

1. H5移动端适配

你的回答已经涵盖了一些H5移动端适配的常见方法和技术，整体方向是正确的，但可以进一步完善和补充。以下是对这个问题的更全面分析和讲解：

适配方法

- rem适配

- 原理是根据根元素（html）的字体大小来计算其他元素的尺寸。通过JavaScript动态设置html元素的font-size，然后在CSS中使用rem作为单位，这样元素的尺寸会随着根字体大小的变化而自适应。例如，如果根字体大小设置为16px，1rem就等于16px，若将根字体大小改为32px，1rem就变为32px。
- 优点是可实现等比例缩放，适配各种屏幕尺寸，缺点是需要手动计算和设置rem值，相对麻烦。

- 原理概述

rem（root em）是相对于根元素（<html>）字体大小的单位。通过设置根元素的字体大小，页面上所有使用rem作为单位的元素尺寸都会随之变化，从而实现页面在不同屏幕尺寸下的等比例缩放。postcss - pxtorem 是一个PostCSS插件，它可以在构建过程中自动将CSS中的px单位转换为rem单位，这样开发者在编写CSS时可以继续使用熟悉的px单位，由插件来完成转换工作，提高开发效率。

在Vue项目中配置rem + postcss - pxtorem 的详细步骤

1. 创建Vue项目

如果你还没有Vue项目，可以使用Vue CLI来创建一个新的项目。首先确保你已经安装了Vue CLI，如果没有安装，可以使用以下命令进行安装：

```
npm install -g @vue/cli
```

然后创建一个新的Vue项目：

```
vue create rem-vue-demo  
cd rem-vue-demo
```

2. 安装依赖

安装 `postcss - pxtorem` 和 `lib - flexible`（用于动态设置根元素字体大小）：

```
npm install postcss-pxtorem lib-flexible --save-dev
```

3. 引入 `lib - flexible`

在 `main.js` 中引入 `lib - flexible`：

```
import 'lib-flexible'
```

`lib - flexible` 会根据不同的屏幕宽度动态设置根元素（`<html>`）的字体大小。

4. 配置 `postcss - pxtorem`

在项目根目录下找到或创建 `postcss.config.js` 文件，添加以下配置：

```
module.exports = {
  plugins: {
    'postcss-pxtorem': {
      rootValue: 16, // 根元素字体大小，这里设置为 16px，可根据实际情况调整
      propList: ['*'], // 需要转换的属性列表，'*' 表示所有属性都转换
      selectorBlackList: [], // 需要忽略的选择器列表，可根据需求添加，如
        ['body'] 表示 body 元素下的 px 不转换
      minPixelValue: 2, // 最小转换像素值，小于该值的 px 不转换
      exclude: /node_modules/i // 排除 node_modules 目录下的文件
    }
  }
}
```

配置解释：

- `rootValue`：根元素的字体大小，这里设置为 16px，即 1rem 等于 16px。在实际开发中，你可以根据设计稿的尺寸进行调整。
- `propList`：指定哪些 CSS 属性需要进行 `px` 到 `rem` 的转换，`['*']` 表示所有属性都转换。
- `selectorBlackList`：指定需要忽略的选择器，这些选择器下的 `px` 不会被转换。
- `minPixelValue`：设置最小的转换像素值，小于该值的 `px` 不会被转换。
- `exclude`：排除某些目录下的文件，通常会排除 `node_modules` 目录。

5. 编写 CSS 代码

现在你可以在组件的 CSS 中正常使用 `px` 单位，`postcss - pxtoem` 会在构建过程中自动将其转换为 `rem` 单位。例如，在 `App.vue` 中：

```
<template>
  <div id="app">
    <div class="box">这是一个测试盒子</div>
  </div>
</template>

<style scoped>
.box {
  width: 200px;
  height: 100px;
  background-color: #f0f0f0;
  margin: 20px auto;
}
</style>
```

在构建后，上述 CSS 中的 `px` 单位会被自动转换为 `rem` 单位。

6. 运行项目

使用以下命令运行项目：

```
npm run serve
```

打开浏览器查看效果，页面会根据不同的屏幕尺寸进行自适应布局。

注意事项

- **设计稿与根字体大小的对应关系：**在实际开发中，需要根据设计稿的尺寸来确定 `rootValue` 的值。例如，如果设计稿是 750px 宽度，通常可以将 `rootValue` 设置为 75，这样在设计稿上量取的 `px` 值可以直接写在 CSS 中，插件会自动转换为合适的 `rem` 值。
- **兼容性问题：**虽然 `rem` 适配方案在大多数现代浏览器中都能正常工作，但在一些旧版本的浏览器中可能存在兼容性问题，需要进行适当的测试和处理。

通过以上步骤，你就可以在 Vue 项目中成功配置 `rem + postcss - pxtoem` 实现移动端适配。

- **vw/vh适配**

- vw是视窗宽度的百分比，vh是视窗高度的百分比。比如，1vw表示屏幕宽度的1%，1vh表示屏幕高度的1%。使用vw/vh可以很方便地实现元素的宽度和高度根据屏幕尺寸自适应。
- 优点是简单直观，能很好地适应不同屏幕宽度，缺点是在一些特殊布局或需要精确控制元素位置时可能不太方便，而且部分旧浏览器兼容性略差。
- **媒体查询**
 - 可以根据不同的媒体类型（如屏幕、打印等）和媒体特性（如屏幕宽度、分辨率等）来应用不同的CSS样式。例如，可以针对不同的屏幕宽度区间设置不同的样式，以实现适配。
 - 优点是可以针对特定的屏幕尺寸范围进行精细的样式调整，缺点是需要编写大量的媒体查询代码，维护起来可能较复杂，而且如果屏幕尺寸变化范围很广，可能需要编写很多组媒体查询。
- **flex布局**
 - 是一种弹性布局模型，通过设置父元素为flex布局，可以方便地控制子元素的排列方式、对齐方式和伸缩比例等，使页面在不同屏幕尺寸下能自适应排列。
 - 优点是强大且灵活，能轻松实现各种复杂的布局自适应，缺点是对一些旧浏览器的兼容性有限，而且在某些特殊布局需求下可能不够直观。

相关库和工具

- **lib-flexible**
 - 是手淘团队开发的一个用于H5页面适配的库。它主要通过动态修改html元素的font-size来实现rem适配，并且能根据不同的屏幕宽度进行合理的计算和设置，简化了rem适配的开发过程。
- **postcss-pxtorem**
 - 是一个PostCSS插件，它可以在构建过程中自动将CSS中的px单位转换为rem单位，大大提高了开发效率，减少了手动计算rem值的工作量。
- **Autoprefixer**
 - 是一个CSS自动添加浏览器前缀的工具。由于不同浏览器对CSS属性的支持和实现方式不同，有些CSS属性需要添加特定的浏览器前缀才能在不同浏览器中正常显示。Autoprefixer可以根据配置的浏览器兼容列表，自动为CSS属性添加相应的前缀，如-webkit-、-moz-、-ms-等，提高了CSS的兼容性。

2. JS和客户端交互方法

你的回答提到了在实际项目中借助客户端提供方法进行交互以及后续数据记录的情况，有一定的实践体现，但可以更加系统全面地展开阐述JS和客户端交互的方法。下面为你

详细介绍常见的交互方式：

1. 原生WebView注入对象交互

原理

在iOS和Android等客户端的WebView（用于在移动应用中显示网页内容的组件）中，可以向网页注入原生对象。这样在网页中的JavaScript代码就能直接调用这些原生对象的方法，实现与客户端的交互。

示例

- Android端：

```
// Java代码
webView.addJavascriptInterface(new JsInterface(), "AndroidObj");

public class JsInterface {
    @JavascriptInterface
    public void showToast(String message) {
        Toast.makeText(MainActivity.this, message,
            Toast.LENGTH_SHORT).show();
    }
}
```

在JavaScript中调用：

```
// 调用Android注入的对象方法
window.AndroidObj.showToast('Hello from JS');
```

- iOS端：

```
// Objective - C代码
WKUserContentController *userContentController =
[[WKUserContentController alloc] init];
WKWebViewConfiguration *config = [[WKWebViewConfiguration alloc] init];
config.userContentController = userContentController;
WKWebView *webView = [[WKWebView alloc] initWithFrame:self.view.bounds
configuration:config];
[userContentController addScriptMessageHandler:self name:@"iOSObj"];

- (void)userContentController:(WKUserContentController
*)userContentController didReceiveScriptMessage:(WKScriptMessage
```

```

*)message {
    if ([message.name isEqualToString:@"iOSObj"]) {
        NSString *msg = message.body;
        NSLog(@"Received message from JS: %@", msg);
    }
}

```

在JavaScript中调用：

```

// 调用iOS注入的对象方法
window.webkit.messageHandlers.iOSObj.postMessage('Hello from JS');

```

2. 通过URL Scheme交互

原理

URL Scheme是一种类似于URL的链接格式，用于在应用间进行跳转和传递数据。JavaScript可以通过修改当前页面的URL为特定的URL Scheme，客户端的WebView捕获到这个URL Scheme后，进行相应的处理。

示例

在JavaScript中触发URL Scheme：

```

function callNativeFunction() {
    window.location.href = 'myapp://action?
param1=value1&param2=value2';
}

```

客户端捕获URL Scheme并处理：

- **Android端**：在WebViewClient的 `shouldOverrideUrlLoading` 方法中拦截URL Scheme进行处理。

```

webView.setWebViewClient(new WebViewClient() {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        if (url.startsWith("myapp://")) {
            // 解析URL参数并处理
            return true;
        }
        return super.shouldOverrideUrlLoading(view, url);
    }
}

```

```
}  
});
```

- **iOS端**：在WKUIDelegate的

`webView:decidePolicyForNavigationAction:decisionHandler:` 方法中拦截 URL Scheme 进行处理。

```
- (void)webView:(WKWebView *)webView decidePolicyForNavigationAction:  
(WKNavigationAction *)navigationAction decisionHandler:(void (^)(  
(WKNavigationActionPolicy))decisionHandler {  
    NSURL *url = navigationAction.request.URL;  
    if ([url.scheme isEqualToString:@"myapp"]) {  
        // 解析URL参数并处理  
        decisionHandler(WKNavigationActionPolicyCancel);  
        return;  
    }  
    decisionHandler(WKNavigationActionPolicyAllow);  
}
```

3. 消息传递机制（如postMessage）

原理

HTML5提供了 `window.postMessage` 方法，可用于在不同窗口或框架之间传递消息。在客户端WebView和网页之间也能借助这个机制进行通信。客户端可以监听网页发送的消息，网页也能监听客户端发送的消息。

示例

- 网页向客户端发送消息：

```
window.postMessage({type: 'share', data: 'Some share data'}, '*');
```

- 客户端监听消息（以Android为例）：

```
webView.evaluateJavascript("window.addEventListener('message',  
function(event) { console.log('Received message from client: ' +  
JSON.stringify(event.data)); });", null);
```

3.如何进行组件的二次封装：

明确封装目的

在开始二次封装之前，要清楚为什么要进行封装。常见的目的包括：

- **提高复用性**：将一些常用的功能或样式封装成一个通用组件，方便在多个项目或页面中使用。
- **简化使用**：对一些复杂的原生组件或第三方组件进行简化，使其使用起来更加方便快捷。
- **统一风格**：确保项目中使用的组件具有统一的外观和交互风格。

二次封装的步骤

1. 选择合适的基础组件

可以是原生 HTML 元素、框架自带的组件，或者第三方开源组件。例如，在 Vue 项目中，可能会对 `input` 元素、`el-input`（Element UI 组件）等进行二次封装。

2. 分析基础组件的属性、方法和事件

了解基础组件提供了哪些属性可以配置、哪些方法可以调用以及会触发哪些事件。这有助于在二次封装时进行合理的继承和扩展。

3. 定义新组件的属性、方法和事件

- **属性**：根据封装目的，定义新组件需要接收的属性。可以对基础组件的属性进行筛选和扩展。例如，封装一个 `Input` 组件时，可以定义 `placeholder`、`value`、`disabled` 等属性，同时还可以添加一些自定义属性，如 `maxLength` 来限制输入的最大长度。

```
<template>
  <input :value="value" :placeholder="placeholder" :disabled="disabled"
  @input="handleInput">
</template>

<script>
export default {
  props: {
    value: {
      type: String,
      default: ''
    },
  },
}
```



```

placeholder: {
  type: String,
  default: '请输入内容'
},
disabled: {
  type: Boolean,
  default: false
},
maxLength: {
  type: Number,
  default: 50
}
},
methods: {
  handleInput(event) {
    this.$emit('input', event.target.value.slice(0, this.maxLength));
  }
}
}
</script>

```

- **方法：**可以定义新组件自己的方法，也可以对基础组件的方法进行封装和调用。例如，封装一个 `Modal` 组件时，可以定义 `open` 和 `close` 方法来控制模态框的显示和隐藏。

```

<template>
  <div v-if="visible" class="modal">
    <!-- 模态框内容 -->
    <button @click="close">关闭</button>
  </div>
</template>

```

```

<script>
export default {
  data() {
    return {
      visible: false
    };
  },
  methods: {
    open() {
      this.visible = true;
    },
    close() {
      this.visible = false;
    }
  }
}

```

```
}  
}  
</script>
```

- **事件**：要处理好基础组件触发的事件，并可以根据需要触发新的自定义事件。例如，在封装 `Input` 组件时，将基础 `input` 元素的 `input` 事件进行处理后，触发自定义的 `input` 事件，方便父组件监听。

4. 样式定制

- **继承基础组件样式**：可以通过 CSS 类名或样式绑定的方式继承基础组件的样式。例如，在封装 `Button` 组件时，可以使用基础按钮的类名，同时添加自定义的类名来修改样式。

```
<template>  
  <button :class="['base-button', customClass]">{{ label }}</button>  
</template>  
  
<script>  
export default {  
  props: {  
    label: {  
      type: String,  
      default: '按钮'  
    },  
    customClass: {  
      type: String,  
      default: ''  
    }  
  }  
}  
</script>  
  
<style scoped>  
.base-button {  
  /* 基础按钮样式 */  
  padding: 8px 16px;  
  background-color: #007bff;  
  color: white;  
  border: none;  
  border-radius: 4px;  
}  
</style>
```

- **提供样式配置选项**：可以通过属性来控制组件的样式，使组件的外观更加灵活。例

如，定义一个 `size` 属性，根据不同的值显示不同大小的按钮。

5. 测试和优化

- 对二次封装的组件进行全面的测试，确保其功能正常、样式符合预期，并且在不同的使用场景下都能稳定工作。
- 根据测试结果进行优化，例如优化性能、修复 bug、改进用户体验等。

总结

二次封装组件需要综合考虑属性、方法、事件和样式的设计，以实现提高复用性、简化使用和统一风格的目的。通过以上步骤，可以创建出高质量的二次封装组件。

4.vue插槽slot

插槽（Slot）是 Vue 中一个非常实用的特性，它允许你在封装组件时为组件提供灵活的内容插入机制，让组件可以根据不同的使用场景插入不同的内容，提高组件的复用性和灵活性。下面从概念、分类、使用方法等方面详细介绍插槽。

插槽的基本概念

想象一下，你封装了一个通用的卡片组件，这个卡片组件有固定的样式和布局，但卡片内部的具体内容可能会根据不同的页面需求而变化。插槽就像是在这个卡片组件上预留的“空位”，在使用这个卡片组件时，你可以往这些“空位”里插入自定义的内容，比如图片、文字、按钮等。

插槽的分类及使用方法

1. 匿名插槽（默认插槽）

- 概念：**匿名插槽是最基本的插槽类型，它没有具体的名称，用于在组件中插入单个默认的内容区域。
- 使用方法**
 - 定义组件：**在组件模板中使用 `<slot></slot>` 标签来定义插槽位置。

```
<!-- CustomCard.vue -->
<template>
  <div class="custom-card">
    <div class="card-header">卡片头部</div>
```

```

    <slot></slot> <!-- 匿名插槽 -->
    <div class="card-footer">卡片底部</div>
  </div>
</template>

<style scoped>
.custom-card {
  border: 1px solid #ccc;
  padding: 10px;
}
.card-header {
  font-weight: bold;
}
.card-footer {
  color: #999;
}
</style>

```

- ****使用组件****: 在使用该组件时, 直接在组件标签内插入要显示的内容, 这些内容会被插入到插槽的位置。

```

<template>
  <div>
    <CustomCard>
      <p>这是卡片的主体内容。</p>
    </CustomCard>
  </div>
</template>

<script>
import CustomCard from './CustomCard.vue';

export default {
  components: {
    CustomCard
  }
};
</script>

```

2. 具名插槽

- **概念**: 当组件需要多个不同的内容插入位置时, 就可以使用具名插槽。每个插槽有一个唯一的名称, 通过名称来区分不同的插入位置。
- **使用方法**

- **定义组件**：在组件模板中使用 `<slot name="插槽名称"></slot>` 标签来定义具名插槽。

```
<!-- CustomLayout.vue -->
<template>
  <div class="custom-layout">
    <header>
      <slot name="header"></slot> <!-- 具名插槽：头部 -->
    </header>
    <main>
      <slot></slot> <!-- 匿名插槽：主体内容 -->
    </main>
    <footer>
      <slot name="footer"></slot> <!-- 具名插槽：底部 -->
    </footer>
  </div>
</template>

<style scoped>
.custom-layout {
  display: flex;
  flex-direction: column;
  height: 100vh;
}
header {
  background-color: #333;
  color: white;
  padding: 10px;
}
main {
  flex: 1;
  padding: 20px;
}
footer {
  background-color: #333;
  color: white;
  padding: 10px;
  text-align: center;
}
</style>
```

- ****使用组件****：在使用该组件时，使用 `<template #插槽名称>`（Vue 3 语法，Vue 2 中使用 `<template slot="插槽名称">`）来指定要插入到哪个具名插槽中。

```
<template>
```

```

<div>
  <CustomLayout>
    <template #header>
      <h1>页面标题</h1>
    </template>
    <p>这是页面的主体内容。</p>
    <template #footer>
      <p>版权所有 &copy; 2024</p>
    </template>
  </CustomLayout>
</div>
</template>

<script>
import CustomLayout from './CustomLayout.vue';

export default {
  components: {
    CustomLayout
  }
};
</script>

```

3. 作用域插槽

- **概念：**作用域插槽允许在父组件中访问子组件的数据。子组件可以将自己的数据通过插槽传递给父组件，父组件可以根据这些数据进行不同的渲染。
- **使用方法**
 - **定义组件：**在组件模板中，通过 `<slot :数据名称="数据值"></slot>` 的方式将数据传递给插槽。

```

<!-- UserList.vue -->
<template>
  <ul>
    <li v-for="user in users" :key="user.id">
      <slot :user="user"></slot> <!-- 作用域插槽，传递 user 数据 -->
    </li>
  </ul>
</template>

<script>
export default {
  data() {
    return {

```

```
      users: [
        { id: 1, name: '张三' },
        { id: 2, name: '李四' },
        { id: 3, name: '王五' }
      ]
    };
  }
};
</script>
```

- ****使用组件****: 在使用该组件时, 通过 `<template #default="{ 数据名称 }">` (Vue 3 语法, Vue 2 中使用 `<template slot-scope="{ 数据名称 }">`) 来接收子组件传递的数据, 并使用这些数据进行渲染。

```
<template>
  <div>
    <UserList>
      <template #default="{ user }">
        <span>{{ user.name }}</span>
      </template>
    </UserList>
  </div>
</template>

<script>
import UserList from './UserList.vue';

export default {
  components: {
    UserList
  }
};
</script>
```

总结

插槽是 Vue 中实现组件灵活复用的重要特性, 通过匿名插槽、具名插槽和作用域插槽, 你可以在封装组件时为不同的内容插入需求提供解决方案, 使组件更加通用和强大

5.静态组件和动态组件区别

在 Vue 中, **动态组件**和**静态组件**的核心区别在于 **组件的渲染时机和灵活性**。以下是详

细对比和实际应用场景分析：

一、核心区别对比

特性	静态组件	动态组件
定义方式	在模板中直接使用组件标签（如 <code><ComponentA/></code> ）	通过 <code><component :is="variable"></code> 动态绑定
渲染时机	编译时确定，直接渲染	运行时根据条件动态决定渲染哪个组件
灵活性	固定，无法切换	高度灵活，可通过变量或逻辑动态切换
生命周期	初始化时创建，销毁时触发销毁钩子	切换时可能触发销毁/重建（除非使用 <code><keep-alive></code> ）
性能影响	高效，无额外开销	频繁切换时可能产生性能开销（需配合缓存优化）

二、使用场景示例

1. 静态组件：固定结构

```
<template>
  <!-- 明确的组件结构 -->
  <Header />
  <ProductList />
  <Footer />
</template>
```

- 适用场景：**页面的固定模块（如导航栏、底部信息）。

2. 动态组件：条件渲染

```
<template>
  <!-- 根据状态动态切换组件 -->
```



```
<component :is="currentTabComponent" />
</template>

<script>
export default {
  data() {
    return {
      currentTab: 'Home',
      tabs: { Home: 'HomeComponent', About: 'AboutComponent' }
    };
  },
  computed: {
    currentTabComponent() {
      return this.tabs[this.currentTab];
    }
  }
};
</script>
```

- **适用场景：**
 - 标签页切换（如不同功能的子页面）
 - 根据权限显示不同组件（如管理员与普通用户视图）
 - 动态加载异步组件（如按需加载弹窗）

三、性能优化技巧

1. 使用 `<keep-alive>` 缓存动态组件

```
<template>
  <!-- 缓存组件状态，避免重复渲染 -->
  <keep-alive>
    <component :is="currentComponent" />
  </keep-alive>
</template>
```

- **效果：**保留组件状态（如表单输入内容），避免重复触发 `created` / `mounted` 生命周期。

2. 按需加载异步组件

```
// Vue 3 语法
const AsyncComponent = defineAsyncComponent(() =>
import('./AsyncComponent.vue'));
```

- 效果：减少初始加载体积，提升首屏速度。

四、底层原理剖析

1. 静态组件编译结果

静态组件在编译阶段会被直接解析为具体的组件引用：

```
// 编译后的渲染函数
import ComponentA from './ComponentA.vue';
export function render() {
  return h(ComponentA);
}
```

2. 动态组件编译结果

动态组件通过 `resolveDynamicComponent` 函数在运行时决定渲染内容：

```
// 编译后的渲染函数
export function render() {
  return h(resolveDynamicComponent(this.currentComponent));
}
```

五、面试回答模板

问题：动态组件和静态组件有什么区别？

回答示例：

静态组件在编译时确定，直接通过标签引用（如 `<Header />`），适合结构固定的场景，性能更高。

动态组件通过 `<component :is>` 动态绑定变量，根据条件渲染不同组件（如标签页切换），灵活性更强。

为了优化动态组件性能，通常会结合 `<keep-alive>` 缓存组件状态，或使用异步组件按需加载。

例如，在管理后台项目中，我通过动态组件实现了权限适配的视图切换，结合 `keep-alive` 缓存高频操作的页面，提升了用户体验。

6. 封装axios接口与网关的调用，包装请求参数、处理错误码、添加接口签名、白名单功能

以下是一个完整的示例，展示如何封装 Axios 接口与网关的调用，包含包装请求参数、处理错误码、添加接口签名和白名单功能。

1. 项目准备

首先，确保你已经安装了 Axios。如果使用 npm，可以通过以下命令安装：

```
npm install axios
```

2. 封装 Axios

创建一个 `axiosWrapper.js` 文件，用于封装 Axios 并实现所需功能。

```
import axios from 'axios';
import CryptoJS from 'crypto-js'; // 用于生成签名，可根据实际需求替换为其他加密库

// 配置信息
const config = {
  baseURL: 'https://your-gateway-url.com', // 网关基础 URL
  timeout: 5000, // 请求超时时间
  secretKey: 'your-secret-key', // 用于接口签名的密钥
  whitelist: ['/public/api1', '/public/api2'] // 白名单接口列表
};

// 创建 Axios 实例
const instance = axios.create({
  baseURL: config.baseURL,
```

```

    timeout: config.timeout
  });

// 生成接口签名
const generateSignature = (data, secretKey) => {
  const sortedKeys = Object.keys(data).sort();
  let signStr = '';
  sortedKeys.forEach(key => {
    signStr += `${key}=${data[key]}&`;
  });
  signStr += `secret=${secretKey}`;
  return CryptoJS.MD5(signStr).toString();
};

// 请求拦截器
instance.interceptors.request.use(config => {
  const { url, data = {} } = config;
  // 检查是否在白名单
  const isInWhiteList = config.whiteList.some(item =>
url.includes(item));
  if (!isInWhiteList) {
    // 添加公共参数
    const timestamp = new Date().getTime();
    const newData = {
      ...data,
      timestamp
    };
    // 生成签名
    const signature = generateSignature(newData, config.secretKey);
    newData.signature = signature;
    config.data = newData;
  }
  return config;
}, error => {
  console.error('请求拦截器错误:', error);
  return Promise.reject(error);
});

// 响应拦截器
instance.interceptors.response.use(response => {
  const { data } = response;
  // 处理错误码
  if (data.code !== 200) {
    switch (data.code) {
      case 401:
        console.error('未授权, 请重新登录');
        break;

```

```

        case 403:
            console.error('禁止访问');
            break;
        case 500:
            console.error('服务器内部错误');
            break;
        default:
            console.error(`未知错误码: ${data.code}`);
    }
    return Promise.reject(new Error(data.message || '请求失败'));
}
return data;
}, error => {
    console.error('响应拦截器错误:', error);
    return Promise.reject(error);
});

// 封装请求方法
const request = {
    get(url, params = {}) {
        return instance.get(url, { params });
    },
    post(url, data = {}) {
        return instance.post(url, data);
    },
    put(url, data = {}) {
        return instance.put(url, data);
    },
    delete(url, params = {}) {
        return instance.delete(url, { params });
    }
};

export default request;

```

3. 使用封装的 Axios

在需要调用接口的地方引入封装好的 `request` 对象并使用。

```

import request from './axiosWrapper';

// 示例: 发送 GET 请求
request.get('/api/getData', { id: 1 })
    .then(response => {

```

```
        console.log('请求成功:', response);
    })
    .catch(error => {
        console.error('请求失败:', error);
    });

// 示例: 发送 POST 请求
request.post('/api/submitData', { name: 'John', age: 30 })
    .then(response => {
        console.log('请求成功:', response);
    })
    .catch(error => {
        console.error('请求失败:', error);
    });
```

代码解释

1. **配置信息**: 定义了网关的基础 URL、超时时间、签名密钥和白名单接口列表。
2. **Axios 实例**: 使用 `axios.create` 创建一个 Axios 实例, 并设置基础 URL 和超时时间。
3. **请求拦截器**:
 - 检查请求的 URL 是否在白名单中。
 - 如果不在白名单, 添加公共参数 (如时间戳), 并生成接口签名。
 - 将签名和公共参数添加到请求数据中。
4. **响应拦截器**:
 - 检查响应数据的错误码。
 - 根据不同的错误码进行相应的处理, 并抛出错误。
5. **封装请求方法**: 提供 `get`、`post`、`put` 和 `delete` 方法, 方便调用。

通过以上步骤, 你可以实现一个功能完善的 Axios 封装, 用于与网关进行接口调用。

7.Vite相较于Webpack的优势

Vite 是一种新型的前端构建工具, 由 Vue.js 核心团队成员尤雨溪开发, 和传统的 Webpack 相比, 它在多个方面展现出显著优势, 以下为你详细介绍:

1. 开发服务器启动速度快

- **Vite 原理：**Vite 利用现代浏览器原生 ES 模块的支持，在开发环境下无需打包，直接启动一个开发服务器。当浏览器请求模块时，Vite 会按需编译并提供这些模块。例如，在一个大型项目中，Vite 可以在瞬间启动开发服务器，让开发者立即开始编码和调试。
- **Webpack 情况：**Webpack 在启动开发服务器时，需要对整个项目进行打包处理，将所有的模块打包成一个或多个文件。对于大型项目，这个过程可能会非常耗时，有时甚至需要等待几十秒到几分钟。

2. 热更新速度快

- **Vite 原理：**Vite 的热更新（HMR）是基于 ES 模块的，当一个模块发生变化时，Vite 可以精确地更新该模块，而不需要重新构建整个应用。这使得热更新的速度极快，几乎是瞬间完成，开发者可以立即看到代码修改后的效果。
- **Webpack 情况：**Webpack 的热更新需要重新构建受影响的模块及其依赖项，对于复杂的项目，热更新可能会有明显的延迟，影响开发效率。

3. 按需编译

- **Vite 原理：**Vite 采用按需编译的方式，只有当浏览器请求某个模块时，才会对该模块进行编译。这种方式减少了不必要的编译工作，提高了开发效率。例如，在开发一个包含大量页面的应用时，开发者可能只关注当前正在开发的页面，Vite 只会编译该页面所依赖的模块。
- **Webpack 情况：**Webpack 在开发环境下通常会对整个项目进行预编译，即使某些模块在当前开发过程中并不会被使用，这会浪费一定的时间和资源。

4. 简单的配置

- **Vite 原理：**Vite 的配置相对简单，对于大多数项目，只需要进行少量的配置即可满足需求。它提供了一些默认的配置选项，并且支持使用 JavaScript 或 TypeScript 进行配置。例如，在创建一个 Vue 项目时，Vite 只需要几行配置就可以完成基本的项目搭建。
- **Webpack 情况：**Webpack 的配置相对复杂，需要了解大量的概念和配置项，如 loader、plugin 等。对于初学者来说，配置 Webpack 可能是一个挑战，而且在项目规模增大时，配置文件会变得越来越复杂，难以维护。

5. 原生支持 ES 模块

- **Vite 原理：**Vite 原生支持 ES 模块，这使得它能够更好地与现代 JavaScript 生态系统集成。开发者可以直接使用 ES 模块的语法进行开发，无需额外的转换。

- **Webpack 情况：**Webpack 虽然也支持 ES 模块，但在处理 ES 模块时需要使用特定的 loader 进行转换，这增加了配置的复杂性。

6. 对不同框架的支持友好

- **Vite 原理：**Vite 对多种前端框架（如 Vue、React、Svelte 等）提供了良好的支持，并且针对不同框架有专门的插件和模板。开发者可以快速搭建不同框架的项目，并且享受到 Vite 的高性能开发体验。
- **Webpack 情况：**Webpack 虽然也可以用于各种前端框架的开发，但需要针对不同的框架进行相应的配置和调整，相对来说不够便捷。

8.webpack的生命周期

Webpack 是一个强大的模块打包工具，它的打包过程包含了一系列钩子（Hook），这些钩子构成了 Webpack 的生命周期。开发者可以通过这些钩子在不同的阶段执行自定义逻辑，如代码压缩、资源处理等。下面为你详细介绍 Webpack 的生命周期。

整体流程概述

Webpack 的打包过程主要分为以下几个大的阶段：初始化、编译、生成资源、输出文件。

各阶段及相关钩子

1. 初始化阶段

- **entry-option**：在 webpack 选项中的 **entry** 配置项处理完成之后执行。这个钩子允许你在入口配置确定后做一些额外的操作，比如动态修改入口文件。

```
const webpack = require('webpack');
const compiler = webpack({
  // webpack 配置
});

compiler.hooks.entryOption.tap('MyPlugin', (context, entry) => {
  console.log('Entry options have been processed:', entry);
  // 可以在这里动态修改 entry
  return true;
});
```


- **after-plugins** : 在所有配置的插件都被应用之后执行。此时可以对已经应用的插件进行一些检查或其他操作。

```
compiler.hooks.afterPlugins.tap('MyPlugin', (compiler) => {  
  console.log('All plugins have been applied.');
```

- **after-resolvers** : 在解析器 (resolver) 设置完成之后执行。解析器用于解析模块路径, 这个钩子可用于自定义解析器的行为。

```
compiler.hooks.afterResolvers.tap('MyPlugin', (compiler) => {  
  console.log('Resolvers have been set up.');
```

2. 编译阶段

- **compile** : 在开始编译之前触发。可以在这里做一些编译前的准备工作, 比如初始化一些变量。

```
compiler.hooks.compile.tap('MyPlugin', (params) => {  
  console.log('Compilation is about to start.');
```

- **make** : 开始构建模块之前触发。这个钩子允许你在模块构建开始前对模块进行一些预处理。

```
compiler.hooks.make.tapAsync('MyPlugin', (compilation, callback) => {  
  console.log('Module building is about to start.');
```

- **after-compile** : 编译完成之后触发。此时可以对编译后的结果进行一些检查或处理。

```
compiler.hooks.afterCompile.tap('MyPlugin', (compilation) => {  
  console.log('Compilation is finished.');
```

3. 生成资源阶段

- **should-emit** : 在资源生成之后、输出之前触发。可以在这里决定是否要输出这

些资源。

```
compiler.hooks.shouldEmit.tap('MyPlugin', (compilation) => {  
  // 根据某些条件决定是否输出资源  
  return true;  
});
```

- **emit**：在生成资源并即将输出到文件系统之前触发。可以在这里对生成的资源进行最后的修改。

```
compiler.hooks.emit.tapAsync('MyPlugin', (compilation, callback) => {  
  console.log('About to emit assets.');
```

// 可以修改 compilation.assets 中的资源

```
  callback();  
});
```

- **after-emit**：在资源输出到文件系统之后触发。可以在这里做一些资源输出后的清理或记录工作。

```
compiler.hooks.afterEmit.tapAsync('MyPlugin', (compilation, callback)  
=> {  
  console.log('Assets have been emitted.');
```

callback();

```
});
```

4. 完成阶段

- **done**：整个打包过程完成之后触发。可以在这里做一些打包完成后的操作，比如发送通知。

```
compiler.hooks.done.tap('MyPlugin', (stats) => {  
  console.log('Webpack build is done.');
```

```
});
```

- **failed**：如果打包过程中出现错误，这个钩子会被触发。可以在这里进行错误处理和记录。

```
compiler.hooks.failed.tap('MyPlugin', (error) => {  
  console.error('Webpack build failed:', error);  
});
```

总结

Webpack 的生命周期钩子提供了丰富的扩展点，开发者可以根据自己的需求在不同的阶段插入自定义逻辑，从而实现各种功能，如代码分割、资源压缩、自定义构建流程等。通过监听这些钩子，开发者能够更好地控制 Webpack 的打包过程。

9.Webpack 的缓存机制是如何工作的？

Webpack 的缓存机制旨在提高构建性能，避免在每次构建时重复处理未发生变化的模块，从而显著减少构建时间。下面详细介绍 Webpack 缓存机制的工作原理、相关配置及不同类型的缓存。

工作原理概述

Webpack 缓存机制的核心思想是识别哪些模块在两次构建之间没有发生变化，然后直接复用之前构建的结果，而不是重新处理这些模块。它通过以下几个关键步骤来实现：

- 记录模块信息：**在首次构建时，Webpack 会记录每个模块的相关信息，如模块的内容哈希值、依赖关系等。
- 对比模块信息：**在后续构建时，Webpack 会重新计算模块的信息，并与之前记录的信息进行对比。
- 复用缓存结果：**如果某个模块的信息没有变化，Webpack 会直接使用之前缓存的构建结果，而不需要重新编译该模块。

缓存类型及配置

1. 持久化缓存（Persistent Caching）

从 Webpack 5 开始，引入了持久化缓存机制，它将构建结果存储在磁盘上，使得在不同的构建会话之间也能复用缓存。

配置示例：

```
// webpack.config.js
module.exports = {
  // 开启持久化缓存
  cache: {
    type: 'filesystem', // 使用文件系统存储缓存
    buildDependencies: {
      // 当 webpack 配置文件发生变化时，缓存将失效
    }
  }
}
```

```
    config: [__filename]
  }
}
};
```

工作原理：

- **缓存存储**：Webpack 会将每个模块的编译结果和相关元数据存储在磁盘的特定目录中。默认情况下，缓存目录位于 `node_modules/.cache/webpack`。
- **缓存验证**：在每次构建开始时，Webpack 会检查每个模块的内容哈希值。如果哈希值与缓存中的值相同，则认为该模块没有变化，直接使用缓存结果。
- **缓存失效**：当模块的内容发生变化、Webpack 配置文件修改或 `buildDependencies` 中指定的文件发生变化时，相关的缓存将失效，Webpack 会重新编译这些模块。

2. 内存缓存 (In - Memory Caching)

Webpack 在内存中也会维护一个缓存，用于在同一次构建过程中复用已经处理过的模块。

工作原理：

- **模块处理**：当 Webpack 处理一个模块时，它会首先检查内存缓存中是否已经存在该模块的处理结果。
- **缓存命中**：如果命中缓存，Webpack 会直接使用内存中的结果，避免重复处理。这种缓存机制在处理大型项目时特别有效，因为它可以减少模块处理的时间开销。

缓存与模块处理流程的结合

当 Webpack 开始构建项目时，会按照以下流程处理模块并利用缓存：

1. **模块解析**：Webpack 首先解析模块的路径，确定模块的位置。在这个过程中，内存缓存可以帮助快速定位已经解析过的模块。
2. **模块加载**：对于每个模块，Webpack 会检查磁盘缓存和内存缓存。如果缓存命中，直接使用缓存结果；否则，加载模块的源代码。
3. **模块转换**：如果模块需要进行转换（如使用 Babel 进行代码转换），Webpack 会对模块进行转换，并将转换结果存储在缓存中，以便后续复用。
4. **模块打包**：最后，Webpack 将处理好的模块打包成最终的文件。

总结

Webpack 的缓存机制通过持久化缓存和内存缓存的结合，有效地减少了构建时间。持久化缓存允许在不同构建会话之间复用缓存结果，而内存缓存则在同一次构建中提高了模块处理的效率。通过合理配置缓存选项，开发者可以进一步优化构建性能。

webpack 热更新原理

以下是面试中可能涉及 Webpack 热更新原理的常见问题及整合回答：

基础概念类

1. 什么是 Webpack 热更新？

Webpack 热更新（Hot Module Replacement, HMR）是一种在应用运行时动态替换、添加或删除模块，而无需完全刷新整个页面的技术。它能显著提升开发效率，因为开发者修改代码后，页面能快速反映这些更改，同时保留应用的当前状态，比如表单输入、滚动位置等。

2. Webpack 热更新和自动刷新有什么区别？

- **自动刷新**：当文件发生变化时，浏览器会重新加载整个页面，这意味着页面上的所有状态都会丢失，比如用户在表单中输入的数据、页面滚动位置等都需要重新设置。
- **热更新**：仅更新发生变化的模块，而不刷新整个页面，能够保留应用的当前状态，让开发者更高效地进行开发和调试。

原理机制类

1. 请简要描述 Webpack 热更新的原理。

Webpack 热更新的原理主要分为以下几个步骤：

- **初始化与监听**：项目启动时，Webpack 读取配置并添加热更新相关的插件和加载器，构建模块依赖图并为每个模块分配唯一 ID。 `webpack - dev - server` 启动，创建 Webpack 编译器实例，同时在浏览器和服务器之间建立 WebSocket 连接，用于实时通信。
- **文件变化检测与编译**：Webpack 持续监听项目文件系统，当检测到文件变化时，重新编译受影响的模块，并生成包含变化模块信息和更新代码的补丁文件。

- **更新信息推送**：服务器通过 WebSocket 连接将更新补丁的相关信息发送给浏览器。
- **模块更新处理**：浏览器端的 HMR 运行时接收到消息后，下载更新补丁文件，找到需要更新的模块，调用其 `update` 方法（如果模块提供）来更新模块内容，同时尽可能保留模块的状态，更新完成后执行回调函数通知开发者。

2. WebSocket 在 Webpack 热更新中起到什么作用？

WebSocket 在 Webpack 热更新中扮演着关键的通信桥梁角色。它为服务器和浏览器之间提供了实时、双向的通信通道。具体作用如下：

- **实时通知**：当文件发生变化，Webpack 重新编译生成更新补丁后，服务器可以通过 WebSocket 立即将更新信息（如变化模块的 ID、补丁文件的路径等）推送给浏览器。
- **高效通信**：相比传统的 HTTP 请求，WebSocket 连接建立后可以持续保持，避免了每次通信都要重新建立连接的开销，使得更新信息能够快速、高效地传输。

3. 热更新时，如何处理模块之间的依赖关系？

Webpack 在构建模块依赖图时，会记录每个模块的依赖信息。当某个模块发生变化时：

- **重新编译**：Webpack 会重新编译受影响的模块及其依赖的模块。
- **更新依赖**：在更新模块时，HMR 运行时会根据模块依赖图，递归地更新所有依赖该模块的其他模块，确保整个应用的依赖关系保持一致。例如，如果模块 A 依赖模块 B，当模块 B 发生变化时，模块 A 也会被更新以反映模块 B 的变化。

配置使用类

1. 如何在 Webpack 中配置热更新？

要在 Webpack 中配置热更新，需要完成以下两个主要步骤：

- **开启热更新选项**：在 `webpack.config.js` 的 `devServer` 配置中，将 `hot` 属性设置为 `true`，表示开启热更新功能。示例如下：

```
module.exports = {  
  // 其他配置...  
  devServer: {  
    hot: true  
  }  
};
```

- **引入热更新插件**：在 `plugins` 数组中添加 `webpack.HotModuleReplacementPlugin` 插件。示例如下：

```
const webpack = require('webpack');

module.exports = {
  // 其他配置...
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ]
};
```

2. 在模块中如何支持热更新？

在需要支持热更新的模块中，可以使用 `module.hot.accept` 方法来处理模块更新。示例代码如下：

```
// 模块代码
const someFunction = () => {
  console.log('This is a function.');
```

```
};
someFunction();

// 支持热更新
if (module.hot) {
  module.hot.accept(() => {
    // 模块更新后的处理逻辑
    console.log('Module updated!');
  });
}
```

在上述代码中，`if (module.hot)` 用于检查当前环境是否支持热更新，`module.hot.accept` 方法中的回调函数会在模块更新时被执行。

问题排查类

1. Webpack 热更新不生效可能有哪些原因？

Webpack 热更新不生效可能由以下原因导致：

- **配置错误**：未正确配置 `devServer.hot` 为 `true` 或未引入 `HotModuleReplacementPlugin` 插件。

- **模块代码问题**：模块中没有正确使用 `module.hot.accept` 方法处理更新，或者处理逻辑存在错误。
- **版本兼容性问题**：Webpack、`webpack-dev-server` 以及相关加载器和插件的版本不兼容，可能导致热更新功能无法正常工作。
- **文件监听问题**：某些环境下，文件系统的监听可能出现异常，导致 Webpack 无法及时检测到文件变化。可以尝试调整 `watchOptions` 配置来解决。

2. 如何调试 Webpack 热更新问题？

可以采用以下方法调试 Webpack 热更新问题：

- **查看控制台日志**：在浏览器和终端的控制台查看详细的日志信息，了解热更新过程中是否有错误或警告信息。
- **检查配置文件**：仔细检查 `webpack.config.js` 文件，确保热更新相关的配置正确无误。
- **使用调试工具**：可以使用浏览器的开发者工具（如 Chrome DevTools）来调试模块更新的过程，查看更新补丁的下载和应用情况。
- **版本检查**：确保 Webpack、`webpack-dev-server` 以及相关插件的版本兼容。可以尝试更新到最新稳定版本或回退到已知兼容的版本。

WebSocket通信实现模块更新的方式

在 Webpack 热更新机制里，WebSocket 通信在实现模块更新方面发挥着关键作用。下面详细介绍其实现模块更新的具体方式：

1. 建立 WebSocket 连接

在项目启动阶段，当使用 `webpack-dev-server` 且开启热更新功能（`hot: true`）时，`webpack-dev-server` 会在浏览器和服务器之间建立 WebSocket 连接。

- **服务器端**：`webpack-dev-server` 启动一个 WebSocket 服务器，等待浏览器的连接请求。这个服务器会持续监听文件系统的变化，并在文件更新时准备发送更新信息。
- **浏览器端**：浏览器中的 HMR 运行时（Hot Module Replacement Runtime）会主动发起 WebSocket 连接请求，与服务器建立双向通信通道。一旦连接成功，双方就可以实时交换数据。

2. 文件变化监听与编译

- **文件监听**：Webpack 会对项目中的文件系统进行监听，当检测到文件发生变化时，会重新编译受影响的模块。
- **生成更新补丁**：重新编译后，Webpack 会生成一个更新补丁，这个补丁包含了发生变化的模块信息（如模块 ID、更新后的代码等）以及更新所需的元数据。

3. 服务器发送更新消息

- **封装消息**：服务器将生成的更新补丁信息封装成特定格式的消息，通常包含变化模块的 ID 列表、补丁文件的 URL 等。
- **发送消息**：通过已建立的 WebSocket 连接，服务器将封装好的更新消息发送给浏览器。例如，消息可能是一个 JSON 对象，如下所示：

```
{
  "type": "hot-update",
  "hash": "123456abcdef",
  "modules": [
    {
      "id": 1,
      "url": "/hot-update/1.js"
    }
  ]
}
```

其中，`type` 表示消息类型为热更新，`hash` 是本次更新的唯一标识，`modules` 数组列出了发生变化的模块及其对应的补丁文件 URL。

4. 浏览器接收并处理更新消息

- **接收消息**：浏览器端的 HMR 运行时通过 WebSocket 接收到服务器发送的更新消息。
- **解析消息**：对收到的消息进行解析，提取出变化模块的信息和补丁文件的 URL。
- **下载补丁文件**：根据补丁文件的 URL，浏览器使用 HTTP 请求下载更新补丁文件。
- **更新模块**：
 - **查找模块**：HMR 运行时根据模块 ID 找到需要更新的模块。
 - **替换模块**：用下载的更新补丁文件中的代码替换原模块的代码。
 - **处理依赖**：递归地更新所有依赖该模块的其他模块，确保整个应用的依赖关系保持一致。
 - **执行回调**：如果模块中定义了 `module.hot.accept` 方法，在模块更新完成后，会执行该方法中的回调函数，开发者可以在回调函数中编写一些额外的处理逻辑，如重新初始化组件状态等。

5. 模块更新完成通知

- **执行回调**：模块更新完成后，HMR 运行时执行一些内置的回调函数，同时也会触发开发者在 `module.hot.accept` 中定义的回调函数，通知开发者模块已经成功更新。
- **界面更新**：浏览器会根据更新后的模块代码，动态更新页面上的相关内容，实现模块的热更新效果，且无需刷新整个页面。

通过以上基于 WebSocket 通信的方式，Webpack 实现了模块的实时更新，大大提高了开发效率。

10. webpack 整合

以下为你整合了面试中常见的 Webpack 问题及详细回答，涵盖 Webpack 基础概念、配置、优化等多个方面：

1. 什么是 Webpack，它的主要作用是什么？

Webpack 是一个现代 JavaScript 应用程序的静态模块打包工具。它的主要作用是将各种类型的模块（如 JavaScript、CSS、图片等）打包成一个或多个静态文件，从而优化前端资源的加载和管理。具体而言，Webpack 可以处理模块之间的依赖关系，将多个小文件合并成大文件以减少 HTTP 请求；对代码进行压缩、混淆等优化操作；还能处理不同类型的文件，如将 Sass 转换为 CSS、将 ES6+ 代码转换为兼容旧浏览器的代码等。

2. Webpack 的核心概念有哪些？

- **入口 (Entry)**：指示 Webpack 从哪个文件开始打包，是打包过程的起点。可以有单个入口或多个入口，多个入口适用于多页面应用。
- **输出 (Output)**：告诉 Webpack 打包后的文件存储在哪里，以及如何命名。包括存储目录（`path`）和文件名（`filename`）等配置。
- **loader**：用于处理不同类型的文件，将它们转换为 Webpack 能够理解的模块。例如，`babel-loader` 用于转换 ES6+ 代码，`css-loader` 和 `style-loader` 用于处理 CSS 文件。
- **插件 (Plugins)**：用于扩展 Webpack 的功能，在 Webpack 构建过程的特定阶段执行特定任务。比如，`HtmlWebpackPlugin` 用于生成 HTML 文件，`MiniCssExtractPlugin` 用于将 CSS 提取到单独的文件中。
- **模式 (Mode)**：有 `development`、`production` 和 `none` 三种模式。`development` 模式下会启用一些有助于开发的特性，如更详细的错误信

息； `production` 模式下会进行代码压缩和优化； `none` 则不使用任何默认优化选项。

3. 如何配置一个基本的 Webpack 项目？

以下是一个基本的 Webpack 配置示例：

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  mode: 'development',
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      },
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ],
  devServer: {
    static: {
      directory: path.join(__dirname, 'dist'),
    },
    hot: true,
  },
}
```

```
    open: true
  }
};
```

这个配置中，入口文件是 `src/index.js`，打包后的文件会存储在 `dist` 目录下，命名为 `bundle.js`。使用 `babel-loader` 处理 JavaScript 文件，`style-loader` 和 `css-loader` 处理 CSS 文件。`HtmlWebpackPlugin` 会根据 `src/index.html` 模板生成一个 HTML 文件，并自动引入打包后的 JavaScript 文件。`devServer` 配置了开发服务器，支持热更新和自动打开浏览器。

4. Webpack 的 loader 和插件有什么区别？

- **loader**：主要用于处理不同类型的文件，将它们转换为 Webpack 能够处理的模块。它是在模块加载时进行转换，针对的是单个模块。例如，`file-loader` 可以处理图片文件，将其复制到输出目录并返回文件的路径。
- **插件**：用于扩展 Webpack 的功能，在 Webpack 构建过程的特定阶段执行特定任务。它可以影响整个构建过程，处理多个模块或生成额外的文件。比如，`CleanWebpackPlugin` 可以在每次构建前清空输出目录。

5. 如何优化 Webpack 的构建速度和打包体积？

构建速度优化

- **使用缓存**：Webpack 5 引入了持久化缓存，将构建结果存储在磁盘上，下次构建时可以直接复用。配置如下：

```
module.exports = {
  cache: {
    type: 'filesystem',
    buildDependencies: {
      config: [__filename]
    }
  }
};
```

- **多线程构建**：使用 `thread-loader` 可以将耗时的 loader 放到独立的线程中处理，提高构建速度。例如：

```
{
  test: /\.js$/,
  use: [
```

```

    'thread-loader',
    {
      loader: 'babel-loader',
      options: {
        presets: ['@babel/preset-env']
      }
    }
  ]
}

```

- **排除不必要的文件：**在 `exclude` 配置中排除 `node_modules` 等不需要处理的目录。

打包体积优化

- **代码分割：**使用 `splitChunks` 配置进行代码分割，将公共代码提取到单独的文件中，减少重复代码。例如：

```

module.exports = {
  optimization: {
    splitChunks: {
      chunks: 'all'
    }
  }
};

```

- **压缩代码：**在 `production` 模式下，Webpack 会自动压缩代码。也可以使用 `TerserPlugin` 进一步优化 JavaScript 代码压缩。
- **Tree Shaking：**对于 ES6 模块，Webpack 可以通过 Tree Shaking 去除未使用的代码，减少打包体积。确保代码使用 ES6 模块语法，并在 `package.json` 中设置 `"sideEffects": false`。

6. Webpack 的缓存机制是如何工作的？

Webpack 有持久化缓存和内存缓存两种机制。

- **持久化缓存：**从 Webpack 5 开始引入，将构建结果存储在磁盘上。在首次构建时，Webpack 会记录每个模块的相关信息，如内容哈希值、依赖关系等。后续构建时，会重新计算模块信息并与缓存中的信息对比，如果模块信息未变，则直接使用缓存结果。当模块内容变化、Webpack 配置文件修改或 `buildDependencies` 中指定的文件发生变化时，缓存会失效。
- **内存缓存：**在同一次构建过程中，Webpack 会在内存中维护一个缓存，处理模块时

会先检查内存缓存，若命中则直接使用结果，避免重复处理，提高处理效率。

7. Webpack 的生命周期是怎样的？

Webpack 的打包过程包含多个阶段，每个阶段有相应的钩子（Hook），开发者可以通过这些钩子在特定阶段执行自定义逻辑。主要阶段和钩子如下：

- **初始化阶段**：包括 `entry-option`（入口配置处理完成后执行）、`after-plugins`（所有插件应用后执行）、`after-resolvers`（解析器设置完成后执行）。
- **编译阶段**：有 `compile`（开始编译前触发）、`make`（开始构建模块前触发）、`after-compile`（编译完成后触发）。
- **生成资源阶段**：`should-emit`（资源生成后、输出前触发，可决定是否输出资源）、`emit`（生成资源并即将输出到文件系统前触发，可修改资源）、`after-emit`（资源输出到文件系统后触发）。
- **完成阶段**：`done`（整个打包过程完成后触发）、`failed`（打包过程出错时触发）。

11.Tree Shaking 原理

Tree Shaking 是一种在打包工具（如 Webpack、Rollup 等）中用于消除未使用代码（dead code）的技术，它可以显著减小打包后的文件体积，提高应用的加载性能。下面将详细介绍如何进行 Tree Shaking 以及其工作原理。

如何进行 Tree Shaking

1. 使用 ES6 模块语法

Tree Shaking 依赖于 ES6 模块的静态结构特性，即模块之间的导入和导出关系在编译时是确定的。因此，要确保你的代码使用 ES6 的 `import` 和 `export` 语法，而不是 CommonJS 的 `require` 和 `module.exports`。

示例代码：

```
// utils.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// main.js
```

```
import { add } from './utils.js';
console.log(add(1, 2));
```

2. 配置打包工具

以 Webpack 为例，需要进行以下配置：

设置 mode 为 production：

在 Webpack 4 及以上版本中，将 `mode` 设置为 `production` 会自动开启一些优化，包括 Tree Shaking。

```
// webpack.config.js
module.exports = {
  mode: 'production',
  // 其他配置...
};
```

确保 package.json 中的 sideEffects 字段正确设置：

`sideEffects` 字段用于告诉打包工具哪些文件有副作用（即执行代码会产生除导出变量之外的其他影响，如修改全局变量、执行一些初始化操作等）。如果你的代码没有副作用，可以将 `sideEffects` 设置为 `false`，这样打包工具可以更激进地进行 Tree Shaking。

```
{
  "name": "my-app",
  "sideEffects": false
}
```

如果有部分文件有副作用，可以列出这些文件的路径：

```
{
  "name": "my-app",
  "sideEffects": [
    "./src/some-file-with-side-effects.js",
    "*.css"
  ]
}
```

3. 使用支持 Tree Shaking 的第三方库

选择使用以 ES6 模块形式发布的第三方库，这样才能保证打包工具可以对其进行 Tree Shaking。例如，Lodash 提供了支持 Tree Shaking 的版本 `lodash-es`。

```
// 使用 lodash-es 进行 Tree Shaking
import { cloneDeep } from 'lodash-es';
const obj = { a: 1 };
const clonedObj = cloneDeep(obj);
```

Tree Shaking 的原理

1. 静态分析

Tree Shaking 的核心是对 ES6 模块进行静态分析，即在不执行代码的情况下，分析模块之间的导入和导出关系。打包工具会解析模块文件，识别出哪些变量被导出，哪些变量被导入和使用。

例如，对于以下代码：

```
// utils.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// main.js
import { add } from './utils.js';
console.log(add(1, 2));
```

打包工具在静态分析时会发现，`main.js` 只导入并使用了 `utils.js` 中的 `add` 函数，而 `subtract` 函数没有被使用。

2. 标记未使用的代码

在静态分析完成后，打包工具会标记出那些未被使用的导出变量和代码块。在上述例子中，`subtract` 函数会被标记为未使用。

3. 消除未使用的代码

最后，打包工具在生成最终的打包文件时，会移除那些被标记为未使用的代码，从而实现 Tree Shaking 的效果。经过 Tree Shaking 处理后，最终的打包文件中不会包含 `subtract` 函数的代码。

需要注意的是，Tree Shaking 只能处理 ES6 模块中静态可分析的代码，对于动态导入（如 `import()`）和一些复杂的运行时依赖，Tree Shaking 的效果可能会受到限制。此外，代码中的副作用也会影响 Tree Shaking 的结果，因此需要正确设置 `sideEffects` 字段。

12.前端工程化、模块化

前端工程化

定义

前端工程化是指运用工程的理念、方法和技术来管理和组织前端开发流程，将前端开发从传统的手工式、零散式开发转变为规范化、标准化、自动化的开发模式，以提高开发效率、保证代码质量、降低维护成本。

主要内容

1. 项目架构设计

- **分层架构设计**
 - **视图层**：负责页面的展示，如使用 Vue、React 等前端框架构建组件化的视图。以 Vue 为例，创建一个个独立的 Vue 组件，每个组件负责特定的 UI 部分，提高代码的可维护性和复用性。
 - **业务逻辑层**：处理业务逻辑，将视图层和数据层解耦。例如在一个电商项目中，商品列表的筛选、排序等逻辑可以放在业务逻辑层处理。
 - **数据访问层**：负责与后端 API 进行交互，获取和处理数据。可以使用 Axios 等工具发送 HTTP 请求，封装请求逻辑，方便统一管理。
- **目录结构规范**
 - 制定清晰的目录结构，如 `src` 目录存放源代码，`public` 目录存放静态资源，`build` 或 `dist` 目录存放打包后的文件。
 - 在 `src` 目录下，可以进一步细分，如 `components` 存放组件，`views` 存放页面视图，`store` 存放状态管理代码（如果使用 Vuex 或 Redux），`utils` 存放工具函数等。

2. 模块化开发

- **代码模块化**：将代码拆分成多个独立的模块，每个模块具有单一的功能和职责，提高代码的可复用性和可维护性。例如使用 ES6 的 `import` 和 `export` 语法实现 JavaScript 模块的导入和导出。
- **样式模块化**：通过 CSS Modules、Sass、Less 等技术实现样式的模块化，避免样式冲突。例如在 CSS Modules 中，每个 CSS 文件都是一个独立的模块，类名会被自动转换为唯一的哈希值。

3. 自动化构建

- **打包工具**：使用 Webpack、Rollup、Parcel 等打包工具将多个模块打包成一个或多个文件，减少浏览器的请求次数，提高页面加载速度。例如 Webpack 可以处理各种类型的模块，如 JavaScript、CSS、图片等，并对它们进行优化和处理。
- **自动化任务**：借助 Gulp、Grunt 等工具实现自动化任务，如代码压缩、文件合并、图片优化、代码检查等。例如使用 Gulp 可以自动压缩 CSS 和 JavaScript 文件，减少文件体积。

4. 版本控制与团队协作

- **版本控制系统**：使用 Git 等版本控制系统对代码进行管理，方便团队成员协作开发、跟踪代码变更和回滚历史版本。例如团队成员可以在不同的分支上进行开发，完成后再合并到主分支。
- **代码审查**：建立代码审查机制，团队成员对提交的代码进行审查，确保代码符合规范和质量要求。例如在 GitHub 上可以通过 Pull Request 进行代码审查。

5. 测试与部署

- **单元测试**：编写单元测试用例，对代码的各个模块进行独立测试，确保代码的正确性。常用的测试框架有 Jest、Mocha 等。例如使用 Jest 可以对 JavaScript 函数进行单元测试。
- **集成测试**：对多个模块进行集成测试，验证它们之间的协作是否正常。
- **自动化部署**：使用 Jenkins、GitLab CI/CD 等工具实现自动化部署，将代码部署到生产环境。例如当代码推送到 Git 仓库时，自动触发部署流程，将代码部署到服务器上。

前端模块化

定义

前端模块化是指将一个复杂的前端应用拆分成多个独立的、可复用的模块，每个模块具有明确的功能和职责，模块之间通过特定的方式进行交互和通信。

发展历程

1. 早期的模块化尝试

- **文件划分**：早期开发者通过将不同功能的代码放在不同的文件中，实现简单的模块化。但这种方式存在全局变量污染、依赖管理困难等问题。

- **命名空间模式**：通过创建全局对象，将相关的函数和变量挂载到该对象上，避免全局变量冲突。例如：

```
var myModule = {
  func1: function() {
    // 函数实现
  },
  func2: function() {
    // 函数实现
  }
};
```

2. CommonJS 规范

- **服务器端模块化**：CommonJS 是服务器端模块规范，主要用于 Node.js 环境。它使用 `require` 函数引入模块，使用 `module.exports` 或 `exports` 导出模块。例如：

```
// math.js
exports.add = function(a, b) {
  return a + b;
};

// main.js
const math = require('./math.js');
console.log(math.add(1, 2));
```

3. AMD 规范 (Asynchronous Module Definition)

- **异步模块加载**：AMD 是浏览器端的异步模块加载规范，主要通过 `define` 函数定义模块，使用 `require` 函数加载模块。例如使用 RequireJS 实现 AMD 规范：

```
// math.js
define(function() {
  return {
    add: function(a, b) {
      return a + b;
    }
  };
});

// main.js
require(['math'], function(math) {
```

```
    console.log(math.add(1, 2));  
  });
```

4. CMD 规范 (Common Module Definition)

- **延迟执行**：CMD 也是一种浏览器端的模块规范，与 AMD 类似，但更强调延迟执行。Sea.js 是实现 CMD 规范的典型库。

```
// math.js  
define(function(require, exports, module) {  
    exports.add = function(a, b) {  
        return a + b;  
    };  
});  
  
// main.js  
seajs.use(['math'], function(math) {  
    console.log(math.add(1, 2));  
});
```

5. ES6 模块化

- **官方标准**：ES6 引入了原生的模块化语法，使用 `import` 和 `export` 关键字实现模块的导入和导出。例如：

```
// math.js  
export const add = (a, b) => a + b;  
  
// main.js  
import { add } from './math.js';  
console.log(add(1, 2));
```

模块化的优势

- **可维护性**：模块具有单一的功能和职责，当某个模块出现问题时，只需要修改该模块，不会影响其他模块。
- **可复用性**：模块可以在不同的项目或同一项目的不同地方重复使用，提高开发效率。
- **依赖管理**：模块化系统可以清晰地管理模块之间的依赖关系，避免依赖冲突。
- **性能优化**：通过按需加载模块，减少首屏加载时间，提高页面性能。

13.前端模块化和 组件化 的区别?

前端模块化和组件化都是前端开发中重要的概念，它们在提高代码的可维护性、可复用性和开发效率等方面发挥着关键作用，但二者存在明显区别，下面从概念、粒度、关注点、使用场景等方面进行详细阐述。

概念差异

- 前端模块化

- 模块化是指将一个复杂的前端应用拆分成多个独立的、可复用的模块，每个模块具有明确的功能和职责，模块之间通过特定的方式进行交互和通信。它更侧重于代码逻辑的组织和管理，将不同功能的代码封装在不同的模块中，使得代码结构清晰，便于开发和维护。
- 例如，在 JavaScript 中，使用 ES6 的 `import` 和 `export` 语法可以实现模块的导入和导出，一个模块可以是一个函数库、一个数据处理模块等。

- 前端组件化

- 组件化是将页面中可复用的 UI 部分封装成独立的组件，每个组件包含自己的 HTML 结构、CSS 样式和 JavaScript 逻辑，组件之间可以嵌套和组合，形成完整的页面。它强调的是 UI 界面的拆分和复用，通过组件化开发可以提高 UI 开发的效率和质量。
- 比如，在 Vue.js 或 React 等框架中，一个按钮、一个表单、一个导航栏都可以封装成一个组件。

粒度差异

- 前端模块化

- 模块化的粒度相对较细，它可以是一个函数、一个类或者一组相关的函数和数据的集合。一个模块可以只负责完成一个特定的小功能，例如一个处理日期格式化的模块，或者一个进行数学计算的模块。

- 前端组件化

- 组件化的粒度相对较粗，通常是一个具有完整 UI 功能的单元。一个组件可能包含多个模块，例如一个商品列表组件，它可能会使用到数据获取模块、数据处理模块、模板渲染模块等。

关注点差异

- 前端模块化

- 主要关注代码的逻辑结构和复用性，通过将代码拆分成多个模块，使得每个模

块的功能单一且明确，便于代码的维护和测试。模块之间的依赖关系清晰，修改一个模块不会对其他模块产生过多的影响。

- 例如，在一个电商项目中，商品数据的获取和处理可以分别封装成不同的模块，这样在需要修改数据获取方式时，只需要修改数据获取模块，而不会影响到数据处理模块。

- **前端组件化**

- 更关注 UI 的设计和交互，将页面拆分成多个组件，每个组件有自己的样式和交互逻辑。组件可以独立开发、测试和部署，提高了 UI 开发的效率和可维护性。
- 例如，在设计一个社交网站时，将评论区封装成一个组件，这个组件包含评论的展示、发布、删除等功能，并且有自己的样式，这样在不同的页面中都可以复用这个评论组件。

使用场景差异

- **前端模块化**

- 适用于处理复杂的业务逻辑，当项目中有大量的代码需要组织和管理时，模块化可以使代码结构更加清晰。例如，在一个大型的 JavaScript 应用中，将不同的业务逻辑封装成不同的模块，方便团队成员分工协作开发。
- 也适用于代码复用，当多个地方需要使用相同的功能时，可以将这些功能封装成模块，在需要的地方进行导入和使用。

- **前端组件化**

- 适用于构建复杂的 UI 界面，当页面中有多个重复的 UI 元素时，组件化可以提高开发效率。例如，在开发一个管理系统时，将表格、表单等常用的 UI 元素封装成组件，在不同的页面中复用这些组件。
- 也适用于快速迭代开发，当需要对 UI 进行修改时，只需要修改相应的组件，而不会影响到其他部分的代码。

14. git

以下为你整合了前端面试中常见的 Git 问题及详细回答：

1. 什么是 Git，它的主要作用是什么？

Git 是一个分布式版本控制系统，用于跟踪文件的变化、管理项目的版本历史，支持多人协作开发。其主要作用包括：

- **版本控制**：记录项目文件的每一次修改，方便开发者随时查看历史版本，回溯到之前的某个状态。
- **团队协作**：多个开发者可以同时在一个项目上进行开发，Git 能有效处理代码的合并与冲突，确保团队协作的高效性。
- **分支管理**：允许开发者创建不同的分支进行独立开发，如开发新功能、修复 bug 等，开发完成后再将分支合并到主分支。

2. 简述 Git 的基本工作流程

- **克隆仓库**：使用 `git clone <repository-url>` 命令将远程仓库复制到本地。
- **创建分支**：使用 `git branch <branch-name>` 创建新分支，使用 `git checkout <branch-name>` 切换到该分支，也可以使用 `git checkout -b <branch-name>` 一步完成创建并切换分支。
- **开发与提交**：在分支上进行代码开发，使用 `git add <file>` 将修改的文件添加到暂存区，使用 `git commit -m "commit message"` 将暂存区的内容提交到本地仓库。
- **同步远程仓库**：使用 `git pull` 从远程仓库拉取最新代码并合并到本地分支，使用 `git push` 将本地分支的代码推送到远程仓库。
- **合并分支**：开发完成后，切换到主分支（如 `master` 或 `main`），使用 `git merge <branch-name>` 将开发分支合并到主分支。

3. Git 中 HEAD、工作区、暂存区和本地仓库分别是什么，它们之间有什么关系？

- **HEAD**：指向当前所在的分支的最新提交，是当前分支指针的引用。
- **工作区**：即你在本地计算机上实际进行文件编辑的目录，是你能直接看到和修改的文件区域。
- **暂存区**：也叫索引（index），是一个中间区域，用于临时保存你所做的修改，你可以选择将部分或全部修改添加到暂存区。
- **本地仓库**：存储了项目的所有版本历史信息，包括提交记录、分支信息等。

它们之间的关系是：你在工作区对文件进行修改后，使用 `git add` 命令将修改添加到暂存区，使用 `git commit` 命令将暂存区的内容提交到本地仓库。`HEAD` 始终指向当前分支的最新提交。

4. 如何解决 Git 合并冲突？

当合并分支时，如果两个分支对同一文件的同一部分进行了不同的修改，就会产生合并

冲突。解决步骤如下：

- **查看冲突文件：**使用 `git status` 命令查看哪些文件发生了冲突。
- **编辑冲突文件：**打开冲突文件，会看到类似下面的标记：

```
<<<<<<< HEAD
// 当前分支的代码
=====
// 要合并的分支的代码
>>>>>>> branch-name
```

手动选择保留哪部分代码，或者将两部分代码进行整合，然后删除冲突标记。

- **标记冲突已解决：**使用 `git add <file>` 将解决冲突后的文件添加到暂存区。
- **完成合并：**使用 `git commit` 完成合并操作。

5. 简述 Git 分支管理策略

常见的分支管理策略有 GitFlow 和 GitHub Flow。

- **GitFlow：**包含主分支（`master` 或 `main`）、开发分支（`develop`）、功能分支（`feature/*`）、发布分支（`release/*`）和热修复分支（`hotfix/*`）。主分支用于生产环境代码，开发分支是所有开发的基础，功能分支从开发分支创建，用于开发新功能，开发完成后合并回开发分支；发布分支从开发分支创建，用于准备新版本发布，测试完成后合并到主分支和开发分支；热修复分支从主分支创建，用于紧急修复生产环境的 bug，修复完成后合并到主分支和开发分支。
- **GitHub Flow：**相对简单，只有主分支（`master` 或 `main`）和功能分支。所有的新功能都在功能分支上开发，开发完成后通过 Pull Request 合并到主分支。主分支的代码始终是可部署的。

6. 如何撤销 Git 中的某次提交？

- **使用 `git revert`：**如果想撤销某次提交，但保留该提交之后的所有提交记录，可以使用 `git revert <commit-hash>`。这个命令会创建一个新的提交，该提交的内容是撤销指定提交所做的修改。
- **使用 `git reset`：**如果想彻底删除某次提交及其之后的所有提交记录，可以使用 `git reset <commit-hash>`。`git reset` 有三种模式：
 - `--soft`：只移动 HEAD 指针，不改变暂存区和工作区的内容。
 - `--mixed`（默认）：移动 HEAD 指针，并重置暂存区，但不改变工作区的内容。

- `--hard`：移动 HEAD 指针，重置暂存区和工作区的内容，使其与指定提交一致。

7. 如何查看 Git 的提交历史？

- 使用 `git log`：显示提交历史记录，默认按时间倒序排列，每条记录包含提交哈希、作者、提交日期和提交信息。
- 常用参数：
 - `--oneline`：将每条提交记录显示为一行，只显示提交哈希的前几位和提交信息。
 - `--graph`：以图形化的方式显示分支和合并历史。
 - `--author=<author-name>`：只显示指定作者的提交记录。
 - `--since=<date>` 和 `--until=<date>`：显示指定日期范围内的提交记录。

8. 什么是 Git 的标签（Tag），如何使用？

Git 标签是指向某个提交的引用，用于标记项目的重要版本，如发布版本。标签分为轻量标签和附注标签。

- 创建轻量标签：使用 `git tag <tag-name>` 在当前提交上创建轻量标签。
- 创建附注标签：使用 `git tag -a <tag-name> -m "tag message"` 创建附注标签，附注标签包含更多的元信息，如标签信息、作者等。
- 查看标签：使用 `git tag` 查看所有标签，使用 `git show <tag-name>` 查看某个标签的详细信息。
- 推送标签到远程仓库：使用 `git push origin <tag-name>` 推送单个标签，使用 `git push origin --tags` 推送所有标签。

9. git rebase

以下为你补充关于 `git rebase` 的相关内容，并将其整合到前面的 Git 面试问答中：

1. 什么是 `git rebase`，它和 `git merge` 有什么区别？

`git rebase` 是 Git 中用于整合分支的一种方式，它将一个分支的修改应用到另一个分支上，通过将提交历史以线性的方式重新排列，使得提交历史看起来更加简洁、清晰。

`git rebase` 和 `git merge` 的主要区别如下：

- 提交历史展示

- `git merge`：会创建一个新的合并提交，将两个分支的历史合并在一起，提交历史呈现出分叉的状态，能够清晰地看到分支的合并点，但可能会使提交历史变得复杂。
- `git rebase`：将一个分支的提交依次应用到另一个分支上，使提交历史呈现线性，更加简洁直观。
- 操作过程
 - `git merge`：操作相对简单，直接将两个分支合并，处理冲突时在合并提交中解决。
 - `git rebase`：操作过程中会逐个应用提交，遇到冲突需要在每个提交上解决，完成冲突解决后继续 `rebase` 操作。

2. 如何使用 `git rebase`？

以下是 `git rebase` 的基本使用步骤：

- 切换到需要被合并的分支：假设你在 `feature` 分支上进行开发，想要将 `feature` 分支的修改应用到 `master` 分支上，首先切换到 `feature` 分支。

```
git checkout feature
```

- 执行 `rebase` 操作：将 `feature` 分支的提交应用到 `master` 分支上。

```
git rebase master
```

- 处理冲突（如果有）：在 `rebase` 过程中，如果遇到冲突，Git 会暂停 `rebase` 操作并提示你解决冲突。解决冲突后，使用 `git add` 将解决冲突的文件添加到暂存区，然后使用 `git rebase --continue` 继续 `rebase` 操作。

```
# 解决冲突后
git add <file>
git rebase --continue
```

- 完成 `rebase`：当所有提交都成功应用后，`rebase` 操作完成。此时，`feature` 分支的提交已经线性排列在 `master` 分支之后。

3. `git rebase` 有哪些应用场景？

- 保持提交历史线性：在团队协作中，为了使提交历史更加清晰、易于理解和审查，使用 `git rebase` 可以将开发分支的提交以线性方式整合到主分支上。
- 清理提交历史：在开发过程中，可能会产生一些临时提交或不必要的提交。使用

`git rebase -i`（交互式 `rebase`）可以合并、修改或删除这些提交，使提交历史更加简洁。例如，将多个小的提交合并成一个有意义的提交：

```
git rebase -i HEAD~3 # 对最近的 3 个提交进行交互式 rebase
```

在弹出的编辑器中，可以选择 `pick`（保留提交）、`squash`（将提交合并到前一个提交）等操作。

4. `git rebase` 有什么风险？

- **数据丢失风险**：如果在 `rebase` 过程中操作不当，例如强制覆盖已经推送到远程仓库的提交，可能会导致数据丢失，其他开发者可能会遇到难以解决的冲突。
- **破坏公共分支历史**：不建议在公共分支（如 `master`）上进行 `rebase` 操作，因为这会改变公共分支的提交历史，可能会影响其他开发者的工作。如果需要在公共分支上应用 `rebase`，应该先与团队成员沟通，确保大家都能正确处理。

10. `git cherry-pick`

以下是关于 `git cherry - pick` 的相关内容，补充到前面的 Git 面试问答集合里：

1. 什么是 `git cherry - pick`

`git cherry - pick` 是 Git 中的一个命令，它允许你从一个或多个提交中挑选出特定的提交，并将这些提交所做的更改应用到当前分支上。这就好比从众多的樱桃（提交）里挑选出你想要的那几颗，然后把它们“种”到当前分支。它提供了一种细粒度的提交合并方式，与 `git merge` 和 `git rebase` 不同，后两者通常是合并整个分支的更改，而 `git cherry - pick` 可以只选择部分提交进行合并。

2. 如何使用 `git cherry - pick`

挑选单个提交

假设你有两个分支 `master` 和 `feature`，`feature` 分支上有一个提交（提交哈希为 `abc123`），你想把这个提交应用到 `master` 分支上，操作步骤如下：

```
# 切换到目标分支
git checkout master
# 执行 cherry - pick 操作
git cherry - pick abc123
```

如果在应用提交的过程中没有冲突，Git 会自动创建一个新的提交，这个新提交的内容和 `abc123` 提交的内容一样，但提交哈希不同。

挑选多个提交

如果要挑选多个连续的提交，可以使用以下语法：

```
git cherry - pick start_commit..end_commit
```

这里的 `start_commit` 是起始提交的哈希，`end_commit` 是结束提交的哈希，但需要注意，这种方式不包含 `start_commit` 本身，只包含从 `start_commit` 之后到 `end_commit` 的所有提交。如果要包含 `start_commit`，可以使用：

```
git cherry - pick start_commit^..end_commit
```

挑选不连续的提交

如果要挑选不连续的多个提交，只需依次列出提交的哈希值即可：

```
git cherry - pick commit1 commit2 commit3
```

3. 处理 `git cherry - pick` 过程中的冲突

当执行 `git cherry - pick` 时，如果提交所做的更改与当前分支的内容发生冲突，Git 会暂停 `cherry - pick` 操作并提示你解决冲突。解决冲突的步骤如下：

- **查看冲突文件：**使用 `git status` 命令查看哪些文件发生了冲突。
- **编辑冲突文件：**打开冲突文件，手动解决冲突内容，删除冲突标记。
- **标记冲突已解决：**使用 `git add` 命令将解决冲突后的文件添加到暂存区。
- **继续 `cherry - pick`：**使用 `git cherry - pick --continue` 命令继续 `cherry - pick` 操作。如果想要放弃本次 `cherry - pick`，可以使用 `git cherry - pick --abort` 命令。

4. `git cherry - pick` 的应用场景

- **修复特定版本的 bug：**假设在 `release` 版本分支上发现了一个 bug，而这个 bug 的修复已经在 `develop` 分支的某个提交中完成。这时可以使用 `git cherry - pick` 将该提交应用到 `release` 分支上，而不需要合并整个 `develop` 分支。

- **同步部分功能**：当你只需要将某个分支上的部分功能同步到另一个分支时，`git cherry - pick` 可以帮助你精确地选择需要的提交进行合并。

5. `git cherry - pick` 的潜在问题

- **重复提交问题**：如果多次对同一个提交执行 `cherry - pick`，可能会在目标分支上产生重复的提交内容。所以在使用前要确保不会出现这种情况。
- **提交历史混乱**：过度使用 `git cherry - pick` 可能会导致提交历史变得复杂和难以理解，因为提交的顺序和逻辑可能会被打乱。因此，在使用时需要谨慎考虑，确保提交历史的可读性和可维护性。

15. flex 布局 and 属性

1. 什么是 Flex 布局

Flex 是 Flexible Box 的缩写，意为“弹性布局”，用于为盒状模型提供最大的灵活性。任何一个容器都可以指定为 Flex 布局。当为父元素设置 `display: flex` 或 `display: inline-flex` 后，子元素的 `float`、`clear` 和 `vertical-align` 属性将失效。

2. 基本概念

采用 Flex 布局的元素，称为 Flex 容器（flex container），简称“容器”。它的所有子元素自动成为容器成员，称为 Flex 项目（flex item），简称“项目”。

容器默认存在两根轴：水平的主轴（main axis）和垂直的交叉轴（cross axis）。主轴的开始位置（与边框的交叉点）叫做 `main start`，结束位置叫做 `main end`；交叉轴的开始位置叫做 `cross start`，结束位置叫做 `cross end`。项目默认沿主轴排列，单个项目占据的主轴空间叫做 `main size`，占据的交叉轴空间叫做 `cross size`。

3. 容器的属性

(1) `display`

定义一个元素为 Flex 容器，有两种取值：

- `flex`：容器为块级元素。
- `inline-flex`：容器为行内元素。

```
.container {
```

```
display: flex; /* 或 inline-flex */
}
```

(2) flex-direction

决定主轴的方向，即项目的排列方向。

- row（默认值）：主轴为水平方向，起点在左端。
- row-reverse：主轴为水平方向，起点在右端。
- column：主轴为垂直方向，起点在上沿。
- column-reverse：主轴为垂直方向，起点在下沿。

```
.container {
  flex-direction: row;
}
```

(3) flex-wrap

定义如果一条轴线排不下，如何换行。

- nowrap（默认值）：不换行。
- wrap：换行，第一行在上方。
- wrap-reverse：换行，第一行在下方。

```
.container {
  flex-wrap: wrap;
}
```

(4) flex-flow

是 flex-direction 和 flex-wrap 的简写形式，默认值为 row nowrap。

```
.container {
  flex-flow: row wrap;
}
```

(5) justify-content

定义了项目在主轴上的对齐方式。

- `flex-start`（默认值）：左对齐。
- `flex-end`：右对齐。
- `center`：居中。
- `space-between`：两端对齐，项目之间的间隔都相等。
- `space-around`：每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。

```
.container {  
  justify-content: center;  
}
```

(6) `align-items`

定义项目在交叉轴上如何对齐。

- `stretch`（默认值）：如果项目未设置高度或设为 `auto`，将占满整个容器的高度。
- `flex-start`：交叉轴的起点对齐。
- `flex-end`：交叉轴的终点对齐。
- `center`：交叉轴的中点对齐。
- `baseline`：项目的第一行文字的基线对齐。

```
.container {  
  align-items: center;  
}
```

(7) `align-content`

定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用。

- `stretch`（默认值）：轴线占满整个交叉轴。
- `flex-start`：与交叉轴的起点对齐。
- `flex-end`：与交叉轴的终点对齐。
- `center`：与交叉轴的中点对齐。
- `space-between`：与交叉轴两端对齐，轴线之间的间隔平均分布。
- `space-around`：每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍。

```
.container {  
    align-content: space-between;  
}
```

4. 项目的属性

(1) `order`

定义项目的排列顺序。数值越小，排列越靠前，默认为 0。

```
.item {  
    order: -1;  
}
```

(2) `flex-grow`

定义项目的放大比例，默认为 0，即如果存在剩余空间，也不放大。

```
.item {  
    flex-grow: 1; /* 如果所有项目的 flex - grow 属性都为 1，则它们将等分剩余空间 */  
}
```

(3) `flex-shrink`

定义了项目的缩小比例，默认为 1，即如果空间不足，该项目将缩小。

```
.item {  
    flex-shrink: 0; /* 如果一个项目的 flex - shrink 属性为 0，其他项目都为 1，则空间不足时，前者不缩小 */  
}
```

(4) `flex-basis`

定义了再分配多余空间之前，项目占据的主轴空间（main size）。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为 `auto`，即项目的本来大小。

```
.item {  
    flex-basis: 200px;  
}
```


(5) flex

是 `flex-grow`、`flex-shrink` 和 `flex-basis` 的简写，默认值为 `0 1 auto`。后两个属性可选。

```
.item {  
    flex: 1; /* 相当于 flex: 1 1 0 */  
}
```

(6) align-self

允许单个项目有与其他项目不一样的对齐方式，可覆盖 `align-items` 属性。默认值为 `auto`，表示继承父元素的 `align-items` 属性，如果没有父元素，则等同于 `stretch`。

```
.item {  
    align-self: flex-end;  
}
```

5. Flex 布局示例

HTML 代码

```
<!DOCTYPE html>  
<html lang="en">  
  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <link rel="stylesheet" href="styles.css">  
    <title>Flexbox Demo</title>  
</head>  
  
<body>  
    <div class="container">  
        <div class="item">1</div>  
        <div class="item">2</div>  
        <div class="item">3</div>  
        <div class="item">4</div>  
        <div class="item">5</div>  
    </div>  
</body>
```

```
</div>
</body>

</html>
```

CSS 代码 (styles.css)

```
.container {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-around;
  align-items: center;
  background-color: #f4f4f4;
  padding: 20px;
}

.item {
  background-color: #3498db;
  color: white;
  font-size: 24px;
  width: 100px;
  height: 100px;
  margin: 10px;
  display: flex;
  justify-content: center;
  align-items: center;
  flex: 1 1 20%; /* 每个项目初始大小为 20%，可伸缩 */
}
```

示例解释

在这个示例中，我们创建了一个 Flex 容器 `.container`，设置了 `flex-wrap` 为 `wrap` 使其可以换行，`justify-content` 为 `space-around` 让项目在主轴上均匀分布，`align-items` 为 `center` 让项目在交叉轴上居中对齐。每个项目 `.item` 的 `flex` 属性设置为 `1 1 20%`，表示它们在有剩余空间时会放大，空间不足时会缩小，初始大小为容器宽度的 20%。

16. Http返回接口状态码

在前端开发中，接口返回码主要分为 HTTP 状态码和业务状态码，以下为你详细介绍：

HTTP 状态码

HTTP 状态码由三位数字组成，第一个数字定义了响应的类别，后两个数字无分类意义。常见的类别有以下五种：

- **1xx (信息性状态码)**：表示临时响应，用来通知客户端请求的第一部分已经收到，客户端应继续发送其余部分。不过这类状态码在实际应用中较少使用。
 - **100 Continue**：客户端应当继续发送请求。这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。
- **2xx (成功状态码)**：表示请求已成功被服务器接收、理解并接受。
 - **200 OK**：最常见的成功状态码，表明请求已经成功处理，返回的信息跟请求所希望的格式一致。
 - **201 Created**：请求已经被实现，而且有一个新的资源已经依据请求的需要而建立，且其 URI 已经随 Location 头信息返回。
 - **204 No Content**：服务器成功处理了请求，但不需要返回任何实体内容。常用于删除操作成功时。
- **3xx (重定向状态码)**：表示要完成请求，需要进一步操作，通常用于重定向。
 - **301 Moved Permanently**：被请求的资源已永久移动到新 URI，客户端应使用新的 URI 重新发起请求。
 - **302 Found**：临时性重定向，请求的资源临时从不同的 URI 响应请求，但客户端应继续使用原有 URI 进行以后的请求。
 - **304 Not Modified**：表示资源未被修改，可以使用缓存的版本。客户端可以继续使用缓存的响应，而无需再次向服务器请求资源。
- **4xx (客户端错误状态码)**：表示客户端可能存在错误，妨碍了服务器的处理。
 - **400 Bad Request**：客户端发送的请求有错误，不能被服务器所识别。例如，请求参数格式错误。
 - **401 Unauthorized**：请求未经授权，需要进行身份验证。通常在访问需要登录的资源时，如果用户未登录，会返回此状态码。
 - **403 Forbidden**：服务器理解请求客户端的请求，但是拒绝执行此请求。可能是因为权限不足，即使进行了身份验证也无法访问该资源。
 - **404 Not Found**：请求的资源不存在，可能是 URI 错误或者资源已被删除。
 - **405 Method Not Allowed**：客户端请求的方法被禁止，例如使用 `POST` 方法请求一个只支持 `GET` 方法的接口。
- **5xx (服务器错误状态码)**：表示服务器在处理请求的过程中发生了错误。
 - **500 Internal Server Error**：最常见的服务器错误状态码，表明服务器在执行请求时发生了内部错误，可能是代码逻辑错误、数据库连接问题等。

- **502 Bad Gateway**: 作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。
- **503 Service Unavailable**: 服务器目前无法使用（由于超载或停机维护），通常只是暂时状态。
- **504 Gateway Timeout**: 网关或者代理服务器未能及时从上游服务器收到响应。

业务状态码

业务状态码是由开发者自定义的，用于表示具体业务逻辑的处理结果。不同的项目可能有不同的业务状态码定义，常见的业务状态码设计规则如下：

- **通用成功码**：一般用 `0` 或 `200` 表示业务处理成功。
 - 示例：

```
{
  "code": 0,
  "message": "操作成功",
  "data": {
    "name": "John",
    "age": 30
  }
}
```

- **通用失败码**：可以用 `-1` 或 `500` 表示业务处理失败。
 - 示例：

```
{
  "code": -1,
  "message": "操作失败，请稍后重试",
  "data": null
}
```

- **具体业务错误码**：根据不同的业务场景定义具体的错误码，例如：
 - `1001`：用户不存在。
 - `1002`：密码错误。
 - `2001`：商品库存不足。

示例：

```
{
  "code": 1002,
  "message": "密码错误",
  "data": null
}
```

vue中使用axios拦截器

在 Vue 项目中使用 Axios 拦截器可以对请求和响应进行统一处理，例如添加请求头、处理错误状态码等。以下是详细的步骤和示例代码。

1. 安装 Axios

首先，确保你已经在 Vue 项目中安装了 Axios。如果还没有安装，可以使用以下命令进行安装：

```
npm install axios
```

2. 创建 Axios 实例并设置拦截器

在项目中创建一个专门的文件（例如 `axios.js`）来配置 Axios 实例和拦截器。

```
// src/axios.js
import axios from 'axios';

// 创建 Axios 实例
const service = axios.create({
  baseURL: 'https://api.example.com', // 设置请求的基础 URL
  timeout: 5000 // 设置请求超时时间
});

// 请求拦截器
service.interceptors.request.use(
  config => {
    // 在发送请求之前做点什么
    // 例如添加请求头
    const token = localStorage.getItem('token');
    if (token) {
      config.headers['Authorization'] = `Bearer ${token}`;
    }
    return config;
  }
);
```

```

    },
    error => {
        // 处理请求错误
        console.error('请求出错:', error);
        return Promise.reject(error);
    }
});

// 响应拦截器
service.interceptors.response.use(
    response => {
        // 对响应数据做点什么
        return response.data;
    },
    error => {
        // 处理响应错误
        if (error.response) {
            // 请求已发送，服务器返回了非 2xx 状态码
            switch (error.response.status) {
                case 401:
                    // 跳转到登录页面
                    window.location.href = '/login';
                    break;
                case 403:
                    console.error('没有权限访问该资源');
                    break;
                case 404:
                    console.error('请求的资源不存在');
                    break;
                case 500:
                    console.error('服务器内部错误，请稍后重试');
                    break;
                default:
                    console.error(`未知错误，状态码：
${error.response.status}`);
            }
        } else if (error.request) {
            // 请求已发送，但没有收到响应
            console.error('没有收到服务器的响应，请检查网络连接');
        } else {
            // 在设置请求时发生了错误
            console.error('请求设置出错:', error.message);
        }
        return Promise.reject(error);
    }
});

```

```
export default service;
```

3. 在 Vue 组件中使用 Axios 实例

在 Vue 组件中引入并使用配置好的 Axios 实例。

```
<template>
  <div>
    <button @click="fetchData">获取数据</button>
  </div>
</template>

<script>
import service from '@/axios'; // 引入 Axios 实例

export default {
  methods: {
    async fetchData() {
      try {
        const response = await service.get('/data');
        console.log('请求成功:', response);
      } catch (error) {
        // 错误已在拦截器中处理
      }
    }
  }
};
</script>
```

4. 全局挂载 Axios 实例（可选）

如果你希望在所有 Vue 组件中都能方便地使用 Axios 实例，可以将其全局挂载到 Vue 实例上。

```
// main.js
import Vue from 'vue';
import App from './App.vue';
import service from './axios';

// 全局挂载 Axios 实例
Vue.prototype.$http = service;

new Vue({
```

```
render: h => h(App)
}).$mount('#app');
```

在 Vue 组件中使用全局挂载的 Axios 实例：

```
<template>
  <div>
    <button @click="fetchData">获取数据</button>
  </div>
</template>

<script>
export default {
  methods: {
    async fetchData() {
      try {
        const response = await this.$http.get('/data');
        console.log('请求成功:', response);
      } catch (error) {
        // 错误已在拦截器中处理
      }
    }
  }
};
</script>
```

通过以上步骤，你可以在 Vue 项目中方便地使用 Axios 拦截器对请求和响应进行统一处理。

17. Http 和 Https的区别？

HTTP（HyperText Transfer Protocol）和 HTTPS（HyperText Transfer Protocol Secure）是互联网中用于传输数据的两种协议，下面从多个方面详细介绍它们并进行对比。

基本概念

- **HTTP**：是一种用于传输超文本的协议，它是互联网上应用最为广泛的一种协议。HTTP 协议以明文方式在客户端和服务端之间传输数据，不提供数据加密功能。
- **HTTPS**：是在 HTTP 协议的基础上加入了 SSL/TLS 协议，通过加密和身份验证机制来保证数据传输的安全性。SSL（Secure Sockets Layer）是早期的安全协议，后来发展为 TLS（Transport Layer Security），现在通常使用的是 TLS 协议。

工作原理

- HTTP
 - 客户端（如浏览器）向服务器发送 HTTP 请求，请求中包含请求方法（如 GET、POST 等）、请求的 URL、请求头和请求体等信息。
 - 服务器接收到请求后，根据请求信息进行处理，并返回 HTTP 响应，响应中包含响应状态码、响应头和响应体等信息。
 - 整个数据传输过程都是以明文形式进行的，容易被中间人窃取或篡改。
- HTTPS
 - **SSL/TLS 握手阶段**：客户端和服务端首先进行 SSL/TLS 握手，协商使用的加密算法、交换密钥等。这个过程包括客户端向服务器发送支持的 SSL/TLS 版本和加密算法列表，服务器选择合适的版本和算法并返回证书等信息。客户端验证服务器证书的有效性，生成会话密钥。
 - **数据传输阶段**：在握手完成后，客户端和服务端使用会话密钥对数据进行加密和解密，确保数据在传输过程中的保密性和完整性。

安全性

- HTTP：由于数据以明文传输，存在以下安全风险：
 - **窃听风险**：中间人可以截取传输的数据，获取用户的敏感信息，如用户名、密码、信用卡号等。
 - **篡改风险**：中间人可以修改传输的数据，例如修改交易金额、篡改网页内容等。
 - **冒充风险**：中间人可以冒充服务器向客户端发送虚假响应，或者冒充客户端向服务器发送虚假请求。
- HTTPS：通过加密和身份验证机制，有效解决了 HTTP 的安全问题：
 - **加密传输**：使用对称加密和非对称加密相结合的方式对数据进行加密，即使数据被截取，中间人也无法解密。
 - **身份验证**：服务器通过 SSL 证书证明自己的身份，客户端可以验证服务器的真实性，防止中间人冒充。

端口号

- HTTP：默认使用端口号 80。
- HTTPS：默认使用端口号 443。

性能

- **HTTP**: 由于不需要进行 SSL/TLS 握手和数据加密，HTTP 的传输速度相对较快，尤其是在对安全性要求不高的场景下，能够提供更高的性能。
- **HTTPS**: SSL/TLS 握手过程会增加一定的延迟，数据加密和解密也会消耗一定的计算资源，因此 HTTPS 的性能相对较低。不过，随着硬件性能的提升和优化技术的发展，HTTPS 的性能损耗已经逐渐减小。

证书

- **HTTP**: 不需要使用 SSL 证书。
- **HTTPS**: 需要使用 SSL 证书来验证服务器的身份。SSL 证书由受信任的证书颁发机构（CA）颁发，分为免费证书和付费证书。付费证书通常具有更高的安全性和可信度，而免费证书可能存在一定的限制。

应用场景

- **HTTP**: 适用于对安全性要求不高的场景，如一些静态网页、新闻网站等。
- **HTTPS**: 适用于涉及用户敏感信息的场景，如电子商务网站、网上银行、社交平台等。使用 HTTPS 可以保护用户的隐私和数据安全，增强用户对网站的信任度。

总结对比

对比项	HTTP	HTTPS
安全性	明文传输，存在窃听、篡改和冒充风险	加密传输，通过身份验证，安全性高
端口号	80	443
性能	速度快，无额外开销	有 SSL/TLS 握手延迟和加密计算开销
证书	不需要	需要 SSL 证书
应用场景	对安全性要求不高的场景	涉及敏感信息的场景

图解 HTTP 和 HTTPS

我将分别描述 HTTP 和 HTTPS 工作原理的流程图结构，并使用 mermaid 代码来生成相应的流程图。

HTTP 工作原理流程图

流程图结构描述

1. 客户端发起 HTTP 请求，包含请求方法、URL、请求头和请求体等信息。
2. 请求通过网络传输到服务器。
3. 服务器接收请求，进行处理。
4. 服务器生成 HTTP 响应，包含响应状态码、响应头和响应体等信息。
5. 响应通过网络传输回客户端。
6. 客户端接收响应并进行处理。

mermaid 代码



HTTPS 工作原理流程图

流程图结构描述

1. 客户端发起 HTTPS 请求。
2. 客户端和服务器进行 SSL/TLS 握手：
 - 客户端发送支持的 SSL/TLS 版本和加密算法列表。
 - 服务器选择合适的版本和算法，并返回证书。
 - 客户端验证证书有效性，生成会话密钥。
3. 握手完成后，进入数据传输阶段：
 - 客户端使用会话密钥加密数据并发送请求。
 - 请求通过网络传输到服务器。
 - 服务器使用会话密钥解密请求，进行处理。
 - 服务器生成响应，使用会话密钥加密响应。
 - 响应通过网络传输回客户端。
 - 客户端使用会话密钥解密响应。
4. 客户端接收并处理响应。

mermaid 代码

这两个流程图分别展示了 HTTP 和 HTTPS 的工作原理。HTTP 流程相对简单，主要是客户端和服务端之间的请求和响应传输。而 HTTPS 流程增加了 SSL/TLS 握手阶段，用

于建立安全连接，之后的数据传输通过加密进行，以确保数据的安全性。

SSL/TLS 证书验证

SSL/TLS 证书验证网站身份是保障 HTTPS 通信安全的关键环节，它涉及多个步骤和技术，下面将详细解释其验证过程。

证书的基本构成

在理解验证过程之前，先了解一下 SSL/TLS 证书的基本构成。一份 SSL/TLS 证书通常包含以下信息：

- **网站域名**：表明该证书所保护的网站域名。
- **证书持有者信息**：可能是个人或组织的详细信息，如公司名称、地址等。
- **公钥**：用于加密通信数据和验证签名。
- **证书颁发机构（CA）信息**：即颁发该证书的 CA 机构名称。
- **证书有效期**：规定了证书的有效使用期限。
- **数字签名**：由 CA 机构使用其私钥对上述信息进行签名生成，用于验证证书的完整性和真实性。

验证网站身份的具体步骤

1. 客户端发起连接请求

当用户在浏览器中输入一个 HTTPS 网址并按下回车键后，浏览器会向该网站的服务器发起连接请求。这个请求中包含了客户端支持的 SSL/TLS 版本和加密算法列表等信息。

2. 服务器发送证书

服务器接收到客户端的请求后，会从其证书存储中选择合适的 SSL/TLS 证书，并将该证书发送给客户端。

3. 客户端验证证书链

客户端（通常是浏览器）接收到服务器发送的证书后，首先会验证证书链的有效性。证书链是由多个证书组成的层级结构，从服务器证书开始，依次链接到中间证书（如果有），最终指向根证书。

- **根证书**：根证书是由受信任的 CA 机构自己颁发的证书，浏览器和操作系统通常会

预先内置一些知名 CA 机构的根证书。这些根证书是信任的基础，用于验证其他证书的真实性。

- **验证过程：**客户端会检查证书链中每个证书的数字签名。具体来说，客户端使用上级证书的公钥来验证下级证书的数字签名。如果签名验证通过，说明该证书确实是由上级 CA 机构颁发的，并且证书内容在传输过程中没有被篡改。例如，客户端会使用根证书的公钥验证中间证书的签名，再使用中间证书的公钥验证服务器证书的签名。如果证书链中的任何一个证书的签名验证失败，客户端会认为该证书不可信。

4. 验证证书的有效期

客户端会检查服务器证书的有效期，确保当前时间在证书规定的有效期限内。如果证书已经过期或者尚未生效，客户端会提示用户该证书存在问题，可能存在安全风险。

5. 验证证书中的域名信息

客户端会将证书中记录的域名信息与用户访问的实际域名进行比对。只有当两者完全一致时，才能确保该证书确实是为当前访问的网站颁发的。例如，如果用户访问的是 `example.com`，而证书中的域名是 `another.com`，那么客户端会认为该证书不匹配，存在安全风险。

6. 扩展验证（针对 EV 证书）

如果服务器使用的是扩展验证（EV）证书，客户端还会进行更严格的验证。CA 机构在颁发 EV 证书之前，会对申请证书的组织进行深入调查，包括核实组织的合法存在、注册地址、联系方式等信息。客户端在验证 EV 证书时，会检查这些额外的组织信息，并且在浏览器地址栏中显示绿色的组织名称，以向用户明确展示网站的真实身份。

验证失败的处理

如果在上述验证过程中任何一个步骤出现问题，客户端（如浏览器）会向用户发出警告，提示用户该网站的证书存在风险。常见的警告信息包括证书过期、证书域名不匹配、证书链验证失败等。用户可以根据这些警告信息决定是否继续访问该网站，但通常不建议在证书验证失败的情况下继续进行敏感操作，如登录账号、进行在线支付等。

通过以上一系列的验证步骤，SSL/TLS 证书能够有效地验证网站的身份，确保用户与真实、合法的网站进行通信，从而保障数据传输的安全性和可靠性。

18. 前端代码部署流程

在公司中，前端项目采用自动化部署能显著提升效率、减少人为失误。以下为你详细介绍常见的前端自动化部署基本流程：

1. 代码管理与提交

- **版本控制**：团队成员使用版本控制系统（如 Git）管理前端代码。一般会在本地开发分支上进行代码编写，完成功能开发或问题修复后，将代码提交到本地仓库。
- **代码审查**：开发人员将本地代码推送到远程仓库的特定分支（如 `feature` 分支）后，发起代码审查（Pull Request）。团队中的其他成员对代码进行审查，检查代码质量、功能实现、是否符合规范等，确保代码的正确性和可维护性。
- **合并代码**：代码审查通过后，将开发分支的代码合并到主分支（如 `master` 或 `main`）。

2. 触发自动化部署

- **配置持续集成/持续部署（CI/CD）工具**：常见的 CI/CD 工具有 Jenkins、GitLab CI/CD、GitHub Actions 等。在项目中配置这些工具，当代码发生特定事件（如代码推送到主分支、创建新的标签等）时，自动触发部署流程。
- **事件监听**：CI/CD 工具会持续监听代码仓库的变化。以 GitHub Actions 为例，在项目的 `.github/workflows` 目录下创建 YAML 配置文件，定义触发条件和部署步骤。例如：

```
name: Frontend Deployment
on:
  push:
    branches:
      - main
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      # 后续步骤将在此处定义
```

3. 环境准备

- **拉取代码**：CI/CD 工具接收到触发事件后，首先会从代码仓库中拉取最新的代码到构建环境。在 GitHub Actions 的配置文件中添加如下步骤：

```
steps:
  - name: Checkout code
```

```
uses: actions/checkout@v2
```

- **安装依赖**：根据项目的 `package.json` 文件，安装项目所需的依赖。对于前端项目，通常使用 `npm` 或 `yarn` 进行安装。

```
- name: Install dependencies  
  run: npm install
```

4. 代码构建

- **打包优化**：使用构建工具（如 Webpack、Vite 等）对前端代码进行打包优化，将多个模块打包成一个或多个文件，压缩代码体积，提高加载速度。以 Vue 项目为例：

```
- name: Build project  
  run: npm run build
```

- **代码检查**：在构建过程中，可以进行代码检查，如使用 ESLint 检查 JavaScript 代码的语法和风格错误，使用 Stylelint 检查 CSS 代码的规范。

```
- name: Lint code  
  run: npm run lint
```

5. 测试

- **单元测试**：编写单元测试用例，使用测试框架（如 Jest、Mocha 等）对代码的各个模块进行独立测试，确保代码的正确性。

```
- name: Run unit tests  
  run: npm test
```

- **集成测试**：对多个模块进行集成测试，验证它们之间的协作是否正常。可以使用 Cypress、TestCafe 等工具进行端到端测试。

6. 部署到目标环境

- **选择目标环境**：目标环境可以是开发环境、测试环境、预发布环境或生产环境。根据不同的需求和配置，将打包好的代码部署到相应的环境中。
- **上传代码**：将构建好的静态文件上传到服务器或静态网站托管服务。如果使用云服务器，可以使用 `scp` 或 `rsync` 命令上传文件；如果使用静态网站托管服务，按

照其提供的 API 或界面进行上传。

```
- name: Deploy to server
  uses: appleboy/ssh-action@master
  with:
    host: ${ secrets.SERVER_HOST }
    username: ${ secrets.SERVER_USERNAME }
    password: ${ secrets.SERVER_PASSWORD }
    script: |
      rm -rf /path/to/deploy/*
      scp -r dist/ /path/to/deploy
```

- **配置服务器：**在服务器上配置 Web 服务器（如 Nginx、Apache），确保静态文件能够正确访问。

7. 监控与反馈

- **监控系统：**部署完成后，使用监控系统（如 Prometheus、Grafana 等）对应用的性能、可用性等进行监控，及时发现和处理问题。
- **反馈机制：**建立反馈机制，将部署结果（成功或失败）通知相关人员。可以通过邮件、Slack 等方式发送通知。

通过以上自动化部署流程，公司可以实现前端代码的快速、稳定发布，提高开发效率和产品质量。

19. 截取最大不连续子字符串

在前端面试中，“截取最大不连续子字符串”通常指的是找出字符串中最长的不包含重复字符的子字符串。下面为你详细介绍解题思路、代码实现及复杂度分析，助你清晰掌握该问题。

解题思路

可以使用滑动窗口算法来解决这个问题。滑动窗口是一种常用的算法技巧，它通过两个指针（左指针和右指针）来维护一个窗口，不断移动右指针扩大窗口，当窗口内出现重复字符时，移动左指针缩小窗口，直到窗口内不再有重复字符。在这个过程中，记录窗口的最大长度以及对应的子字符串。

JavaScript 代码实现


```

function longestNonRepeatingSubstring(s) {
  // 用于存储字符及其在字符串中的索引
  const charIndexMap = new Map();
  // 左指针，初始化为 0
  let left = 0;
  // 最大不重复子字符串的长度
  let maxLength = 0;
  // 最大不重复子字符串
  let maxSubstring = '';

  // 遍历字符串，右指针不断向右移动
  for (let right = 0; right < s.length; right++) {
    const char = s[right];
    // 如果字符已经在 map 中，并且其索引大于等于左指针
    if (charIndexMap.has(char) && charIndexMap.get(char) >= left) {
      // 移动左指针到重复字符的下一个位置
      left = charIndexMap.get(char) + 1;
    }
    // 更新字符的最新索引
    charIndexMap.set(char, right);
    // 当前窗口的长度
    const currentLength = right - left + 1;
    // 如果当前窗口长度大于最大长度
    if (currentLength > maxLength) {
      // 更新最大长度
      maxLength = currentLength;
      // 截取最大不重复子字符串
      maxSubstring = s.slice(left, right + 1);
    }
  }

  return maxSubstring;
}

// 测试示例
const input = "abcabcbb";
const result = longestNonRepeatingSubstring(input);
console.log(result);

```

代码解释

1. 初始化：

- `charIndexMap`：使用 `Map` 对象来存储字符及其在字符串中的索引。
- `left`：左指针，初始化为 0，表示滑动窗口的起始位置。

- `maxLength`：用于记录最大不重复子字符串的长度，初始化为 0。
- `maxSubstring`：用于记录最大不重复子字符串，初始化为空字符串。

2. 遍历字符串：

- 使用 `right` 指针从左到右遍历字符串。
- 对于每个字符 `char`，检查它是否已经在 `charIndexMap` 中，并且其索引大于等于 `left`。如果是，则说明该字符在当前窗口内重复，需要移动 `left` 指针到重复字符的下一个位置。
- 更新 `charIndexMap` 中该字符的最新索引。

3. 更新最大长度和子字符串：

- 计算当前窗口的长度 `currentLength`。
- 如果 `currentLength` 大于 `maxLength`，则更新 `maxLength` 并截取当前窗口对应的子字符串作为 `maxSubstring`。

4. 返回结果：

- 遍历结束后，返回 `maxSubstring`。

复杂度分析

- **时间复杂度**： $O(n)$ ，其中 n 是字符串的长度。因为只需要遍历一次字符串，每个字符最多被访问两次（一次是右指针访问，一次是左指针可能的访问）。
- **空间复杂度**： $O(k)$ ，其中 k 是字符集的大小。在本题中，如果字符串只包含 ASCII 字符，那么 k 为 128；如果包含 Unicode 字符， k 会更大。主要的空间开销是 `charIndexMap` 所占用的空间。

通过上述的思路、代码和解释，你应该能够清晰地掌握如何截取字符串中最大不连续子字符串的方法。在面试中，你可以根据实际情况对代码进行适当的优化和扩展。

20. 对比2个版本号

在前端开发中，对比两个版本号是一个常见的需求，例如在检查软件更新时，需要判断当前版本和最新版本的大小关系。版本号通常由多个数字和点号组成，如 `1.2.3`。下面为你详细介绍解题思路、代码实现以及复杂度分析，帮助你清晰掌握如何对比两个版本号。

解题思路

要对比两个版本号，可以按照以下步骤进行：

1. **分割版本号**：使用点号（.）将版本号分割成数组，数组中的每个元素代表版本号的一部分，如 1.2.3 分割后为 ['1', '2', '3']。
2. **逐位比较**：从数组的第一个元素开始，逐位比较两个版本号的对应部分。
 - 如果对应部分的数字不同，则数字大的版本号更大。
 - 如果对应部分的数字相同，则继续比较下一位。
3. **处理长度不一致的情况**：如果一个版本号的位数比另一个多，需要继续比较多出的部分。如果多出的部分都是 0，则两个版本号相等；否则，位数多且非零部分大的版本号更大。

JavaScript 代码实现

```
function compareVersion(version1, version2) {  
    // 分割版本号为数组  
    const v1 = version1.split('.');  
    const v2 = version2.split('.');  
    // 获取两个数组的最大长度  
    const maxLength = Math.max(v1.length, v2.length);  
  
    for (let i = 0; i < maxLength; i++) {  
        // 将当前位转换为数字，如果该位不存在则默认为 0  
        const num1 = parseInt(v1[i] || 0);  
        const num2 = parseInt(v2[i] || 0);  
  
        if (num1 > num2) {  
            return 1; // version1 大于 version2  
        } else if (num1 < num2) {  
            return -1; // version1 小于 version2  
        }  
        // 如果当前位相等，则继续比较下一位  
    }  
  
    return 0; // 两个版本号相等  
}  
  
// 测试示例  
const version1 = "1.2.3";  
const version2 = "1.2.4";  
const result = compareVersion(version1, version2);  
console.log(result);
```

代码解释

1. **分割版本号**：使用 `split('.')` 方法将版本号字符串分割成数组，方便逐位比较。
2. **获取最大长度**：使用 `Math.max(v1.length, v2.length)` 获取两个版本号数组的最大长度，确保能够处理位数不同的版本号。
3. **逐位比较**：使用 `for` 循环遍历数组，将当前位的字符串转换为数字进行比较。如果 `num1 > num2`，则返回 1；如果 `num1 < num2`，则返回 -1；如果相等，则继续比较下一位。
4. **处理长度不一致的情况**：如果一个版本号的位数比另一个多，在比较时会将缺失的部分视为 0。如果遍历完所有位都相等，则返回 0 表示两个版本号相等。

复杂度分析

- **时间复杂度**： $O(n)$ ，其中 n 是两个版本号中较长的位数。因为只需要遍历一次数组，每个元素最多被访问一次。
- **空间复杂度**： $O(n)$ ，主要的空间开销是存储分割后的版本号数组。

21. 在数组中查找重复出现两次的数字

以下为你详细介绍三种查找数组中重复出现两次数字的方法，包含使用 `Set` 和 `Map` 的方式，每种方法都配有思路、代码示例和复杂度分析，方便你理解和掌握。

方法一：使用 `Set`

思路

利用 `Set` 只能存储唯一值的特性。遍历数组，若数字不在 `Set` 中，将其添加进去；若已存在，则表明该数字重复，把它添加到结果 `Set` 里。最后将结果 `Set` 转换为数组返回。

代码示例

```
function findDuplicatesWithSet(arr) {  
  const seen = new Set();  
  const duplicates = new Set();  
  
  for (const num of arr) {
```

```
        if (seen.has(num)) {
            duplicates.add(num);
        } else {
            seen.add(num);
        }
    }

    return Array.from(duplicates);
}

// 测试
const array1 = [1, 2, 3, 2, 4, 3, 5];
console.log(findDuplicatesWithSet(array1));
```

复杂度分析

- **时间复杂度：** $O(n)$ ，这里的 n 是数组的长度，因为只需要对数组进行一次遍历。
- **空间复杂度：** $O(k)$ ，其中 k 是数组中不同数字的个数。在最坏情况下， k 等于 n 。

方法二：使用 Map

思路

使用 `Map` 来记录每个数字的出现次数。遍历数组，对于每个数字，若它不在 `Map` 中，将其添加并把出现次数设为 1；若已存在，将出现次数加 1。最后遍历 `Map`，找出出现次数为 2 的数字并添加到结果数组。

代码示例

```
function findDuplicatesWithMap(arr) {
    const numMap = new Map();
    const duplicates = [];

    for (const num of arr) {
        if (numMap.has(num)) {
            numMap.set(num, numMap.get(num) + 1);
        } else {
            numMap.set(num, 1);
        }
    }

    // 找出出现次数为 2 的数字
    for (const [num, count] of numMap) {
        if (count === 2) {
            duplicates.push(num);
        }
    }

    return duplicates;
}
```

```

    for (const [num, count] of numMap) {
      if (count === 2) {
        duplicates.push(num);
      }
    }

    return duplicates;
  }

  // 测试
  const array2 = [1, 2, 3, 2, 4, 3, 5];
  console.log(findDuplicatesWithMap(array2));

```

复杂度分析

- **时间复杂度：** $O(n)$ ，需要遍历数组一次来统计数字出现次数，再遍历 `Map` 一次找出出现两次的数字，整体为线性时间复杂度。
- **空间复杂度：** $O(k)$ ， k 是数组中不同数字的个数，主要用于存储 `Map` 中的键值对。

方法三：排序后遍历

思路

先对数组进行排序，这样相同的数字会相邻。然后遍历排序后的数组，比较相邻元素，若相等且不等于前一对相等元素，将其添加到结果数组。

代码示例

```

function findDuplicatesBySorting(arr) {
  const sortedArr = arr.slice().sort((a, b) => a - b);
  const duplicates = [];

  for (let i = 1; i < sortedArr.length; i++) {
    if (sortedArr[i] === sortedArr[i - 1]) {
      if (duplicates.length === 0 || sortedArr[i] !==
duplicates[duplicates.length - 1]) {
        duplicates.push(sortedArr[i]);
      }
    }
  }

  return duplicates;
}

```

```
}

// 测试
const array3 = [1, 2, 3, 2, 4, 3, 5];
console.log(findDuplicatesBySorting(array3));
```

复杂度分析

- **时间复杂度：** $O(n \log n)$ ，主要是排序操作的时间复杂度，之后遍历数组的时间复杂度是 $O(n)$ 。
- **空间复杂度：** $O(m)$ ，其中 m 是重复出现两次的数字的个数，主要用于存储结果数组。

22. Vue的SEO解决方案

Vue 是一个用于构建用户界面的渐进式 JavaScript 框架，由于它构建的单页面应用（SPA）内容通常由 JavaScript 动态渲染，搜索引擎爬虫可能难以抓取完整信息，因此需要采取一些 SEO 解决方案。以下是 5 个常用的 Vue SEO 解决方案：

1. 服务器端渲染（SSR）

- **原理：**服务器端渲染是指在服务器端将 Vue 组件渲染成 HTML 字符串，再将完整的 HTML 页面返回给客户端。搜索引擎爬虫直接抓取服务器返回的 HTML 页面，就能获取到完整的页面内容。
- **实现方式：**可以借助 Nuxt.js 框架，它基于 Vue.js，封装了许多复杂的 SSR 配置，让开发者能更便捷地实现服务器端渲染。例如，使用 Nuxt.js 创建项目后，编写页面组件，框架会自动处理服务器端渲染的流程。
- **优点：**搜索引擎能够抓取到完整的页面内容，极大地有利于 SEO；同时，首屏加载速度快，用户体验好。
- **缺点：**开发和部署过程较为复杂，需要服务器具备一定的性能支持，会增加成本。

2. 静态站点生成（SSG）

- **原理：**在项目构建阶段，提前生成所有页面的静态 HTML 文件。这些静态文件可以直接部署到静态文件服务器上，搜索引擎爬虫可以轻松抓取。
- **实现方式：**使用 VitePress、VuePress 等工具。以 VuePress 为例，按照其规范编写文档和页面，执行构建命令，就会生成静态站点。
- **优点：**生成的静态页面加载速度快，对服务器要求低，部署简单；静态文件便于搜

索引引擎爬虫抓取。

- **缺点：**不适合内容实时更新的场景，每次内容更新都需要重新构建和部署。

3. 预渲染 (Prerendering)

- **原理：**在项目构建过程中，针对特定的路由生成对应的静态 HTML 文件。这些文件会被存放在构建输出目录中，供搜索引擎爬虫抓取。
- **实现方式：**使用 `prerender - spa - plugin` 插件。在 `vue.config.js` 中进行配置，指定需要预渲染的路由和渲染参数，构建时插件会自动生成静态 HTML 文件。
- **优点：**实现相对简单，不需要复杂的服务器配置；生成的静态文件可部署到任意静态文件服务器。
- **缺点：**只适用于内容不经常变动的页面；如果路由过多，构建时间会变长。

4. 动态元标签管理

- **原理：**合理设置页面的元数据，如 `title`（页面标题）、`meta description`（页面描述）、`meta keywords`（页面关键词）等，有助于搜索引擎更好地理解页面内容，从而在搜索结果中更准确地展示页面。
- **实现方式：**使用 `vue - meta` 插件。在 Vue 组件中通过 `metaInfo` 属性来设置元数据，例如：

```
export default {
  metaInfo: {
    title: '这是页面标题',
    meta: [
      {
        name: 'description',
        content: '这是页面的详细描述'
      },
      {
        name: 'keywords',
        content: '关键词1, 关键词2'
      }
    ]
  }
};
```

- **优点：**实现简单，能有效提升页面在搜索引擎中的展示效果。
- **缺点：**只能优化元数据，无法解决爬虫无法获取动态内容的问题，需要结合其他 SEO 方法使用。

5. 结构化数据标记

- **原理：**利用结构化数据标记（如 JSON - LD、Microdata 等）向搜索引擎提供更详细的页面内容信息，帮助搜索引擎更好地理解页面的主题和结构，进而在搜索结果中展示更丰富的信息。
- **实现方式：**在 Vue 组件中添加结构化数据标记。例如使用 JSON - LD：

```
<template>
  <div>
    <!-- 页面内容 -->
  </div>
</template>

<script>
export default {
  mounted() {
    const script = document.createElement('script');
    script.type = 'application/ld+json';
    script.textContent = JSON.stringify({
      "@context": "https://schema.org",
      "@type": "Article",
      "headline": "文章标题",
      "datePublished": "2024-01-01",
      "author": {
        "@type": "Person",
        "name": "作者姓名"
      }
    });
    document.head.appendChild(script);
  }
};
</script>
```

- **优点：**可以提高页面在搜索结果中的展示效果，如显示富媒体搜索结果。
- **缺点：**需要开发者了解结构化数据标记的规范，不同搜索引擎对结构化数据的支持可能存在差异。

23. 手写promise

这段代码是一个自定义的 `Promise` 实现，遵循了 `Promise/A+` 规范。下面我们逐步解析这段代码：

1. 状态常量定义

```
const P = 'pending',  
      F = 'fulfill',  
      R = 'reject'
```

这里定义了三个常量，分别代表 `Promise` 的三种状态：

- `P` (`pending`)：初始状态，表示 `Promise` 还未完成。
- `F` (`fulfill`)：成功状态，表示 `Promise` 已经成功完成。
- `R` (`reject`)：失败状态，表示 `Promise` 已经失败。

2. `resolvePromise` 函数

```
function resolvePromise(promise, x, resolve, reject) {  
  if (x === promise) {  
    reject(TypeError('err'))  
  }  
  else if (typeof x === 'object' && x !== null || typeof x ===  
'function') {  
    let called = false  
    try {  
      const then = x.then  
      if (typeof then === 'function') {  
        const resolvePromiseThen = y => {  
          if (called) return  
          called = true  
          resolvePromise(promise, y, resolve, reject)  
        }  
        const rejectPromiseThen = r => {  
          if (called) return  
          called = true  
          reject(r)  
        }  
        then.call(x, resolvePromiseThen, rejectPromiseThen)  
      }  
    }  
    catch (err) {  
      if (called) return  
      called = true  
      reject(err)  
    }  
  }  
  else {  

```

```
    resolve(x)
  }
}
```

这个函数的作用是解析 `Promise` 的结果，处理各种情况：

- **避免循环引用**：如果 `x` 就是当前 `Promise` 本身，会抛出一个 `TypeError` 错误。
- **处理 `x` 是对象或函数的情况**：尝试获取 `x` 的 `then` 方法，如果 `then` 是一个函数，就调用它，并传入 `resolvePromiseThen` 和 `rejectPromiseThen` 作为回调函数。`called` 变量用于确保 `resolve` 或 `reject` 只被调用一次。
- **其他情况**：如果 `x` 不是对象或函数，直接调用 `resolve(x)`。

3. `myPromise` 类

构造函数 `constructor`

```
constructor(executor) {
  this.state = P
  this.value = null
  this.fulfilledCallbacks = []
  this.rejectedCallbacks = []

  const resolve = value => {
    if (this.state === P) {
      this.state = F
      this.value = value
      this.fulfilledCallbacks.forEach(cb => cb(this.value))
    }
  }

  const reject = reason => {
    if (this.state === P) {
      this.state = R
      this.value = reason
      this.rejectedCallbacks.forEach(cb => cb(this.value))
    }
  }

  try {
    executor(resolve, reject)
  }
  catch (err) {
    throw err
  }
}
```

- 初始化 `Promise` 的状态为 `pending`，值为 `null`，并创建两个数组用于存储成功和失败的回调函数。
- 定义 `resolve` 和 `reject` 函数，用于改变 `Promise` 的状态，并执行相应的回调函数。
- 调用传入的 `executor` 函数，并传入 `resolve` 和 `reject` 作为参数。如果 `executor` 函数抛出错误，会直接抛出该错误。

then 方法

```
then(onFulfilled, onRejected) {
  const F1 = typeof onFulfilled === 'function' ? onFulfilled : value =>
value
  const R1 = typeof onRejected === 'function' ? onRejected : reason =>
{ throw (reason) }
  const p2 = new myPromise((resolve, reject) => {
    const handleF = value => {
      setTimeout(() => {
        try {
          const x = F1(value)
          resolvePromise(p2, x, resolve, reject)
        }
        catch (err) {
          reject(err)
        }
      }, 0);
    }
    const handleR = reason => {
      setTimeout(() => {
        try {
          const x = R1(reason)
          resolvePromise(p2, x, resolve, reject)
        }
        catch (err) {
          reject(err)
        }
      }, 0);
    }
    // 处理同步
    if (this.state === F) {
      handleF(this.value)
    }
    else if (this.state === R) {
      handleR(this.value)
    }
  })
}
```

```

else {
  // 处理异步，将回调函数，放入成功回调 和 失败回调数组中
  this.fulfilledCallbacks.push(value => {
    handleF(value)
  })
  this.rejectedCallbacks.push(reason => {
    handleR(reason)
  })
}
})
return p2
}

```

- 首先，确保 `onFulfilled` 和 `onRejected` 是函数，如果不是则使用默认函数。
- 创建一个新的 `Promise` `p2`，用于链式调用。
- 定义 `handleF` 和 `handleR` 函数，分别处理成功和失败的情况。使用 `setTimeout` 确保异步执行，避免同步调用可能带来的问题。
- 根据当前 `Promise` 的状态，执行相应的处理函数。如果状态是 `pending`，将处理函数存入回调数组中，等待状态改变时执行。
- 最后返回新的 `Promise` `p2`。

4. 使用自定义 `Promise`

```

const p = new myPromise((resolve, reject) => {
  // resolve(1)
  // reject(2)
  setTimeout(() => {
    // resolve(1)
    reject(2)
  }, 1000);
})

p.then().then().then(
  res => {
    console.log('resolve-', res)
  },
  err => {
    console.log('reject-', err)
  }
)

```

- 创建一个新的 `myPromise` 实例 `p`，在 `setTimeout` 中调用 `reject(2)`，模拟异步失败的情况。

- 调用 `then` 方法进行链式调用，最终在最后一个 `then` 方法中处理成功或失败的结果。由于 `p` 被 `reject` 了，所以会执行 `err` 回调函数，输出 `reject- 2`。

综上所述，这段代码实现了一个基本的 `Promise` 功能，包括状态管理、链式调用和异步处理。

24.防抖 节流

```
// 防抖 debounce
function debounce(fn, time) {
  let timer = null
  return function (...args) {
    if (timer) clearTimeout(timer)
    timer = setTimeout(() => {
      fn.apply(this, args)
    }, time);
  }
}

// 节流 定时器
function throttleTimer(fn, limit) {
  let timer = null;
  return function (...args) {
    if (!timer) {
      fn.apply(this, args);
      timer = setTimeout(() => {
        timer = null;
      }, limit);
    }
  };
}

// 节流 throttle 时间戳
function throttle(fn, time) {
  let lastTime = new Date().getTime()
  return function (...args) {
    let now = new Date().getTime()
    if (now - lastTime > time) {
      fn.apply(this, args)
      lastTime = now
    }
  }
}
```

25.js实现forEach

```
Array.prototype.myForEach = function (cb, arg) {
  if (this === null || this === undefined) {
    throw new TypeError('Array.prototype.myForEach called on null or undefined');
  }
  if (typeof cb !== 'function') {
    throw new TypeError(`${cb} is not a function`);
  }
  let arr = Object(this);
  let len = arr.length >>> 0; //>>>确保数组长度是小于32位的整数
  for (let i = 0; i < len; i++) {
    if (i in arr) { //处理稀疏数组 跳过空值
      cb.call(arg, arr[i], i, arr);
    }
  }
};
```

```
// 测试代码
const testArray = [1, 2, , 4];
testArray.myForEach((item, index) => {
  console.log( Index: ${index}, Value: ${item} );
});
```

26. 睡眠函数 sleep() 隔1秒再打印

```
<!-- type="module": 这个属性告诉浏览器，<script> 标签里的代码是一个 ES 模块，这样就能支持顶层 await 特性了。 -->
<script type="module">
  // sleep 函数用于创建一个 Promise，在指定时间后 resolve
  async function sleep(timer) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve(1);
      }, timer);
    });
  }

  // 先打印 1
```

```
console.log(1);  
// 等待 1 秒  
await sleep(1000);  
// 打印 2  
console.log(2);  
</script>
```

27. Hash和History路由模式的区别

在前端开发中，路由模式是单页面应用（SPA）中非常重要的概念，Hash 和 History 是两种常见的路由模式。下面从多个方面详细解说它们的区别，帮助你在面试中让面试官钦佩不已。

1. 基本概念与原理

Hash 路由模式

- **原理：**Hash 路由是基于 URL 中的哈希值（即 `#` 后面的部分）来实现的。当哈希值发生变化时，浏览器不会向服务器发送请求，而是触发 `hashchange` 事件，前端可以监听这个事件并根据不同的哈希值来渲染不同的页面内容。
- **示例：**`http://example.com/#/home`，其中 `#/home` 就是哈希值。当哈希值从 `#/home` 变为 `#/about` 时，前端代码会捕捉到这个变化并进行相应的页面切换。

History 路由模式

- **原理：**History 路由模式利用 HTML5 的 `History API` 来实现。`History API` 提供了 `pushState` 和 `replaceState` 方法，通过这两个方法可以改变浏览器的历史记录，同时更新 URL，但不会向服务器发送请求。当用户点击浏览器的前进、后退按钮或者调用相关的历史操作方法时，会触发 `popstate` 事件，前端可以监听这个事件并根据不同的 URL 来渲染不同的页面内容。
- **示例：**使用 `history.pushState(null, '', '/about')` 可以将当前 URL 变为 `http://example.com/about`，并且在浏览器历史记录中添加一条新记录。

2. URL 表现形式

Hash 路由模式

- URL 中会包含一个 `#` 符号，后面跟着哈希值。哈希值的变化不会影响服务器请求，仅仅是前端用来识别不同页面状态的标识。

- 例如：`http://example.com/#/products/123`，这种 URL 看起来相对不够简洁，并且在某些场景下可能会给用户造成困惑。

History 路由模式

- URL 看起来更像传统的 URL，没有 `#` 符号，更加简洁美观，符合用户对传统网站 URL 的认知。
- 例如：`http://example.com/products/123`，这种 URL 更易于分享和记忆。

3. 服务器端处理

Hash 路由模式

- 由于哈希值的变化不会触发服务器请求，服务器只需要返回单页面应用的入口文件（通常是 `index.html`）即可。服务器不需要对不同的哈希值进行特殊处理，因此对服务器的配置要求较低。
- 无论用户访问 `http://example.com/#/home` 还是 `http://example.com/#/about`，服务器都会返回相同的 `index.html` 文件，然后由前端根据哈希值来渲染不同的页面内容。

History 路由模式

- 当用户直接访问某个 URL 或者刷新页面时，浏览器会向服务器发送请求。由于 History 路由的 URL 看起来像传统的 URL，服务器需要进行相应的配置，确保无论用户访问哪个 URL，都能返回单页面应用的入口文件。
- 例如，在 Node.js 中使用 Express 框架，可以通过以下配置来处理 History 路由：

```
const express = require('express');
const app = express();

// 静态文件服务
app.use(express.static(__dirname + '/public'));

// 处理所有路由，返回 index.html
app.get('*', function(req, res) {
  res.sendFile(__dirname + '/public/index.html');
});

const port = 3000;
app.listen(port, function() {
  console.log(`Server is running on port ${port}`);
});
```

4. 兼容性

Hash 路由模式

- 兼容性非常好，几乎所有的浏览器都支持哈希值的变化和 `hashchange` 事件。因此，在需要兼容旧版本浏览器的项目中，Hash 路由是一个比较稳妥的选择。

History 路由模式

- 依赖于 HTML5 的 `History API`，因此在一些旧版本的浏览器中可能不支持。在使用 History 路由模式时，需要考虑项目的目标用户群体，如果需要兼容旧版本浏览器，可能需要进行额外的处理或者降级到 Hash 路由模式。

5. 浏览器历史记录管理

Hash 路由模式

- 哈希值的变化会在浏览器的历史记录中留下记录，用户可以通过点击浏览器的前进、后退按钮来切换不同的哈希值状态。但是，由于哈希值的变化不会向服务器发送请求，因此历史记录中的每个状态对应的页面内容实际上是由前端代码控制的。

History 路由模式

- 利用 `History API` 可以更加灵活地管理浏览器的历史记录。`pushState` 方法可以在不刷新页面的情况下向历史记录中添加一条新记录，`replaceState` 方法可以替换当前的历史记录。这种方式可以实现更复杂的历史记录管理，例如在不影响用户操作的情况下修改 URL。

总结

Hash 路由和 History 路由各有优缺点，在选择使用哪种路由模式时，需要根据项目的具体需求来决定。如果项目需要兼容旧版本浏览器或者对服务器配置要求较低，可以选择 Hash 路由模式；如果项目追求更简洁的 URL 和更灵活的历史记录管理，并且目标用户群体使用的是较新的浏览器，那么 History 路由模式是一个更好的选择。

28. 前端性能优化

前端性能优化是提升用户体验、提高网站或应用响应速度的关键。以下从多个维度详细

阐述前端性能优化的方法：

代码层面优化

1. HTML 优化

- **精简代码**：去除不必要的注释、空格和换行符，减少 HTML 文件的大小。可以使用工具如 HTMLMinifier 对代码进行压缩。
- **合理使用语义化标签**：使用 `<header>`、`<nav>`、`<main>`、`<article>`、`<section>`、`<footer>` 等语义化标签，有助于搜索引擎优化（SEO）和代码的可读性，同时浏览器能更高效地解析页面结构。

2. CSS 优化

- **压缩代码**：和 HTML 类似，去除 CSS 代码中的冗余空格、注释等，可借助工具如 cssnano 来实现。
- **减少内联样式**：内联样式会增加 HTML 文件的大小，并且不利于代码的维护和复用。尽量将样式写在 CSS 文件中。
- **避免使用 @import**：`@import` 会增加额外的 HTTP 请求，影响页面加载速度。建议使用 `<link>` 标签引入 CSS 文件。

3. JavaScript 优化

- **代码压缩和混淆**：使用工具如 UglifyJS 或 Terser 对 JavaScript 代码进行压缩和混淆，去除不必要的空格、注释，缩短变量名，减小文件体积。
- **减少全局变量**：全局变量会增加命名冲突的风险，并且会延长 JavaScript 引擎查找变量的时间。尽量使用局部变量和闭包。
- **防抖和节流**：在处理高频事件（如滚动、窗口缩放、输入框输入等）时，使用防抖（debounce）和节流（throttle）技术可以减少事件处理函数的执行次数，提高性能。

```
// 防抖函数示例
function debounce(func, delay) {
  let timer;
  return function() {
    const context = this;
    const args = arguments;
    clearTimeout(timer);
    timer = setTimeout(() => {
```

```
        func.apply(context, args);
    }, delay);
};
}

// 节流函数示例
function throttle(func, limit) {
    let inThrottle;
    return function() {
        const context = this;
        const args = arguments;
        if (!inThrottle) {
            func.apply(context, args);
            inThrottle = true;
            setTimeout(() => inThrottle = false, limit);
        }
    };
}
```

资源加载优化

1. 图片优化

- **压缩图片**：使用工具如 TinyPNG、ImageOptim 等对图片进行压缩，在不影响视觉效果的前提下减小图片文件大小。
- **选择合适的图片格式**：根据图片的特点选择合适的格式，如 JPEG 适合照片，PNG 适合图标和透明背景图片，WebP 具有更好的压缩比和质量，在支持的浏览器中优先使用。
- **图片懒加载**：对于页面中不在首屏或用户不可见区域的图片，使用懒加载技术，当图片进入可视区域时再进行加载，减少初始加载的资源量。可以使用 `loading="lazy"` 属性或第三方库如 `vue-lazyload` 实现。

2. 资源合并与分割

- **合并文件**：将多个 CSS 文件和 JavaScript 文件合并成一个，减少 HTTP 请求次数。但要注意合并文件的大小，避免文件过大影响加载速度。
- **代码分割**：使用 Webpack 等打包工具进行代码分割，将大的 JavaScript 文件分割成多个小的文件，实现按需加载，提高首屏加载速度。

3. 异步加载资源

- **CSS 文件异步加载**：对于一些非关键的 CSS 文件，可以使用 `media` 属性或

JavaScript 动态加载的方式实现异步加载，避免阻塞页面渲染。

```
<link rel="stylesheet" href="styles.css" media="print"
onload="this.media='all'">
```

- **JavaScript 文件异步加载**：使用 `<script>` 标签的 `async` 或 `defer` 属性实现 JavaScript 文件的异步加载。`async` 表示脚本在下载时不会阻塞页面渲染，下载完成后立即执行；`defer` 表示脚本在下载时不会阻塞页面渲染，页面解析完成后按顺序执行。

```
<script async src="script.js"></script>
<script defer src="script.js"></script>
```

缓存优化

1. 浏览器缓存

- **强缓存**：通过设置 HTTP 响应头的 `Cache-Control` 和 `Expires` 字段，让浏览器直接从本地缓存中读取资源，而不需要向服务器发送请求。

```
// Node.js 示例
const http = require('http');
const fs = require('fs');

http.createServer((req, res) => {
  fs.readFile('index.html', (err, data) => {
    res.setHeader('Cache-Control', 'max-age=3600'); // 缓存 1 小时
    res.end(data);
  });
}).listen(3000);
```

- **协商缓存**：当强缓存失效时，浏览器会向服务器发送请求，通过设置 HTTP 响应头的 `ETag` 和 `Last-Modified` 字段，服务器根据这些信息判断资源是否有更新，如果没有更新则返回 304 状态码，让浏览器使用本地缓存。

2. 应用内缓存

- **使用 `localStorage` 和 `sessionStorage`**：对于一些不经常变化的数据，如用户偏好设置、静态数据等，可以使用 `localStorage` 或 `sessionStorage` 进行缓存，减少对服务器的请求。

```
// 存储数据
localStorage.setItem('username', 'John');
// 获取数据
const username = localStorage.getItem('username');
```

- **使用 Service Worker**: Service Worker 可以在浏览器和服务器之间充当代理，拦截网络请求，实现离线缓存和资源预缓存，提高应用的响应速度和离线体验。

渲染优化

1. 减少重排和重绘

- **批量修改 DOM**: 尽量减少对 DOM 的频繁操作，将多次修改合并成一次。例如，先创建一个文档片段 (DocumentFragment)，在文档片段中进行 DOM 操作，最后将文档片段添加到页面中。

```
const fragment = document.createDocumentFragment();
const list = document.createElement('ul');
for (let i = 0; i < 10; i++) {
  const item = document.createElement('li');
  item.textContent = `Item ${i}`;
  list.appendChild(item);
}
fragment.appendChild(list);
document.body.appendChild(fragment);
```

- **避免强制同步布局**: 避免在修改 DOM 样式后立即读取布局信息，因为这会导致浏览器强制进行重排。

2. 优化 CSS 动画

- **使用 transform 和 opacity**: transform 和 opacity 属性的动画性能较好，因为它们不会触发重排和重绘，而是在合成层进行处理。

```
.box {
  transition: transform 0.3s ease;
}
.box:hover {
  transform: scale(1.1);
}
```

服务器端优化

1. CDN 加速

使用内容分发网络（CDN）来分发静态资源，如 CSS、JavaScript、图片等。CDN 节点分布在全球各地，可以将资源缓存到离用户最近的节点，减少用户的访问延迟。

2. 服务器配置优化

- **压缩响应内容：**在服务器端开启 Gzip 或 Brotli 压缩，对 HTML、CSS、JavaScript 等文件进行压缩，减小响应内容的大小。
- **优化服务器性能：**选择合适的服务器软件（如 Nginx、Apache），合理配置服务器参数，提高服务器的并发处理能力和响应速度。

29. Vuex浏览器刷新数据丢失的解决方法

在 Vuex 中，Vuex浏览器刷新数据丢失是由于其数据存储在内存中，而浏览器刷新会销毁内存中的实例。这主要和 Vuex 的数据存储机制以及浏览器刷新的特性有关，下面为你详细分析：

1. Vuex 数据存储在内存中

- **原理：**Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。Vuex 的数据是存储在 JavaScript 对象中，而这个对象是保存在浏览器的内存里。
- **示例：**

```
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment(state) {
      state.count++;
    }
  }
});
```

```
}  
});
```

在上述代码中，`state` 里的 `count` 数据存储在内存中。

2. 浏览器刷新的影响

- **原理：**当用户刷新浏览器时，浏览器会重新加载页面。这个过程会销毁当前页面的所有 JavaScript 实例，包括 Vue 实例和 Vuex 实例。由于 Vuex 的数据存储在内存中的 JavaScript 对象里，随着实例的销毁，内存中的数据也会被清除。
- **解释：**刷新浏览器相当于重新启动了整个前端应用程序，所有之前在内存中保存的临时数据（如 Vuex 中的状态）都会丢失，然后重新初始化应用，Vuex 也会重新创建一个初始状态的实例。

解决方案

虽然 Vuex 在浏览器刷新时数据会丢失，但可以通过一些方法来解决这个问题：

1. 使用本地存储（LocalStorage 或 SessionStorage）

- **原理：**本地存储可以将数据存储在浏览器中，即使页面刷新，数据依然存在。可以在 Vuex 的 `mutations` 或者 `actions` 中监听状态的变化，将数据存储在本地存储中；在应用初始化时，从本地存储中读取数据并恢复到 Vuex 的状态中。
- **示例：**

```
import Vue from 'vue';  
import Vuex from 'vuex';  
  
Vue.use(Vuex);  
  
const store = new Vuex.Store({  
  state: {  
    count: 0  
  },  
  mutations: {  
    increment(state) {  
      state.count++;  
      localStorage.setItem('vuexState', JSON.stringify(state));  
    }  
  }  
});  
  
// 初始化时从本地存储恢复数据
```



```
const savedState = localStorage.getItem('vuexState');
if (savedState) {
  store.replaceState(JSON.parse(savedState));
}

export default store;
```

2. 使用 Vuex 插件（如 vuex-persistedstate）

- **原理：** `vuex-persistedstate` 是一个专门用于解决 Vuex 数据持久化问题的插件。它可以自动将 Vuex 的状态存储到本地存储、会话存储或者其他存储引擎中，并在应用初始化时自动恢复状态。
- **示例：**

```
npm install vuex-persistedstate
```

```
import Vue from 'vue';
import Vuex from 'vuex';
import createPersistedState from 'vuex-persistedstate';

Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment(state) {
      state.count++;
    }
  },
  plugins: [createPersistedState()]
});
```

通过以上分析可知，Vuex 浏览器刷新数据丢失是由于其数据存储在内存中，而浏览器刷新会销毁内存中的实例，不过可以采用本地存储或相关插件来解决数据持久化问题。

30. 前端开发 中常见问题及对应的解决方案

以下为你整合了前端开发中常见问题及对应的解决方案，涵盖性能、兼容性、交互、安全等多个方面，你可以根据实际情况进行灵活调整和运用：

1. 性能优化相关问题

问题描述：页面加载速度慢

- **具体表现**：在访问页面时，需要等待较长时间才能看到完整内容，尤其是在网络状况不佳或页面资源较多的情况下，这种延迟更为明显。
- **可能原因**：页面中包含大量未压缩的图片、CSS 和 JavaScript 文件；过多的 HTTP 请求；没有合理利用缓存机制。
- **解决方案**
 - **压缩资源**：使用工具对图片、CSS 和 JavaScript 文件进行压缩。例如，使用 TinyPNG 压缩图片，使用 UglifyJS 压缩 JavaScript，使用 cssnano 压缩 CSS。
 - **合并文件**：将多个 CSS 和 JavaScript 文件合并成一个，减少 HTTP 请求次数。同时，使用代码分割技术，按需加载非关键资源。
 - **缓存机制**：设置合理的缓存策略，利用浏览器的强缓存和协商缓存。对于静态资源，可以设置较长的缓存时间；对于动态数据，可以使用 ETag 和 Last - Modified 进行协商缓存。

问题描述：动画卡顿

- **具体表现**：页面中的动画效果不流畅，出现明显的卡顿、掉帧现象，影响用户体验。
- **可能原因**：动画使用了会触发重排和重绘的 CSS 属性，如 width 、 height 等；JavaScript 代码执行时间过长，阻塞了页面渲染。
- **解决方案**
 - **使用 transform 和 opacity**：优先使用 transform 和 opacity 进行动画效果的实现，因为它们不会触发重排和重绘，性能更好。
 - **优化 JavaScript 代码**：避免在动画过程中进行大量的计算和 DOM 操作。可以使用 requestAnimationFrame 来优化动画的执行，确保动画在浏览器的渲染帧中进行。

2. 兼容性问题

问题描述：不同浏览器显示不一致

- **具体表现**：页面在不同浏览器（如 Chrome、Firefox、Safari 等）或同一浏览器的不同版本中显示效果存在差异，布局错乱、样式缺失等。
- **可能原因**：不同浏览器对 CSS 和 HTML 标准的支持程度不同，一些新的 CSS 属性

和 JavaScript 特性在旧版本浏览器中可能不被支持。

- **解决方案**

- **使用浏览器前缀**：对于一些新的 CSS 属性，使用浏览器前缀（如 `-webkit`、`-moz`、`-ms` 等）来确保在不同浏览器中都能正常显示。
- **使用 Polyfill**：对于一些新的 JavaScript 特性，使用 Polyfill 库（如 Babel Polyfill、core-js 等）来在旧版本浏览器中提供兼容支持。
- **进行多浏览器测试**：在开发过程中，使用 BrowserStack、Sauce Labs 等工具对不同浏览器和版本进行测试，及时发现并解决兼容性问题。

问题描述：移动端和桌面端显示差异大

- **具体表现**：页面在移动端和桌面端的布局、交互效果等方面存在较大差异，影响用户在不同设备上的使用体验。
- **可能原因**：没有进行响应式设计，页面布局和样式没有根据设备屏幕尺寸进行调整；一些桌面端的交互方式（如鼠标悬停）在移动端无法正常使用。
- **解决方案**
 - **响应式设计**：使用媒体查询、弹性布局（Flexbox）和网格布局（Grid）等技术进行响应式设计，确保页面在不同设备上都能有良好的显示效果。
 - **触摸事件处理**：在移动端，使用触摸事件（如 `touchstart`、`touchmove`、`touchend` 等）替代鼠标事件，提供更符合移动端操作习惯的交互方式。

3. 交互体验问题

问题描述：表单提交后无反馈

- **具体表现**：用户提交表单后，页面没有任何反馈信息，用户不知道表单是否提交成功，容易造成重复提交。
- **可能原因**：没有在表单提交时添加相应的提示信息或状态更新；表单提交过程中出现错误，但没有给用户明确的错误提示。
- **解决方案**
 - **添加加载提示**：在表单提交时，显示加载动画或提示信息，告知用户表单正在处理中。例如，使用模态框显示“正在提交，请稍候...”。
 - **反馈提交结果**：根据表单提交的结果，给用户相应的反馈信息。如果提交成功，显示成功提示；如果提交失败，显示具体的错误信息，帮助用户排查问题。

问题描述：滚动加载性能差

- **具体表现：**在实现滚动加载数据时，页面滚动不流畅，出现卡顿现象，尤其是在加载大量数据时更为明显。
- **可能原因：**每次滚动加载时，一次性加载过多的数据，导致 DOM 操作频繁，影响页面性能；没有对滚动事件进行节流处理，导致事件处理函数频繁触发。
- **解决方案**
 - **分页加载：**采用分页加载的方式，每次只加载适量的数据，减少 DOM 操作的负担。可以根据用户的滚动位置，动态加载下一页的数据。
 - **节流处理：**使用节流函数对滚动事件进行处理，减少事件处理函数的执行次数。例如，使用 `throttle` 函数，设置一个固定的时间间隔，在该时间间隔内只执行一次事件处理函数。

4. 安全问题

问题描述：XSS 攻击

- **具体表现：**攻击者通过在网页中注入恶意脚本，当用户访问该页面时，恶意脚本会在用户的浏览器中执行，从而获取用户的敏感信息或进行其他恶意操作。
- **可能原因：**在渲染用户输入的内容时，没有进行适当的过滤和转义，导致恶意脚本被直接执行。
- **解决方案**
 - **输入验证和过滤：**在前端和后端对用户输入的内容进行验证和过滤，只允许合法的字符和格式。例如，使用正则表达式对输入内容进行验证。
 - **HTML 转义：**在渲染用户输入的内容时，对特殊字符进行 HTML 转义，防止恶意脚本的执行。可以使用库如 DOMPurify 来进行 HTML 净化。

问题描述：CSRF 攻击

- **具体表现：**攻击者通过诱导用户在已登录的网站上执行恶意操作，利用用户的身份信息完成非法请求。
- **可能原因：**网站没有对请求进行有效的身份验证和令牌验证，导致攻击者可以伪造请求。
- **解决方案**
 - **使用 CSRF 令牌：**在表单提交或 AJAX 请求中添加 CSRF 令牌，服务器在处理请求时验证令牌的有效性。可以在表单中隐藏一个包含 CSRF 令牌的字段，或者在请求头中添加令牌信息。
 - **同源策略：**确保所有的请求都来自同源的页面，防止跨站请求伪造。可以通过

设置 CSP (Content Security Policy) 来限制页面可以加载的资源 and 执行的脚本来源。

31. 项目亮点

在我过往的项目经历中，有两个移动端相关项目十分亮眼，充分展现了我的技术能力与项目管理水平。

移动端项目：从 Vue 2 到 Vue 3 的升级主导

我曾从 0 到 1 主导了一个移动端项目的开发，该项目原本基于 Vue 2 构建，我决定将其升级为 Vue 3 进行开发。

- **技术创新与高效应用：**引入 Vue 3 及其组合式 API，这一转变让代码结构更加清晰模块化，逻辑复用变得轻而易举。比如，在处理复杂的表单验证逻辑时，通过组合式 API 将验证逻辑封装成独立的函数，在多个组件中复用，大大提高了开发效率。同时，搭配 TypeScript，利用其静态类型检查特性，在开发阶段就捕获并解决了许多潜在的代码错误，显著提升了代码的稳定性和可维护性。
- **性能优化成效显著：**针对移动设备的性能特点，我运用 Webpack 进行代码分割和压缩，减少了初始加载的代码量。采用图片懒加载和 CDN 加速技术，大幅加快了图片资源的加载速度。经过一系列优化，页面首屏加载时间缩短了 40%，有效提升了用户体验。
- **团队协作与高效开发：**在项目中采用敏捷开发流程，每周进行迭代和回顾。组织团队成员进行 Vue 3 技术培训，帮助大家快速掌握新技术。通过 Git 进行版本控制，使用项目管理工具进行任务跟踪和进度监控，确保团队成员之间沟通顺畅、协作高效，项目按时高质量交付。

App 内 H5 活动页面项目：动画驱动流量增长

在 App 内 H5 活动页面项目中，我聚焦于打造出色的动画效果，以吸引用户参与，最终实现了 PV 和 UV 的显著增长。这是团队协作的结果，ui老师的设计图，产品老师的点子，还有大家一起想出来的动画效果等；

- **数据驱动活动优化：**在活动上线后，密切关注数据指标，通过埋点统计 PV、UV、用户停留时间等数据。根据数据分析结果，对活动页面进行针对性优化，如调整动画展示位置、优化活动规则说明等。经过优化，活动的 PV 增长了 60%，UV 增长了 50%，取得了显著的营销效果。

这两个项目不仅锻炼了我的技术能力和项目管理能力，也让我深刻体会到技术创新和用

用户体验对于项目成功的重要性。我相信这些经验将对未来的工作产生积极的影响。

32.项目中常见技术难题，如何解决

在项目开发过程中，会遇到各种各样的技术难题，下面从不同方面为你介绍常见的技术难题：

兼容性难题

1. 浏览器兼容性

- **表现：**不同浏览器（如 Chrome、Firefox、Safari、IE 等）对 HTML、CSS 和 JavaScript 的支持存在差异，导致页面在某些浏览器中显示错乱、功能异常。例如，CSS 的一些新特性（如 Flexbox、Grid 布局）在旧版本浏览器中不被支持；JavaScript 的新 API（如 `fetch`）在部分浏览器中需要使用 Polyfill 才能正常工作。
- **解决办法：**使用浏览器前缀来兼容不同浏览器，例如 `-webkit-` 用于 WebKit 内核浏览器（如 Safari、Chrome），`-moz-` 用于 Firefox。同时，使用 Can I Use 网站查询 CSS 属性和 JavaScript API 的浏览器兼容性，对于不兼容的情况，使用 Polyfill 库（如 Babel Polyfill、whatwg-fetch）来提供支持。在开发过程中，使用 BrowserStack、Sauce Labs 等工具对不同浏览器和版本进行测试。

2. 设备兼容性

- **表现：**不同移动设备（如 iPhone、Android 手机、平板电脑）的屏幕尺寸、分辨率、操作系统版本不同，会导致页面布局和交互出现问题。例如，在大屏幕设备上，页面元素可能显示过大或过小；在低版本 Android 系统上，某些 CSS 动画效果可能无法正常显示。
- **解决办法：**采用响应式设计，使用媒体查询、弹性布局（Flexbox）和网格布局（Grid）来实现页面元素的自适应调整。对于不同设备的特性，可以使用 JavaScript 进行检测并采取相应的处理措施。同时，进行多设备测试，确保页面在各种设备上都能正常显示和使用。

性能难题

1. 页面加载性能

- **表现：**页面加载速度慢，尤其是在网络状况不佳或页面资源较多的情况下，用户需

要等待很长时间才能看到完整的页面内容。这可能是由于图片、CSS、JavaScript 文件过大，HTTP 请求过多，服务器响应慢等原因导致的。

- **解决办法：**对图片进行压缩和优化，选择合适的图片格式（如 WebP）；合并和压缩 CSS、JavaScript 文件，减少 HTTP 请求次数；使用 CDN 加速静态资源的加载；优化服务器配置，提高服务器的响应速度；采用懒加载技术，只在用户需要时加载资源。

2. 内存泄漏

- **表现：**在 JavaScript 应用中，内存泄漏会导致页面占用的内存不断增加，最终可能导致浏览器崩溃或页面性能严重下降。常见的内存泄漏原因包括未释放的定时器、未移除的事件监听器、闭包引用等。
- **解决办法：**在不需要定时器时，使用 `clearTimeout` 或 `clearInterval` 清除定时器；在元素移除时，使用 `removeEventListener` 移除事件监听器；避免不必要的闭包引用，及时释放不再使用的对象和变量。可以使用浏览器的开发者工具（如 Chrome DevTools 的 Memory 面板）来检测和分析内存泄漏问题。

安全难题

1. XSS（跨站脚本攻击）

- **表现：**攻击者通过在网页中注入恶意脚本，当用户访问该页面时，恶意脚本会在用户的浏览器中执行，从而获取用户的敏感信息（如 cookie、会话令牌）或进行其他恶意操作。常见的注入方式包括在表单输入、URL 参数中注入脚本代码。
- **解决办法：**对用户输入进行严格的验证和过滤，只允许合法的字符和格式。在渲染用户输入的内容时，对特殊字符进行 HTML 转义，防止恶意脚本的执行。可以使用 DOMPurify 等库来进行 HTML 净化。同时，设置 CSP（Content Security Policy），限制页面可以加载的资源 and 执行的脚本来源。

2. CSRF（跨站请求伪造）

- **表现：**攻击者通过诱导用户在已登录的网站上执行恶意操作，利用用户的身份信息完成非法请求。例如，攻击者可以构造一个恶意链接，当用户在已登录的网站上点击该链接时，会自动向该网站发送一个伪造的请求。
- **解决办法：**在表单提交或 AJAX 请求中添加 CSRF 令牌，服务器在处理请求时验证令牌的有效性。可以在表单中隐藏一个包含 CSRF 令牌的字段，或者在请求头中添加令牌信息。同时，设置 SameSite 属性，限制 cookie 的跨站发送。

数据交互难题

1. 跨域问题

- **表现：**由于浏览器的同源策略，当页面从一个域名下的页面向另一个域名下的服务器发送请求时，会受到跨域限制，导致请求被阻止。常见的跨域场景包括前后端分离项目中前端页面和后端 API 不在同一个域名下。
- **解决办法：**可以使用 CORS（跨域资源共享），在服务器端设置响应头，允许指定的域名进行跨域请求。也可以使用 JSONP（JSON with Padding），它是一种古老的跨域解决方案，但只支持 GET 请求。另外，还可以使用代理服务器，在同源的服务器上转发请求到目标服务器。

2. 数据格式和编码问题

- **表现：**在前后端数据交互过程中，可能会出现数据格式不匹配、编码不一致的问题。例如，前端发送的数据格式为 JSON，但后端期望的是 XML 格式；或者在传输过程中出现字符编码问题，导致数据显示乱码。
- **解决办法：**前后端约定统一的数据格式，通常使用 JSON 作为数据交换格式。在传输数据时，确保使用统一的字符编码（如 UTF - 8）。在前端发送请求和后端接收请求时，进行必要的数据格式转换和编码处理。

33. keep-alive

`keep-alive` 是 Vue.js 提供的一个内置组件，用于缓存动态组件或路由组件，避免重复渲染，从而提高应用的性能和响应速度。下面我将从基本概念、使用场景、使用方式、生命周期钩子、原理以及注意事项几个方面详细介绍。

基本概念

`keep-alive` 是一个抽象组件，它自身不会渲染一个 DOM 元素，也不会出现在父组件链中。它的主要作用是包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们。

使用场景

- **频繁切换的组件：**在一些需要频繁切换显示的组件场景中，比如 Tab 切换组件，使用 `keep-alive` 可以避免每次切换时都重新创建和销毁组件实例，提高切换的流畅度。
- **需要保留状态的组件：**当组件中有一些用户输入或滚动位置等状态，在组件隐藏后再次显示时需要保留这些状态，使用 `keep-alive` 可以很好地满足需求。例如，在一个电商应用中，商品列表页面滚动到一定位置后切换到商品详情页，再返回商

品列表页时能保持之前的滚动位置。

使用方式

包裹动态组件

```
<template>
  <div>
    <button @click="toggleComponent">切换组件</button>
    <!-- 使用 keep-alive 包裹动态组件 -->
    <keep-alive>
      <component :is="currentComponent"></component>
    </keep-alive>
  </div>
</template>

<script>
import ComponentA from './ComponentA.vue';
import ComponentB from './ComponentB.vue';

export default {
  components: {
    ComponentA,
    ComponentB
  },
  data() {
    return {
      currentComponent: 'ComponentA'
    };
  },
  methods: {
    toggleComponent() {
      this.currentComponent = this.currentComponent === 'ComponentA' ?
'ComponentB' : 'ComponentA';
    }
  }
};
</script>
```

结合路由使用

```
<template>
  <div id="app">
    <!-- 使用 keep-alive 包裹路由视图 -->
```

```
<keep-alive>
  <router-view></router-view>
</keep-alive>
</div>
</template>

<script>
export default {
  name: 'App'
};
</script>
```

生命周期钩子

当组件被 `keep-alive` 缓存时，其生命周期钩子会发生变化，除了常规的 `created`、`mounted` 等钩子外，还会新增两个钩子：

- **activated**：被缓存的组件激活时调用。
- **deactivated**：被缓存的组件停用时调用。

```
<template>
  <div>
    <!-- 组件内容 -->
  </div>
</template>

<script>
export default {
  activated() {
    console.log('组件被激活');
  },
  deactivated() {
    console.log('组件停用');
  }
};
</script>
```

原理

`keep-alive` 内部通过一个 `cache` 对象来存储缓存的组件实例，使用组件的 `key` 作为键。当组件被切换时，会先判断该组件是否已经存在于 `cache` 中，如果存在则直接从缓存中获取并渲染，而不是重新创建组件实例；如果不存在，则创建新的组件实例并添加到 `cache` 中。

注意事项

- **include 和 exclude 属性**：可以使用 `include` 和 `exclude` 属性来指定哪些组件需要被缓存或排除缓存。`include` 表示只有匹配的组件会被缓存，`exclude` 表示匹配的组件不会被缓存。例如：

```
<keep-alive include="ComponentA,ComponentB">
  <component :is="currentComponent"></component>
</keep-alive>
```

- **max 属性**：可以使用 `max` 属性来限制缓存组件的最大数量。当缓存的组件数量超过 `max` 时，会按照 LRU（最近最少使用）策略淘汰最久未使用的组件。例如：

```
<keep-alive max="3">
  <component :is="currentComponent"></component>
</keep-alive>
```

综上所述，`keep-alive` 是 Vue.js 中一个非常实用的组件，合理使用它可以显著提升应用的性能和用户体验。

34.前端安全

前端安全是 Web 开发中至关重要的一环，关乎用户信息安全和网站的正常运行。常见的前端安全问题主要包括跨站脚本攻击（XSS）、跨站请求伪造（CSRF）、点击劫持、SQL 注入（虽更多关联后端，但前端输入验证也很关键）以及不安全的第三方库引入等，以下为你详细介绍：

跨站脚本攻击（XSS）

原理

攻击者通过在目标网站注入恶意脚本，当用户访问该网站时，恶意脚本会在用户的浏览器中执行，从而获取用户的敏感信息，如 Cookie、会话令牌等，或者进行其他恶意操作，像修改页面内容、重定向到恶意网站等。

类型

- **反射型 XSS**：恶意脚本作为参数被包含在 URL 中，当用户访问包含恶意脚本的 URL 时，服务器会将该脚本反射到响应页面中并执行。

- **存储型 XSS**：攻击者将恶意脚本存储在目标网站的数据库中，当其他用户访问包含该恶意脚本的页面时，脚本会被执行。
- **DOM - based XSS**：基于 DOM 操作的 XSS 攻击，通过修改页面的 DOM 结构来注入恶意脚本，不需要与服务器进行交互。

防范措施

- **输入验证和过滤**：在前端和后端都对用户输入进行严格验证和过滤，只允许合法的字符和格式。可以使用正则表达式对输入内容进行检查。
- **输出编码**：在渲染用户输入的内容时，对特殊字符进行 HTML 转义，将其转换为 HTML 实体，防止恶意脚本的执行。例如，将 `<` 转换为 `<`，`>` 转换为 `>`。可以使用库如 DOMPurify 来进行 HTML 净化。
- **设置 CSP（内容安全策略）**：通过设置 CSP 响应头，限制页面可以加载的资源 and 执行的脚本来源，只允许从指定的域名加载脚本，从而防止恶意脚本的注入。

跨站请求伪造（CSRF）

原理

攻击者诱导用户在已登录的网站上执行恶意操作，利用用户的身份信息完成非法请求。攻击者通常会构造一个恶意链接或表单，当用户在已登录的网站上点击该链接或提交表单时，会自动向该网站发送一个伪造的请求。

防范措施

- **使用 CSRF 令牌**：在表单提交或 AJAX 请求中添加 CSRF 令牌，服务器在处理请求时验证令牌的有效性。可以在表单中隐藏一个包含 CSRF 令牌的字段，或者在请求头中添加令牌信息。
- **SameSite 属性**：设置 Cookie 的 `SameSite` 属性，限制 Cookie 的跨站发送。`SameSite` 属性有三个值：`Strict`、`Lax` 和 `None`。`Strict` 表示 Cookie 只能在同站请求中发送，`Lax` 表示在一些安全的跨站请求中可以发送，`None` 表示允许跨站发送，但需要同时设置 `Secure` 属性。
- **验证请求来源**：服务器端验证请求的来源，检查请求的 `Referer` 或 `Origin` 头信息，确保请求来自合法的域名。

点击劫持

原理

攻击者通过创建一个透明的、不可见的 `iframe`，将目标网站嵌入其中，并在其上覆盖一个诱导用户点击的按钮或链接。当用户点击表面的按钮或链接时，实际上点击的是 `iframe` 中的目标网站的某个操作，从而在用户不知情的情况下完成恶意操作。

防范措施

- **设置 HTTP 响应头**：服务器端设置 `X - Frame - Options` 响应头，限制页面是否可以被其他网站通过 `iframe` 嵌入。该响应头有三个值：`DENY` 表示页面不能被任何网站嵌入，`SAMEORIGIN` 表示页面只能被同域名的网站嵌入，`ALLOW - FROM uri` 表示页面只能被指定的域名嵌入。
- **使用 JavaScript 检测**：在页面中使用 JavaScript 检测页面是否被嵌入到 `iframe` 中，如果是则进行相应的处理，如跳转到正常页面或显示警告信息。

SQL 注入（前端输入验证方面）

原理

攻击者通过在表单输入或 URL 参数中注入恶意的 SQL 语句，试图绕过应用程序的验证机制，从而执行非法的数据库操作，如获取、修改或删除数据库中的数据。

防范措施

- **前端输入验证**：在前端对用户输入进行初步验证，限制输入的长度和格式，防止用户输入恶意的 SQL 语句。但前端验证不能作为唯一的安全措施，因为可以被绕过，主要起到提示用户的作用。
- **后端严格处理**：后端使用参数化查询或预编译语句来处理用户输入，避免将用户输入直接拼接到 SQL 语句中，从而防止 SQL 注入攻击。

不安全的第三方库引入

原理

在项目中引入不安全的第三方库，这些库可能存在安全漏洞，攻击者可以利用这些漏洞对项目进行攻击，获取用户信息或破坏系统。

防范措施

- **选择可靠的第三方库**：从官方渠道或知名的开源社区获取第三方库，选择维护活跃、口碑良好的库。
- **及时更新库版本**：关注第三方库的更新信息，及时更新到最新版本，以修复已知的

安全漏洞。

- **安全审查：**在引入新的第三方库之前，对其进行安全审查，检查是否存在已知的安全问题。

综上所述，前端安全需要从多个方面进行防范，通过合理运用各种安全技术和措施，可以有效降低前端应用面临的安全风险。



Syntax error in text
mermaid version 10.2.4