Basics of Redux

Kirill Sukhomlin, kirill_sukhomlin@epam.com

Plan

1. Redux API: Actions, Reducers, Store
2. Redux Data Flow
3. Middleware
4. React applications architecture: "Container & Presentational Components"
5. React-Redux
6. Useful links & questions

Redux v1.0.0

14th of August, 2015

«Love what you're doing with Redux!»
Jing Chen，Flux

«I asked for comments on Redux in FB's internal JS discussion group, and it was universally praised. Really awesome work.»
Bill Fisher，Flux

What is Redux

- The library provides tools for **state management**
- Redux is used with many popular JS-frameworks, including Angular 1 & 2, Vue.js
- The size of Redux is only 2kB, including dependencies

Guidelines

1. Single source of truth of application state – *store*
2. The only way to change state is *action*
3. Functions, which actually change state aka *reducers* should be pure

Redux API

```
import {
    createStore,
    combineReducers,
    bindActionCreators,
    applyMiddleware,
    compose
} from 'redux'

const Store = createStore(/*...*/)

const {getState, dispatch, subscribe, replaceReducer} = Store
```

Redux Store example

```
import {createStore} from 'redux'

const reducer = (state = 1, {type, payload}) => {
    switch (type) {
        case 'INCREMENT':
            return state + 1;

        case 'DECREMENT':
            return state - 1;

        case 'CHANGE_COUNTER':
            return state + payload;

        default:
            return state;
    }
}
const Store = createStore(reducer); // Store.getState() => 1
Store.dispatch({type: 'INCREMENT'}); // Store.getState() => 2
Store.dispatch({type: 'INCREMENT'}); // Store.getState() => 3
Store.dispatch({type: 'DECREMENT'}); // Store.getState() => 2
Store.dispatch({type: 'CHANGE_COUNTER', payload: 20}); // Store.getState() => 22
```

Actions

1. **Actions** are a way to communicate information from app to the Store
2. **Actions** are plain Javascript objects
3. The only required property of an **Action** is `type`, which is type of an action

```
const isPlainObject = require('lodash/isPlainObject');
isPlainObject(action); // => true
action.hasOwnProperty('type'); // => type
```

Flux Standart Action (FSA)

```
// typescript
type Action = {
    type: string|symbol,
    payload?: any,
    meta?: any,
    error?: boolean
}
```

Actions creators

```
const change = val => ({
    type: 'CHANGE_COUNTER',
    payload: val
})

const increment = change(1)
const decrement = change(-1)
```

Bound action creator

```
const Store = createStore(/*...*/)
const {dispatch} = Store;
const bindChangeToDispatch = dispatch => val => dispatch(change(val))
const boundChange = bindChangeToDispatch(dispatch)
const boundIncrement = bindChangeToDispatch(dispatch)(1)

// somewhere in code, as on click handler, for example
Store.dispatch(increment())
boundIncrement()
```

Reducers

1. Reducers process actions
2. Reducers have to be **pure** functions

```
const reducer = (prevState, action) => nextState

//looks like
reduce((acc, val) => acc)
```

An example of check for absence of side-effects

```
'use strict'
// https://www.npmjs.com/package/deepfreeze
const deepfreeze = require('deepfreeze')

const initialState = {/*...*/}
const actionCreator = (...args) => {
    // return some action object
}

const reducer = (state, action) => {
    // do something
}

deepfreeze(initialState)

expect(reducer(initialState, actionCreator(...params))).to.not.throw
```

Combining reducers

```
import {createStore, combineReducers} from 'redux'

const stateSchema = {
    user: {
        name: 'string', lastname: 'string'
    },
    mail: {
        letter: 'any[]'
    }
}

const userReducer = (state, {type, payload}) => {
    /*
    state = { name: 'string', lastname: 'string' }
    */
    return state
}

const mailReducer = (state, {type, payload}) => {
    /*
    state = { letter: 'any[]' }
    */
    return state
}

const Store = createStore(combineReducers({user: userReducer, mail: mailReducer}))
```

Store

1. Manages application data
2. Can be used to read the stata back: `Store.getState()`
3. Updates application data with actions `Store.dispatch(action)`
4. Adds and removes handlers

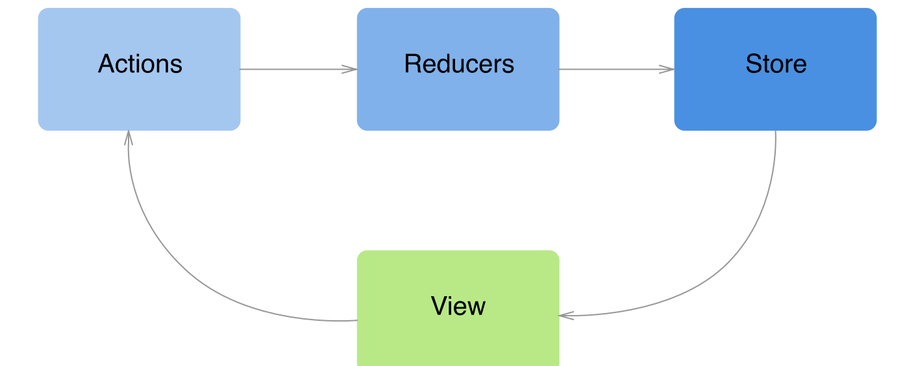
```
const Store = createStore(/*...*/)
const handler = function() {/*...*/}

const unsubscribe = Store.subscribe(handler)

// somewhere later

unsubscribe()
```

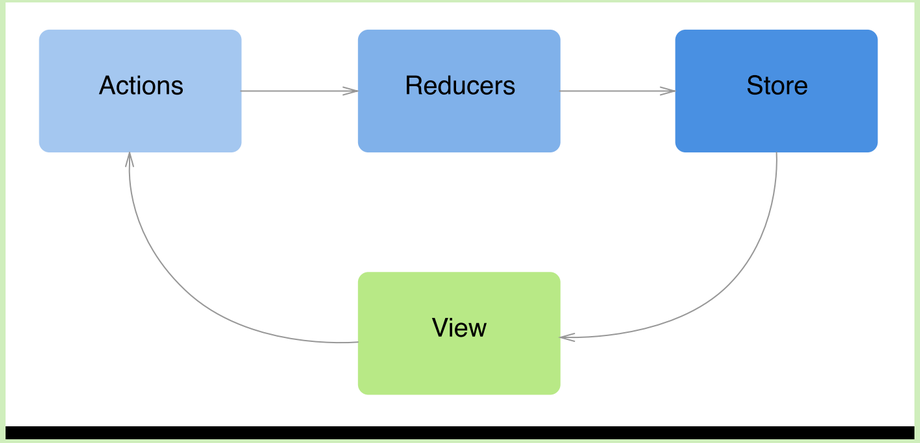
Redux data flow



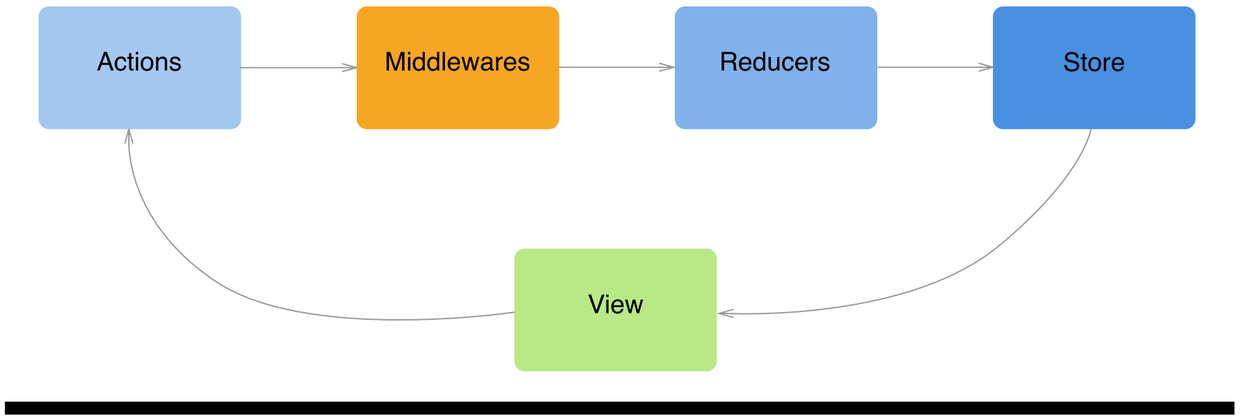
A simple app example

Side-effects

Actions are plain objects and **Reducers** are pure functions which return new state — side effects like ajax requests are not allowed



Middlewares



Middleware is a function, which take existing Store, next chained middleware and an action.

Middleware ain't ought to be pure (and usually don't)

Middleware code examples

```
// logging
const logMiddleware = store => next => action => {
    const {type, ...data} = action;
    console.group(type);
    console.log(JSON.stringify(data));
    console.groupEnd();
    next(action);
};

// async middleware @see redux-thunk
const thunkMiddlware = store => next => action => {
    if (typeof action === 'function') {
        action(store.dispatch);
    } else {
        next(action);
    }
}

// action sample for this middleware
const requestAction = url => dispatch => {
    fetch(`api/${url}`)
        .then(response => dispatch(setResponseAction(response)))
        .catch(err => dispatch(displayErrorAction(err)))
}
```

Off-the-shelf middlewares

1. Redux-logger — logger
2. Redux-thunk — simple side-effects
3. Redux-saga — complext async side effects
4. Redux-debounce — debounce actions
5. Redux-promise — support of async actions `{type: 'any_name', payload: Promise}`

...

Redux API recap

```
import {
    createStore,
    combineReducers,
    bindActionCreators,
    applyMiddleware,
    compose
} from 'redux'

const Store = createStore(/*...*/)

const {getState, dispatch, subscribe, replaceReducer} = Store
```

Redux API recap #2

1. `createStore(reducer, initialState, enhancer)`, initialStore is either defined here or as a default value for reducers,
   `function mail(state = [], action)`
2. `combineReducers(reducers)`, `reducers: {A: reducerA, B: reducerB}`
3. `bindActionCreators(actionCreators, dispatch)`, actionCreators is either function or object `{key => actionCreator}`
4. `applyMiddleware(...middlewares)`, Redux middleware API-compatible functions `({getState, dispatch}) => next => action`
5. or `compose(...function)` — bare function compositions

Application architecture: «Container Components, Presentational Components»

Presentational Components

1. Contains both Presentational and Container Components — often has only HTML/CSS layout
2. Never depend on app model: *actions, state...*
3. Usually are stateless
4. State only for UI-state (like `:hover`)

Container Components

1. Can contain both Presentational and Container Components, trivial renderer.
2. Depend on app model
3. Usually are statefull

Example

```
class UserInput extends React.Component {
    constructor(props) {
        super(props)
        this.state = {user: props.user}
    }
    componentDidMount() {
        fetch(`/api/getUser?id=${this.state.user.id}`)
            .then(response => this.setState({user: response.user}))
    }
    render() {
        return (
            <div className="input">
                <label className="input-label">{this.state.user.label}</label>
                <input className="input-field" type="text" value={this.state.user.name}/>
            </div>
        )
    }
}

//usage
<UserInput user={{id: 8877}}/>
```

Refactoring

```
// Presentational Component
function Input({label, value}) {
    return (<div className="input">
        <label className="input-label">{label}</label>
        <input className="input-field" type="text" value={value}/>
    </div>)
}

import CancelablePromise from 'cancelable-promise'

// Container Component
class UserInput extends React.Component {
    componentWillUnmount() {
        this.request.cancel()
    }
    componentDidMount() {
        this.request = new CancelablePromise(resolve => {
             fetch(`/api/getUser?id=${this.state.user.id}`)
                .then(response => resolve(response))
        })
        this.request.then(response => this.setState({user: response.user}))
    }
    render() {
        return <Input label={this.state.user.label} value={this.state.user.value}/>
    }
}
```

Alternative refactoring

```
// thunk-action
const requestUser = id => dispatch => {
    fetch(`/api/getUser?id=${id}`)
        .then(response => dispatch(setUser(response)))
}

// Presentational Component
function Input({label, value}) {
   return (
           <div className="input">
               <label className="input-label">{label}</label>
               <input className="input-field" type="text" value={value}/>
           </div>
       )
}

// Container Component
class UserInput extends React.Component {
    // or componentDidMount, or componentWillUpdate, or componentWillReceiveProps ...
    constructor(props) {
        super(props)
        requestUser(props.user.id)
    }
    render() {
        const {user: {value, label}} = this.props
        return value ? <SomeLoadingIndicator/> : <Input label={label} value={value}/>
    }
}
```

React-Redux

React-Redux API

1. `<Provider store>` passes store in descendants' context
2. `connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])`
   computes properties from Store
   - `mapStateToProps(state, [ownProps])` — function
   - `mapDispatchToProps(dispatch, [ownProps])` — object|function
   - `mergeProps(stateProps, dispatchProps, ownProps)` — Object.assign by default, resolves conflicts between own and computed properties

React-Redux API usage

```
import {Provider, connect} from 'react-redux';
import AppRootComponent from './app';
import {createStore} from 'redux';

const Store = createStore(/*...*/)

// <Provider><AppRootComponent /><Provider>

# somewhere
@connect(
    state => ({
        users: state.users
    }),
    (dispatch, props) => ({
        onClick() {
            dispatch(someActionCreator(props))
        }
    })
)
class Users extends React.Component {}
// OR
const UserContainer = connect(
    state => ({ users: state.users }),
    (dispatch, props) => ({onClick() {dispatch(someActionCreator(props))}})
)(User)
```

Selectors

- computed properties
- might be cached

Usage of selectors

```
import {createSelector, defaultMemoize} from 'reselect'

const getUsers = state => state.users
const getLetters = state => state.letters

const getMailBoxes = createSelector(getUsers, getLetters, (users, letter) => {
    // filter, find, reduce, e.t.c
})

// selector are memoized functions
const _complexFunction = {/*...*/}
const complexFunction = defaultMemoize(_complexFunction /* [equalityCheck], === by default */)

// using with mapStateToProps
const mapStateToProps = state => ({
    users: getUsers(state),
    letters: getLetters(state),
    mail: getMailBoxes(state)
})
```

Counter example, refactoring

Links, utils

1. Redux, source, docs
2. React-redux, source, docs
3. Redux-ui, source, docs
4. Normalizr, source and API docs, test examples
5. Reselect, source and API docs

Links, articles

1. From official docs
    - video,
    - examples,
    - articles
2. Getting Started with Redux
3. Building React Applications with Idiomatic Redux
4. Overview: Redux without profanity, Redux-UI scetion
5. Container- and bare components, article On architecture React-apps
6. Examples from our lecture straightforward solution, with provider, react-redux integration

Thanks!