项目实战02

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. Antd Pro

知识点

umi

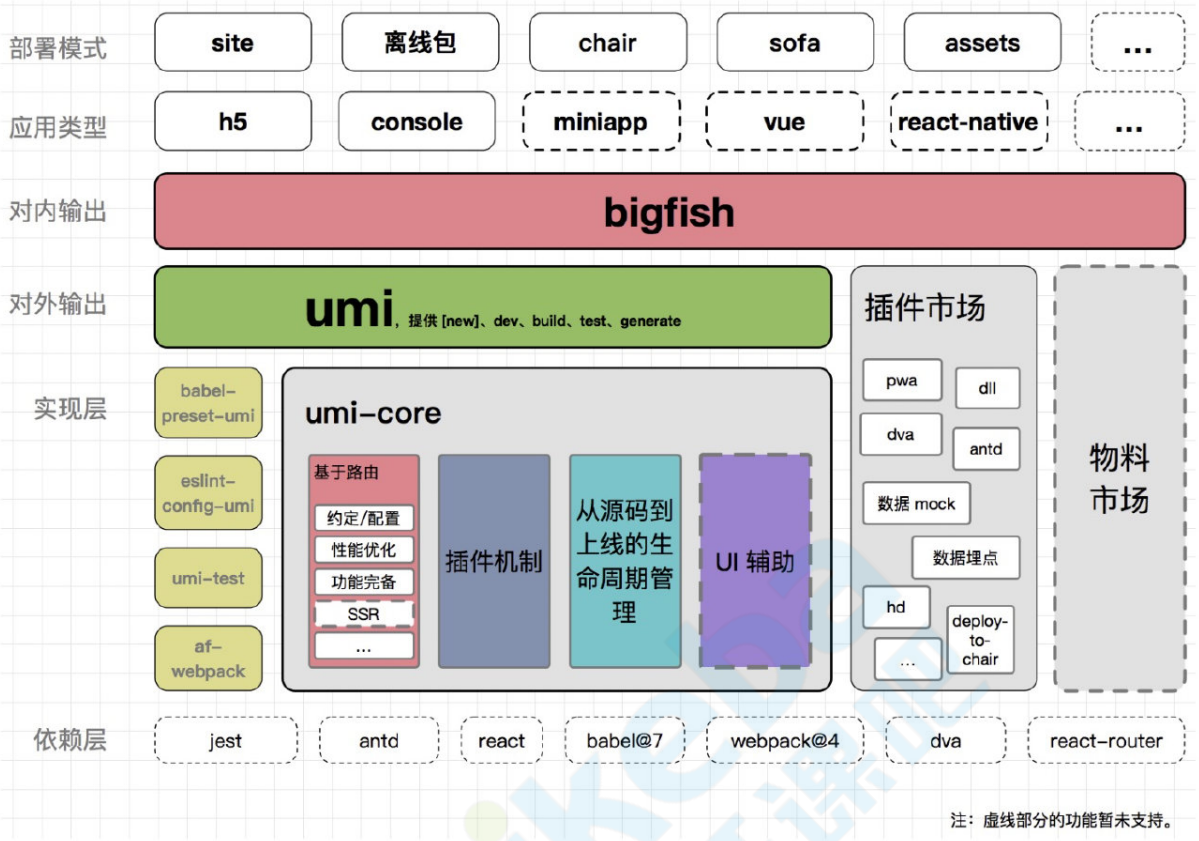
umi，中文可发音为乌米，是一个可插拔的企业级 react 应用框架。

why umi

- 📦 **开箱即用**，内置 react、react-router 等
- 🏈 **类 next.js 且功能完备的路由约定**，同时支持配置的路由方式
- 🎉 **完善的插件体系**，覆盖从源码到构建产物的每个生命周期
- 🚀 **高性能**，通过插件支持 PWA、以路由为单元的 code splitting 等
- 💈 **支持静态页面导出**，适配各种环境，比如中台业务、无线业务、egg、支付宝钱包、云凤蝶等
- 🚄 **开发启动快**，支持一键开启 dll 和 hard-source-webpack-plugin 等
- 🐠 **一键兼容到 IE9**，基于 umi-plugin-polyfills
- 🍁 **完善的 TypeScript 支持**，包括 d.ts 定义和 umi test
- 🌴 **与 dva 数据流的深入融合**，支持 duck directory、model 的自动加载、code splitting 等等

架构

下图是 umi 的架构图。



它和 dva、roadhog 是什么关系?

- roadhog 是基于 webpack 的封装工具，目的是简化 webpack 的配置
- umi 可以简单地理解为 roadhog + 路由，思路类似 next.js/nuxt.js，辅以一套插件机制，目的是通过框架的方式简化 React 开发
- dva 目前是纯粹的数据流，和 umi 以及 roadhog 之间并没有相互的依赖关系，可以分开使用也可以一起使用，个人觉得 umi + dva 是比较搭的

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。

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。

Use umi with dva

自`>= umi@2`起，`dva`的整合可以直接通过 umi-plugin-react 来配置。

特性

- **按目录约定注册 model**，无需手动 `app.model`
- **文件名即 namespace**，可以省去 model 导出的 `namespace` key
- **无需手写 router.js**，交给 umi 处理，支持 model 和 component 的按需加载
- **内置 query-string 处理**，无需再手动解码和编码
- **内置 dva-loading 和 dva-immer**，其中 dva-immer 需通过配置开启

- **开箱即用**，无需安装额外依赖，比如 dva、dva-loading、dva-immer、path-to-regexp、object-assign、react、react-dom 等

dva+umi 的约定

- src 源码
  - pages页面
  - components 组件
  - layout布局
- model
- config 配置
- mock 数据模拟
- test测试等

```
.
├── dist/                          // 默认的 build 输出目录
├── mock/                          // mock 文件所在目录，基于 express
├── config/
    ├── config.js                  // umi 配置，同 .umirc.js，二选一
└── src/                           // 源码目录，可选
    ├── layouts/index.js           // 全局布局
    ├── pages/                     // 页面目录，里面的文件即路由
        ├── .umi/                  // dev 临时目录，需添加到 .gitignore
        ├── .umi-production/       // build 临时目录，会自动删除
        ├── document.ejs           // HTML 模板
        ├── 404.js                 // 404 页面
        ├── page1.js               // 页面 1，任意命名，导出 react 组件
        ├── page1.test.js          // 用例文件，umi test 会匹配所有 .test.js 和 .e2e.js 结尾
        └── page2.js               // 页面 2，任意命名
    ├── global.css                 // 约定的全局样式文件，自动引入，也可以用 global.less
    ├── global.js                  // 可以在这里加入 polyfill
    ├── app.js                     // 运行时配置文件
├── .umirc.js                      // umi 配置，同 config/config.js，二选一
├── .env                           // 环境变量
└── package.json
```

安装

环境要求： node版本>=8.10

antd-pro安装：
新建立一个空文件夹：mkdir lesson6-umi
进入文件夹：cd lesson6-umi
创建：yarn create umi
选择ant-design-pro
选择Javascript
安装依赖： yarn
启动：yarn start或者umi dev

其他例子： 如umi-antd-mobile等

Umi基本使用

建立pages下面的单页面about：

```
umi g page about
```

建立文件夹more(默认是css)：

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

import router from 'umi/router' 跳转
router.push('/user/2')

起服务看效果

```
umi dev
```

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

路由

umi 会根据 `pages` 目录自动生成路由配置。

约定式路由

基础路由

动态路由

`umi g page product/'$id'`

```
import styles from './$id.less';

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

config

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

可选的动态路由

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>
  );
}
```

config

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

嵌套路由

umi 里约定目录下有 `_layout.js` 时会生成嵌套路由，以 `_layout.js` 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。

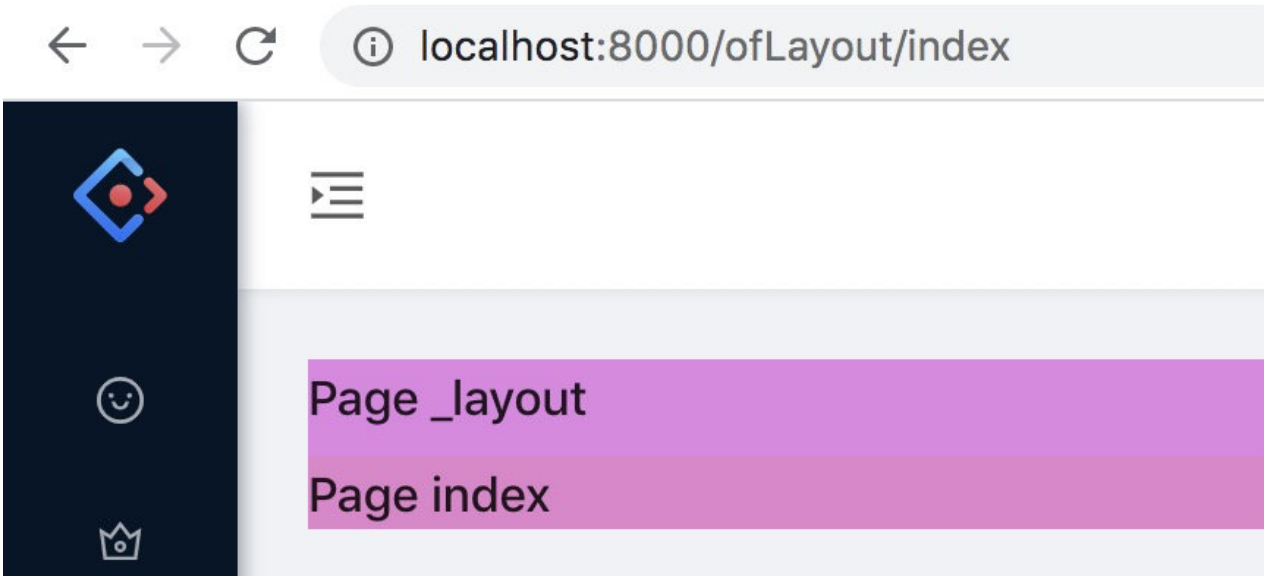
首先创建_layout.js

```
umi g page ofLayout/_layout
```

```
// 创建父组件 umi g page ofLayout/_layout
export default function(props) {
  return (
    <div>
      <h1>Page _layout</h1>
      <div>{props.children}</div>
    </div>
  )
}
```

创建兄弟组件 `umi g page ofLayout/index`

```
export default function() {
  return (
    <div className={styles.normal}>
      <h1>Page index</h1>
    </div>
  );
}
```



全局 layout

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

比如：

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

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 `layouts/index.js` 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 />
    </>
  );
}
```

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

基于 `umi/link`，通常作为 React 组件使用。

```
import Link from 'umi/link';

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

命令式

基于 `umi/router`，通常在事件处理中被调用。

```
import router from 'umi/router';

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

更多命令式的跳转方法，详见 api#umi/router。

启用 Hash 路由

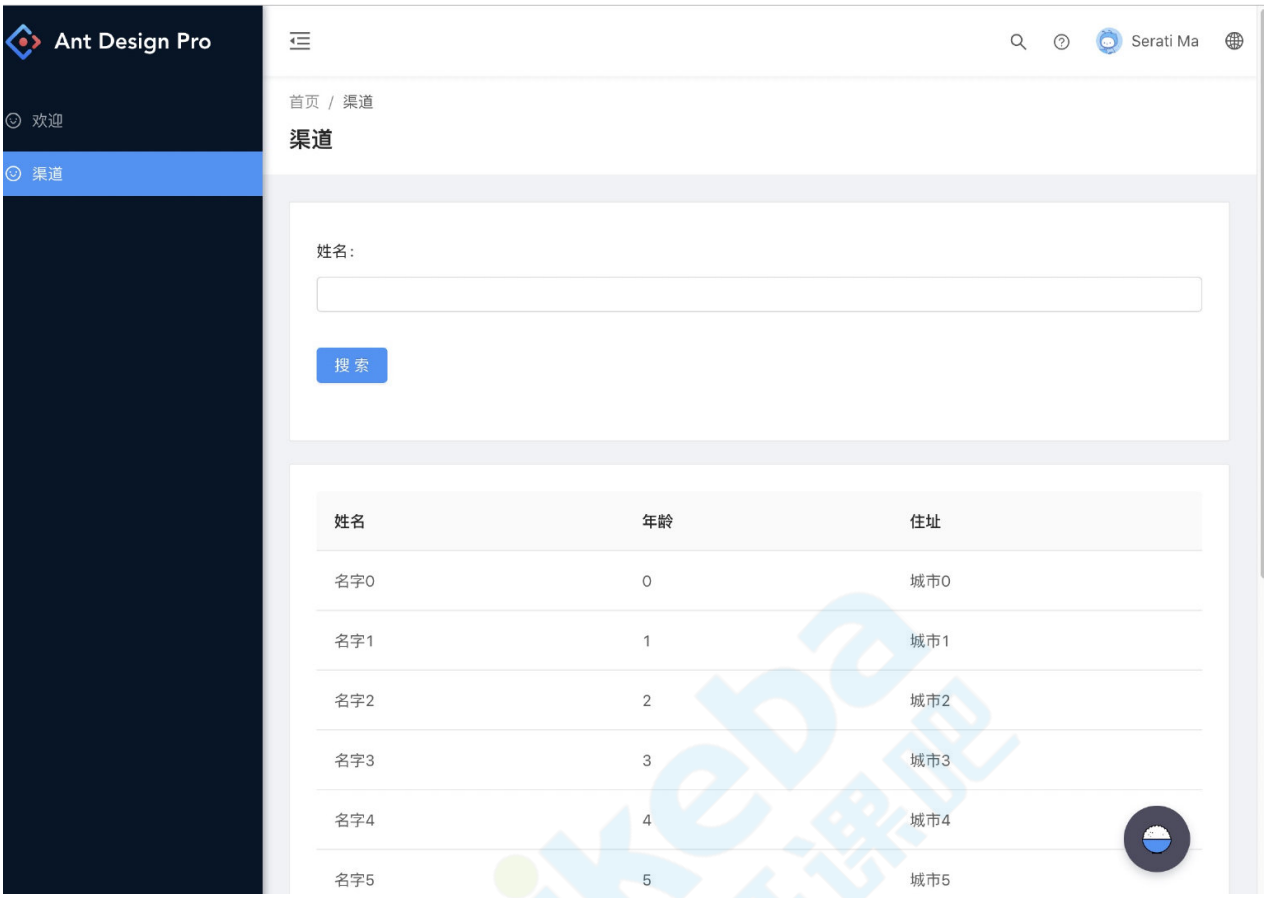
umi 默认是用的 Browser History，如果要用 Hash History，需配置：

```
export default {
  history: 'hash',
}
```

更多配置

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, { Component } from 'react';
import { PageHeaderWrapper } from '@ant-
design/pro-layout';
import { Card, Form, Input, Button, Table }
from 'antd';
import { connect } from 'dva';
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',
  },
];

export default connect(({ more }) => ({ more
}), {
  getMoreData: () => ({ type:
'more/getChannelData' }),
  getMoreDataBySearch: search => ({ type:
'more/getChannelDataBySearch', payload: search
}),
})(
  class More extends Component {
    componentDidMount() {
      this.props.getMoreData();
    }
```

```
    onFinish = values => {
      console.log('values', values); // sy-log
      this.props.getMoreDataBySearch(values);
    };

    onFinishFailed = err => {
      console.log('err', err); // sy-log
    };

    render() {
      const { data } = this.props.more;
      console.log('oo', this.props); // sy-log
      return (
        <PageHeaderWrapper className=
{styles.more}>
          <Card>
            <Form onFinish={this.onFinish}
onFinishFailed={this.onFinishFailed}>
              <Form.Item
                label="姓名"
                name="name"
                rules={[{ required: true,
message: '请输入姓名查询' }]}
              >
                <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 dataSource={data} columns=
{columns} rowKey="id" />
          </Card>
        </PageHeaderWrapper>
      );
    }
  },
);
```

- 更新模型src/models/more.js

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

const model = {
  namespace: 'more',
  state: {
    data: [],
  },
  effects: {
```

```
    *getChannelData({ payload }, { call, put })
{
      const response = yield
call(getChannelData, payload);
      yield put({
        type: 'channelData',
        payload: response,
      });
    },
    *getChannelDataBySearch({ payload }, {
call, put }) {
      const response = yield
call(getChannelDataBySearch, payload);
      console.log('has', response, payload);

      yield put({
        type: 'channelData',
        payload: response,
      });
    },
  },
  reducers: {
    channelData(state, { payload }) {
      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', {
    method: 'get',
  });
}
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/getChannelDataBySearch': (req,
res) => {//搜索
    res.send({
      status: 'ok',
      data: searchChannelData(req.body.name),
    });
  },
};
```

回顾

- **项目实战02**
  - 课堂目标
  - 资源
  - 知识点
    - umi
      - why umi
      - 它和 dva、roadhog 是什么关系?
    - dva
      - dva特性
      - 理解dva
    - Use umi with dva
      - 特性
      - dva+umi 的约定
      - 安装
    - Umi基本使用
      - 路由
        - 约定式路由
          - 基础路由
          - 动态路由
          - 可选的动态路由
          - 嵌套路由
          - 全局 layout
          - 不同的全局 layout
      - 在页面间跳转
        - 声明式

作业

1. 做项目，比如说实现这个effects函数合并，功能多多益善。
2. umi、dva源码尽自己能力去看。