

项目实战02

项目实战02

作业要求

课堂目标

资源

知识点

dva

dva特性

dva数据流向

理解dva

切换 history 为 browserHistory

dynamic

ExamplePage

umi是什么

why umi

什么时候不用 umi?

为什么不是?

create-react-app

next.js

Umi+Dva基本使用

安装

目录结构

src/.umi

src/app.ts

路由

配置路由

约定式路由

动态路由

可选的动态路由

嵌套路由

全局 layout

不同的全局 layout

404 路由

扩展路由属性

在页面间跳转

声明式

命令式

使用按需加载

按需加载组件

按需加载非组件

实例

回顾

作业

下节课内容

作业要求

2. **暗号必须写对**，以课上讲师或者助教打出来的文字为准，注意不要有错别字。
3. **作业必须按照格式提交**，如React阶段必须提交截图!!! 注意作业格式命名为：姓名-日期-课节名称(本节课主题名称)。

4. 不按照以上要求，以后质检员不给你过，**不返钱!!!**

课堂目标

1. 掌握企业级应用框架 - umi
2. 掌握数据流方案 - dva

资源

1. [umi](#)
2. [dva](#)
3. [Why dva and what's dva](#)
4. [Antd Pro](#)

知识点

dva

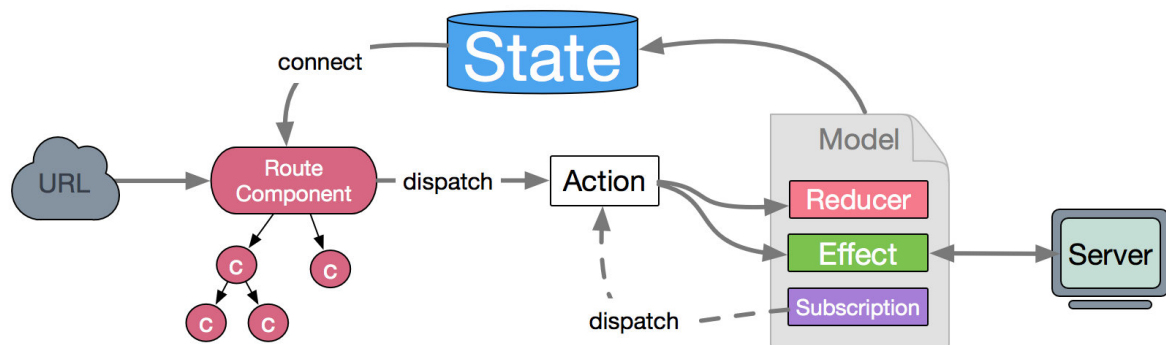
dva 首先是一个基于 [redux](#) 和 [redux-saga](#) 的数据流方案，然后为了简化开发体验，dva 还额外内置了 [react-router](#) 和 [fetch](#)，所以也可以理解为一个轻量级的应用框架。

dva特性

- **易学易用**，仅有 6 个 api，对 redux 用户尤其友好，[配合 umi 使用](#)后更是降低为 0 API
- **elm 概念**，通过 reducers, effects 和 subscriptions 组织 model
- **插件机制**，比如 [dva-loading](#) 可以自动处理 loading 状态，不用一遍遍地写 showLoading 和 hideLoading
- **支持 HMR（模块热替换）**，基于 [babel-plugin-dva-hmr](#) 实现 components、routes 和 models 的 HMR。

dva数据流向

数据的改变发生通常是通过用户交互行为或者浏览器行为（如路由跳转等）触发的，当此类行为会改变数据的时候可以通过 `dispatch` 发起一个 action，如果是同步行为会直接通过 `Reducers` 改变 `State`，如果是异步行为（副作用）会先触发 `Effects` 然后流向 `Reducers` 最终改变 `State`，所以在 dva 中，数据流向非常清晰简明，并且思路基本跟开源社区保持一致（也是来自于开源社区）。



理解dva

软件分层：回顾react，为了让数据流更易于维护，我们分成了store， reducer， action等模块，各司其职，软件开发也是一样

1. Page 负责与用户直接打交道：渲染页面、接受用户的操作输入，侧重于展示型交互性逻辑。
2. Model 负责处理业务逻辑，为 Page 做数据、状态的读写、变换、暂存等。
3. Service 负责与 HTTP 接口对接，进行纯粹的数据读写。

DVA 是基于 redux、redux-saga 和 react-router 的轻量级前端框架及最佳实践沉淀，核心api如下：

1. model

- state 状态
- action
- dispatch
- reducer
- effect 副作用，处理异步

2. subscriptions 订阅

3. router 路由

1. `namespace`：model 的命名空间，只能用字符串。一个大型应用可能包含多个 model，通过 `namespace` 区分
2. `reducers`：用于修改 `state`，由 `action` 触发。reducer 是一个纯函数，它接受当前的 `state` 及一个 `action` 对象。action 对象里面可以包含数据体 (payload) 作为入参，需要返回一个新的 `state`。
3. `effects`：用于处理异步操作（例如：与服务端交互）和业务逻辑，也是由 `action` 触发。但是，它不可以修改 `state`，要通过触发 `action` 调用 `reducer` 实现对 `state` 的间接操作。
4. `action`：是 `reducers` 及 `effects` 的触发器，一般是一个对象，形如 `{ type: 'add', payload: todo }`，通过 `type` 属性可以匹配到具体某个 `reducer` 或者 `effect`，`payload` 属性则是数据体，用于传送给 `reducer` 或 `effect`。

切换 history 为 browserHistory

先安装 history 依赖，

```
$ npm install --save history
```

然后修改入口文件，

```
const createHistory = require("history").createBrowserHistory;

const app = dva({
  history: createHistory()
});
```

目前版本中错误处理：

```
✖ Warning: Please use `require("history").createHashHistory` instead of `index.js:2177`
`require("history/createHashHistory")`. Support for the latter will be removed in the
next major release.
```

打开node_modules/dva/lib/index.js:

找到

```
var _createHashHistory =
  _interopRequireDefault(require("history/createHashHistory"));
```

改成:

```
var _createHashHistory =
  _interopRequireDefault(require("history").createHashHistory);
```

github对应issue: <https://github.com/dvajs/dva/issues/2115>

dynamic

dynamic/index.js

```
import dynamic from "dva/dynamic";
import app from "../app";

export const UserPageDynamic = dynamic({
  app,
  models: () => [import("../models/user")],
  component: () => import("../routes/UserPage")
});
```

ExamplePage

```
import React, { Component } from "react";
import { connect } from "dva";
import { Button, Table } from "antd";
import { Link } from "dva/router";
import { routerRedux } from "dva/router";
import styles from "../ExamplePage.less";

const columns = [
  {
    title: "姓名",
```

```

      dataIndex: "name",
      key: "name"
    },
    {
      title: "年龄",
      dataIndex: "age",
      key: "age"
    },
    {
      title: "住址",
      dataIndex: "city",
      key: "city"
    }
  ];

  @connect(
    state => ({ state, example: state.example }),
    dispatch => {
      return {
        dispatch,
        getProductData: payload =>
          dispatch({ type: "example/getProductData", payload })
      };
    }
  )
  class ExamplePage extends Component {
    dataSearch = () => {
      // 异步获取数据
      this.props.getProductData();
    };

    render() {
      console.log("ExamplePage props", this.props); //sy-log
      const { example, dispatch } = this.props;
      const { data } = example;
      return (
        <div className={styles.example}>
          <h3 className={styles.title}>ExamplePage</h3>
          <button onClick={this.dataSearch}>search</button>
          <Table columns={columns} dataSource={data} rowKey="id" />
          <Link to="/user">go userPage</Link>
          <Link to="/">首页</Link>
          <Button
            onClick={() => {
              dispatch(routerRedux.push("/"));
            }}
          >
            go index
          </Button>
        </div>
      );
    }
  }
}
export default ExamplePage;







```

umi是什么

umi，中文可发音为乌米，是一个可插拔的企业级 react 应用框架。

why umi

它主要具备以下功能：

-  **可扩展**，Umi 实现了完整的生命周期，并使其插件化，Umi 内部功能也全由插件完成。此外还支持插件和插件集，以满足功能和垂直域的分层需求。
-  **开箱即用**，Umi 内置了路由、构建、部署、测试等，仅需一个依赖即可上手开发。并且还提供针对 React 的集成插件集，内涵丰富的功能，可满足日常 80% 的开发需求。
-  **企业级**，经蚂蚁内部 3000+ 项目以及阿里、优酷、网易、飞猪、口碑等公司项目的验证，值得信赖。
-  **大量自研**，包含微前端、组件打包、文档工具、请求库、hooks 库、数据流等，满足日常项目的周边需求。
-  **完备路由**，同时支持配置式路由和约定式路由，同时保持功能的完备性，比如动态路由、嵌套路由、权限路由等等。
-  **面向未来**，在满足需求的同时，我们也不会停止对新技术的探索。比如 dll 提速、modern mode、webpack@5、自动化 external、bundler less 等等。

什么时候不用 umi?

如果你，

- 需要支持 IE 8 或更低版本的浏览器
- 需要支持 React 16.8.0 以下的 React
- 需要跑在 Node 10 以下的环境中
- 有很强的 webpack 自定义需求和主观意愿
- 需要选择不同的路由方案

Umi 可能不适合你。

为什么不是?

[create-react-app](#)

create-react-app 是基于 webpack 的打包层方案，包含 build、dev、lint 等，他在打包层把体验做到了极致，但是不包含路由，不是框架，也不支持配置。所以，如果大家想基于他修改部分配置，或者希望在打包层之外也做技术收敛时，就会遇到困难。

[next.js](#)

next.js 是个很好的选择，Umi 很多功能是参考 next.js 做的。要说有哪些地方不如 Umi，我觉得可能是不够贴近业务，不够接地气。比如 antd、dva 的深度整合，比如国际化、权限、数据流、配置式路由、补丁方案、自动化 external 方面等等一线开发者才会遇到的问题。

Umi+Dva基本使用

安装

环境要求：node版本>=10.13

开课吧web全栈架构师

```
新建一个空文件夹: mkdir lesson6-umi
进入文件夹: cd lesson6-umi
创建: yarn create @umijs/umi-app
安装依赖: yarn
启动: yarn start
```

目录结构

```
├─ package.json
├─ .umirc.ts    配置文件, 包含 umi 内置功能和插件的配置。
├─ .env        环境变量
├─ dist        执行 umi build 后, 产物默认会存放在这里。
├─ mock        存储 mock 文件, 此目录下所有 js 和 ts 文件会被解析为 mock 文件。
├─ public      此目录下所有文件会被 copy 到输出路径。
├─ src
│   └─ .umi
│   └─ layouts/index.tsx 约定式路由时的全局布局文件。
│   └─ pages             所有路由组件存放在这里。
│       └─ index.less
│       └─ index.tsx
└─ app.ts
```

src/.umi

临时文件目录, 比如入口文件、路由等, 都会被临时生成到这里。**不要提交 .umi 目录到 git 仓库, 他们会在 umi dev 和 umi build 时被删除并重新生成。**

src/app.ts

运行时配置文件, 可以在这里扩展运行时的能力, 比如修改路由、修改 render 方法等。

路由

手动创建或者使用下面的命令。

建立pages下面的单页面about:

```
umi g page about
```

建立文件夹more(默认是js和css):

```
umi g page more/index --typescript --less
```

访问index: <http://localhost:8000/>

访问about: <http://localhost:8000/about>

配置路由

路由配置详细查看官方文档: <https://umijs.org/zh-CN/docs/routing>

约定式路由

动态路由

```
umi g page product/[id]
```

```
import React from 'react';
import { IRouteComponentProps } from 'umi';
import styles from './[id].less';

export default (props: IRouteComponentProps) => {
  console.log('product', props); //sy-log
  return (
    <div>
      <h1 className={styles.title}>Page product/[id]</h1>
    </div>
  );
};
```

路由配置

```
{ path: '/product/:id', component: '@/pages/product/[id]' },
```

可选的动态路由

umi3暂不支持，下面是umi2的使用。

umi 里约定动态路由如果带 \$ 后缀，则为可选动态路由。

```
umi g page product/'$id$'
```

比如以下结构：

```
export default function({ location, match }) {
  const { id } = match.params;
  return (
    <div className={styles.normal}>
      <h1>Page $id$</h1>
      <p>{id || '没有id'}</p>
    </div>
  );
}
```

路由配置：

```
{
  path: '/channel/:id?',
  component: './channel/$id$',
},
```

嵌套路由

Umi 里约定目录下有 `_layout.tsx` 时会生成嵌套路由，以 `_layout.tsx` 为该目录的 layout。layout 文件需要返回一个 React 组件，并通过 `props.children` 渲染子组件。

首先创建 `_layout.js`

```
umi g page product/_layout
```

```
import React from 'react';
import { IRouteComponentProps } from 'umi';

export default (props: IRouteComponentProps) => {
  console.log('product', props); //sy-log
  return (
    <div style={{ color: 'red' }}>
      <h1>layout</h1>
      {props.children}
    </div>
  );
};
```

配置路由：

```
{
  path: '/product/:id',
  component: '@/pages/product/_layout',
  routes: [{ path: '/product/:id', component: '@/pages/product/[id]' }],
},
```

layout

Page product/[id]-123

全局 layout

约定 `src/layouts/index.tsx` 为全局路由。返回一个 React 组件，并通过 `props.children` 渲染子组件。比如：

```
import * as React from 'react';
import { IRouteComponentProps } from 'umi';

export default function Layout({ children }: IRouteComponentProps) {
  return (
    <div style={{ color: 'orange' }}>
      <h1>全局layout</h1>
      {children}
    </div>
  );
}
```

路由配置:

```
import { defineConfig } from 'umi';

export default defineConfig({
  nodeModulesTransform: {
    type: 'none',
  },
  routes: [
    {
      path: '/',
      component: '@/layout/index',
      routes: [
        { path: '/', component: '@/pages/index' },
        { path: '/about', component: '@/pages/about' },
        { path: '/more', component: '@/pages/more/index' },
        // { path: '/product/:id', component: '@/pages/product/[id]' },
        {
          path: '/product/:id',
          component: '@/pages/product/_layout',
          routes: [{ path: '/product/:id', component: '@/pages/product/[id]' }],
        },
      ],
    },
  ],
});
```

不同的全局 layout

你可能需要针对不同路由输出不同的全局 layout，Umi 不支持这样的配置，但你仍可以在 `src/layouts/index.tsx` 中对 `location.path` 做区分，渲染不同的 layout。

比如想要针对 `/login` 输出简单布局，

```
export default function(props) {
  if (props.location.pathname === '/login') {
    return <SimpleLayout>{ props.children }</SimpleLayout>
  }

  return (
    <>
      <Header />
      { props.children }
      <Footer />
    </>
  );
}
```

404 路由

约定 `src/pages/404.tsx` 为 404 页面，需返回 React 组件。

```
umi g page 404/index --typescript --less
```

```
{ component: '@pages/404' },
```

扩展路由属性

支持在代码层通过导出静态属性的方式扩展路由。

比如：

```
function HomePage() {
  return <h1>Home Page</h1>;
}

HomePage.title = 'Home Page';

export default HomePage;
```

其中的 `title` 会附加到路由配置中。

在页面间跳转

在 umi 里，页面之间跳转有两种方式：声明式和命令式。

声明式

通过 Link 使用，通常作为 React 组件使用。

```
import { Link } from 'umi';

export default () => (
  <Link to="/list">Go to list page</Link>
);
```

命令式

通过 history 使用，通常在事件处理中被调用。

```
import { history } from 'umi';

function goToListPage() {
  history.push('/list');
}
```

也可以直接从组件的属性中取得 history

```
export default (props) => (
  <Button onClick={()=>props.history.push('/list')}>Go to list page</Button>
);
```

更多命令式的跳转方法，详见 [api#history](#)。

使用按需加载

按需加载组件

通过 Umi 的 `dynamic` 接口实现，比如：

```
import { dynamic } from 'umi';

const delay = (timeout) => new Promise(resolve => setTimeout(resolve, timeout));

const App = dynamic({
  loader: async function() {
    await delay(/* 1s */1000);
    return () => <div>I will render after 1s</div>;
  },
});
```

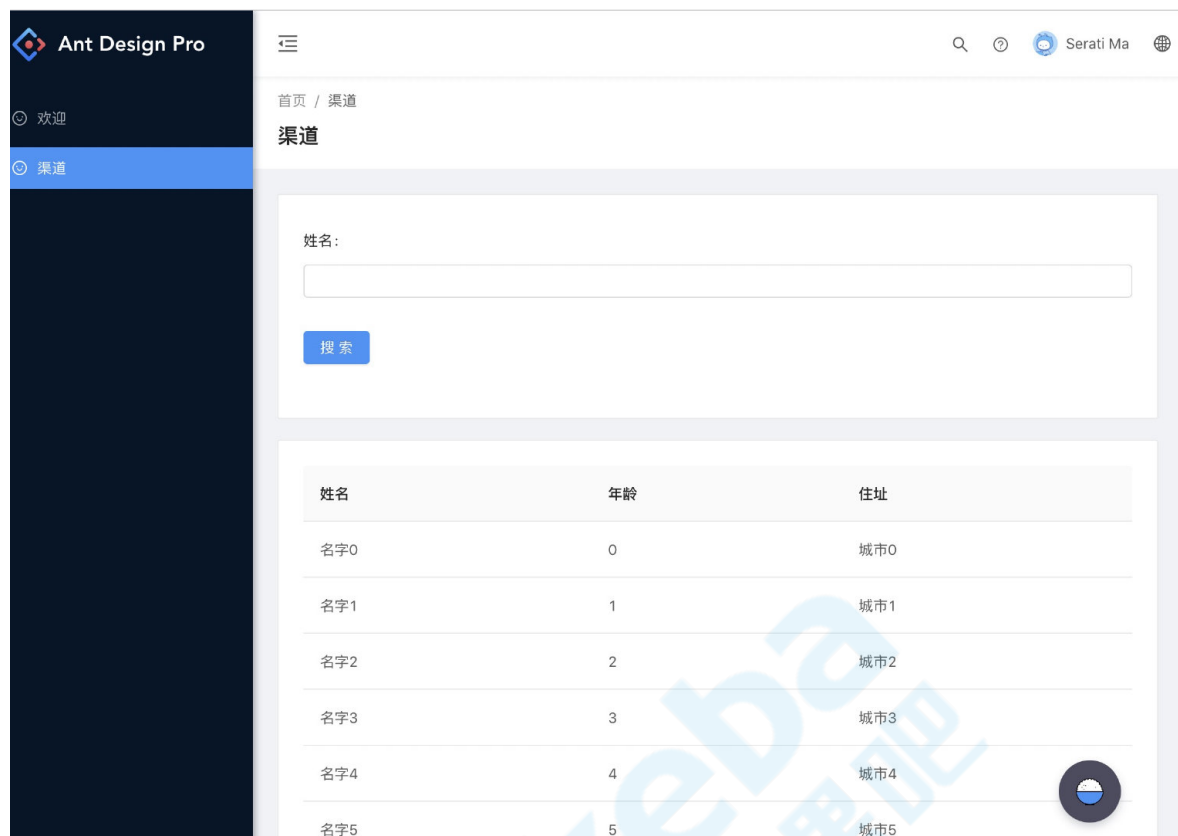
按需加载非组件

通过 `import()` 实现，比如：

```
import('g2').then(() => {
  // do something with g2
});
```

实例

实现如下图：



使用状态：state + connect

- 创建页面more.js: `umi g page more/index --less`

```
import React from 'react';
import { PageHeaderWrapper } from '@ant-design/pro-layout';
import { Form, Input, Button, Card, Table } from 'antd';
import { connect } from 'umi';
import styles from './index.less';
```

```
const columns = [
  {
    title: '姓名',
    dataIndex: 'name',
    key: 'name',
  },
  {
    title: '年龄',
    dataIndex: 'age',
    key: 'age',
  },
  {
    title: '住址',
    dataIndex: 'city',
    key: 'city',
  },
];
```

// UI层和数据层分开

```
class More extends React.Component {
```

开课吧web全栈架构师

```

constructor(props) {
  super(props);
  this.state = {};
}

componentDidMount() {
  this.props.getMoreData({ name: '' });
}

// 成功才会执行这个函数
onFinish = values => {
  console.log('values', values); // sy-log
  // this.props.getMoreDataBySearch(values);
  this.props.getMoreData(values);
};

// 失败才会执行这个函数
onFinishFailed = err => {
  console.log('err', err); // sy-log
};

render() {
  const { data } = this.props.more;
  return (
    <PageHeaderWrapper className={styles.more}>
      <Card>
        <Form onFinish={this.onFinish} onFinishFailed={this.onFinishFailed}>
          <Form.Item
            label="姓名"
            name="name"
            rules={[{ required: true, message: '请输入姓名' }]}
          >
            <Input placeholder="请输入姓名" />
          </Form.Item>
          <Form.Item>
            <Button type="primary" htmlType="submit">
              查询
            </Button>
          </Form.Item>
        </Form>
      </Card>

      <Card>
        <Table columns={columns} dataSource={data} rowKey="id" />
      </Card>
    </PageHeaderWrapper>
  );
}

export default connect(
  // mapStateToProps
  ({ more }) => ({ more }),
  // mapDispatchToProps
  {
    getMoreData: values => ({ type: 'more/getMoreData', payload: values }),
    // getMoreDataBySearch: values => ({
    //   type: 'more/getMoreDataBySearch',
  }

```

```

    // payload: values,
    // }),
  },
)(More);

```

- 更新模型src/models/more.js

```

import { getChannelData, getChannelDataBySearch } from '@services/more';

const model = {
  namespace: 'more',
  state: { data: [] },
  effects: {
    *getMoreData(action, { call, put }) {
      const res = yield call(getChannelData, action.payload);
      // 更新数据
      yield put({ type: 'moreData', payload: res });
    },
    // *getMoreDataBySearch(action, { call, put }) {
    //   const res = yield call(getChannelDataBySearch, action.payload);
    //   // 更新数据
    //   yield put({ type: 'moreData', payload: res });
    // },
  },
  reducers: {
    moreData(state, { payload }) {
      // 返回一个新的state
      return {
        ...state,
        data: payload.data,
      };
    },
  },
};

export default model;

```

- 添加服务: src/service/more.js

```

import request from '@utils/request';

export async function getChannelData(params) {
  return request('/api/getChannelData', {
    data: params,
    method: 'post',
  });
}

// export async function getChannelDataBySearch(params) {
//   return request('/api/getChannelDataBySearch', {
//     method: 'post',
//     data: params,
//   });
// }

```

数据mock: 模拟数据接口

mock目录和src平级，新建mock/channel.js

```
const channelTableData = [];  
for (let i = 0; i < 10; i++) {  
  channelTableData.push({  
    id: i,  
    name: `名字${i}`,  
    age: i,  
    city: `城市${i}`,  
  });  
}  
function searchChannelData(name) {  
  const res = [];  
  for (let i = 0; i < 10; i++) {  
    if (channelTableData[i].name.indexOf(name) > -1) {  
      res.push(channelTableData[i]);  
    }  
  }  
  return res;  
}  
export default {  
  // 支持值为 Object 和 Array  
  // 'GET /api/getChannelData': {  
  //   // 查询表单数据  
  //   data: [...channelTableData],  
  // },  
  'POST /api/getChannelData': (req, res) => {  
    // 搜索  
    console.log('asas', res.body); //sy-log  
    const data = res.body  
      ? [...channelTableData]  
      : searchChannelData(req.body.name);  
    res.send({  
      status: 'ok',  
      data: data, //searchChannelData(req.body.name),  
    });  
  },  
};
```

回顾

项目实战02

[作业要求](#)

[课堂目标](#)

[资源](#)

[知识点](#)

[dva](#)

[dva特性](#)

[dva数据流向](#)

[理解dva](#)

[切换 history 为 browserHistory](#)

[dynamic](#)

- ExamplePage
- umi是什么
 - why umi
 - 什么时候不用 umi?
- 为什么不是?
 - create-react-app
 - next.js
- Umi+Dva基本使用
 - 安装
 - 目录结构
 - src/.umi
 - src/app.ts
 - 路由
 - 配置路由
 - 约定式路由
 - 动态路由
 - 可选的动态路由
 - 嵌套路由
 - 全局 layout
 - 不同的全局 layout
 - 404 路由
 - 扩展路由属性
 - 在页面间跳转
 - 声明式
 - 命令式
 - 使用按需加载
 - 按需加载组件
 - 按需加载非组件
- 实例
- 回顾
- 作业
- 下节课内容

作业

1. 使用pro table实现图片，不再把数据存在model state。

下节课内容

React源码第一节，实现createElement、render、Component，实现class组件、函数组件、原生节点、文本节点、Fragment渲染。