⾃⼰⽹站的重要⼿段 ，可以通过内容创作和社交媒 体推⼴来获得更多⾼质量的外部链接。

6. Schema 标记：使⽤结构化数据标记（Schema Markup）来标识⽹⻚内容 ，帮助搜索引擎更好地 理解和展⽰⽹⻚信息。可以使⽤ JSON-LD、 Microdata 或 RDFa 等标记格式。

7. XML ⽹站地图：创建和提交 XML ⽹站地图 ，提供⽹站的结构和⻚⾯信息 ，帮助搜索引擎更好地索 引⽹站内容。

8. Robots.txt ⽂件：通过 Robots.txt ⽂件来指⽰搜索引擎哪些⻚⾯可以被爬取和索引 ，哪些⻚⾯不可 访问。

9. HTTPS 加密：使⽤ HTTPS 协议来加密⽹站通信 ，确保数据安全和⽤⼾隐私 ，同时搜索引擎更倾向 于收录和排名使⽤ HTTPS 的⽹站。

10. 移动友好性：优化⽹站在移动设备上的显⽰和⽤⼾体验 ，确保⽹站具备响应式设计或移动版⽹ 站 ，以及快速加载和友好的操作性。

这些是深度 SEO 优化的⼀些常⻅技术层⾯的策略 ，通过综合运⽤这些策略 ，可以提升⽹站的搜索引擎 可⻅性和排名。需要根据具体情况和搜索引擎的最佳

21. ⼩程序为什么会有两个线程

⼩程序之所以有两个线程 ，是为了实现⼩程序的⾼效运⾏和良好的⽤⼾体验。

1. 渲染线程（ UI 线程）：

渲染线程负责⼩程序界⾯的渲染和响应⽤⼾的交互。 它使⽤ WebView 进⾏⻚⾯渲染 ，包括解析和 绘制 DOM、布局、样式计算和渲染等操作。渲染线程是单线程的 ，所有的界⾯操作都在这个线程 中进⾏。

2. 逻辑线程（JS 线程）：

逻辑线程负责⼩程序的逻辑运算和数据处理。 它是基于 JavaScript 运⾏的 ，负责处理⽤⼾交互、业 务逻辑、数据请求、事件处理等操作。逻辑线程是独⽴于渲染线程的 ，可以并⾏处理多个任务 ，避 免阻塞界⾯的渲染和响应。

将界⾯渲染和逻辑运算分离成两个线程的设计有以下好处：

• 响应速度：逻辑线程和渲染线程分开 ，可以并⾏执⾏ ，提⾼了⼩程序的响应速度和⽤⼾体验。

• 防⽌阻塞：逻辑线程的运⾏不会阻塞渲染线程 ，避免了⻓时间的计算或数据处理导致界⾯卡顿或⽆ 响应的情况。

• 资源隔离：渲染线程和逻辑线程是独⽴的 ，它们有各⾃的资源和运⾏环境 ，可以避免相互⼲扰和影 响。

需要注意的是 ，⼩程序的渲染线程和逻辑线程之间通过微信客⼾端进⾏通信和交互。逻辑线程可以发 送请求给微信客⼾端 ，然后客⼾端将渲染指令发送给渲染线程进⾏界⾯渲染 ，同时渲染线程可以将⽤ ⼾的交互事件发送给逻辑线程进⾏处理。这种通信⽅式保证了渲染和逻辑的协同⼯作 ，实现了⼩程序 的正常运⾏。

⼩程序之所以有两个线程 ，是为了提⾼渲染速度、避免阻塞和资源隔离。渲染线程负责界⾯渲染 ，逻 辑线程负责业务逻辑和数据处理 ，两者通过微信客⼾端进⾏通信和交互 ，共同实现⼩程序的功能和性 能。

22. web 应⽤中如何对静态资源加载失败的场景做降级处理 【热度: 1,093】

在 Web 应⽤中 ，可以使⽤以下⽅法对静态资源加载进⾏降级处理 ，即在某个资源加载失败时使⽤备⽤ 的静态资源链接：

1. 使⽤多个 CDN 链接：在 HTML 中使⽤多个静态资源链接 ，按照优先级顺序加载 ，如果其中⼀个链 接加载失败 ，则尝试加载下⼀个链接。

|  |
| --- |
| 1 <script src="https://cdn1.example.com/script.js"</script<script  src="<https://cdn2.example.com/script.js>"</script<script  src="<https://cdn3.example.com/script.js>"</script |

在加载 JavaScript 脚本时 ，浏览器会按照给定的顺序尝试加载各个链接 ，如果某个链接加载失败 ，浏 览器会⾃动尝试加载下⼀个链接。

1. 使⽤备⽤资源路径：在 JavaScript 中使⽤备⽤的资源路径 ， 当主要的资源路径加载失败时 ，切换到 备⽤路径。

|  |
| --- |
| 1  script = document.createElement('script');  2 script.src = 'https://cdn.example.com/script.js';  3 script.onerror = () {*// 主要资源加载失败，切换到备⽤资源路径*  4 script.src = 'https://backup.example.com/script.js';  5 };  6 document.head.appendChild(script); |

在加载 JavaScript 脚本时 ，可以通过监听

|  |
| --- |
| one r ro r |

事件 ，在主要资源加载失败时切换到备⽤资源

路径 ，保证资源的可靠加载。

1. 使⽤动态加载和错误处理：使⽤ JavaScript 动态加载静态资源 ，并处理加载失败的情况。

• function loadScript(src, backupSrc) {return new Promise(function(resolve, reject) {var script = document.createElement('script');

script.src = src;

script.onload = resolve;

script.onerror = function() {if (backupSrc) {***/****主要资源加载失败，切换到备⽤资源路径* script.src = backupSrc;

} else {

reject(new Error('Failed to load script: 'src)); }

};document.head.appendChild(script); });

}

***/****使⽤⽰例*

loadScript('<https://cdn.example.com/script.js',> '<https://backup.example.com/script.js'>)

.then(function() {***/****资源加载成功* })

.catch(function(error) {***/****资源加载失败*console.error(error); });

通过动态加载脚本的⽅式 ，可以在资源加载失败时切换到备⽤资源路径或处理加载错误。 除了前⾯提到的⽅法外 ，还有以下⼀些降级处理的⽅法：

1. 本地备份资源：在 Web 应⽤的服务器上存储备份的静态资源⽂件 ，并在主要资源加载失败时 ，从 本地服务器上加载备份资源。这种⽅法需要在服务器上维护备份资源的更新和⼀致性。

2. 使⽤浏览器缓存：如果静态资源被浏览器缓存 ，则在资源加载失败时 ，浏览器可以使⽤缓存中的资 源。可以通过设置合适的缓存策略 ，例如设置资源的 Cache-Control 头字段 ，让浏览器缓存资源并 在需要时从缓存中加载。

3. 使⽤ Service Worker：使⽤ Service Worker 技术可以在浏览器中拦截⽹络请求 ，从⽽实现更⾼级 的降级处理。 当主要资源加载失败时 ，可以使⽤ Service Worker 拦截请求并返回备⽤资源 ，或者 动态⽣成代替资源。

4. 使⽤资源加载管理⼯具：使⽤像 Webpack 这样的资源加载管理⼯具 ，可以通过配置多个资源⼊⼝ 点和插件来实现资源加载的灵活控制。在资源加载失败时 ，可以通过配置⾃动切换到备⽤资源或通 过插件实现⾃定义的降级逻辑。

这些⽅法可以根据具体的需求和场景选择适合的降级处理策略。降级处理的⽬的是确保⽹⻚应⽤的正 常运⾏ ，提⾼⽤⼾体验 ，并减少对单⼀资源的依赖性。

23. html 中前缀为 data- 开头的元素属性是什么？

HTML 中前缀为 data-  开头的元素属性被称为⾃定义数据属性（Custom Data Attributes） 或者数

据属性（ Data Attributes）。

这些属性的命名以 data-  开头 ，后⾯可以跟上任意⾃定义的名称。这样的属性可以⽤来存储与元素

相关的⾃定义数据 ，以便在 JavaScript 或 CSS 中进⾏访问和操作。 ⾃定义数据属性的命名应该遵循以下规则：

• 属性名必须以

|  |
| --- |
| data- |

开头。

• 属性名可以包含任意

|  |
| --- |
| 字⺟ 、数字、连字符（ -） 、下划线（\_）和⼩数点（ .） |

。

• 属性名不应该包含⼤写字⺟ ，因为 HTML 属性名是不区分⼤⼩写的。

通过⾃定义数据属性 ，我们可以在 HTML 元素中嵌⼊⾃定义的数据 ，然后在 JavaScript 中使⽤

|  |
| --- |
| getAttribute |

()

⽅法或直接通过元素对象的

|  |
| --- |
| dataset |

属性来访问这些数据。

例如 ，在 HTML 中定义了⼀个⾃定义数据属性

|  |
| --- |
| data-color="red" |

:

|  |
| --- |
| 1 <div id="myDiv" data-color="red"</div |

在 JavaScript 中可以通过以下⽅式获取该⾃定义数据属性的值：

|  |
| --- |
| 1  myDiv = document.getElementById('myDiv');  2  color = myDiv.getAttribute('data-color'); *// 获取属性值为* *"red"*  *dataset = myDiv.dataset; // 获取包含所有⾃定义数据属性的对象* *{ color: "red" }* *colorValue = dataset.color; // 获取属性值为* *"red"* |

通过⾃定义数据属性 ，我们可以将相关的数据绑定到 HTML 元素上 ，⽅便在 JavaScript 中进⾏处理和 操作 ，增强了 HTML 和 JavaScript 之间的交互性。

24. 移动端如何实现上拉加载 ，下拉刷新？【热度: 718】

移动端实现上拉加载和下拉刷新通常使⽤⼀些特定的库或框架来简化开发。 以下是两种常⻅的实现⽅ 式：

1. 使⽤第三⽅库：⼀些流⾏的移动端UI库（如iScroll、 BetterScroll、Ant Design Mobile等）提供了 上拉加载和下拉刷新的功能 ，你可以使⽤它们来实现。这些库通常提供了易于使⽤的API和配置选 项 ，可以在你的应⽤中轻松地集成上拉加载和下拉刷新功能。

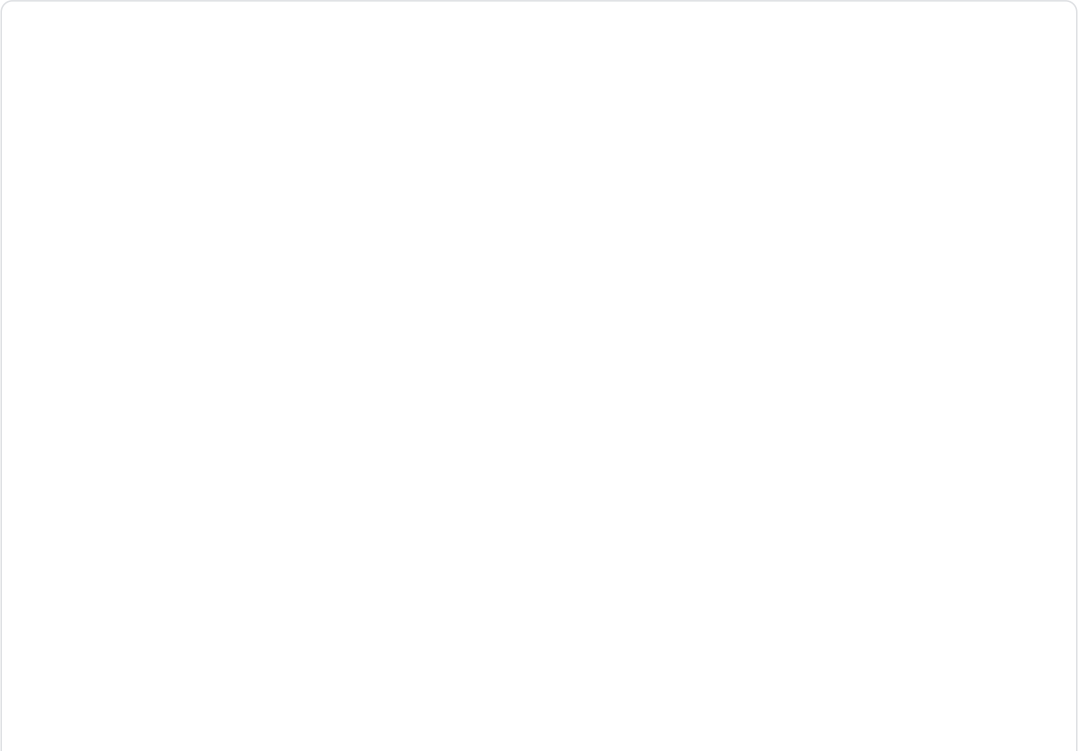
2. ⾃定义实现：如果你想更⾃定义地实现上拉加载和下拉刷新 ，可以使⽤原⽣的触摸事件（如

touchstart、touch move、touchend等）和滚动事件（如scroll）来监测⽤⼾的⼿势操作和滚动⾏ 为 ，并根据这些事件来触发相应的加载或刷新逻辑。你可以监听触摸事件来检测⽤⼾的下拉或上拉 ⼿势 ， 当达到⼀定的阈值时 ，触发刷新或加载的操作。 同时 ，你还需要监听滚动事件来判断当前滚 动位置是否已经到达⻚⾯底部 ，从⽽触发上拉加载的操作。

当⾃定义实现上拉加载和下拉刷新时 ，你可以使⽤JavaScript和HTML/CSS来编写代码。下⾯是⼀个简 单的⽰例 ，演⽰了如何通过原⽣事件来实现上拉加载和下拉刷新的功能：

HTML 结构：

|  |
| --- |
| 1 <html<head<title上拉加载和下拉刷新⽰例</title<style 2 */\* ⽤于展⽰加载和刷新状态的样式* *\*/*  3 .loading {text-align: center;padding: 10px;background-color: #f1f1f1;}  4  5 .refresh {text-align: center;padding: 10px;background-color: #f1f1f1;}  6 </style</head<body<div id="content"*<!-- 内容区域* *-->*</div<div id="loading" class="loading"  7 加载中 ...</div<div id="refresh" class="refresh"  8 下拉刷新</div<script src="your\_script.js"</script</body</html |

JavaScript 代码（your\_script.js）：

1 *// 获取相关元素* *content = document.getElementById('content');* 2  loading = document.getElementById('loading');

3  refresh = document.getElementById('refresh'); 4

5  isRefreshing = false;

6  isLoading = false;

7

8 *// 监听触摸事件* *startY = 0;* 9  moveY = 0;

10 content.addEventListener('touchstart',  (event) {

11 startY = event.touches[0].pageY;

12 });

13 content.addEventListener('touchmove',  (event) {

14 moveY = event.touches[0].pageY;

15 *// 下拉刷新* *(moveY - startY > 100 && !isRefreshing) {*

16 refresh.innerHTML = '释放刷新 ';

17 }

18 *// 上拉加载* *scrollTop = content.scrollTop;* *scrollHeight =*

*content.scrollHeight;* *offsetHeight = content.offsetHeight;* *(scrollTop +*

*offsetHeight >= scrollHeight && !isLoading) {*

19 loading.style.display = 'block';

20 }

21 });

22 content.addEventListener('touchend', (event) {*// 下拉刷新* *(moveY - startY > 100 && !isRefreshing) {*

23 refresh.innerHTML = '刷新中 ...';

24 simulateRefresh();

25 }

26 *// 上拉加载* *scrollTop = content.scrollTop;* *scrollHeight =*

*content.scrollHeight;* *offsetHeight = content.offsetHeight;* *(scrollTop +*

*offsetHeight >= scrollHeight && !isLoading) {*

27 loading.style.display = 'block';

28 simulateLoad();

29 }

30 *// 重置状态*

31 startY = 0;

32 moveY = 0;

33 }); 34

35 *//模拟刷新* *(){*

36 isRefreshing = true;

37 setTimeout( () {*//刷新完成后的操作*

38 refresh.innerHTML = '刷新成功 ';

39 isRefreshing = false; 40 }, 2000);

41 } 42

43 *//模拟加载* *(){*

44 isLoading = true;

45 setTimeout( () {*// 加载完成后的操作*

46 loading.style.display = 'none';

47 isLoading = false;

48 }, 2000);

49 }

上⾯的代码使⽤了

、

|  |
| --- |
| touchstart |

和

|  |
| --- |
| touchmove |

|  |
| --- |
| touchend |

事件来监测⽤⼾的⼿势操作 ，实现

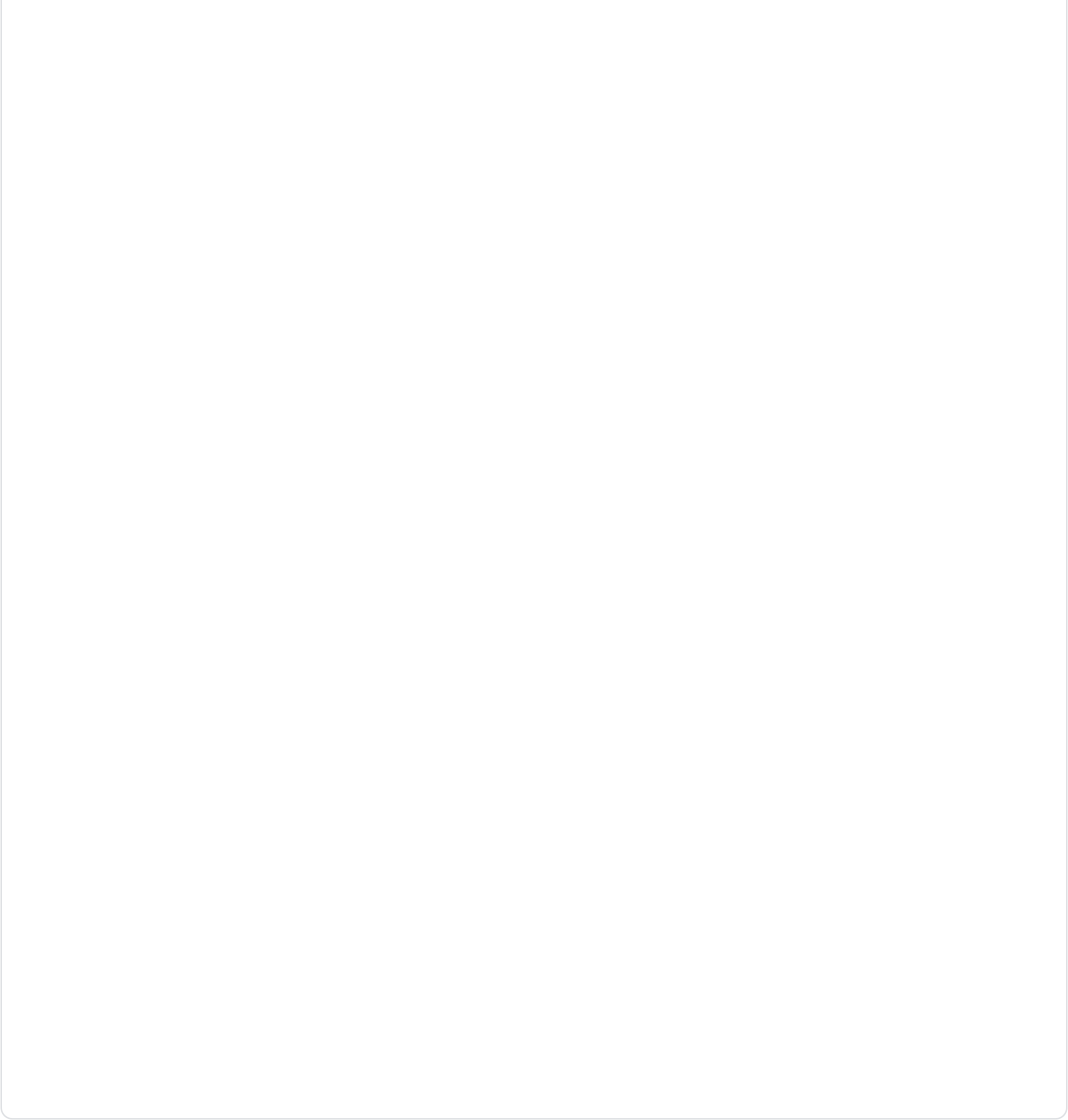
了下拉刷新和上拉加载的功能。通过修改

和

|  |
| --- |
| refresh |

|  |
| --- |
| loading |

元

素的内容和样式 ，可以实现相应的状态展⽰效果。

25. 如何判断dom元素是否在可视区域 【热度: 846】 判断 DOM 元素是否在可视区域可以使⽤以下⽅法：

1. getBoundingClientRect() ⽅法

该⽅法返回元素的⼤⼩及其相对于视⼝的位置 ，包括 top、 right、 bottom、 left 四个属性。我们可以 根据这四个属性来判断元素是否在可视区域内。

|  |
| --- |
| 1 (element) { rect =  element.getBoundingClientRect();  (  2 rect.top >= 0 &&  3 rect.left >= 0 &&  4 rect.bottom <= (window.innerHeight ||  document.documentElement.clientHeight) &&  5 rect.right <= (window.innerWidth || document.documentElement.clientWidth)  6 );  7 }  8  9 *// Example usage* *element = document.getElementById('my-element');*  10  (isInViewport(element)) {console.log('Element is in viewport');  11 }  {console.log('Element is not in viewport');  12 } |

1. IntersectionObserver API

该 API可以观察元素与其祖先元素或视⼝交叉的情况 ，并且可以设置回调函数 ， 当元素的可⻅性发⽣ 变化时会调⽤该回调函数。

|  |
| --- |
| 1 (entries, observer) {  2 entries.forEach(entry => { (entry.isIntersecting) {console.log('Element is  in viewport');  3 }  {console.log('Element is not in viewport');  4 }  5 });  6 }  7  8  observer =  IntersectionObserver(callback); 9  10  element = document.getElementById('my-element');  11 observer.observe(element); |

使⽤ IntersectionObserver API 的优点是可以减少不必要的计算和事件监听 ，提⾼了性能。

26. 前端如何⽤ canvas 来做电影院选票功能 电影院选票功能可以通过 Canvas来实现 ，具体实现步骤如下：

1. 绘制座位图案：使⽤ Canvas 绘制座位图案 ，可以⽤矩形或圆形来表⽰每个座位 ，还可以添加不同 颜⾊来表⽰该座位的状态（ 已售、 已选、可选等）。

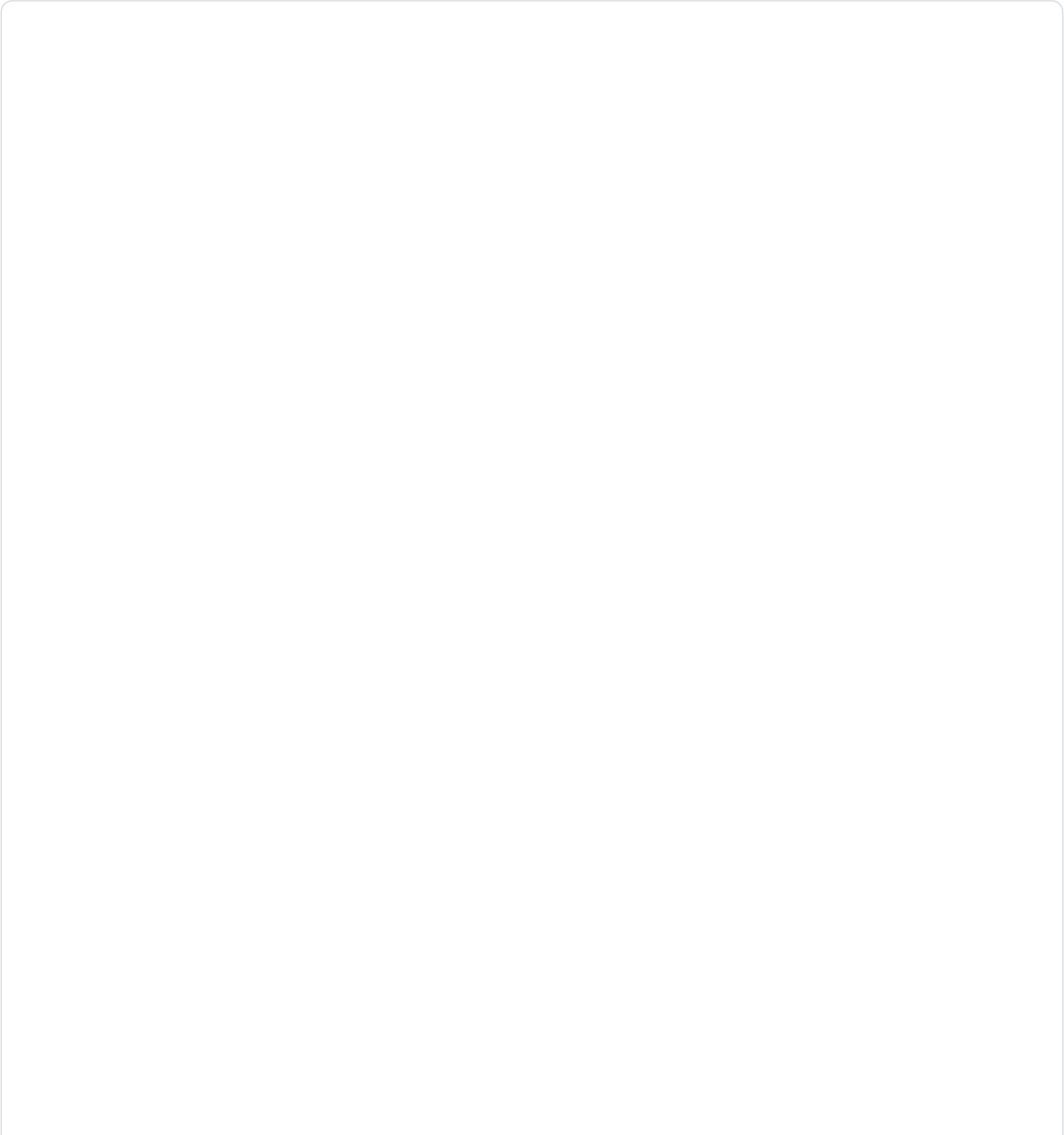
2. 添加⿏标事件：添加⿏标事件 ，如⿏标移动、 ⿏标单击等 ，来实现⽤⼾交互操作。例如 ， 当⽤⼾点 击座位时 ，将该座位的状态改为已选状态 ，并更新座位图案的颜⾊。

3. 统计已选座位：在⽤⼾选票的过程中 ，需要统计已选座位的数量和位置 ，并将选票信息展⽰给⽤ ⼾ 。可以通过遍历座位图案数组来实现。

4. 添加检查功能：为了防⽌⽤⼾在选票过程中出现错误 ，可以添加检查功能 ，如检查座位是否已被售 出或已被其他⼈选中等。

5. 添加确认和⽀付功能： 当⽤⼾选好座位后 ，需要确认并⽀付 ，可以通过弹出确认对话框来实现 ，并 将⽤⼾的选票信息发送⾄后台进⾏处理。

代码实现如下



1 - <!DOCTYPE html><html lang="en"<head<meta charset="UTF-8"

<titleTitle</title</head<body<canvas id="canvas" width="800" height="600" </canvas<button id="btnPay"确认并⽀付</button<script

2 // 获取画布和按钮元素

3 var canvas = document.getElementById('canvas');var btnPay =

document.getElementById('btnPay'); 4 // 获取画布上下⽂和座位数组

5 var ctx = canvas.getContext('2d');var seats = []; 6 // 绘制座位

7 function drawSeat(x, y, state) {switch (state) {case 0:

8 ctx.fillStyle = '#ccc'; // 可选座位

9 break;case 1:

10 ctx.fillStyle = '#f00'; // 已售座位

11 break;case 2:

12 ctx.fillStyle = '#0f0'; // 已选座位

13 break;default:

14 ctx.fillStyle = '#000'; // 其他座位

15 break; 16 }

17 ctx.fillRect(x, y, 30, 30);

18 }

19 // 初始化座位数组

20 function initSeat() {for (var i = 0; i < 10; i++) {

21 seats[i] = [];for (var j = 0; j < 10; j++) {

22 seats[i][j] = 0; // 初始状态为可选

23 drawSeat(i \* 4050, j \* 4050, 0); // 绘制座位 24 }

25 }

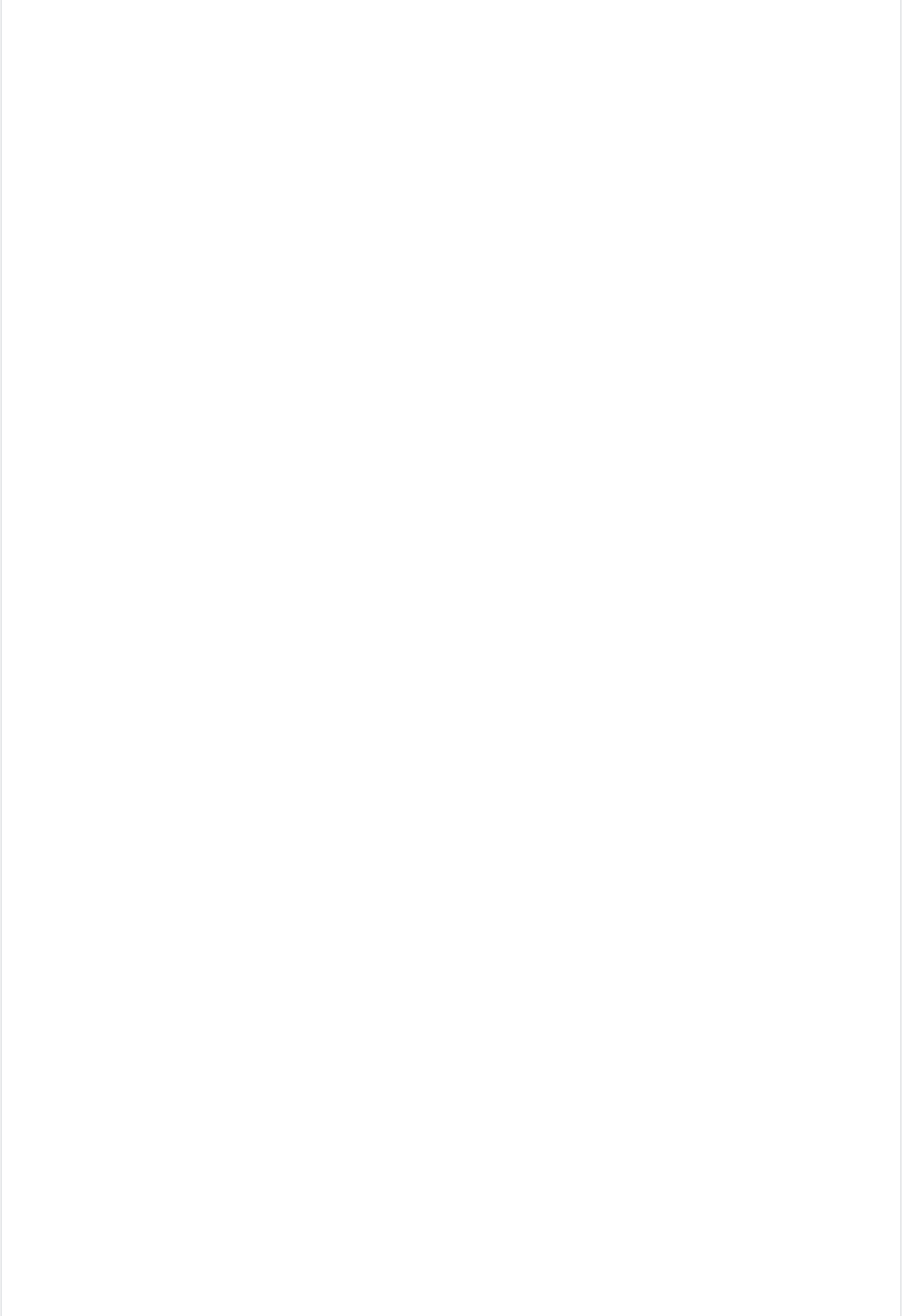
26 }

27 // 统计已选座位数量和位置

28 function countSelectedSeats() {var selectedSeats = [];var count = 0;for (var

i = 0; i < 10; i++) {for (var j = 0; j < 10; j++) {if (seats[i][j] == 2) {

29 selectedSeats.push([i, j]);

30 count++;

31 }

32 }

33 }return [count, selectedSeats];

34 }

35 // 更新座位状态和颜⾊

36 function updateSeat(x, y) {if (seats[x][y] == 0) {

37 seats[x][y] = 2; // 更改为已选状态 38 } else if (seats[x][y] == 2) {

39 seats[x][y] = 0; // 更改为可选状态 40 }

41 drawSeat(x \* 4050, y \* 4050, seats[x][y]); // 更新颜⾊ 42 }

43 // 检查座位状态是否可选

44 function checkSeat(x, y) {if (seats[x][y] == 1) {

45 alert('该座位已售出 ，请选择其他座位！ ');return false; 46 } else if (seats[x][y] == 2) {

47 alert('该座位已被选中 ，请选择其他座位！ ');return false; 48 }return true;

49 }

50 // 点击事件处理函数

51 function handleClick(e) {var x = parseInt((e.clientX - canvas.offsetLeft -

50) / 40);var y = parseInt((e.clientY - canvas.offsetTop - 50) / 40);if (x >=

0 && x < 10 && y >= 0 && y < 10) {if (checkSeat(x, y)) {

52 updateSeat(x, y);var count = countSelectedSeats()[0];if (count > 0) {

53 btnPay.innerHTML = '确认并⽀付（ 已选 'count + ' 座位） '; 54 } else {

55 btnPay.innerHTML = '确认并⽀付 '; 56 }

57 }

58 }

59 }

60 // 确认并⽀付按钮点击事件处理函数

61 function handlePay() {var selectedSeats = countSelectedSeats()[1];if

(selectedSeats.length == 0) {

62 alert('请选择座位！ ');return;

63 }if (confirm('您已选中以下座位： 'selectedSeats.join('、 ') + '，确认⽀付吗？ '))

{// 向后台发送选票信息 ，并进⾏⽀付处理

64 alert('⽀付成功！请前往指定影院取票！ ');

65 initSeat(); // 重新初始化座位

66 btnPay.innerHTML = '确认并⽀付 '; 67 }

68 }

69 // 初始化座位 70 initSeat();

71 // 绑定点击事件和确认并⽀付按钮点击事件

72 canvas.addEventListener('click', handleClick);

73 btnPay.addEventListener('click', handlePay); 74

75 </script</body</html

27. 如何通过设置失效时间清除本地存储的数据？【热度: 1,085】

要清除本地存储的数据 ，可以通过设置失效时间来实现。 以下是⼀种常⻅的⽅法：

1. 将数据存储到本地存储中 ，例如使⽤localStorage或sessionStorage。

2. 在存储数据时 ，同时设置⼀个失效时间。可以将失效时间存储为⼀个时间戳或特定的⽇期时间。

3. 在读取数据时 ，检查当前时间是否超过了失效时间。如果超过了失效时间 ，则认为数据已过期 ，需 要清除。

4. 如果数据已过期 ，则使⽤localStorage.removeItem(key)或sessionStorage.removeItem(key)⽅法 删除该数据。

以下是⼀个⽰例代码：

• ***/****存储数据*function setLocalStorageData(key, data, expiration) {var item = { data: data,

expiration: expiration };

localStorage.setItem(key, JSON.stringify(item)); }

***/****读取数据*function getLocalStorageData(key) {var item = localStorage.getItem(key);if (item) {

item = JSON.parse(item);if (item.expiration && new Date().getTime() > item.expiration) {***/*** *数据已过期，清除数据*

localStorage.removeItem(key);return null; }return item.data;

}return null; }

***/****⽰例⽤法*var data = {name: 'John', age: 30};

var expiration = new Date().getTime() + 36001000;***/****设置失效时间为当前时间后的1⼩时* setLocalStorageData('user', data, expiration);

var storedData = getLocalStorageData('user'); console.log(storedData);

在⽰例代码中 ，setLocalStorageData函数⽤于存储数据 ，并接受⼀个失效时间参数。

getLocalStorageData函数⽤于读取数据 ，并检查失效时间是否已过期。如果数据已过期 ，则清除数 据。⽰例中的失效时间设置为当前时间后的1⼩时。

28. 如果不使⽤脚⼿架 ， 如果⽤ webpack 构建⼀个⾃⼰的 react 应⽤ 【热度: 729】

利⽤ webpack 初始化基本应⽤构建

要在Webpack配置中添加对Less和Ant Design组件库的⽀持 ，需要进⾏以下步骤：

1. 安装所需的依赖。

|  |
| --- |
| 1 npm install less less-loader antd |

1. 在Webpack配置⽂件中添加对Less的⽀持。

|  |
| --- |
| 1 module.exports = {*// ...其他配置module: {*  2 rules: [*// ...其他规则* 3 {  4 test: /\.less$/, *// 匹配Less⽂件*  5 use: ['style-loader', *// 将CSS插⼊到⻚⾯中'css-loader', // 解析CSS⽂* *件'less-loader' // 将Less转换为CSS*  6 ]  7 }  8 ]  9 },  10 *// ...其他配置* 11 }; |

1. 在⼊⼝⽂件中引⼊Ant Design的样式⽂件。

|  |
| --- |
| 1 *// ⼊⼝⽂件* *index.js* *React*  *'react';* 2  ReactDOM  'react-dom';  3  'antd/dist/antd.css'; *// 引⼊Ant Design的样式⽂件* *App*  *'./App';*  4 ReactDOM.render(<App />, document.getElementById('root')); |

现在你可以在React组件中使⽤Ant Design的组件和样式了。例如 ，在 Button组件：

|  |
| --- |
| App . s |

j

中使⽤Ant Design的



|  |
| --- |
| 1 *// App.js* *React*  *'react';*  2  { Button }  'antd';  3  4  () {  (<div  5 <h1Hello, React!</h1  6 <Button type="primary"Click me</Button  7 </div  8 );  9 }  10  11   App; |

重新运⾏Webpack开发服务器 ，你应该能够看到Ant Design的Button组件正常显⽰在⻚⾯上。

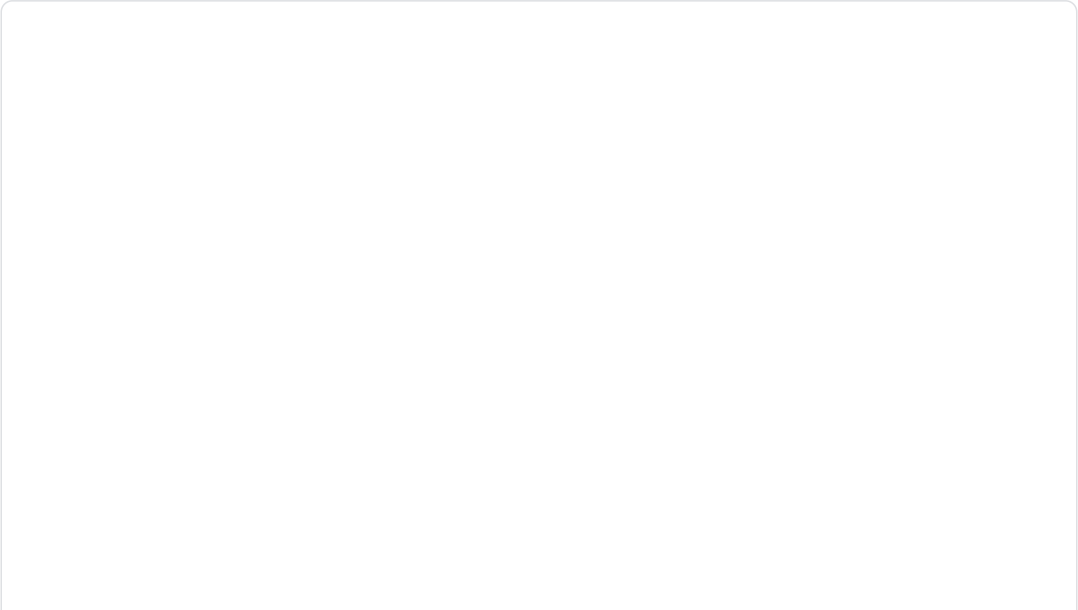
以上是⼀种简单的配置⽅法 ，你还可以根据需要进⾏更⾼级的配置 ，例如按需加载、 ⾃定义主题等。 有关更多信息 ，请参阅Webpack和Ant Design的官⽅⽂档。

使⽤ less 应该如何配置 ， 同时⽀持 css module 和 ⾮ css module

如果你想在Webpack配置中同时⽀持Less和CSS Module（局部作⽤域的CSS） ，可以使⽤以下配置。

1. 安装所需的依赖。

|  |
| --- |
| 1 npm install less less-loader css-loader style-loader |

1. 在Webpack配置⽂件中添加对Less和CSS的⽀持。

1 module.exports = {*// ...其他配置module: {*

2 rules: [*// ...其他规则* 3 {

4 test: /\.less$/, *// 匹配Less⽂件*

5 exclude: /\.module\.less$/, *// 排除CSS Module的Less⽂件*

6 use: ['style-loader', *// 将CSS插⼊到⻚⾯中'css-loader', // 解析CSS⽂* *件'less-loader' // 将Less转换为CSS*

7 ]

8 },

9 {

10 test: /\.module\.less$/, *// 匹配CSS Module的Less⽂件*

11 use: ['style-loader', *// 将CSS插⼊到⻚⾯中* 12 {

13 loader: 'css-loader', *// 解析CSS⽂件*

14 options: {

15 modules: true *// 启⽤CSS Module*

16 }

17 },'less-loader' *// 将Less转换为CSS*

18 ]

19 }

20 ]

21 },

22 *// ...其他配置* 23 };

现在 ，你可以同时使⽤普通的Less⽂件和CSS Module的Less⽂件。例如，

|  |
| --- |
| styles .module .less |

是⼀个CSS Module的Less⽂件，

|  |
| --- |
| styles .less |

是⼀个普通的Less⽂件。

|  |
| --- |
| 1 */\* styles.module.less \*/*.container {background-color: red;  2 } |

|  |
| --- |
| 1 */\* styles.less \*/*.text {color: blue;  2 } |

在React组件中使⽤这些样式：

|  |
| --- |
| 1  React  'react';  2  styles  './styles.module.less'; *// 导⼊CSS Module的样式*  *'./styles.less'; // 导⼊普通的Less样式* *() {* *(<div className= {styles.container}*  3 <h1 className="text"Hello, React!</h1  4 </div  5 );  6 }  7  8   App; |

这样，

|  |
| --- |
| styles .container |

将应⽤CSS Module的样式，

|  |
| --- |
| .text |

将应⽤普通的Less样式。

重新运⾏Webpack开发服务器 ，你应该能够看到样式正常应⽤到组件中。

如何引⼊ antd 组件并且⽀持按需加载

要引⼊antd组件并⽀持按需加载 ，你需要进⾏以下配置。

1. 安装antd和babel插件。



1 npm install antd babel-plugin-import --save

1. 在.babel rc⽂件中配置babel插件。

|  |
| --- |
| 1 {  2 "plugins": [  3 ["import",  4 {  5 "libraryName": "antd",  6 "style": "css"  7 }  8 ]  9 ]  10 } |

1. 在Webpack配置⽂件中添加对Less和CSS的⽀持。

|  |
| --- |
| 1 module.exports = {*// ...其他配置module: {*  2 rules: [*// ...其他规则* 3 {  4 test: /\.less$/, *// 匹配Less⽂件*  5 exclude: /node\_modules/,  6 use: ['style-loader', *// 将CSS插⼊到⻚⾯中'css-loader', // 解析CSS⽂*  *件'less-loader' // 将Less转换为CSS*  7 ]  8 }  9 ]  10 },  11 *// ...其他配置* 12 }; |

1. 在你的组件中引⼊antd组件。

|  |
| --- |
| 1  React  'react';  2  { Button }  'antd'; 3  4 () {  <Button type="primary"Hello, Antd!</Button;  5 }  6  7   App; |

现在 ，你可以使⽤ antd组件并且只加载你需要的组件样式。Webpack会根据需要⾃动按需加载antd组 件的样式⽂件。

29. ⽤ nodejs 实现⼀个命令⾏⼯具 ， 统计输⼊⽬录下⾯指定 代码的⾏数 【热度: 1,732】

要实现⼀个命令⾏⼯具来统计输⼊⽬录下指定代码的⾏数 ，你可以使⽤Node.js的 内容并进⾏⾏数统计。 以下是⼀个简单的实现⽰例：

fs

模块来读取⽂件

|  |
| --- |
| 1  fs = require('fs');  2  path = require('path');  3  4 (dirPath, fileExtension) { totalLines = 0;  5 (filePath) { content = fs.readFileSync(filePath,  'utf-8'); lines = content.split('\n');  6 totalLines += lines.length;  7 }  8 (directoryPath) { files =  fs.readdirSync(directoryPath);  9 files.forEach((file) => { filePath = path.join(directoryPath,  file); stats = fs.statSync(filePath);  10  (stats.isFile() && path.extname(file) === fileExtension) {  11 countLinesInFile(filePath);  12 }  (stats.isDirectory()) {  13 processDirectory(filePath);  14 }  15 });  16 }  17 processDirectory(dirPath);  18  totalLines;  19 } 20  21 *// 命令⾏参数，第⼀个参数是⽬录路径，第⼆个参数是⽂件扩展名* *[\_, , dirPath, fileExtension] = process.argv;*  22  23  linesCount = countLinesInDirectory(dirPath, fileExtension);  24 console.log(`Total lines of ${fileExtension} files  ${dirPath}:  ${linesCount}`); |

你可以将上述代码保存为⼀个JavaScript⽂件 ，⽐如

line-counter .js

。然后 ，在终端中运⾏以

下命令：

是

1 node line-counter.js /path/to/directory .js

其中

/path/to/directory

.js

是你要统计的⽬录路径，

是你要统计的⽂件扩展名。运⾏命令

后 ，程序将会输出指定⽂件类型在指定⽬录中的总⾏数。

你可以根据需要⾃定义输出格式、⽂件过滤规则等。此⽰例只是⼀个基本的实现 ，你可以根据具体需 求进⾏扩展和优化。

30. package.json ⾥⾯ sideEffects 属性的作⽤是啥 【热度: 229】

sideEffects 作⽤

|  |
| --- |
| side Effects |

字段可以有以下⼏种取值：

|  |
| --- |
| package . son |

j

⽂件中的⼀个字段 ，它⽤于指定⼀个模块是否具有副作⽤ 。副作

⽤是指模块在加载时会执⾏⼀些特定的操作 ，⽽不仅仅是导出⼀个函数或对象。

|  |
| --- |
| side Effects |

• true ：表⽰模块具有副作⽤ ，即模块加载时会执⾏⼀些操作。这是默认值 ，如果没有在

|  |
| --- |
| package . son |

j

中明确指定

|  |
| --- |
| side Effects |

字段 ，则假设为 true 。

•  false ：表⽰模块没有副作⽤ ，即模块加载时不会执⾏任何操作。这意味着该模块只导出函数、 对象或其他静态内容 ，并且不依赖于其他模块的副作⽤ 。

• 数组：可以将模块的具体⽂件路径或⽂件匹配模式（使⽤glob模式） 列在数组中 ，以指定哪些⽂件

具有副作⽤ ，哪些⽂件没有副作⽤ 。例如，

[" ./s rc/\* .js", " ! ./s rc/utils/\* .js"]

表

⽰

s rc

.js

⽬录下的所有

⽂件都具有副作⽤ ，但是

s rc/utils

.js

⽬录下的

⽂件没有副作

使⽤ side Effects字段可以帮助构建⼯具（如Webpack） 进⾏优化。如果模块没有副作⽤ ，构建 ⼯具可以进⾏更好的摇树优化（tree shaking） ，即只保留项⽬所需的代码 ，⽽将未使⽤的代码消除， 从⽽减⼩最终打包⽂件的⼤⼩。

⽤

注意：在使⽤

|  |
| --- |
| side Effects |

字段时 ，需要确保你的代码确实没有副作⽤ ，否则可能会导致意外的⾏

为。

sideEffects 是如何辅助 webpack 进⾏优化的？

字段可以帮助Webpack进⾏摇树优化（Tree Shaking） ，从⽽减⼩最终打包⽂件的

|  |
| --- |
| side Effects |

⼤⼩。摇树优化是指只保留项⽬所需的代码 ，⽽将未使⽤的代码消除。

当Webpack打包时 ，它会通过静态分析来确定哪些导⼊的模块实际上被使⽤了 ，然后只保留这些被使 ⽤的代码 ，并将未使⽤的代码从最终的打包⽂件中删除。

在这个过程中 ，Webpack会检查模块的

|  |
| --- |
| side Effects |

字段。如果⼀个模块具有

|  |
| --- |
| side Effects |

字

段 ，并且设置为 false ，Webpack会认为该模块没有副作⽤ 。Webpack会在摇树优化过程中将未使

⽤的导出从该模块中删除 ，因为它不会影响项⽬的功能。

然⽽ ，如果⼀个模块具有

|  |
| --- |
| side Effects |

字段 ，并且设置为

true

或是⼀个数组 ，Webpack会认为

该模块具有副作⽤ 。在摇树优化过程中 ，Webpack会保留该模块的所有导出 ，因为它不能确定哪些代 码是副作⽤的。这样可以确保项⽬中需要的副作⽤代码不会被误删除。

因此 ，通过正确使⽤

|  |
| --- |
| side Effects |

字段 ，可以帮助Webpack更好地优化打包⽂件 ，减少不必要的

代码 ，提⾼应⽤程序的性能。

31. script 标签上有那些属性 ，分别作⽤是啥？【热度: 744】

在HTML中，

|  |
| --- |
| <script> |

标签⽤于引⼊或嵌⼊JavaScript代码。

|  |
| --- |
| <script> |

标签可以使⽤以下属性

来调整脚本的⾏为： 常⽤属性

1.

s rc

：指定要引⼊的外部JavaScript⽂件的URL。例如：

<script s rc="script .js">

</script> 。通过这个属性 ，浏览器会下载并执⾏指定的外部脚本⽂件。

2.

< script>

/

|  |
| --- |
| async |

：可选属性 ，⽤于指⽰浏览器异步加载脚本。这意味着脚本会在下载的同时继续解析

<script s rc="script .js" async>

HTML⽂档 ，不会阻塞其他资源的加载。例如：

。

3.

|  |
| --- |
| defer |

：可选属性 ，⽤于指⽰浏览器延迟执⾏脚本 ，直到⽂档解析完成。这样可以确保脚本在⽂

档完全呈现之前不会执⾏。 例如：

<script s rc="script .js" defer></script>

。

4. type ：指定脚本语⾔的MIME类型。通常是

或者

|  |
| --- |
| text avascript |

/j

|  |
| --- |
| module |

（⽤于ES6模

块） 。如果未指定该属性 ，浏览器默认将其视为JavaScript类型。例如：

<script

type="text/javascript"> . . .</script> 。

5.

charset="UTF-8">< script>

/

|  |
| --- |
| charset |

：指定外部脚本⽂件的字符编码。例如：

<script s rc="script .js"

。

6.

|  |
| --- |
| integrity |

：⽤于指定外部脚本⽂件的Sub resource Integrity（SRI） 。SRI可以确保浏览器在

<script

加载脚本时验证其完整性 ，防⽌通过恶意更改⽂件来执⾏潜在的攻击。例如：

s rc="script .js" integrity="sha256-

qzn LcsROx4GACP2dm0UCKCzCG+HiZ1guq6ZZDob/Tng="></script> 。

不常⽤属性

1.

|  |
| --- |
| crossorigin |

：正常的 script 元素将最⼩的信息传递给 window.onerror ，⽤于那些没有通过

标准 CORS 检查的脚本。要允许对静态媒体使⽤独⽴域名的⽹站进⾏错误记录 ，请使⽤此属性。参 ⻅ CORS 设置属性。

2.

|  |
| --- |
| fetchpriority |

：提供⼀个指⽰ ，说明在获取外部脚本时要使⽤的相对优先级。

3.

|  |
| --- |
| nomodule |

： 这个布尔属性被设置来标明这个脚本不应该在⽀持 ES 模块的浏览器中执⾏。 实际

上 ，这可⽤于在不⽀持模块化 JavaScript 的旧浏览器中提供回退脚本。

4.

|  |
| --- |
| nonce |

: 在 script-s rc Content-Security-Policy (en-US)

中允许脚本的⼀个⼀次

性加密随机数（ nonce） 。服务器每次传输策略时都必须⽣成⼀个唯⼀的 nonce 值。提供⼀个⽆法 猜测的 nonce 是⾄关重要。

5.

|  |
| --- |
| referrerpolicy |

: 表⽰在获取脚本或脚本获取资源时 ，要发送哪个 referrer。

32. 为什么 SPA 应⽤都会提供⼀个 hash 路由 ，好处是什么？ 【热度: 681】

SPA（单⻚应⽤） 通常会使⽤ hash 路由的⽅式来实现⻚⾯的导航和路由功能。这种⽅式将路由信息存

储在 URL 的⽚段标识符（ hash） 中 ，例如：

|  |
| --- |
| [www.example.com # home](http://www.example.com/#/home) |

[/ /](http://www.example.com/#/home)

。

以下是使⽤ hash 路由的 SPA 的⼀些好处：

1. 兼容性： Hash 路由对浏览器的兼容性⾮常好 ，可以在所有主流浏览器上运⾏ ，包括较旧的浏览器 版本。这是因为 hash 路由不需要对服务端进⾏特殊的配置或⽀持。

2. 简单实现：实现 hash 路由⾮常简单 ，只需要在⻚⾯中添加⼀个监听器来监听

|  |
| --- |
| hashchange |

事

件 ，然后根据不同的 hash 值加载对应的⻚⾯内容。这种⽅式不需要对服务器进⾏特殊配置 ，服务 器只需传送⼀个初始⻚⾯ ，之后的⻚⾯切换完全由前端控制。

3. 防⽌⻚⾯刷新：使⽤ hash 路由可以防⽌⻚⾯的完全刷新。 因为 hash 路由只改变 URL 的⽚段标识 符 ，不会引起整个⻚⾯的重新加载 ，所以⽤⼾在不同⻚⾯之间切换时 ，不会丢失当前⻚⾯的状态和 数据。

4. 前进后退⽀持： 由于 hash 路由不会引起⻚⾯的刷新 ，因此可以⽅便地⽀持浏览器的前进和后退操

作。浏览器的前进和后退按钮可以触发

|  |
| --- |
| hashchange |

事件 ，从⽽实现⻚⾯的导航和⻚⾯状态的

管理。

5. ⽆需服务端配置：使⽤ hash 路由 ，不需要对服务端进⾏特殊的配置。所有的路由和⻚⾯切换逻辑 都由前端控制 ，服务器只提供⼀个初始⻚⾯ 。这样可以减轻服务器的负担 ，并且可以将更多的逻辑 放在前端处理 ，提升⽤⼾体验。

虽然 hash 路由有⼀些好处 ，但也有⼀些局限性。例如 ，hash 路由的 URL 不够美观 ，也不利于 SEO （搜索引擎优化） 。为了解决这些问题 ，现代的 SPA 框架通常使⽤更先进的路由⽅式 ，例如 HTML5 的 History API ，它可以在不刷新整个⻚⾯的情况下改变 URL。不过 ，hash 路由仍然是⼀个简单可靠 的选择 ，特别适⽤于简单的 SPA 或需要兼容较旧浏览器的情况。

33. [React] 如何进⾏路由变化监听 【热度: 698】

在 React 中 ，你可以使⽤ React Router 库来进⾏路由变化的监听。 React Router 是 React 的⼀个常⽤ 路由库 ，它提供了⼀组组件和API来帮助你在应⽤中管理路由。

下⾯是⼀个⽰例代码 ，演⽰如何使⽤ React Router 监听路由的变化：

然后 ，在你的 React 组件中 ，使⽤ BrowserRouter 或 HashRouter 组件包裹你的应⽤ ：

|  |
| --- |
| 1  () {  (*//使⽤* *BrowserRouter 或* *HashRouter 包裹你的* *应⽤*  2 <BrowserRouter>  3 {*/\* 在这⾥编写你的应⽤内容* *\*/*}  4 </BrowserRouter>  5 );  6 }  7 export  App; |

当使⽤函数组件时 ，可以使⽤

|  |
| --- |
| use Effect |

钩⼦函数来监听路由变化。下⾯是修改后的⽰例代码：

|  |
| --- |
| 1  React, { useEffect }  'react';  2  { withRouter }  'react-router-dom';  3  4 (props) {  5 useEffect(() => { handleRouteChange = (location, action) => {*// 路由发⽣变* *化时执⾏的处理逻辑console.log('路由发⽣了变化', location, action);*  6 };  7 *// 在组件挂载后，添加路由变化的监听器* *unlisten = props.history.listen(handleRouteChange);*  8 *// 在组件卸载前，移除监听器* *() => {*  9 unlisten();  10 };  11 }, [props.history]);  12  (<div  13 {/\* 在这⾥编写组件的内容 \*/}</div  14 ); 15 }  16  17 *//使⽤* *withRouter ⾼阶组件将路由信息传递给组件*  withRouter(MyComponent); |

在上⾯的代码中 ，我们使⽤了

|  |
| --- |
| use Effect |

钩⼦函数来添加路由变化的监听器。在

|  |
| --- |
| use Effect |

的

回调函数中 ，我们定义了

|  |
| --- |
| handleRouteChange |

⽅法来处理路由变化的逻辑。然后 ，通过

|  |
| --- |
| props .history .listen |

⽅法来添加监听器 ，并将返回的取消监听函数赋值给

|  |
| --- |
| unlisten |

变

量。

|  |  |  |  |
| --- | --- | --- | --- |
| 同时 ，我们还在 | |  | | --- | | use Effect | | 返回的清理函数中调⽤了 |

除监听器。

发⽣变化时（即路由发⽣变化时），

|  |  |  |  |
| --- | --- | --- | --- |
| 需要注意的是 ， 由于 | |  | | --- | | use Effect | | 的依赖数组中包含了 |

|  |
| --- |
| props .history |

新路由变化的监听器。

函数 ，以确保在组件卸载时移

|  |
| --- |
| unlisten |

, 所以每当

|  |
| --- |
| props .history |

的回调函数会被调⽤ ，从⽽更

|  |
| --- |
| use Effect |

总结起来 ，通过使⽤

|  |
| --- |
| use Effect |

钩⼦函数和

|  |
| --- |
| props .history .listen |

⽅法 ，可以在函数组件

中监听和响应路由的变化。

34. [单点登录是是什么， 具体流程是什么](https://gitee.com/yanleweb/interview-question/issues/I7W2KU)

SSO ⼀般都需要⼀个独⽴的认证中⼼（ passport） ，⼦系统的登录均得通过 passport ，⼦系统本⾝将 不参与登录操作 ， 当⼀个系统成功登录以后 ，passport 将会颁发⼀个令牌给各个⼦系统 ，⼦系统可以 拿着令牌会获取各⾃的受保护资源 ，为了减少频繁认证 ，各个⼦系统在被 passport 授权以后 ，会建⽴ ⼀个局部会话 ，在⼀定时间内可以⽆需再次向 passport 发起认证。

具体流程是：

1. ⽤⼾访问系统 1 的受保护资源 ，系统 1 发现⽤⼾未登录 ，跳转⾄ sso 认证中⼼ ，并将⾃⼰的地址作 为参数

2. sso 认证中⼼发现⽤⼾未登录 ，将⽤⼾引导⾄登录⻚⾯

3. ⽤⼾输⼊⽤⼾名密码提交登录申请

4. sso 认证中⼼校验⽤⼾信息 ，创建⽤⼾与 sso 认证中⼼之间的会话 ，称为全局会话 ，同时创建授权 令牌

5. sso 认证中⼼带着令牌跳转会最初的请求地址（ 系统 1）

6. 系统 1 拿到令牌 ，去 sso 认证中⼼校验令牌是否有效

7. sso 认证中⼼校验令牌 ，返回有效 ，注册系统 1

8. 系统 1 使⽤该令牌创建与⽤⼾的会话 ，称为局部会话 ，返回受保护资源

9. ⽤⼾访问系统 2 的受保护资源

10. 系统 2 发现⽤⼾未登录 ，跳转⾄ sso 认证中⼼ ，并将⾃⼰的地址作为参数

11. sso 认证中⼼发现⽤⼾已登录 ，跳转回系统 2 的地址 ，并附上令牌

12. 系统 2 拿到令牌 ，去 sso 认证中⼼校验令牌是否有效

13. sso 认证中⼼校验令牌 ，返回有效 ，注册系统 2

14. 系统 2 使⽤该令牌创建与⽤⼾的局部会话 ，返回受保护资源

35. web ⽹⻚如何禁⽌别⼈移除⽔印 【热度: 540】 关于如何防⽌移除⽔印：

可以通过监听 DOM 的变化来检测是否有⼈删除⽔印 ，可以使⽤

|  |
| --- |
| MutationObserver API |

。

可以观察 DOM 树的变化 ，并在变化发⽣时触发回调函数。你可以在回调函数

|  |
| --- |
| MutationObserver |

中检查是否有⽔印被删除 ，然后采取相应的措施。

以下是⼀个简单的⽰例代码 ，演⽰了如何使⽤ MutationObserver 监听 DOM 变化：

|  |
| --- |
| 1 *// ⽬标节点* *targetNode = document.body;*  2  3 *// 创建* *MutationObserver 实例* *observer =*  *MutationObserver((mutationsList) => {* *(* *mutation of mutationsList) {// 检* *查是否有⼦节点被删除* *(mutation.removedNodes.length > 0) {// 在此处判断是否有⽔印被* *删除// 如果⽔印被删除，则重新插⼊⽔印的* *DOM 元素到⽬标节点//例如：*  *targetNode.appendChild(watermarkElement);*  4 }  5 }  6 });  7  8 *// 配置* *MutationObserver* *config = { childList: true, subtree: true };*  9  10 *// 开始观察⽬标节点*  11 observer.observe(targetNode, config); |

在上述代码中 ，我们创建了⼀个 MutationObserver 实例 ，并通过

|  |
| --- |
| observe |

⽅法将其绑定到⽬标节

点上。在回调函数中 ，我们使⽤

|  |
| --- |
| mutation .removedNodes |

来检查是否有⼦节点被删除 ，如果发

现⽔印被删除 ，则可以采取相应的措施来重新插⼊⽔印的 DOM 元素。

需要注意的是 ，MutationObserver API是现代浏览器提供的功能 ，在⽼旧的浏览器中可能不⽀持。 因 此 ，在实际使⽤时 ，你需要对浏览器的兼容性进⾏测试和处理。

另外 ，如果⽔印被删除后⽴即加回去 ，你可以在检测到⽔印被删除时 ，⽴即执⾏插⼊⽔印的代码 ，以 确保⽔印能够迅速地重新出现在⻚⾯上。

36. ⽤⼾访问⻚⾯⽩屏了 ， 原因是啥 ， 如何排查？【热度: 609】

⽤⼾访问⻚⾯⽩屏可能由多种原因引起 ，以下是⼀些可能的原因和排查步骤：

1. ⽹络问题：⽤⼾的⽹络连接可能存在问题 ，⽆法正确加载⻚⾯内容。可以要求⽤⼾检查⽹络连接， 或者⾃⼰尝试在不同⽹络环境下测试⻚⾯的加载情况。

2. 服务端问题：服务器未正确响应⽤⼾请求 ，导致⻚⾯⽆法加载。可以检查服务器的状态、 ⽇志和错 误信息 ，查看是否有任何异常。 同时 ，可以确认服务器上的相关服务是否正常运⾏。

3. 前端代码问题：⻚⾯的前端代码可能存在错误或异常 ，导致⻚⾯⽆法正常渲染。可以检查浏览器的 开发者⼯具 ，查看是否有任何错误信息或警告。 同时 ，可以尝试将⻚⾯的JavaScript、CSS和

HTML代码分离出来进⾏单独测试 ，以确定具体的问题所在。

4. 浏览器兼容性问题：不同浏览器对于某些代码的⽀持可能不⼀致 ，导致⻚⾯在某些浏览器中⽆法正 常加载。可以尝试在不同浏览器中测试⻚⾯的加载情况 ，同时使⽤浏览器的开发者⼯具检查是否有 任何错误或警告。

5. 第三⽅资源加载问题：⻚⾯可能依赖于某些第三⽅资源（如外部脚本、样式表等），如果这些资源 ⽆法加载 ，可能导致⻚⾯⽩屏。可以检查⽹络请求是否正常 ，是否有任何资源加载失败的情况。

6. 缓存问题：浏览器可能在缓存中保存了旧版本的⻚⾯或资源 ，导致新版本⽆法加载。可以尝试清除 浏览器缓存 ，或者通过添加随机参数或修改⽂件名的⽅式强制浏览器重新加载⻚⾯和资源。

7. 其他可能原因：⻚⾯⽩屏问题还可能由于安全策略（如CSP、CORS等） 限制、跨域问题、 DNS解 析问题等引起。可以使⽤浏览器的开发者⼯具检查⽹络请求和错误信息 ，查找可能的问题。

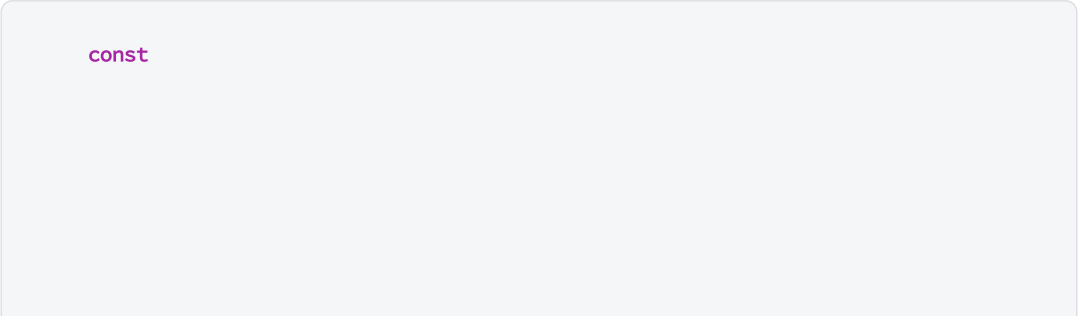
在排查问题时 ，可以根据具体情况逐步进⾏排查 ，并结合浏览器的开发者⼯具、服务器⽇志等⼯具来 辅助定位问题所在 ，并且可以与⽤⼾进⾏进⼀步沟通以获取更多信息。如果问题⽆法解决 ，可以寻求 专业的技术⽀持或咨询。

37. [代码实现] JS 中如何实现⼤对象深度对⽐ 【热度: 906】

在JavaScript中 ，可以使⽤递归的⽅式实现⼤对象的深度对⽐ 。以下是⼀个⽰例函数 ，⽤于⽐较两个 ⼤对象的每个属性是否相等：

|  |
| --- |
| 1 (obj1, obj2) {*//检查类型是否相同* *(* *obj1 !==*  *obj2) {* *false;*  2 }  3 *//检查是否是对象或数组* *(* *obj1 === 'object' && obj1 !== null && obj2 !== null) {//检查对象或数组⻓度是否相同* *(Object.keys(obj1).length !==*  *Object.keys(obj2).length) {* *false;*  4 }  5  ( key  obj1) {*// 递归⽐较每个属性的值* *(!deepEqual(obj1[key], obj2[key])) {* *false;*  6 }  7 }  8  true;  9 }  10 *// ⽐较基本类型的值* *obj1 === obj2;* 11 } |

使⽤⽰例：



1 obj1 = {

2 name: 'John',

3 age: 30,

4 address: {

5 street: '123 Main St',

6 city: 'New York'

7 }

8 };

|  |  |
| --- | --- |
| 9  10  obj2 = {  11 name: 'John',  12 age: 30,  13 address: {  14 street: '123 Main St',  15 city: 'New York'  16 } 17 }; 18  19  obj3 = {  20 name: 'Jane',  21 age: 25,  22 address: {  23 street: '456 Park Ave'  24 city: 'Los Angeles'  25 } 26 }; 27  28 console.log(deepEqual(obj1  *// false* | ,  , obj2)); *// trueconsole.log(deepEqual(obj1, obj3));* |

在上述⽰例中，

|  |
| --- |
| deepEqual |

函数会递归⽐较两个对象的每个属性的值 ，包括嵌套的对象或数组。如

果两个对象是相等的 ，则返回 true ，否则返回 false 。注意 ，该函数不会检查函数、正则表达

式、 ⽇期等复杂类型的值。

38. 如何理解数据驱动视图 ， 有哪些核⼼要素？【热度: 943】 关键词：理解数据驱动视图

数据驱动视图是指将数据作为主要驱动⼒ ，通过对数据的处理和分析 ，动态地更新和呈现视图的过

程。 它强调将数据与视图进⾏解耦 ，使得视图的呈现可以根据数据的变化⾃动更新 ，实现更灵活、可 扩展和可维护的视图。

数据驱动视图的核⼼要素包括：

1. 数据源：数据驱动视图需要有⼀个或多个数据源 ，这些数据源可以是来⾃数据库、API接⼝ 、⽂件 等不同的来源。

2. 数据处理：对数据进⾏处理和分析的过程。这包括对数据进⾏清洗、过滤、转换、计算等操作 ，以 便于后续的视图呈现。

3. 视图模板：视图模板定义了视图的结构和样式 ，并指定了如何将数据展⽰在视图中。视图模板通常 使⽤⼀种模板语⾔ ，可以根据数据的变化⾃动⽣成最终的视图。

4. 视图更新机制：视图更新机制是指如何根据数据的变化⾃动更新视图。这可以基于事件驱动的⽅

式 ， 当数据发⽣变化时主动更新视图；也可以采⽤响应式编程的⽅式 ，通过观察数据的变化来⾃动 更新视图。

5. ⽤⼾交互：数据驱动视图通常与⽤⼾进⾏交互 ，⽤⼾可以通过界⾯操作改变数据 ，从⽽触发视图的 更新。⽤⼾交互可以通过表单、按钮、滑块等不同的⽅式实现。

通过将数据与视图解耦 ，数据驱动视图可以实现更灵活、可扩展和可维护的视图呈现⽅式。 同时 ，它 也可以提⾼开发效率 ，减少开发⼈员对视图的⼿动管理。

39. vue-cli 都做了哪些事⼉ ，有哪些功能？ 【热度: 386】

Vue CLI 是⼀个基于 Vue.js 的命令⾏⼯具 ，⽤于快速搭建、开发和构建 Vue.js 项⽬ 。它提供了⼀系列 的功能来简化 Vue.js 项⽬的开发和部署流程 ，包括：

1. 项⽬脚⼿架：Vue CLI 可以通过简单的命令⾏交互⽅式快速⽣成⼀个新的 Vue.js 项⽬的基础结构， 包括⽬录结构、配置⽂件、⽰例代码等。

2. 开发服务器：Vue CLI 提供了⼀个开发服务器 ，⽤于在本地运⾏项⽬ ，在开发过程中实时预览和调 试应⽤程序。 它⽀持热模块替换（ HMR） ，可以实时更新⻚⾯内容 ，提⾼开发效率。

3. 集成构建⼯具：Vue CLI 集成了 Webpack ，可以⾃动配置和管理项⽬的构建过程。 它通过配置⽂件 可以进⾏定制 ，例如设置打包输出路径、优化代码、压缩资源等。

4. 插件系统：Vue CLI 提供了丰富的插件系统 ，可以通过安装插件来扩展项⽬的功能。这些插件可以 帮助处理样式、路由、状态管理、 国际化等⽅⾯的需求 ，提供更多的开发⼯具和功能⽀持。

5. 测试集成：Vue CLI 集成了测试⼯具 ，可以快速配置和运⾏单元测试和端到端测试。 它⽀持多种测 试框架 ，如 Jest、 Mocha、Cypress等 ，可以帮助开发⼈员编写和运⾏各种类型的测试。

6. 项⽬部署：Vue CLI 提供了命令⾏接⼝ ，可以⽅便地将项⽬部署到不同的环境 ，如开发环境、测试 环境和⽣产环境。 它⽀持⽣成优化过的静态⽂件、 ⾃动压缩和缓存等功能。

提供了⼀整套开发和构建 Vue.js 项⽬的功能和⼯具链 ，可以⼤⼤简化和加速 Vue.js 项⽬的开发过程。

40. JS 执⾏ 100 万个任务 ， 如何保证浏览器不卡顿？【热度: 806】

Web Workers

要确保浏览器在执⾏100万个任务时不会卡顿 ，你可以考虑使⽤Web Workers来将这些任务从主线程中 分离出来。Web Workers允许在后台线程中运⾏脚本 ，从⽽避免阻塞主线程 ，保持⻚⾯的响应性。

以下是⼀个使⽤Web Workers的简单⽰例：

1 *// 主线程代码* *worker =*  *Worker('worker.js'); //创建⼀个新的Web Worker*

2 worker.postMessage({ start: 0, end: 1000000 }); *// 向Web Worker发送消息*

|  |
| --- |
| 3 worker.onmessage =  (event) { result = event.data;console.log('任务 完成： ', result);  4 }; 5  6 *// worker.js - Web Worker代码*  7 onmessage =  (event) { start = event.data.start; end =  event.data.end; sum = 0; ( i = start; i <= end; i++) {  8 sum += i; 9 }  10 postMessage(sum); *// 向主线程发送消息* 11 }; |

在这个⽰例中 ，主线程创建了⼀个新的Web Worker ，并向其发送了⼀个包含任务范围的消息。Web Worker在后台线程中执⾏任务 ，并将结果发送回主线程。

requestAnimationFrame 来实现任务分割

使⽤

|  |
| --- |
| requestAnimation Frame |

来实现任务分割是⼀种常⻅的⽅式 ，它可以确保任务在浏览器的每

⼀帧之间执⾏ ，从⽽避免卡顿。 以下是⼀个使⽤

|  |
| --- |
| requestAnimation Frame |

来分割任务的简单例

⼦：

|  |
| --- |
| 1 *// 假设有⼀个包含⼤量元素的数组* *bigArray = Array.from({ length: 1000000 }, (\_, i) => i + 1);*  2  3 *// 定义⼀个处理函数，例如对数组中的每个元素进⾏平⽅操作* *(chunk) {* *chunk.map(num => num \* num);*  4 } 5  6 *// 分割任务并使⽤requestAnimationFrame* *chunkSize = 1000; // 每个⼩块的⼤⼩* *index = 0;*  7  8  () {  () { chunk = bigArray.slice(index, index + chunkSize); *// 从⼤数组中取出⼀个⼩块* *result = processChunk(chunk); // 处理⼩块任务console.log('处理完成：* *', result);*  9 index += chunkSize;  10  (index < bigArray.length) {  11 requestAnimationFrame(processChunkWithRAF); *// 继续处理下⼀个⼩块*  12 }  13 }  14 requestAnimationFrame(processChunkWithRAF); *// 开始处理⼤数组* 15 }  16 processArrayWithRAF(); |

在这个例⼦中 ，我们使⽤

|  |
| --- |
| requestAnimation Frame |

, 从⽽实现对⼤数组的任务分割。这样可以确保任务在每⼀帧之间执⾏，

来循环执⾏处理⼩块任务的函数

|  |
| --- |
| processChunkWithRAF |

避免卡顿。

针对上⾯的改进⼀下

是不能保证不卡的 ， 那么久需要动态调整

|  |
| --- |
| const chunkSize = 1000; 每个⼩块的⼤⼩ |

//

的⼤⼩ ， 代码可以参考下⾯的⽰范：

|  |
| --- |
| chunkSize |

|  |
| --- |
| 1  $result = document.getElementById("result"); 2  3 *// 假设有⼀个包含⼤量元素的数组* *bigArray = Array.from({ length: 1000000 }, (\_, i) => i + 1);*  4  5 *// 定义⼀个处理函数，对数组中的每个元素执⾏⼀次* *(chunk) {*  6 chunk: ${chunk}  7 ; 8 } 9  10 *// 动态调整* *chunkSize 的优化⽅式* *chunkSize = 1000; // 初始的* *chunkSize* *index = 0;*  11  12  () {  () { startTime = performance.now(); *// 记录结束时间* *(* *i = 0; i <*  *chunkSize; i++) {* *(index < bigArray.length) {* *result = processChunk(bigArray[index]); //对每个元素执⾏处理函数*  13 $result.innerText = result;  14 index++; 15 }  16 } endTime = performance.now(); timeTaken = endTime - startTime; *// 计*  *算处理时间// 根据处理时间动态调整* *chunkSize* *(timeTaken > 16) { // 如果处理时间超过* *⼀帧的时间（16毫秒），则减⼩* *chunkSize*  17 chunkSize = Math.floor(chunkSize \* 0.9); *// 减⼩10%*  18 }  (timeTaken < 16) { *// 如果处理时间远⼩于⼀帧的时间（8毫秒），则增加*  *chunkSize*  19 chunkSize = Math.floor(chunkSize \* 1.1); *// 增加10%*  20 }  21  (index < bigArray.length) {  22 requestAnimationFrame(processChunkWithRAF); *// 继续处理下⼀个⼩块*  23 }  24 }  25 requestAnimationFrame(processChunkWithRAF); *// 开始处理⼤数组* 26 }  27 processArrayWithDynamicChunkSize(); |

在这个例⼦中 ，我们动态调整

|  |
| --- |
| chunkSize |

的⼤⼩ ，根据处理时间来优化任务分割。根据处理时间的

表现 ，动态调整

|  |
| --- |
| chunkSize |

的⼤⼩ ，以确保在处理⼤量任务时 ，浏览器能够保持流畅 ，避免卡顿。

41. JS 放在 head ⾥和放在 body ⾥有什么区别？【热度: 420】

将 JavaScript 代码放在

|  |
| --- |
| <head> |

标签内部和放在

|  |
| --- |
| <body> |

标签内部有⼀些区别：

1. 加载顺序：放在

|  |
| --- |
| <head> |

⾥会在⻚⾯加载之前执⾏ JavaScript 代码 ，⽽放在

|  |
| --- |
| <body> |

⾥会在⻚

⾯加载后执⾏。

2. ⻚⾯渲染：如果 JavaScript 代码影响了⻚⾯的布局或样式 ，放在  <head> ⾥可能会导致⻚⾯渲染 延迟 ，⽽放在 <body> ⾥可以减少这种影响。

3. 代码依赖：如果 JavaScript 代码依赖其他元素 ，放在 <body> ⾥可以确保这些元素已经加载。

4. 全局变量和函数：放在 <head> ⾥的 JavaScript 代码中的全局变量和函数在整个⻚⾯⽣命周期内 都可⽤ 。

以下是⼀个简单的⽰例代码 ，展⽰了如何在

和

|  |
| --- |
| <head> |

|  |
| --- |
| <body> |

中放置 JavaScript 代码：

|  |
| --- |
| 1  <html<head<script  2 console.log("这是在 head 中执⾏的 JavaScript 代码。 "); </script</head<body<script  3 console.log("这是在 body 中执⾏的 JavaScript 代码。 ");</script</body</html |

在这个⽰例中 ，分别在

和

|  |
| --- |
| <head> |

|  |
| --- |
| <body> |

中放置了简单的 JavaScript 代码 ，⽤于在控制台输出

信息 ，以便观察执⾏顺序。

42. [Eslint 代码检查的过程是啥？ 【热度: 111】](https://gitee.com/yanleweb/interview-question/issues/I9GDHV)

ESLint 是⼀个插件化的静态代码分析⼯具 ，⽤于识别 JavaScript 代码中的问题。 它在代码质量和编码 ⻛格⽅⾯有助于保持⼀致性。代码检查的过程通常如下：

1. 配置：

⾸先需要为 ESLint 提供⼀套规则 ，这些规则可以在

|  |
| --- |
| .eslint rc |

配置⽂件中定义 ，或者在项⽬的

package .json

⽂件中的

|  |
| --- |
| eslintConfig |

字段⾥指定。规则可以继承⾃⼀套已有的规则集，

如

|  |
| --- |
| eslint :recommended |

, 或者可以是⼀个流⾏的样式指南 ，如

|  |
| --- |
| airbnb |

。也可以是⾃定义

的规则集。

2. 解析：

当运⾏ ESLint 时 ，它会使⽤⼀个解析器（如espree ，默认的解析器）来解析代码 ，将代码转换

成⼀个抽象语法树（AST） 。AST 是代码结构的⼀个树状表⽰ ，能让 ESLint 理解代码的语义结构。

3. 遍历：

⼀旦代码被转换成 AST ，ESLint 则会遍历该树。 它会查找树的每个节点 ，检查是否有任何规则适⽤ 于该节点。在遍历过程中 ，如果发现违反了某项规则 ，ESLint 将记录⼀个问题（通常称为“ lint 错 误”）。

4. 报告：

在遍历完整个 AST 之后 ，ESLint 会⽣成⼀份报告。这份报告详细说明了它在代码中找到的任何问 题。这些问题会被分类为错误或警告 ，根据配置设置的不同 ，某些问题可能会阻⽌构建过程或者被 忽略。

5. 修复：

对于某些类型的问题 ，ESLint 提供了⾃动修复的功能。这意味着你可以让 ESLint 尝试⾃动修复它 所发现的问题 ，不需⼈⼯⼲预。

6. 集成：

ESLint 可以集成到IDE 中 ，这样就可以在代码编写过程中即时提供反馈。 它也可以被集成到构建⼯ 具如 Webpack 或任务运⾏器 Grunt、Gulp 中 ，作为构建过程或提交代码到版本控制系统前的⼀个 步骤。

通过以上步骤 ，ESLint 帮助开发者在编码过程中遵循⼀致的⻛格和避免出现潜在的错误。

43. 虚拟混动加载原理是什么 ， ⽤ JS 代码简单实现⼀个虚拟 滚动加加载。【热度: 354】

原理

虚拟滚动（Virtual Scrolling）是⼀种性能优化的⼿段 ，通常⽤于处理⻓列表的显⽰问题。在传统的滚 动加载中 ， 当⾯对成千上万项的⻓列表时 ，直接在 DOM 中创建并展⽰所有项会导致严重的性能问题， 因为浏览器需要渲染所有的列表项。⽽虚拟滚动的核⼼原理是仅渲染⽤⼾可视范围内的列表项 ，以此 减少 DOM 操作的数量和提⾼性能。

实现虚拟滚动 ，我们需要：

1. 监听滚动事件 ，了解当前滚动位置。

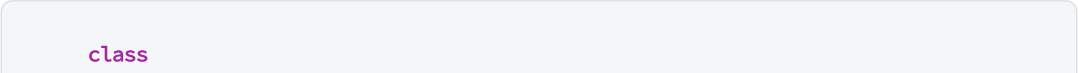
2. 根据滚动位置计算当前应该渲染哪些列表项⽬ （即在视⼝内的项⽬） 。

3. 只渲染那些项⽬ ，并⽤占位符（ ⽐如⼀个空的 div） 占据其它项⽬应有的位置 ，保持滚动条⼤⼩不 变。

4. 当⽤⼾滚动时 ，重新计算并渲染新的项⽬ 。

基础版本实现

以下是⼀个简单的虚拟滚动实现的 JavaScript 代码⽰例：



1 VirtualScroll {

|  |
| --- |
| 2 constructor(container, itemHeight, totalItems, renderCallback)  { .container = container; *// 容器元素* *.itemHeight = itemHeight; // 每个项的* *⾼度* *.totalItems = totalItems; // 总列表项数* *.renderCallback =*  *renderCallback; // 渲染每⼀项的回调函数* *.viewportHeight = container.clientHeight; // 视⼝⾼度* *.bufferSize =*  *Math.ceil(* *.viewportHeight / itemHeight) \* 3; // 缓冲⼤⼩* *.renderedItems = []; // 已渲染项的数组* *.startIndex = 0; // 当前渲染的开始索引* *.endIndex =*   *.bufferSize; // 当前渲染的结束索引*  3 container.addEventListener("scroll", () =>  .onScroll()); .update();  4 }  5 onScroll() { scrollTop =  .container.scrollTop; newStartIndex =  Math.floor(scrollTop /  .itemHeight) -  .bufferSize / 2;  newEndIndex = newStartIndex +  .bufferSize;  6  (newStartIndex !==  .startIndex || newEndIndex !==  .endIndex)  { .startIndex = Math.max(0, newStartIndex); .endIndex =  Math.min( .totalItems, newEndIndex); .update();  7 }  8 }  9 update() {*// 清空已有内容* *.container.innerHTML = "";*  10 *// 计算并设置容器的总⾼度* *totalHeight =*  *.totalItems \**  *.itemHeight;* *.container.style.height =*  11 ${totalHeight}px 12 ;  13 *// 渲染视⼝内的项* *fragment = document.createDocumentFragment();* *(* *i =*  *.startIndex; i <*  *.endIndex; i++) {* *item =*  *.renderCallback(i);*  14 item.style.top = `${i \*  .itemHeight}px`;  15 fragment.appendChild(item);  16 } .container.appendChild(fragment);  17 }  18 }  19  20 *//创建⼀个列表项的函数* *(index) {* *item = document.createElement("div");*  21 item.className = "list-item";  22 item.innerText =  23 Item ${index}  24 ;  25 item.style.position = "absolute";  26 item.style.width = "100%";  item;  27 } 28  29 *// 初始化虚拟滚动* *container = document.querySelector(".scroll-container"); // 容器元素需要预先在HTML中定义* *virtualScroll =*  *VirtualScroll(container, 30, 10000, createItem);* |

这个例⼦中 ，我们创建了⼀个

|  |
| --- |
| VirtualScroll |

类 ，通过传⼊容器、项⾼度、总项数和渲染回调函数

来进⾏初始化。该类的update⽅法⽤于渲染出当前可视范围内部分的项⽬ ，并将它们放到⽂档碎⽚

中 ，然后⼀次性添加到容器中。这样可以避免多次直接操作 DOM ，减少性能消耗。 当滚动时，

|  |
| --- |
| onScroll |

⽅法将计算新的

和

|  |
| --- |
| startIndex |

|  |
| --- |
| endIndex |

, 然后调⽤

|  |
| --- |
| update |

⽅法进⾏更新。请

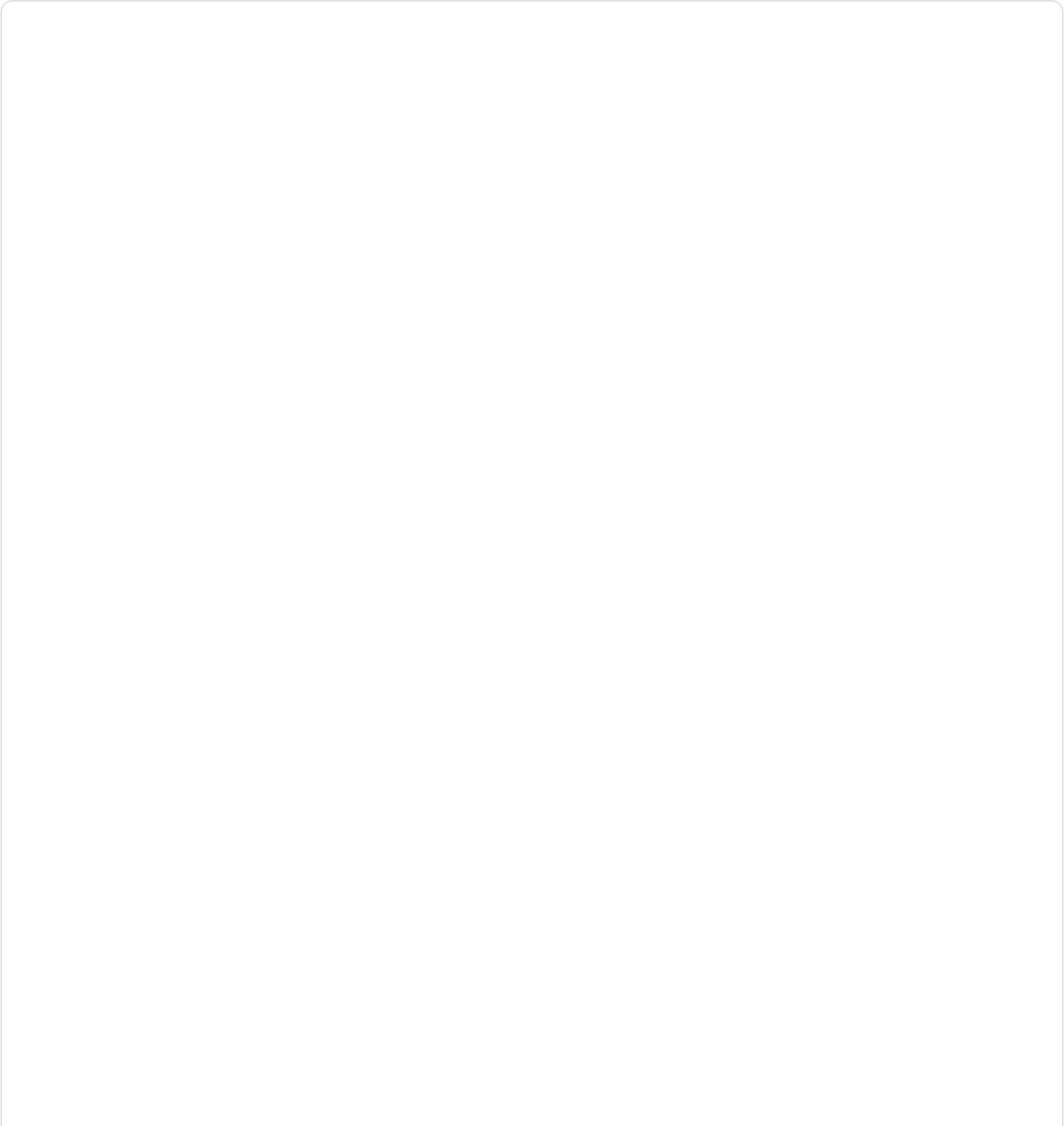
注意 ，实际应⽤可能需要根据具体情况调整缓冲区⼤⼩等参数。 进阶版本：使⽤ IntersectionObserver 来实现

使⽤

|  |
| --- |
| IntersectionObserver |

实现虚拟滚动就意味着我们会依赖于浏览器的API来观察哪些元素

进⼊或离开视⼝ （viewport） ，⽽⾮直接监听滚动事件。这样我们只需在需要时渲染或回收元素。

以下是⼀个简化版使⽤

|  |
| --- |
| IntersectionObserver |

来实现虚拟滚动的例⼦：

1  VirtualScroll {

2 constructor(container, itemHeight, totalItems, renderItem) { .container =

container; .itemHeight = itemHeight; .totalItems = totalItems; .renderItem = renderItem;

3  .observer =  IntersectionObserver( .onIntersection.bind(), {

4 root:  .container,

5 threshold: 1.0,

6 });

7  .items =  Map();

8  .init();

9 }

10 init() {*// 填充初始屏幕的元素* *(* *i = 0; i <*  *.totalItems; i++) {* *placeholder =*

 *.createPlaceholder(i);* *.container.appendChild(placeholder);* *.observer .observe(placeholder);*

11 }

12 }

13 createPlaceholder(index) { placeholder = document.createElement("div");

14 placeholder.style.height = `${ .itemHeight}px`;

15 placeholder.style.width = "100%";

16 placeholder.dataset.index = index; *// store index* *placeholder;*

17 }

18 onIntersection(entries) {

19 entries.forEach((entry) => { index = entry.target.dataset.index;

(entry.isIntersecting) { rendered =

 .renderItem(index); .container.replaceChild(rendered, entry.target); .items.set(index, rendered);

20 }  ( .items.has(index)) { placeholder =

 .createPlaceholder(index); .container.replaceChild(placeholder,

ex);.items.get(index)); .observer.observe(placeholder); .items.delete(ind 21 }

22 });

23 } 24 } 25

26 *// Render item function**(index) {* *item =*

*document.createElement("div");*

27 item.classList.add("item");

28 item.textContent =

29 Item ${index} 30 ;

31 item.dataset.index = index;

32 item.style.height = "30px"; *// Same as your itemHeight in VirtualScroll*

*item;*

33 } 34

35 *// Example usage:* *container = document.getElementById("scroll-*

*container"); // This should be a predefined element in your HTML*

*itemHeight = 30; // Height of each item* *itemCount = 1000; // Total number*

*of items you have* *virtualScroll =*  *VirtualScroll(container,*

*itemHeight, itemCount, renderItem);*

在这⾥我们创建了⼀个

|  |
| --- |
| VirtualScroll |

注册了这些占位元素。

类 ，构造函数接收容器元素、每个项的⾼度、总项⽬数和

⽤于渲染每个项⽬的函数。我们在初始化⽅法中 ，为每个项⽬创建了⼀个占位符元素 ，并且向

|  |
| --- |
| IntersectionObserver |

当⼀个占位元素进⼊到视⼝中时 ，我们就会渲染对应的项 ，并且将它替换这个占位符。 当⼀个项离开 视⼝ ，我们⼜会将它替换回原来的占位符并取消它的注册。

这种⽅法的优势包括：

• 不需要绑定滚动事件 ，防⽌滚动性能问题。

• 浏览器会⾃动优化观察者的回调。

• 不需要⼿动计算当前应该渲染的项⽬ ， 当⽤⼾快速滚动时也不会遇到空⽩内容。

44. [React] react-router 和 原⽣路由区别 【热度: 434】 React Router 和浏览器原⽣ history API在路由管理上主要有以下⼏个区别：

1. 抽象级别:

◦ React Router 提供了更⾼层次的抽象 ，如

、

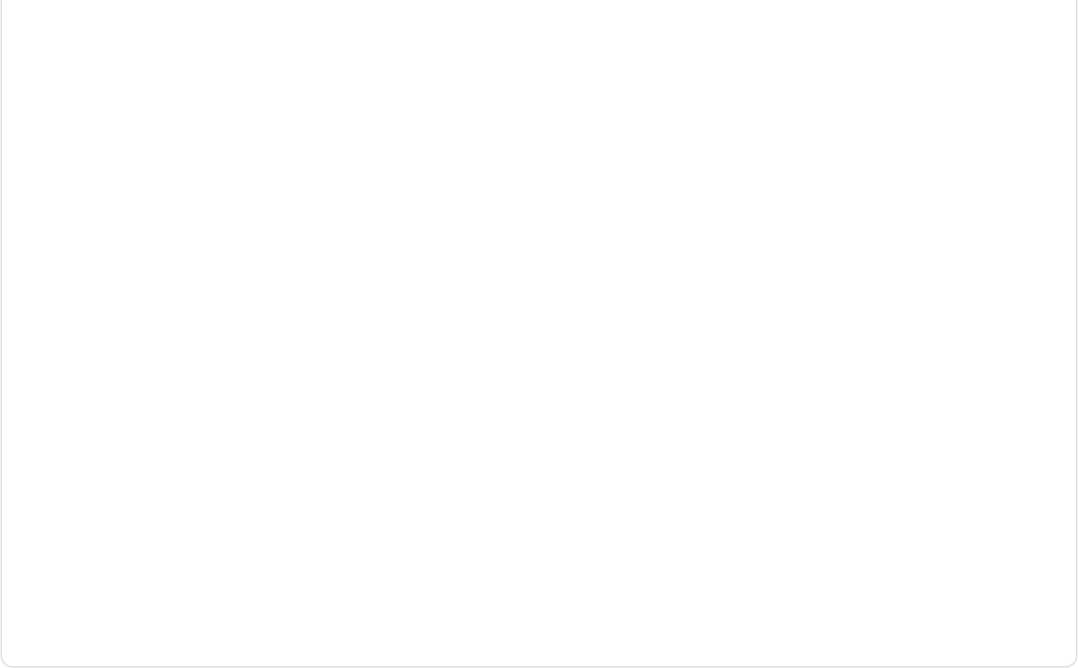
|  |
| --- |
| <Router> |

、和

|  |
| --- |
| <Route> |

|  |
| --- |
| <Link> |

等组件，

这些都是专⻔为了在 React 中更⽅便地管理路由⽽设计的。 它处理了底层 history API 的很多细 节 ，把操作抽象成了 React 组件和 hooks。

◦ 原⽣ history API更底层 ，直接作⽤于浏览器的历史记录栈。使⽤原⽣ history API 需要开发者 ⾃⼰编写更多的代码来管理 history 栈和渲染相应的组件。

2. 便利性:

o React Router提供了声明式导航和编程式导航的选项，并且有大量的社区支持和文档，易于使

用和学习。

。 原 生history API需要开发者自己处理URL与组件之间的关系映射，以及页面渲染的逻辑。

3. 功能：

o React Router除了包含对原生history API的基本封装外，还提供了如路由守卫、路由懒加载、 嵌套路由、重定向等高级功能。

。 原生 history API提供基本的历史记录管理功能，但是不包含上述 React Router提供的高级应 用路由需求。

4. 集成：

o React Router 是专为 React 设计的，与React 的生命周期、状态管理等密切集成。 。 原 生history API与 React 没有直接关联，需要用户手动实现整合。

5. 状态管理：

o React Router 可以将路由状态管理与应用的状态管理(如使用Redux) 结合起来，使路由状态

可预测和可管理。

。 原 生history API通常需要额外的状态管理逻辑来同步UI 和 URL。

6. 服务器渲染：

o React Router 可以与服务器渲染一起使用，支持同构应用程序，即客户端和服务器都可以进行 路由渲染。

o 原 生history API主要是针对客户端的，因此在服务器端渲染中需要额外的处理来模拟routing 行为。

在考虑是否使用React Router 或者原生history API时，通常需要考虑项目的复杂性、团队的熟悉度以 及项目对路由的特定需求。对于大多数 React 项目而言，React Router的便利性和其附加的高级特性 通常使得它成为首选的路由解决方案。

表格对比

|  |  |  |
| --- | --- | --- |
| **特性** | **React** **Router** | **原生**  **HistoryAPI** |
| 抽象级别 | 高层次抽  象，提供了 组件和  hooks | 底层API,直 接操作历史 记录栈 |
| 便利性 | 声明式和编 程式导航， 社区支持和 文档齐全 | 手动处理  URL和组件 映射，以及 渲染逻辑 |
|  | 改由字T |  |

|  |  |  |
| --- | --- | --- |
| 功能 | 由寸上  懒加载、嵌 套路由、重 定向等 | **基本的历史** **记录管理** |
| 集成 | **与React生** **命周期和状** **态管理紧密** **集成** | **需要手动整** **合到React** **中** |
| 状态管理 | **与应用的状** **态管理系统**  **(如**  **Redux)可集** **成，路由状** **态可预测和**  可管理 | **需要额外实** **现状态管理** **逻辑** |
| 服务器渲染 | 支持同构应 用程序，客 户端和服务 器都能渲染 | **主要用于客** **户端，服务** **器端需要模** **拟** |
| 开发者工作 量 | 由库处理大 部分的路由 逻辑，简化 开发者工作 | **需要开发者** **手动编写代** **码管理路由** |
| 社区和资源 | 广泛的社区 和资源，易 于获取帮助 和解决方案 | **相对较少的** **社区资源，** **通常需求独** **立解决** |
| 用户体验 | 通常能提供 更顺畅的用 户体验 | **可能因为实** **现不当导致** **的复杂性和** **用户体验问** **题** |

**45.html** **的行内元素和块级元素的区别【热度：796】**

HTML中的行内元素 (Inline elements) 和块级元素 (Block-level elements) 在布局行为、外观以及 如何参与文档流方面有所不同。以下是它们的主要区别：

|  |  |  |
| --- | --- | --- |
| **特性** | **块级元素** **(Block-**  **level**  **elements)** | **行内元素** **(Inline**  **elements)** |
| **布局** | 通常开始于 新的一行 | **在同一行内** **水平排列** |
| **宽度** | 默认填满父 容器宽度 | **宽度由内容** **决定** |
| **高度** | 可以设置高 度 | **高度通常由** **内容决定** |
| **外边距** **(margin)** | 可以设置上 下左右的外 边距 | **只能设置左** **右外边距** |
| **内边距**  **(padding)** | 可以设置上 下左右的内 边距 | **只能设置左** **右内边距** |
| **内容** | 可以包含其  **他块级或行** **内元素** | **通常包含文** **本或数据** |
| **堆叠方式** | **垂直堆叠** | **水平方堆放** **齐** |
| **盒模型** | **表现为完整** **的盒子模型** | **只表现部分** **盒子模型** |
| **换行** | 前后有换行 空间 | **默认没有前** **后换行空间** |
| **常见标签** | <div>,<p>, <section>, <h1>等 | **<span>,<a>,** **<strong>,<em>** **,<img>等** |



即使块级元素和行内元素默认特征不同，你还是可以通过CSS的displa y 属性来更改它们的行为。

例 如 ，d i s p l a y : i n l i n e ;会 让 块级元素表现得像行内元素，并且它们将在其父容器的同一行内显 示。另一方面，display:block; 会让行内元素表现得像块级元素。

**46.介绍一下requestldleCallback api 热度：290**

requestIdleCallback 是一个Web API,它允许开发者请求浏览器在主线程空闲时执行一些低 优先级的后台任务，这对于执行如分析、整理状态和数据等不紧急的任务是理想的。这种方法可以提 高用户的响应性和页面的整体性能。

以下是requestIdleCallback API的一些关键特点：

何时使⽤ requestIdleCallback

|  |
| --- |
| requestIdleCallback |

⽽不会明显影响⽤⼾体验。例如：

• 清理⼯作：如标记的 DOM 节点删除、数据的本地存储同步等。

• ⾮关键的解析：如解析⼤量数据。

• 状态更新：如发送不紧急的状态变更。 如何使⽤ requestIdleCallback

使⽤

|  |
| --- |
| requestIdleCallback |

, 你需要传递⼀个回调函数给它 ，此函数会在浏览器的空闲时间调

⽤ 。你可以指定⼀个超时参数 ，它定义了浏览器在“ 空闲期”最多可以花费的时间来执⾏你的回调。

|  |
| --- |
| 1 requestIdleCallback(myNonCriticalFunction, { timeout: 5000 }); |

• myNonCriticalFunction: 这是你想要浏览器在空闲时间执⾏的函数。

• timeout: ⼀个可选的参数 ，表⽰回调执⾏时间的上限（ 以毫秒为单位） 。如果超时 ，浏览器可能在 下次空闲机会进⾏执⾏。

回调函数参数

你的回调函数会接收到⼀个

|  |
| --- |
| IdleDeadline |

对象作为参数 ，通常命名为

|  |
| --- |
| deadline |

。这个对象包

含两个属性：

• didTimeout ⼀个布尔值 ，如果超时已经被触发为 true 。

• timeRemaining 返回当前空闲阶段剩余时间的函数 ，单位是毫秒。

|  |
| --- |
| 1  (deadline) { ((deadline.timeRemaining() > 0 || deadline.didTimeout) && someCondition()) {*// 执⾏⼯作直到时间⽤完或下次更新不是必* *要的*  2 }  3 *// 如果还有未完成的⼯作，可以请求下⼀次空闲周期* *(someCondition()){*  4 requestIdleCallback(myNonCriticalFunction);  5 }  6 } |

注意事项

•

|  |
| --- |
| requestIdleCallback |

不保证你的回调会在⼀个特定的时刻被调⽤ ，它只在浏览器需要的时

候调⽤ 。

特别适合那些不直接关联⽤⼾交互及响应的任务 ，这些任务可以延后执⾏

• 执⾏低优先级任务时 ，不应该太过频繁或执⾏时间太⻓ ，以免影响⻚⾯性能。

• 这个 API为了最⼤化性能优化 ，会强制性地结束你的任务 ，在不迟于指定的超时时⻓执⾏结束。 Cross-Browser Compatibility (跨浏览器兼容性)

你可能需要 polyfills（垫⽚库）来确保

|  |
| --- |
| requestIdleCallback |

的兼容性 ，因为它并不是在所有

浏览器中都有原⽣⽀持。

使⽤

|  |
| --- |
| requestIdleCallback |

, 开发者可以更好地利⽤浏览器的空闲序列来执⾏不紧急的任务 ，同

时保持⽤⼾交互的流畅度。

47. documentFragment api 是什么 ， 有哪些使⽤场景？热 度: 115

|  |
| --- |
| DocumentFragment |

是 Web API 中的⼀部分 ，它是

DOM

（⽂档对象模型） 的⼀个⾮常轻量级的

节点 ，代表⼀组 DOM 节点的集合。 它不是⼀个真实存在于 DOM 中的实体 ，因此被认为是“ 没有名 字” 的节点 ，或者说它不在⽂档的主体中渲染 ，通常⽤来作为临时的 DOM 节点仓库。

对于

|  |
| --- |
| DocumentFragment |

的⼀部分内容 ， 当它们在

|  |
| --- |
| DocumentFragment |

之外操作时 ，并不会

引起主 DOM 树的直接重排或重绘。然⽽ ，⼀旦你将整个

|  |
| --- |
| DocumentFragment |

插⼊到 DOM 的⼀个

永久节点上 ，那么在

|  |
| --- |
| DocumentFragment |

内进⾏的更改将会触发 DOM 的重新渲染。

DocumentFragment API 有⼏个关键的特点和⽤途：

1. 轻量级：

|  |
| --- |
| DocumentFragment |

不会引起布局重排 ，因为其不是真实渲染的⼀部分。

2. 节点集合：可以在

DOM

|  |
| --- |
| DocumentFragment |

中节点集合进⾏分组 ，这个集合可以⼀次性插⼊到

的某⼀部分中。

3. 性能优化：通过在⼀个

|  |
| --- |
| DocumentFragment |

中构建好⼀⼤块

DOM

树 ，然后将它整体插⼊到

主

DOM

中 ，从⽽减少重排次数 ，提⾼效率。

4. 事件不冒泡： 因为

|  |
| --- |
| DocumentFragment |

不是真实渲染的⼀部分 ，所以它的事件不会冒泡到上层

的 DOM 元素 ，除⾮它被插⼊到了

DOM

中。

使⽤场景

以下是⼀些使⽤

|  |
| --- |
| DocumentFragment |

的常⻅场景：

• 批量操作： 当你想要⼀次性添加多个节点到

DOM

树中时 ，使⽤

|  |
| --- |
| DocumentFragment |

可以将这

些节点预先堆放在⼀个轻量级对象中 ，然后⼀次性添加。

• 离屏操作：如果你需要创建复杂的

DOM

结构 ，可以通过

|  |
| --- |
| DocumentFragment |

在不触发⻚⾯重

排和重绘的情况下进⾏。

• 内容填充：在填充

DOM

元素内容之前 ，可以先创建⼀个

|  |
| --- |
| DocumentFragment |

完成所有节点的

添加和排序 ，然后把它添加到

DOM

树中。

• 避免内存泄漏：在某些情况下 ，它可以作为防⽌因移动节点⽽造成的内存泄漏的⼀个办法。 ⽰例代码

和

|  |
| --- |
| 1 *// 创建* *DocumentFragment* *fragment = document.createDocumentFragment();*  2  3 *//创建多个节点或元素* *div = document.createElement("div");* 4  p = document.createElement("p");  5  6 *// 将节点添加到* *DocumentFragment 上*  7 fragment.appendChild(div); 8 fragment.appendChild(p); 9  10 *// ⼀次性将* *DocumentFragment 添加到* *DOM 的某个部分* *body = document.querySelector("body");*  11 body.appendChild(fragment); 12  13 *// 这时* *div 和* *p 被添加⾄* *body 元素，⽽不会触发额外的布局重排。* |

|  |
| --- |
| DocumentFragment |

提供了⼀个⾼效的⽅式去操作

DOM

⽽不影响⻚⾯的渲染性能 ，在很多需要

进⾏批量 DOM 操作的场合⾮常有⽤ 。

48. git pull 和 git fetch 有啥区别？ 【热度: 355】

|  |
| --- |
| git pull |

|  |
| --- |
| git fetch |

是 Git 版本控制系统中的两个基本命令 ，它们都⽤于从远程仓库更新本

地仓库的信息 ，但执⾏的具体操作不同。 git fetch

•

|  |
| --- |
| git fetch |

下载远程仓库最新的内容到你的本地仓库 ，但它并不⾃动合并或修改你当前的⼯

作。 它取回了远程仓库的所有分⽀和标签（tags）。

• 运⾏

|  |
| --- |
| git fetch |

后 ，你可以在需要时⼿动执⾏合并操作（使⽤

|  |
| --- |
| git merge |

) 或者重新基于

远程仓库的内容进⾏修改。

•

|  |
| --- |
| fetch |

只是将远程变更下载到本地的远程分⽀跟踪副本中 ，例如

|  |
| --- |
| origin master |

/

。

git pull

•

|  |
| --- |
| git pull |

实际上是

|  |
| --- |
| git fetch |

操作之后紧跟⼀个

|  |
| --- |
| git merge |

操作 ，它会⾃动拉取远程

仓库的新变更 ，并尝试合并到当前所在的本地分⽀中。

• 当你使⽤

|  |
| --- |
| git pull |

, Git 会尝试⾃动合并变更。这可能会引起冲突（conflicts） ， 当然冲突需

要⼿动解决。

•

|  |
| --- |
| git pull |

等价于执⾏了

|  |
| --- |
| git fetch |

|  |
| --- |
| git merge FETCH\_HEAD |

的组合。

使⽤场景

• 当你仅仅想要查看远程仓库的变动⽽不⽴即合并到你的⼯作 ，可以使⽤

|  |
| --- |
| git fetch |

。

• ⽽当你想要⽴即获取远程的最新变动并快速合并到你的⼯作中 ，则可以使⽤

|  |
| --- |
| git pull |

。

总之，

|  |
| --- |
| git pull |

是⼀个更加「激进」 的命令 ，因为它⾃动将远程变更合并到你的当前分⽀ ，⽽

更加「谨慎」 ，它只下载变更到本地 ，不做任何合并操作。

和

|  |
| --- |
| git fetch |

49. 前端如何做 ⻚⾯主题⾊切换 【热度: 538】

⻚⾯主题⾊切换通常涉及到修改⽹⻚中的颜⾊⽅案 ，以提供不同的视觉体验 ，例如从明亮模式切换到 暗⿊模式。实现这⼀功能 ，可以通过配合使⽤ CSS、JavaScript 和本地存储来进⾏。 以下是实施⻚⾯ 主题⾊切换的⼏种⽅法：

使⽤ CSS ⾃定义属性

1. 定义⼀套主题变量：

1 :root {--primary-color: #5b88bd; */\* 明亮主题⾊* *\*/* 2 --text-color: #000; */\* 明亮主题⽂本颜⾊* *\*/*

3 }[data-theme="dark"] {--primary-color: #1e2a34; */\* 暗⿊主题⾊* *\*/* 4 --text-color: #ccc; */\* 暗⿊主题⽂本颜⾊* *\*/*

5 }

1. 应⽤⾃定义属性到CSS 规则中：

|  |
| --- |
| 1 body {background-color: var(--primary-color);color: var(--text-color);  2 } |

1. 使⽤ JavaScript 动态切换主题：

|  |
| --- |
| 1 () { root = document.documentElement;  (root.dataset.theme === "dark") {  2 root.dataset.theme = "light"; 3 }  {  4 root.dataset.theme = "dark"; 5 }  6 } |

使⽤ CSS 类切换

1. 为每个主题创建不同的CSS 类：

|  |
| --- |
| 1 .light-theme {--primary-color: #5b88bd;--text-color: #000;  2 }.dark-theme {--primary-color: #1e2a34;--text-color: #ccc;  3 } |

1. ⼿动切换 CSS 类：

|  |
| --- |
| 1 () { bodyClass = document.body.classList;  (bodyClass.contains("dark-theme")) {  2 bodyClass.replace("dark-theme", "light-theme");  3 }  {  4 bodyClass.replace("light-theme", "dark-theme");  5 }  6 } |

使⽤ LocalStorage 记录⽤⼾主题偏好

|  |
| --- |
| 1 *// 当⽤⼾切换主题时* *(){*  2 localStorage.setItem("theme", document.body.classList.contains("dark-theme")  ? "dark" : "light");  3 }  4  5 *// ⻚⾯加载时应⽤⽤⼾偏好* *(){* *preferredTheme = localStorage.getItem("theme");*  6  (preferredTheme === "dark") {document.body.classList.add("dark-theme");  7 }  {document.body.classList.remove("dark-theme");  8 }  9 }  10 applyThemePreference(); |

使⽤媒体查询⾃动应⽤暗⿊模式

某些现代浏览器⽀持 CSS 媒体查询

|  |
| --- |
| prefers-color-scheme |

。你可以使⽤这个特性来⾃动根据⽤

⼾的系统设置应⽤暗⿊模式或明亮模式 ，⽽⽆须 JavaScript：

1 @ (prefers-color-scheme: dark) {:root {--primary-color: #1e2a34; */\* 暗⿊主* *题⾊* *\*/*

2 --text-color: #ccc; */\* 暗⿊主题⽂本颜⾊* *\*/*

3 } 4 } 5

6 @ (prefers-color-scheme: light) {:root {--primary-color: #5b88bd; */\* 明亮主* *题⾊* *\*/*

7 --text-color: #000; */\* 明亮主题⽂本颜⾊* *\*/*

8 }

9 }

通过以上⽅法 ，开发⼈员能够为前端⻚⾯提供灵活的主题⾊切换功能 ，从⽽增强⽤⼾体验。

50. 前端视⻆ - 如何保证系统稳定性 【热度: 566】

前端视⻆来做稳定性 ， 本是⼀个开放性话题 ，这⾥没有统⼀的解法 ， 作者在此提供⼏个思路和反向：

1. 静态资源多备份（需要有备份）

2. ⾸屏请求缓存

3. 请求异常报警

4. ⻚⾯崩溃报警

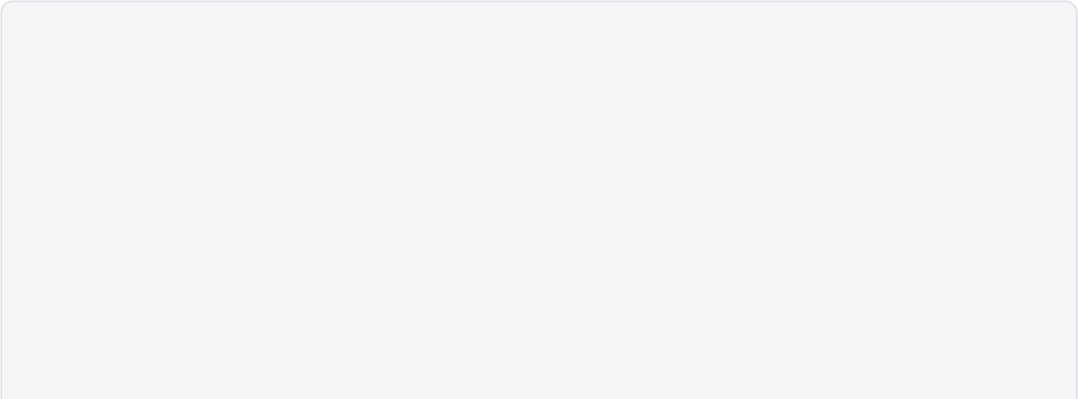
5. E2E 定时全量跑⽤例

51. 如何统计⻓任务时间、 ⻓任务执⾏次数 【热度: 489】

在 JavaScript 中 ，可以使⽤ Performance API 中的 PerformanceObserver 来监视和统计⻓任务

（Long Task） 。⻓任务是指那些执⾏时间超过 50 毫秒的任务 ，这些任务可能会阻塞主线程 ，影响⻚ ⾯的交互性和流畅性。

使⽤ PerformanceObserver 监听⻓任务



1 *//创建⼀个性能观察者实例来订阅⻓任务* *observer =*  *PerformanceObserver((list) => {* *(* *entry of list.getEntries()) {console.log("Long Task*

*detected:");console.log(*

2 Task Start Time: ${entry.startTime}, Duration: ${entry.duration}

3 );

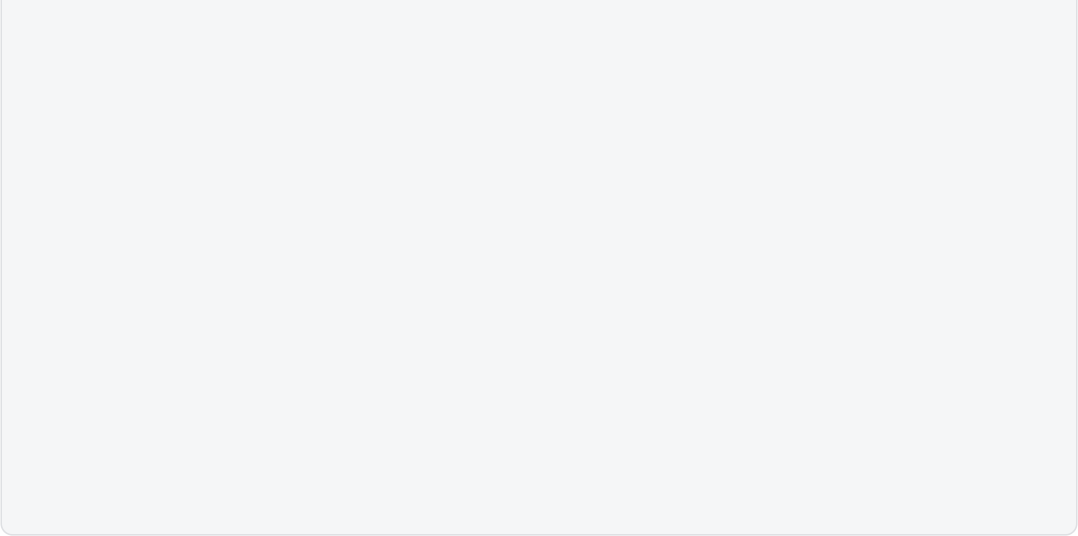
4 }

5 });

6

7 *// 开始观察⻓任务*

8 observer.observe({ entryTypes: ["longtask"] });



9

10 *// 启动⻓任务统计数据的变量* *longTaskCount = 0;* 11  totalLongTaskTime = 0;

12

13 *// 更新之前的性能观察者实例，以增加统计逻辑*

14 observer =  PerformanceObserver((list) => {

15 list.getEntries().forEach((entry) => {

16 longTaskCount++; *// 统计⻓任务次数*

17 totalLongTaskTime += entry.duration; *// 累加⻓任务总耗时// 可以在这⾥添加其他逻* *辑，⽐如记录⻓任务发⽣的具体时间等*

18 }); 19 }); 20

21 *// 再次开始观察⻓任务*

22 observer.observe({ entryTypes: ["longtask"] });

在上⾯的代码中 ，我们创建了⼀个

|  |
| --- |
| PerformanceObserver |

对象来订阅⻓任务。每当检测到⻓任务

时 ，它会向回调函数传递⼀个包含⻓任务性能条⽬的列表。在这个回调中 ，我们可以统计⻓任务的次 数和总耗时。

注意：

|  |
| --- |
| PerformanceObserver |

需要在⽀持该 API 的浏览器中运⾏。 截⾄到我所知道的信息

（2023 年 4 ⽉的知识截点） ，所有现代浏览器都⽀持这⼀ API ，但在使⽤前你应该检查⽤⼾的浏览器 是否⽀持这个特性。

以下是如何在实际使⽤中停⽌观察和获取当前的统计数据：

|  |
| --- |
| 1 *//停⽌观察能⼒*  2 observer.disconnect(); 3  4 *// 统计数据输出console.log(Total number of long tasks: ${longTaskCount});*  5 console.log(Total duration of all long tasks: ${totalLongTaskTime}ms); |

使⽤这种⽅法 ，你可以监控应⽤程序中的性能问题 ，并根据⻓任务的发⽣频率和持续时间进⾏优化。

52. V8 ⾥⾯的 JIT 是什么？ 【热度: 694】

在计算机科学中 ，JIT 是“ Just-In-Time”（即时编译） 的缩写 ，它是⼀种提⾼代码执⾏性能的技术。 具体来说 ，在 V8 引擎（Google Chrome 浏览器和 Node.js 的 JavaScript 引擎） 中 ，JIT 编译器在 JavaScript 代码运⾏时 ，将其编译成机器语⾔ ，以提⾼执⾏速度。

这⾥简要解释下 JIT 编译器的⼯作原理：

1. 解释执⾏：V8 ⾸先通过⼀个解释器（如 Ignition）来执⾏ JavaScript 代码。这个过程中 ，代码不 会编译成机器语⾔ ，⽽是逐⾏解释执⾏。 这样做的优点是启动快 ，但执⾏速度较慢。

2. 即时编译： 当代码被多次执⾏时 ，V8 会认为这部分代码是“ 热点代码”（ Hot Spot），此时 JIT 编 译器（如 TurboFan） 会介⼊ ，将这部分热点代码编译成机器语⾔ 。机器语⾔运⾏在 CPU 上⽐解释 执⾏要快得多。

3. 优化与去优化：JIT 编译器会对热点代码进⾏优化 ，但有时候它会基于错误的假设做出优化（例如 认为某个变量总是某种类型） 。如果后来的执⾏发现这些假设不成⽴ ，编译器需要去掉优化

（Deoptimize） ，重新编译。

JIT 编译器的⼀个关键优点是它能够在不牺牲启动速度的情况下 ，提供接近于或同等于编译语⾔的运⾏ 速度。这使得像 JavaScript 这样原本被认为执⾏效率较低的语⾔能够⽤于复杂的计算任务和⾼性能的 应⽤场景。

随着 V8 和其他现代 JavaScript 引擎的不断进步 ，JIT 编译技术也在持续优化 ，以提供更快的执⾏速度 和更⾼的性能。

53. ⽤ JS 写⼀个 cookies 解析函数 ， 输出结果为⼀个对象 【热度: 137】

以下是⼀个简单的 JavaScript 函数 ，⽤于解析当前⻚⾯的 cookie 并将它们存储到⼀个对象中：

|  |
| --- |
| 1 () {*//创建⼀个空对象来存储解析后的* *cookie* *cookiesObj = {};* 2 *// 获取* *cookie 字符串，然后分割每个键值对* *cookies = document.cookie.split(";");* 3 *//遍历每个键值对*  4 cookies.forEach(  (cookie) {*// 去除键值对前后的空格* *cleanCookie = cookie.trim();//找到键和值之间的等号位置* *separatorIndex =*  *cleanCookie.indexOf("=");*  5 *// 如果找不到等号，则不是有效的键值对，跳过当前循环* *(separatorIndex === -1)* *;* 6 *// 获取键名* *key = cleanCookie.substring(0, separatorIndex);// 获取值* *value*  *= cleanCookie.substring(separatorIndex + 1);* 7 *//解码因为* *cookie 键和值是编码过的*  8 key = decodeURIComponent(key);  9 value = decodeURIComponent(value);  10 *// 将解析后的值存储到对象中*  11 cookiesObj[key] = value;  12 });  13 *//返回解析后的* *cookie 对象* *cookiesObj;* 14 }  15  16 *//使⽤⽰例* *cookies = parseCookies();*  17 console.log(cookies); |

这个函数⾸先会以分号

;

分割

|  |
| --- |
| document .cookie |

字符串来得到各个 cookie 键值对 ，然后移除

键值对前后的任何空格。接着寻找每个键值对中的等号

=

位置 ，以此来分割键和值。最后 ，它会使

⽤

|  |
| --- |
| decodeURIComponent |

函数来解码键名和键值 ，因为通过

|  |
| --- |
| document .cookie |

读取的键名

和键值通常是编码过的。

调⽤

|  |
| --- |
| parseCookies |

函数将返回⼀个对象 ，其中包含了当前⻚⾯的所有 cookie ，键名和值都已被

解码。然后你可以像访问普通对象⼀样访问这些值 ，例如

cookies [ 'username ']

来获取

'username' 键对应的值。

54. vue 中 Scoped Styles 是如何实现样式隔离的 ， 原理是 啥？ 【热度: 244】

在 Vue 中，

.vue

单⽂件组件的

|  |
| --- |
| <style> |

标签可以添加⼀个

|  |
| --- |
| scoped |

属性来实现样式的隔离。

通过这个 scoped 属性 ，Vue 会确保样式只应⽤到当前组件的模板中 ，⽽不会泄漏到外部的其他组件

中。

这个效果是通过 PostCSS 在构建过程中对 CSS 进⾏转换来实现的。基本原理如下： Scoped Styles 的⼯作原理：

1. 当你为

|  |
| --- |
| <style> |

标签添加

|  |
| --- |
| scoped |

属性时 ，Vue 的加载器（ ⽐如

|  |
| --- |
| vue-loader |

) 会处理你

的组件⽂件。

2.

|  |
| --- |
| vue-loader |

使⽤ PostCSS 来处理

|  |
| --- |
| scoped |

的 CSS。 它为组件模板内的每个元素添加⼀个独

特的属性（如

|  |
| --- |
| data-v-f3f3eg9 |

) 。这个属性是随机⽣成的 ，确保唯⼀性（是在 Vue项⽬构建

过程中的 hash 值）。

3. 同时 ，所有的 CSS 规则都会被更新 ，以仅匹配带有相应属性选择器的元素。例如：如果你有⼀个

|  |
| --- |
| .button |

类的样式规则 ，它会被转换成类似

|  |
| --- |
| .button data-v-f3f3eg9 |

[ ]

的形式。这确保

了样式只会被应⽤到拥有对应属性的 DOM 元素上。 ⽰例

假设你在组件

|  |
| --- |
| MyComponent .vue |

内写了如下代码：

|  |
| --- |
| 1 <template<button class="btn"Click Me</button</template<style scoped  2 .btn {background-color: blue;}  3 </style |

|  |
| --- |
| vue-loader |

将处理上述代码 ，模板中的

|  |
| --- |
| <button> |

可能会渲染成类似下⾯的 HTML：

|  |
| --- |
| 1 <button class="btn" data-v-f3f3eg9Click Me</button |

CSS 则会被转换成：

|  |
| --- |
| 1 .btn[data-v-f3f3eg9] {background-color: blue;  2 } |

因此，

.btn

类的样式仅会应⽤于拥有

|  |
| --- |
| data-v-f3f3eg9 |

属性的

|  |
| --- |
| <button> |

元素上。

注意：

• Scoped styles 提供了样式封装 ，但不是绝对的隔离。⼦组件的根节点仍然会受到⽗组件的

|  |
| --- |
| scoped |

CSS 的影响。在⼦组件中使⽤

|  |
| --- |
| scoped |

可以避免这种情况。

• Scoped CSS 不防⽌全局样式影响组件。如果其他地⽅定义了全局样式 ，它们仍然会应⽤到组件 中。

• 当使⽤外部库的类名时 ， scoped 可能会导致样式不被应⽤ ，因为它会期望所有匹配规则的元素 都带有特定的属性。

总的来说 ，Scoped Styles 是 Vue单⽂件组件提供的⼀种⽅便且有效的样式封装⽅式 ，通过 PostCSS 转换和属性选择器来实现组件之间的样式隔离。

55. 样式隔离⽅式有哪些 【热度: 683】

样式隔离意味着在⼀个复杂的前端应⽤中保持组件的样式私有化 ，使得不同组件之间的样式不会互相 影响。 以下是⼀些在前端开发中实现样式隔离的常⻅⽅式：

1. CSS 模块（CSS Modules）

CSS 模块是⼀种在构建时将 CSS 类名局部作⽤域化的技术。每个类名都是独⼀⽆⼆的 ，通常通过添加 哈希值来实现。 当你导⼊⼀个 CSS 模块 ，会得到⼀个包含⽣成的类名的对象。这样可以确保样式的唯 ⼀性 ，并防⽌样式冲突。

2. Shadow DOM

Shadow DOM 是 Web 组件规范的⼀部分 ，它允许将⼀段不受外界影响的 DOM 附加到元素上。在 Shadow DOM 中的样式是局部的 ，不会影响外部的⽂档样式。

3. CSS-in-JS 库

CSS-in-JS 是⼀种技术 ，允许你⽤ JavaScript 编写 CSS ，并在运⾏时⽣成唯⼀的类名。 常⻅的库有 Styled-components、 Emotion 等。这些库通常提供⾃动的样式隔离 ，并且还⽀持主题化和动态样 式。

4. 使⽤ BEM（ Block Element Modifier）命名约定

BEM 是⼀种 CSS 命名⽅法 ，通过使⽤严格的命名规则来保持样式的模块化。通过将样式绑定到特定的 类名上 ，这种⽅法有助于防⽌样式泄露。

5. CSS Scoped

在 Vue.js 中 ，可以为

|  |
| --- |
| <style> |

标签添加

|  |
| --- |
| scoped |

属性 ，这将使⽤ Vue 的编译器来实现样式的作⽤

域。虽然这不是⼀个标准的 Web 特性 ，但它在 Vue⽣态系统中提供了很⽅便的样式隔离。

6. 使⽤ iframe

将组件或部分⻚⾯放在iframe 中可以提供⾮常强的样式和脚本隔离。尽管如此 ，iframe 通常不是最佳 选择 ，因为它们可能导致性能问题 ，⽽且使得组件间的沟通变得更加困难。

7. Web 组件

Web 组件利⽤了⾃定义元素和Shadow DOM 来创建封装的、可复⽤的组件。在 Web 组件中 ，可以使 ⽤ Shadow DOM 实现真正的样式和脚本隔离。

8. 封装的CSS 架构

准确使⽤ CSS 选择器 ，避免使⽤全局标签选择器或基础类 ，⽽是使⽤更具体的类选择器可以部分隔离 样式。此外 ，可以设置严格的 CSS 命名策略 ，不同模块使⽤不同的命名前缀 ，以避免名称冲突。

9. PostCSS 插件

使⽤ PostCSS 插件来处理 CSS ，可以⾃动添加前缀、变量等 ，从⽽实现隔离。例如 ，PostCSS 前缀插 件可以⾃动为CSS 类添加唯⼀的前缀。

各种⽅法有各⾃的优点和限制 ，选择哪种⽅法取决于项⽬的技术栈、 团队的熟悉程度以及特定的项⽬ 需求。

56. 在 JS 中 ， 如何解决递归导致栈溢出问题？ 【热度: 269】

在 JavaScript 中 ，递归如果执⾏过深 ，确实有可能导致“ 栈溢出(stack overflow)”错误 ，因为每次函 数调⽤都会向调⽤栈中添加⼀个新的帧 ，⽽每个线程的调⽤栈都有其最⼤容量限制。 当这个容量被超 过时 ，就会发⽣栈溢出。为了解决这个问题 ，你可以使⽤⼏种不同的⽅法：

尾调⽤优化（Tail Call Optimization）

在 ES6 中 ，引⼊了尾调⽤优化。这意味着如果函数的最后⼀个操作是返回另⼀个函数的调⽤ （即尾调 ⽤） ，那么这个调⽤可以在不增加新栈帧的情况下执⾏。但是 ，截⾄我知识更新的时间 ，⼤多数

JavaScript 引擎还没有实现这项优化 ，或者它在默认情况下并未激活。 转化为循环

⼤多数递归函数都可以重写为循环 ，这样可以避免调⽤栈问题。这种⽅法需要⼿动维护⼀个栈来存储 必要的状态信息 ，⽽这个栈通常是存储在堆（ heap） 中的数组 ，不受调⽤栈⼤⼩限制。

例如 ，下⾯递归计算阶乘的代码：

|  |
| --- |
| 1  (n) { (n === 1)  1; n \* factorial(n - 1);  2 } |

可以重写为循环形式：

|  |
| --- |
| 1  (n) { result = 1; ( i = 2; i <= n; i++) {  2 result \*= i;  3 }  result;  4 } |

⽤ Trampoline 函数

Trampoline 是⼀个⾼阶函数 ，使您可以在递归调⽤的情况下避免栈溢出。 它通过在每个递归步骤返回 ⼀个函数⽽不是值 ，然后持续调⽤这些函数 ，直到获得最终结果为⽌。

|  |
| --- |
| 1  (fn) {  (...args) { result =  fn.apply( , args);  2  ( result === "function") {  3 result = result();  4 }  5  result;  6 };  7 } |

然后 ，将原始递归函数改写为每次递归调⽤返回⼀个函数：

|  |
| --- |
| 1  (args) { (baseCase) {  finalValue;  2 }  {  () {  recursiveFunction(newArgs);  3 };  4 }  5 }  6  7  trampolinedFunction = trampoline(recursiveFunction); |

调⽤

|  |
| --- |
| trampolinedFunction |

会避免栈溢出 ，因为它不是真正的递归调⽤ ，只是同步循环调⽤那

些返回的函数。

⽣成器和 Promises

使⽤ ES6 的⽣成器(generator)和/或 Promises 也可以⽤来避免递归调⽤过深。这些特性可以帮助您⽣ 成异步递归调⽤ ，其允许事件循环（event loop）介⼊ ，避免单次执⾏过多递归调⽤造成的栈溢出。

使⽤异步递归

将递归函数改造成异步函数（async function） ，并确保每⼀次递归调⽤都有机会返回控制权给

JavaScript 事件循环 ，这可以通过

、

|  |
| --- |
| setTimeout |

或者

|  |
| --- |
| setImmediate |

|  |
| --- |
| process .nextTick |

（在 Node.js 环境下） 实现。

例如 ，可以将⼀个同步递归函数改写为：

|  |  |  |  |
| --- | --- | --- | --- |
| 1   (i) { (i < 0)  Promise.resolve();console.log("Recursion ", i);  {  2 setImmediate(() => {  3 resolve(recursiveAsyncFunction(i - 1));  4 });  5 });  6 } |  |  | Promise((resolve) => |

记得确保递归终⽌条件是正确的 ，否则即便以上⽅法也可能导致⽆限循环或者内存泄漏。每⼀种⽅法 都有其适⽤场景 ，具体使⽤哪⼀种⽅法取决于问题的具体需求。

57. 站点如何防⽌爬⾍？ 【热度: 554】

站点防⽌爬⾍通常涉及⼀系列技术和策略的组合。 以下是⼀些常⽤的⽅法：

1. 修改

|  |
| --- |
| robots .txt |

在站点的根⽬录下创建或修改

|  |
| --- |
| robots .txt |

⽂件 ，⽤来告知遵守该协议的爬⾍应该爬取哪些⻚⾯，

哪些不应该爬取。例如：

|  |
| --- |
| 1 User-agent: \*Disallow: / |

然⽽ ，需要注意的是遵守

|  |
| --- |
| robots .txt |

不是强制性的 ，恶意爬⾍可以忽视这些规则。

2. 使⽤ CAPTCHA

对于表单提交、登录⻚⾯等 ，使⽤验证码（CAPTCHA） 可以防⽌⾃动化脚本或机器⼈执⾏操作。

3. 检查⽤⼾代理字符串

服务器可以根据请求的⽤⼾代理（ User-Agent） 字符串来决定是否屏蔽某些爬⾍ 。但⽤⼾代理字符串 可以伪造 ，所以这不是⼀个完全可靠的⽅法。

4. 分析流量⾏为

分析访问者的⾏为 ，⽐如访问频率、访问⻚数、访问时⻓ ，并与正常⽤⼾的⾏为进⾏对⽐ ，从⽽尝试 检测和屏蔽爬⾍。

5. 使⽤ Web 应⽤防⽕墙（WAF）

许多 Web 应⽤防⽕墙提供⾃动化的爬⾍和机器⼈检测功能 ，可以帮助防⽌爬⾍。

6. 服务端渲染和动态 Token

⼀些⽹站使⽤ JavaScript 服务端渲染 ，或将关键内容（ ⽐如令牌） 动态地插⼊到⻚⾯中 ，这可以使得 ⾮浏览器的⾃动化⼯具获取⽹站内容变得更加困难。

7. 添加额外的 HTTP 头

⼀些站点要求每个请求都包括特定的 HTTP 头 ，这些头信息不是常规爬⾍会添加的 ，⽽是通过 JavaScript 动态添加的。

8. IP ⿊名单

如果探测到某个 IP 地址的不正常⾏为 ，就可以将该 IP 地址加⼊⿊名单 ，阻⽌其进⼀步的访问。

9. 限制访问速度

通过限制特定时间内允许的请求次数来禁⽌爬⾍执⾏⼤量快速的⻚⾯抓取。

10. API 限流

对 API 使⽤率进⾏限制 ，⽐如基于⽤⼾ 、IP 地址等实施限速和配额。

11. 使⽤ HTTPS

使⽤ HTTPS 加密您的⽹站 ，这可以避免中间⼈攻击 ，并增加爬⾍的抓取难度。

12. 更改⽹站结构和内容

定期更改⽹站的 URL 结构、 内容排版等 ，使得爬⾍开发⼈员需要不断更新爬⾍程序来跟进⽹站的改 动。

58. ts 项⽬中 ，如何使⽤ node\_modules ⾥⾯定义的全局类型 包到⾃⼰项⽬ src 下⾯使⽤？ 【热度: 377】

关键点在

|  |
| --- |
| types |

属性配置

在 TypeScript 项⽬中导⼊

|  |
| --- |
| node\_modules |

中定义的全局包 ，并在你的

s rc

⽬录下使⽤它 ，通常

遵循以下步骤：

1. 安装包：

使⽤包管理器如 npm 或 yarn来安装你需要的全局包。

npm install <package-name>

*或者*

yarn add <package-name>

1. 类型声明：

确保该全局包具有类型声明。如果该全局包包含⾃⼰的类型声明 ，则 TypeScript 应该能够⾃动找 到它们。如果不包含 ，则可能需要安装对应的 DefinitelyTyped 声明⽂件。

npm install @types/<package-name>

*或者，如果它是⼀个流⾏的库，⼀些库可能已经带有⾃⼰的类型定义。*

1. 导⼊包：

在 TypeScript ⽂件中 ，使⽤

|  |
| --- |
| import |

语句导⼊全局包。

• importas PackageName from "<package-name>";

***/****或者*import PackageName from "<package-name>";

1. tsconfig.json 配置：

确保你的

|  |
| --- |
| tsconfig . son |

j

⽂件配置得当 ，以便 TypeScript 能够找到

|  |
| --- |
| node\_modules |

中的

声明⽂件。

◦ 如果包是模块形式的 ，确保

|  |
| --- |
| "moduleResolution" |

设置为

|  |
| --- |
| "node" |

。

◦ 确保

中的

|  |
| --- |
| compilerOptions |

和

|  |
| --- |
| "types" |

|  |
| --- |
| "typeRoots" |

属性没有配置错误。

2. 使⽤全局包：

现在你可以在你的 s rc ⽬录中的任何⽂件⾥使⽤这个全局包。

记住 ，最好的做法是不要把包当成全局包来使⽤ ，即使它们是全局的。通过显式地导⼊所需的模块， 可以有助于⼯具如 linters 和 bundlers 更好地追踪依赖关系 ，并可以在以后的代码分析和维护中发挥 重要作⽤ 。

此外 ，全局变量或全局模块通常指的是在项⽬的多个部分中⽆需导⼊就可以直接使⽤的变量或模块。 如果你确实需要将某些模块定义为全局可⽤ ，并且⽆法通过导⼊来使⽤ ，你可能需要更新你的

TypeScript 配置⽂件 (

|  |
| --- |
| tsconfig . son |

j

) 来包括这些全局声明。但这通常不是⼀个推荐的做法，

因为它可能会导致命名冲突和代码可维护性问题。

59. 不同标签⻚或窗⼝间的 【主动推送消息机制】 的⽅式有哪 些？（不借助服务端） 【热度: 401】

在不借助服务器端的帮助下 ，实现不同标签⻚或窗⼝间的主动推送消息机制 ，可以使⽤以下客⼾端技 术：

作者备注：

这⾥要注意⼀下 ， 这⾥讨论的不是跨⻚签通信 ，⽽是跨⻚签主动推送信息 。如果仅仅是跨⻚签通 信 ， 那么浏览器的本地存储都可以都可以使⽤了。 所以排除了本地存储类 API 的介绍

BroadcastChannel API:

作者备注

这个很有意思 ， 有⼀个⽂章 ， 国内某⼤佬复刻了《跨窗⼝量⼦纠缠粒⼦效果》就是⽤的 这个 API

[https://juejin.cn/post/7307057492059471899](https://gitee.com/link?target=https%3A%2F%2Fjuejin.cn%2Fpost%2F7307057492059471899)

是⼀种在相同源的不同浏览器上下⽂之间实现简单⾼效通信的⽅法。这

|  |
| --- |
| BroadcastChannel API |

意味着它可以在同⼀⽹站的多个标签⻚或窗⼝之间发送消息。这是由 HTML5 规范引⼊的 ，⽤于改进

Web Workers 中的通信⽅法。

下⾯是如何使⽤

|  |
| --- |
| BroadcastChannel API |

的基本指南及⼏个⽰例。

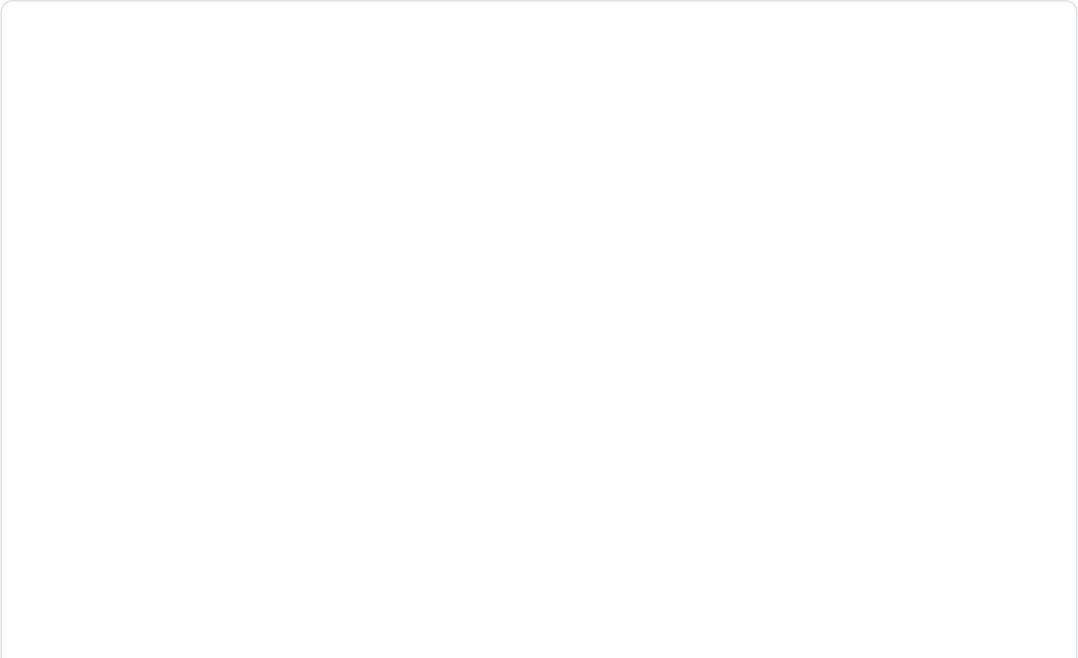
创建与发送消息

|  |
| --- |
| 1 *// 在任何⼀个* *tab 或* *iframe 中创建⼀个⼴播频道* *channel =*  *BroadcastChannel("my-channel-name");*  2  3 *//发送⼀个消息到频道*  4 channel.postMessage("Hello from a tab!"); |

监听消息

|  |
| --- |
| 1 *// 监听这个频道的消息*  2 channel.addEventListener("message",  (event) { (event.data ===  "Hello from a tab!") {console.log("Message received: ", event.data);  3 }  4 }); |

实现频道消息通信

假设你有两个标签⻚ ，并且你想更新每个标签⻚来显⽰另⼀个标签⻚中发⽣的事情 ，⽐如⽤⼾数量计 数器：

1 *// 在第⼀个标签⻚中*

2 self.addEventListener("load", () => { channel = 

BroadcastChannel("visitor-channel"); visitorCount = 0;

3 *// 定时发送随机的⽤⼾活动消息*

4 setInterval(  () {

5 visitorCount++;

6 channel.postMessage(

7 Visitor count increased to: ${visitorCount}

8 );

9 }, 5000);

10 });

11

12 *// 在另⼀个标签⻚中*

13 self.addEventListener("load", () => { channel = 

BroadcastChannel("visitor-channel");

14 *// 监听消息来更新⽤⼾数量*

15 channel.addEventListener("message",  (event) {

(event.data.startsWith("Visitor count")) {*// ⽤接收到的⽤⼾数量更新显⽰*

16 updateVisitorCountDisplay(event.data);

17 }

18 });

19 *// 这个⽅法将设置标签⻚上的⽤⼾计数显⽰* *(message) {// 这⾥写⽤于更新显⽰的代码console.log(message);*

20 }

21 });

在这个例⼦中 ，⼀个标签⻚通过定期发送新的消息来模拟⽤⼾活动的增加 ，这个消息在所有监听该频 道的上下⽂中传递。另⼀个或多个标签⻚将监听这个频道来接收和响应这些更新。

注意事项：

• 频道内的通信 仅在同源浏览器上下⽂（具有相同的协议、域名和端⼝号） 之间有效 ，也就是说 ，不 同的⽹站之间的通信是不被允许的 ，以保护每个⽹站的安全性。

• 频道中的通信是 单向的 ，你可以通过频道向所有连接 Service Workers:

利⽤ Service Workers ，各个标签⻚可以通过

clients .matchAll()

⽅法找到所有其他客⼾端

（如打开的标签⻚） ，然后使⽤

|  |
| --- |
| postMessage |

发送消息。

这个⽅法相⽐

|  |
| --- |
| BroadcastChannel |

提供了在后台运⾏脚本的能⼒ ，这些脚本可以在⽹络受限或没有⽹络的情况下运

更加灵活 ，因为 Service Workers 可以通过

|  |
| --- |
| Focus |

和

事件来控制⻚⾯的焦点和导航等。

|  |
| --- |
| Navigate |

|  |
| --- |
| ServiceWorkers |

⾏ 。当你⽤

|  |
| --- |
| ServiceWorkers |

进⾏⻚⾯间的通信 ，你可以利⽤它们来推送消息到打开的

（如浏览器标签⻚）。

|  |
| --- |
| Clients |

要使⽤

|  |
| --- |
| ServiceWorkers |

实现从不同 Tab 中主动推送信息 ，可以通过以下⼏个步骤：

1. 编写 ServiceWorker ⽂件

⾸先 ，创建名为  sw .js  的 ServiceWorker ⽂件。这个⽂件在你的⽹站⽬录下 ，会在⽤⼾访问⽹站时 注册并激活。

• ***/****sw.js*

self.addEventListener("message", (event) => {if (event.data === "New message from another tab") {

self.clients

.matchAll({

type: "window",

includeUncontrolled: true, })

.then((windowClients) => {

windowClients.forEach((client) => {

client.postMessage("New message for "client.id); });

}); }

});

2. 在主⻚⾯注册 ServiceWorker

在主⻚⾯ （ index.html） 通过 JavaScript 注册这个 ServiceWorker ⽂件。

|  |
| --- |
| 1 *// index.html* *("serviceWorker"*  *navigator) {*  2 navigator.serviceWorker  3 .register("/sw.js")  4 .then((registration) => {console.log("Service Worker registered with  scope:", registration.scope);  5 })  6 .catch((error) => {console.log("Service Worker registration failed:",  error);  7 });  8 } |

3. 监听

|  |
| --- |
| message |

事件

在主⻚⾯使⽤

|  |
| --- |
| navigator .serviceWorker .controller |

来检查是否已经有 ServiceWorker 主

动控制。

|  |
| --- |
| 1  (navigator.serviceWorker.controller) {*// Post a message to the ServiceWorker*  2 navigator.serviceWorker.controller.postMessage("This is from main page");  3 } |

4. 从其他 Tab 推送消息

在其他 Tab 上 ，⼀旦 ServiceWorker 被该⻚⾯控制后 ，可以通过同样的

|  |
| --- |
| postMessage |

⽅法发送消

息。

SharedWorker:

SharedWorker 提供了⼀种更传统的跨⽂档通信机制 ，在不同⽂档间共享状态和数据。你需要创建⼀

个

|  |
| --- |
| SharedWorker |

对象 ，并在所有的⽂档⾥监听来⾃该 worker 的消息。

简单场景的 SharedWorker 的使⽤步骤：

1. 创建和连接:

|  |
| --- |
| 1 *//创建⼀个* *SharedWorker，并指定要加载的脚本* *myWorker =*  *SharedWorker("worker.js");*  2 *// 开启端⼝通信*  3 myWorker.port.start(); |

1. 端⼝通信: 使⽤端⼝接收和发送消息

|  |
| --- |
| 1 *//发送数据给worker*  2 myWorker.port.postMessage({ command: "start", data: [1, 2, 3] }); 3  4 *// 监听来⾃worker的消息*  5 myWorker.port.onmessage =  (event) { (event.data)  {console.log("Result from worker:", event.data);  6 }  7 }; |

1. 实现 worker 逻辑:

在

|  |
| --- |
| worker . s |

j

内 ，通过

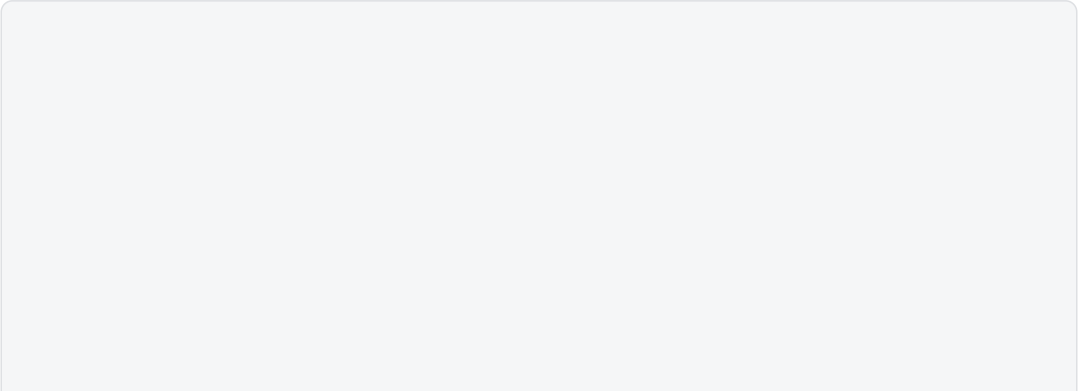
|  |
| --- |
| onconnect |

事件监听端⼝连接 ，并在使⽤

|  |
| --- |
| postMessage |

发送数据的

⻚⾯之间转发消息。



1 *// worker.js// ⾃⾝的事件监听器*

2 self.onconnect =  (event) { port = event.ports[0]; 3 *// 监听端⼝的消息*

4 port.onmessage =  (e) { (e.data.command === "start") { result =

someHeavyComputation(e.data.data);

5 port.postMessage({ result: result });

6 }

7 };

8 };

9

中的

10 *// 在这⾥执⾏⼀些开销较⼤的计算逻辑* *(data) {// 在这⾥进⾏*

*计算...*  *data.reduce(* *(previousValue, currentValue) {* *previousValue + currentValue;*

11 }, 0);

12 }



1. 通知其他⻚⾯更新:

当你希望基于上⽂提到的 SharedWorker 执⾏的计算结果通知其他所有的⻚⾯更新时 ，可以利⽤

|  |
| --- |
| SharedWorkerGlobalScope |

|  |
| --- |
| clients |

对象。

|  |
| --- |
| 1 *// 在* *worker.js 中*  2 self.addEventListener("message", (e) => { (e.data === "Update all clients") {*//遍历所有客⼾端*  3 self.clients.matchAll().then((clients) => {  4 clients.forEach((client) => {*// 发送消息更新它们*  5 client.postMessage("Please update your state");  6 });  7 });  8 }  9 }); |

使⽤ localStorage 的变更监听

虽然

|  |
| --- |
| localStorage |

没有直接提供跨标签⻚推送机制 ，但是可以使⽤

window .addEventListener( 'storage ', listener)

监听

|  |
| --- |
| storage |

事件 ，实现不同标

签⻚间的通信。

|  |
| --- |
| 1 *//标签⻚1修改了* *localStorage*  2 localStorage.setItem("someKey", "someValue"); 3  4 *// 其他标签⻚监听* *storage 事件window.addEventListener("storage",*  *(event) {* *(event.storageArea === localStorage && event.key === "someKey")*  *{console.log(event.newValue);*  5 }  6 }); |

使⽤ iframe 的 message 事件

如果排他性不是问题（所有标签⻚都属于同⼀客⼾端） ，可以使⽤ iframe 来传递消息 ，⽗窗⼝和

iframe 可以使⽤ DOM 中的

|  |
| --- |
| message |

事件系统相互通信。

|  |
| --- |
| message |

要使⽤

|  |
| --- |
| i frame |

的

事件实现不同⻚签之间的通信 ，你需要两个关键项的配合：⽗⻚⾯

和 i frame ⻚⾯之间的协调⼯作。这种通信⾮常灵活 ，因为你可以根据⾃⼰需要进⾏信息的发送和

监听。

⽰例步骤：

2. 创建⼀个⽗⻚⾯

在⽗⻚⾯中 ，我们创建⼀个

|  |
| --- |
| i frame |

并监听

|  |
| --- |
| message |

事件。

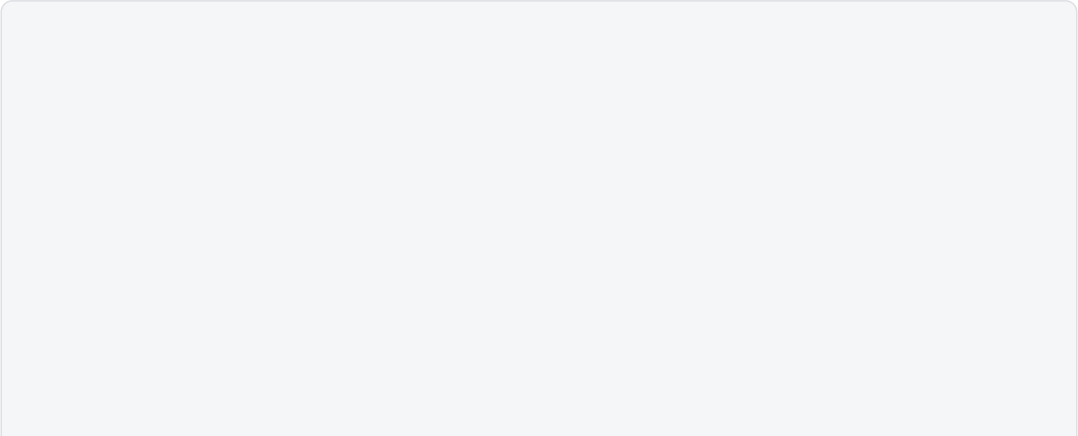
|  |
| --- |
| 1 *<!-- parent.html -->*<html lang="en"<head<meta charset="UTF-8" />  <meta name="viewport" content="width=device-width, initial-scale=1.0" />  <titleParent Page</title</head<body<iframe src="iframe.html"  style="display:none;"</iframe<script  2 *// 监听* *iframe 发送的* *message 事件*  3  .addEventListener("message", function () { ( .origin  !== "http://example.com") {*// 确保消息源是可信的* 4  ;  5 } ( .data &&  .data.greeting) {  6 console. ("Message received from iframe:",  .data);*// 如果iframe*  *向⽗⻚⾯问好（向⽗⻚⾯发送了⼀条消息）*  7 *// 假设我们还想再向iframe发送⼀些信息*  8 document.querySelector("iframe").contentWindow.postMessage(  9 {  10 response: "Hello iframe! This is the parent window speaking.",  11 },"http://example.com"  12 );  13 }  14 });</script</body</html |

3. 创建⼀个 iframe ⻚⾯

在

|  |
| --- |
| i frame .html |

⻚⾯中 ，我们需要发送消息到⽗⻚⾯并监听⽗⻚⾯的消息。



1 *<!-- iframe.html -->*<html lang="en"<head<meta charset="UTF-8" />

<titleIframe Page</title</head<body<script 2 *// 假设我们有⼀些需要发送到⽗⻚⾯的信息*

3 () {

4 parent.postMessage({ greeting: "Hello, I am the iframe!" },

"<http://example.com>");

5 }

6 *// 当⻚⾯加载完成后，发送消息*

7 window.onload =  () {

8 sendMessageToParent();

9 };

10 *// 监听来⾃⽗⻚⾯的消息*

11 window.addEventListener("message",  (event) { (event.origin

!== "http://example.com") {*// 反向验证消息源的可信度* 12  ;

13 } (event.data && event.data.response) {console.log("Message received

from parent:", event.data);*// 可根据消息实现特定的逻辑* 14 }

15 });</script</body</html

60. [React] 在 react 项⽬开发过程中 ， 是否可以不⽤ react- router ， 使⽤浏览器原⽣ history 路由来组织⻚⾯路由？

【热度: 632】

在 React 项⽬中 ，你完全可以不使⽤

|  |
| --- |
| react-router |

⽽是使⽤浏览器原⽣的

|  |
| --- |
| history |

API 来⼿

动管理路由。这通常会涉及以下⼏个步骤：

1. 使⽤

|  |
| --- |
| history.pushState |

()

和 history .replaceState()

⽅法来添加和修改浏览器历

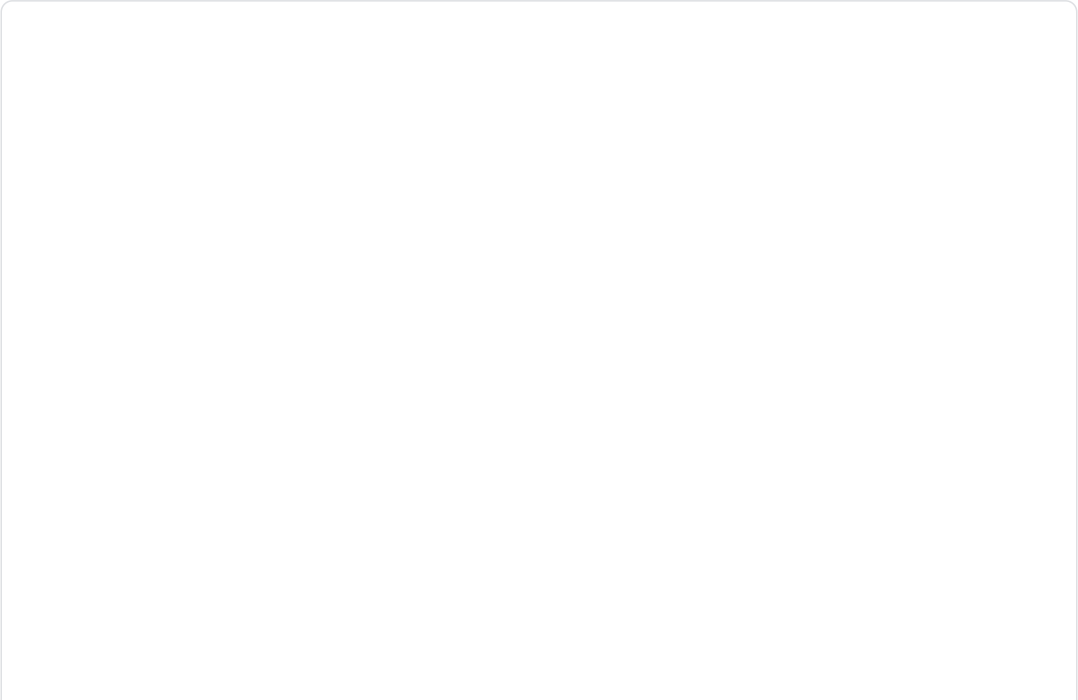
史条⽬ 。

2. 侦听

|  |
| --- |
| popstate |

事件来响应浏览器历史的变化。

3. 根据当前的 URL 状态 ，⼿动渲染对应的 React 组件。

例如 ，下⾯是⼀个简单的例⼦ ，演⽰了如何在没有

|  |
| --- |
| react-router |

的情况下使⽤原⽣

|  |
| --- |
| history |

API来管理路由。

1  App extends React.Component {

2 componentDidMount() {*// 当⽤⼾点击后退/前进按钮时触发路由变化window.onpopstate =*  *.handlePopState;// 初始⻚⾯加载时处理路由* *.route();*

3 }

4 handlePopState = () => {*// 处理路由变化* *.route();* 5 };

6 route() { path = window.location.pathname;*// 根据* *path 渲染不同的组件* *(path) {* *"/page1":// 渲染* *Page1 组件**;* *"/page2":// 渲染* *Page2 组件*

*;// 其他路由分⽀...**:// 渲染默认组件或404⻚⾯**;*

7 }

8 }

9 navigate = (path) => {*// 更新历史记录并触发路由变化*

*window.history.pushState(null, "", path);* *.route();*

10 };

11 render() {  (<div

12 <button onClick={() => this.navigate("/page1")}>Go to Page 1</button

13 <button onClick={() => this.navigate("/page2")}>Go to Page 2</button

14 {/\* 这⾥根据路由渲染对应的组件 \*/}</div

15 );

16 }

17 } 18

19 *// 实际的⻚⾯组件* Page1 = () => <divPage 1</div; Page2 = () => <divPage 2</div;

尽管⼿动管理路由是可能的 ，但使⽤

|  |
| --- |
| react-router |

这类专⻔设计的库通常会⼤⼤简化路由管理的

⼯作。 它为路径匹配、路由嵌套、重定向等提供了便利的抽象 ，并且和 React 的声明式⽅式很好地集 成在⼀起。如果不是为了特别的原因 ，通常推荐使⽤现成的路由库来管理 React 应⽤的路由 ，以避免 重新发明轮⼦。

61. 在表单校验场景中 ， 如何实现⻚⾯视⼝滚动到报错的位置 【热度: 248】

基本原理

⻚⾯是⽤⼾与程序进⾏交互的界⾯ ，在对应表单校验场景中 ，通常会因为有填写错误需要⽤⼾进⾏修 改。为了提⾼⽤⼾体验 ，可以将⻚⾯滚动⾄对应表单报错的位置 ，使得⽤⼾⽴即可⻅错误并进⾏修

改。这通常可以通过 JavaScript 编程实现。

要注意的是 ，实现滚动⾄错误表单 ，⼀般需要⼏个步骤：

1. 记录表单元素的位置：在表单提交前的适当时间⾥记录所有表单元素的错误位置。

2. 滚动到特定错误：错误发⽣时 ，滚动到第⼀个错误的表单元素位置。

3. 优化：可为同⼀元素多次错误滚动优化 ，避免不必要的⽤⼾⼲扰。

以下是这些步骤的代码⽰例 HTML:

|  |
| --- |
| 1 <form id="myForm" onsubmit="return false;"<input type="text" id="name"  name="name" /><input type="text" id="age" name="age" />*<!-- ... 其他表单元素* *... -->*<button type="submit" onclick="handleValidation()"Submit</button</form |

JavaScript:

1 *// ⼀个假设的表单验证函数* *(inputId) {// 调⽤此处的校验逻辑，返回* *是否存在错误// 这⾥以ID "inputId"来获取对应的DOM对象* *el =*

*document.getElementById(inputId);// 此处只是⽰例, 实际上应根据具体的校验逻辑返回⼀个* *布尔类型* *el.value === "预期值";*

2 }

|  |
| --- |
| 3  4  () { valid = true;  5 ["name", "age"].forEach((key) => {*// 进⾏校验判断* *(!validateInput(key)) {console.error(`Validation failed* *: ${key}`);*  6 *//标记校验失败*  7 valid = false;  8 *// 滚动到出现问题的元素位置* *element = document.getElementById(key);*  9 element.scrollIntoView({ block: "center", behavior: "smooth" });  10 *// 增加⼀些提⽰效果, ⽐如错误边框, 可按需实现// element.classList.add('error- highlight');*  11 }  12 });  13 *//检查是否验证失败，如果失败则不提交表单* *valid;* 14 }  15  16 *// 处理表单提交事件，与HTML中的onclick绑定*  *document.getElementById("myForm").addEventListener("submit", (e) => {*  17 e.preventDefault(); *// 阻⽌表单默认提交⾏为*  18 handleValidation();  19 }); |

62. 如何⼀次性渲染⼗万条数据还能保证⻚⾯不卡顿 【热度: 426】

原理其实就是 通过

|  |
| --- |
| requestAnimation Frame |

实现分块⼉加载。

requestAnimationFrame + fragment（ 时间分⽚）

既然定时器的执⾏时间和浏览器的刷新率不⼀致 ，那么我就可以⽤

|  |
| --- |
| requestAnimation Frame |

来解

决

|  |
| --- |
| requestAnimation Frame |



因此

也是个定时器 ，不同于

也是个宏任务 ，前阵⼦⾯试就被问到过这个

|  |
| --- |
| setTimeout |

, 它的时间不需要我们⼈为指

定 ，这个时间取决于当前电脑的刷新率 ，如果是 60Hz ，那么就是 16.7ms 执⾏⼀次 ，如果是 120Hz 那 就是 8.3ms 执⾏⼀次

|  |
| --- |
| requestAnimation Frame |

这么⼀来 ，每次电脑屏幕 16.7ms 后刷新⼀下 ，定时器就会产⽣ 20 个 的刷新保持了⼀致

li

, dom 结构的出现和屏幕

1  total = 100000;

2  ul = document.getElementById("container"); 3  once = 20;

4  page = total / once; 5

|  |
| --- |
| 6 (curTotal) { (curTotal <= 0)  ;  7  pageCount = Math.min(curTotal, once);  8 window.requestAnimationFrame(() => {  ( i = 0; i < pageCount; i++) {  li = document.createElement("li");  9 li.innerHTML = ~~(Math.random() \* total);  10 ul.appendChild(li);  11 }  12 loop(curTotal - pageCount);  13 });  14 }  15 loop(total); |

其实⽬前这个代码还可以优化⼀下 ，每⼀次

|  |
| --- |
| appendChild |

都是新增⼀个新的 li ，也就意味着需要

回流⼀次 ，总共⼗万条数据就需要回流⼗万次

此前讲回流的时候提出过虚拟⽚段

|  |
| --- |
| fragment |

来解决这个问题

|  |
| --- |
| fragment |

for

是虚拟⽂档碎⽚ ，我们⼀次

li

循环产⽣ 20 个

dom

的过程中可以全部把真实

挂载

到

|  |
| --- |
| fragment |

上 ，然后再把

|  |
| --- |
| fragment |

dom

挂载到真实

上 ，这样原来需要回流⼗万次 ，现在只需

要回流

100000 / 20

次

|  |
| --- |
| 1  total = 100000;  2  ul = document.getElementById("container"); 3  once = 20;  4  page = total / once; 5  6 (curTotal) { (curTotal <= 0)  ;  7  pageCount = Math.min(curTotal, once);  8 window.requestAnimationFrame(() => { fragment =  document.createDocumentFragment(); *// 创建⼀个虚拟⽂档碎⽚* *(* *i = 0; i < pageCount; i++) {* *li = document.createElement("li");*  9 li.innerHTML = ~~(Math.random() \* total);  10 fragment.appendChild(li); *// 挂到fragment上* 11 }  12 ul.appendChild(fragment); *// 现在才回流*  13 loop(curTotal - pageCount);  14 });  15 }  16 loop(total); |

进阶： 如果做到极致的话 ， 可以考虑通过动态计算渲染的量 ， ⼀次性渲染多少。

63. [webpack] 打包时 hash 码是如何⽣成的 【热度: 167】

Webpack 在打包过程中⽣成 hash 码主要⽤于缓存和版本管理。主要有三种类型的 hash 码：

1. hash ：是和整个项⽬的构建相关 ，只要项⽬⽂件有修改 ，整个项⽬构建的 hash 值就会更改。这意 味着任何⼀个⽂件的改动都会影响到整体的 hash 值。

2. chunkhash ：与 webpack 打包的 chunk 有关 ，不同的 entry 会⽣成不同的 chunkhash 值。例

如 ，如果你的配置⽣成了多个 chunk（例如使⽤了 code splitting） ，每个 chunk 的更新只会影响 到它⾃⾝的 chunkhash。

3. contenthash ：根据⽂件内容来定义 hash ，内容不变 ，则 contenthash 不变。这在使⽤诸如 CSS 提取到单独⽂件的插件时特别有⽤ ，因此只有当⽂件的内容实际改变时 ，浏览器才会重新下载⽂

件。

⽣成⽅式：

• hash 和 chunkhash 主要是通过某种 hash 算法（默认 MD5）来对⽂件名或者 chunk 数据进⾏编 码。

• contenthash 是通过构建时的 webpack 插件（如 mini-css-extract-plugin）来处理的 ，它会对⽂ 件内容进⾏ hash。

Hash 码的⽣成可以被 webpack 配置的 hashFunction ，hashDigest ，hashDigestLength 等选项影 响。例如 ，你可以选择不同的算法如 SHA256 或者 MD5 ，以及可以决定 hash 值的⻓度。

在 webpack 的配置⽂件中 ，可以通过如下⽅式设定 hash:

|  |
| --- |
| 1 output: {  2 filename: '[name].[chunkhash].js',  3 path: \_\_dirname + '/dist' 4 } |

webpack-dev-

这会将输出的⽂件名设置为⼊⼝名称加上基于每个 chunk 内容的hash。在使⽤

或者

server

|  |
| --- |
| webpack --watch |

时 ，不会⽣成实际的⽂件 ，所以这些 hash 值是在内存中计算并

关联的。

64. 如何从 0 到 1 搭建前端基建 【热度: 404】 如何从 0 到 1 搭建前端基建

有⼀个⾮常经典的⽂章 ， 直接参考即可： [⾮⼤⼚的我们，要如何去搞前端基建?](https://gitee.com/link?target=https%3A%2F%2Fjuejin.cn%2Fpost%2F7144881028661723167)

这⾥简单总结⼀下⽂章⾥⾯的要点 1.什么是基建？

2.为什么要做前端基建？ 业务复⽤；

提升研发效率；

规范研发流程；

团队技术提升；

团队的技术影响⼒； 开源建设；

3.前端基建如何推动落地？

• 要合适的同学（资源）

• 要解决的问题（ 问题）

• 要解决问题⽅案计划书（ ⽅案）

• 要具体执⾏的步骤（执⾏） 技术基建四⼤特性（切记）

• 技术的健全性

• 基建的稳定性

• 研发的效率性

• 业务的体验性

4.前端基建都有什么？

• 前端规范（Standard）

• 前端⽂档（ Document）

• 前端项⽬模板管理（Templates）

• 前端脚⼿架（CLI）

• 前端组件库（ UI Design）

• 前端响应式设计 or ⾃适应设计

• 前端⼯具库（类 Hooks / Utils）

• 前端⼯具⾃动化（Tools）

• 接⼝数据聚合（ BFF）

• 前端 SSR 推进

• 前端⾃动化构建部署（CI/CD）

• 全链路前端监控/数据埋点系统

• 前端可视化平台

• 前端性能优化

• 前端低代码平台搭建

• 微前端（ Micro App）

65. 你在开发过程中 ， 使⽤过哪些 TS 的特性或者能⼒？ 【热 度: 670】

直接上⼲货：

1. Utility Types（ ⼯具类型）：

◦ Partial<T>: 将类型 T 的所有属性变为可选。

◦ Required<T>: 将类型 T 的所有属性变为必选。

◦ Readonly<T>: 将类型 T 的所有属性变为只读。

◦ Record<K, T>: 创建⼀个具有指定键类型 K 和值类型 T 的新对象类型。

◦ Pick<T, K>: 从类型 T 中选择指定属性 K 形成新类型。

◦ Omit<T, K>: 从类型 T 中排除指定属性 K 形成新类型。

◦ Exclude<T, U>: 从类型 T 中排除可以赋值给类型 U 的类型。

◦ Extract<T, U>: 从类型 T 中提取可以赋值给类型 U 的类型。

◦ NonNullable<T>: 从类型 T 中排除 null 和 undefined 类型。

◦ ReturnType<T>: 获取函数类型 T 的返回类型。

◦ Parameters<T>: 获取函数类型 T 的参数类型组成的元组类型。

2. 条件判定类型：

◦ Conditional Types（条件类型） : 根据类型关系进⾏条件判断⽣成不同的类型。

◦ Distribute Conditional Types（分布式条件类型） : 分发条件类型 ，允许条件类型在联合类型上 进⾏分发。

3. Mapped Types（ 映射类型） ：根据已有类型创建新类型 ，通过映射类型可以⽣成新的类型结构。

4. Template Literal Types（模板⽂字类型）：使⽤字符串模板创建新类型。

5. 类型推断关键字：

◦ keyof 关键字：关键字允许在泛型条件类型中推断类型变量。

◦ instanceof：运算符⽤于检查对象是否是特定类的实例。

◦ in ：⽤于检查对象是否具有特定属性。

◦ type guards ：类型守卫是⾃定义的函数或条件语句 ，⽤于在代码块内缩⼩变量的类型范围。

◦ as ：⽤于类型断⾔ ，允许将⼀个变量断⾔为特定的类型。

66. JS 的加载会阻塞浏览器渲染吗？ 【热度: 243】

JavaScript 的加载、解析和执⾏默认情况下会阻塞浏览器的渲染过程。这是因为浏览器渲染引擎和 JavaScript 引擎是单线程的 ，并且⼆者共享同⼀个线程。JavaScript 在执⾏时会阻⽌ DOM 构建 ，因 为 JavaScript 可能会修改 DOM 结构（例如添加、修改或删除节点） 。出于这个原因 ，浏览器必须暂 停 DOM 的解析和渲染 ，直到 JavaScript 执⾏完成。

默认情况下 ， 当浏览器遇到⼀个

|  |
| --- |
| <script> |

标签时 ，会⽴即停⽌解析 HTML ，转⽽下载和执⾏脚本，

然后再继续 HTML 的解析和渲染。这意味着在 HTML ⽂档中的 JavaScript 脚本的下载和执⾏过程中， ⻚⾯的渲染是被阻塞的。

不过 ，你可以⽤下⾯⼏种⽅法调整脚本的加载和执⾏⾏为 ，以减少对浏览器渲染过程的阻塞：

1. 异步脚本（async）：

在

|  |
| --- |
| <script> |

标签中使⽤

|  |
| --- |
| async |

属性可以使得脚本的加载变成异步操作。 当使⽤

|  |
| --- |
| async |

属性

时 ，浏览器会在后台进⾏下载 ，但脚本的执⾏还是会阻塞 DOM 渲染。

|  |
| --- |
| 1 <script async src="script.js"</script |

1. 使⽤ async时 ，脚本会在下载完成后尽快执⾏ ，这可能会在⽂档解析完成之前或之后。

2. 延迟脚本（defer）：

defer 属性使得脚本在 HTML 解析完成之后、 DOMContentLoaded 事件触发之前执⾏ ，不阻塞 HTML 的解析。

|  |
| --- |
| 1 <script defer src="script.js"</script |

1. 使⽤

|  |
| --- |
| defer |

, 脚本的执⾏顺序将按照它们在 DOM 中出现的顺序执⾏。

2. 动态脚本加载：

你可以使⽤ JavaScript 动态创建

|  |
| --- |
| <script> |

元素并添加到 DOM 中 ，这允许你控制脚本的加载和

执⾏时机。

|  |
| --- |
| 1  script = document.createElement("script");  2 script.src = "script.js";  3 document.body.appendChild(script); |

1. 移动脚本位置：

将脚本放在 HTML 的底部 ，即

|  |
| --- |
| <body> |

标签关闭之前 ，⽽不是放在

|  |
| --- |
| <head> |

中 ，可以让⻚⾯内容

先加载显⽰ ，从⽽减少⽤⼾对加载过程的可感知时间。

现代 Web 开发中通常推荐使⽤

|  |
| --- |
| async |

|  |
| --- |
| defer |

属性 ，提⾼⻚⾯加载性能 ，尤其是对于那些需要从

外部服务器加载的⼤型 JavaScript 库来说尤为关键。

67. 浏览器对队头阻塞有什么优化？ 【热度: 368】

队头阻塞（ Head-of-Line Blocking ，缩写 HoLB） 问题主要发⽣在⽹络通信中 ，特别是在使⽤

HTTP/1.1 和以前版本时 ，在⼀个 TCP 连接中同⼀时间只能处理⼀个请求。即使后续的请求已经准备好 在客⼾端 ，它们也必须等待当前处理中的请求完成后才能被发送。这会延迟整个⻚⾯或应⽤的⽹络请 求 ，降低性能。

现代浏览器和协议已经实施了多种优化措施来减少或解决队头阻塞问题：

1. HTTP/2：

为了解决 HTTP/1.x 的诸多问题 ，包括队头阻塞问题 ，HTTP/2 引⼊了多路复⽤ （ multiplexing）功 能。这允许在同⼀ TCP 连接上同时传输多个独⽴的请求-响应消息。与 HTTP/1.1 相⽐ ，HTTP/2 在 同⼀个连接上可以并⾏处理多个请求 ，⼤⼤减少了队头阻塞的问题。

2. 服务器推送：

HTTP/2 还引⼊了服务器推送（server push）功能 ，允许服务器主动发送多个响应到客⼾端 ，⽽不 需要客⼾端明确地为每个资源提出请求。这提⾼了⻚⾯加载的速度 ，因为相关资源可以被预先发送 ⽽⽆需等待浏览器请求。

3. 域名分散（ Domain Sharding）：

这种技术常⽤于 HTTP/1.1 中 ，通过创建多个⼦域 ，使得浏览器可以同时开启更多的 TCP 连接来加 载资源。虽然这种⽅法可以在⼀定程度上减轻队头阻塞 ，但它增加了复杂性 ，并且在 HTTP/2 中由 于多路复⽤功能变得不再必要。

4. 连接重⽤ （Connection Reuse）：

这是 HTTP/1.1 中的⼀个特性 ，即持久连接（ Persistent Connections） ，允许在⼀次 TCP 连接中 发送和接收多个 HTTP 请求和响应 ，⽽⽆需开启新的连接 ，从⽽减少了TCP 握⼿的开销并提升了效 率。

5. 资源优化：

减少资源的⼤⼩通过压缩（如 GZIP） ，优化图⽚ ，减少 CSS 和 JavaScript ⽂件的⼤⼩等 ，可以减 少队头阻塞的影响 ，因为⼩资源⽂件传输更快。

6. 优先级设置：

HTTP/2 允许设置资源的加载优先级 ，使得关键资源（如 HTML ，CSS ，JavaScript） 可以⽐不那么 重要的资源（如图⽚ ，⼴告） 更早加载。

7. 预加载：

浏览器可以通过使⽤

|  |
| --- |
| <link rel="preload"> |

标签预加载关键资源 ，例如字体⽂件和关键脚

本 ，这样可以确保它们在主要内容加载之前已经准备好。

或

8. HTTP/3 和 QUIC 协议：

HTTP/3 是未来的推进⽅向 ，它基于 QUIC 协议 ，⼀个在 UDP 之上的新传输层协议 ， 旨在进⼀步减 少延迟 ，解决 TCP/IP 协议的队头阻塞问题。

总的来说 ，HTTP/2 的特性如多路复⽤ 、服务器推送和优先级设置都有助于减少队头阻塞。⽽ HTTP/3 的引⼊可能会在未来为⽹络通信带来根本性的变化。在使⽤ HTTP/2、 HTTP/3 和浏览器级别的优化 时 ，⽹⻚开发者也需注意资源加载优化的最佳实践 ，以更全⾯地应对队头阻塞问题。

68. Webpack 项⽬中通过 script 标签引⼊资源 ，在项⽬中如何 处理? 【热度: 100】

在使⽤ Webpack 打包的项⽬中 ，通常资源（如 JavaScript、CSS、 图⽚等） 会被 Webpack 处理 ，因 为 Webpack 的设计初衷就是将所有资源视为模块 ，并进⾏有效的管理和打包。但有时候可能需要通

过

|  |
| --- |
| <script> |

标签直接引⼊资源 ，这通常有两种情况：

1. 在 HTML ⽂件中直接引⼊：

可以在项⽬的 HTML ⽂件中直接使⽤

|  |
| --- |
| <script> |

标签来引⼊外部资源：

|  |
| --- |
| 1 *<!-- 若要使⽤* *CDN 上托管的库* *-->*<script src="https://cdn.example.com/library.js" </script |

1. 这种⽅法简单直接 ，但要记住 ， 由于这些资源不会被 Webpack 处理 ，它们不会被包含在 Webpack 的依赖图中 ，并且也不会享受到 Webpack 的各种优化。

2. 使⽤ Webpack 管理：

如果想要 Webpack 来处理这些通过

|  |
| --- |
| <script> |

引⼊的资源 ，可以使⽤⼏种插件和加载器：

◦

|  |
| --- |
| html-webpack-plugin |

可以帮助你⽣成⼀个 HTML ⽂件 ，并在⽂件中⾃动引⼊ Webpack

打包后的 bundles。

◦

|  |
| --- |
| externals |

配置允许你将⼀些依赖排除在 Webpack 打包之外 ，但还是可以通过

|  |
| --- |
| require |

或

|  |
| --- |
| import |

引⽤它们。

◦

|  |
| --- |
| script-loader |

可以将第三⽅全局变量注⼊的库当作模块来加载使⽤ 。

3. 例如 ，使⽤

和

|  |
| --- |
| html-webpack-plugin |

|  |
| --- |
| externals |

, 你可以将⼀个库配置为 external ，然后

通过

|  |
| --- |
| html-webpack-plugin |

将其引⼊：

1 *// webpack.config.js ⽂件* *HtmlWebpackPlugin = require("html-webpack- plugin");*

2

3 module.exports = {*// ...*

4 externals: {

5 libraryName: "LibraryGlobalVariable",

6 },

7 plugins: [ HtmlWebpackPlugin({

8 template: "src/index.html",

9 scriptLoading: "blocking", *// 或者* *'defer'*

10 }),

11 ],

12 };

1. 然后 ，在你的

|  |
| --- |
| index .html |

模板⽂件中可以这样引⼊资源：

|  |
| --- |
| 1 <script src="https://cdn.example.com/library.js"</script |

1. 使⽤

|  |
| --- |
| externals |

的⽅法能让你在 Webpack 打包的模块代码中⽤正常的

或

|  |
| --- |
| import |

|  |
| --- |
| require |

语句来引⽤那个全局变量：

|  |
| --- |
| 1 *//你的* *JavaScript 代码⽂件中* *Library*  *"libraryName"; // 虽然定义了* *external，Webpack依然会处理这个import* |

应根据项⽬需求和现有的架构来决定使⽤哪种⽅法。上述两种⽅法中 ，第⼆种可以更好地利⽤ Webpack 的功能 ，第⼀种则更加简单直接。

69. 应⽤上线后 ， 怎么通知⽤⼾刷新当前⻚⾯？ 【热度: 466】

⾸先第⼀个问题

⽤⼾在没有⻚⾯刷新的情况下 ， 如何去感知前端静态资源已经发⽣了更新？

⾸先要做静态资源版本管理。 这个版本直接给到 html 模板即可 ， 其他 link 打包的资源还是以哈希 code 作为⽂件名称后缀。

就类似于这样⼦的

|  |
| --- |
| 1 xxx.1.0.0.html --> vender.hash\_1.js、 vender.hash\_2.js、 vender.hash\_3.js、 vender.hash\_1.css  2 xxx.1.0.1.html --> vender.hash\_a.js、 vender.hash\_b.js、 vender.hash\_c.js、  vender.hash\_d.css |

如何主动推送给客⼾端

这个实现⽅式就⾮常的多了 ，我这⾥建议让服务端来做处理

因为我们前端静态资源打包之后 ， ⼤多数会上传到云存储服务器上 ， 或者甚⾄是 服务器本地 也⾏。 这个时候 ， 后端给⼀个定时任务 ， ⽐如 1 分钟去执⾏⼀次 ， 看看是否有新的 html 版本的内容⽣成。 如果有新的 html 版本内容⽣成 ， 且当前⽤⼾访问的还是旧版本 ， 那么直接发⼀个服务端信息推送即 可（SSE 允许服务器推送数据到浏览器）。

这样做成本是最低的 ， 甚⾄可以说是⼀劳永逸。 前端是没有任何负债 ， 没有任何性能问题。

那是否还有别的处理⽅式呢？ 当然是有的。

1. WebSockets：

通过 WebSocket 连接 ，服务器可以实时地向客⼾端发送消息 ，包括静态资源更新的通知。收到消息 后 ，客⼾端可以采取相应的措施 ，⽐如显⽰⼀个提⽰信息让⽤⼾选择是否重新加载⻚⾯。

1. Service Workers(推荐)：

Service workers 位于浏览器和⽹络之间 ，可以控制⻚⾯的资源缓存。 它们也可⽤于检测资源更新 ， 当 检测到静态资源更新时 ，可以通过推送通知或在⽹站上显⽰更新提⽰。

1. 轮询：

客⼾端⽤ JavaScript 定时发送 HTTP 请求到服务器 ，查询版本信息。如果检测到新版本 ，可以提醒⽤ ⼾或⾃动刷新资源。

在绝⼤多数情况下 ，使⽤ Service Workers 可能是最稳妥的做法 ，因为它不仅提供了资源缓存和管理 的能⼒ ，⽽且也可以在后台做资源更新的检查 ，即使⽤⼾没有开启⽹⻚也能实现通知和更新的功能。 当然 ，选择哪种⽅案还需考虑应⽤的需求、⽤⼾体验和实现复杂度等因素。

70. Eslint 代码检查的过程是啥？ 【热度: 111】

ESLint 是⼀个插件化的静态代码分析⼯具 ，⽤于识别 JavaScript 代码中的问题。 它在代码质量和编码 ⻛格⽅⾯有助于保持⼀致性。代码检查的过程通常如下：

1. 配置：

⾸先需要为 ESLint 提供⼀套规则 ，这些规则可以在

|  |
| --- |
| .eslint rc |

配置⽂件中定义 ，或者在项⽬的

|  |
| --- |
| package . son |

j

⽂件中的

|  |
| --- |
| eslintConfig |

字段⾥指定。规则可以继承⾃⼀套已有的规则集，

如

|  |
| --- |
| eslint :recommended |

, 或者可以是⼀个流⾏的样式指南 ，如

|  |
| --- |
| airbnb |

。也可以是⾃定义

的规则集。

2. 解析：

当运⾏ ESLint 时 ，它会使⽤⼀个解析器（如espree ，默认的解析器）来解析代码 ，将代码转换

成⼀个抽象语法树（AST） 。AST 是代码结构的⼀个树状表⽰ ，能让 ESLint 理解代码的语义结构。

3. 遍历：

⼀旦代码被转换成 AST ，ESLint 则会遍历该树。 它会查找树的每个节点 ，检查是否有任何规则适⽤ 于该节点。在遍历过程中 ，如果发现违反了某项规则 ，ESLint 将记录⼀个问题（通常称为“ lint 错 误”）。

4. 报告：

在遍历完整个 AST 之后 ，ESLint 会⽣成⼀份报告。这份报告详细说明了它在代码中找到的任何问 题。这些问题会被分类为错误或警告 ，根据配置设置的不同 ，某些问题可能会阻⽌构建过程或者被 忽略。

5. 修复：

对于某些类型的问题 ，ESLint 提供了⾃动修复的功能。这意味着你可以让 ESLint 尝试⾃动修复它 所发现的问题 ，不需⼈⼯⼲预。

6. 集成：

ESLint 可以集成到IDE 中 ，这样就可以在代码编写过程中即时提供反馈。 它也可以被集成到构建⼯ 具如 Webpack 或任务运⾏器 Grunt、Gulp 中 ，作为构建过程或提交代码到版本控制系统前的⼀个 步骤。

通过以上步骤 ，ESLint 帮助开发者在编码过程中遵循⼀致的⻛格和避免出现潜在的错误。

71. HTTP是⼀个⽆状态的协议 ，那么Web应⽤要怎么保持⽤⼾ 的登录态呢？ 【热度: 1,092】

⼤家都知道 ， HTTP 是⼀个⽆状态的协议 ，那么Web应⽤要怎么保持⽤⼾的登录态呢？

如果你对 cookie ， session 和 token 的优缺点不太明⽩ ，或者你想知道在实际中到底怎么实现 登录态 ，那么本⽂将⾮常适合你,本⽂将以发展历程为顺序为⼤家介绍 cookies ， session 以及

token 的优势和缺点。 知识点：

1. cookie ， session ， token(json web token,jwt) 的区别

2. node 中 jwt 的应⽤

我们站在服务器这⼀端 ，⼀个⽤⼾请求过来怎么判断他有没有登录呢？

在验证⽤⼾名和密码之后 ，我们可以发给客⼾端⼀个凭证(isLogin = true) ，如果请求中有这个凭证， 那么他就是登陆之后的⽤⼾ 。 cookie 和 session 的区别在于 ，凭证的存储位置。换⾔之 ，如果凭

证存储在客⼾端 ，那就是 cookie 。如果凭证存储在服务端 ，那就是 session 。 客⼾端存储（cookie）

cookie 其实是 HTTP 头部的⼀个字段 ，本质上可以存储任何信息 ，早年⽤于实现登录态 ，所以有了 ⼀层别的含义⸺客⼾端存储。把凭证存储到 cookie 中 ，每次浏览器的请求会⾃动带上 cookie ⾥ 的凭证 ，⽅便服务端校验

请求调⽤ /login 接⼝ ，验证通过后颁发的登录凭证 isLogin=true 但是这样⾯临的问题是：

⽤⼾本⼈可以通过修改 document.cookie="isLogin = true" 伪造登陆凭证:

服务端存储（session）

session 本意是指客⼾端与服务器的会话状态 ， 由于凭证存储到了服务端 ，后来也把这些存在服务 端的信息称为 session 。

现在服务器决定⾃⼰维护登录状态 ，仅发给客⼾端⼀个 key ，然后在⾃⼰维护⼀个 key-value

表 ，如果请求中有 key ，并且在表中可以找到对应的 value ，则视为合法请求调⽤ /login 接⼝ ，验证 通过后颁发 sessionID

这样即使⾃⾏修改了 sessionID ，也没有对应的记录 ，也⽆法获取数据。

session 是⼀个好的解决⽅案 ，但是他的问题是：如果存在多个服务器如负载均衡时 ，每个服务器 的状态表必须同步 ，或者抽离出来统⼀管理 ，如使⽤ Redis 等服务。

Token

还有其他的⽅法可以实现登陆态吗？

cookie ⽅法不需要服务器存储 ，但是凭证容易被伪造 ，那有什么办法判断凭证是否伪造呢？ 和 HTTPS ⼀样 ，我们可以使⽤签名的⽅式帮助服务器校验凭证。

JSON Web Token（简称JWT） 是以 JSON 格式存储信息的 Token

1. 头部存储 Token 的类型和签名算法（上图中 ，类型是 jwt ，加密算法是 HS256 ）

2. 负载是 Token 要存储的信息（上图中 ，存储了⽤⼾姓名和昵称信息）

3. 签名是由指定的算法 ，将转义后的头部和负载 ，加上密钥⼀同加密得到的。 最后将这三部分⽤ . 号连接 ，就可以得到了⼀个 Token 了。

使⽤ JWT 维护登陆态 ，服务器不再需要维护状态表 ，他仅给客⼾端发送⼀个加密的数据 token ，每 次请求都带上这个加密的数据 ，再解密验证是否合法即可。 由于是加密的数据 ，即使⽤⼾可以修改， 命中⼏率也很⼩。

客⼾端如何存储 token 呢？

1. 存在 cookie 中 ，虽然设置 HttpOnly 可以有效防⽌ XSS 攻击中 token 被窃取 ，但是也就意 味着客⼾端⽆法获取 token 来设置 CORS 头部。

2. 存在 sessionStorage 或者 localStorage 中 ，可以设置头部解决跨域资源共享问题 ，同时 也可以防⽌ CSRF ，但是就需要考虑 XSS 的问题防⽌凭证泄露。

Node 中 JWT 的使⽤

在 Node 中使⽤ JWT 只需要两步：

第⼀步 ，在你的 /login 路由中使⽤ jsonwebtoken 中间件⽤于⽣成 token ：

1 const jwt = require('jsonwebtoken')

2 let token = jwt.sign({

3 name: user name4 }, config.secret, {

5 expiresIn: '24h'

6 })

7 res.cookie('token', token)

具体使⽤⽅法请查看 jsonwebtoken 的 Github

第⼆步 ，在 Node 的⼊⼝⽂件 app.js 中注册 express-jwt 中间件⽤于验证 token ： 1 const expressJwt = require('express-jwt')

2 app.use(expressJwt({

3 secret: config.secret,

4 getToken: (req) => {

5 return req.cookies.token || null 6 }

7 }).unless({

8 path: [

9 '/login'

10 ]

11 }))

如果 getToken 返回 null ，中间件会抛出 UnauthorizedError 异常： 1 app.use(function (err, req, res, next) {

2 ***/****当token验证失败时会抛出如下错误* 3 if (err.name === 'UnauthorizedError') { 4 res.status(401).json({

5 status: 'fail',

6 message: '⾝份校验过期 ，请重新登陆'

7 });

8 }

9 });

具体使⽤语法参考 express-jwt 的 Github

如何实现单点登录

假设我们在电脑和⼿机都使⽤同⼀个⽤⼾登陆 ，对于服务器来说 ，这两次登陆⽣成的 token 都是合法

的 ，尽管他们是同⼀个⽤⼾ 。所以两个 token 不会失效。要实现单点登陆 ，服务器只需要维护⼀张

userId 和token 之间映射关系的表。每次登陆成功都刷 新 token 的值。

在处理业务逻辑之前 ，使⽤解密拿到的 userId去映射表中找到token ，和请求中的 token 对⽐ 就能校验是否合法了。

总结

实现登录态是前端⾮常基础且重要的技能之⼀ 。之前在学习这⼀块的时候 ，分不清 Cookie ， Session 和 Token 的区别。 session 是⽐ cookie 更好的⼀种解决⽅案。 token 成为主流，

是因为他不需要额外的存储管理。但是当涉及到单点登录的时候 ，其实也出现了多个服务器需要同步 映射表的问题。

1 ，cookie 的出现 浏览器和服务器之间的传输使⽤的 HTTP 协议 ，⽽它是⽆状态的。也就是说 ，每个 请求都是独⽴的 ，服务器并不知道2次请求是否是同⼀个⼈。

为了解决这个问题 ，服务器想了⼀个办法：

当客⼾端登录成功后 ，服务器会给客⼾端⼀个令牌凭证 token；客⼾端后续的请求都需要带着这个 token 在服务器做验证。

但⽤⼾不可能只在⼀个⽹站登录 ，于是客⼾端会收到各个⽹站的出⼊证 token。所以客⼾端需要⼀个 “卡包” 来实现以下功能：

能够存放多个 token ，token 可能来⾃不同的⽹站 ，也可能⼀个⽹站有多个 token。 能够⾃动出⽰正 确的token ，客⼾端访问不同⽹站时 ，会⾃动在请求中带着对应的token。 管理 token 的有效期 ，客 ⼾端需要⾃动发现那些过期的token 并移除。 满⾜上述要求的就是 cookie ，每⼀个 token 就是⼀个 cookie。

每个⽹站的 cookie ⼤⼩不超过 4kb。

2 ，cookie 的组成 每⼀个 cookie 都记录了以下信息：（ 除了 key 和 value ，其他⾮必填+顺序⽆关）

key：键 ，⽐如表⽰⾝份编号的字符串token

value ：值 ，⽐如 123abc ，它可以是任何字符串。

domain ：主机（域） ，表⽰这个 cookie 是属于哪个⽹站的 ，⽐如[www.csdn.net](http://www.csdn.net/)。【默认值： 当前 主机 ，也就是 location.host】 MDN参考

path：路径 ，表⽰这个 cookie 是属于该⽹站的哪个路径。【默认值：实测发现是 cookie 所处⽬录的

上级⽬录。 ⽐如⻚⾯是 <http://localhost:3001/a/api/login> ，则 path 为 /a/api】 secure ：是否使⽤安全传输。 MDN参考

httpOnly：表⽰该 cookie 仅能⽤于传输 ，⽽客⼾端通过 document.cookie 获取的是空字符串 ，这对

防⽌跨站脚本攻击（XSS） 会很有⽤ 。

XSS： ⽐如当前⻚⾯打开 iframe ，iframe 可以获取⽗级的 cookie。设置 httponly 可以不允许 js 获取 来防⽌跨站脚本攻击。expires ：过期时间 ，表⽰该 cookie 在什么时候过期。 MDN 参考

max-age：有效期。【默认值：如果 expires 和 max-age 都不设置 ，则为 session ，也就是会话结束 后过期 ，⼤多浏览器关闭（注意不是标签⻚关闭） 意味着会话结束。如果设置其中⼀个 ，cookie 会保

存在硬盘中 ，即便电脑关闭也不会消失。】 expires 和 max-age ⼀般只设置⼀个即可。

浏览器⾃动发送 cookie 的条件 需要同时满⾜以下4个条件：

没有过期。 expires 必须是⼀个有效的GWT时间 ，格林威治标准时间字符串 ，⽐如 Fri, 22 Dec 2023 17:09:13 GMT。到期后浏览器会⾃动删除。 new Date().toGMTString() // Fri, 22 Dec 2023 17:09:13 GMT // 对⽐常⻅的时间格式： new Date() // Sat Dec 23 2023 01:09:13 GMT+0800 (中国标准时间) max-age 是相对有效期 ，⽐如 max-age=1000 ，相当于设置 expires= 当前时间 + 1000s domain 字段 和这次请求的域是匹配的。 设置的 domain 是 [csdn.net](http://csdn.net/) ，则可匹配的请求域有：[csdn.net](http://csdn.net/)、

[www.csdn.net](http://www.csdn.net/)、[blogs.csdn.net](http://blogs.csdn.net/)等。 设置的 domain 是[www.csdn.net](http://www.csdn.net/) ，则只能匹配 [www.csdn.net](http://www.csdn.net/) 这样的请求域。 cookie 是不关⼼端⼝的 ，只要域匹配即可。（所以端⼝不同导致⾮同源⽽产⽣的跨域 并不影响。） ⽆效的域 ，浏览器的是不认的。 ⽐如对[search.jd.com/Search?keyw](http://search.jd.com/Search?keyw) … ⽹⻚来说：

【翻译：通过 Set-Cookie 标头设置 cookie 的尝试被阻⽌ ，因为其域对于当前域⽆效】 path 字段和这次请求的 path 也是匹配的。/ 表⽰匹配所有。如果是 /docs： 匹配的路

径：/docs ，/docs/ ，/docs/Web/ ，/docs/Web/HTTP 不匹配的路径：/ ，/docsets ，/fr/docs secure 字段验证。设置该字段 ，则请求协议必须是 https（否则不发送 cookie） ；不设置则请求协议可以是 https 或 http。 浏览器会将符合条件的 cookie ， ⾃动添加到请求头 Cookie 中。下图可以看到有3个满 ⾜的 cookie ，以 ; 分隔。

3 ，设置 cookie cookie 是保存在浏览器端的 ，有2种设置模式：

*服务器设置：通过设置响应头set-cookie: 123abc，浏览器会⾃动保存在* *“卡包”中。查看⽅式：控* *制台‒>Application‒>Storage‒>Cookies*浏览器设置：这种情况⽐较少⻅ 。举例：⽤⼾关闭了⼴告

时勾选了 【不喜欢】 或其他原因 ，就可以把这种⼩信息直接通过 js 保存到 cookie 中。后续请求服务 器

时 ，服务器会根据这个信息调整⼴告投放。 3.1 ，服务端设置 可在⼀次响应中设置多个 cookie。格式 如下：

键=值; path=?; domain=?; expires=?; max-age=?; secure; httponly 1 举例：

// 服务端 const Koa = require("koa"); const Router = require("koa-router"); const { bodyParser } =

require("@koa/bodyparser");

const app = new Koa(); const router = new Router();

router.post("/api/login", (ctx) => { const { name, pwd } = ctx.request.body; if (name === "下雪天的 夏⻛ " && pwd === "123") { ctx.set("set-cookie", 'token=aaa; domain=[localhost](http://localhost/); max

age=3600;secure; httponly'); ctx.body = "登录成功"; } else { ctx.body = { code: 500, msg: "⽤⼾名或 密码错误", }; } });

router.get("/api/home", (ctx) => { ctx.body = "home";

});app.use(bodyParser()).use(router.routes()); app.listen(3000); 提交

form 表单发送请求登录成功后 ，会⾃动跳转到⻚⾯ <http://localhost:3000/api/login> ，可以看到 cookie 已经设置了：

注意到 path 的默认值是 cookie 所处⽬录的上级⽬录。 expires/max-age 的时间格式保存为 ISO国际 标准时间 new Date() // Sat Dec 23 2023 01:27:53 GMT+0800 (中国标准时间) new

Date().to ISOString() // 2023-12-22T17:27:53.738Z new Date().toGMTString() // Fri, 22 Dec 2023 17:27:53 GMT

再次访问 <http://localhost:3000/api/home> 时 ，会发现请求头中⾃动带上了 cookie：

3.1 ，客⼾端设置 格式和在服务端相同 ，只是 httponly 字段⽆效。 因为该字段本来就是限制在客⼾端 访问的 ，客⼾端设置它没有意义。

document.cookie = 'token=aaa; domain=[localhost](http://localhost/);secure;httponly'

3.3 ，删除 cookie 可以修改 cookie 的过期时间即可： max-age=-1。浏览器会⾃动删除。

可以让服务器响应⼀个同样的domain、 同样的 path、 同样的 key ，只是时间过期的 cookie 即可。 以上⾯的例⼦来说 ，设置如下：

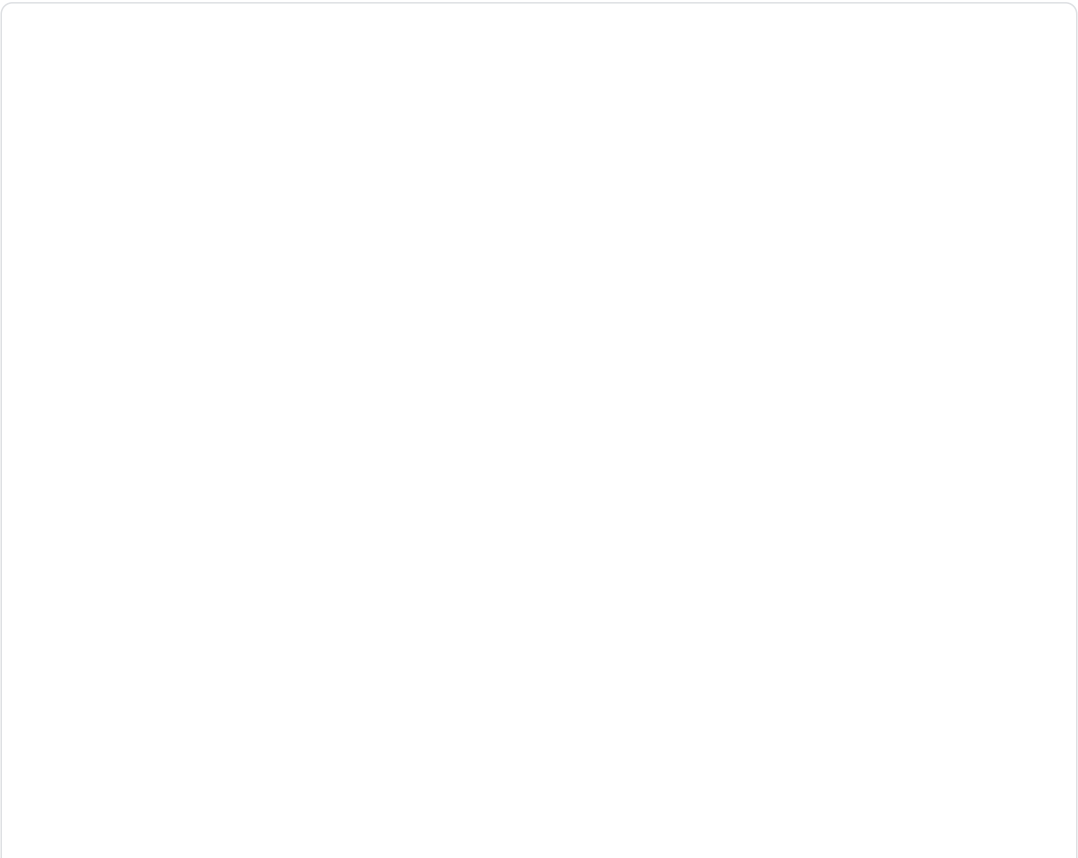
ctx.set("set-cookie", 'token=aaa; domain=[localhost](http://localhost/); max-age=-1'); 1 或客⼾端删除：

document.cookie = 'token=aaa; domain=[localhost](http://localhost/); max-age=-1' 1 注意：⽆论是修改还是删除 ，都 需要注意 domain 和 path ，因为可能存在 domain 和 path 不同但 key 相同的 cookie。

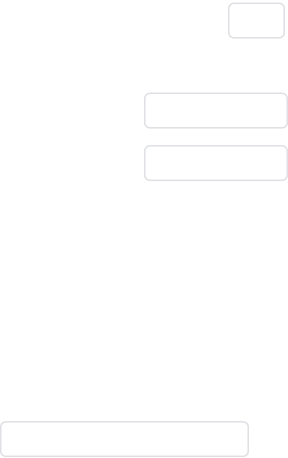
4 ，使⽤流程总结 登录 / 注册请求：

浏览器发送⽤⼾名和密码到服务器。 服务器验证通过后 ，在响应头中设置 cookie ，附带登录认证信息 （ ⼀般为 jwt） 。 浏览器收到 cookie 保存下来。 后续请求 ，浏览器会⾃动将符合的 cookie 附带到请 求中；服务器验证 cookie 后 ，允许其他操作完成业务流程。

72. 如何检测⽹⻚空闲状态(⼀定时间内⽆操作) 【热度: 329】

如何判断⻚⾯是否空闲

⾸先 ，我们要知道什么是空闲？⽤⼾⼀定时间内 ，没有对⽹⻚进⾏任何操作 ，则当前⽹⻚为空闲状 态。



⽤⼾操作⽹⻚ ，⽆⾮就是通过 ⿏标 、 键盘 两个输⼊设备(暂不考虑⼿柄等设备)。 因⽽我们可以监听 相应的输⼊事件 ，来判断⽹⻚是否空闲(⽤⼾是否有操作⽹⻚)。

1. 监听⿏标移动事件 mousemove ；

2. 监听键盘按下事件 mousedown ；

3. 在⽤⼾进⼊⽹⻚后 ，设置延时跳转 ，如果触发以上事件 ，则移除延时器 ，并重新开始。

⽹⻚空闲检测实现

实现点：

1. 需要使⽤防抖⽅式实现 ，避免性能问题

2. 监听 visibilitychange 事件 ，在⻚⾯隐藏时移除延时器 ，然后⻚⾯显⽰时继续计时 ，从⽽解 决这个问题。

实现：

1 - */\*\**

2 *⽹⻚空闲检测*

3 @param *{*() => void*} callback 空闲时执⾏，即⼀定时⻓⽆操作时触发* 4 @param *{*number*} [timeout=15] 时⻓，默认15s，单位：秒*

5 @param *{*boolean*} [immediate=false] 是否⽴即开始，默认* *false* 6 @returns

7 *\*/*const onIdleDetection = (callback, timeout = 15, immediate = false) => {let

pageTimer;let beginTime = 0;const onClearTimer = () => {

8 pageTimer && clearTimeout(pageTimer);

9 pageTimer = undefined;

10 };const onStartTimer = () => {const currentTime = Date.now();if (pageTimer

&& currentTime - beginTime < 100) {return;

11 }

12 onClearTimer();

13 beginTime = currentTime;

14 pageTimer = setTimeout(() => {

15 callback();

16 }, timeout \* 1000);

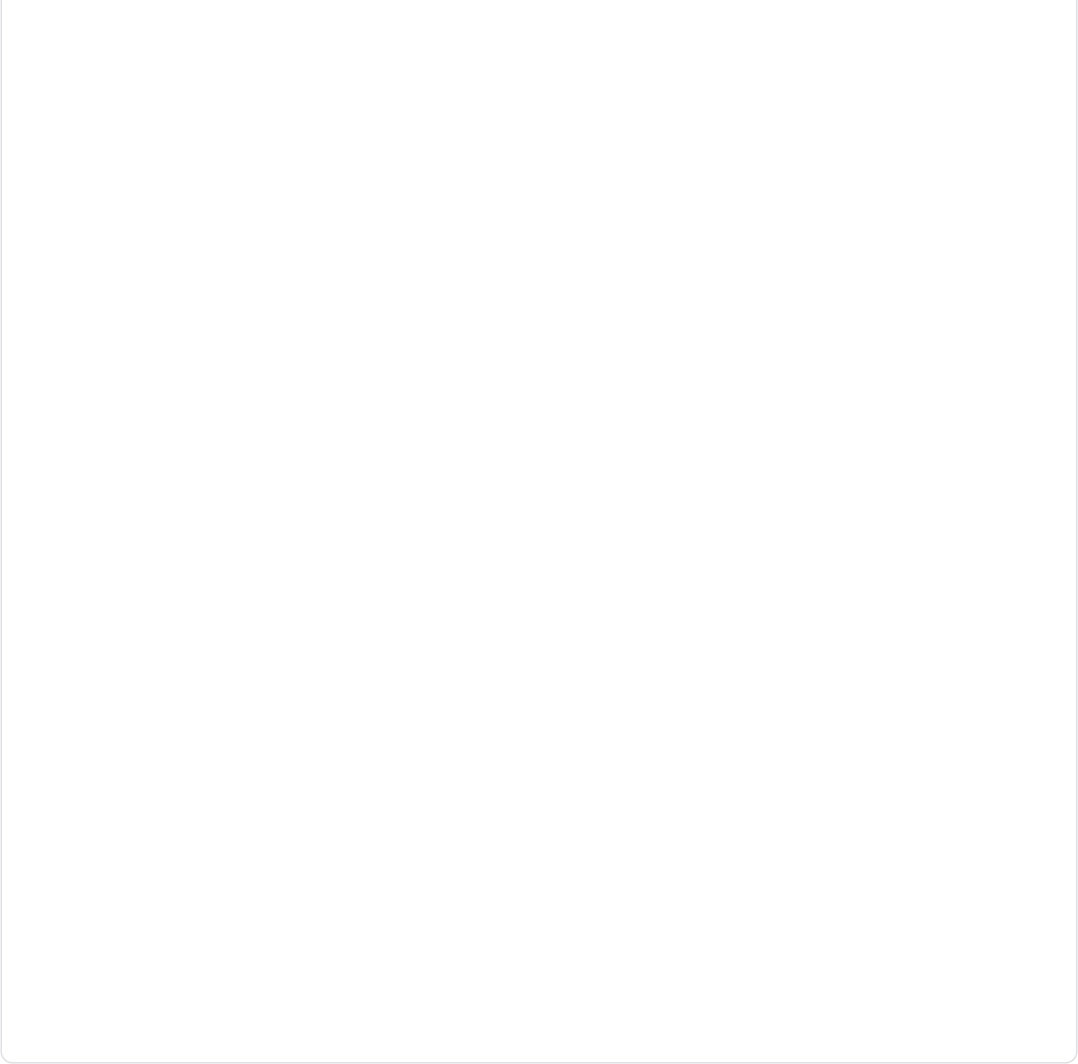
17 };

18 const onPageVisibility = () => {*// ⻚⾯显⽰状态改变时，移除延时器*

19 onClearTimer();

20 if (document.visibilityState === "visible") {const currentTime = Date.now();*// ⻚⾯显⽰时，计算时间，如果超出限制时间则直接执⾏回调函数if (currentTime - beginTime*

*>= timeout \* 1000) {*



21 callback();return; 22 }*// 继续计时*

23 pageTimer = setTimeout(() => {

24 callback();

25 }, timeout \* 1000(currentTime - beginTime));

26 }

27 };

28 const startDetection = () => {

29 onStartTimer();document.addEventListener("mousedown",

onStartTimer);document.addEventListener("mousemove",

onStartTimer);document.addEventListener("visibilitychange", onPageVisibility);

30 };

31 const stopDetection = () => {

32 onClearTimer();document.removeEventListener("mousedown",

onStartTimer);document.removeEventListener("mousemove",

onStartTimer);document.removeEventListener("visibilitychange",

onPageVisibility);

33 };

34 const restartDetection = () => {

35 onClearTimer();

36 onStartTimer();

37 };

38 if (immediate) {

39 startDetection();

40 }

41 return {

42 startDetection,

43 stopDetection,

44 restartDetection, 45 };

46 };

扩展

chrome 浏览器其实提供了⼀个 Idle Detection API ，来实现⽹⻚空闲状态的检测 ，但是这个 API 还是⼀个实验性特性 ，并且 Firefox 与 Safari 不⽀持。

73. 为什么 Vite 速度⽐ Webpack 快？ 【热度: 382】

关键词：vite 编译速度、vite 速度 与 webpack 速度

1、开发模式的差异

在开发环境中 ， Webpack 是先打包再启动开发服务器 ，⽽ Vite 则是直接启动 ，然后再按需编译 依赖⽂件。（ ⼤家可以启动项⽬后检查源码 Sources 那⾥看到）

Webpack 时 ，所有的模块都需要在开发前进⾏打包 ，这会增加启动时间和构建时

这意味着 ， 当使⽤

间。

Vite 则采⽤了不同的策略 ，它会在请求模块时再进⾏实时编译 ，这种按需动态编译的模式极⼤地

⽽



Vite

的优势更为明显。

缩短了编译时间 ，特别是在⼤型项⽬中 ，⽂件数量众多，



主动发起 请求去获取所需⽂件。Vite充分利⽤了这⼀点， 将开发环境下的模块⽂件直接作为浏览器要执⾏的⽂件 ，⽽不是像 Webpack 那样

ES Modules ，会

2、对ES Modules的⽀持 现代浏览器本⾝就⽀持

先打包 ，再交给浏

览器执⾏。 这种⽅式减少了中间环节 ，提⾼了效率。 什么是ES Modules？

import 语句 ，ES Modules 允许在浏览器端导⼊和导出模块。 当使⽤ ES Modules 进⾏开发时 ，开发者实际上是在构建⼀个

通过使⽤

export 和

依赖关系图 ，不同依赖项之间通过导⼊

语句进⾏关联。

主流浏览器（ 除IE外）均⽀持ES Modules ，并且可以通过在 script 标签中设置

type="module" 来

加载模块。默认情况下 ，模块会延迟加载 ，执⾏时机在⽂档解析之后 ，触发DOMContentLoaded事件 前。

3、底层语⾔的差异 Webpack 是基于

Node .js 构建的 ，⽽ Vite 则是基于

esbuild 进⾏预构建依赖。esbuild 是采

⽤

Go 语⾔编写的 ，Go 语⾔是

纳秒 级别的 ，⽽ Node.js 是

毫秒 级别的。 因此 ，Vite 在打包速度上

10-100 倍的提升。 什么是预构建依赖？

相⽐Webpack 有

预构建依赖通常指的是在项⽬

启动或构建 之前 ，对项⽬中所需的依赖项进⾏预先的

处理或构建 。这

直接使⽤ 这些已经预构建好的依赖 ，⽽⽆需再进⾏实时的 编译或构建 ，从⽽提⾼了应⽤程序的运⾏速度和效率。

样做的好处在于 ， 当项⽬实际运⾏时 ，可以

4、热更新的处理

在 Webpack 中 ， 当⼀个模块或其依赖的模块内容改变时 ，需要

重新编译 这些模块。

⽽在 Vite 中 ， 当某个模块内容改变时 ，只需要让浏览器

重新请求 该模块即可 ，这⼤⼤减少了热更新

的时间。

总结

充分利⽤了现代浏览

总的来说 ，Vite 之所以⽐ Webpack 快 ，主要是因为它采⽤了

不同的开发模式 、

并优化了热更新的处理 。这些特点使得

使⽤了更⾼效的底层语⾔ ，

器的 ES Modules ⽀持 、

Vite在⼤型项⽬中具有显著的优势 ，能够快速启动和构建 ，提⾼开发效率。

74. 列表分⻚ ， 快速翻⻚下的竞态问题 【热度: 444】

列表分⻚ ， 快速翻⻚下的竞态问题

问题描述： ⽐如在前端分⻚请求的时候 ， 因为翻⻚很快 ， 所以请求还没有来得及回来的时候 ， 就发 起了下⼀次请求 ， 且请求返回的时间也是不固定的。

如何保证最后⼀次请求结果和其请求⻚码是对应上的。

在处理这种情况时 ，⼀种常⻅的⽅法是使⽤请求标记或唯⼀标识符来确保请求和结果之间的对应关 系。

以下是⼀个⽰例代码⽚段 ，展⽰了⼀种可能的解决⽅案：



*(page) {*

requestId++;

*// 将请求标记与⻚码⼀起发送*

fetch(

/api?requestId=${requestId}&page=${page}

)

.then(response => response.json())

.then(data => {*// 根据请求标记处理返回的数据*

handleResponseData(requestId, data);

});

}

*// 处理返回数据的函数* 

1 *//存储请求的标记* *requestId = 0;*

2

3 *//发起请求的函数*

4

5

6

7

8

9

10

11

12

13

14

15 *(requestId, data) { (requestId*

*=== currentRequestId) {// 在这⾥处理数据并更新⻚⾯*

16 } 17 } 18

19 *// 在翻⻚时调⽤* *sendRequest 函数*

在这个⽰例中 ，每次发起请求时都会增加请求标记 requestId ，并将其与⻚码⼀起发送到服务器。 在处理返回的数据时 ，根据请求标记来确保与当前的请求对应。

另外 ，还可以考虑以下⼏点：

• 对快速翻⻚进⾏限制或优化 ，避免过于频繁的请求。

• 在服务器端处理请求时 ，可以根据请求标记来保证返回的数据与特定的请求相关联。

• 可以使⽤缓存来存储部分数据 ，减少不必要的请求。 保证唯⼀性

保证请求标记的唯⼀性可以通过以下⼏种⽅式：

1. 使⽤递增的数字：就像上⾯⽰例中的

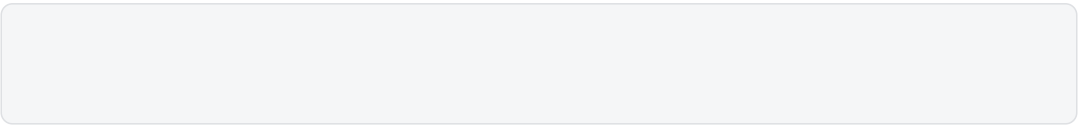
requestId

⼀样 ，每次增加 1。

2. 使⽤随机数：⽣成⼀个随机的数字作为请求标记。

3. 使⽤时间戳：结合当前时间⽣成唯⼀的标记。

4. 组合多种因素：例如 ，将数字、 时间戳或其他相关信息组合起来创建唯⼀标记。 例如 ，使⽤时间戳作为请求标记的⽰例代码如下：

1  requestId = Date.now();



这样每次请求时，

都会是⼀个唯⼀的时间戳值。

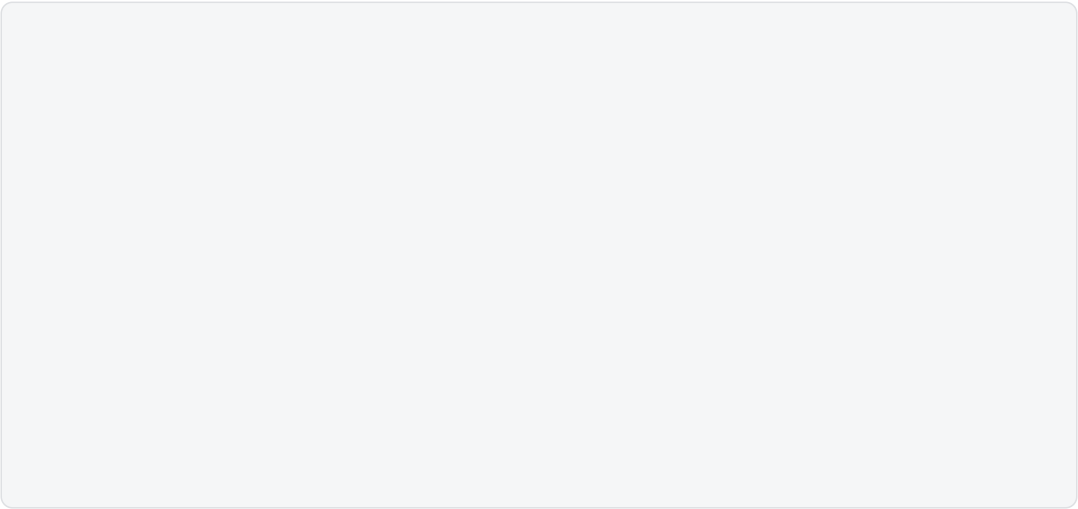
requestId

75. JS 执⾏ 100 万个任务 ， 如何保证浏览器不卡顿？ 【热度: 806】

Web Workers

要确保浏览器在执⾏100万个任务时不会卡顿 ，你可以考虑使⽤Web Workers来将这些任务从主线程中 分离出来。Web Workers允许在后台线程中运⾏脚本 ，从⽽避免阻塞主线程 ，保持⻚⾯的响应性。

以下是⼀个使⽤Web Workers的简单⽰例：

1 *// 主线程代码* *worker =*  *Worker('worker.js'); //创建⼀个新的Web Worker*

2 worker.postMessage({ start: 0, end: 1000000 }); *// 向Web Worker发送消息*

3 worker.onmessage =  (event) { result = event.data;console.log('任务 完成： ', result);

4 }; 5

6 *// worker.js - Web Worker代码*

7 onmessage =  (event) { start = event.data.start; end =

event.data.end; sum = 0; ( i = start; i <= end; i++) {

8 sum += i; 9 }

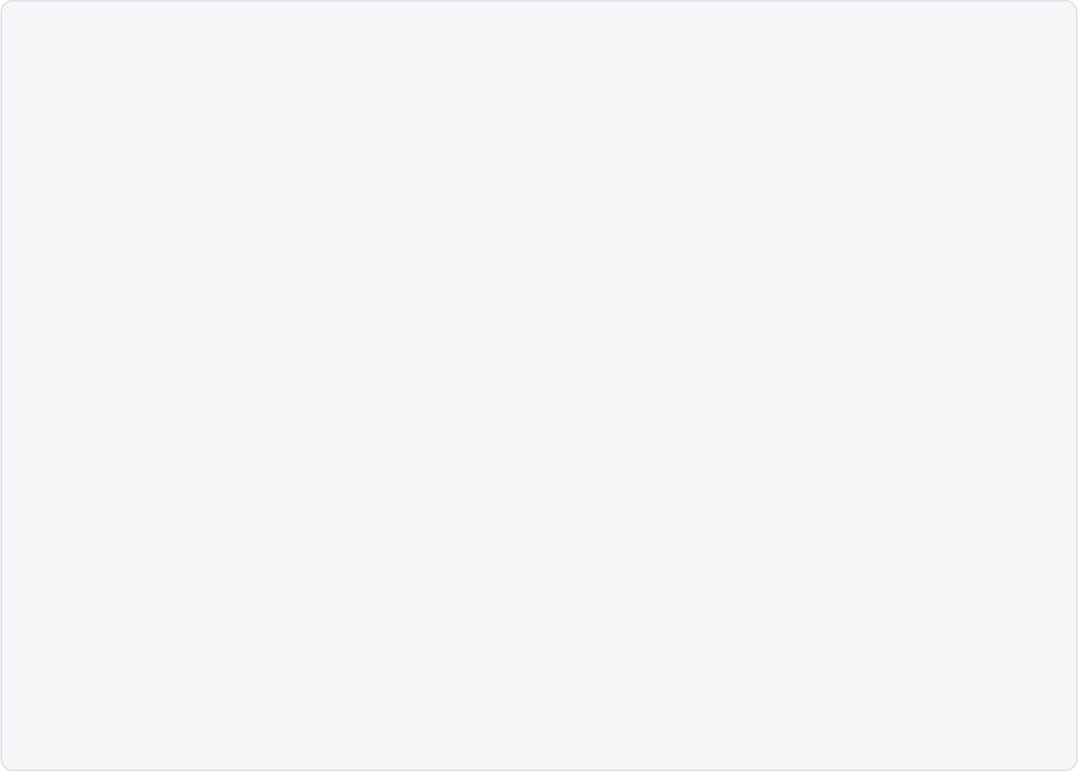
10 postMessage(sum); *// 向主线程发送消息* 11 };

在这个⽰例中 ，主线程创建了⼀个新的Web Worker ，并向其发送了⼀个包含任务范围的消息。Web Worker在后台线程中执⾏任务 ，并将结果发送回主线程。

requestAnimationFrame 来实现任务分割

使⽤ requestAnimation Frame 来实现任务分割是⼀种常⻅的⽅式 ，它可以确保任务在浏览器的每 ⼀帧之间执⾏ ，从⽽避免卡顿。 以下是⼀个使⽤ requestAnimation Frame 来分割任务的简单例 ⼦：





1 *// 假设有⼀个包含⼤量元素的数组* *bigArray = Array.from({ length: 1000000 },*

*(\_, i) => i + 1);*

2

3 *// 定义⼀个处理函数，例如对数组中的每个元素进⾏平⽅操作*

*{ chunk.map(num => num \* num);*

4 } 5

6 *// 分割任务并使⽤requestAnimationFrame chunkSize = 1000; // 每个⼩块的⼤⼩*

*index = 0;*

7

8  () {  () { chunk =

bigArray.slice(index, index + chunkSize); *// 从⼤数组中取出⼀个⼩块* *result =*

*processChunk(chunk); // 处理⼩块任务console.log('处理完成：* *', result);*

9 index += chunkSize;

10  (index < bigArray.length) {

11 requestAnimationFrame(processChunkWithRAF); *// 继续处理下⼀个⼩块*

12 }

13 }

14 requestAnimationFrame(processChunkWithRAF); *// 开始处理⼤数组* 15 }

16 processArrayWithRAF();

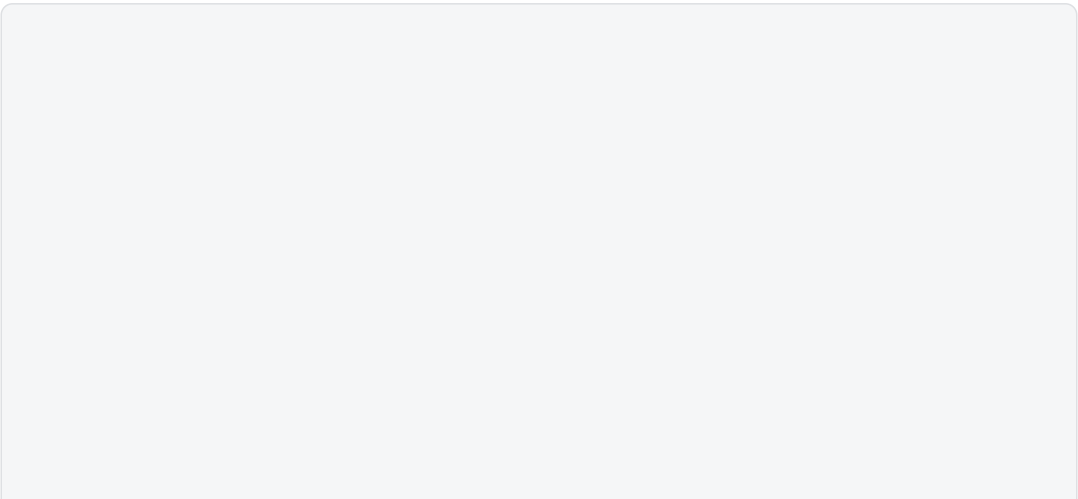
*(chunk)*

在这个例⼦中 ，我们使⽤ requestAnimation Frame 来循环执⾏处理⼩块任务的函数

processChunkWithRAF ，从⽽实现对⼤数组的任务分割。这样可以确保任务在每⼀帧之间执⾏， 避免卡顿。

针对上⾯的改进⼀下

const chunkSize = 1000; // 每个⼩块的⼤⼩ 是不能保证不卡的 ， 那么久需要动态调整 chunkSize 的⼤⼩ ， 代码可以参考下⾯的⽰范：

1  $result = document.getElementById("result"); 2

3 *// 假设有⼀个包含⼤量元素的数组* *bigArray = Array.from({ length: 1000000 }, (\_, i) => i + 1);*

4

5 *// 定义⼀个处理函数，对数组中的每个元素执⾏⼀次* *(chunk) {*

6 chunk: ${chunk}

7 ; 8 } 9

10 *// 动态调整* *chunkSize 的优化⽅式* *chunkSize = 1000; // 初始的* *chunkSize* *index = 0;*

11



12  () {  ()

{ startTime = performance.now(); *// 记录结束时间* *( i = 0; i <*

*chunkSize; i++) { (index < bigArray.length) { result =*

*processChunk(bigArray[index]); //对每个元素执⾏处理函数*

13 $result.innerText = result;

14 index++; 15 }

16 } endTime = performance.now(); timeTaken = endTime - startTime; *// 计*

*算处理时间// 根据处理时间动态调整* *chunkSize* *(timeTaken > 16) { // 如果处理时间超过* *⼀帧的时间（16毫秒），则减⼩* *chunkSize*

17 chunkSize = Math.floor(chunkSize \* 0.9); *// 减⼩10%*

18 }  (timeTaken < 16) { *// 如果处理时间远⼩于⼀帧的时间（8毫秒），则增加*

*chunkSize*

19 chunkSize = Math.floor(chunkSize \* 1.1); *// 增加10%*

20 }

21  (index < bigArray.length) {

22 requestAnimationFrame(processChunkWithRAF); *// 继续处理下⼀个⼩块*

23 }

24 }

25 requestAnimationFrame(processChunkWithRAF); *// 开始处理⼤数组* 26 }

27 processArrayWithDynamicChunkSize();

在这个例⼦中 ，我们动态调整 chunkSize 的⼤⼩ ，根据处理时间来优化任务分割。根据处理时间的 表现 ，动态调整 chunkSize 的⼤⼩ ，以确保在处理⼤量任务时 ，浏览器能够保持流畅 ，避免卡顿。

76. git 仓库迁移应该怎么操作 【热度: 160】

如果你想迁移仓库并保留原始仓库的所有提交历史、分⽀和标签 ，你可以使⽤以下步骤：



git clone

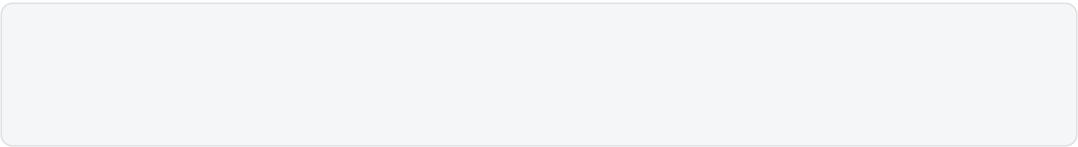
git push

⽅法⼀：使⽤

和

1. 在仓库 B 中创建新的仓库。

2. 在本地克隆仓库 A：



1 git clone --mirror <仓库 A URL>

2 cd <仓库 A ⽬录>

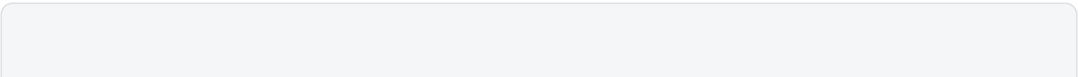


使⽤

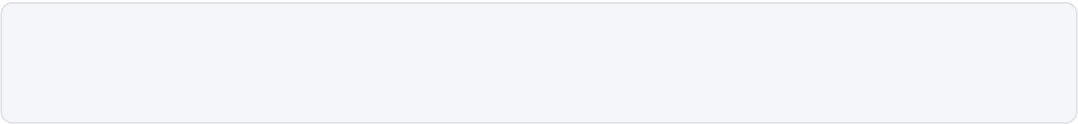
选项克隆仓库会保留所有分⽀ 、标签和提交历史。

--mirror

1. 修改远程仓库地址为仓库 B：

1 git remote set-url --push origin <仓库 B URL>

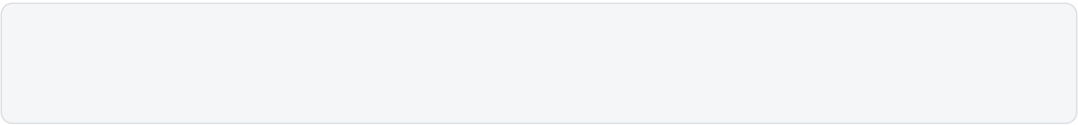
1. 推送到仓库 B：

1 git push --mirror

git bundle

⽅法⼆： 使⽤

1. 在仓库 A 中创建 bundle ⽂件：

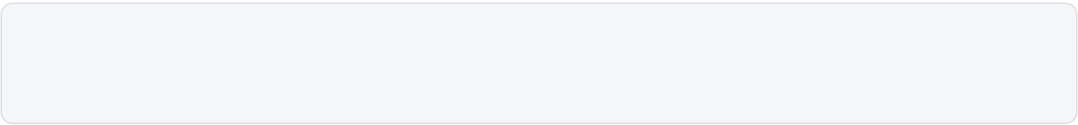


1 git bundle create repoA.bundle --all

repoA .bundle ⽂件传输到仓库 B 所在位置。

1. 将

2. 在仓库 B 中克隆：



1 git clone repoA.bundle <仓库 B ⽬录>

这两种⽅法都会保留所有分⽀ 、标签和提交历史。选择哪种⽅法取决于你的具体需求和迁移环境。 注意：

• 使⽤ --mirror 或 --all 选项在 git clone 或 git bundle 中时 ，会将所有的分⽀和 标签复制到⽬标仓库。

• 在执⾏之前 ，请确保仓库 B 是空的或者是⼀个你可以覆盖的⽬标仓库 ，因为这些操作会覆盖⽬标仓 库的内容。

• 如果仓库 A 中包含⼦模块 ，你可能需要额外处理⼦模块的迁移。

1. 如何禁⽌别⼈调试⾃⼰的前端⻚⾯代码? 【热度: 347】

**⽆限 debugger**

• 前端⻚⾯防⽌调试的⽅法主要是通过不断 debugger 来疯狂输出断点 ，因为 debugger 在控 制台被打开的时候就会执⾏

• 由于程序被 debugger 阻⽌ ，所以⽆法进⾏断点调试 ，所以⽹⻚的请求也是看不到的

• 基础代码如下：

• */\*\**

*基础禁⽌调试代码* *\*/*

(() => {function ban() { setInterval(() => { debugger;

}, 50); }try { ban();

} catch (err) { }

})();

**⽆限 debugger 的对策**

• 如果仅仅是加上⾯那么简单的代码 ，对于⼀些技术⼈员⽽⾔作⽤不⼤



Ctrl + F8

• 可以通过控制台中的

Deactivate breakpoints 按钮或者使⽤快捷键

关闭⽆

debugger

限



breakpoint

• 这种⽅式虽然能去掉碍眼的

debugger ，但是⽆法通过左侧的⾏号添加

禁⽌断点的对策

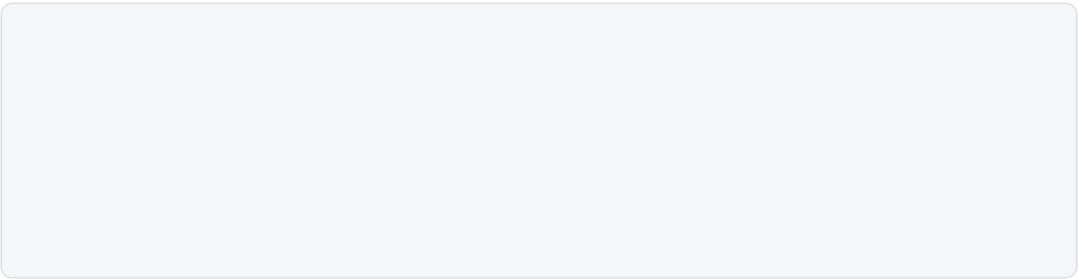
• 如果将

setInterval 中的代码写在⼀⾏ ，就能禁⽌⽤⼾断点 ，即使添加

logpoint 为

false 也⽆⽤

• 当然即使有些⼈想到⽤左下⻆的格式化代码 ，将其变成多⾏也是没⽤的



1 (() => {  () {

2 setInterval(() => { debugger; }, 50);

3 } {

4 ban();

5 }  (err) { }

6 })();

忽略执⾏的代码

• 通过添加 add script ignore list 需要忽略执⾏代码⾏或⽂件

• 也可以达到禁⽌⽆限 debugger

忽略执⾏代码的对策

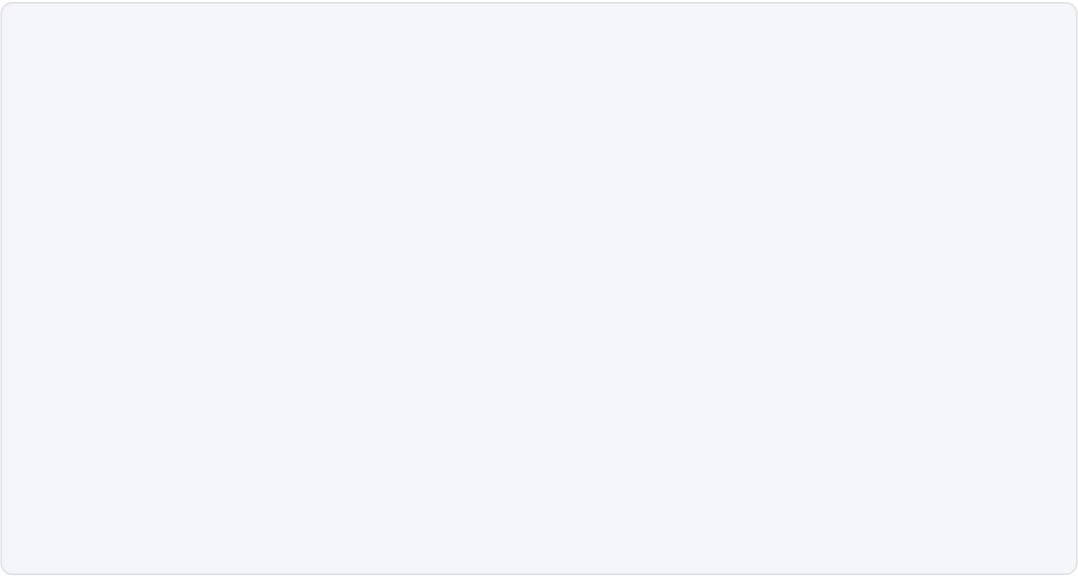
• 那如何针对上⾯操作的恶意⽤⼾呢

• 可以通过将 debugger 改写成 Function("debugger")(); 的形式来应对

• Function 构造器⽣成的 debugger 会在每⼀次执⾏时开启⼀个临时 js ⽂件

• 当然使⽤的时候 ，为了更加的安全 ，最好使⽤加密后的脚本





1 *// 加密前*

2 (() => {  () {

3 setInterval(() => {Function('debugger')();

4 }, 50);

5 } {

6 ban();

7 }  (err) { }

8 })();

9

10 *// 加密后eval( (c,g,a,b,d,e){d=String;**(!"".replace(/^/,String)){*

*-;)e[a]=b[a]||a;b=[ (f){ e[f]}];d=**()*

*{ "\w+"};a=1}*

*RegExp("\b"+d(a)+"\b","g"),b[a])); c}('(()=>{1 0(){2(()=>{3("4")*

*()},5)}6{0()}7(8){}})();',9,9,"block function setInterval Function debugger 50*

*try catch err".split(" "),0,{}));*

*(;a-*

*(;a--;)b[a]&&(c=c.replace(*

终极增强防调试代码

• 为了让⾃⼰写出来的代码更加的晦涩难懂 ，需要对上⾯的代码再优化⼀下

 Function( 'debugger ') .call()

• 将

改成 (function(){return false;})

[ 'constructor ']( 'debugger ') 'call ';

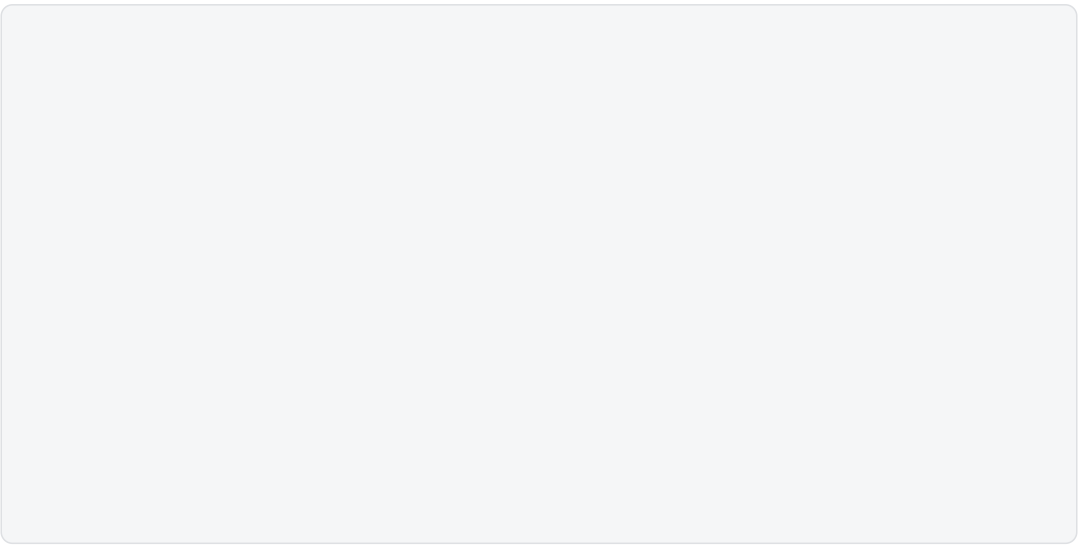
• 并且添加条件 ， 当窗⼝外部宽⾼和内部宽⾼的差值⼤于⼀定的值 ，我把

body

⾥的内容换成指定

内容

• 当然使⽤的时候 ，为了更加的安全 ，最好加密后再使⽤

1 (() => { () { (window.outerHeight - window.innerHeight > 200 || window.outerWidth - window.innerWidth > 200) {document.body.innerHTML = "检测到 ⾮法调试 ,请关闭后刷新重试!";

2 }

3 setInterval(() => {

4 (  () {  false;

5 }

6 ['constructor']('debugger')

7 ['call']());

8 }, 50);

9 } {

10 block();

11 }  (err) { }

12 })();

78.web 系统里面，如何对图片进行优化?【热度：789】

图片作为网页和移动应用中不可或缺的元素之一，对于用户体验和网站性能都有着重要的影响。

加载速度是用户体验的关键因素之一，而大尺寸的图片会增加网页加载时间，导致用户等待时间过

长，从而影响用户的满意度和留存率。通过优化图片，我们可以显著减少页面加载时间，提供更快速 流畅的使用体验。

图片优化是提升用户体验、提高网站性能、减少流量消耗和增加搜索引擎曝光度的关键因素。为了提 供更出色的用户体验，同时也提升网站的性能。总结了一下通用的图片优化首手段。

1.选择合适的图片格式

以下是对常用的图片格式jpg、png 和webp 进行深度对比的表格：

|  |  |  |  |
| --- | --- | --- | --- |
| **特性** | **JPG** | **PNG** | **WebP** |
| 压缩算法 | 有损压缩 | 无损压缩 | 有损压缩 |
| 透明度 | 不支持透明 度 | 支持透明度 | 支持透明度 |
| 图片质量 | 可调整质量 | 无法调整质 量 | 可调整质量 |
| 文件大小 | 相对较小 | 相对较大 | 相对较小 |
| 浏览器支持 | 支持在所有 主流浏览器 上显示 | 支持在所有 主流浏览器 上显示 | 部分浏览器 支持 |
| 动画支持 | 不支持动画 | 不支持动画 | 支持动画 |
| 兼容性 | 兼容性较好 | 兼容性较好 | 兼容性较差 |

请注意，这个表格只是对这些格式的一般特征进行了总结，并不代表所有情况。实际情况可能因图像 内容、压缩设置和浏览器支持等因素而有所不同。因此，在选择图像格式时，您应根据具体要求和应 用场景进行评估和选择。

2. 图片压缩

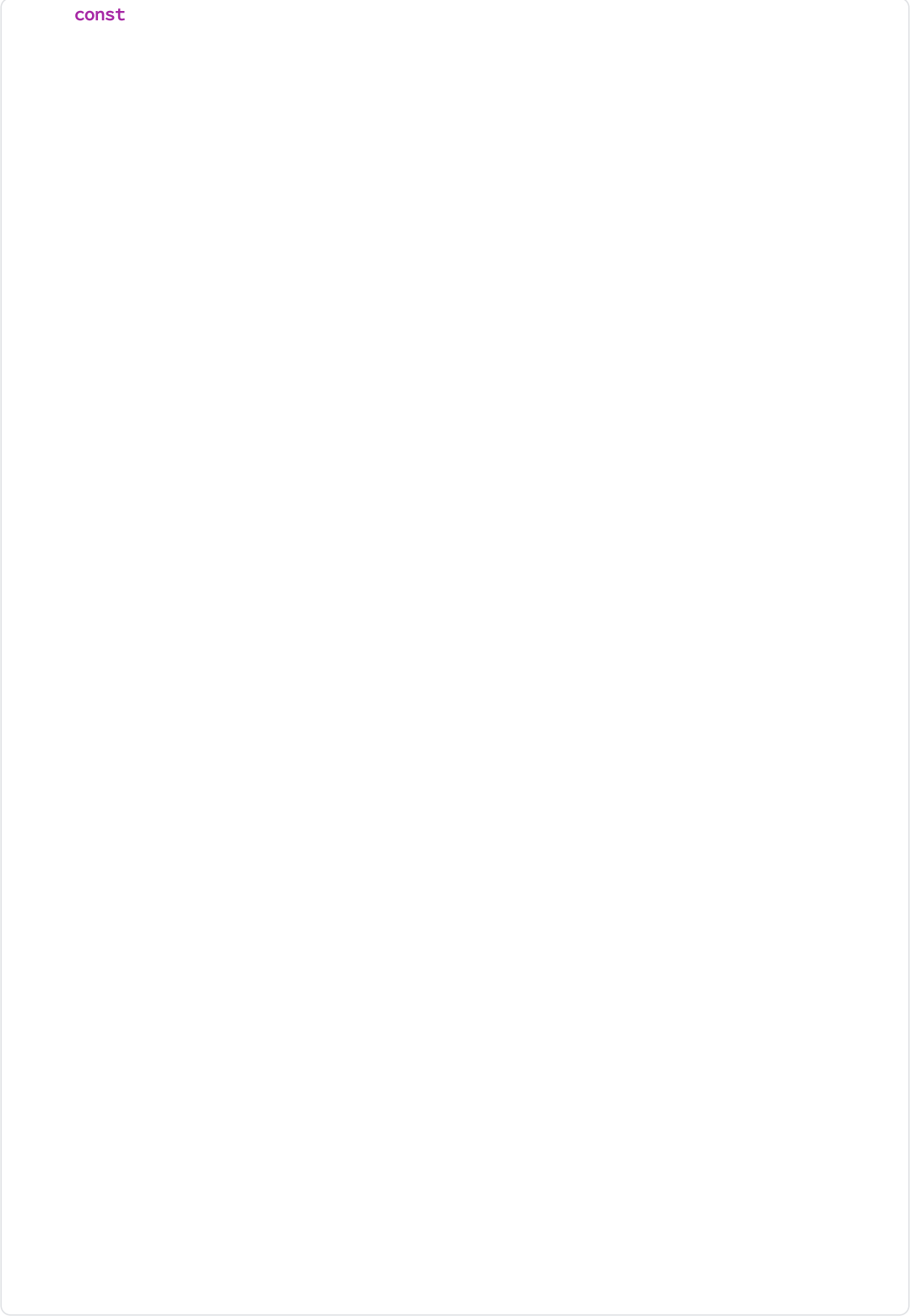
主要介绍webpack 对图片进行压缩，可以使用以下步骤：

1.安装依赖：首先，确保你已经在项目中安装了webpack 和相关的loader。可以使用以下命令安装所 需的loader:

1 npm install --save-dev file-loader image-webpack-loader



1. 配置Webpack: 在 Webpack 的配置文件中进行相关配置。以下是一个简单的示例：

1 path = require('path');

2

3 module.exports = {

4 entry: 'src/index.js',

5 output: {

6 filename: 'bundle.js',

7 path: path.resolve(\_\_dirname, 'dist')

8 },module: {

9 rules: [

10 {

11 test: /\.(png|jpe?g|gif)$/i,

12 use: [

13 {

14 loader: 'file-loader',

15 options: {

16 name: '[name].[ext]',

17 outputPath: 'images/'

18 }

19 },

20 {

21 loader: 'image-webpack-loader',

22 options: {

23 mozjpeg: {

24 progressive: true,

25 quality: 65

26 },*// optipng.enabled: false will disable optipng*

27 optipng: {

28 enabled: false,

29 },

30 pngquant: {

31 quality: [0.65, 0.90],

32 speed: 4

33 },

34 gifsicle: {

35 interlaced: false,

36 },*// the webp option will enable WEBP*

37 webp: {

38 quality: 75

39 }

40 }

41 }

42 ]

43 }

44 ]

45 }

46 };

file-loader 将图⽚复制到输出⽬录 ，并使⽤

上述配置中 ，我们使⽤

image-webpack-

loader 对图⽚进⾏压缩和优化。

1. 运⾏Webpack：现在 ， 当你运⾏Webpack时 ，它将⾃动使⽤

image-webpack-loader 对匹配

到的图⽚进⾏压缩和优化。压缩后的图⽚将被复制到输出⽬录中。

2. 雪碧图

Web图⽚优化的雪碧图（CSS Sprites）是⼀种将多个⼩图⽚合并为⼀个⼤图⽚的技术。通过将多个⼩ 图⽚合并成⼀张⼤图⽚ ，可以减少浏览器发送的请求次数 ，从⽽提⾼⻚⾯加载速度。

background-position 属性 ，将所需的⼩

background-image 和

雪碧图的原理是通过CSS的

图⽚显⽰在指定的位置上。这样 ，只需加载⼀张⼤图 ，就可以显⽰多个⼩图⽚ ，减少了⽹络请求的数 量 ，提⾼了⻚⾯加载速度。

听上去好像很⿇烦 ， 实际上可以使⽤ webpack 插件

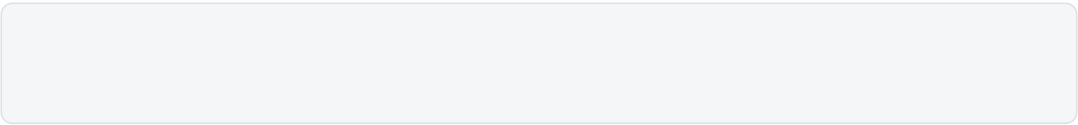
webpack-spritesmith 完成⾃动化处理雪

碧图合成 ，我们在使⽤过程中正常使⽤即可。 以下是使⽤

webpack-spritesmith 插件来⾃动处理雪碧图的步骤：

1. 安装插件：使⽤npm或yarn安装

webpack-spritesmith 插件。

1 npm install webpack-spritesmith --save-dev

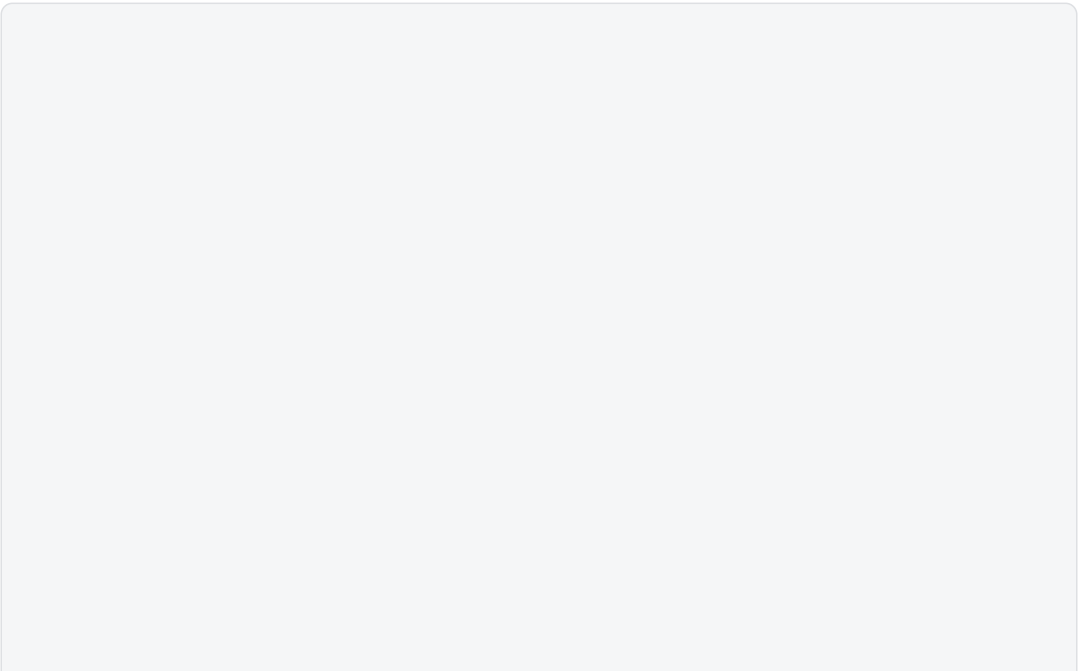


1. 配置Webpack：在Webpack配置⽂件中 ，引⼊

webpack-spritesmith

插件 ，并配置相应的选

项。

1  SpritesmithPlugin = require('webpack-spritesmith'); 2

3 module.exports = {*// ...其他配置*

4 plugins: [ SpritesmithPlugin({

5 src: {

6 cwd: path.resolve(\_\_dirname, 'path/to/sprites'), *// 需要合并的⼩图⽚所在的* *⽬录*

7 glob: '\*.png' *//⼩图⽚的⽂件名格式* 8 },

9 target: {

10 image: path.resolve(\_\_dirname, 'path/to/output/sprite.png'), *// ⽣成的雪* *碧图的路径和⽂件名*

11 css: path.resolve(\_\_dirname, 'path/to/output/sprite.css') *// ⽣成的CSS样* *式表的路径和⽂件名*

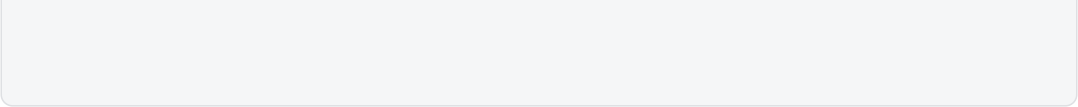
12 },

13 apiOptions: {

14 cssImageRef: 'path/to/output/sprite.png' *// CSS样式表中引⽤雪碧图的路径*

15 }

16 })



17 ]

18 }

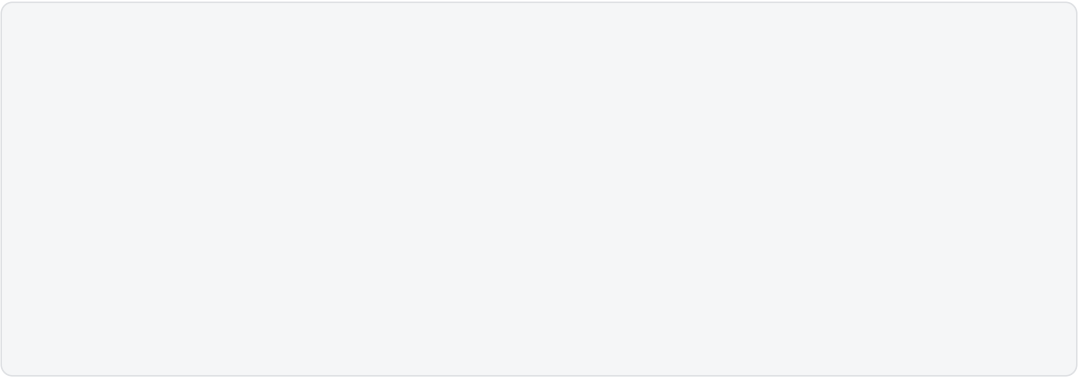
1. 使⽤雪碧图：在HTML中 ，使⽤⽣成的CSS样式类来显⽰相应的⼩图⽚ 。Webpack会⾃动处理雪碧 图的合并和CSS样式的⽣成。例如：

然后 ，你可以按照以下⽅法在CSS中引⽤雪碧图：

CSS⽅式：

1 div {background: url(path/to/output/sprite.png) no-repeat; 2 }.icon-facebook {*/\* 设置⼩图标在雪碧图中的位置和⼤⼩* *\*/*

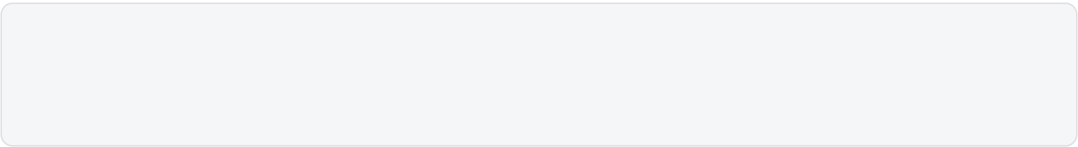
3 width: 32px;height: 32px;background-position: 0 0; */\* 该⼩图标在雪碧图中的位置* *\*/*

4 }.icon-twitter {width: 32px;height: 32px;background-position: -32px 0; */\* 该⼩图* *标在雪碧图中的位置* *\*/*

5 }.icon-instagram {width: 32px;height: 32px;background-position: -64px 0; */\* 该⼩* *图标在雪碧图中的位置* *\*/*

6 }

在HTML中 ，你可以像下⾯这样使⽤对应的CSS类来显⽰相应的⼩图标：



1 <div class="icon icon-facebook"</div<div class="icon icon-twitter"</div<div

class="icon icon-instagram"</div

这样 ，Webpack会根据配置⾃动处理雪碧图 ，并⽣成对应的雪碧图和CSS样式表。CSS中的

background 属性会引⽤⽣成的雪碧图 ，并通过 background-position 来指定显⽰的⼩图标在 雪碧图中的位置。

确保在CSS中指定了每个⼩图标在雪碧图中的位置和⼤⼩ ，以便正确显⽰。

webpack-

使⽤Webpack⾃动处理雪碧图可以简化开发流程 ，并且可以根据需要⾃定义配置。

|  |  |
| --- | --- |
|  |  |
| spritesmith | 是⼀个常⽤的Webpack插件 ，可以帮助⾃动处理雪碧图。 |

2. 图标类型资源推荐使⽤ iconfont

如果你有很多图标类型的图⽚资源 ，并且想使⽤ icon font 来处理这些资源 ，可以按照以下步骤进⾏ 处理：

• 获取图标资源：⾸先 ，你需要获取你想要的图标资源。你可以从 icon font ⽹站或其他图标库中 选择和下载符合需求的图标。这个没有啥好说的 ， 直接推荐: [https://www.iconfont.cn/](https://gitee.com/link?target=https%3A%2F%2Fwww.iconfont.cn%2F)

• ⽣成字体⽂件：接下来 ，你需要将这些图标转换成字体⽂件。你可以使⽤ icon font 提供的在线 转换⼯具 ，将图标⽂件上传并⽣成字体⽂件（包括 .ttf 、 .eot 、 .woff 和 .svg 格式）。

• 引⼊字体⽂件：将⽣成的字体⽂件下载到本地 ，并在你的项⽬中引⼊。通常 ，你需要在CSS⽂件中 通过 @font-face 规则引⼊字体⽂件 ，并为字体定义⼀个唯⼀的名称。

• 使⽤图标：⼀旦字体⽂件引⼊成功 ，你可以在CSS中通过设置 content 属性来使⽤图标。每个图 标都会有⼀个对应的Unicode代码 ，你可以在 icon font 提供的⽹站或字体⽂件中找到对应图标 的Unicode代码 ，并通过设置 content 属性的值为该Unicode代码来使⽤图标。

以下是⼀个简单的⽰例 ，以帮助你更好地理解：

1  {font-family: 'iconfont';src: url('path/to/iconfont.eot'); */\* 引⼊字* *体⽂件* *\*/*

@

2 */\* 其他格式的字体⽂件* *\*/*

3 }.icon {font-family: 'iconfont'; */\* 使⽤定义的字体名称* *\*/*

4 font-size: 16px; */\* 图标⼤⼩* *\*/*

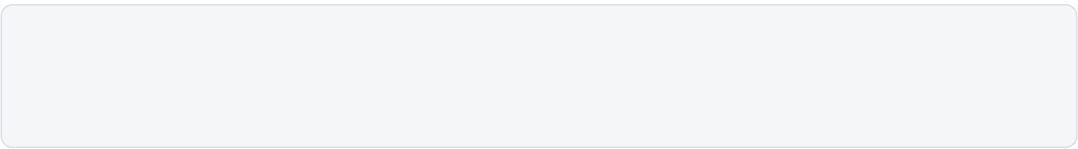
5 line-height: 1; */\* 图标⾏⾼* *\*/*

6 }.icon-facebook::before {content: '\e001'; */\* 使⽤Unicode代码表⽰想要显⽰的图标* *\*/* 7 }.icon-twitter::before {content: '\e002'; */\* 使⽤Unicode代码表⽰想要显⽰的图标* *\*/* 8 }.icon-instagram::before {content: '\e003'; */\* 使⽤Unicode代码表⽰想要显⽰的图标* *\*/* 9 }

在上述⽰例中 ，我们⾸先通过 @font-face 引⼊了字体⽂件 ，并为字体定义了⼀个名称

icon font 。然后 ，我们使⽤该名称作为 font-family 属性的值 ，以便在 .icon 类中使⽤该字 体。最后 ，我们通过在 : :before 伪元素中设置 content 属性为图标的Unicode代码 ，来显⽰相应 的图标。

在HTML中 ，你可以像下⾯这样使⽤对应的CSS类来显⽰相应的图标：

1 <span class="icon icon-facebook"</span<span class="icon icon-twitter"

</span<span class="icon icon-instagram"</span



通过上述步骤 ，你可以使⽤

来处理你的图标资源 ，并在项⽬中⽅便地使⽤它们。确保在

icon font

CSS中设置了图标的字体⼤⼩和⾏⾼ ，以便正确显⽰图标。

3. 使⽤ base64 格式

实际开发过程中 ， 为何会考虑 base64 ？ 使⽤Base64图⽚的优势有以下⼏点：

• 减少HTTP请求数量：通常情况下 ，每个⽹⻚都需要加载多张图⽚ ，因此会发送多个HTTP请求来获 取这些图⽚⽂件。使⽤Base64图⽚可以将图⽚数据嵌⼊到CSS或HTML⽂件中 ，减少了对服务器的 请求次数 ，从⽽提⾼⽹⻚加载速度。

• 减少图⽚⽂件的⼤⼩： Base64是⼀种编码⽅式 ，可以将⼆进制数据转换成⽂本字符串。通过使⽤ Base64 ，可以将图⽚⽂件转换成⽂本字符串 ，并将其嵌⼊到CSS或HTML⽂件中。相⽐于直接引⽤ 图⽚⽂件 ，Base64编码的字符串通常会更⼩ ，因此可以减少图⽚⽂件的⼤⼩ ，从⽽减少了⽹⻚的总 体积 ，加快了⽹⻚加载速度。

• 简化部署和维护：将图⽚数据嵌⼊到CSS或HTML⽂件中 ，可以减少⽂件的数量和复杂性 ，使得部 署和维护变得更加简单和⽅便。此外 ，也不需要处理图⽚⽂件的路径和引⽤相关的问题。

• 实现⼀些特殊效果：通过Base64图⽚ ，可以实现⼀些特殊的效果 ，例如⻚⾯背景渐变、 图标的使⽤ 等。这样可以避免使⽤额外的图⽚⽂件 ，简化了开发过程。

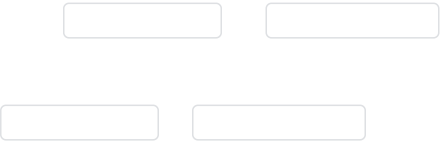
上⾯虽然说饿了挺多有点 ， 但是劣势也是很明显：

• 增加了⽂本⽂件的体积： 因为Base64编码将⼆进制数据转换成⽂本字符串 ，所以会增加CSS或

HTML⽂件的体积。在图⽚较⼤或数量较多时 ，这可能会导致⽂件变得庞⼤ ，从⽽导致⽹⻚加载速 度变慢。

• 缓存问题： 由于Base64图⽚被嵌⼊到了CSS或HTML⽂件中 ，如果图⽚内容有更新 ，那么整个⽂件 都需要重新加载 ，⽽⽆法使⽤缓存。相⽐于独⽴的图⽚⽂件 ，Base64图⽚对缓存的利⽤效率较低。

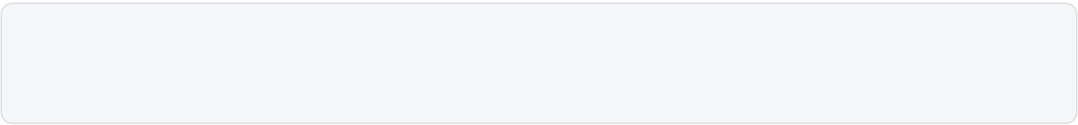
使⽤Base64图⽚在⼀些特定的场景下可以提供⼀些优势 ，但也需要权衡其带来的⼀些缺点。在实际开 发中 ，可以根据具体的需求和情况 ，选择是否使⽤Base64图⽚ 。 所以建议复⽤性很强, 变更率较低 ， 且 ⼩于 10KB 的图⽚⽂件 ， 可以考虑 base64

如何使⽤？ 有要介绍⼀下 webpack 插件了： url-loader 或 file-loader 要使⽤Webpack将图⽚⾃动转换为Base64编码 ，您需要执⾏以下步骤：

1. 安装依赖：⾸先 ，确保您已经安装了 常⽤的加载器。

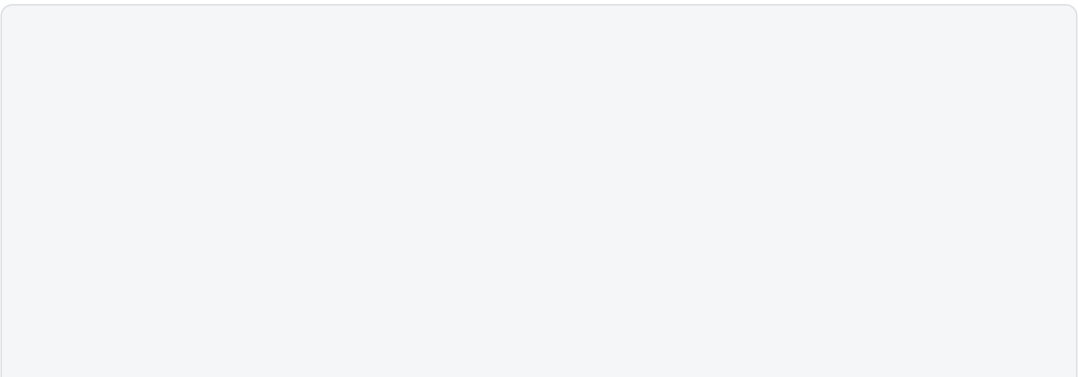
url-loader

或 file-loader ，它们是Webpack的两个

url-loader --save-dev

1 npm install

1. 配置Webpack：在Webpack的配置⽂件中 ，添加对图⽚⽂件的处理规则。您可以在 module .rules 数组中添加⼀个新的规则 ，以匹配图⽚⽂件的后缀。

1 module.exports = {*// ...module: {*

2 rules: [*// ...*

3 {

4 test: /\.(png|jpe?g|gif)$/i,

5 use: [

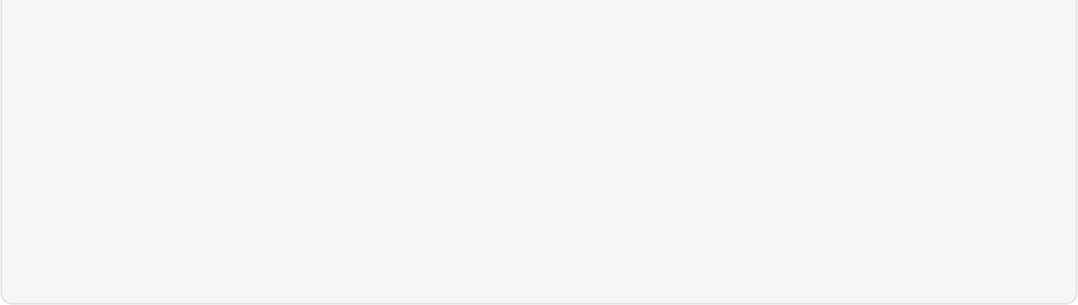
6 {

7 loader: 'url-loader',

8 options: {

9 limit: 8192, *// 设置图⽚⼤⼩的阈值，⼩于该值的图⽚会被转为Base64*

10 outputPath: 'images', *// 输出路径*



11 publicPath: 'images', *// 资源路径*

12 },

13 },

14 ],

15 },

16 ],

17 },*// ...*

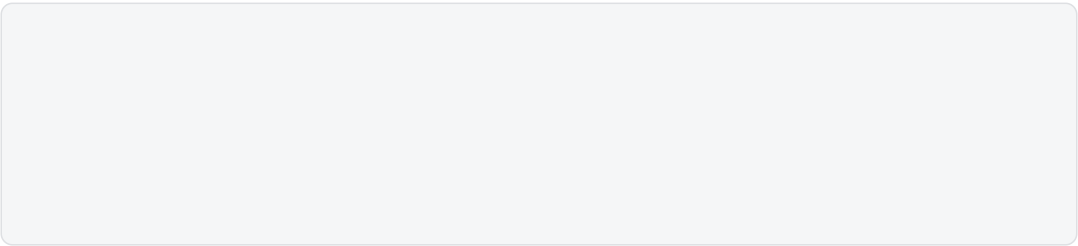
18 };

在上⾯的⽰例中 ，配置了⼀个处理 png 、 jpeg 、 jpg 和 gif 格式图⽚的规则。使⽤ url-

loader加载器 ，并设置了⼀些选项 ，例如 limit 限制了图⽚⼤⼩的阈值 ，⼩于该值的图⽚将会被

转换为Base64编码。

1. 在代码中引⽤图⽚： 在您的代码中 ，可以像引⽤普通图⽚⼀样引⽤图⽚⽂件 ，Webpack会根据配置 ⾃动将其转换为Base64编码。

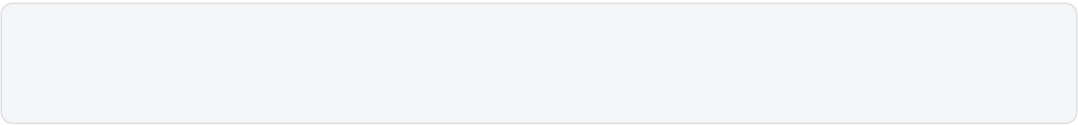
1  imgSrc  './path/to/image.png'; 2

3  imgElement = document.createElement('img');

4 imgElement.src = imgSrc;

5 document.body.appendChild(imgElement);

1. 构建项⽬ ：最后 ，使⽤Webpack构建项⽬ ，它会根据配置⾃动将符合规则的图⽚⽂件转换为 Base64编码 ，并将其嵌⼊到⽣成的输出⽂件中。

1 npx webpack

这样 ，Webpack就会⾃动将图⽚转换为Base64编码 ，并将其嵌⼊到⽣成的输出⽂件中。请注意 ，在使 ⽤Base64图⽚时 ，需要权衡⽂件⼤⼩和性能 ，适度使⽤Base64编码 ，避免过⼤的⽂件导致⽹⻚加载变 慢。

2. 使⽤ CDN 加载图⽚

CND 加载图⽚优势⾮常明显：

• 加速⽹⻚加载速度：CDN通过将图⽚资源分布在全球的多个节点上 ，使⽤⼾能从离⾃⼰最近的节点 获取资源 ，从⽽⼤⼤减少了⽹络延迟和加载时间。这可以提⾼⽹⻚的加载速度和⽤⼾体验。

• 减轻服务器负载：CDN充当了⼀个缓冲层 ， 当⽤⼾请求图⽚资源时 ，CDN会将图⽚资源从源服务器 获取并缓存在节点中 ，下次再有⽤⼾请求同⼀资源时 ，CDN会直接从节点返回 ，减少了对源服务器 的请求 ，分担了服务器的负载。

• 提⾼并发性能：CDN节点分布在不同地区 ，⽤⼾请求图⽚资源时可以从离他们最近的节点获取 ，这 可以减少⽹络拥塞和并发请求 ，提⾼了并发性能。

• 节省带宽成本：CDN的节点之间会⾃动选择最优路径 ，有效利⽤了带宽资源 ，减少了数据传输的成 本 ，尤其在⼤量图⽚资源请求时 ，能够带来显著的成本节省。

• 提供⾼可⽤性：CDN通过分布式存储和负载均衡技术 ，提供了⾼可⽤性和容错能⼒ 。即使某个节点 或源服务器发⽣故障 ，CDN会⾃动切换到其他可⽤节点 ，确保⽤⼾能够正常访问图⽚资源。

总之 ，使⽤CDN加载图⽚可以提⾼⽹⻚加载速度、降低服务器负载、提⾼并发性能、节省带宽成本， 并提供⾼可⽤性 ，从⽽改善⽤⼾体验和⽹站性能。

3. 图⽚懒加载

图⽚懒加载是⼀种在⽹站或应⽤中延迟加载图⽚的技术。 它的主要⽬的是减少⻚⾯的初始加载时间， 并提⾼⽤⼾的浏览体验。

• 原理： 图⽚懒加载的原理是只在⽤⼾需要时加载图⽚ ，⽽不是在⻚⾯初始加载时全部加载。这通常

通过将图⽚的真实地址存储在⾃定义属性（例如

data-s rc ） 中 ，⽽不是在

s rc 属性中。然

后 ，在图⽚进⼊浏览器视图时 ，通过JavaScript动态将

data-s rc 的值赋给

s rc 属性 ，触发图⽚

的加载。

• 优势： 图⽚懒加载可以显著减少初始⻚⾯的加载时间 ，特别是当⻚⾯中有⼤量图⽚时。 它使⻚⾯加 载变得更快 ，提⾼了⽤⼾的浏览体验。此外 ，懒加载还可以节省带宽和减轻服务器负载 ，因为只有 当图⽚进⼊视图时才会加载。

• 实现⽅法： 图⽚懒加载可以通过纯JavaScript实现 ，也可以使⽤现成的JavaScript库 ，如

LazyLoad.js、Intersection Observer API等 。这些库提供了⽅便的API和配置选项， 可以⾃定义懒加载的⾏为和效果。

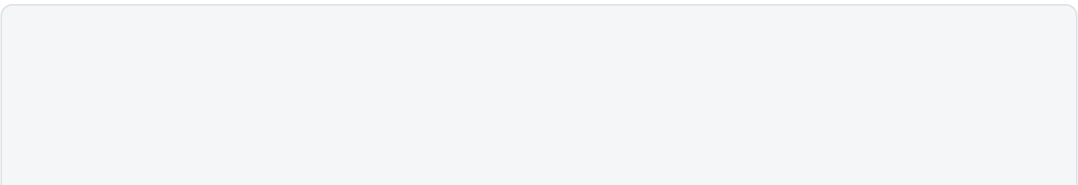
• 最佳实践：在使⽤图⽚懒加载时 ，可以考虑⼀些最佳实践。例如 ，设置⼀个占位符或加载中的动 画 ，以提供更好的⽤⼾体验。另外 ，确保在不⽀持JavaScript的情况下仍然可⽤ ，并为可访问性提 供替代⽂本（alt属性） 。此外 ，对于移动设备 ，可以考虑使⽤响应式图⽚来适应不同的屏幕分辨 率。

实现举例：

图⽚懒加载可以延迟图⽚的加载 ，只有当图⽚即将进⼊视⼝范围时才进⾏加载。这可以⼤⼤减轻⻚⾯ 的加载时间 ，并降低带宽消耗 ，提⾼了⽤⼾的体验。 以下是⼀些常⻅的实现⽅法：

1. Intersection Observer API

Intersection Observer API 是⼀种⽤于异步检查⽂档中元素与视⼝叠加程度的API。可以将 其⽤于检测图⽚是否已经进⼊视⼝ ，并根据需要进⾏相应的处理。

1  observer =  IntersectionObserver(  (entries) {

2 entries.forEach(  (entry) { (entry.isIntersecting) { lazyImage

= entry.target;

3 lazyImage.src = lazyImage.dataset.src;

4 observer.unobserve(lazyImage);

5 }

6 });

7 });

8

9  lazyImages = [...document.querySelectorAll(".lazy")];

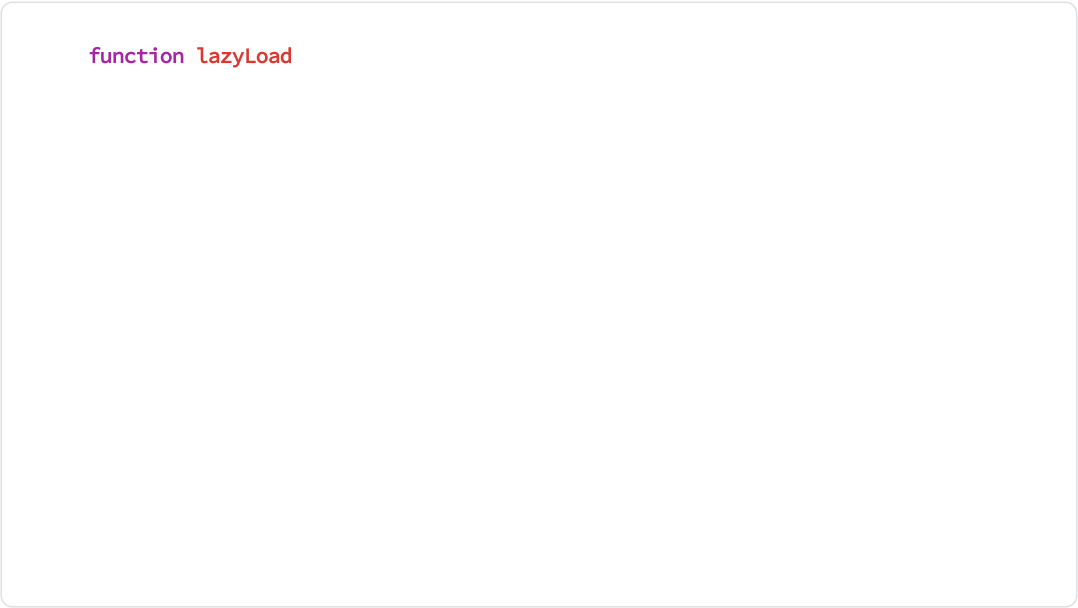
10 lazyImages.forEach(  (image) {

11 observer.observe(image);

12 });

1. ⾃定义监听器

或者 ，可以通过⾃定义监听器来实现懒加载。其中 ，应该避免在滚动事件处理程序中频繁进⾏图⽚加 载 ，因为这可能会影响性能。相反 ，使⽤⾃定义监听器只会在滚动停⽌时进⾏图⽚加载。

1 () { images = document.querySelectorAll(".lazy");

scrollTop = window.pageYOffset;

2 images.forEach((img) => { (img.offsetTop < window.innerHeight + scrollTop)

{

3 img.src = img.dataset.src;

4 img.classList.remove("lazy");

5 }

6 });

7 }

8

9  lazyLoadThrottleTimeout;

10 document.addEventListener("scroll",  () { (lazyLoadThrottleTimeout) {

11 clearTimeout(lazyLoadThrottleTimeout);

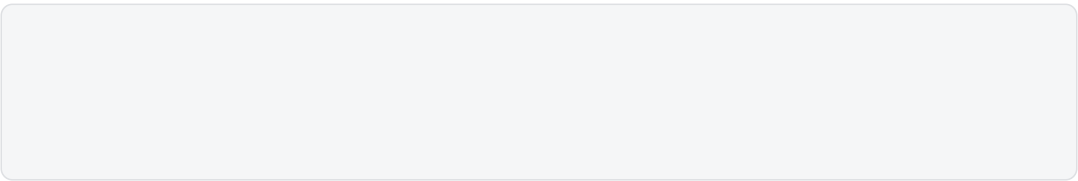
12 }

13 lazyLoadThrottleTimeout = setTimeout(lazyLoad, 20);

14 });

在这个例⼦中 ，我们使⽤了 setTimeout() 函数来延迟图⽚的加载 ，以避免在滚动事件的频繁触发 中对性能的影响。

⽆论使⽤哪种⽅法 ，都需要为需要懒加载的图⽚设置占位符 ，并将未加载的图⽚路径保存在 data 属 性中 ，以便在需要时进⾏加载。这些占位符可以是简单的 div 或样式类 ，⽤于预留图⽚的空间 ，避免⻚ ⾯布局的混乱。

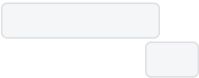
1 *<!-- 占位符⽰例* *-->*<div class="lazy-placeholder" style="background-color: #ddd;height: 500px;"</div*<!-- 图⽚⽰例* *-->*<img class="lazy" data-

src="path/to/image.jpg" alt="预览图 " />

2. 图⽚预加载

图⽚预加载是⼀种在⽹站或应⽤中提前加载图⽚资源的技术。 它的主要⽬的是在⽤⼾实际需要加载图 ⽚之前 ，将其提前下载到浏览器缓存中。

图⽚预加载通常是在⻚⾯加载过程中或在特定事件触发前异步加载图⽚资源。 通过使⽤

JavaScript ，可以在⽹⻚DOM元素中创建⼀个新的 Image 对象 ，并将要预加载的图⽚的URL赋 值给该对象的 s rc 属性。 浏览器在加载过程中会提前下载这些图⽚ ，并将其缓存起来 ，以备将来使 ⽤ 。

图⽚预加载可以使⽤原⽣JavaScript实现 ，也可以使⽤现成的JavaScript库 ，如

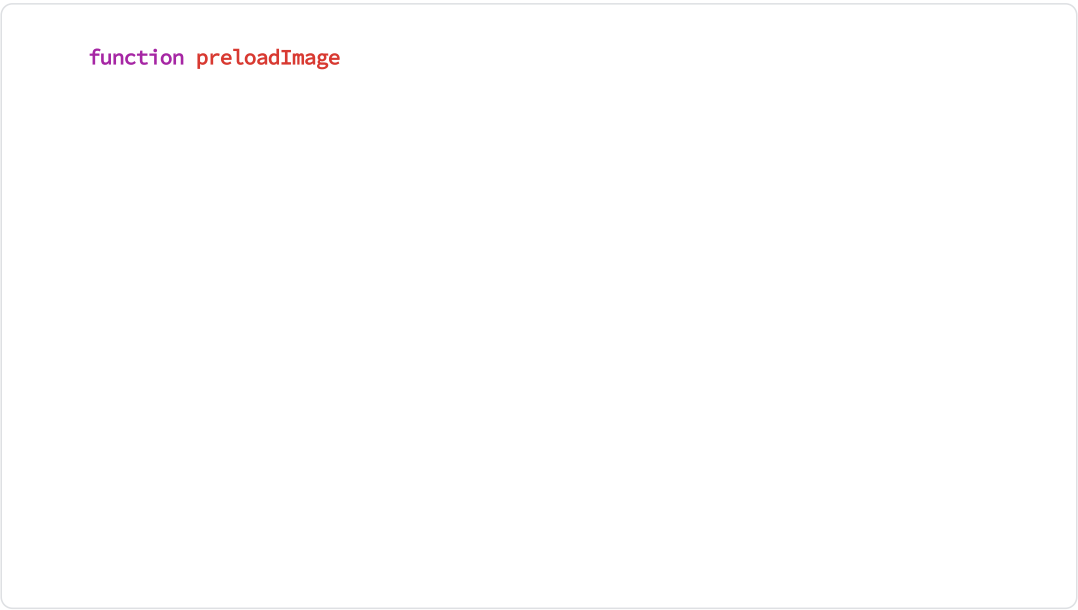
Preload .js、

等。这些库提供了⽅便的API和配置选项 ，可以灵活地控制预加载的⾏为和效果。

LazyLoad .js

实现图⽚预加载可以使⽤原⽣JavaScript或使⽤专⻔的JavaScript库。下⾯分别介绍两种⽅式的实现⽅ 法：

1. 使⽤原⽣JavaScript实现图⽚预加载：

(url) {

Promise(

(resolve, reject) {

1

 Image();

img =

img.onload = resolve; img.onerror = reject; img.src = url;

2

3

4

5

});

}

6

7

*// 调⽤预加载函数*

8

9

10

preloadImage('image.jpg')

.then( 

() {console.log('图⽚加载成功');*// 在此处可以执⾏加载成功后的操作，*

*例*

*如显⽰图⽚等*

})

11

12 () {console.error('图⽚加载失败');*//*

.catch(

*在此处可以执⾏加载失败后的操作，*

*例如显⽰错误信息等*

13 });

在上述代码中 ，我们定义了⼀个 preloadImage 函数 ，它使⽤ Image 对象来加载图⽚资源。通过 onload 事件和 onerror 事件来监听图⽚加载完成和加载错误的情况 ，并使⽤Promise对象进⾏异

步处理。

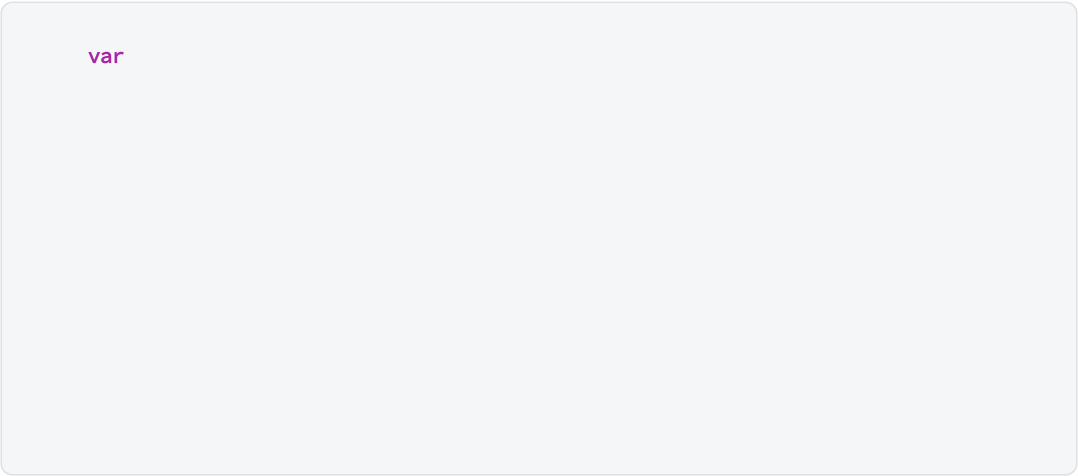
1. 使⽤JavaScript库实现图⽚预加载：

使⽤JavaScript库可以更简便地实现图⽚预加载 ，并提供更多的配置选项和功能。 以下以Preload.js库 为例进⾏说明：

⾸先 ，在HTML⽂件中引⼊Preload.js库：

1 <script src="preload.js"</script

然后 ，在JavaScript代码中使⽤Preload.js库来进⾏图⽚预加载：



1 preload =  createjs.LoadQueue();

2 preload.on("complete", handleComplete);

3 preload.on("error", handleError);

4 preload.loadFile('image.jpg');

5

6 () {console.log('图⽚加载成功');*// 在此处可以执⾏加载成功后的* *操作，例如显⽰图⽚等*

7 } 8

9 () {console.error('图⽚加载失败');*// 在此处可以执⾏加载失败后的* *操作，例如显⽰错误信息等*

10 }

在上述代码中 ，我们⾸先创建⼀个 LoadQueue 对象 ，并使⽤ on ⽅法来监听加载完成和加载错误的 事件。然后使⽤ loadFile ⽅法来指定要预加载的图⽚资源的URL。

当图⽚加载完成时 ， handleComplete 函数会被调⽤ ，我们可以在此处执⾏加载成功后的操作。 当 图⽚加载错误时 ， handle Error 函数会被调⽤ ，我们可以在此处执⾏加载失败后的操作。

以上是两种常⽤的实现图⽚预加载的⽅法 ，根据具体需求和项⽬情况选择合适的⽅式来实现图⽚预加 载。

2. 响应式加载图⽚

要在不同分辨率的设备上显⽰不同尺⼨的图⽚ ，你可以使⽤

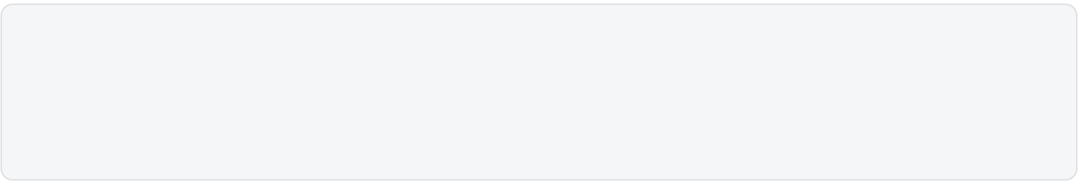
<source>

<picture>

元素和

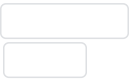
元素来

实现响应式图⽚ 。以下是⼀个⽰例：

1 <picture<source media="(min-width: 1200px)" srcset="large-image.jpg"<source

media="(min-width: 768px)" srcset="medium-image.jpg"<source srcset="small-

image.jpg"<img src="fallback-image.jpg" alt="Fallback Image"</picture

在上⾯的⽰例中 ， <picture> 元素内部有多个 <source> 元素 ，每个 <source> 元素通过

srcset 属性指定了对应分辨率下的图⽚链接。 media 属性可以⽤来指定在哪个分辨率下应⽤对应 的图⽚ 。如果没有任何 <source> 元素匹配当前设备的分辨率 ，那么就会使⽤ <img> 元素的 s rc 属性指定的图⽚链接。

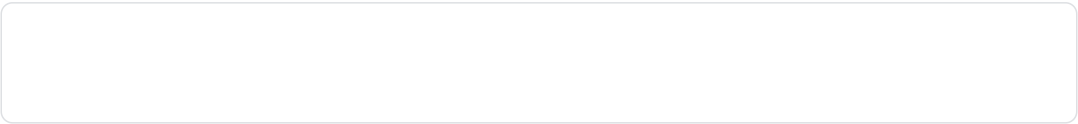
可以根据不同分辨率的设备 ，提供不同尺⼨和质量的图⽚ ，以优化⽤⼾的视觉体验和⻚⾯加载性能。

来实现⾃动根据设备分辨率加载不同的倍图：

有可以使⽤ webpack

responsive-loader

依赖安装:



webpack 配置⽰范

1 npm install responsive-loader sharp --save-dev



1 module.exports = {

2 entry: {...},

3 output: {...},module: {

4 rules: [

5 {

6 test: /\.(jpe?g|png|webp)$/i,

7 use: [

8 {

9 loader: "responsive-loader",

10 options: {

11 adapter: require('responsive-loader/sharp'),

12 sizes: [320, 640, 960, 1200, 1800, 2400],

13 placeholder: true,

14 placeholderSize: 20

15 },

16 },

17 ],

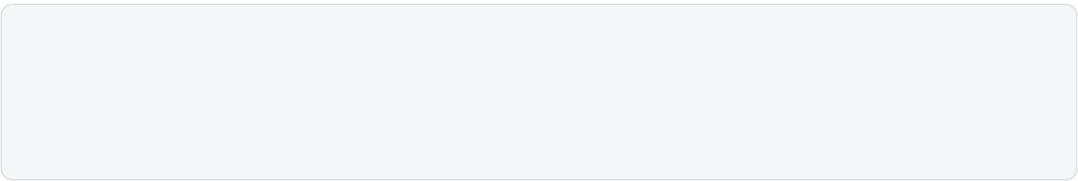
18 }

19 ]

20 },

21 }

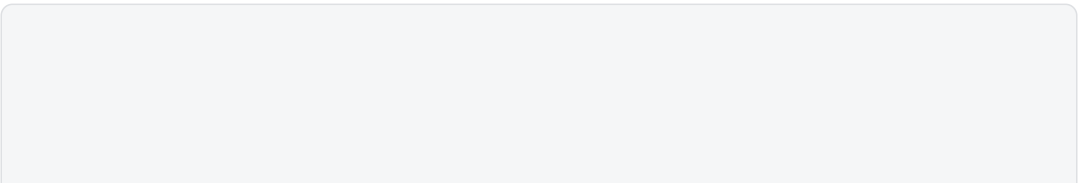
在CSS中使⽤它(如果使⽤多个⼤⼩ ，则只使⽤第⼀个调整⼤⼩的图像)

1 .myImage {background: url('myImage.jpg?size=1140');

2 }@ (max-width: 480px) {.myImage {background: url('myImage.jpg?size=480');}

3 }

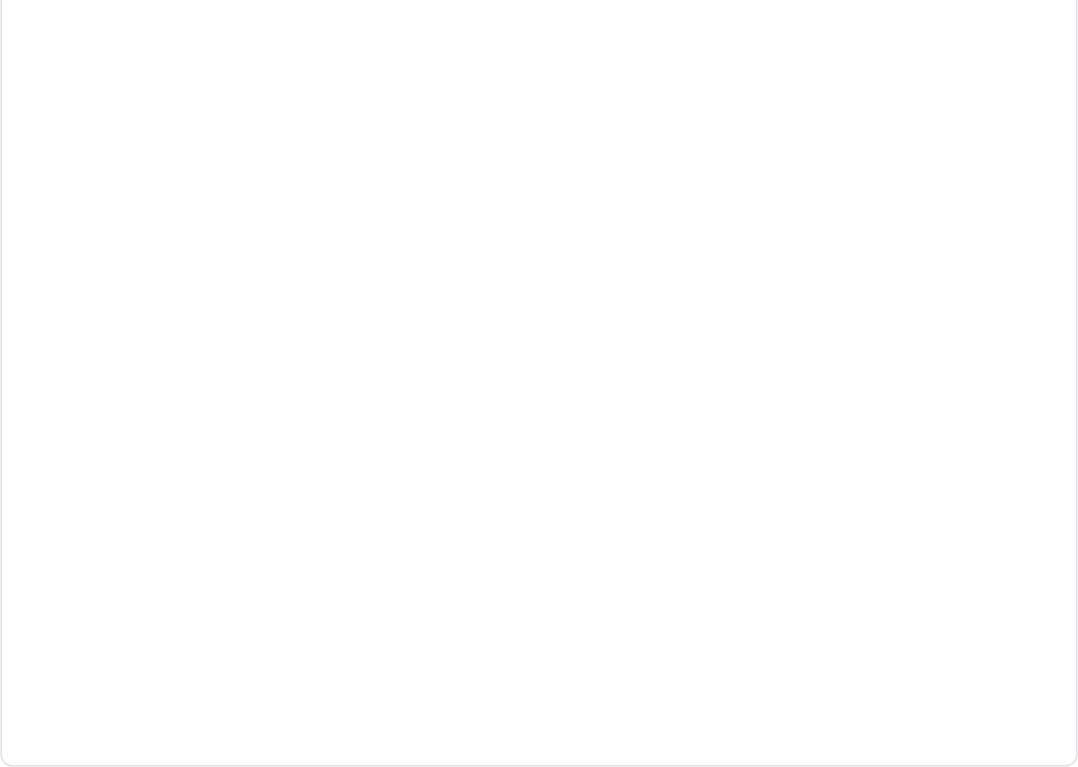
导⼊图⽚到 JS 中：

1  responsiveImage  'img/myImage.jpg?

sizes[]=300,sizes[]=600,sizes[]=1024,sizes[]=2048';

2  responsiveImageWebp  'img/myImage.jpg?

sizes[]=300,sizes[]=600,sizes[]=1024,sizes[]=2048&format=webp';



3

4 *// Outputs// responsiveImage.srcSet => '2fefae46cb857bc750fa5e5eed4a0cde-*

*300.jpg 300w,2fefae46cb857bc750fa5e5eed4a0cde-600.jpg*

*600w,2fefae46cb857bc750fa5e5eed4a0cde-600.jpg 600w ... '//*

*responsiveImage.images => [{height: 150, path:*

*'2fefae46cb857bc750fa5e5eed4a0cde-300.jpg', width: 300}, {height: 300, path:*

*'2fefae46cb857bc750fa5e5eed4a0cde-600.jpg', width: 600} ...]//*

*responsiveImage.src => '2fefae46cb857bc750fa5e5eed4a0cde-2048.jpg'//*

*responsiveImage.toString() => '2fefae46cb857bc750fa5e5eed4a0cde-2048.jpg'*

5 ...

6 <picture><source srcSet={responsiveImageWebp.srcSet} type='image/webp'

sizes='(min-width: 1024px) 1024px, 100vw'/>

7 <img

8 src={responsiveImage.src}

9 srcSet={responsiveImage.srcSet}

10 width={responsiveImage.width}

11 height={responsiveImage.height}

12 sizes='(min-width: 1024px) 1024px, 100vw'

13 loading="lazy" 14 />

15 </picture 16 ...

3. 渐进式加载图⽚

实现渐进式加载的主要思想是先加载⼀张较低分辨率的模糊图⽚ ，然后逐步加载更⾼分辨率的图⽚。 下⾯是实现渐进式加载图⽚的⼀般步骤：

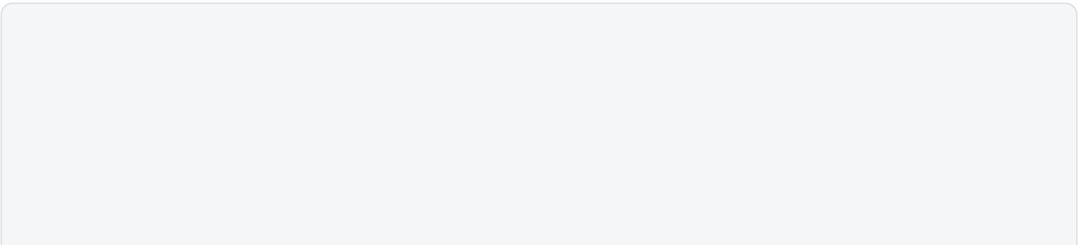
1. 创建⼀张模糊的低分辨率图⽚ 。可以使⽤图⽚处理⼯具将原始图⽚进⾏模糊处理 ，或者使⽤低分辨 率的缩略图作为初始图⽚。

2. 使⽤ <img> 标签将低分辨率的图⽚设置为 s rc 属性。这将⽴即加载并显⽰这张低分辨率的图

3. 。加载低分辨率图⽚时 ，同时加载⾼分辨率的原始图⽚ 。可以将⾼分辨率图⽚的URL设置为 data-s rc 等⾃定义属性 ，或者使⽤JavaScript动态加载⾼清图⽚。

4. 使⽤JavaScript监听图⽚的加载事件 ，在⾼分辨率图⽚加载完成后 ，将其替换低分辨率图⽚的 s rc 属性 ，以实现渐进式加载的效果。

下⾯是⼀个⽰例代码 ，演⽰了如何实现渐进式加载图⽚：

1 *<!-- HTML -->*<img src="blur-image.jpg" data-src="high-res-image.jpg" alt="Image"<script

2 *// JavaScript*

3  image = document.querySelector('img');

4

5 *// 监听⾼分辨率图⽚加载完成事件*

6 image.addEventListener('load', () => {*// 替换低分辨率图⽚的src属性*

7 image.src = image.dataset.src; 8 });

9 </script

在上⾯的⽰例中 ，⼀开始会显⽰⼀张模糊的低分辨率图⽚ ，然后在⾼分辨率图⽚加载完成后 ，将其替 换为⾼分辨率图⽚ ，实现了渐进式加载的效果。

渐进式加载图⽚可以减少⽤⼾等待时间 ，提供更好的⽤⼾体验。然⽽ ，需要注意的是 ，为了实现渐进 式加载 ，需要额外加载⾼分辨率的图⽚ ，这可能会增加⻚⾯加载时间和⽹络带宽消耗。 因此 ，开发者 需要在性能和⽤⼾体验之间进⾏权衡 ，并根据实际情况进⾏选择和优化。

79. OAuth2.0 是什么登录⽅式 【热度: 210】

OAuth2.0并不是⼀种特定的登录⽅式 ，⽽是⼀种授权框架 ，⽤于授权第三⽅应⽤访问⽤⼾的资源。 它 被⼴泛应⽤于⾝份验证和授权的场景中。

OAuth2.0通过引⼊授权服务器、资源服务器和客⼾端等⻆⾊ ，实现了⽤⼾授权和资源访问的分离。具 体流程如下：

1. ⽤⼾向客⼾端发起请求 ，请求访问某个资源。

2. 客⼾端将⽤⼾重定向到授权服务器 ，并携带⾃⼰的⾝份凭证（客⼾端ID）。

3. ⽤⼾在授权服务器登录 ，并授权客⼾端访问特定的资源。

4. 授权服务器验证⽤⼾⾝份 ，并⽣成访问令牌（Access Token）。

5. 授权服务器将访问令牌发送给客⼾端。

6. 客⼾端使⽤访问令牌向资源服务器请求访问资源。

7. 资源服务器验证访问令牌的有效性 ，并根据权限决定是否允许访问资源。

8. 资源服务器向客⼾端返回请求的资源。

在这个过程中 ，OAuth2.0通过访问令牌实现了⽤⼾和资源服务器之间的⾝份授权和资源访问分离。客 ⼾端⽆需知道或存储⽤⼾的凭证（如⽤⼾名和密码） ，⽽是使⽤访问令牌代表⽤⼾向资源服务器请求 资源 ，提供了更安全和便捷的授权⽅式。

以下是使⽤Fetch API来发起请求的⽰例代码：

// 1.客⼾端应⽤程序发起授权请求，重定向⽤⼾到授权服务器的登录⻚⾯

const authorizationEndpoint = 'https://example.com/oauth2/auth'

const clientId = 'your\_client\_id'

const redirectUri = 'https://yourapp.com/callback' //重定向

const scope = 'read write'

const state = 'random\_state\_value'

const authorizationUrl = `${authorizationEndpoint}?client\_id=${clientId}&redirect\_uri=${redirectUri}&scope=${scope}&state=${state}`

// 重定向⽤⼾到授权⻚⾯

window.location.href = authorizationUrl

// 2. 在回调URL中获取授权码

const callbackUrl = window.location.href

const urlParams = new URLSearchParams(callbackUrl.split('?')[1])

const authorizationCode = urlParams.get('code')

// 3. 客⼾端应⽤程序使⽤ 授权码 向授权服务器请求访问令牌

const tokenEndpoint = 'https://example.com/oauth2/token'

const clientSecret = 'your\_client\_secret'

const tokenData = {

  grant\_type: 'authorization\_code',

  code: authorizationCode,//授权码

  redirect\_uri: redirectUri,

  client\_id: clientId,

  client\_secret: clientSecret

}

// 使⽤Fetch API向授权服务器请求访问令牌

fetch(tokenEndpoint, {

  method: 'POST',

  headers: { 'Content-Type': 'application/x-www-form-urlencoded' },

  body: new URLSearchParams(tokenData)

})

  .then((response) => response.json())

  .then((data) => {//获取访问令牌成功

const accessToken = data.access\_token

    // 4. 客⼾端应⽤程序使⽤“访问令牌”向资源服务器请求受保护的资源

    const resourceEndpoint = 'https://example.com/api/resource' // 使⽤Fetch API请求受保护的资源

    fetch(resourceEndpoint, {

      method: 'GET',

      headers: {

        Authorization: `Bearer ${accessToken}`//访问令牌

      }

    })

      .then((response) => response.json())

      .then((resourceData) => {

        // 处理返回的资源数据

        console.log(resourceData)

      })

      .catch((error) => {

        console.error('Failed to retrieve resource:', error)

      })

  })

  .catch((error) => {

    console.error('Failed to retrieve access token:', error)

  })

请注意 ，上述代码使⽤了Fetch API来发送HTTP请求。 它使⽤了 fetch 函数来发送POST请求以获取 访问令牌 ，并使⽤了 Authorization 头部来发送访问令牌获取受保护的资源。确保你的浏览器⽀持 Fetch API ，或者在旧版浏览器中使⽤polyfill库来兼容。与之前的代码⽰例⼀样 ，你需要根据你的情况 替换URL和参数值。

80. 单点登录是如何实现的？ 【热度: 647】

单点登录

单点登录：Single Sign On ，简称SSO。⽤⼾只要登录⼀次 ，就可以访问所有相关信任应⽤的资源。企 业⾥⾯⽤的会⽐较多 ，有很多内⽹平台 ，但是只要在⼀个系统登录就可以。

实现⽅案

• 单⼀域名：可以把 cookie 种在根域名下实现单点登录

• 多域名： 常⽤ CAS来解决 ，新增⼀个认证中⼼的服务。CAS（Central Authentication Service）是 实现SSO单点登录的框架

CAS实现单点登录的流程：

1. ⽤⼾访问系统A ，判断未登录 ，则直接跳到认证中⼼⻚⾯

2. 在认证中⼼⻚⾯输⼊账号 ，密码 ，⽣成令牌 ，重定向到系统A

3. 在系统A拿到令牌到认证中⼼去认证 ，认证通过 ，则建⽴对话

4. ⽤⼾访问系统B ，发现没有有效会话 ，则重定向到认证中⼼

5. 认证中⼼发现有全局会话 ，新建令牌 ，重定向到系统B

6. 在系统B使⽤令牌去认证中⼼验证 ，验证成功后 ，建议系统B的局部会话。

关键点

下⾯是举例来详细说明CAS实现单点登录的流程： ⼀ 、第⼀次访问系统A

1. ⽤⼾访问系统A ([www.app1.com)，](https://gitee.com/link?target=http%3A%2F%2Fwww.app1.com%29%25EF%25BC%258C)跳转认证中⼼ client([www.sso.com)，](https://gitee.com/link?target=http%3A%2F%2Fwww.sso.com%29%25EF%25BC%258C)然后输⼊⽤⼾名 ，密码 登录 ，然后认证中⼼ serverSSO 把 cookieSSO 种在认证中⼼的域名下 ([www.sso.com)，](https://gitee.com/link?target=http%3A%2F%2Fwww.sso.com%29%25EF%25BC%258C)重定向到 系统A ，并且带上⽣成的 ticket 参数 ([www.app1.com?ticket](https://gitee.com/link?target=http%3A%2F%2Fwww.app1.com%3Fticket) =xxx)

2. 系统A ([www.app1.com?ticket](https://gitee.com/link?target=http%3A%2F%2Fwww.app1.com%3Fticket) =xxx)请求系统A的后端 serverA ，serverA 去 serverSSO 验证 ，通 过后 ，将cookieA种在 [www.app1.com下](https://gitee.com/link?target=http%3A%2F%2Fwww.app1.com%25E4%25B8%258B)

⼆ 、第⼆次访问系统A 直接携带 cookieA 去访问后端 ，验证通过后 ，即登录成功。

三、第三次访问系统B

1. 访问系统B ([www.app2.com)，](https://gitee.com/link?target=http%3A%2F%2Fwww.app2.com%29%25EF%25BC%258C) 跳转到认证中⼼ client([www.sso.com)，](https://gitee.com/link?target=http%3A%2F%2Fwww.sso.com%29%25EF%25BC%258C) 这个时候会把认证中⼼的 cookieSSO也携带上 ，发现⽤⼾已登录过 ，则直接重定向到系统B（[www.app2.com](http://www.app2.com/)） ， 并且带上 ⽣成的ticket参数（[www.app2.com?ticket](http://www.app2.com/?ticket) =xxx）

2. 系统B ([www.app2.com?ticket](https://gitee.com/link?target=http%3A%2F%2Fwww.app2.com%3Fticket) =xxx)请求系统B的后端 serverB ，serverB 去 serverSSO 验证 ，通

过后 ，将cookieB种在[www.app2.com](http://www.app2.com/)下 注意cookie⽣成时机及种的位置。

• cookieSSO ，SSO域名下的cookie

• cookieA ，系统A域名下的cookie

• cookieB ，系统B域名下的cookie

81. 常⻅的登录鉴权⽅式有哪些？ 【热度: 557】

前端登录鉴权的⽅式主要有以下⼏种：

1. 基于Session Cookie的鉴权：

◦ cookie: ⽤⼾在登录成功后 ，服务器会⽣成⼀个包含⽤⼾信息的Cookie ，并返回给前端。前端在 后续的请求中会⾃动携带这个Cookie ，在服务器端进⾏验证和识别⽤⼾⾝份。

◦ Session: ⽤⼾登录成功后 ，服务器会在后端保存⽤⼾的登录状态信息 ，并⽣成⼀个唯⼀的

Session ID ，将这个Session ID 返回给前端。前端在后续的请求中需要携带这个Session ID ，服 务器通过Session ID 来验证⽤⼾⾝份。

2. 单点登录（Single Sign-On ，SSO） ：单点登录是⼀种将多个应⽤系统进⾏集成的认证⽅式。⽤⼾ 只需登录⼀次 ，即可在多个系统中完成认证 ，避免了重复登录的⿇烦。 常⻅的单点登录协议有CAS

（Central Authentication Service） 、SAML（Security Assertion Markup Language）等。

3. OpenID Connect（OIDC） ：OIDC是基于OAuth2.0的⾝份验证协议 ，通过在认证和授权过程中引 ⼊⾝份提供者 ，使得⽤⼾可以使⽤第三⽅⾝份提供者（如Google、 Facebook等） 进⾏登录和授 权 ，从⽽实现⽤⼾⾝份验证和授权的功能。

4. OAuth2.0：OAuth2.0是⼀个授权框架 ，⽤于授权第三⽅应⽤访问⽤⼾的资源。 它通过授权服务器 颁发令牌（Token） ，使得第三⽅应⽤可以代表⽤⼾获取资源的权限 ，⽽⽆需知道⽤⼾的真实凭 证。

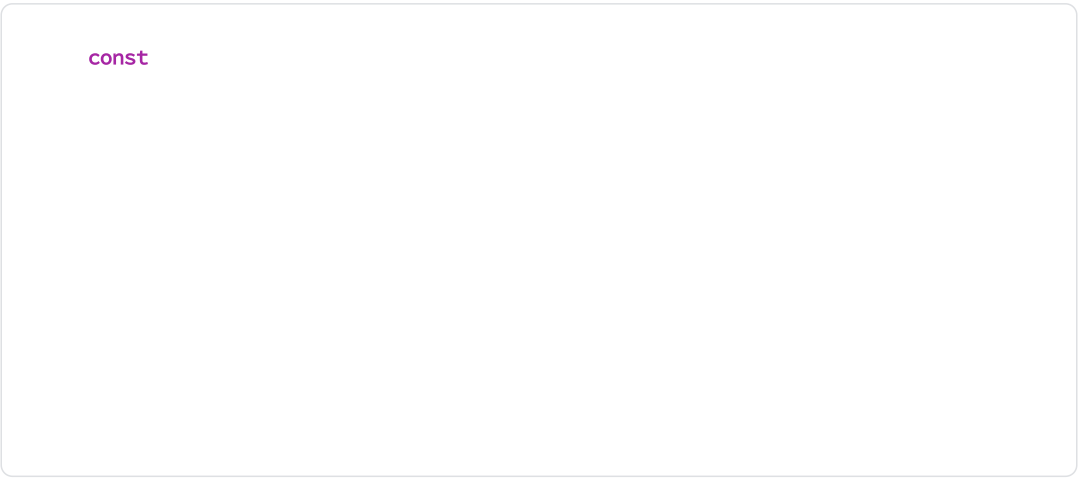
5. LDAP（ Lightweight Directory Access Protocol）： LDAP是⼀种⽤于访问和维护分布式⽬录服务 的协议。在登录鉴权中 ，LDAP常⽤于验证⽤⼾的⾝份信息 ，如⽤⼾名和密码 ，通过与LDAP服务器 进⾏通信来进⾏⽤⼾⾝份验证。

6. 2FA（Two-Factor Authentication） ：⼆次验证是⼀种提供额外安全层的⾝份验证⽅式。与传统的 ⽤⼾名和密码登录不同 ，2FA需要⽤⼾提供第⼆个验证因素 ，如⼿机验证码、指纹识别、硬件令牌 等 ，以提⾼账⼾的安全性。

82. 需要在跨域请求中携带另外⼀个域名下的 Cookie 该如何操 作？ 【热度: 254】

在跨域请求中携带另外⼀个域名下的 Cookie ，需要通过设置响应头部的Access-Control-Allow- Credentials字段为true ，并且请求头部中添加withCredentials字段为true。

在服务端需要设置响应头部的Access-Control-Allow-Origin字段为指定的域名 ，表⽰允许指定域名的 跨域请求携带Cookie。

下⾯是⼀个⽰例代码（ Node.js）：

1 express = require('express');

2  app = express();

3 app.use((req, res, next) => {

4 res.setHeader('Access-Control-Allow-Origin', 'http://example.com');

5 res.setHeader('Access-Control-Allow-Credentials', 'true');

6 next();

7 });

8 app.get('/api/data', (req, res) => {*// 处理请求*

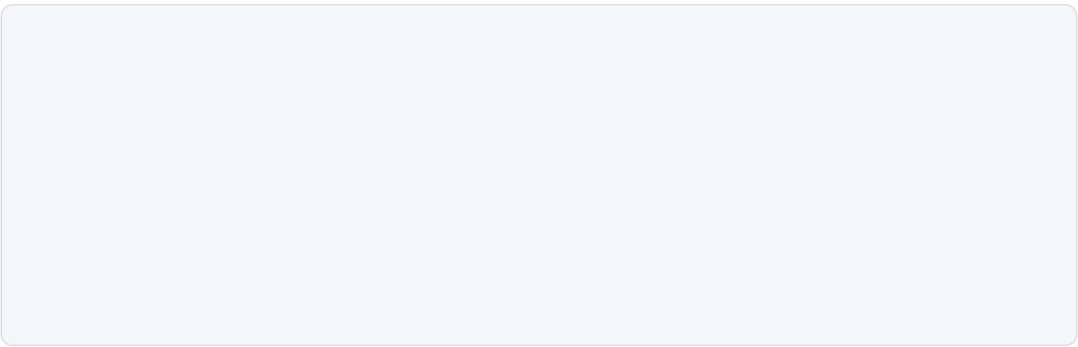
9 res.send('Response Data');

10 });

11 app.listen(3000, () => {console.log('Server is running on port 3000');

12 });

在客⼾端发起跨域请求时 ，需要设置请求头部的withCredentials字段为true ，⽰例代码 （JavaScript）：

1 fetch('http://example.com/api/data', {

2 credentials: 'include',

3 })

4 .then(response => response.text())

5 .then(data => {console.log(data);

6 })

7 .catch(error => {console.error('Error:', error);

8 });

以上代码中 ，Access-Control-Allow-Origin设置为'http://example.com' ，表⽰允许该域名的跨域请求 携带Cookie。fetch请求的参数中 ，credentials设置为'include'表⽰请求中携带Cookie。

83. vite 和 webpack 在热更新上有啥区别？ 【热度: 530】

Vite 和 Webpack 在热更新上有⼀些区别：

1. 模块级别的热更新：Vite 使⽤浏览器原⽣的 ES 模块系统 ，可以实现模块级别的热更新 ，即只更新 修改的模块 ，⽽不需要刷新整个⻚⾯ 。这样可以提供更快的开发迭代速度。⽽在 Webpack 中 ，热 更新是基于⽂件级别的 ，需要重新构建并刷新整个⻚⾯。

2. 开发环境下的⽆构建：Vite 在开发环境下不会对代码进⾏打包构建 ，⽽是直接利⽤浏览器原⽣的模 块导⼊功能 ，通过 HTTP 服务器提供模块的即时响应。这样可以避免了构建和重新编译的时间 ，更 快地反映出代码的修改。⽽在 Webpack 中 ，每次修改代码都需要重新构建和编译 ，耗费⼀定的时 间。

3. 构建环境下的优化：尽管 Vite 在开发环境下不进⾏打包构建 ，但在⽣产环境下 ，它会通过预构建的 ⽅式⽣成⾼性能的静态资源 ，以提⾼⻚⾯加载速度。⽽ Webpack 则通过将所有模块打包成

bundle ⽂件 ，进⾏代码压缩和优化 ，以及使⽤各种插件和配置来优化构建结果。

总的来说 ，Vite 在热更新上⽐ Webpack 更加快速和精细化 ，能够在开发过程中提供更好的开发体验和

更快的反馈速度。但是 ，Webpack 在构建环境下有更多的优化和功能 ，适⽤于更复杂的项⽬需求。 以下是 Vite 和 Webpack 在热更新⽅⾯的对⽐表格：

|  |  |  |
| --- | --- | --- |
| **特点** | **Vite** | **Webpack** |
| 实时热更新 | 支持模块级 别的热更  新，即只更 新修改的模 块，无需刷 新整个页面 | **支持文件级** **别的热更**  **新，修改任** **何文件都会** **触发整个应** **用的重新构** **建和刷新** |
| 构建速度 | 在开发环境  **下，利用浏** **览器原生的** **模块导入功** **能，不需要** **进行打包构** **建，启动速** **度更快** | **需要进行打** **包构建，每** **次修改代码** **都需要重新** **构建和编**  **译，相对较** **慢** |
| 开发体验 | **提供更好的**  开发体验， 修改代码后 快速反馈， 无需等待全 量构建 | **反馈速度较** **慢，需要等** **待每次构建** **和编译完成** |
| 适用场景 | 适用于中小 型项目，追 求开发效率 的前端项目 | **适用于大型** **项目，有复** **杂需求和更** **多构建优化** **的前端项目** |

**84.封装一个请求超时，发起重试的代码【热度：789】**

关键词：请求重试

看过很多请求超时重试的样例，很多都是基于axios interceptors实现的。但是有没有牛逼的原生方 式实现呢?

最近在看fbjs库里面的代码，发现里面有一个超时重试的代码，只有一百多行代码，封装的极其牛 逼。

不过这里的代码是Flow 类型检测的代码，而且有一些外部小依赖，之后要翻译成 ts代码。 这里简单介绍一下fbjs 这个库

fbjs(Facebook JavaScript) 是一个由Facebook开发和维护的JavaScript 工具库。它提供了一组 通用的JavaScript 功能和实用工具，用于辅助开发大型、高性能的JavaScript 应用程序。

说到这⼉了 ， 直接上完整代码

interface InitWithRetries extends RequestInit {

  fetchTimeout?: number | null;

  retryDelays?: number[] | null;

}

const DEFAULT\_TIMEOUT = 1000 \* 1.5;

const DEFAULT\_RETRIES = [0, 0];

const fetchWithRetries = (url: string, initWithRetries?: InitWithRetries): Promise<any> => {

  // fetchTimeout 请求超时时间

  // 请求

  const { fetchTimeout, retryDelays, ...init } = initWithRetries || {};

  // 超时时间

  const \_fetchTimeout = fetchTimeout != null ? fetchTimeout : DEFAULT\_TIMEOUT;

  // 重复时间数组

  const \_retryDelays = retryDelays != null ? retryDelays : DEFAULT\_RETRIES;

  // 开始时间

  let requestStartTime = 0;

  // 重试请求索引

  let requestsAttempted = 0;

  return new Promise((resolve, reject) => {

    // 申明发送请求方法

    const sendTimedRequest = (): void => {

      // 自增索引与请求次数

      requestsAttempted++;

      // 发起请求时间

      requestStartTime = Date.now();

      // 是否需要处理后续请求

      let isRequestAlive = true;

      // 发起请求

      const request = fetch(url, init);

      // 请求超时情况

      const requestTimeout = setTimeout(() => {

        // 需要阻断正常的请求返回

        isRequestAlive = false;

        // 需要重新发起请求

        if (shouldRetry(requestsAttempted)) {

          console.warn("fetchWithRetries: HTTP timeout, retrying.");

          retryRequest();

        } else {

          reject(new Error(

            `fetchWithRetries(): Failed to get response from server, tried ${requestsAttempted} times.`,

          ));

        }

      }, \_fetchTimeout);

      // 正常请求发起

      request.then(response => {

        // 正常请求返回的场景， 清空定时器

        clearTimeout(requestTimeout);

        // 如果进入了超时流程， 那么正常返回的逻辑， 就直接阻断

        if (isRequestAlive) {

          if (response.status >= 200 && response.status < 300) {

            resolve(response);

          } else if (shouldRetry(requestsAttempted)) {

            console.warn("fetchWithRetries: HTTP error, retrying.");

            retryRequest();

          } else {

            const error: any = new Error(`response error.`);

            error.response = response;

            reject(error);

          }

        }

      }).catch(error => {

        clearTimeout(requestTimeout);

        if (shouldRetry(requestsAttempted)) {

          retryRequest();

        } else {

          reject(error);

        }

      });

    };

    // 发起重复请求

    const retryRequest = (): void => {

      // 重复请求 delay 时间

      const retryDelay = \_retryDelays[requestsAttempted - 1];

      // 重复请求开始时间

      const retryStartTime = requestStartTime + retryDelay;

      // 延迟时间

      const timeout = retryStartTime - Date.now() > 0 ? retryStartTime - Date.now() : 0;

      // 重复请求

      setTimeout(sendTimedRequest, timeout);

    };

    // 是否可以发起重复请求

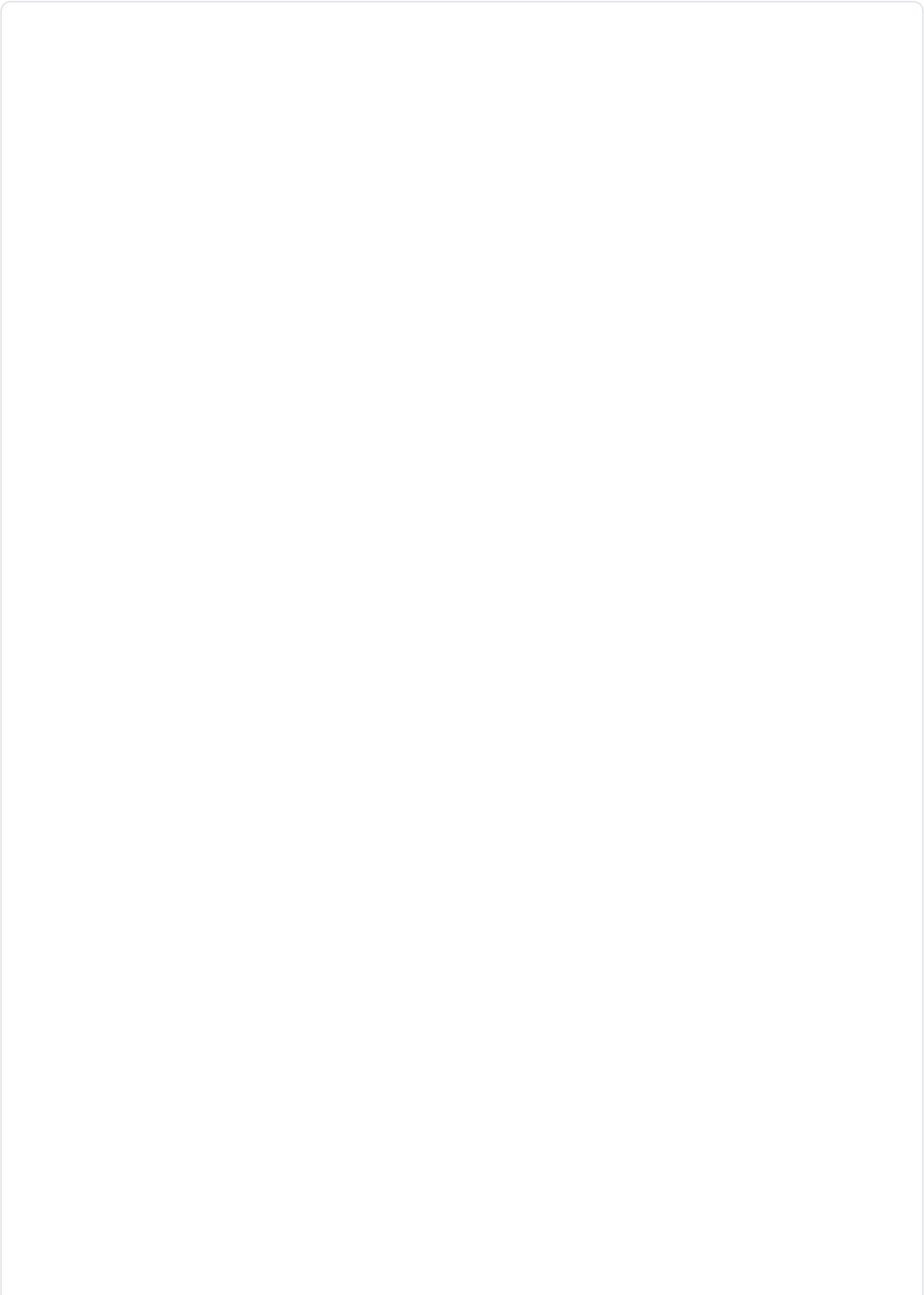
    const shouldRetry = (attempt: number): boolean => attempt <= \_retryDelays.length;

    sendTimedRequest();

  });

};

fetchWithRetries("http://127.0.0.1:3000/user/")

1 - interface InitWithRetries extends RequestInit {

2 fetchTimeout?: number | null;

3 retryDelays?: number| null;

4 }

5

6 const DEFAULT\_TIMEOUT = 10001.5;

7 const DEFAULT\_RETRIES = [0, 0]; 8

9 const fetchWithRetries = (url: string, initWithRetries?: InitWithRetries): Promise<any=> {*// fetchTimeout 请求超时时间// 请求const { fetchTimeout,*

*retryDelays, ...init } = initWithRetries || {};*

10 *// 超时时间const \_fetchTimeout = fetchTimeout != null ? fetchTimeout : DEFAULT\_TIMEOUT;*

11 *// 重复时间数组const \_retryDelays = retryDelays != null ? retryDelays : DEFAULT\_RETRIES;*

12 *// 开始时间let requestStartTime = 0;*

13 *// 重试请求索引let requestsAttempted = 0;*

14 return new Promise((resolve, reject) => {*// 申明发送请求⽅法const sendTimedRequest = (): void => {// ⾃增索引与请求次数*

15 requestsAttempted++; 16 *//发起请求时间*

17 requestStartTime = Date.now();

18 *// 是否需要处理后续请求let isRequestAlive = true;* 19 *// 发起请求const request = fetch(url, init);*

20 *// 请求超时情况const requestTimeout = setTimeout(() => {// 需要阻断正常的请求返回*

21 isRequestAlive = false;

22 *// 需要重新发起请求if (shouldRetry(requestsAttempted))*

*{console.warn("fetchWithRetries: HTTP timeout, retrying.");*

23 retryRequest();

24 } else {

25 reject(new Error(

26 `fetchWithRetries(): Failed to get response from server, tried

${requestsAttempted} times.`,

27 ));

28 }

29 }, \_fetchTimeout);

30 *// 正常请求发起*

31 request.then(response => {*// 正常请求返回的场景，* *清空定时器*

32 clearTimeout(requestTimeout);

33 *// 如果进⼊了超时流程，* *那么正常返回的逻辑，* *就直接阻断if (isRequestAlive) {if (response.status >= 200 && response.status < 300) {*

34 resolve(response);

35 } else if (shouldRetry(requestsAttempted))

{console.warn("fetchWithRetries: HTTP error, retrying.");

36 retryRequest();

37 } else {const error: any = new Error(

38 response error. 39 );

40 error.response = response;

41 reject(error);

42 }

43 }

44 }).catch(error => {

45 clearTimeout(requestTimeout);if (shouldRetry(requestsAttempted)) {

46 retryRequest();

47 } else {

48 reject(error);

49 }

50 });

51 };

52 *//发起重复请求const retryRequest = (): void => {// 重复请求* *delay 时间const retryDelay = \_retryDelays[requestsAttempted - 1];*

53 *// 重复请求开始时间const retryStartTime = requestStartTime + retryDelay;*

54 *// 延迟时间const timeout = retryStartTime - Date.now() > 0 ? retryStartTime - Date.now() : 0;*

55 *// 重复请求*

56 setTimeout(sendTimedRequest, timeout);

57 };

58 *// 是否可以发起重复请求const shouldRetry = (attempt: number): boolean => attempt <= \_retryDelays.length;*

59 sendTimedRequest();

60 });

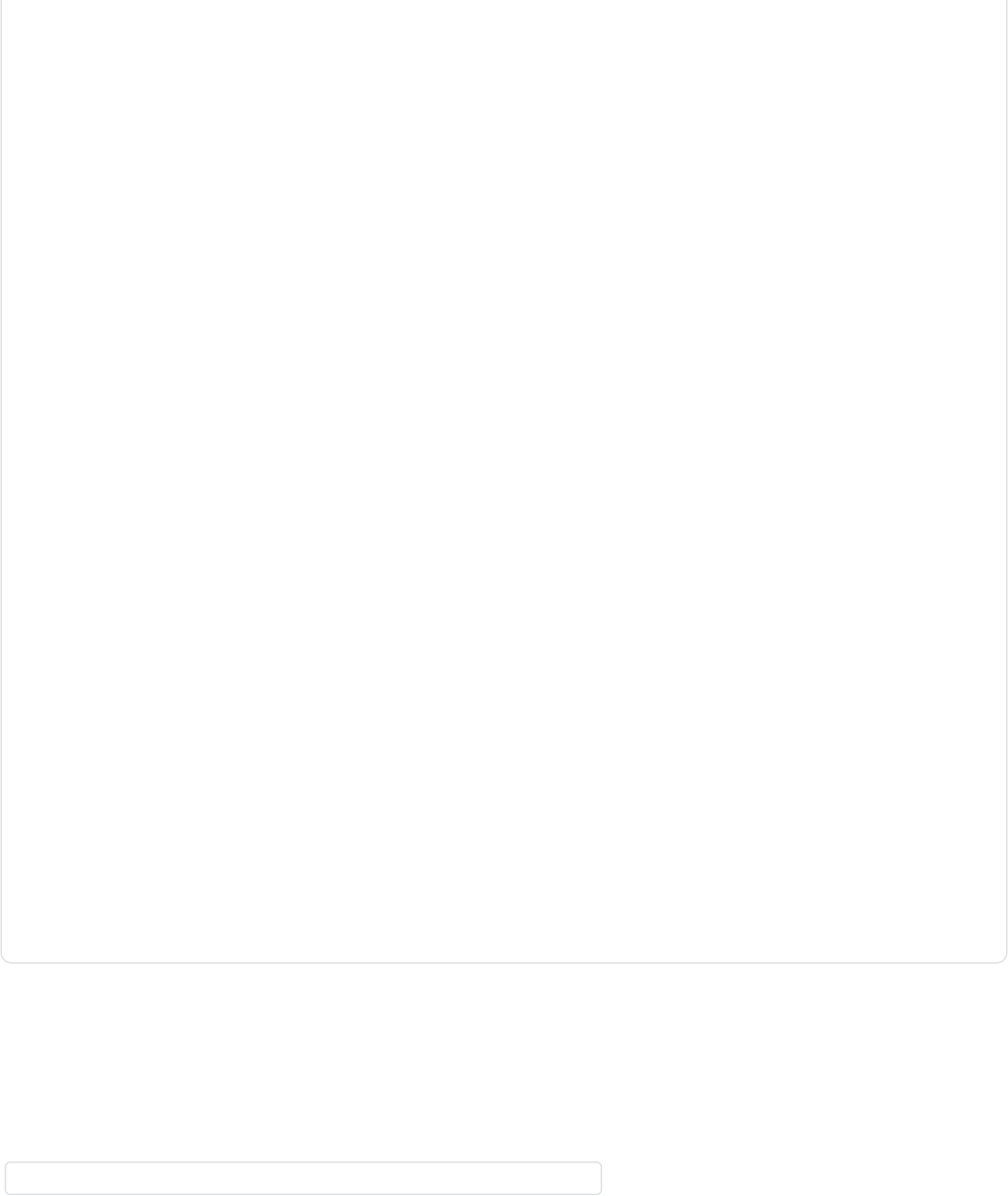
61 };

62 fetchWithRetries("http://127.0.0.1:3000/user/")

85. 前端如何设置请求超时时间timeout 【热度: 890】

关键词：请求超时时间

1. axios全局设置⽹络超时

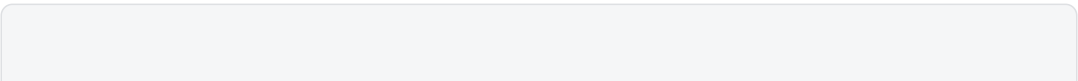
 axios .defaults .timeout = 10 \* 1000; // 10s

2. 单独对某个请求设置⽹络超时

axios .post(url, params, {timeout : 1000}) .then(res => {

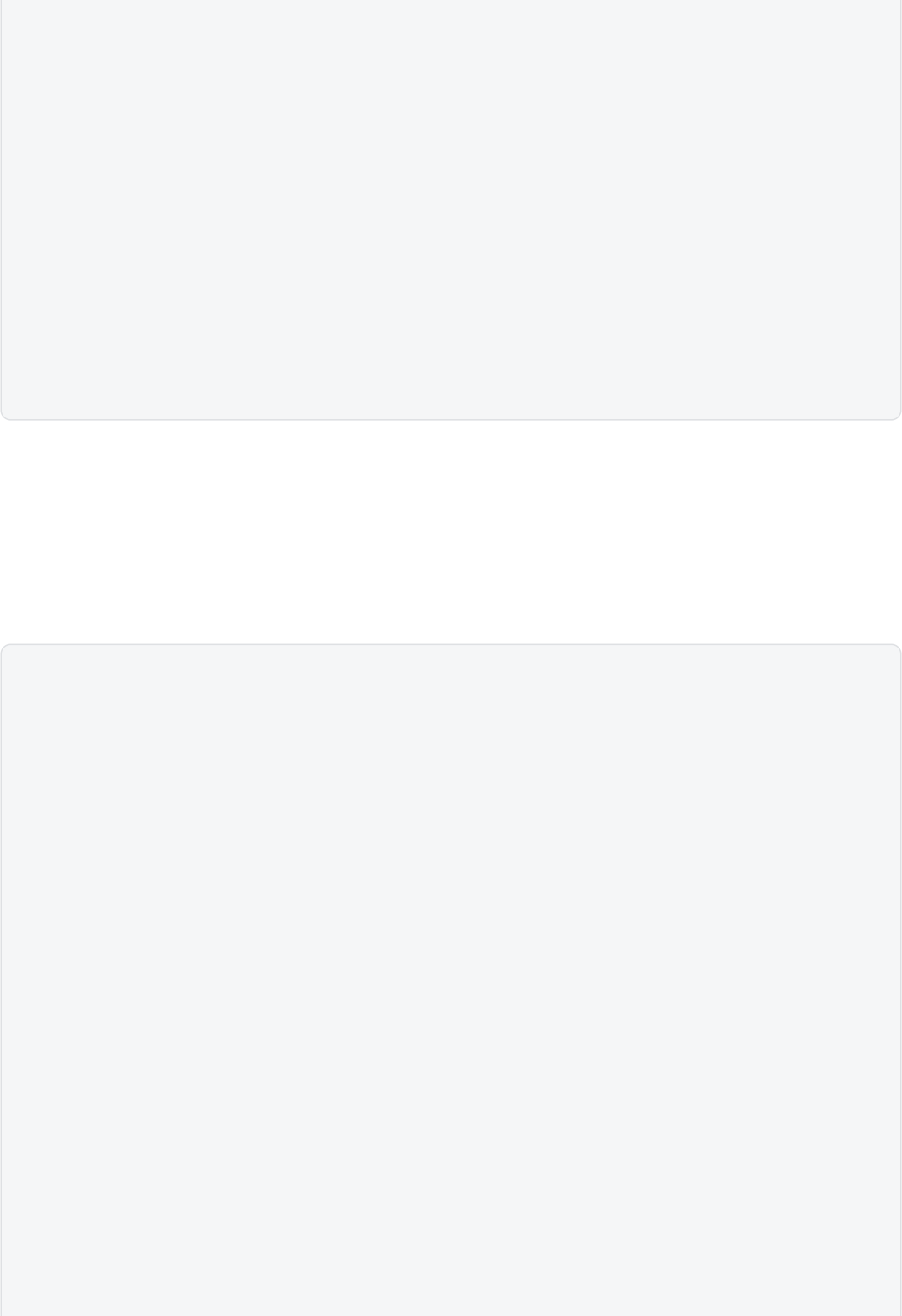
console .log(res); }) .catch(err=> { console .log(err); }) })

3.webpack的dev的proxyTable的超时时间设置

1 dev: { *// Paths*

2 assetsSubDirectory: 'static', *// 静态资源⽂件夹*

3 assetsPublicPath: '/', *//发布路径//代理配置表，在这⾥可以配置特定的请求代理到对* *应的API接⼝//使⽤⽅法：https://vuejs-templates.github.io/webpack/proxy.html*

4 proxyTable: {'/api': {

5 timeout: 30000, *// 请求超时时间*

6 target: 'http://127.0.0.1:3006', *// ⽬标接⼝域名*

7 changeOrigin:  , *// 是否跨域*

8 pathRewrite: {'^/api': '' *// 重写接⼝* 9 }

10 },*// Various Dev Server settings*

11 host: 'localhost', *// can be overwritten by process.env.HOST*

12 port: 4200, *// can be overwritten by process.env.PORT, if port is in use,*

*a free one will be determined*

13 }

4.axios请求超时⾃动重新请求

有时候因项⽬需求 ，要在接⼝请求超时或者获取数据失败时 ，重新请求1次 ，或者更多次。具体的配置 步骤和⽅法如下：

因为是要在请求超时或者获取数据失败时 ，进⾏重新请求设置 ，那么我们肯定是要在请求返回拦截器 ⾥⾯设置

1  axios  "axios";

2

3  Axios = axios.create({ *// 下⾯两个属性，⽤来设置，请求失败或者超时，⾃动重新请求* *的次数和间隙时间*

4 retry: 2, *// 请求次数*

5 retryInterval: 1000 *// 求期间隙* 6 ......

7 });

8 *//请求前拦截*

9 Axios.interceptors.request.use(config => {  config

10 },  (error) {  Promise.reject(error)

11 }

12 );

13 *//请求后返回数据拦截*

14 Axios.interceptors.response.use(res => {  res

15 }, (res) { config = res.config;*//如果配置不存在* *或重试属性未设置，抛出promise错误* *(!config || !config.retry)* 

*Promise.reject(res);//设置⼀个变量记录重新请求的次数*

16 config.retryCount = config.retryCount || 0;*// 检查重新请求的次数是否超过我们*

*设定的请求次数* *(config.retryCount >= config.retry) {* *Promise.reject(res);* 17 }*//重新请求的次数⾃增*

18 config.retryCount += 1;*//创建新的Promise来处理重新请求的间隙* *back =* 

*Promise(**(resolve) {console.log("接⼝"+config.url+"请求超时，重新请求")*

19 setTimeout(  () {

20 resolve();

21 }, config.retryInterval|| 1);

22 });*//返回axios的实体，重试请求* *back.then(**(){*

*Axios(config);*

23 });

24 }

25 );

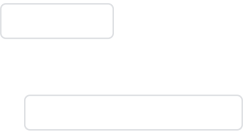
26   Axios

86. nodejs 如何充分利⽤多核 CPU? 【热度: 725】

总所周知 ， NodeJS 是单线程执⾏任务 ， 不同于 浏览器还可以使⽤ web worker 等⼿段多线程执⾏任 务。那么 NodeJS 中 ， 是如何充分利⽤物理机的多核 CPU 呢？

有三种⽅式

在 Node.js 中 ，JS 也是单线程的 ，只有⼀个主线程⽤于执⾏任务。但是 ，在 Node.js 中可以使⽤多进 程来利⽤多核机器 ，以充分利⽤系统资源。

• Node.js 提供了 cluster 模块 ，可以轻松创建⼦进程来处理任务。通过将任务分配给不同的⼦进 程 ，每个⼦进程可以在⾃⼰的线程上执⾏任务 ，从⽽实现多核机器的利⽤ 。

模块 ，可以创建真正的多线程应⽤程序。这个模块允许开

worker\_threads

• Node.js 也提供了

发者创建和管理多个⼯作线程 ，每个线程都可以独⽴地执⾏任务。

• 利⽤的是 Node.js 的事件循环机制和异步⾮阻塞的 I/O 操作。 Node.js 使⽤事件驱动的模型来处理 请求 ， 当有请求到达时 ，Node.js 将其放⼊事件队列中 ，然后通过事件循环来处理这些请求。在等 待 I/O 操作的过程中 ，Node.js 不会阻塞其他请求的处理 ，⽽是继续处理其他请求。这样 ，即使

JavaScript 是单线程的 ，但在实际运⾏中 ，多个请求可以同时处理 ，充分利⽤了多核系统的能⼒。

如果 Nodejs 只写同步代码 ， 是否意味着⽆法充分利⽤多核优势？

如果在 Node.js 的开发过程中只使⽤同步代码⽽不使⽤异步代码或集群模块 ，那么意味着⽆法充分利 ⽤机器多核优势。

Node.js的事件驱动和异步⾮阻塞的特性使得它在处理⼤量并发请求时⾮常⾼效。 当你使⽤异步代码

时 ，可以在等待 I/O 操作的过程中继续处理其他请求 ，从⽽提⾼系统的吞吐量和响应速度。⽽同步代码 会阻塞事件循环 ，使得只能按顺序处理请求 ，⽆法同时处理多个请求 ，⽆法充分利⽤多核系统的能

⼒。

另外 ，如果你不使⽤集群模块 ，那么只有⼀个 Node.js 进程在运⾏ ，⽆法充分利⽤多核系统的资源。 使⽤集群模块可以创建多个⼦进程 ，每个⼦进程在⼀个核⼼上运⾏ ，从⽽并⾏处理请求 ，提⾼系统的 并发能⼒。

为何 nodejs 异步代码就可以充分利⽤多核优势了？

Node.js的异步代码可以充分利⽤多核优势 ，主要有两个原因：

1. 事件驱动和⾮阻塞 I/O： Node.js采⽤事件驱动的模型 ，通过使⽤异步⾮阻塞 I/O 操作 ，可以在等待 I/O 操作完成的同时继续处理其他请求。这意味着在⼀个请求等待 I/O 的过程中 ，Node.js 可以同时 处理其他请求 ，充分利⽤了 CPU 的多核能⼒ 。每个核⼼可以处理⼀个请求 ，从⽽提⾼系统的并发 能⼒和吞吐量。

2. 事件循环机制： Node.js的事件循环机制使得异步代码可以⾼效地处理⼤量并发请求。事件循环机 制通过将请求注册为事件监听器 ，并在合适的时候触发事件处理函数 ，从⽽实现异步处理。这样⼀ 来 ，即使有⼤量并发请求 ，也能够通过事件循环机制避免线程切换的开销 ，提⾼系统的性能。

需要注意的是 ，虽然 Node.js 的事件驱动和异步⾮阻塞的特性使得它能够充分利⽤多核优势 ，但是在 处理 CPU 密集型任务时 ，仍然可能受限于单线程的性能。在这种情况下 ，可以通过使⽤集群模块来创 建多个⼦进程 ，在每个核⼼上运⾏独⽴的 Node.js 进程 ，从⽽实现并⾏处理 ，提⾼系统的性能。

异步就能充分利⽤ CPU 原理是啥？

当Node.js使⽤异步代码时 ，服务器的其他CPU核⼼是在⼯作的。

这是因为Node.js的事件驱动模型和⾮阻塞I/O使得在等待I/O操作完成时 ，可以同时处理其他请求。 当⼀个请求在等待I/O操作时 ，CPU核⼼可以被⽤于处理其他请求 ，⽽不是空闲等待。

这种⽅式可以充分利⽤服务器上的多个CPU核⼼ ，提⾼系统的并发能⼒和吞吐量。通过同时处理多个 请求 ，可以更有效地利⽤服务器的资源 ，提⾼系统的性能。

87. 后端⼀次性返回树形结构数据 ，数据量⾮常⼤, 前端该如何 处理？ 【热度: 171】

当后端⼀次性返回的树形结构数据量⾮常⼤ ，导致前端⼀次性计算和渲染会栈溢出的情况时 ，可以考 虑以下⼏种处理⽅式：

1. 分批处理：将⼤量的树形数据分为多个批次进⾏处理和渲染。前端可以通过递归或循环的⽅式 ，每 次处理⼀部分数据 ，并在渲染完成后再处理下⼀部分数据。这样可以避免⼀次性处理⼤量数据造成 栈溢出的问题。

2. 异步处理：使⽤异步处理的⽅式进⾏数据的计算和渲染。前端可以利⽤JavaScript的异步特性 ，将 数据处理和渲染任务分为多个异步任务 ，并通过事件循环机制依次执⾏这些任务。这样可以避免⼀ 次性计算和渲染⼤量数据导致栈溢出的问题。

3. 虚拟化渲染：使⽤虚拟化渲染技术 ，只渲染当前可⻅区域的树节点 ，⽽不是全部节点。可以根据⻚ ⾯的滚动位置和⽤⼾操作 ，只渲染当前需要展⽰的节点 ，⽽对于不可⻅的节点只保留其占位符。这 样可以减少实际渲染的节点数量 ，降低内存占⽤和渲染时间。

4. 数据分级处理：对于树形结构数据 ，可以考虑对数据进⾏分级处理。将数据根据节点的层级关系进 ⾏分组 ，每次只处理和渲染当前层级的节点数据。这样可以减少每次处理的数据量 ，降低栈溢出的 ⻛险。

根据具体的业务需求和技术实现情况 ，可以选择适合的处理⽅式来解决栈溢出问题。 同时 ，也可以结 合多种处理⽅式来提⾼⻚⾯性能和⽤⼾体验。

88. 你认为组件封装的⼀些基本准则是什么？ 【热度: 625】

组件封装的⼀些基本准则包括：

1. 单⼀职责原则：⼀个组件应该具有单⼀的功能 ，并且只负责完成该功能 ，避免组件过于庞⼤和复 杂。

2. ⾼内聚低耦合：组件内部的各个部分之间应该紧密相关 ，组件与其他组件之间应该尽量解耦 ，减少 对外部的依赖。

3. 易⽤性：组件应该易于使⽤ ，提供清晰的接⼝和⽂档 ，使⽤⼾能够⽅便地使⽤组件。

4. 可扩展性：组件应该具有良好的扩展性 ，能够⽅便地添加新的功能或进⾏修改 ，同时不影响已有的 功能。

5. 可重⽤性：组件应该是可重⽤的 ，能够在多个项⽬中使⽤ ，减少重复开发的⼯作量。

6. ⾼效性：组件应该具有⾼性能和低资源消耗的特点 ，不会成为整个系统的性能瓶颈。

7. 安全性：组件应该具有安全性 ，能够防⽌恶意使⽤或攻击。

8. 可测试性：组件应该容易进⾏单元测试和集成测试 ，以保证组件的质量和稳定性。

89. ⻚⾯加载速度提升（性能优化） 应该从哪些反向来思考？ 【热度: 1,099】

⻚⾯加载优化

「⻚⾯加载链路+流程优化+协作⽅」 的多级分类思考

• ⻚⾯启动

◦ service worker 缓存重要的静态资源

◦ ⻚⾯保活

• 资源加载

◦ ⽹络连接

▪ NDS

• 减少 NDS 解析

• NDA 预解析

▪ HTTP

• 开启 HTTP2 多路复⽤

• 优化核⼼请求链路

◦ HTML 加载

▪ 内容优化

• 代码压缩

• 代码精简(tailwindcss)

• 服务端渲染 SSG

▪ 流程优化

• 服务端渲染 SSR

• 流式渲染 • 预渲染

◦ 静态资源加载

▪ 内容优化

• JS、CSS 代码压缩

• 均衡资源包体积：复⽤代码抽离为⼀份资源打包、 同时开启

• 精简代码 • 雪碧图

• 动态图⽚降质量

• 动态 polyfill (根据浏览器的⽀持情况 ，动态加载需要的 polyfill（填充）脚本)

• 不常变的资源单独打包

• 根据浏览器版本打包 ， ⾼版本浏览器 ， 直接使⽤ es6 作为输出⽂件

▪ 流程优化

• 配置前端缓存: 资源、请求

• 使⽤ CDN

• CDN 优化

• 协调资源加载优先级

• 动态资源转静态 CDN 链接加载(例如⼤图⽚等)

• 静态资源使⽤ service worker 离线存储

• ⾮⾸屏资源懒加载

• 资源和业务请求预加载

• 微前端加载应⽤

• 微组件加载核⼼模块资源

• 代码执⾏

◦ 减少执⾏

▪ 减少重复渲染

▪ ⼤体量计算场景 ， 尽量使⽤缓存函数

▪ 使⽤防抖节流

◦ 速度提升

▪ 使⽤ worker 多线程加速

▪ 充分利⽤异步请求的线下之间来进⾏核⼼代码的加载或者执⾏

▪ wasm 处理⼤量计算场景

▪ 渲染⾼耗时场景 ， 迁移到 canvas 、虚拟 dom 等

▪ 动态渲染：动态渲染可视区内容 ， 例如图⽚懒加载等；

◦ 流程优化

▪ ⾮⾸屏模块 ， 延迟加载与渲染

▪ longtask 任务拆分执⾏

▪ 利⽤请求闲暇时间 ， 请求后续⻚⾯资源

• 数据获取

◦ 内容优化

▪ 减少请求、合并请求、 BFF

▪ ⾸屏数据使⽤模板注⼊到前端应⽤

◦ 流程优化

▪ 数据预请求

▪ 常量数据缓存

▪ ⾮⾸屏请求 ，延迟到⾸屏加载完成之后请求

▪ 请求并⾏

• 渲染相关

◦ 虚拟列表

◦ 延迟渲染

◦ 减少重绘重排

◦ 图⽚预加载到内存

90. 前端⽇志埋点 SDK 设计思路 【热度: 755】

前端⽇志埋点 SDK 设计思路

既然涉及到了⽇志和埋点 ，分析⼀下需求是啥：

• ⾃动化上报 ⻚⾯ PV、 UV。 如果能⾃动化上报⻚⾯性能 ， ⽤⼾点击路径⾏为 ，就更好了。

• ⾃动上报⻚⾯异常。

• 发送埋点信息的时候 ， 不影响性能 ， 不阻碍⻚⾯主流程加载和请求发送。

• 能够⾃定义⽇志发送 ， ⽇志 scope、 key、value。

SDK 设计

sdk 的设计主要围绕以下⼏个话题来进⾏：

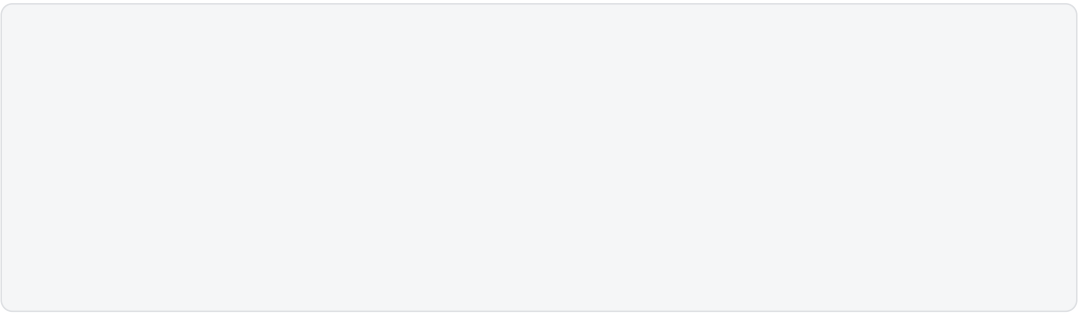
• SDK 初始化

• 数据发送

• ⾃定义错误上报

• 初始化错误监控

• ⾃定义⽇志上报 最基本使⽤

1  StatisticSDK  'StatisticSDK';

2 *//全局初始化⼀次window.insSDK =*  *StatisticSDK('uuid-12345');*

3

4 <button onClick={() => {

5 window.insSDK.event('click', 'confirm');

6 ...// 其他业务代码 7 }}>确认</button

数据发送

数据发送是⼀个最基础的api ，后⾯的功能都要基于此进⾏。 这⾥介绍使⽤

navigator .sendBeacon 来发送请求；具体原因如下 使⽤ navigator .sendBeacon() ⽅法有以下优势：

1. 异步操作： navigator .sendBeacon() ⽅法会在后台异步地发送数据 ，不会阻塞⻚⾯的其他 操作。这意味着即使⻚⾯正在卸载或关闭 ，该⽅法也可以继续发送数据 ，确保数据的可靠性。

2. ⾼可靠性： navigator .sendBeacon() ⽅法会尽可能地保证数据的传输成功。 它使⽤浏览器 内部机制进⾏发送 ，具有更⾼的可靠性和稳定性。即使在⽹络连接不稳定或断开的情况下 ，该⽅法 也会尝试发送数据 ，确保数据的完整性。

navigator .sendBeacon() ⽅法会⾃动处理数据的发送细节 ，⽆需⼿动设置请 求头、响应处理等。 它会将数据封装成 POST 请求 ，并⾃动设置请求头和数据编码 ，使开发者能够 更专注于业务逻辑的处理。

3. ⾃动化处理：

navigator .sendBeacon() ⽅法⽀持跨域发送数据。在⼀些情况下 ，例如使⽤第

4. 跨域⽀持：

三⽅统计服务等 ，可能需要将数据发送到其他域名下的服务器 ，此时使⽤

navigator .sendBeacon()

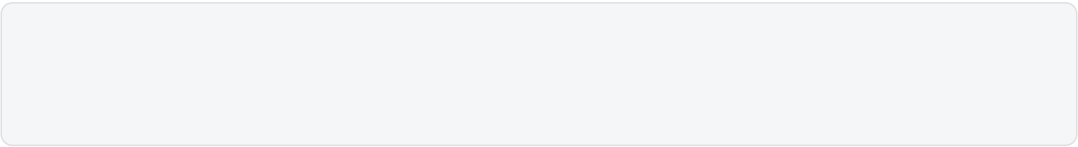
navigator .sendBeacon() ⽅法发送的数据是以 POST 请求的形式发送到服务 器 ，通常会将数据以表单数据或 JSON 格式进⾏封装。 因此 ，后端服务器需要正确处理这些数据 ，并 进⾏相应的解析和处理。

⽅法可以避免跨域问题。 需要注意的是，

navigator .sendBeacon ⽤法

简单介绍⼀下

语法：

1 navigator.sendBeacon(url);

2 navigator.sendBeacon(url, data);

参数 • url

◦ url 参数表明 data 将要被发送到的⽹络地址。

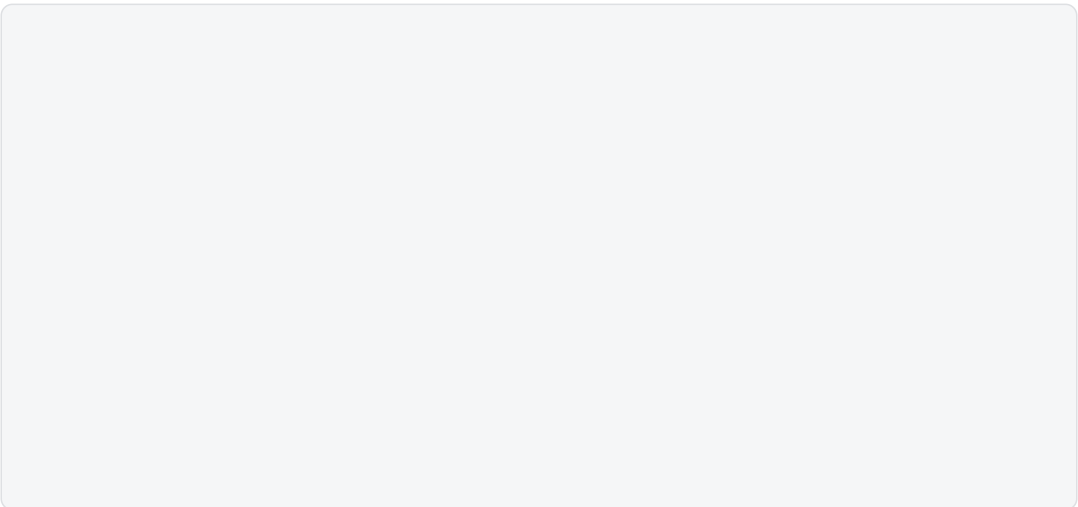
• data 可选

◦ data 参数是将要发送的

ArrayBuffer、ArrayBufferView、 Blob、 DOMString、

|  |  |
| --- | --- |
|  |  |
| FormData 或 URLSearchParams | 类型的数据。 |

发送代码实现如下

1  StatisticSDK {

2 constructor(productID, baseURL) { .productID = productID; .baseURL =

baseURL;

3 }

4 send(query = {}) {

5 query.productID =  .productID;

6  data =  URLSearchParams();  ( [key, value] of

Object.entries(query)) {

7 data.append(key, value);

8 }

9 navigator.sendBeacon( .baseURL, data);

10 }

11 }



⽤⼾⾏为与⽇志上报

⽤⼾⾏为主要涉及到的是事件上报和 pv 曝光 ， 借助 send 实现即可。



性能上报

1 StatisticSDK {

2 constructor(productID, baseURL) { .productID = productID; .baseURL =

baseURL;

3 }

4 send(query = {}) {

5 query.productID =  .productID;

6  data =  URLSearchParams();  ( [key, value] of

Object.entries(query)) {

7 data.append(key, value);

8 }

9 navigator.sendBeacon( .baseURL, data);

10 }

11 event(key, value = {}) { .send({ event: key, ...value })

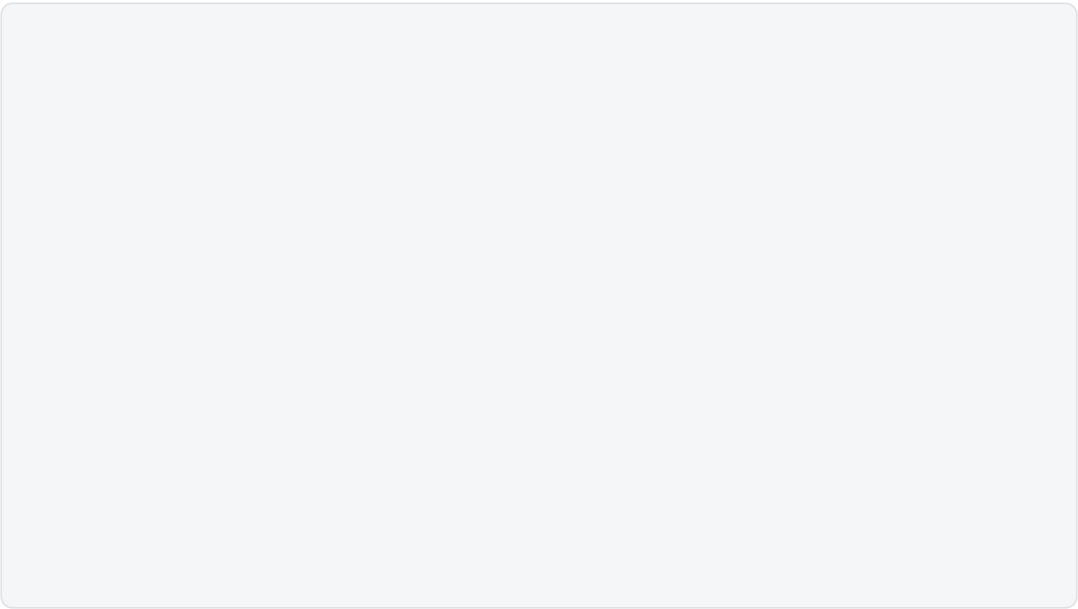
12 }

13 pv() { .event('pv')

14 }

15 }

性能主要涉及的 api 为 performance.timing ⾥⾯的时间内容；

1  StatisticSDK {

2 constructor(productID, baseURL) { .productID = productID; .baseURL =

baseURL;

3 }

4 send(query = {}) {

5 query.productID =  .productID;

6  data =  URLSearchParams();  ( [key, value] of

Object.entries(query)) {

7 data.append(key, value);

8 }

9 navigator.sendBeacon( .baseURL, data);

10 }

11 *// ....*

12 initPerformance() { .send({ event: 'performance', ...performance.timing })

13 }

14 }

错误上报

错误上报分两类：

⼀个是 dom 操作错误与JS 错误报警 ， 也是常说的运⾏时报错。 该类报错直接可以通过

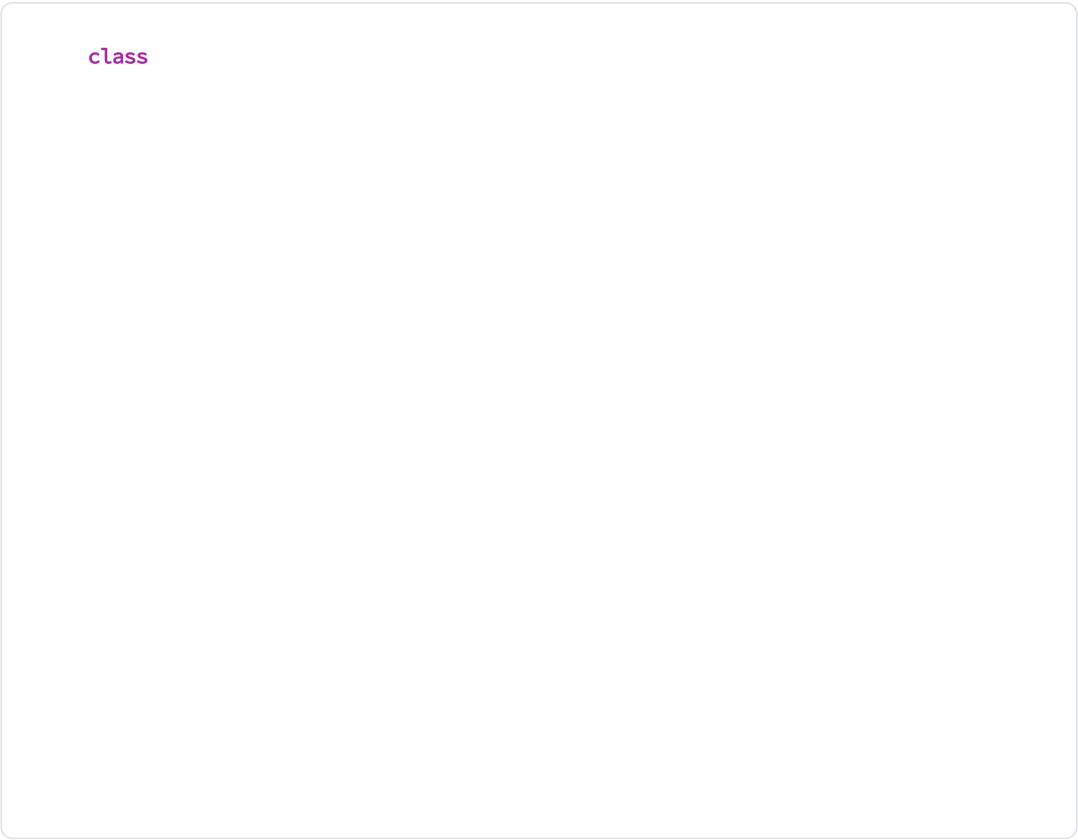
addEventListener( 'error ') 监控即可；

另⼀个是Promise内部抛出的错误是⽆法被error捕获到的 ，这时需要⽤

unhandled rejection

事

件。



1 StatisticSDK {

2 constructor(productID, baseURL) { .productID = productID; .baseURL =

baseURL;

3 }

4 send(query = {}) {

5 query.productID =  .productID;

6  data =  URLSearchParams();  ( [key, value] of

Object.entries(query)) {

7 data.append(key, value);

8 }

9 navigator.sendBeacon( .baseURL, data);

10 }

11 *// ....*

12 error(err, errInfo = {}) { { message, stack } = err; .send({ event:

'error', message, stack, ...errInfo })

13 }

14 initErrorListenner() {window.addEventListener('error', event =>

{ .error(error);

15 })window.addEventListener('unhandledrejection', event => { .error(

Error(event.reason), { type: 'unhandledrejection' })

16 })

17 }

18 }

React 和 vue 错误边界

错误边界是希望当应⽤内部发⽣渲染错误时 ，不会整个⻚⾯崩溃。我们提前给它设置⼀个兜底组件， 并且可以细化粒度 ，只有发⽣错误的部分被替换成这个「兜底组件」 ，不⾄于整个⻚⾯都不能正常⼯

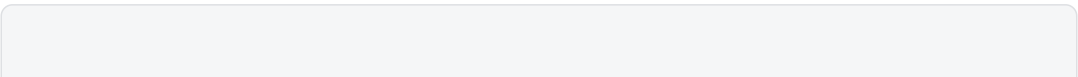
作。

React

可以使⽤类组件错误边界来进⾏处理 ， 涉及到的⽣命周期为： componentDidCatch ；

getDerivedState From Error

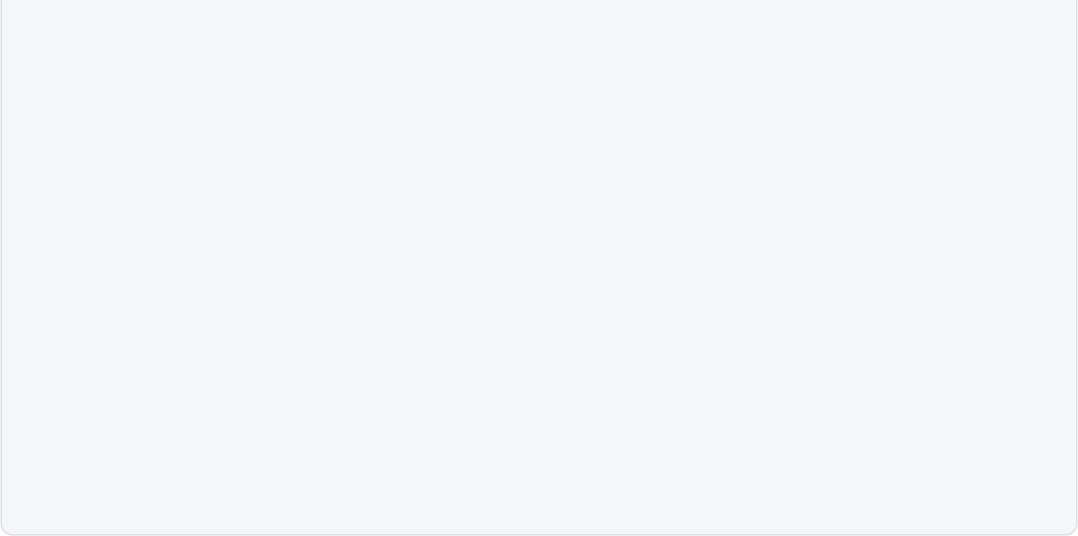
和

1 *// 定义错误边界* *ErrorBoundary extends React.Component {*

Vue

vue也有⼀个类似的⽣命周期来做这件事：

errorCaptured



2 state = { error: null }

3 static getDerivedStateFromError(error) {  { error }

4 }

5 componentDidCatch(error, errorInfo) {*// 调⽤我们实现的SDK实例*

6 insSDK.error(error, errorInfo)

7 }

8 render() { ( .state.error) {  <h2Something went wrong.</h2

9 }

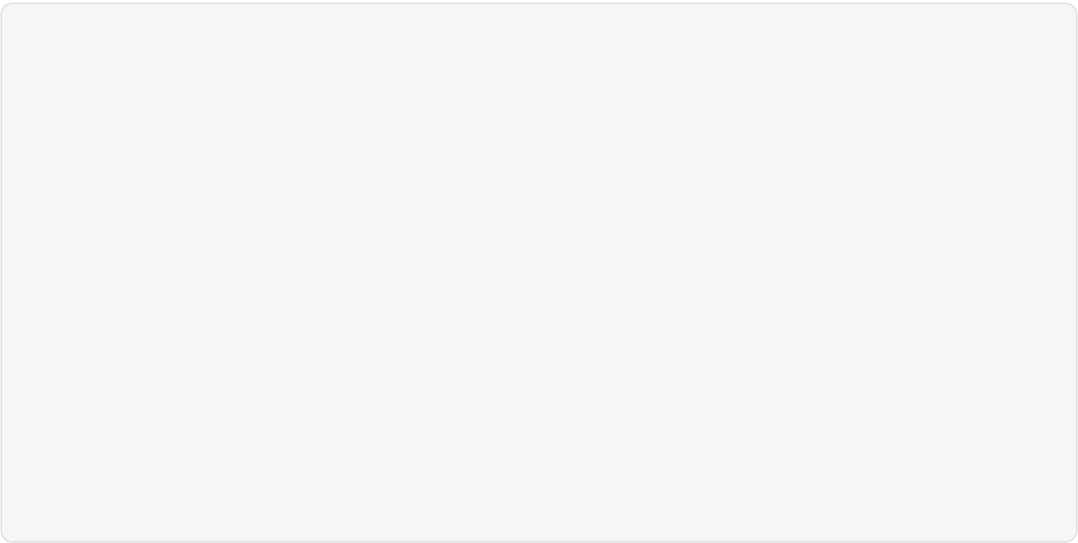
10 return this.props.children

11 } 12 } 13 ...

14 <ErrorBoundary

15 <BuggyCounter /> 16 </ErrorBoundary



1 Vue.component('ErrorBoundary', {

2 data: () => ({ error: null }),

3 errorCaptured (err, vm, info) { .error = `${err.stack}\n\nfound  ${info}

of component`*// 调⽤我们的SDK，上报错误信息*

4 insSDK.error(err,info)  false

5 },

6 render (h) { ( .error) {  h('pre', { style: { color: 'red' }},  .error)

7 }  .$slots.default[0]

8 }

9 })

10 ...

11 <error-boundary><buggy-counter />

12 </error-boundary

91. token 进⾏⾝份验证了解多少？【热度: 942】

token 概念和作⽤

Token是⼀种⽤于⾝份验证和授权的令牌。在Web应⽤程序中 ， 当⽤⼾进⾏登录或授权时 ，服务器会 ⽣成⼀个Token并将其发送给客⼾端。客⼾端在后续的请求中将Token作为⾝份凭证携带 ，以证明⾃⼰ 的⾝份。

Token可以是⼀个字符串 ，通常是经过加密和签名的 ，以确保其安全性和完整性。服务器收到Token

后 ，会对其进⾏解析和验证 ，以验证⽤⼾的⾝份并授权对特定资源的访问权限。 Token的使⽤具有以下特点：

• ⽆状态：服务器不需要在数据库中存储会话信息 ，所有必要的信息都包含在Token中。

• 可扩展性：Token可以存储更多的⽤⼾信息 ，甚⾄可以包含⾃定义的数据。

• 安全性：Token可以使⽤加密算法进⾏签名 ，以确保数据的完整性和安全性。

• 跨域⽀持：Token可以在跨域请求中通过在请求头中添加Authorization字段进⾏传递。

Token在前后端分离的架构中⼴泛应⽤ ，特别是在RESTful API的⾝份验证中常⻅ 。它⽐传统的基于 Cookie的会话管理更灵活 ，并且适⽤于各种不同的客⼾端 ，例如Web、移动应⽤和第三⽅接⼊等。

token ⼀般在客⼾端存在哪⼉ Token⼀般在客⼾端存在以下⼏个地⽅：

• Cookie ：Token可以存储在客⼾端的Cookie中。服务器在响应请求时 ，可以将Token作为⼀个 Cookie发送给客⼾端 ，客⼾端在后续的请求中会⾃动将Token包含在请求的Cookie中发送给服务 器。

• Local Storage/Session Storage：Token也可以存储在客⼾端的Local Storage或Session Storage 中。这些是HTML5提供的客⼾端存储机制 ，可以在浏览器中⻓期保存数据。

• Web Storage API：除了Local Storage和Session Storage ，Token也可以使⽤Web Storage API中 的其他存储机制 ，⽐如IndexedDB、WebSQL等。

• 请求头：Token也可以包含在客⼾端发送的请求头中 ，⼀般是在Authorization头中携带Token。 需要注意的是 ，⽆论将Token存储在哪个地⽅ ，都需要采取相应的安全措施 ，如HTTPS传输、加密存 储等 ，以保护Token的安全性。

存放在 cookie 就安全了吗？

存放在Cookie中相对来说是⽐较常⻅的做法 ，但是并不是最安全的⽅式。存放在Cookie中的Token可 能存在以下安全⻛险：

• 跨站脚本攻击（XSS） ：如果⽹站存在XSS漏洞 ，攻击者可以通过注⼊恶意脚本来获取⽤⼾的 Cookie信息 ，包括Token。攻击者可以利⽤Token冒充⽤⼾进⾏恶意操作。

• 跨站请求伪造（CSRF） ：攻击者可以利⽤CSRF漏洞 ，诱使⽤⼾在已经登录的情况下访问恶意⽹ 站 ，该⽹站可能利⽤⽤⼾的Token发起伪造的请求 ，从⽽执⾏未经授权的操作。

• 不可控的访问权限：将Token存放在Cookie中 ，意味着浏览器在每次请求中都会⾃动携带该

Token。如果⽤⼾在使⽤公共计算机或共享设备时忘记退出登录 ，那么其他⼈可以通过使⽤同⼀个 浏览器来访问⽤⼾的账⼾ 。

为了增加Token的安全性 ，可以采取以下措施：

• 使⽤HttpOnly标识：将Cookie设置为HttpOnly ，可以防⽌XSS攻击者通过脚本访问Cookie。

• 使⽤Secure标识：将Cookie设置为Secure ，只能在通过HTTPS协议传输时发送给服务器 ，避免明 ⽂传输。

• 设置Token的过期时间：可以设置Token的过期时间 ，使得Token在⼀定时间后失效 ，减少被滥⽤ 的⻛险。

• 使⽤其他存储⽅式：考虑将Token存储在其他地⽅ ，如Local Storage或Session Storage ，并采取 加密等额外的安全措施保护Token的安全性。

cookie 和 token 的关系

Cookie和Token是两种不同的概念 ，但它们在⾝份验证和授权⽅⾯可以有关联。

Cookie是服务器在HTTP响应中通过Set-Cookie标头发送给客⼾端的⼀⼩段数据。客⼾端浏览器将

Cookie保存在本地 ，然后在每次对该服务器的后续请求中将Cookie作为HTTP请求的⼀部分发送回服 务器。Cookie通常⽤于在客⼾端和服务器之间维护会话状态 ，以及存储⽤⼾相关的信息。

Token是⼀种⽤于⾝份验证和授权的令牌。 它是⼀个包含⽤⼾⾝份信息的字符串 ，通常是服务器⽣成并 返回给客⼾端。客⼾端在后续的请求中将Token作为⾝份凭证发送给服务器 ，服务器通过验证Token的 有效性来确认⽤⼾的⾝份和权限。

Cookie和Token可以结合使⽤来实现⾝份验证和授权机制。服务器可以将Token存储在Cookie中 ，然 后发送给客⼾端保存。客⼾端在后续的请求中将Token作为Cookie发送给服务器。服务器通过验证

Token的有效性来判断⽤⼾的⾝份和权限。这种⽅式称为基于Cookie的⾝份验证。另外 ，也可以将 Token直接存储在请求的标头中 ，⽽不是在Cookie中进⾏传输 ，这种⽅式称为基于Token的⾝份验 证。

需要注意的是 ，Token相对于Cookie来说更加灵活和安全 ，可以实现跨域⾝份验证 ，以及客⼾端和服 务器的完全分离。⽽Cookie则受到⼀些限制 ，如跨域访问限制 ，以及容易受到XSS和CSRF攻击等。 因 此 ，在实现⾝份验证和授权机制时 ，可以选择使⽤Token替代或辅助Cookie。

92. 在前端应⽤如何进⾏权限设计？ 【热度: 329】

在前端应⽤的权限设计中 ，以下是⼀些建议：

⻆⾊与权限分离

将⽤⼾的权限分为不同的⻆⾊ ，每个⻆⾊拥有特定的权限。

这样可以简化权限管理 ，并且当需求变化时 ，只需要调整⻆⾊的权限 ，⽽不需要逐个修改⽤⼾的权 限。

在⻆⾊与权限分离的设计中 ，可以按照以下⼏个步骤进⾏

1. 确定权限集合：⾸先 ，需要确定系统中所有的权限 ，包括操作、功能、资源等。可以根据系统需 求、业务流程等确定权限的粒度和层次结构。

2. 确定⻆⾊集合：根据系统的⻆⾊需求 ，确定不同的⻆⾊ ，例如管理员、普通⽤⼾ 、编辑等。每个⻆ ⾊代表⼀组权限的集合 ，可以根据业务需求进⾏划分。

3. 分配权限给⻆⾊：将权限与⻆⾊进⾏关联 ，确定每个⻆⾊具备哪些权限。可以通过⻆⾊-权限的映射 表或者通过⻆⾊组的⽅式进⾏管理。

4. ⽤⼾与⻆⾊关联：将⽤⼾与⻆⾊进⾏关联 ，确定每个⽤⼾属于哪些⻆⾊。 可以通过⽤⼾-⻆⾊的映射 表或者通过⽤⼾组的⽅式进⾏管理。

5. 权限验证：在系统中 ，根据⽤⼾的⻆⾊和权限配置进⾏权限验证。在⽤⼾进⾏操作或访问受限资源 时 ，根据⽤⼾的⻆⾊与权限进⾏验证 ，决定是否允许执⾏相应的操作。

功能级权限控制

对于敏感操作或者需要权限控制的功能 ，需要在前端实现功能级的权限控制。

通过在代码中判断⽤⼾是否拥有执⾏该功能的权限 ，来决定是否展⽰或者禁⽤相关功能。

功能级权限控制是指在系统中对⽤⼾进⾏细粒度的权限控制 ，即控制⽤⼾是否能够执⾏某个具体的功 能或操作。

以下是功能级权限控制的设计步骤：

1. 确定功能点：⾸先 ，需要明确系统中的各个功能点 ，例如新增、编辑、删除、查询等。将系统中的 所有功能进⾏明确定义和分类。

2. 定义权限：对于每个功能点 ，定义相应的权限。权限可以使⽤权限名或者权限码进⾏标识 ，例如新 增权限可以使⽤ "add"或者权限码"001"进⾏标识。

3. ⻆⾊与权限关联：将权限与⻆⾊进⾏关联。确定每个⻆⾊具备哪些权限。可以使⽤⻆⾊-权限的映射 表进⾏管理。

4. ⽤⼾与⻆⾊关联：将⽤⼾与⻆⾊进⾏关联。确定每个⽤⼾属于哪些⻆⾊。 可以使⽤⽤⼾-⻆⾊的映射 表进⾏管理。

5. 权限验证：在系统中 ，对⽤⼾进⾏权限验证。 当⽤⼾进⾏某个功能操作时 ，根据⽤⼾的⻆⾊与权限 进⾏验证 ，决定是否允许执⾏该操作。

6. 权限控制界⾯： 提供⼀个权限控制界⾯ ，⽤于管理⻆⾊与权限的关联。管理员可以通过该界⾯对⻆ ⾊的权限进⾏配置和管理。

7. 动态权限控制：可以考虑将权限控制设计成动态的。即在系统运⾏时 ，可以根据⽤⼾⻆⾊的配置动 态控制⽤⼾是否具备某个功能的权限。 这样可以灵活地根据业务需求进⾏权限的调整。

路由级权限控制

对于不同的⻚⾯或路由 ，可以根据⽤⼾的⻆⾊或权限来进⾏权限控制。在前端路由中配置权限信息， 当⽤⼾访问特定路由时 ，前端会检查⽤⼾是否具备访问该路由的权限。

前端路由级权限控制是指在前端⻚⾯中根据⽤⼾的权限配置 ，控制⽤⼾是否可以访问某个路由或者⻚ ⾯。

以下是前端路由级权限控制的设计⽅案：

1. 定义路由表：⾸先 ，需要定义系统中的所有路由和对应的⻚⾯组件。将路由按照功能模块进⾏分 类 ，⽅便后续的权限管理。

2. 定义权限配置：对于每个路由或者⻚⾯ ，定义相应的权限配置。可以使⽤权限名或者权限码进⾏标 识 ，例如"add"、 "edit"等。可以将权限配置与路由表⼀起存放在⼀个配置⽂件中 ，或者存放在后 端数据库中。

3. 获取⽤⼾权限：在登录成功后 ，从后端获取当前⽤⼾的权限信息。可以将⽤⼾的权限信息存放在前 端的状态管理库（如Vuex或Red ux） 中 ，以便在全局范围内进⾏访问。

4. 路由守卫：使⽤前端路由守卫机制 ，在路由跳转前进⾏权限验证。在路由守卫中 ，根据当前⽤⼾的 权限信息和路由配置进⾏判断 ，决定是否允许⽤⼾访问该路由。如果⽤⼾没有相应的权限 ，可以进 ⾏跳转到⽆权限提⽰⻚⾯或者其他处理⽅式。

5. 权限控制组件：可以创建⼀个权限控制组件 ，在需要进⾏权限控制的路由组件上使⽤该组件进⾏包 裹。该组件可以根据当前⽤⼾的权限和路由配置 ，动态显⽰或隐藏路由组件。

6. 动态路由：对于⼀些有权限控制的路由 ，可以在⽤⼾登录时根据权限配置动态⽣成。根据⽤⼾的权 限配置 ，过滤路由表 ，⽣成⽤⼾可以访问的路由列表 ，并将该列表添加到路由配置中。

动态权限管理

在前端应⽤中 ，可以实现动态权限管理 ，即在⽤⼾登录时从服务器获取⽤⼾的权限信息 ，并在前端进 ⾏缓存。这样可以保证⽤⼾权限的实时性 ，同时也便于后端对权限进⾏调整和管理。

UI级的权限控制

对于某些敏感信息或操作 ，可以通过前端的界⾯设计来进⾏权限控制。例如 ，隐藏某些敏感字段或操 作按钮 ，只对具有相应权限的⽤⼾可⻅或可操作。

异常处理与安全验证

在前端应⽤中 ，需要实现异常处理机制 ， 当⽤⼾越权操作时 ，需要给予相应提⽰并记录⽇志。 同时， 对于敏感操作 ，需要进⾏⼆次验证 ，例如通过输⼊密码或短信验证码等⽅式进⾏安全验证。

安全性考虑

在设计前端应⽤的权限时 ，需要考虑安全性 ，例如防⽌跨站脚本攻击（XSS） 、跨站请求伪造

（CSRF）等攻击⽅式。可以采⽤合适的安全措施 ，如输⼊验证、加密传输等来保护应⽤的安全性。

综上所述 ，前端应⽤的权限设计应该考虑⻆⾊与权限分离、功能级与路由级的权限控制、动态权限管 理、 UI级的权限控制、异常处理与安全验证以及安全性考虑等⽅⾯ 。通过合理的权限设计 ，可以确保 系统的安全性和⽤⼾权限的灵活管理。

93. [低代码] 代码平台⼀般渲染是如何设计的？ 【热度: 399】

渲染设计

渲染核⼼本质就是： [schema] + [组件] = [⻚⾯] 整体架构如下

• 协议层：基于《低代码引擎搭建协议规范》 产出的 Schema 作为我们的规范协议。

• 能⼒层：提供组件、 区块、⻚⾯等渲染所需的核⼼能⼒ ，包括 Props解析、样式注⼊ 、条件渲染 等。

• 适配层： 由于我们使⽤的运⾏时框架不是统⼀的 ，所以统⼀使⽤适配层将不同运⾏框架的差异部 分 ，通过接⼝对外 ，让渲染层注册/适配对应所需的⽅法。能保障渲染层和能⼒层直接通过适配层 连接起来 ，能起到独⽴可扩展的作⽤ 。

• 渲染层：提供核⼼的渲染⽅法 ， 由于不同运⾏时框架提供的渲染⽅法是不同的 ，所以其通过适配层 进⾏注⼊ ，只需要提供适配层所需的接⼝ ，即可实现渲染。

• 应⽤层：根据渲染层所提供的⽅法 ，可以应⽤到项⽬中 ，根据使⽤的⽅法和规模即可实现应⽤ 、⻚ ⾯ 、区块的渲染。

设计模式渲染（Simulator）

设计模式渲染就是将编排⽣成的《搭建协议》渲染成视图的过程 ，视图是可以交互的 ，所以必须要处 理好内部数据流、⽣命周期、事件绑定、 国际化等等。

也称为画布的渲染 ，画布是 UI编排的核⼼ ，它⼀般融合了⻚⾯的渲染以及组件/区块的拖拽、选择、快 捷配置。

画布的渲染和预览模式的渲染的区别在于 ，画布的渲染和设计器之间是有交互的。 所以在这⾥我们新增了⼀层 Simulator 作为设计器和渲染的连接器。

Simulator 是将设计器传⼊的 DocumentModel 和组件/库描述转成相应的 Schema 和 组件类。再调 ⽤ Render 层完成渲染。我们这⾥介绍⼀下它提供的能⼒。

• Project：位于顶层的 Project ，保留了对所有⽂档模型的引⽤ ，⽤于管理应⽤级 Schema 的导⼊与 导出。

• Document：⽂档模型包括 Simulator 与数据模型两部分。Simulator 通过⼀份 Simulator Host 协 议与数据模型层通信 ，达到画布上的 UI操作驱动数据模型变化。通过多⽂档的设计及多 Tab 交互 ⽅式 ，能够实现同时设计多个⻚⾯ ，以及在⼀个浏览器标签⾥进⾏搭建与配置应⽤属性。

• Simulator ：模拟器主要承载特定运⾏时环境的⻚⾯渲染及与模型层的通信。

• Node ：节点模型是对可视化组件/区块的抽象 ，保留了组件属性集合 Props 的引⽤ ，封装了⼀系列 针对组件的API ，⽐如修改、编辑、保存、拖拽、复制等。

• Props：描述了当前组件所维系的所有可以「设计」 的属性 ，提供⼀系列操作、遍历和修改属性的 ⽅法。 同时保持对单个属性 Prop 的引⽤ 。

• Prop：属性模型 Prop 与当前可视化组件/区块的某⼀具体属性想映射 ，提供了⼀系列操作属性变更 的 API。

• Settings：SettingField 的集合。

• SettingField： 它连接属性设置器 Setter 与属性模型 Prop ，它是实现多节点属性批处理的关键。

• 通⽤交互模型： 内置了拖拽、活跃追踪、悬停探测、剪贴板、滚动、快捷键绑定。 模拟器

• 运⾏时环境：从运⾏时环境来看 ， ⽬前我们有 React ⽣态、 Rax ⽣态。⽽在对外的历程中 ，我们也 会拥有 Vue ⽣态、Angular ⽣态等。

• 布局模式：不同于 C 端营销⻚的搭建 ，中后台场景⼤多是表单、表格 ，流式布局是主流的选择。对 于设计师、产品来说 ，是需要绝对布局的⽅式来进⾏⻚⾯研发的。

• 研发场景：从研发场景来看 ，低代码搭建不仅有⻚⾯编排 ，还有诸如逻辑编排、业务编排的场景。

94. [低代码] 代码平台⼀般底层协议是怎么设计的 【热度: 263】

低代码引擎体系基于三份协议来构建:

• [《低代码引擎搭建协议规范》](https://gitee.com/link?target=https%3A%2F%2Flowcode-engine.cn%2Fsite%2Fdocs%2Fspecs%2Flowcode-spec)

• [《低代码引擎物料协议规范》](https://gitee.com/link?target=https%3A%2F%2Flowcode-engine.cn%2Fsite%2Fdocs%2Fspecs%2Fmaterial-spec)

• [《低代码引擎资产包协议规范》](https://gitee.com/link?target=https%3A%2F%2Flowcode-engine.cn%2Fsite%2Fdocs%2Fspecs%2Fassets-spec)

95. [Webpack] 有哪些优化项⽬的⼿段？ 【热度: 1,163】

做性能优化 ，分为两个⽅⾯：

围绕

webpack

构建时间优化 、

构建体积优化

构建时间优化

• [缩⼩范围](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E7%BC%A9%E5%B0%8F%E8%8C%83%E5%9B%B4) • [⽂件后缀](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E6%96%87%E4%BB%B6%E5%90%8E%E7%BC%80) • [别名](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E5%88%AB%E5%90%8D)

• [缓存](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E7%BC%93%E5%AD%98)

• [并⾏构建](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E5%B9%B6%E8%A1%8C%E6%9E%84%E5%BB%BA)

• [定向查找第三⽅模块](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E5%AE%9A%E5%90%91%E6%9F%A5%E6%89%BE%E7%AC%AC%E4%B8%89%E6%96%B9%E6%A8%A1%E5%9D%97)

• [构建结果优化](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E6%9E%84%E5%BB%BA%E7%BB%93%E6%9E%9C%E4%BC%98%E5%8C%96)

◦ [压缩 js](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E5%8E%8B%E7%BC%A9-js)

◦ [压缩 css](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E5%8E%8B%E7%BC%A9-css)

◦ [压缩 html](https://gitee.com/yanleweb/interview-question/issues/I8FASH#%E5%8E%8B%E7%BC%A9-html)

◦ 压缩图⽚

◦ 按需加载

◦ prload、 prefetch

◦ 代码分割

◦ tree shaking

◦ gzip

◦ 作⽤域提升

缩⼩范围

我们在使⽤ loader 时 ，可以配置

exclude 缩⼩ loader 对⽂件的搜索范围 ，以此来提

include 、

⾼构建速率。 像

/node\_moudles ⽬录下的体积辣么⼤ ，⼜是第三⽅包的存储⽬录 ，直接

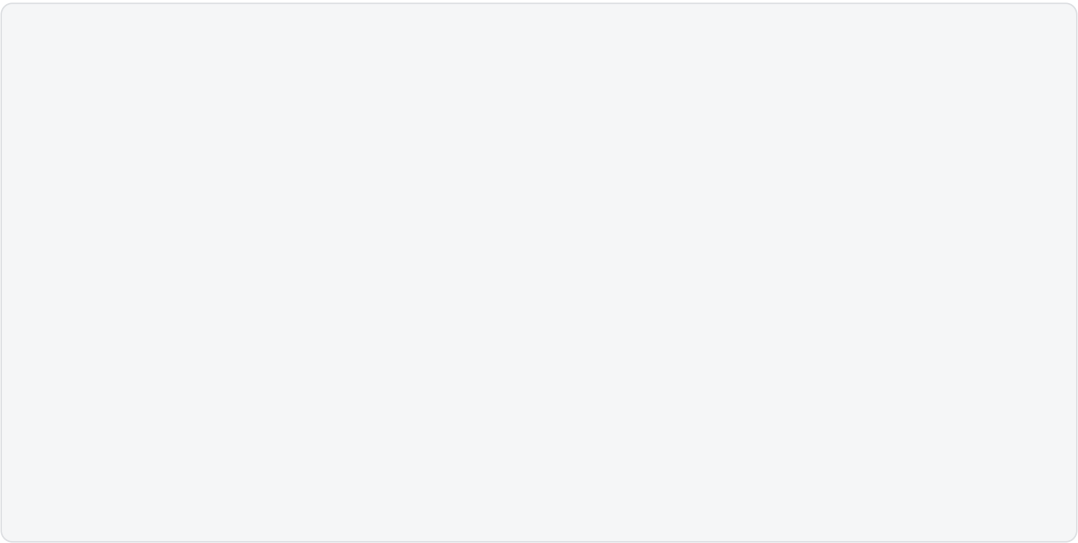
exclude 掉可以节

省⼀定的时间的。 当然

exclude 和

include 可以⼀起配置 ，⼤部分情况下都是只需要使⽤ loader 编译 src ⽬录下

的代码

1 module.exports = {module: {

2 rules: [

3 {

4 test: /\.(|ts|tsx|js|jsx)$/,*// 只解析* *src ⽂件夹下的* *ts、tsx、js、*

*jsx ⽂件// include 可以是数组，表⽰多个⽂件夹下的模块都要解析*

5 include: path.resolve(\_\_dirname, '../src'),

6 use: [ 'thread-loader', 'babel-loader'],

7 *//当然也可以配置* *exclude，表⽰* *loader 解析时不会编译这部分⽂件//同样*

*exclude 也可以是数组*

8 exclude: /node\_modules/,

9 }

10 ]

11 }

12 }

还需注意⼀个点就是要确保 loader 的 准确性 ，⽐如不要使⽤ less-loader 去解析 css ⽂件

⽂件后缀

resolve .extensions 是我们常⽤的⼀个配置 ，他可以在导⼊语句没有带⽂件后缀时 ，可以按照 配置的列表 ， ⾃动补上后缀。我们应该根据我们项⽬中⽂件的实际使⽤情况设置后缀列表 ，将使⽤频 率⾼的放在前⾯ 、同时后缀列表也要尽可能的少 ，减少没有必要的匹配。 同时 ，我们在源码中写导⼊ 语句的时候 ，尽量带上后缀 ，避免查找匹配浪费时间。



1 module.export = {

2 resolve: {*//按照* *tsx、ts、jsx、js 的顺序匹配，若没匹配到则报错*

3 extensions: ['.tsx', '.ts', '.jsx', '.js'],

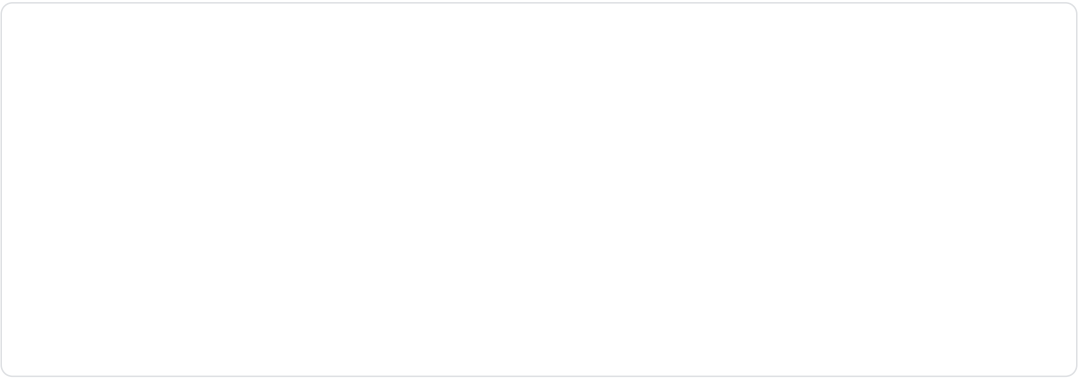
4 }

5 }

别名

通过配置

resolve .alias 别名的⽅式 ，减少引⽤⽂件的路径复杂度





1 module.exports = {

2 resolve: {

3 alias: {*//把* *src ⽂件夹别名为* *@//引⼊* *src 下的⽂件就可以* *import xxx from*

*'@/xxx''@': path.join( dirname, '../src')*

4 }

5 }

6 }

7

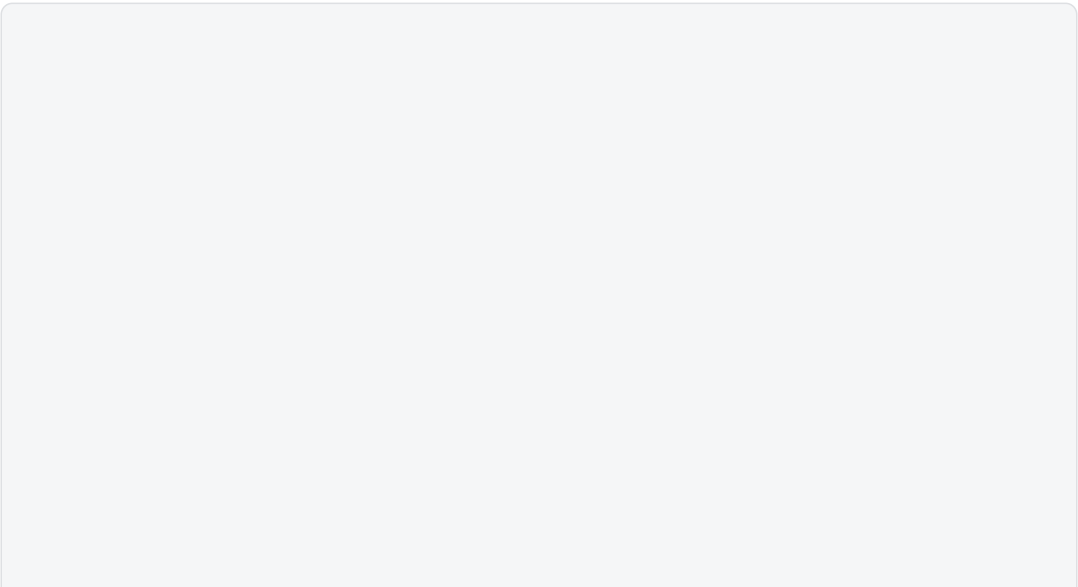
8 *// 引⼊* *src 下的某个模块时* *XXX*  *'@/xxx/xxx.tsx'*

缓存

在优化的⽅案中 ，缓存也是其中重要的⼀环。在构建过程中 ，开启缓存提升⼆次打包速度。

在项⽬中，js ⽂件是占⼤头的 ， 当项⽬越来越⼤时 ，如果每次都需要去编译 JS 代码 ，那么构建的速度 肯定会很慢的 ，所以我们可以配置 babel-loader 的缓存配置项 cacheDirectory 来缓存没有

变过的 js 代码

module.exports = {module: {

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

rules: [

{

test: /.jsx?$/,

use: [

{

loader: 'babel-loader' options: {

,

cacheDirectory: true

,

},

}

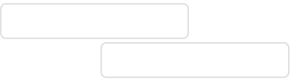
]

}

]

}

}

上⾯的缓存优化只是针对像 babel-loader 这样可以配置缓存的 loader ，那没有缓存配置的 loader 该怎么使⽤缓存呢 ，此时需要 cache-loader

1 module.exports = {module: {

2 rules: [

3 {

4 test: /.jsx?$/,

5 use: ['cache-loader', "babel-loader"

6 ],

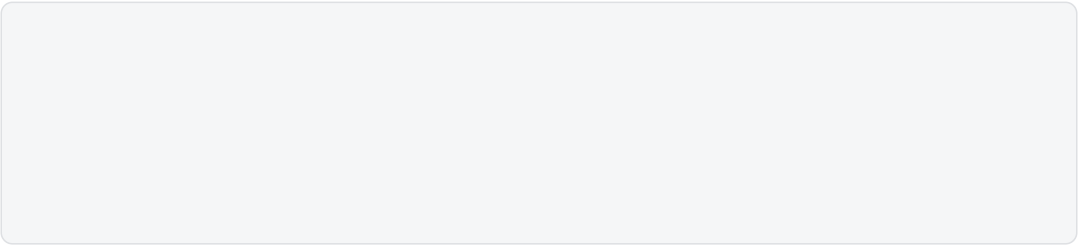
7 }

8 ]

9 }

10 }

编译后同样多⼀个 /node\_modules/.cache/cache-loader 缓存⽬录 当然还有⼀种⽅式 ， webpack5 直接提供了 cache 配置项 ，开启后即可缓存

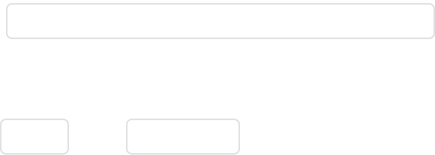
1 module.exports = {

2 cache: {

3 type: 'filesystem'

4 }

5 }

 /node\_modules/.cache/webpack

编译后会多出

缓存⽬录

并⾏构建

是单线程的 ，所以⼀次性只能⼲⼀件事 ，那如果利⽤电脑的多核

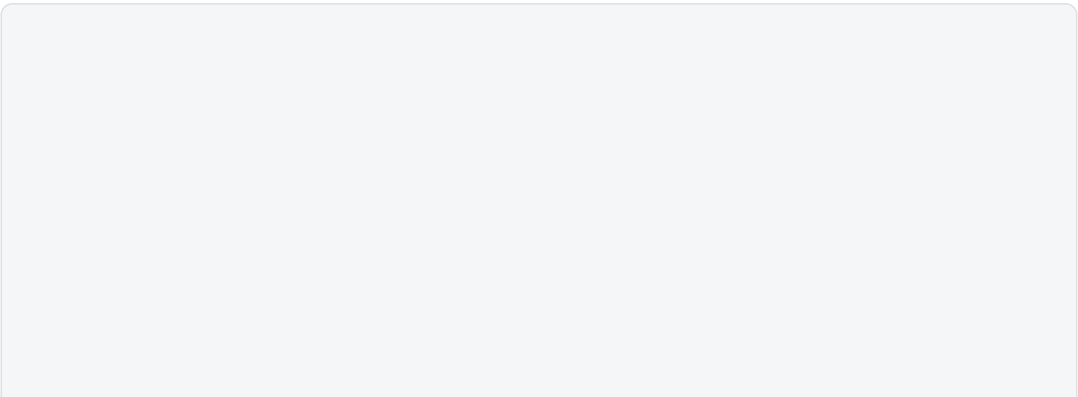
Node

⾸先 ，运⾏在

⾥的

webpack

优势 ，也能提⾼构建速度？ thread-loader可以开启多进程打包

1 module.exports = {module: {

2 rules: [

3 {

4 test: /.jsx?$/,

5 use: [*// 开启多进程打包。*

6 {

7 loader: 'thread-loader',

8 options: {

9 workers: 3 *// 开启* *3个* *进程*

10 }

11 },

12 {

13 loader: 'babel-loader',

14 }

15 ]

16 }

17 ]

18 }

19 }

放置在这个 thread-loader 之后的 loader 就会在⼀个单独的 worker 池(worker pool) 中运⾏。 每个 worker 都是⼀个单独的有 600ms 限制的 node .js 进程。 同时跨进程的数据交换也会被限制。 所以建议仅在耗时的 loader 上使⽤ 。若项⽬⽂件不算多就不要使⽤ ，毕竟开启多个线程也会存在性能 开销。

定向查找第三⽅模块

webpack

配置⽤于指定

去哪些⽬录下寻找第三⽅模块。默认值是

resolve .modules

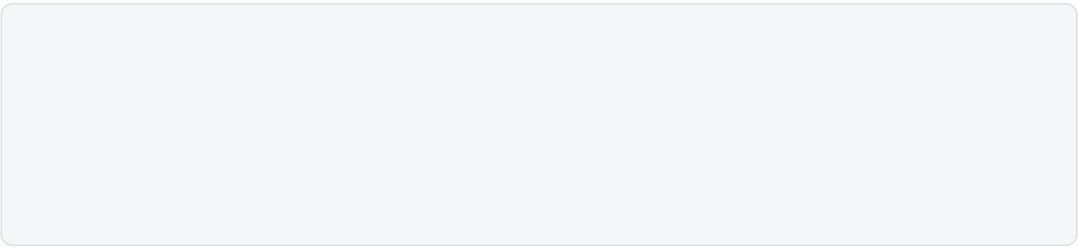
 [ 'node\_modules '] 。⽽在引⼊模块的时候 ，会以

node 核⼼模块 -----> node\_modules

的顺序查找模块。

------> node全局模块

我们通过配置 resolve.modules 指定 webpack 搜索第三⽅模块的范围 ，提⾼构建速率

module.export = {

1

2

3

4

5

resolve: {

modules: [path.resolve(\_\_dirname, 'node\_modules')]

}

}

构建结果优化

压缩 js

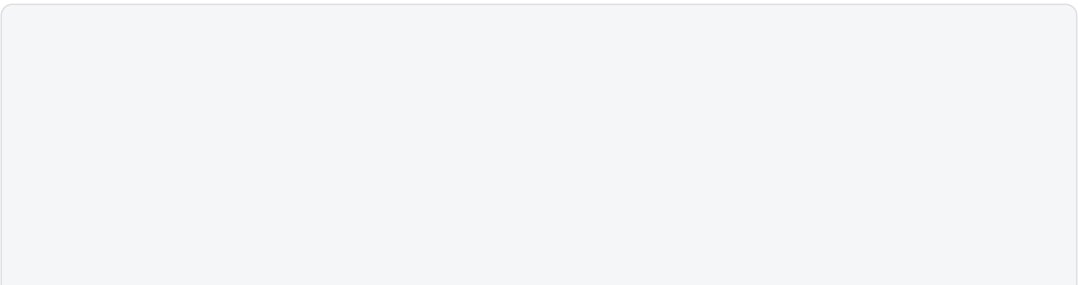
webpack5的话通过

mode : production

terser-webpack-plugin

来压缩 JS ，但在配置了

时 ，会默认开启

1  TerserPlugin = require('terser-webpack-plugin'); 2

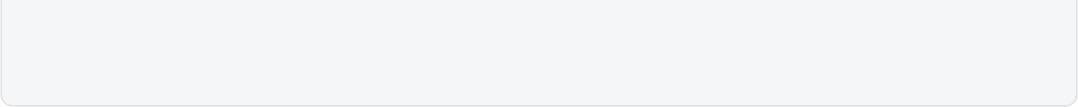
3 module.exports = {

4 optimization: {*// 开启压缩*

5 minimize: true,*// 压缩⼯具*

6 minimizer: [ TerserPlugin({}),

7 ],



8 },

9 }

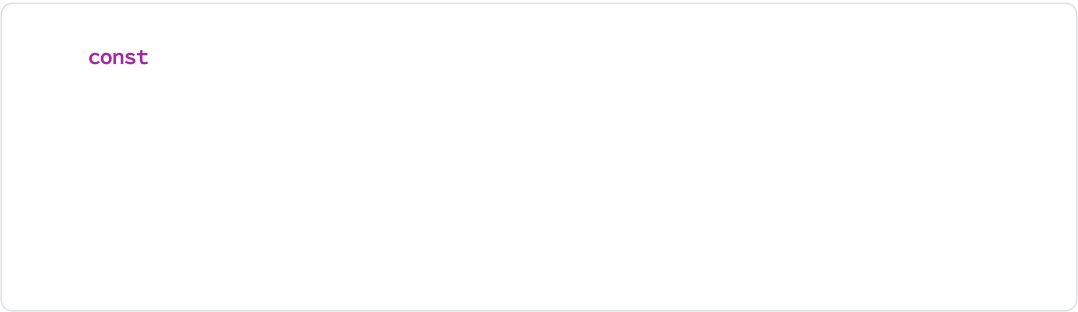
需要注意⼀个地⽅： ⽣产环境会默认配置 terser-webpack-plugin ，所以如果你还有其它压缩插 件使⽤的话需要将 Terser Plugin 显⽰配置或者使⽤ . . . ，否则



terser-webpack-plugin

会

被覆盖。

1 TerserPlugin = require("terser-webpack-plugin");

2 optimization: {

3 minimize: true,

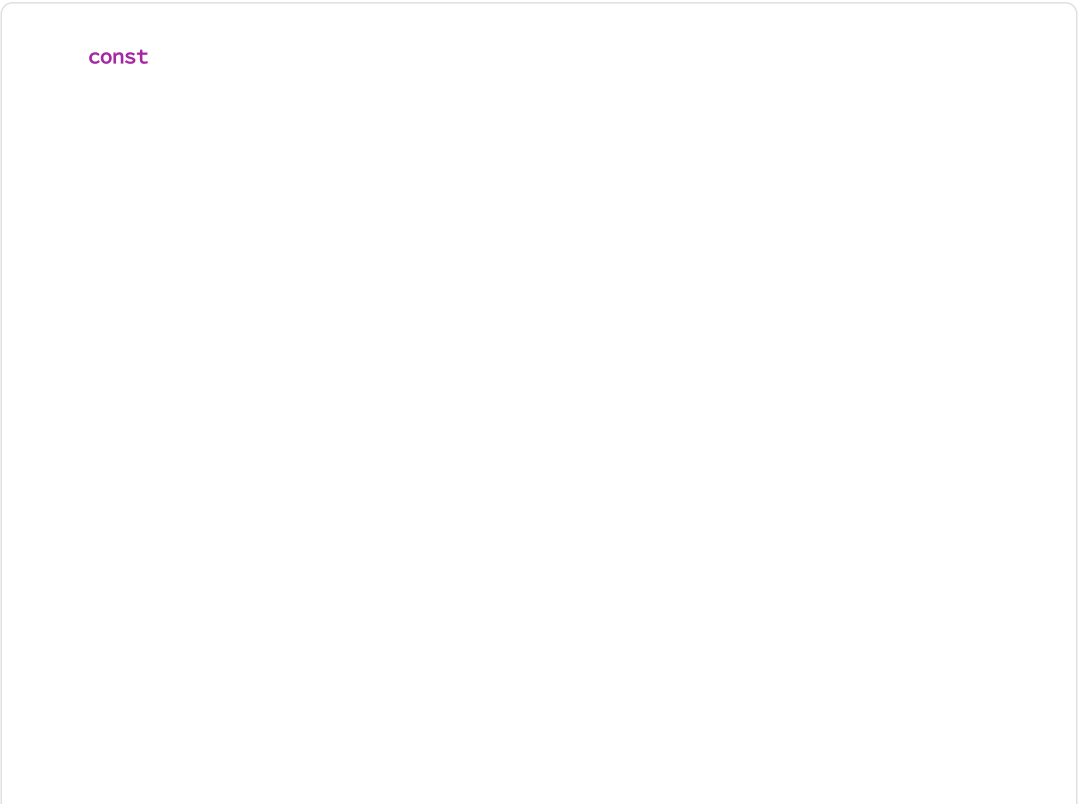
4 minimizer: [ TerserPlugin({}), *// 显⽰配置// "...", // 或者使⽤展开符，启⽤默* *认插件// 其它压缩插件* *CssMinimizerPlugin(),*

5 ],

6 },

压缩 css

压缩 css 我们使⽤ css-minimize r-webpack-plugin

同时 ，应该把 css 提取成单独的⽂件 ，使⽤ mini-css-extract-plugin

1 MiniCssExtractPlugin = require("mini-css-extract-plugin");

2  CssMinimizerPlugin = require("css-minimizer-webpack-plugin");

3

4 module.exports = {module: {

5 rules: [

6 {

7 test: /\.css$/,

8 use: [*//提取成单独的⽂件*

9 MiniCssExtractPlugin.loader,"css-loader" 10 ],

11 exclude: /node\_modules/,

12 },

13 ]

14 },

15 plugins: [ MiniCssExtractPlugin({*// 定义输出⽂件名和⽬录*

16 filename: "asset/css/main.css",

17 })

18 ],

19 optimization: {

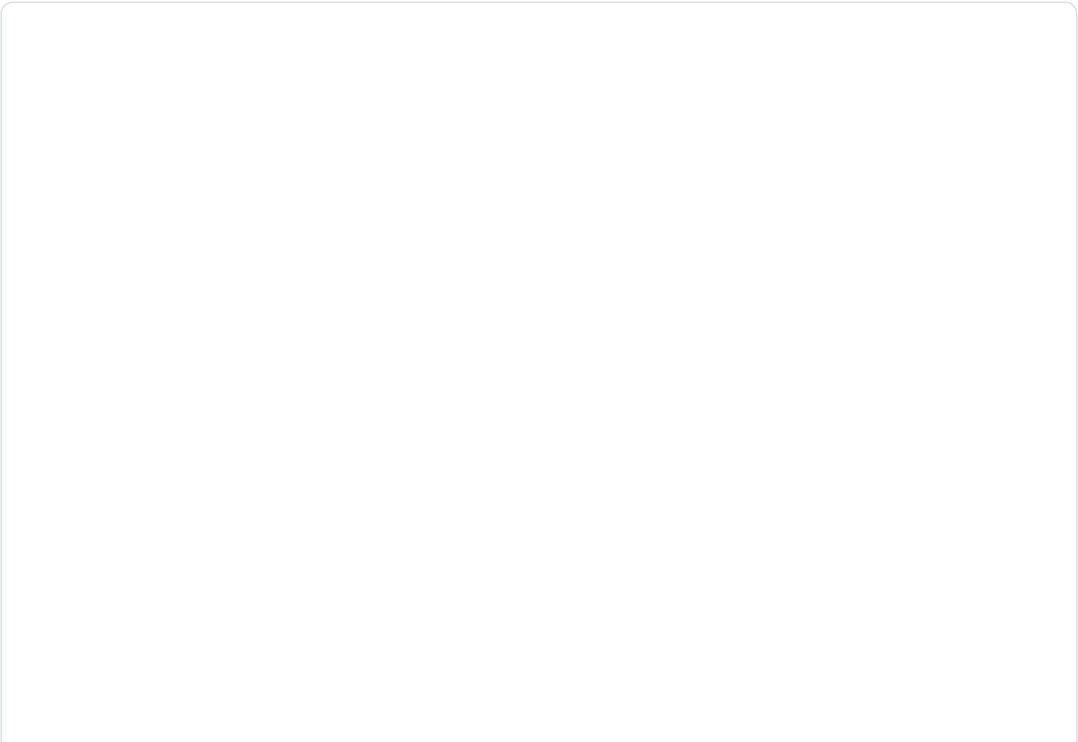
20 minimize: true,

21 minimizer: [*// 压缩* *css* *CssMinimizerPlugin({}),*

22 ],

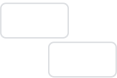
23 },

24 }

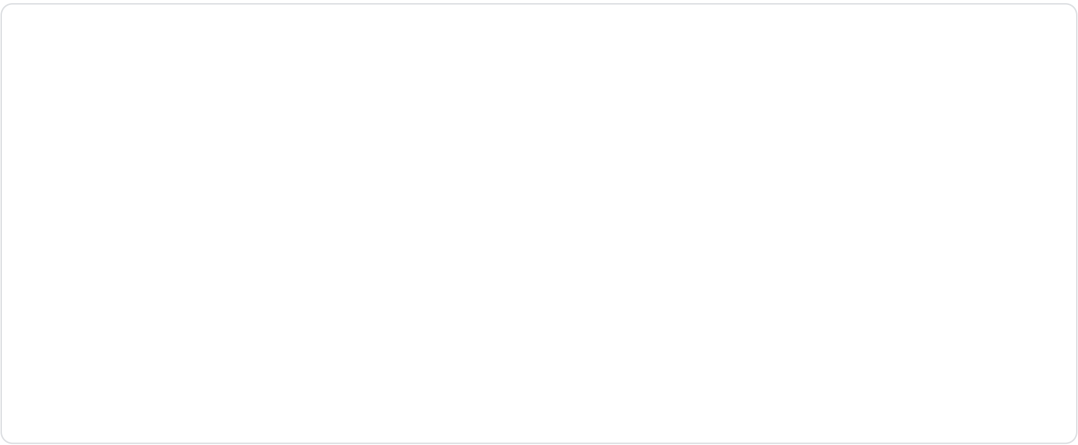
压缩 html



html-webpack-plugin

压缩 html 使⽤的还是 配置压缩 html 。

插件。该插件⽀持配置⼀个 minify 对象 ，⽤来

1 module.export = {

2 plugins: [ HtmlWebpackPlugin({*// 动态⽣成* *html ⽂件*

3 template: "./index.html",

4 minify: {*// 压缩HTML*

5 removeComments: true, *// 移除HTML中的注释*

6 collapseWhitespace: true, *// 删除空⽩符与换⾏符*

7 minifyCSS: true *// 压缩内联css* 8 },

9 })

10 ]

11 }

压缩图⽚

可以通过

image-webpack-loader

来实现

1 module.exports = {module: {

2 rules: [

3 {

4 test: /\.(png|jpg|gif|jpeg|webp|svg)$/,

5 use: ["file-loader",

6 {

7 loader: "image-webpack-loader",

8 options: {

9 mozjpeg: {

10 progressive: true,

11 },

12 optipng: {

13 enabled: false,

14 },

15 pngquant: {

16 quality: [0.65, 0.9],

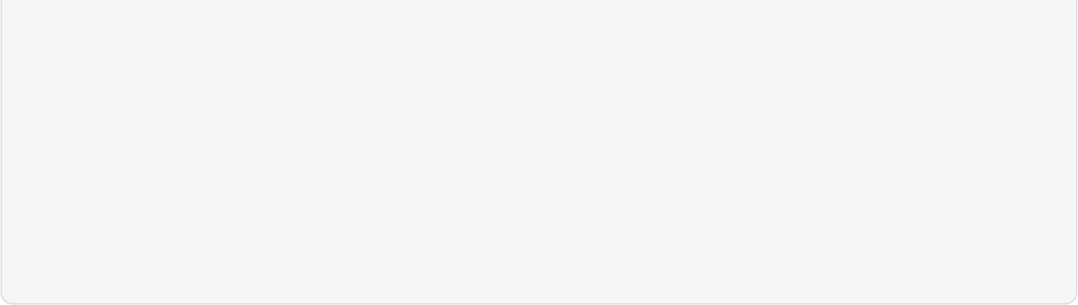
17 speed: 4,

18 },

19 gifsicle: {

20 interlaced: false,

21 },



22 },

23 },

24 ],

25 exclude: /node\_modules/, *//排除* *node\_modules ⽬录*

26 },

27 ]

28 },

29 }

按需加载

很多时候我们不需要⼀次性加载所有的

JS ⽂件 ，⽽应该在不同阶段去加载所需要的代码。

将路由⻚⾯/触发性功能单独打包为⼀个⽂件 ，使⽤时才加载 ，好处是

减轻⾸屏渲染的负担 。因为项

⽬功能越多其打包体积越⼤ ，导致⾸屏渲染速度越慢。

实际项⽬中⼤部分是对懒加载路由 ，⽽懒加载路由可以打包到⼀个 chunk ⾥⾯ 。⽐如某个列表⻚和编 辑⻚它们之间存在相互跳转 ，如果对它们拆分成两个

import() js 资源加载模块 ，在跳转过程中视

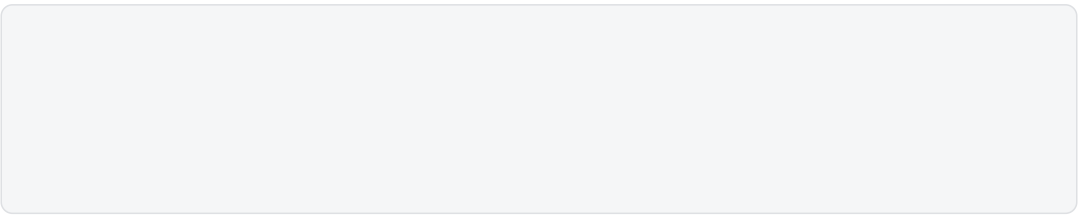
图会出现⽩屏切换过程。

因为在跳转期间 ，浏览器会动态创建 script 标签来加载这个

chunk ⽂件 ，在这期间 ，⻚⾯是没有任

何内容的。

所以⼀般会把路由懒加载打包到⼀个 chunk ⾥⾯

1  List = lazyComponent('list', () =>

(*/\* webpackChunkName: "list" \*/*

(*/\* webpackChunkName: "list" \*/*

'@/pages/list'));

2  Edit = lazyComponent('edit', () =>

'@/pages/edit'));

但需要注意⼀点：动态导⼊ import()⼀个模块 ，这个模块就不能再出现被其他模块使⽤ 同步

import ⽅式导⼊。

⽐如 ，⼀个路由模块在注册 <Route /> 时采⽤动态 import()导⼊ ，但在这个模块对外暴露了⼀些 变量⽅法供其他⼦模块使⽤ ，在这些⼦模块中使⽤了同步 ESModule import ⽅式引⼊ ，这就造成了 动态 import() 的失效。

prload、 prefetch

对于某些较⼤的模块 ，如果点击时再加载 ，那可能响应的时间反⽽延⻓。我们可以使⽤ prefetch 、 preload 去加载这些模块

带

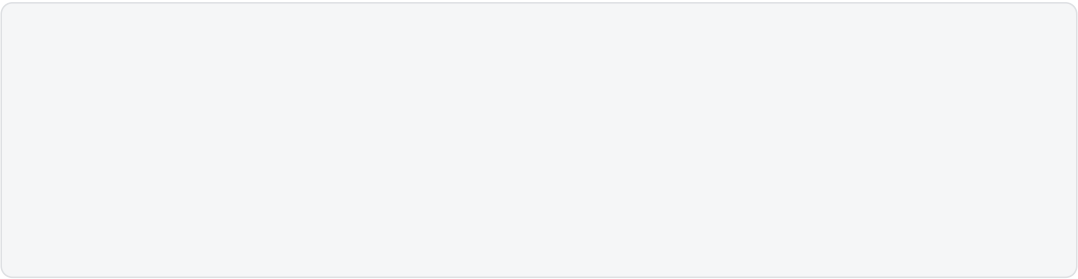
 prefetch ：将来可能需要⼀些模块资源（ ⼀般是其他⻚⾯的代码） ，在核⼼代码加载完成之后 宽空闲的时候再去加载需要⽤到的模块代码。

preload ： 当前核⼼代码加载期间可能需要模块资源（ 当前⻚⾯需要的但暂时还没使⽤到的），其

是和核⼼代码⽂件⼀起去加载的。 只需要通过

魔法注释 即可实现 ，以

prefetch 为例：



1 document.getElementById('btn1').onclick =  () {(

2 */\* webpackChunkName: "btnChunk"*

3 */*

4 *\* /\* webpackPrefetch: true\*/*'./module1.js'

5 ).then(fn => fn.default());

6 }

这⾏代码表⽰在浏览器空闲时加载 module1.js 模块 ，并且单独拆⼀个 chunk ，叫做 btnChunk

可以看到 ，在 head ⾥⾯ ，我们的懒加载模块被直接引⼊了 ，并且加上了 rel= 'prefetch ' 。 这样 ，⻚⾯⾸次加载的时候 ，浏览器空闲的会后会提前加载 module1 .js 。当我们点击按钮的时 候 ，会直接从缓存中读取该⽂件 ，因此速度⾮常快。

代码分割

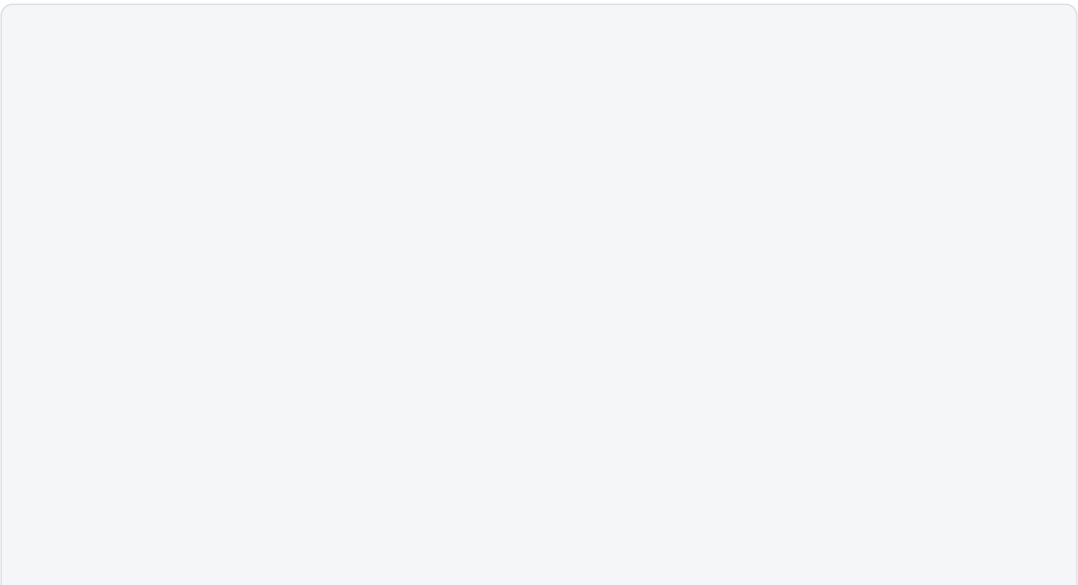
在项⽬中 ，⼀般是使⽤同⼀套技术栈和公共资源。如果每个⻚⾯的代码中都有这些公开资源 ，就会导 致资源的浪费。在每⼀个⻚⾯下都会加载重复的公共资源 ，⼀是会浪费⽤⼾的流量 ，⼆是不利于项⽬ 的性能 ，造成⻚⾯加载缓慢 ，影响⽤⼾体验。

⼀般是把不变的第三⽅库、⼀些公共模块（ ⽐如 util.js）这些单独拆成⼀个 chunk ，在访问⻚⾯的时 候 ，就可以⼀直使⽤浏览器缓存中的资源

webpack ⾥⾯通过

来分割代码

splitChunks

1 module.exports = {*//...*

2 optimization: {

3 splitChunks: {

4 chunks: 'async', *// 值有* *all，async 和* *initial*

5 minSize: 20000, *// ⽣成* *chunk 的最⼩体积（以* *bytes 为单位）。*

6 minRemainingSize: 0,

7 minChunks: 1, *//拆分前必须共享模块的最⼩* *chunks 数。*

8 maxAsyncRequests: 30, *//按需加载时的最⼤并⾏请求数。*

9 maxInitialRequests: 30, *// ⼊⼝点的最⼤并⾏请求数。*

10 enforceSizeThreshold: 50000,

11 cacheGroups: {

12 defaultVendors: {

13 test: /[\/]node\_modules[\/]/, *//第三⽅模块拆出来*

14 priority: -10,

15 reuseExistingChunk: true,

16 },

17 util.vendors: {

18 test: /[\/]utils[\/]/, *//公共模块拆出来*

19 minChunks: 2,

20 priority: -20,

21 reuseExistingChunk: true,

22 },

23 },

24 },

25 },

26 };

tree shaking

tree shaking 的原理细节可以看这篇⽂章：# webpack tree-shaking解析

tree shaking 在⽣产模式下已经默认开启了

只是需要注意下⾯⼏点：

ESM ⽣效

1. 只对

2. 只能是静态声明和引⽤的

ES6 模块 ，不能是动态引⼊和声明的。

3. 只能处理模块级别 ，不能处理函数级别的冗余。

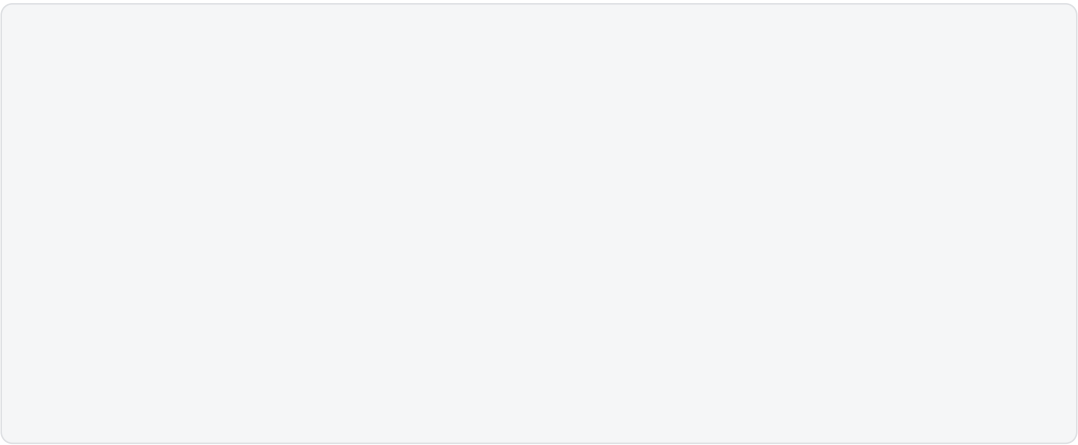
JS 相关冗余代码 ，不能处理

CSS 冗余代码。 ⽽可能样式⽂件⾥⾯有些代码我们也没有使⽤ ，我们可以通过

4. 只能处理

purgecss-webpack-plugin 插件

来对 css 进⾏ tree shaking

 path = require("path");

1

2

3

4

5

6

7

8

 PurgecssPlugin = require("purgecss-webpack-plugin");

 glob = require("glob"); *// ⽂件匹配模式module.exports = {//...*

plugins: [

...  PurgeCSSPlugin({

paths: glob.sync(`${PATH.src}*/\*\*/*\*`, { nodir: true }),

})

*// Add your plugins here// Learn more about plugins from*

*<https://webpack.js.org/configuration/plugins/>*

],

9

10

};

gzip

前端除了在打包的时候将⽆⽤的代码或者 console 、注释剔除之外。我们还可以使⽤ Gzip 对资 源进⾏进⼀步压缩。那么浏览器和服务端是如何通信来⽀持 Gzip 呢？

accept-encoding:gzip ，

request header 中设置

1. 当⽤⼾访问 web 站点的时候 ，会在

表明浏览器是否⽀持

Gzip 。

2. 服务器在收到请求后 ，判断如果需要返回

Gzip 压缩后的⽂件那么服务器就会先将我们的



response headers

JS\CSS 等其他资源⽂件进⾏

Gzip 压缩后再传输到客⼾端 ，同时将

设置

content-encoding:gzip 。反之 ，则返回源⽂件。

3. 浏览器在接收到服务器返回的⽂件后 ，判断服务端返回的内容是否为压缩过的内容 ，是的话则进⾏ 解压操作。



Gzip

⼀般情况下我们并不会让服务器实时

Gzip 压缩 ，⽽是利⽤

webpack 提前将静态资源进⾏

Gzip 资源放到服务器 ， 当请求需要的时候直接将

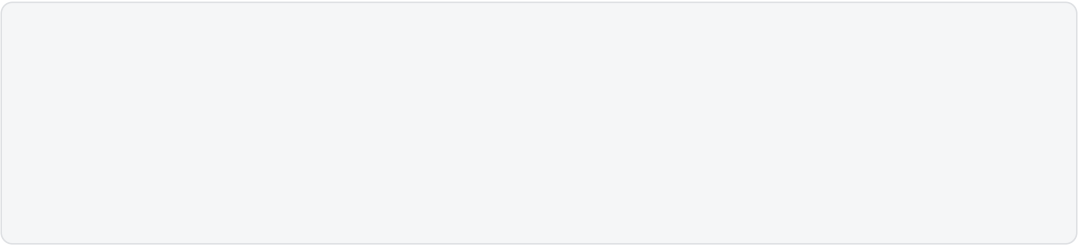
压缩 ，然后将

Gzip 资源发送给客⼾端。

compression-webpack-plugin 并在

我们只需要安装

plugins 配置就可以了



1  CompressionWebpackPlugin = require("compression-webpack-plugin"); *// 需要* *安装module.exports = {*

2 plugins: [ CompressionWebpackPlugin()

3 ]

4 }

作⽤域提升

Scope Hoisting 可以让 webpack 打包出来的代码⽂件体积更⼩ ，运⾏更快。

在开启 Scope Hoisting 后 ，构建后的代码会按照引⼊顺序放到⼀个函数作⽤域⾥ ，通过适当重命 名某些变量以防⽌变量名冲突 ，从⽽减少函数声明和内存花销。

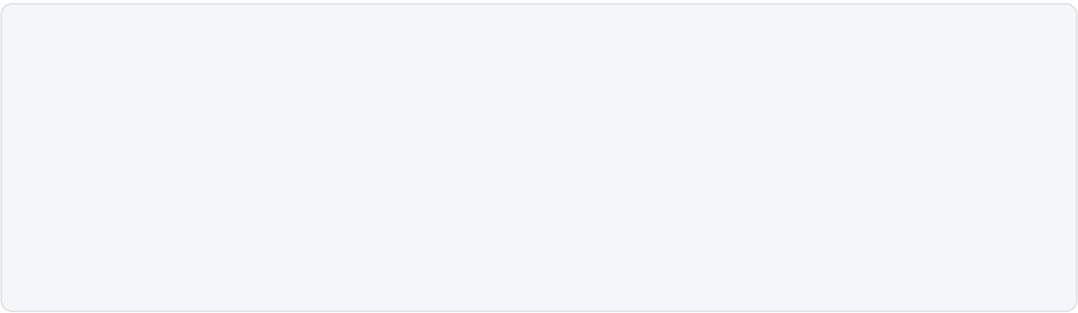
需要注意： Scope Hoisting 需要分析模块之间的依赖关系 ，所以源码必须采⽤ ES6 模块化语法 Scope Hoisting 是 webpack 内置功能 ，只需要在 plugins ⾥⾯使⽤即可 ，或者直接开启⽣产

环境也可以让作⽤域提升⽣效。

1 module.exports = {*//⽅式1*

2 mode: 'production', 3 *//⽅式2*

4 plugins: [*// 开启* *Scope Hoisting 功能*

*webpack.optimize.ModuleConcatenationPlugin()*

5 ]

6 }

96. IndexedDB 存储空间⼤⼩是如何约束的？ 【热度: 116】

IndexedDB 有⼤⼩限制。具体来说 ，IndexedDB 的⼤⼩限制通常由浏览器实现决定 ，因此不同浏览器 可能会有不同的限制。

⼀般来说 ，IndexedDB 的⼤⼩限制可以分为两个⽅⾯：

• 单个数据库的⼤⼩限制：每个 IndexedDB 数据库的⼤⼩通常会有限制 ，这个限制可以是固定的

（如某些浏览器限制为特定的⼤⼩ ，如 50MB） ，也可以是动态的（如某些浏览器根据设备剩余存 储空间来动态调整⼤⼩）。

• 整个浏览器的⼤⼩限制：除了每个数据库的⼤⼩限制外 ，浏览器还可能设置整个 IndexedDB 存储 的总⼤⼩限制。这个限制可以根据浏览器的策略和设备的可⽤存储空间来决定。

需要注意的是 ， 由于 IndexedDB 是在⽤⼾设备上进⾏存储的 ，并且浏览器对存储空间的管理可能会受 到⽤⼾权限和设备限制的影响 ，因此在使⽤ IndexedDB 存储⼤量数据时 ，需要注意数据的⼤⼩和存储 限制 ，以免超过浏览器的限制导致出错或⽆法正常存储数据。

追问：开发者是否可以通过JS代码可以调整 IndexedDB 存储空间⼤⼩？ 实际上 ，在创建数据库时 ，⽆法直接通过 API设置存储空间⼤⼩。

IndexedDB 的存储空间⼤⼩通常由浏览器的策略决定 ，并且在⼤多数情况下 ，开发者⽆法直接控制。 浏览器会根据⾃⾝的限制和规则 ，动态分配和管理 IndexedDB 的存储空间。 因此 ，将存储空间⼤⼩设 置为期望的值不是开发者可以直接控制的。

开发者可以通过以下⽅式来控制 IndexedDB 的存储空间使⽤情况：

1. 优化数据模型：设计合适的数据结构和索引 ，避免存储冗余数据和不必要的索引。

2. 删除不再需要的数据：定期清理不再需要的数据 ，以减少数据库的⼤⼩。

3. 压缩数据：对存储的数据进⾏压缩 ，可以减少存储空间的使⽤ 。

这些⽅法只能间接地影响 IndexedDB 的存储空间使⽤情况 ，具体的存储空间⼤⼩仍然由浏览器决定。

97. 浏览器的存储有哪些 【热度: 814】

在浏览器中 ，有以下⼏种常⻅的存储⽅式：

1. Cookie ：Cookie 是⼀种存储在⽤⼾浏览器中的⼩型⽂本⽂件。 它可以⽤于存储少量的数据 ，并在 浏览器与服务器之间进⾏传输。Cookie 可以设置过期时间 ，可以⽤于维持⽤⼾会话、记录⽤⼾偏 好等功能。

2. Web Storage：Web Storage 是 HTML5 提供的⼀种在浏览器中进⾏本地存储的机制。 它包括两种 存储⽅式：sessionStorage 和 localStorage。

◦ sessionStorage：sessionStorage ⽤于在⼀个会话期间（即在同⼀个浏览器窗⼝或标签⻚中） 存储数据。 当会话结束时 ，存储的数据会被清除。

◦ localStorage： localStorage ⽤于持久化地存储数据 ，即使关闭浏览器窗⼝或标签⻚ ，数据仍 然存在。 localStorage 中的数据需要⼿动删除或通过 JavaScript 代码清除。

3. IndexedDB： IndexedDB 是⼀种⽤于在浏览器中存储⼤量结构化数据的数据库。 它提供了⼀个异步 的 API ，可以进⾏增删改查等数据库操作。 IndexedDB 可以存储⼤量的数据 ，并⽀持事务操作。

4. Cache Storage：Cache Storage 是浏览器缓存的⼀部分 ，⽤于存储浏览器的缓存资源。 它可以⽤ 来缓存⽹⻚ 、脚本、样式表、 图像等静态资源 ，以提⾼⽹⻚加载速度和离线访问能⼒。

5. Web SQL Database ：Web SQL Database 是⼀种已被废弃但仍被⼀些浏览器⽀持的关系型数据 库。 它使⽤ SQL 语⾔来进⾏数据操作 ，可以存储⼤量的结构化数据。

追问：service worker 存储的内容是放在 哪⼉的？

Service Worker 可以利⽤ Cache API 和 IndexedDB API 进⾏存储。具体来说：

1. Cache API：Service Worker 可以使⽤ Cache API 将请求的响应存储在浏览器的 Cache Storage 中。Cache Storage 是浏览器的⼀部分 ，⽤于存储缓存的资源。通过 Cache API ，Service Worker 可以将⽹⻚ 、脚本、样式表、 图像等静态资源缓存起来 ，以提⾼⽹⻚加载速度和离线访问能⼒。

2. IndexedDB API：Service Worker 还可以利⽤ IndexedDB API 在浏览器中创建和管理数据库。

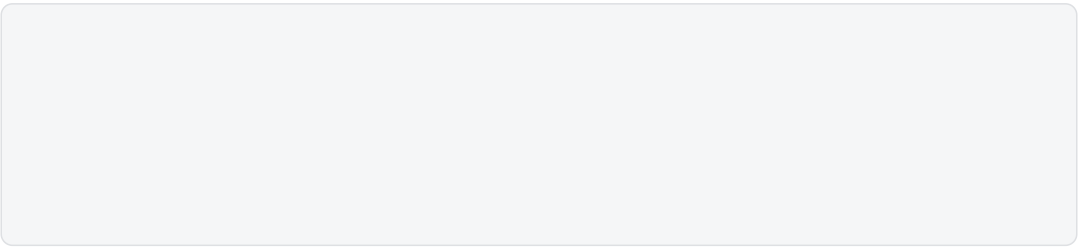
IndexedDB 是⼀种⽤于存储⼤量结构化数据的数据库 ，Service Worker 可以通过 IndexedDB API 进⾏数据的增删改查操作。通过 IndexedDB ，Service Worker 可以将⼤量的数据进⾏持久化存

储 ，以便在离线状态下仍然能够访问和操作数据。

Service Worker 存储的内容并不是放在普通的浏览器缓存或本地数据库中 ，⽽是放在 Service Worker 的全局作⽤域中。Service Worker 运⾏在独⽴的线程中 ，与浏览器主线程分离 ，因此能够独⽴地处理 ⽹络请求和数据存储 ，提供了⼀种强⼤的离线访问和缓存能⼒。

98. [Webpack] 如何打包运⾏时 chunk ， 且在项⽬⼯程中， 如何去加载这个运⾏时 chunk?

Webpack打包运⾏时chunk的⽅式可以通过optimization.runtimeChunk选项来配置。下⾯是⼀个⽰ 例的配置：

1 module.exports = {*// ...*

2 optimization: {

3 runtimeChunk: 'single',

4 },

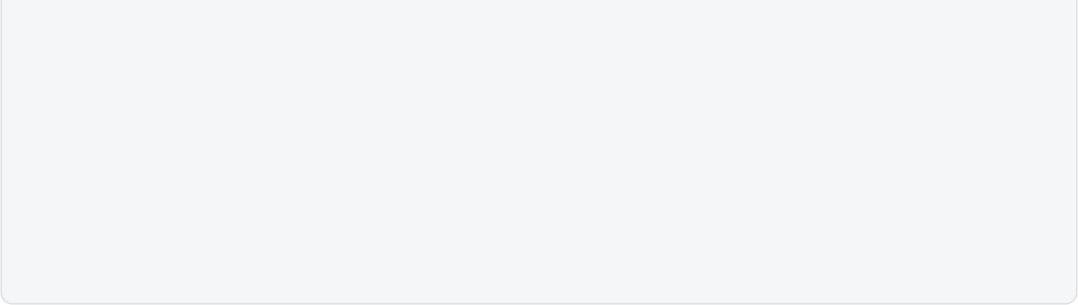
5 };

上述配置中 ，通过设置optimization.runtimeChunk为'single' ，将会把所有的webpack运⾏时代码打 包为⼀个单独的chunk。

在项⽬⼯程中加载运⾏时chunk有两种⽅式：

1. 通过script标签加载：可以使⽤HtmlWebpackPlugin插件来⾃动将运⾏时chunk添加到 HTML ⽂件 中。在webpack配置⽂件中添加以下配置：



1

 HtmlWebpackPlugin = require('html-webpack-plugin');

2

3

module.exports = {*// ...*

plugins: [

4 HtmlWebpackPlugin({*// ...*

5 chunks: ['runtime', 'app'],

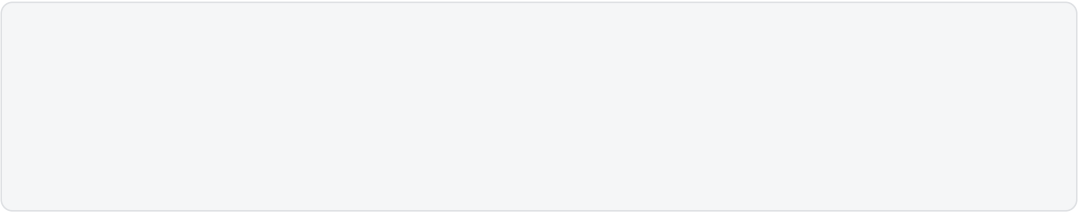
6 }),

7 ],

8 };

上述配置中 ，chunks选项指定了要加载的chunk ，包括运⾏时chunk（ 'runtime'）和其他的业务代码 chunk（ 'app'） 。最终⽣成的HTML⽂件会⾃动引⼊这些chunk。

1. 通过import语句动态加载：可以使⽤动态导⼊的⽅式来加载运⾏时chunk。在需要加载运⾏时 chunk的地⽅ ，使⽤以下代码：



1 (*/\* webpackChunkName: "runtime" \*/* './path/to/runtime').then((runtime) =>

{

2 *// 运⾏时chunk加载完成后的逻辑* 3 });

上述代码中 ，通过import()函数动态加载运⾏时chunk ，通过webpackChunkName注释指定要加载的 chunk名称（这⾥是'runtime'） 。加载完成后 ，可以进⾏相关逻辑处理。

总结：Webpack可以通过optimization.runtimeChunk选项配置打包运⾏时chunk ，可以通过script标 签加载或者使⽤动态导⼊的⽅式来加载运⾏时chunk。

追问

如果只想把某⼏个⽂件打包成运⾏时加载 ， 该如何处理呢？

如果你想将某⼏个⽂件打包成运⾏时加载 ，可以使⽤Webpack的 实现。

entry

import()

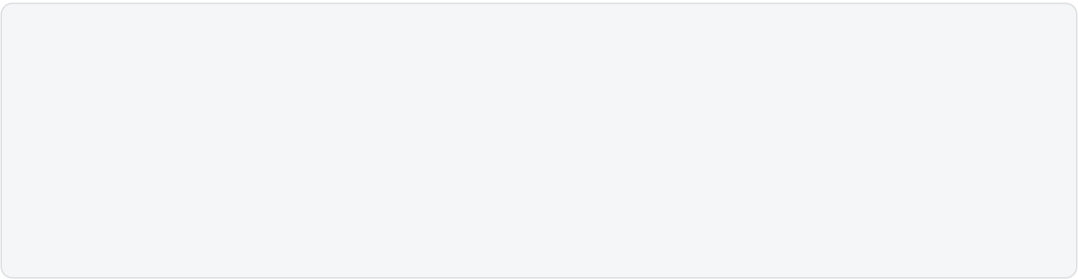
配置和

语法来

⾸先 ，在Webpack的配置⽂件中 ，将这⼏个⽂件指定为单独的

entry

点。例如：

module.exports = {*// ...*

1

2

3

4

5

6

entry: {

main: './src/main.js',

runtime: './src/runtime.js

'

,

},

};

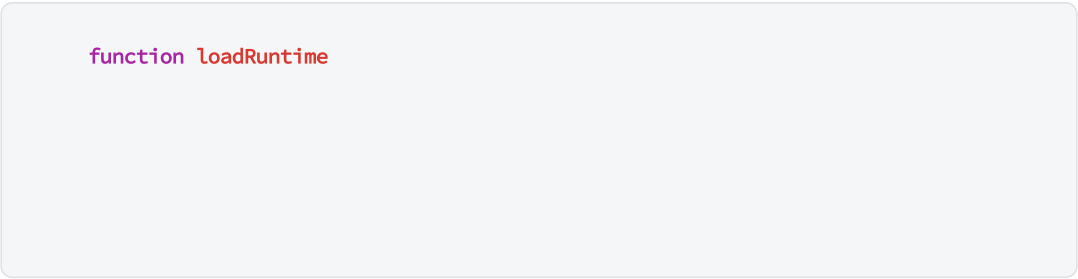
上述配置中 ， main .js 是业务代码的⼊⼝⽂件 ， runtime .js 是你想要打包成运⾏时加载的⽂

件。

然后 ，在你的业务代码中 ，通过

import()

动态导⼊这些⽂件。例如：

1 () {  ('./runtime.js');

2 }

3

4 *//使⽤动态导⼊的⽅式加载运⾏时⽂件*

5 loadRuntime().then(runtime => {*// 运⾏时⽂件加载完成后的逻辑* 6 });

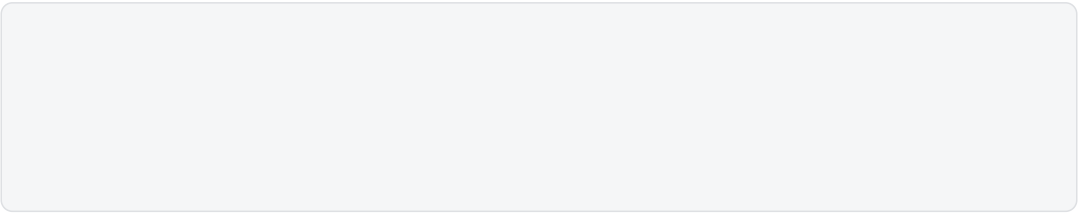
使⽤ import() 会返回⼀个 Promise ，可以通过 .then() 来处理⽂件加载完成后的逻辑。

最后 ，使⽤Webpack进⾏打包时 ，会根据配置的 entry 点和 import() 语法⾃动将这⼏个⽂件打 包成运⾏时加载的模块。运⾏时模块会在需要时动态加载并执⾏。

注意：在使⽤ import() 动态导⼊⽂件时 ，需要确保你的环境⽀持 Promise 和动态导⼊语法。 作为上⾯回复的补充

除了 entry 的⽅式可以处理⾃⼰申明的 runtime ⽂件以外 ， 还可以直接在 import( 'xx ') 的时

候申明； 例如：

1 (*/\* webpackChunkName: "runtime" \*/* './path/to/runtime').then((runtime) =>

{

2 *// 运⾏时chunk加载完成后的逻辑* 3 });



上⾯的⽅式 ， 可以在也可以达到同样的效果 ， 只是在

import

的时候申明runtime⽂件名称⽽已

99. 为何现在市⾯上做表格渲染可视化技术的 ，⼤多数都是 canvas ， ⽽很少⽤ svg 的？ 【热度: 302】

都⽤上了可视化技术做渲染 ， 在这个场景下 ， ⼤多数考虑的是性能；

所以主要基于⼏个⽅⾯去衡量技术⽅案的选择： 性能、动态交互、复杂图形⽀持

• 性能：Canvas 通常⽐ SVG 具有更好的性能。Canvas 是基于像素的绘图技术 ，⽽ SVG 是基于⽮量 的绘图技术。 由于 Canvas 的绘图是直接操作像素 ，所以在⼤规模绘制⼤量图形时 ，Canvas 的性能 优势更为明显。⽽ SVG ⽣成的图形是由 DOM 元素组成 ，每个元素都要进⾏布局和绘制 ，因此在处 理⼤量图形时会有性能瓶颈。

• 动态交互：Canvas 更适合处理动态交互。 由于 Canvas绘制的图形是像素级别的 ，可以直接对图形 进⾏像素级别的操作 ，可以⽅便地进⾏复杂的动画和交互效果。⽽ SVG 的图形是由 DOM 元素组成 的 ，每个元素都要进⾏布局和绘制 ，所以在处理复杂的动态交互时 ，性能⽅⾯可能会受到限制。

• 复杂图形⽀持：Canvas 更适合处理复杂的图形。 由于 Canvas是像素级别的绘制 ，可以直接操作像

素 ，因此可以实现更加灵活和复杂的图形效果 ，⽐如阴影、渐变等。⽽ SVG 的图形是基于⽮量的， 相对来说对复杂图形的⽀持可能会有⼀些限制。

canvas 和 svg 在动态交互上有什么具体的区别？

追问：

元素操作：在 Canvas 中 ，绘制的图形被视为位图 ，⽆法直接访问和操作单个元素 ，需要通过

JavaScript 对整个画布进⾏操作。⽽在 SVG 中 ，每个图形元素都是 DOM 元素 ，可以直接访问和操作 单个元素 ，⽐如修改属性、绑定事件等。

真是场景： ⽐如在 table 开发场景下 ， svg 能通过元素进⾏事件绑定进⾏⽤⼾操作事件驱动 ， ⽐较⽅ 便 ， 但是同样的⽤⼾操作 ， ⽤ canvas 去驱动 ， 显得并不是那么的⽅便。 这个问题 canvas 是如何解 决的？

Table 开发场景下 ，SVG 确实更适合进⾏事件绑定和⽤⼾操作事件驱动。使⽤ SVG ，可以直接操 作每个图形元素 ，为其绑定事件处理程序 ，实现⽤⼾交互。

在

Canvas 在处理⽤⼾操作事件驱动⽅⾯相对不太⽅便 ，因为

⽽

Canvas 绘制的是位图 ，并不直接

⽀持事件绑定。但是可以通过以下⽅式解决这个问题：

• 通过将 Canvas元素放置在 HTML 元素之上 ，再利⽤ CSS 控制其位置和尺⼨ ，实现与⽤⼾交互的感 觉。然后通过监听 HTML 元素的事件 ，通过 JavaScript 判断⽤⼾操作的位置与 Canvas 上的图形元 素是否相交 ，从⽽模拟出⽤⼾交互的效果。

• 使⽤第三⽅库或框架 ，如

Fabric .js、Konva .js 等 ，它们提供了更⾼级的 API 和事件系统 ， 使得在 Canvas上进⾏⽤⼾交互更加⽅便。这些库可以处理⽤⼾操作事件 ，检测点击、拖拽、缩放 等交互操作 ，并提供了事件绑定和管理的⽅法。

通过以上⽅式 ，可以在 Canvas 中实现⼀些基本的⽤⼾交互 ，但相⽐于 SVG 来说 ，Canvas 的事件处理 和⽤⼾交互仍然相对繁琐⼀些。所以在需要⼤量的⽤⼾交互和事件处理的情况下 ，SVG 仍然是更好的 选择。

追问： canvas 如何更为⽅便的提供事件处理能⼒？ 因为 canvas 不能进⾏事件绑定等 ， 显得就

⾮常的不⽅便

在 Canvas 中提供事件处理能⼒ ，可以通过以下两种⽅式更为⽅便：

1. 使⽤第三⽅库或框架：有⼀些流⾏的 Canvas框架可以帮助简化事件处理 ，例如 Fabric.js、

Konva.js 和 EaselJS 等。这些库封装了 Canvas 的底层API ，提供了更⾼级的事件系统和⽅法 ，可 以轻松地为图形元素绑定事件处理程序 ，实现⽤⼾交互。

2. ⼿动实现事件处理：通过监听 HTML 元素的事件（例如⿏标点击、移动、滚轮等），再结合 Canvas 的绘制和坐标计算 ，可以⼿动实现事件处理。 以下是基本的步骤：

◦ 获取⿏标或触摸事件的坐标。

◦ 判断坐标是否在 Canvas绘制区域内。

◦ 找到被点击的图形元素（如果有）。

◦ 根据事件类型执⾏相应的操作 ，如拖拽、缩放、点击等。

追问：Fabric . js是如何进行canvas底层事件api 的封装的?

Fabric.js 是一个强大的 Canvas库，它在提供图形绘制和交互能力的同时，也封装了Canvas的 底层事件API, 简化了事件处理的流程。下面是 Fabric.js 如何封装 Canvas 底层事件 API的 一 些主要方式：

1. 事件监听：Fabric.js 提供了 on 方法，用于在Canvas上注册事件处理程序。可以监听各种 事件，如鼠标点击、移动、滚动、键盘事件等。通过这个方法，可以为整个 Canvas 或图形元素 绑定事件。

2. 事件对象：在事件处理程序中 ，Fabric.js 将底层事件对象进行封装，提供了一个更高级的事 件对象 ( fabric.Event ), 其中包含了更多有用的信息，如事件类型、触发坐标、关联的图形 对象等。

3.坐 标转换：Fabric.js 提供了一系列的方法来处理坐标转换，使得事件处理更加方便。可以通 过get Pointer 方法获取相对于Canvas的坐标，通过localToGlobal 和

globalToLocal 方法在不同坐标系之间进行转换。

4. 交互操作： Fabric .js 提供了一些方便的方法来处理用户交互，如拖拽、缩放、旋转等。通过 drag ging、 scaling 、rotating 等属性和方法，可以轻松地实现这些交互操作，并在事 件处理程序中进行相应的处理。

**100. 在你的项目中，使用过哪些** **webpack plugin,说一下他**

**们的作用**

下表列出了常见的Webpack 插件及其作用：

|  |  |
| --- | --- |
| **插件名称** | **作用** |
| HtmlWebpackP lugin | 自动生成  HTML文件， 并将打包后 的资源自动  注入到  HTML中。 |
| MiniCssExtract Plugin | 将CSS代码 提取到单独 的文件中， 而不是内联 到  JavaScript 中 。 |
| CopyWebpackP lugin | 将指定的文 件或目录复 制到输出目 |

|  |  |
| --- | --- |
|  | 录。 |
| CleanWebpack Plugin | 在每次构建 之前清理输 出目录，避 免旧的文件 残留。 |
| DefinePlugin | 在编译过程 中创建全局 常量，可以 在代码中直 接使用。 |
| HotModuleRepl acementPlugin | 启用热模块 更换(Hot  Module  Replacemen t),在开发 过程中实现 代码修改后 实时更新页 面，无需刷 新。 |
| ProvidePlugin | 自动加载模 块，使模块 在使用时可 以直接使用 对应的全局 变量，无需 |
|  | 引入。 |
| MiniCssExtract Plugin | 将CSS代码 提取到单独 的文件中， 而不是内联 到  JavaScript 中 。 |
| OptimizeCSSAs setsPlugin | 压缩提取出 的CSS文  件。 |
| uglifyjs- webpack- plugin | 压缩  JavaScript 代码。 |
| webpack- | 分析打包后 的文件大  小，并可视 |

|  |  |
| --- | --- |
| bundle- analyzer | **化展示，方** **便优化打包** **结果。** |
| **CompressionW** **ebpackPlugin** | **使用gzip或** **其他压缩算** **法对文件进** **行压缩，减** **小文件大**  **小，加快网** **络传输速**  **度。** |
| CopyWebpackP lugin | 将指定的文 件或目录复 制到输出目 录。 |
| FriendlyErrors WebpackPlugin | 提供友好的  **构建错误提**  示和优化构 建速度的功 能。 |
| ImageminWebp ackPlugin | 压缩图片资 源，减小文 件大小，提 升加载速  度。 |
|  | 启用热模块 更换(Hot  Module  Replacemen t),在开发 过程中实现 代码修改后 实时更新页 面，无需刷 新。 |
| HotModuleRepl acementPlugin |
| HtmlWebpackP lugin | 自动生成  HTML文件， 并将打包后 的资源自动  注入到 HTML中。 |
| lgnorePlugin | 忽略特定的 模块，避免 将其打包到 |

|  |  |
| --- | --- |
|  | **最终的输出** **文件中** |
| **BannerPlugin** | **在打包的文** **件块顶部添** **加自定义的** **注释和信**  **息。** |
| webpack.Defin ePlugin | **在编译过程** **中创建全局** **常量，可以**  在代码中直 接使用。 |
| webpack.Progr essPlugin | 在控制台输 出构建进度 信息。 |
| webpack- bundle- analyzer | 分析打包后 的文件大  小，并可视 化展示，方 便优化打包 结果。 |
| webpackbar | 在命令行中 显示构建进 度条，提供 更直观的构 建进度信  息。 |



这些插件可以根据需要配置在Webpack 的插件列表 ( plugins ) 中，以实现对构建过程的各种增强

和优化操作。

**101. 在你的项目中，使用过哪些** **webpack loader,说一下他**

**们的作用**

|  |  |
| --- | --- |
| **Loader名称** | **作用** |
| babel-loader | 将ES6+代码 转换为ES5 代码，以便 在旧版浏览 器中运行。 |
|  | 解析CSS文 |

|  |  |
| --- | --- |
| css-loader | 件，处理  CSS中的依 赖关系，并 将CSS转换 为JS模块。 |
| style-loader | 将CSS代码 以内联的方 式注入到  HTML页面 中 。 |
| file-loader | 处理文件资 源(如图  片、字体  等 ) , 将 文 件复制到输 出目录，并 返回文件路 径。 |
| url-loader | 与file-loader 类似，但可  以根据文件 大小将文件 转换为Data URL(base6 4格式)或文 件路径。 |
| sass-loader | 解析Sass/ SCSS文件， 并将其转换 为CSS代  码。 |
| less-loader | 解析Less文 件，并将其 转换为CSS 代码。 |
|  | 使用  PostCSS处 理CSS,可 以进行自动 添加前缀、 压缩、CSS Modules等 操作。 |
| postcss-loader |
|  | 将 |

|  |  |
| --- | --- |
| ts-loader | TypeScript 代码转换为 JavaScript 代码。 |
| eslint-loader | 在构建过程 中使用  ESLint进行 代码检查。 |
| stylelint- webpack-  plugin | 在构建过程 中使用  Stylelint进 行CSS/  SCSS代码检 查。 |
| vue-loader | 解析Vue单 文件组件  (.vue文 件 ) , 并 将 其转换为  JavaScript 代码。 |
| image-  webpack- loader | 优化图片资 源，包括压 缩、转换格 式等操作。 |
| html-loader | 解析HTML 文件，处理 其中的引用 资源(如图 片、字体  等 ) , 并 返 回处理后的 HTML代码。 |
| markdown- loader | 将  Markdown  文件转换为 HTML代码。 |
| json-loader | 解析JSON 文件，并返  回解析后的 JavaScript 对象。 |
|  | 在构建过程 中使用 |

|  |  |
| --- | --- |
| eslint-loader | ESLint进行 代码检查。 |
| tslint-loader | 在构建过程 中使用  TSLint进行 TypeScript 代码检查。 |
| prettier-loader | 在构建过程 中使用  Prettier进行  代码格式 化。 |
| stylelint- webpack-  plugin | 在构建过程 中使用  Stylelint进 行CSS/  SCSS代码检 查。 |
| mini-cSs-  extract-plugin | 提 取 C S S 代 码到单独的 文件，而不 是内联到  JavaScript 代码中。 |
| optimize-CSS- assets-  webpack- plugin | 压 缩 C S S 代 码。 |
| terser- webpack- plugin | 压缩  JavaScript 代码。 |

这些Loader 可以根据需要配置在 Webpack 的模块规则 ( module.rules ) 中，以实现对不同类型

文件的处理和转换操作。

**102.[React]** **如何避免不必要的渲染?【热度：632】**

在 React 中，有几种方法可以避免不必要的渲染，以提高性能和优化应用程序的渲染过程：

1. 使 用PureComponent 或 shouldComponentUpdate 方法：继承PureComponent 类或在自定义 组件中实现shouldComponentUpdate 方法，以检查组件的props 和 state是否发生变化。如果 没有变化，则阻止组件的重新渲染。这种方式适用于简单的组件，并且可以自动执行浅比较。

2. 使⽤ React.memo ⾼阶组件：使⽤ React.memo 包装函数组件 ，以缓存组件的渲染结果 ，并仅在 其 props发⽣变化时重新渲染。这种⽅式适⽤于函数组件 ，并且可以⾃动执⾏浅⽐较。

3. 避免在 render ⽅法中创建新对象： 由于对象的引⽤发⽣变化 ，React 将会认为组件的 props或

state 发⽣了变化 ，从⽽触发重新渲染。 因此 ，应尽量避免在 render ⽅法中创建新的对象 ，尤其是 在⼤型数据结构中。

4. 使⽤ key 属性唯⼀标识列表项：在渲染列表时 ，为每个列表项指定唯⼀的 key 属性。这样 ， 当列表 项重新排序、添加或删除时 ，React 可以更准确地确定哪些列表项需要重新渲染 ，⽽不是重新渲染 整个列表。

5. 使⽤ useCallback 和 useMemo 避免不必要的函数和计算：使⽤ useCallback 缓存函数引⽤ ，以确 保只有在其依赖项发⽣变化时才重新创建函数。使⽤ useMemo 缓存计算结果 ，以确保只有在其依 赖项发⽣变化时才重新计算结果。这些钩⼦函数可以帮助避免不必要的函数创建和计算过程 ，从⽽ 提⾼性能。

6. 使⽤ React.lazy 和 Suspense 实现按需加载组件：使⽤ React.lazy 函数和 Suspense 组件可以实现 按需加载组件 ，只在需要时才加载组件代码。这可以减少初始渲染时的资源负载。

103. 全局样式命名冲突和样式覆盖问题怎么解决？【热度: 772】

在前端开发过程中 ，有⼏种常⻅的⽅法可以解决全局样式命名冲突和样式覆盖问题：

1. 使⽤命名空间（ Namespacing） ：给样式类名添加前缀或命名空间 ，以确保每个组件的样式类名不 会冲突。例如 ，在⼀个项⽬中 ，可以为每个组件的样式类名都添加⼀个唯⼀的前缀 ，例



.componentA-button

.componentB-button ，这样可以避免命名冲突。

如

和

2. 使⽤BEM命名规范： BEM（块、元素、修饰符）是⼀种常⽤的命名规范 ，可以将样式类名分成块 （block） 、元素（element）和修饰符（ modifier） 三个部分 ，以确保样式的唯⼀性和可读性。

.button icon 表⽰⼀个元素，

例如，

.button 表⽰⼀个块，

.button--disabled 表

⽰⼀个修饰符。

3. 使⽤CSS预处理器：CSS预处理器（如Sass、 Less） 可以提供变量、嵌套规则和模块化等功能 ，可 以更⽅便地管理样式并避免命名冲突。例如 ，可以使⽤变量来定义颜⾊和尺⼨ ，使⽤嵌套规则来组 织样式 ，并将样式拆分成多个模块。

4. 使⽤CSS模块：CSS模块提供了在组件级别上限定样式作⽤域的能⼒ ，从⽽避免了全局样式的冲突 和覆盖。每个组件的样式定义在组件内部 ，使⽤唯⼀的类名 ，确保样式的隔离性和唯⼀性。

5. 使⽤CSS-in-JS解决⽅案：CSS-in-JS是⼀种将CSS样式直接写⼊JavaScript代码中的⽅法 ，通过将 样式与组件绑定 ，可以避免全局样式的冲突问题。⼀些常⻅的CSS-in-JS解决⽅案包括Styled

Components、 Emotion和CSS Modules with React等。

104. [React] 如何实现专场动画？

这个问题⾮常复杂 ， 我这边⽤⽩话⽂解释⼀下原理 ， 若有不对的地⽅ ， 请⼤家更正：

如果没有专场动画 ， 那么在路由切换的⼀瞬间 ， 加载下⼀个路由⻚⾯的组件 ， 注销上⼀个路由⻚⾯的 组件；

但是如果加上专场动画 ， ⽐如专场动画时间为 500ms ， 那么 ， 在咋合格 500ms过程中 ， ⾸先要加载 下⼀个路由⻚⾯的组件 ， 然后加载上⼀个渐进的动画。

同时不能注销掉当前路由 ， 需要给当前路由加载⼀个渐出的动画。

需要当两个⻚⾯完成动画时间 ， 完成⻚⾯覆盖切换之后 ， 然后注销上⼀个路由⻚⾯的组件； 所以涉及到的知识点：

1. 如何做⻚⾯跳转拦截；

2. 如何在⻚⾯路由组件不跳转的同时 ， 加载下⼀个⻚⾯的组件；

3. 配置⻚⾯层级；

4. 如何执⾏ 、加载、完成专场动画；

5. 动画结束的时候⼿动注销组件；

105. [React]从 React 层⾯上 ， 能做的性能优化有哪些？ 从 React 层⾯上 ，可以进⾏以下性能优化：

1. 使⽤ memoization（记忆化） ：通过使⽤ React.memo() 或 useMemo() 来避免不必要的重新渲 染。这对于纯函数组件和⼤型组件特别有⽤ 。

2. 使⽤ shouldComponentUpdate 或 PureComponent：在类组件中 ，可以通过重写 shouldComponentUpdate ⽅法或使⽤ PureComponent 来避免不必要的重新渲染。

3. 使⽤ React.lazy 和 Suspense：通过使⽤ React.lazy 和 Suspense 来按需加载组件 ，从⽽减少初始 加载时间。

4. 使⽤虚拟化：对于⼤型列表或表格等组件 ，可以使⽤虚拟化技术（如 react-window 或 react- virtualized）来仅渲染可⻅区域内的元素 ，从⽽提⾼性能。

5. 避免不必要的渲染：在函数组件中 ，可以使⽤ useCallback 和 useMemo 来避免不必要的函数创建 和计算, 使⽤ useRef 保持函数应⽤的唯⼀性。

6. 使⽤ key 属性：在使⽤列表或动态元素时 ，确保为每个元素提供唯⼀的 key 属性 ，这有助于 React 有效地识别和更新元素。

7. 使⽤ React DevTools Profiler：使⽤ React DevTools Profiler 来分析组件的渲染性能 ，并找出性能 瓶颈。

8. 使⽤ React.StrictMode ：在开发环境中 ，可以使⽤ React.StrictMode 组件来检测潜在的问题和不 安全的使⽤ 。

9. 避免深层嵌套：尽量避免过多的组件嵌套 ，这可能会导致性能下降。

10. 使⽤组件分割：将⼤型组件拆分成多个⼩组件 ，可以提⾼组件的可维护性和性能。

这些是⼀些常⻅的 React 层⾯上的性能优化技巧 ，根据具体的应⽤场景和需求 ，可能还有其他优化⽅ 式。

106. [Vue] 中为何不要把 v-if 和 v-for 同时⽤在同⼀个元素 上 ， 原理是什么？【热度: 546】



v-



确实 ，将

v-if 和

v-for 同时⽤在同⼀个元素上可能会导致性能问题。原因在于

v-for 具有⽐

if 更⾼的优先级 ，它会在每次渲染的时候都会运⾏。 这意味着 ，即使在某些情况下

v-if 的条件为

v-for 仍然会对数据进⾏遍历和渲染。

false ，

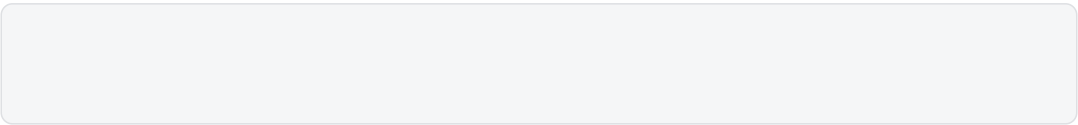
这样会导致⼀些不必要的性能消耗 ，特别是当数据量较⼤时。Vue在渲染时会尽量复⽤已经存在的元 素 ，⽽不是重新创建和销毁它们。但是当

v-for 遍历的数据项发⽣变化时 ，Vue会使⽤具有相同

v-if 的条件可能会影响到之前的元素 ，导致⼀些不符合预期的⾏为。

key 的元素 ，此时

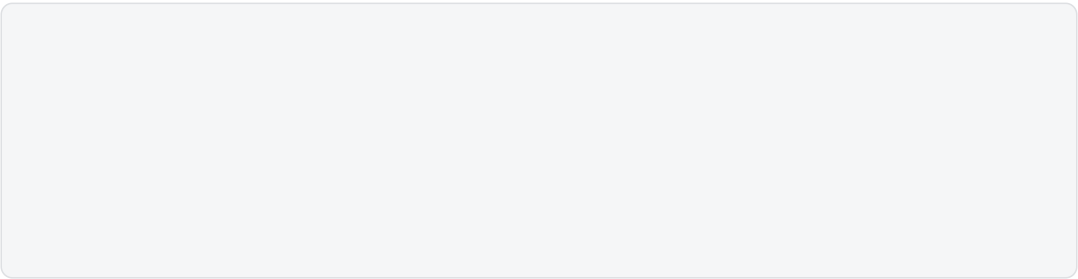
让我们来看⼀个具体的例⼦来说明这个问题。 假设我们有以下的Vue模板代码：

1 <ul<li v-for="item in items" v-if="item.isActive"{{ item.name }}</li</ul

这⾥我们使⽤ v-for 来循环渲染 items 数组 ，并且使⽤ v-if 来判断每个数组项是否是活动状

态。现在 ，让我们看⼀下Vue的源码 ，特别是与渲染相关的部分。

在Vue的渲染过程中 ，它会将模板解析为AST（抽象语法树） ，然后将AST转换为渲染函数。对于上⾯ 的模板 ，渲染函数⼤致如下：

1   () {  \_c('ul',null,

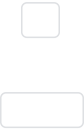
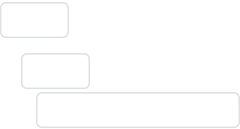
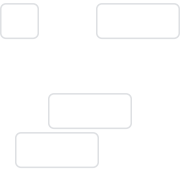
2 \_l(items,  (item) {  item.isActive ? \_c('li', null,

\_v(\_s(item.name))) : \_e();

3 })

4 );

5 }

上⾯的代码中 ， \_l 是由 v-for 指令⽣成的渲染函数。 它接收⼀个数组和⼀个回调函数 ，并在每个 数组项上调⽤回调函数。 回调函数根据 v-if 条件来决定是否渲染 li 元素。



问题出在这⾥： 由于 v-for 的优先级⽐ v-if ⾼ ，所以每次渲染时都会执⾏ v-for 循环 ，⽆论v-

if 的条件是否为 false 。这意味着即使 item .isActive 为 false ，Vue仍然会对它进⾏遍历

和渲染。

v-for 遍历的

此外 ，Vue在渲染时会尽量复⽤已经存在的元素 ，⽽不是重新创建和销毁它们。但是当

item .isActive 从

key 的元素。在上⾯的例⼦中 ，如果

数据项发⽣变化时 ，Vue会使⽤具有相同

v-if 条件。这可能会导致⼀

true 变为

li 元素 ，并在其上应⽤

false ，Vue会尝试复⽤之前的

些不符合预期的⾏为。

v-for 。如果需要根据

v-if 和

为了避免这种性能问题 ，Vue官⽅推荐在同⼀个元素上不要同时使⽤

v-for 的过滤器来处理数据。或者 ，将

条件来决定是否渲染循环的元素 ，可以考虑使⽤计算属性或者

v-for 进⾏循环渲染 ，以确保每次渲染时都能正确地应⽤

条件判断放在外层元素上 ，内层元素使⽤

v-if 条件。

107. 将静态资源缓存在本地的⽅式有哪些？ 【热度: 584】

浏览器可以使⽤以下⼏种⽅式将前端静态资源缓存在本地：

1. HTTP缓存：浏览器通过设置HTTP响应头中的Cache-Control或Expires字段来指定资源的缓存策 略。 常⻅的缓存策略有： no-cache（每次都请求服务器进⾏验证） 、no-store（不缓存资源） 、 max-age（设置资源缓存的最⼤时间）等。浏览器根据这些缓存策略来决定是否将资源缓存在本 地。

2. ETag/If-None-Match ：服务器可以通过在响应头中添加ETag字段 ，⽤于标识资源的版本号。 当浏 览器再次请求资源时 ，会将上次请求返回的ETag值通过If-None-Match字段发送给服务器 ， 由服务 器判断资源是否发⽣了变化。如果资源未发⽣变化 ，服务器会返回304 Not Modified状态码 ，浏览 器则直接使⽤本地缓存的资源。

3. Last-Modified/If-Modified-Since ：服务器可以通过在响应头中添加Last-Modified字段 ，⽤于标识 资源的最后修改时间。浏览器再次请求资源时 ，会将上次请求返回的Last-Modified值通过If-

Modified-Since字段发送给服务器。服务器根据资源的最后修改时间判断资源是否发⽣了变化 ，如 果未发⽣变化 ，则返回304 Not Modified状态码 ，浏览器使⽤本地缓存的资源。

4. Service Worker缓存：使⽤Service Worker可以将前端资源缓存在浏览器的Service Worker缓存

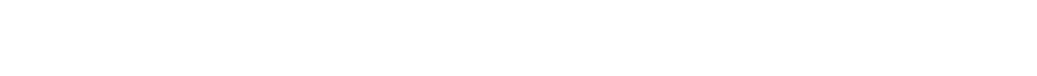
中。Service Worker是运⾏在浏览器后台的脚本 ，它可以拦截和处理⽹络请求 ，因此可以将前端资 源缓存起来 ，并在离线状态下提供缓存的资源。

5. LocalStorage或IndexedDB：对于⼀些⼩的静态资源 ，可以将其存储在浏览器的LocalStorage或 IndexedDB中。这些存储⽅式是浏览器提供的本地存储机制 ，可以将数据以键值对的形式存储在浏 览器中 ，从⽽实现缓存的效果。

如何将静态资源缓存在 LocalStorage或IndexedDB

以下是⼀个使⽤LocalStorage将静态资源缓存的⽰例代码：

• ***/****定义⼀个数组，包含需要缓存的静态资源的URL*var resources =

[['https://example.com/css/style.css','https://example.com/js/main.js','https://example.com/i mages/logo.png'](https://example.com/css/style.css','https://example.com/js/main.js','https://example.com/images/logo.png')

];

***/****遍历资源数组，将资源请求并存储在LocalStorage中* resources.forEach(function(url) {***/****发起资源请求*

fetch(url)

.then(function(response) {***/****检查请求是否成功*if (!response.ok) {throw new Error('Request failed: 'response.status);

}***/****将响应数据存储在LocalStorage中*return response.text(); })

.then(function(data) {***/****将资源数据存储在LocalStorage中，以URL作为键名* localStorage.setItem(url, data);console.log('Resource cached: 'url);

})

.catch(function(error) {console.error(error); });

});

以下是⼀个使⽤IndexedDB将静态资源缓存的⽰例代码：

• ***/****打开或创建⼀个IndexedDB数据库*var request = indexedDB.open('myDatabase', 1);

***/****创建或更新数据库的对象存储空间*

request.onupgradeneeded = function(event) {var db = event.target.result;var objectStore = db.createObjectStore('resources', { keyPath: 'url' });

objectStore.create Index('url', 'url', { unique: true }); };

***/****成功打开数据库后，将资源请求并存储在IndexedDB中*

request.onsuccess = function(event) {var db = event.target.result;var transaction =

db.transaction('resources', 'readwrite');var objectStore = transaction.objectStore('resources');

resources.forEach(function(url) {***/****发起资源请求* fetch(url)

.then(function(response) {***/****检查请求是否成功*if (!response.ok) {throw new Error('Request failed: 'response.status);

}***/****将响应数据存储在IndexedDB中*return response.blob(); })

.then(function(data) {***/****创建⼀个资源对象，以URL作为键名*var resource = { url: url, data: data };***/****将资源对象存储在IndexedDB中*

objectStore.put(resource);console.log('Resource cached: 'url); })

.catch(function(error) {console.error(error);

}); });

***/****完成事务*

transaction.oncomplete = function() {console.log('All resources cached in IndexedDB.'); };

transaction.onerror = function(event) {console.error('Transaction error:', event.target.error); };

以上};代码仅为⽰例 ，实际应⽤中需要根据具体的需求进⾏相应的优化和错误处理。

108. SPA⾸屏加载速度慢的怎么解决 【热度: 868】 影响⾸屏可能得因素

• ⽹络延时问题

• 资源⽂件体积是否过⼤

• 资源是否重复发送请求去加载了

• 加载脚本的时候 ，渲染内容堵塞了

• 加载脚本的时候 ，渲染内容堵塞了

解决⽅案

• 减⼩⼊⼝⽂件积

• 静态资源本地缓存

• UI框架按需加载

• 图⽚资源的压缩

• 组件重复打包

• 开启 GZip 压缩

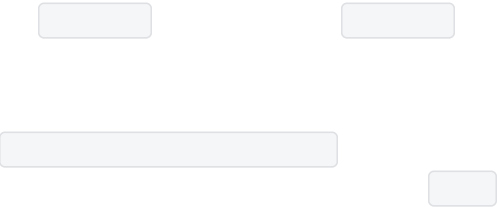
• 使⽤ SSR

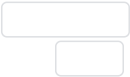
• 启⽤ CDN 加速

109. axios 是如何区分是 nodejs 环境还是 浏览器环境 的？ 【热度: 13】

Axios 是⼀个跨平台的 HTTP 客⼾端库 ，可以在浏览器和 Node.js 中使⽤ 。Axios 通过判断当前环境来 确定是在浏览器还是在 Node.js 环境中运⾏。

在浏览器环境中 ，Axios 默认会使⽤浏览器提供的 XMLHttpRequest 对象来发送 HTTP 请求。

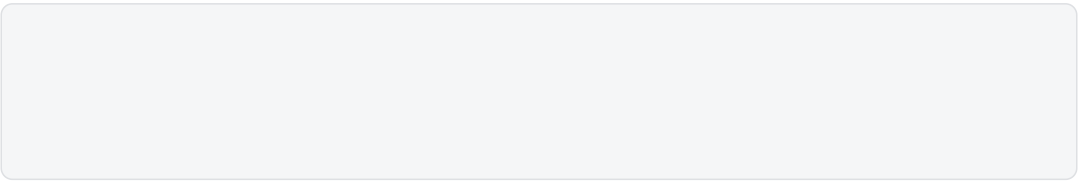
在 Node.js 环境中 ，Axios 会检查是否存在 process 全局对象 ，以及 process 对象中是否存在

nextTick ⽅法。如果存在以上两个条件 ，Axios 就默认在 Node.js 环境中运⾏ ，并使⽤ Node.js 内 置的 http 模块发送 HTTP 请求。

如果需要明确指定运⾏环境 ，可以使⽤ axios .defaults .adapter 属性来设置适配器

（adapter） ，以便在需要时⼿动选择使⽤ XMLHttpRequest 或 Node.js 内置的 http 模块。

例如 ，在 Node.js 环境中可以这样设置适配器：

1  axios = require('axios');

2  httpAdapter = require('axios/lib/adapters/http');

3 axios.defaults.adapter = httpAdapter;

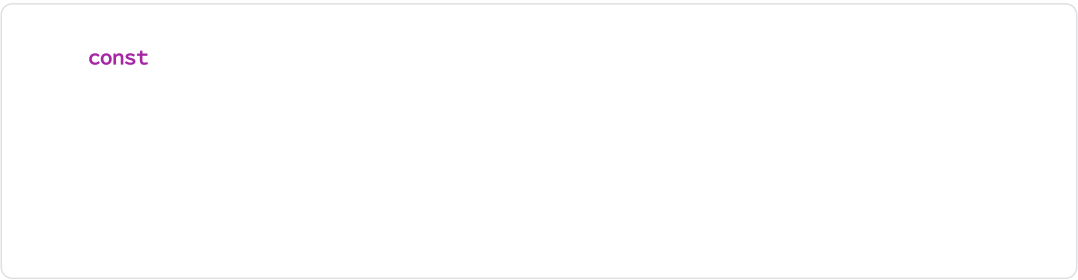
通过上述⽅式 ，Axios 可以根据环境⾃动选择适当的底层实现来发送 HTTP 请求 ，使其在不同的环境中 都能正常⼯作。

110. 如何拦截 web 应⽤的请求 【热度: 487】

在前端拦截和处理 Web 应⽤的所有请求 ，可以使⽤以下⽅法：

1. 使⽤ Fetch 或 XMLHttpRequest：在前端代码中使⽤ Fetch API 或 XMLHttpRequest 对象发送请 求。通过拦截 Fetch 或 XMLHttpRequest 对象的 open 和 send ⽅法 ，可以在请求发出前进⾏拦截

和修改。这样可以捕获请求的相关信息 ，并进⾏相应的处理。 ⽰例代码（使⽤ Fetch API）：



1. 使⽤ Service Worker ：Service Worker 是⼀种在浏览器背后运⾏的脚本 ，可以拦截和处理⽹络请 求。通过注册⼀个 Service Worker ，可以在其中监听和处理请求事件。从⽽实现拦截和处理 Web

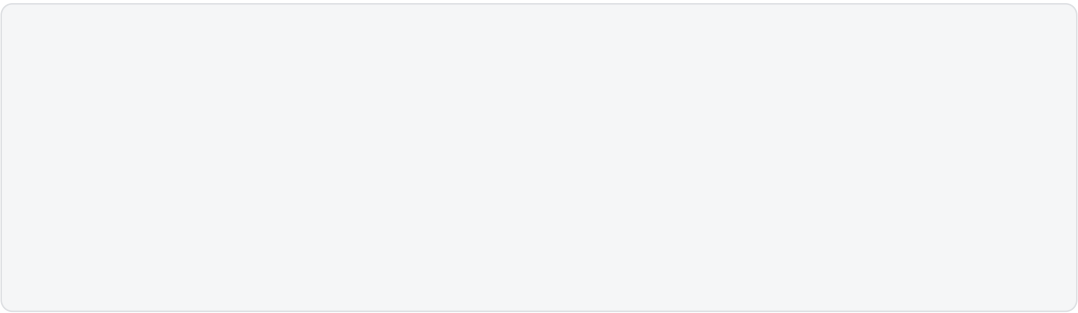
1 originalFetch = window.fetch;

2 window.fetch =  (url, options) {*// 在请求发出前进⾏拦截和处理* *console.log('拦截到请求:', url);*

3 *// 可以修改请求的相关信息// options.headers['Authorization'] = 'Bearer token';* *originalFetch.apply(**, arguments);*

4 };

应⽤的所有请求。 ⽰例代码：

1 self.addEventListener('fetch', (event) {*// 在请求发出前进⾏拦截和处理* *console.log('拦截到请求:', event.request.url);*

2 *// 可以修改请求的相关信息// event.request.headers.set('Authorization', 'Bearer token');*

3

4 event.respondWith(fetch(event.request));

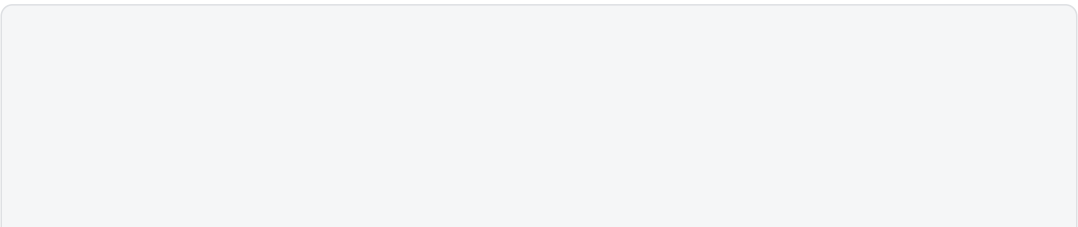
5 });

需要注意的是 ，前端拦截和处理请求只能在客⼾端进⾏ ，对于服务器端的请求⽆法拦截。此外 ，拦截 和处理请求可能会对性能产⽣⼀定的影响 ，因此要根据实际情况进⾏权衡和调优。 同时 ，对于⼀些敏 感信息（如密码、个⼈信息等） ，应该谨慎处理 ，确保安全性。

1. 如果是使⽤的反⽅库 ， ⽐如 aixos ， 可以直接使⽤三⽅库提供的能⼒

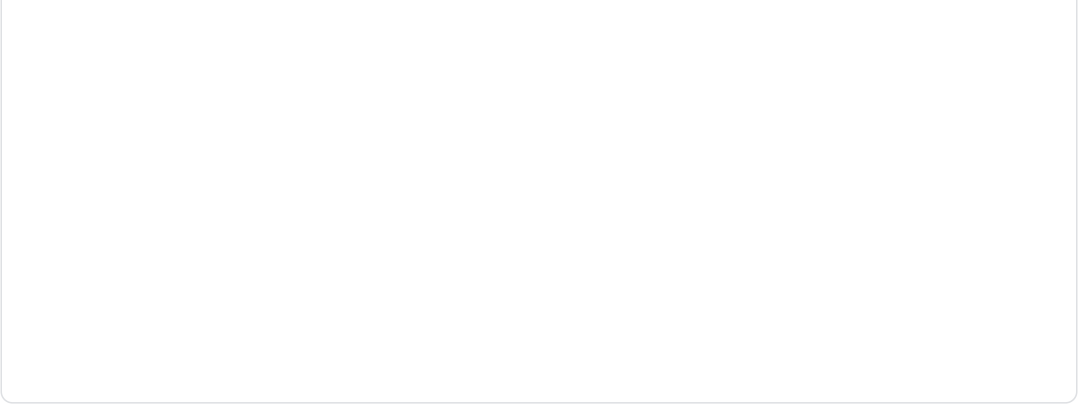
是的 ，使⽤ axios 也可以拦截请求。axios 提供了拦截器（ interceptors） 的功能 ，可以在请求发出前 进⾏拦截和处理。

⽰例代码：

1  axios  'axios'; 2

3 *// 请求拦截器*

4 axios.interceptors.request.use(  (config) {*// 在请求发出前进⾏拦截和处理* *console.log('拦截到请求:', config.url);*



5 *// 可以修改请求的相关信息// config.headers['Authorization'] = 'Bearer token';* *config;*

6 },  (error) {  Promise.reject(error);

7 });

8

9 *//发送请求*

10 axios.get('/api/data')

11 .then(  (response) {console.log(response.data);

12 })

13 .catch(  (error) {console.error(error);

14 });



interceptors .request

在上述代码中 ，通过使⽤

⽅法 ，可以对请求进⾏拦截和处理。在拦截器

函数中 ，可以修改请求的相关信息 ，并返回修改后的配置对象。

使⽤ axios 拦截请求只能在客⼾端进⾏ ，对服务器端的请求⽆法拦截。 同样需要谨慎处理敏感信息 ，并 确保安全性。

111. 前端有哪些跨⻚⾯通信⽅式？ 【热度: 491】

在前端中 ，有多种跨⻚⾯通信的⽅式 ，下⾯列举了其中⼀些常⻅的⽅式：

1. 使⽤URL参数：可以通过URL参数在不同⻚⾯之间传递数据。例如 ，可以在URL中添加查询字符串 参数来传递数据 ，并通过解析URL参数来获取传递的数据。

2. 使⽤localStorage或sessionStorage：可以使⽤浏览器的本地存储（ localStorage或

sessionStorage） 在不同⻚⾯之间共享数据。⼀个⻚⾯可以将数据存储在本地存储中 ，另⼀个⻚⾯ 可以读取该数据。

3. 使⽤Cookies ：可以使⽤Cookies在不同⻚⾯之间共享数据。⼀个⻚⾯可以将数据存储在Cookie 中 ，另⼀个⻚⾯可以读取该Cookie。

4. 使⽤postMessage API： postMessage API允许不同窗⼝或iframe之间进⾏跨⻚⾯通信。可以使⽤ postMessage发送消息 ，接收⽅可以通过监听message事件来接收消息。

5. 使⽤Broadcast Channel API： Broadcast Channel API允许不同⻚⾯或不同浏览器标签之间进⾏⼴ 播式的消息传递。可以使⽤Broadcast Channel发送消息 ，其他订阅同⼀频道的⻚⾯都可以接收到 消息。

6. 使⽤Shared Worker ：Shared Worker是⼀种特殊的Web Worker ，可以在多个⻚⾯之间共享。可 以通过Shared Worker进⾏通信和共享数据。

7. 使⽤WebSocket：WebSocket是⼀种双向通信协议 ，可以在不同⻚⾯之间建⽴持久的连接 ，实现 实时的跨⻚⾯通信。

以上是⼀些常⻅的跨⻚⾯通信⽅式 ，选择适合⾃⼰需求的⽅式来实现跨⻚⾯通信。