

面试问题：

css相关

1. css清除浮动?

1. 通过清除浮动，来防止父级元素高度塌陷，造成页面混乱，有以下几种方式来清除浮动：

1. 给浮动元素的父元素添加高度

2. 给父元素添加overflow:hidden触发父级BFC

1. BFC：**块级格式化上下文**，是一个独立的渲染区域，让处于 BFC 内部的元素与外部的元素相互隔离，使内外元素的定位不会相互影响。

3. 通过伪元素:after / :before 结合 clearfix来实现

2. CSS3的新特性?

1. 过渡：transition

2. 动画：animation

3. 变形：transform

4. 阴影：box-shadow

5. 颜色：rgba

6. 弹性布局：flex

7. 栅格布局：grid

3. CSS布局你用过哪些？适配方案？

1. table,早期，特点：内容可以自动居中，缺点：table会阻止浏览器渲染引擎的渲染顺序，会延迟页面的生成速度，让用户等待更久的时间，体验不好。

2. float，浮动

3. display：block，将元素转为块级元素(独占一行)，inline：将元素转为行内元素(不能设置宽高)，inline-block：行内块，可设置宽高，又在同一行

4. position，定位，static，默认，relative：相对自身定位，absolute：绝对定位，相对最近的父级relative或absolute定位元素，fixed：固定定位，相对于视口，可通过z-index来改变层叠顺序

5. flex，弹性盒，特点：弹性盒是CSS3中新增的一种布局方式，当一个元素设为Flex后，它的子元素的float、clear和vertical-align属性都将失效。

6. 响应式布局(移动端)

4. div如何做垂直水平居中？

1. 使用绝对定位，先将父元素开启相对定位，自身元素再开启绝对定位，left right top bottom都设置为0，margin设置为auto

2. transform，父元素开启相对定位，自身开启绝对定位，left: 50%,top:50%,transform: translate(-50%, -50%);

3. 开启弹性盒，设置主轴居中：justify-content: center;align-items: center;侧轴居中

5. CSS3常见的选择器？

1. 基本选择器：

1. 元素选择器

2. id选择器

3. 类选择器

4. 通配选择器

5. 属性选择器

2. 复合选择器：

1. 交集：同时选中多个选择器元素，语法：选择器1选择器2{}
2. 并集：可以选中多个选择器对应的元素，语法：选择器1，选择器2，选择器3{}
3. 关系选择器
4. 后代元素选择器
5. 子元素选择器 >
6. 相邻兄弟元素选择器： item +
7. 通用兄弟选择器： ~

6. CSS所用到的框架？

1. Bootstrap
2. jQuery-ui
3. Foundation
4. Ant Design
5. UIKit

7. 介绍一下双飞翼布局？

1. 就是将一个盒子，分为了左、中、右三块，左右宽度使用margin固定，中间部分自适应，使用双飞翼布局不需要定位，使用场景也比较多，是我们布局经常用的方式之一。

8. 原生CSS如何做响应式布局(圣杯和双飞翼)？

1. 圣杯布局和双飞翼布局都是实现两边固定中间自适应的三栏式布局
2. 圣杯布局：

```
<div class="box">
  <div class="main">Main</div>
  <div class="aside left">Left</div>
  <div class="aside right">Right</div>
</div>
<!-- 之所以将main元素放在left和right之前，是要保证网页在加载过程中首先加载网页的主体内容，使其首先渲染，避免网页加载时间过长导致主要部分出现空白 -->
.box {
  height: 65px;
  padding-left: 150px; //added
  padding-right: 190px; //added
  line-height: 65px;
  text-align: center;
}
.main {
  float: left;
  width: 100%;
  height: 100%;
  background-color: #0088CC;
}
.aside {
  position: relative; //added
  float: left;
  height: 100%;
}
.left {
  width: 140px;
  left: -150px; //added
  margin-left: -100%;
  background-color: #31B0D5;
}
.right {
```

```
width: 180px;
right: -190px; //added
margin-left: -180px;
background-color: #39B3D7;
}
```

3. 双飞翼布局

```
<div class="box">
  <div class="main_out">
    <div class="main">Main</div>
  </div>
  <div class="aside left">Left</div>
  <div class="aside right">Right</div>
</div>

.main_out, .aside {
  float: left;
  height: 65px;
  line-height: 65px;
  text-align: center;
}
.main_out {
  width: 100%;
}
.aside {
  background-color: #31B0D5;
}
.main {
  margin-left: 150px; //added
  margin-right: 190px; //added
  height: 100%;
  background-color: #0088CC;
}
.left {
  margin-left: -100%; //added
  width: 140px;
}
.right {
  float: right; //added
  margin-left: -190px; //added
  width: 180px;
}
```

4. 双飞翼和圣杯的不同点在于：

1. 圣杯布局是为了中间栏不被遮挡，将main、left和right的外层包裹bd设置了padding-left和padding-right，设置左右栏的position属性为relative，并分别对应左右栏设置left和right属性，一边使左右栏分别左右移动到空出的padding空间，以此来保证不遮挡中间栏。
2. 双飞翼布局是为了中间栏内容不被遮挡，将main元素放在main_out元素内，使main_out元素与left和right元素同级。接下来在main元素中使用margin-left和margin-right为左右栏留出空白位置。
5. 其实也可以用flex来简单实现，将中间的内容区flex设为1即可

9. 把div固定在页面的顶部?

1. 开启固定定位: position:fixed; top:0;

10. flex布局有哪些属性?

1. flex-direction 用来设置容器的主轴方向, 默认为row, 主轴在水平方向
2. flex-wrap 设置主轴是否换行, 默认为nowrap, 不换行
3. justify-content 设置元素在主轴上的对齐方式
4. align-items 设置元素在侧轴上的对齐方式
5. align-content 设置侧轴上留白的分布
6. order: 0 定义弹性盒容器内元素的排列顺序, 默认为0, 数值越小, 排列越靠前
7. flex-grow 定义弹性盒容器内元素的方法比例
8. flex-shrink 定义弹性盒内元素的缩小比例, 默认为1, 即空间不足, 该元素将缩小
9. flex-basis 定义分配多余空间之前, 元素占据的主轴空间, 浏览器根据这个属性, 计算主轴是否有多余空间, 默认为auto
10. flex 是flex-grow, flex-shrink, flex-basis的简写

js相关

1. 闭包理解 及作用?

1. 闭包就是可以访问到外部函数中变量的内部函数, 主要用来隐藏一些不希望外部访问到的变量。

2. 闭包用来做什么? 优缺点?

1. 闭包用来执行内部函数, 也可以用来隐藏一些不希望外部访问到的变量。
2. 闭包的优点: 避免全局变量的污染, 可以读取函数内部的变量, 并且变量的值始终保持在内部中, 不会在外层函数调用后被自动清除。
3. 闭包的缺点: 常驻内存, 增大内存的使用量, 使用不当会造成内存泄漏, 降低程序的性能, 所以, 闭包在不使用时, 需要及时释放

3. this在不同情况下的取值

1. 以函数形式调用, this是window;
2. 以方法形式调用, this是调用方法的对象;
3. 以构造函数形式调用函数时, this是新创建的对象;
4. 使用call或apply调用时, this是call和apply的第一个参数
5. 全局作用域中的this是window
6. 箭头函数, this由外层作用域决定

4. 事件的传播:

1. 捕获: 由外到内
2. 冒泡: 由内到外

5. 节流防抖如何实现?

1. 节流: 在固定的时间内触发事件, 每隔几秒触发一次。使用场景: 抢购点击、DOM元素拖拽功能实现)

```
// 函数节流
function throttle(callback, delay) {
  let start = 0 // 必须保存第一次点击立即调用
  return function (event) { // 事件回调函数
    // this是发生事件的dom元素
    console.log('throttle 事件')
  }
}
```

```

const current = Date.now()
if (current - start > delay) { // 从第2次点击开始，需要间隔时间超过delay
  callback.call(this, event)
  // 将当前时间指定为start，==> 为后面的比较做准备
  start = current
}
}
}

```

2. 防抖：频繁触发，只执行最后一次，几秒内只执行一次。使用场景：输入框实时联想搜索

```

// 函数防抖
function debounce(callback, delay) {
  return function (event) {
    console.log('debounce 事件...')

    // 清除待执行的定时器任务
    if (callback.timeoutId) {
      clearTimeout(callback.timeoutId)
    }
    // 每隔delay的时间，启动一个新的延迟定时器，去准备调用callback
    callback.timeoutId = setTimeout(() => {
      callback.call(this, event)
      // 如果定时器回调执行了，删除标记
      delete callback.timeoutId
    }, delay)
  }
}

```

6. 数组去重，数组合并，数组排序？

```

// 双层for循环实现数组去重
let arr = [1,2,3,4,1,2,3,4,5];
let newArr = [];
for(let i=0; i<arr.length; i++){
  let flag = true; // 创建一个标志位，默认为true
  for(let j=0; j<newArr.length; j++){
    if(arr[i] === newArr[j]){
      flag = false;
      break;
    }
  }
  if(flag){
    newArr[newArr.length] = arr[i];
  }
}

// 利用数组方法indexOf() 和 push() 实现数组去重
let arr = [10,20,30,20,30,50];
let newArr = [];
for(let i=0; i<arr.length; i++){
  if(newArr.indexOf(arr[i]) === -1){
    newArr.push(arr[i])
  }
}

```

```

    }
}

// concat() 数组合并
let arr1 = [1,2,3,4,5];
let arr2 = [10,20,30,40,50];
let arr3 = arr1.concat(arr2); // [1,2,3,4,5,10,20,30,40,50]

// for循环实现数组合并
for(let i in arr2){
    arr1.push(arr2[i]);
}

// apply实现数组合并
arr1.push.apply(arr1,arr2);

// 冒泡实现排序
let arr = [12,8,88,9,62,66];
// 排列次数的最大应该为数组的长度-1
for(var j=0; j<arr.length-1; j++){
    //获取到数组中的每一个元素
    for(var i=0; i<arr.length-1; i++){
        //比较当前元素和后一个元素的大小
        //当前元素 arr[i] 后一个元素arr[i+1]
        if(arr[i] > arr[i+1]){
            // 前边的元素大于后边的元素，交换两个元素的位置
            var temp = arr[i];
            arr[i] = arr[i+1];
            arr[i+1] = temp;

            //可使用解构赋值
            //[arr[i],arr[i+1]] = [arr[i+1],arr[i]];
        }
    }
}
console.log(arr);

// sort() 方法实现排序
let arr = [12,8,88,9,62,66];
arr.sort((item,index) => {
    // return item - index; // 升序
    return index - item; // 降序
})
console.log(arr);

// 递归快排
var arr = [3,4,1,2,6,7,9,0,5,8];
// 对array进行排序，返回一个排好序的新数组
function quickSort(array) {
    //设置基线条件
    if(array.length <= 1){
        return array;
    }
    // 获取一个基准值，获取数组的最后一个元素作为基准值
    var base = array[array.length-1];
    //创建两个数组
    var left = []; //左数组，存储小于base的元素

```

```

var right = []; //右数组, 存储大于base的元素
//比较每个元素和base的大小
for(var i=0; i<array.length - 1; i++){
    // 比较当前值和基准值的大小
    if(array[i] < base){
        //放入到左数组
        left.push(array[i]);
    }else{
        //放入到右数组
        right.push(array[i]);
    }
}
//left base right
// 将left base right 拼接为一个数组返回
return quickSort(left).concat(base, quickSort(right));
}
var result = quickSort(arr);
console.log(result);

```

7. setTimeout (function(){3,0})本质原理

1. JavaScript是单线程执行的, 也就是无法同时执行多段代码, 当某一段代码执行的时候, 所有后续代码都必须等待, 形成一个队列, 一旦当前任务执行完毕, 再从队列中取出下一个任务, 这也被称为"阻塞式执行", 所以, 一次鼠标点击, 或定时器到达时间点, 或是ajax请求完成后触发了回调函数, 这些事件处理程序或回调函数都不会立即执行, 而是立即排队, 一旦线程有空闲就立即执行, 如果代码中设置了setTimeout,那么浏览器便会在合适的时间, 将代码插入任务队列, 如果这个时间设为0, 就代表立即插入队列, 但不会立即执行, 仍然要等前面的代码执行完毕。所以setTimeout并不能保证执行的时间, 是否及时执行取决于js线程是拥挤还是空闲的。也就是说, setTimeout只能保证在指定的时间过后执行, 并不能保证这个任务在什么时候执行, 执行js的线程会在空闲的时候, 自动从队列中取出任务然后执行, js通过这种队列机制, 给我们制造了一个异步执行的假象。应用场景: 当用户在文本框输入字符时, 将输入的内容实时的在页面中显示, 就可以用setTimeout实现

8. js中创建对象的方法

1. 原始模式

```
var person = {name: '小明'}
```

2. 构造函数方式

```
var person = new Object();
person.name = '小明'
```

3. 工厂模式

```
function creatPerson(name){
    var person = new Object();
    person.name = name;
    return person;
}
```

4. 构造函数

```
function Person(name){
  this.name = name
}
let p1 = new Person('小明')
```

5. 原型模式

6. 混合模式(构造、原型)

```
function Person(name){
  this.name = name;
}
Person.prototype = {
  eat: ['面包','鸡蛋'];
  sayName: function(){
    console.log(this.name);
  }
}
```

9. 原型的理解

1. JavaScript是一门面向对象的高级语言，它的面向对象是基于原型，在函数对象中都存在一个属性：prototype，该属性指向一个对象，这个对象就是原型对象，如果函数作为普通函数调用这个prototype没有任何作用，如果函数作为一个构造函数调用时，则它所创建的所有实例中都有一个隐含的属性__proto__指向该对象(实例的隐式原型指向构造函数的显示原型)，所有原型对象就相当于所有实例的一个公共区域，可以将对象中共有的属性/方法统一设置在原型对象中，这样我们只需要设置一次即可让所有的实例都具有这些属性/方法。
2. 在访问一个对象的属性时，js解析器会先在对象本身寻找，如果找到了则使用，如果没有找到则继续在对象的原型中寻找，找到了则使用，如果还没找到，则继续到原型的原型中查找，直到找到Object的原型，它是所有对象的原型，如果还没找到，则返回undefined
3. 原型的作用：数据共享，节约内存

a.b的解析流程：

查看a变量：作用域链查看

不存在：==> 报错

存在：得到它的值

基本类型：

 null/undefined ==> 报错

 number/string/boolean ==> 创建一个包含此值得包装类型对象返回

引用类型/地址值：==> 解析.b ==> 查找b属性

 先在自身找，找到了就返回，如果没有找到

 沿着原型链查找

 找到了返回

 没找到，返回undefined

10. parent构造函数，child怎么利用刚说的原型来实现继承？

11. 利用prototype 将parent的prototype指向child来达到继承的效果

```
function parent(name,age){
  this.name = name;
```



```

    this.age = age;
}
parent.prototype.eat = function(){
    console.log('吃面包')
};
function child(num, score){
    this.num = num;
    this.score = score;
}
child.prototype = new parent('小明',19);
child.prototype.constructor = child
let c1 = new child(123,40);
c1.eat()
// 使用原型继承时，需要将构造器的指向更改为正确的指向

```

12. 字符串转数组?数组转字符串的方法?

1. split() 以指定的字符作为切割点，来将字符串切割为数组返回，如果传的是空串，那么每个字符都会作为数组的元素
2. join() 以指定的连接符，将数组中的元素连接成一个字符串，如果不写，默认以逗号连接。

13. window.onload 和 window.Filereader 是什么? 谁先执行?

1. window.onload()方法用于在网页加载完毕后立即执行的操作，即 当HTML文档加载完毕后，立即执行，通常写在body内的最下方，保证页面加载完成后(图片、css)执行js代码
2. 为什么要用window.onload : 因为JavaScript中的函数方法需要HTML文档渲染完成后才可以执行，如果没有渲染完成此时的DOM树是不完整的，这样在调用一些js代码的时候可能会报错(undefined)
3. window.filereader 可以通过一个file input选择一个图像文件将图片显示在页面中，而不用发送到后端，由后端对其存储再由后端返回一个url地址，通过url来显示图像，它使得js拥有处理文件的能力，它可以异步的读取在电脑中的文件
4. 谁先执行? window.onload

14. 跨域问题，跨域产生的原因?

1. 跨域就是指从a页面想要获取b页面的资源，但是它们的协议、域名、端口号都是不同的，而浏览器为了安全问题一般都限制访问，不允许跨域请求资源
2. 怎么解决跨域问题:
 1. JSONP
 1. 前台：通过script标签，发送GET的http请求，请求地址为目标地址
 2. 后台：处理请求，产生需要返回的数据data
 3. 不足：只能处理GET请求，每个请求都要在后台做处理，比较麻烦
 2. CORS，返回允许浏览器在某个域上发送跨域请求的相关响应头

```
// 使用cors, 允许跨域, 且允许携带跨域cookie
app.use(function (req, res, next) {
  // console.log('----')
  // 允许跨域的地址
  res.header('Access-Control-Allow-Origin', 'http://localhost:5500')
  // 不要是*
  // 允许携带凭证(也就是cookie)
  res.header('Access-Control-Allow-Credentials', 'true')
  // 允许跨域的请求头
  res.set("Access-Control-Allow-Headers", "Content-Type")
  // 放行
  next()
})
```

3. 代理服务器

1. 开发环境: 利用webpack-dev-server中的http-proxy-middle 进行正向代理
2. 生产环境: 利用nginx 进行反向代理

15. 前端请求状态码有哪些? 各自代表什么?

1. 200 - 请求成功, 已经正常处理完毕
2. 301 - 请求永久重定向, 转移到其它URL
3. 302 - 请求临时重定向
4. 304 - 请求被重定向到客户端本地缓存
5. 400 - 客户端请求存在语法错误
6. 401 - 客户端请求没有经过授权
7. 403 - 客户端的请求被服务器拒绝, 一般为客户端没有访问权限
8. 404 - 客户端请求的URL在服务端不存在
9. 500 - 服务端内部错误

301重定向是一种永久重定向, 而302跳转是暂时的跳转。

16. 向后端发送请求, 如何携带token(请求拦截器中加请求头)?

1. ajax添加token到Header中

```
$.ajax({
  type: "GET",
  url: "/access/logout/" + userCode,
  headers: {'Authorization': token}
});

$.ajax({
  type: "GET",
  url: "/access/logout/" + userCode,
  beforeSend: function(request) {
    request.setRequestHeader("Authorization", token);
  },
  success: function(result) {
  }
});
```

2. 一般在axios二次封装的时候, 如果请求成功, 就会把token存入localStorage中, 添加一个请求头header, 把token放在里边

17. http和https的区别？

1. http:超文本传输协议，是互联网上应用最为广泛的一种网络协议，设计HTTP最初的目的是为了提供一种发布和接收HTML页面的方法，它可以使浏览器更加高效，HTTP协议是以明文方式发送信息的，如果被人截取了web浏览器和服务器之间的传输报文，就可以直接获取其中的数据信息
2. HTTPS:是以安全为目的的HTTP通道，是HTTP的安全版，HTTPS的设计目标：数据保密性，保证数据内容在传输过程中不会被第三方查看，(快递/包裹，快递员不知道里面是什么)，数据完整性：及时发现被第三方篡改的传输内容；身份校验安全性：保证数据到达目标位置
3. 区别：
 1. 传输信息安全性不同：
 1. http协议：是超文本传输协议，信息是明文传输，如果攻击者截取了web浏览器和网站服务器之间的传输报文，那就可以直接获取其中的数据信息
 2. https：具有安全性的ssl加密传输协议，为浏览器和服务器之间的通信加密，确保数据传输的安全
 2. 连接方式不同：
 1. http协议：连接很简单，没有状态
 2. https协议：是由ssl + http协议构建的可进行加密传输，身份验证的网络协议
 3. 端口不同：
 1. http协议：使用的端口是80
 2. https协议：使用的端口是443
 4. 证书申请方式不同：
 1. http协议：免费申请
 2. https协议：需要到CA申请证书，一般免费证书很少，需要付费

18. axios 和 ajax的区别？




1. axios：
 1. 是通过promise来实现对ajax技术的一种封装
 2. axios是一个基于promise的HTTP库，可以用在浏览器和node.js中，特点：
 1. 从浏览器中创建XMLHttpRequests
 2. 从node.js创建http请求
 3. 支持promise API
 4. 拦截请求和响应
 5. 客户端支持防御XSRF
2. ajax：
 1. 是对原生XHR的封装，为了达到跨域的目的，增添了对JSONP的支持
 2. 用于快速的创建动态页面，能够实现无刷新更新数据从而提高用户体验
 3. 实现原理：由客户端发送请求给ajax引擎，再由ajax引擎发送请求到服务器，服务器做出一系列响应之后返回给ajax引擎，由ajax引擎决定将这个结果写入到客户端的什么位置，从而实现无刷新更新数据
 4. 核心对象：XMLHttpRequest
 5. 优点：
 1. 无刷新更新数据
 2. 异步与服务器通信
 3. 前端后台负载平衡
 4. 界面与应用分离
 6. 缺点：

1. 安全问题：ajax暴露了与服务器交互的细节
2. 没有浏览历史，不能回退
3. 存在跨域问题
4. ajax不能很好的支持移动设备

```
// 封装ajax函数，用来发送ajax请求，返回promise对象
function promiseAjax(url) {
  return new Promise((resolve, reject) => {
    // 1. 创建对象
    let x = new XMLHttpRequest();
    // 2. 初始化
    x.open('GET', url);
    // 3. 发送
    x.send();
    // 4. 绑定事件，处理结果
    x.onreadystatechange = function () {
      if(x.readyState === 4){
        if(x.status >= 200 && x.status < 300) {
          // 成功
          resolve(x.response)
        }else {
          // 失败
          reject(x.status)
        }
      }
    }
  })
}
```

3. 区别：axios是通过Promise实现对ajax技术的一种封装，简单来说就是ajax技术实现了局部数据的刷新，axios实现了对ajax的封装，axios有的ajax都有，ajax有的axios不一定有。

4. Ajax原理：

1.  image-20201101232558473
2.  image-20201101232638459
3.  image-20201101232843753

19. axios的二次封装

1. 配置通用的基础路径和超时时间
2. 显示请求进度条
3. 成功返回的数据不再是response，而直接时响应体数据response.data
4. 统一处理请求错误，具体请求也可以选择处理或不处理
5. 每个请求自动携带userTepmld的请求头，数据值在state中，在请求拦截器中实现
6. 如果当前有token，自动携带token的请求头

20. 事件轮回机制

一个在分线程执行的模块，一个是待执行的事件队列

单线程 异步???

ES6相关

1. 简述一下promise

1. promise是ES6推出的更好的异步编程解决方法，可以解决回调地狱问题，支持链式调用，可以用promise封装一个异步操作来获取其成功/失败的结果值。有三种状态：初始状态为pending，可以变化为resolved或rejected。
2. promise的执行流程是什么？
 - 首先new了一个Promise对象，状态为pending，然后执行异步操作，
 - 成功了：执行resolve(), 状态由pending变为resolved状态，再执行回调函数onResolved(),返回一个新的promise对象
 - 失败了：执行reject(), 状态由pending变为rejected状态，再执行回调函数onRejected(),返回一个新的promise对象

3. promise详细说一下，以及如何一次发送多个请求，怎么实现？

1. promise是ES6推出的新的异步编程解决方法，可以异步操作启动后，在指定回调函数得到异步结果数据，解决嵌套回调的回调地狱问题，promise对象有三种状态：初始状态为pending，可以变化为resolved或rejected。promise有.then()方法和.all()方法，then()方法返回的是一个新的promise，新的promise的结果状态由then指定的回调函数执行的结果决定。all()方法用来一次性发送多个异步请求，只有当发送的请求都成功，返回的promise才成功，一旦有一个失败，返回的promise就失败了。还有一个race()方法，返回一个新的promise，第一个完成的promise的结果状态就是最终的结果状态。如何一次性发送多个请求？可以通过.all()方法来实现。
2. promise.then()返回的新promise的结果状态由then()指定的回调函数执行的结果决定：
 1. 如果抛出异常，新promise变为rejected, reason为抛出的异常
 2. 如果返回的是非promise的任意值, 新promise变为resolved, value为返回的值
 3. 如果返回的是另一个新promise, 此promise的结果就会成为新promise的结果
3. promise异常穿透：当使用promise的then链式调用时，可以在最后指定失败的回调，前面任何操作出了异常，都会传到最后失败的回调中处理
4. 中断promise链：当使用promise的then链式调用时，在中间中断，不再调用后面的回调函数，在回调函数中返回一个pending状态的promise对象

4. 如果发送3个请求后，再发送第4个请求如何实现？

1. promise的then()方法返回一个新的promise，可以使用all()方法来发送多个请求，再通过then()进行链式调用来实现

```
function ajax(url){
    return axios.get(url)
}
const p1 = ajax(url1)
const p2 = ajax(url2)
const p3 = ajax(url3)
Promise.all([p1,p2,p3])
    .then(values => {
        return ajax(url)
    })
    .then(value => {
        console.log(value) // 就是第四个请求成功的value
    })
    .catch(error => {
    })
})
```

5. all() 方法和 race()

1. all() 返回一个新的promise, 一次性发送多个异步请求, 只有所有的promise都成功返回的promise才成功, 只要有一个失败了就直接失败
2. race() 返回一个新的promise, 第一个完成的promise的结果状态就是最终的结果状态

6. async/await与promise的关系

- async/await用来消灭异步回调的终极武器
- 作用: 简化promise对象的使用, 不用再使用then/catch来指定回调函数
- 和Promise相辅相成
- 执行async函数, 返回promise对象
- await相当于promise的then()
- try..catch可捕获异常, 相当于promise的catch

7. async和await的使用

1. await: 一般在结果为promise的表达式的左侧, 如果await右侧表达式结果不是promise, 直接得到这个结果
2. async: 包含了await最近的函数定义的左侧
 1. 成功的value: await左侧以及下面的代码, 都是在成功之后执行(相当于在成功的回调中执行)
 2. 失败的reason: 使用try..catch中在失败后执行(相当于在失败的回调中执行)

8. const和let 的区别?

1. const 一般用来定义常量, let用来定义变量;
2. 都有块级作用域;
3. const必须声明的同时进行赋值, 且不能修改, 否则会报错;
4. let声明的变量可以重新赋值;
5. 都不会被添加到window上, 也都没有变量提升

9. ES6新增箭头函数的简单使用

1. 没有this, 箭头函数的this指向由外部作用域决定
2. 右侧没有大括号可以省略return, 只有一个形参可以省略小括号()
3. bind不能改变箭头函数的this指向
4. 箭头函数不能作为构造函数使用, 不能new

10. 解构赋值

11. ES6允许按照一定模式, 从数组和对象中提取值, 来对变量进行赋值, 这被称为解构赋值, 当我们频繁使用对象方法, 数组元素, 就可以使用解构赋值的形式

1. 数组的解构:

```
var [c,d,...e] = [1,2,3,4,5,6];
```

2. 对象的解构:

```
var [name, age, ...other] =  
{name:'xxx',age:18,gender:'男',address:'xxxx'};
```

3. 实参的解构:

```
var arr = [1,2,3];  
fn(...arr);
```

12. ES6常用语法

1. const let
2. 箭头函数
3. 解构赋值
4. 形参默认值
5. rest/剩余参数
6. 类语法: class/extends/constructor/static/super
7. 扩展运算符
8. 模板字符串
9. 异步语法
10. 对象的属性与方法简写
11. set/map
12. 模块化语法: export / default / import / import()

工程化相关

1. webpack了解多少?

1. webpack是一个前端资源加载/打包的工具，它将根据模块的依赖关系，进行了静态分析，然后将这些模块按照指定的规则生成对应的静态资源。

2. 几个核心概念:

1. Entry: 入口起点，指示webpack应该使用哪个模块来作为构建其内部依赖图的开始。
2. Output: output 属性告诉 webpack 在哪里输出它所创建的 bundles，以及如何命名这些文件，默认值为 ./dist。
3. Loader: loader 让 webpack 能够去处理那些非 JavaScript 文件（webpack 自身只能解析 JavaScript）。
4. Plugins: 插件则可以用于执行范围更广的任务。插件的范围包括，从打包优化和压缩，一直到重新定义环境中的变量等。
5. Mode: 模式，有生产模式production和开发模式development

3. 理解Loader

- Webpack 本身只能加载JS/JSON模块，如果要加载其他类型的文件(模块)，就需要使用对应的loader 进行转换/加载
- Loader 本身也是运行在 node.js 环境中的 JavaScript 模块
- 常见的几个loader:
 - babel-loader
 - css-loader/style-loader/less-loader/stylus-loader/ sass-loader/postcss-loader
 - file-loader / url-loader
 - eslint-loader
 - vue-loader/vue-style-loader

4. 理解Plugins

- 插件可以完成一些loader不能完成的功能。
- 插件的使用一般是在 webpack 的配置信息 plugins 选项中指定。
- 常见的几个plugin:
 - html-webpack-plugin
 - copy-webpack-plugin 打包文件夹
 - clean-webpack-plugin 删除dist文件夹
 - optimize-css-assets-webpack-plugin 压缩css

5. 区别plugin和loader:

1. 用于加载特定类型的资源文件，webpack本身只能打包js，打包css就需要css-loader/style-loader
 2. plugin 用来扩展webpack其它方面的功能，例如页面引入打包文件需要html-webpack-plugin, 删除文件需要clean-webpack-plugin
6. 配置文件(默认)
- webpack.config.js : 是一个node模块，返回一个 json 格式的配置信息对象

2. webpack的使用(配置及打包)?

1. 初始化项目，生成package.json文件
2. 安装webpack:

```
npm install webpack webpack-cli -g //全局安装,作为指令使用
npm install webpack webpack-cli -D //本地安装,作为本地依赖使用
```

3. 编译打包应用:

1. 创建js文件，创建json文件，创建主页面
2. 运行:

```
开发配置指令: webpack src/js/app.js -o build/js/app.js --
mode=development
功能: webpack能够编译打包js和json文件，并且能将es6的模块化语法转换成浏览器能识别的语法
生产配置指令: webpack src/js/app.js -o build/js/app.js --
mode=production
功能: 在开发配置功能上加上一个压缩代码
```

4. webpack能够编译打包js和json文件，能将es6的模块化语法转为浏览器能识别的语法，可以压缩代码，缺点：不能编译打包css、img等文件，需要使用webpack配置文件解决。
5. 使用webpack配置文件，在项目根目录(webpack.config.js)定义配置文件，通过自定义配置文件，还原以上功能

3. webpack的优点?

1. webpack是以commonJS的形式来书写代码的，但对AMD/CMD的支持也很全面，方便旧项目进行代码迁移
2. 能被模块化的不仅仅是js
3. 开发便捷，能代替部分grunt/gulp的工作，比如：打包、压缩、图片转base64等
4. 扩展性强，插件机制完善
5. 能配置打包多个文件，有效的利用浏览器的缓存机制提升性能。
6. webpack能够将依赖的模块转化成可以代表这些包的静态文件，将依赖的模块分片化，并且按需加载，解决大型项目初始化加载缓慢的问题，每一个静态文件都可以看成一个模块，可以整合，第三方库能够在大型项目中使用

4. 打包上线的时候，需要修改哪些配置文件?

1. config文件夹下的index.js 中的文件中的 assetsPublicPath : './' 改为./ 不然的话会是空白页
2. build/util.js文件下在 'vue-style-loader' 之后加一行: publicPath: './..' 不然的话会有图片显示不出来，字体不能正常使用
3. 如果你有本地json数据的话，把本地json数据中的路径全部改为根路径
4. 本地如果使用了api代理，上线之后就不需要代理了

5. webpack的优化及如何优化打包?

1. 目标:兼容性, 减小打包文件, 懒加载, 预加载, 首屏加载优化
 1. 兼容低版本浏览器: babel, postcss(自动增加浏览器厂商前缀)
 2. 拆分打包压缩: 单独打包第三方模块js & 压缩js, 抽取css单独打包&压缩css
 3. 懒加载: import()动态引入模块 ==> code split,在特定条件下才执行import()
 4. 预加载
 5. 打包文件hash化:利用浏览器缓存
 6. Tree Shaking摇树: 摇掉不需要的文件
2. 优化打包: 加快打包/提升开发调试体验:
 1. loader增加include匹配特定条件(缩小loader查找文件的范围)
 2. dll预打包, 为后面重复打包做铺垫
 3. eslint代码规范检查
 4. oneOf: 让模块只被一个loader处理, 只要匹配上其它的就不看了, 提升打包速度

6. webpack如何做浏览器兼容处理?

1. 使用@babel/core和@babel/preset-env, 只能编译Es6的新语法转换为ES5, 不能处理ES6的新API, 在低版本浏览器中不能运行
2. 使用@babel/polyfill, 内部提供了新API的实现, 来对js代码做兼容处理, 但是默认是将全部打包, 导致打包文件太大
3. 通过配置useBuiltIns: 'usage'实现polyfill的按需引入打包

移动端相关

1. 移动端适配的问题?

1. viewport适配, 不用复杂的计算, 拿到设计稿后, 设置布局视口宽度为设计稿宽度, 然后直接按照设计稿给宽高进行布局, 缺点: 不能使用完整的meta标签, 会导致某些安卓手机上有兼容性问题, 不希望适配的例如边框, 也会强制参与适配, 图片会失真
2. rem适配, 编写样式是统一使用rem为单位, 在不同设备上动态调整根字体大小
 1. 方案一(淘宝、百度):
 1. 设置完美视口, 通过js设置根字体大小, 编写样式时, 直接以rem为单位, 值为 设计值 / 100, 增加js代码进行实时适配, 优势: 编写样式时直接挪动小数点即可。
 2. 方案二(搜狐、唯品会):
 1. 设置完美视口, 通过js设置根字体大小, 编写样式时, 直接以rem为单位, 值为 设计值 / (设计稿宽度 / 10), 增加js代码进行实时适配
3. vw适配:
 1. vw和vh是两个相对单位, 1vw 等于 布局视口宽度的1%, 1vh等于布局视口高度的1%, 也存在一定的兼容问题

2. 如何手动的去实现rem适配?

1. 方案一(淘宝、百度):
 1. 设置完美视口, 通过js设置根字体大小, 编写样式时, 直接以rem为单位, 值为 设计值 / 100, 增加js代码进行实时适配, 优势: 编写样式时直接挪动小数点即可。
2. 方案二(搜狐、唯品会):
 1. 设置完美视口, 通过js设置根字体大小, 编写样式时, 直接以rem为单位, 值为 设计值 / (设计稿宽度 / 10), 增加js代码进行实时适配

Vue相关

1. Vue的生命周期和钩子?

1. 整个生命周期有三个大的阶段：初始化显示，更新显示，卸载
2. 在vm整个生命周期的过程中的特定时刻，会自动调用某个回调函数
3. 生命周期钩子：
 1. 初始化显示：
 1. beforeCreate() 在实例初始化之后调用，此时不能通过this访问data和methods
 2. created() 编译模板，此时可以通过this去操作data和methods里边的数据，也可以在此发ajax请求(较早，此时还不能通过ref找到对应的标签对象)，
 1. 实现数据代理/数据绑定，此时通过vm可以得到data当中的数据
 3. beforeMount() 模板已经在内存中编译，将要挂载到页面，此时还不能通过ref找到对应的标签对象
 4. mounted() 页面已经挂载，可以通过ref找到对应的标签，也可以选择此时发ajax请求
 2. 更新：
 1. beforeUpdate() 在数据更新后，界面更新前调用，只能访问到原有的界面
 2. updated() 在界面更新之后调用，此时可以访问最新的界面
 3. 卸载：
 1. beforeDestroy() 实例销毁之前调用，一般在此执行一些收尾工作：清除定时器等；
 2. destroyed() 实例销毁之后调用，实例已经无法正常工作了
4. 激活与失活(keep-alive相关)
 1. activated 激活，组件激活，被复用
 2. deactivated失活，组件失活，但没有死亡/销毁
5. 捕获子组件错误
 1. errorCaptured 用于捕获子组件的错误
 - 用于捕获子组件的错误,return false可以阻止错误向上冒泡(传递)

2. Vue中性能优化?

1. 路由懒加载：Vue单页面应用，有很多路由引入，使用webpack打包加载后文件很大，会出现首屏加载过慢的情况，用户体验较差。所以，将不同的路由对应的组件分割成不同的代码块，当路由被调用的时候再加载对应的组件，这样可以提升部分性能，优化用户体验。
2. 图片懒加载，为了加快页面加载速度，未出现在可视区域内的图片先不做加载，提高用户体验。
3. 第三方插件按需引入，直接引入整个插件，会导致体积过大，加载时间过长，借助babel-plugin-compeneil 然后可按需引入，减小项目体积
4. 不要将所有的数据都放在data中，data中的数据都会增加getter和setter，又会收集watcher，这样还占内存，不需要响应式的数据我们可以定义在实例上。
5. 在v-for绑定事件的时候可以使用事件代理，将事件绑定到外层元素上。
6. 使用keep-alive缓存组件，防止组件来回的创建和销毁(浪费性能)
 1. 做一个大型项目的时候，有很多router，对应的有很多个页面，在页面的快速切换中，为了保证页面加载的效率，除了缓存机制，还可以使用vue中的keep-alive组件
7. 使用v-show代替v-if指令
8. key最好保证唯一性，提高DOM-diff的复用性能。
9. 打包优化：
 1. 使用cdn加载第三方模块

2. 压缩和缓存

3. v-show 和 v-if 指令的共同点和不同点?

1. v-show和v-if都可以用来隐藏元素，不同的是：v-show是通过控制display样式来隐藏/显示(隐藏时占用内存空间，重新显示速度较快)，而v-if是通过删除DOM对象来隐藏(隐藏时不占用内存空间，重新显示时需要重新创建，需要加载时间)

4. Vue中响应数据添加?

1. 首先我们都知道vue最大的特点就是数据驱动视图，当响应式数据发生变化的时候就会触发视图更新，而非响应式数据发生变化的时候则不会触发视图更新。vue响应式数据原理，也叫数据绑定原理和双向数据绑定原理，底层通过Object.defineProperty()给data中的数据添加数据劫持(setter和getter方法)，当我们通过Object.defineProperty()处理data中数据的时候，数据就添加上了get和set方法，获取数据的时候会触发get，修改数据的时候会触发set，在修改数据的时候，优先触发set，再触发watcher监听，最后通知界面。vue中如果动态添加属性，该属性就没有经历过处理，就没有set和get方法，所以数据变而页面不变。vue提供了一个实例方法：vm.\$set(目标对象, '动态添加的属性名', '属性值')可以添加一个响应式属性，会触发视图的更新。

5. Vue-lazy load 的使用(图片懒加载)?

- 先将img标签中的src链接设置为空，将真正的图片链接放在自定义属性（data-src），
- 当js监听到图片元素进入到可视窗口的时候，将自定义属性中的地址存储到src中，达到懒加载的效果

1. 首先我们需要先知道什么是图片懒加载和在什么情况下我们会用到图片懒加载：当我们打开一个有很多图片的页面时，先只加载页面上可视区域的图片，等页面滚动时，再加载所需要的图片，这就是图片懒加载，使用图片懒加载可以减少/延迟请求次数，缓解浏览器的压力，增强用户体验。

2. vue-lazyload的使用:

1. 首先安装插件npm install vue-lazyload
2. 在main.js中引入插件 import VueLazyLoad from 'vue-lazyload'
3. 使用:

```
Vue.use(VueLazyLoad, {
  error: '', // 加载失败的图片
  loading: '' // 加载中的默认图
})
```

4. 也可以对图片进行单独配置，可以设置三个属性src图片，loading加载状态下的图片，error错误状态下的图片

```
<div v-lazy-container="{ selector: 'img', error: 'xxx.jpg', loading: 'xxx.jpg' }"></div>
```

5. 懒加载就是一种网页性能优化的方式，它能极大的提升用户体验，不仅可以提高浏览器的渲染速度，还可节省用户的流量。

6. Vue的实现原理?

1. vue实现的本质就是数据驱动视图原理, 所谓的数据驱动就是: 当数据发生变化的时候, 界面会相应的发生变化, 不需要手动去修改dom, 其中就是通过v-model双向数据绑定的原理来实现Model层与View层交互的效果。

7. vue-route的使用

1. 先下载vue-router `npm install vue-router`
2. vue-router就是用来跟后端服务器进行交互的一种方式, 通过不同的路径, 来请求不同的资源。
3. 在项目文件夹中引入注册, vueRouter 并且声明使用
4. 在main.js中引用创建好的vueRouter文件, 就可以全局使用

8. Vue的父子组件传值

1. 在vue中实现父子组件传值的方法使用较多的是props, 通过标签属性来传递, 是一种基本的通信方式, 函数属性情况下, 子向父传递数据, 通过子组件调用父组件的方法, 更新父组件的数据, 通过参数将数据传递给父组件; 非函数或一般属性: 父向子传递数据
2. 还有vue的自定义事件也可以实现父子组件间传值, 在说vue的自定义事件之前, 我们需要先理解事件的两个操作, 绑定事件监听: 事件名和回调函数(可以接收任意多个参数数据), 分发/触发事件:事件名(需要与绑定监听的事件名一致)和数据(可以指定任意多个数据),在vue的自定义事件中, 使用\$on('事件名',data)来绑定事件监听(接收数据), 使用\$emit('事件名',data)来分发事件(传递数据),使用\$off(事件名)来解绑事件监听, 事件处理的方法定义在vue原型对象上, 同一个组件对象绑定监听后分发事件, 回调函数才执行
3. slot插槽也可以实现父子组件间传值, 通过父组件向子组件传递标签内容来实现, 插槽有以下几种:

1. 默认插槽: 没有名字的插槽, 一个模板页面中最多只能有一个默认插槽
2. 命名插槽: 有名字的插槽
3. 作用域插槽: 通过父组件读取子组件的数据来决定向子组件传递什么样的结构内容, 子组件通过slot标签属性向父组件传递数据, 父组件通过slot-scope属性来接收所有传递过来的数据, 根据数据决定要传递的结构内容

插槽模板是在父组件解析好之后, 再传递给子组件, 模板内容是通过标签体传递过去的。

9. Vue的自定义事件?

1. vue的自定义事件是用来实现子组件向父组件通信的一种方式, 在说vue的自定义事件之前, 我们需要先理解事件的两个操作, 绑定事件监听: 事件名和回调函数(可以接收任意多个参数数据), 分发/触发事件:事件名(需要与绑定监听的事件名一致)和数据(可以指定任意多个数据), 在vue的自定义事件中, 在父组件中使用\$on('事件名',data)来绑定事件监听(接收数据), 在子组件中使用\$emit('事件名',data)来分发事件(传递数据),使用\$off(事件名)来解绑事件监听, 事件处理的方法定义在vue原型对象上, 同一个组件对象绑定监听后分发事件, 回调函数才执行。据我了解, elment-ui组件的事件监听语法都用到了自定义事件, 我们的一些项目中的组件也用到过不少自定义事件

10. vuex的使用场景, 具体有什么?

1. Vuex是管理状态数据的一种模式, 采用集中式管理状态数据, 用来进行任意组件间的通信操作, 平时我们也说它是一个插件(工具), Vuex可以帮助我们管理共享状态, 并且附带了更多的概念和框架, 这需要对短期和长期效益进行权衡, 也就是说应用简单, 组件比较少就没有必要使用, 但是也可以, 如果应用复杂, 那么使用Vuex就会带来很大的便利。Vuex的核心就是把所有的共享状态数据拿出来放在Vuex中进行集中式管理, Vuex中的4个核心概念:

1. state 代表初始状态数据, 是一个包含了多个状态数据/属性的对象

2. mutations 是一个包含了多个直接修改状态数据的方法的对象
3. actions:是一个包含了多个用户行为回调方法的对象
4. getters: 是一个包含了多个计算属性的方法的对象

只有通过mutations的方法才能去直接修改state里边的状态数据, actions里边是用户操作的行为回调函数, 它的内部可以写异步和判断(mutations不可以)

2. 使用Vuex的流程:

1. 首先需要安装插件: npm install vuex
2. 创建独立的模块来使用vuex
3. 编写四个核心对象
4. 将模块暴露出去
5. 在Vue配置项当中注册vuex对象
6. 在核心对象中写代码

11. vuex如何外部改变内部的值?

1. 可以通过\$store对象的dispatch方法, 分发action, action执行结束后, 可以通过接收的context对象内的属性commit属性触发mutation, 在mutation中修改状态
2. 也可以通过\$store的commit方法提交mutation, 直接修改vuex的状态

12. MVVM 与 MVC ?

1. MVVM即Model-View-ViewModel的简写, Model模型 指的是后台传递过来的数据, View视图 指的是所看到的页面, VM视图模型 是MVVM模式的核心, 它是连接view和model的桥梁, 通过数据绑定来将Model 转换为View, 后台传递过来的数据转化为可视化的页面, 再通过事件监听将View转化成Model, 将所看到的页面转化为后台的数据, 我们也称之为: 双向数据绑定。
2. MVC是Model-View-Controller的简写, 即模型-视图-控制器, M和V的意思和MVVM中M和V的意思一样, Controller指的是页面业务逻辑, 使用MVC的目的就是将M和V的代码分离, MVC是单向通信, 也就是View和Model, 必须通过Controller来承上启下, MVC和MVVM的区别并不是VM完全取代了C, 只是再MVC的基础上添加了一层VM, 只不过弱化了C的概念, ViewModel存在目的在于抽离Controller中展示的业务逻辑, 而不是代替Controller, 其它视图操作业务等还是应该放在Controller中实现, 也就是说MVVM实现的是业务逻辑组件的重用, 使我们开发更高效, 结构更清晰, 增加代码的复用性。
3. MVVM的优势: 不用亲自操作DOM, 数据是响应式的, 一旦数据变化就会更新页面。

13. 路由跳转的方式?

1. 声明式路由: 使用路由链接进行路由跳转

```
<router-link to="/xxx"></router-link>
```

2. 编程式路由跳转

```
router.push/replace(location)
```

跳转路由携带参数的方式:

1. query 即能用name, 又能用path
2. params 需要使用分号占位, 只能用name
3. meta
4. props

耦合度较高, 传参方式和组件联系很密切

14. Vue 中this指向谁?

1. 指向vue组件的实例对象(在组件中的生命周期和vue内置对象内调用this)
2. 在组件函数的script标签直接访问this ==> undefined

15. Vue的数据双向绑定原理?

1. 默认文本框通过v-model指令绑定了msg表达式, 页面中修改了文本框的内容: xxx, 获取文本框的value属性值, 对比msg表达式中的值, 不一样, 那么就需要调用mvvm中的set方法修改data对象中的msg值(vm.msg='新值'),就会自动的进入劫持中的set方法内部, 在这个set方法内部, 当前的msg属性对应着一个dep对象, 这个dep对象的subs数组中有对应的watcher对象, 需要让dep通知内部所有联系的watcher对象, 更新数据, 更新的是当前这个表达式的值, 还需要在内存中重新把html模板中用到的表达式的地方全部都替换一下(最终调用的是updater对象中的方法)

16. Vue中页面闪动怎么解决?

1. 第一种:
 1. 使用watch + nextTick
 2. watch里面的深度监听, 可以监听到对象和数组的变化
 3. 使用watch监听数据的变化, 配合this.\$nextTick 在同一事件循环中数据变化后, DOM完成更新, 立即执行nextTick内的回调
2. 第二种:
 1. 初始化时, 页面加载出现闪动:
 2. 使用v-clock
 3. 这个指令可以隐藏未编译的标签

17. V-model 原理?

1. v-model的本质是将动态data数据通过value属性传给input显示, 实现data到view的绑定
2. 给input标签绑定input监听, 一旦输入改变读取最新的值保存到data对应的属性上, view ==> data的绑定

18. 组件间是如何传递参数的?

1. 组件间传递参数的方法有很多, 常见的父子之间, 或者兄弟之间, 或者任意组件之间的传递, 常用的大概有:
 1. 父子:
 1. props
 2. v-model
 3. vue自定义事件
 4. .sync
 5. \$ref \$children \$parent 插槽
 2. 祖孙:
 1. \$attrs
 2. \$listeners
 3. 任意组件:
 1. 全局事件总线
 2. vuex
2. 主要说一下全局事件总线: 实现任意组件间通信, 它可以在我们函数钩子beforecreate()里面, 去定义作为一个全局事件总线的对象, 然后先绑定, \$on(事件名, 回调函数) 再去组件中分发这个数据, \$emit(事件名, 数据)

3. vuex 也可以实现任意组件间通信，是一个专门为vue应用程序设计的管理多组件共享状态的vue插件，任意组件都可以读取到vuex中store的state对象中的数据，任意组件都可以通过dispatch或commit来触发store去更新state中的数据，一旦state中的数据发生变化，依赖这些数据的事件就会自动更新

19. 路由导航守卫有什么作用?如何实现?

1. 作用:
 1. 监视路由跳转
 2. 控制路由跳转
2. 为什么要用: 在做项目的时候，会有需求，比如说这个页面跳转到另外一个页面的时候，它的状态允不允许它跳转，
3. 应用: 在跳转到界面之前，进行用户权限限制，(是否已登录，是否有访问权限) 后台，在跳转登录界面，判断登录状态
4. 如何实现: 首先要看我们需要的是哪种路由守卫
 1. 全局守卫: 针对任意路由跳转

```
方法: router.beforeEach((to,from,next) => {})  
// to 即将要进入的目标路由对象  
// from 当前导航正要离开的路由  
// next: function 调用方法执行resolve钩子
```

2. 路由守卫:

```
前置守卫: beforeEnter: (to,from,next)=> {}  
组件守卫: 只针对当前
```

20. vuex的模型?

21. v-if和v-for能不能同时使用? 需要注意什么?

1. 能同时使用，但是不建议同时使用，因为v-for比v-if优先级高，所以每次重新渲染的时候会先遍历产生多个标签，然后v-if再对多个标签进行判断，这样对多个标签/对象进行判断，效率大大降低，尤其是在只需要对部分内容进行渲染的时候，同时使用就会存在这样的缺陷，那么最好的办法就是将v-if放在外部父级标签上，这样每次只执行一次，可以提高效率，或者是用计算属性来代替v-if，这样也能极大的提升效率。

22. vue底层是怎么实现数组监视的?

1. 在Observer中，通过Object.defineProperty()将data中所有层次属性添加上getter/setter方法
2. 为每一个属性都创建一个dep对象，用于后面更新
3. 在compile解析模板时，为每个表达式都创建一个用于更新对应节点的watcher
4. 在getter中建立dep与watcher之间的关系，dep和watcher相互包含
5. 在setter中，通过dep去通知所有watcher去更新对应的节点

23. vue项目中使用到了哪些技术?

24. 对keep-alive 的理解及使用场景，项目的哪些地方用到?

1. 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，有以下特性:
 1. 一般结合路由和动态组件一起使用，用于缓存组件;

2. 提供include和exclude属性，两者都支持字符串或正则表达式，include标识只有名称匹配的组件会被缓存，exclude标识任何名称匹配的组件都不会被缓存，器中exclude的优先级比include高；
3. 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
4. keep-alive是一个抽象组件，用来保存组件的渲染状态，它本身不会渲染DOM元素，也不会出现在父组件链中，使用keep-alive包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们，使用场景一般在：用户在某个列表页面选择筛选条件过滤出一份数据列表，由列表页面进入数据详情页面，再返回列表页面，列表页面仍然可以保留用户筛选或选中的状态，这时就可以用keep-alive来解决，keep-alive还可以避免组件反复创建和渲染，有效提升系统性能

25. 说一下redux和vuex的区别？

26. vue的代理路径在什么情况下使用？

27. vuex的计算属性如何使用？

28. vue中watch和computed 的区别

1. computed内部有缓存，多次读取数据只计算一次，当数据发生改变才会再次计算，watch监视器，初始状态不会调用，只有当数据发生变化才会调用，可以用来对数组/对象进行深度监视(deep) 可以通过immediate:true让watch一上来就执行一次。—— 包含异步操作使用watch，同步一般使用computed。

29. mpvue 和 vue 的区别

1. mpvue是由美团点评团队开发的一款基于vue的框架，从整个vue的核心代码上经过二次开发而形成的一个框架，相当于是给vue本身赋值，增加了开发微信小程序的能力

30. delete和vue.delete的区别

1. delete只能删除非响应式数据，不会触发vue的响应式更新
2. vue.delete可以删除响应式数据，删除后界面会发生变化。
3. delete只会删除数组的值，但是它依然会存在内存中
4. vue.delete会删除数组在内存中的占位

31. vuex内部如何监听数据状态？

32. 常用的vue指令？

1. v-model 双向数据绑定
2. v-bind 强制绑定解析表达式，可以省略v-bind 用 : 简写
3. v-if v-else v-else 满足条件才会输出
4. v-show 通过控制display样式来显示/隐藏
5. v-html 更新元素的innerHTML
6. v-text 更新元素的innerText
7. v-on 绑定事件监听
8. v-slot 插槽，提供具名插槽或需要接收prop的插槽
9. v-once 只初始渲染元素和组件一次，用于性能优化

33. vue和react的区别？

1. 相同点：
 1. 都是组件化开发
 2. 都支持props进行父子组件间数据通信
 3. 都支持数据驱动视图，不支持直接操作dom,更新状态数据界面就会自动更新
 4. 都支持服务器端渲染
 5. 都支持native方案，react的React Native，Vue的Weex

2. 不同点:

1. 数据绑定, vue实现了数据的双向绑定, react数据流向是单向的
2. 组件写法不一样: React推荐写法是jsx, 也就是把html和css都写在JavaScript, vue推荐的做法是webpack+vue-loader的单文件组件格式
3. state对象在react应用中不可变, 需要使用setState方法更新状态, 在Vue中, state对象不是必须的, 数据由data属性在vue对象中管理
4. virtual DOM不一样, vue会跟踪每一个组件的依赖关系, 不需要更新渲染整个组件, 而对于React而言, 每当应用的状态被改变时, 全部组件都会重新渲染, 所以react中会需要shouldComponentUpdate这个生命周期函数方法来进行控制

34. vue2脚手架和vue3脚手架的区别?

35. 在vue中如何让下拉框默认选中?

1. 循环出select内的数据, 发现原本默认显示第一条的select框变成了空白, 要选择后才有显示, 那就出现了一个如何让这个下拉框默认选中的问题
2. 在vue中, 使用select下拉框主要靠v-model来绑定选项option的value值
3. 关于select选项的写法:

1. 写在HTML中

```
<select name="xxx" id="stuts" v-model="seletced" option value="" 请选择
data:{selected: "" } 需要默认什么值, 就写什么值
```

2. option选项内容写在js中, 需要v-for来遍历

data 要遍历的数据

created 在组件创建之后, 把默认选项中的value值赋给绑定的属性, 如果没有的话, select中初始化会是空白的, 默认选中无法实现

36. 事件总线怎么用, 需要注意什么?

1. vue组件中, 数据传递最常用的就是父子组件之间的传递, 父传子通过props向下传递, 数据给子组件, 子传父通过\$emit发送时间, 并携带数据给父组件, 两个组件毫无关系, 就用全局事件总线: EventBus, 相当于一个全局的仓库, 任何组件都可以去这个仓库里获取

1. 初始化 `import vue from 'vue'`

2. 因为是一个全局的仓库, 所以初始化需要全局初始化

```
const EventBus = new vue()
```

3. 在已经创建好的vue实例原型中创建一个EventBus

```
vue.prototype.$EventBus = new vue()
```

4. 分发事件

```
this.$bus.$emit('eventName', data)
```

5. 接收事件

```
this.$bus.$on('eventName', 回调函数)
```

2. 如果A组件向事件总线发送了一个事件，并传递了参数。然后B组件对该事件进行了监听，并获取了传递过来的参数，但是，这时我们离开B组件，然后又再次进入B组件，又会触发一次对事件的监听，这时事件总线里就有两个监听了，如果返回进入B组件的话，那么就会有多个监听
3. 总而言之，A组件只向事件总线发送了一次事件，但B组件却进行了多次监听，EventBus容器中有很多个一模一样的事件监听器，事件只触发一次，但监听事件中的回调函数执行了很多次
4. 解决办法：在组件离开，也就是销毁前，将该监听事件移除，以免下次重复监听。

```
this.$bus.$off('要移除的事件名')
```

37. 在vuex 里，state中数据都可以在哪里修改？你的项目里哪里的数据放到了vuex里保存？

38. 开发中遇到的问题？

1. params传参刷新参数丢失问题，vue-router进行路由跳转的时候可通过params传参，如果在注册路由的时候没有使用占位符进行注册:'/home/:id',首次路由跳转可以获取params参数，再次刷新页面params数据丢失，解决方法:注册路由的时候写好占位符
`path: '/home/:id'`
2. Vue单页面应用中刷新页面，Vuex数据丢失：
 1. Vuex数据保存在运行内存中，vue实例初始化的时候为其分配内存
 2. 当刷新页面的时候重新初始化Vue实例，所以重新为Vuex分配内存导致之前保存的数据丢失
 3. 如何解决：将Vuex中的数据每次同步更新保存到sessionStorage中，在页面刷新之前获取vuex的数据，将数据保存在sessionStorage中，页面加载后从sessionStorage中获取。

39. vue响应式原理

1. 说到vue的响应式原理，其实就是在说：当我们修改了data当中的数据时，组件界面是如何自动更新的，下面来详细的说一下
 1. 在初始化的时候，vue内部通过Object.defineProperty()给vm添加与data对象中对应的属性
 2. 在getter中，读取data中对应的属性值返回 ==> 当我们通过this.msg 读取数据的时候，读取的就是data当中对应的属性值
 3. 在setter中，将最新的值保存到data中对应的属性上 ==> 当我们通过this.msg = value修改时，value保存在data中对应的属性上
 1. 数据代理作用：【这样做的目的就是简化了对组件对象内部的数据对象的属性数据的操作(读写)】
 4. 在observer中，通过Object.defineProperty()方法给data中所有层次的属性都添加getter和setter
 5. 给每个属性都创建一个dep对象，用于更新，在模板解析的时候，给每个表达式都创建了一个用来更新对应节点的watcher
 6. 通过getter来建立dep与watcher之间的关系：dep与data中的属性一一对应，watcher与模板中的表达式一一对应，一个dep中保存了多个包含watcher的数组，一个watcher中，保存了多个包含dep的对象
 7. 通过setter，让dep来通知所有的watcher去更新对应的节点

创建MVVM对象的时候，内部进行了数据代理的操作，进而会把data对象中的数据进行劫持，创建劫持对象，内部遍历data中所有的数据，每个属性都会创建对应的dep对象，模板解析的时候内部最终会创建watcher对象，html模板中有一个表达式就会创建一个对应的watcher，在watcher内部获取表达式值的时候，开始建立dep和watcher的关系，如果页面中发生了数据的变化(表达式/data中的属性值改变)，就会找到该属性对应的dep对象，通知内部对应的watcher对象更新数据，watcher内部会调用updater对象中的相关方法进行更新，进而重新渲染界面

40. v-model的双向数据绑定

1. 通过v-model来实现双向数据绑定，v-model的本质就是 将动态的data数据通过value属性传给input显示，就是从data到view，再通过给input标签绑定input监听，一旦输入改变了就会读取最新的值保存到data对应的属性上，双向数据绑定其实就是在单项数据绑定的基础之上加入了input事件监听。

单项数据绑定: data ==> view

双向数据绑定: view ==> data

41. vue3的特点

1. 支持vue2的大多特性
2. 更好的支持TS
3. 打包和渲染时间更快
4. 内存减少54%
5. 使用Proxy代替了defineProperty实现数据响应式
6. 重写了虚拟DOM的实现和Rree-Shaking
7. 使用ref和reactive来代替data，也支持data，ref用来定义基本类型(某一个数据)，reactive用来定义引用类型(处理多个数据)

42. 比较Vue2和Vue3的响应式

1. Vue2:

1. 对象: 通过defineProperty对对象的已有属性值进行读取或修改进行劫持
2. 数组: 通过重写数组的一系列方法来实现对元素修改的劫持
3. 存在的问题:

1. 对象直接新添加的属性或删除已有属性，界面不会自动更新
2. 直接通过下标替换元素或更新length，界面不会自动更新

2. vue3:

1. 通过Proxy代理拦截对data任意属性的任意(13)种操作，包括属性值的读写，属性的添加，属性的删除等
2. 通过Reflect(反射)动态的对代理对象的相应属性进行特定的操作
3. 添加新属性界面会自动更新

43. 组件与Vue的关系

1. 组件是vue的"子类型"
2. 组件对象的原型对象的原型对象是Vue的原型对象
3. 组件对象能访问到vue原型对象上的方法/属性

1. postman与测试接口

1. 定义接口请求函数模块

在vuex的异步action中调用接口请求函数

将API挂载到Vue的原型对象上, 在组件中调用接口请求函数与后台交互: `Vue.prototype.$API = this`

当后台接口还未完成时, 先mock数据, 可以使用mockjs, 当然有的公司可能有自己的一套

2. 功能模块

商品搜索列表
商品详情
购物车
登陆与注册
订单交易/结算
支付
个人中心/订单列表

3. 使用的库

vue
vue-router
vuex
vee-validate
vue-lazyload
element-ui

axios
mockjs
nprogress
uuidjs

swiper
qrcode
lodash

4. 注册流程

1. 前台:

1. 输入注册需要的相关信息(用户名, 密码), 进行前台表单验证, 如果不通过, 提示错误
2. 发送注册的ajax请求(post), 携带注册接口需要的相关数据(用户名, 密码)

2. 后台:

1. 获取到注册请求携带的参数, 去数据库种判断是否已经存在
 1. 如果已存在, 返回用户已存在的提示信息
 2. 如果不存在, 保存到数据库, 返回成功的数据

3. 前台接收到响应:

1. 如果是不成功的数据, 进行提示
2. 如果是成功的数据, 自动跳转到登录页面

5. 登录流程

1. 前台：
 1. 输入登录需要的相关信息(用户名, 密码), 进行前台表单验证, 如果不通过, 提示错误
 2. 发送登录的ajax请求(post), 携带登录接口需要的相关数据(用户名, 密码)
2. 后台: 获取到登录请求携带的参数, 去数据库种查找是否存在
 1. 如果不存在, 返回登录失败的信息
 2. 如果存在, 生成一个新的token字符串, 将token与用户信息一起返回
3. 前台接收到响应:
 1. 如果是不成功的数据, 进行相关提示
 2. 如果是成功的数据:
 1. 将用户信息和token都保存到vuex中
 2. 将token保存到localStorage中
 3. 跳转到首页或redirect页面

6. 自动登录流程

1. 页面一加载, 发送请求根据token获取用户信息
2. 利用全局前置守卫:
 1. 一旦发现当前没有登录, 当之前登录过(有token, 没有用户信息)
 2. 发送请求根据token获取用户信息:
 1. 成功了: 保存用户信息及token, 存入vuex
 2. 失败了: 说明token过期, 清除登录用户信息(token), 强制跳转到登录页面

7. 支付流程

1. 获取订单交易数据
2. 提交下单请求, 获取订单id
3. 根据订单id获取支付信息: 金额, 支付url
4. 支付:
 1. 根据支付url生成支付二维码图片显示
 2. 扫码支付
 3. 轮询请求获取订单状态
5. 获取到支付成功后跳转页面

HTML相关

1. HTML性能优化

1. HTML标签有始终。减少浏览器的判断时间
2. 把script标签移到HTML文件末尾, 因为JS会阻塞后面的页面的显示。
3. 减少iframe的使用, 因为iframe会增加一条http请求, 阻止页面加载, 即使内容为空, 加载也需要时间
4. id和class, 在能看明白的基础上, 简化命名, 在含有关键字的连接词中连接符号用'-', 不要用'_'

5. 保持统一大小写，统一大小写有利于浏览器缓存，虽然浏览器不区分大小写，但是w3c标准为小写
6. 清除空格，虽然空格有助于我们查看代码，但是每个空格相当于一个字符，空格越多，页面体积越大，像google、baidu等搜索引擎的首页去掉了所有可以去掉的空格、回车等字符，这样可以加快web页面的传输。可以借助于DW软件进行批量删除 html内标签之间空格，sublime text中ctrl+a，然后长按shift+tab全部左对齐，清除行开头的空格
7. 减少不必要的嵌套，尽量扁平化，因为当浏览器编译器遇到一个标签时就开始寻找它的结束标签，直到它匹配上才能显示它的内容，所以当嵌套很多时打开页面就会特别慢。
8. 减少注释，因为过多注释不光占用空间，如果里面有大量关键词会影响搜索引擎的搜索
9. 使用css+div代替table布局，去掉格式化控制标签如：strong，b，i等，使用css控制
10. 代码要结构化、语义化
11. css和javascript尽量全部分离到单独的文件中

2. 浏览器兼容问题？

1. 不同的浏览器对同一段代码的解析是不一样的，造成页面显示效果不一致的情况
2. 不同的浏览器，随便写几个标签，不加样式控制的情况下，各自的margin和padding差异较大，如何解决：在一般页面中通常使用*{margin:0;padding:0}来去除默认样式，但是在项目中，一般会引入一个css重置样式文件，来对这些默认的风格进行处理。
3. IE6双边距问题：在ie6中设置了float，同时又设置margin，就会出现边距问题，解决：设置display:inline
4. 当标签的高度设置小于10px，在IE6、IE7中会超出自己设置的高度，解决：超出高度的标签设置overflow:hidden,或者设置line-height的值小于设置的高度
5. 图片默认间距问题，解决：display:block / float:left / 设置图片父元素字体大小为0：font-size:0
6. 外边距重叠：当相邻的两个元素都设置了margin边距时，margin会取最大值显示，舍弃最小值。

3. 前台数据存储

1. cookie

1. 本身用于浏览器和Server通讯
2. 被借用到本地存储
3. 可用document.cookie读取或保存：得到
4. 可用利用cookies工具库简化编码
5. 缺点：

1. 存储大小有限，最大4kb
2. http请求时会自动发送给服务器，增加了请求的数据量。
3. 原生的操作语法不太方便操作cookie
4. 浏览器可用设置禁用

2. sessionStorage：数据保存在当前会话内存中，关闭浏览器则自动清除

3. localStorage：保存在本地文件中，除非编码或手动删除，否则一直存在

4. localStoarge 与 sessionStorage相同点：

1. 纯浏览器端存储，大小不受限制，请求时不会自动携带
2. 只能保存文本，如果是对象或数组，需要转换为JSON
3. API相同：

1. setItem(key,value)
2. getItem(key,value)
3. removeitem(key,value)

5. 注意：session 后台数据存储

6. 区别cookie与session:

1. cookie保存在浏览器端(前台可以操作)
2. session保存在服务器端(前台不能操作)
3. session依赖于cookie(session的id以cookie的形式保存在浏览器端)

4. 从输入url到渲染出页面的整个过程

简单表达:

1. 解析域名
2. 建立 TCP 连接
3. 浏览器发送请求到服务器,
4. 服务器处理请求返回响应给浏览器
5. 浏览器解析渲染页面
6. 断开 TCP 链接

详细表达:

1. DNS解析: 将域名地址解析ip地址

1. 浏览器DNS缓存
2. 计算机DNS缓存
3. 路由器DNS缓存
4. 网络运营商DNS缓存
5. 递归查询

2. TCP链接: TCP三次握手:

1. 客户端发送服务端: 我准备好了, 请你准备一下
2. 服务端发送客户端: 我也准备好了, 请你确认一下
3. 客户端发送服务端: 确认完毕

3. 发送请求给服务器

1. 将请求报文发送过去

4. 服务器返回响应

1. 将响应报文发送过来

5. 解析渲染界面

1. 遇到html, 调用html解析器, 解析成DOM树
2. 遇到CSS, 调用CSS解析器, 解析成CSSOM树
3. 遇到js, 调用js解析器(js引擎), 解析执行js代码
 1. 可能会修改元素节点, 重新调用html解析器, 解析成新的dom树
 2. 可能会修改元素样式节点, 重新调用css解析器, 解析成cssom树
4. 将DOM + CSSOM = Render Tree(渲染树)
5. layout布局: 计算元素的位置和大小信息
6. render渲染: 将颜色/文字/图片等渲染上去

6. 断开链接: TCP四次挥手

断开请求链接2次, 断开响应链接2次

1. 客户端发送服务端: 请求数据发送完毕, 可以断开了
2. 服务端发送客户端: 请求数据接收完毕, 可以断开了
3. 服务端发送客户端: 响应数据发送完毕, 可以断开了
4. 客户端发送服务端: 响应数据接收完毕, 可以断开了

5. 开发中遇到的问题？

1. params传参刷新参数丢失问题，vue-router进行路由跳转的时候可通过params传参，如果在注册路由的时候没有使用占位符进行注册: '/home/:id', 首次路由跳转可以获取params参数，再次刷新页面params数据丢失，解决方法:注册路由的时候写好占位符
`path: '/home/:id'`
2. Vue单页面应用中刷新页面，Vuex数据丢失：
 1. Vuex数据保存在运行内存中，vue实例初始化的时候为其分配内存
 2. 当刷新页面的时候重新初始化Vue实例，所以重新为Vuex分配内存导致之前保存的数据丢失
 3. 如何解决：将Vuex中的数据每次同步更新保存到sessionStorage中，在页面刷新之前获取vuex的数据，将数据保存在sessionStorage中，页面加载后从sessionStorage中获取。
3. localStorage 和 sessionStorage 的区别：
 1. sessionStorage保存的数据在关闭浏览器后自动清除，localStorage保存的数据在关闭后还会存在。

react相关

1. react和redux的联系？

1. 一些小型项目：a ==> b, 随着业务增加，a ==> b ==> c ==> d ==> e, 但是b, c, d不需要这些数据，于是想要找个地方共享数据，存放一个数据对象，外界能访问这个数据，外界也能修改这个数据，当数据有变化的时候通知订阅者
2. react是一个视图层框架，多个组件之间传参很麻烦，数据无法共享。
3. redux：将所有数据放在公共的存储器 ==> store 所有的数据由store存储
4. redux的工作流程：
 1. store的创建
 2. 从redux里引入createStore方法并创建store
 3. reducer的创建，返回一个方法，接收store和action
 4. 将reducer传递给store，两者进行关联
 5. 在组件中使用数据，页面中引入store，并调用store提供的getState()方法
 6. 将项目与redux devtools融合，便于调试redux
 7. 修改store中的数据
 8. 组件修改数据的方法定义一个action，并传入store中的dispatch方法，store接收到action会自动将之前的数据和action派送给reducers，reducers根据action的type类型，来将原来的数据深拷贝之后，再修改并返回新的store数据
 9. reducers返回新的数据给store，在页面组件上，通过在store.subscribe()方法来订阅store中的数据变化，该方法会传入修改组件state的方法，来更新当前组件的state，当store中数据发生变化时，会自动调用，store.Subscribe方法区别实时更新组件的state目的，感知到了store中数据变化，在this.setState中传store.getState() 从store中重新获取数据，

10. Redux的三项基本原则：

1. store是唯一的，项目只能有一个store
2. 只有store能够改变自己的内容，store拿到reducers返回的新数据，自己对自己的数据进行了更新+
3. reducer必须是纯函数(纯函数：给定固定的输入就有固定的输出，而且不会有副作用，及不修改输入)

2. props和setState()

1. React三大属性：state、props、refs
2. state被称为组件的状态，通过更新组件的state重新渲染组件，state内容更新后，必须要调用setState方法重新渲染，达到更新视图的目的
3. props：组件间通信，单项数据流说的就是props，props本身是不可变的，也是不可更改的
4. props一定来源于默认属性或者父组件传递过来

尽量少用state，多用props

props用于定义外部接口，state用于记录内部状态

props的赋值在于外部事件使用组件，state的赋值在于组件内部组件不应该改变props的值，而state存在的目的就是让组件来修改，每一次通过setState函数修改state就改变了组件的状态，但是组件的props不能被修改，如果多个子组件用props修改了那全都会被修改

state状态，只是用来控制这个组件本身自己的状态，用state完成行为的控制，数据的更新，界面的渲染，由于组件不能修改传入的props，所以需要记录自身的数据变化

setState，更新是异步的，还要负责触发重新渲染，要经过diff算法，最终决定是否进行重新渲染

如果让你来独立开发一个项目你该怎么做，有哪些流程？框架？页面？原型图？

扩展

1. data中为什么只能是函数不能是对象？

1. 一个组件中的data有多个实例，多个实例指向同一对象，引用，内部存的是地址值，所以会有影响。
2. 一个组件的多个标签实例的data对象应该是独立的，不能共享。
2. vue中的mixin
 1. 可以用来定义公用的变量，在每个组件中使用，引入组件之后，各个变量是相互独立的，值的修改在组件中不会相互影响。
3. data中的一个数组，v-for遍历完数组，在method里往data中添加数据，新添加的数据是否会显示到页面？
 1. 不会
4. watch有监测不到的情况么？
5. vue过渡动画怎么实现？
6. 子组件在父组件的哪个钩子出现？
7. 接口请求一般放在哪个生命周期？
8. created和mounted区别,在created可以发送请求吗？
9. 钩子的activated和mounted的区别？
 - 初始mounted之后会调用activated，
10. 页面加载会调用几个钩子？
 - beforeCreate，实例初始化之后调用
 - created，编译模板，可以通过this操作data和methods中的数据
 - beforeMount，将要挂载到页面，还不能通过ref找到对应的标签对象
 - Mounted，已经挂载到页面，可以通过ref找到对应的标签，可以在此发送ajax请求
11. vue的生命周期有哪些？
12. 第一个能看到this的生命周期函数？
13. 第一个能发请求的生命周期函数？
14. 第一个能操作dom的生命周期函数
15. created钩子做了什么？
16. keep-alive缓存组件
17. 关于keep-alive组件的使用
18. vue的自定义事件
19. vue组件通信方式
 1. props (父子间相互通信)
 1. 通过标签属性传递，使用的较多，基本通信方式。
 2. 两种情况：
 1. 函数属性：子向父传递数据，子组件调用父组件的方法，更新父组件的数据，通过参数将数据传递给父组件
 2. 非函数/一般属性：父向子传递数据
 3. 缺点：祖孙之间通信，需要逐层传递，兄弟间通信，需要借助于共同的父组件，比较麻烦。
 2. vue的自定义事件(父子间通信)
 1. 事件监听的相关方法：
 1. \$on(event, callback) 监听事件监听
 2. \$emit(eventName) 分发事件
 3. \$off(eventName) 解绑事件监听
 4. \$once(event, callback) 绑定事件监听(只响应一次)

2. 理解事件的两个操作:

1. 绑定事件监听: 事件名和回调函数(可以接收任意多个参数数据)
2. 分发/触发事件: 事件名(需要与绑定监听的事件名一致)和数据(可以指定任意多个数据)

3. 绑定事件监听: `$on('eventName',data) ==>` 用来接收数据

```
<Header @addTodo="addTodo" />
```

4. 分发事件: `$emit('eventName',(data) => {})` ==> 用来传递数据

```
// 分发事件
this.$emit('addTodo', title)
```

5. 事件处理的方法是定义在Vue原型对象上

6. 同一个组件对象来绑定监听后分发事件, 回调函数才执行

7. 用来实现子组件向父组件通信, 功能类似于函数属性

3. 全局事件总线globalEventBus(任意组件间通信)

1. 什么是全局事件总线:

- 是一个对象
- 必须有绑定监听/分发事件/解绑监听的3个方法
- 任意的组件都可以访问这个对象
- 利用globalEventBus实现任意关系组件A向B通信:
 - 方式一: 创建一个新的vm作为总线对象保存到Vue的原型对象上

```
// 创建一个vm对象作为全局事件总线对象
Vue.prototype.$globalEventBus = new Vue()
```

- 方式二: 通过生命周期钩子beforeCreate() 来实现, 直接将最外层vm对象作为事件总线对象

```
const vm = new Vue({
  // vm对象一旦创建就立即调用, 此时通过ref获取不到data中的数据
  beforeCreate() {
    Vue.prototype.$globalEventBus = this // 将当前vm作为全局事件总线对象保存到Vue的原型对象上
  },
  render: h => h(App)
}).$mount('#root')
```

- A 通过总线对象来分发事件

```
// 通过总线对象分发事件
this.$globalEventBus.$emit('deleteTodo',this.index)
```

- B 给总线对象绑定事件监听

```
mounted() {
  // 给总线对象绑定事件监听
  this.$globalEventBus.$on('deleteTodo', this.deleteTodo)
},

beforeDestroy() {
  // 解绑事件监听
  this.$globalEventBus.$off('deleteTodo')
},
```

4. 消息订阅与发布(任意组件间通信)

1. 提供一个全局对象能进行订阅消息与发布消息的操作
2. 订阅消息：绑定事件监听
3. 发布消息：分发事件
4. 下载pubsub工具包：

```
npm install pubsub-js
```

5. 语法：

1. 订阅消息： `PubSub.subscribe('msgName', (msg, data) => {})`
2. 发布消息： `PubSub.publish('msgName', data)`
3. 取消订阅： `PubSub.unsubscribe(token)`

5. slot 插槽

1. Vue中用来实现父组件向子组件传递标签内容

2. 插槽分类：

1. 默认插槽：没有名字的插槽，一个模板页面最多只能有一个默认插槽
2. 命名插槽：有名字的插槽
3. 作用域插槽：通过父组件读取子组件的数据来决定向子组件传递什么样的结构内容
4. 作用域插槽编码：

1. 子组件：通过slot标签属性向父组件传递数据

```
<slot :row="item" :index="index"></slot>
```

2. 父组件：通过slot-scope属性来接收所有传递过来的数据，根据数据决定要传递的结构内容

```
<span :style="{color:scope.index % 2 === 0 ? 'blue' :
'green'}">
  {{scope.index + 1}} --- {{scope.row.text}}
</span>
```

3. 插槽模板是在父组件解析好过后，再传递给子组件
4. 插槽模板内容是通过子组件标签体传递过去的

6. vuex

data当中的数据就叫状态数据

1. 可以实现对状态数据的统一管理，实现任意组件的通信

2. 集中式的管理状态数据，进行任意组件间通信

3. Vuex的使用步骤：

1. 创建目录：

1. src目录中创建一个目录vuex，内部创建一个store.js文件

2. src目录中创建一个目录store，内部创建一个index.js文件

2. 安装vuex的插件，然后引入，声明使用，暴露及注册。

1. 安装： `npm install vuex`

2. 使用：在store.js/index.js 中，引入Vue，引入Vuex,声明使用该插件，实例化Vuex.Store对象，并暴露出去，在main.js/index.js(主文件)中引入暴露出来的对象，并注册到Vue的配置对象中

```
// 引入Vue
import Vue from 'vue'
// 引入Vuex
import Vuex from 'vuex'
// 声明使用插件
Vue.use(Vuex)
// 实例化对象并暴露出去
export default new Vuex.Store({

})
```

```
// 引入store对象
import store from './vuex/store.js'

new Vue({
  // 注册Vuex的store仓库对象
  //store:store // 完整写法
  store // 同名简写
}).$mount('#root')
```

3. Vuex的四个核心概念

1. state 包含了多个初始状态数据

2. mutations 包含了多个直接修改状态数据的方法的对象(同步)

3. actions 包含了多个间接修改状态数据的方法的对象(异步)

4. getters 包含了多个状态数据的计算属性的get方法的对象

20. 父组件如何获取子组件实例对象的方法

1. ref

2. children

21. 父子组件互相调用的方法

1.

22. vue之间组件通信方式

23. 常用的事件修饰符有哪些

1.

24. vue组件间通信的方式

25. 如何实现兄弟组件之间的通信

1. 全局事件总线
26. v-on可以监听多个方法吗
27. vue双向数据绑定原理
28. 解释下v-model怎么实现的
29. \$set和\$nextTick的使用场景
30. Vue.use()做了什么
31. 说说Vue的mixin技术
32. 如何理解Vue中的数据代理

在实例化Vue对象的时候,需要传入配置对象,内部会获取该配置对象中的data对象数据,并且存储到data变量及_data属性中

然后通过Object.keys获取data中的每个属性,进行遍历操作,遍历的过程中内部调用了Object.defineProperty()方法把data对象中的每个属性一个一个的添加到vm实例对象上,从而实现了数据代理

内部遍历了data对象,通过了Object.defineProperty()方法把data中的每个属性一个一个的添加到vm对象上

33. 双向数据绑定的理解

在创建Vue实例对象的时候,内部会进行数据劫持操作,对应个数的属性会创建对应个数的dep对象,开始模版解析,模版解析内部会创建文档碎片对象,内部会把html容器中所有的节点全部的扔进文档碎片对象,在内存中进行操作,遍历所有的节点,判断当前的节点是不是标签,input标签,然后获取里面所有的属性,判断是不是指令,是不是普通指令,如果是普通指令,在调用bind方法,更新文本框的value属性的值,bind方法内部还要创建watcher对象,在为input标签通过addEventListener方法绑定input事件及对应的回调函数

一旦在页面中修改input标签的数据,会自动的进入input事件对应的回调函数内部,修改vm.data下面的msg属性,一旦修改vm.data.msg属性值,就会自动的进入mvm.js文件的Object.defineProperty方法内部的set方法,就会自动的进入到observer.js文件中的Object.defineProperty方法内部的set方法,然后就会调用dep.notify方法让当前的dep对象中的subs数组遍历里面每个sub,sub就是watcher对象,然后watcher对象中的update方法内部调用run方法,然后对比原来数据和新数据是否一致,进行数据的更新再次调用创建watcher对象传入的回调函数,内部实际上是updater对象中的相关方法modelUpdater更新input标签中的value属性的值,同时对应的p标签中的表达式的值也会自动更新

34. 双向数据绑定

- 1). 双向数据绑定是建立在单向数据绑定(model==>View)的基础之上的
- 2). 双向数据绑定的实现流程:
 - * 在解析v-model指令时, 给当前元素添加input监听
 - * 当input的value发生改变时, 将最新的值赋值给当前表达式所对应的data属性

35. mode的两种模式:

1. history 不带#号
2. hash 带#号

36. 自定义事件和原生事件的区别?

- event, 事件对象, 本质上是一个数据, 原生中: 事件数据对象

37. 区别原生事件和自定义事件

38. 区别原生的\$event和自定义的\$event

39. 组件间通信/传值的理解:

1. 一个组件向另一个组件传递数据 ==> 组件间通信/传值

40. 组件间通信方式有哪些？

1. props —— 父子间通信

1. 函数属性：子 ==> 父，子组件调用父组件的方法，更新父组件的数据，需要的数据通过参数传递给父组件
2. 非函数/一般属性：父 ==> 子，父组件向子组件传递数据

2. vue自定义事件总线 —— 父子间通信

1. 相关语法

- a. EventBus: 包含所有功能的全局事件总线对象
- b. EventBus.on(eventName, listener): 绑定事件监听
- c. EventBus.emit(eventName, data): 分发事件(同步)
- d. EventBus.off(eventName): 解绑事件监听

3. 全局事件总线

4. 消息订阅与发布

5. v-model

6. .sync

7. \$attrs与\$listeners

8. \$ref,\$children与\$parent

9. provide与inject

10. Vuex

11. 插槽slot —— 父子组件通信

41. 计算属性computed与监视watch的区别

1. 计算属性computed，定义计算属性方法根据已有的数据进行计算返回一个要显示的新数据在页面中使用{{计算属性名}}来显示返回的数据，计算属性默认相当于只有getter来根据已有数据计算返回一个新数据值，也可以指定setter来监视我们主动修改当前计算属性值
2. 监听watch，通过watch配置或vm的\$watch()来监视指定的属性值的变化，当属性变化时，回调函数自动调用，在函数内部进行特定处理
3. 如果是根据已有数据，来动态同步确定一个新的数据值，那么一般都选择computed，watch可以做异步操作(ajax请求)后才确定新的数据值，watch还有一个特点就是可以对数组或对象进行深度(deep)监视。

1. **computed:** 定义计算属性，初始显示执行第一次，一旦依赖数据发生改变就会再次执行内部定义的属性(方法)。将执行返回的结果作为计算属性值，**this**指向为vm对象(只要是vue控制的回调函数都是vm对象)，**computed**内部有缓存，多次读取只计算一次，若数据发生改变，则会再次计算
2. **watch /vm的\$watch:** 监视器/侦听器，监视指定的属性值的变化，初始值状态不会调用，只有当数据发生改变，才会调用，**watch**可以对数组或对象进行深度(deep)监视
3. 如果是根据已有数据，来动态**同步**确定一个新的数据值一般选择**computed**，如果包含**异步**请求(操作)的数据(**ajax, setTimeout**)，就选择**watch**
4. 计算属性默认相当于只有**getter**来根据已有数据计算返回一个新数据值，也可以指定**setter**来监视当前主动修改的计算属性值

42. 常用数组方法:

1. **split()** 以指定的字符为切割点，将字符串切割/分割为数组返回，参数：指定的切割子串，返回值:返回切割好的数组，如果不传任何参数，那么整体作为数组的一个元素返回，如果传的是空串，那么每个字符都会作为数组的元素
2. **reverse()** 翻转数组，返回翻转后的原数组

3. join() 以指定的连接符，将数组中的元素连接成一个字符串，参数：如果不传，默认以逗号连接，如果传空串，数组元素直接拼接，如果指定字符，那么就以指定字符连接
 4. map() 返回一个由原数组执行了回调函数过后的结果组成的新数组(根据条件，形成一个新数组)
 5. reduce() 对数组中的每个元素都执行一次提供的条件函数，最后汇总为单个返回值(根据条件，给定初始值，返回累加结果，一般用于计算总和或累计)
 6. filter() 返回一个通过条件判断的元素组成的新数组，如果没有，则返回空数组
 7. find() 返回数组中满足条件的第一个元素的值，否则返回undefined
 8. findIndex() 返回数组中满足条件的第一个元素的索引，如果没有找到则返回-1
 9. every() 判断一个数组内的所有元素是否都能通过某个条件判断，如果**全都**满足则返回true，否则返回false
 10. some() 判断一个数组内的所有元素是否有(1个或1个以上)元素通过某个条件判断，如果有则返回true，如果都没有则返回false
43. Vue内部重写了数组的一些方法：
1. push()、pop()、shift()、unshift()、splice()、sort()、reverse()
 2. 通过调用数组原始方法去更新数组元素，再去更新页面
44. 什么是路由：
1. 路由就是一个映射/对应关系(key:value)，key是path/路径, value是函数(function)或者组件(component)
 2. 前台路由：value是路由组件, 用来更新显示当前路由界面
 3. 后台路由：value是回调函数, 用来处理请求, 返回响应数据
 4. 相关API:

创建路由器，注册路由

```
new VueRouter({
  mode: 'hash/history', // 路由的2种模式
  routes: [ // 注册项目中的多个路由
    { // 一般路由
      path: '/about',
      component: About
    },
    { // 自动跳转的路由
      path: '/',
      redirect: '/about'
    }
  ]
})
```

注册路由器

```
import router from './router'
new Vue({
  router,
})
```

2个路由相关组件标签

<router-link>: 路由链接，当点击路由链接时，就能自动显示对应的路由组件界面
<router-view>: 显示当前路由组件界面

45. 路由的传参方式：

1) params参数

注册路由的路径: 'detail/:id/:name'

路由链接的路径: 'detail/3/tom'

读取数据:

```
this.$route.params.id  
this.$route.params.name
```

2). query参数

路由链接的路径: 'detail/3?id2=5&name=tom'

读取数据:

```
this.$route.query.id2  
this.$route.query.name
```

46. 区别push路由跳转与replace路由跳转

push可以返回到原来的路由组件, replace不能。

47. 如何解决vuex数据丢失的问题

48. vuex和redux的区别

49. vuex和store的关系, vuex的五大属性

1. store本身是一个构造函数, 是vuex的实例

2. a,g,s,m,m

50. vuex的原理, 组件传参方式

1. 内部管理状态靠vm

51. 监听vuex自身数据

1. 如何监视vuex内部的数据, 首先读取到组件内, 通过watch, 本身没有set

52. vuex的理解

1. 几大部分组成,

2.

53. vuex如何外部改变内部的值

1. 组件改vuex的值? dispatch

2. 通过actions => mutations => state

54. 对于vuex的理解及使用场景

1. 涉及到多个组件共享状态

55. vue-router路由跳转传参方式路由组件懒加载实现?

56. vue-router中的全局守卫, 路由守卫, 组件守卫

57. vue-router的使用和理解

58. vue-router跳转的原理, 路由守卫

59. vue-router的路由传参

60. 前端路由的实现

61. hash路由和history路由实现原理

62. vue-router中hash和history的区别, 路由传参方式, location跳转

63. vue中\$route和\$router的区别, query和params的区别

64. vue路由模式的区别

65. 路由守卫都有哪些? 哪个守卫可以在next中传函数(beforeRouteEnter)

66. 路由传参的方式?

