面试总结

1、小程序数据双向绑定和 VUE 双向绑定的区别？

设置值：
在 vue 中,只需要在表单元素上加上 v-model,然后再绑定 data 中对应的一个值，当表单元素内容发生变化时，data 中对应的值也会相应改变，这是 vue 非常 nice 的一点。
但是在小程序中，却没有双向绑定这个功能。那怎么办呢？
当表单内容发生变化时，会触发表单元素上绑定的方法，然后在该方法中，通过 this.setData({key:value})来将表单上的值赋值给 data 中的对应值。
取值：
在 vue 中，通过 this.reason 取值；
小程序中，通过 this.data.reason 取值。
Vue 实现双向绑定的原理： Object.defineProperty();

2、小程序视图渲染结束后的回调函数？

setData 可以传递回调函数

```
this.setData({
    text: 'Set some data for updating view.'
}, function() {
   // this is setData callback
})
```

3、addeventlistener 和 onclick 的区别？

1.onclick 事件在同一时间只能指向唯一对象
2.addEventListener 给一个事件注册多个 listener
3.addEventListener 对任何 DOM 都是有效的，而 onclick 仅限于 HTML
4.addEventListener 可以控制 listener 的触发阶段，（捕获/冒泡）。对于多个相同的事件处理器，不会重复触发，不需要手动使用 removeEventListener 清除
5.IE9 使用 attachEvent 和 detachEvent

4、数据单向绑定和双向绑定的区别？

双向数据绑定给人的最大的优越感就是方便。当数据 data 发生变化时，页面自动发生更新。但是有一个缺点也是因为自动更新而导致的，因为这样你就不知道 data 什么时候变了,也不知道是谁变了,变化后也不会通知你,当然你可以 watch

来监听 data 的变化，但是这变复杂了，还不如单向数据绑定。
单向数据绑定的实现思路：
所有数据只有一份
一旦数据变化，就去更新页面(data-页面)，但是没有(页面-data)
如果用户在页面上做了变动，那么就手动收集起来(双向是自动)，合并到原有的数据中。
其实单向绑定也有双向绑定的意味，不过页面变动后数据的变化不会自动更新。
方神解析了这个魔法：双向绑定 = 单向绑定 + UI 事件监听

5、wxss 和 css 的区别？

选择器：
wxss 不支持*选择所有元素
wxss 不能直接通过 css3 中的 background-image 属性来设置显示的背景图片
字体：
支持 font-family 属性
支持自定义字体@font-face，暂时字体文件用 base64 代替实现。使用文件的方式，可能要上传资源到对应服务器。
其他：
支持动画 @keyframes，貌似不能加厂商前缀。
支持过渡 transition。

6、小程序和组件的生命周期

小程序生命周期：
小程序注册完成后，加载页面，触发 onLoad 方法。
页面载入后触发 onShow 方法，显示页面。
首次显示页面，会触发 onReady 方法，渲染页面元素和样式，一个页面只会调用一次。
当小程序后台运行或跳转到其他页面时，触发 onHide 方法。
当小程序有后台进入到前台运行或重新进入页面时，触发 onShow 方法。
当使用重定向方法 wx.redirectTo(OBJECT)或关闭当前页返回上一页
wx.navigateBack()，触发 onUnload
小程序组件生命周期：
created 组件实例化，但节点树还未导入，因此这时不能用 setData
attached 节点树完成，可以用 setData 渲染节点，但无法操作节点
ready 组件布局完成，这时可以获取节点信息，也可以操作节点
moved 组件实例被移动到树的另一个位置
detached 组件实例从节点树中移除

7、页面如何传参？

第一种就是页面跳转，依靠跳转的 url 带参传值，第二种就是本地存取，这里有分为同步或者异步，以及移除或清除本地缓存的 API 接口，还有一种就是将值设置为全局变量，在需要的页面获取，这种的话就不是很建议使用，因为很多需要传递的值是你需要从后台拿数据的。

8、小程序从 0-1 的流程

注册小程序账户
安装微信开发者工具
选框架，新建项目
开发-调试
上传-提交审核-通过后发布

9、线上版本出错怎么补救

登录微信公众平台可以回退到上个版本

10、初次提交小程序审核要多久？

1-7 天

11、闭包

闭包就是能够读取其他函数内部变量的函数，即定义在函数内部的函数，且该函数作为返回值，在本质上就是将函数内部和函数外部连接起来的桥梁。最大用处一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。
使用闭包的注意点
1）由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在 IE 中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。
2）闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

12、Promise

Promise 是一个对象，它代表了一个异步操作的最终完成或者失败。因为大多数人仅仅是使用已创建的 Promise 实例对象。

本质上，Promise 是一个被某些函数传出的对象，我们附加回调函数（callback）使用它，而不是将回调函数传入那些函数内部
Promise 最直接的好处就是链式调用

13、Promise 实现 axios

```
return new Promise() {
       var xhr = new XMLHttpRequest();
       xhr.open(‘get’,’ example.php’,false);
       onreadystatechange 事件中 readyState 属性等于 4。响应的 HTTP 状态为 200(OK)或者 304(Not Modified)。// 这里 resolve
       xhr.send(data);
   }
```

14、组件化和模块化的区别

组件是把重复的代码提取出来合并成为一个个组件，组件最重要的就是重用（复用），位于框架最底层，其他功能都依赖于组件，可供不同功能使用，独立性强。

模块是分属同一功能/业务的代码进行隔离（分装）成独立的模块，可以独立运行，以页面、功能或其他不同粒度划分程度不同的模块，位于业务框架层，模块间通过接口调用，目的是降低模块间的耦合，由之前的主应用与模块耦合，变为主应用与接口耦合，接口与模块耦合。模块就像有多个 USB 插口的充电宝，可以和多部手机充电，接口可以随意插拔。复用性很强，可以独立管理。

组件就像一个个小的单位，多个组件可以组合成组件库，方便调用和复用，组件间也可以嵌套，小组件组合成大组件。

模块就像是独立的功能和项目（如淘宝：注册、登录、购物、直播…），可以调用组件来组成模块，多个模块可以组合成业务框架。

组件化模块化优点是开发调试效率高、可维护性强、避免阻断、版本管理更容易

组件化是指解耦复杂系统时将多个功能模块拆分、重组的过程，有多种属性、状态反映其内部特性。

组件化是一种高效的处理复杂应用系统，更好的明确功能模块作用的方式。

组件是可以单独开发、测试。允许多人同时协作，编写及开发、研究不同的功能模块。

15、Url 链接筛查 key 对应的 value 值，没有就返回 undefined

```
function getUrlParam(name) {
        const reg = new RegExp("(^|&)" + name + "=([^&]*)(&|$)"); //构造一个含有目标参数的正则表达式对象
        const r = window.location.search.substr(1).match(reg); //匹配目标参数
        if (r != null) return unescape(r[2]);
        return undefined; //返回参数值
}
```

16、讲一下一个 HTTP 请求从发起请求到结束的完整过程

首先 http 是一个应用层的协议，在这个层的协议，只是一种通讯规范，也就是因为双方要进行通讯，大家要事先约定一个规范。

http 请求的基本过程是连接、请求、应答、关闭连接。

1.连接：当我们输入这样一个请求时,首先要建立一个 socket 连接,因为 socket 是通过 ip 和端口建立的，所以之前还有一个 DNS 解析过程，把 www.mycompany.com 变成 ip，如果 url 里不包含端口号，则会使用该协议的默认端口号。

DNS 的过程是这样的：首先我们知道我们本地的机器上在配置网络时都会填写 DNS，这样本机就会把这个 url 发给这个配置的 DNS 服务器，如果能够找到相应的 url 则返回其 ip，否则该 DNS 将继续将该解析请求发送给上级 DNS。

整个 DNS 可以看做是一个树状结构，该请求将一直发送到根直到得到结果。现在已经拥有了目标 ip 和端口号，这样我们就可以打开 socket 连接了。

2.请求： 连接成功建立后，开始向 web 服务器发送请求，这个请求一般是 GET 或 POST 命令（POST 用于 FORM 参数的传递）。GET 命令的格式为：GET 路径/文件名 HTTP/1.0。

文件名指出所访问的文件，HTTP/1.0 指出 Web 浏览器使用的 HTTP 版本。现在可以发送 GET 命令：GET /mydir/index.html HTTP/1.0。

3.应答 ：web 服务器收到这个请求，进行处理，从它的文档空间中搜索子目录 mydir 的文件 index.html。如果找到该文件，Web 服务器把该文件内容传送给相应的 Web 浏览器。

为了告知浏览器，Web 服务器首先传送一些 HTTP 头信息，然后传送具体内容（即 HTTP 体信息），HTTP 头信息和 HTTP 体信息之间用一个空行分开。

常用的 HTTP 头信息有：

① HTTP 1.0 200 OK： 这是 Web 服务器应答的第一行，列出服务器正在运行的 HTTP 版本号和应答代码，代码"200 OK"表示请求完成。

② MIME_Version:1.0： 它指示 MIME 类型的版本。

③ content_type:类型： 这个头信息非常重要，它指示 HTTP 体信息的 MIME 类型。如：content_type:text/html 指示传送的数据是 HTML 文档。

④ content_length:长度值： 它指示 HTTP 体信息的长度（字节）。

4.关闭连接：当应答结束后，Web 浏览器与 Web 服务器必须断开，以保证其它 Web 浏览器能够与 Web 服务器建立连接

17、小程序开发过程中提高性能的方法

代码层面

- 拆分组件
- 图片压缩
- 减少不必要数据
- 避免频繁 setData
- 使用 webView 组件开发

项目层面

- 拆分小程序
- 分包预加载
- 尽量升级版本库

18、node 框架层是怎么拿到 http 请求的

19、你是否使用过预编译的 CSS 语言，如 LESS 或者 SASS?你是如何编译他们的?

一：node.js 编译
1.sass
window 系统需先安装 Ruby，然后全局安装 sass：

```
<!-- 终端安装输入： -->
    gem install sass
    <!-- 即可在命令行中运行 sass： -->
    sass styles.scss styles.css
    <!-- 编译成生产环境下的 css 文件： -->
    sass styles.scss styles.css --style compressed
    <!-- 实时监视.scss 文件是及其方便的： -->
    sass --watch styles.scss:styles.css
```

2.less
与 sass 类似，第一步也需全局安装 less：

```
<!-- 终端安装输入： -->
    npm install -g less
```

```
    <!-- 即可在命令行中运行 less： -->
   lessc styles.less styles.css
    <!-- 编译成生产环境下的 css 文件： -->
   lessc -x styles.less styles.css
```

二：工具编译

1、做项目时常使用工具 - 考拉，操作方便，使用简单

2、使用 HBuilder 配置 Less 或者 Sass 自动编译

20、Node 同步和异步区别

* 同步方法调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为

* 异步方法调用一旦开始，方法调用就会立即返回，调用者就可以继续后续的操作。而异步方法通常会在另外一个线程中，整个过程，不会阻碍调用者的工作

21、不知设备宽度的九宫格怎么实现

1、FlexBox:使用 Flex 的一个好处是不用再担心高度塌陷的问题，而且还可以轻松实现子元素横向竖向甚至按比例伸缩扩展的布局。

HTML 结构如下：

```
<div class="square">
  <ul class="square-inner flex">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
    <li>6</li>
    <li>7</li>
    <li>8</li>
    <li>9</li>
  </ul>
</div>
```

抽取公共样式：

```
.square{
  position: relative;
  width: 100%;
  height: 0;
  padding-bottom: 100%; /* padding 百分比是相对父元素宽度计算的 */
  margin-bottom: 30px;
}
.square-inner{
```

```
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%; /* 铺满父元素容器，这时候宽高就始终相等了 */
}
.square-inner>li{
  width: calc(98% / 3);  /* calc 里面的运算符两边要空格 */
  height: calc(98% / 3);
  margin-right: 1%;
  margin-bottom: 1%;
  overflow: hidden;
}
.flex{
  display: flex;
  flex-wrap: wrap;
}
.flex>li{
  flex-grow: 1; /* 子元素按 1/n 的比例进行拉伸 */
  background-color: #4d839c;
  text-align: center;
  color: #fff;
  font-size: 50px;
  line-height: 2;
}
.flex>li:nth-of-type(3n){ /* 选择个数是 3 的倍数的元素 */
  margin-right: 0;
}
.flex>li:nth-of-type(n+7){  /* 选择倒数的三个元素，n 可以取 0 */
  margin-bottom: 0;
}
```

2、Grid：对于网格布局来说，grid 比 flex 更为方便，代码量更少，可以处理更为复杂的结构。

```
<div class="square">
  <div class="square-inner grid">
    <div>1</div>
    <div>2</div>
    <div>3</div>
    <div>4</div>
    <div>5</div>
    <div>6</div>
    <div>7</div>
    <div>8</div>
    <div>9</div>
```

```
  </div>
</div>
.grid{
  display: grid;
  grid-template-columns: repeat(3, 1fr); /* 相当于 1fr 1fr 1fr */
  grid-template-rows: repeat(3, 1fr); /* fr 单位可以将容器分为几等份 */
  grid-gap: 1%; /* grid-column-gap 和 grid-row-gap 的简写 */
  grid-auto-flow: row;
}
.grid>div{
  color: #fff;
  font-size: 50px;
  line-height: 2;
  text-align: center;
  background: linear-gradient(to bottom, #f5f6f6 0%,#dbdce2 21%,#b8bac6
49%,#dddfe3 80%,#f5f6f6 100%);
}
```

3、Float：

```
<div class="square">
  <ul class="square-inner float">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
    <li>6</li>
    <li>7</li>
    <li>8</li>
    <li>9</li>
  </ul>
</div>
.float::after{
  content: "";
  display: block;
  clear: both;
  visibility: hidden;
}
.float>li{
  float: left;
  background-color: #42a59f;
  text-align: center;
  color: #fff;
  font-size: 50px;
  line-height: 2;
```

```
}
.float>li:nth-of-type(3n){
  margin-right: 0;
}
.float>li:nth-of-type(n+7){
  margin-bottom: 0;
}
```

4、Table

```
<div class="square">
  <table class="square-inner table">
    <tbody>
      <tr>
        <td>1</td>
        <td>2</td>
        <td>3</td>
      </tr>
      <tr>
        <td>4</td>
        <td>5</td>
        <td>6</td>
      </tr>
      <tr>
        <td>7</td>
        <td>8</td>
        <td>9</td>
      </tr>
    </tbody>
  </table>
</div>
.table{
  border-collapse: separate;
  border-spacing: 0.57em;
  font-size: 14px;
  empty-cells: hide;
  table-layout: fixed;
}
.table>tbody>tr>td{
  text-align: center;
  background-color: #889ed8;
  overflow: hidden;
}
```

22、typeof 方法输出的值有哪些

'undefined'——未定义
'boolean'——布尔值
'string'——字符串
'number'——数值或 NaN
'object'——对象或 nullhuo 数组
'function'——函数
//typeof 无法识别 null、数组、NaN
//判断数据类型用 Object.prototype.toString.call()或 instanceOf
//实现 instanceOf：

```
function ins(left,right){
    let pro = left.__proto__
    while(true){
        if(pro === right.prototype){
            return true
        }
        if(pro === null){
            return false
        }
        pro = pro.__proto__
    }
}
```

23、react16 新生命周期

static getDerivedStateFromProps(nextProps, prevState)
这个函数会返回一个对象用来更新当前的 state 对象，如果不需要更新可以返回 null

24、手写实现数组扁平化

```
function bp(arr){
return arr.reduce((init,cur)=>{
return !Array.isArray(cur)?init.concat(cur):init.concat(bp(cur))
},[])
}
```

25、JS 强制类型转换和隐式转换有哪些，举例说明

parseInt() parseFloat() toString()
"1" + 1 "1" - 0

26、ES6 里面新增的数组和对象方法，用过哪些，举几个例子说明

```
Array.from()
Array.fill()
Array.includes()//ECMAscript 2016
Object.assign()
Object.keys()
Object.values()
```

27、JS 的 this 指向问题（结合笔试题，考察 this 指向）

this 永远指向最后调用它的对象

28、输出字符串中所有的叠词（几个连续出现的字组成叠词）？

```
var reg= /(.)\1+/ig;
var str_match= str.match(reg);
console.log(str_match);
```

29、股票利润最大值

假设有一个数组，它的第 i 个元素对应第 i 天的价格，如果最多只允许完成一次交易【即买进一次卖出一次】,设计一个算法找出利润最大利润。例 :[7,1,5,3,6,4]

最大利润：5

```
var arr=[7,1,5,3,6,4];
var min_index= arr.indexOf(Math.min.apply(Math,arr));
var min= Math.min.apply(null,arr);
var newArr= arr.slice(min_index);
var max= Math.max.apply(null,newArr);
var lirun= max-min;
console.log(lirun);
```

30、apply,call,bind

apply() 方法调用一个函数, 其具有一个指定的 this 值,以及作为一个数组(或类似数组的对象）提供的参数
apply 和 call 的区别是 call 方法接受的是若干个参数列表，而 apply 接收

的是一个包含多个参数的数组。
bind 绑定函数 this 返回绑定后的函数
都是改变 this 指向，第一个参数都是 this 的指向对象
bind 返回一个新的函数，调用时才会执行
call 和 apply 都是立即执行，apply 第二个参数是数组，代表一个个参数，call 是把参数依次作为第二个第三个参数

31、讲一下 JS 面向对象，原型链与原型继承基本原理

ECMAScript 有两种开发模式：1.函数式(过程化)，2.面向对象(OOP)。面向对象的语言有一个标志，那就是类的概念，而通过类可以创建任意多个具有相同属性和方法的对象。但是，ECMAScript 没有类的概念，因此它的对象也与基于类的语言中的对象有所不同。
js(如果没有作特殊说明,本文中的 js 仅包含 ES5 以内的内容)本身是没有 class 类型的，但是每个函数都有一个 prototype 属性。prototype 指向一个对象，当函数作为构造函数时，prototype 则起到类似 class 的作用。
一. 创建对象
创建一个对象，然后给这个对象新建属性和方法。
var box = new Object(); //创建一个 Object 对象 box.name = 'Lee'; //创建一个 name 属性并赋值 box.age = 100; //创建一个 age 属性并赋值 box.run = function () { //创建一个 run()方法并返回值 return this.name + this.age + '运行中...'; }; alert(box.run()); //输出属性和方法的值
上面创建了一个对象，并且创建属性和方法，在 run()方法里的 this，就是代表 box 对象本身。这种是 JavaScript 创建对象最基本的方法，但有个缺点，想创建多个类似的对象，就会产生大量的代码。
为了解决多个类似对象声明的问题，我们可以使用一种叫做工厂模式的方法，这种方法就是为了解决实例化对象产生大量重复的问题。
function createObject(name, age) { //集中实例化的函数 var obj = new Object(); obj.name = name; obj.age = age; obj.run = function () { return this.name + this.age + '运行中...'; }; return obj; } var box1 = createObject('Lee', 100); //第一个实例 var box2 = createObject('Jack', 200); //第二个实例 alert(box1.run()); alert(box2.run()); //保持独立
工厂模式解决了重复实例化的问题，但是它有许多问题，创建不同对象其中属性和方法都会重复建立，消耗内存；还有函数识别问题等等。
二. 构造函数的方法
构造函数的方法有一些规范：
1）函数名和实例化构造名相同且大写，(PS：非强制，但这么写有助于区分构造函数和
普通函数)；
2）通过构造函数创建对象，必须使用 new 运算符。
function Box(name, age) { //构造函数模式 this.name = name; this.age = age; this.run = function () { return this.name + this.age + '运行中...'; }; } var box1 = new Box('Lee', 100); //new Box()即可 var box2 = new Box('Jack',

200); alert(box1.run()); alert(box1 instanceof Box); //很清晰的识别他从属于 Box
构造函数可以创建对象执行的过程：
1）当使用了构造函数，并且 new 构造函数()，那么就后台执行了 new Object()；
2）将构造函数的作用域给新对象，(即 new Object()创建出的对象)，而函数体内的 this 就
代表 new Object()出来的对象。
3）执行构造函数内的代码；
4）返回新对象(后台直接返回)。
注：
1）构造函数和普通函数的唯一区别，就是他们调用的方式不同。只不过，构造函数也是函数，必须用 new 运算符来调用，否则就是普通函数。
2）this 就是代表当前作用域对象的引用。如果在全局范围 this 就代表 window 对象，如果在构造函数体内，就代表当前的构造函数所声明的对象。
这种方法解决了函数识别问题，但消耗内存问题没有解决。同时又带来了一个新的问题，全局中的 this 在对象调用的时候是 Box 本身，而当作普通函数调用的时候，this 又代表 window。即 this 作用域的问题。
三.原型
我们创建的每个函数都有一个 prototype(原型)属性，这个属性是一个对象，它的用途是包含可以由特定类型的所有实例共享的属性和方法。逻辑上可以这么理解：prototype 通过调用构造函数而创建的那个对象的原型对象。使用原型的好处可以让所有对象实例共享它所包含的属性和方法。也就是说，不必在构造函数中定义对象信息，而是可以直接将这些信息添加到原型中。
function Box() {} //声明一个构造函数 Box.prototype.name = 'Lee'; //在原型里添加属性 Box.prototype.age = 100; Box.prototype.run = function () { //在原型里添加方法 return this.name + this.age + '运行中...'; };

我们经常把属性（一些在实例化对象时属性值改变的），定义在构造函数内；把公用的方法添加在原型上面，也就是混合方式构造对象（构造方法+原型方式）：
var person = function(name){ this.name = name }; person.prototype.getName = function(){ return this.name; } var zjh = new person(‘zhangjiahao’); zjh.getName(); //zhangjiahao
下面详细介绍原型：
1.原型对象
每个 javascript 对象都有一个原型对象，这个对象在不同的解释器下的实现不同。比如在 firefox 下，每个对象都有一个隐藏的__proto__属性，这个属性就是“原型对象”的引用。
2.原型链
由于原型对象本身也是对象，根据上边的定义，它也有自己的原型，而它自己的原型对象又可以有自己的原型，这样就组成了一条链，这个就是原型链，
JavaScritp 引擎在访问对象的属性时，如果在对象本身中没有找到，则会去原型链中查找，如果找到，直接返回值，如果整个链都遍历且没有找到属性，则返回 undefined.原型链一般实现为一个链表，这样就可以按照一定的顺序来查找。

__proto__和 prototype
JS 在创建对象（不论是普通对象还是函数对象）的时候，都有一个叫做__proto__的内置属性，用于指向创建它的函数对象的原型对象 prototype。
constructor
原型对象 prototype 中都有个预定义的 constructor 属性，用来引用它的函数对象。这是一种循环引用
总结：
实例对象的__proto__指向，其构造函数的原型；构造函数原型的 constructor 指向对应的构造函数。构造函数的 prototype 获得构造函数的原型。
有时某种原因 constructor 指向有问题，可以通过
constructor：构造函数名；//constructor : Task
重新指向。
四.继承
继承是面向对象中一个比较核心的概念。其他正统面向对象语言都会用两种方式实现继承：一个是接口实现，一个是继承。而 ECMAScript 只支持继承，不支持接口实现，而实现继承的方式依靠原型链完成。
在 JavaScript 里，被继承的函数称为超类型(父类，基类也行，其他语言叫法)，继承的函数称为子类型(子类，派生类)
1.call+遍历
属性使用对象冒充（call）（实质上是改变了 this 指针的指向）继承基类，方法用遍历基类原型。可以实现多继承。
2.寄生组合继承
主要是 Desk.prototype = new Box(); Desk 继承了 Box，通过原型，形成链条。
主要通过临时中转函数和寄生函数实现。
临时中转函数：基于已有的对象创建新对象，同时还不必因此创建自定义类型
寄生函数：目的是为了封装创建对象的过程
临时中转函数和寄生函数主要做的工作流程：
临时中转函数：返回的是基类的实例对象函数
寄生函数：将返回的基类的实例对象函数的 constructor 指向派生类，派生类的 prototype 指向基类的实例对象函数（是一个函数原型），从而实现继承。

32、输出对称数

打印出 1-1000000 之间的所有对称数(例如 121、1331 等)

```
function judgeNum(start,end){
for(var i = start;i < end; i++){
var strI = i.toString().split("").reverse("").join("");
if(Number(strI) == i && i > 10){
console.log(strI);
}
}
}
judgeNum(1,1000000);
```

33、setTimeout（fuc,1000）是不是1000毫秒之后一定会执行？为什么？JS的定时器不准的原因是什么，结合此问题讲一下事件循环

不一定
js是单线程的，主线程之外有个任务队列用来放异步任务，定时器就是异步的操作，定时器设置的时间，是指在1000毫秒后，可以调用回调函数得到异步操作的结果，但是必须保证主线程中的同步任务已经全部执行完毕，如果主线程还有同步任务在执行，即使1000毫秒到了，也不会去读取异步的任务队列，就是说，这个定时器等待的最短时间是1000毫秒。

34、Promise和setTimeout的区别 [浅析setTimeout与Promise](https://www.jianshu.com/p/1486afd81594) [javascript的宏任务和微任务](https://blog.csdn.net/lc237423551/article/details/79902106)

setTimeout和Promise调用的都是异步任务，这一点是它们共同之处，也即都是通过任务队列进行管理/调度
JavaScript通过任务队列管理所有异步任务，而任务队列还可以细分为MacroTask Queue和MicoTask Queue两类。
MacroTask Queue
MacroTask Queue（宏任务队列）主要包括setTimeout,setInterval, setImmediate, requestAnimationFrame, NodeJS中的`I/O等。
MicroTask Queue
MicroTask Queue（微任务队列）主要包括两类：
独立回调microTask：如Promise，其成功/失败回调函数相互独立；
复合回调microTask：如 Object.observe, MutationObserver 和NodeJs中的process.nextTick ，不同状态回调在同一函数体；
MacroTask和MicroTask
JavaScript将异步任务分为MacroTask和MicroTask，那么它们区别何在呢？
依次执行同步代码直至执行完毕；
检查MacroTask 队列，若有触发的异步任务，则取第一个并调用其事件处理函数，然后跳至第三步，若没有需处理的异步任务，则直接跳至第三步；
检查MicroTask队列,然后执行所有已触发的异步任务,依次执行事件处理函数，直至执行完毕，然后跳至第二步，若没有需处理的异步任务中，则直接返回第二步，依次执行后续步骤；
最后返回第二步，继续检查MacroTask队列，依次执行后续步骤；
如此往复，若所有异步任务处理完成，则结束；

异步任务分为两类：微任务和宏任务，遇到宏任务，先执行宏任务
（宏任务队列）主要包括**setTimeout**,**setInterval**, **setImmediate**, **requestAnimationFrame**, **NodeJS 中的`I/O**等
（微任务队列）主要包括两类：
独立回调 microTask：如**Promise**，其成功 / 失败回调函数相互独立；
复合回调 microTask：如 **Object.observe**, **MutationObserver** 和 **NodeJs**中的 **proces

35、讲一下前端页面的渲染过程，有没有想过为什么渲染引擎会设计成 GUI 和 JS 是两个不同的线程且互斥，如果浏览器渲染引擎不是单线程会有什么问题

36、手写实现去除字符串连续重复值

/*数组去重*/
方法一、

```
function quchong(arr){
    var len = arr.length;
    arr.sort();
    for(var i=len-1;i>0;i--){
        if(arr[i]==arr[i-1]){
            arr.splice(i,1);
        }
    }
    return arr;
}
var a = ["a","a","b",'b','c','c','a','d'];
var b = quchong(a);
console.log(b);
```

方法二、

```
let arr = [12,43,23,43,68,12];let item = [...new Set(arr)];

console.log(item);//[12, 43, 23, 68]
```

/*字符串去重*/
方法一、

```
function quchongstr(str){
    var a = str.match(/\S+/g);//等价于 str.split(/\s+/g)// \s 空白符，\S 非空白符
    a.sort();
    for(var i=a.length-1;i>0;i--){
        if(a[i]==a[i-1]){
            a.splice(i,1);
```

```
            }
        }
        return a.join(" ");
    }
    var str = quchongstr("a a b a b e");
    console.log(str);
```

方法二：indexOf(无兼容问题)

```
function quchong2(str){
var newStr="";
for(var i=0;i<str.length;i++){
if(newStr.indexOf(str[i])==-1){
newStr+=str[i];
}
}
return newStr;
}
```

方法三：search()方法

```
function quchong3(str){
var newStr="";
for(var i=0;i<str.length;i++){
if(newStr.search(str[i])==-1)
newStr+=str[i];
}
return newStr;
}
```

方法四：对象属性

```
function quchong4(str){
var obj={};
var newStr="";
for(var i=0;i<str.length;i++){
if(!obj[str[i]]){
newStr+=str[i];
obj[str[i]]=1;
}
}
return newStr;
}
```

方法五：正则

```
var str="我我是是一个个帅帅帅帅哥！";
var reg = /(.)\1+/gi;
str = str.replace(reg,'$1');
console.log(str);//我是一个帅哥！
```

37、三个数组：arr1=[1,2,3,4],arr2=[4,5,6],arr3=[5],将 arr1 与

arr2 合并，去重，然后返回不包含 arr3 中元素的新数组（用 es6 标准的新特性实现）

```
var arr1=[1,2,3,4];
var arr2= [4,5,6];
var arr3= [5];
// 并集 数组去重
let RemoveSame=[...new Set([...arr1,...arr2])]
console.log(RemoveSame) //[1, 2, 3, 4, 5, 6]
//数组交集，或得两个数组重复的元素
let SamePart=arr1.filter(item=>arr2.includes(item))
console.log(SamePart) //[4]
//差集=并集-交集  去除两个数组相同的元素
let Difference=RemoveSame.filter(item=>!arr3.includes(item))
console.log(Difference)//[1, 2, 3, 6]
```

38、new 运算符的执行过程，实现一个 new

```
//1.创建一个新对象
//2.将构造函数的作用域赋给新对象
//3.执行构造函数中的代码
//4.返回新对象
function new(con,...args){
console.log(args)
let obj = {}
obj.proto  = con.prototype
console.log(obj)
let result = con.apply(obj,args)
console.log(result)
return result
}
```

39、手写深拷贝

```
function deepCopy(target){
    let copyed_objs = []//存放遍历过的目标属性
    function _deepCopy(target){
        if((typeof target !== 'object') || !target){
            return target //非对象类型直接返回值
```

```
        }
        for(let i =0; i<copyed_objs.length;i++){
            if(copyed_objs[i].target === target){
                return copyed_objs[i].copyTarget
            }
        }//每次递归时判断要复制的目标属性是否已经缓存，已缓存直接返回
        let obj = {}
        if(Array.isArray(target)){
            obj = []
        }
        copyed_objs.push({
            target : target,
            copyTarget : obj
        })//将递归过的对象保存
        Object.keys(target).forEach(key=>{
            if(obj[key]){
                return
            }
            obj[key] = _deepCopy(target[key])
        })//遍历复制目标属性对应对象的每个值
        return obj
        //返回复制结果
    }
    return _deepCopy(target)
}
```

40、http 缓存

cache-control:
max-age=xxx: 缓存的内容将在 xxx 秒后失效
no-cache: 需要使用对比缓存来验证缓存数据（后面介绍）
no-store: 所有内容都不会缓存，强制缓存，对比缓存都不会触发

41、实现一个 div 相对于父元素或者屏幕垂直水平居中，多说几种方法

相对于父元素：
将父元素 position 属性设置成 relative，目标元素设为 absolute
一、将目标元素的 top、left、right、bottom 设为相同的值，margin：auto
二、将目标元素 top、left 设为 50%，transform:translate(-50%,-50%)将目标框体向左(上)平移自己宽度(高度)的 50%.
三、已知宽高的子元素
父元素启用弹性盒布局

```
 /* 启用弹性盒布局 */
 display: flex;
/* 使子元素水平居中 */
justify-content: center;
/* 使子元素垂直居中 */
align-items: center;
```

相对屏幕：
一、将目标元素 position 属性设为 fixed，top、left、right、bottom 设为相同的值，margin：auto

42、用 flex 实现左右两边 50%，其中右边是一个田字格

43、讲一下 redux 原理

1、为什么要用 redux
在 React 中,数据在组件中是单向流动的,数据从一个方向父组件流向子组件(通过 props),所以，两个非父子组件之间通信就相对麻烦，redux 的出现就是为了解决 state 里面的数据问题
2、Redux 设计理念
Redux 是将整个应用状态存储到一个地方上称为 store,里面保存着一个状态树 store tree,组件可以派发(dispatch)行为(action)给 store,而不是直接通知其他组件，组件内部通过订阅 store 中的状态 state 来刷新自己的视图。
3、Redux 三大原则
1 唯一数据源
2 保持只读状态
3 数据改变只能通过纯函数来执行
1 唯一数据源
整个应用的 state 都被存储到一个状态树里面，并且这个状态树，只存在于唯一的 store 中
2 保持只读状态
state 是只读的，唯一改变 state 的方法就是触发 action，action 是一个用于

描述以发生时间的普通对象
3 数据改变只能通过纯函数来执行
使用纯函数来执行修改，为了描述 action 如何改变 state 的，你需要编写 reducers
或许你读到这已经不知所云了，没事这只是让你了解一些 redux 到底是干嘛的，后面或详细的讲解各个部分的作用，并且会讲解 redux 实现原理
4、Redux 概念解析
4.1 Store
store 就是保存数据的地方，你可以把它看成一个数据，整个应用只能有一个 store
Redux 提供 createStore 这个函数，用来生成 Store

```
import {createStore} from 'redux'
const store=createStore(fn);
```

4.2 State
state 就是 store 里面存储的数据，store 里面可以拥有多个 state，Redux 规定一个 state 对应一个 View,只要 state 相同，view 就是一样的，反过来也是一样的，可以通过 store.getState( )获取

```
import {createStore} from 'redux'
const store=createStore(fn);
const state=store.getState()
```

4.3 Action
state 的改变会导致 View 的变化，但是在 redux 中不能直接操作 state 也就是说不能使用 this.setState 来操作，用户只能接触到 View。在 Redux 中提供了一个对象来告诉 Store 需要改变 state。Action 是一个对象其中 type 属性是必须的，表示 Action 的名称，其他的可以根据需求自由设置。

```
const action={
  type:'ADD_TODO',
  payload:'redux原理'
}
```

在上面代码中,Action 的名称是 ADD_TODO,携带的数据是字符串‘redux 原理’，Action 描述当前发生的事情，这是改变 state 的唯一的方式
4.4 store.dispatch( )
store.dispatch( )是 view 发出 Action 的唯一办法

```
store.dispatch({
  type:'ADD_TODO',
  payload:'redux原理'
})
```

store.dispatch 接收一个 Action 作为参数，将它发送给 store 通知 store 来改变 state。

4.5 Reducer

Store 收到 Action 以后，必须给出一个新的 state，这样 view 才会发生变化。这种 state 的计算过程就叫做 Reducer。

Reducer 是一个纯函数，他接收 Action 和当前 state 作为参数，返回一个新的 state

注意：Reducer 必须是一个纯函数，也就是说函数返回的结果必须由参数 state 和 action 决定，而且不产生任何副作用也不能修改 state 和 action 对象

```
const reducer =(state,action)=>{
  switch(action.type){
    case ADD_TODO:
        return newstate;
    default return state
  }
}
```

5、Redux 源码

里面的注释就是我一步一步的分析，有点懒没有把代码拆分出来给你们看

```
let createStore = (reducer) => {
    let state;
    //获取状态对象
    //存放所有的监听函数
    let listeners = [];
    let getState = () => state;
    //提供一个方法供外部调用派发action
    let dispath = (action) => {
        //调用管理员reducer得到新的state
        state = reducer(state, action);
        //执行所有的监听函数
        listeners.forEach((l) => l())
    }
    //订阅状态变化事件，当状态改变发生之后执行监听函数
    let subscribe = (listener) => {
        listeners.push(listener);
    }
    dispath();
    return {
        getState,
        dispath,
        subscribe
    }
}
```

```
let combineReducers=(renducers)=>{
    //传入一个renducers管理组，返回的是一个renducer
    return function(state={},action={}){
        let newState={};
        for(var attr in renducers){
            newState[attr]=renducers[attr](state[attr],action)

        }
        return newState;
    }
}
export {createStore,combineReducers};
```

6、Redux 使用案例

html 代码

```
<div id="counter"></div>
  <button id="addBtn">+</button>
  <button id="minusBtn">-</button>
```

js 代码

```
function createStore(reducer) {
    var state;
    var listeners = [];
    var getState = () => state;
    var dispatch = (action) => {
        state = reducer(state, action);
        listeners.forEach(l=>l());
    }
    var subscribe = (listener) => {
        listeners.push(listener);
        return () => {
            listeners = listeners.filter((l) => l != listener)
        }
    }
    dispatch();
    return {
        getState, dispatch, subscribe
    }}
    var reducer = (state = 0, action) => {
    if (!action) return state;
    console.log(action);
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        case 'DECREMENT':
```

```
            return state - 1;
        default:
            return state;
    }}
    var store = createStore(reducer);
store.subscribe(function () {
    document.querySelector('#counter').innerHTML =
store.getState();});

document.querySelector('#addBtn').addEventListener('click', function ()
{
    store.dispatch({type: 'INCREMENT'});});
document.querySelector('#minusBtn').addEventListener('click',
function () {
    store.dispatch({type: 'DECREMENT'});});
```

44、react 父子组件之间如何通信，不相干组件如何通信

首先要知道 React 的组件间通讯是单向的。数组必须是由父级传到子级或者子级传递给父级层层传递。
如果要给兄弟级的组件传递数据，那么就要先传递给父级而后在传递给你要传递到的组件位置。

父级组件向子级组件传递数据
在 React 中，父组件可以向子组件通过传 props 的方式，向子组件进行通讯。

```
import React, { Component } from 'react';
import './App.css';
import Child from './child'
export default class App extends Component {
   constructor(props){
        super(props);
        this.state={
            msg:'父类的消息',
            name:'John',
            age:99
        }
   }
    callback=(msg,name,age)=>{
        // setState 方法,修改 msg 的值,值是由 child 里面传过来的
        this.setState({msg});
        this.setState({name});
        this.setState({age});
    }
```

```
    render() {
     return (
       <div className="App">
         <p> Message:   {this.state.msg}</p>
         <Child callback={this.callback} age={this.state.age}
name={this.state.name}></Child>
       </div>
     );
    }
}
```

父组件中，state 里面有三个属性，分别是 msg，name 和 age，在子组件 child 中，如果想拿到父组件里面的属性，就需要通过 props 传递。

在标签里面添加

name = {this.state.name} age = {this.state.age} 写成 <Child name={this.state.name} age={this.state.age}>

name 和 age 分别是你要传递的属性。

子组件 Child

```
import React from "react";

export default class Child extends React.Component{
    constructor(props){
         super(props);
         this.state={
             name:'Andy',
             age:31,
             msg:"来自子类的消息"
         }
     }
    change=()=>{

    this.props.callback(this.state.msg,this.state.name,this.state.age)
;
    }
     render(){
         return(
             <div>
                 <div>{this.props.name}</div>
                 <div>{this.props.age}</div>
                 <button onClick={this.change}>点击</button>
             </div>
         )
     }
}
```

在子组件中,通过 {this.props.name} {this.props.age}就能拿到父组件里面

的数据。
子组件向父组件通信
子组件向父组件通信，同样也需要父组件向子组件传递 props 进行通讯，只是父组件传递的是作用域为父组件自身的函数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中。
上面的例子中，在子组件 Child 中绑定了 onClick 事件。调用 this.change 方法。注意 change 函数采用了箭头函数的写法 change=() => {}，目的是为了改变 this 的指向。使得在函数单独调用的时候，函数内部的 this 依然指向 child 组件。如果不使用箭头函数，而是采用普通的写法则需要在 constructor 中 bind 一下。
this.change = this.change.bind(this)
或者在 onClick 方法中绑定 this，
onClick={this.change=this.change.bind(this)}
在 change 方法中，通过 props 发送出去一个方法，比如说叫 callback 方法，父组件中去接收这个方法，callback={this.callback}，然后在自身的 callback 函数中进行一系列操作。
本例中，函数 callback 中就是通过调用 setState 方法来改变值。
点击按钮后页面显示：
现在我们就已经将父子组件间的通讯问题给解决了，但是如果是同级别的组件间传递如果还是采用 props 传给父级组件然后再传给子组件这就比较麻烦了，尤其是当项目越来越大的时候。所以这时候我们就可以使用全局的状态管理了 Redux

45、react-redux 是如何把 store 绑定到组件上的，底层实现

46、小程序里面 canvas map 等组件优先级最高，无法被覆盖，如果要在其上加按钮，例如在地图上加按钮，如何做

cover-view 和 cover-image

47、RN 里面 native 和 JS 是如何通信的

一：通过 Callbacks 的方式
二：通过 Promises 的方式
三：通过发送事件的方式

方式	缺点	优点
通过Callbacks的方式	只能传递一次	传递可控，JS模块调用一次，原生模块传递一次
通过Promises的方式	只能传递一次	传递可控，JS模块调用一次，原生模块传递一次
通过发送事件的方式	原生模块主动传递，JS模块被动接收	可多次传递

48、手写实现 Promise.all

49、手写实现 filter

50、讲一下 express 或者 koa 的原理

51、实现继承的方法

js 中继承跟 java 中的继承不太一样，一般通过 call()和 apply()两种方式完成，js 中的继承是以复制的形式完成的，复制一个父对象，而不像 java 中直接继承父对象，还有通过原型的方式完成继承，也有弊端，总之 js 中的继承只是形式上的对面向对象语言的一种模仿，本质上不是继承，但用起来效果是一样的
至于为什么要继承：通常在一般的项目里不需要，因为应用简单，但你要用纯 js 做一些复杂的工具或框架系统就要用到了，比如 webgis、或者 js 框架如 jquery、ext 什么的，不然一个几千行代码的框架不用继承得写几万行，甚至还无法维护
1、原型链继承
2、构造继承
3、组合继承
4、寄生组合继承
5、原型式继承

52、事件循环机制

同步任务直接在主线程中执行。
异步任务进入事件队列中，主线程内的任务执行完毕为空，会去实践队列读取任务，堆入主线程执行，不断循环就是事件循环

53、char 和 varchar：

char 不可变，varchar 可变

54、rem 的原理

rem 是 css3 一种新的长度单位。在 W3C 中有这样的定义：REM（Font size of the root element）是指相对于 html 根元素的字体大小的单位（注意这里泛指是相对于 html 中的根元素标签也就是 html）。一般用于在移动端中解决多种机型适配问题。
在移动端开发时候，开发人员为了解决不同的机型适配问题。常用 mediaQue(媒体查询)依据每种机型写多套适配代码。这种方法固然能解决适配问题。代码量太多了，不美观，且不利于维护。
用 rem 可以解决上述的问题。rem 的核心在于通过动态设置根元素 html 的 font-size。我们知道浏览器的默认的 font-size 是 16px,(有些浏览器例外)。依据 W3C 中的描述相对于 html 中的根元素标签。也就是 1rem = 16px,　16px = 1rem。
代码如下:
1.首先先获取设备屏幕的宽度,以下采用兼容的写法

```
let htmlWidth= document.documentElement.clientWidth || document.body.clientWidth
console.log(htmlWidth)
```

2.通过标签选择器获取根元素 html 标签

```
let htmlDom = document.getElementsByTagName('html')[0]
```

3.用设备的宽度除 10,然后把得到的数值赋值给根元素的 font-size;
这里的除数选择多少都可以,不一定是要 10,这里除 10 的目的在于:假设设备是 iphone6,那宽度就是 375,最后得到根元素的 font-size 就是 37.5px,如果除数越大的话，那最后得到的像素就越小也就是 font-size 越小，而对于浏览器来说，它无法识别那么小的 font-size。所以这里的除数，应该选择一个合适的数值。

```
htmlDom.style.fontSize = htmlWidth / 10 + 'px';
```

4.得到的 htmlDom(根元素)是 37.5px 也就是 1rem ，接下来就好办多了,当你编码的时候,你就可以依据设计稿中标注的尺寸(px) 例如:你设置某个元素的高度,假设是 120px (移动端中设计稿标注的尺寸一般都是乘以 2 倍)，所以这里要除 2,也就是 60px，然后用
60px / 根元素的 font-size 37.5px　=　?rem (这里的问号即是除完后的数值,单位是 rem)
5.如果每次都要手动去除根元素的 font-size 很麻烦，这里可以运用 sass,less 定义一个方法来工作。代码如下

```
@function px2rem($px){
  $rem : 37.5px;　定义一个变量初始值为 37.5。也就是我们之前根据设备宽度 375 px　/ 10 px 得到的根元素的 font-size
  @return ($px / $rem ) + rem;　接下来每次调用这个方法传入的 px 自动除 37.5 最后返回给调　用者
  }
```

例如:

```
.fatherBox {
width: px2rem(160px); 函数执行完之后会自动以 rem 来作为单位返回
height: px2rem(160px);
}
```

55、对一张图片色彩要求比较高选什么格式？

Png：PNG 图片在下载过程中占带宽较小，而且颜色逼真，下载一次可重复使用

56、react 里 setState 是同步还会异步

由 React 控制的事件处理程序，以及生命周期函数调用 setState 不会同步更新 state 。
React 控制之外的事件中调用 setState 是同步更新的。比如原生 js 绑定的事件，setTimeout/setInterval 等。
大部分开发中用到的都是 React 封装的事件，比如 onChange、onClick、onTouchMove 等，这些事件处理程序中的 setState 都是异步处理的。

57、定位描述

固定定位:始终相对于浏览器窗口进行定位
相对定位:用来对标签的位置进行微调,参照的是标签原来的位置
绝对定位:元素的位置相对于最近的已定位祖先元素(relative 对象),如果元素没有已定位的祖先元素,那么它的位置相对于最初的包含块----body

58、钩子函数

一般认为，钩子函数就是回调函数的一种，其实还是有差异的，差异地方就是：触发的时机不同。
先说钩子函数：
钩子（Hook）概念源于 Windows 的消息处理机制，通过设置钩子，应用程序对所有消息事件进行拦截，然后执行钩子函数。

```
let btn = document.getElementById("btn");
btn.onclick = () => {
    console.log("i'm a hook");
}
```

上面的例子，在按钮点击时候立即执行钩子函数。而看下面的例子：

```
btn.addEventListener("click",() =>{
    console.log(this.onclick);//undefined
});
```

给 btn 绑定了一个监听器，只有消息捕获完成之后才能触发回调函数。
很明显的差别就是：钩子函数在捕获消息的第一时间就执行，而回调函数是捕获结束时，最后一个被执行的。
回调函数其实是调用者将回调函数的指针传递给了调用函数，当调用函数执行完

毕后，通过函数指针来调用回调函数。而钩子函数在消息刚发出，没到达目的窗口前就先捕获了该消息，先得到控制权执行钩子函数，所以他可以加工改变该消息，当然也可以不作为，还可以强行结束该消息。

59、sessionStorage、localStorage、cookie 的区别

sessionStorage 与 localStorage 以及 Cookie 共同点：
都是保存在浏览器端、且同源的
区别在于：
1、数据有效期不同，sessionStorage：仅在当前浏览器窗口关闭之前有效;localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据;cookie：只在设置的 cookie 过期时间之前有效，即使窗口关闭或浏览器关闭
2、cookie 数据始终在同源的 http 请求中携带(即使不需要)，即 cookie 在浏览器和服务器间来回传递，而 sessionStorage 和 localStorage 不会自动把数据发送给服务器，仅在本地保存。cookie 数据还有路径(path)的概念，可以限制 cookie 只属于某个路径下
3、存储大小限制也不同，cookie 数据不能超过 4K，同时因为每次 http 请求都会携带 cookie、所以 cookie 只适合保存很小的数据，如会话标识。
sessionStorage 和 localStorage 虽然也有存储大小的限制，但比 cookie 大得多，可以达到 5M 或更大
4、作用域不同，sessionStorage 不在不同的浏览器窗口中共享，即使是同一个页面;localstorage 在所有同源窗口中都是共享的;cookie 也是在所有同源窗口中都是共享的
5、web Storage(sessionStorage,localstorage)支持事件通知机制，可以将数据更新的通知发送给监听者
6、web Storage 的 api 接口使用更方便
sessionStorage 与 localStorage 的用法一致：
window.sessionStorage.clear(all);
window.sessionStorage.removeItem(key);
window.sessionStorage.setItem(key, value);
window.sessionStorage.getItem(key)
cookie 用法：
$.cookie(key);
$.cookie(key,value);

60、Css 优先级

a、用 a 表示选择器中 ID 选择器出现的次数
b、用 b 表示类选择器，属性选择器和伪类选择器出现的总次数。
c、用 c 表示标签选择器、伪元素选择器出现的总次数
d、忽略通用选择器
e、然后计算 a*100+b*10+c 的大小，这就是优先级了。
权重：内联样式 1000》id 选择器 100》class 选择器 10》标签选择器 1

Note:
ID 选择器「如：#header」，Class 选择器「如：.foo」，属性选择器「如：[class]」，伪类「如：:link」，标签选择器「如：h1」，伪元素「如：:after」，选择器「*」
!important > 行内样式 > id 选择器 > 类选择器 > 标签选择器 > 通配符 > 浏览器默认样式 > 继承

61、弹性布局

1 弹性布局简介

弹性布局，又称“Flex 布局”，是由 W3C 老大哥于 2009 年推出的一种布局方式。可以简便、完整、响应式地实现各种页面布局。而且已经得到所有主流浏览器的支持，我们可以放心大胆的使用。

>>> 了解两个基本概念，接下来会频繁提到：

① 容器： 需要添加弹性布局的父元素；

② 项目： 弹性布局容器中的每一个子元素，称为项目；

>>> 了解两个基本方向，这个牵扯到弹性布局的使用：

① 主轴： 在弹性布局中，我们会通过属性规定水平/垂直方向为主轴；

② 交叉轴： 与主轴垂直的另一方向，称为交叉轴。

2 弹性布局的使用

① 给父容器添加 display: flex/inline-flex;属性，即可使容器内容采用弹性布局显示，而不遵循常规文档流的显示方式；

② 容器添加弹性布局后，仅仅是容器内容采用弹性布局，而容器自身在文档流中的定位方式依然遵循常规文档流；

③ display:flex; 容器添加弹性布局后，显示为块级元素；

display:inline-flex; 容器添加弹性布局后，显示为行级元素；

④ 设为 Flex 布局后，子元素的 float、clear 和 vertical-align 属性将失效。但是 position 属性，依然生效。

3 作为父容器的 6 大属性

① flex-direction 属性决定主轴的方向（即项目的排列方向）。

row（默认值）： 主轴为水平方向，起点在左端；

row-reverse： 主轴在水平方向，起点在右端 ；

column：主轴为垂直方向，起点在上沿。

column-reverse：主轴为垂直方向，起点在下沿。

② flex-wrap 属性定义，如果一条轴线排不下，如何换行。

nowrap（默认）：不换行。当容器宽度不够时，每个项目会被挤压宽度；

wrap： 换行，并且第一行在容器最上方；

wrap-reverse： 换行，并且第一行在容器最下方。

③ flex-flow 是 flex-direction 和 flex-wrap 的缩写形式，默认值为：flex-flow: row wrap; 不做过多解释

④ justify-content 属性定义了项目在主轴上的对齐方式。

>>> 此属性与主轴方向息息相关。

主轴方向为：row-起点在左边，row-reverse-起点在右边， column-起点在上边，column-reverse-起点在下边

flex-start（默认值）： 项目位于主轴起点。

flex-end：项目位于主轴终点。

center： 居中

space-between：两端对齐，项目之间的间隔都相等。(开头和最后的项目，与父容器边缘没有间隔)

space-around：每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。(开头和最后的项目，与父容器边缘有一定的间隔)

⑤ align-items 属性定义项目在交叉轴上如何对齐。

flex-start：交叉轴的起点对齐。

flex-end：交叉轴的终点对齐。

center：交叉轴的中点对齐。

baseline: 项目的第一行文字的基线对齐。(文字的行高、字体大小会影响每行的基线)

stretch（默认值）：如果项目未设置高度或设为 auto，将占满整个容器的高度。

⑥ align-content 属性定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用。

(当项目换为多行时，可使用 align-content 取代 align-items)

flex-start：与交叉轴的起点对齐。

flex-end：与交叉轴的终点对齐。

center：与交叉轴的中点对齐。

space-between：与交叉轴两端对齐，轴线之间的间隔平均分布。

space-around：每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍。

stretch（默认值）：轴线占满整个交叉轴。

4 Android 作用于子项目的 6 大属性

① order 属性定义项目的排列顺序。数值越小，排列越靠前，默认为 0。

② flex-grow 属性定义项目的放大比例，默认为 0，即如果存在剩余空间，也不放大。

③ flex-shrink 属性定义了项目的缩小比例，默认为 1，即如果空间不足，该项目将缩小。

④ flex-basis 定义项目占据的主轴空间。(如果主轴为水平，则设置这个属性，相当于设置项目的宽度。 原 width 将会失效。)

⑤ flex 属性是 flex-grow, flex-shrink 和 flex-basis 的简写，默认值为 0 1 auto。后两个属性可选。

此属性有两个快捷设置：auto=(1 1 auto)/none=(0 0 auto)

⑥ align-self:定义单个项目自身在交叉轴上的排列方式，可以覆盖掉容器上的 align-items 属性。

属性值：与 align-items 相同，默认值为 auto，表示继承父容器的 align-items 属性值。

62、背景图充满屏幕

background-size:100% 100%;

63、Css 盒子模型和 IE 的区别

盒子模型的概念说法有很多，个人的简单理解就是——即具备内容、填充、边框、边界这些属性的均可以看作盒子模型，无论是不是块级元素！
那么，主要的两种盒子模型：标准 CSS 盒子模型与 IE 盒子模型的区别是什么？

（盗用两张图）
标准盒子模型：元素的 width 或 height=content 的 width 或 height；
IE 盒子模型：元素的 width 或 height=content 的 width 或 height+padding*2+border*2；

64、Css 动画 animation

65、什么是跨域

“跨域:指的是浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的,是浏览器对 javascript 施加的安全限制。”
例如：a 页面想获取 b 页面资源，如果 a、b 页面的协议、域名、端口、子域名不同，所进行的访问行动都是跨域的，而浏览器为了安全问题一般都限制了跨域访问，也就是不允许跨域请求资源。注意：跨域限制访问，其实是浏览器的限制。理解这一点很重要！！！
同源策略：是指协议，域名，端口都要相同，其中有一个不同都会产生跨域；

66、前端跨域解决方案

1、通过 jsonp 跨域
2、 document.domain + iframe 跨域
3、 location.hash + iframe
4、 window.name + iframe 跨域
5、 postMessage 跨域
6、 跨域资源共享（CORS）
7、 nginx 代理跨域
8、 nodejs 中间件代理跨域
9、 WebSocket 协议跨域
https://segmentfault.com/a/1190000011145364

67、Let 和 var 和 const 的区别

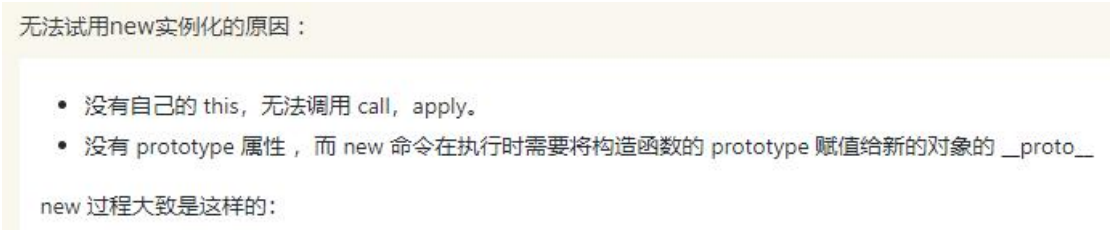
在 javascript 中有三种声明变量的方式：var、let、const。
var 声明变量可以重复声明，而 let 不可以重复声明
var 是不受限于块级的，而 let 是受限于块级
var 会与 window 相映射（会挂一个属性），而 let 不与 window 相映射
var 可以在声明的上面访问变量，而 let 有暂存死区，在声明的上面访问变量会报错

const 声明之后必须赋值，否则会报错
const 定义不可变的量，改变了就会报错
const 和 let 一样不会与 window 相映射、支持块级作用域、在声明的上面访问变量会报错

68、箭头函数和普通函数的区别

1.this 指向不同
1.1 普通函数 this 指向为方法调用的对象，可以通过 bind，call，apply 改变 this 的指向，
将当前函数与指定的对象绑定，并返回一个新函数，这个新函数无论以什么样的方式调用，
其 this 始终指向绑定的对象 bind 和 call，apply 调用方式不同，call 和 apply 传参方式不同
1.2 箭头函数比函数表达式更简洁，箭头函数不会创建自己的 this,它只会从自己的作用域链的上一层继承 this
bind，call，apply 只能调用传递参数，不可修改 this 指向

2.arguments 对象是所有（非箭头）函数中都可用的局部变量
3.箭头函数不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数。
4.箭头函数不能使用 new 命令，且没有 prototype 属性



69、Typescript

TypeScript 是 JavaScript 的一个超集，支持 ECMAScript 6 标准。
TypeScript 由微软开发的自由和开源的编程语言。
TypeScript 设计目标是开发大型应用，它可以编译成纯 JavaScript，编译出来的 JavaScript 可以运行在任何浏览器上。
TypeScript 是一种给 JavaScript 添加特性的语言扩展。增加的功能包括：
类型批注和编译时类型检查
类型推断
类型擦除
接口
枚举
Mixin
泛型编程
名字空间

元组
Await
以下功能是从 ECMA 2015 反向移植而来：
类
模块
lambda 函数的箭头语法
可选参数以及默认参数
JavaScript 与 TypeScript 的区别
TypeScript 是 JavaScript 的超集，扩展了 JavaScript 的语法，因此现有的 JavaScript 代码可与 TypeScript 一起工作无需任何修改,TypeScript 通过类型注解提供编译时的静态类型检查。
TypeScript 可处理已有的 JavaScript 代码，并只对其中的 TypeScript 代码进行编译。

70、border 设为 1px 显示比 1px 要宽怎么解决

由于高清屏的特性，1px 是由 2×2 个像素点来渲染，border:1px 在 Retina 屏下会渲染成 2px 的边框
实现方式
1.图片
2.阴影
3.缩放
4.dpi：umi 是用 dpi 做的 rem 布局 动态修改 initial-scale=1

71、提升网页安全性

选择安全稳定的服务器
修改后台初始密码
使用 HTTPS 协议
使用 CDN 加速

72、数组的深度

数组深度指的是数组嵌套的深度

```
var  ary  =  [9,'kk',['kk',9],['uuu','hhh',['kkk',89]]]
function  fo(arr,len)  {
    var  flag  =  false
    var  arr1  =  []
    for  (let  i  =  0;  i<  arr.length;  i++)  {
        let  isAry  =  Object.prototype.toString.call(arr[i])  ==  '[
object  Array]'
        if  (isAry)  {
```

```
            for(let  j  =  0;  j<  arr[i].length;  j  ++)  {
                arr1.push(arr[i][j])
            }
            flag  =  true
        }
    }
    if  (flag)  {
        len  ++
        return  fo(arr1,len)
    }else  {
        return  len
    }
}
let  lens  =  fo(ary,1)
console.log(lens)
```

73、setImmediate() node

node.js 中的非 IO 的异步 API 提供了四种方法，分别为 setTimeOut(),setInterval(),setImmediate()以及 process.nextTick()，四种方法实现原理相似，但达到的效果略有区别。
这里面 setTimeOut()与 setInterval()除了执行频次外基本相同，都表示主线程执行完一定时间后立即执行，而 setImmediate()与之十分相似，也表示主线程执行完成后立即执行。那么他们之间的区别是什么呢？
两者都代表主线程完成后立即执行，其执行结果是不确定的，可能是 setTimeout 回调函数执行结果在前，也可能是 setImmediate 回调函数执行结果在前，但 setTimeout 回调函数执行结果在前的概率更大些，这是因为他们采用的观察者不同，setTimeout 采用的是类似 IO 观察者，setImmediate 采用的是 check 观察者，而 process.nextTick()采用的是 idle 观察者。
结论：
process.nextTick()，效率最高，消费资源小，但会阻塞 CPU 的后续调用；
setTimeout()，精确度不高，可能有延迟执行的情况发生，且因为动用了红黑树，所以消耗资源大；
setImmediate()，消耗的资源小，也不会造成阻塞，但效率也是最低的。
setImmediate()方法
setImmediate()方法用于中断长时间运行的操作，并在浏览器完成其他操作（如事件和显示更新）后立即运行回调函数。
注意：此功能是非标准的，不要在面向 Web 的生产站点上使用它：它不适用于每个用户。实现之间可能存在很大的不兼容性，并且行为可能在将来发生变化。
setImmediate()方法语法
var immediateID = setImmediate（func，[param1，param2，...]）;var immediateID = setImmediate（func）;
其中 immediateID 是是立即标记的 ID，稍后可以与 window.clearImmediate 一

起使用。
func 是你想要调用的函数能。
所有参数都将直接传递给您的函数。

74、事件轮询

事件轮询（Event Loop）是一个很重要的概念，指的是计算机系统的一种运行机制。JavaScript 语言就是采用的这种机制，来解决单线程运行带来的一些问题。
想要理解 EventLoop，就要从程序的运行模式讲起。运行以后的程序叫做"进程"（process），一般情况下，一个进程一次只能执行一个任务。
如果有很多任务需要执行，不外乎三种解决方法。
（1）排队。因为一个进程一次只能执行一个任务，只好等前面的任务执行完了，再执行后面的任务。
（2）新建进程。使用 fork 命令，为每个任务新建一个进程。
（3）新建线程。因为进程太耗费资源，所以如今的程序往往允许一个进程包含多个线程，由线程去完成任务。
以 JavaScript 语言为例，它是一种单线程语言，所有任务都在一个线程下完成，即采用上面的第一种方法，一旦遇到大量任务或者遇到一个耗时的任务，网页就会出现假死的状态，因为 JavaScript 停不下来，也就无法响应用户的行为。
你也许会问，JavaScript 为什么是单线程，难道不能实现为多线程吗？
这跟历史有关系。JavaScript 从诞生起就是单线程。原因大概是不想让浏览器变得太复杂，因为多线程需要共享资源、且有可能修改彼此的运行结果，对于一种网页脚本语言来说，这就太复杂了。后来就约定俗成，JavaScript 为一种单线程语言。（Worker API 可以实现多线程，但是 JavaScript 本身始终是单线程的。）
多线程不仅占用多倍的系统资源，也闲置多倍的资源，这显然不合理。
Event Loop 就是为了解决这个问题而提出的。Wikipedia 这样定义：
"EventLoop 是一个程序结构，用于等待和发送消息和事件。（aprogramming construct that waits for and dispatches events or messages in aprogram.）"
简单说，就是在程序中设置两个线程：一个负责程序本身的运行，称为"主线程"；另一个负责主线程与其他进程(主要是各种 I/O 操作)的通信，被称为"EventLoop 线程"（可以译为"消息线程"）。

75、弹性布局使用场景

多列盒子浮动时可以采用弹性盒

76、new 发生了什么

1、创建一个新对象
2、将构造函数的作用域赋值给新对象（this 指向这个新对象）

3、执行构造函数中的代码（为这个新对象添加属性）
4、返回新对象
一个普通的构造函数

```
function Person(name){
  this.name = name}var p = new Person("小明");

console.log(p.name) // 小明
console.log(p instanceof Person) // true
```

用 JS 模拟 new 的过程

```
function _new(){
  // 1、创建一个新对象
  let target = {};
  let [constructor, ...args] = [...arguments];  // 第一个参数是构造函数
  // 2、原型链连接
  target.__proto__ = constructor.prototype;
  // 3、将构造函数的属性和方法添加到这个新的空对象上。
  let result = constructor.apply(target, args);
  if(result && (typeof result == "object" || typeof result == "function")){
    // 如果构造函数返回的结果是一个对象，就返回这个对象
    return result
  }
  // 如果构造函数返回的不是一个对象，就返回创建的新对象。
  return target}let p2 = _new(Person, "小花")console.log(p2.name)  // 小花 console.log(p2 instanceof Person) // true
```

77、原型里面声明和 prototype 里面声明的一样 优先级

78、基本数据类型 和引用数据类型的区别和判断

1、基本数据类型和引用数据类型

ECMAScript 包括两个不同类型的值：基本数据类型和引用数据类型。

基本数据类型指的是简单的数据段，引用数据类型指的是有多个值构成的对象。

当我们把变量赋值给一个变量时，解析器首先要确认的就是这个值是基本类型值还是引用类型值。

2、常见的基本数据类型：

Number、String 、Boolean、Null 和 Undefined。基本数据类型是按值访问的，因为可以直接操作保存在变量中的实际值。示例：

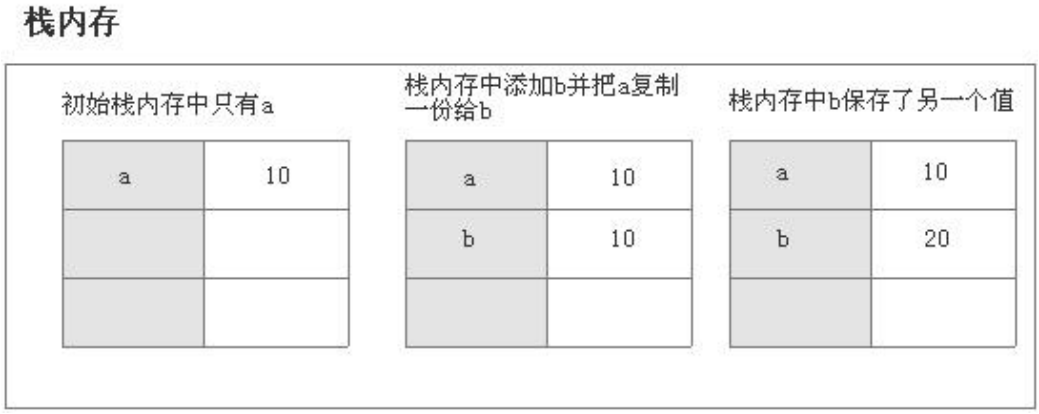
```
var a = 10;
```

```
var b = a;
b = 20;
console.log(a); // 10 值
```

上面，b 获取的是 a 值得一份拷贝，虽然，两个变量的值相等，但是两个变量保存了两个不同的基本数据类型值。

b 只是保存了 a 复制的一个副本。所以，b 的改变，对 a 没有影响。

下图演示了这种基本数据类型赋值的过程：



3、引用类型数据：

也就是对象类型 Object type，比如：Object 、Array 、Function 、Data 等。

javascript 的引用数据类型是保存在堆内存中的对象。

与其他语言的不同是，你不可以直接访问堆内存空间中的位置和操作堆内存空间。只能操作对象在栈内存中的引用地址。

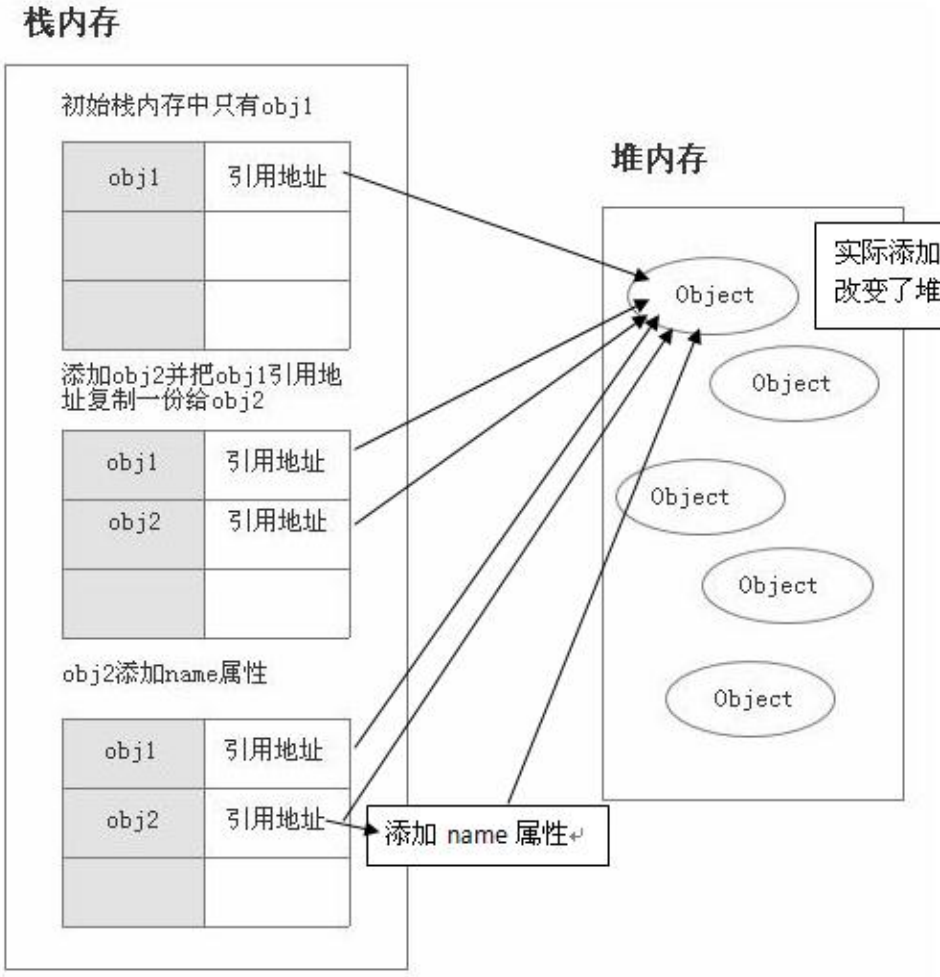
所以，引用类型数据在栈内存中保存的实际上是对象在堆内存中的引用地址。通过这个引用地址可以快速查找到保存中堆内存中的对象。

```
var obj1 = new Object();
var obj2 = obj1;
obj2.name = "我有名字了";
console.log(obj1.name); // 我有名字了
```

说明这两个引用数据类型指向了同一个堆内存对象。obj1 赋值给 onj2，实际上这个堆内存对象在栈内存的引用地址复制了一份给了 obj2，

但是实际上他们共同指向了同一个堆内存对象。实际上改变的是堆内存对象。

下面我们来演示这个引用数据类型赋值过程：



4、总结区别

a 声明变量时不同的内存分配：

1）原始值：存储在栈（stack）中的简单数据段，也就是说，它们的值直接存储在变量访问的位置。

这是因为这些原始类型占据的空间是固定的，所以可将他们存储在较小的内存区域 - 栈中。这样存储便于迅速查寻变量的值。

2）引用值：存储在堆（heap）中的对象，也就是说，存储在变量处的值是一个指针（point），指向存储对象的内存地址。

这是因为：引用值的大小会改变，所以不能把它放在栈中，否则会降低变量查寻的速度。相反，放在变量的栈空间中的值是该对象存储在堆中的地址。

地址的大小是固定的，所以把它存储在栈中对变量性能无任何负面影响。

b 不同的内存分配机制也带来了不同的访问机制

1）在 javascript 中是不允许直接访问保存在堆内存中的对象的，所以在访问一个对象时，

首先得到的是这个对象在堆内存中的地址，然后再按照这个地址去获得这个对象中的值，这就是传说中的按引用访问。

2）而原始类型的值则是可以直接访问到的。

c 复制变量时的不同

1）原始值：在将一个保存着原始值的变量复制给另一个变量时，会将原始值的副本赋值给新变量，此后这两个变量是完全独立的，他们只是拥有相同的value 而已。

2）引用值：在将一个保存着对象内存地址的变量复制给另一个变量时，会把这个内存地址赋值给新变量，

也就是说这两个变量都指向了堆内存中的同一个对象，他们中任何一个作出的改变都会反映在另一个身上。

（这里要理解的一点就是，复制对象时并不会在堆内存中新生成一个一模一样的对象，只是多了一个保存指向这个对象指针的变量罢了）。多了一个指针

d 参数传递的不同（把实参复制给形参的过程）

首先我们应该明确一点：ECMAScript 中所有函数的参数都是按值来传递的。

但是为什么涉及到原始类型与引用类型的值时仍然有区别呢？还不就是因为内存分配时的差别。

1）原始值：只是把变量里的值传递给参数，之后参数和这个变量互不影响。

2）引用值：对象变量它里面的值是这个对象在堆内存中的内存地址，这一点你要时刻铭记在心！

因此它传递的值也就是这个内存地址，这也就是为什么函数内部对这个参数的修改会体现在外部的原因了，因为它们都指向同一个对象。

79、栈和堆

一、程序的内存分配方式不同

栈区（stack）：编译器自动分配释放，存放函数的参数值，局部变量的值等，其操作方式类似于数据结构的栈。

堆区（heap）：一般是由程序员分配释放，若程序员不释放的话，程序结束时可能由 OS 回收，值得注意的是他与数据结构的堆是两回事，分配方式倒是类似于数据结构的链表。

二、申请方式不同

stack 由系统自动分配，heap 需要程序员自己申请。

C 中用函数 malloc 分配空间，用 free 释放，C++用 new 分配，用 delete 释放。

三、申请后系统的响应不同

栈：只要栈的剩余空间大于所申请的空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录内存地址的链表，当系统收到程序的申请时，遍历该链表，寻找第一个空间大于所申请的空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样代码中的 delete 或

free 语句就能够正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会将多余的那部分重新放入空闲链表中。
四、 申请的大小限制不同
栈：在 windows 下，栈是向低地址扩展的数据结构，是一块连续的内存区域，栈顶的地址和栈的最大容量是系统预先规定好的，能从栈获得的空间较小。
堆：堆是向高地址扩展的数据结构，是不连续的内存区域，这是由于系统是由链表在存储空闲内存地址，自然堆就是不连续的内存区域，且链表的遍历也是从低地址向高地址遍历的，堆得大小受限于计算机系统的有效虚拟内存空间，由此空间，堆获得的空间比较灵活，也比较大。
五、申请的效率不同
栈：栈由系统自动分配，速度快，但是程序员无法控制。
堆：堆是有程序员自己分配，速度较慢，容易产生碎片，不过用起来方便。
六、堆和栈的存储内容不同
栈：在函数调用时，第一个进栈的是主函数中函数调用后的下一条指令的地址，然后函数的各个参数，在大多数的 C 编译器中，参数是从右往左入栈的，当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令。
堆：一般是在堆的头部用一个字节存放堆的大小，具体内容由程序员安排。

80、设计模式

设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。
设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。
设计模式的使用
设计模式在软件开发中的两个主要用途。
开发人员的共同平台
设计模式提供了一个标准的术语系统，且具体到特定的情景。例如，单例设计模式意味着使用单个对象，这样所有熟悉单例设计模式的开发人员都能使用单个对象，并且可以通过这种方式告诉对方，程序使用的是单例模式。
最佳的实践
设计模式已经经历了很长一段时间的发展，它们提供了软件开发过程中面临的一般问题的最佳解决方案。学习这些模式有助于经验不足的开发人员通过一种简单快捷的方式来学习软件设计。
设计模式的类型

根据设计模式的参考书 Design Patterns - Elements of Reusable Object-Oriented Software（中文译名：设计模式 - 可复用的面向对象软件元素） 中所提到的，总共有 23 种设计模式。这些模式可以分为三大类：创建型模式（Creational Patterns）、结构型模式（Structural Patterns）、行为型模式（Behavioral Patterns）
设计模式的六大原则
1、开闭原则（Open Close Principle）
开闭原则的意思是：对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。
2、里氏代换原则（Liskov Substitution Principle）
里氏代换原则是面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。
3、依赖倒转原则（Dependence Inversion Principle）
这个原则是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。
4、接口隔离原则（Interface Segregation Principle）
这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。
5、迪米特法则，又称最少知道原则（Demeter Principle）
最少知道原则是指：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。
6、合成复用原则（Composite Reuse Principle）
合成复用原则是指：尽量使用合成/聚合的方式，而不是使用继承。

81、Webworker

什么是 Web Worker?
Web Worker 是 Html5 提出的能够在后台运行 javascript 的对象，独立于其他脚本，不会影响页面的性能，也不会影响你继续对于页面进行操作。通俗点讲，就是后台打杂的小工。
Why Web Worker?
Javascript 是单线程执行的，即某一时刻，一次只能做一件事情。Javascript 的单线程是为了保证对 dom 操作的统一性，即同一时刻不会既有删除和添加同一个 dom 的操作，为了保证
dom 树不会混乱，但是单线程执行是对于当下强大的多核 CPU 的一种浪费，无法充分发挥计算机的性能。为了

解决上述问题，就产生了 web worker，但是既要满足保证对 dom 树的统一性，又要支持多线程，这就决定了 web worker 在程序运行时的地位（特性）。打个比方，程序的主线程就像是只有一个厨师长的厨房，厨师长要亲自操刀整个做菜流程，配菜，加料等。web worker 就像是厨师长招来的学徒，可以帮忙切菜，帮忙看火候，帮忙调制佐料，但又不会影响整个做菜的主流程（没有权限）。所以 web worker 可以用于负责处理数据，或者执行可以延后的任务。

如何编写 Web Worker

可以用 VSCode 创建一个空白的文件夹，添加 index.html,index.js,worker.js 等文件。在 index.html 中引入 index.js，接着在 index.js 中编写如下代码

```
function  main(){
        let  worker =  new  Worker('worker.js');
        worker.postMessage('start');
        setTimeout(()=>worker.postMessage('end'), 5000);

        worker.onmessage =  function(event){
                console.log(event.data);
                worker.terminate();
                console.log(worker);
        }
}
```

创建一个 worker 线程，只要通过 new 关键字创建一个 worker 对象就行，传递的参数是 woker 线程要执行的 js 脚本，即我们现在给我们的小工分配了工作任务。

主线程和 worker 如何通信(厨师长如何给学徒安排任务)

主线程和 worker 之间的通信只能通过 postMessage 和 onMessage 来进行，主线程通过 postMessage 来向 worker 传递信息,worker 线程也是通过 postMessage 来向主线程传递信息，主线程通过 onMessage 来接收 worker 传递过来的信息。worker 线程通过 addEventListener('message', callback) 来监听主线程通过 postMessage 传递的信息。

```
let  number = 0;
let  intervalId = 0;

this.addEventListener('message', (e)=>{
        const data = e.data;
        console.log(e.data);

        switch(data){
                case  'start':
                        this.startCountNumber();  break;
                case  'end':
                        this.endCountNumber();  break;
                default:
                        break;
        }
```

```
})

function  startCountNumber() {
        this.intervalId =
setInterval(()=>{number++;console.log(number)}, 1000);
}

function  endCountNumber() {
        clearInterval(this.intervalId);
        self.postMessage('done')
}
```

这是一个用来计数的 worker，但是开始结束都由主线程开始，有兴趣的可以改写一下，主线程通知开始任务，当 worker 完成后，worker 通知主线程任务完成。（厨师长安排任务给学徒，学徒做完后通知厨师长）。

当编写完这两个 js 文件后，可以通过打开浏览器访问 index.html 来对这个 worker 进行测试。

这里有一个值得注意的点，如果测试的时候是直接打开本地 html 文件，那么不能用 chrome，chrome 出于安全问题，会没法加载 worker.js（CORS）。换用其它的浏览器即可，或者搭建本地服务器去运行亦可。

Web Worker 注意点

为什么用厨师长和学徒的例子，就是为了体现 web worker 的几个注意点。

1. 学徒无法直接干预厨师长的工作(worker 只能通过 postMessage 来向 mai'n thread 传递信息)
2. worker 没法直接读取或者操作 dom，也没法使用 window，parent 等对象，但可以读取 navigator, location 对象。
3. web worker 加载的脚本必需和主线程脚本同源（厨师长和学徒需要是一个厨房里的）
4.厨师长可以招多个学徒（主线程可以创建多个 worker 线程)

web worker 又分 dedicated worker 和 shared worker，有兴趣的可以深入了解下。

以及 web worker 的错误处理 onError, web worker 发送异步请求等。

82、dva 的原理

dva 是基于现有应用架构 (redux + react-router + redux-saga 等)的一层轻量封装，没有引入任何新概念，全部代码不到 100 行。

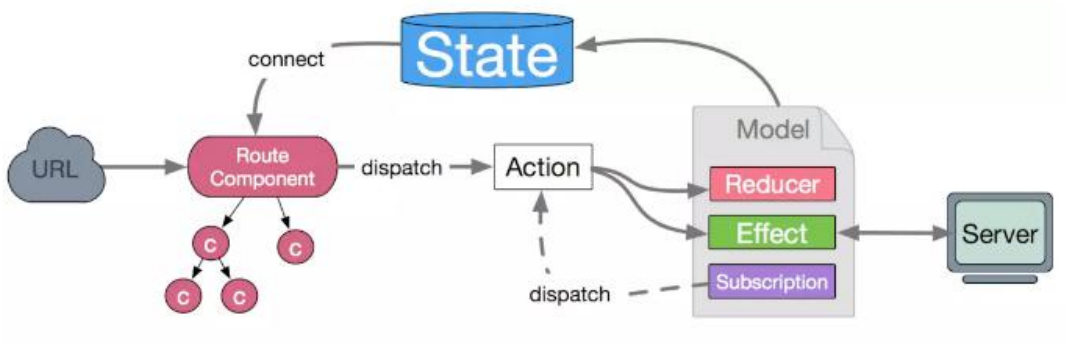
框架，而非类库

基于 redux, react-router, redux-soga 的轻量级封装

借鉴 elm 的概念, Reducer, Effect 和 Subscription

...

可以说，dva 是基于 react+redux 最佳实践上实现的封装方案，简化了 redux 和 redux-saga 使用上的诸多繁琐操作。



83、koa 中间件

中间件就是匹配路由（匹配任何路由或者特定的路由，其作用比如打印日志，查看权限）之前或者匹配路由完成之后所得一系列操作，功能有：
1.执行任何代码
2.修改请求和和响应对象
3.终结请求-响应循环
4.调用堆栈中的下一个中间件
通过 next 来实现
1.应用级中间件：匹配路由之前所做的一系列操作
2.路由级中间件：由于这个中间件只对这一个路由有作用，而不是对整个应用的路由都有作用，所以叫做路由级中间件
3.错误处理中间件
4.中间件的执行顺序
5 第三方中间

84、git 工作流

https://blog.csdn.net/qq_35865125/article/details/80049655

Git 工作流可以理解为团队成员遵守的一种代码管理方案，在 Git 中有以下几种常见工作流：

- 集中式工作流
  功能开发工作流
  Gitflow 工作流
  Forking 工作流

85、http 码

HTTP状态码分类

分类	分类描述
1**	信息，服务器收到请求，需要请求者继续执行操作
2**	成功，操作被成功接收并处理
3**	重定向，需要进一步的操作以完成请求
4**	客户端错误，请求包含语法错误或无法完成请求
5**	服务器错误，服务器在处理请求的过程中发生了错误

1**(信息类)：表示接收到请求并且继续处理
100—客户必须继续发出请求
101—客户要求服务器根据请求转换 HTTP 协议版本

2**(响应成功)：表示动作被成功接收、理解和接受
200—表明该请求被成功地完成，所请求的资源发送回客户端
201—提示知道新文件的 URL
202—接受和处理、但处理未完成
203—返回信息不确定或不完整
204—请求收到，但返回信息为空
205—服务器完成了请求，用户代理必须复位当前已经浏览过的文件
206—服务器已经完成了部分用户的 GET 请求

3**(重定向类)：为了完成指定的动作，必须接受进一步处理
300—请求的资源可在多处得到
301—本网页被永久性转移到另一个 URL
302—请求的网页被转移到一个新的地址，但客户访问仍继续通过原始 URL 地址，重定向，新的 URL 会在 response 中的 Location 中返回，浏览器将会使用新的 URL 发出新的 Request。
303—建议客户访问其他 URL 或访问方式
304—自从上次请求后，请求的网页未修改过，服务器返回此响应时，不会返回网页内容，代表上次的文档已经被缓存了，还可以继续使用
305—请求的资源必须从服务器指定的地址得到
306—前一版本 HTTP 中使用的代码，现行版本中不再使用
307—申明请求的资源临时性删除

4**(客户端错误类)：请求包含错误语法或不能正确执行
400—客户端请求有语法错误，不能被服务器所理解
401—请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用
402—保留有效 ChargeTo 头响应
403—禁止访问，服务器收到请求，但是拒绝提供服务
404—一个 404 错误表明可连接服务器，但服务器无法取得所请求的网页，请求资源不存在。eg：输入了错误的 URL
405—用户在 Request-Line 字段定义的方法不允许
406—根据用户发送的 Accept 拖，请求资源不可访问

```
    407—类似 401，用户必须首先在代理服务器上得到授权
    408—客户端没有在用户指定的饿时间内完成请求
    409—对当前资源状态，请求不能完成
    410—服务器上不再有此资源且无进一步的参考地址
    411—服务器拒绝用户定义的 Content-Length 属性请求
    412—一个或多个请求头字段在当前请求中错误
    413—请求的资源大于服务器允许的大小
    414—请求的资源 URL 长于服务器允许的长度
    415—请求资源不支持请求项目格式
    416—请求中包含 Range 请求头字段，在当前请求资源范围内没有 range 指示值，请求也不包含 If-Range 请求头字段
    417—服务器不满足请求 Expect 头字段指定的期望值，如果是代理服务器，可能是下一级服务器不能满足请求长。

  5**(服务端错误类)：服务器不能正确执行一个正确的请求
    500 - 服务器遇到错误，无法完成请求
    502 - 网关错误
    503：由于超载或停机维护，服务器目前无法使用，一段时间后可能恢复正常
```

86、工程目录怎么搭建的

87、HTML5 的 key feature 是什么？ 与 HTML 有哪些不同？ HTML5 有哪些优缺点？

88、使用 HTML5 的页面，交互数据是如何存储的？

1. 本地存储 localstorage
2. 本地存储 sessionstorage
3. 离线缓存（application cache）
4. Web SQL 关系数据库
5. IndexedDB 索引数据库

89、如何优化一个网站需要的 assets？

1. 文件合并（目的是减少 http 请求）：使用 css sprites 合并图片，一个网站经常使用小图标和小图片进行美化，但是很遗憾这些小图片占用了大量的 HTTP 请求，因此可以采用 sprites 的方式把所有的图片合并成一张图片 ，可以通过相关工具在线合并，也可以在 ps 中合并。

2.使用 CDN（内容分发网络）加速，降低通信距离。
3.缓存的使用，添加 Expire/Cache-Control 头。
4.启用 Gzip 压缩文件。
压缩 js 和 css 可以通过服务器动态脚本进行也可以更简单的使用 apache 服务器可以在网站根目录 .htaccess 中加入以下代码 AddOutputFilterByType DEFLATE text/html text/css text/plain text/xml application/x-javascript application/json
Header append Vary Accept-Encoding
这段代码的意思是调用服务器的压缩模块对以上文件输出之前进行 GZIP 压缩，gzip 的压缩之后所有文件都应该能减少 30%以上的体积。特别是对于大量使用 js 的博客有了 gzip 保驾护航之后速度能提高不少。
5.将 css 放在页面最上面。
6.将 script 放在页面最下面。
7.避免在 css 中使用表达式。
8.将 css, js 都放在外部文件中。
9.减少 DNS 查询。
10.文件压缩：最小化 css, js，减小文件体积。
11.避免重定向。
12.移除重复脚本。
13.配置实体标签 ETag。
14.使用 AJAX 缓存，让网站内容分批加载，局部更新。

90、HTML5 中，form 表单有哪些新增加的 element？

新的 input 的类型有。
email（自动验证 email 格式）
url（自动验证 url 格式）
number（只能输入数字）
range（类似音量滑动条）
Date pickers (date, month, week, time, datetime, datetime-local)（自带日期选择）
search（搜索域）
color（颜色选择）
datalist（自动验证内容是否在可选择选项中）

91、HTML5 中，Canvas 与 SVG 有什么区别？ 怎样在 Canvas 上画一条直线？

区别一：
svg 绘制出来的每一个图形的元素都是独立的 DOM 节点，能够方便的绑定事件或用来修改。canvas 输出的是一整幅画布；
区别二：

svg 输出的图形是矢量图形，后期可以修改参数来自由放大缩小，不会是真和锯齿。而 canvas 输出标量画布，就像一张图片一样，放大会失真或者锯齿
svg 是矢量图，canvas 是点阵图
```
var canvas = document.getElementById("myCanvas");
//获取对应的 CanvasRenderingContext2D 对象(画笔)
var ctx = canvas.getContext("2d");
//注意，Canvas 的坐标系是：Canvas 画布的左上角为原点(0,0)，向右为横坐标，向下为纵坐标，单位是像素(px)。
//开始一个新的绘制路径
ctx.beginPath();
//定义直线的起点坐标为(10,10)
ctx.moveTo(10, 10);
//定义直线的终点坐标为(50,10)
ctx.lineTo(50, 10);
//沿着坐标点顺序的路径绘制直线
ctx.stroke();
//关闭当前的绘制路径
ctx.closePath();
```

92、HTML5 中常使用的 API 有哪些？请解释下 Geolocation API 的使用

多媒体 API、拖放 API、文件 API、canvas、SVG、地理定位 geolocation
Element.classList
classList API 提供了我们多年来一种使用 JavaScript 工具库来实现的控制 CSS 的基本功能
```
// 增加一个 CSS 类
myElement.classList.add("newClass");
// 删除一个 CSS 类
myElement.classList.remove("existingClass");
// 检查是否拥有一个 CSS 类
myElement.classList.contains("oneClass");
// 反转一个 CSS 类的有无
myElement.classList.toggle("anotherClass");
```
ContextMenu API
这个新的 ContextMenu API 非常的有用：它并不会替换原有的右键菜单，而是将你的自定义右键菜单添加到浏览器的右键菜单里：
```
<section contextmenu="mymenu">
<!-- 添加菜单 -->
<menu type="context" id="mymenu">
<menuitem label="Refresh Post" onclick="window.location.reload();"
icon="/images/refresh-icon.png"></menuitem>
<menu label="Share on..." icon="/images/share_icon.gif">
<menuitem label="Twitter" icon="/images/twitter_icon.gif"
```

onclick="goTo('//twitter.com/intent/tweet?text=' + document.title + ':' + window.location.href);"></menuitem>
<menuitem label="Facebook" icon="/images/facebook_icon16x16.gif" onclick="goTo('//facebook.com/sharer/sharer.php?u=' + window.location.href);"></menuitem>
</menu>
</menu>
</section>
Element.dataset
使用 dataset API，程序员可以方便的获取或设置 data-*自定义属性：

window.postMessage API
即使是 IE8 也对 postMessage API 支持多年了，postMessage API 的功能是可以让你在两个浏览器窗口或 iframe 之间传递信息数据
autofocus 属性
autofocus 属性能够让 BUTTON, INPUT, 或 TEXTAREA 元素在页面加载完成时自动成为页面焦点：
addEventListener 添加事件监听函数
setInterval();

93、JS 中有哪些数据类型？

js 中有 5 种数据类型：Undefined、Null、Boolean、Number 和 String。
还有一种复杂的数据类型 Object，Object 本质是一组无序的名值对组成的。
Undefined 类型只有一个值，即 undefined，使用 var 声明变量，但是未对初始化的，这个变量就是 Undefined 类型的

94、Web 优化

使用 CDN
减少外部 http 协议
使用预获取
压缩 HTML、css 和 JavaScript
优化图片
Ajax 请求方式

95、缓存处理

html
只要加在头部就可以了.
<HEAD>
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">

<META HTTP-EQUIV="Cache-Control" CONTENT="no-cache">
<META HTTP-EQUIV="Expires" CONTENT="0">
</HEAD>
说明：HTTP 头信息“Expires”和“Cache-Control”为应用程序服务器提供了一个控制浏览器和代理服务器上缓存的机制。HTTP 头信息 Expires 告诉代理服务器它的缓存页面何时将过期。HTTP1.1 规范中新定义的头信息 Cache-Control 可以通知浏览器不缓存任何页面。当点击后退按钮时，浏览器重新访问服务器已获取页面。如下是使用 Cache-Control 的基本方法：

1）no-cache:强制缓存从服务器上获取新的页面

2）no-store: 在任何环境下缓存不保存任何页面

HTTP1.0 规范中的 Pragma:no-cache 等同于 HTTP1.1 规范中的 Cache-Control:no-cache，同样可以包含在头信息中。

在需要打开的 url 后面增加一个随机的参数：

增加参数前：url=test/test.jsp

增加参数后：url=test/test.jsp?ranparam=random()

说明：因为每次请求的 url 后面的参数不一样，相当于请求的是不同的页面，用这样的方法来曲线救国，清除缓存。

96、递归

先递进，再回归——这就是「递归」

所谓递归，就是会在函数内部代码中，调用这个函数本身，所以，我们必须要找出递归的结束条件，不然的话，会一直调用自己，进入无底洞。也就是说，我们需要找出当参数为啥时，递归结束，之后直接把结果返回，请注意，这个时候我们必须能根据这个参数的值，能够直接知道函数的结果是什么。

递归的三大要素

第一要素：明确你这个函数想要干什么

第二要素：寻找递归结束条件

第三要素：找出函数的等价关系式

97、Es6 class 静态方法

不需要实例化类，即可直接通过该类来调用的方法，即称之为“静态方法”。将类中的方法设为静态方法也很简单，在方法前加上 static 关键字即可。这样该方法就不会被实例继承！

```
class Box{
    static a(){
        return "我是Box类中的，实例方法，无须实例化，可直接调用！"
    }
}
//通过类名直接调用
console.log(Box.a());//我是Box类中的，实例方法，无须实例化，可直接调用!
```

父类的静态方法， 可以被子类继承

倘若想通过子类的静态方法调用父类的静态方法，需要从 super 对象上

```
class Box{
   constructor(){
       this.name="实例属性"
   }
}
Box.prop1="静态属性1";
Box.prop2="静态属性2";
console.log(Box.prop1,Box.prop2);//静态属性1  静态属性2
```

调用

静态属性指的是 Class 本身的属性， 即 Class.propname， 而不是定义在实例对象（ this） 上的属性。

98、https 协议

http 协议与 https 协议的区别?

1、https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。

2、http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。

3、http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

4、http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

https 协议的工作原理?

我们都知道 HTTPS 能够加密信息，以免敏感信息被第三方获取，所以很多银行网站或电子邮箱等等安全级别较高的服务都会采用 HTTPS 协议。

客户端在使用 HTTPS 方式与 Web 服务器通信时有以下几个步骤，如图所示。

（1）客户使用 https 的 URL 访问 Web 服务器，要求与 Web 服务器建立 SSL 连接。

（2）Web 服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。

（3）客户端的浏览器与 Web 服务器开始协商 SSL 连接的安全等级，也就是信息加密的等级。

（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

（5）Web 服务器利用自己的私钥解密出会话密钥。

（6）Web 服务器利用会话密钥加密与客户端之间的通信。

https 的优缺点?

虽然说 HTTPS 有很大的优势，但其相对来说，还是存在不足之处的：

（1）HTTPS 协议握手阶段比较费时，会使页面的加载时间延长近 50%，增加 10% 到 20% 的耗电；

（2）HTTPS 连接缓存不如 HTTP 高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；

（3）SSL 证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。

（4）SSL 证书通常需要绑定 IP，不能在同一 IP 上绑定多个域名，IPv4 资源不可能支撑这个消耗。

（5）HTTPS 协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL 证书的信用链体系并不安全，特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。

99、vue 的生命周期

beforeCreate（创建前） 在数据观测和初始化事件还未开始

created（创建后） 完成数据观测，属性和方法的运算，初始化事件，$el 属性还没有显示出来

beforeMount（载入前） 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。注意此时还没有挂载 html 到页面上。

mounted（载入后） 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。

beforeUpdate（更新前） 在数据更新之前调用，发生在虚拟 DOM 重新渲染和打补丁之前。可以在该钩子中进一步地更改状态，不会触发附加的重渲染过程。

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。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。

beforeDestroy（销毁前） 在实例销毁之前调用。实例仍然完全可用。

destroyed（销毁后） 在实例销毁之后调用。调用后，所有的事件监听器会被

移除，所有的子实例也会被销毁。该钩子在服务器端渲染期间不被调用。

1.什么是 vue 生命周期？

答： Vue 实例从创建到销毁的过程，就是生命周期。从开始创建、初始化数据、编译模板、挂载 Dom→渲染、更新→渲染、销毁等一系列过程，称之为 Vue 的生命周期。

2.vue 生命周期的作用是什么？

答：它的生命周期中有多个事件钩子，让我们在控制整个 Vue 实例的过程时更容易形成好的逻辑。

3.vue 生命周期总共有几个阶段？

答：它可以总共分为 8 个阶段：创建前/后, 载入前/后,更新前/后,销毁前/销毁后。

4.第一次页面加载会触发哪几个钩子？

答：会触发 下面这几个 beforeCreate, created, beforeMount, mounted 。

5.DOM 渲染在 哪个周期中就已经完成？

答：DOM 渲染在 mounted 中就已经完成了。

100、css 只在当前组件起作用

在 style 标签中写入 **scoped** 即可 例如：<style scoped></style>

101、v-if 和 v-show 区别

v-if 按照条件是否渲染，v-show 是 display 的 block 或 none；

102、vue 常用的修饰符？

.prevent: 提交事件不再重载页面；.stop: 阻止单击事件冒泡；.self: 当事件发生在该元素本身而不是子元素的时候会触发；.capture: 事件侦听，事件发生的时候会调用

103、vue 中 key 值的作用？

当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用"就地复用"策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序，而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素。key 的作用主要是为了高效的更新虚拟 DOM。

104、什么是 vue 的计算属性？

在模板中放入太多的逻辑会让模板过重且难以维护，在需要对数据进行复杂处理，且可能多次使用的情况下，尽量采取计算属性的方式。好处：①使得数据处理结构清晰；②依赖于数据，数据更新，处理结果自动更新；③计算属性内部 this 指向 vm 实例；④在 template 调用时，直接写计算属性名即可；⑤常用的是 getter 方法，获取数据，也可以使用 set 方法改变数据；⑥相较于 methods，不管依赖的数据变不变，methods 都会重新计算，但是依赖数据不变的时候 computed 从缓存中获取，不会重新计算。

105、vue 等单页面应用及其优缺点

优点：Vue 的目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件，核心是一个响应的数据绑定系统。MVVM、数据驱动、组件化、轻量、简洁、高效、快速、模块友好。
缺点：不支持低版本的浏览器，最低只支持到 IE9；不利于 SEO 的优化（如果要支持 SEO，建议通过服务端来进行渲染组件）；第一次加载首页耗时相对长一些；不可以使用浏览器的导航按钮需要自行实现前进、后退。

106、对于 MVVM 的理解？

MVVM 是 Model-View-ViewModel 的缩写。

Model 代表数据模型，也可以在 Model 中定义数据修改和操作的业务逻辑。

View 代表 UI 组件，它负责将数据模型转化成 UI 展现出来。

ViewModel 监听模型数据的改变和控制视图行为、处理用户交互，简单理解就是一个同步 View 和 Model 的对象，连接 Model 和 View。

在 MVVM 架构下，View 和 Model 之间并没有直接的联系，而是通过 ViewModel 进行交互，Model 和 ViewModel 之间的交互是双向的，因此 View 数据的变化会同步到 Model 中，而 Model 数据的变化也会立即反应到 View 上。

ViewModel 通过双向数据绑定把 View 层和 Model 层连接了起来，而 View 和 Model 之间的同步工作完全是自动的，无需人为干涉，因此开发者只需关注业务逻辑，不需要手动操作 DOM, 不需要关注数据状态的同步问题，复杂的数据状态维护完全由 MVVM 来统一管理。

107、Vue 实现数据双向绑定

vue 实现数据双向绑定主要是：采**用数据劫持结合发布者-订阅者模式**的方式，通过 **Object.defineProperty**（）来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应监听回调。当把一个普通 Javascript 对象传给 Vue 实例来作为它的 data 选项时，Vue 将遍历它的属性，用 Object.defineProperty 将它们转为 getter/setter。用户看不到 getter/setter，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。

vue 的数据双向绑定 将 MVVM 作为数据绑定的入口，整合 Observer，Compile 和 Watcher 三者，通过 Observer 来监听自己的 model 的数据变化，通过 Compile 来解析编译模板指令（vue 中是用来解析 {{}}），最终利用 watcher 搭起 observer 和 Compile 之间的通信桥梁，达到数据变化 —>视图更新；视图交互变化（input）—>数据 model 变更双向绑定效果。

108、Vue 组件之间数据传递

1.父组件与子组件传值

父组件传给子组件：子组件通过 props 方法接受数据;

子组件传给父组件：$emit 方法传递参数

2.非父子组件间的数据传递，兄弟组件传值

eventBus，就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件。项目比较小时，用这个比较合适。（虽然也有不少人推荐直接用 VUEX，具体来说看需求咯。技术只是手段，目的达到才是王道。）

109、Vue、react、angula 区别？

1.与 AngularJS 的区别

相同点：
都支持指令：内置指令和自定义指令；都支持过滤器：内置过滤器和自定义过滤器；都支持双向数据绑定；都不支持低端浏览器。

不同点：

AngularJS 的学习成本高，比如增加了 Dependency Injection 特性，而 Vue.js 本身提供的 API 都比较简单、直观；在性能上，AngularJS 依赖对数据做脏检查，

所以 Watcher 越多越慢；Vue.js 使用基于依赖追踪的观察并且使用异步队列更新，所有的数据都是独立触发的。

2.与 React 的区别

相同点：

React 采用特殊的 JSX 语法，Vue.js 在组件开发中也推崇编写.vue 特殊文件格式，对文件内容都有一些约定，两者都需要编译后使用；中心思想相同：一切都是组件，组件实例之间可以嵌套；都提供合理的钩子函数，可以让开发者定制化地去处理需求；都不内置列数 AJAX，Route 等功能到核心包，而是以插件的方式加载；在组件开发中都支持 mixins 的特性。

不同点：

React 采用的 Virtual DOM 会对渲染出来的结果做脏检查；Vue.js 在模板中提供了指令，过滤器等，可以非常方便，快捷地操作 Virtual DOM。

110、Vue 路由钩子函数

首页可以控制导航跳转，beforeEach，afterEach 等，一般用于页面 title 的修改。一些需要登录才能调整页面的重定向功能。

beforeEach 主要有 3 个参数 to，from，next：

to：route 即将进入的目标路由对象，

from：route 当前导航正要离开的路由

next：function 一定要调用该方法 resolve 这个钩子。执行效果依赖 next 方法的调用参数。可以控制网页的跳转。

111、vue 框架与 jQuery 类库的区别

Vue 直接操作视图层，它通过 Vue 对象将数据和 View 完全分离开来了。对数据进行操作不需要引用相应的 DOM 节点，只需要关注逻辑，完全实现了视图层和逻辑层的解耦；

Jquery 的操作是基于 DOM 节点的操作，jQuery 是使用选择器（$）选取 DOM 对象，对其进行赋值、取值、事件绑定等操作，其实和原生的 js 的区别只在于可以更方便的选取和操作 DOM 对象，而数据和界面是在一起的。它的优势在于良好的封装和兼容，使调用简单方便。

112、vue-cli 是什么

`Vue.js`提供一个官方命令行工具，可用于快速搭建大型单页应用（在一个完成的应用或者站点中，只有一个完整的 HTML 页面，这个页面有一个容器，可以把需要加载的代码（以组件的方式）插入到该容器中）。

该工具提供开箱即用的构建工具配置，带来现代化的前端开发流程。只需几分钟即可创建并启动一个带热重载、保存时静态检查以及可用于生产环境的构建配置的项目。

113、Computed 和 watch 什么区别

- **computed:**

1、computed 是计算属性,也就是计算值
2、computed 具有缓存性,computed 的值在 getter 执行后是会缓存的，只有在它依赖的属性值改变之后，下一次获取 computed 的值时才会重新调用对应的 getter 来计算
3、computed 适用于计算比较消耗性能的计算场景

- ** watch**

1、Vue 提供了一种更通用的方式来观察和响应 Vue 实例上的数据变动：侦听属性。
2、无缓存性，页面重新渲染时值不变化也会执行

114、Vuex 是什么？简述 action 和 mutation 的区别？

Vuex 类似 Redux 的状态管理器，用来管理 Vue 的所有组件状态。
vuex 就是一个存放多个组件共用的一个数据的存放、更改、处理的一个容器，就是说来存放处理公共数据的工具，存放的数据一变，各个组件都会更新，也就是说存放的数据是响应式的。

115、Vue 组件中 data 为什么是一个函数

组件是可复用的 vue 实例，一个组件被创建好之后，就可能被用在各个地方，而组件不管被复用了多少次，组件中的 data 数据都应该是相互隔离，互不影响的，基于这一理念，组

件每复用一次，data 数据就应该被复制一次，之后，当某一处复用的地方组件内 data 数据被改变时，其他复用地方组件的 data 数据不受影响。

类似于给每个组件实例创建一个私有的数据空间，让各个组件实例维护各自的数据。而单纯的写成对象形式，就使得所有组件实例共用了一份 data，就会造成一个变了全都会变的结果

116、说出至少 5 个 ES6 的新特性，并简述它们的作用

1、let 关键字，用于声明只在块级作用域起作用的变量。

2、const 关键字，用于声明一个常量。

3、解构赋值，一种新的变量赋值方式。常用于交换变量值，提取函数返回值，设置默认值。

4、Symbol 数据类型，定义一个独一无二的值。

5、Proxy 代理，用于编写处理函数，来拦截目标对象的操作。

6、for...of 遍历，可遍历具有 iterator 接口的数据结构。

7、Set 结构，存储不重复的成员值的集合。

8、Map 结构，键名可以是任何类型的键值对集合。

9、Promise 对象，更合理、规范地处理异步操作。

10、Class 类定义类和更简便地实现类的继承。

117、 promise 对象的用法，手写一个 promise

```
var promise = new Promise((resolve,reject) => {
    if (操作成功) {
        resolve(value)
    } else {
        reject(error)
    }
})
promise.then(function (value) {
    // success
},function (value) {
    // failure
})
```

118、es5 和 es6 的区别，说一下你所知道的 es6

ECMAScript5，即 ES5，是 ECMAScript 的第五次修订，于 2009 年完成标准化
ECMAScript6,即 ES6,是 ECMAScript 的第六次修订,于 2015 年完成,也称 ES2015
ES6 是继 ES5 之后的一次改进，相对于 ES5 更加简洁，提高了开发效率
ES6 新增的一些特性：
1.let 声明变量和 const 声明常量，两个都有块级作用域
ES5 中是没有块级作用域的，并且 var 有变量提升，在 let 中，使用的变量一定要进行声明
2.箭头函数
ES6 中的函数定义不再使用关键字 function(),而是利用了()=>来进行定义
3.模板字符串
模板字符串是增强版的字符串，用反引号（`）标识，可以当作普通字符串使用，也可以用来定义多行字符串
4.解构赋值
ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值
5.for of 循环
for...of 循环可以遍历数组、Set 和 Map 结构、某些类似数组的对象、对象，以及字符串
6.import、export 导入导出
ES6 标准中，Js 原生支持模块(module)。将 JS 代码分割成不同功能的小块进行模块化，将不同功能的代码分别写在不同文件中，各模块只需导出公共接口部分，然后通过模块的导入的方式可以在其他地方使用
7.set 数据结构
Set 数据结构，类似数组。所有的数据都是唯一的，没有重复的值。它本身是一个构造函数
8....展开运算符
可以将数组或对象里面的值展开；还可以将多个值收集为一个变量
9.修饰器 @
decorator 是一个函数，用来修改类甚至于是方法的行为。修饰器本质就是编译时执行的函数
10.class 类的继承
ES6 中不再像 ES5 一样使用原型链实现继承，而是引入 Class 这个概念
11.async、await
使用 async/await, 搭配 promise,可以通过编写形似同步的代码来处理异步流程, 提高代码的简洁性和可读性
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
12.promise
Promise 是异步编程的一种解决方案，比传统的解决方案（回调函数和事件）更合理、强大
13.Symbol
Symbol 是一种基本类型。Symbol 通过调用 symbol 函数产生，它接收一个

可选的名字参数，该函数返回的 symbol 是唯一的
14. Proxy 代理
使用代理（Proxy）监听对象的操作，然后可以做一些相应事情

119、介绍下 Set、Map 的区别？

应用场景 Set 用于数据重组，Map 用于数据储存
Set：
（1）成员不能重复
（2）只有键值没有键名，类似数组
（3）可以遍历，方法有 add, delete,has
Map:
（1）本质上是健值对的集合，类似集合
（2）可以遍历，可以跟各种数据格式转换

120、ECMAScript 6 怎么写 class ，为何会出现 class？

ES6 的 class 可以看作是一个语法糖，它的绝大部分功能 ES5 都可以做到，新的 class 写法只是让对象原型的写法更加清晰、更像面向对象编程的语法

```
//定义类 class Point {

    constructor(x,y){ //构造方法

        this.x = x; //this 关键字代表实例对象

        this.y = y;

    }toString(){

        return '(' + this.x + ',' + this.y + ')';

    }}
```

121、promise 有几种状态，什么时候会进入 catch？

三个状态：pending、fulfilled、reject

两个过程：padding -> fulfilled、padding -> rejected

当 pending 为 rejectd 时，会进入 catch

122、Promise 中 reject 和 catch 处理上有什么区别

reject 是用来抛出异常，catch 是用来处理异常

reject 是 Promise 的方法，而 catch 是 Promise 实例的方法

reject 后的东西，一定会进入 then 中的第二个回调，如果 then 中没有写第二个回调，则进入 catch

网络异常（比如断网），会直接进入 catch 而不会进入 then 的第二个回调

123、forEach、for in、for of 三者区别

forEach 更多的用来遍历数组

for in 一般常用来遍历对象或 json

for of 数组对象都可以遍历，遍历对象需要通过和 Object.keys()

for in 循环出的是 key，for of 循环出的是 value

124、何时在 ES6 中使用箭头函数？

以下是一些经验分享：

- 在全局作用域内和 Object.prototype 属性中使用 function 。
- 为对象构造函数使用 class。
- 其它情况使用箭头函数。

为啥大多数情况都使用箭头函数？

- 作用域安全性:当箭头函数被一致使用时，所有东西都保证使用与根对象相同的 thisObject。如果一个标准函数回调与一堆箭头函数混合在一起，那么作用域就有可能变得混乱。
- 紧凑性:箭头函数更容易读写。
- 清晰度:使用箭头函数可明确知道当前 this 指向。

125、.call 和.apply 有什么区别？

.call 和.apply 均用于调用函数，并且第一个参数将用作函数中 this 的值。但是，.call 将逗号分隔的参数作为下一个参数，而.apply 将参数数组作为下一个参数。简单记忆法：C 用于 call 和逗号分隔，A 用于 apply 和参数数组。

126、为什么要使用 ES6？

选择使用类的一些原因：

语法更简单，更不容易出错。

使用新语法比使用旧语法更容易(而且更不易出错)地设置继承层次结构。

class 可以避免构造函数中使用 new 的常见错误（如果构造函数不是有效的对象，则使构造函数抛出异常）。

- 用新语法调用父原型方法的版本比旧语法要简单得多，用 super.method() 代替 ParentConstructor.prototype.method.call(this) 或 Object.getPrototypeOf(Object.getPrototypeOf(this)).method.call(this)

127、解释一下 Object.freeze() 和 const 的区别

const 和 Object.freeze 是两个完全不同的概念。

const 声明一个只读的变量，一旦声明，常量的值就不可改变：

Object.freeze 适用于值，更具体地说，适用于对象值，它使对象不可变，即不能更改其属性。

128、什么时候不使用箭头函数？

不应该使用箭头函数一些情况：

当想要函数被提升时(箭头函数是匿名的)

要在函数中使用 this/arguments 时，由于箭头函数本身不具有 this/arguments，因此它们取决于外部上下文

使用命名函数(箭头函数是匿名的)

使用函数作为构造函数时(箭头函数没有构造函数)

当想在对象字面是以将函数作为属性添加并在其中使用对象时，因为咱们无法访问 this 即对象本身。

129、能否比较模块模式与构造函数/原型模式的用法？

模块模式通常用于命名空间，在该模式中，使用单个实例作为存储来对相关函数和对象进行分组。这是一个不同于原型设计的用例,它们并不是相互排斥,咱们可

以同时使用它们(例如，将一个构造函数放在一个模块中，并使用 new MyNamespace.MyModule.MyClass(arguments))。

构造函数和原型是实现类和实例的合理方法之一。它们与模型并不完全对应，因此通常需要选择一个特定的 scheme 或辅助方法来实现原型中的类。

130、举一个柯里化函数，并说明柯里化的好处

柯里化是一种模式，其中一个具有多个参数的函数被分解成多个函数，当被串联调用时，这些函数将一次累加一个所需的所有参数。这种技术有助于使用函数式编写的代码更容易阅读和编写。需要注意的是，要实现一个函数，它需要从一个函数开始，然后分解成一系列函数，每个函数接受一个参数。

```
function curry(fn) {
  if (fn.length === 0) {
    return fn;
  }

  function _curried(depth, args) {
    return function(newArgument) {
      if (depth - 1 === 0) {
        return fn(...args, newArgument);
      }
      return _curried(depth - 1, [...args, newArgument]);
    };
  }

  return _curried(fn.length, []);
}

function add(a, b) {
  return a + b;
}

var curriedAdd = curry(add);
var addFive = curriedAdd(5);

var result = [0, 1, 2, 3, 4, 5].map(addFive); // [5, 6, 7, 8, 9, 10]
```

131、Vue 的路由实现：hash 模式 、 history 模式、abstract 模式

hash 模式：在浏览器中符号“#”，#以及#后面的字符称之为 hash，用 window.location.hash 读取；

特点：hash 虽然在 URL 中，但不被包括在 HTTP 请求中；用来指导浏览器动作，对服务端安全无用，hash 不会重加载页面。
hash 模式下，仅 hash 符号之前的内容会被包含在请求中，
如 http://www.xxx.com，因此对于后端来说，即使没有做到对路由的全覆盖，也不会返回 404 错误。
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.abc.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）。

history 模式：history 采用 HTML5 的新特性；且提供了两个新方法：pushState（），replaceState（）可以对浏览器历史记录栈进行修改，以及 popState 事件的监听到状态变更。
history 模式下，前端的 URL 必须和实际向后端发起请求的 URL 一致，
如 http://www.xxx.com/items/id。后端如果缺少对 /items/id 的路由处理，将返回 404 错误。Vue-Router 官网里如此描述：“不过这种模式要玩好，还需要后台配置支持……所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。”
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）。

abstract 模式 ：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.
(后续补上)

132、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。
有两种常见的试图改变一个 prop 的情形 ：

这个 prop 用来传递一个初始值；这个子组件接下来希望将其作为一个本地的 prop 数据来使用。 在这种情况下，最好定义一个本地的 data 属性并将这个 prop 用作其初始值：

```
props: ['initialCounter'],
data: function () {
  return {
    counter: this.initialCounter
  }
}
```

这个 prop 以一种原始的值传入且需要进行转换。 在这种情况下，最好使用这

个 prop 的值来定义一个计算属性 **props: ['size'],computed: {**

```
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

133、虚拟 DOM 优缺点及实现原理

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 的实现原理主要包括以下 3 部分：
①用 JavaScript 对象模拟真实 DOM 树，对真实 DOM 进行抽象；
②diff 算法 — 比较两棵虚拟 DOM 树的差异；
③pach 算法 — 将两个虚拟 DOM 对象的差异应用到真正的 DOM 树。

134、Vue 项目优化

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

135、vue3.0 特性的了解

1.监测机制的改变
Vue3.0 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：①只能监测属性，不能监测对象；②检测属性的添加和删除；③检测数组索引和长度的变更；④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，还有一些其它的更改：
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 其它部分渲染组建内容）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。

136、Vue 路由跳转方式有哪几种？

1.router-link

```
**不带参数**
<router-link :to="{name:'home'}">
<router-link :to="{path:'/home'}"> //name,path 都行, 建议用 name
// 注意: router-link 中链接如果是'/'开始就是从根路由开始, 如果开始不带'/', 则从当前路由开始。
带参数
<router-link :to="{name:'home', params: {id:1}}">       // params 传参数 (类似 post)
// 路由配置 path: "/home/:id" 或者 path: "/home:id"
// 不配置 path ,第一次可请求,刷新页面 id 会消失
// 配置 path,刷新页面 id 会保留
// html 取参  $route.params.id
// script 取参  this.$route.params.id
<router-link :to="{name:'home', query: {id:1}}"> // query 传参数 (类似 get,url 后面会显示参数)
// 路由可不配置
// html 取参  $route.query.id
// script 取参  this.$route.query.id
```

2.this.$router.push() (函数里面调用)

```
不带参数
this.$router.push('/home')this.$router.push({name:'home'})this.$router.push({path:'/home'})
query 传参 this.$router.push({name:'home',query:{id:'1'}})this.$router.push({path:'/home',query: {id:'1'}}) // html 取参 $route.query.id// script 取参  this.$route.query.id
```

2.1 params 传参

```
this.$router.push({name:'home',params: {id:'1'}})  // 只能用 name// 路由配置 path: "/home/:id" 或者 path: "/home:id" ,// 不配置 path ,第一次可请求,刷新页面 id 会消失// 配置 path,刷新页面 id 会保留// html 取参  $route.params.id// script 取参  this.$route.params.id
```

2.2 query 和 params 区别

```
query 类似 get, 跳转之后页面 url 后面会拼接参数,类似?id=1, 非重要性的可以这样传, 密码之类还是用 params 刷新页面 id 还在 params 类似 post, 跳转之后页面 url 后面不会拼接参数 , 但是刷新页面 id 会消失
```

3.this.$router.replace() (用法同上,push)

4. this.$router.go(n) ()

137、vue 编译

首先说一下 vue 项目如何编译，其实很简单，cd 到项目文件夹，然后执行命令：

npm run build

vue 项目编译打包

1. npm run build

2. npm install -g serve

3.serve dist

138、什么是 jsx

React 使用 JSX 来替代常规的 JavaScript。

JSX 是一个看起来很像 XML 的 JavaScript 语法扩展。

我们不需要一定使用 JSX，但它有以下优点：

JSX 执行更快，因为它在编译为 JavaScript 代码后进行了优化。

它是类型安全的，在编译过程中就能发现错误。

使用 JSX 编写模板更加简单快速。

我们先看下以下代码：

```
const element = <h1>Hello, world!</h1>;
```

这种看起来可能有些奇怪的标签语法既不是字符串也不是 HTML。

它被称为 JSX，一种 JavaScript 的语法扩展。我们推荐在 React 中使用 JSX 来描述用户界面。

JSX 是在 JavaScript 内部实现的。

我们知道元素是构成 React 应用的最小单位，JSX 就是用来声明 React 当中的元素。

与浏览器的 DOM 元素不同，React 当中的元素事实上是普通的对象，React DOM 可以确保 浏览器 DOM 的数据内容与 React 元素保持一致。

要将 React 元素渲染到根 DOM 节点中，我们通过把它们都传递给 ReactDOM.render() 的方法来将其渲染到页面上

139、主浏览器兼容性问题？

浏览器兼容问题一：不同浏览器的标签默认的外补丁和内补丁不同

问题症状：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。

碰到频率:100%

解决方案：CSS 里 *{margin:0;padding:0;}

备注：这个是最常见的也是最易解决的一个浏览器兼容性问题，几乎所有的 CSS 文件开头都会用通配符*来设置各个标签的内外补丁是 0。

浏览器兼容问题二：块属性标签 float 后，又有横行的 margin 情况下，在 IE6 显示 margin 比设置的大

问题症状:常见症状是 IE6 中后面的一块被顶到下一行

碰到频率：90%（稍微复杂点的页面都会碰到，float 布局最常见的浏览器兼容问题）

解决方案:在 float 的标签样式控制中加入 display:inline;将其转化为行内属性

备注：我们最常用的就是 div+CSS 布局了，而 div 就是一个典型的块属性标签，横向布局的时候我们通常都是用 div float 实现的，横向的间距设置如果用 margin 实现，这就是一个必然会碰到的兼容性问题。

浏览器兼容问题三：设置较小高度标签（一般小于 10px），在 IE6，IE7，遨游中高度超出自己设置高度

问题症状：IE6、7 和遨游里这个标签的高度不受控制，超出自己设置的高度

碰到频率：60%

解决方案:给超出高度的标签设置 overflow:hidden;或者设置行高 line-height 小于你设置的高度。

备注：这种情况一般出现在我们设置小圆角背景的标签里。出现这个问题的原因是 IE8 之前的浏览器都会给标签一个最小默认的行高的高度。即使你的标签是空的，这个标签的高度还是会达到默认的行高。

浏览器兼容问题四：行内属性标签，设置 display:block 后采用 float 布局，又有横行的 margin 的情况，IE6 间距 bug

问题症状：IE6 里的间距比超过设置的间距

碰到几率：20%

解决方案：在 display:block;后面加入 display:inline;display:table;

备注:行内属性标签，为了设置宽高，我们需要设置 display:block;(除了 input 标签比较特殊)。在用 float 布局并有横向的 margin 后，在 IE6 下，他就具有了块属性 float 后的横向 margin 的 bug。不过因为它本身就是行内属性标签，所以我们再加上 display:inline 的话，它的高宽就不可设了。这时候我们还需要在 display:inline 后面加入 display:talbe。

浏览器兼容问题五：图片默认有间距

问题症状：几个 img 标签放在一起的时候，有些浏览器会有默认的间距，加了问题一中提到的通配符也不起作用。

碰到几率：20%

解决方案：使用 float 属性为 img 布局

备注：因为 img 标签是行内属性标签，所以只要不超出容器宽度，img 标签都会排在一行里，但是部分浏览器的 img 标签之间会有个间距。去掉这个间距使用 float 是正道。（我的一个学生使用负 margin，虽然能解决，但负 margin 本身就是容易引起浏览器兼容问题的用法，所以我禁止他们使用）

浏览器兼容问题六：标签最低高度设置 min-height 不兼容

问题症状：因为 min-height 本身就是一个不兼容的 CSS 属性，所以设置 min-height 时不能很好的被各个浏览器兼容

碰到几率：5%

解决方案：如果我们要设置一个标签的最小高度 200px，需要进行的设置为：{min-height:200px; height:auto !important; height:200px; overflow:visible;}

备注：在 B/S 系统前端开时，有很多情况下我们又这种需求。当内容小于一个值（如 300px）时。容器的高度为 300px；当内容高度大于这个值时，容器高度被撑高，而不是出现滚动条。这时候我们就会面临这个兼容性问题。

5.超链接访问过后 样式就混乱了，hover 样式不出现了。其实主要是其 CSS 属性的排序问题。

解决方案：最好按照这个顺序：L-V-H-A

a:link{} a:visited{} a:hover{} a:active{}

6.chrome 下默认会将小于 12px 的文本强制按照 12px 来解析。

解决办法是给其添加属性：-webkit-text-size-adjust: none;

9.opacity 透明度问题：一般就直接 opacity: 0.6 ; IE 就 filter: alpha(opacity=60)。ie6 下：filter:progid:DXImageTransform.Microsoft.Alpha(style=0,opacity=60);

10.IE6 下 div 高度无法小于 10px

解决方案：verflow:hidden | zoom:0.08 | line-height:1px|font-size:0

11.由于浮动引起的无法识别父盒子高度问题

解决方案：4 种清除浮动的方法

清除浮动方法 1：给浮动的元素的祖先元素加高度。有高度的盒子，才能关住浮动。

清除浮动方法 2：clear:both;但是有致命缺陷，margin 失效。

清除浮动方法 3：隔墙法。在两个浮动的 div 间增加一个空 div，就像一个屏障隔开了 2 个浮动，使两个浮动间互不影响。缺点：额外增加了很多的标签，对页面结构及其的不好。

清除浮动方法 4：overflow:hidden;zoom：1；

缺点： 里面尽肯能的不能有定位。 如果有定位，可能会切掉一部分。

清除浮动方法 5：利用 after 伪类清除浮动

12.行内块元素之间空白缝隙的问题:
解决方案：
(1.)利用 margin 负值，例如 Margin-left:-8px;
(2.)把行内块写到一行上去。
(3.)给父盒子加:font-size:0;
(4.)利用 js 来清除缝隙。

javascript ：

1.HTML 对象获取问题

FireFox：document.getElementById(“idName”);

ie:document.idname 或者 document.getElementById(“idName”).

解决办法：统一使用 document.getElementById(“idName”);

2.const 问题

说明:Firefox 下,可以使用 const 关键字或 var 关键字来定义常量;

IE 下,只能使用 var 关键字来定义常量.

解决方法：统一使用 var 关键字来定义常量.

3.event.x 与 event.y 问题

说明:IE 下,event 对象有 x,y 属性,但是没有 pageX,pageY 属性;

Firefox 下,event 对象有 pageX,pageY 属性,但是没有 x,y 属性.

解决方法：使用 mX(mX = event.x ? event.x : event.pageX;)来代替 IE 下的 event.x 或者 Firefox 下的 event.pageX.

4.window.location.href 问题

说明:IE 或者 Firefox2.0.x 下,可以使用 window.location 或 window.location.href;

Firefox1.5.x 下,只能使用 window.location.

解决方法：使用 window.location 来代替 window.location.href.

5.firefox 与 IE 的父元素(parentElement)的区别

IE：obj.parentElement

firefox：obj.parentNode

解决方法: 因为 firefox 与 IE 都支持 DOM,因此使用 obj.parentNode 是不错选择.

6.集合类对象问题

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

解决方法：统一使用 [] 获取集合类对象。

7.自定义属性问题

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

解决方法：统一通过 getAttribute() 获取自定义属性。

8.input.type 属性问题

问题说明：IE 下 input.type 属性为只读；但是 Firefox 下 input.type 属性为读写。

解决办法：不修改 input.type 属性。如果必须要修改，可以先隐藏原来的 input，然后在同样的位置再插入一个新的 input 元素。

9.event.srcElement 问题

问题说明：IE 下，even 对象有 srcElement 属性，但是没有 target 属性；Firefox 下，even 对象有 target 属性，但是没有 srcElement 属性。

解决方法：使用 srcObj = event.srcElement ?event.srcElement : event.target;

140、Electron

Electron 可以让你使用纯 JavaScript 调用丰富的原生 APIs 来创造桌面应用。你可以把它看作是专注于桌面应用而不是 web 服务器的，io.js 的一个变体。

这不意味着 Electron 是绑定了 GUI 库的 JavaScript。相反，Electron 使用 web 页面作为它的 GUI，所以你能把它看作成一个被 JavaScript 控制的，精简版的 Chromium 浏览器。

141、服务器端

1.服务器与客户端

1.1：服务器是一台装有特殊服务的电脑，该电脑上装有服务的相关软件，提供服务，这样

我们就可以把这台电脑叫服务器

2.2：通过服务器享受服务的电脑就叫客户端，比如我们在浏览百度，那么百度就叫服务器端，我们

此时用的电脑就叫客户端。

2.ip 每台计算机都有自己 ip 地址，并且是唯一的，互联网就是用网线

将电脑连接起来，域名就是 ip 对应的别名，为了方便用户记忆，dns 是

将域名对应上 ip 地址，端口是服务器中的门牌号，每个端口有自己特定的服务，比如 80

端口提供浏览器服务，3003 端口提供数据库服务。。。url 里默认端口就是 80

3.静态网页与动态网页的区别：

静态网页：通过 http 协议发送请求，直接会通过 web 服务器处理会直接返回，没有数据交互，开发技术，html，css，js

页面上的东西都是写死的，一成不变，可以直接在本地打开，files:// 的形式

动态网页：请求的是动态资源，通过 web 服务器交给后端 php，java。。进行处理之后，在

通过 web 服务器返回给客户端

4.为什么需要 web 服务器:

不管什么 web 资源，想被远程计算机访问，都必须有一个与之对应的网络通信程序，当用户来访问时，这个网络通信程序

会读取 web 资源请求，并把数据发送来来访者

5.单引号与双引号的区别：

php 中单引号不能够解析变量，最好用双引号。

6.打开百度 baidu.com，发生了哪些过程

1.输入正确得网址

2.通过 dns 找到域名对应得 ip 地址，现在本机 hosts 文件中找，如果没有，再去 dns 服务器找

3.找到之后，先是浏览器与服务器进行三次握手，①客户端向服务器发送一个连接请求②服务器向客户端返回一个确认信息

③客户端将请求及这个确认信息发送服务器，如果断开连接会进行四次握手，通过握手之后，通过 http 协议建立起连接

4.浏览器会向服务器发送请求

5.这时候会有报文得传输过程，浏览器发出请求报文，（请求行，请求头，请求体），服务器会通过解析等处理进行响应，会返回

响应行，响应头，响应体

6.浏览器接收了响应之后，会进行加载和渲染页面，打印绘制输出。

second_day:

1.get 请求与 post 请求区别

应用得场景不同:

get:一般常用在直接输入网址；src/href script/img/a from 表单中 method="get"

post:像论坛发帖子，注册账号，报名（个人照片上传，身份证）涉及到一些敏感数据

安全性:

get:参数会跟在 url 地址中？得后面以键值得形式用&隔开进行传递，相对来说很不安全

post:参数可以在请求头 headers 的 form-data 里看到数据

数据大小:

get:一般为 4kb 左右

post：理论上无限制，根据服务器的性能了

执行效率：

get 比 post 高

2.表单中 name 和 value 属性:

name 属性是必需的，后台获取数据，比如$_GET["name 的属性名"]来拿，而 value 是默认值,如果你没有输入数据，那么后台拿到的

数据就是 username=value

还有单选框配合 name 属性来用，选了男就不能选女

3.json 文件:

1.必需是双引号: "name":"wangliang"

2.整个文件时一个字符串，一般两种形式[]{}

3.不能写注释

4.相关操作的函数:json_encode(将数组转成 json 字符串)json_decode(将 json 字符串转成数组)

4.http 协议:

http 协议:是一种超文本传输协议,是一个基于 TCP/IP 通信来传递数据

特点： 1、简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。

由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。

2、灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。

3.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

4.无状态：HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。

5、支持 B/S 及 C/S 模式。

请求报文与响应报文:

请求报文:

get 方式的请求报文：

请求行：请求方式 url 地址 http 版本

请求头：一些相关的信息,Host:127.0.0.1 主机的服务器地址

user-agent：用户代理，展示浏览器一些相关的

信息

没有请求体！！！因为 get 方式请求，参数写在 url 上

post 方式的请求报文:

请求行：同 get

请求头：get 里所没有的，且很重要的一个属性：Content-Type:用来描述发送数据的编码格式

请求体:就是 get 请求的参数 username=123&psd=1241

响应报文:

get 和 post 方式的响应报文一样:

响应状态行:http 版本 200 ok(成功响应返回的) 状态码： + 200 成功找到

+ 404 找不到页面

+ 500 服务器出错了

+ 302 重定向

响应头:一些相关信息 Content-Type：规范当前返回值的类型 一般是 text/html

响应体:就是代码片段了

两种会话技术，cookie，session

会话：从客户端与服务器建立连接一直到连接断开，这个过程中会话一直存在。

为什么有 cookie 和 session：因为 http 协议是无状态的，没有记忆能力，而有时候我们需要浏览器记忆一些东西，比如，用户浏览一个网站

，每打开一个页面都要登录一次。

什么是 cookie 和 session：

cookie：通俗讲，是访问某些网站后在本地存储的一些网站相关信息，下次访问时减少一些步骤，是浏览器保持状态的一种方案

cookie：工作原理:

当用户第一次访问服务器时，客户端还没有存 cookie，这时在服务器端的表现是第一次来，这时，在响应头里向

客户端写入一个 cookie 值，第二次访问时，客户端会在请求头里携带 cookie 值，服务器端的表现为第二次来，能

检测到这个 cookie。

设置 cookie 函数:setcookie(参 1，参 2，参 3)参 1：键 参 2：值，参 3：有效时间（可选）如果没有设置参 3，则他的

生命周期为浏览器关闭就没了

删除 cookie，setcookie("名",值,time()-1)注意点：如何设置，就要如何删除，

session：工作原理:

当用户第一次请求一个登录页面,服务器返回响应，用户输入用户名密码，再次发送请求，这时候，服务器会做相应

的验证，通过后，服务器会将数据保存到临时文件，并生成一个 sessionid 以 cookie 的方式将这个 sessionid 返回给浏览器

第三次请求 main.php，会携带这个 sessionid，服务器拿到这个 id 做验证，通过之后，会根据这个 id 取出对应的数据，将

响应返回给用户

区别：

存放位置：cookie 存在浏览器端，session 存在服务器端

应用场景：

处于安全性的考虑，因为 cookie 在存在浏览器端的，用户可以访问到，并伪造 cookie，很不安全，而 session 存在服务器端

只向用户暴露出一个 id，这样安全性提高，因为安全性不同，所以应用场景也不一样，像涉及到敏感信息，会使用 session

生命周期：

cookie：如果没有设置时间，默认浏览器一关闭就没了，

session：在配置文件中，默认设置的时间是 24 分钟，浏览器一关闭就没了

存储：cookie 只能存储字符串类型的对象，session 可以存储任意对象

大小：cookie 大约存 4kb 左右，很多浏览器限定一个站点最多存放 20 个 cookie，而 session 理论上无限制，但是 session 是存在服务器

上的，如果太大，会给服务器带来很大压力。

142、交互体验、产品设计的的理解

用户体验设计最终在做的就是用户体验，其实在现在的社会里，人们说到底使用一个产品最终就是为了获得一种体验，比如用手机是为了获得可以和别人及时沟通的体验等等，手机本身如果什么体验都没有的话，那它的价值几乎是没有的，所以对于一个产品来说，好的设计理念，好的造型，材质等等所有的一切都是围绕用户体验来的，都是为了让用户用起来舒服，用起来体验好为最终目的的，所以软件也是一样，UED 做软件也是同样如此，这种情况下，单纯的交互是满足不了用户愈来愈严苛的要求的。

做交互首先要具备的就是逻辑，当然逻辑越强越好，交互对最终用户体验的影响是很大的，一个动效的形式，快慢等等，都有可能让用户体验变得很差或者很好，要考虑这个界面有什么东西，另一个界面有什么东西，两个界面怎么切换，手势改用什么样的手势

143、Js 常用排序

1. 冒泡排序

原理：从第一个元素开始，把当前元素和下一个索引元素进行比较。如果当前元素大，那么就交换位置，重复操作直到比较到最后一个元素

```
function bubbleSort(arr) {
     if (Array.isArray(arr)) {
       for (var i = arr.length - 1; i > 0; i--) {
         for (var j = 0; j < i; j++) {
           if (arr[j] > arr[j + 1]) {
             [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
           }
         }
       }
```

```
      return arr;
    }
  }
```

2.插入排序

原理：第一个元素默认是已排序元素，取出下一个元素和当前元素比较，如果当前元素大就交换位置。那么此时第一个元素就是当前的最小数，所以下次取出操作从第三个元素开始，向前对比，重复之前的操作。

```
function insertSort(arr) {
    if (Array.isArray(arr)) {
      for (var i = 1; i < arr.length; i++) {
        var preIndex = i - 1;
        var current = arr[i]
        while (preIndex >= 0 && arr[preIndex] > c) {
          arr[preIndex + 1] = arr[preIndex];
          preIndex--;
        }
        arr[preIndex + 1] = current;
      }
      return arr;
    }
  }
```

3.选择排序

原理：遍历数组，设置最小值的索引为 0，如果取出的值比当前最小值小，就替换最小值索引，遍历完成后，将第一个元素和最小值索引上的值交换。如上操作后，第一个元素就是数组中的最小值，下次遍历就可以从索引 1 开始重复上述操作。

```
function selectSort(arr) {
    if (Array.isArray(arr)) {
      for (var i = 0; i < arr.length - 1; i++) {
        var minIdex = i;
        for (var j = i + 1; j < arr.length; j++) {
          minIdex = arr[j] < arr[minIdex] ? j : minIdex;
        }
        [arr[i], arr[minIdex]] = [arr[minIdex], arr[i]];
      }
      return arr;
    }
  }
```

4.快速排序

原理：在数据集之中，找一个基准点，建立两个数组，分别存储左边和右边的数组，利用递归进行下次比较。

```
function quickSort(arr) {
```

```
    if (!Array.isArray(arr)) return;
    if (arr.length <= 1) return arr;
    var left = [], right = [];
    var num = Math.floor(arr.length / 2);
    var numValue = arr.splice(num, 1)[0];
    for (var i = 0; i < arr.length; i++) {
      if (arr[i] > numValue) {
        right.push(arr[i]);
      } else {
        left.push(arr[i]);
      }
    }
    return [...quickSort(left), numValue, ...quickSort(right)]
  }
```

5. 希尔排序

原理：

选择一个增量序列 t1，t2，……，tk，其中 ti > tj, tk = 1；
按增量序列个数 k，对序列进行 k 趟排序；
每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

```
function shellSort(arr) {
    var len = arr.length,
      temp,
      gap = 1;
    // 动态定义间隔序列，也可以手动定义，如 gap = 5;
    while (gap < len / 5) {
      gap = gap * 5 + 1;
    }
    for (gap; gap > 0; gap = Math.floor(gap / 5)) {
      for (var i = gap; i < len; i++) {
        temp = arr[i];
        for (var j = i - gap; j >= 0 && arr[j] > temp; j -= gap) {
          arr[j + gap] = arr[j];
        }
        arr[j + gap] = temp;
      }
    }
    return arr;
  }
```

6.归并排序

原理：

（1） 把长度为 n 的输入序列分成两个长度为 n/2 的子序列；
（2）对这两个子序列分别采用归并排序；
（3） 将两个排序好的子序列合并成一个最终的排序序列。

```
function mergeSort(arr) {  //采用自上而下的递归方法
    var len = arr.length;
    if (len < 2) {
      return arr;
    }
    var middle = Math.floor(len / 2),
      left = arr.slice(0, middle),
      right = arr.slice(middle);
    return merge(mergeSort(left), mergeSort(right));
  }

  function merge(left, right) {
    var result = [];
    while (left.length && right.length) {
      // 不断比较 left 和 right 数组的第一项，小的取出存入 res
      left[0] < right[0] ? result.push(left.shift()) :
result.push(right.shift());
    }
    return result.concat(left, right);
  }
```

js十大排序算法

排序算法说明:

(1) 对于评述算法优劣术语的说明

稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；
不稳定：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；

内排序：所有排序操作都在内存中完成；
外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；

时间复杂度：一个算法执行所耗费的时间。
空间复杂度：运行完一个程序所需内存的大小。

(2)排序算法图片总结:

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

144、为什么选择 mpvue 开发小程序？

mpvue 是一款使用 Vue.js 开发微信小程序的前端框架。使用此框架，开发者将得到完整的 Vue.js 开发体验，同时为 H5 和小程序提供了代码复用的能力。如果想将 H5 项目改造为小程序，或开发小程序后希望将其转换为 H5，mpvue 将是十分契合的一种解决方案。mpvue 是一套定位于开发小程序的前端开发框架，其核心目标是提高开发效率，增强开发体验。使用该框架，开发者只需初步了解小程序开发规范、熟悉 Vue.js 基本语法即可上手。框架提供了完整的 Vue.js 开发体验，开发者编写 Vue.js 代码，mpvue 将其解析转换为小程序并确保其正确运行。此外，框架还通过 vue-cli 工具向开发者提供 quick start 示例代码，开发者只需执行一条简单命令，即可获得可运行的项目。
在小程序内测之初，我们计划快速迭代出一款对标 H5 的产品实现，核心诉求是：快速实现、代码复用、低成本和高效率... 随后经历了多个小程序建设，结合业务场景、技术选型和小程序开发方式，我们整理汇总出了开发阶段面临的主要问题：
组件化机制不够完善
代码多端复用能力欠缺
小程序框架和团队技术栈无法有机结合
小程序学习成本不够低
组件机制：小程序逻辑和视图层代码彼此分离，公共组件提取后无法聚合为单文件入口，组件需分别在视图层和逻辑层引入，维护性差；组件无命名空间机制，事件回调必须设置为全局函数，组件设计有命名冲突的风险，数据封装不强。开发者需要友好的代码组织方式，通过 ES 模块一次性导入；组件数据有良好的封装。成熟的组件机制，对工程化开发至关重要。
多端复用：常见的业务场景有两类，通过已有 H5 产品改造为小程序应用或反之。从效率角度出发，开发者希望通过复用代码完成开发，但小程序开发框架却无法做到。我们尝试过通过静态代码分析将 H5 代码转换为小程序，但只做了视图层转换，无法带来更多收益。多端代码复用需要更成熟的解决方案。
引入 Vue.js：小程序开发方式与 H5 近似，因此我们考虑和 H5 做代码复用。沿袭团队技术栈选型，我们将 Vue.js 确定为小程序开发规范。使用 Vue.js 开发小程序，将直接带来如下开发效率提升：
H5 代码可以通过最小修改复用到小程序
使用 Vue.js 组件机制开发小程序，可实现小程序和 H5 组件复用
技术栈统一后小程序学习成本降低，开发者从 H5 转换到小程序不需要更多学习。

145、Vue 跳转路由如何传递一个对象过去？

// 跳转路由传递对象参数
Var arr=JSON.stringify(this.songList)
this.$router.push('/shop/'+encodeURIComponent(arr))
这里跳转路由的时候，先用 JSON.stringify 将参数转换一下，有人会将转换的参数直接传递过去，然后在那边接收的时候用 JSON.parse 会报错，这里用 JSON.stringify 转换完毕后再用 encodeURIComponent()将参数再次转换一下，然后就可以传递了。
// 获取传过来的参数

var list = decodeURIComponent(this.$route.params.obj); this.songList = JSON.parse(list);
这里接收参数的时候，先用 decodeURIComponent()将传递过来的参数转换一下，然后再用 JSON.parse 再次转换，这样，一个对象就完整的传递过来了，然后可以开心的使用各种参数，不用再次去请求数据了。 但是上传文件的文件流数据 file 用这个方法不好使，file 数据不能用 JSON 来转换，一转换就为空了，所以也不能用本地存贮了，可以使用 vuex 来保存。

146、Vue 中 route 和 router 的区别？

1.router 是 VueRouter 的一个对象，通过 Vue.use(VueRouter)和 VueRouter 构造函数得到一个 router 的实例对象，这个对象中是一个全局的对象，他包含了所有的路由包含了许多关键的对象和属性。

举例：history 对象

$router.push({path:'home'});本质是向 history 栈中添加一个路由，在我们看来是 切换路由，但本质是在添加一个 history 记录

方法：

$router.replace({path:'home'});//替换路由，没有历史记录

2.route 是一个跳转的路由对象，每一个路由都会有一个 route 对象，是一个局部的对象，可以获取对应的 name,path,params,query 等

$route.path
字符串，等于当前路由对象的路径，会被解析为绝对路径，如 "/home/news" 。

$route.params
对象，包含路由中的动态片段和全匹配片段的键值对

$route.query
对象，包含路由中查询参数的键值对。例如，对于 /home/news/detail/01?favorite=yes ，会得到$route.query.favorite == 'yes' 。

$route.router
路由规则所属的路由器（以及其所属的组件）。

$route.matched
数组，包含当前匹配的路径中所包含的所有片段所对应的配置参数对象。

$route.name
当前路径的名字，如果没有使用具名路径，则名字为空。

$route.path, $route.params, $route.name, $route.query 这几个属性很容易理解，主要用于接收路由传递的参数

147、Vue 中 router 跳转和 location.href 有什么区别？

1、使用 location.href='/url'来跳转，简单方便，但是刷新了页面。

2、使用 history.pushState('/url')，无刷新页面，静态跳转。

3、引进 router，然后使用 router.push('/url')来跳转，使用了 diff 算法，实现了按需加载，减少了 dom 的消耗。

其实使用 router 跳转和使用 history.pushState()没什么差别的，因为 vue-router 就是用了 history.pushState()，尤其是在 history 模式下。

148、Webpack 什么插件可以解决 css 兼容性问题？

css 兼容性处理：postcss --> postcss-loader postcss-preset-env
1.下载 postcss-loader postcss-preset-env 两个插件
npm i postcss-loader postcss-preset-env -D
2.webpack.config.js 文件内配置
3.在 package.json 配置 browserslist，帮 postcss 找到 package.json 中 browserslist 里面的配置，通过配置加载指定的 css 兼容性样式

149、Window.onload 和 jquery 里的${document}.ready 有什么区别？

1、执行时间：
window.onload 必须等到页面内包含图片的所有元素加载完毕后才能执行，所以如果页面的图片很多，则等待的时间可能越长。
但是 ready()是 DOM 结构绘制完毕后就执行，不必等到资源加载完毕后才执行，不过也有弊端，DOM 结构加载完毕，很有可能资源图片没有下载结束，这时候去读取图片的宽高不一定能读取的到。
2、编写个数不同：
window.onload 不能同时编写多个，如果有多个 window.onload 方法，只会执行最后一个。因为会覆盖。
$(document).ready()可以同时编写多个，并且都可以得到执行。因为 Jquery 对事件绑定做了封装处理。
简化写法：window.onload 没有简化写法，但是
$(document).ready(function(){})可以简写成$(function(){})。

150、http2 相较于 http1 有什么改进？

（1）单连接多资源的方式，减少服务端的链接压力，内存占用更少，链接吞吐量更大；
（2）由于 TCP 链接的减少而使网络拥塞状况得以改善，同时慢启动时间的减少，使拥塞和丢包回复速度更快；
（3）引入了“服务端推（server push）”的概念，允许服务端在客户端需要数据之前就主动将数据发送到客户端缓存，从而提高性能；
（4）HTTP2 提供了更多加密支持；
（5）增加了头压缩（header compression），因此即使非常小的请求，其请求和相应的 header 都只会占用很小的带宽。

151、Webpack 里的 sourcemap

SourceMap 一个存储源代码与编译代码对应位置映射的信息文件

152、http 请求头

· Accpet
· 告诉服务端,客户端接收什么类型的响应
· Referer
· 表示这是请求是从哪个 URL 进来的,比如想在网上购物,但是不知道选择哪家电商平台,你就去问度娘,说哪家电商的东西便宜啊,然后一堆东西弹出在你面前,第一给就是某宝,当你从这里进入某宝的时候,这个请求报文的 Referer 就是 www.baidu.com
· Cache-Control
· 对缓存进行控制,如一个请求希望响应的内容在客户端缓存一年,或不被缓可以通过这个报文头设置
· Accept-Encoding
·
这个属性是用来告诉服务器能接受什么编码格式,包括字符编码,压缩形式(一般都是压缩形式)

例如:Accept-Encoding:gzip, deflate(这两种都是压缩格式)
· Host
· 指定要请求的资源所在的主机和端口
· User-Agent 作用：告诉服务器，客户端使用的操作系统、浏览器版本和名称

153、WebSocket

目的：即时通讯，替代轮询
应用场景：网站上的即时通讯是很常见的，比如网页的 QQ，聊天系统等。按照以往的技术能力通常是采用轮询、Comet 技术解决。
原理：WebSocket 同 HTTP 一样也是应用层的协议，但是它是一种双向通信协议，是建立在 TCP 之上的。

连接过程 —— 握手过程
1. 浏览器、服务器建立 TCP 连接，三次握手。这是通信的基础，传输控制层，若失败后续都不执行。
2. TCP 连接成功后，浏览器通过 HTTP 协议向服务器传送 WebSocket 支持的版本号等信息。（开始前的 HTTP 握手）
3. 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。
4. 当收到了连接成功的消息后，通过 TCP 通道进行传输通信。
WebSocket 与 HTTP 的关系
相同点
都是一样基于 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 是需要握手进行建立连接的。
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。
WebSocket 与 Socket 的关系
Socket 其实并不是一个协议，而是为了方便使用 TCP 或 UDP 而抽象出来的一层，是位于应用层和传输控制层之间的一组接口。Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket 其实就是一个门面模式，它把复杂的 TCP/IP 协议族隐藏在 Socket 接口后面，对用户来说，一组简单的接口就是全部，让 Socket 去组织数据，以符合指定的协议。当两台主机通信时，必须通过 Socket 连接，Socket 则利用 TCP/IP 协议建立 TCP 连接。TCP 连接则更依靠于底层的 IP 协议，IP 协议的连接则依赖于链路层等更低层次。WebSocket 则是一个典型的应用层协议。Socket 是传输控制层协议，WebSocket 是应用层协议。

154、浅拷贝和深拷贝的区别

深拷贝和浅拷贝最根本的区别在于是否真正获取一个对象的复制实体，而不是引用。
假设 B 复制了 A，修改 A 的时候，看 B 是否发生变化：
如果 B 跟着也变了，说明是浅拷贝，拿人手短！（修改堆内存中的同一个值）
如果 B 没有改变，说明是深拷贝，自食其力！（修改堆内存中的不同的值）
浅拷贝（shallowCopy）只是增加了一个指针指向已存在的内存地址，
深拷贝（deepCopy）是增加了一个指针并且申请了一个新的内存，使这个增加的指针指向这个新的内存，
使用深拷贝的情况下，释放内存的时候不会因为出现浅拷贝时释放同一个内存的错误。

浅复制：仅仅是指向被复制的内存地址，如果原地址发生改变，那么浅复制出来的对象也会相应的改变。
深复制：在计算机中开辟一块新的内存地址用于存放复制的对象。
浅拷贝实例：

```
//此递归方法不包含数组对象
var obj = { a:1, arr: [2,3] };
var shallowObj = shallowCopy(obj);

function shallowCopy(src) {
  var newobj = {};
  for (var prop in src) {
    if (src.hasOwnProperty(prop)) {
      newobj[prop] = src[prop];
    }
  }
  return newobj;
}
```

因为浅复制只会将对象的各个属性进行复制，并不会进行递归复制，而 JavaScript 存储对象是存地址的，所以浅复制会导致 Obj.arr 和 shallowObj.arr 指向同一块内存地址：
导致的结果就是：

```
shallowObj.arr[1] = 5;
console.log(obj.arr[1]);       //5
```

深拷贝实例：

```
var obj = {
a:1,
arr: [1,2],
nation : '中国',
birthplaces:['北京','上海','广州']
};
var obj2 = {name:'杨'};
obj2 = deepCopy(obj,obj2);
console.log(obj2);
//深复制，要想达到深复制就需要用递归
function deepCopy(o, c){
var c = c || {};
for(var i in o){
if(typeof o[i] === 'object'){
if(o[i].constructor === Array){
//这是数组
c[i] = [];
}else{
//这是对象
c[i] = {};
}
deepCopy(o[i], c[i]);
}else{
c[i] = o[i];
```

```
}
}
return c;
}
```

而深复制则不同，它不仅将原对象的各个属性逐个复制出去，而且将原对象各个属性所包含的对象也依次采用深复制的方法递归复制到新对象上。这就不会存在 obj 和 shallowObj 的 arr 属性指向同一个对象的问题。

155、