自我介绍

你好，我叫马超，是一名 19 级的本科生，来自于哈尔滨理工大学，本科专业是数据科学与大数据技术，应聘的岗位是 web 前端实习生，我在大三开始自学前端，前端的基础课程大约都学过了，之前在联想前端实习四个月。这些就是我的基本情况。

要问的问题

1. 如果去实习做什么工作，需要做 node 的工作嘛。老项目新项目
2. 技术栈
3. 导师
4. 转正，三方，转正周期

待遇多少薪
五险一金
人才补贴，住房补贴，落户补贴
吃住
时间，加班

1.城市发展前景
2.城市风景
3.压力
4.吃住
5.产业发展空间

1) 薪资
2) 薪资结构，绩效
3) 多少薪
4) 试用期多久，试用期工资
5) 福利补贴，五险一金
6) 上班时间，单双休，加班费

实习经历

项目名称：`PCRManagementAndTrackingSystem`

项目内容

使用基于 Vue2+ElementUI 开发和维护联想日常办公系统，该系统为联想内部自用的任务派发系统，本人主要负责管理员模块下的前端部分的新需求开发，以及在使用中发现的 bug 的解决，使系统的任务派发，审批功能的正常运转。

实习职责

(1) 部门领导会根据内部员工的使用体验提出改进方案，我的任务就是实现每周提出的新需求所对应的模块。

(2) 由于是旧系统的迁移，日常使用中会出现小 bug，需要解决这些 bug，还包括系统完善及维护工作，以及项目改动之后的前端代码部署到服务器上。

(3) 旧系统的代码冗余，没有很好的代码约束，团队做出了新的代码规范，需要去优化代码书写和命名规则。

(4) 每周例会分配下的新需求，需要去分配前后端具体任务点，以及上周需求实现的系统展示。

1. 直接点击链接跳转新页面
2. 点击复制链接打开新窗口
3. 微软封装的库
4. 对话框弹出链接，点击跳转 js 开的新窗口，后端 tomcat 配置开放路径

实习总结

在目前的实习中，接触了团队开发，熟悉了团队开发办公的流程。知道了代码规范整洁意识对一个团队的重要性，也了解了一些项目部署相关的操作。也发现了自己的不足，对于知识的掌握牢记程度还是不够，有时需要再去查阅，在日后的学习中需要对知识的深度和广度加深学习。另外学习了 Git 团队开发流程。

项目 1

基本情况

前端应用 react 全家桶和 react-redux 配合后端 Node Express 和 MySQL, sequelize 实现了一个前后端分离的博客系统。

具体

后端是使用 Node 配合框架 Express 搭建，数据存储用的是 MySQL 数据库，采用了 sequelize 来连接数据库，初始化数据模型，并建立数据模型关系。项目整体采用分层架构，模块化开发，采用 MVC 模式对系统架构进行划分，降低层与层之间的依赖，复用代码逻辑，使系统功能逻辑清晰。在登陆时采用 jsonwebtoken 加签解签的方式对访问其他接口加以验证，自定义了中间件处理机制，如用户认证处理中间件，错误处理中间件，路由匹配错误中间件来处理系统中产生的各类

错误。接口的形式采用 restFul 形式，利用 ApiFox 进行接口测试。并配置了 cors 来解决跨越问题

前端主要采用的是 react 全家桶，使用 create react app 脚手架进行项目结构搭建，项目中所请求回来的数据使用 react-redux 进行数据状态管理。数据交互自定义封装了 fetch 请求工具来请求后端数据。使用 connect-react-router 对路由进行封装，在 redux 使用路由，并对页面跳转路由进行监听，由于封装了路由守卫组件，在页面跳转时进行登陆状态判断，未登录则跳转登陆界面，组件加载方式采用了懒加载的方式，页面无关组件被拆分，无关组件被加载时才会下载到本地。

功能

注册和登录，密码校验修改，个人信息修改和常用的增删改查功能。文章的创建删除编辑获取功能，添加取消关注，添加取消喜欢，创建删除获取评论功能，创建获取删除标签。

Blog 系统首页组件涉及的业务逻辑比较复杂，首先是由 react-redux 搭建 UI 容器，然后需要获取所有标签，然后判断当前登录状态，根据登录状态向后台请求不同的标识，如果未登录就获取全局文章，如果已经登陆就获取作者已关注作者的文章。然后将选项卡标签绑定点击事件，根据不同的标签获取不同标签下的文章，点击后还包括同步标签同步页码同步文章数据操作，还需要将同步获取来的数据传递给分页组件进行组件间的传值供子组件使用。

后台获取文章数据时，需要将关联查询到的信息包括文章喜欢的数量，是否喜欢等进行数据的封装然后一并返回给前端使用，包括处理 tag，进行数据脱敏，返回作者信息等操作。

还包括组件复用，组件间传值的功能，调用的接口比较多，和一些细节上的处理，包括防抖节流，数据校验等。

路由：编程式路由 this.$router.push
路由模式：history 模式，在 store 中引入了 history

模块

登录注册个人信息修改模块，文章的增删改查模块，关注模块，喜欢模块，评论模块，标签模块

难点解决

(1) Jwt token 验证

使用 jsonwebtoken 将用户信息在登陆时生成一个 token，在路由模块访问其他接口时对 token 的存在性和格式进行验证，存在就进行解签操作，达到安全传输信息的目的。

(2) 在 redux 里面封装路由

为了能直接在 action 里面请求数据并直接跳转路由，使用 routerMiddleware 在 store 上挂载了 history 路由，并使用 connectRouter 将 router 和 react-redux 连结起来，达到在 redux 里面派发路由的目的。

(3) 登录信息 token 存储

由于组件嵌套生命周期执行顺序原因，在根组件获取登录信息，子组件生命周期先于父组件执行，会导致子组件生命周期加载获取不到登录信息，所以将登录信息初始化在处理登录的 reducer 里面，当子组件生命周期加 载就会直接拿到登录的用户信息。

(3) 项目分层架构，模块化开发

采用 MVC 模式对系统架构进行划分，降低层与层之间的依赖，复用代码逻辑，使系统功能逻辑清晰。

(4) 静态服务

将系统所用到的样式包括 bootstrap，字体图标 font-awesome 集成在后端静态资源文件夹，当后端启动服务之后，静态资源中的样式供前端使用。

Bug：

1.数据库数据定义：DataTypes.TEXT 不能做主键，顺便解决了 logging = true 带来的 bug

```
  username: {
        type: DataTypes.STRING,
        unique: "username", //
        allowNull: false
    },
```

2.嵌套层级过多 props 丢失 不增强组件

3.封装 router 页面不刷新 react 版本不兼容

4.点击跳转路由，url 发生变化页面不变化，强制刷新

3.注册不上去，react 版本

4.组件嵌套，都在生命周期里面获取数据，componentDidMount，内部的生命周期不会执行，所以第一次渲染内部组件无数据，将获取数据都放到父组件里面

5.生命周期执行顺序(先执行内部的操作（render，didMount，actionReducer，全局 didMount），在执行全局外部的同步 user，导致没有数据),将获取 currentUser 函数放到 userReducer 里面,默认 currentUser 为获取 currentUser 函数

6.组件嵌套 componentDidMount 内部的生命周期不执行,创建评论首次获取不到 username，刷新一下才能好使(添加评论)

项目 2

基本情况

应用 Vue2.x 全家桶和 ElementUI 配合 Node 和 MongoDB 实现了一个前后端分离的教师学生管理系统。

具体

后端是使用 Node 配合它的框架 Express 搭建，数据存储用的是 MongoDB 数据库，因为涉及 MongoDB 验证和其他的业务逻辑比较麻烦，所以采用了 Mongoose，因为 Mongoose 里面封装了 MongoDB 的一些基本操作功能。

前端主要采用的是 Vue2.x，使用 VueCLI3.x 配合 ElementUI 进行项目结构搭建，项目中所需要的数据使用 Vuex 进行数据状态管理，项目整体采用 webpack 打包压缩。数据交互封装了 axios 组件来请求数据，并配置了 devServer 来解决跨越问题。

功能

注册和登录，密码校验，密码修改和常用的增删改查功能。

学生和教师组件涉及的业务逻辑比较复杂，首先是使用 ElementUI 搭建组件，前端传参进行接口调用请求数据，后端根据前端传过来的参数进行数据处理，并返回给前端，前端接受后端传过来的数据并渲染组件，功能包括新会话窗口添加，编辑和更新数据，和不同种请求数据的方式，还有一些组件复用，组件间传值的功能，调用的接口比较多，和一些细节上的处理，包括防抖节流，数据校验规则，延迟回调等。

组件：分页组件 el-pagination
路由：声明式路由 router-link 和编程式路由 this.$router.push 都有使用，router-view
路由模式：history 模式，在路由中指定了 mode:history

模块

登录注册模块，教师的增删改查模块，学生的增删改查模块，和密码的校验修改模块

难点解决

1. 通过路由守卫钩子在登陆时进行拦截，在本地 vuex 查看是否有 token，如果没有就通过 dispach 提交 GetUserInfo 异步获取用户信息，vuex 通过 commit 调用 SET_USER 将从后台获取的 token 通过 localstorage 存储为 TOKEN。

2. 连续点击新增数据按钮会在数据库中插入多条数据，如果实在大型应用中会造成服务器压力过大和系统性能问题，所以需要对其进行处理对点击操作进行防抖处理，在当前设置的时间内再次点击会取消前一次的点击，来避免插入多条数据的 bug。
3. 在页面未刷新时，表中的数据依然存在，由于新增和编辑是同一个组件的衍生，点击新增路由无法进行判断是新增还是编辑。可以添加一个_id:null 标识，在添加数据之后对页面中的数据进行清空，实现下一次新增数据不会对已有数据进行编辑。

虚拟 DOM 和 diff 算法

虚拟 DOM

起初我们在使用 JS/JQuery 时，会大量操作 DOM，而 DOM 的变化又会引发回流或重绘，降低渲染性能。虚拟 DOM 出现的主要目的就是通过对比新旧虚拟 DOM，复用没有变化的 DOM 节点，减少 DOM 操作，进而提高页面性能。

Diff 算法（Vue2）

根据真实 DOM 生成一颗虚拟 DOM 树，当虚拟 DOM 某个节点的发生改变后会生成一个新的虚拟 DOM，然后新老节点进行从上至下进行同层比对，如果上层已经不同了，那么下面的 DOM 全部重新渲染，提升算法效率，对比的过程就是调用 patch 函数，patch 函数会生成一个补丁包，用来描述新老节点改变的内容，然后将这个补丁打到真实 DOM 上更新 DOM。在 diff 的同时进行 patch。

首先，我们拿到新旧虚拟 DOM 节点的数组，然后初始化四个指针，分别指向新旧节点的开始位置和结束位置，进行两两对比，1.若是新的开始节点和旧开始节点相同，则都向后面移动，2.若是结尾节点相匹配，则都前移指针。3.若是旧开始节点和新的结束节点相匹配，则会将新的结束节点移动到旧结束节点的后面。4.若是旧结尾节点和新开始节点匹配上了，则会将新开始节点移动到旧的开始节点前。若是上述节点都没配有匹配上，则会进行一个循环判断，判断开始节点是否在旧节点中，若是存在则复用，若是不存在则创建。最终跳出循环，进行裁剪或者新增，若是旧的开始节点小于旧的结束节点，则会删除之间的节点，反之则是新增新的开始节点到新的结束节点。

key 值的作用：

每个 DOM 节点都有一个唯一的 key，通过判断新旧 DOM 的 key 是否相等，决定这个节点是进行复用还是重新创建新的节点，达到减少 DOM 操作节约性能开销的目的。
错误的用法：

1. index 做 key：当涉及排序删除节点时，key 值不会随节点的改变而改变顺序，依旧是按 0123 进行排序，达不到复用节点的目的。

2. index 拼接其他值做 key：index 拼接其他值作为索引的时候，当顺序发生改变而 key 不会改变而导致所有的节点都找不到相应的 key 而不能复用节点。

3. 随机数作为 key：新旧 DOM 节点都会生成不同的 key 导致所有的节点都会被删除重新创建节点导致严重的性能问题。

Vue3

Vue2.x 中的虚拟 DOM 是进行全量的对比，会遍历判断虚拟 DOM 所有节点，包括一些不会变化的节点有没有发生变化；虽然说 diff 算法确实减少了多 DOM 节点的直接操作，但是这个减少是有成本的，如果是复杂的大型项目，会不断地递归调用 patchVNode，最终就会造成 DOM 更新缓慢。

而 Vue3 在 DOM-Diff 过程中，根据新老节点索引列表找到最长稳定序列，通过最长增长子序列的算法比对，找出新旧节点中不需要移动的节点，原地复用，仅对需要移动或已经 patch 的节点进行操作，是对 diff 算法的一个优化。

1.最长前置与后置的预处理：找出列表中前后最长相邻可复用的节点

2.判断是否需要移动：先根据新列表剩余节点个数，创建一个 source 数组，并将数组填充满-1。将新列表中的节点对应在旧列表中的位置下标替换 source 中的-1，如果值还为-1，说明是新创建的节点。根据递增法判断如何移动，如果老节点下标小于新节点下标，说明其要进行移动。将其命名为 a，将 a 的 DOM 节点移动到 a 在新列表中前一个 VNode 对应的真实 DOM 的结点之后。

3.如何移动：根据 source 数组得到索引形式的最长递增子序列 lis。从后向前遍历 source 数组的每一项，1.当前的值为-1，这说明该节点是全新的节点，由于我们是从后向前遍历，直接创建好 DOM 节点插入到队尾。2. 当前的**索引**为最长递增子序列中的值，说明该节点不需要移动，原地复用。3. 当前的索引不是最长递增子序列中的值，说明该 DOM 节点需要移动，直接将 DOM 节点插入到队尾，因为队尾是排好序的。

4.最长递增子序列：

dp = new Array(len).fill(1)根据数组长度创建一个充满 1 的数组。

res.push([i]) 创建数组长度个子数组，数组里面用于存储递增子元素的下标。

双层循环，第一层存储当前元素

```
for (let i = len - 1; i >= 0; i--) {
    let cur = arr[i]
```

第二层存储下一元素

```
for (let j = i + 1; j < len; j++) {
        let next = arr[j]
```

满足递增条件

```
        if (cur < next) {
          let max = dp[j] + 1 最大长度加一
```

如果当前长度大于原本长度，更换原本长度，记录满足条件的值，对应在数组中的 index

```
            if (max > dp[i]) {
```

```
                dp[i] = max
                nextIndex = j
            }
```

返回最长递增子数组的下标

res[i].push(...res[nextIndex])dp.reduce((prev, cur, i, arr) => cur > arr[prev] ? i : prev, dp.length - 1)

返回最长递增子序列

React

1. 双层循环递增思想找到每一个需要移动的节点：第一层遍历新列表，记录元素位置，第二层遍历旧列表，记录元素位置，如果元素相等就找到了节点，如果旧节点下标小于当前节点下标就需要移动节点，否则可以复用节点。

```
let lastIndex = 0
    for (let i = 0; i < nextList.length; i++) {
        let nextItem = nextList[i];
        for (let j = 0; j < prevList.length; j++) {
            let prevItem = prevList[j]
            if (nextItem === prevItem) {
                if (j < lastIndex) {
                    // 需要移动节点
```

2.移动节点：将其命名为 a，将 a 的 DOM 节点移动到 a 在新列表中前一个 VNode 对应的真实 DOM 的结点之后。因为我们已经遍历过新列表，该节点之前的所有节点都是排好序的，如果该节点需要移动，那么只需要将 DOM 节点移动到前一个 vnode 节点之后就可以，因为 a 在新列表中的位置就是这样的。

3.添加节点：在遍历新列表的时候给每一个节点加上一个标识 `find = false`，如果在旧列表中找到了这个节点，将其变为 `find = true`，进行 patchNode，如果没有找到 `let refNode = i <= 0 ? prevChildren[0].el : nextChildren[i - 1].el.nextSibling` 判断将其插入到队首还是队列中间，然后进行节点挂载。

4.移除节点：遍历旧列表并记录每个元素的 key，如果在新列表中没有这个元素，就删除掉这个节点。

```
for (let i = 0; i < prevChildren.length; i++) {
        let prevChild = prevChildren[i],
            key = prevChild.key,
            has = nextChildren.find(item => item.key === key);
        if (!has) parent.removeChild(prevChild.el)
    }
```

5.优化与不足：目前的 `reactDiff` 的时间复杂度为 `O(m*n)`，我们可以用空间换时间，把 `key` 与 `index` 的关系维护成一个 `Map`，从而将时间复杂度降低为 `O(n)`