## webpack工作原理

### 1、核心概念

（1）entry：一个可执行模块或者库的入口。

（2）chunk：多个文件组成一个代码块。可以将可执行的模块和他所依赖的模块组合成一个chunk，这是打包。

（3）loader：文件转换器。例如把es6转为es5，scss转为css等

（4）plugin：扩展webpack功能的插件。在webpack构建的生命周期节点上加入扩展hook，添加功能

### 2、webpack构建流程（原理）

从启动构建到输出结果一系列过程：

（1）初始化参数：解析webpack配置参数，合并shell传入和webpack.config.js文件配置的参数，形成最后的配置结果。

（2）开始编译：上一步得到的参数初始化compiler对象，注册所有配置的插件，插件监听webpack构建生命周期的事件节点，做出相应的反应，执行对象的 run 方法开始执行编译。

（3）确定入口：从配置的entry入口，开始解析文件构建AST语法树，找出依赖，递归下去。

（4）编译模块：递归中根据文件类型和loader配置，调用所有配置的loader对文件进行转换，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。

（5）完成模块编译并输出：递归完事后，得到每个文件结果，包含每个模块以及他们之间的依赖关系，根据entry配置生成代码块chunk。

（6）输出完成：输出所有的chunk到文件系统。

注意：在构建生命周期中有一系列插件在做合适的时机做合适事情，比如UglifyPlugin会在loader转换递归完对结果使用UglifyJs压缩覆盖之前的结果

## webpack的loader和plugin区别

【Loader】：用于对模块源码的转换，loader描述了webpack如何处理非javascript模块，并且在buld中引入这些依赖。loader可以将文件从不同的语言（如TypeScript）转换为JavaScript，或者将内联图像转换为data URL。比如说：CSS-Loader，Style-Loader等。

loader的使用很简单：

在webpack.config.js中指定loader。module.rules可以指定多个loader，对项目中的各个loader有个全局概览。

loader是运行在NodeJS中，可以用options对象进行配置。plugin可以为loader带来更多特性。loader可以进行压缩，打包，语言翻译等等。

loader从模板路径解析，npm install node\_modules。也可以自定义loader，命名XXX-loader。

语言类的处理器loader：CoffeeScript，TypeScript，ESNext（Bable）,Sass,Less,Stylus。任何开发技术栈都可以使用webpack。

【Plugin】：目的在于解决loader无法实现的其他事，从打包优化和压缩，到重新定义环境变量，功能强大到可以用来处理各种各样的任务。webpack提供了很多开箱即用的插件：CommonChunkPlugin主要用于提取第三方库和公共模块，避免首屏加载的bundle文件，或者按需加载的bundle文件体积过大，导致加载时间过长，是一把优化的利器。而在多页面应用中，更是能够为每个页面间的应用程序共享代码创建bundle。

## Webpack怎么优化开发环境？

**开发环境常见的问题有**

启动慢。

编译慢，尤其是当项目变大时，修改一行代码要等好几秒甚至十几秒才能看到效果。

占用内存高，当模块数变多时，项目运行占用内存飙升，导致电脑卡顿，影响开发效率。

**优化措施**

使用缓存：可以使用Webpack的缓存功能，将打包结果缓存起来以避免重复构建。可以使用cache-loader或hard-source-webpack-plugin等插件来实现缓存。

使用 DllPlugin：DllPlugin 是 Webpack 的一个插件，它可以将一些不经常变动的第三方库预先打包好，然后在开发过程中直接使用。这样可以减少每次构建时对这些库的重复打包，提高构建速度。

配置合适的SourceMap策略：在开发环境下，开启 SourceMap 可以帮助我们快速定位错误和调试代码。但是开启 SourceMap 会降低构建速度，所以需要权衡是否开启。

多线程并行打包：可以使用thread-loader或happypack开启多线程并行构建，但是并不是一定会提升性能，需要结合场景来自行取舍，比较适合单个耗时长的任务。

配置模块解析：Webpack 在模块解析时会搜索 node\_modules 目录，这个过程比较耗时。为了减少搜索时间，我们可以使用 resolve.alias 配置选项来告诉 Webpack 直接使用特定的路径来查找模块。

使用新技术，比如Webpack5或者Vite这些性能更好的构建工具。

## 如何优化 Webpack 的构建速度？

1. 使用高版本的 Webpack 和 Node.js
2. 多进程/多实例构建：HappyPack(不维护了)、thread-loader
3. 压缩代码

多进程并行压缩

webpack-paralle-uglify-plugin

uglifyjs-webpack-plugin 开启 parallel 参数 (不支持ES6)

terser-webpack-plugin 开启 parallel 参数

通过 mini-css-extract-plugin 提取 Chunk 中的 CSS 代码到单独文件，通过 css-loader 的 minimize 选项开启 cssnano 压缩 CSS。

1. 图片压缩

使用基于 Node 库的 imagemin (很多定制选项、可以处理多种图片格式)

配置 image-webpack-loader

1. 缩小打包作用域：

exclude/include (确定 loader 规则范围)

resolve.modules 指明第三方模块的绝对路径 (减少不必要的查找)

resolve.mainFields 只采用 main 字段作为入口文件描述字段 (减少搜索步骤，需要考虑到所有运行时依赖的第三方模块的入口文件描述字段)

resolve.extensions 尽可能减少后缀尝试的可能性

noParse 对完全不需要解析的库进行忽略 (不去解析但仍会打包到 bundle 中，注意被忽略掉的文件里不应该包含 import、require、define 等模块化语句)

IgnorePlugin (完全排除模块)

合理使用alias

1. 提取页面公共资源：

基础包分离：

使用 html-webpack-externals-plugin，将基础包通过 CDN 引入，不打入 bundle 中

使用 SplitChunksPlugin 进行(公共脚本、基础包、页面公共文件)分离(Webpack4内置) ，替代了 CommonsChunkPlugin 插件

1. DLL：

使用 DllPlugin 进行分包，使用 DllReferencePlugin(索引链接) 对 manifest.json 引用，让一些基本不会改动的代码先打包成静态资源，避免反复编译浪费时间。

HashedModuleIdsPlugin 可以解决模块数字id问题

1. 充分利用缓存提升二次构建速度：

babel-loader 开启缓存

terser-webpack-plugin 开启缓存

使用 cache-loader 或者 hard-source-webpack-plugin

1. Tree shaking

打包过程中检测工程中没有引用过的模块并进行标记，在资源压缩时将它们从最终的bundle中去掉(只能对ES6 Modlue生效) 开发中尽可能使用ES6 Module的模块，提高tree shaking效率

禁用 babel-loader 的模块依赖解析，否则 Webpack 接收到的就都是转换过的 CommonJS 形式的模块，无法进行 tree-shaking

使用 PurifyCSS(不在维护) 或者 uncss 去除无用 CSS 代码

purgecss-webpack-plugin 和 mini-css-extract-plugin配合使用(建议)

1. Scope hoisting

构建后的代码会存在大量闭包，造成体积增大，运行代码时创建的函数作用域变多，内存开销变大。Scope hoisting 将所有模块的代码按照引用顺序放在一个函数作用域里，然后适当的重命名一些变量以防止变量名冲突

必须是ES6的语法，因为有很多第三方库仍采用 CommonJS 语法，为了充分发挥 Scope hoisting 的作用，需要配置 mainFields 对第三方模块优先采用 jsnext:main 中指向的ES6模块化语法

1. 动态Polyfill

建议采用 polyfill-service 只给用户返回需要的polyfill，社区维护。 (部分国内奇葩浏览器UA可能无法识别，但可以降级返回所需全部polyfill)

## vue和react的异同

react和vue大体上是相同的，比如**都使用[虚拟DOM](https://so.csdn.net/so/search?q=%E8%99%9A%E6%8B%9FDOM&spm=1001.2101.3001.7020" \t "/Users/gdj/Documents\\x/_blank)高效的更新视图，提倡组件化，数据驱动视图，diff算法，对diff算法进行了优化，有router库实现url到组件的映射，有状态管理**等等

### 组件化的异同

相同点：

react和vue都推崇组件化，通过将页面拆分成一个一个小的可复用单元来提高代码的复用率和开发效率。在开发时react和vue有相同的套路，比如都有父子组件传参，都有数据状态管理，都有前端路由等。

不同点：

React推荐的做法是JSX + inline style, 也就是把 HTML 和 CSS 全都写进 JavaScript 中,即 all in js;

Vue 推荐的做法是 template 的单文件组件格式(简单易懂，从传统前端转过来易于理解),即 html,css,JS 写在同一个文件(vue也支持JSX写法)

### 2.虚拟DOM异同

相同点：

Vue与React都使用了 Virtual DOM + Diff算法， 不管是Vue的Template模板+options api 写法， 还是React的Class或者Function写法,最后都是生成render函数，而render函数执行返回VNode(虚拟dom树, 虚拟DOM的数据结构，本质上是棵树)。

当每一次UI更新时，总会根据render重新生成最新的虚拟dom树，然后跟以前缓存起来老的虚拟dom树进行比对，再使用Diff算法（框架核心）去真正更新真实DOM（虚拟DOM是JS对象结构，同样在JS引擎中，而真实DOM在浏览器渲染引擎中，所以操作虚拟DOM比操作真实DOM开销要小的多）

不同点：

**react函数式组件思想，在发生数据（setState）更改后，会重新生成新的虚拟dom树，然后进行新旧虚拟dom树的diff（自顶向下的全量diff）**

**vue是组件响应思想，采用代理监听（watcher）数据，当一个组件内数据更改，可以明确知道并响应这个组件进行diff比较（局部订阅）**

#### diff算法源码异同

在处理老节点部分，都需要把节点处理 key - value 的 Map 数据结构，方便在往后的比对中可以快速通过节点的 key 取到对应的节点。同样在比对两个新老节点是否相同时，key 是否相同也是非常重要的判断标准。所以不管是 React, 还是 Vue，在写动态列表的时候，都需要设置一个唯一值 key，这样在 diff 算法处理的时候性能才最大化。

不同点：react 会自顶向下全diff。vue会跟踪每一个组件的依赖关系,不需要重新渲染整个组件树。

在react中，当状态发生改变时，组件树就会自顶向下的全diff, 重新render页面， 重新生成新的虚拟dom tree, 新旧dom tree进行比较， 进行patch打补丁方式，局部更新dom。所以react为了避免父组件更新而引起不必要的子组件更新， 可以在shouldComponentUpdate做逻辑判断，减少没必要的render， 以及重新生成虚拟dom，做差量对比过程。

在vue中， 通过Object.defineProperty 把 data 属性全部转为 getter/setter。同时watcher实例对象会在组件渲染时，将属性记录为dep, 当dep 项中的 setter被调用时，通知watch重新计算，使得关联组件更新。（vue中的diff是用双端比较的方式实现的，但是vue2与vue3有区别）

### 3.数据驱动视图

#### vue中的数据驱动视图

Vuejs的数据驱动是通过MVVM这种框架来实现的。MVVM框架主要包含3个部分:model、view和 viewModel。

* Model:指的是数据部分，对应到前端就是javascript对象
* View:指的是视图部分，对应前端就是dom
* ViewModel:就是连接视图与数据的中间件

ViewModel是实现数据驱动视图的核心，当数据变化的时候，ViewModel能够监听到这种变化，并及时的通知view做出修改。同样的，当页面有事件触发时，ViewModel也能够监听到事件，并通知model进行响应。ViewModel就相当于一个观察者，监控着双方的动作，并及时通知对方进行相应的操作。

首先，vuejs在实例化的过程中，会对遍历传给实例化对象选项中的data 选项，遍历其所有属性并使用 Object.defineProperty 把这些属性全部转为 getter/setter。

同时每一个实例对象都有一个watcher实例对象，他会在模板编译的过程中,用getter去访问data的属性，watcher此时就会把用到的data属性记为依赖，这样就建立了视图与数据之间的联系。当之后我们渲染视图的数据依赖发生改变（即数据的setter被调用）的时候，watcher会对比前后两个的数值是否发生变化，然后确定是否通知视图进行重新渲染。这样就实现了所谓的数据对于视图的驱动。

#### react的数据驱动视图

首先了解一些列内容：

* pending：当前所有等待更新的state队列。
* isBatchingUpdates：React中用于标识当前是否处理批量更新状态，默认false。
* dirtyComponent：当前所有待更新state的组件队列。

React通过setState实现数据驱动视图，通过setState来引发一次组件的更新过程从而实现页面的重新渲染(除非shouldComponentUpdate返回false)。

* setState()首先将接收的第一个参数state存储在pending队列中；（state）
* 判断当前React是否处于批量更新状态，是的话就将需要更新state的组件添加到dirtyComponents中；（组件）
* 不是的话，它会遍历dirtyComponents的所有组件，调用updateComponent方法更新每个dirty组件（开启批量更新事务）

## React调用 setState 之后发生了什么？

在代码中调用 setState 函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发所谓的调和过程（Reconciliation）。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个 UI 界面。在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

## shouldComponentUpdate 是做什么的，（react 性能优化是哪个周期函数？）

shouldComponentUpdate 这个方法用来判断是否需要调用 render 方法重新描绘 dom。因为 dom 的描绘非常消耗性能，如果我们能在 shouldComponentUpdate 方法中能够写出更优化的 dom diff 算法，可以极大的提高性能。

## 为什么虚拟 dom 会提高性能?(必考)

虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 dom diff 算法避免了没有必要的 dom 操作，从而提高性能。

用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异把 2 所记录的差异应用到步骤 1 所构建的真正的 DOM 树上，视图就更新了。

## react diff 原理（常考，必考）

* 把树形结构按照层级分解，只比较同级元素。
* 给列表结构的每个单元添加唯一的 key 属性，方便比较。
* React 只会匹配相同 class 的 component（这里面的 class 指的是组件的名字）
* 合并操作，调用 component 的 setState 方法的时候, React 将其标记为 dirty.到每一个事件循环结束, React 检查所有标记 dirty 的 component 重新绘制.
* 选择性子树渲染。开发人员可以重写 shouldComponentUpdate 提高 diff 的性能。

## React组件怎样可以返回多个组件

* 使用HOC（高阶函数）
* 使用React.Fragment,可以让你将元素列表加到一个分组中，而且不会创建额外的节点（类似vue的template)
* 使用数组返回

## React性能优化手段

1. shouldComponentUpdate
2. memo
3. getDerviedStateFromProps
4. 使用Fragment
5. 循环元素使用正确的key
6. 拆分尽可能小的可复用组件，ErrorBoundary
7. 使用React.lazy和React.Suspense延迟加载不需要立马使用的组件

## 什么是useCallback和useMemo？

useCallback 和 useMemo 都是react可用于性能优化的内置hooks。

两者的区别在于：useCallback缓存的是一个函数，而useMemo缓存的是计算结果。

### 为什么使用useCallback和useMemo?

在函数式组件中，每次UI的变化，都是通过重新执行整个函数来完成的，这和传统的类组件有很大区别：函数组件中并没有一个直接的方式在多次渲染之间维持一个状态。

在重新执行整个函数组件的过程中，其中的函数和引用类型的变量会创建新的（指向新的引用），导致函数组件在re-render前后，其中函数和引用类型变量是不相等的，这又会导致其他非必要的re-render。

### 什么时候使用useCallback和useMemo?

#### useCallback

当子组件接收一个函数props时，一般会使用useCallback来缓存这个函数，减少不必要的re-render。以下例子：向子组件传递一个函数，在父组件每次re-render的时候，函数会重新创建新的，这会导致使用这个函数props的子组件也re-render，但这是不必要的，可以用useCallback来解决。

#### useMemo

useMemo常用在以下两种场景的优化中：

1. 引用类型的变量
2. 需要大量时间执行的计算函数。

useMemo 和 useCallback 接收的参数都是一样,第一个参数为回调 第二个参数为要依赖的数据

### 异同点

共同作用：

1.仅仅 依赖数据 发生变化, 才会重新计算结果，也就是起到缓存的作用。

两者区别：

1.useMemo 计算结果是 return 回来的值, 主要用于 缓存计算结果的值 ，应用场景如： 需要 计算的状态

2.useCallback 计算结果是 函数, 主要用于 缓存函数，应用场景如: 需要缓存的函数，因为函数式组件每次任何一个 state 的变化 整个组件 都会被重新刷新，一些函数是没有必要被重新刷新的，此时就应该缓存起来，提高性能，和减少资源浪费。

注意： 不要滥用会造成性能浪费，react中减少render就能提高性能，所以这个仅仅只针对缓存能减少重复渲染时使用和缓存计算结果。

### 总结

useMemo返回的是一个变量的值，useCallback返回的是一个函数。对应到function组件最后要return的html代码部分，useMemo就是作为一个值来使用的，而useCallback则是被绑定到onClick上，作为要执行的函数。这就是它俩的本质区别。

useCallback(fn, deps) 等价于 useMemo(() => fn, deps).

## setState是同步还是异步？

**v18之前：**

React是希望setState表现为异步的，因为批量更新可以优化性能。因此**在React能够管控到的地方，比如生命周期钩子和合成事件回调函数内，表现为异步**。

**在定时器和原生事件里，因为React管控不到，所以表现为同步。**

在某些情况下，我们需要立即获取更新后的状态，这时可以使用第二个可选参数callback，在状态更新后立即执行回调函数来获取更新后的状态。例如：

this.setState({ counter: this.state.counter + 1 }, () => {

console.log(this.state.counter); // 输出更新后的值

});

**v18之后：**

React18之后，默认所有的操作都放到批处理中，因此setState不管在那儿调用都是异步的了。

如果希望同步更新，可以使用flushSync这个API。

## 使用Hooks有踩过哪些坑？

useEffect中没有正确设置依赖数组导致死循环。

useEffect中没有清除副作用导致内存泄漏。

在条件语句和循环中使用Hooks导致报错。

闭包陷阱。

## React逻辑复用方式有哪些？

Mixin：有很多缺点，已被弃用，可以不考虑。

HOC(高阶组件)：高阶组件是一个函数，它接收一个组件作为参数并返回一个新的组件。高阶组件可以将一些通用的逻辑（如：数据获取、权限验证、错误处理等）封装到一个函数中，并将其作为高阶组件的参数传递给其他组件使用，HOC一般以withXxx命名，并可以结合装饰器优雅地使用。

Render Props：通过在组件中传递一个函数作为prop，该函数将用于渲染组件的内容。这个函数可以接收组件需要的数据和方法，并返回React元素。

Hooks：自定义Hooks，将通用逻辑封装到useXxx函数中，可以在多个组件内使用，常见的像数据请求、表单、防抖节流、拖拽等。

## useRef有什么作用？

获取DOM。

存储上一次渲染的值，可以用useRef创建一个对象来存储setState前的旧值。

## React代码层面有哪些性能优化的方式？

React.memo()：可以缓存组件的渲染结果，避免不必要的重渲染。它接受一个函数组件，并返回一个新的组件，新组件将只在props发生变化时才重新渲染。

useMemo和useCallback。

shouldComponentUpdate：在类组件中，可以通过实现shouldComponentUpdate()方法来判断组件是否需要重新渲染。SCU接收两个参数：nextProps和nextState，我们可以在这个方法中比较当前props和state与下一个props和state的变化来决定是否需要重新渲染组件。

使用React.lazy()和Suspense进行组件懒加载。

对于大型应用，可以使用不可变数据的三方库比如Immer.js结合shouldComponentUpdate来做性能优化。

## 原型链

每个对象（object）都有一个私有属性指向另一个名为原型（prototype）的对象

js分为函数对象和普通对象，每个对象都有\_\_proto\_\_属性，但是只有函数对象才有prototype属性

Object、Function都是js内置的函数, 类似的还有我们常用到的Array、RegExp、Date、Boolean、Number、String

属性\_\_proto\_\_是一个对象，它有两个属性，constructor和\_\_proto\_\_；

原型对象prototype有一个默认的constructor属性，用于记录实例是由哪个构造函数创建；

js原型、原型链默认遵从以下两个准则

* 准则1：原型对象（即Person.prototype）的constructor指向构造函数本身
* 准则2：实例（即per0）的\_\_proto\_\_和原型对象指向同一个地方

## 作用域

js中的作用域说的是变量的可访问性和可见性。也就是说整个程序中哪些部分可以访问这个变量，或者说这个变量都在哪些地方可见。

* 全局作用域
* 函数作用域
* 块级作用域

## 作用域链

当执行一段JavaScript代码（全局代码或函数）时，JavaScript引擎会为其创建一个作用域又称为执行上下文（Execution Context），在页面加载后会首先创建一个全局的作用域，然后每执行一个函数，会建立一个对应的作用域，从而形成了一条**作用域链**。**每个作用域都有一条对应的作用域链，链头是全局作用域，链尾是当前函数作用域。**

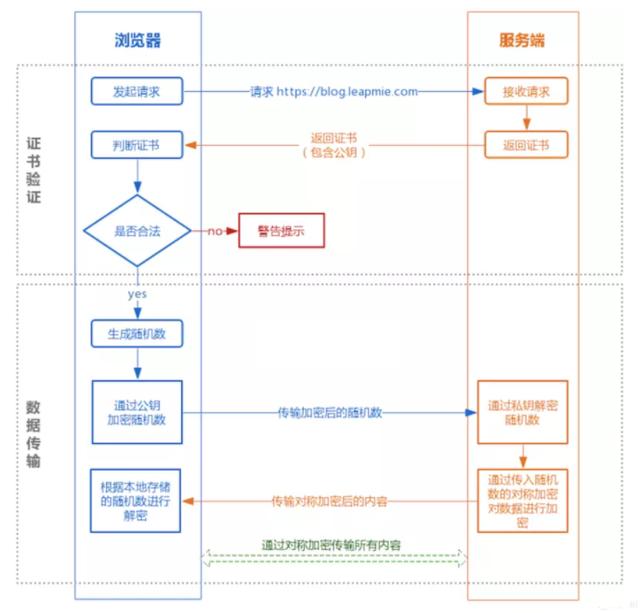
作用域链的作用是用于**解析标识符**，当函数被创建时（不是执行），会将this、arguments、命名参数和该函数中的所有局部变量添加到该当前作用域中，当JavaScript需要查找变量X的时候（这个过程称为变量解析），它首先会从作用域链中的链尾也就是当前作用域进行查找是否有X属性，如果没有找到就顺着作用域链继续查找，直到查找到链头，也就是全局作用域链，仍未找到该变量的话，就认为这段代码的作用域链上不存在x变量，并抛出一个引用错误（ReferenceError）的异常。

* 作用域分为全局作用域和局部作用域。
* 作用域其实就是规定了当前作用域中的变量和函数可被作用的范围。
* 作用域链其实就是规定了变量和函数的查找规则、是当前执行上下文的变量对象以及所有父级执行上下文的变量对象的集合。

当查找一个变量时，先从作用域的顶端查找，一直查找到作用域的底端，若查找完仍未找到，抛出错误。

## HTTPS工作流程

HTTPS协议是安全的，因为HTTPS协议会对传输的数据进行加密，而加密过程是使用了非对称加密实现。但是HTTPS在内容传输使用的是对称加密，在证书验证阶段使用非对称加密。所以HTTPS加密方式是：对称加密 + 非对称加密混合。



① 证书验证阶段：

1）浏览器发起 HTTPS 请求；

2）服务端返回 HTTPS 证书；

3）客户端验证证书是否合法，如果不合法则提示告警。

② 数据传输阶段：

1）当证书验证合法后，在本地生成随机数；

2）通过公钥加密随机数，并把加密后的随机数传输到服务端；

3）服务端通过私钥对随机数进行解密；

1. 服务端通过客户端传入的随机数构造对称加密算法，对返回结果内容进行加密后传输。

## 网页前端性能优化的方式有哪些？

1.压缩 css, js, 图片

2.减少 http 请求次数， 合并 css、js 、合并图片（雪碧图）

3.使用 CDN

4.减少 dom 元素数量

5.图片懒加载

6.静态资源另外用无 cookie 的域名

7.减少 dom 的访问（缓存 dom）

8.巧用事件委托

9.样式表置顶、脚本置低

## 移动端性能优化？

尽量使用css3动画，开启硬件加速

适当使用touch事件代替click时间

避免使用css3渐变阴影效果

可以用transform: translateZ(0) 来开启硬件加速

不滥用float。float在渲染时计算量比较大，尽量减少使用

不滥用web字体。web字体需要下载，解析，重绘当前页面

合理使用requestAnimationFrame动画代替setTimeout

css中的属性（css3 transitions、css3 3D transforms、opacity、webGL、video）会触发GUP渲染，耗电

## 节流与防抖的原理和应用场景

节流：就是限制一个动作在一段时间内只能执行一次

防抖：就是 当一个动作连续触发，只执行最后一次

/\*\*

\* 防抖

\*

\* @export

\* @param {\*} handle

\* @param {\*} delay

\* @returns

\*/

export function debounce(handle, delay) {

var timer = null;

return function () {

var \_self = this,

\_args = arguments;

clearTimeout(timer);

timer = setTimeout(function () {

handle.apply(\_self, \_args)

}, delay)

}

}

/\*\*

\* 节流

\*

\* @export

\* @param {\*} handler

\* @param {\*} wait

\* @returns

\*/

export function throttle(handler, wait) {

var lastTime = 0;

return function (e) {

var nowTime = new Date().getTime();

if (nowTime - lastTime > wait) {

handler.apply(this, arguments);

lastTime = nowTime;

}

}

}

## 浏览器的缓存

**流程**：浏览器中的缓存主要分为强缓存和协商缓存。

1. 浏览器在加载资源时，根据请求头的 expires 和 cache-control 判断是否命中强缓存，是则直 接从缓存读取资源，不会发请求到服务器。
2. 如果没有命中强缓存，浏览器一定会发送一个请求到服务器，通过 last-modified 和 etag 验证 资源是否命中协商缓存，如果命中，服务器会将这个请求返回，但是不会返回这个资源的数据，依 然是从缓存中读取资源。
3. 如果前面两者都没有命中，直接从服务器加载资源。

### 缓存机制

浏览器发送请求前，根据请求头的expires和cache-control判断是否命中（包括是否过期）强缓存策略，如果命中，直接从缓存获取资源，并不会发送请求。如果没有命中，则进入下一步。 2. 没有命中强缓存规则，浏览器会发送请求，根据请求头的last-modified和etag判断是否命中协商缓存，如果命中，直接从缓存获取资源。如果没有命中，则进入下一步。 3. 如果前两步都没有命中，则直接从服务端获取资源。

### 2、强缓存

强缓存：不会向服务器发送请求，直接从缓存中读取资源。

#### 2.1 强缓存原理

强制缓存就是向浏览器缓存查找该请求结果，并根据该结果的缓存规则来决定是否使用该缓存结果的过程，强制缓存的情况主要有三种(暂不分析协商缓存过程)，如下：

* 第一次请求，不存在缓存结果和缓存标识，直接向服务器发送请求
* 存在缓存标识和缓存结果，但是已经失效，强制缓存是啊比，则使用协商缓存（暂不分析）
* 存在该缓存结果和缓存标识，且该结果尚未失效，强制缓存生效，直接返回该结果

那么强制缓存的缓存规则是什么？ 当浏览器向服务器发起请求时，服务器会将缓存规则放入HTTP响应报文的HTTP头中和请求结果一起返回给浏览器，控制强制缓存的字段分别是Expires和Cache-Control，其中Cache-Control优先级比Expires高。

### 协商缓存

协商缓存就是强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程，主要有以下两种情况：

* 协商缓存生效，返回304和Not Modified
* 协商缓存失效，返回200和请求结果

## cookie、sessionStorage、localStorage

### cookie

* 会话性：默认不设置过期时间，保存在内存中；关闭浏览器、用户手动清除 则cookie失效
* 持久性：设置了过期时间，保存在硬盘中；过期时间到、用户手动清除 则cookie失效
* 同一浏览器下的同一个域名下共享(同一浏览器的不同网页窗口共享)
* 单个4KB
* 每次请求都会携带在http请求头中，存储过多cookie会造成性能的浪费
* 可用于登录识别用户

### session

* 关闭浏览器的网页窗口、用户手动清除 则失效
* 同一浏览器的不同网页窗口不共享，只在当前网页窗口有效
* 单个5MB
* 仅在客户端（即浏览器）中保存，不参与和服务器的通信

### localStorage

* 长期有效，不管是关闭网页窗口还是浏览器都不会失效，只能手动清除
* 单个5MB
* 仅在客户端（即浏览器）中保存，不参与和服务器的通信

### 总结

| **类型** | **时效性** | **共享性** | **大小** | **参与** |
| --- | --- | --- | --- | --- |
| cookie | 设置过期时间，时间到则失效；未设置过期时间，浏览器关闭失效 | 同域名下共享 | 单个4KB，一般可存储50个，Opera为30个 | 参与，携带在http请求头中 |
| sessionStorage | 网页窗口关闭则失效 | 只在当前网页窗口中可访问 | 单个5MB | 不参与 |
| localStorage | 永久存储，除非手动清除 | 在特定页面协议内访问 | 单个5MB | 不参与 |

从**数据存储位置、生命周期、存储大小、写入方式、数据共享、发送请求时是否携带、应用场景**这几个方面来回答。

1.存储位置：Cookie、SessionStorage、 LocalStorage都是浏览器的本地存储。 它们的共同点：都是存储在浏览器本地的，它们的区别：cookie是由服务器端写入的，而SessionStorage、 LocalStorage都是由前端写入的，

2.生命周期：cookie的生命周期是由服务器端在写入的时候就设置好的，LocalStorage是写入就一直存在，除非手动清除，SessionStorage是页面关闭的时候就会自动清除。

3.存储大小：cookie的存储空间比较小大概4KB，SessionStorage、LocalStorage存储空间比较大，大概5M。

4.数据共享：Cookie、SessionStorage、LocalStorage数据共享都遵循同源原则，SessionStorage还限制必须是同一个页面。

5.是否携带：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

总结：由于它们的以上区别，所以它们的应用场景也不同，Cookie一般用于存储登录验证信息SessionID或者token，LocalStorage常用于存储不易变动的数据，减轻服务器的压力，SessionStorage可以用来检测用户是否是刷新进入页面，如音乐播放器恢复播放进度条的功能。

## AST抽象语法树

概念

AST 全称为 Abstract Syntax Tree，译为抽象语法树。在 JavaScript 中，任何一个对象（变量、函数、表达式等）都可以转化为一个抽象语法树的形式。抽象语法树本质就是一个树形结构的对象。

用途

* 常用各类转义、编译的插件中。比如最典型的 ES6 转换为 ES6 工具 、JSX 语法转换为 JavaScript 语法。即 babel 模块。
* 代码语法的检查，比如代码规范工具 ESLint 模块。
* 各类 JS/CSS/HTML 压缩工具。
* 代码的格式化、高亮。
* 代码错误提示。
* 代码自动补全。

## 闭包问题

得分点： 变量背包、作用域链、局部变量不销毁、函数体外访问函数的内部变量、内存泄漏、内存溢出、形成块级作用域、柯里化、构造函数中定义特权方法、Vue中数据响应式Observer

标准回答：

**一个函数和词法作用域的引用捆绑在一起，这样的组合**就是闭包（closure）。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。

### 应用场景

1. 函数防抖
2. 使用闭包设计单例模式
3. 为多个组件独立属性
4. 设置私有变量
5. 拿到正确的值

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

setTimeout(function(){

console.log(i)//10个10

},1000)

}

// 正确做法

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

((j)=>{

setTimeout(function(){

console.log(j)//1-10

},1000)})(i)

}

## 匿名函数

匿名函数是没有名称的函数。

匿名函数可以用作其他函数的参数或作为立即调用的函数执行。

(function() {

console.log(111);

})();

(function(s) {

console.log(s);

})(‘222’);

### 应用场景

1. **事件**

btn.onclick = function () { //给按钮增加点击事件

alert("123");

}

1. **赋值给变量**

var fn = function () {

console.log(123);

}

fn() // 123

1. **回调函数 （比如定时器的回调函数）**

setInterval(function () {

console.log('我是一个回调函数，每次1秒钟会被执行一次');

}, 1000)

**4 返回值 （return一个匿名函数）**

function fn() {

return function () {

console.log(123);

}

}

fn()() // 123

### 作用

1、实现闭包

2、模拟块级作用域，减少全局变量，降低命名冲突

3、执行完匿名函数，存储在内存中的相关变量会被销毁，从而节省内存

## ajax、axios 前后端通信的过程（原生的ajax怎么写）

原生ajax请求步骤get：

1.创建核心的对象（new xhrhttprequest对象）

2.编写回调函数（判断是否请求成功 状态码等于4或者等于200）

3.调用open方法 设置请求方式和请求路径

4.调用send方法 发送

post：要多设置一个请求头setrequestheader（在发送的前面）

## map和forEach的异同

### 相同点

* 都只能遍历数组
* 都是循环遍历数组中的每一项。
* 每次执行匿名函数都支持三个参数，参数分别为item(当前每一项)，index(索引值)，arr(原数组)。
* 匿名函数中的this都是指向window。

不同点

* forEach()方法没有返回值，会更改原数组。
* map() 有返回值，返回一个新数组，不会改变原数组，map() 不会对空数组进行检测。
* 总的来说 map 的速度大于 forEach()
* 性能上来说for > forEach > map
* forEach返回undefined，map会返回新的数组。
* forEach没办法中止循环，但是map可以通过返回false或者出错来中止。

## JavaScript事件循环

JavaScript作为一门单线程语言，如何通过**事件循环（Event Loop）和任务队列（Task Queue）**的机制，高效地处理异步任务，保证用户体验的流畅性。

### 事件循环

JavaScript是单线程执行模型，执行的时候将会区分为**主线程和任务队列**。主线程执行完毕，会从任务队列中读取新的任务放入主线程进行执行，这个读取过程是**循环读取**，所以也叫**事件循环**。

任务队列分为宏任务和微任务，同层次，**先执行微任务，再执行宏任务**。

在事件循环中，当主线程执行完当前的同步任务后，会检查事件队列中是否有待处理的事件。如果有，主线程会取出事件并执行对应的回调函数。这个循环的过程被称为事件循环（Event Loop），它由主线程和任务队列两部分组成。主线程负责执行同步任务，而异步任务则通过任务队列进行处理。这种机制保证了异步任务在适当的时机能够插入执行，从而实现了JavaScript的非阻塞异步执行。

#### 事件循环流程如下：

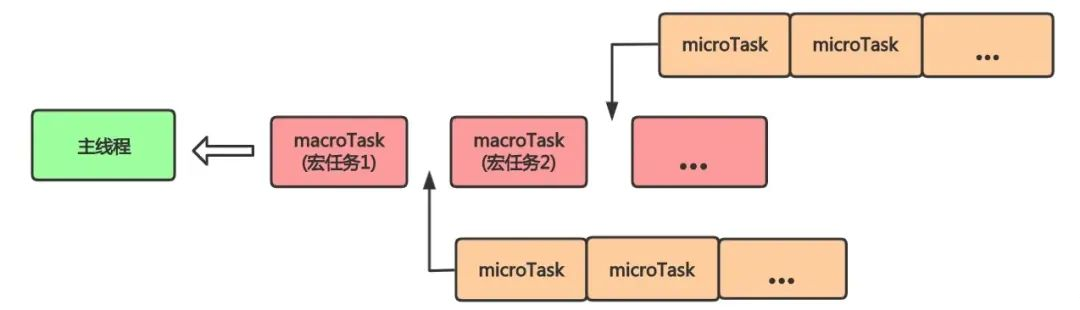
主线程读取JavaScript代码，形成相应的堆和执行栈。

当主线程遇到异步任务时，将其委托给对应的异步进程（如Web API）处理。

异步任务完成后，将相应的回调函数推入任务队列。

主线程执行完同步任务后，检查任务队列，如果有任务，则按照先进先出的原则将任务推入主线程执行。

重复执行以上步骤，形成事件循环。



#### 同步任务

同步任务是按照代码的书写顺序一步一步执行的任务。当主线程执行同步任务时，会阻塞后续的代码执行，直到当前任务执行完成。典型的同步任务包括函数调用、变量赋值、算术运算等。

#### 异步任务

异步任务是在主线程执行的同时，通过回调函数或其他机制委托给其他线程或事件来处理的任务。在执行异步任务时，主线程不会等待任务完成，而是继续执行后续代码。包括：

回调函数 callback

Promise/async await

Generator

事件监听

发布/订阅

计时器

requestAnimationFrame

MutationObserver

process.nextTick

I/O操作

不得不说，异步执行的机制使得 JavaScript 能够更好地处理耗时操作，保持页面的响应性。

### 任务队列

上面我们讨论了同步任务和异步任务的执行过程，接下来我们将进一步探讨任务队列，了解它的最小颗粒度是如何执行的。

在JavaScript中，任务队列是**一个执行异步任务的机制**，由浏览器内部的“任务队列管理器”管理。JavaScript中的任务分为两类：同步任务和异步任务。同步任务会在当前的执行栈中执行，而异步任务会被添加到任务队列中，等待当前的同步任务执行完毕之后再执行。

#### 任务队列类型

任务队列分为宏任务队列（macrotask queue）和微任务队列（microtask queue）两种。JavaScript 引擎遵循事件循环的机制，在执行完当前宏任务后，会检查微任务队列，执行其中的微任务，然后再取下一个宏任务执行。这个过程不断循环，形成事件循环。

#### 宏任务和微任务

事件循环由宏任务和在执行宏任务期间产生的所有微任务组成。完成当下的宏任务后，会立刻执行所有在此期间入队的微任务。

这种设计是为了给紧急任务一个插队的机会，否则新入队的任务永远被放在队尾。区分了微任务和宏任务后，本轮循环中的微任务实际上就是在插队，这样微任务中所做的状态修改，在下一轮事件循环中也能得到同步。

常见的**宏任务**有：

script（整体代码）

setTimout

setInterval

setImmediate(node 独有)

requestAnimationFrame(浏览器独有)

IO

UI render（浏览器独有）

常见的**微任务**有：

process.nextTick(node 独有)

Promise.then()

Object.observe

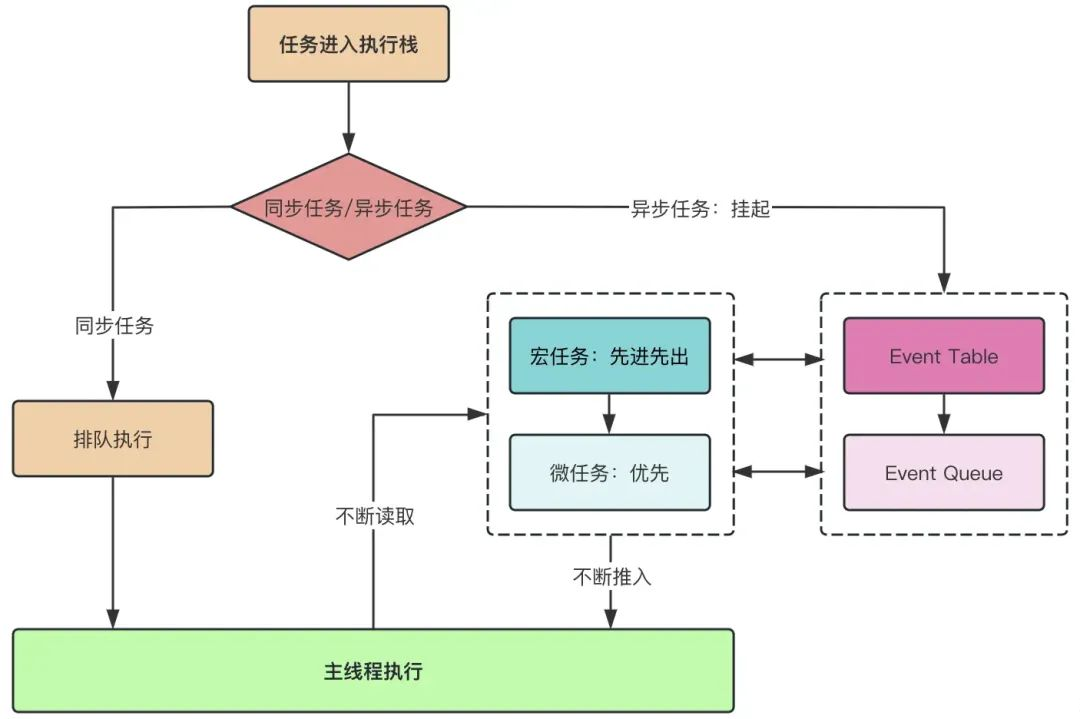
MutationObserver

Generator函数

#### 任务执行过程

首先，必须要明确，在JavaScript中，所有任务都在主线程上执行。任务执行过程分为同步任务和异步任务两个阶段。异步任务的处理经历两个主要阶段：Event Table（事件表）和 Event Queue（事件队列）。

Event Table存储了宏任务的相关信息，包括事件监听和相应的回调函数。当特定类型的事件发生时，对应的回调函数被添加到事件队列中，等待执行。



#### 任务队列的执行流程

同步任务在主线程排队执行，异步任务在事件队列排队等待进入主线程执行。

遇到宏任务则推进宏任务队列，遇到微任务则推进微任务队列。

执行宏任务，执行完毕后检查当前层的微任务并执行。

继续执行下一个宏任务，执行对应层次的微任务，直至全部执行完毕。

这个流程确保了异步任务能够在适当的时机插入执行，保持程序的高效性和响应性。

## get和post请求区别

GET在浏览器回退时是无害的，而POST会再次提交请求。

GET产生的URL地址可以被添加书签(Bookmark)，而POST不可以。

GET请求会被浏览器主动cache，而POST不会，除非手动设置。

GET请求只能进行url编码，而POST支持多种编码方式。

GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

GET请求在URL中传送的参数是有长度限制的，而POST没有。

对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

GET参数通过URL传递，POST放在Request body中。

GET产生一个TCP数据包；POST产生两个TCP数据包。

GET与POST都有自己的语义，不能随便混用。

## promise 与 async await

async/await是写异步代码的新方式，以前的方法有回调函数和Promise。

　　async/await是基于Promise实现的，它不能用于普通的回调函数。

　　async/await与Promise一样，是非阻塞的。

　　async/await使得异步代码看起来像同步代码，这正是它的魔力所在

函数前面多了一个aync关键字。await关键字只能用在aync定义的函数内。async函数会隐式地返回一个promise，该promise的reosolve值就是函数return的值。(示例中reosolve值就是字符串”done”)

为什么Async/Await更好？

1）使用async函数可以让代码简洁很多，不需要像Promise一样需要些then，不需要写匿名函数处理Promise的resolve值，也不需要定义多余的data变量，还避免了嵌套代码。

2）错误处理：

　　　　Async/Await 让 try/catch 可以同时处理同步和异步错误。在下面的promise示例中，try/catch 不能处理 JSON.parse 的错误，因为它在Promise中。我们需要使用 .catch，这样错误处理代码非常冗余。并且，在我们的实际生产代码会更加复杂

3）使用async/await的话，代码会变得异常简单和直观。

4）错误栈

　 如果 Promise 连续调用，对于错误的处理是很麻烦的。你无法知道错误出在哪里。

async/await中的错误栈会指向错误所在的函数。在开发环境中，这一点优势并不大。但是，当你分析生产环境的错误日志时，它将非常有用。这时，知道错误发生在makeRequest比知道错误发生在then链中要好。

5）调试

　 async/await能够使得代码调试更简单。2个理由使得调试Promise变得非常痛苦:

《1》不能在返回表达式的箭头函数中设置断点

《2》如果你在.then代码块中设置断点，使用Step Over快捷键，调试器不会跳到下一个.then，因为它只会跳过异步代码。

使用await/async时，你不再需要那么多箭头函数，这样你就可以像调试同步代码一样跳过await语句。

## Async和Await原理

async、await从字面上理解，async就是异步的意思，await就是等待的意思，而两者的用法上也是这样的，async用于申明一个function是异步的，而await用于等待一个异步方法执行完成

过程就是：

1，async就是一个async函数，而await只能在这个函数中使用

2，await表示这里等待await后面的操作执行完毕，再执行下一句代码

3，await后面紧跟的最好是一个定时操作或者一个异步操作

async和await的优点

1，解决了回调地狱的问题

2，支持并发执行

3，可以添加返回值 return xxx

4，可以在代码中添加try/catch捕获错误

## 说一说promise是什么与使用方法？

九大方法: resolve、reject、then、catch、finally、all、allSettled、race、any

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：

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）。

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

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

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

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。

Promise.allSettled()：入参和Promise.all、Promise.race一样，接受一个promise 对象的数组作为参数,也是同时开始、并行执行的。但是Promise.allSettled的返回值需要注意以下几点：Promise.allSettled不会走进catch，当所有输入Promise都被履行或者拒绝时， statusesPromise 会解析一个具有具体完成状态的数组

{ status: 'fulfilled', value:value } ：如果相应的promise被履行

{ status: 'rejected', reason: reason }：如果相应的promise被拒绝

## Vue 的生命周期

Vue实例有⼀个完整的⽣命周期，也就是从**开始创建、初始化数据、编译模版、挂载Dom -> 渲染、更新->渲染、卸载**等⼀系列过程，称这是Vue的⽣命周期。

1.beforeCreate（创建前）：数据观测和初始化事件还未开始，此时 data 的响应式追踪、event/watcher 都还没有被设置，也就是说不 能访问到 data、computed、watch、methods 上的方法和数据。

2.created（创建后） ：实例创建完成，实例上配置的 options 包 括 data、computed、watch、methods 等都配置完成，但是此时渲染 得节点还未挂载到 DOM，所以不能访问到 $el 属性。

3.beforeMount（挂载前）：在挂载开始之前被调用，相关的 render 函数首次被调用。实例已完成以下的配置：编译模板，把 data 里面 的数据和模板生成 html。此时还没有挂载 html 到页面上。

4.mounted（挂载后）：在 el 被新创建的 vm.$el 替换，并挂载到实 例上去之后调用。实例已完成以下的配置：用上面编译好的 html 内 容替换 el 属性指向的 DOM 对象。完成模板中的 html 渲染到 html 页 面中。此过程中进行 ajax 交互。

5.beforeUpdate（更新前）：响应式数据更新时调用，此时虽然响应 式数据更新了，但是对应的真实 DOM 还没有被渲染。

6.updated（更新后）：在由于数据更改导致的虚拟 DOM 重新渲染和 打补丁之后调用。此时 DOM 已经根据响应式数据的变化更新了。调 用时，组件 DOM 已经更新，所以可以执行依赖于 DOM 的操作。然而 在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更 新无限循环。该钩子在服务器端渲染期间不被调用。

7.beforeDestroy（销毁前）：实例销毁之前调用。这一步，实例仍 然完全可用，this 仍能获取到实例。

8.destroyed（销毁后）：实例销毁后调用，调用后，Vue 实例指示 的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例 也会被销毁。该钩子在服务端渲染期间不被调用。

另外还有 keep-alive 独有的生命周期，分别为 activated 和

deactivated。用 keep-alive 包裹的组件在切换时不会进行销毁，而 是缓存到内存中并执行 deactivated 钩子函数，命中缓存渲染后会执 行 activated 钩子函数。

## 深拷贝和浅拷贝

## 深拷贝&浅拷贝&赋值

### 1.深拷贝

深拷贝是将一个对象从内存中完整的拷贝一份出来，从堆内存中开辟一个新的区域存放新对象（新旧对象不共享同一块内存），且修改新对象不会影响原对象（`深拷贝采用了在堆内存中申请新的空间来存储数据，这样每个对象可以避免指针悬挂`）

### 2.浅拷贝

如果属性是基本类型，拷贝的就是基本类型的值。如果属性是引用类型，那么拷贝的就是内存地址（新旧对象共享同一块内存），所以如果其中一个对象改变了这个地址，就会影响到另一个对象（`只是拷贝了指针，使得两个指针指向同一个地址`）

### 3.赋值与浅拷贝的区别

1. 把一个对象赋值给一个新的变量的时候，赋的其实是该对象在栈中的地址，而不是栈中的数据。也就是这两个对象指的是同一个储存空间，不论哪个对象发生改变，其实都是改变储存空间里的内容。因此，两个对象是联动的。

2. 浅拷贝是按位拷贝对象的，他会创建一个新的对象，这个对象有着原始对象属性值的一份精确拷贝。

* 如果属性是基本类型，拷贝的就是基本类型的值。
* 如果属性是内存地址（引用类型），拷贝的就是内存地址，因此如果一个对象改变了这个地址，就会影响到另一个对象。
* 即默认拷贝构造函数只是对对象进行浅拷贝复制（逐个成员依次拷贝），即只复制对象空间而不复制资源。

### 4.浅拷贝的实现【当拷贝对象只有一层的时候，是深拷贝】

* 展开运算符...
* Object.assign()
* Array.prototype.concat()
* Array.prototype.slice()

### 5.深拷贝的实现

* JSON.parse(JSON.stringify())
* 手写递归方法：(递归方法实现深度克隆原理：遍历对象、数组直到里边都是基本数据类型，然后再去复制，就是深度拷贝)
* jQuery的extend方法实现深拷贝
* 函数库lodash的\_.cloneDeep方法

### 6.JSON.parse(JSON.stringify())有什么缺陷？

数据里面有特殊符号、正则这种符号。json.parse进行拷贝可能会出问题

## 怎样解决深拷贝中的循环拷贝

**使用weakmap**：解决循环引用问题，我们可以额外开辟一个存储空间，来存储当前对象和拷贝对象的对应关系这个存储空间，需要可以存储 key-value 形式的数据，且 key 可以是一个引用类型，我们可以选择 weakMap 这种数据结构:

**可以使用 set**：发现相同的对象直接赋值，也可以用 Map

### 赋值不属于拷贝

首先，大家需要区分，赋值不属于拷贝：

let arr = [1,2,3]

let arr1 = arr

// 这里仅仅是把数组的内存地址赋值给arr1，这里不叫拷贝

简单来说，如果b复制了a，当修改a时，看b是否会发生变化，如果b也变了，说明这是浅拷贝，如果b没变，说明这是深拷贝。

## http1.0，http1.1，http2，http3

### http1.0

HTTP1.0最早在网页中使用是在1996年，那个时候只是使用一些较为简单的网页上和网络请求上,是一种**无状态、无连接**的应用层协议，几年后被HTTP1.1代替并广泛使用

### http1.1

http1.1基于**文本解析**,把所有请求和响应作为纯文本

http1.1加入了**缓存处理（强缓存和协商缓存）**

http1.1拥有长连接，并支持请求**管道化（pipelining）**，

http1.1流控制基于**tcp连接**。当连接建立时，两端通过系统默认机制建立缓冲区。并通过ack报文来通知对方接收窗口大小，因为http1.1 依靠传输层来避免流溢出，每个tcp连接需要一个独立的流控制机制

### 缓存处理（强缓存和协商缓存）

浏览器缓存能优化性能，而浏览器缓存分为强缓存和协商缓存，都是从客户端读取缓存

#### 强缓存

强缓存不发送请求，直接读取资源，可以获得返回200的状态码

利用http头中的Expires和Cache-Control两个字段来控制，都用来表示资源的缓存时间，Expires能设置失效时间，而Cache-Control能做到更多选项更细致，如果同时设置的话，其优先级高于Expires

#### 协商缓存

通过服务器来确定缓存资源是否可用，通过request header判断是否命中请求，命中后返回304状态码，并返回新的request header通知客户端从缓存里取

普通刷新会启用弱缓存，忽略强缓存。只有在地址栏或收藏夹输入网址、通过链接引用资源等情况下，浏览器才会启用强缓存

如果时间过期，则向服务器发送header带有If-None-Match和If-Modified-Since的请求，回到1

### http2

http2相比于http1.1，**性能**大幅度提升

http2通过一个连接来**多路复用**

http2拥有**头部压缩**

http2拥有**新的二进制格式**，使用二进制框架层把所有消息封装成二进制，且仍然保持http语法

http2允许客户端和服务器端实现他们自己的流控制机制，而不是依赖传输层,两端在传输层交换可用的缓冲区大小，来让他们在多路复用流上设置自己的接收窗口

http2让服务器可以将响应主动“**推送**”到客户端缓存中

#### htpp2头部压缩

http2头部压缩又称为HAPCK，设计简单而灵活，是因为HPACK格式有意地简单且不灵活能降低由于实现错误而导致的互操作性或安全问题的风险

http1.1没有头部压缩，随着请求增加，冗余头部字段会不必要地占用带宽，从而显着增加延迟，而头部压缩可消除冗余报头字段，限制已知安全攻击的漏洞，并且在受限环境中使用有限的内存要求

#### http2多路复用

http 性能优化的关键并不在于高带宽，而是低延迟

tcp 连接会随着时间进行自我「调谐」，起初会限制连接的最大速度，如果数据成功传输，会随着时间的推移提高传输的速度,这种调谐则被称为 tcp 慢启动,由于这种原因，让原本就具有突发性和短时性的 http 连接变的十分低效

http/2 通过让所有数据流共用同一个连接，可以更有效地使用 tcp 连接，让高带宽也能真正的服务于 http 的性能提升。而http1.1存在低性能的线头阻塞，一旦有一个请求超时，便会出现阻塞等待的情况

### http3

之前说了http2，那么http3就是为了解决http2相关问题而诞生，它基于一个新的传输层协议QUIC，而http3就是建立一个在QUIC上运行的HTTP新规范，而http3之前的版本都是基于TCP，QUIC就是为了替代TCP，解决TCP的一些缺陷

### tcp

1. **不支持流级复用**，TCP会将所有对象序列化在同一个流中，因此，它不知道TCP段的对象级分区，无法在同一个流中复用数据包
2. **会产生冗余通信**，tco三次连接握手会有冗余的消息交换序列
3. **可能会间歇性地挂起数据传输**，tcp中有个因为序列顺序处理丢失的问题的缺陷称为行头阻塞

### QUIC

同样拥有头部压缩，并优化了对乱序发送的支持，也优化了压缩率

放弃tcp，通过udp建立，提高了连接建立的速度，降低了延迟

tcp本身是无法解决队头拥塞，quic则解决了这个问题

Connection ID使得http3支持连接迁移以及NAT的重绑定

### HTTP1.0对比HTTP1.1

HTTP1.1主要改进了以下几点内容

* keep-alive
* 客户端缓存
* 连接代宽优化
* 请求Host域
* 请求状态码
* 请求方法

#### keep-alive

在HTTP 1.0中，客户端的每次请求都要求建立一次单独的连接，在处理完本次请求后，就自动释放连接。

在HTTP 1.1中则可以在一次连接中处理多个请求，并且多个请求可以重叠进行，不需要等待一个请求结束后再发送下一个请求。

建立起一个 TCP 连接需要经过“三次握手”：

第一次握手：客户端发送syn包(syn=j)到服务器，并进入SYN\_SEND状态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握 手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连 接之前，TCP 连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开TCP连接的请求，断开过程需要经过“四次握手”（过程就不细写 了，就是服务器和客户端交互，最终确定断开）

HTTP 1.1持久连接（PersistentConnection）在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭TCP连接的消耗和延迟，在HTTP1.1中默认开启Connection： keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

#### 客户端缓存

在HTTP/1.0中，使用Expire头域来判断资源的fresh或stale，并使用条件请求（conditional request）来判断资源是否仍有效。例如，cache服务器通过If-Modified-Since头域向服务器验证资源的Last-Modefied头域是否有更新，源服务器可能返回304（Not Modified），则表明该对象仍有效；也可能返回200（OK）替换请求的Cache对象。

此外，HTTP/1.0中还定义了Pragma:no-cache头域，客户端使用该头域说明请求资源不能从cache中获取，而必须回源获取。

HTTP/1.1在1.0的基础上加入了一些cache的新特性，当缓存对象的Age超过Expire时变为stale对象，cache不需要直接抛弃stale对象，而是与源服务器进行重新激活（revalidation）。

HTTP/1.0中，If-Modified-Since头域使用的是绝对时间戳，精确到秒，但使用绝对时间会带来不同机器上的时钟同步问题。而HTTP/1.1中引入了一个ETag头域用于重激活机制，它的值entity tag可以用来唯一的描述一个资源。请求消息中可以使用If-None-Match头域来匹配资源的entitytag是否有变化。

为了使caching机制更加灵活，HTTP/1.1增加了Cache-Control头域（请求消息和响应消息都可使用），它支持一个可扩展的指令子集：例如max-age指令支持相对时间戳；private和no-store指令禁止对象被缓存；no-transform阻止Proxy进行任何改变响应的行为。

Cache使用关键字索引在磁盘中缓存的对象，在HTTP/1.0中使用资源的URL作为关键字。但可能存在不同的资源基于同一个URL的情况，要区别它们还需要客户端提供更多的信息，如Accept-Language和Accept-Charset头域。为了支持这种内容协商机制(content negotiation mechanism)，HTTP/1.1在响应消息中引入了Vary头域，该头域列出了请求消息中需要包含哪些头域用于内容协商。

#### 强缓存、协商缓存

我们在使用浏览器访问一个web页面的时候，浏览器会将该网页中的资源和相应的服务端response header强制缓存起来，在这些头信息中有两个字段来控制强缓存Cache-Control和Expires，如果这两个字段验证通过就使用缓存，如果没有通过就会带着另外两个字段Etag/If-None-Match和Last-Modified/If-Modified-Since去向服务端发送一个验证（协商）请求，如果验证通过就返回304继续使用缓存，如果没有通过就返回200和对应的资源。

#### 连接代宽优化

HTTP1.1支持传送内容的一部分。比方说，当客户端已经有内容的一部分，为了节省带宽，可以只向服务器请求一部分。

HTTP/1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了。例如，客户端只需要显示一个文档的部分内容，又比如下载大文件时需要支持断点续传功能，而不是在发生断连后不得不重新下载完整的包。

HTTP/1.1中在请求消息中引入了range头域，它允许只请求资源的某个部分。

"Content-Type": "application/octet-stream",

"Content-Range": \`bytes ${startPosition}-${endPosition}/${file.size}\`

在响应消息中Content-Range头域声明了返回的这部分对象的起始位置和长度。如果服务器相应地返回了对象所请求范围的内容，则响应码为206（Partial Content），它可以防止Cache将响应误以为是完整的一个对象。我的文件并发上传项目有使用到这个请求方式。

节省带宽资源的一个非常有效的做法就是压缩要传送的数据。Content-Encoding是对消息进行端到端（end-to-end）的编码，它可能是资源在服务器上保存的固有格式（如jpeg图片格式）；在请求消息中加入Accept-Encoding头域，它可以告诉服务器客户端能够解码的编码方式。

而Transfer-Encoding是逐段式（hop-by-hop）的编码，如Chunked编码。在请求消息中加入TE头域用来告诉服务器能够接收的transfer-coding方式，

#### 请求Host域

\*\*Host \*\*请求头指明了请求将要发送到的服务器主机名和端口号。HTTP1.0并没有包含Host，

如果没有包含端口号，会自动使用被请求服务的默认端口（比如HTTPS URL使用443端口，HTTP URL使用80端口）。

如果使用的HTTP1.0我们就不能给一个服务器部署多个服务，HTTP1.1解决了这个问题。

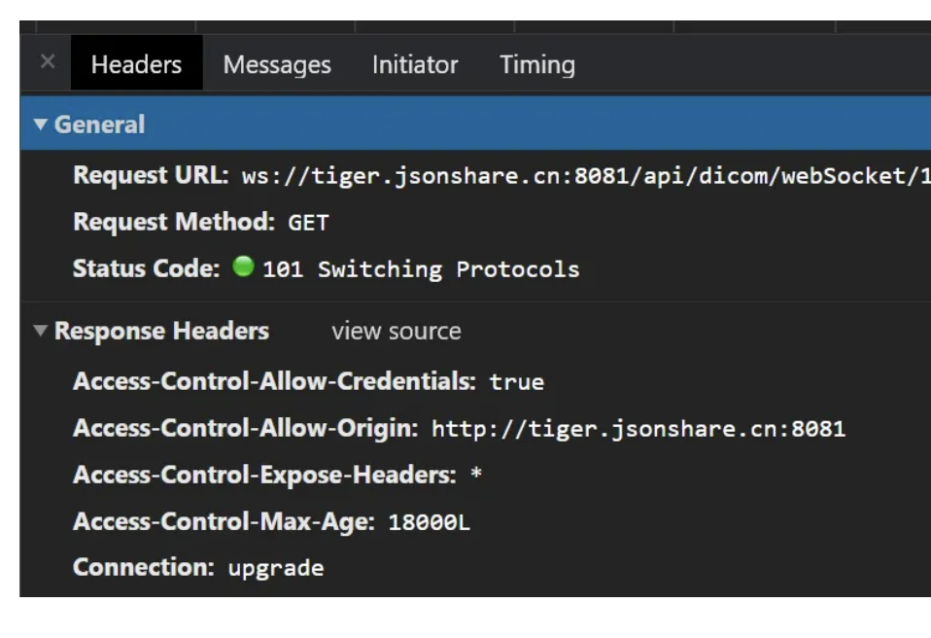
所有HTTP/1.1 请求报文中必须包含一个Host头字段。对于缺少Host头或者含有超过一个Host头的HTTP/1.1 请求，可能会收到400（Bad Request）状态码。

#### 请求状态码

HTTP1.1新增了两个1.0中没有的状态码，

100 (Continue) 状态代码的使用，允许客户端在发request消息body之前先用request header试探一下server，看server要不要接收request body，再决定要不要发request body。

101 (Switching Protocols) 当我们在网页中使用websocket的时候就会出现一个这样的状态码



#### 请求方法

HTTP1.0 定义了三种请求方法： GET, POST 和 HEAD方法。

HTTP1.1 新增了六种请求方法：OPTIONS、PUT、PATCH、DELETE、TRACE 和 CONNECT 方法。

### HTTP2.0

http2.0是一种安全高效的下一代http传输协议。安全是因为http2.0建立在https协议的基础上，高效是因为它是通过二进制分帧来进行数据传输。

他对于HTTP1.x主要有以下改进

* 二进制分帧（Binary Format）
* 头部压缩（Header Compression）
* 服务端推送（Server Push）
* 多路复用 (Multiplexing) / 连接共享，让多个请求合并在同一 TCP 连接内
* 请求优先级（Request Priorities）

#### 二进制分帧

有别于HTTP/1.1在连接中的明文请求，HTTP/2与SPDY一样，将一个TCP连接分为若干个流（Stream），每个流中可以传输若干消息（Message），每个消息由若干最小的二进制帧（Frame）组成。这也是HTTP/1.1与HTTP/2最大的区别所在。 HTTP/2中，每个用户的操作行为被分配了一个流编号(stream ID)，这意味着用户与服务端之间创建了一个TCP通道

启用http2.0后会给性能带来很大的提升，但同时也会带来新的性能瓶颈。因为现在所有的压力集中在底层一个TCP连接之上，TCP很可能就是下一个性能瓶颈，比如TCP分组的队首阻塞问题，单个TCP packet丢失导致整个连接阻塞，无法逃避，此时所有消息都会受到影响。

具体内容有机会可以单独总结一下，我在另一篇文章总结了HTTP和HTTPS区别。

#### 多路复用

同一域名下多个请求公用同一个连接，不限制请求数量，双向数据流，多个请求之前数据是无序的，浏览器最终会根据每一帧的编号进行组装。

#### header压缩

在HTTP1中每次传输的header都属文本形式传输，当header中的参数比较多时，对于压缩就显得很有必要了，头部会在连接上始终存在，对于多个请求中重复的字段会自动过滤。

#### Server Push

服务器会对于客户端必需的资源在双方协商好的前提下会主动推送。

### HTTP3

HTTP3是一个基于UDP协议的应用层协议，并且集成了HTTP2的加密传输、多路复用的几个特点

QUIC (Quick UDP Internet Connections), 快速 UDP 互联网连接。

QUIC是基于UDP协议的。

#### 线头阻塞(HOL)问题的解决更为彻底：

基于TCP的HTTP/2，尽管从逻辑上来说，不同的流之间相互独立，不会相互影响，但在实际传输方面，数据还是要一帧一帧的发送和接收，一旦某一个流的数据有丢包，则同样会阻塞在它之后传输的流数据传输。而基于UDP的QUIC协议则可以更为彻底地解决这样的问题，让不同的流之间真正的实现相互独立传输，互不干扰。

#### 切换网络时的连接保持

当前移动端的应用环境，用户的网络可能会经常切换，比如从办公室或家里出门，WiFi断开，网络切换为3G或4G。基于TCP的协议，由于切换网络之后，IP会改变，因而之前的连接不可能继续保持。而基于UDP的QUIC协议，则可以内建与TCP中不同的连接标识方法，从而在网络完成切换之后，恢复之前与服务器的连接。

总结：基于UDP和应用层实现了TCP协议。

## 什么是websocket

WebSocket是HTML5下一种新的协议（websocket协议本质上是一个基于tcp的协议）

它实现了浏览器与服务器全双工通信，能更好的节省服务器资源和带宽并达到实时通讯的目的

Websocket是一个持久化的协议

### websocket的原理

websocket约定了一个通信的规范，通过一个握手的机制，客户端和服务器之间能建立一个类似tcp的连接，从而方便它们之间的通信

在websocket出现之前，web交互一般是基于http协议的短连接或者长连接

websocket是一种全新的协议，不属于http无状态协议，协议名为"ws"

### websocket与http的关系

相同点：

都是基于tcp的，都是可靠性传输协议

都是应用层协议

不同点：

WebSocket是双向通信协议，模拟Socket协议，可以双向发送或接受信息

HTTP是单向的

WebSocket是需要浏览器和服务器握手进行建立连接的

而http是浏览器发起向服务器的连接，服务器预先并不知道这个连接

联系：

WebSocket在建立握手时，数据是通过HTTP传输的。但是建立之后，在真正

传输时候是不需要HTTP协议的

总结（总体过程）：

首先，客户端发起http请求，经过3次握手后，建立起TCP连接；http请求里存放WebSocket支持的版本号等信息，如：Upgrade、Connection、WebSocket-Version等；

然后，服务器收到客户端的握手请求后，同样采用HTTP协议回馈数据；

最后，客户端收到连接成功的消息后，开始借助于TCP传输信道进行全双工通信。

### websocket解决的问题

#### 1.http存在的问题

http是一种无状态协议，每当一次会话完成后，服务端都不知道下一次的客户端是谁，需要每次知道对方是谁，才进行相应的响应，因此本身对于实时通讯就是一种极大的障碍

http协议采用一次请求，一次响应，每次请求和响应就携带有大量的header头，对于实时通讯来说，解析请求头也是需要一定的时间，因此，效率也更低下

最重要的是，需要客户端主动发，服务端被动发，也就是一次请求，一次响应，不能实现主动发送

#### 2.long poll(长轮询)

对于以上情况就出现了http解决的第一个方法——长轮询

基于http的特性，简单点说，就是客户端发起长轮询，如果服务端的数据没有发生变更，会 hold

住请求，直到服务端的数据发生变化，或者等待一定时间超时才会返回。返回后，客户端又会立即再次发起下一次长轮询

优点是解决了http不能实时更新的弊端，因为这个时间很短，发起请求即处理请求返回响应，实现了“伪·长连接”

张三取快递的例子，张三今天一定要取到快递，他就一直站在快递点，等待快递一到，立马取走

从例子上来看有个问题：

假如有好多人一起在快递站等快递，那么这个地方是否足够大，（抽象解释：需要有很高的并发，同时有很多请求等待在这里）

总的来看：

推送延迟。服务端数据发生变更后，长轮询结束，立刻返回响应给客户端。

服务端压力。长轮询的间隔期一般很长，例如 30s、60s，并且服务端 hold 住连接不会消耗太多服务端资源。

#### 3.Ajax轮询

基于http的特性，简单点说，就是规定每隔一段时间就由客户端发起一次请求，查询有没有新消息，如果有，就返回，如果没有等待相同的时间间隔再次询问

优点是解决了http不能实时更新的弊端，因为这个时间很短，发起请求即处理请求返回响应，把这个过程放大n倍，本质上还是request =

response

举个形象的例子（假设张三今天有个快递快到了，但是张三忍耐不住，就每隔十分钟给快递员或者快递站打电话，询问快递到了没，每次快递员就说还没到，等到下午张三的快递到了，but，快递员不知道哪个电话是张三的，（可不是只有张三打电话，还有李四，王五），所以只能等张三打电话，才能通知他，你的快递到了）

从例子上来看有两个问题：

假如说，张三打电话的时间间隔为10分钟，当他收到快递前最后一次打电话，快递员说没到，他刚挂掉电话，快递入库了（就是到了），那么等下一次时间到了，张三打电话知道快递到了，那么这样的通讯算不算实时通讯？很显然，不算，中间有十分钟的时间差，还不算给快递员打电话的等待时间（抽象的解释：每次request的请求时间间隔等同于十分钟，请求解析相当于等待）

假如说张三所在的小区每天要收很多快递，每个人都采取主动给快递员打电话的方式，那么快递员需要以多快的速度接到，其他人打电话占线也是问题（抽象解释：请求过多，服务端响应也会变慢）

总的来看，Ajax轮询存在的问题：

推送延迟。

服务端压力。配置一般不会发生变化，频繁的轮询会给服务端造成很大的压力。

推送延迟和服务端压力无法中和。降低轮询的间隔，延迟降低，压力增加；增加轮询的间隔，压力降低，延迟增高

#### 4.websocket的改进

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端中断连接前，不需要客户端和服务端重新发起连接请求。在海量并发及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，实现了“真·长链接”，实时性优势明显。