HTML

简述一下你对 HTML 语义化的理解?

用正确的标签做正确的事情。
html 语义化让页面的内容结构化，结构更清晰，便于对浏览器、搜索引擎解析；即使在没有样式 CSS 情况下也以一种文档格式显示，并且是容易阅读的;
搜索引擎的爬虫也依赖于 HTML 标记来确定上下文和各个关键字的权重，利于 SEO;
使阅读源代码的人对网站更容易将网站分块，便于阅读维护理解。

标签上 title 与 alt 属性的区别是什么?

alt 是给搜索引擎识别，在图像无法显示时的替代文本;
title 是关于元素的注释信息，主要是给用户解读。
当鼠标放到文字或是图片上时有 title 文字显示。（因为 IE 不标准）在 IE 浏览器中 alt 起到了 title 的作用，变成文字提示。
在定义 img 对象时，将 alt 和 title 属性写全，可以保证在各种浏览器中都能正常使用。

iframe的优缺点?

优点:

解决加载缓慢的第三方内容如图标和广告等的加载问题
Security sandbox
并行加载脚本
缺点:

iframe会阻塞主页面的Onload事件
即时内容为空，加载也需要时间
没有语意

href 与 src?

href (Hypertext Reference)指定网络资源的位置，从而在当前元素或者当前文档和由当前属性定义的需要的锚点或资源之间定义一个链接或者关系。（目的不是为了引用资源，而是为了建立联系，让当前标签能够链接到目标地址。）

src source（缩写），指向外部资源的位置，指向的内容将会应用到文档中当前标签所在位置。

href与src的区别

1、请求资源类型不同:

href 指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的联系。在请求 src 资源时会将其指向的资源下载并应用到文档中，比如 JavaScript 脚本，img 图片；\

2、作用结果不同:

href 用于在当前文档和引用资源之间确立联系；src 用于替换当前内容;

3、浏览器解析方式不同:

当浏览器解析到src ，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架等也如此，类似于将所指向资源应用到当前内容。这也是为什么建议把 js 脚本放在底部而不是头部的原因。

CSS

介绍一下 CSS 的盒子模型?

有两种， IE 盒子模型、W3C 盒子模型;
盒模型: 内容(content)、填充(padding)、边界(margin)、 边框(border);
区 别: IE 的 content 部分把 border 和 padding 计算了进去;

css 选择器优先级?

!important > 行内样式（比重1000）> ID 选择器（比重100） > 类选择器（比重10） > 标签（比重1） > 通配符 > 继承 > 浏览器默认属性

垂直居中几种方式?

单行文本: line-height = height
图片: vertical-align: middle;
absolute 定位: top: 50%;left: 50%;transform: translate(-50%, -50%);
flex: display:flex;margin:auto

简明说一下 CSS link 与 @import 的区别和用法?

link 是 XHTML 标签，除了加载CSS外，还可以定义 RSS 等其他事务；@import 属于 CSS 范畴，只能加载 CSS。
link 引用 CSS 时，在页面载入时同时加载；@import 需要页面网页完全载入以后加载。
link 是 XHTML 标签，无兼容问题；@import 是在 CSS2.1 提出的，低版本的浏览器不支持。
link 支持使用 Javascript 控制 DOM 去改变样式；而@import不支持。

rgba和opacity的透明效果有什么不同?

opacity 会继承父元素的 opacity 属性，而 RGBA 设置的元素的后代元素不会继承不透明属性。

display:none和visibility:hidden的区别?

display:none 隐藏对应的元素，在文档布局中不再给它分配空间，它各边的元素会合拢，就当他从来不存在。
visibility:hidden 隐藏对应的元素，但是在文档布局中仍保留原来的空间。

position的值， relative和absolute分别是相对于谁进行定位的?

relative:相对定位，相对于自己本身在正常文档流中的位置进行定位。
absolute:生成绝对定位，相对于最近一级定位不为static的父元素进行定位。
fixed: （老版本IE不支持）生成绝对定位，相对于浏览器窗口或者frame进行定位。
static:默认值，没有定位，元素出现在正常的文档流中。
sticky:生成粘性定位的元素，容器的位置根据正常文档流计算得出。

画一条0.5px的直线?

考查的是css3 的 transform

height: 1px;
transform: scale(0.5);

calc, support, media各自的含义及用法?

@support 主要是用于**检测浏览器是否支持CSS的某个属性**，其实就是**条件判断**，如果支持某个属性，你可以写一套样式，如果不支持某个属性，你也可以提供另外一套样式作为替补。
calc() **函数用于动态计算长度值。** calc()函数支持 “+”, “-”, “*”, “/” 运算；
@media 查询，你可以针**对不同的媒体类型定义不同的样式。**

1rem、1em、1vh、1px各自代表的含义?

rem
rem是**全部的长度都相对于根元素元素。**通常做法是给html元素设置一个字体大小，然后其他元素的长度单位就为rem。
em
子元素字体大小的em**是相对于父元素字体大小**
元素的width/height/padding/margin用em的话是相对于该元素的font-size
vw/vh
全称是 Viewport Width 和 Viewport Height，视窗的宽度和高度，相当于 屏幕宽度和高度的 1%，不过，处理宽度的时候%单位更合适，处理高度的 话 vh 单位更好。
px
px像素（Pixel）。相对长度单位。像素px是相对于显示器屏幕分辨率而言的。
一般电脑的分辨率有{19201024}等不同的分辨率
19201024 前者是屏幕宽度总共有1920个像素,后者则是高度为1024个像素

no画一个三角形?

这属于简单的css考查，平时在用组件库的同时，也别忘了原生的css

```
.a {
    width: 0;
    height: 0;
    border-width: 100px;
    border-style: solid;
    border-color: transparent #0099CC transparent transparent;
    transform: rotate(90deg); /*顺时针旋转90°*/
}
<div class="a"></div>
```

HTML / CSS 混合篇

HTML5、CSS3 里面都新增了那些新特性?

HTML5

新的语义标签
article 独立的内容。
aside 侧边栏。
header 头部。
nav 导航。
section 文档中的节。
footer 页脚。
画布(Canvas) API
地理(Geolocation) API
本地离线存储 localStorage 长期存储数据，浏览器关闭后数据不丢失；
sessionStorage 的数据在浏览器关闭后自动删除

新的技术webworker, websocket, Geolocation
拖拽释放(Drag and drop) API
音频、视频API(audio,video)
表单控件，calendar、date、time、email、url、searc

CSS3

2d，3d变换
Transition, animation
媒体查询
新的单位（rem, vw，vh 等）
圆角（border-radius），阴影（box-shadow），对文字加特效（text-shadow），线性渐变（gradient），旋转（transform）
transform:rotate(9deg) scale(0.85,0.90) translate(0px,-30px) skew(-9deg,0deg);//旋转,缩放,定位,倾斜
rgba

BFC 是什么?

BFC 即 Block Formatting Contexts (块级格式化上下文)，它属于普通流，即：元素按照其在 HTML 中的先后位置至上而下布局，在这个过程中，行内元素水平排列，直到当行被占满然后换行，块级元素则会被渲染为完整的一个新行，除非另外指定，否则所有元素默认都是普通流定位，也可以说，普通流中元素的位置由该元素在 HTML 文档中的位置决定。
可以把 BFC 理解为一个封闭的大箱子，箱子内部的元素无论如何翻江倒海，都不会影响到外部。
只要元素满足下面任一条件即可触发 BFC 特性

body 根元素
浮动元素：float 除 none 以外的值
绝对定位元素：position (absolute、fixed)
display 为 inline-block、table-cells、flex
overflow 除了 visible 以外的值 (hidden、auto、scroll)

常见兼容性问题?

浏览器默认的margin和padding不同。解决方案是加一个全局的*{margin:0;padding:0;}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.

JS

JS 数据类型？

数据类型主要包括两部分：

基本数据类型： Undefined、Null、Boolean、Number 和 String
引用数据类型： Object (包括 Object 、Array 、Function)
ECMAScript 2015 新增:Symbol(创建后独一无二且不可变的数据类型)

判断一个值是什么类型有哪些方法?

typeof 运算符
instanceof 运算符
Object.prototype.toString 方法

null 和 undefined 的区别?

null 表示一个对象被定义了，值为“空值”；
undefined 表示不存在这个值。
（1）变量被声明了，但没有赋值时，就等于undefined。 （2) 调用函数时，应该提供的参数没有提供，该参数等于undefined。 （3）对象没有赋值的属性，该属性的值为undefined。 （4）函数没有返回值时，默认返回undefined。

怎么判断一个变量arr的话是否为数组（此题用 typeof 不行）？

arr instanceof Array
arr.constructor == Array
Object.protype.toString.call(arr) == ‘[Object Array]’

“ ===”、“ ==”的区别?

==，当且仅当两个运算数相等时，它返回 true，即不检查数据类型

===，只有在无需类型转换运算数就相等的情况下，才返回 true，需要检查数据类型

“eval是做什么的?

它的功能是把对应的字符串解析成 JS 代码并运行；
应该避免使用 eval，不安全，非常耗性能（2次，一次解析成 js 语句，一次执行）。

箭头函数有哪些特点?

不需要function关键字来创建函数
省略return关键字
改变this指向

var、let、const 区别?

var 存在变量提升。
let 只能在块级作用域内访问。
const 用来定义常量，必须初始化，不能修改（对象特殊）

new操作符具体干了什么呢?

1、创建一个空对象，并且 this 变量引用该对象，同时还继承了该函数的原型。
2、属性和方法被加入到 this 引用的对象中。
3、新创建的对象由 this 所引用，并且最后隐式的返回 this 。

JSON 的了解?

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。
它是基于JavaScript的一个子集。数据格式简单, 易于读写, 占用带宽小
{'age':'12', 'name':'back'}

documen.write 和 innerHTML 的区别?

document.write 只能重绘整个页面
innerHTML 可以重绘页面的一部分

ajax过程?

(1)创建XMLHttpRequest对象,也就是创建一个异步调用对象.
(2)创建一个新的HTTP请求,并指定该HTTP请求的方法、URL及验证信息.
(3)设置响应HTTP请求状态变化的函数.
(4)发送HTTP请求.
(5)获取异步调用返回的数据.
(6)使用JavaScript和DOM实现局部刷新.

请解释一下 JavaScript 的同源策略?

概念:同源策略是客户端脚本（尤其是Netscape Navigator2.0，其目的是防止某个文档或脚本从多个不同源装载。
这里的同源策略指的是：协议，域名，端口相同，同源策略是一种安全协议。
指一段脚本只能读取来自同一来源的窗口和文档的属性。

介绍一下闭包和闭包常用场景?

闭包是指有权访问另一个函数作用域中的变量的函数，创建闭包常见方式，就是在一个函数的内部创建另一个函数
使用闭包主要为了设计私有的方法和变量，闭包的优点是可以避免变量的污染，缺点是闭包会常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。在js中，函数即闭包，只有函数才会产生作用域的概念。
闭包有三个特性：
函数嵌套函数
函数内部可以引用外部的参数和变量
参数和变量不会被垃圾回收机制回收
应用场景，设置私有变量的方法
不适用场景：返回闭包的函数是个非常大的函数
闭包的缺点就是常驻内存，会增大内存使用量，使用不当会造成内存泄漏

javascript的内存(垃圾)回收机制?

垃圾回收器会每隔一段时间找出那些不再使用的内存，然后为其释放内存
一般使用标记清除方法(mark and sweep), 当变量进入环境标记为进入环境，离开环境标记为离开环境
垃圾回收器会在运行的时候给存储在内存中的所有变量加上标记，然后去掉环境中的变量以及被环境中变量所引用的变量（闭包），在这些完成之后仍存在标记的就是要删除的变量了
还有引用计数方法(reference counting), 在低版本IE中经常会出现内存泄露，很多时候就是因为其采用引用计数方式进行垃圾回收。引用计数的策略是跟踪记录每个值被使用的次数，当声明了一个 变量并将一个引用类型赋值给该变量的时候这个值的引用次数就加1，如果该变量的值变成了另外一个，则这个值得引用次数减1，当这个值的引用次数变为0的时 候，说明没有变量在使用，这个值没法被访问了，因此可以将其占用的空间回收，这样垃圾回收器会在运行的时候清理掉引用次数为0的值占用的空间。
在IE中虽然JavaScript对象通过标记清除的方式进行垃圾回收，但BOM与DOM对象却是通过引用计数回收垃圾的， 也就是说只要涉及BOM及DOM就会出现循环引用问题。

JavaScript原型，原型链？有什么特点?

```
任何对象都有 proto 隐式原型，等于 构造函数 的 prototype
const obj = {}
obj.proto === Object.prototype // true

任何函数都有 prototype 显示原型 等于 原型对象(就是一个普通对象包含公共属性)
*(通过Function.prototype.bind方法构造出来的函数是个例外，它没有prototype属性)
function Person () {}
```

```
Person.prototype = 原型对象

Person.prototype.constructor === Person // true

const person1 = new Person
person1.proto === Person.prototype // true
person1.constructor == Person // true
```

对象还具有 constructor 属性，指向构造函数（Person.prototype.constructor == Person）
原型链是依赖于proto, 查找一个属性会沿着 proto 原型链向上查找，直到找到为止。
特殊

```
// 原型链最终点是 null

Object.prototype.proto === null // true

obj.proto.proto === null // true
```

每个对象都会在其内部初始化一个属性，就是prototype(原型)，当我们访问一个对象的属性时，
如果这个对象内部不存在这个属性，那么他就会去prototype里找这个属性，这个prototype又会有自己的prototype，
于是就这样一直找下去，也就是我们平时所说的原型链的概念。
关系：instance.constructor.prototype = instance.proto
特点：
JavaScript对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变。

用js递归的方式写1到100求和?

```
function add(num1, num2) {
const num = num1 + num2;
if(num2 === 100) {
    return num;
} else {
    return add(num, num2 + 1)
}
}
var sum = add(1, 2);
```

事件队列（宏任务微任务）

可以分为微任务（micro task）队列和宏任务（macro task）队列。

微任务一般比宏任务先执行，并且微任务队列只有一个，宏任务队列可能有多个。另外我们常见的点击和键盘等事件也属于宏任务。

下面我们看一下常见宏任务和常见微任务。

常见宏任务：

setTimeout()
setInterval()
setImmediate()

常见微任务：

promise.then()、promise.catch()
new MutaionObserver()
process.nextTick()
微任务和宏任务的本质区别。

宏任务特征：有明确的异步任务需要执行和回调；需要其他异步线程支持。
微任务特征：没有明确的异步任务需要执行，只有回调；不需要其他异步线程支持。

```
setTimeout(function () {
    console.log("1");
}, 0);
async function async1() {
    console.log("2");
    const data = await async2();
    console.log("3");
    return data;
}
async function async2() {
    return new Promise((resolve) => {
        console.log("4");
        resolve("async2的结果");
    }).then((data) => {
        console.log("5");
        return data;
    });
```

```
}
async1().then((data) => {
    console.log("6");
    console.log(data);
});
new Promise(function (resolve) {
    console.log("7");
  resolve()
}).then(function () {
    console.log("8");
});

// 2 4 7 5 8 3 6 async2的结果 1
```

async/await

async 是一个通过异步执行并隐式返回 Promise 作为结果的函数。是Generator函数的语法糖，并对Generator函数进行了改进。
改进：

内置执行器，无需手动执行 next() 方法。
更好的语义
更广的适用性：co模块约定，yield命令后面只能是 Thunk 函数或 Promise 对象，而async函数的await命令后面，可以是 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时会自动转成立即 resolved 的 Promise 对象）。
返回值是 Promise，比 Generator 函数返回的 Iterator 对象方便，可以直接使用 then() 方法进行调用。
async 隐式返回 Promise 作为结果的函数，那么可以简单理解为，await后面的函数执行完毕时，await会产生一个微任务(Promise.then是微任务)。

JavaScript 是单线程的，浏览器是多进程的

每打开一个新网页就会创建一个渲染进程
渲染进程是多线程的
负责页面渲染的 GUI 渲染线程
负责JavaScript的执行的 JavaScript 引擎线程，
负责浏览器事件循环的事件触发线程，注意这不归 JavaScript 引擎线程管
负责定时器的定时触发器线程，setTimeout 中低于 4ms 的时间间隔算为4ms
负责XMLHttpRequest的异步 http 请求线程
GUI 渲染线程与 JavaScript 引擎线程是互斥的
单线程JavaScript是因为避免 DOM 渲染的冲突，web worker 支持多线程，但是 web worker 不能访问 window 对象，document 对象等。

VUE

谈谈你对MVVM开发模式的理解?

MVVM分为Model、View、ViewModel三者。
Model 代表数据模型，数据和业务逻辑都在Model层中定义；
View 代表UI视图，负责数据的展示；
ViewModel 负责监听 Model 中数据的改变并且控制视图的更新，处理用户交互操作；
Model 和 View 并无直接关联，而是通过 ViewModel 来进行联系的，Model 和 ViewModel 之间有着双向数据绑定的联系。因此当 Model 中的数据改变时会触发 View 层的刷新，View 中由于用户交互操作而改变的数据也会在 Model 中同步。
这种模式实现了 Model 和 View 的数据自动同步，因此开发者只需要专注对数据的维护操作即可，而不需要自己操作 dom。

v-if 和 v-show 有什么区别?

v-if 是真正的条件渲染，会控制这个 DOM 节点的存在与否。因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建；也是惰性的：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。
v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 的 “display” 属性进行切换。
当我们需要经常切换某个元素的显示/隐藏时，使用v-show会更加节省性能上的开销；当只需要一次显示或隐藏时，使用v-if更加合理。

你使用过 Vuex 吗?

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。每一个 Vuex 应用的核心就是 store（仓库）。“store” 基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。

（1）Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。
（2）改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样使得我们可以方便地跟踪每一个状态的变化。
主要包括以下几个模块：

State => 基本数据，定义了应用状态的数据结构，可以在这里设置默认的初始状态。
Getter => 从基本数据派生的数据，允许组件从 Store 中获取数据，mapGetters 辅助函数仅仅是将 store 中的 getter 映射到局部计算属性。
Mutation => 是唯一更改 store 中状态的方法，且必须是同步函数。
Action => 像一个装饰器，包裹mutations，使之可以异步。用于提交 mutation，而不是直接变更状态，可以包含任意异步操作。
Module => 模块化Vuex，允许将单一的 Store 拆分为多个 store 且同时保存在单一的状态树中。

说说你对 SPA 单页面的理解，它的优缺点分别是什么?

SPA（ single-page application ）仅在 Web 页面初始化时加载相应的 HTML、JavaScript 和 CSS。一旦页> 面加载完成，SPA 不会因为用户的操作而进行页面的重新加载或跳转；取而代之的是利用路由机制实现 > HTML 内容的变换，UI 与用户的交互，避免页面的重新加载。

优点：
用户体验好、快，内容的改变不需要重新加载整个页面，避免了不必要的跳转和重复渲染；
基于上面一点，SPA 相对对服务器压力小；
前后端职责分离，架构清晰，前端进行交互逻辑，后端负责数据处理；
缺点：
初次加载耗时多：为实现单页 Web 应用功能及显示效果，需要在加载页面的时候将 JavaScript、CSS 统一> 加载，部分页面按需加载；
前进后退路由管理：由于单页应用在一个页面中显示所有的内容，所以不能使用浏览器的前进后退功能，所> 有的页面切换需要自己建立堆栈管理；
SEO 难度较大：由于所有的内容都在一个页面中动态替换显示，所以在 SEO 上其有着天然的弱势。

Class 与 Style 如何动态绑定?

Class 可以通过对象语法和数组语法进行动态绑定：

对象语法：

```
data: {
    isActive: true,
    hasError: false
}
```

数组语法：

```
data: {
  activeClass: 'active',
  errorClass: 'text-danger'
}
```

Style 也可以通过对象语法和数组语法进行动态绑定：

对象语法：

```
data: {
  activeColor: 'red',
  fontSize: 30
}
```

数组语法：

```
data: {
  styleColor: {
     color: 'red'
   },
  styleSize:{
     fontSize:'23px'
  }
```

怎样理解 Vue 的单向数据流?

所有的 prop 都使得其父子 prop 之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。
这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。
额外的，每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。
这意味着你不应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器的控制台中发出警告。
子组件想修改时，只能通过 $emit 派发一个自定义事件，父组件接收到后，由父组件修改。

computed 和 watch 的区别和运用的场景?

computed： 是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值；
watch： 更多的是「观察」的作用，类似于某些数据的监听回调 ，每当监听的数据变化时都会执行回调进行后续操作；
运用场景：
当我们需要进行数值计算，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；
当我们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，使用 watch 选项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。

直接给一个数组项赋值，Vue 能检测到变化吗?

由于 JavaScript 的限制，Vue 不能检测到以下数组的变动：

当你利用索引直接设置一个数组项时，例如：vm.items[indexOfItem] = newValue
当你修改数组的长度时，例如：vm.items.length = newLength
为了解决第一个问题，Vue 提供了以下操作方法：

```
// Vue.set
Vue.set(vm.items, indexOfItem, newValue)
// vm.$set，Vue.set的一个别名
vm.$set(vm.items, indexOfItem, newValue)
// Array.prototype.splice
vm.items.splice(indexOfItem, 1, newValue)
```

为了解决第二个问题，Vue 提供了以下操作方法：

```
// Array.prototype.splice
vm.items.splice(newLength)
```

谈谈你对 Vue 生命周期的理解?

生命周期是什么?
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，我们称这是 Vue 的生命周期。

- **各个生命周期的作用**

生命周期	描述
beforeCreate	组件实例被创建之初，组件的属性生效之前
created	组件实例已经完全创建，属性也绑定，但真实 dom 还没有生成，$el 还不可用
beforeMount	在挂载开始之前被调用：相关的 render 函数首次被调用
mounted	el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子
beforeUpdate	组件数据更新之前调用，发生在虚拟 DOM 打补丁之前
updated	组件数据更新之后
activited	keep-alive 专属，组件被激活时调用
deadctivated	keep-alive 专属，组件被销毁时调用
beforeDestory	组件销毁前调用
destoryed	组件销毁后调用

Vue 的父组件和子组件生命周期钩子函数执行顺序?

Vue 的父组件和子组件生命周期钩子函数执行顺序可以归类为以下 4 部分：

加载渲染过程：
父 beforeCreate -> 父 created -> 父 beforeMount -> 子 beforeCreate -> 子 created -> 子 beforeMount -> 子 mounted -> 父 mounted
子组件更新过程：
父 beforeUpdate -> 子 beforeUpdate -> 子 updated -> 父 updated
父组件更新过程：
父 beforeUpdate -> 父 updated
销毁过程：
父 beforeDestroy -> 子 beforeDestroy -> 子 destroyed -> 父 destroyed

父组件可以监听到子组件的生命周期吗?

比如有父组件 Parent 和子组件 Child，如果父组件监听到子组件挂载 mounted 就做一些逻辑处理，可以通过以下写法实现：

```
// Parent.vue
<Child @mounted="doSomething"/>

// Child.vue
mounted() {

  this.$emit("mounted");

}
```

以上需要手动通过 $emit 触发父组件的事件，更简单的方式可以在父组件引用子组件时通过 @hook 来监听即可，如下所示：

```
//  Parent.vue
<Child @hook:mounted="doSomething" ></Child>

doSomething() {
   console.log('父组件监听到 mounted 钩子函数 ...');
},

//  Child.vue
mounted(){
   console.log('子组件触发 mounted 钩子函数 ...');
},

// 以上输出顺序为：
// 子组件触发 mounted 钩子函数 ...
// 父组件监听到 mounted 钩子函数 ...
```

当然 @hook 方法不仅仅是可以监听 mounted，其它的生命周期事件，例如：created，updated 等都可以监听。

谈谈你对 keep-alive 的了解?

keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，避免重新渲染 ，其有以下特性：

一般结合路由和动态组件一起使用，用于缓存组件；
提供 include 和 exclude 属性，两者都支持字符串或正则表达式， include 表示只有名称匹配的组件会被缓存，exclude 表示任何名称匹配的组件都不会被缓存 ，其中 exclude 的优先级比 include 高；
对应两个钩子函数 activated 和 deactivated ，当组件被激活时，触发钩子函数 activated，当组件被移除时，触发钩子函数 deactivated。

组件中 data 为什么是一个函数?

为什么组件中的 data 必须是一个函数，然后 return 一个对象，而 new Vue 实例里，data 可以直接是一个对象?
因为组件是用来复用的，且 JS 里对象是引用关系，如果组件中 data 是一个对象，那么这样作用域没有隔离，子组件中的 data 属性值会相互影响，
如果组件中 data 选项是一个函数，那么每个实例可以维护一份被返回对象的独立的拷贝，组件实例之间的 data 属性值不会互相影响；而 new Vue 的实例，是不会被复用的，因此不存在引用对象的问题。

v-model 的原理?

我们在 vue 项目中主要使用 v-model 指令在表单 input、textarea、select 等元素上创建双向数据绑定，我们知道 v-model 本质上不过是语法糖，v-model 在内部为不同的输入元素使用不同的属性并抛出不同的事件：

text 和 textarea 元素使用 value 属性和 input 事件；
checkbox 和 radio 使用 checked 属性和 change 事件；
select 字段将 value 作为 prop 并将 change 作为事件。
以 input 表单元素为例：

1
相当于

1
如果在自定义组件中，v-model 默认会利用名为 value 的 prop 和名为 input 的事件，如下所示：

父组件：

子组件：

```
<div>{{value}}</div>

props:{
    value: String
},
methods: {
  test1(){
     this.$emit('input', '小红')
  },
},

```

Vue 组件间通信有哪几种方式?

Vue 组件间通信是面试常考的知识点之一，这题有点类似于开放题，你回答出越多方法当然越加分，表明你对 Vue 掌握的越熟练。

Vue 组件间通信只要指以下 3 类通信：父子组件通信、隔代组件通信、兄弟组件通信，下面我们分别介绍每种通信方式且会说明此种方法可适用于哪类组件间通信。
（1）props / $emit 适用 父子组件通信

这种方法是 Vue 组件的基础，相信大部分同学耳闻能详，所以此处就不举例展开介绍。
（2）ref 与 $parent/$children适用 父子组件通信

ref：如果在普通的 DOM 元素上使用，引用指向的就是 DOM 元素；如果用在子组件上，引用就指向组件实例
$parent/$children：访问父 / 子实例
（3）EventBus （$emit/$on）适用于 父子、隔代、兄弟组件通信

这种方法通过一个空的 Vue 实例作为中央事件总线（事件中心），用它来触发事件和监听事件，从而实现任何组件间的通信，包括父子、隔代、兄弟组件。
（4）$attrs/$listeners适用于 隔代组件通信

$attrs$：包含了父作用域中不被$prop$所识别(且获取)的特性绑定($class$和$style$除外)。当一个组件没有声明任何$prop$时，这里会包含所有父作用域的绑定($cl
attrs" 传入内部组件。通常配合 inheritAttrs 选项一起使用。
$listeners$：包含了父作用域中的(不含$.native$修饰器的)$v-on$事件监听器。它可以通过$v-on =$"listeners" 传入内部组件
（5）provide / inject适用于 隔代组件通信

祖先组件中通过 provider 来提供变量，然后在子孙组件中通过 inject 来注入变量。provide / inject API 主要解决了跨级组件间的通信问题，不过它的使用场景，主要是子组件获取上级组件的状态，跨级组件间建立了一种主动提供与依赖注入的关系。
（6）Vuex适用于 父子、隔代、兄弟组件通信

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。每一个 Vuex 应用的核心就是 store（仓库）。“store” 基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。
Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。
改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样使得我们可以方便地跟踪每一个状态的变化。

使用过 Vue SSR 吗？说说 SSR?

Vue.js 是构建客户端应用程序的框架。默认情况下，可以在浏览器中输出 Vue 组件，进行生成 DOM 和操作 DOM。然而，也可以将同一个组件渲染为服务端的 HTML 字符串，将它们直接发送到浏览器，最后将这些静态标记"激活"为客户端上完全可交互的应用程序。
即：SSR大致的意思就是vue在客户端将标签渲染成的整个 html 片段的工作在服务端完成，服务端形成的html 片段直接返回给客户端这个过程就叫做服务端渲染。
服务端渲染 SSR 的优缺点如下：

（1）服务端渲染的优点：
更好的 SEO：因为 SPA 页面的内容是通过 Ajax 获取，而搜索引擎爬取工具并不会等待 Ajax 异步完成后再抓取页面内容，所以在 SPA 中是抓取不到页面通过 Ajax 获取到的内容；而 SSR 是直接由服务端返回已经渲染好的页面（数据已经包含在页面中），所以搜索引擎爬取工具可以抓取渲染好的页面；
更快的内容到达时间（首屏加载更快）：SPA 会等待所有 Vue 编译后的 js 文件都下载完成后，才开始进行页面的渲染，文件下载等需要一定的时间等，所以首屏渲染需要一定的时间；SSR 直接由服务端渲染好页面直接返回显示，无需等待下载 js 文件及再去渲染等，所以 SSR 有更快的内容到达时间；
(2) 服务端渲染的缺点：
更多的开发条件限制：例如服务端渲染只支持 beforCreate 和 created 两个钩子函数，这会导致一些外部扩展库需要特殊处理，才能在服务端渲染应用程序中运行；并且与可以部署在任何静态文件服务器上的完全静态单页面应用程序 SPA 不同，服务端渲染应用程序，需要处于 Node.js server 运行环境；
更多的服务器负载：在 Node.js 中渲染完整的应用程序，显然会比仅仅提供静态文件的 server 更加大量占用CPU 资源 (CPU-intensive - CPU 密集)，因此如果你预料在高流量环境 (high traffic) 下使用，请准备相应的服务器负载，并明智地采用缓存策略。

vue-router 路由模式有几种?

vue-router 有 3 种路由模式：hash、history、abstract，对应的源码如下所示：

```
switch (mode) {
  case 'history':
    this.history = new HTML5History(this, options.base)
    break
  case 'hash':
    this.history = new HashHistory(this, options.base, this.fallback)
    break
  case 'abstract':
    this.history = new AbstractHistory(this, options.base)
    break
  default:
    if (process.env.NODE_ENV !== 'production') {
      assert(false, `invalid mode: ${mode}`)
    }
```

其中，3 种路由模式的说明如下：

hash: 使用 URL hash 值来作路由。支持所有浏览器，包括不支持 HTML5 History Api 的浏览器；
history : 依赖 HTML5 History API 和服务器配置。具体可以查看 HTML5 History 模式；
abstract : 支持所有 JavaScript 运行环境，如 Node.js 服务器端。如果发现没有浏览器的 API，路由会自动强制进入这个模式.

能说下 vue-router 中常用的 hash 和 history 路由模式实现原理吗?

（1）hash 模式的实现原理

早期的前端路由的实现就是基于 location.hash 来实现的。其实现原理很简单，location.hash 的值就是 URL 中 # 后面的内容。比如下面这个网站，它的 location.hash 的值为 ‘#search’：
https://www.word.com#search
hash 路由模式的实现主要是基于下面几个特性：

URL 中 hash 值只是客户端的一种状态，也就是说当向服务器端发出请求时，hash 部分不会被发送；
hash 值的改变，都会在浏览器的访问历史中增加一个记录。因此我们能通过浏览器的回退、前进按钮控制hash 的切换；
可以通过 a 标签，并设置 href 属性，当用户点击这个标签后，URL 的 hash 值会发生改变；或者使用 JavaScript 来对 loaction.hash 进行赋值，改变 URL 的 hash 值；
我们可以使用 hashchange 事件来监听 hash 值的变化，从而对页面进行跳转（渲染）。

（2）history 模式的实现原理

HTML5 提供了 History API 来实现 URL 的变化。其中做最主要的 API 有以下两个：history.pushState() 和 history.repalceState()。这两个 API 可以在不进行刷新的情况下，操作浏览器的历史纪录。
唯一不同的是，前者是新增一个历史记录，后者是直接替换当前的历史记录，如下所示：
window.history.pushState(null, null, path);
window.history.replaceState(null, null, path);
history 路由模式的实现主要基于存在下面几个特性：

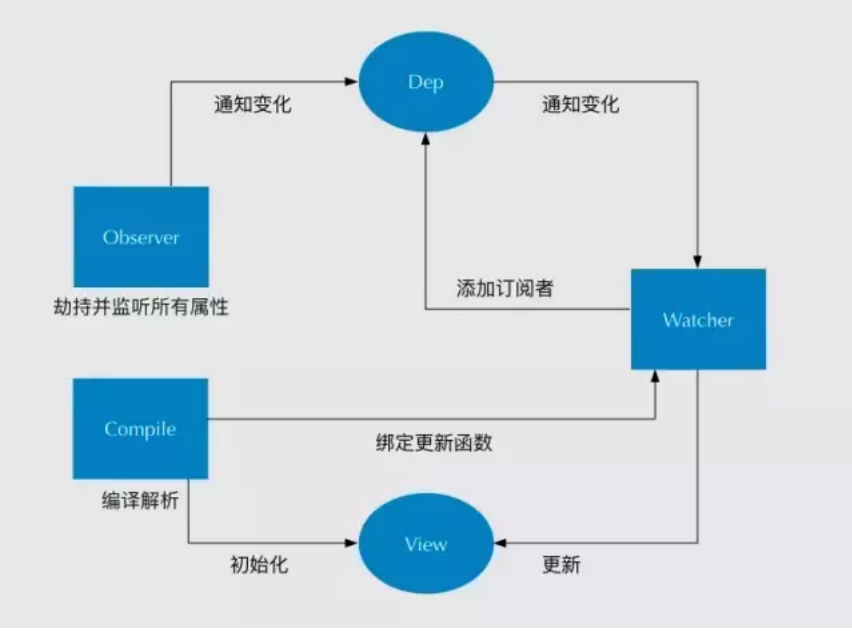
pushState 和 repalceState 两个 API 来操作实现 URL 的变化；
我们可以使用 popstate 事件来监听 url 的变化，从而对页面进行跳转（渲染）；
history.pushState() 或 history.replaceState() 不会触发 popstate 事件，这时我们需要手动触发页面跳转（渲染）。

Vue 框架怎么实现对象和数组的监听?

Vue 数据双向绑定主要是指：数据变化更新视图，视图变化更新数据。
即：

输入框内容变化时，Data 中的数据同步变化。即 View => Data 的变化。
Data 中的数据变化时，文本节点的内容同步变化。即 Data => View 的变化。
其中，View 变化更新 Data ，可以通过事件监听的方式来实现，所以 Vue 的数据双向绑定的工作主要是如何根据 Data 变化更新 View。
Vue 主要通过以下 4 个步骤来实现数据双向绑定的：

实现一个监听器 Observer：对数据对象进行遍历，包括子属性对象的属性，利用 Object.defineProperty() 对属性都加上 setter 和 getter。这样的话，给这个对象的某个值赋值，就会触发 setter，那么就能监听到了数据变化。
实现一个解析器 Compile：解析 Vue 模板指令，将模板中的变量都替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，调用更新函数进行数据更新。
实现一个订阅者 Watcher：Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁 ，主要的任务是订阅 Observer 中的属性值变化的消息，当收到属性值变化的消息时，触发解析器 Compile 中对应的更新函数。
实现一个订阅器 Dep：订阅器采用 发布-订阅 设计模式，用来收集订阅者 Watcher，对监听器 Observer 和 订阅者 Watcher 进行统一管理。



Vue 是如何实现数据双向绑定的?

如果被问到 Vue 怎么实现数据双向绑定，大家肯定都会回答 通过 Object.defineProperty() 对数据进行劫持，但是 Object.defineProperty() 只能对属性进行数据劫持，不能对整个对象进行劫持。
同理无法对数组进行劫持，但是我们在使用 Vue 框架中都知道，Vue 能检测到对象和数组（部分方法的操作）的变化，那它是怎么实现的呢？我们查看相关代码如下：

```
/**

   * Observe a list of Array items.
     */
       observeArray (items: Array<any>) {

    for (let i = 0, l = items.length; i < l; i++) {
      observe(items[i])  // observe 功能为监测数据的变化
    }

  }
```

```
12
13   /**
14
15    * 对属性进行递归遍历
16      */
17        let childOb = !shallow && observe(val) // observe 功能为监测数据的变化
```

通过以上 Vue 源码部分查看，我们就能知道 Vue 框架是通过遍历数组 和递归遍历对象，从而达到利用 Object.defineProperty() 也能对对象和数组（部分方法的操作）进行监听。

Vue 怎么用 vm.$set() 解决对象新增属性不能响应的问题？

受现代 JavaScript 的限制，Vue 无法检测到对象属性的添加或删除。
由于 Vue 会在初始化实例时对属性执行 getter/setter 转化，所以属性必须在 data 对象上存在才能让 Vue 将它转换为响应式的。
但是 Vue 提供了 Vue.set (object, propertyName, value) / vm.$set (object, propertyName, value)来实现为对象添加响应式属性，那框架本身是如何实现的呢?

我们查看对应的 Vue 源码：vue/src/core/instance/index.js

```
1  export function set (target: Array<any> | Object, key: any, val: any): any {
2    // target 为数组
3    if (Array.isArray(target) && isValidArrayIndex(key)) {
4      // 修改数组的长度, 避免索引>数组长度导致splcie()执行有误
5      target.length = Math.max(target.length, key)
6      // 利用数组的splice变异方法触发响应式
7      target.splice(key, 1, val)
8      return val
9    }
10   // key 已经存在，直接修改属性值
11   if (key in target && !(key in Object.prototype)) {
12     target[key] = val
13     return val
14   }
15   const ob = (target: any).__ob__
16   // target 本身就不是响应式数据, 直接赋值
17   if (!ob) {
18     target[key] = val
19     return val
20   }
21   // 对属性进行响应式处理
22   defineReactive(ob.value, key, val)
23   ob.dep.notify()
24   return val
25 }
```

我们阅读以上源码可知，vm.$set 的实现原理是：

如果目标是数组，直接使用数组的 splice 方法触发相应式；
如果目标是对象，会先判读属性是否存在、对象是否是响应式，最终如果要对属性进行响应式处理，则是通过调用 defineReactive 方法进行响应式处理（ defineReactive 方法就是 Vue 在初始化对象时，给对象属性采用 Object.defineProperty 动态添加 getter 和 setter 的功能所调用的方法）

虚拟 DOM 的优缺点?

优点：

保证性能下限： 框架的虚拟 DOM 需要适配任何上层 API 可能产生的操作，它的一些 DOM 操作的实现必须是普适的，所以它的性能并不是最优的；但是比起粗暴的 DOM 操作性能要好很多，因此框架的虚拟 DOM 至少可以保证在你不需要手动优化的情况下，依然可以提供还不错的性能，即保证性能的下限；
无需手动操作 DOM： 我们不再需要手动去操作 DOM，只需要写好 View-Model 的代码逻辑，框架会根据虚拟 DOM 和 数据双向绑定，帮我们以可预期的方式更新视图，极大提高我们的开发效率；
跨平台： 虚拟 DOM 本质上是 JavaScript 对象,而 DOM 与平台强相关，相比之下虚拟 DOM 可以进行更方便地跨平台操作，例如服务器渲染、weex 开发等等。
缺点:

无法进行极致优化： 虽然虚拟 DOM + 合理的优化，足以应对绝大部分应用的性能需求，但在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化。

虚拟 DOM 实现原理?

虚拟 DOM 的实现原理主要包括以下 3 部分：

用 JavaScript 对象模拟真实 DOM 树，对真实 DOM 进行抽象；
diff 算法 — 比较两棵虚拟 DOM 树的差异；
pach 算法 — 将两个虚拟 DOM 对象的差异应用到真正的 DOM 树。

Vue 中的 key 有什么作用?

key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速。
Vue 的 diff 过程可以概括为：oldCh 和 newCh 各有两个头尾的变量 oldStartIndex、oldEndIndex 和 newStartIndex、newEndIndex，它们会新节点和旧节点会进行两两对比，即一共有4种比较方式：newStartIndex 和oldStartIndex 、newEndIndex 和 oldEndIndex 、newStartIndex 和 oldEndIndex 、newEndIndex 和 oldStartIndex，如果以上 4 种比较都没匹配，如果设置了key，就会用 key 再进行比

较，在比较的过程中，遍历会往中间靠，一旦 StartIdx > EndIdx 表明 oldCh 和 newCh 至少有一个已经遍历完了，就会结束比较。
所以 Vue 中 key 的作用是：key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速!

更准确：因为带 key 就不是就地复用了，在 sameNode 函数 a.key === b.key 对比中可以避免就地复用的情况。所以会更加准确。
更快速：利用 key 的唯一性生成 map 对象来获取对应节点，比遍历方式更快，源码如下：

```
function createKeyToOldIdx (children, beginIdx, endIdx) {
  let i, key
  const map = {}
  for (i = beginIdx; i <= endIdx; ++i) {
    key = children[i].key
    if (isDef(key)) map[key] = i
  }
  return map
}
```

你有对 Vue 项目进行哪些优化?

（1）代码层面的优化

v-if 和 v-show 区分使用场景
computed 和 watch 区分使用场景
v-for 遍历必须为 item 添加 key，且避免同时使用 v-if
长列表性能优化
事件的销毁
图片资源懒加载
路由懒加载
第三方插件的按需引入
优化无限列表性能
服务端渲染 SSR or 预渲染
（2）Webpack 层面的优化

Webpack 对图片进行压缩
减少 ES6 转为 ES5 的冗余代码
提取公共代码
模板预编译
提取组件的 CSS
优化 SourceMap
构建结果输出分析
Vue 项目的编译优化
（3）基础的 Web 技术的优化

开启 gzip 压缩
浏览器缓存
CDN 的使用
使用 Chrome Performance 查找性能瓶颈

对于 vue3.0 特性你有什么了解的吗?

Vue 3.0 的目标是让 Vue 核心变得更小、更快、更强大，因此 Vue 3.0 增加以下这些新特性：

（1）监测机制的改变
3.0 将带来基于代理 Proxy 的 observer 实现，提供全语言覆盖的反应性跟踪。这消除了 Vue 2 当中基于 Object.defineProperty 的实现所存在的很多限制：

只能监测属性，不能监测对象
检测属性的添加和删除；
检测数组索引和长度的变更；
支持 Map、Set、WeakMap 和 WeakSet。
新的 observer 还提供了以下特性：

用于创建 observable 的公开 API。这为中小规模场景提供了简单轻量级的跨组件状态管理解决方案。
默认采用惰性观察。在 2.x 中，不管反应式数据有多大，都会在启动时被观察到。如果你的数据集很大，这可能会在应用启动时带来明显的开销。在 3.x 中，只观察用于渲染应用程序最初可见部分的数据。
更精确的变更通知。在 2.x 中，通过 Vue.set 强制添加新属性将导致依赖于该对象的 watcher 收到变更通知。在 3.x 中，只有依赖于特定属性的 watcher 才会收到通知。
不可变的 observable：我们可以创建值的“不可变”版本（即使是嵌套属性），除非系统在内部暂时将其“解禁”。这个机制可用于冻结 prop 传递或 Vuex 状态树以外的变化。
更好的调试功能：我们可以使用新的 renderTracked 和 renderTriggered 钩子精确地跟踪组件在什么时候以及为什么重新渲染。
（2）模板
模板方面没有大的变更，只改了作用域插槽，2.x 的机制导致作用域插槽变了，父组件会重新渲染，而 3.0 把作用域插槽改成了函数的方式，这样只会影响子组件的重新渲染，提升了渲染的性能。
同时，对于 render 函数的方面，vue3.0 也会进行一系列更改来方便习惯直接使用 api 来生成 vdom 。

（3）对象式的组件声明方式
vue2.x 中的组件是通过声明的方式传入一系列 option，和 TypeScript 的结合需要通过一些装饰器的方式来做，虽然能实现功能，但是比较麻烦。
3.0 修改了组件的声明方式，改成了类式的写法，这样使得和 TypeScript 的结合变得很容易。
此外，vue 的源码也改用了 TypeScript 来写。其实当代码的功能复杂之后，必须有一个静态类型系统来做一些辅助管理。

现在 vue3.0 也全面改用 TypeScript 来重写了，更是使得对外暴露的 api 更容易结合 TypeScript。静态类型系统对于复杂代码的维护确实很有必要。

（4）其它方面的更改
vue3.0 的改变是全面的，上面只涉及到主要的 3 个方面，还有一些其他的更改：

支持自定义渲染器，从而使得 weex 可以通过自定义渲染器的方式来扩展，而不是直接 fork 源码来改的方式。
支持 Fragment（多个根节点）和 Protal（在 dom 其他部分渲染组建内容）组件，针对一些特殊的场景做了处理。
基于 treeshaking 优化，提供了更多的内置功能。

响应式原理（变化侦测）

使用发布订阅模式将数据劫持和模板编译结合，实现双向绑定

1、observer: 封装 Object.defineProperty 方法用来劫持对象属性的getter和setter，以此来追踪数据变化。

2、读取数据时触发getter来收集依赖(Watcher)到Dep。
3、修改数据时触发setter，并遍历依赖列表，通知所有相关依赖（Watcher）
4、Dep 类为依赖找一个存储依赖的地方，用来收集和管理依赖，在getter中收集，在setter中通知。
5、Watcher 类就是收集的依赖，实际上是一个订阅器，Watcher会将自己的实例赋值给window.target（全局变量）上，然后去主动访问属性，触发属性的getter，getter中会将此Watcher收集到Dep中，Watcher的update方法会在Dep的通知方法中被调用，触发更新。
6、Observer 类用来将一个对象的所有属性和子属性都变成响应式的，通过递归调用defineReactive来实现。
7、由于无法侦测对象上新增/删除属性，所以提供 set和delete API5。

Object.defineProperty怎么用， 三个参数? ，有什么作用啊?

Object.defineProperty() 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回此对象。

```
obj：需要定义属性的对象
prop：需要定义的属性
{}：要定义或修改的属性描述符。
    value: "18",          // 设置默认值 （与 get() 互斥）
    enumerable: true,     //这一句控制属性可以枚举 enumerable 改为true 就可以参与遍历了    默认值false
    writable: true,       // 该属性是否可写    默认值false （与 set() 互斥）
    configurable: true,  // 该属性是否可被删除    默认值false
     get // 当有人读取 prop 的时候  get函数就会调用,并且返回就是 sss 的值
     set // 当有人修改 prop 的时候  set函数就会调用，有个参数这个参数就是修改后的值
```

vue2和vue3的响应式原理都有什么区别呢?

vue2 用的是 Object.defindProperty 但是vue3用的是Proxy

Object.defindProperty 缺点：

一次只能对一个属性进行监听，需要遍历来对所有属性监听
对于对象的新增属性，需要手动监听
对于数组通过push、unshift方法增加的元素，也无法监听
Proxy就没有这个问题，可以监听整个对象的数据变化，所以用vue3.0会用Proxy代替definedProperty。

30. Vue的patch diff 算法
    patch将新老VNode节点进行比对，然后将根据两者的比较结果进行最小单位地修改视图，而不是将整个视图根据新的VNode重绘。
    patch的核心在于diff算法，这套算法可以高效地比较virtual DOM的变更，得出变化以修改视图。

diff算法核心是通过同层的树节点进行比较而非对树进行逐层搜索遍历的方式，所以时间复杂度只有O(n)，是一种相当高效的算法。

同层级比较（只比较同一层级，不跨级比较）
tag 不相同，则直接删除重建，不在深度比较
tag 和 key，两个都相同，则认为是相同节点，会进行深度比较

Vue 模板编译原理

模板字符串 转换成 element AST（解析器）
Vue-loader 切割解析 .vue 文件（parseHTML按标签以出栈入栈形式切割（自闭合不入栈直接处理），出栈时维护父子关系）生成 AST（抽象语法树）

使用大量正则匹配开始结束标签，while指针定位解析位置，

对 AST 进行静态节点标记，主要用来做虚拟DOM的渲染优化（优化器）
在dom更新时不需 diff 静态节点。

使用 element AST 生成 render 函数代码字符串（代码生成器）
Vue-template-compiler再解析成render（可执行函数字符串-with(this)=>{return _c('div')}），new Function 生成函数，传递给组件的 render

在组件渲染的时候直接调用 render 即可

Vue原理总结

【模板编译】将template模板，经过编译系统后生成VNode，（模板字符串→AST→Render函数）

【渲染】然后再通过渲染系统来将VNode生成真实DOM（document.createElement && Mount挂载到真实DOM节点上）

【响应式】通过响应式系统对数据进行监听，当数据发生改变时，触发依赖项（组件）

【Diff & Patch】组件内收到通知后，会通过diff算法对比VNode的变化，尽可能复用代码，找出最小差异，保证性能消耗最小。

【渲染】拿到需要新增/删除/修改的VNode后，逐一去操作真实DOM进行修改（通过选择器选择到对应真实DOM节点进行修改）

webpack

谈谈你对Webpack的理解（Webpack是什么？）

Webpack 是一个 模块打包器，可以分析各个模块的依赖关系，最终打包成bundle静态文件（js、css、jpg）。

webpack 是一个静态模块打包器，当 webpack 处理应用程序时，会递归构建一个依赖关系图，其中包含应用程序需要的每个模块，然后将这些模块打包成一个或多个 bundle。

webpack 就像一条生产线,要经过一系列处理流程(loader)后才能将源文件转换成输出结果。 这条生产线上的每个处理流程的职责都是单一的,多个流程之间有存在依赖关系,只有完成当前处理后才能交给下一个流程去处理。
插件就像是一个插入到生产线中的一个功能,在特定的时机对生产线上的资源做处理。 webpack 在运行过程中会广播事件,插件只需要监听它所关心的事件,就能加入到这条生产线中,去改变生产线的运作。

Webpack的打包过程/打包原理/构建流程?

初始化：启动构建，读取与合并配置参数，加载plugin,实例化Compiler
编译：从Entry出发，针对每个Module串行调用对应的Loader去翻译文件中的内容，再找到该Module依赖的Module，递归的进行编译处理
输出：将编译后的Module组合成Chunk,将Chunk转换成文件，输出到文件系统中
细节：

Webpack CLI 通过 yargs模块解析 CLI 参数，并转化为配置对象option（单入口：Object，多入口：Array），调用 webpack(option) 创建 compiler 对象。

如果有 option.plugin，则遍历调用plugin.apply()来注册 plugin，

判断是否开启了 watch，如果开启则调用 compiler.watch，否则调用 compiler.run，开始构建。

创建 Compilation 对象来收集全部资源和信息，然后触发 make 钩子。

make阶段从入口开始递归所有依赖，

每次遍历时调用对应Loader翻译文件中内容，然后生成AST，遍历AST找到下个依赖继续递归，

根据入口和模块之间关系组装chunk，输出到dist中的一个文件内。

在以上过程中，webpack会在特定的时间点（使用tapable模块）广播特定的事件，插件监听事件并执行相应的逻辑，并且插件可以调用webpack提供的api改变webpack的运行结果

loader的作用

webpack中的loader是一个函数，主要为了实现源码的转换，所以loader函数会以源码作为参数，比如，将ES6转换为ES5，将less转换为css，然后再将css转换为js，以便能嵌入到html文件中。

有哪些常见的Loader？他们是解决什么问题的?

file-loader：把文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件
url-loader：和 file-loader 类似，但是能在文件很小的情况下以 base64 的方式把文件内容注入到代码中去
source-map-loader：加载额外的 Source Map 文件，以方便断点调试
image-loader：加载并且压缩图片文件
babel-loader：把 ES6 转换成 ES5
css-loader：加载 CSS，支持模块化、压缩、文件导入等特性
style-loader：把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS。
eslint-loader：通过 ESLint 检查 JavaScript 代码

plugin的作用

plugin是一个类，类中有一个apply()方法，主要用于Plugin的安装，可以在其中监听一些来自编译器发出的事件，在合适的时机做一些事情。

有哪些常见的Plugin？他们是解决什么问题的?

html-webpack-plugin：可以复制一个有结构的html文件，并自动引入打包输出的所有资源（JS/CSS）
clean-webpack-plugin：重新打包自动清空 dist 目录
mini-css-extract-plugin：提取 js 中的 css 成单独文件
optimize-css-assets-webpack-plugin：压缩css
uglifyjs-webpack-plugin：压缩js
commons-chunk-plugin：提取公共代码
define-plugin：定义环境变量

Webpack中Loader和Plugin的区别

运行时机1.loader运行在编译阶段2.plugins 在整个周期都起作用

使用方式Loader:1.下载 2.使用Plugin:1.下载 2.引用 3.使用

webpack的热更新是如何做到的？说明其原理？

1、在 webpack 的 watch 模式下，文件系统中某一个文件发生修改，webpack 监听到文件变化，根据配置文件对模块重新编译打包，并将打包后的代码通过简单的 JavaScript 对象保存在内存中。
2、webpack-dev-server 和 webpack 之间的接口交互，而在这一步，主要是 dev-server 的中间件webpack-dev-middleware 和 webpack 之间的交互，webpack-dev-middleware 调用 webpack 暴露的 API对代码变化进行监控，并且告诉 webpack，将代码打包到内存中。
3、webpack-dev-server 对文件变化的一个监控，这一步不同于第一步，并不是监控代码变化重新打包。当我们在配置文件中配置了 devServer.watchContentBase 为 true 的时候，Server 会监听这些配置文件夹中静态文件的变化，变化后会通知浏览器端对应用进行 live reload。注意，这儿是浏览器刷新，和 HMR 是两个概念
4、webpack-dev-server 代码的工作，该步骤主要是通过 sockjs（webpack-dev-server 的依赖）在浏览器端和服务端之间建立一个 websocket 长连接，将 webpack 编译打包的各个阶段的状态信息告知浏览器端，
同时也包括第三步中 Server 监听静态文件变化的信息。浏览器端根据这些 socket 消息进行不同的操作。当然服务端传递的最主要信息还是新模块的 hash 值，后面的步骤根据这一 hash 值来进行模块热替换。
webpack-dev-server/client 端并不能够请求更新的代码，也不会执行热更模块操作，而把这些工作又交回给了 webpack，
webpack/hot/dev-server 的工作就是根据 webpack-dev-server/client 传给它的信息以及 dev-server 的配置决定是刷新浏览器呢还是进行模块热更新。当然如果仅仅是刷新浏览器，也就没有后面那些步骤了。HotModuleReplacement.runtime 是客户端 HMR 的中枢，它接收到上一步传递给他的新模块的 hash 值，它通过 JsonpMainTemplate.runtime 向 server 端发送 Ajax 请求，服务端返回一个 json，该 json 包含了所有要更新的模块的 hash 值，获取到更新列表后，该模块再次通过 jsonp 请求，获取到最新的模块代码。
5、决定 HMR 成功与否的关键步骤，在该步骤中，HotModulePlugin 将会对新旧模块进行对比，决定是否更新模块，在决定更新模块后，检查模块之间的依赖关系，更新模块的同时更新模块间的依赖引用。最后一步，当 HMR 失败后，回退到 live reload 操作，也就是进行浏览器刷新来获取最新打包代码。

如何解决循环依赖问题

Webpack 中将 require 替换为 webpack_require，会根据 moduleId 到 installedModules 找是否加载过，加载过则直接返回之前的 export，不会重复加载。

如何提高Webpack构建速度

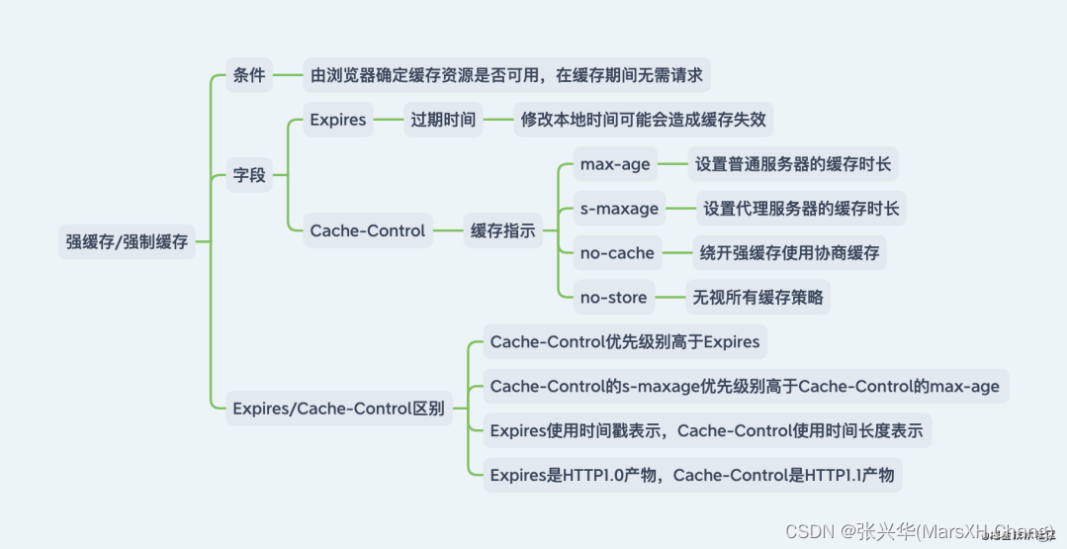
组件懒加载、路由懒加载、开启gzip、公共的第三方包上cdn、配置include/exclude缩小Loader对文件的搜索范围、配置cache缓存Loader对文件的编译副本、配置resolve提高文件的搜索速度（@: src）

性能优化篇

1. ## 浏览器缓存优化

为了让浏览器缓存发挥最大作用，该策略尽量遵循以下五点就能发挥浏览器缓存最大作用。

「考虑拒绝一切缓存策略」：Cache-Control:no-store
「考虑资源是否每次向服务器请求」：Cache-Control:no-cache
「考虑资源是否被代理服务器缓存」：Cache-Control:public/private
「考虑资源过期时间」：Expires:t/Cache-Control:max-age=t,s-maxage=t
「考虑协商缓存」：Last-Modified/Etag
缓存策略通过设置HTTP报文实现，在形式上分为**「强缓存/强制缓存」和「协商缓存/对比缓存」**。为了方便对比，笔者将某些细节使用图例展示，相信你有更好的理解。



整个缓存策略机制很明了，先走强缓存，若命中失败才走协商缓存。若命中强缓存，直接使用强缓存；若未命中强缓存，发送请求到服务器检查是否命中协商缓存；若命中协商缓存，服务器返回304通知浏览器使用本地缓存，否则返回最新资源。

有两种较常用的应用场景值得使用缓存策略一试，当然更多应用场景都可根据项目需求制定。

「频繁变动资源」：设置Cache-Control:no-cache，使浏览器每次都发送请求到服务器，配合Last-Modified/ETag验证资源是否有效
「不常变化资源」：设置Cache-Control:max-age=31536000，对文件名哈希处理，当代码修改后生成新的文件名，当HTML文件引入文件名发生改变才会下载最新文件

渲染层面性能优化

「渲染层面」的性能优化，无疑是如何让代码解析更好执行更快。因此笔者从以下五方面做出建议。

「CSS策略」：基于CSS规则
「DOM策略」：基于DOM操作
「阻塞策略」：基于脚本加载
「回流重绘策略」：基于回流重绘
「异步更新策略」：基于异步更新
上述五方面都是编写代码时完成，充满在整个项目流程的开发阶段里。因此在开发阶段需时刻注意以下涉及到的每一点，养成良好的开发习惯，性能优化也自然而然被使用上了。

渲染层面的性能优化更多表现在编码细节上，而并非实体代码。简单来说就是遵循某些编码规则，才能将渲染层面的性能优化发挥到最大作用。

「回流重绘策略」在渲染层面的性能优化里占比较重，也是最常规的性能优化之一。上年笔者发布的掘金小册《玩转CSS的艺术之美》使用一整章讲解回流重绘，本章已开通试读，更多细节请戳这里。

CSS策略
避免出现超过三层的嵌套规则
避免为ID选择器添加多余选择器
避免使用标签选择器代替类选择器
避免使用通配选择器，只对目标节点声明规则
避免重复匹配重复定义，关注可继承属性
DOM策略
缓存DOM计算属性
避免过多DOM操作
使用DOMFragment缓存批量化DOM操作
阻塞策略
脚本与DOM/其它脚本的依赖关系很强：对设置defer
脚本与DOM/其它脚本的依赖关系不强：对设置async
回流重绘策略
缓存DOM计算属性
使用类合并样式，避免逐条改变样式
使用display控制DOM显隐，将DOM离线化
异步更新策略
在异步任务中修改DOM时把其包装成微任务

性能优化六大指标

六大指标基本囊括大部分性能优化细节，可作为九大策略的补充。笔者根据每条性能优化建议的特征将指标划分为以下六方面。

「加载优化」：资源在加载时可做的性能优化
「执行优化」：资源在执行时可做的性能优化
「渲染优化」：资源在渲染时可做的性能优化
「样式优化」：样式在编码时可做的性能优化
「脚本优化」：脚本在编码时可做的性能优化
「V8引擎优化」：针对V8引擎特征可做的性能优化

其他杂项篇

常见的浏览器内核有哪些?

主要分成两部分：渲染引擎(layout engineer或Rendering Engine)和JS引擎。
渲染引擎：负责取得网页的内容（HTML、XML、图像等等）、整理讯息（例如加入CSS等），以及计算网页的显示方式，然后会输出至显示器或打印机。浏览器的内核的不同对于网页的语法解释会有不同，所以渲染的效果也不相同。所有网页浏览器、电子邮件客户端以及其它需要编辑、显示网络内容的应用程序都需要内核。
JS引擎则：解析和执行javascript来实现网页的动态效果。
最开始渲染引擎和JS引擎并没有区分的很明确，后来JS引擎越来越独立，内核就倾向于只指渲染引擎。
常见内核
Trident 内核：IE, MaxThon, TT, The World, 360, 搜狗浏览器等。[又称 MSHTML]
Gecko 内核：Netscape6 及以上版本，FF, MozillaSuite / SeaMonkey 等
Presto 内核：Opera7 及以上。 [Opera内核原为：Presto，现为：Blink;]
Webkit 内核：Safari, Chrome等。 [Chrome的：Blink（WebKit 的分支）]

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

网页从输入网址到渲染完成经历了哪些过程?

大致可以分为如下7步：

输入网址；
发送到DNS服务器，并获取域名对应的web服务器对应的ip地址；
与web服务器建立TCP连接；
浏览器向web服务器发送http请求；
web服务器响应请求，并返回指定url的数据（或错误信息，或重定向的新的url地址）；
浏览器下载web服务器返回的数据及解析html源文件；
生成DOM树，解析css和js，渲染页面，直至显示完成；

线程与进程的区别?

一个程序至少有一个进程,一个进程至少有一个线程.
线程的划分尺度小于进程，使得多线程程序的并发性高。
另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。
线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。
从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

HTTP常见的状态码?

100 Continue 继续，一般在发送post请求时，已发送了http header之后服务端将返回此信息，表示确认，之后发送具体参数信息
200 OK 正常返回信息
201 Created 请求成功并且服务器创建了新的资源
202 Accepted 服务器已接受请求，但尚未处理
301 Moved Permanently 请求的网页已永久移动到新位置。
302 Found 临时性重定向。
303 See Other 临时性重定向，且总是使用 GET 请求新的 URI。
304 Not Modified 自从上次请求后，请求的网页未修改过。
400 Bad Request 服务器无法理解请求的格式，客户端不应当尝试再次使用相同的内容发起请求。
401 Unauthorized 请求未授权。
403 Forbidden 禁止访问。
404 Not Found 找不到如何与 URI 相匹配的资源。
500 Internal Server Error 最常见的服务器端错误。
503 Service Unavailable 服务器端暂时无法处理请求（可能是过载或维护）。

图片懒加载?

当页面滚动的时间被触发 -> 执行加载图片操作 -> 判断图片是否在可视区域内 -> 在，则动态将data-src的值赋予该图片

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时间代替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

TCP 传输的三次握手、四次挥手策略

三次握手：
为了准确无误地吧数据送达目标处，TCP协议采用了三次握手策略。用TCP协议把数据包送出去后，TCP不会对传送后的情况置之不理，他一定会向对方确认是否送达，握手过程中使用TCP的标志：SYN和ACK
发送端首先发送一个带SYN的标志的数据包给对方
接收端收到后，回传一个带有SYN/ACK标志的数据包以示传达确认信息
最后，发送端再回传一个带ACK的标志的数据包，代表"握手"结束
如在握手过程中某个阶段莫明中断，TCP协议会再次以相同的顺序发送相同的数据包
断开一个TCP连接需要"四次挥手"
第一次挥手：主动关闭方发送一个FIN，用来关注主动方到被动关闭方的数据传送，也即是主动关闭方告诫被动关闭方：我已经不会再给你发数据了（在FIN包之前发送的数据，如果没有收到对应的ACK确认报文，主动关闭方依然会重发这些数据）。但是，此时主动关闭方还可以接受数据
第二次挥手：被动关闭方收到FIN包后，发送一个ACK给对方，确认序号收到序号 +1（与SYN相同，一个 FIN占用一个序号）
第三次挥手：被动关闭方发送一个 FIN。用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉主动关闭方，我的数据也发送完了，不会给你发送数据了
第四次挥手：主动关闭方收到FIN后，发送一个ACK给被动关闭方，确认序号为收到序号+1，至此，完成四次挥手

HTTP 和 HTTPS，为什么HTTPS安全?

HTTP协议通常承载与 TCP协议之上，在HTTP和TCP之间添加一个安全协议层（SSL或TSL），这个时候，就成了我们常说的HTTPS
默认HTTP的端口号为80，HTTPS的端口号为443
因为网络请求需要中间有很多的服务器路由的转发，中间的节点都可能篡改信息，而如果使用HTTPS，密钥在你和终点站才有，https之所有说比http安全，是因为他利用ssl/tls协议传输。包含证书，流量转发，负载均衡，页面适配，浏览器适配，refer传递等，保障了传输过程的安全性

axios和fetch区别对比

axios 是一个基于Promise 用于浏览器和 nodejs 的 HTTP 客户端，本质上也是对原生XHR的封装，只不过它是Promise的实现版本，符合最新的ES规范，它本身具有以下特征

从浏览器中创建 XMLHttpRequest
支持 Promise API
客户端支持防止CSRF
提供了一些并发请求的接口（重要，方便了很多的操作）
从 node.js 创建 http 请求
拦截请求和响应
转换请求和响应数据
取消请求
自动转换JSON数据
fetch优势：

语法简洁，更加语义化
基于标准 Promise 实现，支持 async/await
同构方便，使用 isomorphic-fetch
更加底层，提供的API丰富（request, response）
脱离了XHR，是ES规范里新的实现方式
fetch存在问题

fetch是一个低层次的API，你可以把它考虑成原生的XHR，所以使用起来并不是那么舒服，需要进行封装。
fetch只对网络请求报错，对400，500都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。
fetch默认不会带cookie，需要添加配置项： fetch(url, {credentials: 'include'})
fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费
fetch没有办法原生监测请求的进度，而XHR可以

主观题篇

你都做过什么项目呢？具体聊某一个项目中运用的技术.

注意：用心找自己做的项目中自己感觉最拿出来手的（复杂度最高，用的技术最多的项目），描述的时候尽可能往里面添加一些技术名词
布局我们用html5+css3
我们会用reset.css重置浏览器的默认样式
JS框架的话我们选用的是jQuery(也可能是Zepto)
我们用版本控制工具git来协同开发
我们会基于gulp搭建的前端自动化工程来开发（里面包含有我们的项目结构、我们需要引用的第三方库等一些信息，我们还实现了sass编译、CSS3加前缀等的自动化）
我们的项目中还用到了表单验证validate插件、图片懒加载Lazyload插件

1. 你遇到过比较难的技术问题是？你是如何解决的?
2. 常使用的库有哪些？常用的前端开发工具？开发过什么应用或组件?
3. 除了前端以外还了解什么其它技术么？你最最厉害的技能是什么?

4. 对前端开发工程师这个职位是怎么样理解的？它的前景会怎么样？

   前端是最贴近用户的程序员，比后端、数据库、产品经理、运营、安全都近。

   1、实现界面交互

   2、提升用户体验

   3、有了Node.js，前端可以实现服务端的一些事情

   前端是最贴近用户的程序员，前端的能力就是能让产品从 90分进化到 100 分，甚至更好，

   参与项目，快速高质量完成实现效果图，精确到1px；

   与团队成员，UI设计，产品经理的沟通；

   做好的页面结构，页面重构和用户体验；

   处理hack，兼容、写出优美的代码格式；

   针对服务器的优化、拥抱最新前端技术。