vuex 怎么合理规范管理数据,及 mutations 和 actions 区别：

1.不同于 redux 只有一个 action, vuex 单独拎出了一个 mutations, 它认为 更新数据必须是同步的, 也就是只要调用了提交数据方法, 在 mutation 里面才可以修改数据。那么如果我们想做 异步请求,怎么做? 这里 vuex 提供了专门做异步请求的模块,action, 当然 action 中也可以做同步操作, 只不过 分工更加明确, 所有的数据操作 不论是同步还是异步 都可以在 action 中完成, mutation 只负责接收状态, 同步完成 数据快照，所以可以认为

state => 负责存储状态 mutations => 负责同步更新状态

actions => 负责获取 处理数据（若有异步操作必须在 action 处理再到 mutation），提交到 mutation 进行状态更新。

2.vuex 模块化 module 管理：

使用单一的状态树，应用的所有状态都会集中在一个比较大的对象上面，随着项目需求的不断增加，状态树也会变得越来越臃肿，增加了状态树维护的复杂度,而且代码变得沉长；因此需要 modules(模块化)来为我们的状态树分隔成不同的模块，每个模块拥有自己的 state，getters，mutations，actions；且允许每个 module 里面嵌套子 module。

需要注意，原来使用 vuex 辅助函数 mapMutations/mapActions 引入的是全局的的 mutations 和 actions，并且我们 vuex 子模块 也就是 module1,module2 ... 这些模块的 aciton /mutation 也注册了全局, 也就是若 module1 中定义了 loginMutation, module2 中也定义了 loginMutation, mutation 就冲突了，会报错了。如果不想冲突, 各个模块管理自己的 action 和 mutation ,需要给子模块一个 属性 namespaced: true 。

```
 modules: {   // 模块:{  一套 state action mutation }
   m1: {
     namespaced: true,   //开启命名空间: 是 m1 下的 不会影响其它模块
     // 模块内容（module assets）
     state: { // 模块内的状态已经是嵌套的了，使用 `namespaced` 属性不会对其产生影响
       m1Name:"我是 m1 模块的 m1Name"
     },
     actions: {
       loginAction () {
           console.log('m1 的 action')
       } // -> dispatch('m1/loginAction')
     },
     mutations: {
       loginMutation () {
         console.log('loginMutation-执行啦')
       } // -> commit('m1/loginMutation')
     }
   },。。。。。。}
```

可以将模块的空间名称字符串作为第一个参数传递给 map...函数，这样所有绑定都会自动将该模块作为上下文

```
 methods:{
       //不是 modules 里面的的直接写   ...mapMutations( ['loginMutaton])
     ...mapMutations('m1', ['loginMutation']), // 解构出全局 m1 模块下的 loginMutation，放到 this 上，简化了代码
      add(){
        console.log('add',this)
       //   this.$store.commit("m1/loginMutation")
       // 或者下面的  先 mapMutations 相当于帮你写了 commit
       // this.loginMutation() //可以直接调用，帮你简化了代码
     }
  }
```

组件是本质是什么? 组件就是一个单位的 HTML 结构 + 数据逻辑 + 样式的 操作单元

Vue 的组件 继承自 Vue 对象, Vue 对象中的所有的属性和方法,组件可自动继承.

组件的要素 template =>作为页面的模板结构 script =>作为数据及逻辑的部分 style =>作为该组件部分的样式部分

封装 Vue 组件的步骤？着手模板的结构设计及搭建,也就是 html 结构部分，先完成静态的 html 结构；
结构完成，着手数据结构的设计及开发，数据结构一般存储于组件的 data 属性 或者 vuex 状态共享的数据结构；
数据设计完成，着手完成数据和模块的结合,利用 vuejs 中指令和 插值表达式的特性 将静态结构 动态化；
展现的部分完成，着手完成交互部分,即利用组件的生命周期钩子函数和事件驱动完成逻辑及数据的处理与操作；
最后组件完成,进行测试及使用。
常用的组件属性 => data/ methods/filters/ components/watch/created/mounted/beforeDestroy/computed/props
常用组件指令: v-if/v-on/v-bind/v-model/v-text/v-once

Vue 中的 data 是以函数的形式还是对象的形式表示（为什么组件要求必须是带返回值的函数）
如 data(){ return { name:"xxx" }? 因为组件在实例化的时候，会直接将 data 数据作用在视图上。对组件实例化，会导致我们组件的 data 数据进行共享，这显然不符合我们的程序设计要求，我们希望组件内部的数据是相互独立的,且互不响应,所以 采用 return {} 每个组件实例都返回新对象实例的形式,保证每个组件实例的唯一性，返回新对象，所以如果该组件使用多次，用 函数形式，每次都是返回新对象，组件间就不会互相影响。否则如果不是函数写法，直接写对象，那么地址（索引）都是一样的，就会互相影响。

解决跨域问题的方式有：
1.服务端设置，但这种方式依赖服务端的设置,在前后分离的场景下，不太方便
```
Access-Control-Allow-Origin：*
Access-Control-Allow-Methods："POST, GET, OPTIONS, DELETE"
```
2.jsonp 形式，可以利用 script 标签的特性解决同源策略带来的跨域问题,但这是这种方案对于请求的类型有限制,只能 get。
3.可以在开发环境(本地调试)期间,进行代理，就是通过在本地通过 nodejs 启动一个微型服务，然后先请求这个微型服务，微型服务是服务端，服务端代理去请求相应的跨域地址，因为服务端是不受同源策略的限制的，具体到开发中,打包工具 webpack 集成了代理的功能,可以采用配置 webpack 的方式进行解决，但是这种仅限于本地开发期间，等项目上线时,还是需要另择代理。
```
// 以下为 webpack 配置代理的配置，在 vue.config.js
module.exports = {
  // 修改的配置
  devServer: {
      proxy: {
          '/api': {
              target: 'http://122.51.238.153',
              changeOrigin: true,
              pathRewrite: {
                '^/api': ''
              }
          }
      }
  }
}
```
target：接口域名；
changeOrigin： 如果设置为 true,那么本地会虚拟一个服务端接收你的请求并代你发送该请求；
pathRewrite：如果接口中是没有 api 的，那就直接置空，如果接口中有 api，就需要写成{‘^/api’:’’}

Vue 中的 watch 如何深度监听某个对象，两种方式：
1.字符串嵌套方式
```
export default {
    data () {return {a: {b: {c :'张三'}}}},
```

```
    watch: {
        //想监听 c  此时 数据 是 a.b.c  比较深   深度监听
        "a.b.c": function (newValue, oldValue) {   ......
         }
    }
}
```

2.启用深度监听方式

```
export default {
    data () {return {a: {b: {c :'张三'}}}},
    watch: {
        a: {
            deep: true //  开启深度监听  a 对象里面任何数据变化都会触发 handler 函数,
            handler(){
                // handler 是一个固定写法
            }
        }
    }
}
```

Vue 的 keep-alive 的使用：keep-alive 是 Vue 提供的一个全局组件, Vue 的组件是有销毁机制的，如条件渲染、路由跳转时，组件都会经历销毁，再次回到页面时,又会重生，这一过程保证了生命周期钩子函数各个过程都会在这一生命周期中执行。但获取的数据会归零,这影响了交互的体验, 所以 keep-alvie 出现了, 可以帮助我们缓存想要缓存的组件实例, 用 keep-alive 包裹要缓存的组件实例, 此时组件创建之后,就不会再进行销毁, 组件数据和状态得以保存。但没有销毁，就失去了重生的环节，失去原有的钩子函数, 所以 keep-alive 包裹的组件都获取了另外两个事件，即如果缓存组件需要重新获取数据：

1.唤醒 activated 重新唤醒休眠组件实例时执行 2.休眠 deactivated 组件实例进入休眠状态时执行

可以针对组件容器 router-view 这个组件进行缓存, 一般的策略是在路由的元信息 meta 对象中设置是否缓存的标记, 然后根据标记决定是否进行缓存。

在根目录》src 文件夹》router 文件夹》index.js 文件：

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Home from '../views/Home.vue'
Vue.use(VueRouter)
const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home,
    // meta  标识符  是路由配置里面可以写的，可以用来  判断一些操作
    meta:{   //  名字：值
      isAlive:false // false 代表不缓存  需要缓存的就写  isAlive:true
    }
  },
  {
    path: '/about',
    name: 'About',
    component: () => import(/* webpackChunkName: "about" */ '../views/About.vue'),
    meta:{
      isAlive:true // about 组件  需要缓存
```

```
    }
  }
]
const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})
export default router
```

在 vue 组件里：

```
  <div id="app">
    <keep-alive>
      <!-- 里面是当需要缓存时    显示组件的 router-view-->
      <router-view   v-if="$route.meta.isAlive" />
    </keep-alive>
     <!-- 外面是不需要缓存时 -->
    <router-view   v-if="!$route.meta.isAlive" />
  </div>
```

要注意，被缓存的组件中如果还有子组件，那么子组件也会拥有 激活和唤醒事件，且这些事件会在同时执行。

vue 的双向数据绑定原理：Vue 的双向绑定原理其实就是 MVVM 的实现原理， Vuejs 官网已经说明，实际就是通过 Object.defineProperty 方法 完成了对于 Vue 实例中数据的劫持，通过对于 data 中数据 set 的监听，然后通过观察者模式，通知 对应的绑定节点 进行节点数据更新，完成数据驱动视图的更新；同理，通过对于节点的表单值改变事件的监听，执行对于数据的修改。

简单概述 ：通过 Object.defineProperty 完成对于数据的劫持，通过观察者模式，完成对于节点的数据更新

```
// 那 vue 的 data 那么多数据怎么办：vue 里面 就 for 循环执行下面大概的代码 全部 data 里的数据监听到
Object.defineProperty(this.data,'name',{
  get:function(){   // 当获取监听对象的某个 值  就可以 执行 get 函数
    console.log('get 获取了值')
  },
  set:function(newval){ // 设置的新值
      console.log('set 设置了值',newval)
      // 当然 vue 没有这么简单去找 他写了很多正则表达式去替换，但是思路是这个
    // 只需要 监听到 name 值改了  就去页面修改 对应的地方就行 变成新值
    let con=document.getElementById("con")
    // con.innerHTML  获取内容  name 的值是:{{name}}  .replace("查找的字符串","替换成这个")
    let str=con.innerHTML.replace("{{name}}",newval)
    // 重新修内容  innerHTML 是获取内容 设置内容的
    con.innerHTML=str
  }
})
```

如何在组件使用全局数据？

1 在 html 范围 直接 $store.state.名字 2 在 js 范围 this.$store.state.名字

页面刷新了之后 vuex 中的数据消失怎么解决：vuex 数据位于内存，页面的刷新重置会导致数据的归零,也就是所谓的消失，本地持久化可以解决这个问题。

本地持久化用到的技术也就是 本次存储 sesstionStorage 或者 localStorage 。若需要保持的更长久，如浏览器关掉再打开依然存有数据，需要使用后者。

实施方案: state 的持久化 也就是分别需要在 state 数据初始化 /更新 的时候 进行读取和设置本地存储操作：
代码如下

```
export default new Vuex.store({
state: {
    user: localStorge.getItem('user')  // 初始化时读取 本地存储
},
mutations: {
    updateUser (state, payload) {
        state.user = payload.user
        localStoregae.setItem('user',payload.user) // 数据更新时 设置本地存储
    }
} })
```

nuxt.js ： 是 vue 语法版本的 服务器渲染(ssr)框架
为什么要做服务端渲染：有对应的需求，即对 实时到达时间(页面访问时间)的绝对需求。如果只是简单的一个管理系统, 区区几百毫秒的优化 显得十分小题大做。而且 vue 单页面应用渲染是从服务器获取所需 js，在客户端将其解析生成 html 挂载于 id 为 app 的 DOM 元素上，这样会存在两大问题：由于资源请求量大（或 js 文件可能很大），造成网站首屏加载缓慢，不利于用户体验；由于页面内容通过 js 插入，对于内容性网站来说，搜索引擎无法抓取网站内容，不利于 SEO。而 Nuxt.js 是一个基于 Vue.js 的通用应用框架，预设了利用 Vue.js 开发服务端渲染的应用所需要的各种配置。可以将 html 在服务端渲染，合成完整的 html 文件再输出到浏览器。

服务端渲染有一个成熟优秀的框架 nuxt.js ，nuxt 是 vue 服务端渲染的优秀解决方案
nuxt 的出现可以让渲染内容完全服务端化，解决 seo 不够友好，首屏渲染速度不够迅速的问题。
要注意: 不是所有页面都需要服务端渲染，因为服务端渲染比重大，对于服务器的访问处理能力要求也会增大。
nuxt 脚手架 不需要 安装，nodejs 默认自带了
1 脚手架 npx create-nuxt-app <项目名> 2 进入项目 yarn dev 启动开发 3 上线 yarn build/yarn start

nuxt 与 vue 其它的区别：
1.路由 nuxt 按照 pages 文件夹的目录结构自动生成路由，如 http://localhost:3000/user/reg 相当于去访问 pages 文件夹下的 user 文件夹下的 reg.vue；而 vue 需在 src/router/index.js 手动配置路由
2.入口页面 nuxt 页面入口为 layouts/default.vue vue 页面入口为 src/App.vue
3.nuxt（标签）类似 router-view，nuxt-link 类似 router-link
4.nuxt 内置 webpack，允许根据服务端需求，在 nuxt.config.js 中的 build 属性自定义构建 webpack 的配置，覆盖默认配置；vue 关于 webpack 的配置存放在 build 文件夹下
5.asyncData 里面发送 ajax 这个东西跟生命周期这些都是平级的：
cnpm install @nuxtjs/axios --save plugins 目录新建 axios.js：

```
import * as axios from 'axios'
let options ={}
//需要全路径才能工作
if(process.server){
    options.baseURL=http://${process.env.HOST || 'localhost'}:${process.env.PORT || 3000}/api
}
export default axios.create(options)
```

Nuxt.config.js 里增加 axios 配置：

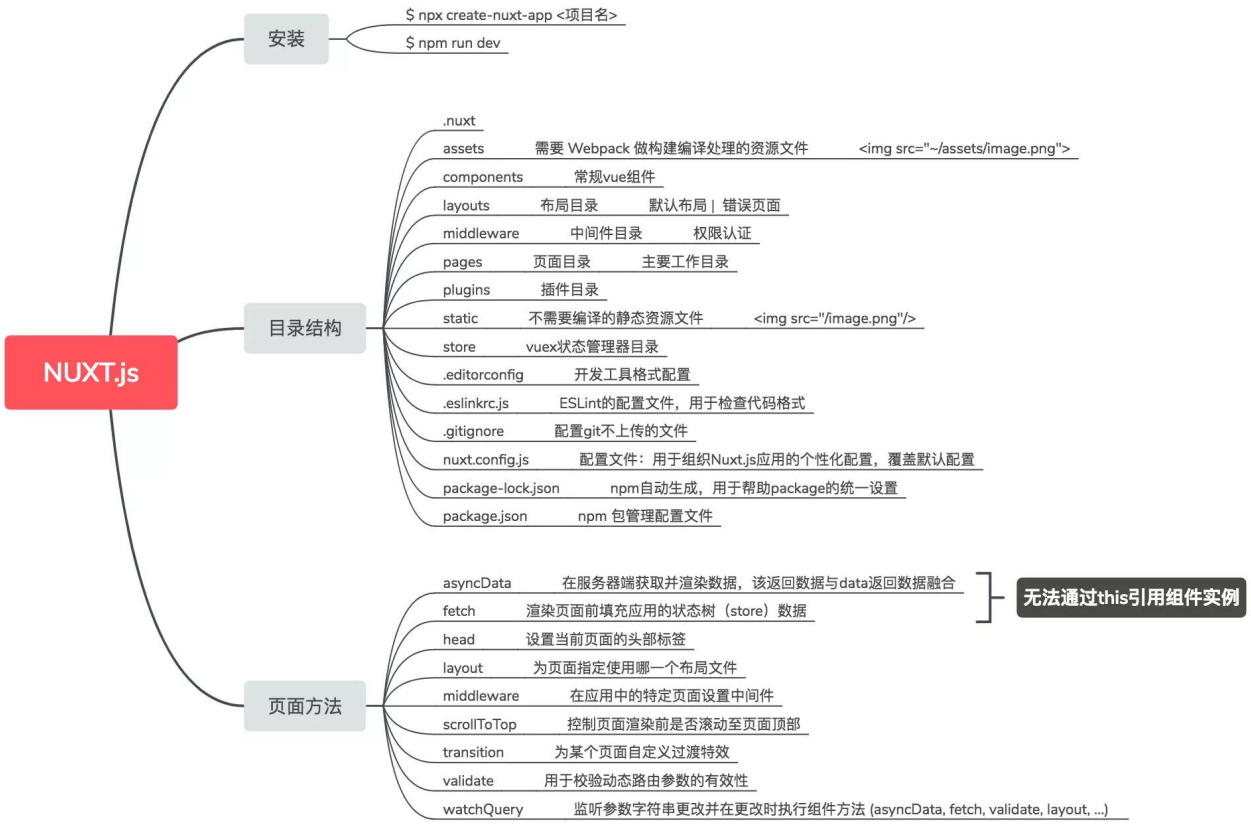
```
modules:[
    '@nuxtjs/axios'
],
```

组件里，使用 asyncData 里面发送 ajax 这个东西跟生命周期这些都是平级的 在页面渲染之前：

```
export default {
```

```
  async asyncData({app}){
            let res =await app.$axios({
                headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
                method: 'get',
                url: `http://test.yms.cn/testjson.asp`,
                data: ''
            })
            console.log('res',res.data)
            return {
                testData:res.data.title
            }
    },
    created(){
      console.log('nuxt reg 组件')
    }
}
```

关于 nuxt 的简单概述：



vue-router 传参（vue-router 传值）可以通过地址传值

最简单的就是 url 传值, url 传值有两种, params 和 query 参数传值：

1.params 传值，指动态路由传值：

```
{ path: '/user/:id' }  // 定义一个路由参数
<router-link to="/user/123"></router-link>  // 传值，'/user/:id'  ---123 就是 id
this.$route.params.id  // 取值 专门获取 :id 这种参数 params 参数
```

2.query 传值，指通过?后面的拼接参数传值

{ path: '/user' } // 定义一个路由参数
<router-link to="/user?id=123&name=zs"></router-link> // 传值
this.$route.query.id // 取值 获取 ?后面的参数 query 参数 查询字符

前端鉴权一般思路（前后分离的鉴权思路）（为什么要在前端鉴权）：
1.传统项目都是在后端鉴权，然后通过进行拦截 跳转 进行对应操作。因为做的并不是传统的项目，而是前后分离项目，即前端项目和后端服务进行了剥离，后端没有办法用 session 来存储任意一个前端项目域名下的身份信息，所以 jwt 鉴权模式应运而生，即后端不再提供会话的身份存储，而是通过一个鉴权接口将用户的身份，登录时间，请求端口，协议头等信息 组装成一个加密的串 返给前端请求，前端拿到了这个串，就可以认为登录成功，这个加密串就成了前端用户是否登录的成功标志，这就是 token，那么在接下来的接口请求中，几乎都要携带这个加密串，因为它是唯一能证明身份的信息。
2.为了方便，一般在请求工具 ，如 axios，的拦截器中统一注入 token，减少代码的重复；token 同时具有时效性，要在此时对 token 过期进行处理，一旦出现过期的请求码，就需要进行换取新 token 或者重新登录的解决方案。
3.除此之外，还要依据有无加密串 在前端对于某些页面的访问进行限制，这个会用到 Vue-Router 中的导航守卫。vue 单页项目涉及到多角色用户权限问题，不同的角色用户拥有不同的功能权限，不同的功能权限对应的不同的页面。一开始 有一些 默认的路由，登录后 若你是总经理 后台会返回给前端 总经理能看见的 路由页面地址（数组），前端在 router.beforeEach 路由导航守卫里面 拿到返回的地址 使用 router.addRouter 动态加上 这个项目路由就行

```
// 路由导航守卫
router.beforeEach((to, from, next) => {
        //判断 user 信息是否已经获取，已经登录，登录后就把后台给.的路由数组 addRouter 就行
    if (token) {
        //根据用户的角色类型来生成对应的新路由，在这里要用 登录时候后台返回的 路由数组
         // 建议把那个数组 写在 vuex 里，从 vuex 拿出 登录时候存的 newRouter
         // const newRouter = [{path:"/xxx" ...} ..]
         //将新路由添加到路由中， router.addRoutes 是 vue 自带的专门用来追加路由的
         //   router.addRoutes(newRouter)
        //为了正确渲染导航,将对应的新的路由添加到 vuex 中
          // 渲染对应的侧边栏
    }
})
```

如果是前端 就 先写一个 全的 所有的 路由数组，登录之后 后台 返回 当前的登录人的 数据，如路由列表或者这个人的权限，前端 拿到路由列表 去循环 和那个全的 路由 对比 拿到相关的 在 addRouter 添加就行。

vue 数据流 和 react 数据流（不是问你双向绑定）应该说：
1.在 vue 和 React 中数据流向是单向的，由父节点流向子节点，如果父节点的 props 发生了改变，那么 React 会递归遍历整个组件，父组件通过绑定 props 的方式，将数据传递给子组件，但是子组件自己并没有权利修改这些数据，如果要修改，只能把修改的这个行为通过 event 的方式报告给父组件，由父组件本身决定改如何处理数据。
2.vue 中，有另一个概念 v-model 双向数据，无论数据改变或用户操作，都能带来互相的变动，自动更新。v-model 一般用在 表单元素，核心原理是绑定 value + oniput 文本框值改变的事件。

如何在组件中监听 Vuex 的数据变化？
首先，Vuex 是为了解决组件间状态共享而出现的一个框架。其中有几个要素是组成 Vuex 的关键, state(状态) mutations actions。state 表示 需要共享的状态数据；mutations 表示 更改 state 的方法集合 只能是同步更新 不能写 ajax 等异步请求；actions 如果需要做异步请求 可以在 actions 中发起 然后提交给 mutations mutation 再做同步更新。
即 state 负责管理状态， mutation 负责同步更新状态，action 负责 异步获取数据 提交给 mutation；所以 组件监听 Vuex 数据变化 就是 监听 Vuex 中 state 的变化：
1.第一种方案，可以在组件中通过组件的 watch 方法来做，因为组件可以将 state 数据映射到 组件的计算属性上，然后 监听 映射的计算属性即可。

```
// vuex 中的 state 数据
```

```
  state: {
    count: 0
  },。。。。。。。
//   A 组件中映射 state 数据到计算属性
  computed: {
  // mapState 方法 把全局  count 变成 可以直接使用的 数据
    ...mapState(['count'])
  }
// A 组件监听 count 计算属性的变化
   watch: {
     // watch 可以监听 data 数据 也可以监听 全局 vuex 数据
    count () {
      // 用本身的数据进行一下计数
      this.changeCount++
    }
  }
```

2.第二种方案，vuex 中 store 对象本身提供了 watch 函数 ,可以利用该函数进行监听

watch(fn: Function, callback: Function, options?: Object): Function

响应式地侦听 fn 的返回值，当值改变时调用回调函数。fn 接收 store 的 state 作为第一个参数，其 getter 作为第二个参数。最后接收一个可选的对象参数表示 Vue 的 vm.$watch 方法的参数。

```
  created () {
    this.$store.watch((state, getters) => {
      return state.count
    }, () => {
      this.changeCount++
    })
  }
```

Vue 的多页面的使用：配置多入口文件，只需要 配置 vue.config.js 和对应好文件夹就行。一个项目分成很多小 vue 项目，其实也可以直接创建两个项目。

单页面应用和多页面应用的根本区别：单页面即所有的模块统统置于一个 html 文件之上，切换模块不会重新对 html 文件和资源进行再次请求，服务器不会对换页面的动作产生任何反应, 所以没有任何的刷新动作，速度和体验很畅快；多页面应用，即多个 html 页面 共同的使用，可认为一个页面即一个模块，但是不排除 多个单页应用混合到一起的组合情况 ，多页面切换一定会造成 页面资源的重新加载，这意味着多页面之间切换，会造成数据的重置。

1.新建多个文件夹 每个文件夹里的页面是一个单独的小 vue 类型项目

2 配置 多入口页面文件在 vue.config.js 里写上这些（重点是入口选择对应页面的 main.js）：

```
module.exports = {
  pages: {
    index: {
      // page 的入口
      entry: "src/views/index/main.js",
      // 模板来源
      template: "public/index.html",
      // 在 dist/index.html 的输出
      filename: "index.html",
      // 使用 title 选项时，template 中的 title 标签需要是 <title><%= htmlWebpackPlugin.options.title %></title>
      title: "Index Page"
    },
```

```
    ui: {
      // page 的入口
      entry: "src/views/ui/main.js",
      // 模板来源
      template: "public/ui.html",
      // 在 dist/ui.html 的输出
      filename: "ui.html",
      // 使用 title 选项时，template 中的 title 标签需要是 <title><%= htmlWebpackPlugin.options.title %></title>
      title: "ui Page"
    }
  }
};
```

3.public 写上不同的渲染的 html 文件

4.main.js 不同的入口 对应上相应的 根组件和 页面元素

5.通过 a 标签跳转，如

```
<div id="app">
    另一个 ui 页面啊
    <a href="home.html">去 home 页面</a>
  </div>
```