

# Redux

JavaScript 狀態容器，提供可預測化的狀態管理

# 目录

1. Redux 核心
2. React + Redux
3. Redux 中间件
4. Redux 常用中间件
5. Redux 综合案例

# Redux 介绍

# 1. Redux 核心

## 1.1 Redux 介绍

JavaScript 状态容器，提供可预测化的状态管理



```
const state = {  
  modelOpen: "yes",  
  btnClicked: "no",  
  btnActiveClass: 'active',  
  page: 5,  
  size: 10  
};
```

# 1. Redux 核心

## 1.2 获取 Redux

官网

CDN



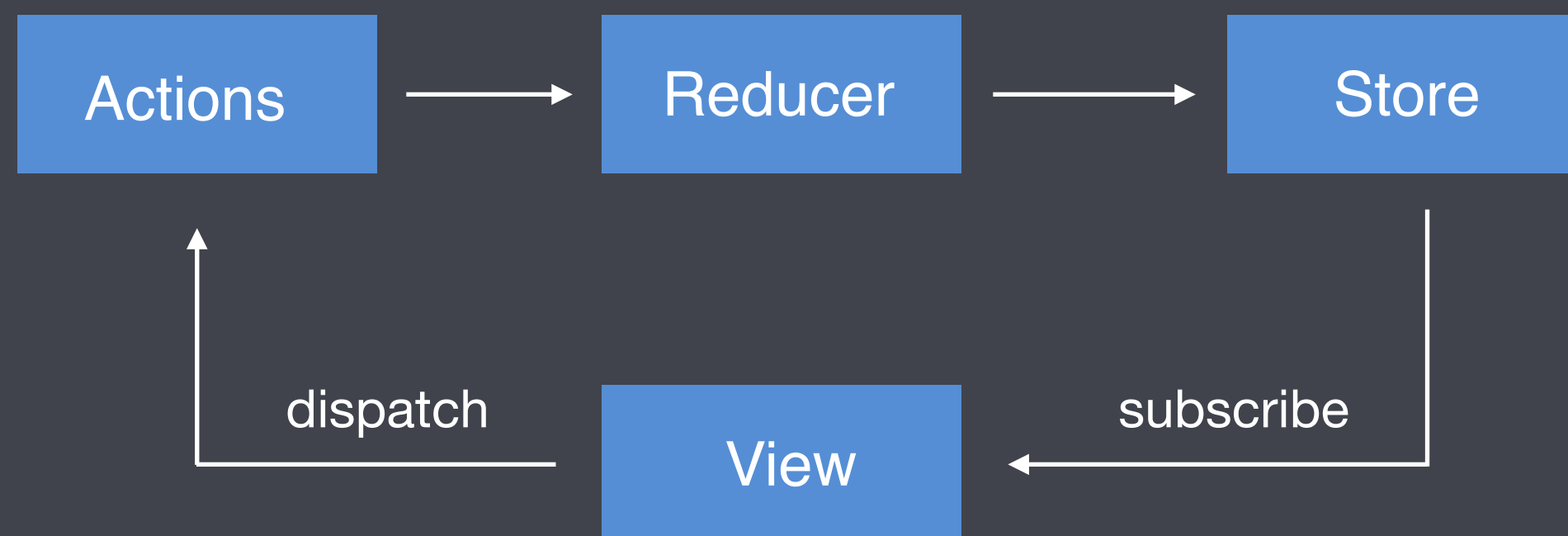
```
<script src="https://cdn.bootcss.com/redux/4.0.5/redux.min.js"></script>
```

# Redux 核心概念及工作流程

# 通过计数器案例学习 Redux

# 1. Redux 核心

## 1.3 Redux 核心概念及工作流程



Store: 存储状态的容器, JavaScript对象

View: 视图, HTML页面

Actions: 对象, 描述对状态进行怎样的操作

Reducers: 函数, 操作状态并返回新的状态



# Redux 核心 API 总结

# 1. Redux 核心

## 1.4 Redux 核心 API



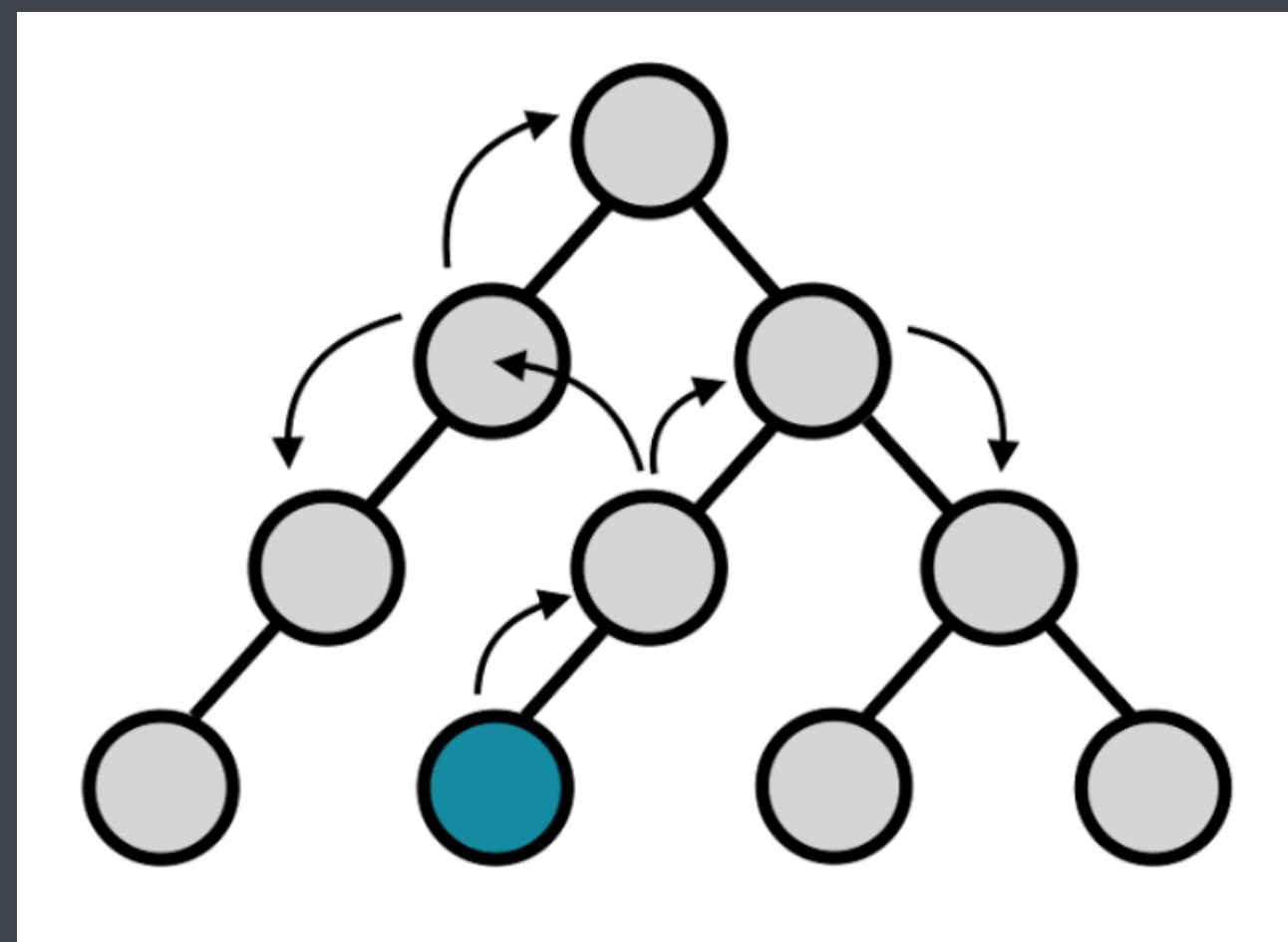
```
// 创建 Store 状态容器
const store = Redux.createStore(reducer);
// 创建用于处理状态的 reducer 函数
function reducer (state = initialState, action) {}
// 获取状态
store.getState();
// 订阅状态
store.subscribe(function () {});
// 触发Action
store.dispatch({type: 'description ... '});
```

# Redux 解决了什么问题

## 2. React + Redux

### 2.1 在 React 中不使用 Redux 时遇到的问题

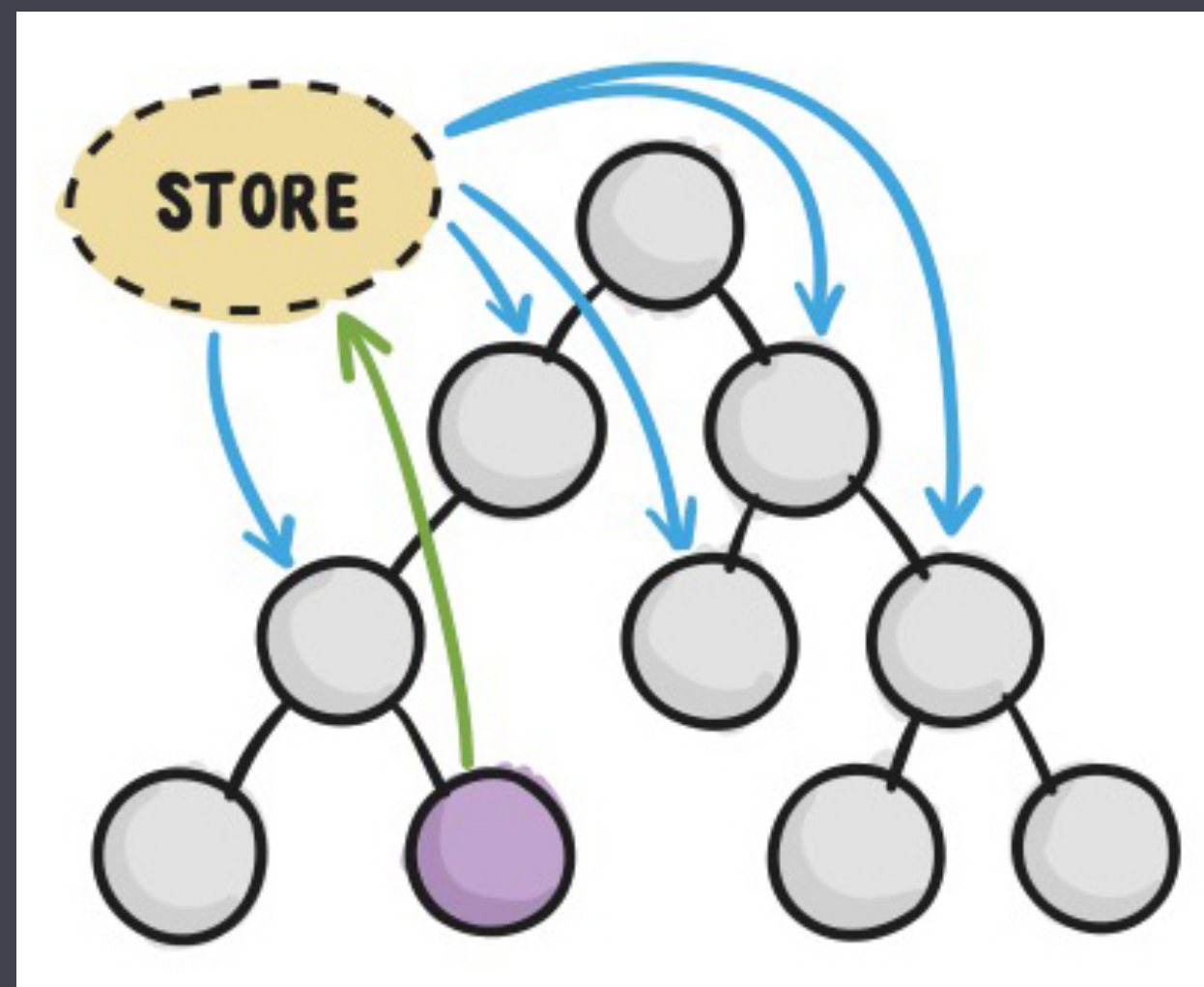
在React中组件通信的数据流是单向的, 顶层组件可以通过props属性向下层组件传递数据, 而下层组件不能向上层组件传递数据, 要实现下层组件修改数据, 需要上层组件传递修改数据的方法到下层组件. 当项目越来越大的时候, 组件之间传递数据变得越来越困难.



# 2. React + Redux

## 2.2 在 React 项目中加入 Redux 的好处

使用Redux管理数据，由于Store独立于组件，使得数据管理独立于组件，解决了组件与组件之间传递数据困难的问题。



# 在 React 中使用 Redux

# 2. React + Redux

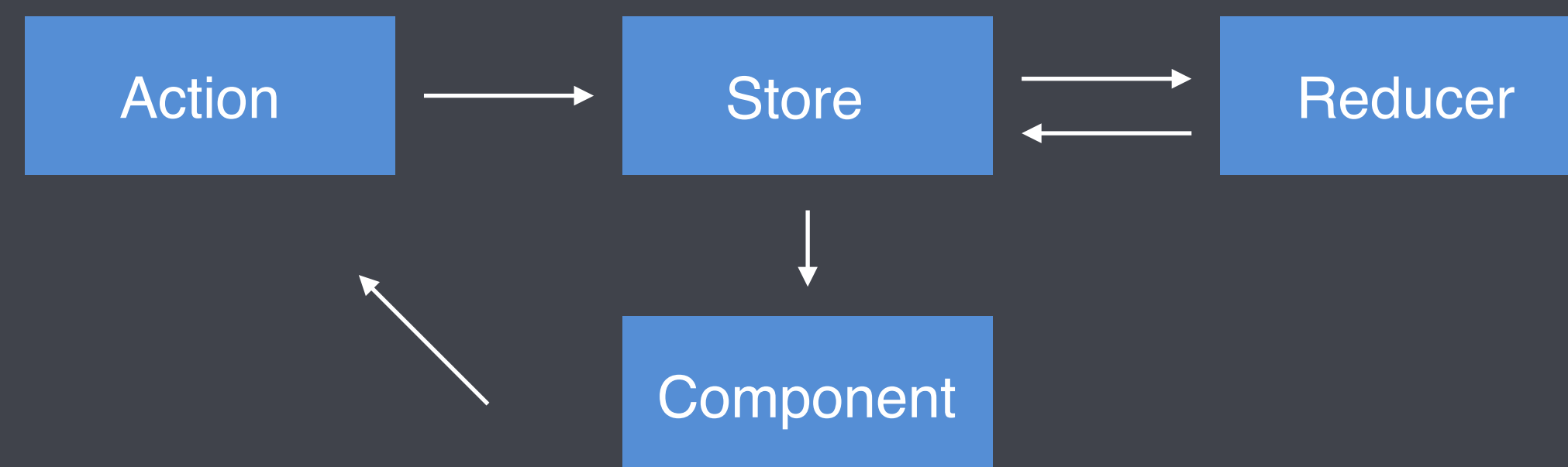
## 2.3 下载 Redux

```
npm install redux react-redux
```

# 2. React + Redux

## 2.4 Redux 工作流程

1. 组件通过 dispatch 方法触发 Action
2. Store 接收 Action 并将 Action 分发给 Reducer
3. Reducer 根据 Action 类型对状态进行更改并将更改后的状态返回给 Store
4. 组件订阅了Store中的状态，Store中的状态更新会同步到组件





# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.1 创建 Store 和 Reducer

1. 创建 Store 需要使用 createStore 方法, 方法执行后的返回值就是Store, 方法需要从 redux 中引入
2. createStore 方法的第一个参数需要传递reducer
3. reducer 是一个函数, 函数返回什么, store中就存储什么. 函数名称自定义.



```
import { createStore } from "redux";
const store = createStore(reducer);
function reducer() {
  return { count: 1 };
}
```

# Provider 组件与 connect 方法

# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.2 组件获取Store中的数据

1. 将store中的数据放在Provider组件中, Provider组件是存储共享数据的地方



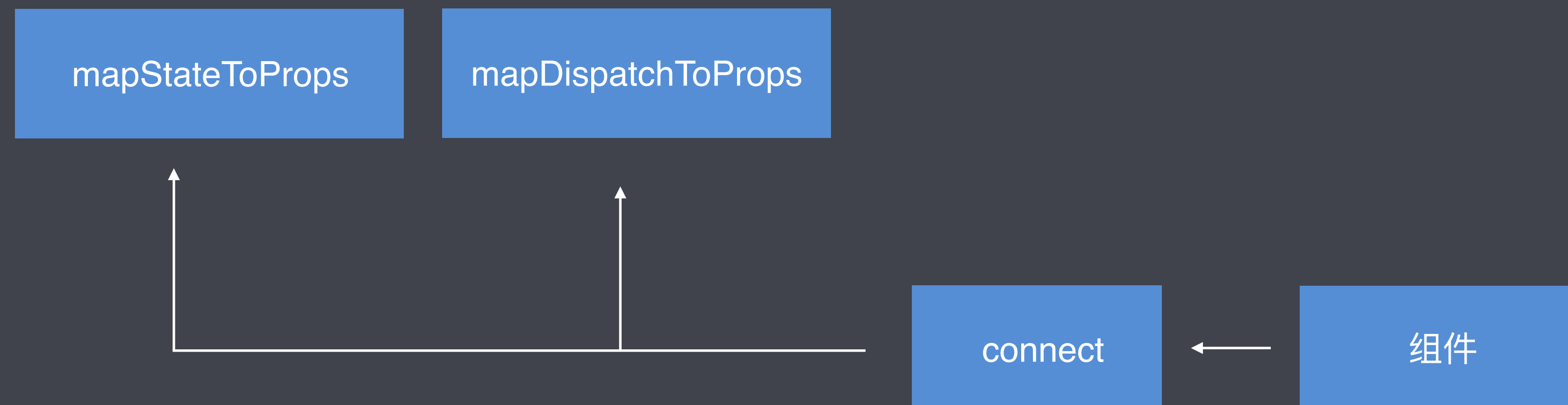
```
import { Provider } from 'react-redux';  
ReactDOM.render(<Provider store={store}><App /></Provider>);
```

# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.2 组件获取Store中的数据

2. 组件使用connect方法获取数据并将数据通过props传递进组件



# connect 方法的第二个参数

# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.2 组件获取Store中的数据

2. 组件使用connect方法获取数据并将数据通过props传递进组件



```
import { connect } from 'react-redux';  
const mapStateToProps = state => ({  
  count: state.count  
});  
export default connect(mapStateToProps)(组件名称);
```

# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.3 组件更改Store中的数据

#### 1. 定义 action



```
{type: '描述对数据要进行什么样的操作'}
```

action是改变状态的唯一途径

# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.3 组件更改Store中的数据

#### 2. 组件触发 action



```
this.props.dispatch({ type: '描述对数据进行设么样的操作' })
```



# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.3 组件更改Store中的数据

3. reducer 接收 action, 针对action对数据进行处理并返回



```
const initialState = { count: 0 };
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case "描述对数据进行设么样的操作":
      return { count: state.count + 1 };
    default:
      return state;
  }
};
```

# Action 传递参数

# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.4 为 action 传递参数

#### 1. 传递参数



```
<button onClick={() => increment(5)}>+ 1</button>
```

# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.4 为 action 传递参数

#### 2. 接收参数, 传递reducer



```
export const increment = payload => ({type: INCREMENT, payload});
```

# 2. React + Redux

## 2.5 Redux 使用步骤

### 2.5.4 为 action 传递参数

3. reducer根据接收到的数据进行处理



```
export default (state, action) => {  
  switch (action.type) {  
    case INCREMENT:  
      return { count: state.count + action.payload };  
  }  
};
```

# 2. React + Redux

## 2.6 优化

### 2.6.1 使用 Action Creator 函数将触发Action的代码独立成函数

在组件中通过调用`this.props.dispatch({type: '描述对数据进行设么样的操作'})`方法触发action. 造成HTML模板在视觉上的混乱.



```
const { increment, decrement } = this.props;  
<button onClick={increment}>+1</button>  
<button onClick={decrement}>-1</button>
```

# bindActionCreators 方法

# 2. React + Redux

## 2.6 优化

### 2.6.1 使用 Action Creator 函数将触发Action的代码独立成函数

在组件中通过调用`this.props.dispatch({type: '描述对数据进行设么样的操作'})`方法触发action. 造成HTML模板在视觉上的混乱.



```
const mapDispatchToProps = dispatch => ({
  increment () {
    dispatch({type: 'increment count'})
  },
  decrement () {
    dispatch({type: 'increment count'})
  }
});
export default connect(mapStateToProps, mapDispatchToProps)(组件名称);
```



# 代码重构

# 2. React + Redux

## 2.6 优化

### 2.6.2 Action Creators 函数绑定

触发Action的函数, 内部代码重复率非常高, 所以React提供了方法帮我们生成这些函数, 代替开发者手写.



```
// store/actions/counter.actions.js
export const increment = () => ({type: 'increment count'})
export const decrement = () => ({type: 'decrement count'})
// 组件
import { bindActionCreators } from 'redux';
import * as counterActions from '../store/actions/counter.action';
const mapDispatchToProps = dispatch => ({
  ...bindActionCreators(counterActions, dispatch)
});
```

# 2. React + Redux

## 2.6 优化

### 2.6.3 将Action类型字符串独立成常量

Action类型字符串组件在触发Action时需要使用, Reducer在接收Action时也需要使用, 由于字符串不存在代码提示, 存在写错的风险, 所以要将它独立成常量.

# Redux 弹出框

# 拆分 Reducer

# 2. React + Redux

## 2.6 优化

### 2.6.4 拆分Reducer

当要管理的数据越来越多时, reducer中的代码将会变得越来越庞大.

React允许将一个大的reducer拆分成若干个小的reducer, 最后进行合并使用.



```
import { combineReducers } from 'redux';

export default combineReducers({
  counter: counterReducer,
  user: userReducer
});

/* { counter: {count: 0}, user: {name:'张三', age: 0} } */
```

# 中间件概念介绍

# 3. Redux 中间件

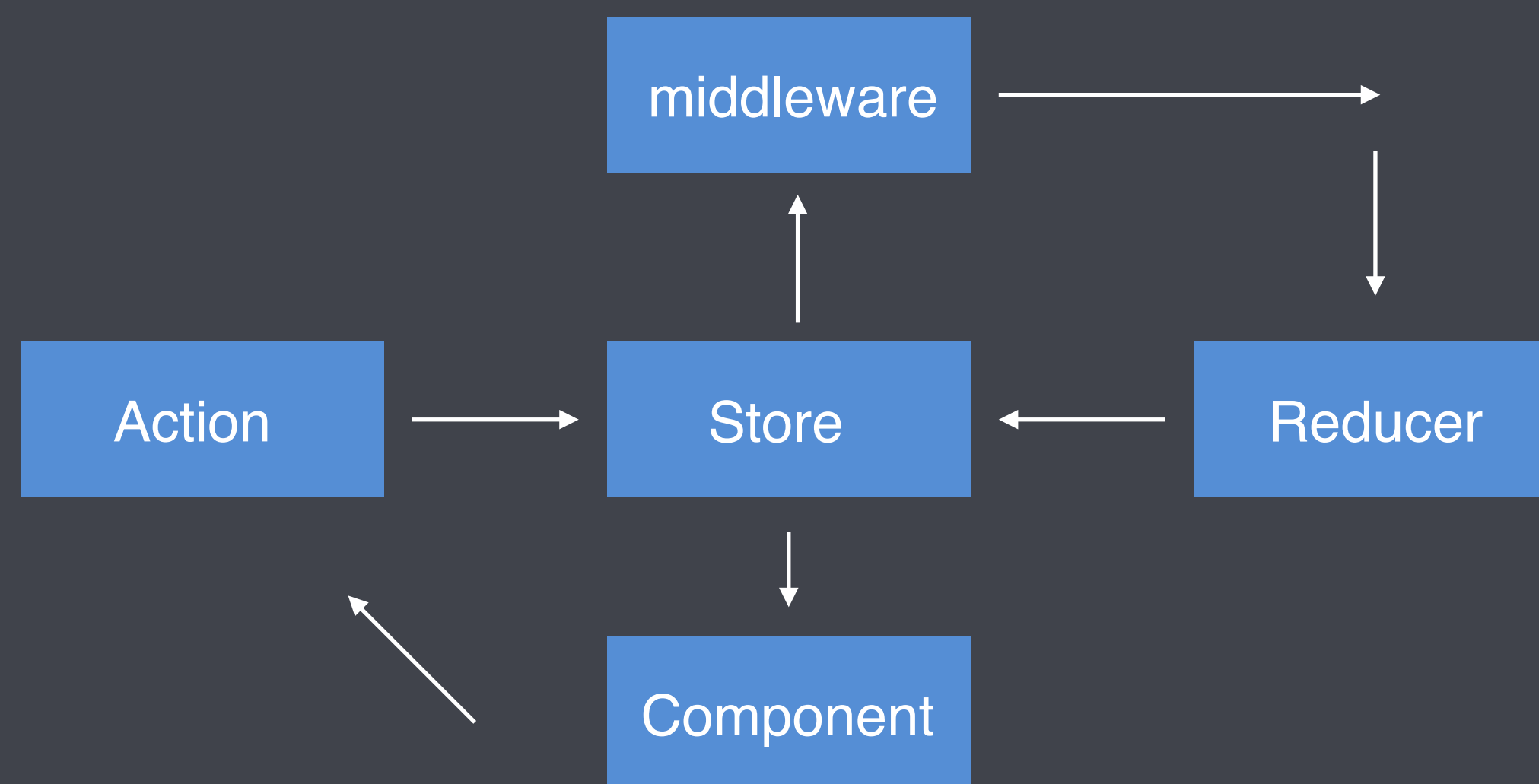
## 3.1 什么是中间件？

中间件允许我们扩展redux应用程序。



# 3. Redux 中间件

## 3.2 加入了中间件 Redux 工作流程



# 开发 Redux 中间件

# 3. Redux 中间件

## 3.3 开发 Redux 中间件

开发中间件的模板代码



```
export default store => next => action => { }
```

# 3. Redux 中间件

## 3.4 注册中间件

中间件在开发完成以后只有被注册才能在Redux的工作流程中生效



```
import { createStore, applyMiddleware } from 'redux';  
import logger from './middlewares/logger';  
  
createStore(reducer, applyMiddleware(  
  logger  
));
```

# 中间件开发实例

# 3. Redux 中间件

## 3.5 中间件开发实例 thunk

thunk 中间件可以让我们在 Redux 的工作流程中加入异步代码



```
export default ({dispatch, getState}) => next => action => {  
  if (typeof action === 'function') {  
    return action(dispatch, getState)  
  }  
  next(action);  
}
```

# Redux-thunk

# 4. Redux 常用中间件

## 4.1 redux-thunk

### 4.1.1 redux-thunk 下载

```
npm install redux-thunk
```



# 4. Redux 常用中间件

## 4.1 redux-thunk

### 4.1.2 引入 redux-thunk



```
import thunk from 'redux-thunk';
```

# 4. Redux 常用中间件

## 4.1 redux-thunk

### 4.1.3 注册 redux-thunk




```
import { applyMiddleware } from 'redux';  
createStore(rootReducer, applyMiddleware(thunk));
```

# 4. Redux 常用中间件

## 4.1 redux-thunk

### 4.1.4 使用 redux-thunk 中间件



```
const loadPosts = () => async dispatch => {  
  const posts = await axios.get('/api/posts').then(response => response.data);  
  dispatch({type: LOADPOSTSSUCCESS, payload: posts });  
};
```

# Redux-saga

# Redux-saga 传参

# 4. Redux 常用中间件

## 4.2 redux-saga

### 4.2.1 redux-saga 解决的问题

redux-saga 可以将异步操作从 Action Creator 文件中抽离出来，放在一个单独的文件中。

# 4. Redux 常用中间件

## 4.2 redux-saga

### 4.2.2 redux-saga 下载

```
npm install redux-saga
```

# 4. Redux 常用中间件

## 4.2 redux-saga

### 4.2.3 创建 redux-saga 中间件



```
import createSagaMiddleware from 'redux-saga';  
const sagaMiddleware = createSagaMiddleware();
```



# 4. Redux 常用中间件

## 4.2 redux-saga

### 4.2.4 注册 sagaMiddleware



```
createStore(reducer, applyMiddleware(sagaMiddleware));
```

# 4. Redux 常用中间件

## 4.2 redux-saga

### 4.2.5 使用 saga 接收 action 执行异步操作



```
import { takeEvery, put } from 'redux-saga/effects';

function* load_posts () {
  const { data } = yield axios.get('/api/posts.json');
  yield put(load_posts_success(data));
}

export default function* postSaga () {
  yield takeEvery(LoadPosts, load_posts)
}
```

# 4. Redux 常用中间件

## 4.2 redux-saga

### 4.2.6 启动saga




```
import postSaga from './store/sagas/post.saga';  
sagaMiddleware.run(postSaga);
```

# Redux-saga 拆分合并

# 4. Redux 常用中间件

## 4.2 redux-saga

### 4.2.7 合并sagas



```
import { all } from 'redux-saga/effects';
import counterSaga from './counte.saga';
import postSaga from './post.saga';
export default function* rootSaga () {
  yield all([
    counterSaga(),
    postSaga()
  ])
};
import rootSaga from './root.saga';
sagaMiddleware.run(rootSaga);
```

# Redux-actions

# 4. Redux 常用中间件

## 4.3 redux-actions

### 4.3.1 redux-actions 解决的问题

redux流程中大量的样板代码读写很痛苦, 使用redux-actions可以简化Action和Reducer的处理.

# 4. Redux 常用中间件

## 4.3 redux-actions

### 4.3.2 redux-actions 下载

```
npm install redux-actions
```



# 4. Redux 常用中间件

## 4.3 redux-actions

### 4.3.3 创建 Action



```
import { createAction } from 'redux-actions';  
  
const increment_action = createAction('increment');  
const decrement_action = createAction('increment');
```

# 4. Redux 常用中间件

## 4.3 redux-actions

### 4.3.4 创建 Reducer



```
import { handleActions as createReducer } from 'redux-actions';
import { increment_action, decrement_action } from '../actions/counter.action';

const initialState = {count: 0};
const counterReducer = createReducer({
  [increment_action]: (state, action) => ({count: state.count + 1}),
  [decrement_action]: (state, action) => ({count: state.count - 1})
}, initialState);
export default counterReducer;
```

# 项目初始化

# 搭建 Redux workflow

# 商品列表数据展示

加入商品到购物车

# 购物车列表数据展示

# 从购物车中删除商品



# 更改购物车中商品数量

# 更正视图中图片显示错误

# 计算商品总价

# Redux 源码之核心逻辑

# Redux 源码之类型约束

# Redux 源码之 Enhancer

# Redux 源码之 applyMiddleware

# Redux 源码之 bindActionCreatorsCreators



# Redux 源码之 combineReducers