2025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. 首先在页面中创建一个 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，比如页面的某个区域、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，实现了对整个页面的截图：

```
XML
htmlCopy code<!DOCTYPE html><html<head<meta charset="UTF-8"<title
截图示例</title<style
      #canvas {position: fixed;left: 0;top: 0;z-index: 9999;}
    </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
      const canvas = document.getElementById('canvas');const ctx =
canvas.getContext('2d');const btn =
document.getElementById('btn');
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
      btn.addEventListener('click', () => {
        ctx.clearRect(0, 0, canvas.width, canvas.height);
        ctx.drawImage(document.documentElement, 0, 0);const
imgData = canvas.toDataURL();console.log(imgData);
      });</script</body</html
```

这个例子中，在页面中创建了一个 `canvas` 元素，并设置其宽高和样式，将其放在页面最上方。在点击“截图”按钮时，通过 `toDataURL()` 方法将整个页面的截图转换为 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、耗时长的操作异步处理，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。

以上是一些常见的优化手段，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.js` 或 `bignumber.js`，这些库可以处理任意精度的数值。

例如，使用 `big.js` 库可以将两个超过 `Number.MAX_VALUE` 的数相加：

```JavaScript
const big = require('big.js');

const x = new big('9007199254740993');
const y = new big('10000000000000000');

const result = x.plus(y);

console.log(result.toString()); // 输出：10000900719492547409 3
```

这里创建了两个 `big.js` 对象 `x` 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：

```JavaScript
const 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，返回不同的页面或数据。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 页面或接口数据。
3. 如果是 PC 设备，可以返回一个 web 应用页面或接口数据。

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. 如何保证用户的使用体验

【如何保证用户的使用体验】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. 用户线上问题反馈，线上 on call 的思考
3. 用户使用体验的思考， 交互体验使用方向

4. 提升用户能效方向思考

6. 如何解决页面请求接口大规模并发问题

如何解决页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： 比如提取页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： 只是对当前访问页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： 从服务端把页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

页性能。通过 Performance API，可以获取页面各个阶段的时间、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. 对于请求耗时较长的接口，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，允许用户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：

```
JavaScript
// performance.jsconst performance = {
  timings: {},
  config: {
    reportUrl: '/report',
  },
  init() {// 监听所有请求的开始时间
window.addEventListener('fetchStart', (event) =>
{this.timings[event.detail.id] = {
        startTime: Date.now(),
      };
    });
// 监听所有请求的结束时间，并计算请求耗时
```

```
window.addEventListener('fetchEnd', (event) => {const id =
event.detail.id;if (this.timings[id]) {const timing =
this.timings[id];
        timing.endTime = Date.now();
        timing.duration = timing.endTime - timing.startTime;
// 将耗时信息发送到服务端const reportData = {
          url: event.detail.url,
          method: event.detail.method,
          duration: timing.duration,
        };this.report(reportData);
    }
    });
  },
  report(data) {// 将耗时信息发送到服务端const xhr = new
XMLHttpRequest();
    xhr.open('POST', this.config.reportUrl);
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.send(JSON.stringify(data));
  },
};

export default performance;
```

在应用入口处引入该模块:

```
JavaScript
// main.jsimport performance from './performance';
performance.init();
```

在每个请求的响应拦截器中触发 `fetchEnd` 事件:

```
JavaScript
// fetch.jsimport EventBus from './EventBus';

const fetch = (url, options) => {const id =
Math.random().toString(36).slice(2);const fetchStartEvent = new
CustomEvent('fetchStart', {
    detail: {
      id,
      url,
      method: options.method || 'GET',
    },
  });
  EventBus.dispatchEvent(fetchStartEvent);
```

```
return window.fetch(url, options)
    .then((response) => {const fetchEndEvent = new
CustomEvent('fetchEnd', {
        detail: {
          id,
          url,
          method: options.method || 'GET',
        },
      });
      EventBus.dispatchEvent(fetchEndEvent);
return response;
    });
};

export default 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

客户端 JS 代码实现如下

```
JavaScript
function submitUpload() {var chunkSize = 210241024;//分片大小 2Mvar
```

```
file = document.getElementById('f1').files[0];var chunks = [], //
保存分片数据
    token = (+new Date()),//时间戳
    name = file.name, chunkCount = 0, sendChunkCount = 0;
//拆分文件 像操作字符串一样 if (file.size > chunkSize) {//拆分文件var
start = 0, end = 0;while (true) {
      end += chunkSize;var blob = file.slice(start, end);
      start += chunkSize;
//截取的数据为空 则结束 if (!blob.size) {//拆分结束 break;
      }
      chunks.push(blob);//保存分段数据
    }
  } else {
    chunks.push(file.slice(0));
  }
  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
for (var i = 0; i < chunkCount; i++) {var fd = new FormData();
//构造 FormData 对象
    fd.append('token', token);
    fd.append('f1', chunks[i]);
    fd.append('index', i);
    xhrSend(fd, function() {
      sendChunkCount += 1;if (sendChunkCount === chunkCount) {//上
传完成，发送合并请求 console.log('上传完成，发送合并请求');var formD =
new FormData();
        formD.append('type', 'merge');
        formD.append('token', token);
        formD.append('chunkCount', chunkCount);
        formD.append('filename', name);
        xhrSend(formD);
      }
    });
  }
}

function xhrSend(fd, cb) {
var xhr = new XMLHttpRequest();   //创建对象
  xhr.open('POST', 'http://localhost:8100/', true);
  xhr.onreadystatechange = function() {console.log('state change',
xhr.readyState);if (xhr.readyState == 4)
{console.log(xhr.responseText);
      cb && cb();
```

```
    }
  }
  xhr.send(fd);//发送
}

//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，代码见 fnMergeFile 方法。

```
JavaScript
//二次处理文件，修改名称
app.use((ctx) => {var body = ctx.request.body;var files =
ctx.request.files ? ctx.request.files.f1 : [];//得到上传文件的数组
var result = [];var fileToken = ctx.request.body.token;// 文件标识
var fileIndex = ctx.request.body.index;//文件顺序if (files
&& !Array.isArray(files)) {//单文件上传容错
    files = [files];
  }
  files && files.forEach(item => {var path = item.path;var fname =
item.name;//原文件名称var nextPath = path.slice(0,
path.lastIndexOf('/') + 1) + fileIndex + '-'fileToken;if
(item.size > 0 && path) {//得到扩展名var extArr =
fname.split('.');var ext = extArr[extArr.length - 1];//var
nextPath = path + '.' + ext;//重命名文件
      fs.renameSync(path, nextPath);
      result.push(uploadHost +
nextPath.slice(nextPath.lastIndexOf('/') + 1));
    }
  });
if (body.type === 'merge') {//合并分片文件var filename =
body.filename,
      chunkCount = body.chunkCount,
      folder = path.resolve(__dirname, '../static/uploads') + '/';
var writeStream = fs.createWriteStream(
${folder}${filename}
);
var cindex = 0;
//合并文件function fnMergeFile() {var fname = ${folder}${cindex}-
${fileToken};var readStream = fs.createReadStream(fname);
      readStream.pipe(writeStream, { end: false });
      readStream.on("end", function() {
```

```
        fs.unlink(fname, function(err) {if (err) {throw err;
          }
        });if (cindex + 1 < chunkCount) {
          cindex += 1;
          fnMergeFile();
        }
      });
    }
    fnMergeFile();
    ctx.body = 'merge ok 200';
  }
});
```

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`

客户端 JS 代码：

```
JavaScript
//获得本地缓存的数据function getUploadedFromStorage() {return
JSON.parse(localforage.getItem(saveChunkKey) || "{}");
}

//写入缓存function setUploadedToStorage(index) {var obj =
getUploadedFromStorage();
  obj[index] = true;
  localforage.setItem(saveChunkKey, JSON.stringify(obj));
}

//分段对比var uploadedInfo = getUploadedFromStorage();//获得已上传
的分段信息for (var i = 0; i < chunkCount; i++)
```

```
{console.log('index', i, uploadedInfo[i] ? '已上传过' : '未上传');
if (uploadedInfo[i]) {//对比分段
    sendChunkCount = i + 1;//记录已上传的索引continue;//如果已上传则跳过
  }var fd = new FormData();    //构造 FormData 对象
  fd.append('token', token);
  fd.append('f1', chunks[i]);
  fd.append('index', i);
  (function(index) {
    xhrSend(fd, function() {
      sendChunkCount += 1;//将成功信息保存到本地
      setUploadedToStorage(index);if (sendChunkCount === chunkCount) {console.log('上传完成，发送合并请求');var formD = new FormData();
        formD.append('type', 'merge');
        formD.append('token', token);
        formD.append('chunkCount', chunkCount);
        formD.append('filename', name);
        xhrSend(formD);
      }
    });
  })(i);
}
```

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，客户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页在不同的移动设备上显示效果相同。下面是一些常见的 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，来控制页面的缩放比例和宽度，以适配不同的设备。例如：

```
XML
<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：

```
Plain Text
arduinoCopy code@media screen and (max-width: 640px) {/* 样式 */
}
```

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：

```
CSS
html {font-size: 16px;
}@media screen and (max-width: 640px) {html {font-size: 14px;}
}

div {width: 10rem;
}
```

其中 `font-size: 16px` 表示将网页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：

```
XML
arduinoCopy code// index.html
<script src="https://cdn.bootcdn.net/ajax/libs/lib-flexible/0.3.4/flexible.js"</script
// index.js
```

```
import 'lib-flexible/flexible.js'
```

其中 `flexible.js` 会在页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：

```
CSS
div {width: 100px;height: 100px;
}
```

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 动态修改页面的样式，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，都需要在设计页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 实现网页换肤可以使用 CSS 变量和 Less 变量。CSS 变量的语法如下：

```
CSS
:root {--primary-color: #007bff;
}.btn {background-color: var(--primary-color);
}
```

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：

```
Plain Text
lessCopy code@primary-color: #007bff;
```

```
.btn {background-color: @primary-color;
}
```

通过 Less 变量来实现网页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 样式应用到页面上。

以下是一段实现通过 Less 变量来实现网页换肤的示例代码：

```
Plain Text
// base.less 文件@primary-color: #007bff;

.btn {background-color: @primary-color;
}

// dark.less 文件@primary-color: #343a40;
```

```
XML
<!-- index.html 文件 --><!DOCTYPE html><html<head<meta
charset="UTF-8"<title 网页换肤示例</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
    function changeSkin() {// 修改 Less 变量的值
      less.modifyVars({'@primary-color': '#28a745'
      }).then(() => {console.log('换肤成功');
      }).catch(() => {console.error('换肤失败');
      });
    }</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. 如何实现网页加载进度条?

监听静态资源加载情况

可以通过 `window.performance` 对象来监听页面资源加载进度。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()` 方法获取页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:

```
JavaScript
const resources = window.performance.getEntriesByType('resource');
const totalResources = resources.length;
let loadedResources = 0;
resources.forEach((resource) => {if (resource.initiatorType !==
'xmlhttprequest') {// 排除 AJAX 请求
    resource.onload = () => {
      loadedResources++;const progress =
Math.round((loadedResources / totalResources) * 100);
      updateProgress(progress);
    };
  }
});

function updateProgress(progress) {// 更新进度条
}
```

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，因为它们不属于页面资源。

当所有资源加载完成后，页面就会完全加载。

实现进度条

网页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页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。

```XML
<progress id="progressBar" value="0" max="100"</progress
```

然后在 JavaScript 中，监听页面加载事件和资源加载事件，实时更新 `progress` 元素的 `value` 属性。

```JavaScript
const progressBar = document.getElementById('progressBar');

window.addEventListener('load', () => {
  progressBar.value = 100;
});

document.addEventListener('readystatechange', () => {const
progress = Math.floor((document.readyState / 4) * 100);
  progressBar.value = progress;
});
```

2. 使用第三方库

使用第三方库可以更加方便地实现网页加载进度条，下面以 `nprogress` 库为例：

1. 安装 `nprogress` 库

```Bash
bashCopy codenpm install nprogress --save
```

1. 在页面中引入 `nprogress.css` 和 `nprogress.js`

```XML
<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` 并监听页面加载事件和资源加载事件

```JavaScript
// 初始化 nprogress
NProgress.configure({ showSpinner: false });

// 监听页面加载事件window.addEventListener('load', () => {
  NProgress.done();
});
```

```
// 监听资源加载事件document.addEventListener('readystatechange', ()
=> {if (document.readyState === 'interactive') {
    NProgress.start();
  } else if (document.readyState === 'complete') {
    NProgress.done();
  }
});
```

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. 常见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。这可以大大减轻页面的加载时间，并降低带宽消耗，提高了用户的体验。以下是一些常见的实现方法:

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。

```
JavaScript
let observer = new IntersectionObserver(function (entries) {
  entries.forEach(function (entry) {if (entry.isIntersecting)
{const lazyImage = entry.target;
      lazyImage.src = lazyImage.dataset.src;
      observer.unobserve(lazyImage);
    }
  });
});

const lazyImages = [...document.querySelectorAll(".lazy")];
lazyImages.forEach(function (image) {
  observer.observe(image);
});
```

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。

```
JavaScript
```

```
function lazyLoad() {const images =
document.querySelectorAll(".lazy");const scrollTop =
window.pageYOffset;
  images.forEach((img) => {if (img.offsetTop < window.innerHeight
+ scrollTop) {
      img.src = img.dataset.src;
      img.classList.remove("lazy");
    }
  });
}

let lazyLoadThrottleTimeout;
document.addEventListener("scroll", function () {if
(lazyLoadThrottleTimeout) {
    clearTimeout(lazyLoadThrottleTimeout);
  }
  lazyLoadThrottleTimeout = setTimeout(lazyLoad, 20);
});
```

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，避免页面布局的混乱。

```
XML
<!-- 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，能够显著提高网页的性能和用户体验。

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，用于在客户端存储和读取信息。cookie 是由服务器发送的，客户端可以使用浏览器 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：

```
Nginx
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" 的域名。这个 cookie 就会被存储在客户端，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. 用户在 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 端的页面上。

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

3. 一旦用户扫描了二维码，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，则该中转站将用户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户进行认证，并创建一个与该用户关联的会话状态。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. 当用户在 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，从而允许用户在 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，并使用户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` 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）为其用户提供 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 出现的风险，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页面上，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页面上，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页内容的来源和版本。

添加明水印手段有哪些

可以参考这个文档： https://zhuanlan.zhihu.com/p/374734095

总计一下：

1. 重复的 dom 元素覆盖实现： 在页面上覆盖一个 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， 如何防止用户删除对应的 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户删除对应的 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。以下是一些常见的方法：

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，可以避免用户通过编辑页面 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，这种方式会影响网页的可读性，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，可以防止用户直接删掉该类名，进而删除水印。但是，用户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，可以使得用户删除水印较为困难。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， 如果用户改变了该 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，从而避免用户直接在网页源文件中删除 CSS 代码。但需要注意的是，这种方案会稍稍加重网页的加载速度，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，然后将其作为网页的背景图片，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 图形嵌入到网页中进行展示。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见的一些元素：

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。以下是一些常见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、业务领域或者页面进行划分。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. 页面划分：将前端系统按照页面进行划分，每个页面对应一个领域模型。每个页面可以有自己的实体、值对象和关联关系，以及与页面相关的业务逻辑。

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。组件可以是页面级别的，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。功能可以是用户操作的具体功能模块，例如登录、注册、购物车等。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风险

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风险。以下是一些需要注意的风险：

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户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。

为了减轻这些风险，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风险。

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风险或全局命名冲突。

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，以下是一些常见的方式：

1. 网站结构优化：优化网站的结构，确保每个页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页面的内容。

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）等技术手段来减少页面加载时间。

3. 页面渲染优化：确保搜索引擎可以正常渲染和索引使用 JavaScript 技术构建的单页面应用（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页面内容。

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长的 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，使页面之间相互连接。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）来标识网页内容，帮助搜索引擎更好地理解和展示网页信息。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，提供网站的结构和页面信息，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页面可以被爬取和索引，哪些页面不可访问。

9. HTTPS 加密：使用 HTTPS 协议来加密网站通信，确保数据安全和用户隐私，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户体验，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 优化的一些常见技术层面的策略，通过综合运用这些策略，可以提升网站的搜索引擎可见性和排名。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户体验。

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户的交互。它使用 WebView 进行页面渲染，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，负责处理用户交互、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户体验。
- 防止阻塞：逻辑线程的运行不会阻塞渲染线程，避免了长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户端进行通信和交互。

逻辑线程可以发送请求给微信客户端，然后客户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户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，两者通过微信客户端进行通信和交互，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。

```
XML
<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。

```
JavaScript
var script = document.createElement('script');
script.src = 'https://cdn.example.com/script.js';
script.onerror = function() {// 主要资源加载失败，切换到备用资源路径
  script.src = 'https://backup.example.com/script.js';
};
document.head.appendChild(script);
```

在加载 JavaScript 脚本时，可以通过监听 `onerror` 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页应用的正常运行，提高用户体验，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"`：

```XML
<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：

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

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。以下是两种常见的实现方式：

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、touchmove、touchend 等）和滚动事件（如 scroll）来监测用户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户的下拉或上拉手势，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页面底部，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：

```
XML
<!DOCTYPE html><html<head<title 上拉加载和下拉刷新示例</title<style
    /* 用于展示加载和刷新状态的样式 */
    .loading {text-align: center;padding: 10px;background-color:
#f1f1f1;}

    .refresh {text-align: center;padding: 10px;background-color:
#f1f1f1;}
  </style</head<body<div id="content"<!-- 内容区域 --></div<div
id="loading" class="loading"
    加载中...</div<div id="refresh" class="refresh"
    下拉刷新</div<script src="your_script.js"</script</body</html
```

JavaScript 代码（your_script.js）：

```
JavaScript
// 获取相关元素var content = document.getElementById('content');
var loading = document.getElementById('loading');
var refresh = document.getElementById('refresh');

var isRefreshing = false;
var isLoading = false;

// 监听触摸事件var startY = 0;
var moveY = 0;
content.addEventListener('touchstart', function(event) {
  startY = event.touches[0].pageY;
});
content.addEventListener('touchmove', function(event) {
  moveY = event.touches[0].pageY;
// 下拉刷新if (moveY - startY > 100 && !isRefreshing) {
```

```
    refresh.innerHTML = '释放刷新';
  }
// 上拉加载var scrollTop = content.scrollTop;var scrollHeight =
content.scrollHeight;var offsetHeight = content.offsetHeight;if
(scrollTop + offsetHeight >= scrollHeight && !isLoading) {
    loading.style.display = 'block';
  }
});
content.addEventListener('touchend', function(event) {// 下拉刷新
if (moveY - startY > 100 && !isRefreshing) {
    refresh.innerHTML = '刷新中...';
    simulateRefresh();
  }
// 上拉加载var scrollTop = content.scrollTop;var scrollHeight =
content.scrollHeight;var offsetHeight = content.offsetHeight;if
(scrollTop + offsetHeight >= scrollHeight && !isLoading) {
    loading.style.display = 'block';
    simulateLoad();
  }
// 重置状态
  startY = 0;
  moveY = 0;
});

// 模拟刷新function simulateRefresh() {
  isRefreshing = true;
  setTimeout(function() {// 刷新完成后的操作
    refresh.innerHTML = '刷新成功';
    isRefreshing = false;
  }, 2000);
}

// 模拟加载function simulateLoad() {
  isLoading = true;
  setTimeout(function() {// 加载完成后的操作
    loading.style.display = 'none';
    isLoading = false;
  }, 2000);
}
```

上面的代码使用了 `touchstart`、`touchmove` 和 `touchend` 事件来监测用户的手势操作，实现了下拉刷新和上拉加载的功能。通过修改 `refresh` 和 `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。

```
JavaScript
function isInViewport(element) {const rect =
element.getBoundingClientRect();return (
    rect.top >= 0 &&
    rect.left >= 0 &&
    rect.bottom <= (window.innerHeight ||
document.documentElement.clientHeight) &&
    rect.right <= (window.innerWidth ||
document.documentElement.clientWidth)
  );
}

// Example usageconst element = document.getElementById('my-
element');
if (isInViewport(element)) {console.log('Element is in viewport');
} else {console.log('Element is not in viewport');
}
```

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，当元素的可见性发生变化时会调用该回调函数。

```
JavaScript
function callback(entries, observer) {
  entries.forEach(entry => {if (entry.isIntersecting)
{console.log('Element is in viewport');
    } else {console.log('Element is not in viewport');
    }
  });
}

const observer = new IntersectionObserver(callback);
```

```
const element = document.getElementById('my-element');
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，来实现用户交互操作。例如，当用户点击座位时，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：在用户选票的过程中，需要统计已选座位的数量和位置，并将选票信息展示给用户。可以通过遍历座位图案数组来实现。

4. 添加检查功能：为了防止用户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：当用户选好座位后，需要确认并支付，可以通过弹出确认对话框来实现，并将用户的选票信息发送至后台进行处理。

代码实现如下

```
XML
- <!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
  // 获取画布和按钮元素
  var canvas = document.getElementById('canvas');var btnPay =
document.getElementById('btnPay');
// 获取画布上下文和座位数组
  var ctx = canvas.getContext('2d');var seats = [];
// 绘制座位
  function drawSeat(x, y, state) {switch (state) {case 0:
        ctx.fillStyle = '#ccc'; // 可选座位
        break;case 1:
        ctx.fillStyle = '#f00'; // 已售座位
        break;case 2:
        ctx.fillStyle = '#0f0'; // 已选座位
```

```
        break;default:
        ctx.fillStyle = '#000'; // 其他座位
        break;
    }
    ctx.fillRect(x, y, 30, 30);
  }
// 初始化座位数组
  function initSeat() {for (var i = 0; i < 10; i++) {
      seats[i] = [];for (var j = 0; j < 10; j++) {
        seats[i][j] = 0; // 初始状态为可选
        drawSeat(i * 4050, j * 4050, 0); // 绘制座位
      }
    }
  }
// 统计已选座位数量和位置
  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) {
          selectedSeats.push([i, j]);
          count++;
        }
      }
    }return [count, selectedSeats];
  }
// 更新座位状态和颜色
  function updateSeat(x, y) {if (seats[x][y] == 0) {
      seats[x][y] = 2; // 更改为已选状态
    } else if (seats[x][y] == 2) {
      seats[x][y] = 0; // 更改为可选状态
    }
    drawSeat(x * 4050, y * 4050, seats[x][y]); // 更新颜色
  }
// 检查座位状态是否可选
  function checkSeat(x, y) {if (seats[x][y] == 1) {
      alert('该座位已售出，请选择其他座位！');return false;
    } else if (seats[x][y] == 2) {
      alert('该座位已被选中，请选择其他座位！');return false;
    }return true;
  }
// 点击事件处理函数
  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)) {
        updateSeat(x, y);var count = countSelectedSeats()[0];if
(count > 0) {
          btnPay.innerHTML = '确认并支付（已选 'count + ' 座位）';
        } else {
          btnPay.innerHTML = '确认并支付';
        }
      }
    }
  }
// 确认并支付按钮点击事件处理函数
  function handlePay() {var selectedSeats =
countSelectedSeats()[1];if (selectedSeats.length == 0) {
      alert('请选择座位！');return;
    }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
      alert('支付成功！请前往指定影院取票！');
      initSeat(); // 重新初始化座位
      btnPay.innerHTML = '确认并支付';
    }
  }
// 初始化座位
  initSeat();
// 绑定点击事件和确认并支付按钮点击事件
  canvas.addEventListener('click', handleClick);
  btnPay.addEventListener('click', handlePay);

</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。以下是一种常见的方法：

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。

```
Bash
npm install less less-loader antd
```

1. 在 Webpack 配置文件中添加对 Less 的支持。

```
JavaScript
module.exports = {// ...其他配置 module: {
    rules: [// ...其他规则
      {
        test: /\.less$/, // 匹配 Less 文件
        use: ['style-loader', // 将 CSS 插入到页面中'css-loader', //
解析 CSS 文件'less-loader' // 将 Less 转换为 CSS
        ]
      }
    ]
  },
// ...其他配置
};
```

1. 在入口文件中引入 Ant Design 的样式文件。

```
JavaScript
// 入口文件 index.jsimport React from 'react';
import ReactDOM from 'react-dom';
import 'antd/dist/antd.css'; // 引入 Ant Design 的样式文件import App
from './App';
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 `App.js` 中使用 Ant Design 的 Button 组件：

```
JavaScript
// App.jsimport React from 'react';
import { Button } from 'antd';

function App() {return (<div
      <h1Hello, React!</h1
      <Button type="primary"Click me</Button
    </div
  );
}

export default App;
```

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

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。

```
Bash
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。

```
JavaScript
module.exports = {// ...其他配置module: {
    rules: [// ...其他规则
      {
        test: /\.less$/, // 匹配Less 文件
        exclude: /\.module\.less$/, // 排除CSS Module 的Less 文件
        use: ['style-loader', // 将CSS 插入到页面中'css-loader', //
解析CSS 文件'less-loader' // 将Less 转换为CSS
        ]
      },
      {
        test: /\.module\.less$/, // 匹配CSS Module 的Less 文件
        use: ['style-loader', // 将CSS 插入到页面中
          {
            loader: 'css-loader', // 解析CSS 文件
            options: {
              modules: true // 启用CSS Module
            }
          },'less-loader' // 将Less 转换为CSS
        ]
      }
    ]
  },
// ...其他配置
};
```

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。

```
Plain Text
/* styles.module.less */.container {background-color: red;
}
```

```
Plain Text
/* styles.less */.text {color: blue;
}
```

在 React 组件中使用这些样式：

```
JavaScript
import React from 'react';
import styles from './styles.module.less'; // 导入 CSS Module 的样式
import './styles.less'; // 导入普通的 Less 样式function App()
{return (<div className={styles.container}
      <h1 className="text"Hello, React!</h1
    </div
  );
}

export default 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。

```
Bash
npm install antd babel-plugin-import --save
```

1. 在.babelrc 文件中配置 babel 插件。

```
JSON
{
  "plugins": [
```

```
    ["import",
      {
        "libraryName": "antd",
        "style": "css"
      }
    ]
  ]
}
```

1. 在 Webpack 配置文件中添加对 Less 和 CSS 的支持。

```
JavaScript
module.exports = {// ...其他配置module: {
    rules: [// ...其他规则
      {
        test: /\.less$/, // 匹配Less 文件
        exclude: /node_modules/,
        use: ['style-loader', // 将CSS 插入到页面中'css-loader', //
解析CSS 文件'less-loader' // 将Less 转换为CSS
        ]
      }
    ]
  },
// ...其他配置
};
```

1. 在你的组件中引入 antd 组件。

```
JavaScript
import React from 'react';
import { Button } from 'antd';

function App() {return <Button type="primary"Hello, Antd!</Button;
}

export default 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 `fs` 模块来读取文件内容并进行行数统计。以下是一个简单的实现示例：

```JavaScript
const fs = require('fs');
const path = require('path');

function countLinesInDirectory(dirPath, fileExtension) {let
totalLines = 0;
function countLinesInFile(filePath) {const content =
fs.readFileSync(filePath, 'utf-8');const lines =
content.split('\n');
    totalLines += lines.length;
  }
function processDirectory(directoryPath) {const files =
fs.readdirSync(directoryPath);
    files.forEach((file) => {const filePath =
path.join(directoryPath, file);const stats =
fs.statSync(filePath);
if (stats.isFile() && path.extname(file) === fileExtension) {
        countLinesInFile(filePath);
      } else if (stats.isDirectory()) {
        processDirectory(filePath);
      }
    });
  }
  processDirectory(dirPath);
return totalLines;
}

// 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const [_,
__, dirPath, fileExtension] = process.argv;

const linesCount = countLinesInDirectory(dirPath, fileExtension);
console.log(`Total lines of ${fileExtension} files in ${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：

```Plain Text
node line-counter.js /path/to/directory .js
```

其中`/path/to/directory`是你要统计的目录路径，`.js`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

`sideEffects` 是 `package.json` 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。

`sideEffects` 字段可以有以下几种取值：

- `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.json` 中明确指定 `sideEffects` 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，`["./src/*.js", "!./src/utils/*.js"]`表示 `src` 目录下的所有 `.js` 文件都具有副作用，但是 `src/utils` 目录下的 `.js` 文件没有副作用。

使用 `sideEffects` 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。

注意：在使用 `sideEffects` 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？

`sideEffects` 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 `sideEffects` 字段。如果一个模块具有 `sideEffects` 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 `sideEffects` 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 `sideEffects` 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. `src`：指定要引入的外部 JavaScript 文件的 URL。例如：`<script src="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. `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 HTML 文档，不会阻塞其他资源的加载。例如：`<script src="script.js" async></script>`。
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 src="script.js" defer></script>`。
4. `type`：指定脚本语言的 MIME 类型。通常是 `text/javascript` 或者 `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`：指定外部脚本文件的字符编码。例如：`<script src="script.js" charset="UTF-8"></script>`。
6. `integrity`：用于指定外部脚本文件的 Subresource Integrity（SRI）。SRI 可以确保浏览器在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：`<script src="script.js" integrity="sha256-qznLcsROx4GACP2dm0UCKCzCG+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。参见 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-src 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（单页应用）通常会使用 hash 路由的方式来实现页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`。

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，只需要在页面中添加一个监听器来监听 `hashchange` 事件，然后根据不同的 hash 值加载对应的页面内容。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页面，之后的页面切换完全由前端控制。

3. 防止页面刷新：使用 hash 路由可以防止页面的完全刷新。因为 hash 路由只改变 URL 的片段标识符，不会引起整个页面的重新加载，所以用户在不同页面之间切换时，不会丢失当前页面的状态和数据。

4. 前进后退支持：由于 hash 路由不会引起页面的刷新，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，从而实现页面的导航和页面状态的管理。

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。所有的路由和页面切换逻辑都由前端控制，服务器只提供一个初始页面。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，提升用户体验。

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页面的情况下改变 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：

```
Plain Text
import React from 'react';import { BrowserRouter, HashRouter }
from 'react-router-dom';function App() {return (// 使用
BrowserRouter 或 HashRouter 包裹你的应用
    <BrowserRouter>
      {/* 在这里编写你的应用内容 */}
    </BrowserRouter>
  );
}
export default App;
```

当使用函数组件时，可以使用 `useEffect` 钩子函数来监听路由变化。下面是修改后的示例代码：

```
JavaScript
import React, { useEffect } from 'react';
import { withRouter } from 'react-router-dom';

function MyComponent(props) {
  useEffect(() => {const 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);
    };
// 在组件挂载后，添加路由变化的监听器 const unlisten =
props.history.listen(handleRouteChange);
// 在组件卸载前，移除监听器 return () => {
      unlisten();
```

```
    };
  }, [props.history]);
return (<div
      {/* 在这里编写组件的内容 */}</div
  );
}

// 使用 withRouter 高阶组件将路由信息传递给组件export default
withRouter(MyComponent);
```

在上面的代码中，我们使用了 `useEffect` 钩子函数来添加路由变化的监听器。在 `useEffect` 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 `useEffect` 返回的清理函数中调用了 `unlisten` 函数，以确保在组件卸载时移除监听器。

需要注意的是，由于 `useEffect` 的依赖数组中包含了 `props.history`，所以每当 `props.history` 发生变化时（即路由发生变化时），`useEffect` 的回调函数会被调用，从而更新路由变化的监听器。

总结起来，通过使用 `useEffect` 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

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. 用户访问系统 1 的受保护资源，系统 1 发现用户未登录，跳转至 sso 认证中心，并将自己的地址作为参数
2. sso 认证中心发现用户未登录，将用户引导至登录页面
3. 用户输入用户名密码提交登录申请
4. sso 认证中心校验用户信息，创建用户与 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户的会话，称为局部会话，返回受保护资源
9. 用户访问系统 2 的受保护资源
10. 系统 2 发现用户未登录，跳转至 sso 认证中心，并将自己的地址作为参数
11. sso 认证中心发现用户已登录，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户的局部会话，返回受保护资源

35. web 网页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`。

`MutationObserver` 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:

```
JavaScript
// 目标节点const targetNode = document.body;

// 创建 MutationObserver 实例const observer = new
MutationObserver((mutationsList) => {for (let mutation of
mutationsList) {// 检查是否有子节点被删除if
(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);
        }
    }
});

// 配置 MutationObserverconst config = { childList: true, subtree:
true };

// 开始观察目标节点
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页面上。

36. 用户访问页面白屏了， 原因是啥， 如何排查？【热度: 609】

用户访问页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：用户的网络连接可能存在问题，无法正确加载页面内容。可以要求用户检查网络连接，或者自己尝试在不同网络环境下测试页面的加载情况。

2. 服务端问题：服务器未正确响应用户请求，导致页面无法加载。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：页面的前端代码可能存在错误或异常，导致页面无法正常渲染。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，可以尝试将页面的 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，导致页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页面的加载情况，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：页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，可能导致页面白屏。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页面或资源，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页面和资源。

7. 其他可能原因：页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，并且可以与用户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：

```JavaScript
function deepEqual(obj1, obj2) {// 检查类型是否相同if (typeof obj1 !== typeof obj2) {return false;
  }
  // 检查是否是对象或数组if (typeof obj1 === 'object' && obj1 !== null && obj2 !== null) {// 检查对象或数组长度是否相同if (Object.keys(obj1).length !== Object.keys(obj2).length) {return false;
    }
  for (let key in obj1) {// 递归比较每个属性的值if (!deepEqual(obj1[key], obj2[key])) {return false;
      }
    }
    return true;
  }
  // 比较基本类型的值return obj1 === obj2;
}
```

使用示例：

```JavaScript
const obj1 = {
  name: 'John',
  age: 30,
  address: {
    street: '123 Main St',
    city: 'New York'
  }
};

const obj2 = {
  name: 'John',
  age: 30,
```

```
  address: {
    street: '123 Main St',
    city: 'New York'
  }
};

const obj3 = {
  name: 'Jane',
  age: 25,
  address: {
    street: '456 Park Ave',
    city: 'Los Angeles'
  }
};

console.log(deepEqual(obj1, obj2)); //
trueconsole.log(deepEqual(obj1, obj3)); // false
```

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. 用户交互：数据驱动视图通常与用户进行交互，用户可以通过界面操作改变数据，从而触发视图的更新。用户交互可以通过表单、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），可以实时更新页面内容，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，保持页面的响应性。

以下是一个使用 Web Workers 的简单示例：

```
JavaScript
// 主线程代码const worker = new Worker('worker.js'); // 创建一个新的 Web Worker
worker.postMessage({ start: 0, end: 1000000 }); // 向Web Worker 发送消息
worker.onmessage = function(event) {const result = event.data;console.log('任务完成：', result);
};

// worker.js - Web Worker 代码
onmessage = function(event) {const start = event.data.start;const end = event.data.end;let sum = 0;for (let i = start; i <= end; i++) {
    sum += i;
  }
  postMessage(sum); // 向主线程发送消息
};
```

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 `requestAnimationFrame` 来实现任务分割是一种常见的方式，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 `requestAnimationFrame` 来分割任务的简单例子：

```
JavaScript
// 假设有一个包含大量元素的数组const bigArray = Array.from({ length: 1000000 }, (_, i) => i + 1);

// 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function processChunk(chunk) {return chunk.map(num => num * num);
}

// 分割任务并使用requestAnimationFrameconst chunkSize = 1000; // 每个小块的大小let index = 0;
```

```
function processArrayWithRAF() {function processChunkWithRAF()
{const chunk = bigArray.slice(index, index + chunkSize); // 从大数
组中取出一个小块const result = processChunk(chunk); // 处理小块任务
console.log('处理完成: ', result);
    index += chunkSize;
if (index < bigArray.length) {
      requestAnimationFrame(processChunkWithRAF); // 继续处理下一个
小块
    }
  }
  requestAnimationFrame(processChunkWithRAF); // 开始处理大数组
}
processArrayWithRAF();
```

在这个例子中，我们使用 `requestAnimationFrame` 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:

```
JavaScript
  const $result = document.getElementById("result");

// 假设有一个包含大量元素的数组const bigArray = Array.from({ length:
1000000 }, (_, i) => i + 1);

// 定义一个处理函数，对数组中的每个元素执行一次function
processChunk(chunk) {return
chunk: ${chunk}
;
}

// 动态调整 chunkSize 的优化方式let chunkSize = 1000; // 初始的
chunkSizelet index = 0;

function processArrayWithDynamicChunkSize() {function
processChunkWithRAF() {let startTime = performance.now(); // 记录结
束时间for (let i = 0; i < chunkSize; i++) {if (index <
bigArray.length) {const result = processChunk(bigArray[index]); //
```

```
对每个元素执行处理函数
        $result.innerText = result;
        index++;
      }
    }let endTime = performance.now();let timeTaken = endTime -
startTime; // 计算处理时间// 根据处理时间动态调整 chunkSizeif
(timeTaken > 16) { // 如果处理时间超过一帧的时间（16 毫秒），则减小
chunkSize
      chunkSize = Math.floor(chunkSize * 0.9); // 减小 10%
    } else if (timeTaken < 16) { // 如果处理时间远小于一帧的时间（8
毫秒），则增加 chunkSize
      chunkSize = Math.floor(chunkSize * 1.1); // 增加 10%
    }
if (index < bigArray.length) {
      requestAnimationFrame(processChunkWithRAF); // 继续处理下一个
小块
    }
  }
  requestAnimationFrame(processChunkWithRAF); // 开始处理大数组
}
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>` 里会在页面加载之前执行 JavaScript 代码，而放在 `<body>` 里会在页面加载后执行。
2. 页面渲染：如果 JavaScript 代码影响了页面的布局或样式，放在 `<head>` 里可能会导致页面渲染延迟，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页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 `<head>` 和 `<body>` 中放置 JavaScript 代码：

```
XML
<!DOCTYPE html><html<head<script
    console.log("这是在 head 中执行的 JavaScript 代码。
");</script</head<body<script
    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 `<head>` 和 `<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】

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风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 `.eslintrc` 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风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，通常用于处理长列表的显示问题。在传统的滚动加载中，当面对成千上万项的长列表时，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户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. 当用户滚动时，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：

```
JavaScript
class VirtualScroll {
  constructor(container, itemHeight, totalItems, renderCallback)
{this.container = container; // 容器元素this.itemHeight =
itemHeight; // 每个项的高度this.totalItems = totalItems; // 总列表
项数this.renderCallback = renderCallback; // 渲染每一项的回调函数
this.viewportHeight = container.clientHeight; // 视口高度
this.bufferSize = Math.ceil(this.viewportHeight / itemHeight) * 3;
// 缓冲大小this.renderedItems = []; // 已渲染项的数组
this.startIndex = 0; // 当前渲染的开始索引this.endIndex =
```

```
this.bufferSize; // 当前渲染的结束索引
    container.addEventListener("scroll", () =>
this.onScroll());this.update();
  }
  onScroll() {const scrollTop = this.container.scrollTop;const
newStartIndex = Math.floor(scrollTop / this.itemHeight) -
this.bufferSize / 2;const newEndIndex = newStartIndex +
this.bufferSize;
if (newStartIndex !== this.startIndex || newEndIndex !==
this.endIndex) {this.startIndex = Math.max(0,
newStartIndex);this.endIndex = Math.min(this.totalItems,
newEndIndex);this.update();
    }
  }
  update() {// 清空已有内容this.container.innerHTML = "";
// 计算并设置容器的总高度const totalHeight = this.totalItems *
this.itemHeight;this.container.style.height =
${totalHeight}px
;
// 渲染视口内的项const fragment =
document.createDocumentFragment();for (let i = this.startIndex; i
< this.endIndex; i++) {const item = this.renderCallback(i);
      item.style.top = `${i * this.itemHeight}px`;
      fragment.appendChild(item);
    }this.container.appendChild(fragment);
  }
}

// 创建一个列表项的函数function createItem(index) {const item =
document.createElement("div");
  item.className = "list-item";
  item.innerText =
Item ${index}
;
  item.style.position = "absolute";
  item.style.width = "100%";return item;
}

// 初始化虚拟滚动const container = document.querySelector(".scroll-
container"); // 容器元素需要预先在HTML中定义const virtualScroll =
new 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 `startIndex` 和 `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：

```
JavaScript
class VirtualScroll {
  constructor(container, itemHeight, totalItems, renderItem)
{this.container = container;this.itemHeight =
itemHeight;this.totalItems = totalItems;this.renderItem =
renderItem;
this.observer = new
IntersectionObserver(this.onIntersection.bind(this), {
      root: this.container,
      threshold: 1.0,
    });
this.items = new Map();
this.init();
  }
  init() {// 填充初始屏幕的元素for (let i = 0; i < this.totalItems;
i++) {const placeholder =
this.createPlaceholder(i);this.container.appendChild(placeholder);
this.observer.observe(placeholder);
    }
  }
  createPlaceholder(index) {const placeholder =
document.createElement("div");
    placeholder.style.height = `${this.itemHeight}px`;
    placeholder.style.width = "100%";
    placeholder.dataset.index = index; // store indexreturn
placeholder;
  }
  onIntersection(entries) {
    entries.forEach((entry) => {const index =
entry.target.dataset.index;if (entry.isIntersecting) {const
rendered =
```

```
this.renderItem(index);this.container.replaceChild(rendered,
entry.target);this.items.set(index, rendered);
      } else if (this.items.has(index)) {const placeholder =
this.createPlaceholder(index);this.container.replaceChild(placehol
der,
this.items.get(index));this.observer.observe(placeholder);this.ite
ms.delete(index);
      }
    });
  }
}

// Render item functionfunction renderItem(index) {const item =
document.createElement("div");
  item.classList.add("item");
  item.textContent =
Item ${index}
;
  item.dataset.index = index;
  item.style.height = "30px"; // Same as your itemHeight in
VirtualScrollreturn item;
}

// Example usage:const container =
document.getElementById("scroll-container"); // This should be a
predefined element in your HTMLconst itemHeight = 30; // Height of
each itemconst itemCount = 1000; // Total number of items you
haveconst virtualScroll = new VirtualScroll(container, itemHeight,
itemCount, renderItem);
```

在这里我们创建了一个 `VirtualScroll` 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` 注册了这些占位元素。

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，当用户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>`、`<Route>`、和 `<Link>` 等组件，这些都是专门为了在 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:
   - 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:
   - 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:
   - 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:
   - 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:
   - 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。
   - 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 History API
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
	路由守卫、懒加载、嵌	基本的历史

点击图片可查看完整电子表格

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	可以设置上	只能设置左

点击图片可查看完整电子表格

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 `display` 属性来更

改它们的行为。例如，`display: inline;`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 requestIdleCallback 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` 特别适合那些不直接关联用户交互及响应的任务，这些任务可以延后执行而不会明显影响用户体验。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。

```JavaScript
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。

```JavaScript
function myNonCriticalFunction(deadline) {while ((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
  }
// 如果还有未完成的工作，可以请求下一次空闲周期if (someCondition()) {
    requestIdleCallback(myNonCriticalFunction);
  }
}
```

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。
- 执行低优先级任务时，不应该太过频繁或执行时间太长，以免影响页面性能。
- 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长执行结束。

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，同时保持用户交互的流畅度。

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 `DocumentFragment` 中节点集合进行分组，这个集合可以一次性插入到 `DOM` 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` 的常见场景：

- 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` 在不触发页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

```JavaScript
// 创建 DocumentFragmentvar fragment =
document.createDocumentFragment();

// 创建多个节点或元素var div = document.createElement("div");
var p = document.createElement("p");

// 将节点添加到 DocumentFragment 上
fragment.appendChild(div);
fragment.appendChild(p);

// 一次性将 DocumentFragment 添加到 DOM 的某个部分var body =
document.querySelector("body");
body.appendChild(fragment);
```

```
// 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 而不影响页面的渲染性能，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 和 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 和 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 git fetch 更加「谨慎」，它只下载变更到本地，不做任何合并操作。

49. 前端如何做 页面主题色切换【热度: 538】

页面主题色切换通常涉及到修改网页中的颜色方案，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。以下是实施页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：

```CSS
:root {--primary-color: #5b88bd; /* 明亮主题色 */
  --text-color: #000; /* 明亮主题文本颜色 */
}[data-theme="dark"] {--primary-color: #1e2a34; /* 暗黑主题色 */
  --text-color: #ccc; /* 暗黑主题文本颜色 */
}
```

1. 应用自定义属性到 CSS 规则中：

```CSS
body {background-color: var(--primary-color);color: var(--text-color);
}
```

1. 使用 JavaScript 动态切换主题：

```JavaScript
function toggleTheme() {const root = document.documentElement;if (root.dataset.theme === "dark") {
    root.dataset.theme = "light";
  } else {
    root.dataset.theme = "dark";
  }
}
```

使用 CSS 类切换

1. 为每个主题创建不同的 CSS 类：

```CSS
.light-theme {--primary-color: #5b88bd;--text-color: #000;
}.dark-theme {--primary-color: #1e2a34;--text-color: #ccc;
}
```

1. 手动切换 CSS 类：

```
JavaScript
function toggleTheme() {const bodyClass =
document.body.classList;if (bodyClass.contains("dark-theme")) {
    bodyClass.replace("dark-theme", "light-theme");
  } else {
    bodyClass.replace("light-theme", "dark-theme");
  }
}
```

使用 LocalStorage 记录用户主题偏好

```
JavaScript
// 当用户切换主题时function saveThemePreference() {
  localStorage.setItem("theme",
document.body.classList.contains("dark-theme") ? "dark" :
"light");
}

// 页面加载时应用用户偏好function applyThemePreference() {const
preferredTheme = localStorage.getItem("theme");
if (preferredTheme === "dark") {document.body.classList.add("dark-
theme");
  } else {document.body.classList.remove("dark-theme");
  }
}
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户的系统设置应用暗黑模式或明亮模式，而无须 JavaScript：

```
CSS
@media (prefers-color-scheme: dark) {:root {--primary-color:
#1e2a34; /* 暗黑主题色 */
    --text-color: #ccc; /* 暗黑主题文本颜色 */
  }
}

@media (prefers-color-scheme: light) {:root {--primary-color:
#5b88bd; /* 明亮主题色 */
    --text-color: #000; /* 明亮主题文本颜色 */
```

```
  }
}
```

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

50. 前端视角 - 如何保证系统稳定性【热度: 566】

前端视角来做稳定性， 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. 页面崩溃报警
5. E2E 定时全量跑用例

51. 如何统计长任务时间、长任务执行次数【热度: 489】

在 JavaScript 中，可以使用 Performance API 中的 PerformanceObserver 来监视和统计长任务（Long Task）。长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，影响页面的交互性和流畅性。

使用 PerformanceObserver 监听长任务

```
JavaScript
// 创建一个性能观察者实例来订阅长任务let observer = new
PerformanceObserver((list) => {for (const entry of
list.getEntries()) {console.log("Long Task
detected:");console.log(
Task Start Time: ${entry.startTime}, Duration: ${entry.duration}
);
  }
});

// 开始观察长任务
```

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

// 启动长任务统计数据的变量let longTaskCount = 0;
let totalLongTaskTime = 0;

// 更新之前的性能观察者实例，以增加统计逻辑
observer = new PerformanceObserver((list) => {
  list.getEntries().forEach((entry) => {
    longTaskCount++; // 统计长任务次数
    totalLongTaskTime += entry.duration; // 累加长任务总耗时// 可以在这里添加其他逻辑，比如记录长任务发生的具体时间等
  });
});

// 再次开始观察长任务
observer.observe({ entryTypes: ["longtask"] });
```

在上面的代码中，我们创建了一个 `PerformanceObserver` 对象来订阅长任务。每当检测到长任务时，它会向回调函数传递一个包含长任务性能条目的列表。在这个回调中，我们可以统计长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户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：

```
JavaScript
// 停止观察能力
observer.disconnect();

// 统计数据输出console.log(Total number of long tasks:
${longTaskCount});
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，并根据长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，用于解析当前页面的 cookie 并将它们存储到一个对象中：

```
JavaScript
function parseCookies() {// 创建一个空对象来存储解析后的 cookievar
cookiesObj = {};
// 获取 cookie 字符串，然后分割每个键值对var cookies =
document.cookie.split(";");
// 遍历每个键值对
  cookies.forEach(function (cookie) {// 去除键值对前后的空格var
cleanCookie = cookie.trim();// 找到键和值之间的等号位置var
separatorIndex = cleanCookie.indexOf("=");
// 如果找不到等号，则不是有效的键值对，跳过当前循环if (separatorIndex
=== -1) return;
// 获取键名var key = cleanCookie.substring(0, separatorIndex);//
获取值var value = cleanCookie.substring(separatorIndex + 1);
```

```
// 解码因为 cookie 键和值是编码过的
    key = decodeURIComponent(key);
    value = decodeURIComponent(value);
// 将解析后的值存储到对象中
    cookiesObj[key] = value;
  });
// 返回解析后的 cookie 对象return cookiesObj;
}

// 使用示例var cookies = parseCookies();
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，其中包含了当前页面的所有 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：

```
XML
<template<button class="btn"Click Me</button</template<style
scoped
  .btn {background-color: blue;}
</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：

```
XML
<button class="btn" data-v-f3f3eg9Click Me</button
```

CSS 则会被转换成：

```
CSS
.btn[data-v-f3f3eg9] {background-color: blue;
}
```

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见方式：

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。常见的库有 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

将组件或部分页面放在 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：

```
JavaScript
function factorial(n) {if (n === 1) return 1;return n *
factorial(n - 1);
}
```

可以重写为循环形式：

```
JavaScript
function factorial(n) {let result = 1;for (let i = 2; i <= n; i++)
{
    result *= i;
  }return result;
}
```

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。

```
JavaScript
function trampoline(fn) {return function (...args) {let result =
fn.apply(this, args);
while (typeof result === "function") {
      result = result();
    }
return result;
  };
}
```

然后，将原始递归函数改写为每次递归调用返回一个函数：

```
JavaScript
function recursiveFunction(args) {if (baseCase) {return
finalValue;
  } else {return function () {return recursiveFunction(newArgs);
    };
  }
}

const 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`、`setImmediate` 或者 `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：

```
JavaScript
function recursiveAsyncFunction(i) {if (i < 0) return
Promise.resolve();console.log("Recursion ", i);return new
Promise((resolve) => {
```

```
    setImmediate(() => {
      resolve(recursiveAsyncFunction(i - 1));
    });
  });
}
```

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页面，哪些不应该爬取。例如：

```
HTTP
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、登录页面等，使用验证码（CAPTCHA）可以防止自动化脚本或机器人执行操作。

3. 检查用户代理字符串

服务器可以根据请求的用户代理（User-Agent）字符串来决定是否屏蔽某些爬虫。但用户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、访问页数、访问时长，并与正常用户的行为进行对比，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）动态地插入到页面中，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页面抓取。

10. API 限流

对 API 使用率进行限制，比如基于用户、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 `src` 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。

- **import**as PackageName from "<package-name>";

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

1. tsconfig.json 配置：

确保你的 `tsconfig.json` 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 `compilerOptions` 中的 `"types"` 和 `"typeRoots"` 属性没有配置错误。

2. 使用全局包：

现在你可以在你的 `src` 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.json`）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. 不同标签页或窗口间的 【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，实现不同标签页或窗口间的主动推送消息机制，可以使用以下客户端技术：

> 作者备注:
> 这里要注意一下， 这里讨论的不是跨页签通信，而是跨页签主动推送信息。如果仅仅是跨页签通信， 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

`BroadcastChannel API` 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。这意味着它可以在同一网站的多个标签页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

```
JavaScript
// 在任何一个 tab 或 iframe 中创建一个广播频道const channel = new
BroadcastChannel("my-channel-name");

// 发送一个消息到频道
channel.postMessage("Hello from a tab!");
```

监听消息

```
JavaScript
// 监听这个频道的消息
channel.addEventListener("message", function (event) {if
(event.data === "Hello from a tab!") {console.log("Message
received: ", event.data);
  }
});
```

实现频道消息通信

假设你有两个标签页，并且你想更新每个标签页来显示另一个标签页中发生的事情，比如用户数量计数器：

```
JavaScript
// 在第一个标签页中
self.addEventListener("load", () => {const channel = new
BroadcastChannel("visitor-channel");let visitorCount = 0;
// 定时发送随机的用户活动消息
  setInterval(function () {
    visitorCount++;
    channel.postMessage(
Visitor count increased to: ${visitorCount}
);
```

```
  }, 5000);
});

// 在另一个标签页中
self.addEventListener("load", () => {const channel = new
BroadcastChannel("visitor-channel");
// 监听消息来更新用户数量
  channel.addEventListener("message", function (event) {if
(event.data.startsWith("Visitor count")) {// 用接收到的用户数量更新
显示
    updateVisitorCountDisplay(event.data);
    }
  });
// 这个方法将设置标签页上的用户计数显示function
updateVisitorCountDisplay(message) {// 这里写用于更新显示的代码
console.log(message);
  }
});
```

在这个例子中，一个标签页通过定期发送新的消息来模拟用户活动的增加，这个消息在所有监听该频道的上下文中传递。另一个或多个标签页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，各个标签页可以通过 `clients.matchAll()` 方法找到所有其他客户端（如打开的标签页），然后使用 `postMessage` 发送消息。

这个方法相比 `BroadcastChannel` 更加灵活，因为 Service Workers 可以通过 `Focus` 和 `Navigate` 事件来控制页面的焦点和导航等。

`ServiceWorkers` 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行。当你用 `ServiceWorkers` 进行页面间的通信，你可以利用它们来推送消息到打开的 `Clients`（如浏览器标签页）。

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

户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. 在主页面注册 ServiceWorker

在主页面（index.html）通过 JavaScript 注册这个 ServiceWorker 文件。

```
JavaScript
// index.htmlif ("serviceWorker" in navigator) {
  navigator.serviceWorker
    .register("/sw.js")
    .then((registration) => {console.log("Service Worker registered with scope:", registration.scope);
    })
    .catch((error) => {console.log("Service Worker registration failed:", error);
    });
}
```

3. 监听 `message` 事件

在主页面使用 `navigator.serviceWorker.controller` 来检查是否已经有 ServiceWorker 主动控制。

```
JavaScript
if (navigator.serviceWorker.controller) {// Post a message to the ServiceWorker
  navigator.serviceWorker.controller.postMessage("This is from main page");
}
```

4. 从其他 Tab 推送消息

在其他 Tab 上，一旦 ServiceWorker 被该页面控制后，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:

```
JavaScript
// 创建一个 SharedWorker，并指定要加载的脚本var myWorker = new
SharedWorker("worker.js");
// 开启端口通信
myWorker.port.start();
```

1. 端口通信: 使用端口接收和发送消息

```
JavaScript
// 发送数据给 worker
myWorker.port.postMessage({ command: "start", data: [1, 2, 3] });

// 监听来自 worker 的消息
myWorker.port.onmessage = function (event) {if (event.data)
{console.log("Result from worker:", event.data);
  }
};
```

1. 实现 worker 逻辑:

在 `worker.js` 内，通过 `onconnect` 事件监听端口连接，并在使用 `postMessage` 发送数据的页面之间转发消息。

```
JavaScript
// worker.js// 自身的事件监听器
self.onconnect = function (event) {var port = event.ports[0];
// 监听端口的消息
  port.onmessage = function (e) {if (e.data.command === "start")
{var result = someHeavyComputation(e.data.data);
      port.postMessage({ result: result });
    }
  };
};
```

```
// 在这里执行一些开销较大的计算逻辑 function
someHeavyComputation(data) {// 在这里进行计算...return
data.reduce(function (previousValue, currentValue) {return
previousValue + currentValue;
  }, 0);
}
```

1. 通知其他页面更新:

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页面更新时，可以利用 `SharedWorkerGlobalScope` 中的 `clients` 对象。

```
JavaScript
// 在 worker.js 中
self.addEventListener("message", (e) => {if (e.data === "Update
all clients") {// 遍历所有客户端
    self.clients.matchAll().then((clients) => {
      clients.forEach((client) => {// 发送消息更新它们
        client.postMessage("Please update your state");
      });
    });
  }
});
```

使用 localStorage 的变更监听

虽然 `localStorage` 没有直接提供跨标签页推送机制，但是可以使用 `window.addEventListener('storage', listener)` 监听 `storage` 事件，实现不同标签页间的通信。

```
JavaScript
// 标签页 1 修改了 localStorage
localStorage.setItem("someKey", "someValue");

// 其他标签页监听 storage 事件window.addEventListener("storage",
function (event) {if (event.storageArea === localStorage &&
event.key === "someKey") {console.log(event.newValue);
  }
});
```

使用 iframe 的 message 事件

如果排他性不是问题（所有标签页都属于同一客户端），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。

要使用 `iframe` 的 `message` 事件实现不同页签之间的通信，你需要两个关键项的配合：父页面和 `iframe` 页面之间的协调工作。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. 创建一个父页面

在父页面中，我们创建一个 `iframe` 并监听 `message` 事件。

```
XML
<!-- parent.html --><!DOCTYPE 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
      // 监听 iframe 发送的 message 事件
      window.addEventListener("message", function (event) {if
(event.origin !== "http://example.com") {// 确保消息源是可信的
          return;
        }if (event.data && event.data.greeting) {
          console.log("Message received from iframe:",
event.data);// 如果 iframe 向父页面问好（向父页面发送了一条消息）
          // 假设我们还想再向 iframe 发送一些信息

document.querySelector("iframe").contentWindow.postMessage(
            {
              response: "Hello iframe! This is the parent window
speaking.",
            },"http://example.com"
          );
        }
      });</script</body</html
```

3. 创建一个 iframe 页面

在 `iframe.html` 页面中，我们需要发送消息到父页面并监听父页面的消息。

```
XML
<!-- iframe.html --><!DOCTYPE html><html lang="en"<head<meta
charset="UTF-8" /><titleIframe Page</title</head<body<script
      // 假设我们有一些需要发送到父页面的信息
      function sendMessageToParent() {
```

```
        parent.postMessage({ greeting: "Hello, I am the
iframe!" }, "http://example.com");
      }
// 当页面加载完成后，发送消息
      window.onload = function () {
        sendMessageToParent();
      };
// 监听来自父页面的消息
      window.addEventListener("message", function (event) {if
(event.origin !== "http://example.com") {// 反向验证消息源的可信度
          return;
        }if (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
        }
      });</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 路由来组织页面路由？【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。

```
JavaScript
class App extends React.Component {
  componentDidMount() {// 当用户点击后退/前进按钮时触发路由变化
window.onpopstate = this.handlePopState;// 初始页面加载时处理路由
this.route();
  }
  handlePopState = () => {// 处理路由变化this.route();
  };
```

```
  route() {const path = window.location.pathname;// 根据 path 渲染不同的组件switch (path) {case "/page1":// 渲染 Page1 组件break;case "/page2":// 渲染 Page2 组件break;// 其他路由分支...default:// 渲染默认组件或 404 页面break;
    }
  }
  navigate = (path) => {// 更新历史记录并触发路由变化
window.history.pushState(null, "", path);this.route();
  };
  render() {return (<div
      <button onClick={() => this.navigate("/page1")}>Go to Page 1</button
      <button onClick={() => this.navigate("/page2")}>Go to Page 2</button
      {/* 这里根据路由渲染对应的组件 */}</div
    );
  }
}

// 实际的页面组件const Page1 = () => <divPage 1</div;const Page2 = () => <divPage 2</div;
```

尽管手动管理路由是可能的，但使用 react-router 这类专门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， 如何实现页面视口滚动到报错的位置【热度: 248】

基本原理

页面是用户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户进行修改。为了提高用户体验，可以将页面滚动至对应表单报错的位置，使得用户立即可见错误并进行修改。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，避免不必要的用户干扰。

以下是这些步骤的代码示例

HTML:

```
XML
<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:

```
JavaScript
// 一个假设的表单验证函数function validateInput(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var 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return el.value === "预期值";
}

function handleValidation() {var valid = true;
  ["name", "age"].forEach((key) => {// 进行校验判断if
(!validateInput(key)) {console.error(`Validation failed for:
${key}`);
// 标记校验失败
      valid = false;
// 滚动到出现问题的元素位置var element =
document.getElementById(key);
      element.scrollIntoView({ block: "center", behavior:
"smooth" });
// 增加一些提示效果，比如错误边框，可按需实现//
element.classList.add('error-highlight');
    }
  });
// 检查是否验证失败，如果失败则不提交表单return valid;
}

// 处理表单提交事件，与HTML 中的onclick 绑定
document.getElementById("myForm").addEventListener("submit", (e)
=> {
```

```
  e.preventDefault(); // 阻止表单默认提交行为
  handleValidation();
});
```

62. 如何一次性渲染十万条数据还能保证页面不卡顿【热度: 426】

原理其实就是 通过 `requestAnimationFrame` 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 `requestAnimationFrame` 来解决

`requestAnimationFrame` 也是个定时器，不同于 `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

> 因此 `requestAnimationFrame` 也是个宏任务，前阵子面试就被问到过这个

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 `li`，`dom` 结构的出现和屏幕的刷新保持了一致

```
JavaScript
const total = 100000;
let ul = document.getElementById("container");
let once = 20;
let page = total / once;

function loop(curTotal) {if (curTotal <= 0) return;
let pageCount = Math.min(curTotal, once);
window.requestAnimationFrame(() => {for (let i = 0; i < pageCount;
i++) {let li = document.createElement("li");
      li.innerHTML = ~~(Math.random() * total);
      ul.appendChild(li);
    }
    loop(curTotal - pageCount);
  });
}
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` 是虚拟文档碎片，我们一次 `for` 循环产生 20 个 `li` 的过程中可以全部把真实 `dom` 挂载到 `fragment` 上，然后再把 `fragment` 挂载到真实 `dom` 上，这样原来需要回流十万次，现在只需要回流 `100000 / 20` 次

```JavaScript
const total = 100000;
let ul = document.getElementById("container");
let once = 20;
let page = total / once;

function loop(curTotal) {if (curTotal <= 0) return;
let pageCount = Math.min(curTotal, once);
window.requestAnimationFrame(() => {let fragment =
document.createDocumentFragment(); // 创建一个虚拟文档碎片 for (let
i = 0; i < pageCount; i++) {let li = document.createElement("li");
      li.innerHTML = ~~(Math.random() * total);
      fragment.appendChild(li); // 挂到fragment上
    }
    ul.appendChild(fragment); // 现在才回流
    loop(curTotal - pageCount);
  });
}
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 值的长度。

在 webpack 的配置文件中，可以通过如下方式设定 hash:

```
JavaScript
output: {
  filename: '[name].[chunkhash].js',
  path: __dirname + '/dist'
}
```

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 `webpack-dev-server` 或者 `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?

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，页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。

```XML
<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。

```XML
<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。

```JavaScript
var script = document.createElement("script");
script.src = "script.js";
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，可以让页面内容先加载显示，从而减少用户对加载过程的可感知时间。

现代 Web 开发中通常推荐使用 `async` 或 `defer` 属性，提高页面加载性能，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户端，它们也必须等待当前处理中的请求完成后才能被发送。这会延迟整个页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户端，而不需要客户端明确地为每个资源提出请求。这提高了页面加载的速度，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。

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，网页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：

```
XML
<!-- 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 `html-webpack-plugin` 和 `externals`，你可以将一个库配置为 external，然后通过 `html-webpack-plugin` 将其引入：

```JavaScript
// webpack.config.js 文件const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {// ...
  externals: {
    libraryName: "LibraryGlobalVariable",
  },
  plugins: [new HtmlWebpackPlugin({
      template: "src/index.html",
    scriptLoading: "blocking", // 或者 'defer'
    }),
  ],
};
```

1. 然后，在你的 `index.html` 模板文件中可以这样引入资源：

```XML
<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 `import` 或 `require` 语句来引用那个全局变量：

```JavaScript
// 你的 JavaScript 代码文件中import Library from "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， 怎么通知用户刷新当前页面？【热度: 466】

首先第一个问题

用户在没有页面刷新的情况下， 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

```
Plain Text
xxx.1.0.0.html --> vender.hash_1.js、 vender.hash_2.js、
vender.hash_3.js、vender.hash_1.css
xxx.1.0.1.html --> vender.hash_a.js、 vender.hash_b.js、
vender.hash_c.js、vender.hash_d.css
```

如何主动推送给客户端

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， 且当前用户访问的还是旧版本， 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户端发送消息，包括静态资源更新的通知。收到消息后，客户端可以采取相应的措施，比如显示一个提示信息让用户选择是否重新加载页面。

1. Service Workers(推荐):

Service workers 位于浏览器和网络之间，可以控制页面的资源缓存。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:

客户端用 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，可以提醒用户或自动刷新资源。

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，即使用户没有开启网页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、用户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风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`.eslintrc`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风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 应用要怎么保持用户的登录态呢？【热度: 1,092】

大家都知道， HTTP 是一个无状态的协议，那么 Web 应用要怎么保持用户的登录态呢？

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，一个用户请求过来怎么判断他有没有登录呢?

在验证用户名和密码之后，我们可以发给客户端一个凭证(isLogin = true)，如果请求中有这个凭证，

那么他就是登陆之后的用户。 cookie 和 session 的区别在于，凭证的存储位置。换言之，如果凭

证存储在客户端，那就是 cookie 。如果凭证存储在服务端，那就是 session 。

客户端存储（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—客户端存储。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:

用户本人可以通过修改 document.cookie="isLogin = true" 伪造登陆凭证:

服务端存储（session）

session 本意是指客户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，仅发给客户端一个 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户登陆，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户。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：

当客户端登录成功后，服务器会给客户端一个令牌凭证 token；客户端后续的请求都需要带着这个

token 在服务器做验证。

但用户不可能只在一个网站登录，于是客户端会收到各个网站的出入证 token。所以客户端需要一个

“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，客户端访问不同网站时，会自动在请求中带着对应的 token。管理 token 的有效期，客

户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。【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。比如页面是 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，而客户端通过 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：比如当前页面打开 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（注意不是标签页关闭）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 // 对比常见的时间格式： 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，则可匹配的请求域有：

csdn.net、

www.csdn.net、blogs.csdn.net 等。 设置的 domain 是 www.csdn.net，则只能匹配 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… 网页来说：

【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：这种情况比较少见。举例：用户关闭了广告

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

夏风" && pwd === "123") { ctx.set("set-cookie", 'token=aaa; domain=localhost; max

age=3600;secure; httponly'); ctx.body = "登录成功"; } else { ctx.body = { code: 500, msg: "用户名或

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，会自动跳转到页面 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().toISOString() // 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，客户端设置 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户端

访问的，客户端设置它没有意义。

document.cookie = 'token=aaa; domain=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; max-age=-1'); 1 或客户端删除：

document.cookie = 'token=aaa; domain=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：

浏览器发送用户名和密码到服务器。 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. 如何检测网页空闲状态(一定时间内无操作)【热度: 329】

如何判断页面是否空闲

首先，我们要知道什么是空闲？用户一定时间内，没有对网页进行任何操作，则当前网页为空闲状态。

用户操作网页，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，来判断网页是否空闲(用户是否有操作网页)。

1. 监听鼠标移动事件 `mousemove`；
2. 监听键盘按下事件 `mousedown`；
3. 在用户进入网页后，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。

网页空闲检测实现

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，在页面隐藏时移除延时器，然后页面显示时继续计时，从而解决这个问题。

实现：

```
JavaScript
- /**
网页空闲检测
@param {() => void} callback 空闲时执行，即一定时长无操作时触发
@param {number} [timeout=15] 时长，默认 15s，单位：秒
@param {boolean} [immediate=false] 是否立即开始，默认 false
@returns
*/const onIdleDetection = (callback, timeout = 15, immediate =
```

```
false) => {let pageTimer;let beginTime = 0;const onClearTimer = ()
=> {
    pageTimer && clearTimeout(pageTimer);
    pageTimer = undefined;
  };const onStartTimer = () => {const currentTime = Date.now();if
(pageTimer && currentTime - beginTime < 100) {return;
    }
    onClearTimer();
    beginTime = currentTime;
    pageTimer = setTimeout(() => {
      callback();
    }, timeout * 1000);
  };
const onPageVisibility = () => {// 页面显示状态改变时，移除延时器
    onClearTimer();
if (document.visibilityState === "visible") {const currentTime =
Date.now();// 页面显示时，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) {
        callback();return;
      }// 继续计时
      pageTimer = setTimeout(() => {
        callback();
      }, timeout * 1000(currentTime - beginTime));
 }
};
const startDetection = () => {
    onStartTimer();document.addEventListener("mousedown",
onStartTimer);document.addEventListener("mousemove",
onStartTimer);document.addEventListener("visibilitychange",
onPageVisibility);
  };
const stopDetection = () => {
    onClearTimer();document.removeEventListener("mousedown",
onStartTimer);document.removeEventListener("mousemove",
onStartTimer);document.removeEventListener("visibilitychange",
onPageVisibility);
  };
const restartDetection = () => {
    onClearTimer();
    onStartTimer();
  };
if (immediate) {
    startDetection();
  }
```

```
return {
    startDetection,
    stopDetection,
    restartDetection,
  };
};
```

扩展

chrome 浏览器其实提供了一个 `Idle Detection`API，来实现网页空闲状态的检测，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）

这意味着，当使用 `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间。

而 `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缩短了编译时间，特别是在大型项目中，文件数量众多，`Vite` 的优势更为明显。

2、对 ES Modules 的支持

现代浏览器本身就支持 `ES Modules`，会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?

通过使用 `export` 和 `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 在打包速度上相比 Webpack 有 `10-100` 倍的提升。

什么是预构建依赖?

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。这样做的好处在于，当项目实际运行时，可以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。

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

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、充分利用了现代浏览器的 `ES Modules` 支持、使用了更高效的底层语言，并优化了热更新的处理。这些特点使得 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. 列表分页， 快速翻页下的竞态问题【热度: 444】

列表分页， 快速翻页下的竞态问题

> 问题描述：比如在前端分页请求的时候， 因为翻页很快， 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页码是对应上的。

在处理这种情况时，一种常见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：

```
JavaScript
```

```
// 存储请求的标记let requestId = 0;

// 发起请求的函数function sendRequest(page) {
  requestId++;
// 将请求标记与页码一起发送
  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function handleResponseData(requestId, data)
{if (requestId === currentRequestId) {// 在这里处理数据并更新页面
  }
}

// 在翻页时调用 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`，并将其与页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：

- 对快速翻页进行限制或优化，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：

```
JavaScript
```

```
let requestId = Date.now();
```

这样每次请求时，`requestId` 都会是一个唯一的时间戳值。

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，保持页面的响应性。

以下是一个使用 Web Workers 的简单示例：

```
JavaScript
// 主线程代码const worker = new Worker('worker.js'); // 创建一个新的 Web Worker
worker.postMessage({ start: 0, end: 1000000 }); // 向Web Worker 发送消息
worker.onmessage = function(event) {const result = event.data;console.log('任务完成: ', result);
};

// worker.js - Web Worker 代码
onmessage = function(event) {const start = event.data.start;const end = event.data.end;let sum = 0;for (let i = start; i <= end; i++) {
    sum += i;
  }
  postMessage(sum); // 向主线程发送消息
};
```

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 `requestAnimationFrame` 来实现任务分割是一种常见的方式，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 `requestAnimationFrame` 来分割任务的简单例子：

```
JavaScript
```

```
// 假设有一个包含大量元素的数组const bigArray = Array.from({ length:
1000000 }, (_, i) => i + 1);

// 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function
processChunk(chunk) {return chunk.map(num => num * num);
}

// 分割任务并使用requestAnimationFrameconst chunkSize = 1000; // 每
个小块的大小let index = 0;

function processArrayWithRAF() {function processChunkWithRAF()
{const chunk = bigArray.slice(index, index + chunkSize); // 从大数
组中取出一个小块const result = processChunk(chunk); // 处理小块任务
console.log('处理完成：', result);
    index += chunkSize;
if (index < bigArray.length) {
      requestAnimationFrame(processChunkWithRAF); // 继续处理下一个
小块
    }
  }
  requestAnimationFrame(processChunkWithRAF); // 开始处理大数组
}
processArrayWithRAF();
```

在这个例子中，我们使用 `requestAnimationFrame` 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:

```
JavaScript
  const $result = document.getElementById("result");

// 假设有一个包含大量元素的数组const bigArray = Array.from({ length:
1000000 }, (_, i) => i + 1);

// 定义一个处理函数，对数组中的每个元素执行一次function
processChunk(chunk) {return
chunk: ${chunk}
;
```

```
}

// 动态调整 chunkSize 的优化方式let chunkSize = 1000; // 初始的
chunkSizelet index = 0;

function processArrayWithDynamicChunkSize() {function
processChunkWithRAF() {let startTime = performance.now(); // 记录结
束时间for (let i = 0; i < chunkSize; i++) {if (index <
bigArray.length) {const result = processChunk(bigArray[index]); //
对每个元素执行处理函数
    $result.innerText = result;
      index++;
      }
    }let endTime = performance.now();let timeTaken = endTime -
startTime; // 计算处理时间// 根据处理时间动态调整 chunkSizeif
(timeTaken > 16) { // 如果处理时间超过一帧的时间（16 毫秒），则减小
chunkSize
      chunkSize = Math.floor(chunkSize * 0.9); // 减小 10%
    } else if (timeTaken < 16) { // 如果处理时间远小于一帧的时间（8
毫秒），则增加 chunkSize
      chunkSize = Math.floor(chunkSize * 1.1); // 增加 10%
    }
if (index < bigArray.length) {
      requestAnimationFrame(processChunkWithRAF); // 继续处理下一个
小块
    }
  }
  requestAnimationFrame(processChunkWithRAF); // 开始处理大数组
}
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：

方法一：使用 `git clone` 和 `git push`

1. 在仓库 B 中创建新的仓库。
2. 在本地克隆仓库 A:

```
Bash
git clone --mirror <仓库 A URL>
cd <仓库 A 目录>
```

使用 `--mirror` 选项克隆仓库会保留所有分支、标签和提交历史。

1. 修改远程仓库地址为仓库 B:

```
Bash
git remote set-url --push origin <仓库 B URL>
```

1. 推送到仓库 B:

```
Bash
git push --mirror
```

方法二：使用 `git bundle`

1. 在仓库 A 中创建 bundle 文件:

```
Bash
git bundle create repoA.bundle --all
```

1. 将 `repoA.bundle` 文件传输到仓库 B 所在位置。
2. 在仓库 B 中克隆:

```
Bash
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。

77. 如何禁止别人调试自己的前端页面代码？【热度: 347】

无限 debugger

- 前端页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，所以网页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
- 可以通过控制台中的 `Deactivate breakpoints` 按钮或者使用快捷键 `Ctrl + F8` 关闭无限 `debugger`
- 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 `breakpoint`

禁止断点的对策

- 如果将 `setInterval` 中的代码写在一行，就能禁止用户断点，即使添加 `logpoint` 为 `false` 也无用
- 当然即使有些人想到用左下角的格式化代码，将其变成多行也是没用的

```JavaScript
(() => {function ban() {
    setInterval(() => { debugger; }, 50);
  }try {
    ban();
  } catch (err) { }
```

```
})();
```

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户呢
- 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

```
JavaScript
// 加密前
(() => {function ban() {
    setInterval(() => {Function('debugger')();
    }, 50);
  }try {
    ban();
  } catch (err) { }
})();

// 加密后
eval(function(c,g,a,b,d,e){d=String;if(!"".replace(/^/,String)){fo
r(;a--;)e[a]=b[a]||a;b=[function(f){return
e[f]}];d=function(){return"\w+"};a=1}for(;a--
;)b[a]&&(c=c.replace(new RegExp("\b"+d(a)+"\b","g"),b[a]));return
c}('(()=>{1 0(){2(()=>{3("4")()},5)}6{0()}7(8){}})();',9,9,"block
function setInterval Function debugger 50 try catch err".split("
"),0,{}));
```

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
- 将 `Function('debugger').call()`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

```JavaScript
(() => {function block() {if (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!";
    }
    setInterval(() => {
      (function () {return false;
      }
      ['constructor']('debugger')
      ['call']());
    }, 50);
  }try {
    block();
  } catch (err) { }
})();
```

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

点击图片可查看完整电子表格

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：

```
Fortran
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：

```
JavaScript
const path = require('path');

module.exports = {
  entry: 'src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },module: {
    rules: [
      {
        test: /\.(png|jpe?g|gif)$/i,
        use: [
```

```
        {
          loader: 'file-loader',
          options: {
            name: '[name].[ext]',
            outputPath: 'images/'
          }
        },
        {
          loader: 'image-webpack-loader',
          options: {
            mozjpeg: {
              progressive: true,
              quality: 65
            },// optipng.enabled: false will disable optipng
            optipng: {
              enabled: false,
            },
            pngquant: {
              quality: [0.65, 0.90],
              speed: 4
            },
            gifsicle: {
              interlaced: false,
            },// the webp option will enable WEBP
            webp: {
              quality: 75
            }
          }
        }
      ]
    }
  ]
};
```

上述配置中，我们使用 `file-loader` 将图片复制到输出目录，并使用 `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

高页面加载速度。

雪碧图的原理是通过 CSS 的 `background-image` 和 `background-position` 属性，将所需的小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，提高了页面加载速度。

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。

```
Bash
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。

```
JavaScript
const SpritesmithPlugin = require('webpack-spritesmith');

module.exports = {// ...其他配置
  plugins: [new SpritesmithPlugin({
      src: {
        cwd: path.resolve(__dirname, 'path/to/sprites'), // 需要合并的小图片所在的目录
        glob: '*.png' // 小图片的文件名格式
      },
      target: {
        image: path.resolve(__dirname, 'path/to/output/sprite.png'), // 生成的雪碧图的路径和文件名
        css: path.resolve(__dirname, 'path/to/output/sprite.css') // 生成的CSS 样式表的路径和文件名
      },
      apiOptions: {
        cssImageRef: 'path/to/output/sprite.png' // CSS 样式表中引用雪碧图的路径
      }
    })
  ]
}
```

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：

```
CSS
div {background: url(path/to/output/sprite.png) no-repeat;
}.icon-facebook {/* 设置小图标在雪碧图中的位置和大小 */
  width: 32px;height: 32px;background-position: 0 0; /* 该小图标在雪碧图中的位置 */
}.icon-twitter {width: 32px;height: 32px;background-position: -32px 0; /* 该小图标在雪碧图中的位置 */
}.icon-instagram {width: 32px;height: 32px;background-position: -64px 0; /* 该小图标在雪碧图中的位置 */
}
```

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：

```
XML
<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。

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。`webpack-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 `iconfont` 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 `iconfont` 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/
- 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 `iconfont` 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 `iconfont` 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：

```
CSS
@font-face {font-family: 'iconfont';src:
url('path/to/iconfont.eot'); /* 引入字体文件 */
  /* 其他格式的字体文件 */
}.icon {font-family: 'iconfont'; /* 使用定义的字体名称 */
  font-size: 16px; /* 图标大小 */
  line-height: 1; /* 图标行高 */
}.icon-facebook::before {content: '\e001'; /* 使用Unicode 代码表示想要显示的图标 */
}.icon-twitter::before {content: '\e002'; /* 使用Unicode 代码表示想要显示的图标 */
}.icon-instagram::before {content: '\e003'; /* 使用Unicode 代码表示想要显示的图标 */
}
```

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`iconfont`。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：

```
XML
<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`iconfont`来处理你的图标资源，并在项目中方便地使用它们。确保在 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，每个网页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，从而提高网页加载速度。

- 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，从而减少了网页的总体积，加快了网页加载速度。
- 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，例如页面背景渐变、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，从而导致网页加载速度变慢。
- 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 `url-loader` 或 `file-loader`，它们是 Webpack 的两个常用的加载器。

```Bash
npm install url-loader --save-dev
```

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。

```JavaScript
module.exports = {// ...module: {
   rules: [// ...
     {
       test: /\.(png|jpe?g|gif)$/i,
       use: [
         {
```

```
            loader: 'url-loader',
            options: {
              limit: 8192, // 设置图片大小的阈值，小于该值的图片会被转为Base64
              outputPath: 'images', // 输出路径
              publicPath: 'images', // 资源路径
            },
          },
        ],
      },
    ],
  },// ...
};
```

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。

```
JavaScript
import imgSrc from './path/to/image.png';

const imgElement = document.createElement('img');
imgElement.src = imgSrc;
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。

```
Bash
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页加载变慢。

2. 使用 CDN 加载图片

CND 加载图片优势非常明显：

- 加速网页加载速度：CDN 通过将图片资源分布在全球的多个节点上，使用户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。这可以提高网页的加载速度和用户体验。

- 减轻服务器负载：CDN 充当了一个缓冲层，当用户请求图片资源时，CDN 会将图片资源从源服务器获取并缓存在节点中，下次再有用户请求同一资源时，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，用户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，确保用户能够正常访问图片资源。

总之，使用 CDN 加载图片可以提高网页加载速度、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，从而改善用户体验和网站性能。

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页面的初始加载时间，并提高用户的浏览体验。

- 原理：图片懒加载的原理是只在用户需要时加载图片，而不是在页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-src`）中，而不是在 `src` 属性中。然后，在图片进入浏览器视图时，通过 JavaScript 动态将 `data-src` 的值赋给 `src` 属性，触发图片的加载。
- 优势：图片懒加载可以显著减少初始页面的加载时间，特别是当页面中有大量图片时。它使页面加载变得更快，提高了用户的浏览体验。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，以提供更好的用户体验。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。这可以大大减轻页面的加载时间，并降低带宽消耗，提高了用户的体验。以下是一些常见的实现方法：

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。

```
JavaScript
let observer = new IntersectionObserver(function (entries) {
  entries.forEach(function (entry) {if (entry.isIntersecting)
{const lazyImage = entry.target;
      lazyImage.src = lazyImage.dataset.src;
      observer.unobserve(lazyImage);
    }
  });
});

const lazyImages = [...document.querySelectorAll(".lazy")];
lazyImages.forEach(function (image) {
  observer.observe(image);
});
```

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。

```
JavaScript
function lazyLoad() {const images =
document.querySelectorAll(".lazy");const scrollTop =
window.pageYOffset;
  images.forEach((img) => {if (img.offsetTop < window.innerHeight
+ scrollTop) {
      img.src = img.dataset.src;
      img.classList.remove("lazy");
    }
  });
}

let lazyLoadThrottleTimeout;
document.addEventListener("scroll", function () {if
(lazyLoadThrottleTimeout) {
    clearTimeout(lazyLoadThrottleTimeout);
  }
  lazyLoadThrottleTimeout = setTimeout(lazyLoad, 20);
});
```

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，避免页面布局的混乱。

```XML
<!-- 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户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页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`，可以在网页 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 `src` 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、LazyLoad.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。

实现图片预加载可以使用原生 JavaScript 或使用专门的 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：

```JavaScript
function preloadImage(url) {return new Promise(function(resolve, reject) {var img = new Image();
   img.onload = resolve;
   img.onerror = reject;
   img.src = url;
 });
}

// 调用预加载函数
preloadImage('image.jpg')
 .then(function() {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
 })
 .catch(function() {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
});
```

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：

```
XML
<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：

```
JavaScript
var preload = new createjs.LoadQueue();
preload.on("complete", handleComplete);
preload.on("error", handleError);
preload.loadFile('image.jpg');

function handleComplete() {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
}

function handleError() {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
}
```

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，`handleError` 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`<picture>`元素和

`<source>`元素来实现响应式图片。以下是一个示例：

```
XML
<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 `src` 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，以优化用户的视觉体验和页面加载性能。

有可以使用 webpack `responsive-loader` 来实现自动根据设备分辨率加载不同的倍图：

依赖安装：

```
GraphQL
npm install responsive-loader sharp --save-dev
```

webpack 配置示范

```
JavaScript
module.exports = {
  entry: {...},
  output: {...},module: {
    rules: [
      {
        test: /\.(jpe?g|png|webp)$/i,
         use: [
          {
            loader: "responsive-loader",
            options: {
              adapter: require('responsive-loader/sharp'),
              sizes: [320, 640, 960, 1200, 1800, 2400],
              placeholder: true,
              placeholderSize: 20
            },
          },
        ],
      }
```

```
    ]
  },
}
```

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)

```
CSS
.myImage {background: url('myImage.jpg?size=1140');
}@media (max-width: 480px) {.myImage {background:
url('myImage.jpg?size=480');}
}
```

导入图片到 JS 中:

```
JavaScript
import responsiveImage from
'img/myImage.jpg?sizes[]=300,sizes[]=600,sizes[]=1024,sizes[]=2048
';
import responsiveImageWebp from
'img/myImage.jpg?sizes[]=300,sizes[]=600,sizes[]=1024,sizes[]=2048
&format=webp';

// 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'
...
  <picture><source srcSet={responsiveImageWebp.srcSet}
type='image/webp' sizes='(min-width: 1024px) 1024px, 100vw'/>
    <img
      src={responsiveImage.src}
      srcSet={responsiveImage.srcSet}
      width={responsiveImage.width}
      height={responsiveImage.height}
      sizes='(min-width: 1024px) 1024px, 100vw'
      loading="lazy"
    />
```

```
  </picture
...
```

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`src`属性。这将立即加载并显示这张低分辨率的图片。
3. 在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-src`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`src`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：

```
XML
<!-- HTML --><img src="blur-image.jpg" data-src="high-res-
image.jpg" alt="Image"<script
// JavaScript
const image = document.querySelector('img');

// 监听高分辨率图片加载完成事件
image.addEventListener('load', () => {// 替换低分辨率图片的 src 属性
  image.src = image.dataset.src;
});
</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户等待时间，提供更好的用户体验。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，这可能会增加页面加载时间和网络带宽消耗。因此，开发者需要在性能和用户体验之间进行权衡，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户的资源。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、资源服务器和客户端等角色，实现了用户授权和资源访问的分离。具体流程如下：

1. 用户向客户端发起请求，请求访问某个资源。
2. 客户端将用户重定向到授权服务器，并携带自己的身份凭证（客户端 ID）。
3. 用户在授权服务器登录，并授权客户端访问特定的资源。
4. 授权服务器验证用户身份，并生成访问令牌（Access Token）。
5. 授权服务器将访问令牌发送给客户端。
6. 客户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. 资源服务器向客户端返回请求的资源。

在这个过程中，OAuth2.0 通过访问令牌实现了用户和资源服务器之间的身份授权和资源访问分离。客户端无需知道或存储用户的凭证（如用户名和密码），而是使用访问令牌代表用户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：

```
JavaScript
// 1. 客户端应用程序发起授权请求，重定向用户到授权服务器的登录页面const
authorizationEndpoint = 'https://example.com/oauth2/auth';
const clientId = 'your_client_id';
const redirectUri = 'https://yourapp.com/callback';
const scope = 'read write';
const state = 'random_state_value';

const authorizationUrl =
${authorizationEndpoint}?client_id=${clientId}&redirect_uri=${redi
rectUri}&scope=${scope}&state=${state}
;

// 重定向用户到授权页面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. 客户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,
  redirect_uri: redirectUri,
  client_id: clientId,
  client_secret: clientSecret
};

// 使用Fetch API 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 => {const accessToken = data.access_token;
// 4. 客户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}
      }
    })
      .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。用户只要登录一次，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. 用户访问系统 A，判断未登录，则直接跳到认证中心页面
2. 在认证中心页面输入账号，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. 用户访问系统 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. 用户访问系统 A (www.app1.com)， 跳转认证中心 client(www.sso.com)， 然后输入用户名，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)， 重定向到系统 A，并且带上生成的 ticket 参数 (www.app1.com?ticket =xxx)

2. 系统 A (www.app1.com?ticket =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

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)，跳转到认证中心 client(www.sso.com)，这个时候会把认证中心的 cookieSSO 也携带上，发现用户已登录过，则直接重定向到系统 B（www.app2.com），并且带上生成的 ticket 参数（www.app2.com?ticket =xxx）

2. 系统 B (www.app2.com?ticket =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 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. 常见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: 用户在登录成功后，服务器会生成一个包含用户信息的 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户身份。
    - Session: 用户登录成功后，服务器会在后端保存用户的登录状态信息，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 来验证用户身份。
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。用户只需登录一次，即可在多个系统中完成认证，避免了重复登录的麻烦。常见的单点登录协议有 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，使得用户可以使用第三方身份提供者（如 Google、Facebook 等）进行登录和授权，从而实现用户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户的资源。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户获取资源的权限，而无需知道用户的真实凭证。
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 常用于验证用户的身份信息，如用户

名和密码，通过与 LDAP 服务器进行通信来进行用户身份验证。

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。与传统的用户名和密码登录不同，2FA 需要用户提供第二个验证因素，如手机验证码、指纹识别、硬件令牌等，以提高账户的安全性。

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）：

```JavaScript
const express = require('express');
const app = express();
app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin',
'http://example.com');
  res.setHeader('Access-Control-Allow-Credentials', 'true');
  next();
});
app.get('/api/data', (req, res) => {// 处理请求
  res.send('Response Data');
});
app.listen(3000, () => {console.log('Server is running on port
3000');
});
```

在客户端发起跨域请求时，需要设置请求头部的 withCredentials 字段为 true，示例代码（JavaScript）：

```JavaScript
fetch('http://example.com/api/data', {
  credentials: 'include',
})
  .then(response => response.text())
  .then(data => {console.log(data);
```

```
})
.catch(error => {console.error('Error:', error);
});
```

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，而不需要刷新整个页面。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页面。

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，以提高页面加载速度。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	需要进行打包构建，每次修改代码都需要重新

点击图片可查看完整电子表格

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

```
TypeScript
- interface InitWithRetries extends RequestInit {
  fetchTimeout?: number | null;
  retryDelays?: number| null;
}

const DEFAULT_TIMEOUT = 10001.5;
```

```
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/")
```

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

```
C#
dev: {     // Paths
    assetsSubDirectory: 'static', // 静态资源文件夹
    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
    proxyTable: {'/api': {
        timeout: 30000, // 请求超时时间
        target: 'http://127.0.0.1:3006', // 目标接口域名
        changeOrigin: true, // 是否跨域
        pathRewrite: {'^/api': '' // 重写接口
        }
      },// Various Dev Server settings
    host: 'localhost', // can be overwritten by process.env.HOST
    port: 4200, // can be overwritten by process.env.PORT, if port is in use, a free one will be determined
  }
```

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

```
JavaScript
import axios from "axios";

const 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
  retry: 2, // 请求次数
  retryInterval: 1000 // 求期间隙
  ......
});
//请求前拦截
Axios.interceptors.request.use(config => {return config
  },function(error) {return Promise.reject(error)
  }
);
//请求后返回数据拦截
Axios.interceptors.response.use(res => {return res
```

```
  },function axiosRetryInterceptor(res) {var config =
res.config;//如果配置不存在或重试属性未设置，抛出promise 错误if
(!config || !config.retry) return Promise.reject(res);//设置一个变
量记录重新请求的次数
      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if (config.retryCount >= config.retry)
{return Promise.reject(res);
      }//重新请求的次数自增
      config.retryCount += 1;// 创建新的Promise 来处理重新请求的间隙
var back = new Promise(function(resolve) {console.log("接口
"+config.url+"请求超时，重新请求")
          setTimeout(function() {
              resolve();
          }, config.retryInterval|| 1);
      });//返回axios 的实体，重试请求return back.then(function()
{return Axios(config);
      });
  }
);
export default 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。
- Node.js 也提供了 `worker_threads` 模块，可以创建真正的多线程应用程序。这个模块允许开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，只渲染当前可见区域的树节点，而不是全部节点。可以根据页面的滚动位置和用户操作，只渲染当前需要展示的节点，而对于不可见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，降低栈溢出的风险。

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页面性能和用户体验。

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，使用户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. 页面加载速度提升（性能优化）应该从哪些反向来思考？【热度: 1,099】

页面加载优化

「页面加载链路+流程优化+协作方」的多级分类思考

- 页面启动
  - service worker 缓存重要的静态资源
  - 页面保活
- 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， 请求后续页面资源

- 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 页面 PV、UV。 如果能自动化上报页面性能， 用户点击路径行为，就更好了。
- 自动上报页面异常。
- 发送埋点信息的时候， 不影响性能， 不阻碍页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

```
JavaScript
import StatisticSDK from 'StatisticSDK';
// 全局初始化一次window.insSDK = new StatisticSDK('uuid-12345');

<button onClick={() => {
  window.insSDK.event('click', 'confirm');
...// 其他业务代码
}}>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

页面的其他操作。这意味着即使页面正在卸载或关闭，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。

3. 自动化处理：`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。

4. 跨域支持：`navigator.sendBeacon()` 方法支持跨域发送数据。在一些情况下，例如使用第三方统计服务等，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()`
方法可以避免跨域问题。

需要注意的是，`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。

简单介绍一下 `navigator.sendBeacon` 用法

语法：

```JavaScript
navigator.sendBeacon(url);
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

```JavaScript
class StatisticSDK {
  constructor(productID, baseURL) {this.productID =
productID;this.baseURL = baseURL;
  }
  send(query = {}) {
    query.productID = this.productID;
```

```
let data = new URLSearchParams();for (const [key, value] of
Object.entries(query)) {
      data.append(key, value);
    }
    navigator.sendBeacon(this.baseURL, data);
  }
}
```

用户行为与日志上报

用户行为主要涉及到的是事件上报和 pv 曝光， 借助 send 实现即可。

```
JavaScript
class StatisticSDK {
  constructor(productID, baseURL) {this.productID =
productID;this.baseURL = baseURL;
  }
  send(query = {}) {
    query.productID = this.productID;
let data = new URLSearchParams();for (const [key, value] of
Object.entries(query)) {
      data.append(key, value);
    }
    navigator.sendBeacon(this.baseURL, data);
  }
  event(key, value = {}) {this.send({ event: key, ...value })
  }
  pv() {this.event('pv')
  }
}
```

性能上报

性能主要涉及的 api 为 performance.timing 里面的时间内容;

```
JavaScript
class StatisticSDK {
  constructor(productID, baseURL) {this.productID =
productID;this.baseURL = baseURL;
  }
  send(query = {}) {
    query.productID = this.productID;
let data = new URLSearchParams();for (const [key, value] of
```

```
Object.entries(query)) {
      data.append(key, value);
    }
    navigator.sendBeacon(this.baseURL, data);
  }
// ....
  initPerformance() {this.send({ event:
'performance', ...performance.timing })
  }
}
```

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 `unhandledrejection` 事件。

```
JavaScript
class StatisticSDK {
  constructor(productID, baseURL) {this.productID =
productID;this.baseURL = baseURL;
  }
  send(query = {}) {
    query.productID = this.productID;
let data = new URLSearchParams();for (const [key, value] of
Object.entries(query)) {
      data.append(key, value);
    }
    navigator.sendBeacon(this.baseURL, data);
  }
// ....
  error(err, errInfo = {}) {const { message, stack } =
err;this.send({ event: 'error', message, stack, ...errInfo })
  }
  initErrorListenner() {window.addEventListener('error', event =>
{this.error(error);
    })window.addEventListener('unhandledrejection', event =>
{this.error(new Error(event.reason), { type:
'unhandledrejection' })
    })
```

```
  }
}
```

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，不会整个页面崩溃。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」，不至于整个页面都不能正常工作。

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：`getDerivedStateFromError` 和 `componentDidCatch`;

```
JavaScript
// 定义错误边界class ErrorBoundary extends React.Component {
  state = { error: null }
  static getDerivedStateFromError(error) {return { error }
  }
  componentDidCatch(error, errorInfo) {// 调用我们实现的 SDK 实例
    insSDK.error(error, errorInfo)
  }
  render() {if (this.state.error) {return <h2Something went
wrong.</h2
    }
    return this.props.children
  }
}
...
<ErrorBoundary
  <BuggyCounter />
</ErrorBoundary
```

Vue

vue 也有一个类似的生命周期来做这件事：`errorCaptured`

```
JavaScript
Vue.component('ErrorBoundary', {
  data: () => ({ error: null }),
  errorCaptured (err, vm, info) {this.error =
`${err.stack}\n\nfound in ${info} of component`// 调用我们的 SDK，上
报错误信息
    insSDK.error(err,info)return false
  },
```

```
  render (h) {if (this.error) {return h('pre', { style: { color:
'red' }}, this.error)
    }return this.$slots.default[0]
  }
})
...
<error-boundary><buggy-counter />
</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，当用户进行登录或授权时，服务器会生成一个 Token 并将其发送给客户端。客户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，以验证用户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 可以存储更多的用户信息，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 的身份验证中常见。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户端，例如 Web、移动应用和第三方接入等。

token 一般在客户端存在哪儿

Token 一般在客户端存在以下几个地方：

- Cookie：Token 可以存储在客户端的 Cookie 中。服务器在响应请求时，可以将 Token 作为一个 Cookie 发送给客户端，客户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 也可以存储在客户端的 Local Storage 或 Session Storage 中。这些是 HTML5 提供的客户端存储机制，可以在浏览器中长期保存数据。
- 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 也可以包含在客户端发送的请求头中，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见的做法，但是并不是最安全的方式。存放在 Cookie 中的 Token 可能存在以下安全风险：

- 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户的 Cookie 信息，包括 Token。攻击者可以利用 Token 冒充用户进行恶意操作。
- 跨站请求伪造（CSRF）：攻击者可以利用 CSRF 漏洞，诱使用户在已经登录的情况下访问恶意网站，该网站可能利用用户的 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。如果用户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户的账户。

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，减少被滥用的风险。
- 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 标头发送给客户端的一小段数据。客户端浏览器将 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 通常用于在客户端和服务器之间维护会话状态，以及存储用户相关的信息。

Token 是一种用于身份验证和授权的令牌。它是一个包含用户身份信息的字符串，通常是服务器生成并返回给客户端。客户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 的有效性来确认用户的身份和权限。

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，然后发送给客户端保存。客户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 的有效性来判断用户的身份和权限。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，以及客户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：

角色与权限分离

将用户的权限分为不同的角色，每个角色拥有特定的权限。

这样可以简化权限管理，并且当需求变化时，只需要调整角色的权限，而不需要逐个修改用户的权限。

在角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. 确定角色集合：根据系统的角色需求，确定不同的角色，例如管理员、普通用户、编辑等。每个角色代表一组权限的集合，可以根据业务需求进行划分。
3. 分配权限给角色：将权限与角色进行关联，确定每个角色具备哪些权限。可以通过角色-权限的映射表或者通过角色组的方式进行管理。
4. 用户与角色关联：将用户与角色进行关联，确定每个用户属于哪些角色。可以通过用户-角色的映射表或者通过用户组的方式进行管理。
5. 权限验证：在系统中，根据用户的角色和权限配置进行权限验证。在用户进行操

作或访问受限资源时，根据用户的角色与权限进行验证，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户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户进行细粒度的权限控制，即控制用户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. 角色与权限关联：将权限与角色进行关联。确定每个角色具备哪些权限。可以使用角色-权限的映射表进行管理。
4. 用户与角色关联：将用户与角色进行关联。确定每个用户属于哪些角色。可以使用用户-角色的映射表进行管理。
5. 权限验证：在系统中，对用户进行权限验证。当用户进行某个功能操作时，根据用户的角色与权限进行验证，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，用于管理角色与权限的关联。管理员可以通过该界面对角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，可以根据用户角色的配置动态控制用户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

对于不同的页面或路由，可以根据用户的角色或权限来进行权限控制。在前端路由中配置权限信息，当用户访问特定路由时，前端会检查用户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页面中根据用户的权限配置，控制用户是否可以访问某个路由或者页面。

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页面组件。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：对于每个路由或者页面，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. 获取用户权限：在登录成功后，从后端获取当前用户的权限信息。可以将用户的权限信息存放在前端的状态管理库（如 Vuex 或 Redux）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，根据当前用户的权限信息和路由配置进行判断，决定是否允许用户访问该路由。如果用户没有相应的权限，可以进行跳转到无权限提示页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户的权限和路由配置，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，可以在用户登录时根据权限配置动态生成。根据用户的权限配置，过滤路由表，生成用户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，即在用户登录时从服务器获取用户的权限信息，并在前端进行缓存。这样可以保证用户权限的实时性，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户可见或可操作。

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，当用户越权操作时，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角色与权限分离、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户权限的灵活管理。

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] = [页面]

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、页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、页面、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，它一般融合了页面的渲染以及组件/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页面，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页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 端营销页的搭建，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页面研发的。
- 研发场景：从研发场景来看，低代码搭建不仅有页面编排，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》
- 《低代码引擎物料协议规范》

- 《低代码引擎资产包协议规范》

95. [Webpack] 有哪些优化项目的手段？【热度: 1,163】

围绕 `webpack` 做性能优化，分为两个方面：`构建时间优化`、`构建体积优化`

构建时间优化

- 缩小范围
- 文件后缀
- 别名
- 缓存
- 并行构建
- 定向查找第三方模块
- 构建结果优化
  - 压缩 js
  - 压缩 css
  - 压缩 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 `include`、`exclude` 缩小 loader 对文件的搜索范围，以此来提高构建速率。

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

```
JavaScript
module.exports = {module: {
        rules: [
            {
                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
                include: path.resolve(__dirname, '../src'),
                use: [ 'thread-loader', 'babel-loader'],
                //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
                exclude: /node_modules/,
            }
        ]
    }
}
```

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。

```
JavaScript
module.export = {
  resolve: {// 按照 tsx、ts、jsx、js 的顺序匹配，若没匹配到则报错
    extensions: ['.tsx', '.ts', '.jsx', '.js'],
  }
}
```

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

```
JavaScript
```

```
module.exports = {
    resolve: {
        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')
        }
    }
}

// 引入 src 下的某个模块时import XXX from '@/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

```
JavaScript
module.exports = {module: {
    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`

```
JavaScript
module.exports = {module: {
    rules: [
      {
```

```
        test: /.jsx?$/,
        use: ['cache-loader', "babel-loader"
        ],
      }
    ]
  }
}
```

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

```
JavaScript
module.exports = {
  cache: {
    type: 'filesystem'
  }
}
```

编译后会多出 `/node_modules/.cache/webpack` 缓存目录

并行构建

首先，运行在 `Node` 里的 `webpack` 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优势，也能提高构建速度？ thread-loader 可以开启多进程打包

```
JavaScript
module.exports = {module: {
    rules: [
      {
        test: /.jsx?$/,
        use: [// 开启多进程打包。
          {
            loader: 'thread-loader',
            options: {
              workers: 3 // 开启 3 个 进程
            }
          },
          {
            loader: 'babel-loader',
          }
        ]
      }
    ]
```

```
  }
}
```

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

`resolve.modules` 配置用于指定 `webpack` 去哪些目录下寻找第三方模块。默认值是 `['node_modules']`。而在引入模块的时候，会以 `node 核心模块 -----> node_modules ------> node 全局模块` 的顺序查找模块。

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

```JavaScript
module.export = {
  resolve: {
    modules: [path.resolve(__dirname, 'node_modules')]
  }
}
```

构建结果优化

压缩 js

webpack5 的话通过 `terser-webpack-plugin` 来压缩 JS，但在配置了 `mode: production` 时，会默认开启

```JavaScript
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  optimization: {// 开启压缩
    minimize: true,// 压缩工具
    minimizer: [new TerserPlugin({}),
    ],
  },
}
```

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 `TerserPlugin` 显示配置或者使用`...`，否则 `terser-webpack-plugin` 会被覆盖。

```
JavaScript
const TerserPlugin = require("terser-webpack-plugin");
optimization: {
  minimize: true,
  minimizer: [new TerserPlugin({}), // 显示配置// "...", // 或者使用展开符，启用默认插件// 其它压缩插件new CssMinimizerPlugin(),
  ],
},
```

压缩 css

压缩 css 我们使用 `css-minimizer-webpack-plugin`

同时，应该把 css 提取成单独的文件，使用 `mini-css-extract-plugin`

```
JavaScript
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");

module.exports = {module: {
    rules: [
      {
        test: /\.css$/,
        use: [// 提取成单独的文件
            MiniCssExtractPlugin.loader,"css-loader"
        ],
        exclude: /node_modules/,
      },
    ]
  },
  plugins: [new MiniCssExtractPlugin({// 定义输出文件名和目录
      filename: "asset/css/main.css",
    })
  ],
  optimization: {
    minimize: true,
    minimizer: [// 压缩 cssnew CssMinimizerPlugin({}),
    ],
  },
```

```
}
```

压缩 html

压缩 `html` 使用的还是 `html-webpack-plugin` 插件。该插件支持配置一个 minify 对象，用来配置压缩 `html`。

```
JavaScript
module.export = {
  plugins: [new HtmlWebpackPlugin({// 动态生成 html 文件
    template: "./index.html",
      minify: {// 压缩HTML
        removeComments: true, // 移除HTML 中的注释
        collapseWhitespace: true, // 删除空白符与换行符
        minifyCSS: true // 压缩内联 css
      },
    })
  ]
}
```

压缩图片

可以通过 `image-webpack-loader` 来实现

```
JavaScript
module.exports = {module: {
    rules: [
      {
        test: /\.(png|jpg|gif|jpeg|webp|svg)$/,
        use: ["file-loader",
          {
            loader: "image-webpack-loader",
            options: {
              mozjpeg: {
                progressive: true,
              },
              optipng: {
                enabled: false,
              },
              pngquant: {
                quality: [0.65, 0.9],
                speed: 4,
              },
```

```
            gifsicle: {
              interlaced: false,
            },
          },
        },
      ],
      exclude: /node_modules/, //排除 node_modules 目录
    },
  ]
 },
}
```

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。

将路由页面/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。比如某个列表页和编辑页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，页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

```JavaScript
const List = lazyComponent('list', () => import(/*
webpackChunkName: "list" */ '@/pages/list'));
const Edit = lazyComponent('edit', () => import(/*
webpackChunkName: "list" */ '@/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长。我们可以使

用 prefetch、preload 去加载这些模块

prefetch：将来可能需要一些模块资源（一般是其他页面的代码），在核心代码加载完成之后带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（当前页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：

```
JavaScript
document.getElementById('btn1').onclick = function() {import(
  /* webpackChunkName: "btnChunk"
/
*  /* webpackPrefetch: true*/'./module1.js'
  ).then(fn => fn.default());
}
```

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，页面首次加载的时候，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。如果每个页面的代码中都有这些公开资源，就会导致资源的浪费。在每一个页面下都会加载重复的公共资源，一是会浪费用户的流量，二是不利于项目的性能，造成页面加载缓慢，影响用户体验。

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，在访问页面的时候，就可以一直使用浏览器缓存中的资源

webpack 里面通过 splitChunks 来分割代码

```
JavaScript
module.exports = {//...
  optimization: {
    splitChunks: {
      chunks: 'async', // 值有 all，async 和 initial
      minSize: 20000, // 生成 chunk 的最小体积（以 bytes 为单位）。
      minRemainingSize: 0,
      minChunks: 1, // 拆分前必须共享模块的最小 chunks 数。
```

```
      maxAsyncRequests: 30, // 按需加载时的最大并行请求数。
      maxInitialRequests: 30, // 入口点的最大并行请求数。
      enforceSizeThreshold: 50000,
      cacheGroups: {
        defaultVendors: {
          test: /[\/]node_modules[\/]/,  //第三方模块拆出来
          priority: -10,
          reuseExistingChunk: true,
        },
        util.vendors: {
          test: /[\/]utils[\/]/, //公共模块拆出来
          minChunks: 2,
          priority: -20,
          reuseExistingChunk: true,
        },
      },
    },
  },
};
```

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：

1. 只对 `ESM` 生效
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。
4. 只能处理 `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 `purgecss-webpack-plugin` 插件来对 css 进行 tree shaking

```
JavaScript
const path = require("path");
const PurgecssPlugin = require("purgecss-webpack-plugin");
const glob = require("glob"); // 文件匹配模式module.exports =
{//...
  plugins: [
    ...new PurgeCSSPlugin({
      paths: glob.sync(`${PATH.src}/**/*`, { nodir: true }),
```

```
    })
// Add your plugins here// Learn more about plugins from
https://webpack.js.org/configuration/plugins/
  ],
};
```

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?

1. 当用户访问 web 站点的时候，会在 `request header` 中设置 `accept-encoding:gzip`，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 JS\CSS 等其他资源文件进行 `Gzip` 压缩后再传输到客户端，同时将 `response headers` 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。

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` 压缩，然后将 `Gzip` 资源放到服务器，当请求需要的时候直接将 `Gzip` 资源发送给客户端。

我们只需要安装 `compression-webpack-plugin` 并在 `plugins` 配置就可以了

```
JavaScript
const CompressionWebpackPlugin = require("compression-webpack-
plugin"); // 需要安装module.exports = {
  plugins: [new CompressionWebpackPlugin()
  ]
}
```

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。

```JavaScript
module.exports = {//方式1
  mode: 'production',
//方式2
  plugins: [// 开启 Scope Hoisting 功能new
webpack.optimize.ModuleConcatenationPlugin()
  ]
}
```

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 是在用户设备上进行存储的，并且浏览器对存储空间的管理可能会受到用户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，有以下几种常见的存储方式：

1. Cookie：Cookie 是一种存储在用户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，可以用于维持用户会话、记录用户偏好等功能。
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页中）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页，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。它可以用来缓存网页、脚本、样式表、图像等静态资源，以提高网页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 可以将网页、脚本、样式表、图像等静态资源缓存起来，以提高网页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：

```
JavaScript
module.exports = {// ...
  optimization: {
    runtimeChunk: 'single',
  },
};
```

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：

```
JavaScript
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {// ...
  plugins: [new HtmlWebpackPlugin({// ...
      chunks: ['runtime', 'app'],
    }),
  ],
};
```

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：

```JavaScript
import(/* webpackChunkName: "runtime" */
'./path/to/runtime').then((runtime) => {
  // 运行时 chunk 加载完成后的逻辑
});
```

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 `entry` 配置和 `import()`语法来实现。

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：

```JavaScript
module.exports = {// ...
  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：

```JavaScript
function loadRuntime() {return import('./runtime.js');
}

// 使用动态导入的方式加载运行时文件
loadRuntime().then(runtime => {// 运行时文件加载完成后的逻辑
});
```

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:

```
JavaScript
import(/* webpackChunkName: "runtime" */
'./path/to/runtime').then((runtime) => {
  // 运行时 chunk 加载完成后的逻辑
});
```

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?

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。

追问： `canvas 和 svg 在动态交互上有什么具体的区别?`

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户操作事件驱动， 比较方便， 但是同样的用户操作， 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？

在 `Table` 开发场景下，SVG 确实更适合进行事件绑定和用户操作事件驱动。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，实现用户交互。

而 `Canvas` 在处理用户操作事件驱动方面相对不太方便，因为 `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，实现与用户交互的感觉。然后通过监听 HTML 元素的事件，通过 JavaScript 判断用户操作的位置与 Canvas 上的图形元素是否相交，从而模拟出用户交互的效果。
- 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 上进行用户交互更加方便。这些库可以处理用户操作事件，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户交互，但相比于 SVG 来说，Canvas 的事件处理和用户交互仍然相对繁琐一些。所以在需要大量的用户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，实现用户交互。
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 `getPointer` 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 `dragging`、`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
HtmlWebpackPlugin	自动生成 HTML 文件，并将打包后的资源自动注入到 HTML 中。
MiniCssExtractPlugin	将 CSS 代码提取到单独的文件中，而不是内联到

点击图片可查看完整电子表格

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。
css-loader	解析 CSS 文件，处理 CSS 中的依赖关系，并将 CSS 转换为 JS 模块。

点击图片可查看完整电子表格

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，有几种常见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，例如`.componentA-button`和`.componentB-button`，这样可以避免命名冲突。

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。例如，`.button`表示一个块，`.button__icon`表示一个元素，`.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。一些常见的 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， 加载下一个路由页面的组件， 注销上一个路由页面的组件；

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页面的组件， 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。

需要当两个页面完成动画时间， 完成页面覆盖切换之后， 然后注销上一个路由页面的组件；

所以涉及到的知识点：

1. 如何做页面跳转拦截；
2. 如何在页面路由组件不跳转的同时， 加载下一个页面的组件；
3. 配置页面层级；
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）来仅渲染可见区域内的元素，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。

这些是一些常见的 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】

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 `v-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 `false`，`v-for` 仍然会对数据进行遍历和渲染。

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 `key` 的元素，此时 `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。

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：

```
XML
<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：

```
JavaScript
function render() {return _c('ul',null,
    _l(items, function(item) {return item.isActive ? _c('li',
null, _v(_s(item.name))) : _e();
    })
  );
}
```

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。

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 `v-for` 遍历的数据项发生变化时，Vue 会使用具有相同 `key` 的元素。在上面的例子中，如果 `item.isActive` 从 `true` 变为 `false`，Vue 会尝试复用之前的 `li` 元素，并在其上应用 `v-if` 条件。这可能会导致一些不符合预期的行为。

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-if` 和 `v-for`。如果需要根据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` 的过滤器来处理数据。或者，将条件判断放在外层元素上，内层元素使用 `v-for` 进行循环渲染，以确保每次渲染时都能正确地应用 `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。常见的缓存策略有：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 URLvar resources = ['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.createIndex('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。