React数据流管理神器—Redux

2017年3月 无线南京 柳凯

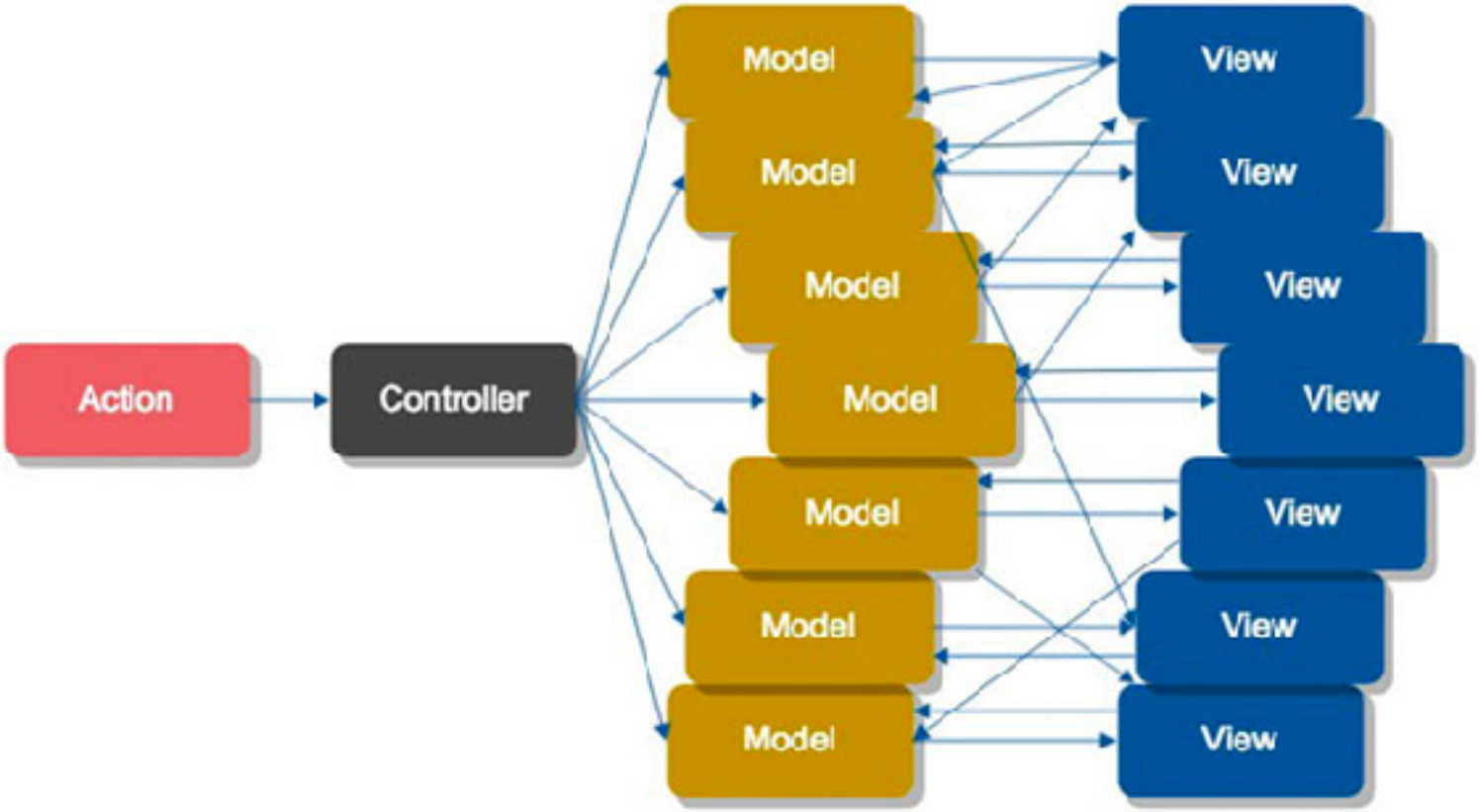
JD.京东.COM

双向数据流

在传统MVC前端框架，如angularjs, backbone中，通常使用双向绑定的方式来将Model的数据展现到View。当Model中的数据发生变化时，一个或多个View会发生变化；当View接受了用户输入时，Model中的数据则会发生变化。

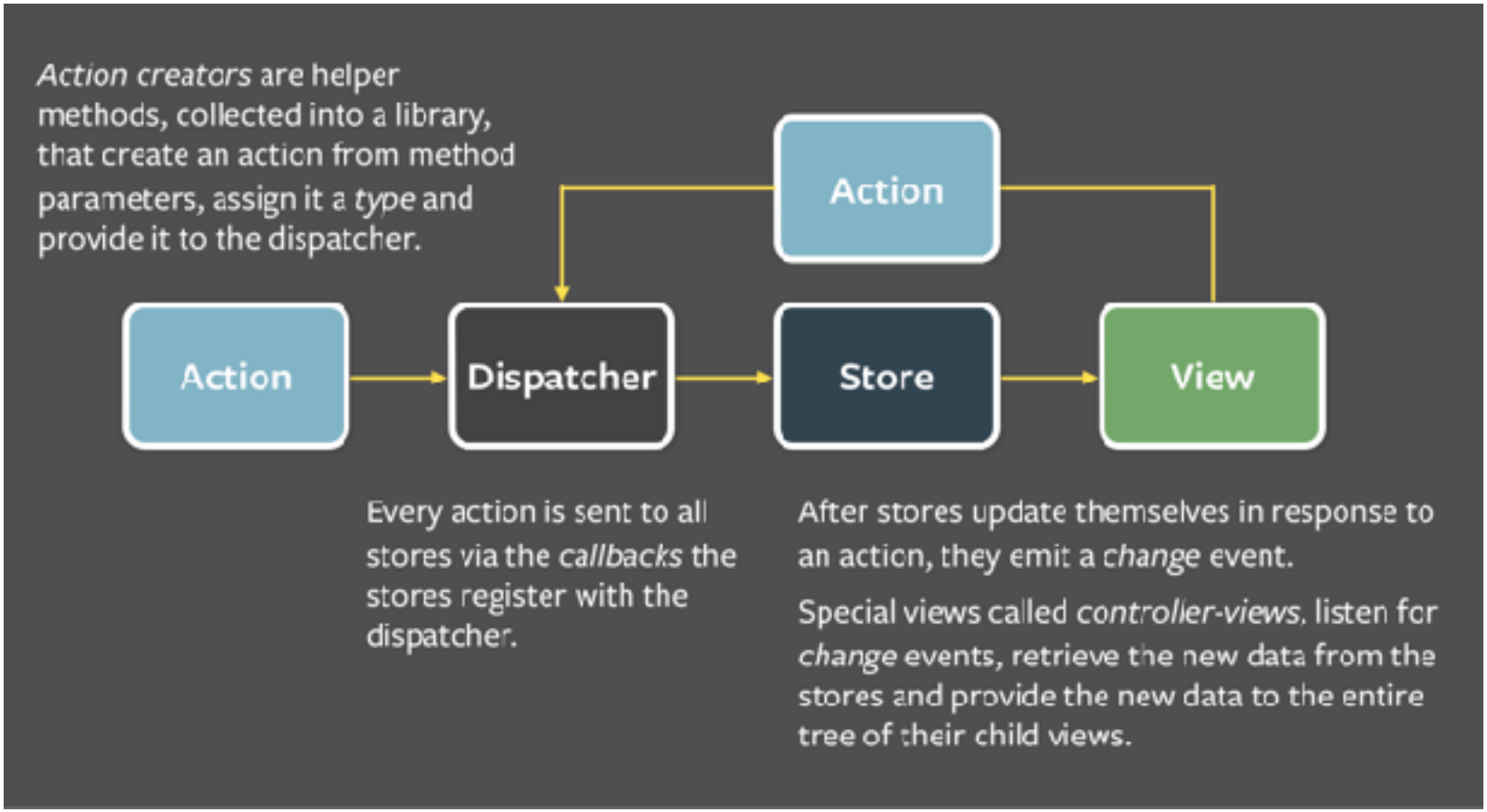
在实际的应用中，当一个Model中的数据发生变化时，也有可能另一个相关的Model中的数据会被同步更新。这样，很容易出现的一个现象就是连锁更新（Cascading Update），Model可以更新Model，Model可以更新View，View也可以更新Model。你很难去推断一个界面的变化究竟是由哪个局部的功能代码引起。

如下图所示， Model 和 View 之间的关系错综复杂，导致出现问题时很难调试；实现新功能时也需要时刻注意代码是否会产生副作用。



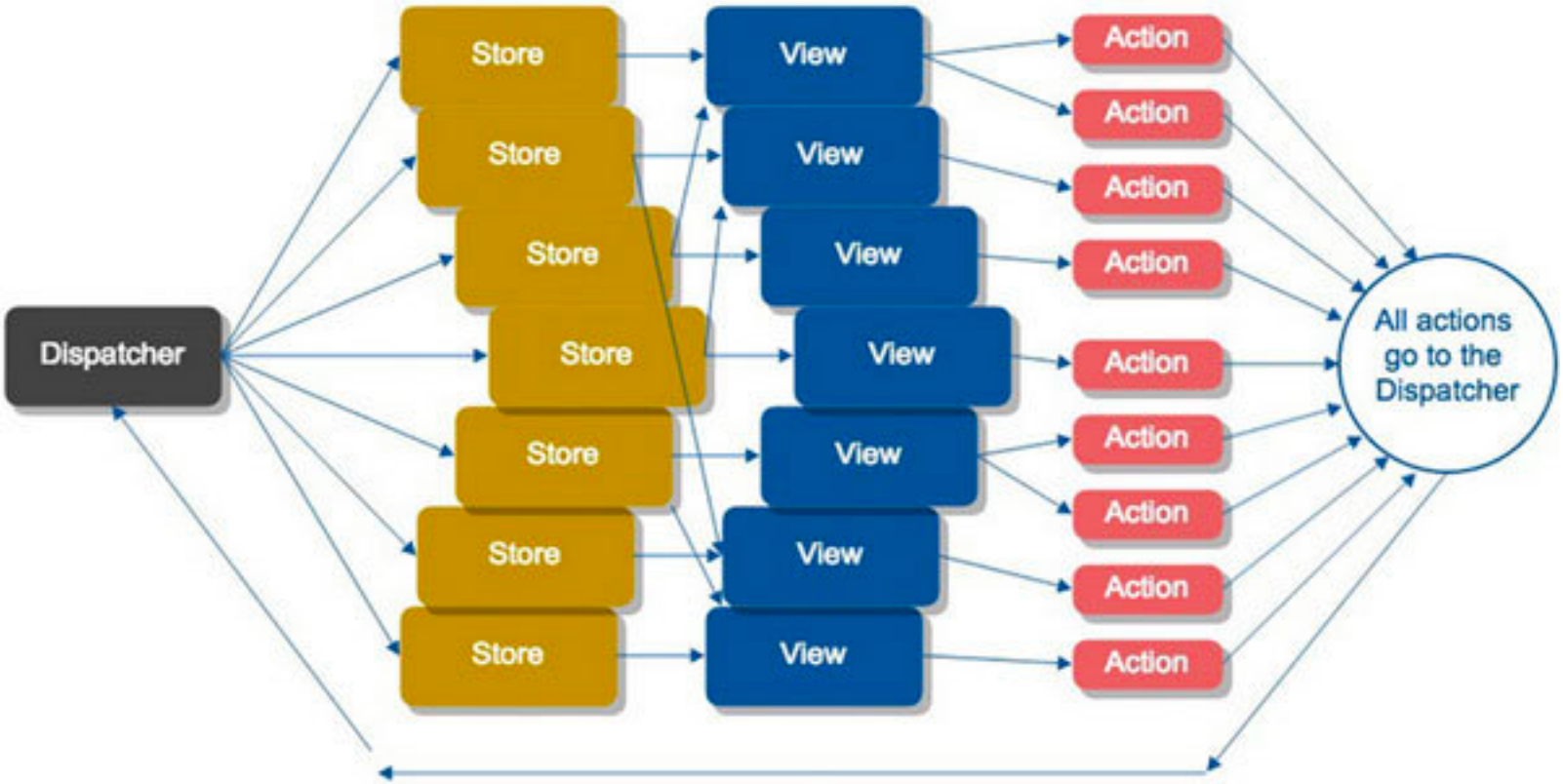
2013年，Fackbook推出来React来构建大型前端项目。React本身只是一个简单易用的UI库，其特性是数据驱动UI，但React本身并没有涉及到数据流的管理。

Flux 正是 FB 官方给出的管理React应用数据的应用架构，他推崇的是单向数据流动模式。



Flux中引入Store、Action、Action Creators和Dispatcher等概念来管理信息流。

相较于双向数据流，Flux在项目中应该是这样的



下面是一个Todo的案例，在这个界面上，需要实现添加、标记完成/未完成、删除、涮选等操作。组件的划分如下

Todo示例

JD.COM 京东

在这个Todo的例子中，如果不采用数据流管理框架而存粹在React组件内部管理数据，非常麻烦，比如在TodoItem中出发完成/未完成如何修改到TodoFooter。而采用Flux，就比较方便了。

```
// TodoApp.js
function getTodoState() {
  return {
    allTodos: TodoStore.getAll(),
    areAllComplete: TodoStore.areAllComplete()
  };
}

var TodoApp = React.createClass({
  getInitialState: function() {
    return getTodoState();
  },
  componentDidMount: function() {
    TodoStore.addChangeListener(this._onChange);
  },
  componentWillUnmount: function() {
    TodoStore.removeChangeListener(this._onChange);
  },
  render: function() {
    return ...
  },
  _onChange: function() {
    this.setState(getTodoState());
  }
});
```

```
// TodoItem.js
var TodoItem = React.createClass({
  render: function() {
    return (
      <div>
        <input type="text"
        value={this.state.text}
        onChange={this._onChange}
        onKeyUp={this._onKeyup}
        />
      </div>
    );
  },

  _onKeyup: function(e) {
    if (e.keyCode !== 13) {
      return;
    }
    TodoActions.create(this.state.text);
  }
  ...
});
```

Todo示例

JD.COM 京东

```
// TodoStore.js
var _todos = [];
var TodoStore = assign({},
EventEmitter.prototype, {
  getAll: function() {
    return _todos;
  },

  emitChange: function() {
    this.emit(CHANGE_EVENT);
  },

  addChangeListener: function(callback) {
    this.on(CHANGE_EVENT, callback);
  },

  removeChangeListener: function(callback)
{
    this.removeListener(CHANGE_EVENT,
callback);
  }
});

function create(text) {
  _todos.push({
    id: Date.now(),
    text: text,
    completed: false
  });
}
```

```
...
Dispatcher.register(function(action) {
  var text;
  switch(action.actionType) {
    case TodoConstants.TODO_CREATE:
      text = action.text.trim();
      if (text !== '') {
        create(text);
        TodoStore.emitChange();
      }
      ...
      break;
  }
});
```

```
// TodoActions.js
var TodoActions = {
  create: function(text) {
    Dispatcher.dispatch({
      actionType: TodoConstants.TODO_CREATE,
      text: text
    });
  },
  ...
}
```

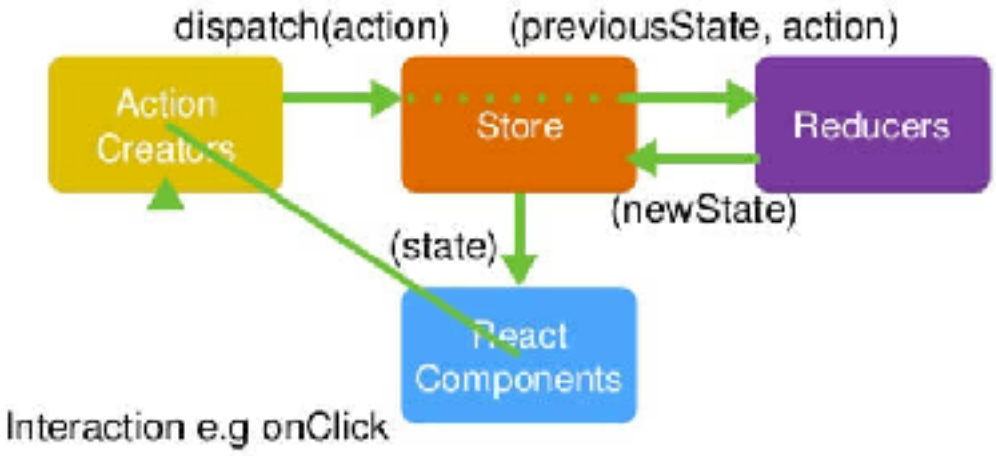
Redux 由 Flux 演变而来，但受 Elm的启发，避开了 Flux 的复杂性。

和 Flux 一样，Redux 让应用的状态变化变得更加可预测。如果你想改变应用的状态，就必须 dispatch 一个 action。对于 Flux 和 Redux 来说，这些概念都是相似的。

Redux是flux概念的子集，两者主要区别是：

- 应用中所有的 state 都以一个对象树的形式储存在一个**单一的** store 中。
- 你的应用状态是不可变的，惟一改变 state 的办法是触发action，一个描述发生什么的对象。
- Redux 没有 dispatcher 的概念，它依赖纯函数来替代事件处理器。
- Redux有代码热替换和时间旅行的功能

下图是Redux的数据流图：



Action：Action是把数据从应用传到 store 的有效载荷。它是 store 数据的唯一来源。一般来说你会通过 store.dispatch() 将 action 传到 store。

Reducer：reducer 就是一个纯函数，接收旧的 state 和 action，返回新的 state。

Store：Store 是把Action和Reducer联系到一起的对象。Store 有以下职责：

- 维持应用的 state；
- 提供 getState() 方法获取 state；
- 提供 dispatch(action) 方法更新 state；
- 通过 subscribe(listener) 注册监听器；
- 通过 subscribe(listener) 返回的函数注销监听器。

Redux 和 React 之间没有关系，它只是一个数据流管理框架，也可以用于Vue等其他框架。
尽管如此，Redux 还是和 React这类框架搭配起来用最好，因为这类框架允许你以 state 函数的形式来描述界面，Redux 通过 action 的形式来发起 state 变化。

在React中使用Redux时，我们可以使用React-Redux来做React和Redux之间的粘合，它帮我们屏蔽掉了很多底层细节，更方便开发。
比如说，使用React-Redux，你不需要手动调用store.subscribe(fn)来更新state，也不需要在componentWillUnmount中手动移除监听，在View中，你也不需要通过store.dispatch(action)来dispatch一个action，可以直接通过props来dispatch一个action

Todo示例

JD.COM 京东

```
// index.js
const store = createStore(reducer); // 创建store
render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)

// App.js
const App = ({todos, actions}) => (
  <div>
    <TodoHeader addTodo={actions.addTodo} />
    <TodoList todos={todos} actions={actions} />
    <TodoFooter todos={todos} actions={actions} />
  </div>
)
const mapStateToProps = state => ({
  todos: state.todos
})
const mapDispatchToProps = dispatch => ({
  actions: bindActionCreators(TodoActions, dispatch)
})
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(App)
```

```
// TodoItem.js
var TodoItem = React.createClass({
  render: function() {
    return (
      <div>
        <input type="text"
        value={this.state.text}
        onChange={this._onChange}
        onKeyUp={this._onKeyup}
        />
      </div>
    );
  },

  _onKeyup: function(e) {
    if (e.keyCode !== 13) {
      return;
    }
    this.props.addTodo(this.state.text);
  }
  ...
});
```

Todo示例

JD.COM 京东

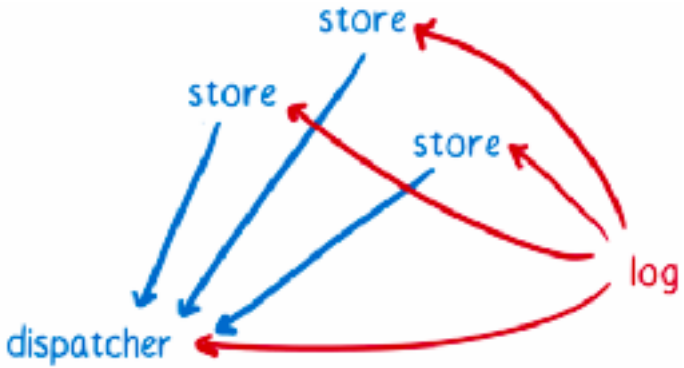
```
// TodoActions.js
import * as types from '../constants/ActionTypes'

export const addTodo = text => ({ type: types.ADD_TODO, text })
...

// Reducer.js
const initialState = []

export default function todos(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        {
          id: state.reduce((maxId, todo) => Math.max(todo.id, maxId), -1) + 1,
          completed: false,
          text: action.text
        },
        ...state
      ]
      ...
  }
}
```

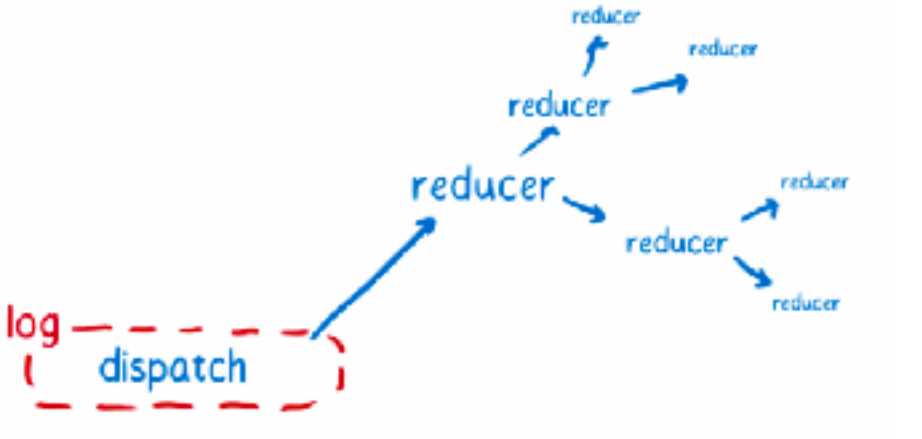
reducer是一个纯函数，不要在reducer中做有副作用的操作。

在一个复杂项目中，如果把所有的操作数据逻辑都写在一个函数中，必然造成这个函数的难以维护，我们可以把reducer拆分成很多小的子函数，通过combineReducers组合成一个rootReducer

在项目中，通常会有一些通用功能，比如说在每个action发起的前后打印state的值。
最简单的方式是在业务代码中直接插入打印log的代码，但这样侵入性太强，不利于以后的修改。在flux中，这样的日志功能不是一个第三方插件能够轻易实现的，实现该功能会侵入业务代码。

将这个架构的部分功能包装进其他的对象中将使得我们的需求变得更容易实现。这些「其他对象」在架构原有的功能基础之上添加了自己的功能。你可以把这种扩展点看做是一个增强器或者高阶对象，亦或者中间件。Redux可以采用这种中间件的形式来扩展功能。





Redux中间件

JD.COM 京东

```
// logger.js
const logger = store => next => action => {
  console.group(action.type)
  console.info('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  console.groupEnd(action.type)
  return result;
}

// store.js
let store = createStore(
  rootReducer,
  applyMiddleware(
    logger,
  )
)
```

目前，npm上有大量的Redux中间件可直接使用：

- redux-thunk — 用最简单的方式搭建异步 action 构造器
- redux-logger — 记录所有 Redux action 和下一次 state 的日志
- redux-saga — Redux 应用的另一种副作用 model
- redux-promise — 遵从 FSA 标准的 promise 中间件
- redux-observable — Redux 的 RxJS 中间件
- redux-rx — 给 Redux 用的 RxJS 工具，包括观察变量的中间件
- redux-immutable-state-invariant — 开发中的状态变更提醒
- redux-axios-middleware — 使用 axios HTTP 客户端获取数据的 Redux 中间件

……

异步Action

JD.COM 京东

在一个应用中，一般都会有和后台的交互，此时需要用到ajax来请求数据，ajax是一个异步的操作，在Redux中如何处理这种异步操作呢?

当调用异步 API 时，有两个非常关键的时刻：发起请求的时刻，和接收到响应的时刻 。这两个时刻都可能会更改应用的 state；为此，你需要 dispatch 普通的同步 action。一般情况下，每个 API 请求都需要 dispatch 至少三种 action：

- 一种通知 reducer 请求开始的 action。
- 一种通知 reducer 请求成功的 action。
- 一种通知 reducer 请求失败的 action。

上面三个action的同步方式是这样的

```
export const addTodo = text => ({ type: types.ADD_TODO_STSRT, text })
export const addTodoSuccess = () => ({ type: types.ADD_TODO_SUCCESS })
export const addTodoFailed = () => ({ type: types.ADD_TODO_FAILED })
```

异步Action

JD.COM 京东

如何把之前定义的同步 action 创建函数和网络请求结合起来呢？标准的做法是使用 redux-thunk middleware。要引入 redux-thunk 这个专门的库才能使用。
通过使用redux-thunk，action创建函数除了返回 action 对象外还可以返回函数。这时，这个 action 创建函数就成为了 thunk。

```
export const addTodoSuccess = () => ({ type: types.ADD_TODO_SUCCESS })
export const addTodoFailed = () => ({ type: types.ADD_TODO_FAILED })
export const addTodo = text => {
  return (dispatch, getState) {
    dispatch({ type: types.ADD_TODO_STSRT, text });
    fetch('/api/v1/add', {text})
      .then(response => response.json())
      .then(() => {
        dispatch(addTodoSuccess);
      }, () => {
        dispatch(addTodoFailed);
      })
  }
}
```

Thanks!