2020/11/10 笔试面试题-2

1、水平垂直居中

2、屏幕分辨率计算

3、rem 和 em、px 区别？

4、右边宽度固定，左边自适应，编写 CSS 代码

5、js 原型继承的几种方式

6、a 标签的几种状态？

7、怎么理解盒模型？（这是整理过的用这个）

8、回流，重绘

9、M-V-VM（Model-View-ViewModel）

10、vue 中双向数据绑定(v-model)的原理

11、vue 中使用 sass 的配置什么

12、12、axios 封装、Promise 原理、async　await、Promise 与 asynce await 区别

13、Vuex 属性

14、在 jquery 中想要找到所有元素的同辈元素

15、http 请求出现“请求的对应资源禁止被访问”的问题

16、在 vue 中，第一次加载时会触发哪几个钩子

17、vue-router 有哪些组件

18、关于 vue 的正确说法

19、synce

20、vuex 相关与 localtorage 区别

21、数组循环几种区别

22、闭包

23、vue 优势

24、Event Loop&宏任务微任务

25、什么是防抖与节流

27、_proto_与 prototype 区别关系，作用域和作用域链，原型与原型链

28、用 css 画三角形

29、写一段 VUE 原码里的代码

30、computed 与 watch 区别使用场景

31、keep-alive 几个钩子函数

32、$nextTick

33、http 与 https 区别（原理）

34、jquery 与 vue 相比有什么不同

35、Vue 中的 diff 算法?

36、回流，重绘

37、call 与 apply 与 bind 区别

38、用过视频 video 兼容问题

39、箭头函数和普通函数区别

40、css3 用过哪些属性:box-shadow 和 ransition 分别有哪些属性值

41、css 如何选择 ul 下的第 50 个 li

42、代码实现把一个 span 的 click 事件，委托给父级 div#abc（原生 js，不用考虑兼容性）

44.、简述 Vue 的响应式原理

45、Vue 中给 data 中的对象属性添加一个新的属性时会发生什么，如何解决？

46、axios 是什么？怎么使用？描述使用它实现登录功能的流程？

47、Vue 组件中 data 为什么必须是函数？

1、如水平垂直居中:

（1）用 position 和负边距：父节点相对定位。子节 position:absolute;top:50%;margin-身高度的一半;（2）多行文本居中： vertical-align: middle;display: table-cell;（3）文本居中 line-height

Display:flex 弹性布局”，用来为盒子模型提供最大的灵活性;

CSS 水平垂直居中常见方法总结

1、文本水平居中

line-height,text-align:center(文字)

元素水平居中 margin:0 auo

方案 1：position 元素已知宽度

父元素设置为：position: relative;

子元素设置为：

position: absolute; left: 50%;top: 50%;margin: -50px 0 0 -50px;

margin 各减去上下距离的一半

方案 2：position transform 元素未知宽度

子元素：margin: -50px 0 0 -50px;替换为：transform: translate(-50%,-50%);

方案 3：flex 布局

父元素加：

display: flex; //flex 布局

justify-content: center; //使子项目水平居中

align-items: center; //使子项目垂直居中

2、屏幕分辨率计算

当前屏幕宽：document.documentElement.clientWidth

html 代吗为 document.documentElement

2、rem 和 em、px 区别?

适配的几种方法

- 使用 px，结合 Media Query 进行阶梯式的适配；
- 使用%，按百分比自适应布局；
- 使用 rem，结合 html 元素的 font-size 来根据屏幕宽度适配；
- 使用 vw、vh，直接根据视口宽高适配。

Em：EM 是相对于父元素来设计字体大小 需要转换的像素值 / 父元素的 font-size s= em 值

如父 font-size:24px, 子 30px 等于 1.25em = 30/24

如父 font-size:20px, 子 width:200px 等于 10em = 200/20

通过设置父级字号其他不好使来定子级 em 值

1. body 选择器中声明 Font-size=62.5%;
2. 2. 将你的原来的 px 数值除以 10，然后换上 em 作为单位；*12px=1.2em,*

*注*意：任意浏览器的默认字体高都是 16px。所有未经调整的浏览器都符合：1em=16px。body 选择器中声明 Font-size=62.5%相于于 10px，这样 12px=1.2em, 10px=1em, 也就是说只需要将你的原来的 px 数值除以 10，然后换上 em 作为单位就行了。

Rem：https://www.cnblogs.com/dannyxie/p/6640903.html

```
<script type="text/javascript">
    document.documentElement.style.fontSize = document.documentElement.clientWidth / 640*100 + 'px';
</script>
```

问题：为什么是 clientWidth/640?为什么要乘以 100?

1.是因为这里是作为一个基础数值，换个方向去想，这里先不乘以 100 以免产生误解。

例如：设计稿宽度是 640px，设备屏幕宽度是 320px,rem=320/640=0.5px 是 html {font-size:0.5px} 但在不设置时默认 html 是 16px ，（320/640）*32 =16

如 640px 宽就是 html{font-size:16px;} 那么 640px 的稿就是 640/32=20rem;

2.一般浏览器的最小字体是 12px，如果 html 的 font-size=（320/640）px，相当于 font-size=0.5px，那么这个数值就小于 12px，会造成一些计算的错误，和一些奇怪的问题，*100 后，font-size 是 50px，就可以解决这种字体小于 12px 的问题。

3. 为了计算方便　我们后面把比率乘以了 100，（320/640）*100，那么相对应这个元素在设置数值的时候就需要除以 100 了（640/100）,这样可以保证最后出来的数值不变.

实际证明 （document.documentElement.clientWidth=375/414/320 ）*100 后计算 rem 与屏幕宽没关系，*100 后与设计稿宽有关都是用像素值除以 100 就是要的 rem 值，如设计稿是 750，页面宽就是 7.5rem,如设计稿是 640，页面宽就是 6.4rem

附如设计稿宽度是 750px，设备屏幕宽度是 414px，rem=414/750=0.552px 是 html{font-size:0.552px}但在不设置时默认 html 是 16px ，（414/750）*29 =16

如 750px 宽就是 html{font-size:16px;} 那么 750px 的稿就是 750/29=25.86rem;

总：默认 16px 情况下：需要转换值÷16=rem 值是不对的，这根据屏幕宽与设计稿宽有关，在开发中要动态获取计算

px 稳定和精确。问题就是缩放页面时布局会打破

适配各种移动设备，使用 rem

```
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
 user-scalable=0" />
<meta http-equiv="cache-control" content="max-age=0" />
<meta http-equiv="cache-control" content="no-cache" />
```

支持横竖屏切换改字号

```
//rem 自动计算
(function (doc, win) {
 var docEl = doc.documentElement,
 resizeEvt = 'orientationchange'in window?'orientationchange':'resize',
 recalc = function () {
            var clientWidth = docEl.clientWidth;
            if (!clientWidth) return;
            docEl.style.fontSize = 100 * (clientWidth / 750) + 'px';
 };
   if (!doc.addEventListener) return;
   win.addEventListener(resizeEvt, recalc, false);
   doc.addEventListener('DOMContentLoaded', recalc, false);
})(document, window);
```

3、右边宽度固定，左边自适应，编写 CSS 代码

（1）

```
<div class='wraper'>
    <div class="left">自适应区</div>
    <div class="right">固定宽度区</div>
</div>
.wraper {display: table;}
.left { display: table-cell;border: 1px solid #00f;width: 100%;}
.right {width: 200px;border: 1px solid #f00;display:table;}
```

或：

```
.wraper { display: flex;}
.left {border: 1px solid #00f;flex: 1;}
.right { width: 200px;border: 1px solid #f00;}
```

4、js 原型继承的几种方式

```
var AClass = function(){
      this.id=123;
}
var BClass=function(id){
     AClass.call(this,id)
}
var bInstance = new BClass;
BClass.prototype.sayId=function(){
    console.log(this.id)
}
bInstance.sayId();
```

在 JavaScript 中，继承都是源于原型，有多种实现方式有 call 的都是在子类里写的

1、原型链继承：重点圈起来：将父类实例赋值给子类原型对象

```
function Parent() {
    this.name = "a";
}
function A() {
    this.age = "20";
}
//将父类实例赋值给子类原型对象 A 继承了 Parent,通过原型，形成链条
A.prototype = new Parent();
// 将 constructor 指向本身，保证原型链不断。
A.prototype.constructor = A;
var a = new A();
console.log(a.name)//结果：a
```

2、构造继承：重点圈起来：执行父构造，将 This 指向本身，拉取父私有属性

```
// 父级
function Parent(name) {
    this.name = name;
```

```
}
// 创建子类、添加子类属性。
function A(name) {
    Parent.call(this, name) // 执行父构造，将 This 指向本身，拉取父私有属性；
}
A.prototype.getName = function () {
    console.log('我叫' + this.name)
}
var a = new A('李四');
console.log(a.name)  //李四
a.getName()          //
我叫李四
```

3、组合继承

重点圈起来：组合继承（原型链
继承+构造函数继承）

用 Parent.call()也用

Parent 函数中如 this.age=12,结果 age=12,父类没有取自己的，自己没有 undefined

```
// 父
function Parent(age) {
    this.name = ['Lee', 'Jack', 'Hello']
    this.age = age;
}
Parent.prototype.run = function () {
    return this.name + this.age;
};
// 子
function A(age) {
    Parent.call(this, age); //对象冒充 子类 A 中的 age 指向 Parent 里的
}
A.prototype = new Parent(); //原型链继承
var a = new A(100);    //Parent 函数中如 this.age=12,结果 age=12,父类没有取自己的，自己没有 undefined
console.log(a.run());//结果：Lee,Jack,Hello100
```

4、寄生组合继承

重点圈起来：将父类原型对象直接赋值给一个空属性的构造函数，再将空属性的构造函数实例赋值给子类原型对象,其根本是为了解决父实例继承的出现的两次构造。

https://blog.csdn.net/qq_42926373/article/details/83149347

https://www.cnblogs.com/luoguixin/p/6195902.html

6、a 标签的几种状态？

a:link 普通的、未被访问的链接

a:visited 用户已访问的链接

a:hover 鼠标指针位于链接的上方

a:active 链接被点击的时刻

7、怎么理解盒模型？（这是整理过的用这个）

Box-sizing:content-box/border-box

标准模型：最终元素的大小为=设置的 width+padding+border ,width=content（(默认是 content-box)

如:content 的 width 是 100px，padding:20px;border:10px,最后元素大小=100+60 是 160px

IE 盒子模型：**border-box** ：最后元素大小=width， padding 和 border 在 width 里边缩

最后元素大小=width = content+padding+border,content<=width

怎么让 div 的红色背景#ff0000 半透明，但 div 里文字正常

```
background: rgba(255, 0, 0, 0.2)
```

8、回流，重绘

回流：一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建。这就称为回流(reflow)。

重绘：当一些元素需要更新属性，而这些属性只是影响元素的外观，风格，而不会影响布局的，比如 background-color。则就叫称为重绘。

9、M-V-VM（Model-View-ViewModel）

M（Model）模型：数据保存—存放着各种数据，有的是 data 里的属性，有的是从后端返回的数据

V（View）视图 ：用户界面，也就是 DOM。即 template 或 html 写页面部分也就是 UI 效果

VM（View-Model）视图模型:连接 View 和 Model 的桥梁，当数据变化时，ViewModel 够监听到数据的变化（通过 Data Bindings），自动更新视图，而当用户操作视图，ViewModel 也能监听到视图的变化（通过 DOM Listeners），然后通知数据做改动，这就实现了数据的双向绑定。

也是 vue 的实例对象，充当一个 UI 视配器的角色。也就是说 view 中每一个 UI 元素，都应该在 ViewModel 中找到与之相对应的属性。把 data 里的数据给到 html 页面渲染当中实现双向数据绑定。

为什么用 mvvm:

统一团队的开发标准，框架层面上清晰，代码层面干净

MVVM 的实现方式

更新数据方式通常做法是 `vm.set('property', value)`。

`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，触发相应的监听回调。

10、vue 中双向数据绑定(v-model)的原理

本质是 value 和 v-on 的结合体,就是绑定他的 value,通过 v-on 触发实现双向绑定，依赖于 Object.defineProperty(),通过这个函数可以监听到 get,set 事件，然后用他的 set,get 方法来通知订阅者，触发 update 方法，从而实现更新视图

11、vue 中使用 sass 的配置什么比 less 多配置了 node-sassss

运行如下命令

```
npm install --save-dev sass-loader
npm install --save-dev node-sass
```

build 文件夹下的 webpack.base.conf.js 添加如下代码：

```
{
  test: /\.sass$/,
  loaders: ['style', 'css', 'sass']
}
```

修改 style：`<style lang="scss">`

12、axios 封装、Promise 原理、async await、Promise 与 asynce await 区别

ajax 封装：

```
var methods = {
    //全站 ajax 请求状态处理
    ajax: function (url, data, successCallback, errorCallBack) {
        $.ajax({
            "type": "post",
            "url": AJAXURL + url,
            "async": true,
            "data": data,
            "success": function (res) {
                //返回状态处理
                if (res.code == 200) {
                    if (successCallback) {
                        successCallback(res);
                    }
                } else if (res.code == 406) {
                    //跳转至登录
                    methods.toLogin();
                } else {
                    if (errorCallBack) {
                        errorCallBack(res)
                    } else {
                        methods.prompt(res.msg);
                    }
                }
            }
        });
    },
}
```

axios 全局封装

```
Vue.prototype.$axios = function (config) {
    return new Promise((resolve, reject) => {
        axios({
            method: config.method ? config.method : "post",
            url: config.url ? config.url : "",
            data: config.data ? config.data : ""
        }).then(res => {
            console.log(config.data)
            resolve(res.data);
        }).catch(error => {
```

```
            // 请求失败，
            reject(error.data);
        });
    })
}
var vm = new Vue({
    el: '#app',
    data: {},
    created() {
        this.$axios({
            url: 'https://mall.faw-vw.com/Dealer/Index/shop',
            data: {
                dealer_id: 969
            }
        }).then(res => {

        });
    }
})
```

Promise 原理:

promise 是构造函数，通过 new 关键字实例化对象，代表某个将要发生的事情（通常是异步操作）

异步操作以同步的形式表示出来，解决回调地狱问题，

Promise 是个函数接收两个参数，resolve 成功，reject 失败，返回 promise 对象

Promise 的两个属性 state ，result，

Promise 三种状态：pending 准备），fullfilled 成功、rejected 失败，Promise 状态的改变指向实例对象

Promise 有隐示属性_proto_原型上有 catch then 方法,可以访问，then 方法是一个函数有两个参数，参数都是是函数，第一个表 resolve 成功回调，第二个失败回调

支持链式操作，

promise 的状态不改变时为 pending，不会执行 then 方法

Promise 调 resolve()，then 里 return 可以将 t 实例状态改成 fulfilled 成功状态

promise 出错时调用两种写法，then 第二个参数或 catch 方法，推荐用 catch

catch 三种情况下会调用:reject，报错时，throw new Error

promise 封装

```
function getData(url, data) {
    return new Promise((resolve, reject) => {
     $.ajax({
         url: url,
         type: 'GET',
         data: data ? data : {},
         success: function (res) {
             resolve(res)
```

```
        },
        error: function (err) {
            reject(err);
        }
     })
    })
}
let p = getData('data1.json').then(data => {
  let {id} = data;
  console.log(id)
  // return 的 promise 下个 then
  return getData('data2.json')
}).then(data => {
  let {username} = data;
  console.log(username)
  return getData('data3.json')
}).then(data => {
    let {email} = data;
    console.log(email)
})
```

async await

async 是异步的简写，await 是 async await 的简写，所以 async 用于声明一个异步的 function，而 await 用于等待一异步方法执行完成，

简单理解：

async 用于声明异步方法，返回 Promise

await 等待这个异步方法执行完成，await 必须用于 async 异步方法里

async await 代码读起来更像同步代码，所有的 async 都有 promise 的返回值，

```
async function test() {
    return new Promise((resolve, reject) => {
     setTimeout(() => {
         var name = 'yjui';
         resolve(name);
     }, 1000)
    })
}
//获取上边 async 异步方法里的数据所以用时下边方法必须也是异步方法
async function main() {
//等待上边 async 异步方法执行完返回 data 值，不加 await 是 promise，加 await 是值
  let data = await test(); //用时一句话就搞定异步执行
  console.log(data)
}
main() //yjui
```

Promise 与 aynce await 区别

它们都是异步编程的解决方式，Promise 解决传统 callback 函数回调地或问题，Promise 有隐示属性_proto_原型上有 catch then 方法，所以支持链式操作，调用时用 then 的方式，Promise 有 3 种状态，一旦执行 Promise 状态值不能改变，遇到复杂业务逻辑 Promise 要一直.then 不是很美观。像一个新的回调地域，精髓是状态传递。

Aynce await 是基于 Promise 实现的基于 Generator 函数语法糖，拥有内置执行器返回 Promise 对象，可以异步看起来像同步一样，更方便阅读和理解代码解决了 Promise 里不停 then 的问题，区别 async await 需要 try...catch 去捕获异常，Promise 更多用在函数封装中，async 用在函数使用中。

fetch 用法:

加 headers 给服务器看的不加解析不对

往服务器传数据必须是字符串所有 body 要 JSON.stringify(json 数据)

不带参数的请求

```
fetch('/fetch_test.php')
    .then(response => {// response 为 HTTP 响应
     return response.json() // 将 HTTP 响应转为 json 数据
    }).then(data => {
       console.log(data) // 打印 json 数据
    }).catch(error => {
     // 捕获异常
      console.log(error)
    })
},
```

带参数的请求(以 post 为例)

```
fetch('/fetch_getuser_test.php', {
  body: JSON.stringify({ id: 666 }), //这里是要传递的参数
  headers: {
  //添加头文件
  'content-type': 'application/json' //必加，指定数据类型和编码,
  'Authorization': 'Bearer ' + localStorage.access_token//按需选择，在请求信息中添加
assess_token 验证码,
    },
  method: 'POST'
}).then(response => response.json()) //返回的 res 为原生 Promise 对象,需要转换
.then(data => {
console.log(data)
})
```

线程和异步方法的区别　　（异步应该去跟同步比较才对。）

13、单线单线程与多线程的区别

单线程：只有一个线程，代码顺序执行，容易出现代码阻塞（页面假死）

多线程程序：有多个线程，线程间独立运行，能有效地避免代码阻塞，并且提高程序的运行性能

在 jquery 中想要找到所有元素的同辈元素 siblings()

14、http 请求出现“请求的对应资源禁止被访问”的问题，请问返回状态码是（403）400 语法错误，401 请求要求身份认证，403 拒绝执行此请求，404 找不到资

15、向上取整　Math.ceil(25.5) 26

四舍五入 Math.round(25.4）　25

向下取整 Math.floor(25.9)　25

16、在 vue 中，第一次加载时会触发哪几个钩子：beforeCreate, created, beforeMount, mounted 。

附：vue 生命周期：

beforeCreate、created 创建 初始化数据事件、beforeMount、mounted 载入 DOM 渲染完成、beforeUpdate、updated 更新、beforeDestroy、destroyed 销毁

17、vue-router 有哪些组件：

<router-link :to='' class='active-class'>　　//路由声明式跳转 ，active-class 是标签被点击时的样式

<router-view>　　//渲染路由的容器

<keep-alive>　　//缓存组件

18、判断

1、关于 vue 的正确说法：

A)V-show 通过修改元素不 display css 属性让其显示或隐藏　正

B)Vue.js 双向绑定是采用数据劫持结合发布-订阅的方式，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，触发相应的监听回调。 正

C)<style scoped>　让 css 只在当前组件起作用　局部　正

D)v-bind 作用是动态地绑定一个或多个特性，或一个组件 prop 到表达式　（改成正确语法不是 v-model）

v-model 一个组件上的 v-model 默认会利用名为 value 的 prop 和名为 input 的事件 正

2、关于 vuex 说法正确的：

A)getters 可以对 state 进行计算操作，它就是 Store 的计算属性 正

B)Vuex 的 mutation 特性可以直接变更 state 里的数据（状态），是同步操作　正

C)state 里面存放的数据是非响应式的，Vue 组件从 store 中读取数据。若是 store 中的数据发生改变，依赖这个数据的组件不会发生更新　　错

D)提交 mutation 是更改状态的唯一方法，并且这个过程是异步的（错，在 dispatch 之前可以 this.$store,dispatch()方法更改，strict：true 严格下也能改。mutation 同步）

3、关于 webpack 说法：

A)使用 Happypack 实现多线程加速编译　　正（这是提高 webpack 的构建速度方式，要注意的第一点是，它对 file-loader 和 url-loader 支持不好，所以这两个 loader 就不需要换成 happypack 了，其他 loader 可以类似地换一下）

B)使用 html-webpack-plugin 为 html 文件中引入的外部资源，可以生成创建 html 入口文件　　正

C)通过 externals 配置来提取常用库 正（这是提高 webpack 的构建速度方式）

D)Delete-webpack-plugin：打包器添理源目录文件，在 webpack 打包器清理 dist 目录　错（改说的是 clean-webpack-plugin）

4、v-on 可以监听多个方法　（正）

如：<input type="text" :value="name" @input="onInput" @focus="onFocus" @blur="onBlur" />

5、在 vuex 中，Action 提交的是 mutation，而不是直接变更状态，Action 可以包含任意异步操作　　（正）

6、jquery 的 load()方法中 data 参数是必须的（错）

`$(selector).load(url,data,function(response,status,xhr))` ，*url 必须，其他可选*

7、<router-link/>设置 replace 属性的话，当点击时，导航后不会留下 history 记录（正）

设置 replace 属性的话，当点击时，会调用 router.replace() 而不是 router.push()，于是导航后不会留下 history 记录。：<router-link :to="{ path: '/abc'}" replace></router-link>

15、webpack 的 babel-loader 是将 ES5 转成 ES6(错，改是配合 babel 将 es6 转成 es5)

19、synce

.synce 用于实现双向绑定，vue 的 prop 单向下行绑定，父级 prop 更新会向下流动到子组件中，反过来不行，有时需要对 prop 进行双向绑定，这时用 synce，和 v-model 差不多，在它的基础上扩展要多少有多少，v-model 一个标签上就一个

20、vuex 相关与 localtorage 区别

Vuex 实现了一个单向数据流 actions->mutations->state

Vue Components：是 vue 组件，只能读 State 不是写，写改写唯一办法是通过 Dispatch 间接调用

this.$store.dispatch(调用哪个 action，值)一般在 methods 中调，使用{{$store.state}}

Actions：异步，mutations 不能直接调用，想调用 mutations 要先调用 action 通过 commit 调用 mutations 完成复杂操作[功能]，比如登录，找服务器请求数据这种操作都是异步操作通过 actions

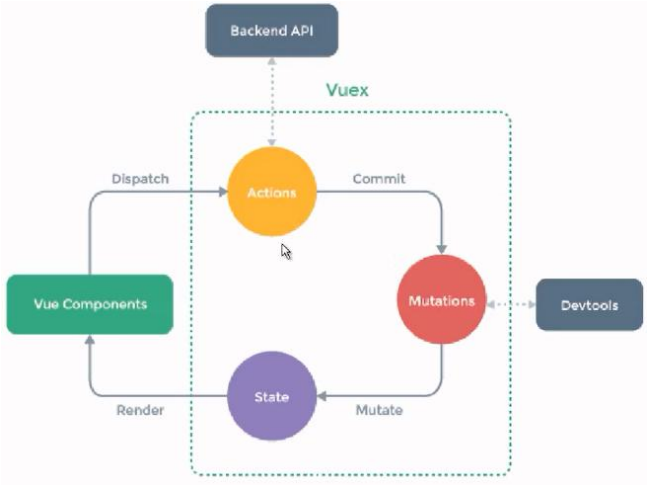
Mutations：同步，mutations 专门修改 state,

state：全局存储状态数据 ，【初始化的】

最后调 render()函数进行服务器端渲染返回 VNode 给组件。dispatch 和 commit 与 render 是方法

简要介绍各模块在流程中的功能：

dispatch 一个动作到 action ，action 做异步处理，action 调用之后 mutation 改变 state,state 改变完之后组件 view 内存也会发生变化



vuex 存储在内存，localstorage（本地存储）

应用场景:vuex 用于组件之间的传值,localstorage,sessionstorage 则主要用于不同页面之间的传值。

永久性：vuex 当刷新页面(F5 刷新,属于清除内存了)就没了，sessionstorage 页面关闭后就清除掉了，localstorage 不会。

vuex 全局变量，刷页面就全变量就不存在了。用户信息建议存储到 localStorage 里面存储一份

localStorage.setItem(key,value)//存储数据

21、数组循环几种区别

写法：

arr.forEach/filter/map/every/some/forEach/(function(currentValue,index,arr),thisValue)

forEach():数组每个元素都执行一次回调函数即：迭代(循环) 遍历，不支持 break 跳出循环用 try/catch/every/some 代替,返 undefined

foreach 遍历数组的话，使用 break 不能中断循环，使用 return 也不能返回到外层函数

filter() ：筛选数组，创建一个新的数组，返回符合条件所有元素的新数组。遇到 return true 不会终止迭代

map()：返回原数组处理后的新数组。不改变原数组

every()：检查每个元素是否都符合条件,回值 true/false,都满足才 true

some():查找数组中是否有满足条件的元素,有一个满足就返 true,遇 return 会终止遍历

forin 与 forof 与 Object.keys()区别

forin 与 forof 相同点写法：

for(let I in/of arr){}

Object.keys(obj)

For in 遍历数组或者对象的属性，包各原型方法 method 和 name 属性

for of 是 ES6 新增的循环方法

遍历对象时：forin 取属性名，for of 无法循环遍历对象报错

遍历数组时：for in 遍历的是数组的索引（即键名），而 for of 遍历的是数组元素值。

自定义属性：for in 可以，for of 输出不来

for in 遍历数组包括原型方法属性，for of 遍历只是数组内元素，不包括数组原型属性

for in 更适合遍历对象取键名，for of 适合遍历数组取元素

Object.keys()更适合遍历对象，返回属性名组成的数组，属性名直接放数组里不用 push 了而 forin 需要 push 才能把键名放数组里

22、闭包：

正常下：函数内部可以直接读取全局变量，但是在函数外部无法读取函数内部的局部变量

闭包：外部可以读取函数内部的变量，可以让变量的值始终保持在内存中。

缺点：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。

闭包使用场景:子函数可以使用父函数的变量

（1）采用函数引用方式的 setTimeout 调用。?例子

（2）将函数关联到对象的实例方法。

（3）封装相关的功能集。

闭包，内部函数使用外部函数的变量

```
function  f1()  {
        var  n  =  999;
        function  f2()  {
              console.log(n);
        }
        return  f2;
}
var  result  =  f1();
result();//999
```

```
或function f1() {
      var n = 999;
      function f2() { console.log(n);}
      return f2();
}f1();//999
```

闭包使用场景:子函数可以使用父函数的变量

使用场景

* 模块化: 封装一些数据以及操作数据的函数, 向外暴露一些行为
* 循环遍历加监听
* JS 框架(jQuery)大量使用了闭包

* 缺点:

* 变量占用内存的时间可能会过长
* 可能导致内存泄露
* 解决:及时释放 : f = null; //让内部函数对象成为垃圾对象

(1)setTimeout

原生的 setTimeout 传递的第一个函数不能带参数，通过闭包可以实现传参效果。

```
function  f1(a)  {
       function  f2()  {
              console.log(a);
       }
       return  f2;
}
var  fun  =  f1(1);
setTimeout(fun,1000);//一秒之后打印出 1
```

(2)回调

定义行为，然后把它关联到某个用户事件上（点击或者按键）。代码通常会作为一个回调（事件触发时调用的函数）绑定到事件。 当点击数字时，字体也会变成相应的大小。

```
<a  href="#"  id="size-12">12</a>
<a  href="#"  id="size-20">20</a>
<a  href="#"  id="size-30">30</a>
<script  type="text/javascript">
       function  changeSize(size){
            return  function(){
                 document.body.style.fontSize  =  size  +  'px';
            };
       }
       var  size12  =  changeSize(12);
       var  size14  =  changeSize(20);
       var  size16  =  changeSize(30);
       document.getElementById('size-12').onclick  =  size12;
       document.getElementById('size-20').onclick  =  size14;
       document.getElementById('size-30').onclick  =  size16;
</script>
```

(3)使用场景三:封装相关功能集

23、vue 优势：

易于开发：上手快，简单易学，提供页面数据渲染模板引擎如 v-if,v-for,v-show 等，事件绑定如@click 等，代码清晰明了，逻辑简单，单页面，组件复用，前后分离

高性能：Vue 提出虚拟 DOM 概念，以及数据驱动 DOM 思想，减少页面渲染成本，大幅度提高性能。

生态完善：许多开发者提供了对 vue 支持插件如 ElementUI，完善 vue 的生态，故使用 vue 进行开发遇到难题会很少。

开源社区活跃：这提供了 vue 未来更加强大的可能性。

$option

过滤如果不用 filter 怎么用 js 实现过滤 提到$option

24、Event Loop&宏任务微任务

1、什么是 Event Loop(事件循环)？

Event Loop 指的是 js 的执行机制，js 是单线程的,任务分为同步任务和异步任务，

同步任务会放执行栈中也就是主线程。异步任务会提交给异步进程来处理，异步进程处理内容如 ajax(回调函数也有触发)、DOM 事件、setTimeout/setInterval。它们被触发后就放到任务队列里。

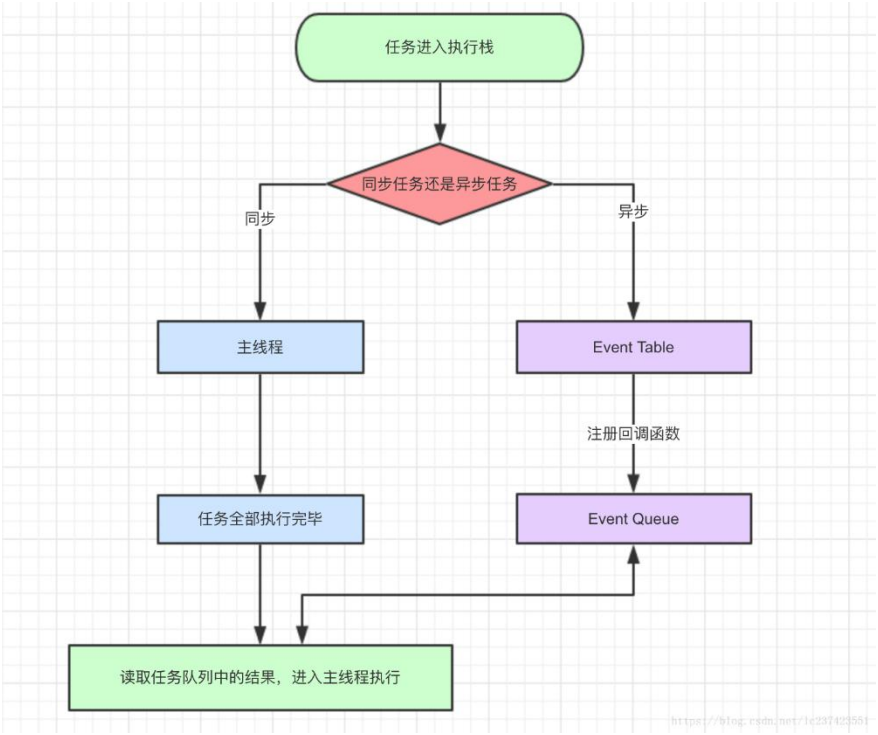
主线程的同步任务都执行完后在到任务队列里看有没有异步任务,如果有就取回来放执行栈中执行,

在回头看还有没有异步任务如有在取回在执行，这种循环的过程叫事件循环 Event Loop

javascript 是单线程同步非阻塞同一时间只能做一个件事，这样导致的体验差的问题。阻塞-异步-放任务队列

什么时候入任务队列中？

任务队列中放的都是异步任务。当异步函数达到触发条件时，比如定时器到时间，Ajax 请求返回数据，根据异步模块类型压入到任务队列里。



2、异步任务

异步任务又分为宏任务和微任务、他们之间区别主要是执行顺序的不同。

微任务：Promise 中的.then .catch .finally 都是微任务。 但 new Promise(是同步主线程的要先执行没有任何宏/微任务说法)

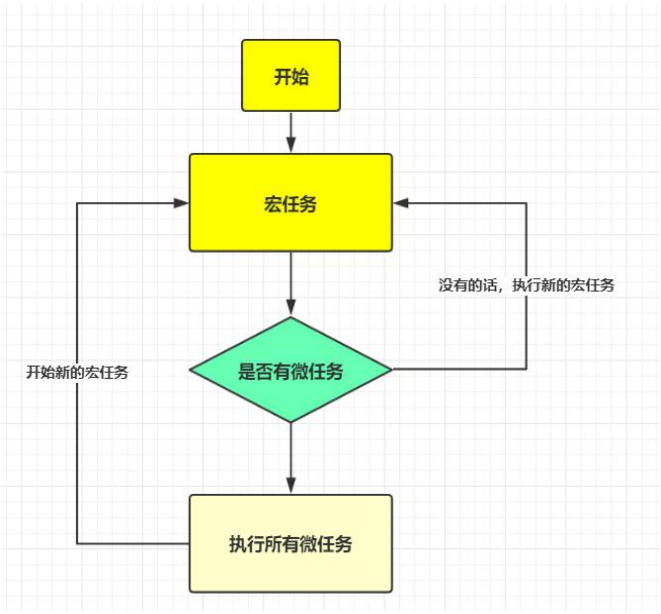
宏任务:script 是一个大的宏任务,代表 setTimeout、setInterval【是浏览器、node 发起的,包含 script(整体代码)，setImmediate(node 里的东西)、I/O（读写文件）】

在当前的微任务没有执行完成时，是不会执行下一个宏任务的，微任务先于下一个宏任务

执行宏任务时顺便把微任务执行完

执行顺序：

主线程(同步)>微任务>下一个宏任务



25、什么是防抖与节流

含义:防抖&节流是 JS 中常见的性能优化手段，如果我们操作浏览器窗口大小会进行 resize，scroll 的一个滚顶事件；或者说我们移动鼠标，对鼠标的 mousemove,mouseover 进行操作;或者说对 input 输入框的 keypress 时，如果我们一直调用绑定在事件上 callback 函数时它会一直调用一直执行，就会浪费 js 性能，严重会导致页面的卡顿。或者说这个 callback 函数是调用后端的接口，那滑动一下就会调几百个后端接口对后端也会有很大负担。所以为了优化性能，防抖和节流会对这类事件进行调用次数的限制。

定义：

防抖(debounce)：指在事件触发 n 秒后再执行回调函数，如果在这 n 秒内又被触发则重新计算

防抖记录的是每次鼠标触发的时间间隔，这次点击距离下次点击几间隔 2 秒每次点击都重新计算，就是 2 秒一次请求

通过 setTimeout 的方式，在一定时间间隔内，将多次触发变成一次触发，限制函数的执行次数

原理：利用定时器设置时间间隔来降低调用频率，

节流 throttle：指每隔一段时间，只执行一次函数，减少一段时间的触发频率（理解为节流阀控制流量的大小，我们没办法减少用户的点击次数，但能减少用户的触发频率）

节流是这次点击请求了再请求只能是 n 秒间隔之后，【假如说上次点击时是 1 点，下次再请求就得 1 点零 2S 了，不论中间这 2 秒点了多少次，记录的总是上次请求时间那个点，而不是每次鼠标触发那个点。这过程中会记录中间隔了多久，记录距离上次请求的一个时间段）

使用场景

防抖使用场景：连续事件，只需触发一次回调的场景【都是前端判断】

input 输入多次发起请求提交(手机号，邮箱的输入验证)，onresize 监听窗口大小时，监听滚动条情况

当页面发生滚动时，由于滚动事件是连续触发的，这样下面代码中的 handler 会被执行很多次，每次都执行这个 handler 中的代码，也许这些代码修改了 DOM 等，这样很消耗性能。

节流使用场景：间隔一段时间执行一次回调的场景【都是后台接口相关】

1、滚动加载，加载更多的操作

2、表单的多次点击提交

防抖

```
document.querySelector('#input').addEventListener('click', debounce(submit, 3000),
false)
function submit() {
console.log(1) // ajax 数据
}
function debounce(fn, timer) {
  let t = null;
  return function () {
     var firstClick = !t;
     if (t) clearTimeout(t);
     if (firstClick) {
        fn.apply(this, arguments)
     }
     t = setTimeout(() => {
        t = null;
     }, timer)
  }
}
//节流
document.querySelector('#input').addEventListener('click', throttle(submit, 2000),
false)

function submit() {
  console.log(1)
}
function throttle(fn, delay) {
     begin = 0;
     return function () {
      var cur = new Date().getTime()
      console.log(cur + '-' + begin, cur - begin)
      if (cur - begin > delay) {
        console.log('大于-可以触发了')
        fn.apply(this, arguments)
        begin = cur
      }
     }
}
```

27、__proto__与 prototype 区别关系

Blue 老师：class(类)上的叫 prototype，实例上的叫__proto__，

看图

prototype：是构造函数(function Foo())和原型对象(Foo.prototype)之间的一个属性

【每个函数都有属性 prototype，该属性所存储的就是原型对象保存共享属性和方法,来实现扩展和继承】

__proto__：是链式的，实例对象(f1)指向原型对象(Foo.prototype)；但原型对象(Foo.prototype)本身又是另一个原型(Object.prototype)的实例对象,(Object.prototype)在往上指__proto__就是 null。这个链就是原型链。这里_proto_就是原型链的重要部分

__proto__是私有属性在 javascript 上没法通过这个属性直接访问某个属性指向的对象，

【对象必然有_proto_属性 new 出来的，但不一定有 prototype】

例:

```
var f1= new Foo(); foo.name
```

如果 foo 本身没有 name 属性会通过私有_proto_向上找【看图】->Foo.prototype(原型)->Object.prototype 到头了还没有返回 undefined

有两个特例

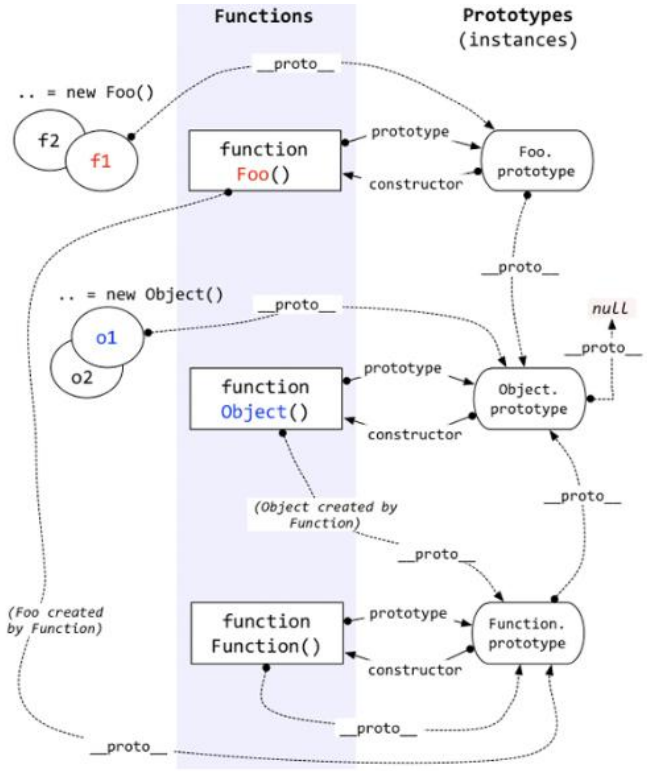
Function.__proto___ === Function.prototype 返回 true,浏览器让它自己构造了自己.

Object.prototype.__proto__ === null 返回 true,万物需要有终点,浏览器让它指向了 null.

每个实例对象的__proto__指向其构造函数的 prototype,prototype 中储存着公用方法和属性.

也就是说：f1 是 new 出来的实例有__proto__，Foo 是构造函数也称 class(类)有 prototype

```
function Foo() {
    this.name = 'yjui';
}
var f1 = new Foo();
console.log(Foo.prototype) //{constructor: f}
console.log(f1.__proto__) //{constructor: f}
console.log(f1.__proto__ === Foo.prototype) //true
console.log(Foo.prototype.__proto__ === Object.prototype) //true
console.log(Object.prototype.__proto__ === null) //true
```



作用域和作用域链？

作用域一个变量的可用范围，比如我们创建了一个函数，函数里面又包含了一个函数，那么现在就有三个作用域，一直会找到全局变量 a，这个查找的过程就叫作用域链。

全局作用域==>函数 1 作用域==>函数 2 作用域

作用域的特点就是，先在自己的变量范围中查找，如果找不到，就会沿着作用域往上找。

在函数体内声明的所有变量都是可见的，这种特性也被称为“声明提前”赋值不提前

如：

```
var a = 1;
function b() {
    var a = 2;
    function c() {
      var a = 3;
      console.log(a);
    }
    c();
}
b();
```

最后打印出来的是 3，因为执行函数 c（）的时候它在自己的范围内找到了变量 a 所以就不会越上继续查找，如果在函数 c()中没有找到则会继续向上找，一直会找到全局变量 a，这个查找的过程就叫作用域链。

原型与原型链？

所有函数都有一个特别的属性:prototype：显式原型属性

所有实例对象都有一个特别的属性:__proto__：隐式原型属性

原型对象即为当前实例对象的父对象

所有的实例对象都有__proto__属性，它指向的就是原型对象

这样通过__proto__属性就形成了一个链的结构----->原型链

28、用 css 画三角形

```
.up {
width: 0; height: 0;
border: 50px solid transparent;/* 设置边框 transparent 为透明 */
border-bottom: 50px solid #f00;/* 要设置的位置 */}
.down {width: 0;height: 0;border: 50px solid transparent;
border-top: 50px solid red;}/* 箭头向右 */
.right {width: 0;height: 0;border: 50px solid transparent;
border-left: 50px solid red;}
.left {width: 0;height: 0;border: 50px solid transparent;
border-bottom: 50px solid red;}/* 箭头向左 */
```

29、写一段原型里的代码

30、computed 与 watch 区别使用场景

Blue:最主要的区别是 computed 像变量一样使用，watch 类似于事件，一个是主动的一个是被动的

computed：是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值；

watch：更多的是「观察」的作用，类似于某些数据的监听回调 ，每当监听的数据变化时都会执行回调进行后续操作；

运用场景：

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

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

31、keep-alive 几个钩子函数

<keep-alive>是 Vue 的内置组件，动态组件上使用 keep-alive，能在组件切换过程中将状态保留在内存中保持切换后的状态而不是最初初始状态，避免组件重复渲染。用 Keep-alive 包裹关于

<keep-alive><component :is="com"></component></keep-alive>

只要有 keep-alive 页面缓存，第一次走 mounted 和 activated，第二次及以后只走 activeted 且组件里有效果

当引入 keep-alive 的时候，页面第一次进入，钩子的触发顺序 created-> mounted-> activated，退出时触发 deactivated。当再次进入 activated。也就是说进入初始 A 是 activated,切到 B 是 deactivated,再进 A 不审 activated

activated，deactivated：这两个钩子函数在普通的组件当中是不存在的，只有当组件被 keep-alive 缓存的时候才会有。可以用来解决再次进入页面的触发。

32、**$nextTick**Blue 老师:这个是等 **vue 完成本次渲染的时候回调用的**，一般用来等待渲染完成，比如说

假设你有个 arr

arr=[]

然后它是 li 身上的

<li v-for="arr">

然后你比如

arr.push(55);

li[0].xxx

这时候因为你数据变了之后 li 不可能立即渲染完成所以就错了

但这样就对

```
arr.push(55);
this.$nexttick(()=>{
 li[0].xxx
});
```

33、http 与 https 区别（原理）

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/tls 加密传输协议。

3、HTTPS 标准端口 443，HTTP 标准端口 80;

4、http 的连接很简单，是无状态的；HTTPS 协议是由 SSL/TLS+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。

客户端在使用 HTTPS 方式与 Web 服务器通信时的步骤

（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/TLS 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。

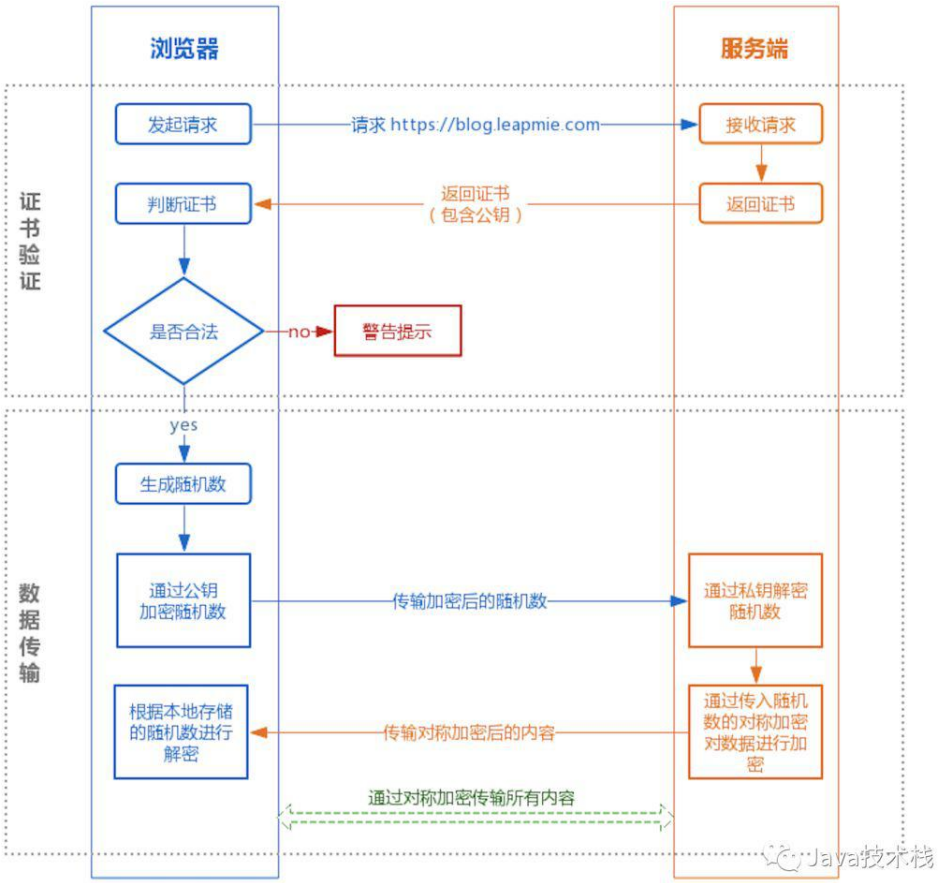
（5）Web 服务器利用自己的私钥解密出会话密钥。

（6）Web 服务器利用会话密钥加密与客户端之间的通信。

HTTPS 的实现原理

大家可能都听说过 HTTPS 协议之所以是安全的是因为 HTTPS 协议会对传输的数据进行加密，而加密过程是使用了非对称加密实现。但其实，HTTPS 在内容传输的加密上使用的是对称加密，非对称加密只作用在证书验证阶段。

HTTPS 的整体过程分为证书验证和数据传输阶段，具体的交互过程如下：



① 证书验证阶段

浏览器发起 HTTPS 请求

服务端返回 HTTPS 证书

客户端验证证书是否合法，如果不合法则提示告警

② 数据传输阶段

1.当证书验证合法后，在本地生成随机数

2.通过公钥加密随机数，并把加密后的随机数传输到服务端

3.服务端通过私钥对随机数进行解密

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

34、jquery 与 vue 相比有什么不同

jq 优点：比原生 js 更易书写，封装了很多 api，有丰富的插件库；缺点：每次升级与之前版本不兼容，只能手动开发，操作 DOM 很慢，不方便，变量名污染，作用域混淆等.

vue 优缺点：易于开发：上手快，简单易学，提供页面数据渲染模板引擎如 v-if,v-for,v-show 等，事件绑定如@click 等，代码清晰明了，双向绑定，diff 算法，组件的复用，单页面，

高性能：虚拟 DOM，减少 DOM 操作减少页面渲染成本，通信方便，便于协同开发提高了开发效率,提高性能

生态完善：许多开发者提供了对 vue 支持插件如 ElementUI，完善 vue 的生态，故使用 vue 进行开发遇到难题会很少。

35、Vue 中的 diff 算法?

要知道渲染真实 DOM 的开销是很大的，比如有时候我们修改了某个数据，如果直接渲染到真实 dom 上会引起整个 dom 树的重绘和重排,有没有可能我们只更新我们修改的那一小块 dom 而不要更新整个 dom 呢？diff 算法能够帮助我们。

我们先根据真实 DOM 生成一颗 virtual DOM，当 virtual DOM 某个节点的数据改变后会生成一个新的 Vnode,然后 Vnode 和 oldVnode 作对比,发现有不一样的地方就直接修改在真实的 DOM 上,然后使 oldVnode 的值为 Vnode

diff 算法包含 tree diff， component diff,element diff

tree diff： 新旧两棵 DOM 树逐层对比的过程，就是 tree diff,当整颗 DOM 逐层对比完毕，则所有需要被按需更新的元素，必然能够找到。每一层节点进行对比，只负责每一层怎么对比它不管

component diff：每一层有各种组件，每个组件进行对比叫做 compnent Diff

如果对比前后，组件的类型相同，则暂时认为此组件不需要被更新

如果对比前后，线件类型不同（前组件 A，后组件 B），则需要移除旧组件，创建新组件，并追加到页面上

element diff：在进行组件对比的时候，如果两组件类型相同，则需要进行元素级别的对比，这中 element Diff

diff 算法的本质是**找出两个对象之间的差异**

key 的作用主要是 ：决定节点是否可以复用、建立 key-index 的索引，主要是替代遍历，提升性能。

virtual DOM 和真实 DOM 的区别?

virtual DOM 是将真实的 DOM 的数据抽取出来，以对象的形式模拟树形结构

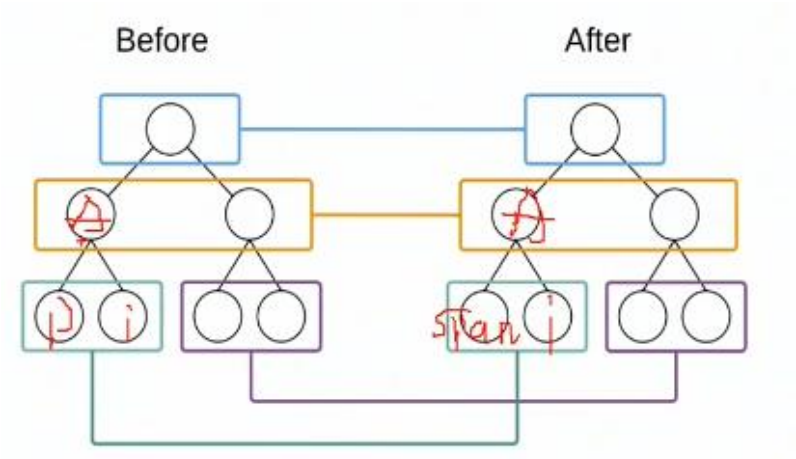
虚拟 DOM 提供 DOM 树

diff 算法比较新旧节点时，比较只会在同层级进行，不会跨层级比较。

123

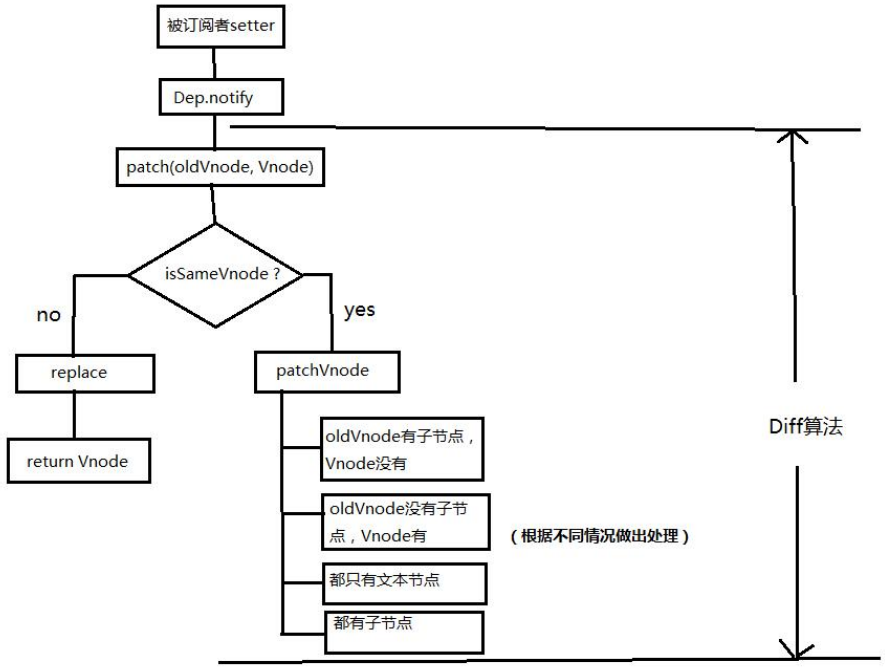
456

上面的代码会分别比较同一层的两个 div 以及第二层的 p 和 span，但是不会拿 div 和 span 作比较。在别处看到的一张很形象的图:



diff 流程图

当数据发生改变时，set 方法会让调用 Dep.notify 通知所有订阅者 Watcher，订阅者就会调用 patch 给真实的 DOM 打补丁，更新相应的视图。



37、call 与 apply 与 bind 区别

作用：1、调用函数 2、改变 this 指向

call()方法 第一个参数和 apply()方法的一样都是 this 的指向不参与参数的传递,第二个参数列举出来

语法：函数.call(this 指定,参数 1,参数 2,..)

例：

调用函数 fn.call()的 this 指向是 window

fn.call(o)的 this 指向 o 对象，

```
function fn(x,y) {
    console.log(this);
    console.log(x+y);
}
var o = {
    name: 'yjui'
}
fn.call()  //this 是 window
fn.call(o,1,2) //this 是 {name: "yjui"}  3
```

apply()方法 第一个参数和 call 相同，第二个参数是数组。

语法：apply(this 指向， [必须是数组]...);

如果你想生成一个新的函数长期绑定某个函数给某个对象使用，则可以使用 const newFn = fn.bind(thisObj); newFn(arg1, arg2...)

Bind()方法：改变 this 指向,不会调用函数要赋值输出

Fun.bind(thisArg,arg1,arg2...)返回原函数改变 this 后产生的新函数

不同点例

```
function  add(c,d){
          return  this.a  +  this.b  +  c  +  d;
}
```

```
var  s  =  {a:1,  b:2};
console.log(add.call(s,3,4));  //  1+2+3+4  =  10
console.log(add.apply(s,[5,6]));  //  1+2+5+6  =  14
```

38、用过视频 video 兼容问题

1、自动播放

PC 端：没法在页面加载完成的时候播放音频或视频。这是 Chrome 66 的新特性

解决：静音就可以了，Chrome 66 为了避免标签产生随机噪音

<video muted></video>

//Js 写法

var video = document.getElementById("video"); video.play();

// Jquery 写法

$("#video")[0].play();//或$("#video").get(0).play();

IOS 本身是禁止自动播放的，需要由用户手动触发，IOS 微信浏览器中可以借助 jssdk 的 wx.ready(()=>{})里面进行自动播放去实现，但是这种方法在安卓微信上有失效了。（安卓现在还在用 X5 核？）

移动端：

移动端视频需要用户手动触发才能播放，这样就导致进入有视频的界面是黑色的

移动端 H5 页面浏览器机制下，视频浮层一直在最上方；微信下不是最上方

解决办法：

只要有点击事件就可以，可以在视频上放一视频封面背景图，在加播放按钮图标，这样触发最上层遮罩时视频也会被触发 ，因为有用户点击事件在

```
$("#videoPlay").click(function () {
        var video= document.getElementById("video");
        video.play();
}
```

2、画中画功能

<video>视频画中画功能，火狐或一些其他浏览器是不支持的，移动端画中画不同浏览器手机效果也不同，

解决：pc 端画中画功能可以用改变视频大小，拖拽自行实现来模拟大致效果，移动端同理

39、箭头函数和普通函数区别

1. 普通函数的 this：指向它的调用者，如果没有调用者则默认指向 window.

2.箭头函数的 this: 指向箭头函数定义时所处的对象，而不是箭头函数使用时所在的对象，默认使用父级的 this.

箭头函数：

A)更简洁的语法：

箭头函数 var a = ()=>{return 1;}相当于普通函数 function a(){return 1;}

B)没有 this

在箭头函数出现之前，每个新定义的函数都有其自己的 this 值

```
var myObject = {
  value: 1,
  getValue: function () {
    //方法 this 指向 myObject 对象
    console.log(this.value)
  },
  double: function () {
    //方法 this 指向 myObject 对象
    return function () {
      //函数内 this 指向 window
      console.log(this.value = this.value * 2);
    }
  }
}
myObject.double()();  //NaN
myObject.getValue();  //1
```

使用箭头函数

```
// 使用箭头函数
    var myObject = {
      value: 1,
      double: function () {
        //this 指向 myObject 对象
        return () => {
          console.log(this.value = this.value * 2); //this 指向 myObject 对象
        }
      }
    }
```

C) 不能使用 new 构造函数：箭头函数作为匿名函数,是不能作为构造函数的,不能使用 new

```
var B = () => { value: 1;}
var b = new B(); //TypeError: B is not a constructor
```

D) 不绑定 arguments，用 rest 参数...解决

```
/*常规函数使用 arguments*/
function test1(a) {console.log(arguments);   //1}
/*箭头函数不能使用 arguments*/
var test2 = (a) => { console.log(arguments) }  //ReferenceError: arguments is not d
efined
/*箭头函数使用 reset 参数...解决*/
let test3 = (...a) => { console.log(a[1]) } //22
test1(1);
test2(2);
test3(33, 22, 44);
```

E)使用 call()和 apply()调用

由于 this 已经在词法层面完成了绑定，通过 call() 或 apply() 方法调用一个函数时，只是传入了参数而已，对 this 并没有什么影响：

```
var obj = {
  value: 1,
  add: function (a) {
    //this 是 obj
    var f = (v) => v + this.value;//this 是 obj,a==v,3+1
    return f(a);
  },
  addThruCall: function (a) {
    //this 是 obj
    var f = (v) => v + this.value;//此 this 指向 obj 4+1
    var b = { value: 2 };
    // this 已经在词法层面完成了绑定 call()无作用相当return f(a),f 函数并非指向 b,只是传入了 a 参数而已
    return f.call(b, a);
  }
}
console.log(obj.add(3));    //4
console.log(obj.addThruCall(4));    //5
```

普通的

```
var obj = {
  value: 1,
  add: function (a) {
    // this 是 obj
    var f = function (v) {
      //this 是 window，3+window
      return v + this.value
    }
    return f(a);
  },
  addThruCall: function (a) {
    //  this 是 obj
    var f = function (v) {
      // this 是 b{value:2}  4+2
      return v + this.value
    }
    var b = { value: 2 };
    return f.call(b, a);
  }
}
console.log(obj.add(3));    //NaN
console.log(obj.addThruCall(4));    //6
```

E) 捕获其所在上下文的 this 值，作为自己的 this 值,同指向

```
var obj = {
      a: 10,
      b: function () {
        console.log(this.a); //10  this->obj
      },
      c: function () {
        return () => {
          console.log(this.a); //10  this->obj
        }
      }
    }
    obj.b();
    obj.c()();
```

F) 箭头函数没有原型属性

```
var a = () => { return 1;}
function b() {return 2;}
console.log(a.prototype);//undefined
console.log(b.prototype);//{constructor: ƒ}
```

H)箭头函数不能当做 Generator 函数,不能使用 yield 关键字

K)箭头函数不能换行

40、css3 用过哪些属性:box-shadow 和 transition 分别有哪些属性值

box-shadow：边框阴影

text-overflow 文本溢出

{text-overflow:ellipsis;overflow:hidden;white-space:nowrap;width: 400px;}

box-sizing 盒模型两个值 content-box、border-box

E:nth-child(n){}第 n 个元素

border-radius：圆角边框

background-size：指定背景图片尺寸

background-clip：指定背景图片从什么位置开始裁剪

text-shadow：文本阴影

word-wrap：自动换行

transform 新增动画效果变换效果：

transition 过渡效果

box-shadow: h-shadow v-shadow blur spread color inset;

h-shadow 必需。水平阴影的位置。允许负值

v-shadow 必需。垂直阴影的位置。允许负值。

blur 可选。模糊距离。

spread 可选。阴影的尺寸。

color 可选。阴影的颜色。请参阅 CSS 颜色值

inset 可选。将外部阴影 (outset) 改为内部阴影。

transition: property duration timing-function delay;

transition-property 规定设置过渡效果的 CSS 属性的名称。

transition-duration　规定完成过渡效果需要多少秒或毫秒。

transition-timing-function　规定速度效果的速度曲线。

transition-delay 定义过渡效果何时开始。

划过过宽度变 300px

```
div{
width:100px;
height:100px;
background:blue;
transition:width 2s;
-moz-transition:width 2s; /* Firefox 4 */
-webkit-transition:width 2s; /* Safari and Chrome */
-o-transition:width 2s; /* Opera */
}
div:hover{width:300px;}
```

41、css 如何选择 ul 下的第 50 个 li

ul li:first-child{margin-left:0;}　第一个

ul li:last-child{margin-left:0;}　最后一个

ul li:nth-child(5){margin-left:0;} 指定第几个

ul li:nth-child(3n+1){background:orange;}/*匹配第 1、第 4、第 7、…、每 3 个为一组的第 1 个 LI*/

ul li:nth-child(odd){background:orange;}　基数

ul li:nth-child(even){background:orange;}　偶数

42、代码实现把一个 span 的 click 事件，委托给父级 div#abc（原生 js，不用考虑兼容性）

事件委托：利用事件冒泡的特性子元素上的处理事件注册在父元素

事件委托的优点：

1、可以节省大量内存占用，减少事件注册减少 DOM 操作。节约性能不需要为每个元素都绑定事件监听器。

2、当新增对象时，无需再对其进行事件绑定

```
var oDiv = document.querySelector('#abc');
oDiv.onclick = function(ev) {
    var ev = ev || window.event;
    var target = ev.srcElement || ev.target;
    if (target.nodeName.toLowerCase() == 'span') {
     alert('span');
    }
};
```

改成 jquery

```
var oDiv = $('#abc');
oDiv[0].onclick=function(){}
```

或

```
var oDiv = $('#abc');
 oDiv.on('click', function() {})
```

DOM 事件标准定义了两种事件流：

冒泡事件流（从里向外）：从当前触发的事件目标外向传递，依次触发事件直到 document 为止。

阻止冒泡：`if(e.stopPropagation){e.stopPropagation();}else{e.cancelBubble = true;}`

捕获时间流（从外向里）：顶级对象 document 开始触发、一级一级往里传递直到事件目标为止，这个外向内的过程为事件捕获阶段。

通过 addEventListener() 的第三个属性来设置事件是通过捕获阶段注册的（true），还是冒泡阶段注册的（false）。默认情况下是 false。

例：

```
<div id="div">
    <span id="span">
       <button id="btn">button</button>
    </span>
</div>
```

//冒泡 btn span div

```
var oDiv = document.querySelector('#div');
var ospan = document.querySelector('#span');
 var obtn = document.querySelector('#btn');
  oDiv.addEventListener('click',function() {
        console.log('div');
  },false);
ospan.addEventListener('click',function() {
      console.log('span');
},false);
 obtn.addEventListener('click',function() {
       console.log('btn');
 },false);
//阻止冒泡只输出 btn 写法
obtn.addEventListener('click',function(ev) {
     var e = ev || window.event;
     if (e.stopPropagation) {
         e.stopPropagation();
      } else {
         e.cancelBubble = true;
      }
      console.log('btn');
},false);
```

//改成 true 就是捕获 结果为 div span btn

事件冒泡捕获事件委托三者的关系是怎样的呢

一、事件捕获和冒泡是现代浏览器的执行事件的两个不同阶段

二、事件委托是利用冒泡阶段的运行机制来实现的

Event 对象提供了一个属性叫 target，可以返回事件的目标节点，称为事件源。target 表示当前的事件操

作的 dom，但是不是真正操作 dom。有兼容性，标准浏览器用 ev.target，IE 浏览器用 event.srcElement

事件委托代码实例：

```
$(function() {
    var oSpan = document.getElementById('span');
    oSpan.onclick = function(ev) {
        var ev = ev || window.event;
        var target = ev.target || ev.srcElement;
        if (target.nodeName.toLowerCase() == 'span') {
            target.style.background = 'red';
            console.log(target);
        }
    };
});
```

44. 简述 Vue 的响应式原理

当一个 Vue 实例创建时，vue 会遍历 data 选项的属性，用 Object.defineProperty 将它们转为 getter/setter 并且在内部追踪相关依赖，在属性被访问和修改时通知变化。

每个组件实例都有相应的 watcher 程序实例，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新。

typeof 返回类型与以上依次对应为 number object boolean string undefined

45、Vue 中给 data 中的对象属性添加一个新的属性时会发生什么，如何解决？

解：点击 button 会发现， obj.b 已经成功添加，但是视图并未刷新：

因为 vue 底层的东西本身就不支持对数组某一个特定元素的监控，push，pop 这些都能监控到。

对 json 或数组的添加和删除是监听不到的

不是 VUE 的 bug，是 VUE 依赖底层库的问题，

vue 依赖机制 observe --数据监听

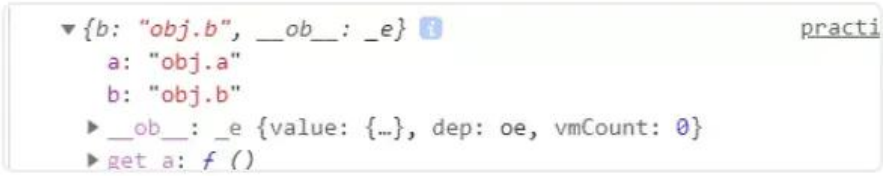
并不是所有的数据操作都能被监听到：

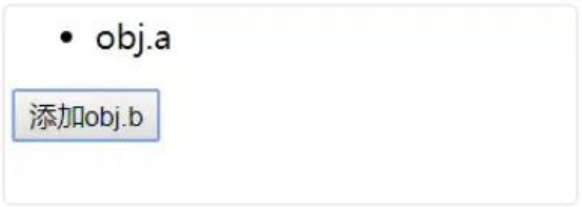
这时就需要使用 Vue 的全局 api——$set()：

vue 实例.$set(数据，key，val)

Vue.set(数据，key，val) // 直接在 Vue 类上

```
addObjB() {
      // this.obj.b = 'obj.b'    下面两种写法
      this.$set(this.obj, 'b', 'obj.b')
      Vue.set(this.obj, 'b', 'obj.b')
     console.log(this.obj)
}
```






```
<div id='app'>
    <ul><li v-for="value in obj" :key="value">{{value}}</li> </ul>
    <button @click="addObjB">添加 obj.b</button>
</div>
<script src='vue2.js'></script>
<script type="text/javascript">
    var vm = new Vue({
        el: '#app',
        data() {
                obj: {
                    a: 'obj.a'
                }
        },
        methods: {
            addObjB() {
                this.obj.b = 'obj.b'
                console.log(this.obj)
            }
        }
    })
</script>
```

46、axios 是什么？怎么使用？描述使用它实现登录功能的流程？

https://chuansongme.com/n/394228451820

答：请求后台资源的模块。

Axios 是一个基于 promise 的 HTTP 库，可以工作于浏览器中，也可以在 node.js 中使用，提供了一个 API 用来处理 XMLHttpRequests 和 node 的 http 接口

可能很多人会疑问：用 jquery 的 get/post 不就很好了，为什么要用 Axios？原因主要有：

（1）Axios 支持 node.js，jquery 不支持

（2）Axios 基于 promise 语法标准，jquery 在 3.0 版本中才全面支持

（3）Axios 是一个小巧而专业的 HTTP 库，jquery 是一个大而全的库，如果有些场景不需要使用 jquery 的其他功能，只需要 HTTP 相关功能，这时使用 Axios 会更适合

除了 get/post，还可以请求 delete,head,put,patch

47、Vue 组件中 data 为什么必须是函数？

答：在 new Vue() 中，data 是可以作为一个对象进行操作的，然而在 component 中，data 只能以函数的形式存在，不能直接将对象赋值给它。 当 data 选项是一个函数的时候，每个实例可以维护一份被返回对象的独立的拷贝，这样各个实例中的 data 不会相互影响，是独立的。

组件中的 `data` 写成一个函数，数据以函数返回值形式定义，这样每复用一次组件，就会返回一份新的 `data`，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。