Table of Contents

dva 文档

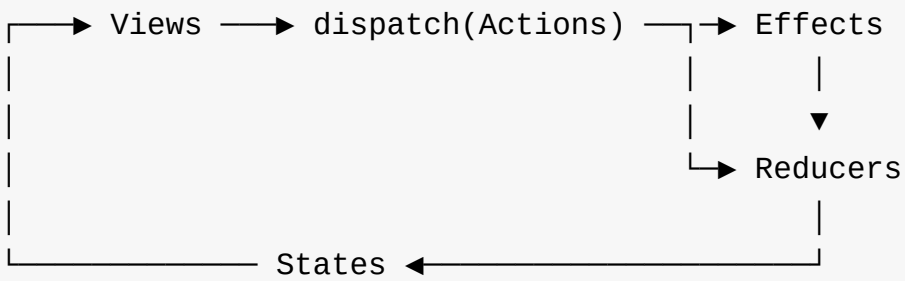
来自 https://github.com/dvajs/dva-docs

基本概念

View this in English

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）会先出发 `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）。



Models

State

```
type State = any
```

State 表示 Model 的状态数据，通常表现为一个 javascript 对象（当然它可以是任何值）；操作的时候每次都要当作不可变数据（immutable data）来对待，保证每次都是全新对象，没有引用关系，这样才能保证 State 的独立性，便于测试和追踪变化。

在 dva 中你可以通过 dva 的实例属性 `_store` 看到顶部的 state 数据，但是通常你很少会用到:

```
const app = dva();
console.log(app._store); // 顶部的 state 数据
```

Action

`type AsyncAction = any`

Action 是一个普通 javascript 对象，它是改变 State 的唯一途径。无论是从 UI 事件、网络回调，还是 WebSocket 等数据源所获得的数据，最终都会通过 dispatch 函数调用一个 action，从而改变对应的数据。action 必须带有 `type` 属性指明具体的行为，其它字段可以自定义，如果要发起一个 action 需要使用 `dispatch` 函数；需要注意的是 `dispatch` 是在组件 connect Models以后，通过 props 传入的。

```
dispatch({
  type: 'add',
});
```

dispatch 函数

`type dispatch = (a: Action) => Action`

dispatching function 是一个用于触发 action 的函数，action 是改变 State 的唯一途径，但是它只描述了一个行为，而 dipatch 可以看作是触发这个行为的方式，而 Reducer 则是描述如何改变数据的。

在 dva 中，connect Model 的组件通过 props 可以访问到 dispatch，可以调用 Model 中的 Reducer 或者 Effects，常见的形式如：

```
dispatch({
  type: 'user/add', // 如果在 model 外调用，需要添加 namespace
  payload: {}, // 需要传递的信息
});
```

Reducer

```
type Reducer<S, A> = (state: S, action: A) => S
```

Reducer（也称为 reducing function）函数接受两个参数：之前已经累积运算的结果和当前要被累积的值，返回的是一个新的累积结果。该函数把一个集合归并成一个单值。

Reducer 的概念来自于是函数式编程，很多语言中都有 reduce API。如在 javascript 中：

```
[{x:1},{y:2},{z:3}].reduce(function(prev, next){
    return Object.assign(prev, next);
})
//return {x:1, y:2, z:3}
```

在 dva 中，reducers 聚合积累的结果是当前 model 的 state 对象。通过 actions 中传入的值，与当前 reducers 中的值进行运算获得新的值（也就是新的 state）。需要注意的是 Reducer 必须是纯函数，所以同样的输入必然得到同样的输出，它们不应该产生任何副作用。并且，每次一的计算都应该使用immutable data，这种特性简单理解就是每次操作都是返回一个全新的数据（独立，纯净），所以热重载和时间旅行这些功能才能够使用。

Effect

Effect 被称为副作用，在我们的应用中，最常见的就是异步操作。它来自于函数编程的概念，之所以叫副作用是因为它使得我们的函数变得不纯，同样的输入不一定获得同样的输出。

dva 为了控制副作用的操作，底层引入了redux-sagas做异步流程控制，由于采用了generator的相关概念，所以将异步转成同步写法，从而将effects转为纯函数。至于为什么我们这么纠结于 纯函数，如果你想了解更多可以阅读Mostly adequate guide to FP，或者它的中文译本JS函数式编程指南。

Subscription

Subscriptions 是一种从 源 获取数据的方法，它来自于 elm。

Subscription 语义是订阅，用于订阅一个数据源，然后根据条件 dispatch 需要的 action。数据源可以是当前的时间、服务器的 websocket 连接、keyboard 输入、geolocation 变化、history 路由变化等等。

```
import key from 'keymaster';
...
app.model({
  namespace: 'count',
  subscriptions: {
    keyEvent(dispatch) {
      key('⌘+up, ctrl+up', () => { dispatch({type:'add'}) });
    },
  }
});
```

Router

这里的路由通常指的是前端路由，由于我们的应用现在通常是单页应用，所以需要前端代码来控制路由逻辑，通过浏览器提供的 History API 可以监听浏览器url的变化，从而控制路由相关操作。

dva 实例提供了 router 方法来控制路由，使用的是react-router。

```
import { Router, Route } from 'dva/router';
app.router(({history}) =>
  <Router history={history}>
    <Route path="/" component={HomePage} />
  </Router>
);
```

Route Components

在组件设计方法中，我们提到过 Container Components，在 dva 中我们通常将其约束为 Route Components，因为在 dva 中我们通常以页面纬度来设计 Container Components。

所以在 dva 中，通常需要 connect Model的组件都是 Route Components，组织在 `/routes/` 目录下，而 `/components/` 目录下则是纯组件（Presentational Components）。

参考引申

- redux docs
- redux docs 中文
- Mostly adequate guide to FP
- JS函数式编程指南
- choo docs
- elm

快速上手

View this in English

> 本章节会引导开发者快速搭建 dva 项目，并熟悉他的所有概念。

最终效果：

这是一个测试鼠标点击速度的 App，记录 1 秒内用户能最多点几次。顶部的 Highest Record 纪录最高速度；中间的是当前速度，给予即时反馈，让用户更有参与感；下方是供点击的按钮。

看到这个需求，我们可能会想：

1. 该如何创建应用?
2. 创建完后，该如何一步步组织代码?
3. 开发完后，该如何构建、部署和发布?

在代码组织部分，可能会想：

1. 如何写 Component ?
2. 如何写样式?
3. 如何写 Model ?
4. 如何 connect Model 和 Component ?
5. 用户操作后，如何更新数据到 State ?
6. 如何处理异步逻辑? (点击之后 +1，然后延迟一秒 -1)
7. 如何处理路由?

以及：

1. 不想每次刷新 Highest Record 清 0，想通过 localStorage 记录，这样刷新之后还能保留 Highest Record。该如何处理?
2. 希望同时支持键盘的点击测速，又该如何处理?

我们可以带着这些问题来看这篇文章，但不必担心有多复杂，因为全部 JavaScript 代码只有 70 多行。

安装 dva-cli

> 你应该会更希望关注逻辑本身，而不是手动敲入一行行代码来构建初始的项目结构，以及配置开发环境。

那么，首先需要安装的是 dva-cli 。dva-cli 是 dva 的命令行工具，包含 init、new、generate 等功能，目前最重要的功能是可以快速生成项目以及你所需要的代码片段。

```
$ npm install -g dva-cli
```

安装完成后，可以通过 `dva -v` 查看版本，以及 `dva -h` 查看帮助信息。

创建新应用

安装完 dva-cli 后，我们用他来创建一个新应用，取名 `myApp` 。

```
$ dva new myApp --demo
```

注意：`--demo` 用于创建简单的 demo 级项目，正常项目初始化不加要这个参数。

然后进入项目目录，并启动。

```
$ cd myApp
$ npm start
```

几秒之后，会看到这样的输出：

```
          proxy: listened on 8989
     livereload: listening on 35729
  173/173 build modules
webpack: bundle build is now finished.
```

(如需关闭 server，请按 Ctrl-C.)

在浏览器里打开 http://localhost:8989/ ，正常情况下，你会看到一个 "Hello Dva" 页面。

定义 model

接到需求之后推荐的做法不是立刻编码，而是先以上帝模式做整体设计。

1. 先设计 model
2. 再设计 component
3. 最后连接 model 和 component

这个需求里，我们定义 model 如下：

```
app.model({
  namespace: 'count',
  state: {
    record : 0,
    current: 0,
  },
});
```

namespace 是 model state 在全局 state 所用的 key，state 是默认数据。然后 state 里的 record 表示 `highest record` ， `current` 表示当前速度。

完成 component

完成 Model 之后，我们来编写 Component 。推荐尽量通过 stateless functions 的方式组织 Component，在 dva 的架构里我们基本上不需要用到 state 。

```
import styles from './index.less';
const CountApp = ({count, dispatch}) => {
  return (
    <div className={styles.normal}>
      <div className={styles.record}>Highest Record: {count.reco
rd}</div>
      <div className={styles.current}>{count.current}</div>
      <div className={styles.button}>
        <button onClick={() => { dispatch({type: 'count/add'});
}}>+</button>
      </div>
    </div>
  );
};
```

注意：

1. 这里先 `import styles from './index.less';` ，再通过 `styles.xxx` 的方式声明 css classname 是基于 css-modules 的方式，后面的样式部分会用上
2. 通过 props 传入两个值， `count` 和 `dispatch` ， `count` 对应 model 上的 state，在后面 connect 的时候绑定， `dispatch` 用于分发 action
3. `dispatch({type: 'count/add'})` 表示分发了一个 `{type: 'count/add'}` 的 action，至于什么是 action，详见：Actions@redux.js.org

更新 state

更新 state 是通过 reducers 处理的，详见 Reducers@redux.js.org。

reducer 是唯一可以更新 state 的地方，这个唯一性让我们的 App 更具可预测性，所有的数据修改都有据可查。reducer 是 pure function，他接收参数 state 和 action，返回新的 state，通过语句表达即 `(state, action) => newState` 。

这个需求里，我们需要定义两个 reducer， `add` 和 `minus` ，分别用于计数的增和减。值得注意的是 `add` 时 record 的逻辑，他只在有更高的记录时才会被记录。

> 请注意，这里的 `add` 和 `minus` 两个action，在 `count` model 的定义中是不需要加 namespace 前缀的，但是在自身模型以外是需要加 model 的 namespace

```
app.model({
  namespace: 'count',
  state: {
    record: 0,
    current: 0,
  },
+ reducers: {
+   add(state) {
+     const newCurrent = state.current + 1;
+     return { ...state,
+       record: newCurrent > state.record ? newCurrent : state.r
ecord,
+       current: newCurrent,
+     };
+   },
+   minus(state) {
+     return { ...state, current: state.current - 1};
+   },
+ },
});
```

注意：

1. `{ ...state }` 里的 `...` 是对象扩展运算符，类似 `Object.extend` ，详见：对象的扩展运算符
2. `add(state) {}` 等同于 `add: function(state) {}`

绑定数据

> 还记得之前的 Component 里用到的 count 和 dispatch 吗？会不会有疑问他们来自哪里？

在定义了 Model 和 Component 之后，我们需要把他们连接起来。这样 Component 里就能使用 Model 里定义的数据，而 Model 中也能接收到 Component 里 dispatch 的 action 。

这个需求里只要用到 `count` 。

```
function mapStateToProps(state) {
  return { count: state.count };
}
const HomePage = connect(mapStateToProps)(CountApp);
```

这里的 connect 来自 react-redux。

定义路由

> 接收到 url 之后决定渲染哪些 Component，这是由路由决定的。

这个需求只有一个页面，路由的部分不需要修改。

```
app.router(({history}) =>
  <Router history={history}>
    <Route path="/" component={HomePage} />
  </Router>
);
```

注意：

1. `history` 默认是 hashHistory 并且带有 `_k` 参数，可以换成 browserHistory，也可以通过配置去掉 `_k` 参数。

现在刷新浏览器，如果一切正常，应该能看到下面的效果：

添加样式

默认是通过 css modules 的方式来定义样式，这和普通的样式写法并没有太大区别，由于之前已经在 Component 里 hook 了 className，这里只需要在 `index.less` 里填入以下内容：

```
.normal {
  width: 200px;
  margin: 100px auto;
  padding: 20px;
  border: 1px solid #ccc;
  box-shadow: 0 0 20px #ccc;
}

.record {
  border-bottom: 1px solid #ccc;
  padding-bottom: 8px;
  color: #ccc;
}

.current {
  text-align: center;
  font-size: 40px;
  padding: 40px 0;
}

.button {
  text-align: center;
  button {
    width: 100px;
    height: 40px;
    background: #aaa;
    color: #fff;
  }
}
```

效果如下：

异步处理

在此之前，我们所有的操作处理都是同步的，用户点击 + 按钮，数值加 1。

现在我们要开始处理异步任务，dva 通过对 model 增加 effects 属性来处理 side effect(异步任务)，这是基于 redux-saga 实现的，语法为 generator。(但是，这里不需要我们理解 generator，知道用法就可以了)

在这个需求里，当用户点 + 按钮，数值加 1 之后，会额外触发一个 side effect，即延迟 1 秒之后数值 1 。

```
app.model({
  namespace: 'count',
+ effects: {
+   *add(action, { call, put }) {
+     yield call(delay, 1000);
+     yield put({ type: 'minus' });
+   },
+ },
...
+function delay(timeout){
+  return new Promise(resolve => {
+    setTimeout(resolve, timeout);
+  });
+}
```

注意：

1. `*add() {}` 等同于 `add: function*(){}`
2. call 和 put 都是 redux-saga 的 effects，call 表示调用异步函数，put 表示 dispatch action，其他的还有 select, take, fork, cancel 等，详见 redux-saga 文档
3. 默认的 effect 触发规则是每次都触发(`takeEvery`)，还可以选择 `takeLatest` ，或者完全自定义 `take` 规则

刷新浏览器，正常的话，就应该已经实现了最开始需求图里的所有要求。

订阅键盘事件

> 在实现了鼠标测速之后，怎么实现键盘测速呢?

在 dva 里有个叫 subscriontions 的概念，他来自于 elm。

Subscription 语义是订阅，用于订阅一个数据源，然后根据条件 dispatch 需要的 action。数据源可以是当前的时间、服务器的 websocket 连接、keyboard 输入、geolocation 变化、history 路由变化等等。

dva 中的 subscriptions 是和 model 绑定的。

```
+import key from 'keymaster';
...
app.model({
  namespace: 'count',
+ subscriptions: {
+   keyboardWatcher({ dispatch }) {
+     key('⌘+up, ctrl+up', () => { dispatch({type:'add'}) });
+   },
+ },
});
```

这里我们不需要手动安装 keymaster 依赖，在我们敲入 `import key from 'keymaster';` 并保存的时候，dva-cli 会为我们安装 `keymaster` 依赖并保存到 `package.json` 中。输出如下：

```
use npm: tnpm
Installing `keymaster`...
[keymaster@*] installed at node_modules/.npminstall/keymaster/1.
6.2/keymaster (1 packages, use 745ms, speed 24.06kB/s, json 2.98
kB, tarball 15.08kB)
All packages installed (1 packages installed from npm registry,
use 755ms, speed 23.93kB/s, json 1(2.98kB), tarball 15.08kB)
  2/2 build modules
webpack: bundle build is now finished.
```

所有代码

index.js

```
import dva, { connect } from 'dva';
import { Router, Route } from 'dva/router';
import React from 'react';
import styles from './index.less';
import key from 'keymaster';

const app = dva();
```

```
app.model({
  namespace: 'count',
  state: {
    record: 0,
    current: 0,
  },
  reducers: {
    add(state) {
      const newCurrent = state.current + 1;
      return { ...state,
        record: newCurrent > state.record ? newCurrent : state.r
ecord,
        current: newCurrent,
      };
    },
    minus(state) {
      return { ...state, current: state.current - 1};
    },
  },
  effects: {
    *add(action, { call, put }) {
      yield call(delay, 1000);
      yield put({ type: 'minus' });
    },
  },
  subscriptions: {
    keyboardWatcher({ dispatch }) {
      key('⌘+up, ctrl+up', () => { dispatch({type:'add'}) });
    },
  },
});

const CountApp = ({count, dispatch}) => {
  return (
    <div className={styles.normal}>
      <div className={styles.record}>Highest Record: {count.reco
rd}</div>
      <div className={styles.current}>{count.current}</div>
      <div className={styles.button}>
        <button onClick={() => { dispatch({type: 'count/add'});
```

```
}}>+</button>
      </div>
    </div>
  );
};

function mapStateToProps(state) {
  return { count: state.count };
}
const HomePage = connect(mapStateToProps)(CountApp);

app.router(({history}) =>
  <Router history={history}>
    <Route path="/" component={HomePage} />
  </Router>
);

app.start('#root');


// ---------
// Helpers

function delay(timeout){
  return new Promise(resolve => {
    setTimeout(resolve, timeout);
  });
}
```

构建应用

我们已在开发环境下进行了验证，现在需要部署给用户使用。敲入以下命令：

```
$ npm run build
```

输出：

```
> @ build /private/tmp/dva-quickstart
> atool-build

Child
    Time: 6891ms
        Asset       Size  Chunks             Chunk Names
    common.js    1.18 kB       0  [emitted]  common
     index.js     281 kB    1, 0  [emitted]  index
    index.css  353 bytes    1, 0  [emitted]  index
```

该命令成功执行后，编译产物就在 dist 目录下。

下一步

通过完成这个简单的例子，大家前面的问题是否都已经有了答案？以及是否熟悉了 dva 包含的概念：model, router, reducers, effects, subscriptions ？

下一步可以进入 tutorial 了解更多。

概要

前言

需要了解的是 dva 是对 redux 的一层浅封装，所以虽然我们不要求一定要了解 redux 才能学会使用 dva，但是如果你对 redux 有所了解，再使用 dva 一定驾轻就熟，并且会了解很多潜在的知识点。redux的社区较为成熟，文档也比较健全，可以访问 http://redux.js.org（中文文档）查看更多内容，其中会介绍整个生态系统的相关框架与设计思路，值得一看。

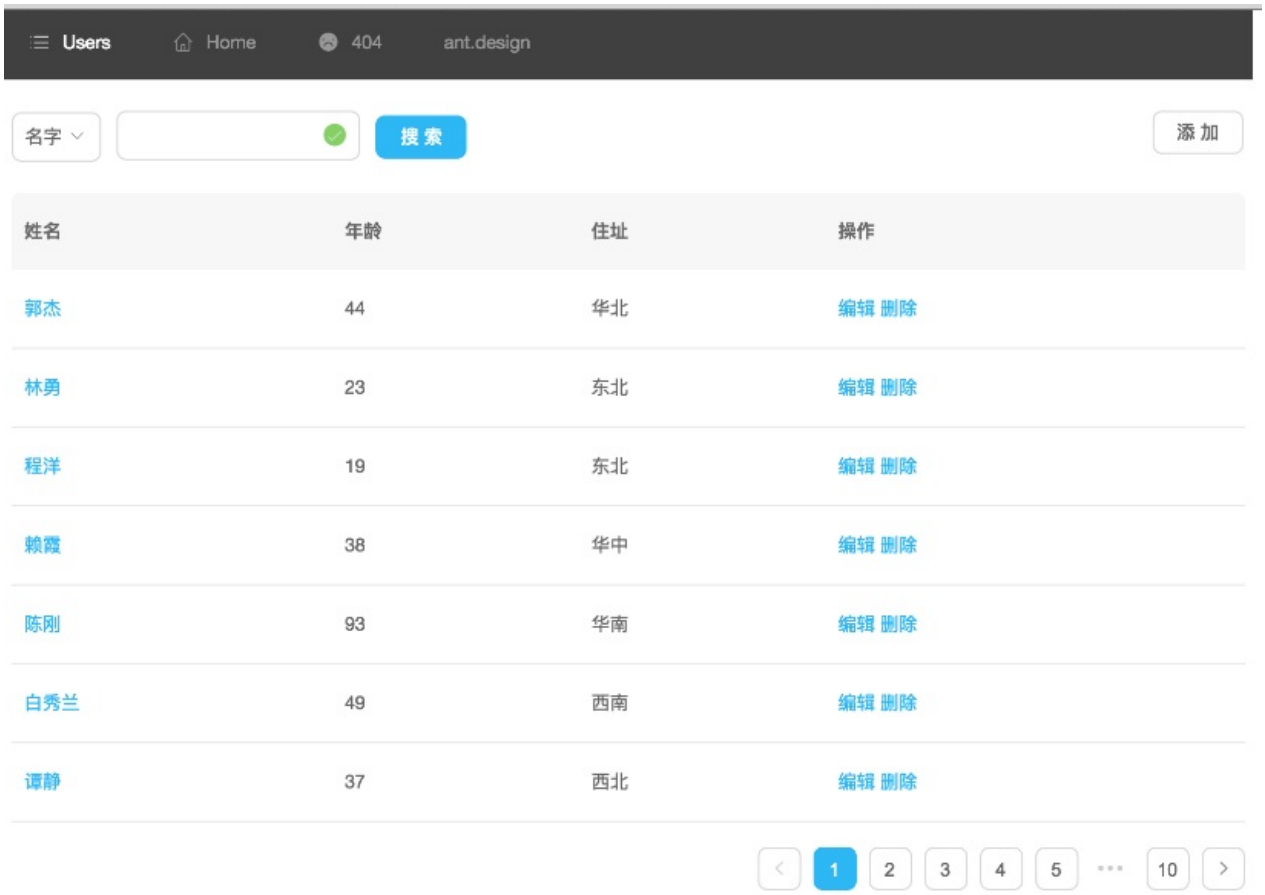
开始

在 快速上手 中我们已经对 dva 有了一定的认识，接下来我们会一起完成一个较为完善的例子，在完成这个例子的过程中，我们逐步完成以下内容：

1. 划分结构
2. 设计 Model
3. 组件设计
4. 添加样式
5. 添加 Reducers
6. 添加 Effects
7. 定义 Service
8. mock 数据
9. 添加样式
10. 设计布局

第一步，我们会划分一下整体的项目结构，熟悉每一部分是什么概念；接下来我们会说如何抽离 model，以及 model 设计的一些思路；然后我们会根据项目的情况说明如何合理的设计你的组件，以及组件中样式的处理；在设计好了组件之后，就会进入数据相关的内容，包含了同步和异步的情况，以及异步请求的处理方式，在最后我们还会介绍在dva中mock数据的的方式以及布局的设计。

下面就是我们要做的简单用户管理项目的样子：



例子的源码可以在 user-dashboard 上访问到，你也可以对照来看，不过本章的代码每一步都是可以运行的，我们建议动手跟着教程一起来。

下一步，进入划分结构。

结构划分

很多同学在搭建项目的时候，往往忽略项目结构的划分，实际上合理的项目划分往往能够提供规范的项目搭建思路。在 dva 架构的项目中，我们推荐的目录基本结构为：

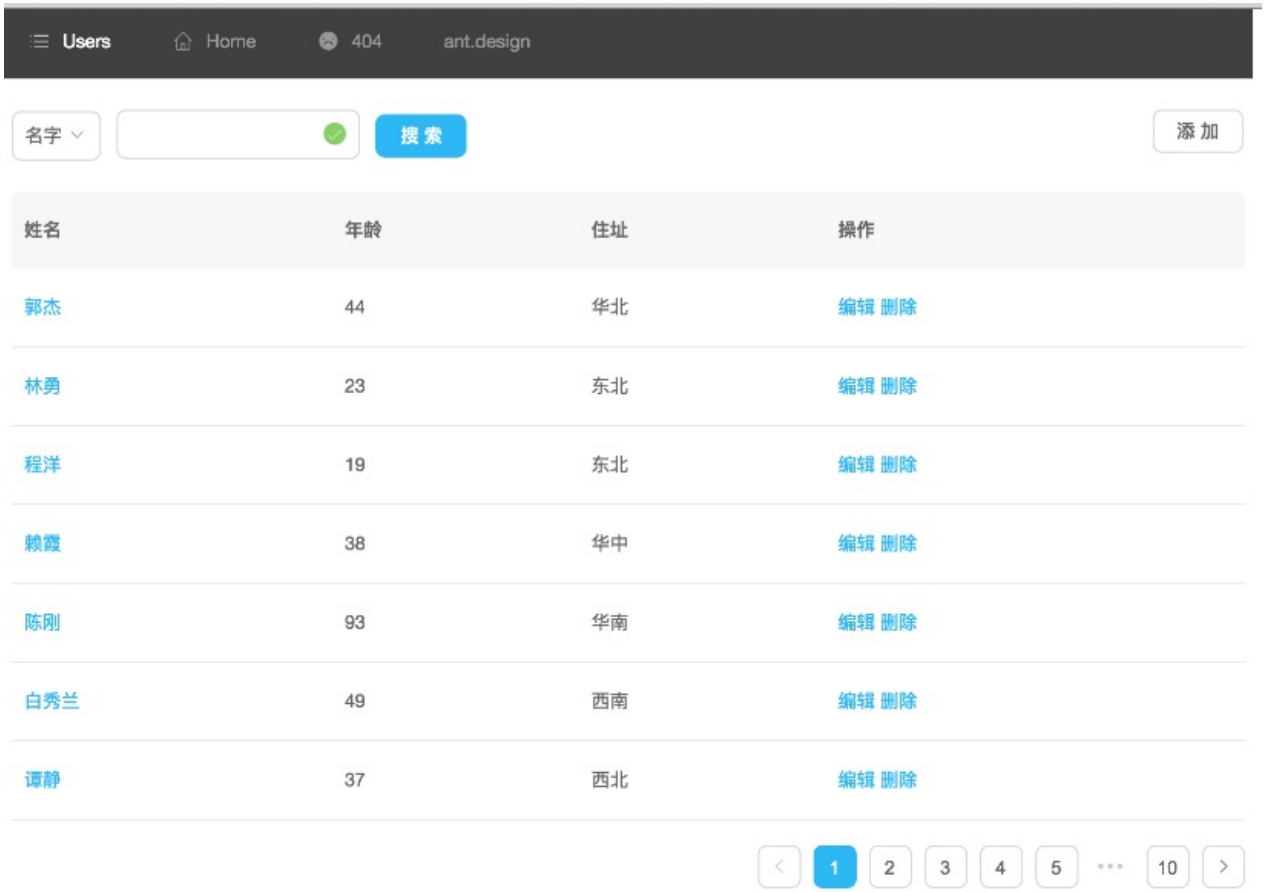
```
.
├── /mock/              # 数据mock的接口文件
├── /src/               # 项目源码目录
│ ├── /components/      # 项目组件
│ ├── /routes/          # 路由组件（页面纬度）
│ ├── /models/          # 数据模型
│ ├── /services/        # 数据接口
│ ├── /utils/           # 工具函数
│ ├── route.js          # 路由配置
│ ├── index.js          # 入口文件
│ ├── index.less
│ └── index.html
├── package.json        # 项目信息
└── proxy.config.js     # 数据mock配置
```

大家可以发现，dva 将项目中所有可能出现的功能性都映射到了对应的目录当中，并且对整个项目的功能从目录上也有了清晰的体现，所以我们建议你的项目也按照这个目录来组织文件，如果你是用的是 dva-cli 工具生成的 dva 的脚手架模板，那么会帮你按照这样目录生成好。

下一步，进入设计Model。

设计 Model

在了解了项目基本的结构划分以后，我们将要开始设计 model，在设计 model 之前，我们来回顾一下我们需要做的项目是什么样的：



Model 的抽象

从设计稿中我们可以看出，这部分功能基本是围绕 以用户数据的基础 的操作，其中包含：

1. 用户信息的展示（查询）
2. 用户信息的操作（增加，删除，修改）

有经验的同学不难发现，无论是多复杂的项目也基本上是围绕着数据的展示和操作，复杂一点的无非是组合了很多数据，有关联关系，只要分解开来，model 的结构层次依旧会很清晰，便于维护。

所以抽离的 model 的原则就是抽离数据模型，很明显在这里，我们首先会抽离一个 `users` 的 model。

Model 的设计

在抽离了 `users` 以后，我们来看下如何设计，通常有以下两种方式：

1. 按照数据纬度
2. 按照业务纬度

数据纬度

按照数据纬度的 model 设计原则就是抽离数据本身以及相关操作的方法，比如在本例的 `users` ：

```
// models/users.js

export default {
  namespace: 'users',
  state: {
    list: [],
    total: null,
  },
    effects: {
        *query(){},
        *create(){},
        // 因为delete是关键字
        *'delete'(){},
        *update(){},
    },
    reducers: {
        querySuccess(){},
        createSuccess(){},
        deleteSuccess(){},
        updateSuccess(){},
    }
}
```

如果你写过后台代码，你会发现这跟我们常常写的后台接口是很类似的，只关心数据本身，至于在使用 `users` model 的组件中所遇到的状态管理等信息是跟 model 无关，而是作为组件自身的state维护。

这种设计方式使得 model 很纯粹，在设计通用数据信息 model 的时候比较适用，比如当前用户登陆信息等数据 model。但是在数据跟业务状态交互比较紧密，数据不是那么独立的时候会有些不那么方便，因为在数据跟业务状态紧密相连的场景下，将状态放到 model 里面维护会使得我们的代码更加清晰可控，而这种方式就是下面将要介绍的 `业务纬度` 方式的设计。

业务纬度

按照业务纬度的 model 设计，则是讲数据以及使用强关联数据的组件中的状态统一抽象成 model 的方法，在本例中，`users` model设计如下：

```
// models/users.js

export default {
  namespace: 'users',

  state: {
    list: [],
    total: null,
    loading: false, // 控制加载状态
    current: null, // 当前分页信息
    currentItem: {}, // 当前操作的用户对象
    modalVisible: false, // 弹出窗的显示状态
    modalType: 'create', // 弹出窗的类型（添加用户，编辑用户）
  },
    effects: {
        *query(){},
        *create(){},
        *'delete'(){},
        *update(){},
    },
    reducers: {
        showLoading(){}, // 控制加载状态的 reducer
        showModal(){}, // 控制 Modal 显示状态的 reducer
        hideModal(){},
        querySuccess(){},
        createSuccess(){},
        deleteSuccess(){},
        updateSuccess(){},
    }
}
```

需要注意的是，有可能初次接触到 `name(){}` 的语法有些陌生，这种写法可以看成是：

```
name: function(){}
```

的简写，另外，`*name(){}` 前面的 `*` 号，表示这个方法是一个 **Generator**函数，具体可以参看Generator 函数的含义与用法

回到代码，可以看到，我们将业务状态也一并放到 model 当中去了，这样所有状态的变化都会在 model 中控制，会跟容易跟踪和操作。

根据样例项目的情况，由于数据跟业务状态关联性强，所以我们采用 `业务纬度` 的方式来设计我们的 model，在设计好了 `users` model 的基本形态以后，接下来在写代码的过程中我们会不断完善。

下一步，进入组件设计方法。

组件设计方法

在初步确定了 model 的设计方法以后，让我们来看看如何设计 dva 中的 React 组件。

组件设计

React 应用是由一个个独立的 Component 组成的，我们在拆分 Component 的过程中要尽量让每个 Component 专注做自己的事。

一般来说，我们的组件有两种设计：

1. Container Component
2. Presentational Component

Container Component

Container Component 一般指的是具有 `监听数据行为` 的组件，一般来说它们的职责是 `绑定相关联的 model 数据` ，以数据容器的角色包含其它子组件，通常在项目中表现出来的类型为：Layouts、Router Components 以及普通 Containers 组件。

通常的书写形式为：

```
import React, { Component, PropTypes } from 'react';

// dva 的 connect 方法可以将组件和数据关联在一起
import { connect } from 'dva';

// 组件本身
const MyComponent = (props)=>{};
MyComponent.propTypes = {};

// 监听属性，建立组件和数据的映射关系
function mapStateToProps(state) {
  return {...state.data};
}

// 关联 model
export default connect(mapStateToProps)(MyComponent);
```

Presentational Component

Presentational Component 的名称已经说明了它的职责，展示形组件，一般也称作：Dump Component，它不会关联订阅 modal 上的数据，而所需数据的传递则是通过 props 传递到组件内部。

通常的书写形式：

```
import React, { Component, PropTypes } from 'react';

// 组件本身
// 所需要的数据通过 Container Component 通过 props 传递下来
const MyComponent = (props)=>{}
MyComponent.propTypes = {};

// 并不会监听数据
export default MyComponent;
```

对比

对组件分类，主要有两个好处：

1. 让项目的数据处理更加集中；
2. 让组件高内聚低耦合，更加聚焦；

试想如果每个组件都去订阅数据 model，那么一方面组件本身跟 model 耦合太多，另一方面代码过于零散，到处都在操作数据，会带来后期维护的烦恼。

除了写法上订阅数据的区别以外，在设计思路上两个组件也有很大不同。`Presentational Component` 是独立的存粹的，这方面很好的例子，大家可以参考 ant.design UI组件的React实现 ，每个组件跟业务数据并没有耦合关系，只是完成自己独立的任务，需要的数据通过 `props` 传递进来，需要操作的行为通过接口暴露出去。 而 `Container Component` 更像是状态管理器，它表现为一个容器，订阅自组件需要的数据，组织子组件的交互逻辑和展示。

更多的相关内容，可以看看Redux作者（facebook的程序员）Dan Abramov 的看法：

- https://github.com/reactjs/redux/issues/756#issuecomment-141683834
- https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.231v4pdgr

接下来我们会进入本例中组件的设计开发，从实践一步一步看看如何贯穿起来。

下一步，进入组件设计实践。

组件设计实践

准备

建议对照完整代码一起看 user-dashboard。

按照之前快速上手的内容，我们可以使用 dva-cli 工具快速生成规范的目录，在命令行中输入：

```
$ mkdir myApp && cd myApp
$ dva init
```

现在，规范的样例模板我们已经有了，接下来我们一步一步添加自己的东西，看看如何完成我们的组件设计。

设置路由

在准备好了 dva 的基本框架以后，需要为我们的项目配置一下路由，这里首先设置 Users Router Container 的访问路径，并且在 `/routes/` 下创建我们的组件文件 `User.jsx` 。

```
// .src/router.js
import React, { PropTypes } from 'react';
import { Router, Route } from 'react-router';
import Users from './routes/Users';

export default function({ history }) {
  return (
    <Router history={history}>
      <Route path="/users" component={Users} />
    </Router>
  );
};
```

```
// .src/routes/User.jsx
import React, { PropTypes } from 'react';

function User() {
  return (
    <div>User Router Component</div>
  );
}

User.propTypes = {
};

export default User;
```

其它路由可以自行添加，关于路由更多信息，可以查看 react-router 获取更多内容。

Users Container Component 的设计

基础工作都准备好了，接下来就开始设计 Users Container Component。在本项目中 Users Container 的表现为 Route Components（这也是 dva 推荐的结构划分），可以理解页面维度的容器，所以我们在 `/routes/` 下加入 `Users.jsx` 。

我们采用 `自顶向下` 的设计方法，修改 `./src/routes/Users.jsx` 如下：

```
// ./src/routes/Users.jsx
import React, { Component, PropTypes } from 'react';

// Users 的 Presentational Component
// 暂时都没实现
import UserList from '../components/Users/UserList';
import UserSearch from '../components/Users/UserSearch';
import UserModal from '../components/Users/UserModal';

// 引入对应的样式
// 可以暂时新建一个空的
import styles from './Users.less';

function Users() {

  const userSearchProps = {};
  const userListProps = {};
  const userModalProps = {};

  return (
    <div className={styles.normal}>
      {/* 用户筛选搜索框 */}
      <UserSearch {...userSearchProps} />
      {/* 用户信息展示列表 */}
      <UserList {...userListProps} />
      {/* 添加用户 & 修改用户弹出的浮层 */}
      <UserModal {...userModalProps} />
    </div>
  );
}

export default Users;
```

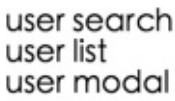
其中，`UserSearch`，`UserList`，`UserModal` 我们还未实现，不过我们可以暂时让他们输出一段话，表示占位，基本的结构表现的很清楚，`Users Router Container` 由这三个 `Presentational Components` 组成。（其中{...x}的用法可以参看es6）

```
// ./src/components/Users/UserSearch.jsx
import React, { PropTypes } from 'react';
export default ()=><div>user search</div>;
```

```
// ./src/components/Users/UserList.jsx
import React, { PropTypes } from 'react';
export default ()=><div>user list</div>;
```

```
// ./src/components/Users/UserModal.jsx
import React, { PropTypes } from 'react';
export default ()=><div>user modal</div>;
```

现在如果你的本地环境是成功的，访问 http://127.0.0.1:8989/#/users 浏览器中看到：



需要注意的是，定义我们的组件一般有三种方式：

```
// 1. 传统写法
const App = React.createClass({});

// 2. es6 的写法
class App extends React.Component({});

// 3. stateless 的写法（我们推荐的写法）
const App = (props) => ({});
```

其中第1种是我们不推荐的写法，第2种是在你的组件涉及 react 的生命周期方法的时候采用这种写法，而第3种则是我们一般推荐的写法。详细内容可以参看Stateless Functions。

在确定了最简陋的结构以后，接下来需要做的事情，就是完善 Users Container 中的组件，在这里我们优先实现 `UserList` 组件。

Userlist 组件

暂时放下 `<UserSearch />` 和 `<UserModal />` ，先来看看 `<UserList />` 的实现，这是一个用户的展示列表，我们期望只需要把数据传入进去，修改 `./src/components/Users/UserList.jsx` ：

```
// ./src/components/Users/UserList.jsx
import React, { Component, PropTypes } from 'react';

// 采用antd的UI组件
import { Table, message, Popconfirm } from 'antd';

// 采用 stateless 的写法
const UserList = ({
    total,
        current,
        loading,
        dataSource,
}) => {
  const columns = [{
    title: '姓名',
    dataIndex: 'name',
    key: 'name',
    render: (text) => <a href="#">{text}</a>,
  }, {
    title: '年龄',
    dataIndex: 'age',
    key: 'age',
  }, {
    title: '住址',
    dataIndex: 'address',
    key: 'address',
  }, {
    title: '操作',
    key: 'operation',
    render: (text, record) => (
      <p>
        <a onClick={()=>{}}>编辑</a>
         
```

```
          <Popconfirm title="确定要删除吗？" onConfirm={()=>{}}>
            <a>删除</a>
          </Popconfirm>
        </p>
      ),
    }];

    // 定义分页对象
    const pagination = {
    total,
    current,
    pageSize: 10,
    onChange: ()=>{},
  };

  return (
    <div>
      <Table
        columns={columns}
        dataSource={dataSource}
        loading={loading}
        rowKey={record => record.id}
        pagination={pagination}
      />
    </div>
  );
}

export default UserList;
```

为了方便起见，我们这里使用一个优秀的UI组件库 antd。 `antd` 提供了 table 组件，可以让我们方便的展示相关数据，具体使用方式可以参看其文档。

需要注意的是，由于我们采用了 antd，所以我们需要在我们的代码中添加样式，可以在 `./src/index.jsx` 中添加一行：

```
+ import 'antd/dist/antd.css';
```

这样我们使用的的 antd 组件就可以展示出样子了：

其中我们发现，在我们设计 `UserList` 的时候，需要将分页信息 `total`、`current` 以及加载状态信息 `loading` 也传入进来，所以现在使用 `UserList` 就需要像这样：

```
<UserList
    current={current}
    total={total}
    dataSource={list}
    loading={loading}
/>
```

接下来，我们回到 `Users Router Container` 模拟一些静态数据，传入 UserList，让其展现数据。

给 UserList 添加静态数据

```
// ./src/routes/Users.jsx
import React, { Component, PropTypes } from 'react';

// Users 的 Presentational Component
// 暂时都没实现
import UserList from '../components/Users/UserList';
import UserSearch from '../components/Users/UserSearch';
import UserModal from '../components/Users/UserModal';

// 引入对应的样式
// 可以暂时新建一个空的
import styles from './Users.less';
```

```
function Users() {

  const userSearchProps={};
  const userListProps={
    total: 3,
    current: 1,
    loading: false,
    dataSource: [
      {
        name: '张三',
        age: 23,
        address: '成都',
      },
      {
        name: '李四',
        age: 24,
        address: '杭州',
      },
      {
        name: '王五',
        age: 25,
        address: '上海',
      },
    ],
  };
  const userModalProps={};

  return (
    <div className={styles.normal}>
      {/* 用户筛选搜索框 */}
      <UserSearch {...userSearchProps} />
      {/* 用户信息展示列表 */}
      <UserList {...userListProps} />
      {/* 添加用户 & 修改用户弹出的浮层 */}
      <UserModal {...userModalProps} />
    </div>
  );
}

Users.propTypes = {
```

```
  users: PropTypes.object,
};

export default Users;
```

传入了静态数据以后，组件的表现如下：

user search

姓名	年龄	住址	操作
张三	23	成都	编辑 删除
李四	24	杭州	编辑 删除
王五	25	上海	编辑 删除

user modal

组件设计小结

虽然我们上面实现的代码很简单，但是已经包含了组件设计的主要思路，可以看到 `UserList` 组件是一个很纯粹的 `Presentation Component`，所需要的数据以及状态是通过 `Users Router Component` 传递的，我们现在还是用的静态数据，接下来我们来看看如何在 modal 创建 **reducer** 来将我们的数据抽象出来。

下一步，进入添加Reducers。

添加Reducers

理解 Reducers

首先需要理解什么是 **reducer**，dva 中 reducer 的概念，主要是来源于下层封装的 redux，在 dva 中 reducers 主要负责修改 model 的数据（state）。

也许你在迷惑，为什么会叫做 reducer 这个名字，你或许知道 reduce 这个方法，在很多程序语言中，数组类型都具备 reduce 方法，而这个方法的功能就是聚合，比如下面这个在 javascript 中的例子：

```
[{x:1},{y:2},{z:3}].reduce(function(prev, next){
    return Object.assign(prev, next);
})
//return {x:1, y:2, z:3}
```

可以看到，在上面的这个例子中，我将三个对象合并成了一个对象，这就是 reducer 的思想，model 的数据就是通过我们分离出来的 reducer 创建出来的，这样可以让每个 reducer 专注于相关数据的修改，但是最终会构建出完整的数据。

如果你想了解更多，可以参看 Redux Reducers。

给 Users Model 添加 Reducers

回到我们之前的 `/models/users.js`，我们在之前已经定义好了它的 state，接下来我们看看如何根据新的数据来修改本身的 state，这就是 reducers 要做的事情。

```
export default {
  namespace: 'users',

  state: {
    list: [],
    total: null,
    loading: false, // 控制加载状态
    current: null, // 当前分页信息
    currentItem: {}, // 当前操作的用户对象
```

```
    modalVisible: false, // 弹出窗的显示状态
    modalType: 'create', // 弹出窗的类型（添加用户，编辑用户）
  },
  effects: {
    *query(){},
    *create(){},
    *'delete'(){},
    *update(){},
  },
  reducers: {
    showLoading(){}, // 控制加载状态的 reducer
    showModal(){}, // 控制 Modal 显示状态的 reducer
    hideModal(){},
    // 使用静态数据返回
    querySuccess(state){
      const mock = {
        total: 3,
        current: 1,
        loading: false,
        list: [
          {
            name: '张三',
            age: 23,
            address: '成都',
          },
          {
            name: '李四',
            age: 24,
            address: '杭州',
          },
          {
            name: '王五',
            age: 25,
            address: '上海',
          },
        ],

      };
      return {...state, ...mock, loading: false};
    },
```

```
    createSuccess(){},
    deleteSuccess(){},
    updateSuccess(){},
  }
}
```

我们把之前 `UserList` 组件中模拟的静态数据，移动到了 reducers 中，通过调用 `'users/query/success'` 这个 reducer，我们就可以将 Users Modal 的数据变成静态数据，那么我们如何调用这个 reducer，能够让这个数据传入 UserList 组件呢，接下来需要做的是：关联**Model**。

关联 Model

```
// ./src/routes/users.jsx
import React, { Component, PropTypes } from 'react';

// 引入 connect 工具函数
import { connect } from 'dva';

// Users 的 Presentational Component
// 暂时都没实现
import UserList from '../components/Users/UserList';
import UserSearch from '../components/Users/UserSearch';
import UserModal from '../components/Users/UserModal';

// 引入对应的样式
// 可以暂时新建一个空的
import styles from './Users.less';

function Users({ location, dispatch, users }) {

  const {
    loading, list, total, current,
    currentItem, modalVisible, modalType
    } = users;

  const userSearchProps={};
  const userListProps={
```

```
        dataSource: list,
        total,
        loading,
        current,
    };
  const userModalProps={};

  return (
    <div className={styles.normal}>
      {/* 用户筛选搜索框 */}
      <UserSearch {...userSearchProps} />
      {/* 用户信息展示列表 */}
      <UserList {...userListProps} />
      {/* 添加用户 & 修改用户弹出的浮层 */}
      <UserModal {...userModalProps} />
    </div>
  );
}

Users.propTypes = {
  users: PropTypes.object,
};

// 指定订阅数据，这里关联了 users
function mapStateToProps({ users }) {
  return {users};
}

// 建立数据关联关系
export default connect(mapStateToProps)(Users);
```

在之前的 组件设计 中讲到了 `Presentational Component` 的设计概念，在订阅了数据以后，就可以通过 `props` 访问到 modal 的数据了，而 UserList 展示组件的数据，也是 `Container Component` 通过 `props` 传递的过来的。

组件和 model 建立了关联关系以后，如果在组件中获取 reduers 的数据呢，或者如何调用 reducers呢，就是需要发起一个 action。

发起 Actions

actions 的概念跟 reducers 一样，也是来自于 dva 封装的 redux，表达的概念是发起一个修改数据的行为，主要的作用是传递信息：

```
dispatch({
    type: '', // action 的名称，与 reducers（effects）对应
    ... // 调用时传递的参数，在 reducers（effects）可以获取
});
```

需要注意的是：**action**的名称（**type**）如果是在 **model** 以外调用需要添加 **namespace**。

通过 dispatch 函数，可以通过 type 属性指定对应的 actions 类型，而这个类型名在 reducers（effects）会一一对应，从而知道该去调用哪一个 reducers（effects），除了 type 以外，其它对象中的参数随意定义，都可以在对应的 reducers（effects）中获取，从而实现消息传递，将最新的数据传递过去更新 model 的数据（state）。

相关的更多信息，可以参看 Redux Actions。

回到例子中，目前传入 UserList 组件的只是默认空数据，那么如何调用 reducers 获取刚才定义的静态数据呢？发起一个 **actions**：

```
dispatch({
    type: 'users/querySuccess', // 调用哪个一个actions
    payload: {}, // 调用时传递的参数
});
```

知道了如何发起一个 action，那么剩下的就是发起的时机了，通常我们建议在组件内部的生命周期发起，如：

```
...
componentDidMount() {
    this.props.dispatch({
        type: 'model/action',
    });
}
...
```

不过在本例中采用另一种发起 action 的场景，在本例中获取用户数据信息的时机就是访问 /users/ 这个页面，所以我们可以监听路由信息，只要路径是 /users/ 那么我们就会发起 action，获取用户数据：

```
// ./src/models/users.js
import { hashHistory } from 'dva/router';

export default {
  namespace: 'users',

  state: {
    list: [],
    total: null,
    loading: false, // 控制加载状态
    current: null, // 当前分页信息
    currentItem: {}, // 当前操作的用户对象
    modalVisible: false, // 弹出窗的显示状态
    modalType: 'create', // 弹出窗的类型（添加用户，编辑用户）
  },

  // Quick Start 已经介绍过 subscriptions 的概念，这里不在多说
  subscriptions: {
    setup({ dispatch, history }) {
      history.listen(location => {
        if (location.pathname === '/users') {
          dispatch({
            type: 'querySuccess',
            payload: {}
          });
        }
      });
    },
  },

  effects: {
    *query(){},
    *create(){},
    *'delete'(){},
    *update(){},
  },
```

```
  reducers: {
    showLoading(){}, // 控制加载状态的 reducer
    showModal(){}, // 控制 Modal 显示状态的 reducer
    hideModal(){},
    // 使用静态数据返回
    querySuccess(state){
      const mock = {
        total: 3,
        current: 1,
        loading: false,
        list: [
          {
            name: '张三',
            age: 23,
            address: '成都',
          },
          {
            name: '李四',
            age: 24,
            address: '杭州',
          },
          {
            name: '王五',
            age: 25,
            address: '上海',
          },
        ],

      };
      return {...state, ...mock, loading: false};
    },
    createSuccess(){},
    deleteSuccess(){},
    updateSuccess(){},
  }
}
```

以上代码在浏览器访问 /users 路径的时候就会发起一个 action，数据准备完毕，别忘了回到 `index.js` 中，添加我们的 models：

```
// ./src/index.js
import './index.html';
import './index.less';
import dva, { connect } from 'dva';

import 'antd/dist/antd.css';

// 1. Initialize
const app = dva();

// 2. Model
app.model(require('./models/users.js'));

// 3. Router
app.router(require('./router'));

// 4. Start
app.start(document.getElementById('root'));
```

如果一切正常，访问：http://127.0.0.1:8989/#/users ，可以看到：

小结

在这个例子中，我们在 合适的时机（进入 /users/ ）发起（dispatch）了一个 action，修改了 model 的数据，并且通过 Container Components 关联了 model，通过 props 传递到 Presentation Components，组件成功显示。如果你想了解更多关于 reducers & actions 的信息，可以参看 redux。

下一步，进入添加Effects。

添加Effects

在之前的教程中，我们已经完成了静态数据的操作，但是在真实场景，数据都是从服务器来的，我们需要发起异步请求，在请求回来以后设置数据，更新 state，那么在 dva 中，这一切是怎么操作的呢，首先我们先来简单了解一下 Effects。

理解 Effects

Effects 来源于 dva 封装的底层库 redux-sagas 的概念，主要指的是处理 `Side Effects` ，指的是副作用（源于函数式编程），在这里可以简单理解成异步操作，所以我们是不是可以理解成 Reducers 处理同步，Effects 处理异步？这么理解也没有问题，但是要认清 Reducers 的本质是修改 model 的 state，而 Effects 主要是 控制数据流程 ，所以最终往往我们在 Effects 中会调用 Reducers。

> 在函数式编程中，我们强调纯函数的使用：纯函数是这样一种函数，即相同的输入，永远会得到相同的输出，而且没有任何可观察的副作用；简单理解就是每个函数职责纯粹，所有行为就是返回新的值，没有其他行为，真实情况最常见的副作用就是异步操作，所以 dva 提供了 effects 来专门放置副作用，不过可以发现的是，由于 effects 使用了 Generator Creator，所以将异步操作同步化，也是纯函数。更多相关可以参看 JS函数式编程指南。

给 Users Model 添加 Effects

```
// ./src/models/users.js
import { hashHistory } from 'dva/router';
//import { create, remove, update, query } from '../services/use
rs';

// 处理异步请求
import request from '../utils/request';
import qs from 'qs';
async function query(params) {
  return request(`/api/users?${qs.stringify(params)}`);
}

export default {
```

```
  namespace: 'users',

  state: {
    list: [],
    total: null,
    loading: false, // 控制加载状态
    current: null, // 当前分页信息
    currentItem: {}, // 当前操作的用户对象
    modalVisible: false, // 弹出窗的显示状态
    modalType: 'create', // 弹出窗的类型（添加用户，编辑用户）
  },

  subscriptions: {
    setup({ dispatch, history }) {
      history.listen(location => {
        if (location.pathname === '/users') {
          dispatch({
            type: 'querySuccess',
            payload: {}
          });
        }
      });
    },
  },

  effects: {
    *query({ payload }, { select, call, put }) {
      yield put({ type: 'showLoading' });
      const { data } = yield call(query);
      if (data) {
        yield put({
          type: 'querySuccess',
          payload: {
            list: data.data,
            total: data.page.total,
            current: data.page.current
          }
        });
      }
    },
```

```
    *create(){},
    *'delete'(){},
    *update(){},
  },
  reducers: {
    showLoading(state, action){
      return { ...state, loading: true };
    }, // 控制加载状态的 reducer
    showModal(){}, // 控制 Modal 显示状态的 reducer
    hideModal(){},
    // 使用静态数据返回
    querySuccess(state, action){
      return {...state, ...action.payload, loading: false};
    },
    createSuccess(){},
    deleteSuccess(){},
    updateSuccess(){},
  }
}
```

首先我们需要增加 `*query` 第二个参数 `*query({ payload }, { select, call, put })` ，其中 call 和 put 是 dva 提供的方便操作 effects 的函数，简单理解 call 是调用执行一个函数而 put 则是相当于 dispatch 执行一个 action，而 select 则可以用来访问其它 model，更多可以参看 redux-saga-in-chinese。

而在 `query` 函数里面，可以看到我们处理异步的方式跟同步一样，所以能够很好的控制异步流程，这也是我们使用 Effects 的原因，关于相关的更多内容可以参看 Generator 函数的含义与用法。

这里我们把请求的处理直接写在了代码里面，接下来我们需要把它拆分到 `/services/` 里面统一处理：

```
import request from '../utils/request';
import qs from 'qs';
async function query(params) {
  return request(`/api/users?${qs.stringify(params)}`);
}
```

关于 async 的用法，可以参看 async 函数的含义和用法，需要注意的是，无论是 Generator 函数，yield 亦或是 async 目的只有一个： 让异步编写跟同步一样 ，从而能够很好的控制执行流程。

下一步，进入定义Services。

定义Services

之前我们已经：

1. 设计好了 model state -> 抽象数据
2. 完善了组件 -> 完善展示
3. 添加了 Reducers -> 数据同步处理
4. 添加了 Effects -> 数据异步处理

接下来就是将请求相关（与后台系统的交互）抽离出来，单独放到 `/services/` 中，进行统一维护管理，所以我们只需要将之前定义在 Effects 的以下代码，移动到 `/services/users.js` 中即可：

```
// request 是我们封装的一个网络请求库
import request from '../utils/request';
import qs from 'qs';

export async function query(params) {
  return request(`/api/users?${qs.stringify(params)}`);
}
```

```
// ./src/utils/request.js
import fetch from 'dva/fetch';

function parseJSON(response) {
  return response.json();
}

function checkStatus(response) {
  if (response.status >= 200 && response.status < 300) {
    return response;
  }

  const error = new Error(response.statusText);
  error.response = response;
  throw error;
}

/**
 * Requests a URL, returning a promise.
 *
 * @param  {string} url       The URL we want to request
 * @param  {object} [options] The options we want to pass to "fe
tch"
 * @return {object}           An object containing either "data"
 or "err"
 */
export default function request(url, options) {
  return fetch(url, options)
    .then(checkStatus)
    .then(parseJSON)
    .then((data) => ({ data }))
    .catch((err) => ({ err }));
}
```

然后在 users model 中引入：

```
import { query } from '../services/users';
```

之后无论是更新，删除、添加等操作，跟用户相关的都可以统一放置在 `/services/users.js` 中。

代码写到这里，你可以看到浏览器中的显示是这样的：

user search

user modal

没错，我们虽然有了接口，但是我们还没有数据，在 dva 中，我们配套的工具能够很方便的模拟数据，这样就可以脱离服务器复杂的环境进行模拟的本地调试开发。下面一节就会一起来看下，如何 mock 数据。

下一步，进入mock数据。

mock数据

我们们采用了 dora-plugin-proxy 工具来完成了我们的数据 mock 功能。

在 `package.json` 中：

```
  "scripts": {
    "start": "dora --plugins \"proxy,webpack,webpack-hmr\"",
    "lint": "eslint --fix --ext .js,.jsx .",
    "build": "atool-build"
  }
```

的start命令中，可以看到使用 dora 工具的相关内容，其中 `proxy` 就是dora的一个插件，在你的项目不需要代理的时候，去除proxy插件即可。

mock文件如下：

```
// ./src/mock/users.js
'use strict';

const qs = require('qs');

// 引入 mock js
const mockjs = require('mockjs');

module.exports = {
  'GET /api/users' (req, res) {
    const page = qs.parse(req.query);

    const data = mockjs.mock({
      'data|100': [{
        'id|+1': 1,
        name: '@cname',
        'age|11-99': 1,
        address: '@region'
      }],
      page: {
        total: 100,
        current: 1
      }
    });

    res.json({
      success: true,
      data,
      page: {
        current: 1,
        total: 100,
      }
    });
  },
};
```

在完整样例中，mock 数据更为全面。

具体更多的使用方法，可以参看 dora-plugin-proxy 文档。

到此为止，我们围绕 UserList 组件的实现，从各方面展示了 dva 项目的设计思路以及方法，剩下的样例项目内容大同小异，关于应用中其它组件的实现和相关内容，请参看完整内容。

下一步，进入添加样式。

添加样式

添加样式的方案

在 dva 中，所有的页面都是基于组件的。因此，我们希望样式依附于组件，不同组件的样式相互之间不会造成污染。

在 dva 中，我们推荐使用 CSS Modules 的解决方案。配合 webpack 的 css-loader 进行打包，会为所有的 class name 和 animation name 加 local scope，避免潜在冲突。

样式引入示例

参考 example 中的示例：

`UserSearch` 组件：

```
// /components/Users/UserSearch.jsx
...
import styles from './UserSearch.less';
...

function UserSearch({
  form, field, keyword,
  onSearch,
  onAdd
  }) {
  ...
  return (
    <div className={styles.normal}>
      <div className={styles.search}>
        ...
      </div>
      <div className={styles.create}>
        <Button type="ghost" onClick={onAdd}>添加</Button>
      </div>
    </div>
  )
}

...
```

对应的 `UserSearch.less` 样式文件：

```
/* /components/Users/UserSearch.less */
.normal {
  display: flex;
  margin-bottom: 20px;
}

.search {
  flex: 1;
}

.create {
}
```

`CSS Modules` 会给组件的 `className` 加上hash字符串，来保证 `className` 仅对引用了样式的组件有效，如 `styles.normal` 可能会输出为 `normal___39QwY` 。

`className` 的输出格式可以通过 `webpack.config` 进行修改。

下一步，进入设计布局。

设计布局

在项目中，我们通常都会有布局组件的概念，常见的场景是整个项目通用的头尾，侧边栏，以及整体布局结构等，这些布局内容被抽象成组件，包含一些布局样式，用于组合其它组件搭建成页面。

说白了，其实它本质上还是一种组件，讲布局样式抽象成组件，能够保持子组件和父组件的独立性，不用在其中关联到布局信息。

如我们的样例项目中的 `MainLayout.jsx` ：

```
// ./components/MainLayout/MainLayout.jsx
import React, { PropTypes } from 'react';
import styles from './MainLayout.less';
import Header from './Header';

// 包含默认头部的布局组件
function MainLayout({ children, location }) {
  return (
    <div className={styles.normal}>
      <Header location={location} />
      <div className={styles.content}>
        <div className={styles.main}>
          {children}
        </div>
      </div>
    </div>
  );
}

MainLayout.propTypes = {
  children: PropTypes.element.isRequired,
  location: PropTypes.object,
};

export default MainLayout;
```

基础教程到这里就结束了，更多相关内容可参看 样例完整代码 。