# vue框架

vue是一套用于构建用户界面的渐进式框架

渐进式：因为vue.js它本身的功能很局限，但是可以配合其他的工具来增强能力

vue-router、vuex等

## **vue的核心**

1. 数据驱动视图。修改绑定的数据，就可以对应的更新视图，减少DOM操作，提高开发效率
2. MVVM模式。vue基于MVVM思想来实现数据的双向绑定。（Object.defineProperty()）

M：数据模型（model）

V：视图（view）

VM：数据模型和视图的控制器viewmodel

MVVM：view中数据变化将自动反映到viewmodel上，model中数据变化也将会自动展示在页面上，ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model

## **vue指令**

v-on：用于vue中绑定事件（@）

v-bind：用于vue中绑定属性（：）

v-for：通过数据动态遍历标签

v-model：双向数据绑定

采用数据劫持和发布者-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

## vue常用属性

1. 过滤器 filter

在差值表达式中使用，可以对输出的内容进行格式的转换

1. 计算属性 computed

根据data当中的数据，经过一定的逻辑处理，得到一项新数据

当data中的数据发生变化的时候，计算属性也会更新

计算属性也是响应式数据，改变的时候也会驱动视图的更新

当多次获取计算属性的时候，处理逻辑不会重新执行，因为有缓存

1. 属性检测 watch

监控一个值的变换,并调用因为变化需要执行的方法。通过watch动态改变关联的状态。

Vue常见优化手段

1、使用key

通过循环生成的列表，应给每个列表项一个稳定且唯一的key，有利于在列表变动时，尽量少的删除、新增、改动元素

2、使用冻结的对象 Object.freeze(obj)

冻结的对象不会被响应化

3、使用函数式组件

参考vue2官方文档中的 函数式组件

4、使用计算属性 Computed

5、非实时绑定的表单项

当使用一个v-model绑定一个表单项时，当用户改变表单项的状态时，也会随之改变数据，从而导致vue发生重渲染（rerender），会带来性能的开销

<!-- 使用v-model -->

    <input type="text"

        @keypress.enter="addTodo"

        v-model="newTodoContent"

        class="todo-content"

        placeholder="input todo"

    />

可以通过使用lazy或不使用v-model的方式解决该问题，但是这样可能会导致在某一个时间段内数据和表单项的值是不一样的

<!-- 使用v-model.lazy -->

    <input type="text"

        @keypress.enter="addTodo"

        v-model.lazy="newTodoContent"

        class="todo-content"

        placeholder="input todo"

    />

6、保持对象引用稳定

function hasChanged(x, y) {

    if (x === y) {

        return x === 0 && 1 / x !== 1 / y   // +0 === -0  true

    } else {

        return x === x || y === y  // NaN !== NaN  true

    }

}

7、v-show替代v-if

8、延迟装载

使用H5中的requestAnimationFrame() 分批渲染内容，延迟装载组件，让组件按照指定的先后顺序依次一个一个渲染出来

export default function (maxFrameCount) {

    return {

        data() {

            return {

                frameCount: 0

            }

        },

        mounted() {

            const refreshFrameCount = () => {

                requestAnimationFrame(() => {

                    this.frameCount++

                    if (this.frameCount < maxFrameCount) {

                        refreshFrameCount()

                    }

                })

            }

            refreshFrameCount()

        },

        methods: {

            // 判断目前画的次数 有没有 超过某个次数

            defer(showInFrameCount) {

                return this.frameCount >= showInFrameCount

            }

        }

    }

}

9、使用keep-alive

10、长列表优化

11、打包体积优化

# Object.defineProperty

对目标对象进行拦截操作。直接在一个对象上定义一个新属性，或者修改一个已存在的属性。

接收三个参数：Object.defineProperty(属性所在的对象, “属性的名字”, { 描述符对象 })

1、属性所在的对象

2、属性的名字

3、描述符对象

描述符对象：-

数据属性：

1. configurable：能否通过delete删除属性从而重新定义属性
2. enumerable：能否通过for in 循环遍历属性
3. writable：能否修改属性的值
4. value：包含这个属性的数据值

let student = {

    name: 'lisi',

    age: 24

}

Object.defineProperty(student, 'name', {

    configurable: false, // 不可配置

    enumerable: false, // 不可枚举

    writable: false, // 不可修改

    value: 'lisi' // 读取属性值

})

访问器属性：

1. get：在读取属性时调用的函数
2. set：在写入属性时调用的函数

let student = {}

Object.defineProperty(student, 'name', {

    get(){ return 'lisi' },

    set(v){ s1.age = v }

})

注意点：

1. value和get是同一个作用，在创建实例对象的时候只能使用其一。
2. writable和set是同一个作用，也只能用其一

两者区别：

1. Object.defineProperty拦截的是对象的属性，会改变原对象；Proxy拦截整个对象，通过new一个新对象，不会改变原对象。

2、Proxy可以监听Object.defineProperty监听不到的操作。如监听数组、监听对象属性的新增、删除等

# Vuex

## vuex是什么

vuex是一个专门为vue.js应用程序开发的状态管理模式。它对所有组件的状态进行一个集中式的存储管理，并以相应的规则保证状态以一种可预测的方式发生变化。

## Vuex的5个属性

1. State

公共数据源。vuex使用单一状态树，每一个应用最好仅包含一个store实例，不建议直接修改state的值，最好是通过commit方法调用mutation任务进行修改，方便后期数据的追踪

1. Mutations

定义方法动态修改state中的数据。不建议包含逻辑业务处理，处理一些同步任务，通过使用$store.commit()调用mutations中的方法

1. Actions

定义方法来执行异步任务，一些较为复杂的逻辑代码。view层通过$store.dispath()调用Actions任务

1. Getter

类似vue实例中的计算属性特点，用来过滤规范改造数据

1. Module

模块。项目特别复杂的时候使用，每一个模块拥有自己的state，mutation，Action，getter，代码逻辑更加清晰

# Vue3

1. 响应式原理的改变：

使用Proxy代替Object.defineProperty()来实现响应式数据

1. 新增一个Composition API：

将零散的data，methods代码重新组合，一个功能的代码放一块，并且可以单独拆分出函数

1. 新增一个setup：

作为Composition API 的入口点。接收两个参数setup(props, context){}，这个函数的返回是一个对象，。这个函数是在beforeCreate之前创建的，所以没有this

1. 生命周期

setup里面的生命周期：onBeforeMount  -> onMounted  -> onBeforeUpdate  -> onUpdated  -> onBeforeUnmount  -> onUnmounted  -> onErrorCaptured

1. provide、inject

提供依赖注入，实现祖孙级组件通信

1. 全局API

vue2：在vue2中使用全局api的话都是挂载到Vue原型上。Vue.config、Vue.prototype等

import Vue from 'vue'

import App from './App.vue'

const app = new Vue(App)

app.$mount()

vue3：通过CreateApp创建vue实例

import { createApp } from 'vue'

import App from './App.vue'

createApp(App).mount('#app')

# 深拷贝和浅拷贝

深拷贝：将一个对象完整的从一个内存中拷贝出来，从堆内存中开辟一个新的区域存放新对象，且修改新对象不会影响原对象。

浅拷贝：创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是引用类型，拷贝的就是指向内存的地址。如果其中一个对象改变了这个引用类型的值，就会影响到另一个对象。

深拷贝的实现：

1. **递归**

文本

描述已自动生成

+

1. **Object.assgin()：对象中没有嵌套对象时，才可以实现深拷贝，否则为浅拷贝**



# 数组操作的方法及返回值

1. push()：向数组的末尾添加一个或多个元素；返回的是数组的新长度。  
   
2. unshift()：向数组的开头添加一个或多个元素；返回的是数组的新长度。  
   
3. shift()：删除数组的第一个元素，并返回被删除的（即第一个元素）内容。  
   
4. pop()：删除数组的最后一个元素，并返回被删除的（即最后一个元素）内容。  
   图表

   描述已自动生成
5. shift、pop如果是空数组则返回underfind。  
   图片包含 文本

   描述已自动生成
6. join()：数组转为[字符串](https://so.csdn.net/so/search?q=%E5%AD%97%E7%AC%A6%E4%B8%B2&spm=1001.2101.3001.7020)，是通过指定的分隔符进行分隔。返回一个字符串。  
   手机屏幕的截图

   描述已自动生成
7. reverse() ：方法用于颠倒数组中元素的顺序。返回顺序颠倒后的数组。  
   手机屏幕的截图

   描述已自动生成
8. slice() ：截取数组想要选定的元素。slice(开始截取位置，结束截取位置)。返回的是截取区间的数组。（左闭右开）  
   

如果只写了一个参数，即将从该参数位置开始截取，直到数组最后一项。  
手机屏幕的截图

中度可信度描述已自动生成

1. splice() : splice(a, b)从位置a删除b个项。返回的是被删除的b个项组成的数组

图表, 散点图

描述已自动生成

1. concat()：将两个数组合并为同一个数组,并返回一个新数组。  
   图片包含 图表

   描述已自动生成
2. sort()：数组排序

钟表的特写

描述已自动生成

# Vue-Router

它是vue.js官方的路由管理器，它和vue.js是深度集成的，适用于构建单页面应用

在使用vue.js开发应用的时候，它是把整个应用拆分成了各个独立的组件，而我们要做的就是把路由映射到各个组件，再通过vue-router把组件渲染到正确的地方。

通过<router-link>和<router-view>

SPA：单一页面应用程序，只有一个完整的页面；加载页面时，不会加载整个页面，意思就是更新视图而不重新请求页面

实现原理：hash模式、history模式

# Hash和history

## Hash模式

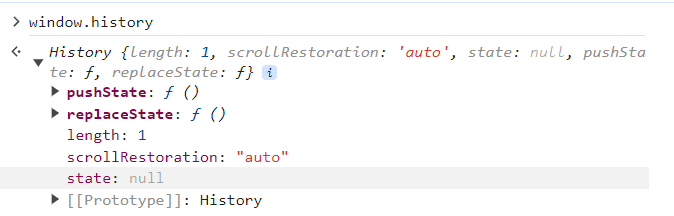
定义：hash 模式是一种把前端路由的路径用 # 拼接在真实 url 后面的模式。当 # 后面的路径发生变化时，浏览器并不会重新发起请求，而是会触发 onhashchange 事件。使用window.location.hash属性以及浏览器的onhashchange事件，实现监听浏览器hash值变化（window.addEventListener(‘hashChange’, ()=>{}），执行相应的切换网页操作

特点：

1. hash指的是url地址中#及后面的字符，用来做页面跳转定位。
2. 改变hash值，不会重新加载页面
3. 页面刷新时，发送的请求中的url是不带 #和后面的字符
4. 监听window的hashchange事件，当#后面发生改变时，通过window.location.hash来获取和设置hash值
5. Location.hash值的变化会直接反应到地址栏中

## History模式

**定义：**history API 是 H5 提供的新特性，允许开发者**直接更改前端路由**，即更新浏览器 URL 地址而**不重新发起请求**。（需要后台做处理）



特点：

1. window.history属性指向History对象，表示当前窗口的浏览历史。当发生改变时，只会改变页面的路径，不会刷新页面。
2. History对象保存了当前窗口访问过的所有页面网址。

属性：

1. History.length：当前窗口访问过的网址数量
2. History.state：History堆栈最上层的状态值

History.pushState(object, title, url)：

History.replaceState(object, title, url):

用户在历史中添加一条记录。pushState方法不会触发页面刷新，只是导致History对象发生变化，地址栏会变化

Object：对象。通过pushState方法将该对象内容传递到新页面中

Title：标题。传一个空字符串

url：新的网址。必须与当前页面处在同一个域

popstate事件：当History对象发生变化时触发。

1. 调用pushState()方法和replaceState()方法不会触发该事件。
2. 当用户点击浏览器前进按钮或后退按钮时，触发该事件
3. JS调用History.back()、History.forward()、History.go()方法时触发。
4. 页面第一次加载的时候，浏览器不会触发popstate事件

使用：

为popstate事件指定回调函数，参数是一个event事件对象，state属性指向当前的state对象。

window.addEventListener('popstate', function(e){

    // e.state相当于history.state

    console.log('state: ' + JSON.stringify(e.state));

    console.log(history.state);

})



# 原型和原型链

## Prototype 原型

每一个函数（箭头函数除外）都有一个叫做prototype的属性，这个属性指向的是一个对象（原型对象）：用来存放实例对象的公有属性和公有方法

function Person(name, age){

this.name = name;

    this.age = age

}

Person.prototype.species = '人类'

Person.prototype.say = function(){

    console.log('Hello!');

}

var person1 = new Person('张三', 18)

console.log(person1.species); // 人类

console.log(person1.\_\_proto\_\_ === Person.prototype); // true

## \_\_proto\_\_ 原型链

某个对象查找属性，自己和原型对象上都没有的话，就会继续到原型对象的原型对象Object.prototype上面去查找，形成一个链条一样的结构，这个结构，就是原型链

function Person() {}

var person = new Person()

console.log(person.\_\_proto\_\_ === Person.prototype); // true

## constructor 构造器

function Person() {}

console.log(Person === Person.prototype.constructor); // true

// 当要写多个方法的时候，修改了原来的原型对象，给原型对象prototype赋值的是一个对象，就需要手动的利用constructor指回原来的构造函数

Star.prototype = {

    constructor: Star,  // 利用constructor指向构造函数

    sing: function () {

        console.log('唱歌');

    },

    movie: function () {

        console.log('电影')

    }

}

}

## 实例与原型

function Person() {}

Person.prototype.name = '张三'

var person = new Person()

person.name = '李四'

console.log(person.name); // 李四

delete person.name

console.log(person.name); // 张三

1. 当一个对象查找属性和方法时会从自身查找,如果查找不到则会通过\_\_proto\_\_指向被实例化的构造函数的prototype
2. 隐式原型也是一个对象,是指向我们构造函数的原型
3. 除了最顶层的Object对象没有\_\_proto\_，其他所有的对象都有\_\_proto\_\_,这是隐式原型
4. 隐式原型\_\_proto\_\_的作用是让对象通过它来一直往上查找属性或方法，直到找到最顶层的Object的\_\_proto\_\_属性，它的值是null,这个查找的过程就是原型链

5、原型对象prototype === 对象原型\_\_proto\_\_

// 构造函数是使用了new关键字的函数，用来创建对象，所有函数都是Function()的实例

// 原型对象是用来存放实例对象的公有属性和公有方法的一个公共对象，所有原型对象都是Object()的实例

// 原型链又叫隐式原型链，是由\_\_proto\_\_属性串联起来，原型链的尽头是Object.prototype

// 公共属性定义到构造函数里面，公共方法放到原型对象上面

# New操作符

1. 创建了一个新对象 var newObj = new object()
2. 设置原型，将对象的原型设置为函数的prototype对象 （newObj.\_\_proto\_\_ === object.prototype）
3. 让函数的this指向这个对象，执行构造函数的代码
4. 判断函数的返回值类型，如果是值类型，返回创建的对象；如果是引用类型，就返回这个引用类型的对象

# 箭头函数和普通函数

1. 箭头函数全都是匿名函数；普通函数可以有匿名函数，也可以有具名函数
2. 箭头函数不具有prototype原型对象，所以不能用于构造函数；普通函数可以用于构造函数，创建对象实例
3. 箭头函数中this在定义的时候就已经确定了，不再改变；在普通函数中，this总是指向调用它的对象，如果用作构造函数，它指向创建的对象实例
4. 箭头函数不具有arguments对象；普通函数调用后都具有一个arguments对象，用来存储实际传递的参数

# 闭包

## 变量作用域

变量作用域分为两种：全局变量、局部变量

在js中，函数内部可以直接读取全局变量，而在函数外部无法读取函数内部的变量

在函数内部声明变量的时候，需要用var来声明，否则声明的为全局变量

## 什么是闭包

闭包：就是能够读取函数 内部变量 的函数，可以简单理解为定义在一个函数内部的函数；在本质上，闭包是将函数内部和函数外部连接起来的一个桥梁

function fn1(){

    var value = 999

    function fn2(){

        console.log(value); // 999

    }

    fn2()

}

fn1()

## 闭包的作用

1）、读取函数内部的变量，让变量始终保持在内存中，不会在外部函数调用后被自动清除

2）、延长外部函数变量对象的生命周期

3）、从函数外部访问函数内部的私有变量

## 4、使用注意

1、闭包会使得函数中变量都会保存在内存中，导致内存消耗很大，影响网页的性能

2、闭包会在父函数外部，改变父函数内部变量的值，如果把父函数当做对象（Object）使用，把闭包当做它的公用方法（Public Method）,把内部变量当做它的私有属性，这时不能随便改变父函数内部变量的值

# Promise

## 什么是Promise

Promise是一个解决异步的方法，本质上是一个构造函数，可以用它实例化一个对象。对象身上有resolve、reject，all方法，race方法，原型上有then、catch方法。Promise对象有三种状态：pending（进行中）、fulfilled（成功）、rejected（失败）。创造Promise实例后，会立即执行。

## 2、Promise对象的特点

1、对象的状态不受外界的影响。Promise对象代表一个异步操作，只有异步操作的结果可以决定Promise对象的状态，其他任何操作都不会改变这个状态。

2、状态一旦改变，就不会再变，且只会发生其中一个变化；同时调用resolve函数和reject函数，默认第一个调用。pending => fulfilled 、pending => rejected

## 3、Promise的优点

1、支持链式调用：可以将异步操作以同步的方式显示出来，避免回调地狱

回调地狱：多个回调函数层层嵌套，外层回调函数异步返回的结果是内层回调函数执行的条件。

2、指定函数回调的方式更灵活：在异步任务执行完毕之后再指定回调

## 4、Promise的缺点

1、一旦建立就会立即执行，无法中途取消

2、如果不给它设置回调函数，内部的抛出错误无法显示

3、当处于pending状态的时候，无法得知当前进行到哪一步

# Map和Set

## Map对象

1. 初始化Map时传入数据

let myMap = new Map([['name', '张三'], ['age', 20],[400, ‘InvalidParameter’]])

1. 插入数据

myMap.set('name', '猪猪侠')

1. 获取长度

myMap.size

1. 获取值

myMap.get('name')

1. 删除值

myMap.delete(‘'name'’)

1. 判断值是否存在

myMap.has('name') // 返回为bool值

特点：

1. Map对象这种数据结构和对象类似，都以键值对的形式存储数据，即key-value
2. Map对象存储的数据是有序的，而平常使用的对象时无序的。
3. Map对象的键值可以是任意类型，而平常使用的对象只能使用字符串作为键

状态码字典

let errors = new Map([

[400, 'InvalidParameter'],

[404, 'Not found'],

[500, 'InternalError']

]);

console.log(errors);

{400 => 'InvalidParameter', 404 => 'Not found', 500 => 'InternalError'}

## Set对象

1. 初始化Set对象

let mySet = new Set(['张三', 12, true])

1. 插入数据

mySet.add('飞猪')

1. 获取长度

mySet.size

1. 获取值

mySet.forEach((item)=>{

console.log(item)

})

1. 删除值

mySet.delete(12)

1. 判断值是否存在

mySet.has('张三') // 返回为bool值

特点：

1. Set对象是一个类数组对象，长得很像数组
2. Set对象存储的值是不重复的，所以用Set来实现数组去重

let arr = [1, 1, 2, 2, 2, 3, 4, 6, 7]

[...new Set(arr)] // [1, 2, 3, 4, 6, 7]

1. Set对象存储的数据不是键值对的形式，可以存储任何类型的数据

## Set和Map区别

1. Set和Map数据结构都是以构造函数的形式出现，所以通常使用new Set() 或者 new Map() 的形式进行初始化
2. Map对象初始化的值为一个二维数组，Set对象初始化的值为一维数组
3. Map对象和Set对象都不允许键重复
4. Map对象的键是不能改的，但是值能改；Set对象只能通过迭代器来更改值

# JS数组中ForEach()方法和map()方法

## 返回值

forEach()方法返回undefined，需要传递一个空数组来创建一个新的转换后的数组；

map()方法返回一个包含已转换元素的新数组

const arr = [1, 2, 3, 4, 5]

// forEach()方法

const foreach = []

arr.forEach(item => foreach.push(item \* item))

console.log(foreach); // [ 1, 4, 9, 16, 25 ]

// map()方法

const map = arr.map(item => item \* item)

console.log(map); // [ 1, 4, 9, 16, 25 ]

## 链接其他方法

forEach()是一个终端方法，所以不能与其他方法链接；而map()方法可以与其他方法链接在一起。

// 数组元素平方和

const number = [1, 2, 3, 4, 5]

// forEach() 方法

const sq = []

let sumSqFor = 0

number.forEach(item => sq.push(item \* item))

sq.forEach(item => sumSqFor += item)

console.log(sumSqFor); // 55

// map()方法

const sumSqMap = number.map(it => it \* it).reduce((total, value) => total + value)

console.log(sumSqMap); // 55

## 中断遍历

forEach()和map()都不能使用break进行中断遍历，只能使用for循环才可

# js数据存储LocalStorage和sessionStorage

## localStorage

localStorage的生命周期是永久的，除非手动去清除，否则永远都在，存储大小为5MB，仅在客户端浏览器上存储，不参与服务器的通信。

以键值对的方式进行存储，通常存储为字符串

// 设置localStorage保存到本地，第一个为变量名，第二个为值

localStorage.setItem('name', '张三')

// 获取数据

localStorage.getItem('name')

// 删除保存的数据

localStorage.removeItem('name')

// 清除所有保存的数据

localStorage.clear()

## sessionStorage

在当前会话下有效。引入了一个“浏览器窗口”的概念。sessionStorage是在同源的同窗口中，始终存在的数据，只要浏览器不关闭，即使是刷新或者进入同源的另一个页面，数据还在。同时打开“独立”的窗口，即使是同一个页面，sessionStorage的对象也是不同的。关闭窗口后sessionStorage就会被销毁。

以键值对的方式进行存储，通常存储为字符串

// 设置sessionStorage保存到本地，第一个为变量名，第二个为值

sessionStorage.setItem('name', '张三')

// 获取数据

sessionStorage.getItem('name')

// 删除保存的数据

sessionStorage.removeItem('name')

// 清除所有保存的数据

sessionStorage.clear()

## webStorage

webStorage是以字符串的格式进行存储的，所以不能直接存储对象和数组类型，需要使用JSON.stringify()将对象转换成字符串。

# Webpack中的核心：Loader 和 Plugin

webpack是一个运行在nodejs端的静态模块打包工具，将所有后缀为.js的文件打包成一个bundle.js。主要功能就是代码的分割、模块化，提取公共代码。

因为webpack本身只能编译js文件，loader 就是帮助 webpack 将不同类型的文件转换为 webpack 可识别的模块

## Loader

加载器。在Webpack中将一切文件都视为模块，而webpack原生是只能解析js文件，这个时候需要用到loader。它的作用就是让webpack拥有加载和解析非JavaScript文件的能力。

loader它只专注于转化文件（transform）这一个领域，完成压缩，打包，语言翻译

loader运行在打包文件之前（loader为在模块加载时的预处理文件）

## Plugin

插件。Plugin可以扩展webpack的功能，让webpack具有更多的灵活性。在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果

plugin不仅只局限在打包，资源的加载上，还可以打包优化和压缩，重新定义环境变量等

plugins在整个编译周期都起作用

# 地址栏输入url回车

1. 输入URL
2. DNS解析
3. 发起TCP连接（三次握手）

第一次握手：

客户端发送syn包(Seq=x)到服务器，并进入SYN\_SEND状态，等待服务器确认

第二次握手：

服务器收到syn包，必须确认客户的SYN（ack=x+1），同时自己也发送一个SYN包（Seq=y），即SYN+ACK包，此时服务器进入SYN\_RECV状态

第三次握手：

客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=y+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手

1. 关闭TCP连接（四次挥手）

第一次挥手：

客户端发送一个FIN包给服务器，用来关闭客户端到服务器的数据传送。

第二次挥手：

服务器收到FIN包，发送一个ACK给客户端并且带上自己的序列号seq

第三次挥手：

服务器发送一个FIN包给客户端，用来关闭服务器到客户端的数据传送

第四次挥手：

客户端收到FIN包，发送一个ACK给服务器。当服务器收到了客户端发出的确认，进入CLOSED状态。

1. 发送HTTP请求

构建HTTP请求报文，并通过TCP协议中发送到服务器指定端口。

请求报文：由请求行、请求头、请求正文组成

请求方法：POST、GET、PUT

GET：产生一个TCP数据包。浏览器会把http、header和data一起发送出去，服务器响应200（返回数据）

POST：产生两个TCP数据包。浏览器先发送header，服务器响应一段时间之后，浏览器再发送data，服务器响应200（返回数据）

1. 服务器处理请求并返回HTTP报文（状态码、响应报头、响应报文）
2. 浏览器解析并渲染页面
3. 连接结束

# 性能优化

1. 加载优化（减少http请求）

1）、合并图片

2）、合并压缩css样式表和js脚本

3）、去掉不必要的请求

4）、预加载（loading）

2、图片优化

1）、使用png格式图片

2）、进行图片的延迟加载

3）、避免img、iframe标签的src属性为空

3、使用公共的css样式

4、减少DOM节点

5、SEO优化

1）、标题字数不要过长

2）、描述中要包括目标关键词

3）、网站代码尽量简洁

# Computed和Watch

1. 计算属性有缓存，当缓存的依赖发生变化时，重新计算，返回一个新值
2. 计算属性有getter和setter，默认为getter.监听属性不分set和get
3. 计算属性用来同步执行计算数据，监听属性可以有异步事件

Computed：如果一个数据需要经过复杂的计算使用

Watch：如果一个数据需要被监听，并且对数据做一些操作就用watch

# Vue生命周期

## vue的生命周期是什么

vue有一个完整的生命周期，就是从开始创建、初始化数据、编译模板、挂载Dom -> 渲染、卸载等一系列过程。这个过程就是vue的生命周期

## vue的钩子

1、beforeCreate（创建前） ：数据观测和初始化事件还未开始，此时 data 的响应式追踪、event/watcher 都还没有被设置，也就是说不能访问到data、computed、watch、methods上的方法和数据。

2、created（创建后） ：实例创建完成，实例上配置的 options 包括 data、computed、watch、methods 等都配置完成，但是此时渲染得节点还未挂载到 DOM，所以不能访问到 `$el` 属性。

3、beforeMount（挂载前） ：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。此时还没有挂载html到页面上。

4、mounted（挂载后） ：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。

5、beforeUpdate（更新前） ：响应式数据更新时调用，此时虽然响应式数据更新了，但是对应的真实 DOM 还没有被渲染。

6、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。此时 DOM 已经根据响应式数据的变化更新了。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。

7、beforeDestroy（销毁前） ：实例销毁之前调用。这一步，实例仍然完全可用，`this` 仍能获取到实例。

8、destroyed（销毁后） ：实例销毁后调用，调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务端渲染期间不被调用。

# 父子组件传值

1. 父传子

父：v-bind绑定一个变量（v-bind:变量名 = ‘值’）

子：props接收

1. 子传父

子：this.$emit("getList",this.urlLists)

父：在子组件中绑定一个方法@getList="getList"，用来接收子组件传过来的方法，再在methods中调用该方法getList(){}

1. ref/refs

在DOM元素上添加ref=”属性名”，通过 this.$refs.属性名 来获取

# $nextTick

$nextTick 是在下次 DOM 更新循环结束之后执行延迟回调，在修改数据之后使用 $nextTick，则可以在回调中获取更新后的 DOM

# Keep-alive

Keep-alive是vue内置的一个组件，作用就是用来缓存不活动的组件。组件在进行切换的时候，会被销毁，当有需求要做到切换组件时组件不被销毁，可以使用keep-alive。

<keep-alive>

   <router-view />

</keep-alive>

# Slot插槽

Slot插槽是vue组件的封装者提供的能力。允许开发者在封装组件时，把不确定的、希望由用户指定的部分定义为插槽。

1. 普通插槽

<slot></slot>

1. 具名插槽

子组件：定义插槽名称

<slot name="XXX"></slot>

父组件：插入到插槽的内容

<template v-slot:XXX>

    {/\* 放在插槽中的内容 \*/}

</template>

1. 作用域插槽

在子组件的slot上动态绑定一个值（:key=’value’），父组件通过v-slot:name = ‘values’将值赋值给values，最后通过{{values.key}}获取数据

//子组件 ： (假设名为：ebutton)

<template>

  <div class= 'button'>

      <button>按钮</button>

      <slot name='one' :value1='child1'>默认值1</slot>//绑定child1数据

      <slot :value2='child2'> 默认值2 </slot>  //绑定child2数据

  </div>

</template>

//父组件：（引用子组件 ebutton）

<template>

  <div class= 'app'>

     <ebutton>

        // 通过v-slot的语法 将插槽 one 的值赋值给slotonevalue

        <template v-slot:one = 'slotonevalue'>

           {{ slotonevalue.value1 }}

        </template>

        // 同上，由于子组件没有给slot命名，默认值就为default

        <template v-slot:default = 'slottwovalue'>

           {{ slottwovalue.value2 }}

        </template>

     </ebutton>

  </div>

</template>

# ES6常见语法

1. let命令、块级作用域、const命令（变量必须要先声明，再使用）

let 命令：定义的变量可以改变值，不可重复声明

const 命令：静态变量（常量），不可以被修改，不可重复声明（只读），声明时需赋值

var命令：定义的变量都是全局作用域，变量提升到作用域最顶端

块级作用域：let命令定义的变量只在当前的代码块生效，作用域之外不能访问

1. 模板字符串

反引号（``）、变量（${}）

const msg = 'Hello World'

const str = `你好 ${msg} 再见`

console.log(str);

1. 解构赋值

从数组和对象中提取值，对变量进行赋值

// 数组解构（有序，取值由位置决定）

let [x, y, z] = [1, 2, 3]

// 对象解构（无序，变量与属性同名）

let {name: myName, age} = {name: 'jack', age: 12}

// 字符串解构

let strr = 'Good'

let [a, b, c, d] = strr

1. 函数

函数参数的默认值（允许参数有默认值）

function teach(x, y = 'world') {

    console.log(x, y);

}

teach('Hello') // Hello world

teach('Hello', 'China') // Hello China

rest参数（形式为 …变量名 ），只能作为最后一个参数

function add(...values) {

    let sum = 0

    for(var val of values) { sum+=val }

    return sum

}

add(2, 3, 5) // 10

name属性——返回改函数的函数名

function foo(){}

console.log(foo.name); // 'foo'

// 如果为匿名函数，返回实际的函数名

var f = function (){}

console.log(f.name);  // 'f'

// 如果是具名函数，返回具名函数的函数名

var bar = function baz(){}

console.log(bar.name);  // 'baz'

箭头函数——没有自己的this，不可以当做构造函数（不能使用new命令）

var sum = (num1, num2) => {

    return num1 + num2

}

const arr1 = [1, 2, 4, 2, 3]

var arr = arr1.sort((a, b) => b - a) // [1, 2, 2, 3, 4]

function Timer() {

    this.s1 = 0;

    this.s2 = 0;

    // 箭头函数 —— this指向的是 绑定定义时所在的作用域(Timer函数)

    setInterval(() => this.s1++, 1000);

    // 普通函数 —— this指向的是 运行时所在的作用域(全局对象)

    setInterval(function () {

        this.s2++;

    }, 1000);

}

var timer = new Timer();

setTimeout(() => console.log('s1: ', timer.s1), 3100);  // s1: 3

setTimeout(() => console.log('s2: ', timer.s2), 3100);  // s2: 0

1. 数组

扩展运算符（ … ）—— 将数组（对象）转为用逗号分隔的参数序列

function add(x, y){

    return x + y

}

const arr = [2, 4]

add(...arr)  // 6

扩展运算符的应用

1. 复制数组

const a1 = [1, 2]

const a2 = [...a1]  // [1, 2]

1. 合并数组（浅拷贝）——原数组改变，新数组也会改变

const a1 = [1, 2]

const a2 = [3, 4]

const a3 = [...a1, ...a2]  // [1, 2, 3, 4]

1. 与解构赋值结合

cosnt [first, ...rest] = [1, 2, 3, 4]

// first => 1

// rest => [2, 3, 4]

// 若将扩展运算符用于数组赋值，只能放在参数的最后一位

cosnt [...rest, first] = [1, 2, 3, 4]

// 报错

1. 将字符串转成数组

[...'hello']

// ['h', 'e', 'l', 'l', 'o']

Array.from()——将两类对象转为数组（类对象、可遍历对象）（有length属性的对象）

let arrayLike = {

    '0': 'a',

    '1': 'b',

    '2': 'c',

    length: 3

};

Array.from(arrayLike)  // [ 'a', 'b', 'c' ]

Array.of()——将一组值，转换为数组，若没有参数，则返回空数组

Array.of(2, 3, 4)  // [2, 3, 4]

Array.of(2, 3, 4).length  // 3

实例方法：find()、findIndex()、findLast()、findLastIndex()

// find() 返回 从前往后找出第一个符合条件的数组成员

[1, 2, -4, -3, 5].find((x) => x < 0)  // -4

// findIndex()  返回 从前往后找出第一个符合条件的数组成员的 下标index

[1, 2, -4, -3, 5].findIndex((x) => x < 0)  // 2

// findLast() 返回 从后往前找出第一个符合条件的数组成员

[1, 2, -4, -3, 5].findLast((x) => x < 0)  // -3

// findLastIndex()  返回 从后往前找出第一个符合条件的数组成员的 下标index

[1, 2, -4, -3, 5].findLastIndex((x) => x < 0)  // 3

实例方法：entries()、keys()、values()——遍历数组，可以用for…of进行遍历

// entries() 遍历键值对

for(let item of ['a', 'b'].entries()) {

    console.log(item);

}

// [0, 'a']

// [1, 'b']

// keys() 遍历键名

for(let index of ['a', 'b'].keys()) {

    console.log(index);

}

// 0

// 1

// values() 遍历键值

for(let value of ['a', 'b'].values()) {

    console.log(value);

}

// a

// b

实例方法：includes() —— 判断数组中是否包含给定的值，返回true、false

[1, 2, 3].includes(3) // true

[1, 2, 3].includes(0) // false

实例方法：flat()、flatMap()

// flat(x)  不影响原数组

// 将嵌套的数组拉平，变成一维数组，返回新数组

[1, 2, [3, 4]].flat()  // [1, 2, 3, 4]

// 参数 x 为整数，表示拉平的层数，默认为1(可不写)

[1, [2, 3, [4, 5]]].flat(2) // [1, 2, 3, 4, 5]

// 若不论嵌套多少层，都变成一维数组，则参数为 Infinity

[1, [2, [3, [4]]]].flat(Infinity) // [1, 2, 3, 4]

// flatMap() 只能展开一层数组，返回新数组，不影响原数组

// 对原数组的每一个成员执行一个函数，然后对返回值组成的数组执行flat()

[2, 3, 4].flatMap((x) => [x, x \* 2])

// 相当于 [[2, 4], [3, 6], [4, 8]].flat()

// [2, 4, 3, 6, 4, 8]

1. Set和Map数据结构
2. Set —— 类似于数组，成员的值都是唯一的，没有重复值

// Set 本身为构造函数，用来生成Set数据结构

const s = new Set()

const arr = [1, 2, 2, 3, 4, 4, 5]

arr.forEach(x => s.add(x))   // 通过add()方法，向Set结构加入成员

s  // { 1, 2, 3, 4, 5 }

// Set 可用于数组去重，也可用于字符串重复字符去重

// 数组：[...new Set(array)]

// 字符串：[...new Set('aabbcc')].join('')  --> 'abc'

const set = new Set(arr)

console.log([...set]);  // [1, 2, 3, 4, 5]

1. Set实例的属性

Set.prototype.constructor   // 构造函数，默认就是Set函数

Set.prototype.size          // 返回Set实例的成员总数

1. Set实例的方法——操作方法（用于操作数据）、遍历方法（用于遍历成员）

操作方法：-----------

const set = new Set()

Set.prototype.add(value)  // 添加某个值，返回Set结构本身

set.add('width', 'height')  // Set: { 'width', 'height' }

Set.prototype.delete(value)  // 删除一个值，返回布尔值，表示是否删除成功

set.delete('width')         // true

Set.prototype.has(value)   // 判断该值是否为Set成员，返回布尔值

set.has('height')           // true

Set.prototype.clear()     //  清除所有成员，没有返回值

set.clear()

// Array.from()方法  可以将Set结构转为数组

const items = new Set([1, 1, 2, 2, 3])   //  去重：Set: { 1, 2, 3 }

const array = Array.from(items)    //  [ 1, 2, 3 ]

遍历方法：-----------

Set.prototype.keys()：返回键名的遍历器——Set结构的键名就是键值

let set = new Set(['green', 'red', 'blue'])

for(let item of set.keys()) {

    console.log(item)

}

// green

// red

// blue

Set.prototype.values()：返回键值的遍历器

let set = new Set(['green', 'red', 'blue'])

for(let item of set.values()) {

    console.log(item);

}

// green

// red

// blue

Set.prototype.entries()：返回键值对的遍历器

let set = new Set(['green', 'red', 'blue'])

for(let item of set.entries()) {

    console.log(item);

}

// ['green', 'green']

// ['red', 'red']

// ['blue', 'blue']

Set.prototype.forEach()：使用回调函数遍历每个成员

let set = new Set([1, 4, 9])

set.forEach((key, value) => console.log(key + ' : ' + value))

// 1 : 1

// 4 : 4

// 9 : 9

1. 遍历的应用

// 扩展运算符(...)内部使用 for...of遍历 用于数组去重

let arr = [1, 1, 2, 2, 3, 4]

let newArr = [...new Set(arr)]   // [1, 2, 3, 4]

1. Map——类似于对象，也是键值对的集合，键可以为各种类型的值（包括对象）

const map = new Map()

const obj = { p: 'Hello' }

map.set(obj, 'content') // 将对象obj作为map的一个键名，'content'作为键值

map.set('foo', true)

// Map: {

//     {

//         key: { p: 'Hello' },

//         value: 'content'

//     },

//     {

//         key: 'foo',

//         value: true

//     }

// }

map.has(obj)  // true

map.get(obj)  // 'content'

map.delete(obj) // true

map.size  // 2

map.clear()  // 清除所有成员，没有返回值

Map.prototype.keys()：返回键名的遍历器

const map = new Map([

    ['F', 'no'],

    ['T', 'yes']

])

for(let key of map.keys()){

    console.log(key);

}

// 'F'

// 'T'

Map.prototype.values()：返回键值的遍历器

const map = new Map([

    ['F', 'no'],

    ['T', 'yes']

])

for(let value of map.values()){

    console.log(value);

}

// 'no'

// 'yes'

Map.prototype.entries()：返回所有成员的遍历器

const map = new Map([

    ['F', 'no'],

    ['T', 'yes']

])

for(let item of map.entries()){

    console.log(item);

}

// 'F' 'no'

// 'T' 'yes'

Map 转为 数组

const map = new Map()

map.set(true, 7)

map.set({foo: 3}, ['abc'])

const newMap = [...map]   // [ [ true, 7], [ {foo: 3}, ['abc]] ]

数组 转为 Map

new Map([

    [true, 7],

    [ {foo: 3}, ['abc']]

])

// Map {

//     true => 7,

//     Object {foo: 3} => ['abc']

// }

1. 对象的扩展

方法的name属性

const person = {

    sayName(){}

}

person.sayName.name  // 'sayName'

获取——属性的描述符

let obj = { foo: '123' }

Object.getOwnPropertyDescriptor(obj) // 获取属性描述符

// {

//     configurable:true,  是否可修改描述符

//     enumerable:true,    是否可遍历

//     writable:true,      是否可重写属性值

//     value:"123",        读取值

// }

设置——属性的描述符

// 设置对象的属性描述符

let obj = {

    a: 101

}

Object.defineProperty(obj, 'a', {

    value: 10,

    writable: false,

    enumerable: false,

    configurable: false

})

obj.a  // 10

解构赋值（浅拷贝——拷贝的对象中含有嵌套对象）

let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 }

// x  1 y  2

// z  { a: 3, b: 4}

// 解构赋值要求等号右边为对象，否则报错

let { ...z } = null  // 报错

// 解构赋值必须为最后一个参数，否则报错

let { x, ...y, ...z } = { x: 1, { a: 2}, { b: 3} }  // 报错

1. 对象的新增方法

Object.is()：比较两个值是否严格相等（===）

Object.is('foo', 'foo')  // true

Object.is({}, {})  // false

Object.assign()：对象的合并，将源对象的所有可枚举属性，复制到目标对象（target）

const target = { a: 1}

const obj1 = { b: 2 }

const obj2 = { c: 3 }

Object.assign(target, obj1, obj2)

target // { a: 1, b: 2, c: 3 }

// 嵌套对象时遇到同名属性，将该属性值全部替换

const target = { a: { x: 1, y: 2 } }

const obj = { a: { x: 'haha' } }

Object.assign(target, obj)

target  // { a: { x: 'haha' } }

// 处理数组

const target = [1, 2, 3]

const arr = [4, 5]

Object.assign(target, arr)

target   // [4, 5, 3]

Object.getOwnPropertyDescriptors()：返回指定对象的所有自身属性的描述对象

const obj = {

    foo: '123'

}

Object.getOwnPropertyDescriptors(obj)

// {

//     foo: {

//         configurable: true,

//         enumerable: true,

//         writable: true,

//         value: '123'

//     }

// }

Object.setPrototypeOf()：设置一个对象的原型对象，返回参数对象本身

Object.setPrototypeOf(object, prototype)  // 将 prototype对象 设为 object对象 的原型

// 等同于下面的函数

function setPrototypeOf(obj, proto) {

    obj.\_\_proto\_\_ = proto

    return obj

}

Object.getPrototypeOf()：和Object.setPrototypeOf()方法配套，用于读取一个对象的原型对象

Object.getPrototypeOf(obj)

Object.keys()：遍历对象，返回含有对象所有可遍历属性的键名的数组

var obj = { foo: 'bar', baz: '24'}

Object.keys(obj)  // [ 'foo', 'baz' ]

Object.values()：遍历对象，返回含有对象所有可遍历属性的键值的数组

var obj = { foo: 'bar', baz: '24'}

Object.values(obj)  // [ 'bar', '24' ]

Object.entries()：遍历对象，返回含有对象所有可遍历属性的键值对的数组

var obj = { foo: 'bar', baz: '24' }

Object.entries(obj)  // [ 'foo', 'bar' ], [ 'baz', '24' ]

Object.fromEntries()：为Object.entries()方法的逆操作，将一个键值对数组转为对象

Object.fromEntries([

    ['foo', 'bar'],

    ['baz', '24']

])

// { foo: 'bar', baz: '24'}

Object.hasOwn()：静态方法，判断是否为自身的属性，返回true、false

// Object.hasOwn(参数1, 参数2)

// 参数1 为判断的对象，参数2 为属性名

const foo = Object.create({ a: 123 });

foo.b = 456;

Object.hasOwn(foo, 'a') // false

Object.hasOwn(foo, 'b') // true

1. 字符串的方法

includes()：判断字符串中，是否包含某个字符

let str = 'I need you'

console.log(str.includes('need')); // true

startsWith()：判断字符串，是否以某个字符开头

let str = 'I need you'

console.log(str.startsWith('I')); // true

endsWith()：判断字符串，是否以某个字符结束

let str = 'I need you'

console.log(str.endsWith('you')); // true

padStart()：补全字符串

// 当字符串不够指定长度时，在头部补全字符串

'x'.padStart(5, 'ab')  // 'ababx'

// 当补全字符串和原字符串长度之和超过最大长度，截去超过位数的补全字符串

'xx'.padStart(6, '12345') // 1234xx

// 若省略第二个参数，默认使用空格补全长度

'xx'.padEnd(4) // '  xx'

padEnd()：当字符串不够指定长度时，在尾部补全字符串

// 当字符串不够指定长度时，在尾部补全字符串

'x'.padEnd(5, 'ab')  // 'xabab'

// 当补全字符串和原字符串长度之和超过最大长度，截去超过位数的补全字符串

'xx'.padEnd(6, '12345') // xx1234

// 若省略第二个参数，默认使用空格补全长度

'xx'.padEnd(4) // 'xx  '

trimStart()：消除字符串头部的空格，返回新的字符串，不修改原字符串

trimEnd()：消除字符串尾部的空格，返回新的字符串，不修改原字符串

let str = '  abc  '

trim(str)  // 'abc'

str.trimStart(str)  // 'abc  '

str.trimEnd(str)  // '  abc'

1. 数值

Number.isFinite()：检查一个数值是否为有限的（finite），返回true/false

Number.isFinite(10) // true

Number.isFinite(NaN)  // false

Number.isNaN()：检查一个值是否为 NaN

Number.isNaN(NaN)  // true

Number.isNaN(10) // false

Number.parseInt()：取整

Number.parseInt('12.34#') // 12

Number.parseFloat()：取小数

Number.parseFloat('12.34#') // 12.34

Math.trunc()：去除一个数的小数部分，返回整数部分

Math.trunc(4.6) // 4

Math.trunc(true) // 1

Math.sign()：判断一个数为正数（返回1）、负数（返回-1）、0（返回0）

Math.sign(5) // 1

Math.sign(-5) // -1

Math.sign(0) // 0

1. 运算符的扩展

指数运算符（ \*\* ）

2\*\*3  // 8

2\*\*4  // 16

链判断运算符（ ?. ）

// 链式逐层判断值是否存在

const firstName = message?.body?.user?.firstName || ‘default’;

// 判断对象方法是否存在，如果存在立即执行

interator.return?.()

Null判断运算符（ ?? ）--- 多和链判断运算符（ ?. ）配合使用

// Null判断运算符( ?? )

// 只有当运算符左侧的属性的值为 null、undefined ,运算符右侧的值生效

const firstName = message?.body?.user?.firstName ?? 'hello';

// 或运算符( || )

// 当运算符左侧的属性的值为 null、undefined、空字符串、false、0 ，

// 运算符右侧的值生效

const headerText = response.settings.headerText || 'Hello world!'

// && || ??  多运算符同时使用时，必须要用括号表明优先级，否则报错

lhs && (middle ?? rhs);

(lhs ?? middle) && rhs;

逻辑赋值运算符（ ||= &&= ??= ）

// 先进行逻辑运算，再根据运算结果视情况进行赋值运算

// 或赋值运算符

x ||= y   ==>   x || (x = y)

// 与赋值运算符

x &&= y   ==>   x && (x = y)

// Null赋值运算符

x ??= y   ==>   x ?? (x = y)

1. Symbol
2. 什么是Symbol

原始数据类型：Symbol---表示独一无二的值

其他数据类型：String、Number、Boolean、Null、undefined、Object、BigInt

// String  Number  Boolean  undefined  Null  Object

// BigInt

// Symbol

// Symbol 值 通过Symbol()函数生成

// Symbol()函数不能使用new命令，否则报错(生成的Symbol值为原始类型的值，不是对象)

let s = Symbol()

typeof s  // 'symbol'

// Symbol()函数可以接收一个字符串作为参数，参数用于描述

let s1 = Symbol('foo')   // s1 这个值的描述就是 foo

let s2 = Symbol('foo')

let s3 = Symbol('bar')

s1  // Symbol(foo)

s2  // Symbol(foo)

s1 === s2  // false

s3  // Symbol(bar)

1. Symbol.prototype.description —— 用于获取 Symbol值 的描述

const sym = Symbol('foo')

sym.description  // 'foo

1. 作为属性名的Symbol

// 每一个Symbol值都是不相等的

// 将Symbol值作为标识符，用于对象的属性名，就不会出现同名的属性

let mySymbol = Symbol()

let a = {}

a[mySymbol] = 'Hello'

Object.defineProperty(a, mySymbol, { value: 'Hello'})

// a[mySymbol]  'Hello'

// Symbol值作为对象属性名时，不能用点运算符

// 点运算符后面为字符串，导致属性名为字符串，而不是一个Symbol值

const mySymbol = Symbol()

const a = {}

a.mySymbol = 'Hello'

a[mySymbol]  // undefined

a['mySymbol']  // 'Hello'

1. 属性名的遍历

Object.getOwnPropertySymbols()

// Object.getOwnPropertySymbols()  循环遍历获取Symbol属性名的方法

// for…in、for…of和Object.getOwnPropertyNames()都不能获取到Symbol属性名

// 返回一个数组，包含获取指定对象的所有的Symbol属性名

const obj = {}

let a = Symbol('a')

let b = Symbol('b')

obj[a] = 'Hello'

obj[b] = 'World'

const objectSymbols = Object.getOwnPropertySymbols(obj)

objectSymbols  // [ Symbol(a), Symbol(b) ]

Reflect.ownKeys()

// Reflect.ownKeys()  返回所有类型的键名，包括常规键名和Symbol键名

let obj = {

    [Symbol('my\_key')]: 1,

    enum: 2,

    nonEnum: 3

}

Reflect.ownKeys(obj)   //  [ 'enum', 'nonEnum', Symbol(my\_key) ]

1. Proxy

// var proxy = new Proxy(target, handler)

// target 要拦截的目标对象  handler 配置对象，用来定制拦截行为

var proxy = new Proxy({}, {

get: function (target, propKey) {

// target 目标对象  propKey 要访问的属性

        return 35

    }

})

proxy.time  // 35

proxy.name  // 35

proxy.title // 35

Proxy实例作为其他对象的原型

var proxy = new Proxy({}, {

    get: function(target, handler) {

        return 35

    }

})

let obj = Object.create(proxy)  // 将proxy对象作为obj对象的原型

obj.time = 35 //obj对象上没有time属性，根据原型链，会在原型对象上读取该属性

Proxy实例的方法

get(target，propKey)：拦截某个属性的读取

// get(target, propKey)

// 拦截某个属性的读取 target为目标对象 propKey为属性名

var person = {

    name: '张三'

}

var proxy = new Proxy(person, {

    get: function(target, propKey) {

        if(propKey in target) {

            return target[propKey]

        } else {

            return new ReferenceError('defalut')

        }

    }

})

proxy.name  // '张三'

proxy.age   // 抛出一个错误

person.age  // undefined

set(target，propKey，propValue)：拦截某个属性的赋值操作

// set(target, propKey, propValue)  拦截某个属性的赋值操作

// target为目标对象  propKey为属性名  propValue为属性值

let descript = {

    set: function(target, propKey, propValue) {

        if(propKey === 'age') {

            if(!Number.isInteger(propValue)) {

                throw new TypeError('The age is not integer')

            }

            if(propValue > 200) {

                throw new RangeError('The age can not')

            }

        }

        target[propKey] = propValue

        return true

    }

}

let person = new Proxy({}, descript)

person.age = 100

person.age  // 100

person.age = 'yong' // 报错

person.age = '300'  // 报错

apply(目标对象，目标对象的上下文对象（this），目标对象的参数数组)：拦截函数的调用

// 执行proxy()函数，就会被apply方法拦截

var twice = {

    apply: function (target, ctx, args) {

        return Reflect.apply(...arguments) \* 2

    }

}

function sum(left, right) {

    return left + right

}

var proxy = new Proxy(sum, twice)

proxy(1, 2) // 6

proxy.call(null, 5, 6) // 22

proxy.apply(null, [9, 10]) // 38

has(目标对象，需查询的属性名)

// has(target, key)  target为目标对象  key为属性名

// 判断对象中是否具有某个属性，其操作就是  in运算符

var handler = {

    has(target, key) {

        if(key[0] === '\_') {

            return false

        }

        return key in target

    }

}

var target = { \_prop: 'foo', prop: 'bar'}

var proxy = new Proxy(target, handler)

'\_prop' in proxy  // false

'prop' in proxy   // true

construct(目标对象，构造函数的参数数组， new命令作用的构造函数)

/\*\*

 \* construct()方法  用于拦截new命令  返回必须为对象  接收以下三个参数

 \*\* target：目标对象

 \*\* args：构造函数的参数数组

 \*\* newTarget：创造实例对象时，new命令作用的构造函数

 \*/

const p = new Proxy(function () { }, {

    construct: function (target, args) {

        console.log('called: ' + args.join(', '));

        return { value: args[0] \* 10 }

    }

})

(new p(1)).value

// 'called: 1'

// 10

1. Reflect
2. Reflect对象和Proxy对象一样，用来操作对象，有以下几个目的

* 将Object对象的一些明显属于语言内部的方法（Object.defineProperty），放到Reflect对象上
* 修改某些Object方法的返回结果，让其变得合理

Object.defineProperty(obj, name, desc)  // 在无法定义属性时，报错

Reflect.defineProperty(obj, name, desc) // 返回false

* 让Object操作都变成函数行为

// 某些命令式的Object操作

name in obj

delete obj[name]

// Reflect将这些操作变成了函数行为

Reflect.has(obj, name)

Reflect.deleteProperty(obj, name)

* Reflect对象的方法和Proxy对象的方法一一对应，只要是Proxy对象的方法，就能在Reflect对象上找到对应的方法。

// 每一个Proxy对象的拦截操作(get、delete、has)，内部都调用对应的Reflect方法

var loggedObj = new Proxy(obj, {

    get(target, name) {

        console.log('get');

        return Reflect.get(target, name)

    },

    deleteProperty(target, name) {

        console.log('delete');

        return Reflect.deleteProperty(target, name)

    },

    has(target, name) {

        console.log('has');

        return Reflect.has(target, name)

    }

})

1. 静态方法

* Reflect.get(target, name)

// Reflect.get(target, name) 查找并返回target对象的name属性

var myObject = {

    foo: 1,

    bar: 2,

    get baz(){

        return this.foo + this.bar

    }

}

Reflect.get(myObject, 'foo')  // 1

Reflect.get(myObject, 'bar')  // 2

Reflect.get(myObject, 'baz')  // 3

* Reflect.set(target, name, value)

// Reflect.set(target, name, value)  设置target对象的name属性等于value

var myObject = {

    foo: 1,

    set bar(value) {

        return this.foo = value

    }

}

myObject.foo  // 1

Reflect.set(myObject, 'foo', 2)

myObject.foo  // 2

Reflect.set(myObject, 'bar', 3)

myObject.foo  // 3

* Reflect.has(target, name)

// Reflect.has(target, name)  判断target对象中是否存在name属性

var myObject = {

    foo: 1

}

Reflect.has(myObject, 'foo')  // true

* Reflect.deleteProperty(target, name)

// Reflect.deleteProperty(target, name)  删除target对象中的name属性

const myObj = { foo: 'bar'}

Reflect.deleteProperty(myObj, 'foo')  // true

* Reflect.defineProperty(target, propertyKey)

// Reflect.defineProperty(target, propertyKey, desc)  为对象定义属性

function myDate() {

    /\* ... \*/

}

Reflect.defineProperty(myDate, 'now', {

    value: () => Date.now()

})

* Reflect.ownKeys(target)

// Reflect.ownKeys(target)  用于返回target对象的所有属性

var myObject = {

    foo: 1,

    bar: 2,

    [Symbol.for('baz')]: 3

}

Reflect.ownKeys(myObject)  // ['foo', 'bar', Symbol(baz)]

1. 使用Proxy实现观察者模式

* 观察者模式——函数自动观察数据对象，一旦对象发生变化，函数就会自动执行

// person对象时观察目标，print函数时观察者

// 一旦数据对象发生变化，print就会自动执行

const person = {

    name: '张三',

    age: 20

}

function print(){

    console.log(`${person.name}, ${person.age}`);

}

observe(print)

person.name = '李四'

// 输出

// 李四，20

1. Promise
2. Promise对象是一个构造函数，用于解决异步编程的方案

* 对象的状态不受外界的影响：pending（进行中）、fulfilled（已成功）、rejected（已失败），只有异步操作的结果，可以决定当前是哪一种状态，其他的任何操作都无法改变这个状态。
* 状态一旦改变，就不会再变：pending 🡺 fulfilled、pending 🡺 rejected

1. 构造一个Promise实例

const promise = new Promise(function (resolve, reject) {

    //  .. do something

    if (/\* 异步操作成功 \*/) {

        resolve(value)

    } else {

        reject(error)

    }

})

* 异步加载图片

function loadImageAsync(url) {

    return new Promise(function (resolve, reject) {

        const image = new Image()

        image.onload = function () {

            resolve(image)

        }

        image.onerror = function () {

            reject(new Error('Could not load image'))

        }

        image.src = url

    })

}

* Promise新建后就会立即执行

let promise = new Promise(function(resolve, reject){

    console.log('Promise');

    resolve()

})

// Promise.resolve.then()  将任务加入到微队列中

promise.then(function(){

    console.log('Resolve');

})

console.log('Hi');

// 打印顺序

// Promise

// Hi

// Resolve

1. Promise.prototype.then()

// Promise.prototype.then()

// then()方法定义在原型对象Promise.prototype上

// Promise对象状态为成功，调用then()方法指定的回调函数

getJSON('/posts.json').then(function (json) {

    return json.post

}).then(function (post) {

    // ...

})

1. Promise.prototype.catch()

// Promise.prototype.catch()

// catch()方法定义在原型对象Promise.prototype上

// Promise对象状态为失败，调用catch()方法指定的回调函数

getJSON('posts.json').then(function (posts) {

    // ...

}).catch(function (error) {

    console.log('发生错误!', error);

})

1. Promise.all()

// Promise.all()

// 将多个Promise实例存入数组作为参数，包装成一个新的Promise实例

const p = Promise.all([p1, p2, p3])

// p的状态由 p1, p2, p3 决定

/\* 当 p1, p2, p3 的状态都变成fulfilled，p的状态才为fulfilled，此时将

    p1, p2, p3的返回值组成一个数组，传递给p的回调函数

\*/

/\* 当 p1, p2, p3 中有一个被rejected，p的状态就变成rejected，此时第一个

    被rejected的实例的返回值，会传递给p的回调函数

\*/

const promises = [2, 3, 5].map(function (id) {

    return getJSON('/post/' + id + '.json')

})

Promise.all(promises).then(function (posts) {

    // fulfilled 执行的回调函数

    // ...

}).catch(function (reason) {

    // rejected 执行的回调函数

    // ...

})

1. Promise.resolve()

// Promise.resolve()  将一个对象，转为Promise对象，状态为 resolved

Promise.resolve('foo')

// 等价于

new Promise(resolve => resolve('foo'))

1. Promise.reject()

// Promise.reject()  返回一个新的Promise实例，状态为 rejected

const p = Promise.reject('出错了')

// 等同于

const p = new Promise((resolve, reject) => reject('出错了'))

p.then(null, function (s) {

    console.log(s);  // 出错了

})

1. Iterator遍历器
2. Iterator的遍历过程

* 创建一个指针对象，指向当前数据结构的起始位置
* 第一次调用指针对象的next方法，将指针指向数据结构的第一个成员
* 第二次调用指针对象的next方法，指针就指向数据结构的第二个成员
* 不断调用指针对象的next方法，直到它指向数据结构的结束位置

// 每次调用next方法，都会返回数据结构的当前成员的信息

// 每次返回一个包含value和done两个属性的对象

// value属性为当前成员的值，done属性为布尔值，表示遍历是否结束

var it = makeIterator(['a', 'b'])

it.next()  // { value: 'a', done: false }

it.next()  // { value: 'b', done: false }

it.next()  // { value: undefined, done: true }

function makeIterator(array) {

    let nextIndex = 0

    return {

        next: function () {

            return nextIndex < array.length ?

                { value: array[nextIndex++], done: false } :

                { value: undefined, done: true }

        }

    }

}

1. 默认Iterator接口

* Iterator接口的目的，就是为所有数据结构提供统一的访问机制，即for…of循环
* 一个数据结构只要部署了Symbol.Iterator属性，就是具有Iterator接口
* 一种数据结构只要有Iterator接口，这种数据结构就是“可遍历的”
* 默认的Iterator接口部署在数据结构的Symbol.Iterator属性
* 原生具备Iterator接口的数据结构：Array、Map、Set、String

// Array数组的Symbol.Iterator属性

let arr = ['a', 'b', 'c']

let iter = arr[Symbol.iterator]()

iter.next()  // { value: 1, done: false}

// String字符串是一个类似数组的对象，也具有Iterator接口

let str  = 'hi'

typeof str[Symbol.iterator]  // 'function'

let iter = str[Symbol.iterator]()

iter.next()  // {value: 'h', done: false}

1. 调用Iterator接口的场景

* 解构赋值 默认调用Iterator接口

let set = new Set().add('a').add('b').add('c')

let [x, y] = set  // x = 'a'; y = 'b'

* 扩展运算符（…） 扩展运算符内部调用Iterator接口

var str = 'hello'

[...str]  // ['h', 'e', 'l', 'l', 'o']

1. for…of循环

// for...in循环，遍历获取键名，主要用来遍历对象，不适用于遍历数组

// for...of循环，遍历获取键值，可以配合break、continue、return使用

var arr = ['a', 'b', 'c']

for (let a in arr) {

    console.log(a);  // 0 1 2

}

for (let a of arr) {

    console.log(a); // a b c

}

1. Set结构、Map结构

// Set结构和Map结构遍历的顺序，是成员添加进数据结构的顺序

// Set结构  返回一个值

let set = new Set(['green', 'blue', 'red'])

for(let e of set) {

    console.log(e); // green  blue  red

}

// Map结构  返回一个数组，数组成员为键名和键值 [键名name, 键值value]

let map = new Map()

map.set('green', 6);

map.set('blue', 7);

map.set('red', 8)

for(let [name, value] in map) {

    console.log(name + ": " + value);

}

// green: 6

// blue: 7

// red: 8

1. Async函数
2. async函数的使用

// async函数返回一个Promise对象，可以使用then方法添加回调函数

async function getPriceName(name) {

    const symbol = await getSymbol(name)

    const price = await getPriceName(symbol)

    return price

}

getPriceName('goog').then(function (reslut) {

    console.log(reslut);

})

// 调用async函数，会立即返回一个Promise对象

async function timeout(ms) {

    return new Promise((resolve, reject)=>{

        setTimeout(resolve, ms)

    })

}

async function asyncPrint(value, ms) {

    await timeout(ms)

    console.log(value);

}

asyncPrint('hello world', 2000)

1. async函数的使用形式

// 函数声明

async function foo(){}

// 函数表达式

const foo = async function(){}

// 对象的方法

let obj = { async foo() {}}

obj.foo().then()

// 箭头函数

const foo = async ()=>{}

1. async函数语法

* 返回Promise对象

// async 函数内部return语句返回的值（将promise对象变成resolve状态），会成为then方法回调函数的参数

// async 函数内部抛出的错误（将promise对象变成reject状态），抛出的错误对象会被catch方法回调函数接收

async function foo() {

    return 'hello world'

    // throw new Error('出错了')

}

foo().then(

    v => console.log('resolve', v),   // 'hello world'

    // e => console.log('reject', e)  // reject Error: 出错了

)

* Promise对象的状态变化

// async 函数内部的异步操作全部执行完后，才会执行then方法指定的回调函数

// 若async函数内部遇到return语句或者抛出错误时，则跳出函数体

async function getTitle(url) {

    let response = await fetch(url);

    let html = await response.text();

    return html.match(/<title>([\s\S]+)<\/title>/i)[1];

}

getTitle('https://tc39.github.io/ecma262/').then(console.log)

* await命令

// await命令后面是一个对象的实例，await会将其视为Promise对象处理

function sleep(interval) {

    return new Promise((resolve) => {

        setTimeout(resolve, interval)

    })

}

async function oneFive() {

    for (let i = 1; i <= 5; i++) {

        console.log(i);

        await sleep(1000)

    }

}

oneFive()

// 任何一个await语句后面的Promise对象变为reject状态，整个async函数都会中断执行

async function fn() {

    await Promise.reject('出错了')

    await Promise.resolve('hello world')  // 这一行不会执行

}

fn()

    .then(v => console.log(v))

    .catch(e => console.log(e))   //  出错了

有两种方法可以使await函数不影响后面的异步操作

// 第一种方法：将该await函数放入try...catch中

async function fn() {

    try {

        await Promise.reject('出错了')

    } catch (e) { }

    return await Promise.resolve('hello world')

}

fn().then(v => console.log(v))  // 'hello world'

// 第二种方法：将await后面的Promise对象再跟一个catch方法，处理可能出现的错误

async function fn() {

    await Promise.reject('出错了')

        .catch(e => console.log(e))

    return Promise.resolve('hello world')

}

fn().then(v => console.log(v))

// 出错了

// hello world

// 如果await操作成功，使用break语句退出循环

// 如果await操作失败，会被catch语句捕捉，进入下一轮循环

const superagent = require('superagent')

const NUM = 3

async function test() {

    for (let i = 0; i < NUM; i++) {

        try {

            await superagent.get('http://google.com')

            break

        } catch (e) {}

    }

    console.log(i);

}

test()

1. 使用注意点

* await命令后面的Promise对象,运行结果可能是rejected，所以最好把await命令放在try…catch代码块中
* 多个await命令后面的异步操作，如果不存在继发关系，最好让它们同时触发

let foo = await getFoo()

let bar = await getBar()

// 若要同时触发，可使用 Promise.all()

let [foo, bar] = await Promise.all([getFoo(), getBar()])

* await命令只能用在async函数之中，不能用在普通函数，否则报错

\*\*\*习题：在一个DOM元素上面，部署了一系列的动画，前一个动画结束，后一个动画才能开始执行，当其中一个动画出错，就不再往下执行，并返回上一个动画的返回值\*\*\*

async function animationAsync(elem, animations) {

    // 变量rest用来保存上一个动画的返回值

    let rest = null

    try {

        // 循环遍历所有的动画

        for (let anim of animations) {

            // 当一个动画执行成功后，将返回值赋给rest，继续执行下一个动画

            // 当一个动画执行失败后，跳出循环，执行catch方法

            rest = await anim(elem)

        }

    } catch (err) { }

    return rest

}

1. Class类 的基本语法

1、定义一个类

// 定义一个类(Point)

class Point {

    // 构造方法：constructor()

    constructor(x, y) {

        // this  实例对象

        this.x = x

        this.y = y

    }

    toString() {

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

    }

}

const point = new Point(1, 2)   // Point  { x: 1, y: 2 }

调用类的实例上面的方法，就是调用原型上的方法

class B {}

const b = new B()

b.constructor === B.prototype.constructor  // true

使用Object.assign()方法，向prototype对象上面添加新方法

// Object.assign()  向对象上添加属性或方法

class Point {

    constructor() { }

}

Object.assign(Point.prototype, {

    toString() {},

    toValue() {}

})

类的内部定义的所有的方法，都不能通过遍历获取

class Point {

    constructor() {

        // ...

    }

    fn1() {}

    fn2() {}

    fn3() {}

}

const point = new Point()

Object.keys(point)  // []

2、constructor()方法

* 一个类中必须有constructor()方法，如果没有显式定义，会默认添加一个空的

class Point {}

// 等同于

class Point {

    constructor() {}

}

3、类必须通过new来调用，否则报错

class Foo{

    constructor(){}

}

// 报错

let foo = Foo()

// 正确

let foo = new Foo()

4、实例属性的新写法

class Foo {

    // 原来的写法，将 \_count属性 定义在constructor()方法的this上

    constructor(){

        this.\_count = 0

        this.\_num = 0

    }

    get value(){

        return this.\_count

    }

}

class Foo{

    // 直接将 \_count属性 定义在类的最顶层，能清晰看出类中有哪些实例属性

    \_count = 0

    \_num = 0

    get value(){

        return this.\_count

    }

}

5、取值函数（getter）和存值函数（setter）

* 使用get和set关键字，对某个属性设置取值函数和存值函数，拦截属性存取行为

class MyClass {

    constructor() {}

    get prop() {

        return 'getter'

    }

    set prop(value) {

        console.log('setter: ' + value);

    }

}

let inst = new MyClass()

inst.prop = 123  // setter: 123

inst.prop        //  getter

6、属性表达式

* 类的属性名，可以采用表达式

// 将属性名定义在类外部，使用表达式获取属性名

let methodName = 'getArea'

class Square {

    constructor(length) {

        //...

    }

    [methodName]() {

        // ...

    }

}

7、Class表达式

* 类 可以使用表达式的形式定义

// 定义一个类，类的名字为Me，Me只在Class内部可用

// 在Class外部，这个类只能用MyClass引用

const MyClass = class Me {

    getClassName() {

        return Me.name

    }

}

let inst = new MyClass()

inst.getClassName()  // Me

Me.name  // 报错

* 使用Class表达式，可以写出立即执行的Class

// 使用表达式，写出一个立即执行的函数

let person = new class {

    constructor(name) {

        this.name = name

    }

    sayName() {

        console.log(this.name);

    }

}('张三')

person.sayName()  // '张三'

8、静态方法（static）

* 在类中定义的方法，都会被实例继承；如果在方法前面加上static关键字，该方法就不会被继承，而是直接通过类来调用

class Foo {

    static classMethod() {

        return 'hello'

    }

}

Foo.classMethod()  // 'hello'

let foo = new Foo()

foo.classMethod()  // 报错

* 静态方法包含this关键字，这个this指的是类，而不是实例

// 静态方法中的this，指的是类（this指向的地方，必须为静态方法中）

class Foo {

    static bar() {

        this.baz()

    }

    static baz() {

        console.log('hello');

    }

    baz() {

        console.log('world');

    }

}

Foo.bar()  // 等同于  Foo.baz() 'hello'

9、静态属性（static）

* 在实例属性前面，加static关键字

class Foo {

    static prop = 1  // 使用static 将属性变为静态属性，只在类内部使用

}

10、私有属性、私有方法

* 私有属性——在属性名前使用 # 表示

// #count 为私有属性，只能在类的内部使用(this.#count)

// 不管在类的内部还是外部，读取一个不存在的私有属性，都会报错

class IncreasingCounter {

    #count = 0

    get value() {

        console.log('Getter the current value!');

        // 读取一个不存在的私有属性 #MyCount，报错

        return this.#MyCount  // 报错

    }

    increment() {

        this.#count++

    }

}

const counter = new IncreasingCounter()

counter.#count  // 报错

counter.#count = 10  // 报错

counter.#MyCount  // 私有属性 #MyCount 不存在，报错

* 私有方法——在方法名前面使用 # 表示

class Foo {

    #a;

    #b;

    constructor(a, b) {

        this.#a = a

        this.#b = b

    }

    #sum() {

        return this.#a + this.#b

    }

    printSum() {

        console.log(this.#sum());

    }

}

1. Class的继承

* 子类使用extends关键字实现继承父类

class Point{

}

// 使用 extends 实现继承

class ColorPoint extends Point{

}

* 子类中使用super关键字，调用父类的方法

// 子类必须在constructor()中调用super()，否则子类得不到自己的this对象

class Point { /\* ... \*/ }

class ColorPoint extends Point {

    constructor(x, y, color) {

        super(x, y)  // 调用父类的constructor(x, y)

        this.color = color

    }

    toString() {

        return super.toString()  // 调用父类的toString()

    }

}

* 在子类的构造函数中，只有调用super()之后，才可以使用this关键字。子类实例的创建，必须先完成父类的继承，通过super()才能让子类实例继承父类

class Point {

    constructor(x, y) {

        this.x = x

        this.y = y

    }

}

class ColorPoint extends Point {

    constructor(x, y, color) {

        this.color = color  // 报错

        super(x, y)

        this.color = color // 正确

    }

}

* 私有属性和私有方法，不可以被子类继承；如果父类定义了私有属性的读写方法，子类可以通过这些方法，读写私有属性

class Foo {

    #p = 1

    getP() {

        return this.#p

    }

}

class Bar extends Foo {

    constructor() {

        super()

        console.log(this.getP());  // 1

    }

}

let bar = new Bar()

* 静态属性和静态方法，可以被子类继承；静态属性是通过浅拷贝继承的，如果拷贝的静态属性的值是一个对象，那么子类的静态属性也会指向这个对象

class A {

    static foo = { n: 100 }

    static bar = 'a'

}

class B extends A {

    constructor(){

        super()

        B.foo.n = 99

        B.bar = 'b'

    }

}

const b = new B()

B.foo.n // 99

B.bar   // 'b'

A.foo.n // 99

A.bar   // 'a'

1. Module的语法

1、ES6模块的使用

// 从fs模块中，加载3个方法

import { stat, exists, readFile } from 'fs'

2、严格模式

* 变量必须声明后再使用
* 函数的参数不能有同名属性，否则报错
* 不能使用with语句
* 不能对只读属性赋值，否则报错
* 不能删除 不可删除的属性，否则报错
* 不能使用前缀0表示八进制数，否则报错
* 不能删除变量delete prop，会报错，只能删除属性 delete global[prop]
* eval不会在它的外层作用域引入变量
* eval和arguments不能被重新赋值
* arguments不会自动反映函数参数的变化
* 不能使用arguments.callee
* 不能使用arguments.caller
* 禁止this指向全局对象
* 不能使用fn.caller和fn.arguments获取函数调用的堆栈
* 增加了保留字（比如protected、static和interface）

3、export命令

* export命令用于规定模块的对外接口
* import命令用于输入其他模块提供的功能

// profill.js  JS文件

var firstName = 'MIchael'

var lastName = 'Jackson'

var year = 1958

export { firstName, lastName, year };

// main.js 其他JS文件

import { firstName, lastName, year } from './profill.js'

function setName(element) {

    element.textContent = firstName + ' ' + lastName

}

* export命令个输出函数或类

export function multiply(x, y) {

    return x \* y

}

4、import命令

* import命令输入的变量都是只读的，不允许在加载模块的脚本中修改

import { a } from './xxx.js'

a = {} // 报错

5、模块的整体加载

* 用 \* 指定一个对象，所有的输出值都加载在这个对象上面

// file.js  JS文件

export function foo(num1) {

    return num1++

}

export function bar(num2) {

    return num2++

}

// main.js  JS文件

import \* as circle from './file.js'

console.log(circle.foo(2));

console.log(circle.bar(4));

6、export default命令

* 为模块指定默认输出

// export.js

// 默认输出一个函数

export default function () {

    console.log('foo');

}

// main.js

// 使用import命令，用任意名称指向export.js输出的方法

import funName from './export.js'

funName()  // 'foo'

1. 编程风格

1、块级作用域

* let取代var

let i = 0;

* const定义全局常量

const [a, b, c] = [1, 2, 3]

2、字符串

* 静态字符串使用 单引号

const a = 'foobar'

* 动态字符串使用 反引号，和 ${} 一起使用

const a = 'foobar'

const b = `print: ${a}`

3、解构赋值

* 使用数组成员对变量赋值时，优先使用解构赋值

const arr = [1, 2, 3]

const [first, second] = arr

* 函数的参数如果是对象的成员，优先使用解构赋值

// 将对象成员解构，作为函数参数

function getFullName({ firstName, lastName }) {}

* 如果函数返回多个值，优先使用 对象的解构赋值

// 使用对象的解构赋值，获取函数的多个返回值

const { left, right } = processInput(input)

4、对象

* 使用Object.assign()方法，向对象中添加新的属性

const a = {}

Object.assign(a, { x: 3 })

* 对象的属性和方法，采用简洁表达式

let ref = 'some value'

const atom = {

    ref,

    value: 1,

    addValue(value) {

        return this.value + value

    }

}

5、数组

* 使用扩展运算符（…）拷贝数组

const newArr = [...arr]

* 使用Array.from方法，将类似数组的对象转为数组

const foo = document.querySelectorAll('.foo')

const nodes = Array.from(foo)

6、函数

* 立即执行函数写成箭头函数的形式

(() => {

    console.log('hello');

})()

* 用箭头函数代替使用匿名函数作为参数的场合

[1, 2, 3].map(x => x \* x)

* 用箭头函数取代Function.prototype.bind绑定this

const boundMethod = (...params) => method.apply(this, params)

* 所有的配置项都应该集中在一个对象，放在最后一个参数

function divide(a, b, { option = false } = {}) {}

* 使用默认值语法 设置函数参数的默认值

function foo(opts = {}) {}

7、Map结构

* 如果只是需要key:value的数据结构，使用Map结构

let arr = [['a', 1], ['b', 2], ['c', 3]]

let map = new Map(arr)

map.keys()  // 获取全部的键名

map.values()  // 获取全部的键值

map.entries() // 获取所有的键值对

# 页面渲染流程

1. 解析HTML

浏览器在开始解析之前，会启动一个预解析线程，当解析过程中遇到CSS，会同步解析CSS，不影响HTML的解析过程；当解析过程中遇到JS，会暂停解析HTML，优先解析JS，JS解析完成之后，再继续解析HTML（JS代码在执行过程中，可能会修改DOM树）

生成DOM树、CSSOM树

1. 样式计算

主线程会遍历所有DOM树的节点，为每一个节点计算出最终的样式

1. 布局

主线程会遍历DOM树的每一个节点，计算出每个节点的几何信息，最后生成布局树

若节点没有几何信息（display：none），就不会生成到布局树中

1. 分层

主线程会对整个布局树进行一个分层，当某一层发生变化时，其他层不受影响。

1. 绘制

主线程为每一层生成如何绘制的指令，形成一个指令集，完成绘制后，主线程将每个图层的绘制信息提交给合成线程，主线程结束任务。

1. 分块

将每一层都分成多个块，分块任务会交给多个线程同时进行

1. 光栅化

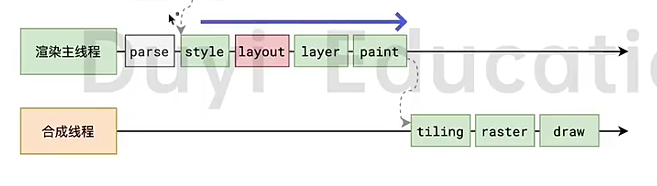
将每一个块都变成一个位图，优先处理靠近视口的块，这个过程在GPU进程中完成

1. 画

合成线程计算出每个位图在屏幕上的位置，交给GPU进程完成最终呈现

# 重排（reflow）、重绘（repaint）

1. 什么是重排和重绘



1）、重排：重新计算布局树。当布局树更新且节点的尺寸发生了变化，浏览器会使布局树中受到影响的部分失效，并重新计算布局树

2）、重绘：重新根据分层信息计算绘制指令。在一个元素的样式被改变所触发的浏览器行为，浏览器会根据元素的新属性重新绘制，使元素呈现新的外观

1. 引起重排的属性和方法

添加或删除可见的DOM元素

元素位置改变，或者使用动画

元素的尺寸改变（外边距、内边距、边框厚度、宽高等几何属性）

内容改变（例如：文本改变或图片被另一个不同尺寸的图片替代、在input框输入内容）

激活CSS伪类（如 :hover）

1. 引起重绘的属性和方法

外观属性包括界面、文字等可用状态向量描述的属性

 界面：appearance、outline、background、mask、box-shadow、box-reflect、filter、opacity、clip、border-radius、background-size、visibility

文字：text、font、word

1. 减少重排和重绘

1）、避免一条一条的修改DOM的样式，可以直接修改DOM的className

2）、避免把DOM结点的属性值放在一个循环里当成循环里的变量

3）、避免在大量元素上使用:hover

4）、避免在布局信息改变时查询布局信息

5）、使用transform属性。

Transform不会影响布局，也不会影响绘制指令，只会改变draw，而draw是在合成线程上进行，对渲染主线程没有影响。

# Flex弹性布局

容器的属性

/\* 设置弹性布局 \*/

display: flex;

/\* 设置方向 \*/

flex-direction: row;  /\* 行 \*/

flex-direction: row-reverse; /\* 颠倒行 \*/

flex-direction: column;  /\* 列 \*/

flex-direction: column-reverse;  /\* 颠倒列 \*/

/\* 设置是否换行 \*/

flex-wrap: nowrap;  /\* 不换行 \*/

flex-wrap: wrap;  /\* 换行 第一行在最上面 \*/

flex-wrap: wrap-reverse;  /\* 换行  第一行在最下面 \*/

/\* 设置X轴的对齐方式 \*/

justify-content: center;  /\* X轴居中 \*/

justify-content: flex-start;  /\* 左对齐 \*/

justify-content: flex-end;  /\* 右对齐 \*/

justify-content: space-around;  /\* 每个项目左右两边的间隔相等 \*/

justify-content: space-between; /\* 第一个项目和最后一个项目 到边框的距离为0 \*/

/\* 设置Y轴的对齐方式 \*/

align-items: center;  /\* Y轴居中 \*/

align-items: flex-start;  /\* 上对齐 \*/

align-items: flex-end;  /\* 下对齐 \*/

项目的属性

/\* 布局容器中，每个项目的属性 \*/

/\* 项目的排列顺序 \*/

order: 0;  /\* 数值从小到大，排列从左到右 \*/

/\* 项目的放大比例 \*/

flex-grow: 0; /\* 数值越大，放大比例越大  默认为0 \*/

/\* 项目的缩小比例 \*/

flex-shrink: 1;  /\* 数值越大，缩小比例越大  默认为1 \*/

**display：inline-block，block，inline**

1、块级元素

每个块级元素会占领页面的一行，其后多个block元素自动换行，可以设置width、height、padding、margin等属性

常见的块级元素：div、h1-h6、img、li、p等

2、行级元素

与其他元素在同一行上，高度，行高以及底边距不可改变，且高度就是内容文字或者图片的高度，不可以改变。

常见的行级元素：em、strong、br、input等

3、display：block

将元素设置为块级元素，可以操控元素的宽高以及内外边距，每个块级元素都是从新的一行开始

4、display：inline

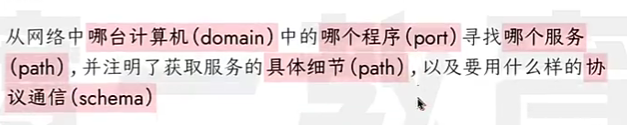
将元素设置为行级元素，高度、行高以及底边距不可改变，高度就是内容文字或者图片的高度

5、display：inline-block

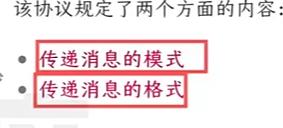
将元素设置为行内块级元素，既有行级元素的特性，也有块级元素的特性，因此在同一行，能设置宽高、margin、padding

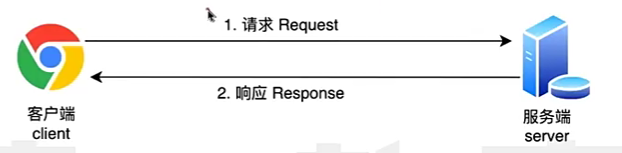
**网络**





HTTP协议





# Rem适配

rem是指相对于根元素的字体大小的单位。可以简单理解为一个相对大小的单位。

px：html中css的一个绝对单位

em：相对于父元素的字体的大小的单位

rem：相对于根元素的字体的大小的单位

rem适配：

1. 在得到设计稿之后，一般情况下，把设计稿的尺寸定为750px，同时我们也需要把浏览器的模拟器也调整为750px
2. 严格按照设计稿，以rem为单位，把设计稿还原出来。首先需要确定的是html的font-size，通常定为100px，叫作rem的标准基
3. 把写好的页面，迁移到其它屏幕上，完成适配

在750px的屏幕上，html的font-size地大小为100px

在375px的屏幕上，html的font-size地大小为50px

# 媒体查询

媒体查询就是样式表的if语句。通过各种条件去判断到底执行哪一段CSS

## 媒体类型种类：

all：全部媒体类型

@media all max-width:600px {

    ...

}

screen：通常指PC、手机、pad

@media screen max-width:600px {

    ...

}

## 可验证特征条件

width、max-width、min-width、color等

## 媒体查询格式

@media (条件1) [逻辑符] (条件2) ... {

    ...

}

## 媒体查询逻辑

与关系（and）：将多个媒体类型与特征联系起来，条件全为真，查询成立

@media (min-width:300px) and (max-width:600px) {

    ...

}

或关系（，）：有一个条件满足，样式就生效

@media (min-width:300px) , (color:red) {

    ...

}

非（not）：取反操作，意思是除了

@media not (min-width:300px) {

    ...

}

# Diff算法

Diff算法可以看做是一种对比算法。对比的对象是新旧虚拟DOM。它可以找到新旧虚拟DOM之间的差异，再根据对比后的结果更新真实DOM。

流程：

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。

Vue中key的作用：

对比两个虚拟节点时，判断其是否为相同节点。

加了key以后，我们可以更为明确的判断两个节点是否为同一个虚拟节点，是的话判断子节点是否有变更（有变更更新真实Dom），不是的话继续比

key可以有效减少真实Dom的变动，从而减少页面重绘和回流的频率，进而提高页面更新的效率。

# 虚拟列表实现

实现：

虚拟列表的实现，实际上就是在首次加载的时候，只加载可视区域内需要的列表项，当滚动发生时，动态通过计算获得可视区域内的列表项，并将非可视区域内存在的列表项删除

过程：

1. 外层容器高度固定（550）,并且设置（overflowY: ‘auto’）
2. 计算内层容器放下全部数据的高度（total \* 55）
3. 计算在这个固定容器高度下可视区可以装多少条数据（limit = 550 / 55）
4. 滚动的时候，通过参数scrollTop，可以计算可视区展示的第一条数据是第几条（startIndex = scrollTop / 55 ）和最后一条数据是第几条（endIndex = limit + startIndex）
5. 优化：加一个缓冲条数bufferSize，防止鼠标滚动过快的时候出现空白区域

# 防抖和节流

## 防抖

防止抖动，避免事件的重复触发。等用户高频事件完成之后，再进行事件操作（search输入框输入）

实现：事件触发后开启一个定时器，如果事件在这个定时器限定的时间内被触发，则清除定时器，然后再写一个定时器，定时时间到则触发。

function debounce(fn, delay) {

    let timer = null

    return function () {

        clearTimeout(timer)

        timer = setTimeout(() => {

            fn.apply(this, arguments)

        }, delay);

    }

}

## 节流

减少流量。将频繁触发的事件减少，并每隔一段时间执行，控制事件触发的频率。（某个操作希望上一次的完成后再进行下一次，或者希望隔一段时间触发一次）

实现：

设计一种类似控制阀门一样定期开放的函数，事件触发时让函数执行一次，然后关闭这个阀门，过了一段时间后再将这个阀门打开，再次触发事件

function throttle(fn, delay) {

    let valid = true

    return function() {

        if(valid) {

            setTimeout(() => {

                fn.apply(this, arguments)

                valid = true

            }, delay);

            valid = false

        }

    }

}

## 应用场景

防抖：

search搜索联想：用户在不断输入值时，用防抖来节约请求资源

节流：

鼠标不断点击触发，mousedown（单位时间内只触发一次）

监听滚动事件，滚动到底部自动加载更多

## 异同点

相同点：

都是为了阻止操作高频触发，避免浪费性能

不同点：

防抖是触发高频事件后n秒内函数只会执行一次，如果n秒内高频事件再次被触发，则重新计算时间。适用于可以多次触发但触发只生效最后一次的场景

节流是高频事件触发，但在n秒内只会执行一次，如果n秒内触发多次函数，只有一次生效，节流会稀释函数的执行频率

# 内存泄漏

JS内存泄漏：

1. 意外的全局变量：未声明的变量会被自动创建为全局对象的属性，使用var可能导致意外的全局变量
2. 未解除的事件监听器(addEventListener)：如果一个DOM元素被移除，但相关的事件监听器仍然保持在内存中，就会发生内存泄漏
3. 闭包：如果一个函数内部定义了另一个函数，并且这个内部函数引用了外部函数的变量，当外部函数执行完毕后，这些变量不会被垃圾回收，内部函数依然可以访问
4. 未移除的定时器或回调函数：设置了定时器或者异步回调函数，但是没有在不需要时清除它们

解决内存泄漏的方法：

1. 使用let和const声明变量，避免意外的全局变量
2. 在移除DOM元素前，先移除相关的事件监听器(removeEventListener)
3. 避免闭包中的错误引用，或者手动将闭包中的变量设为null
4. 清理定时器和异步回调，使用clearTimeout()和clearInterval()移除定时器；如果为异步操作，确保在不需要时回调函数不会执行

# 父子组件生命周期钩子函数执行顺序

1. 加载渲染过程

父beforeCreate -> 父created -> 父beforeMount -> 子beforeCreate -> 子created -> 子beforeMount -> 子mounted -> 父mounted

1. 子组件跟新过程

父beforeUpdate -> 子beforeUpdate -> 子updated -> 父updated

1. 父组件更新过程

父beforeUpdate -> 父updated

1. 销毁过程

父beforeDestroy -> 子beforeDestroy -> 子destroyed -> 父destroyed

# 检测数据类型

在JS中，数据类型分为基本数据类型和引用数据类型

基本数据类型：String、Number、Boolean、Null、Undefined、Symbol、BigInt

引用数据类型：Object、Array、Function、Date

检测数据类型：

1. typeof：用于检测基本数据类型
2. instanceof：用于引用数据类型，判断某个对象是由哪个类（构造函数）产生的
3. constructor：几乎可以用来判断基本数据类型和引用数据类型，但是如果创建一个对象来改变它的原型，就不能用来判断数据类型
4. Object.prototype.toString.call()：可以用来判断

# slice、splice、split

## slice：截取功能

新数组 = 原数组.slice(开始位置的索引，结束位置的索引) （ 左闭右开 -> [a, b ) ）

1. 截取数组为主，也可以截取字符串
2. 返回新的数组，包含截取的元素
3. 不改变原数组

## splice：数组增删查改

新数组 = 原数组.splice(起始索引index，需要删除的个数)

1. 只能对数组进行增删查改，对字符串无效
2. 返回新的数组，内容是被删除的元素
3. 会改变原数组

## split：字符串=>数组

新数组 = 原数组.split(分隔符)

1. 字符串的方法，不是数组的方法
2. 返回一个字符串数组

# 类数组转化成数组

1. 通过call调用数组的slice方法

Array.prototype.slice.call(arrayLike)

1. 通过call调用数组的splice方法

Array.prototype.splice.call(arraylike, 0)

1. 通过apply调用数组的concat方法

Array.prototype.concat.apply([], arrayLike)

1. 通过Array.from()

Array.from(arrayLike)

# v-if和v-show

v-if：通过控制DOM节点的存在与否，来控制元素的显隐，将DOM节点删除

v-show：直接控制DOM节点的显隐（display:none）

# 数组去重

let arr = [1, 1, "1", "1", true, true, "true", {}, {}, "{}", null, null, undefined, undefined]

1. Set数据结构

let uniqueOne = Array.from(new Set(arr))

1. forEach循环遍历

let uniqueTwo = arr => {

let map = new Map(); //或者用空对象 let obj = {} 利用对象属性不能重复得特性

let brr = []

arr.forEach(item => {

if (!map.has(item)) { //如果是对象得话就判断 !obj[item]

map.set(item, true) //如果是对象得话就obj[item] =true 其他一样

brr.push(item)

}

})

return brr

}

# 数组中出现最多的元素

function fn(arr) {

    //如果数组中没有值，直接返回

    if (!arr.length) return;

    //如果只有一个值，返回1，表示出现了1次

    if (arr.length === 1) return 1;

    var result = {};

    //对数组进行遍历

    for (var i = 0; i < arr.length; i++) {

        if (!result[arr[i]]) {

            result[arr[i]] = 1;

        } else {

            result[arr[i]]++;

        }

    }

    //遍历result对象

    var keys = Object.keys(result);

    var maxNum = 0,

        maxElement;

    for (var i = 0; i < keys.length; i++) {

        if (result[keys[i]] > maxNum) {

            maxNum = result[keys[i]];

            maxElement = keys[i];

        }

    }

    return (

        "在数组中出现最多的元素是" + maxElement + ",共出现了" + maxNum + "次"

    );

}

var array = [1, 2, 3, 3, 3, 6, 6, 6, 6, 6, 7, 8, 9];

# Proxy

## 什么是Proxy

Proxy是ES6中提供的一个构造函数，字面意思就是“代理”，但是更像是一种“拦截器”。在访问、赋值、属性查找等基本操作时会先到我们定义好的拦截方法中，根据访问的信息去执行我们想要的操作。

## Proxy使用

// new Proxy(target, handler) // target -> 目标对象   handler -> 用户自定义的行为对象

let handler = {

    get(target, propkey, receiver) {

        console.log('getting values');

        return target[propkey] || 'value is not defined'

    },

    set(target, propkey, value, receiver) {

        console.log('setting values');

        return target[propkey] = value

    }

}

var proxy = new Proxy({}, handler)

console.log(proxy.name);

// 输出

// getting values

// value is not defined

proxy.name = 'marker'

// 输出

// setting values

console.log(proxy.name);

// getting values

// marker

定义了一个拥有 **get** 和 **set** 的代理，当我们在访问 **proxy** 对象中的 **name** 时，会进入 **handler**中的 **get** 方法并执行。同样赋值给 **proxy** 赋值时，则会进入 **hander** 中的 **set** 方法中

# SPA单页面

## 什么是SPA

SPA就是单页应用，通过动态重写当前页面来与用户交互，避免了页面之间切换打断用户体验。JS框架vue就是SPA。

## 单页应用的优缺点

优点：

1. 用户体验好。内容的改变不需要重新加载整个页面
2. 前后端分离

缺点：

1. 不利于搜索引擎的抓取
2. 首屏加载过慢

# 首屏性能优化

首屏加载：指的是浏览器从响应用户输入地址，到首屏内容渲染完成的过程。此时整个网页不一定要全部渲染完成，但要展示当前视窗需要的内容。

问题出现：

1. 页面请求未回来，导致loading加载缓慢，滑动造成页面不流畅
2. 页面请求过多，导致加载变慢

解决方法：

1. 使用vue-router路由懒加载
2. 采用按需加载。当模块没有出现在屏幕中，先用空白div占位。
3. UI库采用按需加载

# 白屏问题

加loading

# 前端登录流程

1. 调用后端登录接口请求并发送用户名和密码，后端接收请求返回token和用户信息。
2. 前端将token存储到vuex，然后从vuex中将token值存入浏览器的Cookies中
3. 将用户信息存储到vuex中，再存入浏览器的localStorage中，然后跳转到下一个页面
4. 根据后端要求跳转页面需要判断Cookies中是否有token，没有就跳转到登录页，有就跳转到相应的页面

注意：在每次发送post/get请求的时候，都应该加上token。在项目utils/service.js中添加全局拦截器，将token的值放入请求头中，后端判断请求头中有无token，有的话就拿到并验证是否过期，若过期会返回无效token，并跳回登录页面重新登录，并清除本地用户信息。

# HTTP和HTTPS

HTTP：

超文本传输协议，是一个基于请求与响应，无状态的，应用层的协议，基于TCP/IP协议传输数据。

特点：

1. 无状态：协议对客户端没有状态存储，对事物处理没有‘记忆’能力。访问一个网站需要反复进行登录操作。
2. 无连接：每次请求需要通过TCP三次握手四次挥手，和服务器重新建立连接
3. 基于请求和响应：由客户端发起请求，服务端响应
4. 简单快速、灵活

HTTPS是在HTTP上建立SSL加密层，并对传输数据进行加密。通过计算机网络进行安全通信的传输协议。

特点：

1. 内容加密：采用混合加密技术，无法直接查看明文内容
2. 验证身份：通过证书认证客户端访问的是自己的服务器
3. 保护数据完整性：防止传输的内容被中间人冒充或篡改

# 强缓存、协商缓存

## 强缓存

浏览器不会向服务器发送请求，直接从本地缓存中读取文件并返回状态码200

Header参数

1. Expires属性：指定资源的过期时间（服务器时间）。若服务器时间与客户端时间不一致，则会影响缓存命中的结果。
2. Cache-Control属性：精确控制资源的缓存。

public（共有的）：资源可以被任何对象缓存。

private（私有的）：资源只能被用户浏览器缓存，不允许任何代理服务器缓存

## 协商缓存

向服务器发送请求，服务器会根据这个请求的 request header 的一些参数来判断是否命中协商缓存。如果命中，则返回304状态码并带上新的 response header 通知浏览器从缓存中读取资源

命中方式：

1. max-age = xxx 过期
2. Cache-Control 值为 no-store

# HTTP状态码

2XX：成功，操作被成功接收并处理

200：请求成功

201：已创建。成功请求并创建了新的资源

202：已接受。已经接受请求，但未处理完成

203：非授权信息。请求成功

204：无内容。服务器成功处理，但未返回内容

205：重置内容。服务器处理成功，浏览器重置文档视图

206：部分内容。服务器成功处理了部分内容

3XX：重定向，需要进一步的操作来完成请求

301：资源（网页等）被永久转移到其他URL

302：临时移动

303：查看其它地址

304：未修改。所请求的资源未修改

305：使用代理。所请求的资源必须通过代理访问

4XX：客户端错误，请求包含语法错误或无法完成请求

400：客户端请求的语法错误，服务器无法理解

401：请求要求用户的身份认证

402：保留，将来使用

403：服务器拒绝执行客户端请求

404：请求资源（网页等）不存在

405：客户端请求中地方法被禁止

5XX：服务器错误。服务器在处理请求的过程中发生了错误

500：服务器内部错误，无法完成请求

501：服务器不支持请求的功能，无法完成请求

502：服务器执行请求时，从远程服务器接收到了一个无效的响应

503：由于某种原因，服务器暂时无法处理客户端的请求

504：服务器未及时从远端服务器获取请求

# Git常见操作

* **git add [文件名]** 将指定文件添加到缓冲区
* **git commit -m ‘注释’**  将缓冲区内容提交到版本仓库并加上注释
* **git pull** 从远程更新源码仓库到本地
* **git push** 将本地更新同步到远程仓库
* **git** **clone [远程仓库地址]** 克隆（复制）一个已存在的远程仓库到本地
* **git branch** 查看本地所有分支及其对应的提交记录
* **git branch -r** 查看远程所有分支
* **git branch -a** 查看本地和远程所有分支
* **git init** 在当前目录初始化一个新的空仓库。
* **git status** 查看工作区、暂存区与最近一次提交之间的状态变动情况。
* **git diff** 显示工作区与暂存区或者两个不同版本之间的差异。
* **git log** 查看项目的提交日志。
* **git checkout [分支名/commit ID]** 切换到指定分支或特定的提交记录。
* **git merge [分支名]** 合并指定分支到当前分支。
* **git remote add origin [远程仓库地址]** 关联本地仓库与远程仓库。
* **git fetch** 获取远程仓库的最新更新。
* **git reset --hard HEAD~n** 回退到上一次提交之前的第 n 次提交。
* **git stash** 将未完成的工作保存起来，用于后续恢复。
* **git clean -f** 清除未被 Git 管理的文件。
* **git rm [文件名]** 删除指定文件。
* **git mv [原文件名] [新文件名]** 重命名文件。

# 设计模式

常见的七种设计模式：

1. 单例模式：一个类只有一个实例，且该类能自行创建这个实例的一种模式
2. 工厂方法模式：实例化对象不是用new，用工厂方法替代。将选择实现类，创建对象统一管理和控制。从而将调用者跟我们的实现类解耦
3. 抽象工厂模式：提供了一个创建一系列相关或者相互依赖对象的接口，无需指定它们具体的类
4. 代理模式：由于某些原因需要给某个对象提供一个代理用来控制对该对象的访问
5. 装饰器模式：动态的将新功能附件到对象上。
6. 观察者模式：对象间的一种一对多依赖关系。当一个对象状态发生改变时，与其相关的依赖对象都能够得到通知并自动更新
7. 责任链模式：让多个处理器都有机会处理请求，直到其中的某个处理成功

# object.prototype.toString.call()

用来检测数据类型，基本数据类型和引用数据类型都可以被检测

# CSS选择器

ID选择器

类选择器

属性选择器

伪类选择器

标签选择器

优先级：

!important -> 内联样式 -> ID选择器 -> 类选择器/属性选择器/伪类选择器 -> 元素选择器

# CSS3新特性

1. 过渡：transition: all, 0.5s
2. 动画：Animation
3. 形状转换：Transform
4. 选择器：nth-of-type()
5. 阴影：text-shadow
6. 边框：border-image:url()
7. 弹性布局：display:flex
8. 媒体查询：@media

# async/await

async：异步。用来声明一个异步方法，返回一个Promise对象。

await：用来等待异步方法执行。后面接的如果是一个Promise对象，返回这个对象的结果，并阻塞后面的代码；如果接的不是Promise对象，就返回对应的值，不阻塞后面的代码。

async function f1(){

    console.log(1);

    await f2()

    console.log(2); // 代码在await后面，被阻塞

}

async function f2(){

    console.log('f2');

}

f1()

console.log(3);

// 打印

// 1 -> f2 -> 3 -> 2

await阻塞后面的代码执行（加入微任务队列）。先执行async外面的同步代码，同步代码执行完（获取到Promise对象resolve的值），再回到async函数中，执行之前阻塞的代码。

# 遍历树

图示

描述已自动生成

方法：在遍历树的时候，只需记录当前节点的值，然后针对左子树和右子树进行递归遍历。在遍历过程中，针对特点节点，需遍历：当前节点、左子树、右子树

1. 先序遍历 5 -> 3 -> 1 -> 2 -> 4 -> 6

前：代表当前节点需要在最前面进行遍历

顺序：当前节点 -> 左子树的前序遍历 -> 右子树的前序遍历

方法：递归、循环

屏幕上有字

描述已自动生成

电脑屏幕截图

描述已自动生成

1. 中序遍历 1 -> 2 -> 3 -> 4 -> 5 -> 6

中：在遍历时，将当前节点放在中间遍历

顺序：中序遍历左子树 -> 当前节点 -> 中序遍历右子树

方法：递归

1. 后序遍历 2 -> 1 -> 4 -> 3 -> 6 -> 5

后：将当前节点放在最后遍历

顺序：后续遍历左子树 -> 后续遍历右子树 -> 当前节点

方法：递归、循环

电脑的屏幕

描述已自动生成

文本

描述已自动生成

# Bootstrap

Bootstrap是一个基于HTML、CSS和JS的一个简洁、直观的前端开发框架

它是由动态的CSS语言less写成的。

它的CSS有几个特性：全局的CSS设置、可以定义基本的HTML样式、可扩展的class类

它里面包含了很多种可复用的组件：菜单栏、导航、提示框、弹出框等等

它相当于可以在很短的时间内，快速的搭建一个功能较为完善的网站

# jQuery

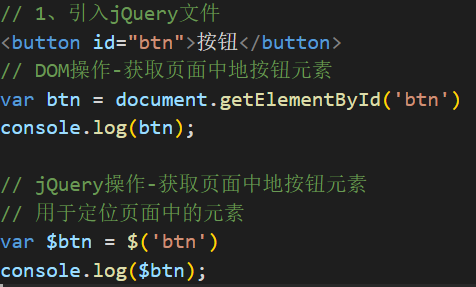
jQuery是一个JavaScript类库

目前这个jQuery库里面主要包含核心库、UI、插件和一些其他的模块

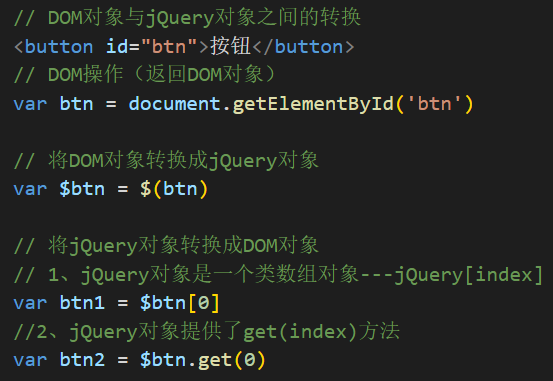
jQuery有一个工厂函数，这个函数也是作为jQuery操作的一个入口，通过这个工厂函数可以实现定位页面的元素，将DOM对象包装成一个jQuery对象（类数组对象）。

它有两种写法：$(selector) / jQuery(selector)

1. 定位页面元素



2、DOM对象和jQuery对象相互转换



# Node.js

NodeJs是JavaScript的运行环境，它是用来支持JavaScript代码的执行的。

在此之前，JS运行环境是浏览器，也就是JS必须要在网页中才能运行，现在可以在nodejs环境运行，所以可以用js去开发移动端的应用。

简单来说，NodeJs可以用来控制系统文件的读写，网络的输入输出。

NodeJS的包管理生态是 Npm，可以下载包，安装包，卸载包等

webpack是一个运行在nodejs端的静态模块打包工具，将所有后缀为.js的文件打包成一个bundle.js。主要功能就是代码的分割、模块化，提取公共代码。

所以打包首先第一件事就是文件的读取（获取所有后缀为.js的文件）

vue-cli也是运行在nodejs环境下的一个脚手架。因为nodejs有文件读写的能力，所以vue-cli可以帮助集成（打包）vue核心模块，并将代码转义、压缩。

# Ajax

它是一种异步请求数据的开发技术，不需要重新刷新当前的页面

它的核心对象就是XMLHttpRequest。

原理：

1. 浏览器让xhr向服务器请求数据
2. 浏览器继续干其他的事情
3. Xhr向服务器请求数据
4. 服务器返回数据给xhr
5. Xhr通知浏览器数据请求回来了
6. 浏览器接收xhr请求的数据并渲染页面

如何使用：

1. 创建ajax核心对象XMLHttpRequest
2. 向服务器发送请求（GET / POST）
3. 服务器响应处理，返回状态码和数据

# Axios

它是一个基于promise的网络请求库，可以用于浏览器和nodejs

文本

描述已自动生成

# npm

它是nodejs的包管理工具，用来安装各种nodejs的扩展

npm官网

# vue-cli

它是vue.js官方提供的项目脚手架

它可以快速搭建vue开发环境以及对应的webpack配置

通过 npm install -g @vue/cli 全局安装

# 跨域

什么是跨域：浏览器不能执行其他网站的脚本

当一个请求url的协议、域名、端口三者中的任意一个与当前页面不同就是跨域

跨域限制访问 🡪 浏览器的限制

解决跨域：

1. JSONP：使用script标签来请求跨域数据

文本

描述已自动生成

1. CORS：跨域资源共享。允许浏览器向服务器发送XMLHttpRequest请求，解除ajax只能同源使用的限制。

通过在HTTP请求头header中添加特殊的一个字段来允许浏览器跨域访问

# nginx

它是一个高性能的HTTP和反向代理web服务器

代理服务器：指局域网内部的机器通过代理服务发送请求到互联网上的服务，一般作用于客户端。

正向代理：为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标（原始服务器），然后代理向原始服务器转交请求并将获得的内容返回给客户端

反向代理服务器：在服务器端接收客户端的请求，然后把请求分发给具体的服务器进行处理，然后再将服务器的响应结果反馈给客户端