# 自我介绍

我叫石本仕，我是16年毕业于武昌工学院通信工程专业，最近的一份工作呢是在中软国际入职时间呢是7月30，中软国际呢是一家外包公司，接的呢是华为端的项目，最近做的几个项目有精准上门服务坐席端，精准上门服务管理端，移动端故障图像上传，三者是同步开发的，精准上门服务呢是用于华为热线客服辅助客户解决设备故障，管理端呢主要是对坐席端页面实现相关内容的动态配置，移动端故障图像上传则是用于客户端上传故障内容，主要用到的技术点呢包括websocket实现双向通信，sdk实现屏幕共享，提示，涂鸦，图片的拖拽 放大 缩小 旋转 删除等，rem适配不同移动端屏幕大小

# vue

## 对MVVM的

MVVM为前端端视图分离式开发概念

M为数据层，V为视图层，VM为控制器，将数据层和视图层建立关系，VM实际上就是Vue实例，数据层和视图层之间的交互式双向的，数据层发生改变视图层跟着改变，视图层发生改变数据层也会跟着改变，而View 和 Model 之间的同步工作完全是自动的，无需人为干涉

## Vue生命周期

1. beforeCreate:

在实例初始化之后调用，此时vue对象被创建了，但是vue对象的属性（data属性，computed属性）还没有绑定，即没有值，此时还没有数据和真实的dom

1. created()

vue对象的属性有值，但是DOM还没有生成，$el属性不存在

如果数据的初始值来自后端此时可发送 ajax请求

1. beforeMount()

模板编译，数据挂载之前执行

this $el 有值，但数据还没有挂载到页面上，即此时页面中的 {{ }} 里的变量还没有被数据替换

1. Mount()

此时数据挂载到了页面上

一般我们在此处发送异步请求（ajax， axios）获取服务器数据，显示在DOM

1. beforeUpdate()

模板上出现的数据更新后触发该函数，组件更新前

1. update ()

组件更新之后执行

1. beforeDestroy()

vue 组件销毁之前执行

1. update ()

vue组件销毁后执行

## vue基本原理

当一个vue实例创建时，vue会遍历data中的属性，用Object.defineProperty(vue3.0使用proxy)将它转为getter/setter,并且在内部追踪相关依赖，在属性被访问和修改时通知变化。每一个组件实例都有相应的watcher程序实例，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的setter被调用时，会通知watcher重新计算，从而致使组件得以更新。

## Vue2.0双向数据绑定的原理与缺陷

VUE实现双向数据绑定的原理主要是采用数据劫持结合发布者订阅者模式通过object对象当中的defineProperty（）来劫持各个属性当中的setter getter，在数据变动时发布消息给订阅者触发相应的监听回调，vue2.0版本使用的是defineProperty，vue3.0使用的是ES6的语法 Proxy对象来实现的

加分回答 Object.defineProperty的缺点

1. 一次性递归到底开销很大，如果数据很大，大量的递归导致调用栈溢出

2. 不能监听对象的新增属性和删除属性

3. 无法正确的监听数组的方法，当监听的下标对应的数据发生改变时

4.双向数据绑定原理代码：

<body>

    <div id="app">

    <input type="text" id="txt">

    <p id="show"></p>

</div>

</body>

<script type="text/javascript">

    var obj = {}

    Object.defineProperty(obj, 'txt', {

        get: function () {

            return obj

        },

        set: function (newValue) {

            document.getElementById('txt').value = newValue

            document.getElementById('show').innerHTML = newValue

        }

    })

    document.addEventListener('keyup', function (e) {

        obj.txt = e.target.value

    })

</script>

## Vue3.0双向数据绑定的原理与缺陷

Vue3.0 是通过Proxy实现的数据双向绑定，Proxy是ES6中新增的一个特性，实现的过程是在目标对象之前设置了一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。

用法： ES6 原生提供 Proxy 构造函数，用来生成 Proxy 实例。 var proxy = new Proxy(target, handler)

target: 是用Proxy包装的被代理对象（可以是任何类型的对象，包括原生数组，函数，甚至另一个代理）。

handler: 是一个对象，其声明了代理target 的一些操作，其属性是当执行一个操作时定义代理的行为的函数。

加分回答 `Object.defineProperty` 的问题在于无法监控数组下标的变化，通过数组下标给数组设置值时，不能实时响应。Object.defineProperty只能劫持对象的属性,我们需要对每个对象的每个属性进行遍历。Vue是通过递归以及遍历data对象来实现对数据的监控的，如果属性值也是对象那么需要深度遍历

Proxy的两个优点：可以劫持整个对象，并返回一个新对象，有13种劫持

能劫持一个完整的对象，对操作性和性能有很大的提升。

--------------------------------------------------------

（<https://juejin.cn/post/6844903545599164423>）

简单来说:Proxy对象就是可以让你去对JavaScript中的一切合法对象的基本操作进行自定义.然后用你自定义的操作去覆盖其对象的基本操作.也就是当一个对象去执行一个基本操作时,其执行的过程和结果是你自定义的,而不是对象的.

首先Proxy的语法是: let p = new Proxy(target, handler);

Proxy的作用

1. 拦截和监视外部对对象的访问
2. 降低函数或类的复杂度
3. 在复杂操作前对操作进行校验或对所需资源进行管理

使用场景（<https://www.w3cplus.com/javascript/use-cases-for-es6-proxies.html>）

## Vue组件之间的通信

1. **父组件向子组件传值**

父组件通过属性将值传递给子组件，

子组件通过props来接收父组件传递过来的值

1. **子组件向父组件传值**

子组件通过$emit自定义事件将值传递给父组件，

父组件通过监听子组件的事件，并通过$event拿到子组件传递过来的值

1. **兄弟组件之间的传值（$emit/$on/$off）**

通过一个空的vue实例作为事件中心，用它来触发和监听事件，实现兄弟/父子，跨级组件之间的通信，建议慎用会出现难以维护，项目比较大时建议选择vuex进行状态管理

用法：首先新建一个js,js做相关处理，将相关处理放入到vue的原型中，然后在入口函数中导入js,通过vue.use进行全局注册，在A组件的周期函数created中注册方法this.$bus.on(‘add’, 在A组件中定义的方法名)，在beforeDestroy中注销方法this.$bus.off(‘add’, 在A组件中定义的方法名),B组件中通过this.$bus.emit(‘方法名’，参数)

加分回答 EventBus的优缺点，

缺点：

1. 由于vue是单页应用，在某一个页面刷新了之后，与之相关的EventBus会被移除，将导致业务无法走下去。
2. 如果页面中有反复操作的业务，EventBus会触发很多次，
3. 在vue页面销毁时，同时移除EventBus事件监听。

优点，解决了多层组件之间繁琐的事件传播，代码量少。适合业简单，组件传递数据较少的项目，大型项目业务复杂的还是尽量使用VueX

## v-if和v-show的异同点

相同点：两者都可控制指定元素的显示与隐藏

不同点：v-if需要不断的在dom树中创建和删除节点，v-show则是直接通过修改display属性当一个元素会被频繁的显示和隐藏时，使用v-show。

## computed和watch的区别

1. **computed**:支持缓存，不支持异步，计算属性的值默认走缓存，如果计算属性的属性值是函数，那么默认使用get方法，函数的返回值就是属性的属性值，在计算属性中有一个get方法和set方法，当数据发生变化时，会调用set方法

2. watch: 不支持缓存，支持异步，监听data中的数据或者props中的数据

Computed和watch在哪个周期时被触发

1. 页面首次加载时，computed会执行一次，并且是在beforeMount之后，mounted之前
2. 在页面数据发生变化时

如果不是由点击事件造成的数据变化，执行顺序为：watch beforeUpdate computed updated

如果是由点击事件造成的数据变化，执行顺序为：methods watch beforeUpdate computed updated

加分回答

computed应用场景：需要进行数值计算，并且依赖于其它数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算；

watch应用场景：需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，watch 允许我们执行异步操作 ( 访问一个 API )，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

## Vue当中的data为什么是一个函数

为了保证组件的独立性 和 可 复用性，data 是一个函数，组件实例化的时候这个函数将

会被调用，返回一个对象，计算机会给这个对象分配一个内存地址，你实例化几次，就分

配几个内存地址，他们的地址都不一样，所以每个组件中的数据不会相互干扰，改变其中

一个组件的状态，其它组件不变。

## Vue 列表（v-for）为什么加 key？

为了性能优化,首先要清楚vue是虚拟DOM,更新DOM时将会对节点一一比对，必有有很多li元素，要在某个位置插入一个li，但没有加key，那么在运算时，将会把所有li元素重新渲染一遍，如果有key，那么就会按照key一一比对li元素，只需创建新的li，插入即可，不需要对其他元素进行修改和重新渲染。

加分回答 key也不能是index，因为如果给数组插元素，它的下标是0，那么和原来的第一个元素重复了，整个数组的key都发生了改变，这样就跟没有key的情况一样了

## Slot

Slot就是子组件的模板标签元素，它是否展示以及怎么展示由父组件决定

Slot分为三类：默认插槽，具名插槽，作用域插槽，默认插槽只能有一个，具名插槽作用域插槽可以有多个，作用域插槽可以将子组件内部的数据传递给父组件，父组件根据子组件传递过来的数据决定如何渲染该插槽

实现原理：

子组件实例化时，获取到父组件传入的slot标签的内容，存放在vm.$slot中，当组件执行渲染函数时，遇到slot标签，使用$slot中的内容进行替换，此时可以为插槽传递数据，若存放数据，则可称该插槽为作用域插槽。

## Vuex

什么是vuex:

Vuex是实现组件全局状态管理的一种机制，可以方便的实现组件之间的数据共享

优点：

1. 能够在vuex中集中管理共享数据，便于开发和后期维护
2. 能够高效的实现组件之间的数据共享提高开发效率
3. 存储在vuex中的数据是响应式的，当数据发生改变时，页面中的数据也会同步更新

特性：

1. State: 提供唯一公共数据源，所有共享的数据都要统一放到state当中
2. Mutation: 用于修改state中的数据
3. Action: 用于执行异步操作，mutation是不能用于异步操作的，会导致一些问题，如：会导致vue调试器的显示出错
4. 用于对state中的数据进行包装处理形成新的数据，不会修改state中的数据，当state中的数据发生改变，getter生成的内容也会随之改变

Vue模块化：  
由于使用单一状态树，应用的所有状态会集中到一个比较大对象。当应用变得非常复杂时，store对象就可能变得相当臃肿。

为了解决以上问题，vuex允许我们将store分割成模块（module）。每个模块拥有自己的state mutation action getter。

虽然我们在实例化vuex.store时创建了不同的模块，但是最终生成的store对象所有数据（包含子模块的数据）依然存储在同一个state mutation action getter中

命名冲突问题

Vuex中的modules支持命名空间，设置了命名空间的模块与没有设置命名空间的模块存在一定的区别

## Mixin

**（**[**https://juejin.cn/post/7076340796361801759**](https://juejin.cn/post/7076340796361801759)**）**

将组件的公共逻辑或者配置提取出来，哪个组件需要用到时，直接将提取的这部分混入到组件内部即可。这样既可以减少代码冗余度，也可以让后期维护起来更加容易。

混入和vuex的区别

1. Vuex公共状态管理，如果在一个组件中更改了Vuex中的某个数据，那么其它所有引用了Vuex中该数据的组件也会跟着变化。
2. Mixin中的数据和方法都是独立的，组件之间使用后是互相不影响的。

局部混入

局部混入和组件的按需加载有点类似，需要用到mixin中的代码时，在组件中导入

混入特点：

1. Mixin中的生命周期函数会和组件的生命周期函数一起合并执行
2. Mixin中的data数据在组件中也可以使用
3. Mixin中的方法在组件内部可以直接调用
4. 生命周期函数合并后执行顺序：先执行mixin中的，后执行组件的

问题提出：

一个组件中改动了mixin中的数据，另一个引用了mixin的组件会受影响么？

不会（不同组件中的mixin是互相独立的）

全局混入

在入口函数main.js中导入，import { mixins } from ‘./mixin/index.js’

通过vue.minxin(mixins),进行注册

不推荐使用全局混入，因为它会影响每一个单独创建的vue实例（包括第三方组件）

Mixin中定义的属性或方法的名称与组件中定义的名称冲突（冲突分几种情况）

1. 生命周期函数

确切的说这种不算冲突，因为生命周期函数的名称都是固定的，默认的合并策略如下：

先执行mixin中生命周期函数中的代码，然后再执行组件内部的代码

1. Data数据冲突

组件中的data数据会覆盖mixin中数据，没有冲突的数据将合并

1. 方法冲突

方法名冲突很常见，加载规则优先调用组件中的方法

我们可以自定义合并规则，但是没必要做那么复杂

Mixin的优缺点

优点：

1. 提高代码复用性
2. 无需传递状态
3. 维护方便，只需要修改一个地方即可

缺点：

1. 命名冲突
2. 滥用的话后期很难维护
3. 不好追溯源，排查问题稍显麻烦
4. 不能轻易的重复代码

## Keep-alive

**（**[**https://juejin.cn/post/6844904014178418702**](https://juejin.cn/post/6844904014178418702)**）**

缓存组件，可以让其包裹的组件不销毁，起到缓存的作用

接收参数

1. Max 缓存的最大组件数量
2. Include 需要缓存的组件 名称匹配的组件会被缓存 支持字符串 数组 正则
3. Exclude 不需要缓存的组件 名称匹配的组件都不会被缓存 支持字符串 数组 正则

Keep-alive的两个周期函数（当组件在 <keep-alive> 内被切换，周期函数将会被触发）

1. Activated ：第一次进入缓存路由/组件，在mounted后面，beforeRouteEnter守卫传给 next的回调函数之前调用
2. Deactivated：处理用户将要离开的操作 使用了keep-alive就不会调用beforeDestroy(组件销毁前钩子)和destroyed(组件销毁)，因为组件没被销毁，被缓存起来了。

使用场景：

在用户从首页跳转到分类页面，再从分类页面跳转回到首页的时候，希望首页滚动的位置用户上一次离开的位置

1. 给路由外面包裹keep-alive，让home组件切换出去后不被销毁，而是缓存
2. 在home组件中，我们需要在activated第一次进入组件的时候给他一个固定值
3. 在deactivated中处理用户将要离开的操作

**在 2.2.0 及其更高版本中，activated 和 deactivated 将会在 <keep-alive> 树内的所有嵌套组件中触发**

## Vue中$nextTick 作用与原理

Vue 在更新 DOM 时是异步操作，在修改数据后，视图不会马上更新的，而是等同一事件中的所有数据变化完成之后，再统一进行视图更新。所以修改完数据之后，如果立即在方法中获取DOM，获取的仍然是未修改的DOM。

$nextTick的作用是：让方法中的代码在当前渲染完成后执行，解决异步渲染获取不到最新DOM的问题了。

原理：$nextTick本质是返回一个Promise

加分回答 应用场景：在created()当中想要获取操作Dom，就把操作DOM的方法放在$nextTick中

## 虚拟DOM

虚拟DOM是有程序员手动模拟实现的类似于浏览器中的DOM,但是有本质的区别

用传统的原生API或者jQuery去操作DOM浏览器会从构建DOM树开始从头到尾执行流程，如此在操作DOM上代价是非常大的，频繁的操作会出现页面卡顿影响用户体验的问题

虚拟DOM的作用就是为了解决浏览器这一性能问题

虚拟DOM的算法步骤主要分为三步

1. 使用javascript对象结构表示DOM树结构
2. 用这个树结构创建一颗真正的DOM树，插入到文档中，当状态发生改变，构造一颗心的对象树
3. 用新的树和旧的树进行比较，记录两颗树之间的差异，并将差异用到所构建的真正的DOM树上从而实现视图的更新，比较两颗树之间的差异用的是diff算法，核心为深搜

浏览器引擎工作流程

创建DOM tree 创建style Rules 构建Render tree 布局layout 绘制Painting

## Diff算法

Diff算法比较过程

第一步：patch函数中对新老节点进行比较 如果新节点不存在就销毁老节点 如果老节点不存在，直接创建新的节点 当两个节点是相同节点的时候，进入 patctVnode 的过程，比较两个节点的内部

第二步：patchVnode函数比较两个虚拟节点内部 如果两个虚拟节点完全相同，返回 当前vnode 的children 不是textNode，再分成三种情况 - 有新children，没有旧children，创建新的 - 没有新children，有旧children，删除旧的 - 新children、旧children都有，执行`updateChildren`比较children的差异，这里就是diff算法的核心 当前vnode 的children 是textNode，直接更新text

第三步：updateChildren函数子节点进行比较 - 第一步 头头比较。若相似，旧头新头指针后移（即 `oldStartIdx++` && `newStartIdx++`），真实dom不变，进入下一次循环；不相似，进入第二步。 - 第二步 尾尾比较。若相似，旧尾新尾指针前移（即 `oldEndIdx--` && `newEndIdx--`），真实dom不变，进入下一次循环；不相似，进入第三步。 - 第三步 头尾比较。若相似，旧头指针后移，新尾指针前移（即 `oldStartIdx++` && `newEndIdx--`），未确认dom序列中的头移到尾，进入下一次循环；不相似，进入第四步。 -

第四步 尾头比较。若相似，旧尾指针前移，新头指针后移（即 `oldEndIdx--` && `newStartIdx++`），未确认dom序列中的尾移到头，进入下一次循环；不相似，进入第五步。

第五步 若节点有key且在旧子节点数组中找到sameVnode（tag和key都一致），则将其dom移动到当前真实dom序列的头部，新头指针后移（即 `newStartIdx++`）；否则，vnode对应的dom（`vnode[newStartIdx].elm`）插入当前真实dom序列的头部，新头指针后移（即 `newStartIdx++`）。 - 但结束循环后，有两种情况需要考虑： - 新的字节点数组（newCh）被遍历完（`newStartIdx > newEndIdx`）。那就需要把多余的旧dom（`oldStartIdx -> oldEndIdx`）都删除，上述例子中就是`c,d`； - 新的字节点数组（oldCh）被遍历完（`oldStartIdx > oldEndIdx`）。那就需要把多余的新dom（`newStartIdx -> newEndIdx`）都添加。

## 模板是如何编译的

<https://fe.ecool.fun/topic/3e12b5bf-53ed-4b71-a199-49d7935f87b4?orderBy=updateTime&order=desc&tagId=14>

我们知道 <template></template> 这个是模板，不是真实的 HTML，浏览器是不认识模板的，所以我们需要把它编译成浏览器认识的原生的 HTML

这一块的主要流程就是：

1. 提取出模板中的原生 HTML 和非原生 HTML，比如绑定的属性、事件、指令等等
2. 经过一些处理生成 render 函数
3. render 函数再将模板内容生成对应的 vnode
4. 再经过 patch 过程( Diff )得到要渲染到视图中的 vnode
5. 最后根据 vnode 创建真实的 DOM 节点，也就是原生 HTML 插入到视图中，完成渲染

## CSS scoped 的原理

Scoped的作用：使样式只作用于当前组件，避免组件之间样式污染

原理：待研究

## vue3中怎么设置全局变量？

**方法1：使用config.globalProperties**

vue2.x挂载全局是使用 Vue.prototype.$xxxx=xxx 的形式来挂载，然后通过 this.$xxx来获取挂载到全局的变量或者方法。

这在 Vue 3 中，就等同于 config.globalProperties。这些 property 将被复制到应用中作为实例化组件的一部分

**方法2：使用provide/inject**

vue3新的 provide/inject 功能可以穿透多层组件，实现数据从父组件传递到子组件。

可以将全局变量放在根组件的 provide 中，这样所有的组件都能使用到这个变量。

如果需要变量是响应式的，就需要在 provide 的时候使用 ref 或者 reactive 包装变量。

## 刷新浏览器后，Vuex的数据是否存在？如何解决？

页面刷新之后vuex中的数据将恢复初始状态

原因：因为 store 里的数据是保存在运行内存中的，当页面刷新时，页面会重新加载vue实例，store里面的数据就会被重新赋值初始化

解决办法：

1. 使用vuex-along
2. 保存到localStorage 或者 sessionStroage

## $route 和 $router 有什么区别

this.$router: 是路由实例，通过它访问路由，相当于获取了整个路由文件，可调用路由方法(路由导航守卫)

this.route: 当前激活的路由信息对象。这个属性是只读的，里面的属性是 immutable (不可变) 的，不过可以 watch (监测变化) 它。

通过 this.$route 访问的是当前路由，获取和当前路由有关的信息

fullPath: "" // 当前路由完整路径，包含查询参数和 hash 的完整路径

hash: "" // 当前路由的 hash 值 (锚点)

matched: [] // 包含当前路由的所有嵌套路径片段的路由记录

meta: {} // 路由文件中自赋值的meta信息

name: "" // 路由名称

params: {} // 一个 key/value 对象，包含了动态片段和全匹配片段就是一个空对象。

path: "" // 字符串，对应当前路由的路径

query: {} // 一个 key/value 对象，表示 URL 查询参数。跟随在路径后用'?'带的参数

## 自定义指令是什么？有哪些应用场景？

自定义指令也像组件那样存在钩子函数：

* bind：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置
* inserted：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)
* update：所在组件的 VNode 更新时调用，但是可能发生在其子 VNode 更新之前。指令的值可能发生了改变，也可能没有。但是你可以通过比较更新前后的值来忽略不必要的模板更新
* componentUpdated：指令所在组件的 VNode 及其子 VNode 全部更新后调用
* unbind：只调用一次，指令与元素解绑时调用

所有的钩子函数的参数都有以下：

* el：指令所绑定的元素，可以用来直接操作 DOM
* binding：一个对象，包含以下 property：
  + name：指令名，不包括 v- 前缀。
  + value：指令的绑定值，例如：v-my-directive="1 + 1" 中，绑定值为 2。
  + oldValue：指令绑定的前一个值，仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。
  + expression：字符串形式的指令表达式。例如 v-my-directive="1 + 1" 中，表达式为 "1 + 1"。
  + arg：传给指令的参数，可选。例如 v-my-directive:foo 中，参数为 "foo"。
  + modifiers：一个包含修饰符的对象。例如：v-my-directive.foo.bar 中，修饰符对象为 { foo: true, bar: true }
* vnode：Vue 编译生成的虚拟节点
* oldVnode：上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用

除了 el 之外，其它参数都应该是只读的，切勿进行修改。如果需要在钩子之间共享数据，建议通过元素的 dataset 来进行

## Vue项目中有封装过axios吗？怎么封装的？

为什么要封装：

axios 的 API 很友好，你完全可以很轻松地在项目中直接使用。

不过随着项目规模增大，如果每发起一次HTTP请求，就要把这些比如设置超时时间、设置请求头、根据项目环境判断使用哪个请求地址、错误处理等等操作，都需要写一遍

这种重复劳动不仅浪费时间，而且让代码变得冗余不堪，难以维护。为了提高我们的代码质量，我们应该在项目中二次封装一下 axios 再使用

如果每个页面都发送类似的请求，都要写一堆的配置与错误处理，就显得过于繁琐了这时候我们就需要对axios进行二次封装，让使用更为便利

如何封装：

封装的同时，你需要和 后端协商好一些约定，请求头，状态码，请求超时时间.......

设置接口请求前缀：根据开发、测试、生产环境的不同，前缀需要加以区分

请求头 : 来实现一些具体的业务，必须携带一些参数才可以请求(例如：会员业务)

状态码: 根据接口返回的不同status ， 来执行不同的业务，这块需要和后端约定好

请求方法：根据get、post等方法进行一个再次封装，使用起来更为方便

请求拦截器: 根据请求的请求头设定，来决定哪些请求可以访问

响应拦截器： 这块就是根据 后端`返回来的状态码判定执行不同业务

## Vue中组件和插件有什么区别

待研究

## 指令是什么，vue为什么会出现闪烁

指令就是自定义属性，vue 中的自定义属性都以 v- 开头

代码在加载过程中先加载HTML,在加载HTML过程中将插值表达式作为HTML的内容显示在

页面，然后加载完 js 之后才将插值表达式替换掉，所以为出现闪烁的问题

使用 v-cloak 指令可解决

## 对v-model的理解

vue 中的 v-model 主要用于双向数据绑定

数据改变页面内容跟着改变，页面内容改变数据也跟着改变

实现原理：也就是给表单控件绑定了一个 input 事件和 v-bind动态属性绑定，当表单

中的value值发生改变将触发input事件，事件处理函数将最新的 value值

覆盖旧的value值，由于表单的value值为v-bind动态属性绑定，当data中的

值发生改变时表单中的value值也会跟着改变

# Js

## 变量

变量是程序在内存中申请的一块存储数据的空间，变量名为这个空间的名字，赋值表示在这个空间中放进内容，类似酒店住进了人

## JS数据类型有哪些,区别是什么

简单数据类型： Number、String、Boolean、Null、Undefined、BigInt、Symbol

复杂数据类型： Object （代表，普通对象，数组，正则，日期，Math数学函数都属于Object。）

不同的数据类型占用的内存空间是不一样的，数据类型分简单数据类型 和复杂数据类型

简单数据类型变量存放的为值本身，复杂数据类型变量存放的为值所对应的地址

## Continue break return区别

Continue：跳出本次循环继续下一次循环，

break：跳出整个循环，后面的循环不执行

rerurn：不仅可以退出循环，还能够返回return语句中的值，同时还可以结束当前函

数体内部的代码

## 作用域，作用域链

作用域是在程序运行时代码中的某些特定部分中变量、函数和对象的**可访问性**。

作用域就是变量的使用范围

全局作用域：作用于所有代码执行的环境（整个script标签内部）

局部作用域：作用于函数内部的代码环境就是局部作用域，

快级作用域：ES6才出现快级作用域

作用域链：

多个作用域对象连续引用形成的链式结构。

使用方面解释：Js使用一个变量时，js引擎会尝试在当前作用域去寻找改变量，如果没有，则会去的上层作用域寻找，以此类推直到找到该变量或已经到了全局作用域，如果在全局作用域仍然没有找到该变量，就会报错

存储方面解释：作用域链在js内部以数组的形式存储，数组的第一个索引对应函数本身的执行期上下文，也就是当前执行代码所在环境的变量对象，下一个索引对应的空间存储的是该对象的外部执行环境，以此类推，一直到全局执行环境

如果函数中还有函数，那么在这个作用域中就又可以诞生一个作用域

根据在内部函数中可以访问外部函数中变量的这种机制，用链式查找决定哪

些数据能被内部函数访问，就称做作用域链

## Var let const的区别

1. Var声明的范围是函数作用域，let和const 声明的范围是块作用域
2. Var声明的变量会被提升到函数作用域的顶部，let 和 const 声明的变量不存在提升，且具有暂时性死区特征
3. Var 允许在同一个作用域中重复声明同一个变量，let 和 const 不允许

在全局作用域中使用var声明的变量会成为window对象的属性，let 和 const 声明的变量则不会

1. Const 的行为与let基本相同，唯一一个重要的区别是，使用const 声明的变量必须进行初始化，且不能被修改

## 原型，原型链

原型

每一个对象实例都有一个指向原型对象的指针（\_\_proto\_\_）它会在创建对象时自动创建，我们访问对象上的属性/方法时，如果对象实例上有该属性/方法，就返回该属性/方法，如果没有，就去原型指针指向的原型对象上查找对应属性/方法，依次往上查找形成一条原型链

构造函数

如果说对象实例的原型指针（\_\_proto\_\_）只是一个指向原型对象的指针，那就说明在此之前原型对象已经创建了，那么原型对象是什么时候被创建的呢?这就要引入构造函数的概念

其实构造函数也只是一个普通的函数，如果这个函数可以使用new关键字来创建它的实例对象，那么我们就把这种函数称为构造函数

原型对象是在构造函数被声明时一同创建的。构造函数被声明时，原型对象也一同完成创建，然后挂载到构造函数的prototype属性上

原型对象被创建时，会自动生成一个constructor属性，指向创建它的构造函数。这样他两的关系就关联起来了

## JS变量提升

变量提升是指变量和函数声明会在代码编译期，提升到代码的最前面。

只有使用var关键字申明的变量才会出现变量提升，提升的只是变量名，赋值不会提升

函数的提升优先级比变量要高

变量提升的结果：是可以在变量初始化之前访问该变量，返回的是undefined。在函数声明前可以调用该函数。

加分回答 使用let和const声明的变量不会出现变量提升，let const声明的变量会形成暂时性死区，在初始化之前访问将会报错。

## JavaScript判断变量的类型

判断变量的类型4种方法: 分别是typeof、instanceof、Object.prototype.toString.call()（对象原型链判断方法）、 constructor (用于引用数据类型)

typeof：常用于判断基本数据类型，对于引用数据类型除了function返回’function‘，其余全部返回’object'。

instanceof：主要用于区分引用数据类型，不太适合用于简单数据类型的检测，检测过程繁琐且对于简单数据类型中的undefined, null, symbol检测不出来。

constructor：用于检测引用数据类型，检测方法是获取实例的构造函数判断和某个类是否相同，如果相同就说明该数据是符合那个数据类型的，这种方法不会把原型链上的其他类也加入进来，避免了原型链的干扰。

Object.prototype.toString.call()：该方法可对各个类型进行检测，返回的是该数据类型的字符串。

加分回答

instanceof的实现原理：验证当前类的原型prototype是否会出现在实例的原型链\_\_proto\_\_上，只要在它的原型链上，则结果都为true。因此，`instanceof` 在查找的过程中会遍历左边变量的原型链，直到找到右边变量的 `prototype`，找到返回true，未找到返回false。

Object.prototype.toString.call()原理： Object.prototype.toString 表示一个返回对象类型的字符串，call()方法可以改变this的指向，那么把Object.prototype.toString()方法指向不同的数据类型上面，返回不同的结果

## null 和 undefined 的区别

undefind 是全局对象的一个属性，

当一个变量没有被赋值—返回undefind

当一个函数没有返回值—返回undefined

当访问某个对象中不存在的属性—返回undefind

当函数定义了形参但没有传递实参这时候为undefind

undefined通过typeof判断类型是'undefined'。undefined == undefined undefined === undefined 。

null代表对象的值未设置，相当于一个对象没有设置指针地址就是null。

null通过typeof判断类型是'object'。null === null null == null null == undefined null !== undefined

undefined 表示一个变量初始状态值，而 null 则表示一个变量被人为的设置为空对象，而不是原始状态。

在实际使用过程中，不需要对一个变量显式的赋值 undefined，当需要释放一个对象时，直接赋值为 null 即可。 让一个变量为null，直接给该变量赋值为null即可。

加分回答 null 其实属于自己的类型 Null，而不属于Object类型，typeof 之所以会判定为 Object 类型，是因为JavaScript 数据类型在底层都是以二进制的形式表示的，二进制的前三位为 0 会被 typeof 判断为对象类型，而 null 的二进制位恰好都是 0 ，因此，null 被误判断为 Object 类型。

对象被赋值了null 以后，对象对应的堆内存中的值就是游离状态了，GC 会择机回收该值并释放内存。因此，需要释放某个对象，就将变量设置为 null，即表示该对象已经被清空，目前无效状态。

## Js中this的指向

This 的指向取决于他的调用者

普通函数的调用this指向window

构造函数调用this指向实例对象

对象方法的调用this指向该方法所属对象

事件绑定方法this指向绑定事件对象

定时器函数this指向window

立即执行函数this指向window

关于this 的指向可通过call（）apply（）bind（）等方法进行改变

## call apply bind的区别

Call（）: 改变this指向的同时会调用函数，函数的参数用逗号隔开，常用于继承

Apply（）：改变this指向的同时调用函数，函数的参数为数组类型，常用于处理数组

Bind（）：改变this的指向但是不会调用函数

## 箭头函数于普通函数的区别

1. 箭头函数不会创建自己的this（它的this指向定义时外层执行环境的this,箭头函数中this的指向在它被定义的时候就已经确定了，之后永远不会改变）
2. Call apply bind 无法改变箭头函数中this指向
3. 箭头函数不能作为构造函数使用
4. 箭头函数没有自己的arguments（箭头函数中访问arguments实际获取到的是外层局部函数执行环境中的值）
5. 箭头函数没有原型prototype

## New做了啥

1. Js内部会创建一个空对象
2. 再把函数中的this指向该对象
3. 执行构造函数中的语句
4. 返回该对象

## Map和forEach的区别

map有返回值，可以开辟新空间，return出来一个length和原数组一致的数组，即便数组元素是undefined或者是null。

forEach默认无返回值，返回结果为undefined，可以通过在函数体内部使用索引修改数组元素。

加分回答 map的处理速度比forEach快，而且返回一个新的数组，方便链式调用其他数组新方法，比如filter、reduce

let arr = [1, 2, 3, 4, 5];

let arr2 = arr.map(value => value \* value).filter(value => value > 10); // arr2 = [16, 25]

## Js执行机制（事件循环Event loop），宏任务与微任务

1. 先执行执行栈中的同步任务
2. 异步任务（回调函数）会放在任务队列中
3. 一但执行栈中的所有同步任务执行完成之后，系统会按次序读取任务队列中的异步任务被读取到的异步任务将结束等待状态进去执行栈中开始执行

---------------------------------------------------------------

浏览器的事件循环：执行js代码的时候，遇见同步任务，直接推入调用栈中执行，遇到异步任务，将该任务挂起，等到异步任务有返回之后推入到任务队列中，当调用栈中的所有同步任务全部执行完成，将任务队列中的任务按顺序一个一个的推入并执行，重复执行这一系列的行为。

异步任务又分为宏任务和微任务。

宏任务：任务队列中的任务称为宏任务，每个宏任务中都包含了一个微任务队列。

微任务：等宏任务中的主要功能都完成后，渲染引擎不急着去执行下一个宏任务，而是执行当前宏任务中的微任务

宏任务包含：执行script标签内部代码、setTimeout/setInterval、ajax请、postMessageMessageChannel、setImmediate，I/O（Node.js）

微任务包含：Promise、MutonObserver、Object.observe、process.nextTick（Node.js）

加分回答 浏览器和Node 环境下，microtask 任务队列的执行时机不同 - Node端，microtask 在事件循环的各个阶段之间执行 - 浏览器端，microtask 在事件循环的 macrotask 执行完之后执行

## cookie sessionStorage localStorage 的区别

共同点：数据都是存储在浏览器本地的,数据共享遵循同源原则，SessionStorage还限制必须是同一个页面

区别：

cookie是由服务器端写入的（存储大小大概4k），而SessionStorage、 LocalStorage都是由前端写入的（存储大小大概5M）；

SessionStorage保存的数据页面关闭后自动清除，LocalStorage保存的数据需要手动清除；cookie的生命周期是由服务器端在写入的时候就设置好的；

在前端给后端发送请求的时候会自动携带Cookie中的数据，但是SessionStorage、 LocalStorage不会；

Cookie一般用于存储登录验证信息，常用于存储不易变动的数据，减轻服务器的压力，SessionStorage可以用作刷新页面时数据回显

## localstorage数据过期处理？

localStorage保存的数据没有过期时间，除非手动删除。所以要实现localStorage缓存数据的过期处理重点是如何清理过期的缓存。 目前有两种方法，一种是惰性删除，另一种是定时删除。

惰性删除是指某个键值过期后，该键值不会被马上删除，而是等到下次被使用时，如果被检查到已过期，才会删除。

实现方法是：存储一个对象，对象有两个key，一个是要存储的value值，另一个是当前时间。获取数据的时候，将存储的时间和当前时间做对比，如果过期则清除

定时删除是指，每隔一段时间执行一次删除操作，并通过限制删除操作执行的次数和频率，来减少删除操作对CPU的长期占用。另一方面定时删除也有效的减少了因惰性删除带来的对localStorage空间的浪费。

实现过程，获取所有设置过期时间的key判断是否过期，过期就存储到数组中，遍历数组，每隔1S（固定时间）删除5个（固定个数），直到把数组中的key从localstorage中全部删除。

加分回答 LocalStorage清空应用场景：token存储在LocalStorage中，要清空

## token 能放在cookie中吗？

token一般是用来判断用户是否登录的，它内部包含的信息有：

uid(用户唯一的身份标识)、

time(当前时间的时间戳)、

sign（签名，token 的前几位以哈希算法压缩成的一定长度的十六进制字符串）

token可以存放在`Cookie`中，`token` 是否过期，应该由后端来判断，不该前端来判断，所以`token`存储在`cookie`中只要不设置`cookie`的过期时间就ok了，如果 `token` 失效，就让后端在接口中返回固定的状态表示`token` 失效，需要重新登录，再重新登录的时候，重新设置 `cookie` 中的 `token` 就行。

加分回答 token认证流程

1. 客户端使用用户名跟密码请求登录

2. 服务端收到请求，去验证用户名与密码

3. 验证成功后，服务端签发一个 token ，并把它发送给客户端

4. 客户端接收 token 以后会把它存储起来，比如放在 cookie 里或者 localStorage 里

5. 客户端每次发送请求时都需要带着服务端签发的 token（把 token 放到 HTTP 的 Header 里）

6. 服务端收到请求后，需要验证请求里带有的 token ，如验证成功则返回对应的数据

## 对闭包的理解

闭包：闭包是一个函数，一般就是一个函数A，return其内部的函数B，被return出去的B函数能够在外部访问A函数内部的变量，这时候就形成了一个B函数的变量背包，A函数执行结束后这个变量背包也不会被销毁，并且这个变量背包在A函数外部只能通过B函数访问。

闭包形成的原理：作用域链，当前作用域可以访问上级作用域中的变量

闭包解决的问题：能够让函数作用域中的变量在函数执行结束之后不被销毁，同时也能在函数外部可以访问函数内部的局部变量。

闭包带来的问题：由于垃圾回收器不会将闭包中变量销毁，于是就造成了内存泄露，内存泄露积累多了就容易导致内存溢出。

加分回答，闭包的应用：能够模仿块级作用域，能够实现柯里化，在构造函数中定义特权方法、Vue中数据响应式Observer中使用闭包等。

## Promise

**简介（https://juejin.cn/post/7043758954496655397）**

Promise 是一种处理异步操作使其不会陷入回调地狱的方式

Promise对象代表一个异步操作，它有三种状态：

pending（进行中）: 初始状态，既没有被兑现，也没有被拒绝

fulfilled（已成功）： 操作成功

rejected（已失败）：操作失败

**promise如何运行**

当promise被调用后它会以padding状态开始，执行成功后将调用resolve回调（传给**then** ）进入已完成状态，否则将执行reject回调（传给**catch**）进入被拒绝状态，如果一直未调用resolve和reject回调，则promise会一直处于pending状态（pending状态下的promise不会执行回调函数then()）

注意：必须给Promise对象传入一个执行函数，否则将会报错

**如何手写promise**

手写Promise包含以下知识点

1. Promise
2. Class类
3. 改变this指向（call,apply,bind）

--------------------------------------------------------------------------

Promise的作用：Promise是异步微任务，解决了异步多层嵌套回调的问题，让代码的可读性更高，更容易维护

Promise使用：

Promise是ES6提供的一个构造函数，可以使用Promise构造函数new一个实例，Promise构造函数接收一个函数作为参数，这个函数有两个参数，分别是两个函数 `resolve`和`reject`，

`resolve`将Promise的状态由等待变为成功，将异步操作的结果作为参数传递过去；

`reject`则将状态由等待转变为失败，在异步操作失败时调用，将异步操作报出的错误作为参数传递过去。

实例创建完成后，可以使用`then`方法分别指定成功或失败的回调函数，也可以使用catch捕获失败，then和catch最终返回的也是一个Promise，所以可以链式调用。

Promise的特点：

1. 对象的状态不受外界影响（Promise对象代表一个异步操作，有三种状态）。 - pending（执行中） - Resolved（成功，又称Fulfilled） - rejected（拒绝） 其中pending为初始状态，fulfilled和rejected为结束状态（结束状态表示promise的生命周期已结束）。

2. 一旦状态改变，就不会再变，任何时候都可以得到这个结果。 Promise对象的状态改变，只有两种可能（状态凝固了，就不会再变了，会一直保持这个结果）： - 从Pending变为Resolved - 从Pending变为Rejected

3. resolve 方法的参数是then中回调函数的参数，reject 方法中的参数是catch中的参数

4. then 方法和 catch方法 只要不报错，返回的都是一个fullfilled状态的promise

加分回答 Promise的其他方法：

Promise.resolve() :返回的Promise对象状态为fulfilled，并且将该value传递给对应的then方法。

Promise.reject()：返回一个状态为失败的Promise对象，并将给定的失败信息传递给对应的处理方法。

Promise.all()：返回一个新的promise对象，该promise对象在参数对象里所有的promise对象都成功的时候才会触发成功，一旦有任何一个iterable里面的promise对象失败则立即触发该promise对象的失败。

Promise.any()：接收一个Promise对象的集合，当其中的一个 promise 成功，就返回那个成功的promise的值。

Promise.race()：当参数里的任意一个子promise被成功或失败后，父promise马上也会用子promise的成功返回值或失败详情作为参数调用父promise绑定的相应句柄，并返回该promise对象。

## 递归

如果一个函数在内部可以调用其本身，那么这个函数就是递归函数。

简单理解：函数内部自己调用自己，这个函数就是递归函数，递归函数的作用和循环效果一样，由于递归很容易发生栈溢出错误，所以必须要加退出条件

## 跨域是什么？如何解决？

跨域：当前页面某个接口的请求地址和当前页面地址如果在协议、域名、端口三者当中有一项不同，就说明该接口跨域了。

跨域限制的原因：浏览器同源策略所导致的，目的是为了保证网页的安全。

跨域解决方案:

JSONP：

原理是script标签可以跨域请求资源，将回调函数作为参数拼接在url中。后端收到请求，调用该回调函数，并将数据作为参数返回去，注意设置响应头返回文档类型，应该设置成javascript。

CORS（跨域资源共享）：目前最常用的一种解决办法，通过设置后端允许跨域实现。  
res.setHeader('Access-Control-Allow-Origin', '\*');  
res.setHeader("Access-Control-Allow-Methods", "GET, PUT, OPTIONS, POST");

node中间件、nginx反向代理：跨域限制的时候浏览器不能跨域访问服务器，node中间件和nginx反向代理，都是让请求发给代理服务器，静态页面面和代理服务器是同源的，然后代理服务器再向后端服务器发请求，服务器和服务器之间不存在同源限制。

postmessage：H5新增API，通过发送和接收API实现跨域通信。

跨域场景：前后端分离式开发、调用第三方接口

-------------------------------------------------------

原因：跨域造成的原因是浏览器同源政策导致的

同源：两个页面拥有相同的协议 域名 端口，若同源，则一个页面可访问另一个页面的数据

跨域是指绕过浏览器**同源策略**约束请求资源的方式。

为什么需要同源政策：主要防止XSS(跨站脚本攻击)、CSRF(跨站请求伪造) 等攻击。

如何解决跨域

1. JSONP

原理：使用<script>标签中的src属性发送请求，src属性不受同源政策的限制

封装JSONP

客户端：

<1>点击按钮调用JSONP函数，传递参数给ajax

<2>函数内容

1. 通过document.createElementScript 创建一个script标签
2. 拼接请求参数的字符串
3. 通过Math.random().toString创建一个随机函数名
4. 通过window[名字]将这个函数名添加到window全局范围的属性，属性值为自定义的success回调函数
5. 为script标签追加src属性，属性值为地址 回调函数名和参数
6. 将script标签添加到页面中通过document.body.appendChild
7. 为script标签添加onLoad事件加载完成之后通过

document.body.removeChild(script)删除标签

服务器端

1. 通过req.query.callBack接收回调函数的名称
2. 将函数名称对应的函数调用以字符串的形式返回给客户端并携带响应数据

可直接写成 res.jsonp（响应数据）

通过res.jsonp（响应数据）将数据响应给客户端

1. CORS跨域资源共享

原理：同源政策是浏览器给予ajax的机制，服务器端不受限制，所以我们让自己的服务器去别的服务器获取数据，然后让客户端在自己的服务器中获取自己服务器从别的服务器获取的数据

实现过程：

1. 客户端依旧发送ajax请求
2. 自己的服务器端通过第三方模块request向其它服务器发送请求（参数为地址， 回调函数）
3. CORS跨域的关键是其它的服务器有没有允许我们的服务器对其进行数据请求，其它服务器端响应头需要设置的三个内容为

<1>res.header(Access-Control-Allow-Origin, 地址)允许哪个客户端的服务器对其访问

<2>res.header(Access-Control-Allow-Methods, 方法)允许用什么方法进行访问

<3>res.header(Access-Contral-Allow-Credentials, true)允许跨域请求时携带cookie

## JS实现异步的方法

所有异步任务都是在同步任务执行结束之后，从任务队列中依次取出执行。

回调函数是异步操作最基本的方法，比如AJAX回调，

回调函数的优点是简单、容易理解和实现，

缺点是不利于代码的阅读和维护，各个部分之间高度耦合，使得程序结构混乱、流程难以追踪（尤其是多个回调函数嵌套的情况），而且每个任务只能指定一个回调函数。此外它不能使用 try catch 捕获错误，不能直接 return

Promise包装了一个异步调用并生成一个Promise实例，当异步调用返回的时候根据调用的结果分别调用实例化时传入的resolve 和 reject方法，then接收到对应的数据，做出相应的处理。

Promise不仅能够捕获错误，而且也很好地解决了回调地狱的问题，

缺点是无法取消 Promise，错误需要通过回调函数捕获。

Generator 函数是 ES6 提供的一种异步编程解决方案，Generator 函数是一个状态机，封装了多个内部状态，可暂停函数, yield可暂停，next方法可启动，每次返回的是yield后的表达式结果。

优点是异步语义清晰，

缺点是手动迭代`Generator` 函数很麻烦，实现逻辑有点绕 async/awt是基于Promise实现的，

async/await使得异步代码看起来像同步代码，所以优点是，使用方法清晰明了，

缺点是awaitt 将异步代码改造成了同步代码，如果多个异步代码没有依赖性却使用了 await 会导致性能上的降低，

代码没有依赖性的话，完全可以使用 Promise.all 的方式。

加分回答

JS 异步编程进化史：callback -> promise -> generator/yield -> async/await。

async/await函数对 Generator 函数的改进，体现在以下三点： -

内置执行器。 Generator 函数的执行必须靠执行器，而 async 函数自带执行器。也就是说，async 函数的执行，与普通函数一模一样，只要一行。 -

更广的适用性。 yield 命令后面只能是 Thunk 函数或 Promise 对象，而 async 函数的 awt 命令后面，可以跟 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。 -

更好的语义。 async 和 await，比起星号和 yield，语义更清楚了。async 表示函数里有异步操作，await表示紧跟在后面的表达式需要等待结果。 目前使用很广泛的就是promise和async/ await

## 数组去重方法

第一种方法：利用对象属性key排除重复项：遍历数组，每次判断对象中是否存在该属性，不存在就存储在新数组中，并且把数组元素作为key，设置一个值，存储在对象中，最后返回新数组。这个方法的优点是效率较高，缺点是占用了较多空间，使用的额外空间有一个查询对象和一个新的数组

第二种方法：利用Set类型数据无重复项：new 一个 Set，参数为需要去重的数组，Set 会自动删除重复的元素，再将 Set 转为数组返回。这个方法的优点是效率更高，代码简单，思路清晰，缺点是可能会有兼容性问题

第三种方法：filter+indexof 去重：这个方法和第一种方法类似，利用 Array 自带的 filter 方法，返回 arr.indexOf(num) 等于 index 的num。原理就是 indexOf 会返回最先找到的数字的索引，假设数组是 [1, 1]，在对第二个1使用 indexOf 方法时，返回的是第一个1的索引0。这个方法的优点是可以在去重的时候插入对元素的操作，可拓展性强。

第四种方法：这个方法比较巧妙，从头遍历数组，如果元素在前面出现过，则将当前元素挪到最后面，继续遍历，直到遍历完所有元素，之后将那些被挪到后面的元素抛弃。这个方法因为是直接操作数组，占用内存较少。

第五种方法：reduce +includes去重：这个方法就是利用reduce遍历和传入一个空数组作为去重后的新数组，然后内部判断新数组中是否存在当前遍历的元素，不存在就插入到新数组中。这种方法时间消耗多，内存空间也有额外占用。 方法还有很多，常用的、了解的这些就可以

加分回答 以上五个方法中，在数据低于10000条的时候没有明显的差别，高于10000条，第一种和第二种的时间消耗最少，后面三种时间消耗依次增加，由于第一种内存空间消耗比较多，且现在很多项目不再考虑低版本浏览器的兼容性问题，所以建议使用第二种去重方法，简洁方便。

## 伪数组和数组的区别

伪数组的类型不是Array，而是Object，可以使用length属性查看长度，也可以使用[index]获取某个元素，但是不能使用数组的其他方法，也不能改变长度，遍历使用for in方法。

伪数组的常见场景：

函数的参数arguments

原生js获取DOM：document.querySelector('div') 等

jquery获取DOM：$(“div”)等

加分回答 伪数组转换成真数组方法

Array.prototype.slice.call(伪数组)

[].slice.call(伪数组)

Array.from(伪数组) 转换后的数组长度由 `length` 属性决定。索引不连续时转换结果是连续的，会自动补位。

## Defer和async的区别

浏览器会立即加载JS文件并执行指定的脚本，“立即”指的是在渲染该 script 标签之下的文档元素之前，也就是说不等待后续载入的文档元素，读到就加载并执行 `` 加上async属性，加载JS文档和渲染文档可以同时进行（异步），当JS加载完成，JS代码立即执行，会阻塞HTML渲染。 `` 加上defer，加载后续文档元素的过程将和 script.js 的加载并行进行（异步），当HTML渲染完成，才会执行JS代码。

加分回答 渲染阻塞的原因： 由于 JavaScript 是可操纵 DOM 的,如果在修改这些元素属性同时渲染界面（即 JavaScript 线程和 UI 线程同时运行）,那么渲染线程前后获得的元素数据就可能不一致了。因此为了防止渲染出现不可预期的结果,浏览器设置 GUI 渲染线程与 JavaScript 引擎为互斥的关系。当浏览器在执行 JavaScript 程序的时候,GUI 渲染线程会被保存在一个队列中,直到 JS 程序执行完成,才会接着执行。如果 JS 执行的时间过长,这样就会造成页面的渲染不连贯,导致页面渲染加载阻塞的感觉

------------------------------------------------------------------

在 HTML 中会遇到以下三类 script：

<script src='xxx'></script>

<script src='xxx' async></script>

<script src='xxx' defer></script>

script标签用于加载脚本与执行脚本，直接使用script脚本时，html会按照顺序来加载并执行脚本，在脚本加载&执行的过程中，会阻塞后续的DOM渲染。

比如现在大家习惯于在页面中引用各种第三方脚本，但如果第三方服务商出现了一些小问题，比如延迟之类的，就会使得页面白屏。

针对上述情况，script标签提供了两种方式来解决问题，就是加入属性async以及defer，这两个属性使得script标签加载都不会阻塞DOM的渲染。

defer：此布尔属性被设置为向浏览器指示脚本在文档被解析后执行。

async：设置此布尔属性，以指示浏览器如果可能的话，应异步执行脚本。

Defer

如果script标签设置了defer属性，则浏览器会异步下载该文件并且不会影响后续DOM的渲染。

如果有多个设置了defer属性的script标签存在，则会按照顺序执行所有的script，defer脚本会在文档渲染完毕后，DOMContentLoaded事件调用前执行。

Async

async属性会使得script脚本异步的加载并在允许的情况下执行，而async的执行并不会按照script标签在页面中的顺序来执行，而是谁先加载完谁先执行。

## 事件函数是怎么传参的

如果事件直接绑定函数名称，那么默认会传递事件对象作为事件函数的第一个参数

如果事件绑定函数调用，那么事件对象必须作为最后一个参数显示传递，

并且事件对象的名称必须是$event

# jQuery

## 什么是就jQuery

简单理解就是一个JS文件，里面对我们原生的js代码进行了封装存放在里面。这样我们可以快速高效的使用这些封装好的功能了

## 怎么使用jQuery

下载文件，然后在项目中导入文件，jQuery的顶级对象为$,该符号为jQuery的别名，把元素利用$ 包装成jQuery对象，就可以调用jQuery的方法了

## 什么是jQuery对象和DOM对象

使用原生JS获取的对象为DOM对象

使用jQuery方法获取的元素就是jQuery对象，本质就是利用$对DOM对象包装后产生的对象（伪数组形式存储）

## jQuery对象和DOM对象之间的转换

由于原生js比jQuery更大，原生的一些属性和方法jQuery没有给我们封装，要想使用这些方法需要将jQuery对象转换为DOM对象才能使用

## DOM对象转换为jQuery对象

直接使用$对原生的DOM对象进行包装即可

## jQuery对象转换为DOM对象

 jQuery对象[索引值] var domObject1 = $('div')[0]

jQuery对象.get(索引值) var domObject2 = $('div').get(0)

## jQuery常用的API

隐式迭代：遍历DOM元素（伪数组形式存储）的过程就叫做隐式迭代

简单理解就是给匹配到的元素进行遍历循环，每一元素都执行相应的方法而不用我们使用for进行循环，简化了我们的操作方便我们调用

要对元素进行操作首先要获取元素，此时将涉及到选择器筛选方法的使用

Parent（） children（） find（） siblings（） eq（）

**样式操作**

简单的样式：通过**css（）**方法来进行操作

样式很多：通过设置类来修改样式

元素.addClass（‘类名’） 元素.remoevClass（‘类名’）

元素.toggleClass（‘类名’），切换类有着删除没有就添加

原生JS中的className会覆盖元素原先里面的类名

jQuery里面类操作只是对指定类进行操作，不影响原先的类名

**效果**

**显示与隐藏**

Show（） hide（） toggle（）

**滑动效果**

slideDown（） slideUp（） slideToggle（）

**淡入淡出**

fadeIn（） fadeout（） fadeToggle（） fadeTo（）

**自定义动画**

Animate（）函数中的参数为对象的数据类型盒子的有定位

**属性操作**

Prop（“属性名”）获取属性值 prop（“属性名”，“属性值”）设置属性

Attr（“属性名”）获取自定义的属性值 attr（“属性名”，“属性值”）修改自定义属性值

Data（）可以在指定元素上存储数据，但不会修改DOM元素结构，页面跟新，存储的数据

将会被移除

**操作元素内容**

Html（）获取普通元素的内容 小括号为空表示得到元素内容，有值表示设置内容

Text（）

Val（） 获取表单里面的值

**操作元素**

**怎么遍历，创建， 添加， 删除元素**

**遍历**： 元素.each（回调函数（index，返回的每一DOM元素））

$.each（object，function（index，element）{ }）

$.each可以用于遍历任何对象，主要用于数据处理，比如数组，对象

**创建元素**：$(“标签”)

**添加元素**：

内部添加 element.append（“内容”）把内容放入匹配元素内部的最后面

Element.prepend（“内容”）内部添加放入到匹配元素内部的前面

外部添加：element.after（“内容”）把内容放入目标元素的后面

Element.before（“内容”）把内容放入到目标元素的前面

**删除元素**：element.remove（）删除匹配的元素删除本身

Element.empty（）删除匹配元素集合中所有的子节点

Element.html（“”）可以删除匹配元素里面的子节点 孩子

**事件**

**事件注册：**

**On（）：**在匹配元素上绑定一个或多个事件的处理函数

可以进行事件委派操作，事件委派就是把原来加给子元素身上的事件绑定在

父元素身上，就是把事件委派给父元素

可以给动态创建的元素绑定事件

**One（）**：使用one绑定的事件只会触发一次

**Triggle（事件）：**自动触发事件 element.click（）也可以自动触发事件

**triggerHandler（）：不会触发元素的默认行为**

**事件注销**：off（）

**事件对象：**

**Event.preventDefault（） 或者 return false 阻止默认行为**

**Event.stopPropagation（） 阻止冒泡**

**拷贝对象**

$.extend（[deep],target,object,[objectN]） deep表示深拷贝默认false浅拷贝

浅拷贝只是把复杂数据类型中的地址拷贝给了目标对象修改目标对象会影响原对象

深拷贝是完全克隆，拷贝的是对象而不是地址，修改目前对象不会影响原对象

**多库共存**

jQuery使用$作为标示符，随着jQuery的流行其他的js库也会用这个标识符，这样便

会起冲突，解决方案：把里面的$符号统一修改为jQuery,或者通过$noConflict()自定

义一个符号

**插件**

**jQuery插件常用网站**

jQuery 插件库 <http://www.jq22.com/>

jQuery 之家 http://www.htmleaf.com/

使用步骤

引入jQuery文件和插件文件

复制相关html css js （调用插件）

**瀑布流插件 轮播图插件**

**尺寸位置操作**

**Width（）height（）本身的宽高**

**innerWidth（）innerHeight（）包含padding值**

**outWidth（）outHeight（）包含padding和边框**

**outWidth（true）包含padding 边框 还有margin**

**括号里面没有值表示获取元素的宽高，有值则是修改宽高**

**位置：**

**Offset（） 设置或返回被选元素相对于文档的偏移坐标，跟父级没有关系**

**position（） 返回被选元素相对于带有定位的父级偏移坐标，不能设置偏移**

**scrollTop（）/scrollLeft（）：设置或返回被选元素被卷去的头部**

**$(“body,html”).stop().animate（{ scrollTop：0 }）是返回到顶部有动画效果**

# H5C3

## 语义化标签

Header aside footer article section video audio dialog nav...

## Px rem em vx的区别

1. px：相对长度单位，像素px是相对于显示器屏幕分辨率而言的
2. em：是相对长度单位（相对父级元素字体大小），相对于当前对象内文本的字体尺寸。如果当前对象内文本字体尺寸未被人为设置，则相对于浏览器的默认字体大小

1em指的是一个字体的大小，它会继承父级元素的字体大小，因此并不是一个固定值，任何浏览器的默认字体大小都是16px,因此 12px=0.75em

1. rem：rem相对的只是HTML根元素
2. vh vw 根据窗口的宽高，分成100等份，100vh就表示满高

## 盒模型

标准盒模型和怪异盒模型

content-box，border和padding不计算入width之内（标准盒模型）

padding-box，padding计算入width内

border-box，border和padding计算入width之内，其实就是怪异模式了~

## 未知宽高元素水平垂直居中方法

未知宽高元素水平垂直都居中的实现方法：  
1. 设置元素相对父级定位`position:absolute;left:50%;right:50%`，让自身平移自身高度50% `transform: translate(-50%,-50%);`，这种方式兼容性好，被广泛使用的一种方式

2. 设置元素的父级为弹性盒子`display:flex`，设置父级和盒子内部子元素水平垂直都居中`justify-content:center; align-items:center` ，这种方式代码简洁，但是兼容性ie 11以上支持，由于目前ie版本都已经很高，很多网站现在也使用这种方式实现水平垂直居中

3. 设置元素的父级为网格元素`display: grid`，设置父级和盒子内部子元素水平垂直都居中`justify-content:center; align-items:center` ，这种方式代码简介，但是兼容性ie 10以上支持

4. 设置元素的父级为表格元素`display: table-cell`，其内部元素水平垂直都居中`text-align: center;vertical-align: middle;` ，设置子元素为行内块`display: inline-block; `，这种方式兼容性较好  
  
加分回答

京东的移动端顶部京东logo，使用的是`display:flex`这种模式

## Src和href的区别

Src: 引入资源，资源会嵌入标签坐在的位置，浏览器解析在识别到该位置时，将停止对其他资源的下载和处理,直到该资源下载，编译，执行完毕

Href: 引入资源，资源和该标签将会建立链接关系，浏览器解析到该位置时不会停止文档的加载和处理

## 样式优先级规则

CSS样式的优先级应该分成四大类

第一类`!important`：无论引入方式是什么，选择器是什么，它的优先级都是最高的。

第二类引入方式：行内样式的优先级要高于嵌入和外链，嵌入和外链如果使用的选择器相同就看他们在页面中插入的顺序，在后面插入的会覆盖前面的。

第三类选择器： 选择器优先级：id选择器>（类选择器 | 伪类选择器 | 属性选择器 ）> （后代选择器 | 伪元素选择器 ）> （子选择器 | 相邻选择器） > 通配符选择器 。

第四类继承样式：是所有样式中优先级比较低的。

第五类浏览器默认样式优先级最低。

加分回答 使用!important要谨慎：一定要优先考虑使用样式规则的优先级来解决问题而不是 `!important` -

只有在需要覆盖全站或外部 CSS 的特定页面中使用 `!important` -

永远不要在你的插件中使用 `!important` -

永远不要在全站范围的 CSS 代码中使用 `!important` 优先级的比较指的是相同的样式属性，不同样式属性优先级比较失效，比如：在设置`max-width`时注意，已经给元素的`max-width`设置了`!important`但是还不生效，很有可能就是被width覆盖了 举例：`div`最终的宽度还是`200px` div { max-width: 400px !important; height: 200px;background-color: tomato; width: 200px; }

## 浮动

浮动的作用：

设置浮动的图片，可以实现文字环绕图片，

设置了浮动的块级元素可以排列在同一行，

设置了浮动的行内元素可以设置宽高，同时可以按照浮动设置的方向对齐排列盒子。

设置浮动元素的特点： -

设置了浮动，该元素脱标。元素不占位置 –

浮动可以进行模式转换（行内块元素）

浮动造成的影响，使盒子脱离文档流，如果父级盒子没有设置高度，需要被子盒子撑开，那么这时候父级盒子的高度就塌陷了，同时也会造成父级盒子后面的兄弟盒子布局受到影响。如果浮动元素后面还有其他兄弟元素，其他兄弟元素的布局也会受到影响。

清除浮动的方法： -

伪元素清除浮动：给浮动元素父级增加

.clearfix::after { content: ''; display: table; clear: both; } /\*兼容IE低版本 \*/

.clearfix { \*zoom: 1; } overflow：hidden`：给浮动元素父级增加`overflow：hidden`属性

额外标签法：给浮动元素父级增加标签

加分回答 三种清除浮动的特点和影响 –

伪元素清除浮动：不会新增标签，不会有其他影响，是当下清除浮动最流行的方法

-`overflow：hidden`：不会新增标签，但是如果父级元素有定位元素超出父级，超出部分会隐藏，在不涉及父级元素有超出内容的情况，overflow：hidden比较常用，毕竟写法方便简洁 –

标签插入法：清除浮动的语法加在新增标签上，由于新增标签会造成不必要的渲染，所以这种方法目前不建议使用

## 三栏布局的实现方案

三栏布局，要求左右两边盒子宽度固定，中间盒子宽度自适应，盒子的高度都是随内容撑高的，一般都是中间盒子内容较多，为了保证页面渲染快，在写结构的时候，需要把中间盒子放在左右盒子的前面。 实现三栏布局的方法通常是圣杯布局和双飞翼布局。

圣杯布局的实现方案：三个元素放在同一个父级元素中，代表中间盒子的元素放在最前面，父级盒子设置左右`padding`，三个盒子全部浮动，设置中间盒子宽度100%，左右盒子设置固定宽度，设置左边盒子左边距-100%同时相对自身定位，右边平移自身宽度，右边盒子设置右边距-自身宽度，最后设置父级盒子清除浮动，否则父级盒子的高度无法被撑开

双飞翼布局的实现方案：三个盒子对应三个元素，其中中间盒子套了两层，中间盒子内部盒子设置`margin`,三个盒子全部浮动，设置中间盒子宽度100%，左右盒子设置固定宽度,设置左边盒子左边距-100%,右边盒子设置右边距-自身宽度，最后设置父级盒子清除浮动，否则父级盒子的高度无法被撑开

加分回答

圣杯布局: - 优点：不需要添加dom节点 - 缺点：正常情况下是没有问题的，但是特殊情况下就会暴露此方案的弊端，当middle部分的宽小于left部分时就会发生布局混乱。

双飞翼布局： - 优点：不会像圣杯布局那样变形，CSS样式代码更简洁 - 缺点：多加了一层dom节点

## 页面布局（移动WEB开发布局）

目前市场上常见的移动端开发有单独制作移动端页面和响应式页面两种方案，其中单独制作移动端页面为主流

## 单独制作移动端页面的布局方式

1. **流式布局也称百分比布局，**

流式布局使用百分比宽度来限定布局元素，根据客户端分辨率的大小来合理显示页面。

流式布局主要针对的是宽度布局，且文字大小是固定的

1. **flex布局**

CSS3引入了一种新的布局模式，通过给父盒子添加flex属性，来控制子盒子的位置和排列方式，针对的是宽度

优点：操作方便，布局简单，移动端应用广泛

缺点：pc端浏览器支持情况较差，ie11或更低版本不支持或者仅部分支持

建议：如果是pc端页面布局，建议选择传统布局，传统布局虽然稍微繁琐些但是兼容性好，如果是移动端布局或者不考虑兼容性问题的pc端页面布局，推荐使用flex布局

1. **rem适配布局（less + rem + 媒体查询布局）**
2. **什么是rem适配布局**

rem适配布局可以通过修改html里面的而文字大小来改变页面中元素的大小

rem适配布局，宽和高都能适配，文字也能跟着屏幕的大小而变化。

rem是一个相对单位类似于em，rem的基准是相对于html元素的字体大小，

**2.如何根据不同设备的尺寸来修改html元素的文字的大小**

该过程将使用到媒体查询，媒体查询是C3的新语法可以用来检测设备的宽度，媒体查询可以针对不同的屏幕尺寸设置不同的样式，或者直接根据不同的屏幕尺寸引入不同的css样式文件

@media screen and (min-width: 320px) {

html {

font-size: 320px / @no;

}

}

**3. Less**

Less是css的一种形式的扩展，它引入了变量，运算，以及函数等功能，简化了css的编写降低了css的维护成本

**4. Rem的适配方案**

**方案一：rem + 媒体查询 + less**

一般情况下我们以一套或两套效果图适应大部分的屏幕

动态设置不同屏幕下的html元素字体大小

1.假设设计稿是750px，以这个为基准  
2.假设我们把整个屏幕划分为15等份（划分的份数可自定义）  
3.每一份作为html字体大小，这里就是50px，定义一个变量@baseFont值为 50

4. 页面元素的rem值 =  页面元素值（rem） /   @baseFont 即可

// 我们此次定义的划分的份数 为 15

@no: 15;

// 320

@media screen and (min-width: 320px) {

html {

font-size: 320px / @no;

}

}

5. 样式文件中通过 @import 路径 引入其他的样式文件 它和link相似link是将样式文件引入到html页面里面

**方案二：**

1. 使用flexible.js插件，使用了它我们可以不用书写媒体查询了，它帮助我们查询，只是当屏幕大于750的时候希望插件不要再去重置html文字大小了，所以自己媒体查询设置一下并且要把权重提到最高
2. VScode中安装插件ceerem 他可以直接帮助我们计算出rem

**4. 混合布局**

## 响应式页面兼容移动端

1. **响应式页面布局原理**

就是使用媒体查询针对不同宽度的设备进行布局和样式的设置，从而适配不同设备的目的。

1. **媒体查询是怎么控制页面里面每个元素的大小以及样式**

如果直接通过媒体查询来控制每一个页面元素的大小变化是比较困难的，一般在响应式里面需要准备一个布局容器作为父级元素让父级元素的变化来实现子元素的大小和布局样式的变化，原理就是在不同屏幕下，通过媒体查询来改变布局容器的大小，再改变里面子元素的排列方式和大小，从而实现在不同屏幕下。看到不同的页面布局和样式变化。

布局容器的划分有：超小屏 小屏 中等屏 大屏

1. **Bootstrap前端开发框架**

**Bootstrap 使用四步曲**：

创建文件夹结构

创建 html 骨架结构

引入相关样式文件

书写内容

**栅格系统：**Bootstrap提供了一套响应式、移动设备优先的流式栅格系统，随着屏幕或视口（viewport）尺寸的增加，系统会自动分为最多12列。

栅格系统用于通过一系列的行（row）与列（column）的组合来创建页面布局，你的内容就可以放入这些创建好的布局中。

**响应式工具: .hidden-xs hidden-sm hidden-md hidden-lg**

**双飞翼布局（圣杯布局）**

三列布局，两边定宽，中间自适应

实现原理：中间部分宽度100%，左右预留一个padding值，左右两边定宽通过负边距或者浮动，定位，盒模型等方法分别放到左边和右边

**多兰布局**

对于多栏布局建议使用bootstrap的栅格系统，

C3也可以实现多栏布局标签中填充很多内容，在标签的样式中设置colum样式

Colum-count：3；控制栏数

Colum-rule:1px solid red；控制栏与栏之间

Colum-gap：100px；控制栏与栏之间的间距

Colum-width：600px；控制每一栏的宽度，会和栏数发生冲突，如果设置的宽度大于栏数自适应的宽度的话，会按照所设置的宽度来分割，如果设置的宽度小于

栏数自适应的宽度的话，还是按照栏数来分隔

**瀑布流布局**

瀑布流布局是流式布局的一种他的特点是等宽不等高

如果只是简单的页面展示建议使用colum多栏布局和flex布局来实现瀑布流，如果需要动态

的添加数据或者动态设置列数就需要使用 JS + JQuery 来实现瀑布流

# Axios

## axios的封装

面试官问这个问题的目的是：测试面试者是不是真有项目经验

对axios封装的封装只是做一些简单封装，主要是为了对请求接口进行统一管理，在项目开发过程中如果是多人开发，如果每个人都是直接调用axios进行请求发送那么很可能会出现代码混乱的现象，代码也很可能出现冗余，比如说我们发送请求过程中肯定是会有一个请求的url对吧，一般情况下这个url是相同的，如果每个人都通过直接使用axios进行请求发送那这个url肯定是会有冗余的现象，所以可以针对类似这种情况对axios做一些简单的封装

简单的封装暂时不考虑，取消重复请求，重复发送请求，请求缓存等

取消重复请求：比如页面上有个按钮，点击按钮发送请求，用户可能疯狂点击按钮，会出现上一次请求还没发送，或者请求还未响应又发送了一个请求，通常情况下是需要将这种多次的相同请求给取消掉

重复发送请求：比如一个请求发送失败了，我们可以让发重复几次，这种主要是为了避免因网络波动导致的请求失败

请求缓存：对请求的数据进行缓存，优化页面加载速度

对于axios的封装呢主要是针对几个部分：

1. 实现请求拦截

2. 实现响应拦截

3. 常见错误信息处理

4. 请求头设置

5. api集中管理

具体封装流程：

首先安装axios库，在src文件夹中新建ap文件夹，新建axios.js在文件中导入axios库，导入vue,导入其它（如：store, i18n等），使用axios.create（）函数传入一个对象，对象包含baseUrl, timeout等创建一个请求服务器，将创建的请求服务器挂载到vue原型上，也可以设置一些默认请求头的参数等，请求拦截器对每个请求设置token,响应拦截器对常见错误信息进行处理，

在api文件夹中创建一个request.js，所有请求的api放入的该文件中

## axios的拦截器原理及应用

axios的拦截器的应用场景：

请求拦截器用于在接口请求之前做的处理，比如为每个请求带上相应的参数（token，时间戳等）。

响应拦截器用于在接口返回之后做的处理，比如对返回的状态进行判断（token是否过期）。

axios为开发者提供了这样一个API：拦截器。拦截器分为 请求（request）拦截器和 响应（response）拦截器。

拦截器原理：创建一个chn数组，数组中保存了拦截器相应方法以及dispatchRequest（dispatchRequest这个函数调用才会真正的开始下发请求），把请求拦截器的方法放到chn数组中dispatchRequest的前面，把响应拦截器的方法放到chn数组中dispatchRequest的后面，把请求拦截器和相应拦截器forEach将它们分unshift,push到chn数组中，为了保证它们的执行顺序，需要使用promise，以出队列的方式对chn数组中的方法挨个执行。 加分回答 Axios 是一个基于 promise 的 HTTP 库，可以用在浏览器和 node.js 中。从浏览器中创建 XMLHttpRequests,从 node.js 创建 http 请求,支持 Promise API,可拦截请求和响应，可转换请求数据和响应数据，可取消请求，可自动转换 JSON 数据，客户端支持防御 XSRF

## 创建ajax的过程

创建ajax过程：

1. 创建XHR对象：new XMLHttpRequest()

2. 设置请求参数：request.open(Method, 服务器接口地址);

3. 发送请求: request.send()，如果是get请求不需要参数，post请求需要参数request.send(data)

4. 监听请求成功后的状态变化：根据状态码进行相应的处理。

XHR.onreadystatechange = function () { if (XHR.readyState == 4 && XHR.status == 200) { console.log(XHR.responseText); // 主动释放,JS本身也会回收的 XHR = null; } };

加分回答 POST请求需要设置请求头 readyState值说明

0：初始化,XHR对象已经创建,还未执行open

1：载入,已经调用open方法,但是还没发送请求

2：载入完成,请求已经发送完成

3：交互,可以接收到部分数据

4：数据全部返回 status值说明 200：成功 404：没有发现文件、查询或URl 500：服务器产生内部错误

## 前后端实时通信

保持前后端实时通信的方法有以下几种： 轮询是客户端和服务器之间会一直进行连接，每隔一段时间就询问一次。其缺点也很明显：连接数会很多，一个接受，一个发送。而且每次发送请求都会有Http的Header，会很耗流量，也会消耗CPU的利用率。优点就是实现简单，无需做过多的更改。缺点是轮询的间隔过长，会导致用户不能及时接收到更新的数据；轮询的间隔过短，会导致查询请求过多，增加服务器端的负担 长轮询是对轮询的改进版，客户端发送HTTP给服务器之后，如果没有新消息，就一直等待。有新消息，才会返回给客户端。在某种程度上减小了网络带宽和CPU利用率等问题。由于http数据包的头部数据量往往很大（通常有400多个字节），但是真正被服务器需要的数据却很少（有时只有10个字节左右），这样的数据包在网络上周期性的传输，难免对网络带宽是一种浪费。优点是做了优化，有较好的时效性。缺点是保持连接会消耗资源; 服务器没有返回有效数据，程序超时。 iframe流方式是在页面中插入一个隐藏的iframe，利用其src属性在服务器和客户端之间创建一条长连接，服务器向iframe传输数据（通常是HTML，内有负责插入信息的javascript），来实时更新页面。优点是消息能够实时到达；浏览器兼容好。缺点是服务器维护一个长连接会增加开销；IE、chrome、Firefox会显示加载没有完成，图标会不停旋转。 WebSocket是类似Socket的TCP长连接的通讯模式，一旦WebSocket连接建立后，后续数据都以帧序列的形式传输。在客户端断开WebSocket连接或Server端断掉连接前，不需要客户端和服务端重新发起连接请求。在海量并发和客户端与服务器交互负载流量大的情况下，极大的节省了网络带宽资源的消耗，有明显的性能优势，且客户端发送和接受消息是在同一个持久连接上发起，实时性优势明显。缺点是浏览器支持程度不一致，不支持断开重连。 SSE(Server-Sent Event)是建立在浏览器与服务器之间的通信渠道，然后服务器向浏览器推送信息。SSE 是单向通道，只能服务器向浏览器发送，因为 streaming 本质上就是下载。 优点是SSE 使用 HTTP 协议，现有的服务器软件都支持。SSE 属于轻量级，使用简单；SSE 默认支持断线重连； 加分回答 轮询适用于：小型应用，实时性不高 长轮询适用于：一些早期的对及时性有一些要求的应用：web IM 聊天 iframe适用于：客服通信等 WebSocket适用于：微信、网络互动游戏等 SSE适用于：金融股票数据、看板等

## Fetch请求方式

fetch是一种HTTP数据请求的方式，是XMLHttpRequest的一种替代方案。Fetch函数就是原生js，没有使用XMLHttpRequest对象。

fetch()方法返回一个Promise解析Response来自Request显示状态（成功与否）的方法。

加分回答 XMLHttpRequest的问题 -所有的功能全部集中在一个对象上, 容易书写出混乱而且不容易维护的代码 -采用传统的事件驱动模式, 无法适配新的 Promise API Fetch API的特点 -精细的功能分割: 头部信息, 请求信息, 响应信息等均分布到不同的对象, 更利于处理各种复杂的数据交互场景 -使用Promise API, 更利于异步代码的书写 -同源请求也可以自定义不带 cookie，某些服务不需要 cookie 场景下能少些流量

## 状态码

1. 200：表明请求已经成功，默认情况下状态码为200的响应可以被缓存
2. 404：服务器端无法找到所请求的资源（服务器内为相关接口或者前端接口写错了）
3. 500：服务器端发生错误
4. 502：网关或代理角色的服务器，从上游服务器中接收到的响应式无效的

# 路由

## HashRouter 和 HistoryRouter的区别和原理？

HashRouter和 HistoryRouter的区别：

1. history和hash都是利用浏览器的两种特性实现前端路由，

history是利用浏览历史记录栈的API实现，

hash是监听location对象hash值变化事件来实现

2. history的url没有'#'号，hash反之

1. 相同的url，history会触发添加到浏览器历史记录栈中，hash不会触发，history需要后端配合，如果后端不配合刷新新页面会出现404，hash不需要。

HashRouter的原理：通过`window.onhashchange`方法获取新URL中hash值，再做进一步处理

HistoryRouter的原理：通过`history.pushState `使用它做页面跳转不会触发页面刷新，使用`window.onpopstate` 监听浏览器的前进和后退，再做其他处理

加分回答 hash模式下url会带有#，需要url更优雅时，可以使用history模式。 需要兼容低版本的浏览器时，建议使用hash模式。 需要添加任意类型数据到记录时，可以使用history模式。

-------------------------------------------------------------------

**hash模式：**在浏览器中符号“#”，#以及#后面的字符称之为hash，用window.location.hash读取；  
特点：hash虽然在URL中，但不被包括在HTTP请求中；用来指导浏览器动作，对服务端安全无用，hash不会重加载页面。hash 模式下，仅 hash 符号之前的内容会被包含在请求中，如 [http://www.xxx.com](http://www.xxx.com/)，因此对于后端来说，即使没有做到对路由的全覆盖，也不会返回 404 错误。

**history模式：**history采用HTML5的新特性；且提供了两个新方法：pushState（），replaceState（）可以对浏览器历史记录栈进行修改，以及popState事件的监听到状态变更。  
history 模式下，前端的 URL 必须和实际向后端发起请求的 URL 一致，如 <http://www.xxx.com/items/id>。后端如果缺少对 /items/id 的路由处理，将返回 404 错误。

**Vue-Router 官网里如此描述：**“不过这种模式要玩好，还需要后台配置支持……所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。”

路由导航守卫（<https://juejin.cn/post/6844903641866829838#heading-10>）

Vue-router全局有三个守卫

1. router.beforeEach 全局前置守卫，进入路由之前
2. router.beforeResolve 全局解析守卫（2.5.0+）在beforeRouteEnter调用之后调用
3. router.afterEach 全局后置钩子 进入路由之后，不用代用next

三个参数（to from next）

1. to:：将要进入的路由对象，路由对象指的是平时通过this.$route获取到的路由对象
2. from：将要离开的路由对象
3. next：Function 这个参数是函数，且必须调用，否则不能进入路由（页面空白）

next()进入该路由

next(false)：取消进入路由，url地址重置为from路由地址（也就是将要离开的路由地址）

next(跳转新路由)：当前的导航被中断，重新开始一个新的导航

使用方法：next(‘path地址’)或者next( { path: ‘ ’ } )或者 next( { name: ‘ ’ } )

路由组件内的守卫（路由独享守卫）

1. beforeRouteEnter 进入路由前
2. beforeRouteUpdate （2.2）路由复用同一个组件时
3. beforeRouteLeave离开当前路由时

完整的路由导航解析流程（不包括其他生命周期）

1. 触发进入其他路由
2. 调用要离开路由的组件守卫 beforeRouteLeave
3. 调用全局前置守卫 beforeEach
4. 在重用的的组件里调用 beforeRouteUpdate
5. 调用路由独享守卫 beforeEnter
6. 解析异步路由组件
7. 在将要进入的路由组件中调用 beforeRouteEnter
8. 调用全局解析守卫 beforeResolve
9. 导航被确认
10. 调用全局后置钩子的 afterEach 钩子
11. 触发DOM更新（mounted）
12. 执行beforeRouteEnter守卫中传给next的回调函数

## vue路由懒加载的方法

vue-router 实现懒加载的方法有两种：

ES6的import方式:

component: () => import(/\* webpackChunkName: "about" \*/ '../views/About.vue'),

VUE中的异步组件进行懒加载方式:

component: resolve=>(require(['../views/About'],resolve))

加分回答 vue-router 实现懒加载的作用：性能优化，不用到该路由，不加载该组件。

# 浏览器

## 浏览器如何渲染页面的？

浏览器拿到HTML后，先将HTML转换成dom树，再将CSS样式转换成stylesheet，根据dom树和stylesheet创建布局树，对布局树进行分层，为每个图层生成绘制列表，再将图层分成图块，紧接着光栅化将图块转换成位图，最后合成绘制生成页面。

加分回答

分层的目的：是为了避免对整个页面进行渲染，把页面分成多个图层，尤其是动画的时候，把动画独立出一个图层，渲染时只渲染该图层就ok，transform，z-index等，浏览器会自动优化生成图层

光栅化目的：页面如果很长但是可视区很小，避免渲染非可视区的样式造成资源浪费，所以将每个图层又划分成多个小个子，当前只渲染可视区附近区域

## 浏览器输入URL发生了什么？

输入地址，浏览器查找域名的 IP 地址。

浏览器向 该 IP 地址的web 服务器发送一个 HTTP 请求，在发送请求之前浏览器和服务器建立TCP的三次握手，

判断是否是HTTP缓存，如果是强制缓存且在有效期内，不再向服务器发请求，如果是HTTP协商缓存向后端发送请求且和后端服务器对比，在有效期内，服务器返回304，直接从浏览器获取数据，如果不在有效期内服务器返回200，返回新数据。

请求发送出去服务器返回重定向，浏览器再按照重定向的地址重新发送请求。

如果请求的参数有问题，服务器端返回404，如果服务器端挂了返回500。

如果获取到数据，当浏览器拿到服务器的数据之后，开始渲染页面同时获取HTML页面中图片、音频、视频、CSS、JS，在获取到JS文件之后，会直接执行JS代码，阻塞浏览器渲染，因为渲染引擎和JS引擎互斥，不能同时工作，所以通常把Script标签放在body标签的底部。

渲染过程就是先将HTML转换成dom树，再将CSS样式转换成stylesheet，根据dom树和stylesheet创建布局树，对布局树进行分层，每个图层将生成绘制列表，再将图层分成图块，紧接着光栅化将图块转换成位图，最后合成绘制生成页面。

## http协议

http协议实际上就是客户端和服务器端沟通的规范，

为什么要有这个规范呢，应为网站运用在运行的过程当中服务器端需要知道客户端请求了什么，客户端需要知道服务器端响应了什么

打个比方来说，中国人与中国人沟通使用汉语，俄国人与俄国人沟通使用俄语，那么中国人与俄国人怎么沟通呢这就需要一门通用的双方都能听得懂的语言了，那么客户端和服务器端的进行沟通也是一样的需要一个统一的规范就是值http协议

http协议：就是超文本传输协议它规定了如何从网站服务器传输超文本到本地浏览器，它基于客户端服务器架构工作是客户端和服务器端请求和响应的标准

超文本：就是超级文本，在文本当中呢除了可以包含文字意外，还可以包含图片，音频，视频等等，实际上指的就是html文本

传输协议：实际上就是规定了如何从网站服务器传送html文本到浏览器，http协议呢它基于客户端服务器架构的，是客户端和服务器端进行请求和响应的标准也就是说在http协议当中规定了客户端和服务器端要如何进行沟通

在协议当中呢有报文的概念，所谓报文就是客户端和服务器端在进行请求和响应的过程当中所携带的数据块这个数据块呢实际上就是两者对话的说明以及对话的内容，报文分为请求报文和响应报文

请求报文：就是客户端向服务器端发送请求时所携带的数据块，实际就是客户端向服务器端说的话

响应报文：就是服务器端向客户端进行响应的时候所携带的数据块，就是服务器端向客户端说的话

在这个数据块当中啊包含了请求和响应的相关信息，比如用户在登录的时候输入的用户名和密码就是包含在请求的数据块中进行传递的，那么当前的登录时成功呢还是失败呢这个消息啊就是包含在响应数据块中，报文在传输过程中它还要遵循规定好的一些格式实际上就是以冒号分割的键值对，那么对于开发人员来讲，报文信息在哪里可以看到呢？开发人员可以在开发者工具中的network当中看到，response Headers为响应报文，request Headers为请求报文

比如我们点开request Headers可以看到一些信息都是以冒号分割的键值对，那么客户端和服务器端说了什么呢，

比如第一项accept(接收)，它告诉服务器端当前的客户端它能够接收什么类型的文件（如：html类型的文件，图片，以及其它的文件格式），

比如说accept.language:它表示告诉服务器端当前的客户端它希望能接收到什么语言类型的响应（比如: 中文）

那么在实际开发过程中有哪些需要我们掌握呢

请求报文：

1. 请求方式（get：请求数据， post：发送数据）：用来告诉服务器端当前这个请求要做的这个事情的类型，比如说获取数据的类型我们一般用get，如果是添加数据的请求用post

响应报文：

Http状态码：对于客户端的每次请求呢，服务器端都会进行响应，在响应的时候要告诉客户端当前这次请求到底是成功还是失败，那么这个通知呢就是通过http状态码的形式进行通知的

内容类型：

Accept： 可接受的响应内容类型（Content-Types）  
Accept-Charset： 可接受的字符集  
Accept-Encoding： 可接受的响应内容的编码方式。  
Accept-Language： 可接受的响应内容语言列表。  
Accept-Datetime： 可接受的按照时间来表示的响应内容版本  
Authorization： 用于表示 HTTP 协议中需要认证资源的认证信息  
Cache-Control： 用来指定当前的请求/回复中的，是否使用缓存机制。  
Connection： 客户端（浏览器）想要优先使用的连接类型  
Cookie： 由之前服务器通过Set-Cooki（e 见下文）设置的一个HTTP协议Cookie  
Content-Length： 以 8 进制表示的请求体的长度  
Content-MD5： 请求体的内容的二进制 MD5 散列值（数字签名），以 Base64 编码的结果  
Content-Type： 请求体的 MIME 类型 （用于 POST 和 PUT 请求中）  
Date： 发送该消息的日期和时间（以 RFC 7231 中定义的"HTTP 日期"格式来发送）  
Expect： 表示客户端要求服务器做出特定的行为  
From： 发起此请求的用户的邮件地址  
Host： 表示服务器的域名以及服务器所监听的端口号。如果所请求的端口是对应的服务的标准端口（80），则端口号可以省略。  
If-Match： 仅当客户端提供的实体与服务器上对应的实体相匹配时，才进行对应的操作。主要用于像 PUT 这样的方法中，仅当从用户上次更新某个资源后，该资源未被修改的情况下，才更新该资源。  
If-Modified-Since： 允许在对应的资源未被修改的情况下返回 304 未修改  
If-None-Match： 允许在对应的内容未被修改的情况下返回 304 未修改（ 304 Not Modified ），参考 超文本传输协议 的实体标记  
If-Range： 如果该实体未被修改过，则向返回所缺少的那一个或多个部分。否则，返回整个新的实体  
If-Unmodified-Since： 仅当该实体自某个特定时间以来未被修改的情况下，才发送回应。  
Max-Forwards限制该消息可被代理及网关转发的次数。  
Origin： 发起一个针对跨域资源共享的请求（该请求要求服务器在响应中加入一个 Access-Control-Allow-Origin 的消息头，表示访问控制所允许的来源）。  
Pragma： 与具体的实现相关，这些字段可能在请求/回应链中的任何时候产生。  
Proxy-Authorization： 用于向代理进行认证的认证信息。  
Range： 表示请求某个实体的一部分，字节偏移以 0 开始。  
Referer： 表示浏览器所访问的前一个页面，可以认为是之前访问页面的链接将浏览器带到了当前页面。Referer 其实是 Referrer 这个单词，但 RFC制作标准时给拼错了，后来也就将错就错使用 Referer 了。  
TE： 浏览器预期接受的传输时的编码方式：可使用回应协议头  
Transfer-Encoding： 中的值（还可以使用"trailers"表示数据传输时的分块方式）用来表示浏览器希望在最后一个大小为 0 的块之后还接收到一些额外的字段。  
User-Agent： 浏览器的身份标识字符串  
Upgrade： 要求服务器升级到一个高版本协议。  
Via： 告诉服务器，这个请求是由哪些代理发出的。  
Warning： 一个一般性的警告，表示在实体内容体中可能存在错误。

定义浏览器如何向web服务器发送请求，web服务器如何向浏览器进行响应。简言之，就是client端与server端进行数据传输的一种规则。（）

<https://juejin.cn/post/6987767774076633118>

http工作原理

 分析链接

 浏览器向dns请求分析www.baidu.com的ip

 dns将分析出的112.80.248.75返回给浏览器

 浏览器与服务器建立tcp连接

 浏览器向服务器请求：GET /index.html

 服务器向浏览器回复：index.html内容

 释放tcp连接

 浏览器解析html元素并显示

简言之：**建链->请求->响应->断开**

**Websocket协议**

 websocket是HTML5开始提供的一种在单个tcp连接上进行的全双工的通讯协议；

 websocket api被w3c定为标准

 基于tcp的应用层协议

 建立连接时会使用http协议

 在websocket api中，客户端与服务端只需要一个握手动作，就能在客户端与服务端之间建立一个快速通道，两者之间就可以进行数据的相互传送（可以由双向主动发起）

## 浏览器强缓存和协商缓存的区别

强缓存：不会向服务器发送请求，直接从缓存中读取资源，在chrome控制台的network选项中可以看到该请求返回200的状态码

协商缓存：向服务器发送请求，服务器会根据这个请求的request header的一些参数来判断是否命中协商缓存，如果命中则返回304状态码并带上新的request header通知浏览器从缓存中读取资源

两者的共同点：都是从客户缓存中读取资源，区别是强缓存不会发送请求，协商缓存会发送请求

## 浏览器垃圾回收机制

浏览器垃圾回收机制根据数据的存储方式分为栈垃圾回收和堆垃圾回收。

栈垃圾回收的方式非常简便，当一个函数执行结束之后，JavaScript 引擎会通过向下移动 ESP 来销毁该函数保存在栈中的执行上下文，遵循先进后出的原则。

堆垃圾回收，当函数直接结束，栈空间处理完成了，但是堆空间的数据虽然没有被引用，但是还是存储在堆空间中，需要垃圾回收器将堆空间中的垃圾数据回收。为了使垃圾回收达到更好的效果，根据对象的生命周期不一样，使用不同的垃圾回收的算法。在 V8 中会把堆分为新生代和老生代两个区域，新生代中存放的是生存时间短的对象，老生代中存放的生存时间久的对象。新生区中使用Scavenge算法，老生区中使用标记-清除算法和标记-整理算法。

加分回答 Scavenge算法：

1. 标记：对对象区域中的垃圾进行标记

2. 清除垃圾数据和整理碎片化内存：副垃圾回收器会把这些存活的对象复制到空闲区域中，并且有序的排列起来，复制后空闲区域就没有内存碎片了

3. 角色翻转：完成复制后，对象区域与空闲区域进行角色翻转，也就是原来的对象区域变成空闲区域，原来的空闲区域变成了对象区域，这样就完成了垃圾对象的回收操作，同时这种角色翻转的操作还能让新生代中的这两块区域无限重复使用下去 标记-清除算法： 1. 标记：标记阶段就是从一组根元素开始，递归遍历这组根元素，在这个遍历过程中，能到达的元素称为活动对象，没有到达的元素就可以判断为垃圾数据。 2. 清除：将垃圾数据进行清除。 3. 产生内存碎片：对一块内存多次执行标记 - 清除算法后，会产生大量不连续的内存碎片。而碎片过多会导致大对象无法分配到足够的连续内存。 标记-整理算法 1. 标记：和标记 - 清除的标记过程一样，从一组根元素开始，递归遍历这组根元素，在这个遍历过程中，能到达的元素标记为活动对象。 2. 整理：让所有存活的对象都向内存的一端移动 3. 清除：清理掉端边界以外的内存 V8 是使用副垃圾回收器和主垃圾回收器处理垃圾回收的，不过由于 JavaScript 是运行在主线程之上的，一旦执行垃圾回收算法，都需要将正在执行的 JavaScript 脚本暂停下来，待垃圾回收完毕后再恢复脚本执行。我们把这种行为叫做全停顿。 为了降低老生代的垃圾回收而造成的卡顿，V8 将标记过程分为一个个的子标记过程，同时让垃圾回收标记和 JavaScript 应用逻辑交替进行，直到标记阶段完成，我们把这个算法称为增量标记（Incremental Marking）算法

## 重排区别如何避免

重排 ：当DOM的变化影响了元素的几何信息(元素的的位置和尺寸大小)，浏览器需要重新计算元素的几何属性，将其安放在界面中的正确位置，这个过程叫做重排。

重绘：当一个元素的外观发生改变，但没有改变布局，重新把元素外观绘制出来的过程，所以重绘跳过了创建布局树和分层的阶段。

重排需要重新计算布局树，重绘不需要，重排必定发生重绘，但是涉及到重绘不一定要重排。涉及到重排对性能的消耗更多一些。

触发重排的方法： -页面初始渲染，这是开销最大的一次重排 -添加/删除可见的DOM元素 -改变元素位置 -改变元素尺寸，比如边距、填充、边框、宽度和高度等 -改变元素内容，比如文字数量，图片大小等 -改变元素字体大小 -改变浏览器窗口尺寸，比如resize事件发生时 -激活CSS伪类（例如：`:hover`） -设置 style 属性的值，因为通过设置style属性改变结点样式的话，每一次设置都会触发一次reflow -查询某些属性或调用某些计算方法：offsetWidth、offsetHeight等 避免重排的方式 -样式集中改变 -使用 absolute 或 fixed 脱离文档流 -使用GPU加速:transform 加分回答 GPU的过程是以下这几步 : 1. 获取DOM并将其分割成多个层(renderLayer) 2. 将每个层栅格化，并独立的绘制进位图中 3. 将这些位图作为纹理上传至GPU 4. 复合多个层来生成最终的屏幕图像(最后的layer) 开启了GPU加速的元素被独立出来，不会再影响其他dom的布局，因为它改变之后，只是相当于被贴上了页面。

# 兼容性问题

## 集合类对象问题

问题说明：IE下，可以使用 () 或 [] 获取集合类对象；Firefox下，只能使用 [ ]获取集合类对象。**解决方法：**统一使用 [] 获取集合类对象。

## 自定义属性问题

问题说明：IE下，可以使用获取常规属性的方法来获取自定义属性，也可以使用getAttribute() 获取自定义属性；Firefox下，只能使用getAttribute() 获取自定义属性。  
解决方法：统一通过getAttribute() 获取自定义属性。

png24位的图片在iE6浏览器上出现背景，解决方案是做成PNG8.

浏览器默认的margin和padding不同。解决方案是加一个全局的样式来统一

Chrome 中文界面下默认会将小于12px的文本强制按照12px显示,可通过加入CSS属性-webkit-text-size-adjust: none; 解决。

超链接访问过后hover样式就不出现了，被点击访问过的超链接样式不在具有hover和active了

解决方法是改变CSS属性的排列顺序:L-V-H-A : a:link {} a:visited {} a:hover {} a:active {}

IE6双边距问题，IE6在浮动之后又有横向的margin,此时该元素的外边距是其值的两倍。解决办法：display: block;

IE6浏览器图片下方有空隙。解决办法：给img设置一个display: block;

# 性能优化

## 前端性能优化手段

前端性能优化分为两类，一类是提高文件加载速度，另一类是提高文件渲染速度。

提高文件加载速度的方法： 让传输的数据包更小（压缩文件/图片）：图片压缩和文件压缩 减少网络请求的次数：雪碧图/精灵图、节流防抖

减少渲染的次数：缓存（HTTP缓存、本地缓存、Vue的keep-alive缓存等）

渲染更快的方法：

提前渲染：ssr服务器端渲染 避免渲染阻塞：

CSS放在HTML的head中

JS放在HTML的body底部

避免无用渲染：懒加载

减少渲染次数：对dom查询进行缓存、将dom操作合并、使用减少重排的标签

加分回答 雪碧图的应用场景一般是项目中不常更换的一些固定图标组合在一起，比如logo、搜索图标、切换图标等。 电商项目中最常用到的懒加载，一般在查看商品展示的时候通常下拉加载更多，因为商品数据太多，一次性请求过来数据太大且渲染的时间太长。

--------------------------------------------------------------------------

* content方面
  + 减少HTTP请求：合并文件、CSS精灵、inline Image
  + 减少DNS查询：DNS查询完成之前浏览器不能从这个主机下载任何任何文件。方法：DNS缓存、将资源分布到恰当数量的主机名，平衡并行下载和DNS查询
  + 避免重定向：多余的中间访问
  + 使Ajax可缓存
  + 非必须组件延迟加载
  + 未来所需组件预加载
  + 减少DOM元素数量
  + 将资源放到不同的域下：浏览器同时从一个域下载资源的数目有限，增加域可以提高并行下载量
  + 减少iframe数量
  + 不要404
* Server方面
  + 使用CDN
  + 添加Expires或者Cache-Control响应头
  + 对组件使用Gzip压缩
  + 配置ETag
  + Flush Buffer Early
  + Ajax使用GET进行请求
  + 避免空src的img标签
* Cookie方面
  + 减小cookie大小
  + 引入资源的域名不要包含cookie
* css方面
  + 将样式表放到页面顶部
  + 不使用CSS表达式
  + 不使用IE的Filter
* Javascript方面
  + 将脚本放到页面底部
  + 将javascript和css从外部引入
  + 压缩javascript和css
  + 删除不需要的脚本
  + 减少DOM访问
  + 合理设计事件监听器
* 图片方面
  + 优化图片：根据实际颜色需要选择色深、压缩
  + 优化css精灵
  + 不要在HTML中拉伸图片
  + 保证favicon.ico小并且可缓存

## 性能优化有哪些性能指标，如何量化？

常用的性能优化指标

-Speed Index（lighthouse，速度指数） - TTFB（Network，第一个请求响应时间） - 页面加载时间 - 首次渲染 - 交互动作的反馈时间 - 帧率FPS（动画 ctrl+shift+p） - 异步请求完成时间 使用性能测量工具进行量化 - Chrome DevTools - 开发调试、性能评测 - Audit(Lighthouse) - Throttling 调整网络吞吐 - Performance 性能分析 - Network 网络加载分析 - Lighthouse - 网站整体质量评估 - 还可以提出优化建议 - WebPageTest - 测试多地点(球各地的用户访问你的网站的性能情况) - 全面性能报告（first view,repeat view,waterfall chart 等等） - WebPageTest 还可以进行本地安装，让你的应用在还没上线的时候就可以测试。 加分回答 常用的性能测量API DNS 解析耗时: domnLookupEnd - domnLookupStart TCP 连接耗时: connectEnd - connectStart SSL 安全连接耗时: connectEnd - secureConnectionStart 网络请求耗时 (TTFB): responseStart - requestStart 数据传输耗时: responseEnd - responseStart DOM 解析耗时: domInteractive - responseEnd 资源加载耗时: loadEventStart - domContentLoadedEventEnd First Byte时间: responseStart - domnLookupStart 白屏时间: responseEnd - fetchStart 首次可交互时间: domInteractive - fetchStart DOM Ready 时间: domContentLoadEventEnd - fetchStart 页面完全加载时间: loadEventStart - fetchStart http 头部大小： transferSize - encodedBodySize 重定向次数：performance.navigation.redirectCount 重定向耗时: redirectEnd – redirectStart

## 什么是内存泄漏？什么原因会导致呢？

内存泄露的解释：程序中己动态分配的堆内存由于某种原因未释放或无法释放。

* 根据JS的垃圾回收机制，当内存中引用的次数为0的时候内存才会被回收
* 全局执行上下文中的对象被标记为不再使用才会被释放

内存泄漏的几种场景：

* 全局变量过多。**通常是变量未被定义或者胡乱引用了全局变量**
* 闭包。 未手动解决必包遗留的内存引用。**定义了闭包就要消除闭包带来的副作用**。
* 事件监听未被移除
* 缓存。建议所有缓存都设置好过期时间。

## 如果一个列表有 100000 个数据，这个该怎么进行展示？

我们需要思考的问题：该处理是否必须同步完成？数据是否必须按顺序完成？

解决办法：

（1）将数据分页，利用分页的原理，每次服务器端只返回一定数目的数据，浏览器每次只对一部分进行加载。

（2）使用懒加载的方法，每次加载一部分数据，其余数据当需要使用时再去加载。

（3）使用数组分块技术，基本思路是为要处理的项目创建一个队列，然后设置定时器每过一段时间取出一部分数据，然后再使用定时器取出下一个要处理的项目进行处理，接着再设置另一个定时器。

## SPA（单页应用）首屏加载速度慢怎么解决

什么是首屏加载

首屏时间（First Contentful Paint），指的是浏览器从响应用户输入网址地址，到首屏内容渲染完成的时间，此时整个网页不一定要全部渲染完成，但需要展示当前视窗需要的内容

首屏加载可以说是用户体验中**最重要**的环节

加载慢的原因：

在页面渲染的过程，导致加载速度慢的因素可能如下：

* 网络延时问题
* 资源文件体积是否过大
* 资源是否重复发送请求去加载了
* 加载脚本的时候，渲染内容堵塞了

解决方案：

常见的几种SPA首屏优化方式

* 减小入口文件积
* 静态资源本地缓存
* UI框架按需加载
* 图片资源的压缩
* 组件重复打包
* 开启GZip压缩
* 使用SSR

# 模块化

## 传统开发模式的主要问题

1. 命名冲突

2. 文件依赖

## 模块化解决的问题

**1. 模块化**就是把单独的一个功能封装到一个模块（文件）中，模块之间相互隔离，但是可以通过特定的接口公开内部成 员，也可以依赖别的模块

2. 模块化开发的好处：方便代码的重用，从而提升开发效率，并且方便后期的维护

## 模块化相关规范

**浏览器端模块化规范：**

AMD： Require.js (<http://www.requirejs.cn/>)

CMD: Sea.js (<https://seajs.github.io/seajs/docs/>)

AMD 和 CMD 适用于浏览器端的 Javascript 模块化

**服务器端模块化规范: CommonJS**

1. 模块分为 单文件模块 与 包

2. 模块成员导出：module.exports 和 exports

3. 模块成员导入：require('模块标识符')

CommonJS 适用于服务器端的 Javascript 模块化

**大一统的模块化规范 – ES6模块化**

AMD 和 CMD 适用于浏览器端的 Javascript 模块化, CommonJS 适用于服务器端的 Javascript 模块化, ES6 模块化规范，是浏览器端与服务器端通用的模块化开发规范。

**ES6模块化规范中定义：**

1. 每个 js 文件都是一个独立的模块

2. 导入模块成员使用 import 关键字

3. 暴露模块成员使用 export 关键字

**Node.js 中通过 babel 体验 ES6 模块化**

① npm install --save-dev @babel/core @babel/cli @babel/preset-env @babel/node

② npm install --save @babel/polyfill

③ 项目跟目录创建文件 babel.config.js

④ babel.config.js 文件内容如右侧代码

⑤ 通过 npx babel-node index.js 执行代码

# Webpack

## 当前 Web 开发面临的困境

1. 文件依赖关系错综复杂

2. 静态资源请求效率低

3. 模块化支持不友好

4. 浏览器对高级 Javascript 特性兼容程度较低

5. etc…

## webpack 概述

webpack 是一个流行的前端项目构建工具（打包工具），可以解决当前 web 开发中所面临的困境。

webpack 提供了友好的模块化支持，以及代码压缩混淆、处理 js 兼容问题、性能优化等强大的功能，从而让程序员把 工作的重心放到具体的功能实现上，提高了开发效率和项目的可维护性

目前绝大多数企业中的前端项目，都是基于 webpack 进行打包构建的

## webpack 的基本使用

① 运行 npm install webpack webpack-cli –D 命令，安装 webpack 相关的包

② 在项目根目录中，创建名为 webpack.config.js 的 webpack 配置文件

③ 在 webpack 的配置文件中，初始化如下基本配置：

④ 在 package.json 配置文件中的 scripts 节点下，新增 dev 脚本如下：

⑤ 在终端中运行 npm run dev 命令，启动 webpack 进行项目打包。

webpack 的 4.x 版本中默认约定：

1. 打包的入口文件为 src -> index.js

2. 打包的输出文件为 dist -> main.js

如果要修改打包的入口与出口，可以在 webpack.config.js 中进行配置

## Webpack.config.js

const path = require('path') // // 导入 node.js 中专门操作路径的模块

const htmlWebpackPlugin = require('html-webpack-plugin') // 配置 html-webpack-plugin 生成预览页面

module.exports = {

    mode: 'development', // mode 用来指定构建模式

    entry: path.join(\_\_dirname, './src/main.js'), // 打包入口文件的路径

    output: {

        path: path.join(\_\_dirname, './dist'), // 输出文件的存放路径

        filename: 'bundle.js', // 输出文件的名称

    },

    // 内存中生成页面

    plugins: [ // plugins 数组是 webpack 打包期间会用到的一些插件列表

        new htmlWebpackPlugin({ // 创建插件的实例对象

            template: path.join(\_\_dirname, './src/index.html'), // 指定要用到的模板文件

            filename: 'index.html', // 指定生成的文件的名称，该文件存在于内存中，在目录中不显示

        }),

    ],

    module: {

        rules: [

            { test: /\.css$/, use: ['style-loader', 'css-loader'] },

            { test: /\.scss$/, use: ['style-loader', 'css-loader?modules', 'sass-loader'] },

            { test: /\.(png|gif|bmp|jpg)$/, use: 'url-loader?limit=5000' },

            { test: /\.jsx?$/, use: 'babel-loader', exclude: /node\_modules/ },

        ],

    },

}

## webpack 中的加载器

在实际开发过程中，webpack 默认只能打包处理以 .js 后缀名结尾的模块，其他非 .js 后缀名结 尾的模块，webpack 默认处理不了，需要调用 loader 加载器才可以正常打包，否则会报错！

loader 加载器可以协助 webpack 打包处理特定的文件模块，比如：

1. less-loader 可以打包处理 .less 相关的文件

2. sass-loader 可以打包处理 .scss 相关的文件

3. url-loader 可以打包处理 css 中与 url 路径相关的文件

## 如何使用webpack优化前端性能？

1. 压缩代码。删除多余的代码、注释、简化代码的写法等等方式

2. 利用 CDN 加速。在构建过程中，将引用的静态资源路径修改为 CDN 上对应的路径

3. 删除死代码，将代码中永远不会走到的片段删除掉

4. 优化图片，对于小图可以使用 base64 的方式写入文件中

5. 按照路由拆分代码，实现按需加载，提取公共代码

6. 给打包出来的文件名添加哈希，实现浏览器缓存文件

# 其它

## Import 和 require 的区别

遵循规范：

1. Require是 AMD规范方式引入
2. Import 是es6的语法标准，如果要兼容浏览器的话必须转化为es5的语法

调用时间

1. Require是运行时调用，所以Require理论上可以运用在代码的任何地方
2. Import 是在编译是调用，所以必须放在文件开头

本质

1. Require是赋值过程，其实Require的结果就是对象，数字，字符串，函数等，再把Require的结果赋值给某个变量
2. Import是解构过程，但是目前所有的引擎都还没有实现Import，我们在node中使用babel支持ES6,也仅仅是将ES6转码为ES6再执行，Import语法会被转码为Require