dva.js 知识导图

> 注：如果你使用 dva@2，请先忽略这里的路由部分，待更新。

不知大家学 react 或 dva 时会不会有这样的疑惑：

- es6 特性那么多，我需要全部学会吗?
- react component 有 3 种写法，我需要全部学会吗?
- reducer 的增删改应该怎么写?
- 怎么做全局/局部的错误处理?
- 怎么发异步请求?
- 怎么处理复杂的异步业务逻辑?
- 怎么配置路由?
- ...

这篇文档梳理了基于 dva-cli 使用 dva 的最小知识集，让你可以用最少的时间掌握创建类似 dva-hackernews 的全部知识，并且不需要掌握额外的冗余知识。

目录

JavaScript 语言

变量声明

const 和 let

不要用 `var`，而是用 `const` 和 `let`，分别表示常量和变量。不同于 `var` 的函数作用域，`const` 和 `let` 都是块级作用域。

```
const DELAY = 1000;

let count = 0;
count = count + 1;
```

模板字符串

模板字符串提供了另一种做字符串组合的方法。

```
const user = 'world';
console.log(`hello ${user}`);  // hello world

// 多行
const content = `
  Hello ${firstName},
  Thanks for ordering ${qty} tickets to ${event}.
`;
```

默认参数

```
function logActivity(activity = 'skiing') {
  console.log(activity);
}
```

```
logActivity();  // skiing
```

箭头函数

函数的快捷写法，不需要通过 `function` 关键字创建函数，并且还可以省略 `return` 关键字。

同时，箭头函数还会继承当前上下文的 `this` 关键字。

比如：

```
[1, 2, 3].map(x => x + 1);  // [2, 3, 4]
```

等同于：

```
[1, 2, 3].map((function(x) {
  return x + 1;
}).bind(this));
```

模块的 Import 和 Export

`import` 用于引入模块，`export` 用于导出模块。

比如：

```
// 引入全部
import dva from 'dva';

// 引入部分
import { connect } from 'dva';
import { Link, Route } from 'dva/router';

// 引入全部并作为 github 对象
import * as github from './services/github';

// 导出默认
export default App;
// 部分导出，需 import { App } from './file'; 引入
export class App extend Component {};
```

ES6 对象和数组

析构赋值

析构赋值让我们从 Object 或 Array 里取部分数据存为变量。

```
// 对象
const user = { name: 'guanguan', age: 2 };
const { name, age } = user;
console.log(`${name} : ${age}`);  // guanguan : 2

// 数组
const arr = [1, 2];
const [foo, bar] = arr;
console.log(foo);  // 1
```

我们也可以析构传入的函数参数。

```
const add = (state, { payload }) => {
  return state.concat(payload);
};
```

析构时还可以配 alias，让代码更具有语义。

```
const add = (state, { payload: todo }) => {
  return state.concat(todo);
};
```

对象字面量改进

这是析构的反向操作，用于重新组织一个 Object 。

```
const name = 'duoduo';
const age = 8;

const user = { name, age };  // { name: 'duoduo', age: 8 }
```

定义对象方法时，还可以省去 `function` 关键字。

```
app.model({
  reducers: {
    add() {}  // 等同于 add: function() {}
  },
  effects: {
    *addRemote() {}  // 等同于 addRemote: function*() {}
  },
});
```

Spread Operator

Spread Operator 即 3 个点 `...`，有几种不同的使用方法。

可用于组装数组。

```
const todos = ['Learn dva'];
[...todos, 'Learn antd'];  // ['Learn dva', 'Learn antd']
```

也可用于获取数组的部分项。

```
const arr = ['a', 'b', 'c'];
const [first, ...rest] = arr;
rest;  // ['b', 'c']

// With ignore
const [first, , ...rest] = arr;
rest;  // ['c']
```

还可收集函数参数为数组。

```
function directions(first, ...rest) {
  console.log(rest);
}
directions('a', 'b', 'c');  // ['b', 'c'];
```

代替 apply。

```
function foo(x, y, z) {}
const args = [1,2,3];

// 下面两句效果相同
foo.apply(null, args);
foo(...args);
```

对于 Object 而言，用于组合成新的 Object 。(ES2017 stage-2 proposal)

```
const foo = {
  a: 1,
  b: 2,
```

```
};
const bar = {
  b: 3,
  c: 2,
};
const d = 4;

const ret = { ...foo, ...bar, d };  // { a:1, b:3, c:2, d:4 }
```

此外，在 JSX 中 Spread Operator 还可用于扩展 props，详见 Spread Attributes。

Promises

Promise 用于更优雅地处理异步请求。比如发起异步请求：

```
fetch('/api/todos')
  .then(res => res.json())
  .then(data => ({ data }))
  .catch(err => ({ err }));
```

定义 Promise 。

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

delay(1000).then(_ => {
  console.log('executed');
});
```

Generators

dva 的 effects 是通过 generator 组织的。Generator 返回的是迭代器，通过 `yield` 关键字实现暂停功能。

这是一个典型的 dva effect，通过 `yield` 把异步逻辑通过同步的方式组织起来。

```
app.model({
  namespace: 'todos',
  effects: {
    *addRemote({ payload: todo }, { put, call }) {
      yield call(addTodo, todo);
      yield put({ type: 'add', payload: todo });
    },
  },
});
```

React Component

Stateless Functional Components

React Component 有 3 种定义方式，分别是 `React.createClass`, `class` 和 `Stateless Functional Component`。推荐尽量使用最后一种，保持简洁和无状态。这是函数，不是 Object，没有 `this` 作用域，是 pure function。

比如定义 App Component 。

```
function App(props) {
  function handleClick() {
    props.dispatch({ type: 'app/create' });
  }
  return <div onClick={handleClick}>${props.name}</div>
}
```

等同于：

```
class App extends React.Component {
  handleClick() {
    this.props.dispatch({ type: 'app/create' });
  }
  render() {
    return <div onClick={this.handleClick.bind(this)}>${this.props.name}</div>
  }
}
```

JSX

Component 嵌套

类似 HTML，JSX 里可以给组件添加子组件。

```
<App>
  <Header />
  <MainContent />
  <Footer />
</App>
```

className

`class` 是保留词，所以添加样式时，需用 `className` 代替 `class` 。

```
<h1 className="fancy">Hello dva</h1>
```

JavaScript 表达式

JavaScript 表达式需要用 `{}` 括起来，会执行并返回结果。

比如：

```
<h1>{ this.props.title }</h1>
```

Mapping Arrays to JSX

可以把数组映射为 JSX 元素列表。

```
<ul>
  { this.props.todos.map((todo, i) => <li key={i}>{todo}</li>) }
</ul>
```

注释

尽量别用 `//` 做单行注释。

```
<h1>
  {/* multiline comment */}
  {/*
    multi
    line
    comment
    */}
  {
    // single line
  }
  Hello
</h1>
```

Spread Attributes

这是 JSX 从 ECMAScript6 借鉴过来的很有用的特性，用于扩充组件 props 。

比如：

```
const attrs = {
  href: 'http://example.org',
  target: '_blank',
};
<a {...attrs}>Hello</a>
```

等同于

```
const attrs = {
  href: 'http://example.org',
  target: '_blank',
};
<a href={attrs.href} target={attrs.target}>Hello</a>
```

Props

数据处理在 React 中是非常重要的概念之一，分别可以通过 props, state 和 context 来处理数据。而在 dva 应用里，你只需关心 props 。

propTypes

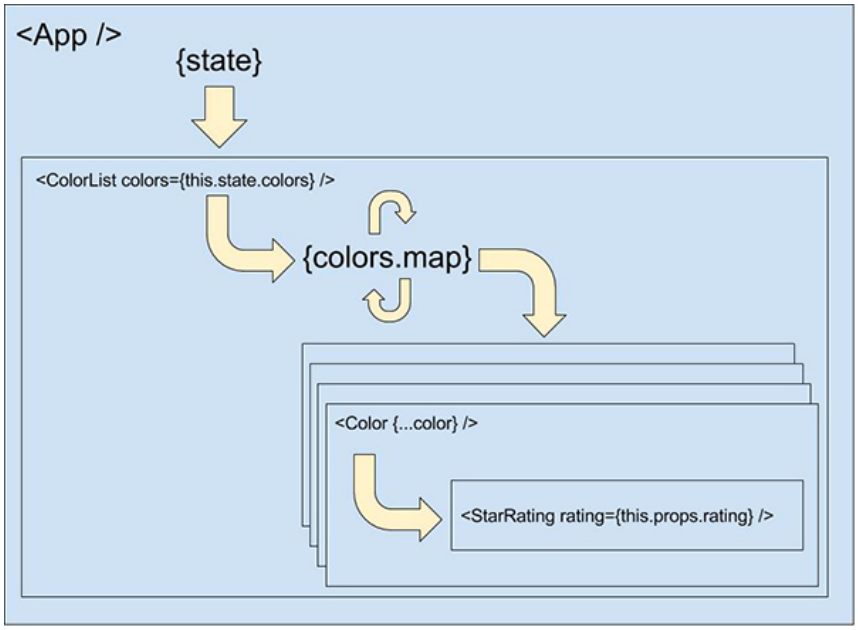
JavaScript 是弱类型语言，所以请尽量声明 propTypes 对 props 进行校验，以减少不必要的问题。

```
function App(props) {
  return <div>{props.name}</div>;
}
App.propTypes = {
  name: React.PropTypes.string.isRequired,
};
```

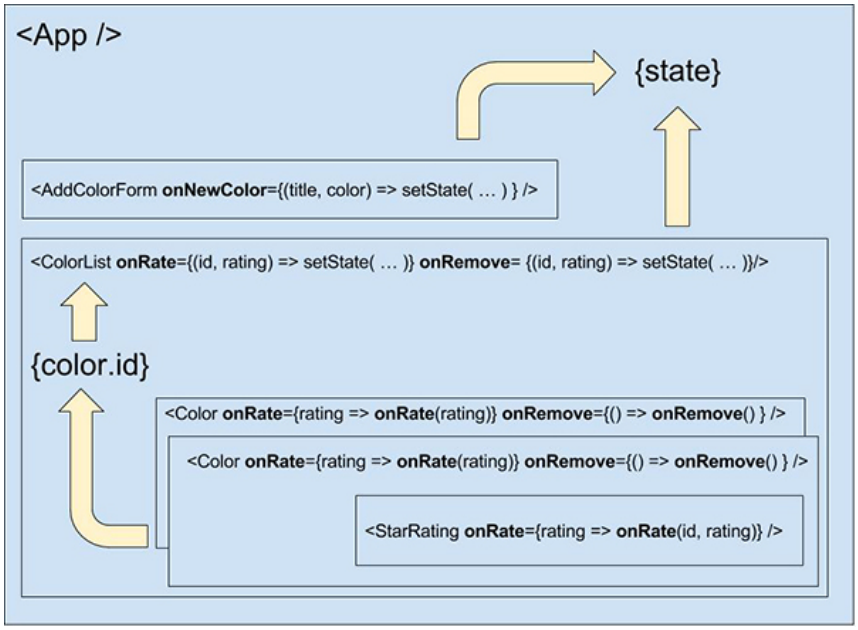
内置的 prop type 有：

- PropTypes.array
- PropTypes.bool
- PropTypes.func
- PropTypes.number
- PropTypes.object
- PropTypes.string

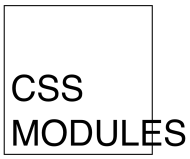
往下传数据



往上传数据

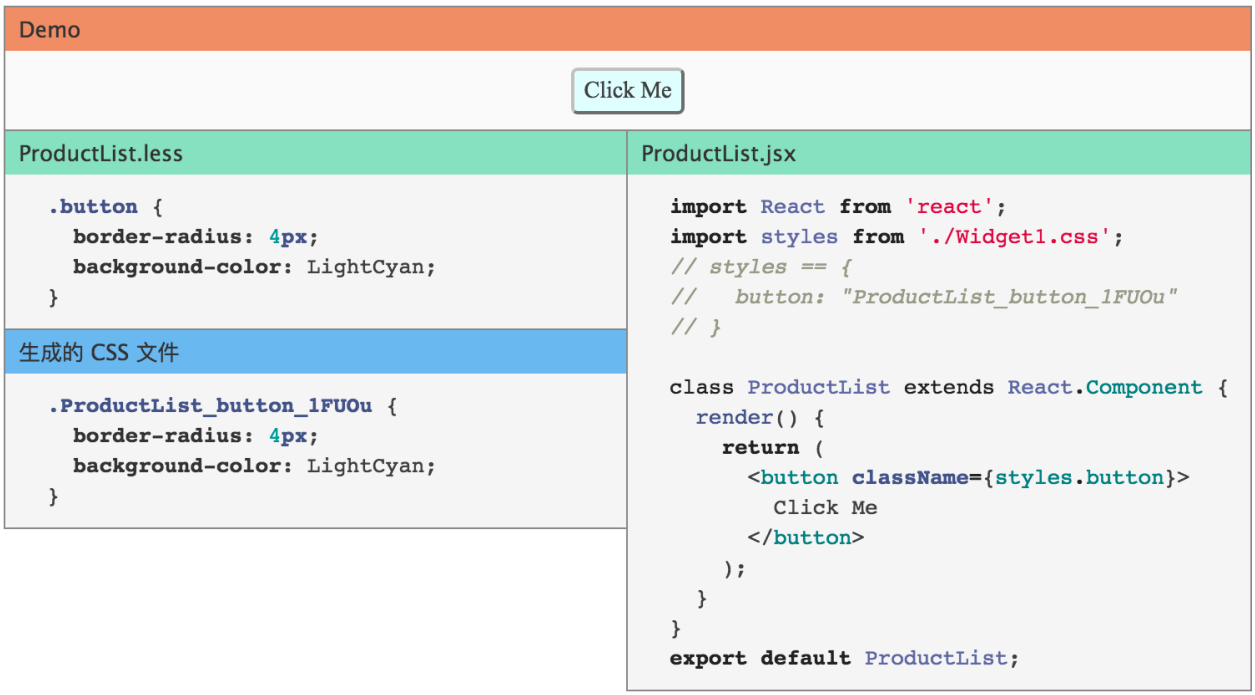


CSS Modules



理解 **CSS Modules**

一张图理解 CSS Modules 的工作原理：



`button` class 在构建之后会被重命名为 `ProductList_button_1FU0u` 。`button` 是 local name，而 `ProductList_button_1FU0u` 是 global name 。你可以用简短的描述性名字，而不需要关心命名冲突问题。

然后你要做的全部事情就是在 css/less 文件里写 `.button {...}`，并在组件里通过 `styles.button` 来引用他。

定义全局 **CSS**

CSS Modules 默认是局部作用域的，想要声明一个全局规则，可用 `:global` 语法。

比如：

```
.title {
  color: red;
}
:global(.title) {
  color: green;
}
```

然后在引用的时候：

```
<App className={styles.title} /> // red
<App className="title" />        // green
```

`classnames` Package

在一些复杂的场景中，一个元素可能对应多个 className，而每个 className 又基于一些条件来决定是否出现。这时，classnames 这个库就非常有用。

```
import classnames from 'classnames';
const App = (props) => {
  const cls = classnames({
    btn: true,
    btnLarge: props.type === 'submit',
    btnSmall: props.type === 'edit',
  });
  return <div className={ cls } />;
}
```

这样，传入不同的 type 给 App 组件，就会返回不同的 className 组合：

```
<App type="submit" /> // btn btnLarge
<App type="edit" />   // btn btnSmall
```

Reducer

reducer 是一个函数，接受 state 和 action，返回老的或新的 state 。即：`(state, action) => state`

增删改

以 todos 为例。

```
app.model({
  namespace: 'todos',
  state: [],
  reducers: {
    add(state, { payload: todo }) {
      return state.concat(todo);
    },
    remove(state, { payload: id }) {
      return state.filter(todo => todo.id !== id);
    },
    update(state, { payload: updatedTodo }) {
      return state.map(todo => {
        if (todo.id === updatedTodo.id) {
          return { ...todo, ...updatedTodo };
        } else {
          return todo;
        }
      });
    },
  },
};
```

嵌套数据的增删改

建议最多一层嵌套，以保持 state 的扁平化，深层嵌套会让 reducer 很难写和难以维护。

```
app.model({
  namespace: 'app',
  state: {
    todos: [],
    loading: false,
  },
  reducers: {
    add(state, { payload: todo }) {
      const todos = state.todos.concat(todo);
      return { ...state, todos };
    },
  },
});
```

下面是深层嵌套的例子，应尽量避免。

```
app.model({
  namespace: 'app',
  state: {
    a: {
      b: {
        todos: [],
        loading: false,
      },
    },
  },
  reducers: {
    add(state, { payload: todo }) {
      const todos = state.a.b.todos.concat(todo);
      const b = { ...state.a.b, todos };
      const a = { ...state.a, b };
      return { ...state, a };
    },
  },
});
```

Effect

示例：

```
app.model({
  namespace: 'todos',
  effects: {
    *addRemote({ payload: todo }, { put, call }) {
      yield call(addTodo, todo);
      yield put({ type: 'add', payload: todo });
    },
  },
});
```

Effects

put

用于触发 action 。

```
yield put({ type: 'todos/add', payload: 'Learn Dva' });
```

call

用于调用异步逻辑，支持 promise 。

```
const result = yield call(fetch, '/todos');
```

select

用于从 state 里获取数据。

```
const todos = yield select(state => state.todos);
```

错误处理

全局错误处理

dva 里，effects 和 subscriptions 的抛错全部会走 `onError` hook，所以可以在 `onError` 里统一处理错误。

```
const app = dva({
  onError(e, dispatch) {
    console.log(e.message);
  },
});
```

然后 effects 里的抛错和 reject 的 promise 就都会被捕获到了。

本地错误处理

如果需要对某些 effects 的错误进行特殊处理，需要在 effect 内部加 `try catch` 。

```
app.model({
  effects: {
    *addRemote() {
      try {
        // Your Code Here
      } catch(e) {
        console.log(e.message);
      }
    },
  },
});
```

异步请求

异步请求基于 whatwg-fetch，API 详见：https://github.com/github/fetch

GET 和 POST

```
import request from '../util/request';

// GET
request('/api/todos');

// POST
request('/api/todos', {
  method: 'POST',
  body: JSON.stringify({ a: 1 }),
});
```

统一错误处理

假如约定后台返回以下格式时，做统一的错误处理。

```
{
  status: 'error',
  message: '',
}
```

编辑 `utils/request.js`，加入以下中间件：

```
function parseErrorMessage({ data }) {
  const { status, message } = data;
```

```
  if (status === 'error') {
    throw new Error(message);
  }
  return { data };
}
```

然后，这类错误就会走到 `onError` hook 里。

Subscription

`subscriptions` 是订阅，用于订阅一个数据源，然后根据需要 dispatch 相应的 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。格式为 `({ dispatch, history }) => unsubscribe` 。

异步数据初始化

比如：当用户进入 `/users` 页面时，触发 action `users/fetch` 加载用户数据。

```
app.model({
  subscriptions: {
    setup({ dispatch, history }) {
      history.listen(({ pathname }) => {
        if (pathname === '/users') {
          dispatch({
            type: 'users/fetch',
          });
        }
      });
    },
  },
});
```

`path-to-regexp` Package

如果 url 规则比较复杂，比如 `/users/:userId/search`，那么匹配和 userId 的获取都会比较麻烦。这是推荐用 path-to-regexp 简化这部分逻辑。

```
import pathToRegexp from 'path-to-regexp';

// in subscription
const match = pathToRegexp('/users/:userId/search').exec(pathname);
if (match) {
  const userId = match[1];
  // dispatch action with userId
}
```

Router

Config with JSX Element (router.js)

```
<Route path="/" component={App}>
  <Route path="accounts" component={Accounts}/>
  <Route path="statements" component={Statements}/>
</Route>
```

详见：react-router

Route Components

Route Components 是指 `./src/routes/` 目录下的文件，他们是 `./src/router.js` 里匹配的 Component。

通过 **connect** 绑定数据

比如：

```
import { connect } from 'dva';
function App() {}

function mapStateToProps(state, ownProps) {
  return {
    users: state.users,
  };
}
export default connect(mapStateToProps)(App);
```

然后在 App 里就有了 `dispatch` 和 `users` 两个属性。

Injected Props (e.g. location)

Route Component 会有额外的 props 用以获取路由信息。

- location
- params
- children

更多详见：react-router

基于 action 进行页面跳转

```
import { routerRedux } from 'dva/router';

// Inside Effects
yield put(routerRedux.push('/logout'));

// Outside Effects
dispatch(routerRedux.push('/logout'));

// With query
routerRedux.push({
  pathname: '/logout',
  query: {
    page: 2,
  },
});
```

除 `push(location)` 外还有更多方法，详见 react-router-redux

dva 配置

Redux Middleware

比如要添加 redux-logger 中间件：

```
import createLogger from 'redux-logger';
const app = dva({
  onAction: createLogger(),
});
```

注：onAction 支持数组，可同时传入多个中间件。

history

切换 **history** 为 **browserHistory**

```
import { browserHistory } from 'dva/router';
const app = dva({
  history: browserHistory,
});
```

去除 **hashHistory** 下的 **_k** 查询参数

```
import { useRouterHistory } from 'dva/router';
import { createHashHistory } from 'history';
const app = dva({
  history: useRouterHistory(createHashHistory)({ queryKey: false }),
});
```

工具

通过 **dva-cli** 创建项目

先安装 dva-cli 。

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

然后创建项目。

```
$ dva new myapp
```

最后，进入目录并启动。

```
$ cd myapp
$ npm start
```