

浙江大学

本科生实验报告

课程名称 :	B/S体系软件设计
姓 名 :	张潇予
学 院 :	计算机科学与技术
系 :	计算机科学与技术
专 业 :	计算机科学与技术
学 号 :	3180105574
指导教师 :	胡晓军

2021 年 6 月 28 日

系统概述

eIoT System是一个基于B/S体系结构的物联网应用管理系统，提供用户友好的网页界面。用户可通过注册和登录进入系统，添加、配置、删除物联网终端设备，查看统计数据。

eIoT System的前端编写借助Umi框架，数据流方案使用DvaJS，UI组件库使用antd及antd-pro。

后端用node.js语言编写，数据库使用非关系型数据库MongoDB。为了简化MQTT数据包的处理，还使用了Mosquito服务器作为消息中间件。

您可以访问eIoT System的[GitHub仓库](#)查看源代码。

前端设计

路由控制

路由控制是前端最重要的部分之一，当用户在不同的页面之间跳转时，前端需要显示不同的内容。但出于效率和用户体验的考虑，我们并不希望每次用户点击时都加载一个全新的HTML文件。路由控制可以解决这一问题，事先配置路径和组件的对应关系，用户做出切换页面的操作时，路径改变，浏览器渲染新组建，达成切换页面的效果。

Umi框架提供了比原生HTML更方便的路由控制方法，在config/routes.js文件中按照指定格式编写URL和组件的对应关系，就可以自动渲染出侧边菜单栏。在组件中如果想进行页面跳转，可以从Umi库中引用history，利用history.push()和history.replace()等方法改变当前路径。

eIoT System的路由分为三组：登录页面、主页面、出错页面。下面一一进行分析：

1. 登录页面

登录页面的目标url为/user/login，路由分为/user和/user/login两层，第一层渲染登陆界面的背景组件，第二层渲染操作框。

```
{  
  path: '/user',  
  component: '../layouts/UserLayout',  
  routes: [  
    {  
      name: 'login',  
      path: '/user/login',  
      component: './User/login',  
    },  
  ],  
},
```

2. 主页面

主页面包含四个目标url，分别对应首页、数据管理、数据地图和设备配置。如果直接访问根目录，会重定向到首页(/home)。值得注意的是，在四个目标url的外部包裹了两个组件SecurityLayout和BasicLayout，前者用于用户鉴权，如果鉴权不通过会重定向到登录界面；后者是页头、页脚和侧边栏菜单，它们对于几个界面是一样的。

```
{  
  path: '/',  
  redirect: '/home',  
},  
{  
  path: '/home',  
  name: '首页',  
  icon: 'barChart',  
  component: './Home',  
},  
{  
  path: '/data',  
  name: '数据管理',  
  icon: 'dashboard',  
  component: './DataList',  
},  
{  
  name: '数据地图',  
  icon: 'global',  
  path: '/map',  
  component: './Map',  
},  
{  
  name: '设备配置',  
  icon: 'setting',  
  path: '/devices',  
  component: './DeviceConfig',  
},  
{  
  component: './404',  
},  
],}
```

3. 出错页面

如果url与任何已经定义的url都不匹配，就会跳转到出错页。

```
{  
  component: './404',  
},
```

数据流控制

前端提供的可视化、数据筛选等服务需要数据支持。数据可以通过HTTP请求从后端获取，但前端仍需解决两个问题：后端获取的数据存放在哪里；组件需要数据时从哪里获取。Dva是一种很流行的数据流方案，eIoT System是用了Umi中集成的DvaJS。

Dva的实现需要model、service和component三者的配合。其中model尤为重要，可以把它拆解为四个部分：
namespace：命名空间，其他部分可以通过此命名访问到它；state：一个对象，可以储存各种数据类型，state中储存的也就是此model需要维护的数据内容；effects：一组异步函数，通常包含调用service获取后端数据，因为异步操作使得同样的输入可能获得不同的输出，effects函数被独立出来。通常在获取到数据后会调用reducers改变state；reducers：用于改变state值的一组函数。reducer接受两个参数：之前的state和当前传入的值，两者经过累加后产生新的state。

service中包含一系列函数，它们的作用是接收参数、向后端发送HTTP请求、返回后端response。

其他路由组件如果想使用model中的state，需要使用connect()方法，绑定对应的model，也可以只绑定model中的某些属性。除了state中声明的属性，组件还可以绑定一个特殊的loading属性，这个属性记录了某个model或者某个effect函数是否正在执行异步操作（如等待后端回复），这个属性在提升用户体验上很有用，我们可以在loading时显示加载动画。

另外一个重要的函数是dispatch()，它用于触发action函数，进一步触发effects，或reducers，从而改变state。一般在页面首次刷新时候会调用dispatch，我在程序中使用hook函数useEffect()，以在组件挂载前调用dispatch，以数据管理组件为例，代码如下：

```
useEffect(() => {  
  if (dispatch) {  
    dispatch({  
      type: 'record/fetch',  
    });  
  }  
}, [ ]);
```

eIoT System的前端使用了5个model，下面一一介绍。

1. global

■ state

```
state: {  
  collapsed: false  
}
```

`collapsed`记录主菜单状态（收起或展开）。

- reducers

```
changeLayoutCollapsed(  
  state = {  
    collapsed: true,  
  },  
  { payload },  
) {  
  return { ...state, collapsed: payload };  
}
```

改变`collapsed`状态，点击菜单收起/展开按钮会调用此函数。

2. login

- state

```
state: {  
  status: undefined,  
},
```

`status`记录登录状态。

- effects

```
*login({ payload }, { call, put }) {  
  const response = yield call(accountLogin, payload);  
  yield put({  
    type: 'user/saveCurrentUser',  
    payload: response.currentUser,  
  })  
  yield put({  
    type: 'changeLoginStatus',  
    payload: response,  
  });  
  
  if (response.status === 'ok') {  
    ...  
  
    if (redirect) {  
      ...  
      if (redirectUrlParams.origin === urlParams.origin) {  
        ...  
      } else {  
        ...  
      }  
    }  
  
    history.replace(redirect || '/');  
  }  
},
```

向后端发送用户名及密码，后端验证后发送回登录状态及其他信息，调用reducers将信息储存在login和user两个model中。最后根据跳转到登录界面前的路径设置新路由。

```
logout() {
  const { redirect } = getPageQuery();

  if (window.location.pathname !== '/user/login' && !redirect) {
    history.replace({
      pathname: '/user/login',
      search: stringify({
        redirect: window.location.href,
      }),
    });
  }
}
```

退出函数。清除登录状态并跳转到登录页面。

- reducers

```
changeLoginStatus(state, { payload }) {
  return { ...state, status: payload.status };
}
```

修改登录状态。

3. user

- state

```
state: {
  currentUser: {},
  subDevices: []
},
```

currentUser储存当前用户的各种信息；

subDevices储存当前用户订阅的设备ID列表。

- effects

```
*register({ payload }, { call, _ }) {
  const response = yield call(registerUser, payload);
  if (response.status === 'ok') {
    message.success('🎉🎉🎉 注册成功! ');
  }
  if (response.status === 'nameDup') {
    message.error('用户名重复! ');
  }
  ...
},
```

向后端发送注册信息，根据回复显示提示信息。

- reducers

```
saveCurrentUser(state, action) {
  return {
    ...state,
    currentUser: { ...action.payload },
  }
},
```

修改currentUer。

4. device

- state

```
state: {
  total: 0,
  online: 0,
  devices: [],
}
```

total储存设备总量； online储存在线设备量； devices储存所有设备信息；

- effects

```
*fetch(_, { call, put }) {
  ...
},
```

从后端获取设备信息。

- reducers

```
saveDevices(state, action) {
  const { online, data: devices } = action.payload

  ...

  devices.filter((item) => {
    ...
  });

  ...

  Object.keys(tmpData).forEach((key) => {
    newData.push({
      x: key,
      y: tmpData[key],
    })
  });

  return {
    ...state,
```

```
        total: devices.length,
        online: online,
        devices: devices,
        newData: newData,
        newTotal: newTotal
    }
}
```

计算近10日新增设备总数和每日新增数，修改state。

5. record

■ state

```
state: {
  records: [],
}
```

records储存所有设备发送的消息。

■ effects

```
*fetch(_, { call, put }) {
  const response = yield call(queryRecords);
  yield put({
    type: 'saveRecords',
    payload: response,
  });
}
```

从后端获取设备发送的消息。

■ reducers

```
saveRecords(state, action) {
  const { data } = action.payload
  const records = data.filter(function(item){ // 去除空消息
    return item.alert !== undefined
  });
  return {
    ...state,
    records: records
  },
},
```

去除空消息，修改state。

组件实现

eIoT System由5个路由页面组成：登录界面和四个功能界面，每个页面中可能包含若干子组件，我将在此节中详细介绍。



图：登陆界面



图：四个功能界面对应的侧边栏

1. User: 登录界面

使用antd提供的Tabs实现切换效果。User组件中维护一个type state，值为account或register，对应登录和注册。点击Tabs触发type改变，页面根据type值选择渲染登录或注册表单。



使用antd-pro提供的ProForm和ProFormText，用rules属性对输入字段进行约束，例如密码是必填项：

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

邮箱地址必须含有@，@前的部分由字母、数字、下划线等字符组成，@后必须有.，必须以两个及以上的数字和字母组合结尾：

```
{
  pattern: /^( [A-Za-z0-9_\\-\\.])+@[ [A-Za-z0-9_\\-\\.])+\\.( [A-Za-z]{2,4})$/,
  message: "不合法的邮箱地址!"
}
```

点击提交出发提交事件，在提交事件中使用dispatch触发登录/注册动作。

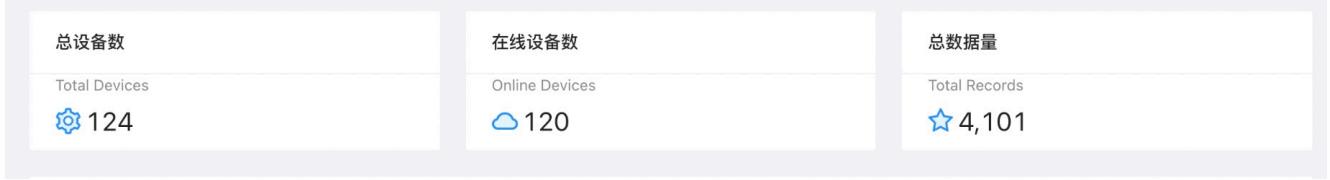
```
const handleSubmit = (values) => {
  const { dispatch } = props;
  if (type === 'account') {
    dispatch({
      type: 'login/login',
      payload: { ...values, type },
    });
  } else { // register
    dispatch({
      type: 'user/register',
      payload: { ...values, type },
    });
  }
};
```

2. Home: 首页

首页由三个子组件构成，布局如下：

```
<div>
  <IntroduceRow>
  ...
  </IntroduceRow>
  <DeviceOverview>
  ...
  </DeviceOverview>
  <RecordOverview>
  ...
  </RecordOverview>
</div>
```

■ IntroduceRow



此组件使用antd提供的Card，三个Card用Row、Col结合的方式布局，每个Card显示不同的数据。

■ DeviceOverview

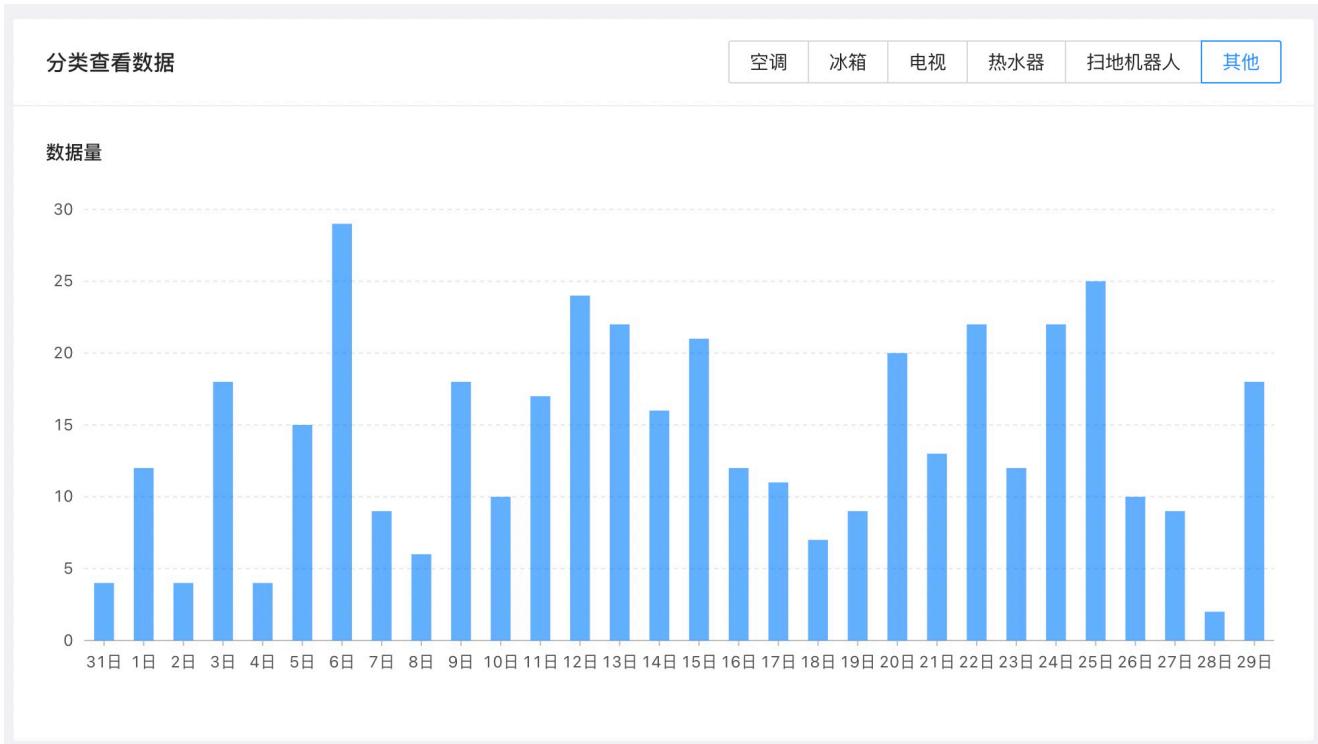


与IntroduceRow类似，此组件也是在Row和Col中包裹了两个Card。在线率进度条和近10日新增曲线图使用antd-pro提供的MiniProgress和MiniArea。两者使用的数据都来自于父组件User，User则通过连接model获取相应数据。

```
<ChartCard
  title="在线率"
  total={onlineRate.toString() + "%"}
  contentHeight={46}>
  <MiniProgress percent={onlineRate} strokeWidth={8} target={onlineRate} />
</ChartCard>

...
<ChartCard
  title="近10日新增"
  total={<Statistic value={newTotal} />}
  contentHeight={46}>
  <MiniArea height={45} line={true} data={newData} />
</ChartCard>
```

▪ RecordOverview



此组件外部使用Card，通过extra参数配置选项条，点击更改组件state，显示不同数据。柱状图使用antd-pro提供的Bar，输入数据是近三十天设备发送的数据量。日期的计算使用moment库。

3. DataList: 数据管理界面

设备ID:		请输入	上报时间:		请选择	重置	查询
查询表格							
设备ID	数值	状态	上报时间	经度	纬度	消息	
device0010	12	● 正常	2021-06-17	30.42	120.47	Device Data 2021/06/17 20:54:37	
device0010	25	● 正常	2021-06-17	30.17	120.30	Device Data 2021/06/17 20:54:38	
device0003	73	● 正常	2021-06-17	30.47	120.46	Device Data 2021/06/17 20:55:08	
device0007	87	● 告警	2021-06-17	30.32	120.27	Device Data 2021/06/17 20:55:08	
device0002	93	● 告警	2021-06-17	30.46	120.02	Device Data 2021/06/17 20:55:09	
device0002	42	● 正常	2021-06-17	30.28	120.23	Device Data 2021/06/17 20:55:09	
device0003	24	● 正常	2021-06-17	30.27	119.98	Device Data 2021/06/17 20:55:09	
device0010	31	● 正常	2021-06-17	30.40	120.24	Device Data 2021/06/17 20:55:11	
device0002	28	● 正常	2021-06-17	30.46	120.22	Device Data 2021/06/17 20:55:11	
device0003	59	● 正常	2021-06-17	30.35	119.99	Device Data 2021/06/17 20:55:12	

第 1-10 条/总共 483 条 < 1 2 3 4 5 ... 49 > 10 条/页

数据管理界面主要使用antd-pro提供的ProTable，ProTable封装了搜索、筛选、排序、设置每页展示条数等功能，需要提供两个重要的参数：columns、request。columns声明表头内容（即设备ID、数值等），列表中每一项包含属性标题、ID、数据类型、约束、是否可以查找、是否可以筛选等等，ProTable根据这些信息渲染表头和显示数据。以经度一栏为例：

```
{
    title: '经度',
    search: false,
    dataIndex: 'lng',
    valueType: 'textarea',
    renderText: (value) => value.toFixed(2),
},
```

数据类型为文本，index为'lat'，表示将来把输入ProTable的数据中index为'lat'的数据显示在这一列。renderText函数将原始的经度数据格式化为两位小数。

ProTable中最为核心的参数是request函数，这一函数接收params、sorter、filter三个参数，三个参数中包含了用户搜索的字段、选择排序的属性、选择筛选的属性等，request函数根据参数对数据进行处理，返回一个对象，此对象中至少包含两个Key：success和data。success标志获取数据是否成功，data列表中则包含将要显示的数据。

request核心逻辑如下：

- 筛选出用户订阅的设备发送的信息

```
const deviceList = currentUser.devices
data = data.filter((item) => {
    return deviceList.indexOf(item.clientId) !== -1
});
```

- 处理搜索

```
const searchKeys = Object.keys(params).slice(2)
searchKeys.forEach((key) => {
    const searchVal = params[key]
    data = data.filter((item) => {
        if (key === 'clientId') return item[key].indexOf(searchVal) !== -1
        else {
            return moment(item[key]).format('YYYY-MM-DD') === searchVal
        }
    });
})
```

- 处理排序

```
if (sorter) {
    data = data.sort((prev, next) => {
        Object.keys(sorter).forEach((key) => {
            if (sorter[key] === 'descend') {
                ...
            } else if (sorter[key] === 'ascend') {
                ...
            }
        });
        return sortNumber;
    });
}
```

■ 处理筛选

```
if (filter) {
  if (Object.keys(filter).length > 0) {
    data = data.filter((item) => {
      return Object.keys(filter).some((key) => {
        ...
      });
    });
  }
}
```

4. Map: 数据地图界面

首页 / 数据地图

数据地图



地图组件使用antv L7封装的mapbox，数据的标注使用Marker组件。为Marker传入经纬度，作为MapScene的子组件传入，即可在地图上显示标注。

■ MapboxScene参数设置

因为上传的数据集中在杭州市西湖区附近，调整地图初始中心位置和缩放比例，让用户进入页面时得到一个比较舒适的视觉体验。

```
<MapboxScene
  map={{
    center: [120.14002669582967, 30.245842227935793],
    pitch: 0,
    style: 'normal',
    zoom: 11,
  }}>
```

```

    style={{
      position: 'relative',
      width: '100%',
      height: '520px',
    }}
    option={{
      logoVisible: false
    }}
  >

```

■ Marker生成

为每一条记录生成一个Marker组件，为避免记录太多造成卡顿，最多显示近100条数据。遍历记录生成标记，根据是否告警设置标记颜色。

```

data.map((item) => {
  count += 1;
  let color = item.alert === 1 ? '#9a325e' : '#6790f2';
  if (count <= 100) {
    markers.push(<Marker option={{ color: color }} lnglat={[item.lng, item.lat]} />);
  }
})

```

5. DeviceConfig: 设备配置界面

设备配置

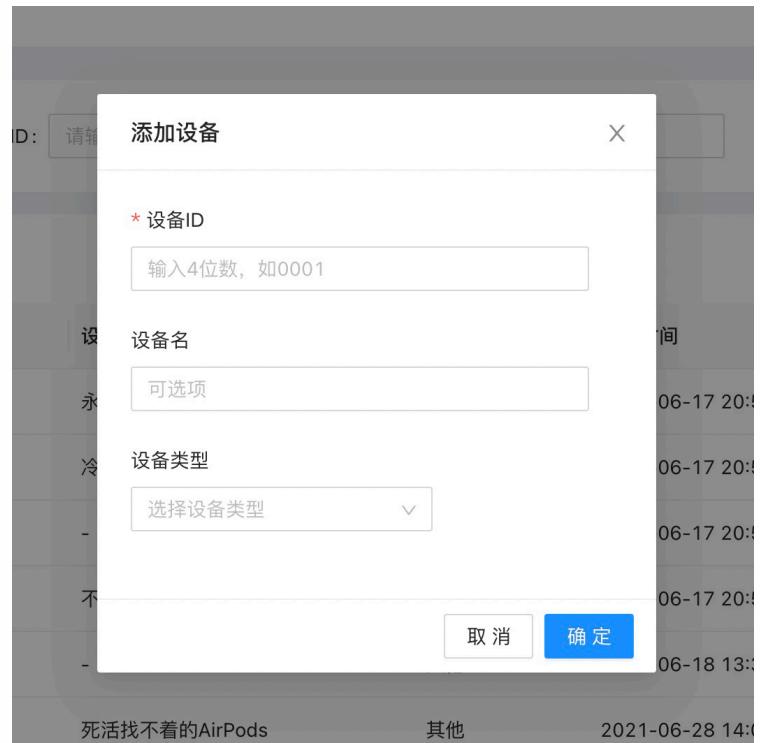
The screenshot shows the 'Device Configuration' interface. At the top, there are two input fields: '设备ID:' and '设备名:', both with placeholder text '请输入'. To the right of these are '重置' and '查询' buttons. Below this is a '查询表格' (Query Table) section. It contains a header row with columns: '设备ID', '设备名', '类型', '激活时间', and '操作'. Below the header is a table with 8 rows of data. Each row includes a '操作' column with '配置' (Configure) and '删除' (Delete) buttons. The table data is as follows:

设备ID	设备名	类型	激活时间	操作
device0010	永远在工作的扫地机器人	扫地机器人	2021-06-17 20:54:37	配置 删除
device0003	冷漠热水器	热水器	2021-06-17 20:55:08	配置 删除
device0007	-	其他	2021-06-17 20:55:08	配置 删除
device0002	不想被定义的水族箱	其他	2021-06-17 20:55:09	配置 删除
device0022	-	其他	2021-06-18 13:39:14	配置 删除
device0100	死活找不着的AirPods	其他	2021-06-28 14:00:39	配置 删除
device0120	-	其他	2021-06-28 14:00:43	配置 删除

At the bottom of the table area, there is a pagination bar with the text '第 1-7 条/总共 7 条' and a page number '1'.

设备配置界面与数据管理界面类似，同样使用了ProTable，增加了添加设备和配置/删除设备逻辑。

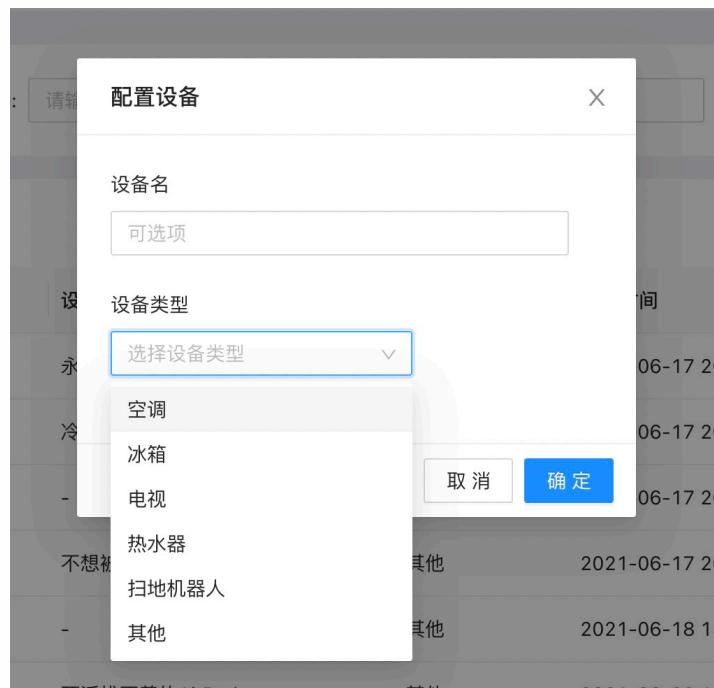
■ 添加设备



添加设备界面使用antd-pro提供的ModalForm，设备ID为必填项，输入后前端会先检查此设备是否为已订阅设备，如果是，则弹出错误提示，否则向后端发送添加请求。

点击添加设备按钮弹出ModalForm效果用state实现。维护一个`createModalVisible`状态，将它作为ModalForm组件的`visible`参数输入，点击添加设备后`createModalVisible`变为true，ModalForm随即显示出来。

■ 配置设备



实现方式与添加设备类似，点击确定后发送`dispatch`触发后端更新。

- 删除设备



为防止误操作，点击删除后弹出确认对话框，用户可选择取消或确认。确认对话框使用antd提供的Modal.confirm。

为了实现添加/配置/删除设备后立即刷新的效果，在处理这些事件的最后，都使用dispatch获取新数据，用户无需刷新页面即可看到最新消息。

后端设计

MQTT服务器

- Mosquitto

MQTT是基于发布/订阅范式的消息协议，工作在TCP/IP协议族上，是为硬件性能低下的远程设备及带宽有限的网络状况设计的消息协议，需要一个消息中间件。

eIoT System后端连接轻量级的Mosquitto，通过简单的配置即可完成通信。

1. 配置Mosquitto监听端口

```
# =====
# Listeners
# =====

listener 8883/127.0.0.1
allow_anonymous true
```

2. 配置终端设备属性，连接到8883端口

```
1 devices=120
2 server=tcp://localhost:8883
3 topic=testapp
4 prefix=device
```

3. 打开服务器

```
[→ ~ brew services start mosquitto
==> Successfully started `mosquitto` (label: homebrew.mxcl.mosquitto)
```

- Node.js后端

1. 使用mqtt库，创建连接。

```

var mqtt = require('mqtt')
var json = require('json5')
var opt = {
  port: 8883
}

var client = mqtt.connect('tcp://127.0.0.1', opt)

```

2. 连接Mosquitto，订阅两个topic。其中'testapp'是终端设备推送数据的topic，'\$SYS/broker/clients/connected'是Mosquitto的系统topic，在连接数发生变化时发布消息，消息内容为新的连接数量。

```

client.on('connect', function () {
  client.subscribe('testapp');
  client.subscribe('$SYS/broker/clients/connected');
});

```

3. 监听消息，处理消息。

```

client.on('message', async function (topic, msg) {
  if (topic === 'testapp') {
    const message = json.parse(msg)
    const { clientId } = message
    const device = await db.deviceModel.findOne({ clientId });
    if (!device) { // 如果设备不存在，注册该设备
      const { timestamp } = message
      await db.deviceModel({ clientId, timestamp }).save();
    } else if (!device.timestamp) { // 如果设备未激活，激活该设备
      const { timestamp } = message
      await db.deviceModel.findOneAndUpdate(
        { clientId },
        {
          $set: { timestamp }
        }
      )
    }
    await db.dataModel({ ...message }).save();
  }
  else { // 在线连接数
    let onlineNum = json.parse(msg) > 0 ? json.parse(msg) - 1 : 0;
    module.exports.onlineNum = onlineNum
  }
});

```

如果是发布终端推送的消息，则对设备和数据数据库进行增加或修改操作；如果是连接数消息，改变全局变量onlineNum的值。

数据库服务器

- MongoDB

开启MongoDB服务器。

```
[→ target brew services start mongodb-community@4.2  
==> Successfully started `mongodb-community@4.2` (label: homebrew.mxcl.mongodb-c]
```

- Node.js后端

1. 连接MongoDB

使用mongoose库，数据库名为eIoT。

```
const mongoose = require('mongoose');  
const db = require('./schema');  
  
mongoose.set('useFindAndModify', false);  
mongoose.set('useCreateIndex', true);  
mongoose.connect('mongodb://localhost/eIoT', {useNewUrlParser: true,  
useUnifiedTopology: true});  
const conn = mongoose.connection;  
conn.on('error', console.error.bind(console, 'connection error'));  
conn.once('open', () => {  
    console.log("connected");  
});  
  
module.exports = db;
```

2. 设计数据库schema

MongoDB的每一个数据库下有若干collections，类似于SQL中的表。后端为每个collection建立一个schema.js文件。

- userSchema

储存用户信息，包括用户名、邮箱等字段，另外保存了一个列表，储存用户订阅的设备ID。

```
const userSchema = new Schema({  
    name: {type: String, unique: true},  
    password: {type: String},  
    email: {type: String, unique: true},  
    phone: {type: String, unique: true},  
  
    devices: [{type: String}]  
});
```

- deviceSchema

储存设备信息，包括设备ID、激活时间的时间戳等字段，其中设备类型是一个枚举类型，默认值为5，即其他。

```
const deviceSchema = new Schema({
  clientId: {type: String, unique: true},
  timestamp: {type: Number},
  deviceType: {type: String, enum: [0, 1, 2, 3, 4, 5], default: 5},
  deviceName: {type: String},
});
```

■ dataSchema

储存设备上报的消息，包括是否告警、设备ID等信息。

```
const dataSchema = new Schema({
  alert: {type: Number},
  clientId: {type: String},
  info: {type: String},
  lat: {type: Number},
  lng: {type: Number},
  timestamp: {type: Number},
  value: {type: Number},
})
```

3. 操作数据库

对数据库的增删改查操作应在Model上进行，先对每个schema生成一个Model：

```
exports.userModel = mongoose.model('user', userSchema);
exports.deviceModel = mongoose.model('device', deviceSchema);
exports.dataModel = mongoose.model('data', dataSchema);
```

引用Model即可在其他模块操作数据库，常用API有save()，find()，findAndUpdate()等。

HTTP响应

eIoT System后端使用express框架，定义/users, /datas, devices三个路由：

```
const express = require('express');

const datasRouter = require('./routes/datas');
const usersRouter = require('./routes/users');
const devicesRouter = require('./routes/devices');

const app = express();

app.use('/users', usersRouter);
app.use('/datas', datasRouter);
app.use('/devices', devicesRouter);
```

路由中提供不同的回调函数以处理请求。

1. /users

- /login

从请求中解构出用户名和密码，在userModel中寻找该用户名对应的用户，如果找到，比对数据库中储存的密码和前端提供的密码，如果用户名不存在或密码错误，回复'error'；否则回复对应的用户记录。

```
router.post('/login', async (req, res) => {
  const { password, name } = req.body;
  const user = await db.userModel.findOne({ name });
  if (!user || user.password !== password) {
    return res.json({
      status: 'error',
    })
  } else {
    return res.json({
      status: 'ok',
      currentUser: user,
    });
  }
});
```

- /register

从请求中解构出用户名、邮箱和手机号，在数据库中查找它们是否已经注册过，如果是，返回对应的错误信息，否则向useModel插入一条新记录。

```
router.post('/register', async (req, res) => {
  const { body } = req;
  const { name, email, phone } = body;
  if (await db.userModel.findOne({ name })) { return res.json({ status: 'nameDup' }) }
  if (await db.userModel.findOne({ email })) { return res.json({ status: 'emailDup' }) }
  if (await db.userModel.findOne({ phone })) { return res.json({ status: 'phoneDup' }) }
  await db.userModel({ ...body }).save();
  return res.json({ status: 'ok' });
});
```

2. /datas

- /fetch

从dataModel中获取所有消息数据，返回给前端。

```
router.post('/fetch', async function (req, res, next) {
  const records = await db.dataModel.find({}, { _id: 0, __v: 0 })
  return res.json({
    data: records,
  })
});
```

3. /devices

- /fetch

从deviceModel中获取所有设备数据，返回给前端。

```
router.post('/fetch', async function (req, res, next) {  
  const devices = await db.deviceModel.find({}, { _id: 0, __v: 0 })  
  let onlineNum = require('../bin/www');  
  return res.json({  
    online: onlineNum.onlineNum,  
    data: devices,  
  })  
});
```

- /add

指定用户添加一个设备。从请求中解构出用户名，从userModel中找到该用户对应的记录，并在该用户的devices(订阅设备)字段中压入新的设备ID。

由于用户添加的设备可能没有被激活过，对这种情况，需要向deviceModel中插入一条新数据。

```
router.post('/add', async function (req, res, next) {  
  const { deviceName, deviceType, userName } = req.body;  
  
  await db.userModel.findOneAndUpdate(  
    { name: userName },  
    {  
      $push: { devices: [clientId] }  
    }  
  )  
  
  const device = await db.deviceModel.findOne({ clientId });  
  if (!device) {  
    if (deviceName) await db.deviceModel({ clientId, deviceType, deviceName }).save();  
    else await db.deviceModel({ clientId, deviceType }).save();  
  }  
  
  return res.json({  
    success: true  
  })  
});
```

- /update

用户更新设备信息。设备的设备名和设备类型可以被修改。从请求中解构出设备ID、设备名和类型，找到数据库中对应记录进行更新。

```
router.post('/update', async function (req, res, next) {  
  const { clientId, deviceName, deviceType } = req.body
```

```

if (deviceName) {
    await db.deviceModel.findOneAndUpdate(
        { clientId: clientId },
        { $set: { deviceName: deviceName } }
    )
}
device = await db.deviceModel.findOne({ clientId });
if (deviceType !== undefined) {
    await db.deviceModel.findOneAndUpdate(
        { clientId },
        { $set: { deviceType } }
    )
}

return res.json({
    success: true
})
);

```

- /remove

用户取消订阅设备，注意并非从deviceModel中将设备删除，而是将设备从用户的devices列表中移除。

```

router.post('/remove', async function (req, res, next) {
    const { clientId, userName } = req.body

    const user = await db.userModel.findOne({ name: userName });
    const index = user.devices.indexOf(clientId)
    if (index !== -1) user.devices.splice(index, 1)

    await db.userModel.findOneAndUpdate(
        { name: userName },
        { $set: { devices: user.devices } }
    )

    return res.json({
        success: true
    })
);

```

实验结果

演示视频附在当前文件夹中，在此不再赘述。