Web面试题

作者：小新老师

Web面试题

- MVC 是什么
- MVVM 是什么
- vue 双向绑定原理
- angular 双向绑定原理
- 单向绑定 与 双向绑定的好处和劣势
- Vuex 是什么
- Vuex 原理
- Vue-router 原理
- router-link 和 $router.push 实现跳转的原理
- promise
- promise 和 await/async 区别
- jQuery 链式操作原理
- 栅格布局原理
- 简述 ES6 使用到的新语法
- v-if 和 v-show 的区别
- vue 的生命周期有哪些？使用场景？
- vue 获取数据在哪个周期函数
- 兼容性问题
- vue 之 keep-alive
- vue 的父子传参
- vue 的子父传参
- vue 的兄弟传参
- 垂直居中在不知道高度时怎么解决
- 代码管理工具
- 什么是单页应用
- Vue的权限管理
- 高度坍塌的解决方式
- Http 的工作过程
- 说出 URL URI URN 的区别
- HTML5 的新特性有哪些
- 闭包及应用场景
- 用 css 画一条 0.5px 的线
- 用 css 画一个三角形
- 跨域问题
- PC端 与 手机端 的自适应
- 页面图片很多，访问很慢，怎么优化
- 微信小程序的生命周期
- 小程序的 bindtap 和 catchtap 区别
- 小程序的文件结构类型
- vue 路由守卫
- 解释 vue 的 nextTick
- 深拷贝 与 浅拷贝？如何实现深拷贝
- Echarts 中实现区域染色
- 原型 与 原型链

let const var 的差别 与 使用场景
箭头函数，可以改变 this 指向吗
token相关
数组常用方法
http 状态码
http 与 https 区别
浏览器的缓存方式
网络安全：csrf
webpack的理解
set 和 map 数据结构
vue 的 computed 特性
vue 的 watch 是否可以监听数组
防抖 与 节流
ajax 超时断开
严格模式 与 非严格模式的 区别
apply bind call 的区别
vue 与 react 的区别
前端的优化方案
手写一个递归函数
前后端分离的意义
前端工程化
get 和 post 的区别
Restful 的请求有哪些方式
rem 是什么
写一个验证手机号的正则
冒泡排序

MVC 是什么

MVC 是Model-View-Controller的缩写.

主要目的是对代码解耦. 把混合在一起的代码拆分成 3 部分;

让html中不存在任何逻辑代码，没有JavaScript代码痕迹.

以原生 HTML 为例：

- Model：数据模型层

  > 早期前端： 弱化的Model. 不关注 Model 层，数据都是从 服务器 请求下来，直接使用即可.
  > 现在前端：使用 WebStorage，框架中的Vuex，Redux等管理数据
  > 在TypeScript语言中，新增了数据类型声明特征，才让 Model 在前端变得尤为重要.

- View：视图层

  > 书写普通的html. 不掺杂任何 JS 代码.
  > 例如： `<button id="tedu">Tedu</button>`
  > 注意：此按钮 没有 onclick 的事件写法.

- Controller：控制器层

> 控制 HTML 的具体行为，具体为script代码范围，例如 为id="tedu"的按钮添加事件的写法：

```
var btn = document.getElementById("tedu");
btn.onclick = function(){ alert(123456) }
```

MVVM 是什么

MVVM 是Model-View-ViewModel的简写。它本质上就是 MVC 的改进版。它本质上就是 MVC 的改进版。MVVM 就是将其中的 View 的状态和行为抽象化，让我们将视图 UI 和业务逻辑分开。

以 vue 为例：

- Model：数据模型层

  > script 部分的 data 属性，专门管理数据

- View：视图层

  > 即 template 中的代码，负责 UI 的构建

- ViewModel：视图模型层

  > new Vue({}) 部分. 自动管理数据和视图.
  >
  > 重点是双向数据绑定功能，实现了 数据变化视图自动变更. 视图变化,数据自动联动.

vue 双向绑定原理

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，触发相应的监听回调。

具体步骤如下：

1. 首先，需要对observe的数据对象进行递归遍历，包括子属性对象的属性，都加上setter getter。这样的话，给这个对象的某个属性赋值，就会触发setter，那么就能监听到数据变化。（其实是通过Object.defineProperty()实现监听数据变化的）
2. 然后，需要compile解析模板指令，将模板中的变量替换成数据，接着初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者。一旦数据有变动，订阅者收到通知，就会更新视图
3. 接着，Watcher订阅者是Observer和Compile之间通信的桥梁，主要负责：
   1）在自身实例化时，往属性订阅器（Dep）里面添加自己
   2）自身必须有一个update()方法
   3）待属性变动，dep.notice()通知时，就调用自身的update()方法，并触发Compile中绑定的回调
4. 最后，viewmodel(vue实例对象)作为数据绑定的入口，整合Observer、Compile、Watcher三者，通过Observer来监听自己的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化（ViewModel)-》视图更新(view)；视图变化(view)-》数据(ViewModel)变更的双向绑定效果。

angular 双向绑定原理

脏值检查（angular.js）

angular.js是通过脏值检测的方式，对比数据是否有变更，从而决定是否更新视图。最简单的方式就是通过setInterval()定时轮询检测数据变动。angular.js只有在指定的事件触发时，进入脏值检测，大致如下：
- DOM事件，譬如用户输入文本，点击按钮等（ng-click）
- XHR响应事件（$http）
- 浏览器location变更事件（$location）
- Timer事件（$timeout,$interval）
- 执行$digest()或$apply()

单向绑定 与 双向绑定的好处和劣势

- 单向数据绑定

  以输入框为例，React 框架采用的是 单向绑定，需要配合 onChange 事件．才能实现类似 vue 双向绑定效果

```
<input type="text" value={this.state.word} onChange={this._onChange} />

_onChange = (event) => {
  let val = event.target.value;
  this.setState({ word: val });
};
```

  - 优点：

    > 单向数据流，所有状态变化都可以被记录、跟踪，状态变化通过手动调用通知，源头易追溯
    > 例如：通过 _onChange 方法可以实时监听输入框数据变更．

  - 缺点：

    > 代码量会相应的上升，数据的流转过程变长，从而出现很多类似的样板代码。
    > 例如：每个输入框 都要添加对应的 方法监听变更． 大型表单项目会导致代码非常啰嗦．

- 双向数据绑定：Vue 框架采用的是 双向数据绑定

```
<input type="text" v-model="uname" />
```

  - 优点：

    > 在表单交互较多的场景下，会简化大量业务无关的代码。
    > 例如：React中的事件绑定 onChange 都可以省略

- 缺点：

  > 无法实时掌控数据的状态变化
  > 例如：数据的更新都是自动化操作，是无感的．要想实现 纯数字 的输入需求，就需要更多操作．

Vuex 是什么

Vuex 是一个专为 Vue.js 应用程序开发的 状态管理模式．

它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

使用场景：

- 组件间的状态共享：登录状态
- 组件间的数据共享：购物车的数据，登录的token

5个核心属性

- state：数据存放
- getters：相当于计算属性
- mutation：同步操作，修改数据
- action：异步操作
- modules：模块化

Vuex 原理

vuex实现了一个单项数据流，在全局又有一个state存放数据，

当组建要更改state中的数据时，必须通过Mutation进行，mutation同时提供了订阅者模式供外部插件调用获取state数据的更新。

而当所有异步操作（常见于调用后台接口异步获取更新数据）或批量的同步操作需要走Action，

但Action也是无法直接修改state的，还是需要通过mutation来修改state的数据。

最后根据state的变化，渲染到视图上。

Vue-router 原理

vue-router通过hash与history两种方式实现前端路由

更新视图但不重新请求页面是前端路由原理的核心之一.

目前在浏览器环境中这一功能的实现主要有两种方式：

1. hash：利用 URL 中的 hash. 形式上会多个#

```
http://localhost:8080/#/login
```

hash("#") 的作用是加载 URL 中指示网页中的位置。

本身以及它后面的字符称之为 hash，可通过 window.location.hash 获取

   - hash 虽然出现在 url 中，但不会被包括在 http 请求中，它是用来指导浏览器动作的，对服务器端完全无用，因此，改变 hash 不会重新加载页面。
   - 每一次改变 hash(window.localtion.hash)，都会在浏览器访问历史中增加一个记录。

   利用 hash 的以上特点，就可以来实现前端路由"更新视图但不重新请求页面"的功能了。

2. history：html5 中新增的方法. 形式上比 hash更好看

```
http://localhost:8080/login
```

History interface 是浏览器历史记录栈提供的接口，通过back()、forward()、go()等方法，我们可以读取浏览器历史记录栈的信息，进行各种跳转操作。

从 HTML5开始，History interface 提供了2个新的方法：pushState()、replaceState() 使得我们可以对浏览器历史记录栈进行修改

这2个方法有个共同的特点：当调用他们修改浏览器历史栈后，虽然当前url改变了，但浏览器不会立即发送请求该url，这就为单页应用前端路由，更新视图但不重新请求页面提供了基础

> history模式需要后端服务器进行 路径重写 处理. 否则会出现 404 错误

router-link 和 $router.push 实现跳转的原理

router-link

- 默认会渲染为 a 标签，可以通过 tag 属性修改为其他标签
- 自动为 a 标签添加 click 事件，然后执行 `$router.push()` 实现跳转

$router.push

- 根据路由配置的 mode 确定使用 `HTML5History` 还是 `HashHistory` 实现跳转
  - `HTML5History` ：调用 `window.history.pushState()` 跳转
  - `HashHistory` ：调用 `HashHistory.push()` 跳转

promise

Promise 对象代表一个异步操作，有三种状态：

- pending：初始状态，不是成功或失败状态。
- fulfilled：意味着操作成功完成。
- rejected：意味着操作失败。

优点

- 将异步操作以同步操作的流程表达出来,避免了层层嵌套的回调函数； 避免回调地狱
- Promise 对象提供统一的接口，使得控制异步操作更加容易。

缺点

- 无法取消 Promise，一旦新建它就会立即执行，无法中途取消.
- 如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。
- 当处于 Pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

基础用法

奇数报错，偶数正常

```
function demo() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let num = Math.round(Math.random() * 100);

      if (num % 2 == 0) {
        resolve("num是偶数" + num);
      } else {
        reject(new Error("num是奇数" + num));
      }
    }, 500);
  });
}

console.log("运行中...");

demo()
  .then((res) => {
    console.log(res);
  })
```

```
  .catch((err) => {
    console.log(err);
  });

```

封装 ajax

```
function ajax(url) {
  return new Promise((resolve, reject) => {
    let req = new XMLHttpRequest();

    req.open("GET", url, true);

    req.onload = function () {
      if (req.status == 200) {
        let data = JSON.parse(req.responseText);
        resolve(data);
      } else {
        reject(new Error(req.statusText));
      }
    };

    req.onerror = function () {
      reject(new Error(req.statusText));
    };

    req.send();
  });
}

ajax("http://101.96.128.94:9999/mfresh/data/news_select.php")
  .then((res) => console.log(res))
  .catch((res) => console.log(res));

```

封装Promise

```
function myPromise(callback) {
  // 用 常量 保存状态值. 使用时有 代码提示, 不容易写错.
  const PENDING = "pending";
  const REJECTED = "rejected";
  const FULFILLED = "fulfilled";

  this.status = PENDING; //初始状态值: pending
  this.msg; // 存储执行结果

  this.thens = []; //存储多个 then();  xxx().then().then().then().
  this.func_error;

  // 通过 .then(成功回调, 失败回调) 接受用户传入的回调方法
  this.then = function (func_success, func_error) {
    this.thens.push(func_success);
    this.func_error = func_error;

    // 返回this, 支持链式写法. 例如: xxx().then().then().then()...
```

```
    return this;
  };

  // 接受 失败回调 函数
  this.catch = function (func_error) {
    this.func_error = func_error;
  };

  // 成功时触发，注意必须箭头函数 保持 this 指向
  let resolve = (msg) => {
    // 只有状态为 pending 时，才能更改
    if (this.status != PENDING) return;

    this.status = FULFILLED; //修改状态值
    this.msg = msg;

    // 触发 终结 方法
    this.complete();
  };

  // 失败时触发，注意必须箭头函数 保持 this 指向
  let reject = (msg) => {
    if (this.status != PENDING) return;

    this.status = REJECTED; //修改状态值
    this.msg = msg;

    // 触发 终结 方法
    this.complete();
  };

  // callback 为用户传入的函数，即异步操作所在位置
  callback(resolve, reject);

  // 统一出口：失败和成功 最终都调用此方法；便于维护
  this.complete = () => {
    if (this.status == FULFILLED) {
      let first_time = true; //首次
      let res;

      // 考虑多个 then() 的情况
      this.thens.forEach((item) => {
        if (first_time) {
          res = item(this.msg); //首次执行
          first_time = false;
        } else {
          // 每次 then 都是上一次的返回值
          res = item(res);
        }
      });
    }
    if (this.status == REJECTED && this.func_error) {
      this.func_error(this.msg);
    }
  };
```

```
}
////////////////////////////////////////
/////////////// 测试方式 /////////////////

console.log("运行中...");

function demo() {
  return new myPromise((resolve, reject) => {
    setTimeout(() => {
      let num = Math.round(Math.random() * 100);

      if (num % 2 == 0) {
        resolve(num);
      } else {
        reject(new Error("奇数" + num));
      }
    }, 300);
  });
}

demo()
  .then((res) => {
    console.log("1" + res);
    return res;
  })
  .then((res) => {
    console.log(res / 2);
    return res / 2;
  })
  .then((res) => {
    console.log("3." + res);
    return res / 2;
  })
  .then((res) => {
    console.log("4." + res);
    return res / 2;
  })
  .then((res) => {
    console.log("5." + res);
    return res / 2;
  })
  .catch((err) => console.log(err));

// demo()
//   .then((res) => {
//     console.log(res);
//   })
//   .catch((err) => {
//     console.log(err);
//   });
```

promise 和 await/async 区别

区别主要在于按顺序调用多个 异步函数 时的写法 和 报错获取

Promise方式

```
ajax().then(func1).then(func2).then(func3).then(func4)
```

await/async方式

```
async function demo(){
  await res = ajax();
  await res = func1(res);
  await res = func2(res);
  await res = func3(res);
  await res = func4(res);
}
```

总结：

- 当遇到多个异步函数时
  - Promise 方式需要很多 `.then` ，会导致代码不易读 且 结构复杂
  - await/async 方式让异步代码的格式 与 同步代码一样，更易读
- 报错读取
  - Promise 使用 `.catch` 抓取报错
  - await/async 使用 `try...catch...` 方式抓取报错

jQuery 链式操作原理

链式写法

```
$('#id').css().append().xxx()
```

原理

每个函数调用后的返回值，都是当前对象，主要依赖每个函数结尾的 `return this`

详细参考

```
<body>
  <div id="d1">
    <i>Hello World!</i>
  </div>

  <script>
    function MyQuery(id) {
      this.el = document.getElementById(id);

      this.css = function (property, value) {
        this.el.style[property] = value;
        return this; //关键点
      };

      this.append = function (content) {
        this.el.innerHTML += content;
```

```
        return this; //关键点
      };
    }

    const $ = function (id) {
      return new MyQuery(id);
    };

    $("d1").css("color", "red").append("<b>haha</b>");
    // $("d1").css("color", "red")  的返回值是 this
    // 所以 append 相当于是 this.append().  而 this 就代表 MyQuery 对象
  </script>
</body>
```

栅格布局原理

随着屏幕设备或视口（viewport）尺寸的增加，系统会自动分为最多12列(也可以自己定制多少列都行)。

通过一系列的行（row）与列（column）的组合创建页面布局

通过定义容器大小，平分12份，再调整内外边距，最后结合媒体查询，实现强大的响应式网格系统。

简述 ES6 使用到的新语法

1. let：块级作用域
2. const：常量；块级作用域；一旦声明，则运行期间无法修改.
3. 模板字符串
4. 解构赋值： `let {name, age} = {name:'dongdong', age:33}`
5. `...` ：代替 arguments 变量，接受函数的多余参数. `function name(...args){}`
6. 箭头函数：匿名函数，自带 this 保持为定义所在对象.
7. `...` 扩展对象，取代 `Object.assign()`

```
let a = {name:'xiaoxin', age:32};

let b = Object.assign(a, {gender: 1});
//等价于下方. 可以看到 ... 更简单
let c = {...a, {gender:1}}
```

8. Promise：异步编程的一种方案，解决 回调地狱
9. class 面向对象

v-if 和 v-show 的区别

区别

- v-if
  - 通过删除DOM元素实现元素的隐藏
  - 惰性：只有条件为真时，才会加载元素到DOM
- v-show:

- 通过设置元素的css样式： `display:none` 实现元素的隐藏，不操作DOM.
- 非惰性：不管条件真与假，都会加载元素到 DOM

所以

- v-if 的开销比 v-show 更大
- v-show 有更高的初始化渲染消耗

适用场景

- 一个元素频繁进行 隐藏 和 显示 操作，使用 `v-show` 更加合适
- 一个元素不频繁进行 隐藏 和 显示操作，使用 `v-if` 更合适.

  例如：需要网络请求 成功后才显示的内容

vue 的生命周期有哪些？使用场景？

加载时

- `beforeCreate` ：开始创建
  - data 和 methods 都未创建，此处不能使用
- `created` ：创建完毕
  - data 和 methods 创建完毕，最早的可以使用处
- `beforeMount` ：开始挂载
  - 内存中已编译好所有内容，准备显示到页面
- `mounted` ：挂载完毕
  - 组件脱离创建阶段，真正显示到页面上. 操作页面的DOM 最早可以在这里进行

更新

- `beforeUpdate` ：更新前
- `updated` ：更新完毕

keep-alive相关

- `activated` ：被 keep-alive 缓存的组件激活时调用。
- `deactivated` ：被 keep-alive 缓存的组件停用时调用。

销毁

- `beforeDestory` ：销毁前
- `destroyed` ：销毁完毕
  - data 和 methods 此处已消失，无法使用

vue 获取数据在哪个周期函数

理论上，应该在 created 周期中进行网络请求. 因为这是最早的的 methods 与 data 加载完毕的时机.

在 created 发送请求，可以比 mounted 周期发送请求，提前几毫秒的时间拿到数据.

而实际开发中，几毫秒的提前对用户来讲，没有任何差异. 所以 created 和 mounted 发送请求都可以.

兼容性问题

兼容性问题主要分为三大类：

- 操作系统兼容：Mac Windows android iOS...
- 不同品牌浏览器的兼容：Chrome, Firefox, Safari, 毒瘤IE

- 设备分辨率的兼容：大屏幕，小屏幕，手机屏幕，平板屏幕...

参考网址：https://www.cnblogs.com/zhoudawei/p/7497544.html

vue 之 keep-alive

参考网址： https://www.jianshu.com/p/9523bb439950

> keep-alive是一个抽象组件：它自身不会渲染一个DOM元素，也不会出现在父组件链中；使用keep-alive包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们。

一个场景

用户在某个列表页面选择筛选条件过滤出一份数据列表，由列表页面进入数据详情页面，再返回该列表页面，我们希望：列表页面可以保留用户的筛选（或选中）状态。
keep-alive就是用来解决这种场景。当然keep-alive不仅仅是能够保存页面/组件的状态这么简单，它还可以避免组件反复创建和渲染，有效提升系统性能。总的来说，keep-alive用于保存组件的渲染状态。

- 在动态组件中的应用

```
<keep-alive :include="whiteList" :exclude="blackList" :max="amount">
    <component :is="currentComponent"></component>
</keep-alive>
```

- 在vue-router中的应用

```
<keep-alive :include="whiteList" :exclude="blackList" :max="amount">
   <router-view></router-view>
</keep-alive>
```

include定义缓存白名单，keep-alive会缓存命中的组件；exclude定义缓存黑名单，被命中的组件将不会被缓存；max定义缓存组件上限，超出上限使用LRU的策略置换缓存数据。

> 内存管理的一种页面置换算法，对于在内存中但又不用的数据块（内存块）叫做LRU，操作系统会根据哪些数据属于LRU而将其移出内存而腾出空间来加载另外的数据。

vue 的父子传参

- 父传递参数

```
<Son name='xiaoxin' :age="18" />
```

- 子组件

```
<script>
export default {
  props: ['name', 'age'],
  // 或者 规定类型写法
  props: { name: {type: String}, age:{type: Number} }
}
</script>
```

vue 的子父传参

流程：

- 子组件

```
<button v-on:click="$emit('show', 'Hi, petter')">我是按钮</button>
```

- 父组件

```
<Son @show="sayHi" />

<script>
export default {
  methods: {
    sayHi(msg){
      console.log(msg)
    }
  }
}
</script>
```

流程解析：

- 子组件中，点击按钮． `$emit(事件名, 参数)` 触发 show 事件绑定的方法，传入参数．
- show 方法在 父组件中定义． `@show="sayHi"` ，子的 show 方法绑定了 父的 `sayHi`
- 子中的参数通过 show 事件绑定的 `sayHi` 方法传入父中

vue 的兄弟传参

兄弟组件间无法直接通信，通信方式有两种：子传父 + 父传子 和 事件车

- 子传父 + 父传子：此方式效率较低，不推荐

  依赖共同的父组件进行信息的转达．

  假设 A 和 B 组件为兄弟组件，A 要向 B 中传值：

  - 父组件 通过 A 的事件方式传递 父的函数给A
  - A组件 通过 `$emit() 方式` 触发父传入的事件，并传入参数
  - 父组件 收到A 的参数之后，再通过修改 传递给 B组件 的属性．实现B的属性修改

  总结

  - 父和A组件，通过子父传参进行信息交互．
  - 父和B组件，通过父子传参进行信息的交互．
- 事件车：此方式效率高，推荐使用．

  参考：https://blog.csdn.net/qq_42455145/article/details/106466367

  - 向 Vue 的原型中，注入一个 专门负责监听事件的 Vue 实例

    ```
    Vue.prototype.EventBus = new Vue();
    ```

  - A 组件中注册 引入 `EventBus.js` 模块，并向其中注册 事件

    ```
    this.EventBus.$emit('change', msg)
    ```

  - B 组件中注册 `change` 事件的监听

```
this.EventBus.$on('change', changeMsg(msg))

methods:{
  changeMsg(msg){
    //此处就能收到 msg, A组件传入的
  }
}
```

垂直居中在不知道高度时怎么解决

- 方式1：绝对定位

```
parentElement{
  position:relative;
}

childElement{
  position: absolute;
  top: 50%;
  transform: translateY(-50%);
}
```

- 方式2：弹性盒子布局

```
parentElement{
  display:flex;/*Flex布局*/
  display: -webkit-flex; /* Safari */
  align-items:center;/*指定垂直居中*/
}
```

代码管理工具

代码管理工具有早期的 SVN 和 现在的 GIT.

我目前使用的是 Git 工具管理代码. Git是一个开源的分布式版本控制系统。

Git工具的主要功能有：

- 暂存功能，实现新旧代码的对比，代码的回退
- 版本功能，代码形成多个版本，记录每日的工作，快捷的版本回退.
- 分支功能，能够互不影响的并行开发多个不同功能，团队合作.
- 合并功能，快速合并不同的分支 并 解决冲突
- 远程仓库，通过 码云 和 Github 实现代码的云存储. 快速进行团队合作

Git的常用命令有：

- 本地仓库
  - 初始化： `git init`
  - 暂存： `git add 文件名` 或 `git add .`
  - 提交版本： `git commit -m '版本描述'`
  - 分支： `git branch`
  - 合并： `git merge`
- 远程仓库

- 克隆： `git clone 远程仓库地址`
- 刷新： `git fetch`
- 更新： `git pull`
- 上传： `git push`

什么是单页应用

单页应用的全称是 Single Page Application，简称 SPA

通过路由的变更，局部切换网页内容 取代 整个页面的刷新操作.

三大框架 React Vue Angular 均采用单页应用模式.

- 优点：
  1. 用户操作体验好，用户不用刷新页面。
  2. 局部更新，对服务器压力小.
  3. 良好的前后端分离．后端不再负责页面渲染和输出工作.
- 缺点：
  1. 首次加载耗时长，速度慢.
  2. SEO不友好，需要采用 prerender 服务进行完善

Vue的权限管理

参考地址： https://www.jb51.net/article/185275.htm

后台管理系统 一般都会有权限模块，用来控制用户能访问哪些页面 和 哪些数据接口.

整体思路：

后端返回用户权限，前端根据用户权限处理得到左侧菜单；所有路由在前端定义好，根据后端返回的用户权限筛选出需要挂载的路由，然后使用 `addRoutes` 动态挂载路由。

具体思路：

1. 路由定义，分为初始路由和动态路由，一般来说初始路由只有 login，其他路由都挂载在 home 路由之下需要动态挂载.
2. 用户登录，登录成功之后得到 token，保存在 sessionStorage，跳转到 home，此时会进入路由拦截根据 token 获取用户权限列表。
3. 全局路由拦截，根据当前用户有没有 token 和 权限列表进行相应的判断和跳转，当没有 token 时跳到 login，当有 token 而没有权限列表时去发请求获取权限等等逻辑。
4. 使用 Vuex 管理路由表，根据 Vuex 动态渲染侧边栏组件

高度坍塌的解决方式

高度坍塌：在流式布局中十分常见。当父元素没有高度，子元素全部设置float时。
原因：子元素脱离文档流，无法撑开父元素

- 方式1：添加一个div标签到子元素末尾

```
<div style="clear:both"></div>
```

- 方式2：完美方案

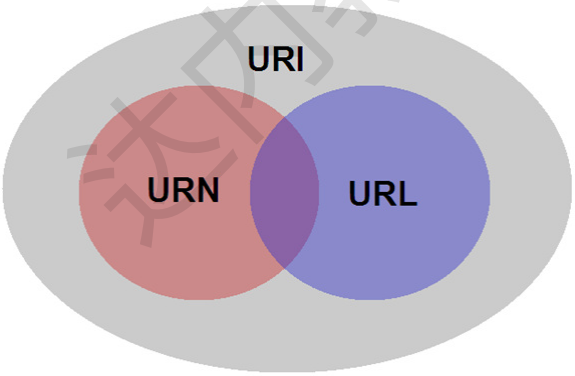
```css
.box:after{
  clear: both;
  content: '';
  display: block;
  height: 0;
  overflow: hidden;
}

.box{ zoom:1; } /** 兼容ie触发hasLayout */
```

Http 的工作过程

参考地址：https://blog.csdn.net/hguisu/article/details/8680808

1. 地址解析
2. 封装 http 请求数据包
3. 封装成 TCP 包，建立 TCP 连接（TCP 的三次握手）
4. 客户机发送请求命令
5. 服务器响应
6. 服务器关闭TCP连接
   - 特殊场景：`keep-alive` 添加此关键词，则可以保持连接.

说出 URL URI URN 的区别



URI：Universal Resource Identifier 统一资源标识符，用来唯一的标识一个资源，是一种语义上的抽象概念。

URL：Universal Resource Locator 统一资源定位符，它是一种具体的URI，即URL可以用来标识一个资源，而且还指明了如何访问到这个资源

URN：Universal Resource Name（统一资源名称）是标准格式的URI，指的是资源而不指定其位置或是否存在。

> 举个容易理解的例子：
>
> URI：国家说，我们要指定一个规则，来找到某个人.
>
> URL：制定地址规则，实现国家需求：xx省xx市xx区xx小区xxx楼xx单元xxx号房间的张三
>
> URN：制定唯一原则，实现国家需求：姓名张三 + 身份证号xxxxx

HTML5 的新特性有哪些

参考地址： https://www.cnblogs.com/binguo666/p/10928907.htmllocal

1. 语义标签
2. 增强型表单
3. 视频和音频
4. Canvas绘图
5. SVG绘图
6. 地理定位
7. 拖放API
8. WebWorker
9. WebStorage
10. WebSocket

闭包及应用场景

闭包函数：声明在一个函数中的函数，叫做闭包函数。

闭包：内部函数总是可以访问其所在的外部函数中声明的参数和变量，即使在其外部函数执行完毕.

```
function funA(){
  var a = 10;  // funA的活动对象之中;
  return function(){   //匿名函数的活动对象;
    console.log(a);
  }
}
var b = funA();
b();  //10
```

闭包的使用场景

- 读取函数内部的变量
- 父函数中的变量始终保持在内存中存活，不会因为函数执行结束而消失.

优点

- 函数中的变量长期存在
- 避免全局变量污染
- 变量成为 私有成员属性的存在

缺点

常驻内存 会增大内存的使用量 使用不当会造成内存泄露

用 css 画一条 0.5px 的线

移动端开发时，由于屏幕是 retina，即高清屏幕. 当写 1px 时，实际的线宽为 2px. 会显得很粗.

此时就有了 0.5px 的需求：主要应对 iPhone

```
<style>
.parent{
  position: relative;
  &:after{
    /* 绝对定位到父元素最低端，可以通过left/right的值控制边框长度或者定义width:100%;*/
    position: absolute;
```

```
    bottom: 0;
    left: 0;
    right: 0;
    /* 初始化边框 */
    content: '';
    box-sizing: border-box;
    height: 1px;
    border-bottom: 1px solid rgba(0, 0, 0, 0.2);
    /* 以上代码，实现了一个边框为1px的元素，下面实现0.5px边框*/
    transform: scaleY(0.5); // 元素y方向缩小为原来的0.5
    transform-origin: 0 0; // CSS属性让你更改一个元素变形的原点。
  }
}
</style>

<div class="parent"></div>
```

用 css 画一个三角形

参考：https://www.cnblogs.com/chengxs/p/11406278.html

这是border 边框放大后的样子

当内容宽高都为0时：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      #triangle-up {
        width: 0;
        height: 0;
        border-left: 50px solid transparent;
        border-right: 50px solid transparent;
        border-bottom: 100px solid red;
```

```
      }
    </style>
  </head>
  <body>
    <div id="triangle-up"></div>
  </body>
</html>
```

跨域问题

原因：浏览器的同源策略

浏览器从一个域名的网页去请求另一个域名的资源时，域名、端口、协议任一不同，都是跨域

> 网址格式：协议名://域名:端口号/...
>
> 例如：http://localhost:8080/... 协议http 域名localhost 端口号8080

常见的解决方案有3种：

- cors
  - 由服务器解决，添加 cors 功能模块.
  - 前端：无操作
- jsonp：利用 script 脚本的 src 不受同源策略限制的特点

  参考教程：https://www.runoob.com/json/json-jsonp.html
  - 服务器：返回特定的 jsonp 格式数据

```php
<?php
header('Content-type: application/json');
//获取回调函数名
$jsoncallback = htmlspecialchars($_REQUEST ['jsoncallback']);
//json数据
$json_data = '["customername1","customername2"]';
//输出jsonp格式的数据
echo $jsoncallback . "(" . $json_data . ")";
?>
```

  - 前端：发送特定的 jsonp 格式数据到服务器

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JSONP 实例</title>
</head>
<body>
<div id="divCustomers"></div>
<script type="text/javascript">
function callbackFunction(result, methodName)
{
    var html = '<ul>';
```

```
    for(var i = 0; i < result.length; i++)
    {
        html += '<li>' + result[i] + '</li>';
    }
    html += '</ul>';
    document.getElementById('divCustomers').innerHTML = html;
}
</script>
<script type="text/javascript"
src="https://www.runoob.com/try/ajax/jsonp.php?
jsoncallback=callbackFunction"></script>
</body>
</html>
```

- 代理proxy
  - vue，angualr 都提供固定的方式设定代理

```
  //vue-cli3.0 里面的 vue.config.js做配置
devServer: {
    proxy: {
        '/rng': {     //这里最好有一个 /
            target: 'http://45.105.124.130:8081',  // 后台接口域名
            ws: true,        //如果要代理 websockets，配置这个参数
            secure: false,  // 如果是https接口，需要配置这个参数
            changeOrigin: true,  //是否跨域
            pathRewrite:{
                '^/rng':''
            }
        }
    }
  }
```

更多的方式：

- html5 新增的 postMessage 特性
- websocket 方式
- location.hash + iframe
- window.name + iframe
- document.domain + iframe

PC端 与 手机端 的自适应

关键词：媒体查询 @media

- Bootstrap 这种框架就是依赖 媒体查询，实现布局随设备宽度自动切换.
- 字体大小 元素大小都使用 rem 或 em 这种相对单位. 不使用px这种固定单位
- 关键标签：`<meta name="viewport" content="width=device-width, initial-scale=1" />`
- 尽量使用流动布局方式
- 根据屏幕宽度 加载不同的css文件
- 图片的自动缩放，例如 `img{ max-width: 100%;}` ，根据不同屏幕分辨率加载不同大小的图片

页面图片很多，访问很慢，怎么优化

1. 开启 web 服务的传输压缩，通过压缩减小图片大小，加快数据传输，提高网页加载速度。
2. 采用 CDN 加速
3. 图片懒加载：刚启动时不加载图片，图片暂时使用默认背景图．页面加载完毕后再加载图片．
4. 使用GIF格式的图片．质量比JPG,PNG略差，但是小很多．对没有特别要求美观的网站比较适用．

微信小程序的生命周期

页面的生命周期

属性	说明
onLoad	生命周期回调-监听页面加载
onShow	生命周期回调-监听页面显示
onReady	生命周期回调-监听页面初次渲染完成
onHide	生命周期回调-监听页面隐藏
onUnload	生命周期回调-监听页面卸载

组件的生命周期

属性	说明
created	在组件实例刚刚被创建时执行
attached	在组件实例进入页面节点树时执行
ready	在组件在视图层布局完成后执行
moved	在组件实例被移动到节点树另一个位置时执行
detached	在组件实例被从页面节点树移除时执行
error	每当组件方法抛出错误时执行

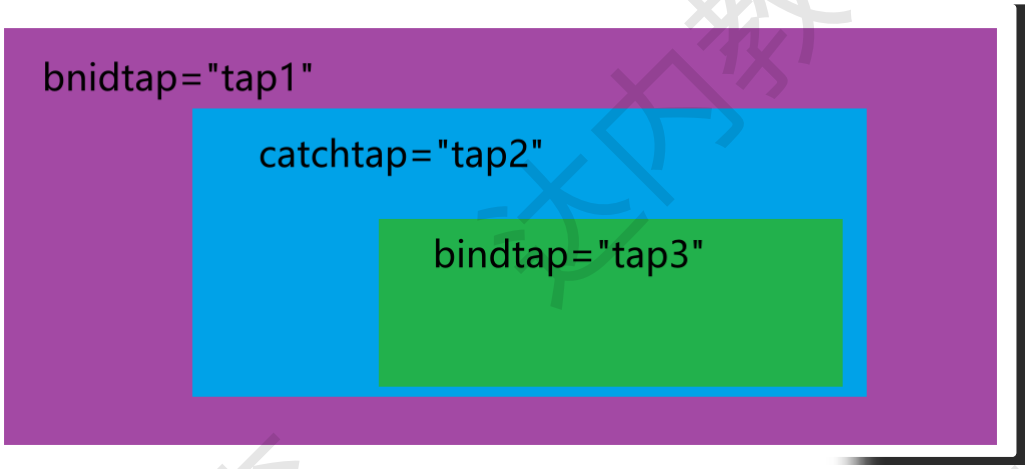
小程序的 bindtap 和 catchtap 区别

bind：允许事件冒泡

catch：阻止事件冒泡

例如下图：

- 点击绿色：触发 tap3 和 tap2
- 点击蓝色：触发 tap2
- 点击紫色：触发 tap1



小程序的文件结构类型

文件	必需	作用
app.js	是	小程序逻辑
app.json	是	小程序公共配置
app.wxss	否	小程序公共样式表

一个小程序页面由四个文件组成，分别是：

文件类型	必需	作用
js	是	页面逻辑
wxml	是	页面结构
json	否	页面配置
wxss	否	页面样式表

vue 路由守卫

参考网址：https://router.vuejs.org/zh/guide/advanced/navigation-guards.html

完整的导航解析流程

1. 导航被触发。
2. 在失活的组件里调用 `beforeRouteLeave` 守卫。
3. 调用全局的 `beforeEach` 守卫。
4. 在重用的组件里调用 `beforeRouteUpdate` 守卫 (2.2+)。
5. 在路由配置里调用 `beforeEnter`。
6. 解析异步路由组件。
7. 在被激活的组件里调用 `beforeRouteEnter`。
8. 调用全局的 `beforeResolve` 守卫 (2.5+)。
9. 导航被确认。
10. 调用全局的 `afterEach` 钩子。
11. 触发 DOM 更新。

12. 调用 `beforeRouteEnter` 守卫中传给 `next` 的回调函数，创建好的组件实例会作为回调函数的参数传入。

- 全局前置守卫

```
const router = new VueRouter({ ... })

router.beforeEach((to, from, next) => {
  // ...
})
```

- 全局解析守卫

  在 2.5.0+ 你可以用 `router.beforeResolve` 注册一个全局守卫。这和 `router.beforeEach` 类似，区别是在导航被确认之前，同时在所有组件内守卫和异步路由组件被解析之后，解析守卫就被调用。

- 全局后置钩子

```
router.afterEach((to, from) => {
  // ...
})
```

- 路由独享守卫

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})
```

- 组件内的守卫

```
beforeRouteEnter (to, from, next) {
  // 在渲染该组件的对应路由被 confirm 前调用
  // 不！能！获取组件实例 `this`
  // 因为当守卫执行前，组件实例还没被创建
},
beforeRouteUpdate (to, from, next) {
  // 在当前路由改变，但是该组件被复用时调用
  // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，
  // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。
  // 可以访问组件实例 `this`
```

```
},
beforeRouteLeave (to, from, next) {
  // 导航离开该组件的对应路由时调用
  // 可以访问组件实例 `this`
}
```

解释 vue 的 nextTick

vue 更新 DOM 是异步操作. `$nextTick()` 可以监听DOM更新完毕的时机.

```
<template>
  <div>
    <h3 id="nn">{{ name }}</h3>
  </div>
</template>

<script>
export default {
  data() {
    return {
      name: "东东",
    };
  },
  mounted() {
    this.name = "然然";

    // 异步渲染机制. 只要 mounted 方法执行完毕后, name 才会更新到DOM
    let el = document.getElementById("nn");
    console.log(el.innerText); // 东东

    this.$nextTick(() => {
      // 这里是DOM 渲染完成后的回调函数
      let el = document.getElementById("nn");
      console.log(el.innerText); // 然然
    });
  },
};
</script>

<style></style>

```

深拷贝 与 浅拷贝? 如何实现深拷贝

> 浅拷贝理解为: 昵称. 比如 张东 东东 东神 东哥 都是一个人呢
>
> 深拷贝裂解为: 克隆体 比如 东哥的 大乔 和 然哥的 大乔 长得一样, 但不是同一个角色.

- 浅拷贝有两种方式

1. 把一个对象里面的所有的属性值和方法都复制给另一个对象

```
let a = { boss: {name:'wenhua'} };
let b = Object.assign({}, a);
b.boss.name = 'WenHua';
console.log(a); // WenHua
```

2. 直接把一个对象赋给另一个对象，使得两个都指向同一个对象。

```
let a = {age: 11};
let b = a;
b.age = 22;
console.log(a);  // 22
```

- 深拷贝

  把一个对象的属性和方法一个个找出来，在另一个对象中开辟对应的空间，一个个存储到另一个对象中。

```
var obj1 = {
    age: 10,
    sex: "男",
    car: ["奔驰", "宝马", "特斯拉", "奥拓"],
    dog: {
        name: "大黄",
        age: 5,
        color: "黑白色"
    }
};

var obj2 = {};//空对象
// 通过函数实现,把对象a中的所有的数据深拷贝到对象b中
// 使用递归函数
function deepCopy(obj,targetObj){
    for (let key in obj){
        let item = obj[key];
        if (item instanceof Array){//if array
            targetObj[key] = [];
            deepCopy(item,targetObj[key]);
        }else if (item instanceof Object){//if object
            targetObj[key] = {};
            deepCopy(item,targetObj[key]);
        }else {//normal attribute
            targetObj[key] = obj[key];
        }
    }
}
deepCopy(obj1,obj2);
console.dir(obj1);
console.dir(obj2);
```

Echarts 中实现区域染色

参考网址： https://echarts.apache.org/examples/zh/editor.html?c=map-usa

使用的数据： https://echarts.apache.org/examples/data/asset/geo/USA.json

原型 与 原型链

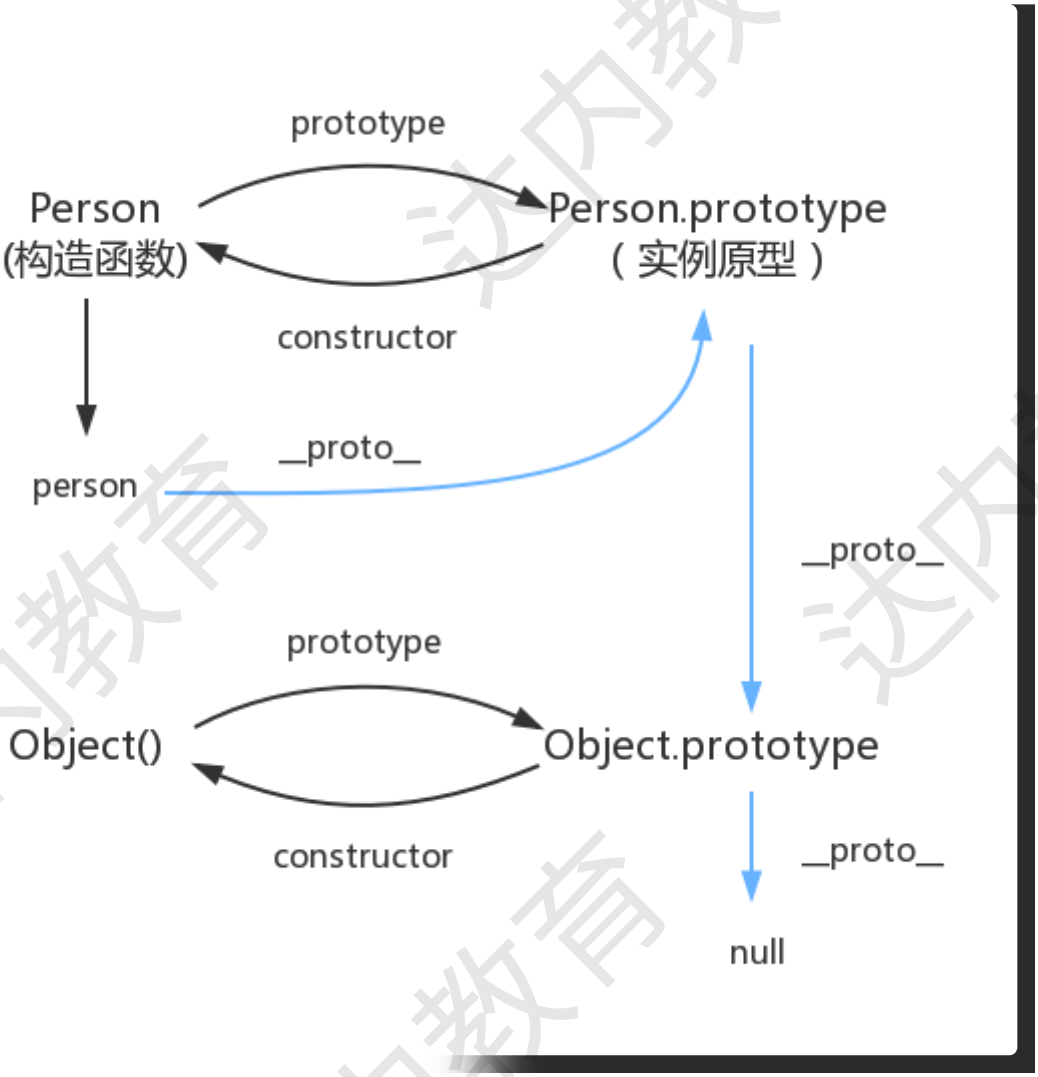
ES6之前并没有引入 class 面向对象的概念，JavaScript 通过构造函数来创建实例.

构造函数：

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

  this.sayName = function () {
    console.log(this.name);
  }
}

var person = new Person('xiaoxin', 32);

console.log(Person.prototype);
```

```
▼Object i                                                    02.js:12
  ▼constructor: ƒ Person(name, age)
      arguments: null
      caller: null
      length: 2
      name: "Person"
    ▶ prototype: {constructor: ƒ}
    ▶ __proto__: ƒ ()
      [[FunctionLocation]]: 02.js:1
    ▶ [[Scopes]]: Scopes[1]
  ▶ __proto__: Object
```

- 原型：每个函数都有一个prototype属性，这个属性指向函数的原型对象。
- 原型链：__proto__ 这是每个对象(除null外)都会有的属性，叫做__proto__，这个属性会指向该对象的原型。



let const var 的差别 与 使用场景

- var：变量提升,无块级作用域概念.
- let：ES6新增，块级作用域；
- const：ES6新增，块级作用域；声明的变量在运行期间不可修改.

箭头函数，可以改变 this 指向吗

箭头函数无法修改this指向.

普通函数 可以通过 apply，call，bind 修改 this 指向

token相关

> 场景：用户登录成功后，需要反复到服务器获取 敏感数据.
>
> 服务器对每次请求都要验证是哪位用户发送的，且用户是否合法，需要反复查询数据库，对数据库造成过大压力.

token的具体流程：

用户登录成功后，在服务器可以查询到此用户的相关信息. 服务器通过一些加密算法 把 用户信息，token 的有效期等，加密成一个字符串. 然后发送给用户. 这个字符串就是 token.

具体加密算法只有服务器知道，服务器可以对 token 进行解密，还原成原始值.

重点：用户每次请求都必须携带 token，服务器直接解密token 就可以知道用户的相关信息，省去查询数据库的操作， 减轻数据库压力！

优势 相较于 cookie：

- 支持跨域访问：cookie是不允许跨域访问的，token支持
- 无状态：token不需要服务器保存任何相关信息，token自身就携带所有值。
- 去耦：不需要绑定特定的身份验证方案
- 更适合移动应用：cookie不支持手机端访问
- 性能：网络传输的过程中，性能更好
- 基于标准化:JWT

缺陷：

- 占带宽：比session_id 大，消耗更多的流量
- 无法在服务端注销：很难解决劫持问题。
- 性能问题：JWT标准消耗更多的 CPU 资源

数组常用方法

参考文档： https://www.cnblogs.com/jinzhou/p/9072614.htm

- map：此方法是将数组中的每个元素调用一个提供的函数，结果作为一个新的数组返回，并没有改变原来的数组
- forEach：此方法是将数组中的每个元素执行传进提供的函数，没有返回值，注意和map方法区分
- filter：此方法是将所有元素进行判断，将满足条件的元素作为一个新的数组返回
- every：此方法是将所有元素进行判断返回一个布尔值，如果所有元素都满足判断条件，则返回true，否则为false
- some：此方法是将所有元素进行判断返回一个布尔值，如果存在元素都满足判断条件，则返回true，若所有元素都不满足判断条件，则返回false
- reduce：此方法是所有元素调用返回函数，返回值为最后结果,传入的值必须是函数类型
- push：此方法是在数组的后面添加新加元素，此方法改变了数组的长度
- pop：此方法在数组后面删除最后一个元素，并返回数组，此方法改变了数组的长度
- shift:此方法在数组后面删除第一个元素，并返回数组，此方法改变了数组的长度
- unshift：此方法是将一个或多个元素添加到数组的开头，并返回新数组的长度
- isArray：判断一个对象是不是数组，返回的是布尔值
- concat:此方法是一个可以将多个数组拼接成一个数组
- toString：此方法将数组转化为字符串
- join：此方法也是将数组转化为字符串
- splice(开始位置， 删除的个数，元素)：万能方法，可以实现增删改

http 状态码

- 200（OK） - 表示已在响应中发出
- 204（无内容） - 资源有空表示
- 301（Moved Permanently） - 资源的URI已被更新
- 303（See Other） - 其他（如，负载均衡）
- 304（not modified）- 资源未更改（缓存）
- 400 （bad request）- 指代坏请求（如，参数错误）
- 404 （not found）- 资源不存在
- 406 （not acceptable）- 服务端不支持所需表示
- 500 （internal server error）- 通用错误响应
- 503 （Service Unavailable）- 服务端当前无法处理请求

http 与 https 区别

http协议和https协议的区bai别：传输du信息安全性不同、连接方zhi式不dao同、端口不同、证书zhuan申请方式不同

- 传输信息安全性不同

  1、http协议：是超文本传输协议，信息是明文传输。如果攻击者截取了Web浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息。

  2、https协议：是具有安全性的ssl加密传输协议，为浏览器和服务器之间的通信加密，确保数据传输的安全。

- 连接方式不同

  1、http协议：http的连接很简单，是无状态的。

  2、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。

- 端口不同

  1、http协议：使用的端口是80。

  2、https协议：使用的端口是443

- 证书申请方式不同

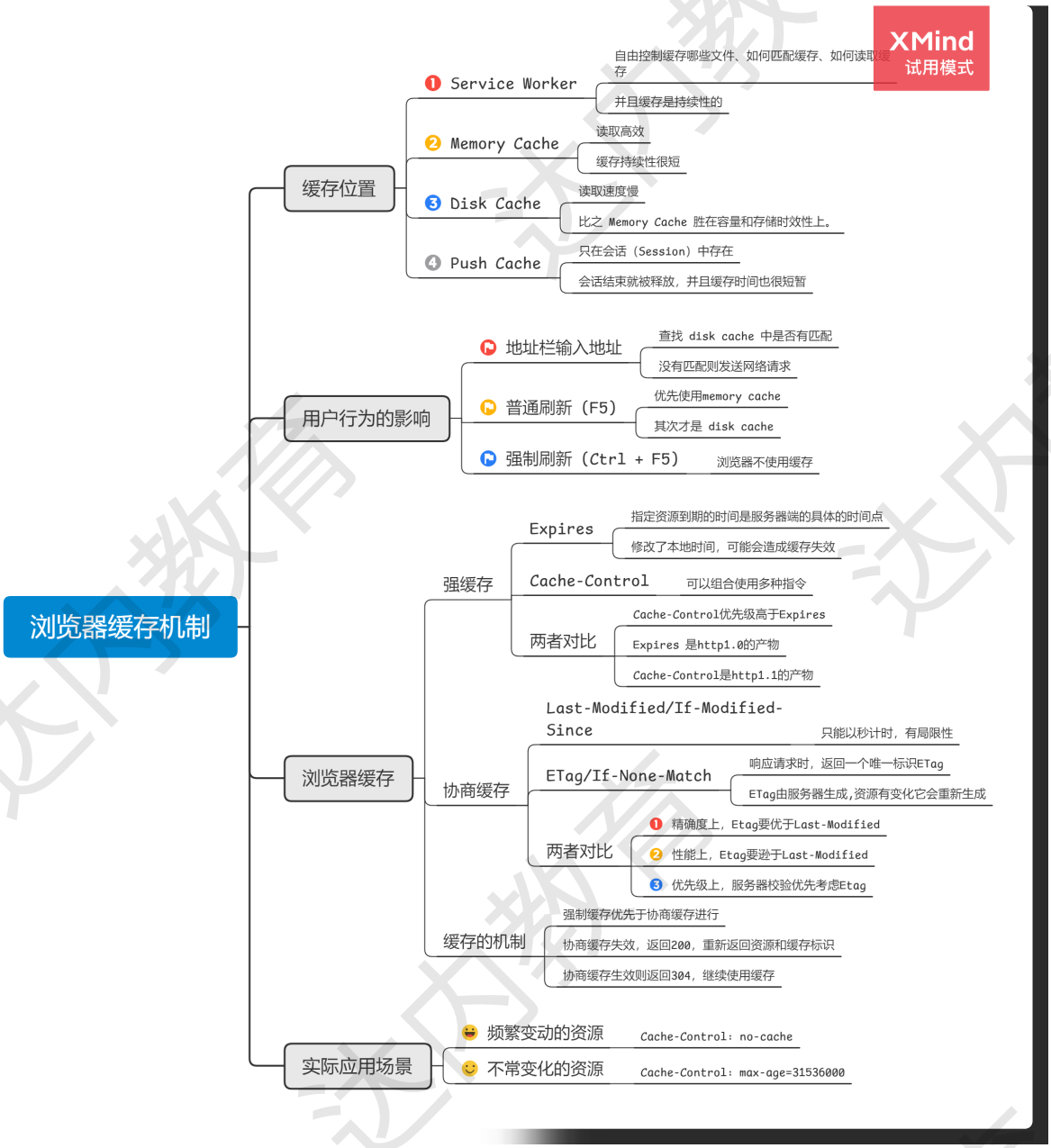
  1、http协议：免费申请。

  2、https协议：需要到ca申请证书，一般免费证书很少，需要交费。

浏览器的缓存方式

缓存可以说是性能优化中简单高效的一种优化方式了。一个优秀的缓存策略可以缩短网页请求资源的距离，减少延迟，并且由于缓存文件可以重复利用，还可以减少带宽，降低网络负荷。

对于一个数据请求来说，可以分为发起网络请求、后端处理、浏览器响应三个步骤。浏览器缓存可以帮助我们在第一和第三步骤中优化性能。



网络安全：csrf

跨站请求伪造（英语：Cross-site request forgery），缩写为 CSRF，是一种劫持受信任用户向服务器发送非预期请求的攻击方式.

原理

> 东东到提款机，插如银行卡 输入密码取钱．此时东东离开提款机忘记拔卡．然然直接用提款机取钱．提款机是不知道取钱人是否为东东本人的.

1. 用户打开浏览器，访问受信任网站A，输入用户名和密码请求登录网站A
2. 在用户信息通过验证后，网站A产生Cookie信息并返回给浏览器，此时用户登录网站A成功，可以正常发送请求到网站A
3. 用户未退出网站A之前，在同一浏览器中，打开一个TAB页访问网站B；
4. 网站B接收到用户请求后，返回一些攻击性代码，并发出一个请求要求访问第三方站点A；
5. 浏览器在接收到这些攻击性代码后，根据网站B的请求，在用户不知情的情况下携带Cookie信息，向网站A发出请求。网站A并不知道该请求其实是由B发起的，所以会根据用户C的Cookie信息以C的权限处理该请求，导致来自网站B的恶意代码被执行。

防御手段

- Cookie 的 SameSite 属性用来限制第三方Cookie，从而减少安全风险.
- 同源检测：Http 请求的 `Origin Header` 和 `Referer Header` 属性
- 在请求地址中添加 token 并验证

webpack的理解

是什么

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

为什么用

- 像sass，JSX等代码虽然极大的提高了开发效率，但是本身并不被浏览器所识别，需要我们对其进行编译和打包，变成浏览器识别的代码
- 模块化(让我们可以把复杂的代码细化为小的文件)
- 优化加载速度(压缩和合并代码来提高加载速度，压缩可以减少文件体积，代码合并可以减少http请求)

主要特性

- 同时支持CommonJS和AMD模块（对于新项目，推荐直接使用CommonJS）；
- 串联式模块加载器以及插件机制，让其具有更好的灵活性和扩展性，例如提供对CoffeeScript、ES6的支持；
- 可以基于配置或者智能分析打包成多个文件，实现公共模块或者按需加载；
- 支持对CSS，图片等资源进行打包
- 开发时在内存中完成打包，性能更快，完全可以支持开发过程的实时打包需求；
- 对source map有很好的支持。

  Source map就是一个信息文件，里面储存着位置信息。也就是说，转换后的代码的每一个位置，所对应的转换前的位置。有了它，出错的时候，除错工具将直接显示原始代码，而不是转换后的代码，这将给开发者带来了很大方便。

set 和 map 数据结构

set 和 map 都是 ES6新增特性

- map映射:也称 dictionary字典. 一个键值结构. 类似于 js 的对象类型.

```
let map = new Map();
map.set("name", "东东");
map.set("age", 22);
map.set("gender", 1);
console.log(map); //Map { 'name' => '东东', 'age' => 22, 'gender' => 1 }
console.log(map.get("name")); //东东
```

- set集合：特点为内部元素不重复. 会自动去重.

```
let a = new Set([1, 1, 2, 2, 3, 3]);
console.log(a); //Set { 1, 2, 3 }
```

vue 的 computed 特性

计算属性就是当其依赖属性的值发生变化时，这个属性的值会自动更新，与之相关的DOM部分也会同步自动更新。

使用场景

在模板中绑定一些数据，这些数据需要经过一些复杂处理之后再展示.

但是模板中只能进行简单逻辑处理，表达式过长 或 逻辑复杂 会变得臃肿，难以阅读及维护.

此时就把处理数据的逻辑放在计算属性中进行.

具体用法

```
<template>
  <div>
    <h3>总价：{{ total }}</h3>
  </div>
</template>

<script>
export default {
  data() {
    return {
      goods: [
        { name: "iPhone12", price: 8999, count: 4 },
        { name: "小米11", price: 3999, count: 2 },
        { name: "Mate40", price: 8000, count: 1 },
      ],
    };
  },
  computed: {
    total() {
      let total = 0;

      this.goods.forEach((item) => {
        total += item.price * item.count;
      });

      return total;
    },
  },
};
</script>

<style></style>

```

vue 的 watch 是否可以监听数组

能监听

- 数组的元素增删：例如 `push` 和 `splice` 操作
- 数组元素内部的变化：必须手动开启 `deep:true` 配置，才能监听到

```
export default {
  data() {
```

```
    return {
      emps: [
        { name: "lucy", age: 22, skills: ["lucy", "lily"] },
        { name: "lucy", age: 22, skills: ["lucy", "lily"] },
        { name: "lucy", age: 22, skills: ["lucy", "lily"] },
      ],
    };
  },
  methods: {
    change() {
      this.emps[0].name = "lala";
      this.emps[0].skills.push(333);
    },
  },
  watch: {
    emps: {
      handler: (xx) => {
        console.log(xx);
      },
      deep: true, //允许监听 内容的变化
    },
  },
};
```

不能监听

- 数组中已有值的替换

```
export default {
  data() {
    return {
      emps: [
        { name: "lucy", age: 22, skills: ["lucy", "lily"] },
        { name: "lucy", age: 22, skills: ["lucy", "lily"] },
        { name: "lucy", age: 22, skills: ["lucy", "lily"] },
      ],
    };
  },
  methods: {
    change() {
      // 此操作, 替换 下标0 的值, 不会被 watch 监听
      this.emps[0] = { name: "222", age: 333 };
    },
  },
  watch: {
    emps(){
      console.log(this.emps)
    }
  },
};
```

防抖 与 节流

参考文档： https://www.cnblogs.com/fs0196/p/12685422.html

日常开发过程中，滚动事件做复杂计算频繁调用回调函数很可能会造成页面的卡顿，这时候我们更希望把多次计算合并成一次，只操作一个精确点，JS把这种方式称为debounce（防抖）和throttle（节流）

- 函数防抖

  当持续触发事件时，一定时间段内没有再触发事件，事件处理函数才会执行一次，如果设定时间到来之前，又触发了事件，就重新开始延时。也就是说当一个用户一直触发这个函数，且每次触发函数的间隔小于既定时间，那么防抖的情况下只会执行一次。

```
function debounce(fn, wait) {
    var timeout = null;      //定义一个定时器
    return function() {
        if(timeout !== null) clearTimeout(timeout);  //清除这个定时器
        timeout = setTimeout(fn, wait);
    }
}
// 处理函数
function handle() {
    console.log(Math.random());
}
// 滚动事件
window.addEventListener('scroll', debounce(handle, 1000));
```

> 效果：页面滚动停止1秒后，才会打印随机数字.
>
> 在滚动过程中并没有持续执行，有效减少了性能的损耗

- 函数节流

  当持续触发事件时，保证在一定时间内只调用一次事件处理函数，意思就是说，假设一个用户一直触发这个函数，且每次触发小于既定值，函数节流会每隔这个时间调用一次
  用一句话总结防抖和节流的区别：防抖是将多次执行变为最后一次执行，节流是将多次执行变为每隔一段时间执行
  实现函数节流我们主要有两种方法：

  - 时间戳

```
var throttle = function(func, delay) {
  var prev = Date.now();
  return function() {
      var context = this;   //this指向window
      var args = arguments;
      var now = Date.now();
      if (now - prev >= delay) {
          func.apply(context, args);
          prev = Date.now();
      }
  }
}

function handle() {
  console.log(Math.random());
}
```

```
17
18  window.addEventListener('scroll', throttle(handle, 1000));
```

- 定时器

```
1  var throttle = function(func, delay) {
2      var timer = null;
3      return function() {
4          var context = this;
5          var args = arguments;
6          if (!timer) {
7              timer = setTimeout(function() {
8                  func.apply(context, args);
9                  timer = null;
10             }, delay);
11         }
12     }
13 }
14 function handle() {
15     console.log(Math.random());
16 }
17 window.addEventListener('scroll', throttle(handle, 1000));
```

ajax 超时断开

当进行前后端通信时，如果响应没有设置结束导致请求一直处于被挂起的状态，或者超出了我们设置的时间，就会发生通信超时。

我们可以通过设置请求的 timeout属性 来设置超时时间：

```
request.timeout = 2000;
```

超时时间必须设置在open方法执行以后，send方法执行之前。

当超时发生时， timeout事件 将会被触发。

```
request.addEventListener("timeout",timeoutHandler);
```

当超时发生以后，我们需要断开通信连接，这时需要使用abort方法：

```
request.abort();
```

综合运用示例：

```
1  var xhr = new XMLHttpRequest();
2  xhr.addEventListener("readystatechange", readyStateChangeHandler);
3  xhr.addEventListener("timeout", timeoutHandler); //侦听超时事件
4
5  xhr.open("POST", "http://10.9.72.236:4010");
6  xhr.timeout = 5000; //设置超时时间
7  xhr.send("a=1&b=2");
8
9  function readyStateChangeHandler(e) {
10   if (xhr.readyState === 4 && xhr.status === 200) {
```

```
    console.log("通信完成并且成功");
  } else if (xhr.readyState === 4) {
    console.log("通信完成，但是通信可能有误");
  } else {
    console.log("通信的过程");
  }
}

function timeoutHandler(e) {
  console.log("超时了");
  xhr.abort(); //断开连接
}
```

严格模式 与 非严格模式的 区别

严格模式 strict mode

使用 `use strict` 指令开启严格模式

```
"use strict";  //整个js代码都是以严格模式执行
//... js 代码
```

- 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;
- 消除代码运行的一些不安全之处，保证代码运行的安全;
- 提高编译器效率，增加运行速度;
- 为未来新版本的Javascript做好铺垫。

常见区别

1. 变量必须先声明 后使用
2. 不能使用 `delete` 关键词删除变量或对象
3. 函数的参数名不能重复
4. 不允许使用 八进制
5. 对象的属性名不能重复
6. arguments 差别

```
"use strict";
function fn(a, obj) {
  arguments[0] = 2;
  arguments[1].b = 2;
  console.log(a); // 严格模式为1; 非严格模式为2
  console.log(obj.b); // 2,因为js中object是地址传递
}
fn(1, { b: 1 });
```

7. `arguments` 不能做变量名 或 函数名

apply bind call 的区别

为 非箭头函数 设置函数体中的 this 对象

```
```

```
function demo(wife, phone) {
  console.log(`${this.name}的${wife}电话是${phone}`);
}

let obj = { name: "然然" };

demo("小乔", "10086"); // undefined的小乔电话是10086

// 然然的小乔电话是10086
demo.apply(obj, ["小乔", "10086"]);
demo.call(obj, "小乔", "10086");

let a = demo.bind(obj, "小乔", "10086");
a()

```

总结：

- apply：函数中的 this 替换成参数1，其余参数放数组中．直接触发函数
- call：函数中的 this 替换成 参数1，其余参数依次摆放．直接触发函数
- bind：替换函数中的 this 指向 并 传入其他参数，返回新的函数．不会直接触发函数！

vue 与 react 的区别

设计思想

- react

  react整体是函数式的思想，把组件设计成纯组件，状态和逻辑通过参数传入，所以在react中，是单向数据流。react在setState之后会重新走渲染的流程，如果shouldComponentUpdate返回的是true，就继续渲染，如果返回了false，就不会重新渲染

- vue

  vue的思想是响应式的，基于是数据可变的，通过对每一个属性建立Watcher来监听，当属性变化的时候，响应式的更新对应的虚拟dom。

总之，react的性能优化需要手动去做，而vue的性能优化是自动的，但是vue的响应式机制也有问题，就是当state特别多的时候，Watcher也会很多，会导致卡顿，所以大型应用（状态特别多的）一般用react，更加可控。

实现方式

- react

  react的思路是all in js，通过js来生成html，所以设计了jsx，还有通过js来操作css

- vue

  vue是把html，css，js组合到一起，用各自的处理方式，vue有单文件组件，可以把html、css、js写到一个文件中，html提供了模板引擎来处理。

代码书写

- react

  采用面向对象方式制作组件，api要求很少．书写比较随意．

- vue

  采用 声明式 写法，通过大量的固定 options，api 生成页面

  - 例如：`methods` ，`data` ，`filter` ，`directive` ，`component` ...

外援

- react

  react本身提供很少的功能，大多数高阶功能都依赖于社区．例如 状态管理要用 redux

- vue

  本身集成了超多功能，使用方便．例如 状态管理的 Vuex

前端的优化方案

主要优化方案分类

- 减少请求次数 和 请求大小
- 代码优化，优化目标：
  - 利用SEO
  - 利于拓展维护
  - 提高性能
- DNS 及 HTTP通信方式的优化

详细方案：

- 尽量减少闭包的使用
- 进行 js 和 css 文件的合并，减少http请求次数，进行可能讲文件压缩，减少请求大小
  - webpack工具会自动实现这种操作
  - 移动端开发过程中，代码量不多，则直接合并 html css js 到一个文件中书写
- 使用字体图标和svg图标，代替传统的png格式
- 减少DOM操作：主要减少DOM的重绘和重排
- js避免 嵌套循环
- 采用图片 懒加载，加快页面启动速度
  - 加载页面时先不加载图片．使用一张背景图占位．等页面加载完毕后，再加载图片．
- 利用浏览器和服务端的缓存技术(304缓存)，把一些不经常变更的资源进行缓存，例如 js 和 css 文件．目的是减少请求大小
- 尽可能使用事件委托来处理绑定操作，减少DOM的频繁操作
  - 事件委托：为父元素添加事件，利用冒泡机制，让父元素处理所有子元素的事件
- 减少 css 表达式的使用
- 减少 css 标签选择器的使用
- css 雪碧图 技术
- 避免重定向（301：资源永久转移/302：暂时转移）
- 减少 cookie 的使用
- 页面数据获取方式 采用异步 和 延迟分批加载
- 页面出现 音视频 标签，让这些资源懒加载．

  方案：只需设置 `preload="none"` ,页面加载完时就会开始加载。
- 数据尽可能使用 json 格式传递． 因为此格式比 `xml` 小
- 进行 js 封装，尽量复用代码．减少代码冗余
- css中设置定位后，最好使用 `z-index` 改变层级．让盒子在不同平面
- css 中尽量减少 filter 属性滤镜的使用
- css 的导入尽量减少 @import 操作，此操作是同步的．而 link 是异步的
- 避免使用iframe

- 开启服务器的 gzip 压缩

手写一个递归函数

```
// 计算阶乘 5 * 4 * 3 * 2 * 1

function jie(n) {
  if (n > 1) {
    return n * jie(n - 1);
  }

  return 1;
}

console.log(jie(5));
```

前后端分离的意义

职责分离

- 后端：
  1. 提供数据和服务
  2. 处理复杂的业务
  3. 关注服务层
  4. 开发和充分利用服务器的性能
- 前端：
  1. 接收数据和服务
  2. 简单处理一些小业务,数据，model，view.
  3. 关注客户端页面渲染，性能，交互
  4. 优化SEO,性能，加载等

多端开发

- 前后端不分离项目 适合 web 开发，提高 SEO 能力.
- 但是目前的业务通常要求一个网站带有 web 和 app 至少两个端.

  此时如果 服务器单独为 app 开发接口，会加大工作量.

  前后端分离后，就不需要为 App 单独增加工作量.

前端工程化

前端工程化是使用软件工程的技术和方法来进行前端项目的开发、维护和管理.

> 早期的非工程化前端开发方式，与小作坊相似：
>
> - 按照个人习惯制作 html 页面
> - 使用 jQuery 等技术添加一些动态效果与数据
> - 随便找个 框架 改一改
>
> 总之：没有一个固定的规矩可以遵循，没有标准化的操作流程．很难保证质量.

前端工程化就是形成一套规矩，把前端网站的制作标准化，大概分为以下措施：

- 模块化

把耦合在一起的大文件 拆分成功能独立的小文件，再进行统一的拼装和加载，这样才能多人协作。

  - 例如 JS 的模块化操作： `commonJS` ， `AMD` ， `CMD`
  - webpack 工具：进行模块的打包
- 组件化

  代码的设计层面，把不同的功能解耦合，设计成可插拔的组件。

  - 相当于：台式机与笔记本的差别。台式机的各个零件都可以随意替换 而 不会影响其他组件
- 规范化

  设定一个规范，让所有参与人员的代码统一风格，便于团队协作与维护。

  - 目录结构的制定
  - 代码规范
  - 前后端接口规范
  - 文档规范
  - 组件管理
  - git分支管理
  - commit 描述规范
  - 定期 Code Review
  - 视觉图标规范
  - ....
- 自动化

> 任何简单机械的重复劳动 都应该让机器自动完成

  - 图标合并:webpack -- 雪碧图
  - 自动化构建：脚手架
  - 自动化部署：脚手架
  - 自动化测试：脚手架
  - ...

get 和 post 的区别

- GET在浏览器回退时是无害的，而POST会再次提交请求。
- GET产生的URL地址可以被Bookmark，而POST不可以。
- GET请求会被浏览器主动cache，而POST不会，除非手动设置。
- GET请求只能进行url编码，而POST支持多种编码方式。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
- GET请求在URL中传送的参数是有长度限制的，而POST么有。
- 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
- GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
- GET参数通过URL传递，POST放在Request body中。

> 参考：https://www.cnblogs.com/logsharing/p/8448446.html

Restful 的请求有哪些方式

RESTFUL是一种网络应用程序的设计风格和开发方式

RESTFUL特点包括：

1、每一个URI代表1种资源；

2、客户端使用GET、POST、PUT、DELETE4个表示操作方式的动词对服务端资源进行操作：GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源；

rem 是什么

rem（font size of the root element）是指相对于根元素的字体大小的单位。简单的说它就是一个相对单位。看到rem大家一定会想起em单位，em（font size of the element）是指相对于父元素的字体大小的单位。它们之间其实很相似，只不过一个计算的规则是依赖根元素一个是依赖父元素计算。

> rem最适合的场景就是 web app. 即在手机端上浏览的网页.
>
> 利用 JS 根据设备自动更改根元素字体大小，就可以实现全局的自动适配
>
> 参考：http://caibaojian.com/web-app-rem.html

写一个验证手机号的正则

- 简单规则：首位 1，第二位 3~9，共 11 位

```
^1[3-9]\d{9}$
```

- 复杂规则：

中国电信号段

> 133、153、173、177、180、181、189、190、191、193、199

中国联通号段

> 130、131、132、145、155、156、166、167、171、175、176、185、186、196

中国移动号段

> 134(0-8)、135、136、137、138、139、1440、147、148、150、151、152、157、158、159、172、178、182、183、184、187、188、195、197、198

中国广电号段

> 192

其他号段

> 14号段部分为上网卡专属号段：中国联通145，中国移动147，中国电信149

虚拟运营商：

> 电信：1700、1701、1702、162
> 移动：1703、1705、1706、165
> 联通：1704、1707、1708、1709、171、167
> 卫星通信：1349、174
> 物联网：140、141、144、146、148

```
/^1(3\d|4[014-9]|5[0-35-9]|6[2567]|7[0-8]|8\d|9[0-35-9])\d{8}$/
```

冒泡排序

参考网址： https://blog.csdn.net/fe_dev/article/details/79600448

```
var arr = [3, 4, 1, 2];
function bubbleSort (arr) {
  var max = arr.length - 1;
  for (var j = 0; j < max; j++) {
    // 声明一个变量，作为标志位
    var done = true;
    for (var i = 0; i < max - j; i++) {
      if (arr[i] > arr[i + 1]) {
        var temp = arr[i];
        arr[i] = arr[i + 1];
        arr[i + 1] = temp;
        done = false;
      }
    }
    if (done) {
      break;
    }
  }
  return arr;
}
bubbleSort(arr);
```