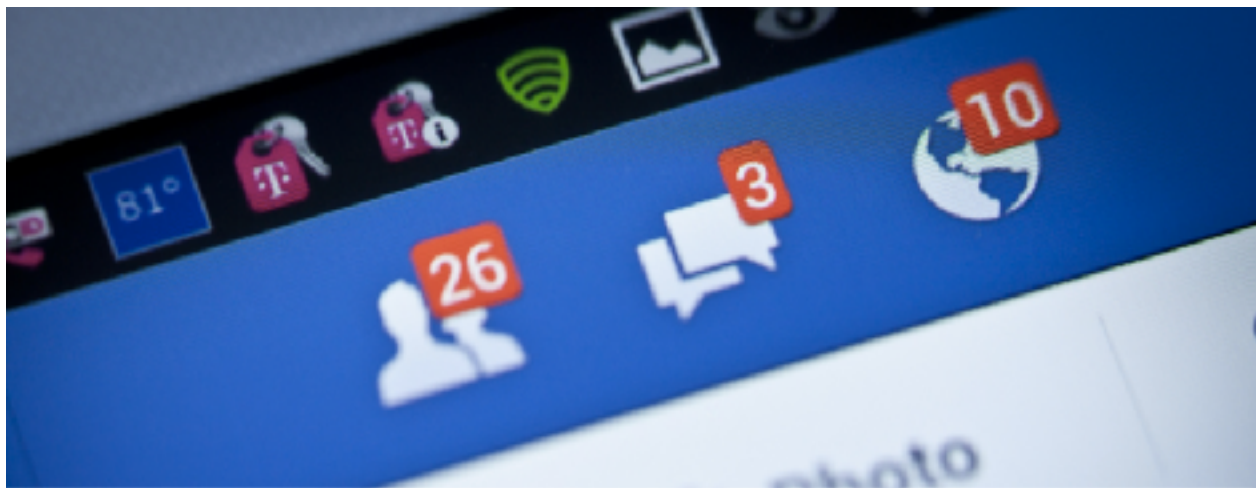


# React 从入门到精通

最全面的 React 技术栈及最佳实践教程

# React 出现的历史背景及特性介绍

# “简单”功能一再出现 Bug



# 问题出现的根源

1. 传统 UI 操作关注太多细节
2. 应用程序状态分散在各处，难以追踪和维护

# 传统 DOM API 关注太多细节

Selectors	Attributes/CSS	Manipulation	Traversing
Basics * .class element #id selector1, selectorN, ...	Attributes .attr() .prop() .removeAttr() .removeProp() .val()	Copying .clone()  DOM Insertion, Around .wrap() .wrapAll() .wrapInner()	Filtering .eq() .filter() .first() .has() .is() .last() .map() .not()
Hierarchy parent > child ancestor descendant prev + next prev ~ siblings	CSS .addClass() .css() jQuery.cssHooks .hasClass() .removeClass()	DOM Insertion, Inside .append() .appendTo() .html() .prepend() .prependTo() .text()	Miscellaneous Traversing .add() .andSelf() .contents() .each() .end()
Basic Filters :animated :eq() :even :first :gt() :header :lang()	Dimensions .height() .innerHeight() .innerWidth() .outerHeight() .outerWidth() .width()	DOM Removal .detach() .empty() .remove() .unwrap()	Tree Traversal .addBack() .children() .closest() .find()

# React：始终整体“刷新”页面

无需关心细节

# 局部刷新

```
{ text: 'message1' }  
{ text: 'message2' }
```

```
{ text: 'message3' }
```



```
<ul>  
  <li>message1</li>  
  <li>message2</li>  
</ul>
```

Append:  
<li>message3</li>

# React : 整体刷新

```
{ text: 'message1' }  
{ text: 'message2' }  
{ text: 'message3' }
```



```
<ul>  
  <li>message1</li>  
  <li>message2</li>  
  <li>message3</li>  
</ul>
```

# React 很简单

1个新概念

4个必须 API

单向数据流

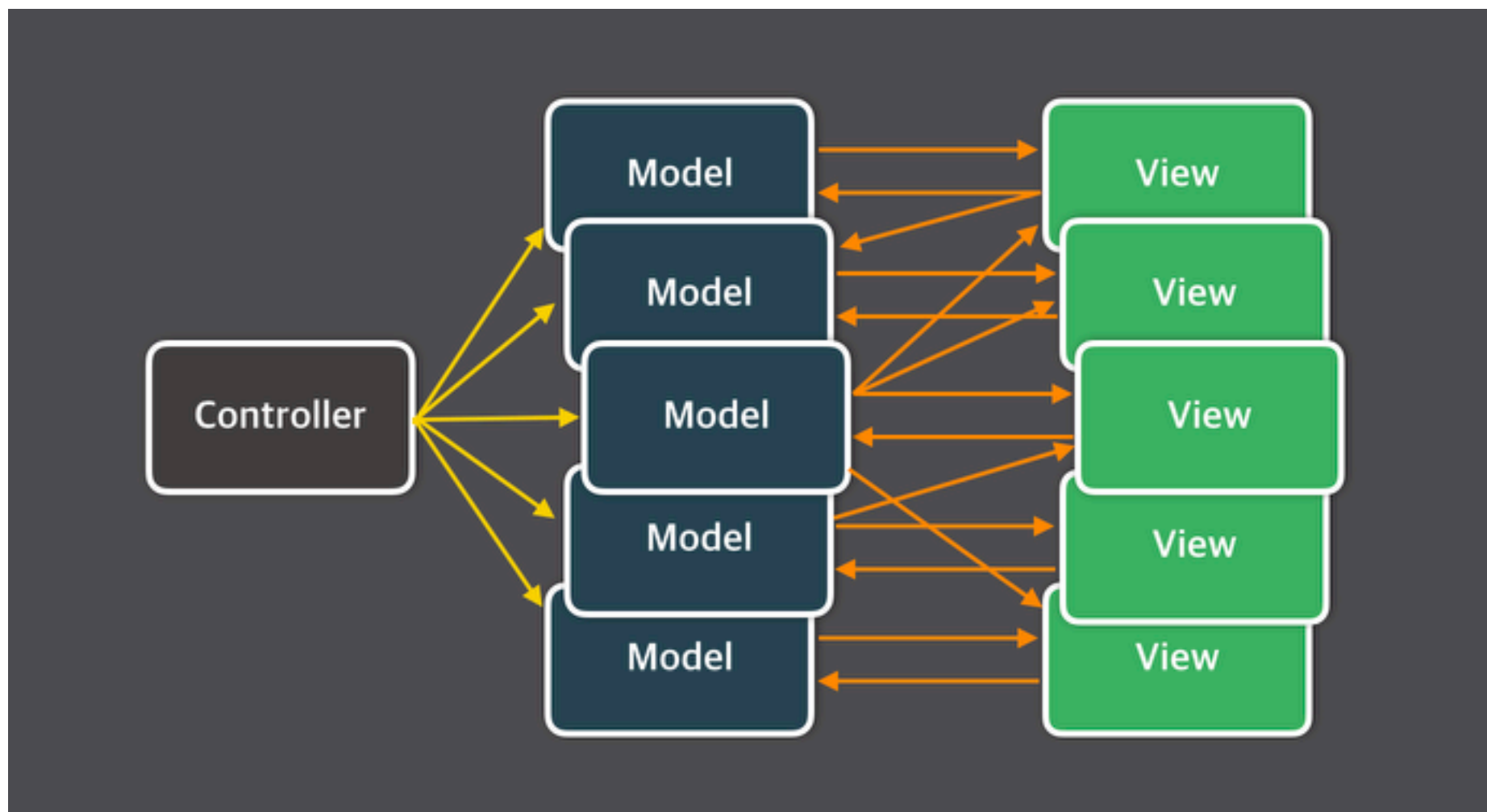
完善的错误提示



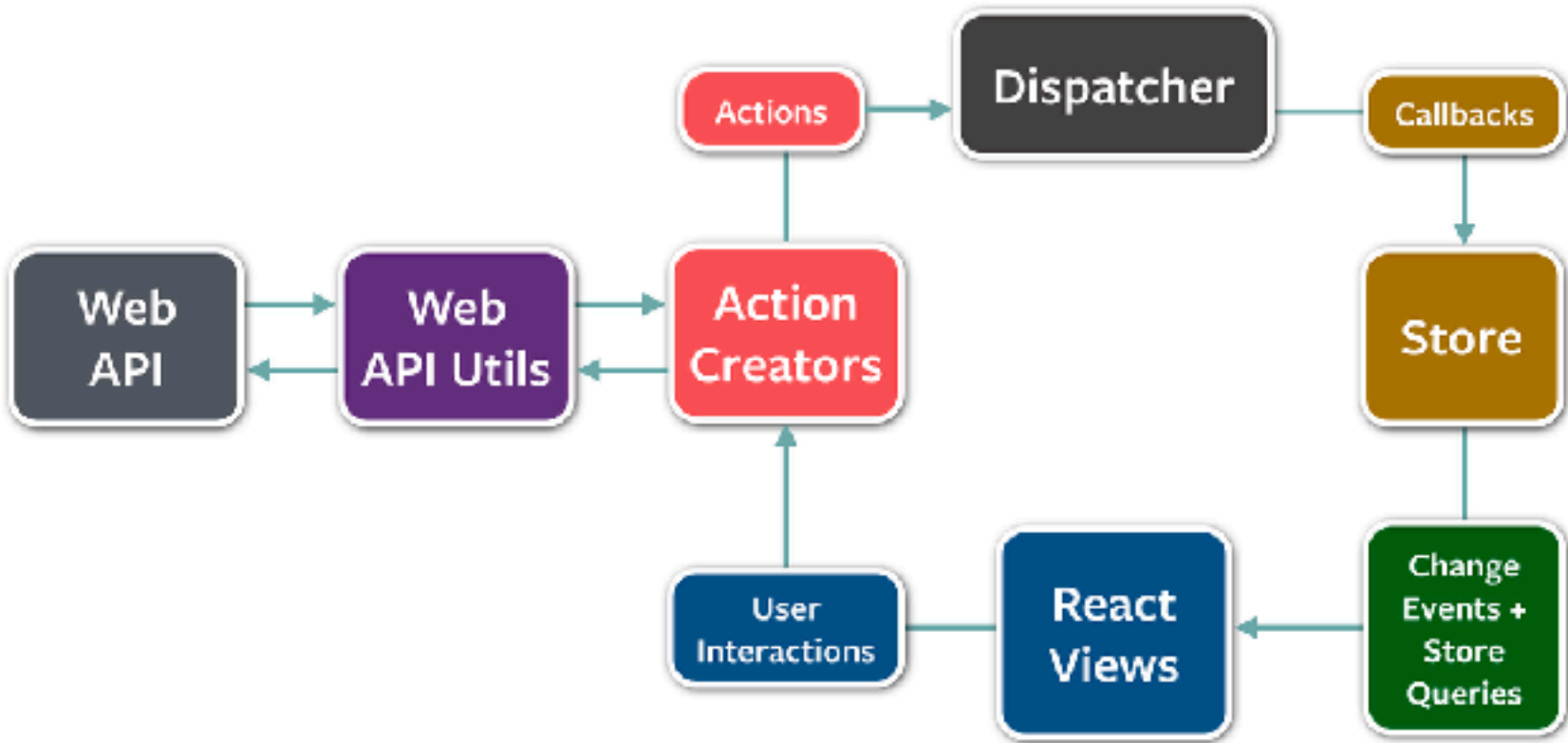
# React 解决了 UI 细节问题

## 数据模型如何解决？

# 传统 MVC 难以扩展和维护



# Flux架构：单向数据流



# Flux 架构的衍生项目



**Redux**



**MobX**

# 小结

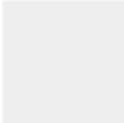
1. 传统 Web UI 开发的问题
2. React : 始终整体刷新页面
3. Flux : 单向数据流

# 以组件方式考虑 UI 的构建

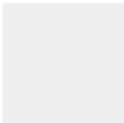
# 以组件方式考虑 UI 的构建

## Comments (3)


---



[Nate](#)  
Hello React! This is a sample comment.

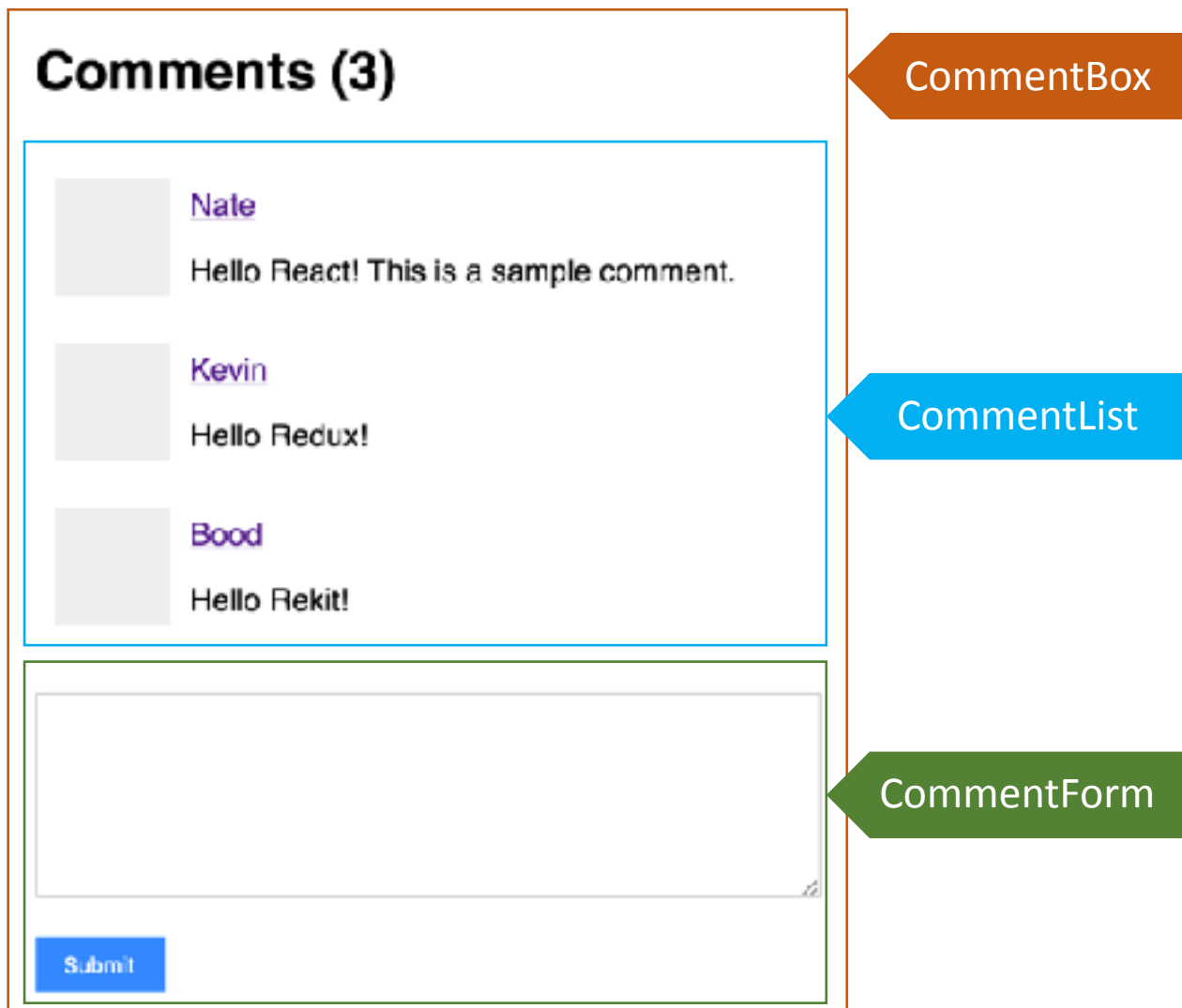


[Kevin](#)  
Hello Redux!



[Bood](#)  
Hello Rekit!

# 将 UI 组织成组件树的形式



```
class CommentBox extends Component {  
  render() {  
    return (  
      <div className="comment-box">  
        <h1>Comments</h1>  
        <CommentList />  
        <CommentForm />  
      </div>  
    );  
  }  
}
```



# 理解 React 组件



1. React组件一般不提供方法，而是某种状态机
2. React组件可以理解为一个纯函数
3. 单向数据绑定

# 创建一个简单的组件：TabSelect

Select color:

1. 创建静态 UI
2. 考虑组件的状态组成
3. 考虑组件的交互方式

# 受控组件 vs 非受控组件

表单元素状态由使用者维护

```
<input
  type="text"
  value={this.state.value}
  onChange={evt =>
    this.setState({ value: evt.target.value})
  }
/>
```

表单元素状态 DOM 自身维护

```
<input
  type="text"
  ref={node => this.input = node}
/>
```

# 何时创建组件：单一职责原则

1. 每个组件只做一件事
2. 如果组件变得复杂，那么应该拆分成小组件

# 数据状态管理：DRY 原则

- 1.能计算得到的状态就不要单独存储
- 2.组件尽量无状态，所需数据通过 props 获取

# 小结

1. 以组件方式思考 UI 的构建
2. 单一职责原则
3. DRY 原则

理解 JSX：不是模板语言，只是一种语法糖

# JSX : 在 JavaScript 代码中直接写 HTML 标记



```
const name = 'Nate Wang';  
const element = <h1>Hello, {name}</h1>;
```



# JSX 的本质：动态创建组件的语法糖



```
const name = 'Nate Wang';  
const element = <h1>Hello, {name}</h1>;
```

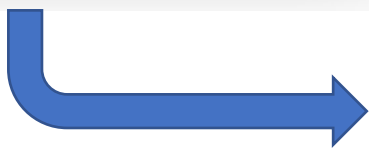


```
const name = 'Josh Perez';  
const element = React.createElement(  
  'h1',  
  null,  
  'Hello, ',  
  name  
);
```

# JSX 的本质：动态创建组件的语法糖

```
class CommentBox extends React.Component {
  render () {
    return (
      <div className="comments">
        <h1>Comments ({this.state.items.length})</h1>
        <CommentList data={this.state.items}/>
        <CommentForm />
      </div>
    );
  }
}

ReactDOM.render(<CommentBox topicId="1" />, mountNode);
```



```
class CommentBox extends React.Component {
  render() {
    return React.createElement(
      "div",
      { className: "comments" },
      React.createElement(
        "h1",
        null,
        "Comments (",
        this.state.items.length,
        ")"
      ),
      React.createElement(CommentList, { data: this.state.items }),
      React.createElement(CommentForm, null)
    );
  }
}

ReactDOM.render(
  React.createElement(CommentBox, { topicId: "1" }),
  mountNode
);
```

# 在 JSX 中使用表达式

## 1. JSX 本身也是表达式

```
const element = <h1>Hello, world!</h1>;
```

## 2. 在属性中使用表达式

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

## 3. 延展属性

```
const props = {firstName: 'Ben', lastName: 'Hector'};  
const greeting = <Greeting {...props} />;
```

## 4. 表达式作为子元素

```
const element = <li>{props.message}</li>;
```

# 对比其它模板语言

```
<html ng-app="todoApp">
  <head>
    <script src="angular.min.js"></script>
    <script src="todo.js"></script>
    <link rel="stylesheet" href="todo.css">
  </head>
  <body>
    <h2>Todo</h2>
    <div ng-controller="TodoListController as todoList">
      <span>
        {{todoList.remaining()}} of {{todoList.todos.length}} remaining
      </span>
      [ <a href="#" ng-click="todoList.archive()">archive</a> ]
      <ul class="unstyled">
        <li ng-repeat="todo in todoList.todos">
          <input type="checkbox" ng-model="todo.done">
          <span class="done-{{todo.done}}">{{todo.text}}</span>
        </li>
      </ul>
      <form ng-submit="todoList.addTodo()">
        <input type="text" ng-model="todoList.todoText" size="30"
          placeholder="add new todo here">
        <input class="btn-primary" type="submit" value="add">
      </form>
    </div>
  </body>
</html>
```

# JSX 优点

1. 声明式创建界面的直观
2. 代码动态创建界面的灵活
3. 无需学习新的模板语言

# 约定：自定义组件以大写字母开头

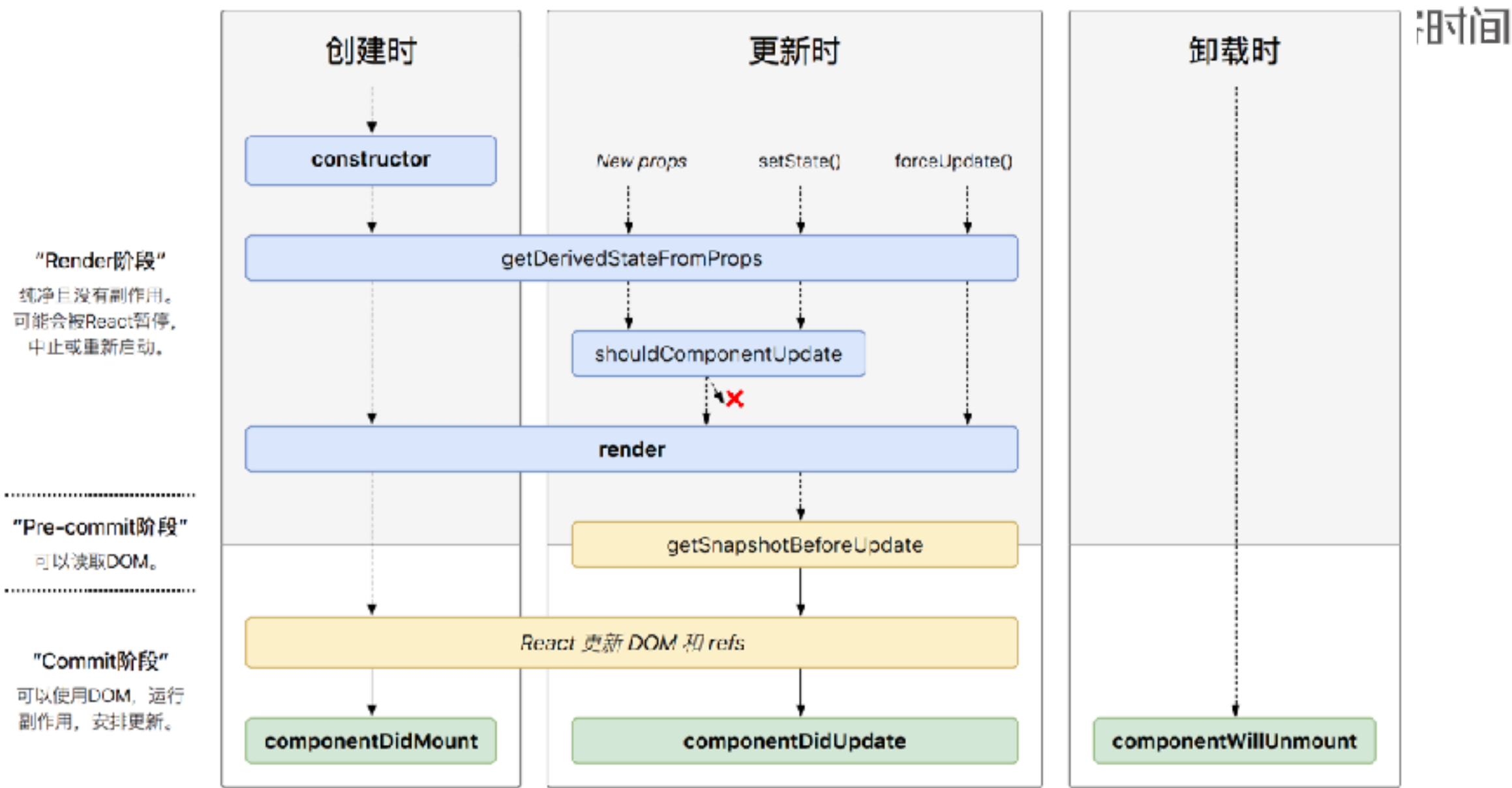
1. React 认为小写的 tag 是原生 DOM 节点，如 div
2. 大写字母开头为自定义组件
3. JSX 标记可以直接使用属性语法，例如 `<menu.Item />`

# 小结

1. JSX 的本质
2. 如何使用 JSX
3. JSX 的优点

# React 组件的生命周期及其使用场景





图片来源：<http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>

# constructor

- 1.用于初始化内部状态，很少使用
- 2.唯一可以直接修改 state 的地方

# getDerivedStateFromProps

1. 当 state 需要从 props 初始化时使用
2. 尽量不要使用：维护两者状态一致性会增加复杂度
3. 每次 render 都会调用
4. 典型场景：表单控件获取默认值

# componentDidMount

1. UI 渲染完成后调用
2. 只执行一次
3. 典型场景：获取外部资源

# componentWillUnmount

1. 组件移除时被调用
2. 典型场景：资源释放

# getSnapshotBeforeUpdate

- 1.在页面 render 之前调用，state 已更新
- 2.典型场景：获取 render 之前的 DOM 状态

# componentDidUpdate

1. 每次 UI 更新时被调用
2. 典型场景：页面需要根据 props 变化重新获取数据

# shouldComponentUpdate

1. 决定 Virtual DOM 是否要重绘
2. 一般可以由 PureComponent 自动实现
3. 典型场景：性能优化



DEMO

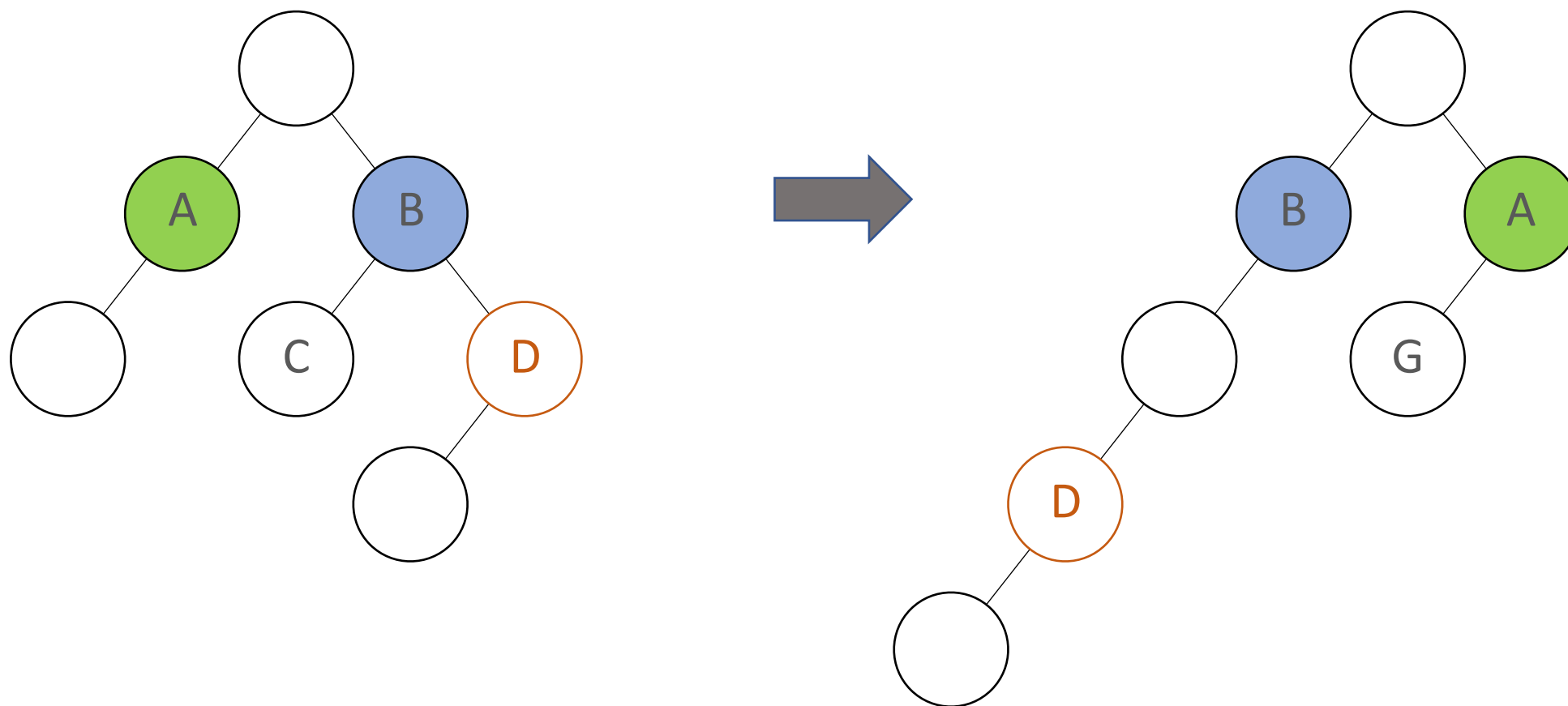
# 小结

1. 理解 React 组件的生命周期方法
2. 理解生命周期的使用场景

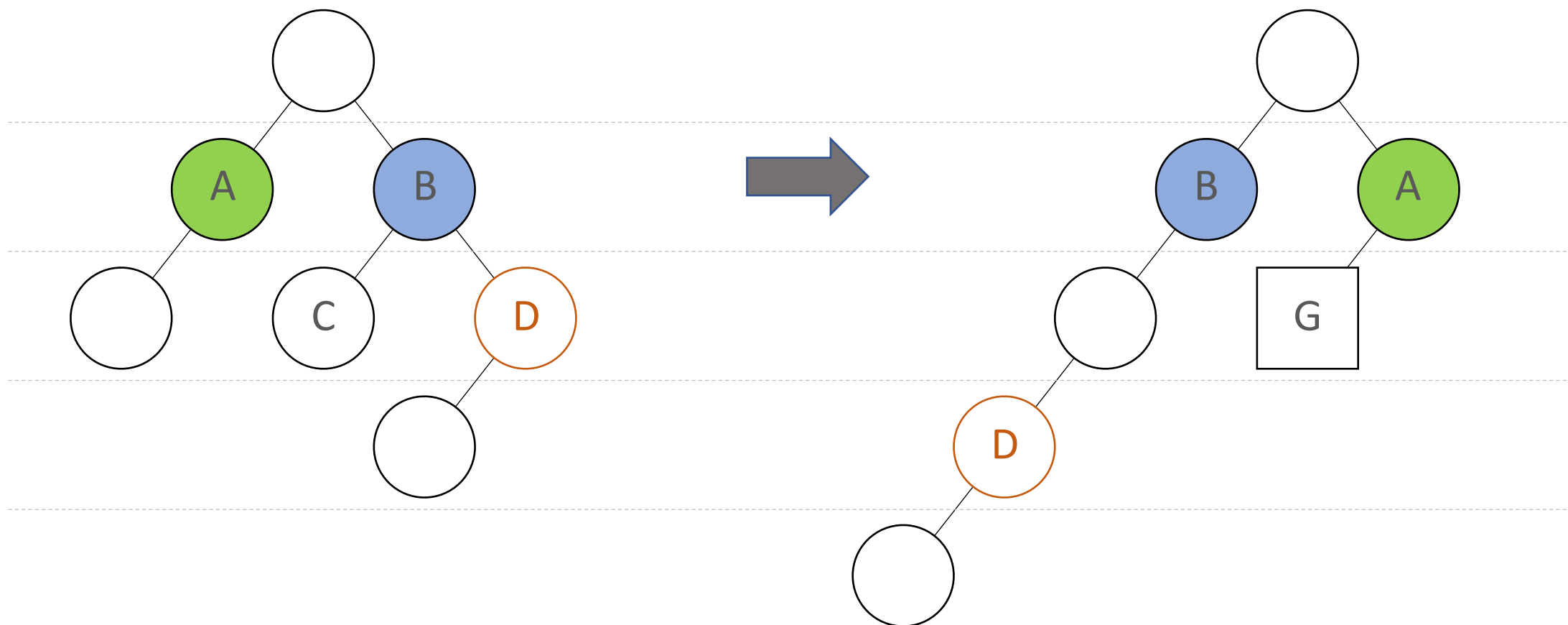
理解 Virtual DOM 的工作原理，  
理解 key 属性的作用

# JSX 的运行基础：Virtual DOM

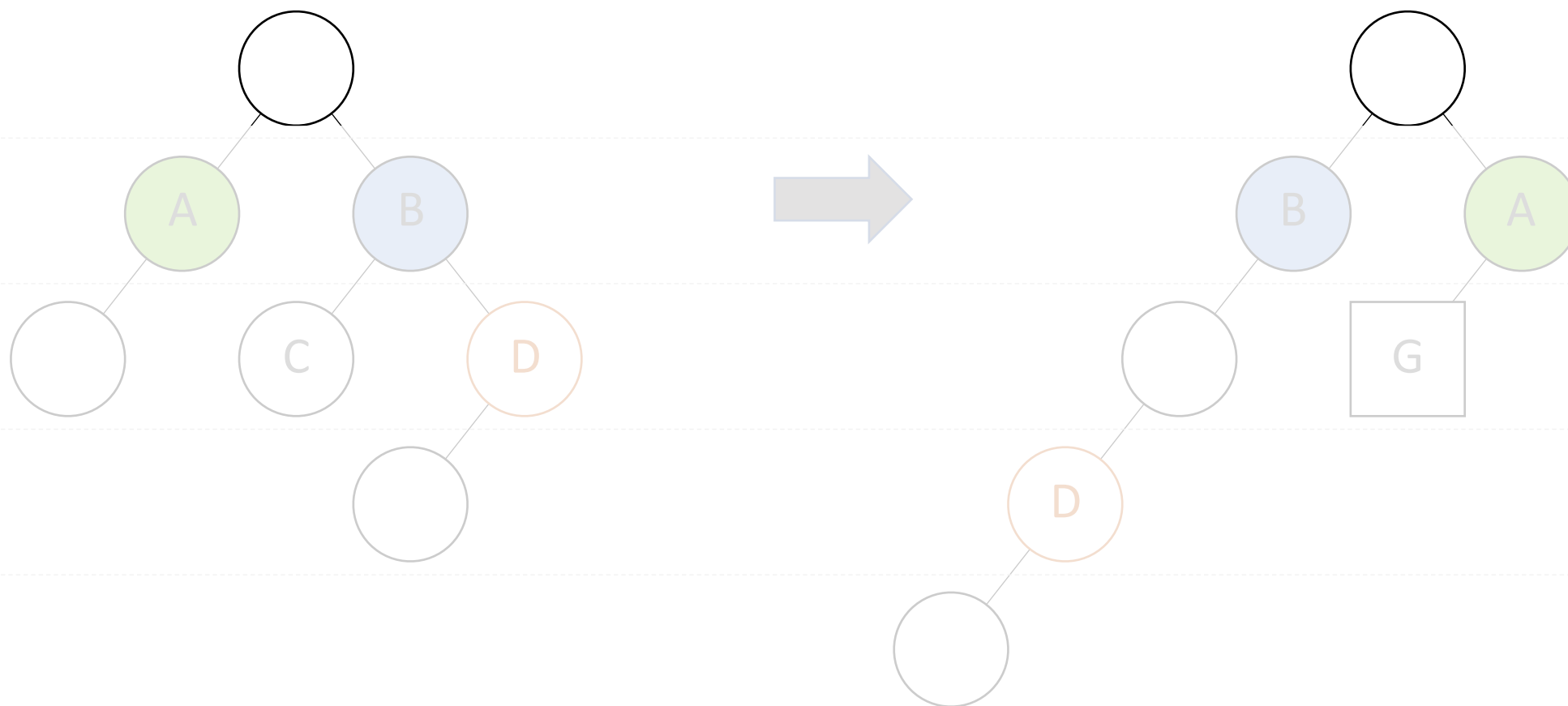
# 虚拟 DOM 是如何工作的



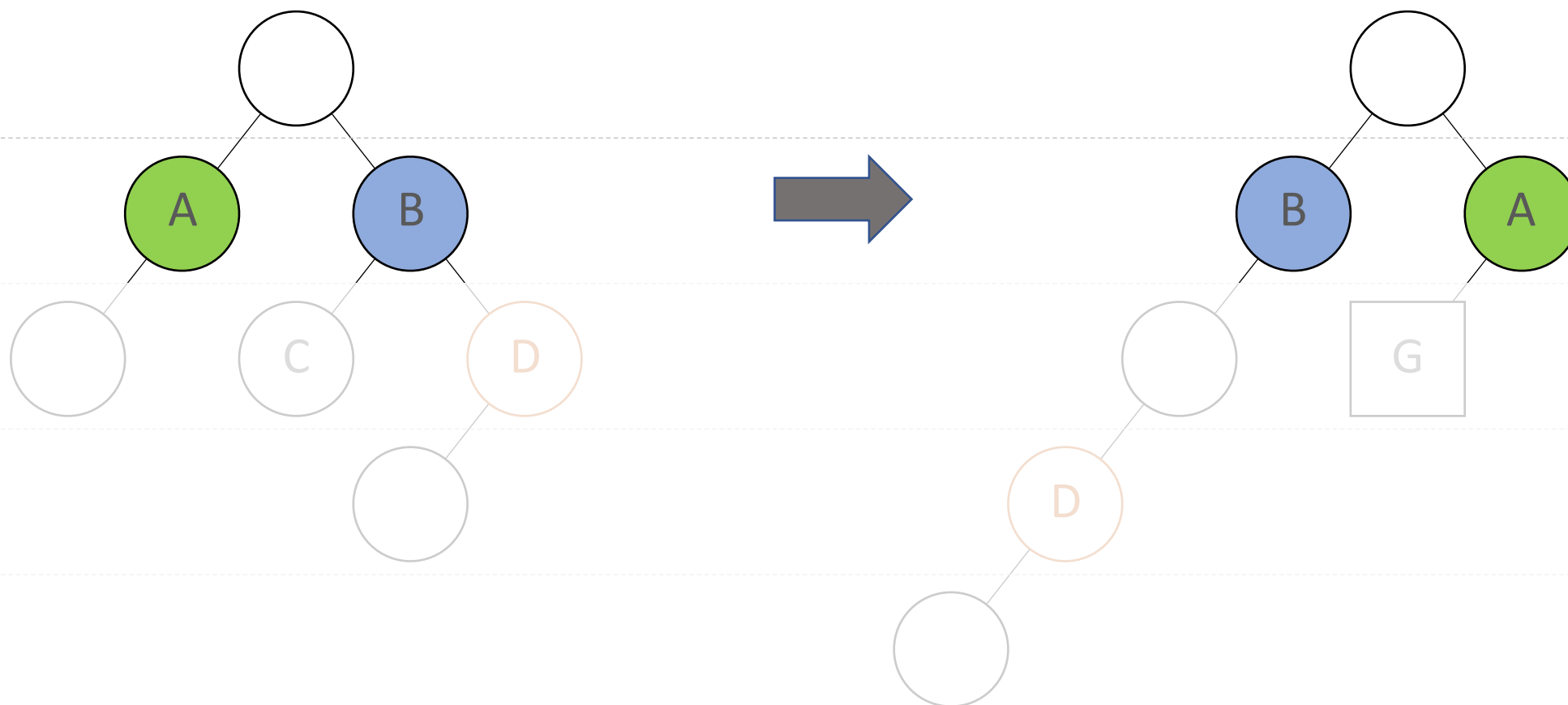
# 广度优先分层比较



# 根节点开始比较

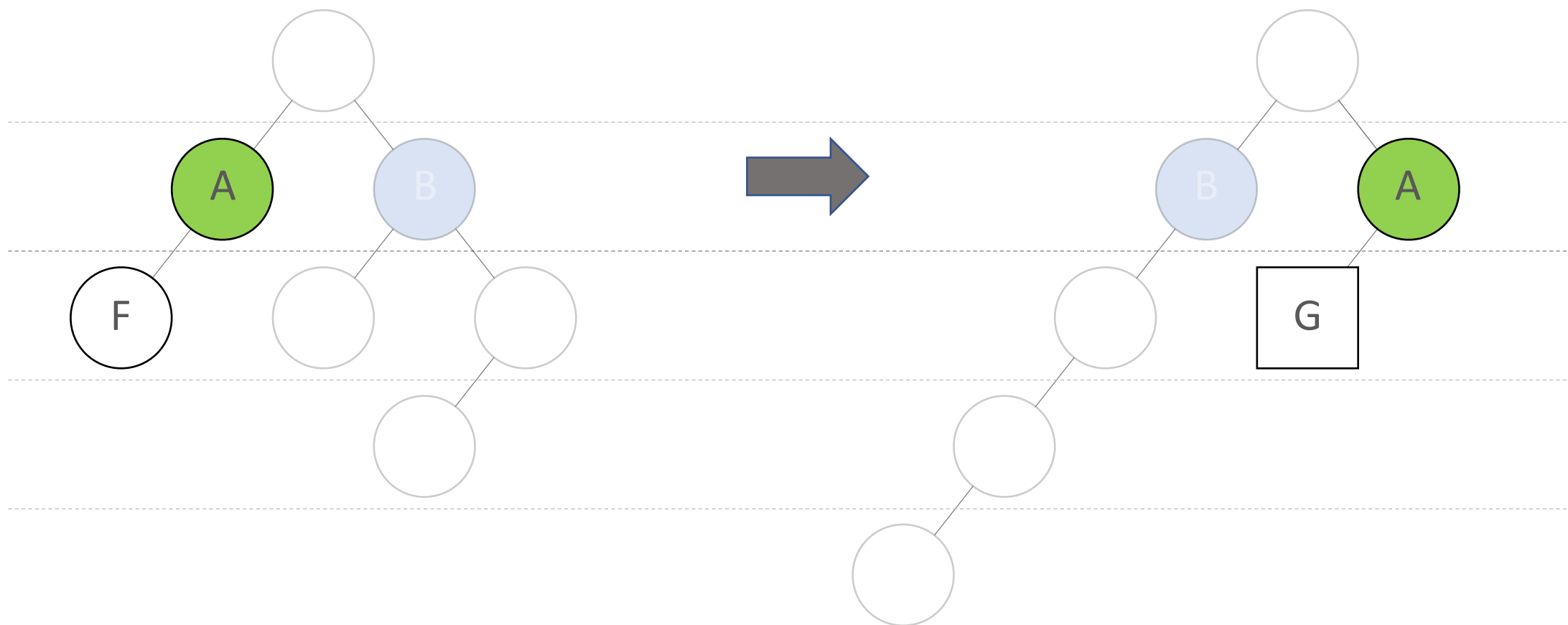


# 属性变化及顺序

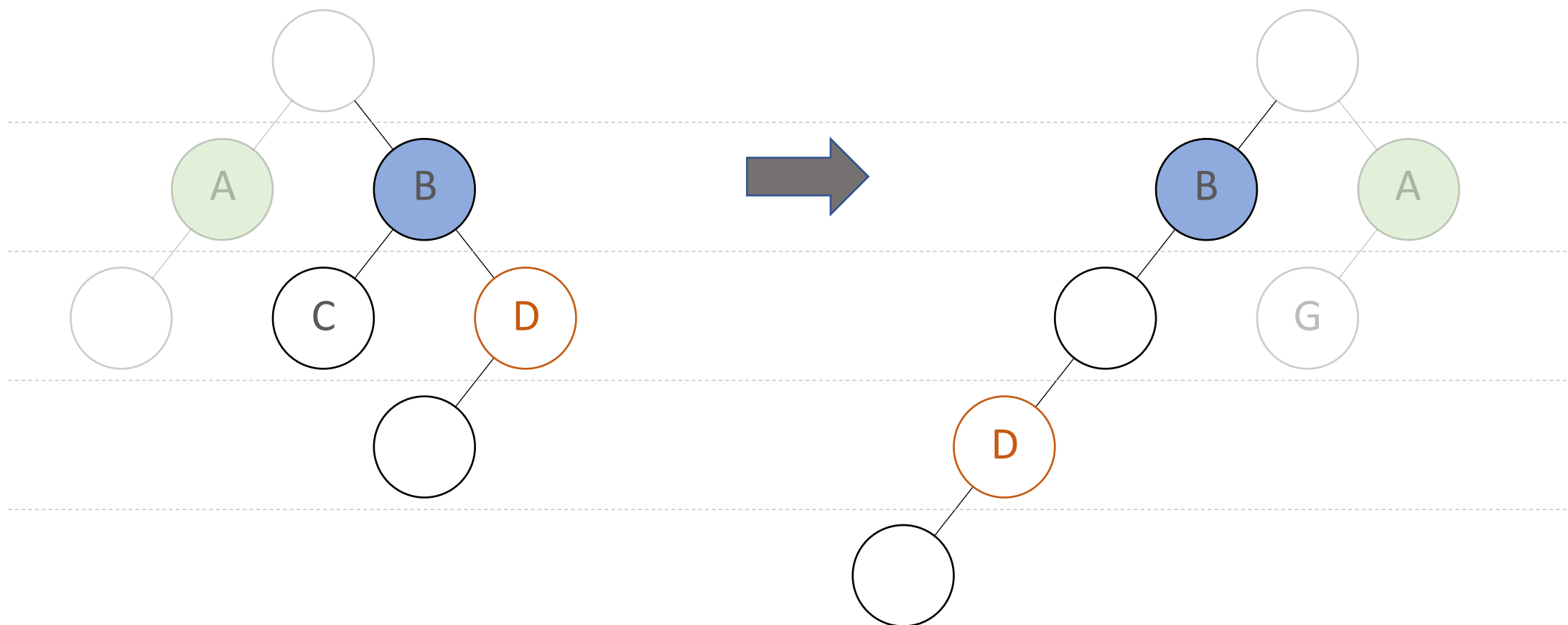




# 节点类型发生变化



# 节点跨层移动



# 虚拟 DOM 的两个假设

1. 组件的 DOM 结构是相对稳定的
2. 类型相同的兄弟节点可以被唯一标识

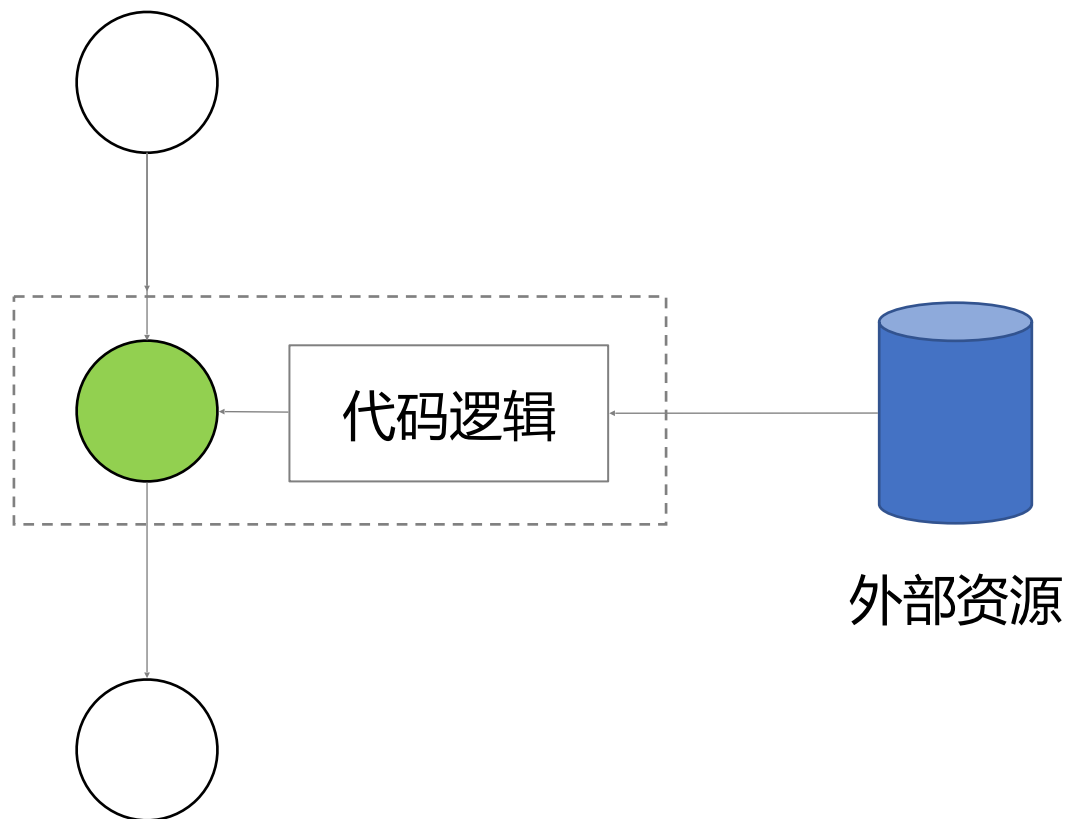
DEMO

# 小结

1. 算法复杂度为  $O(n)$
2. 虚拟 DOM 如何计算 diff
3. key 属性的作用

组件复用的另外两种形式：  
高阶组件和函数作为子组件

# 高阶组件 ( HOC )



```
const EnhancedComponent =  
  higherOrderComponent(WrappedComponent);
```

高阶组件接受组件作为参数，  
返回新的组件。

DEMO



# 函数作为子组件

```
class MyComponent extends React.Component {  
  render() {  
    return (  
      <div>  
        {this.props.children('Nate Wang')}  
      </div>  
    );  
  }  
}  
  
<MyComponent>  
  {(name) => (  
    <div>{name}</div>  
  )}  
</MyComponent>
```

Select color:



Select animal:



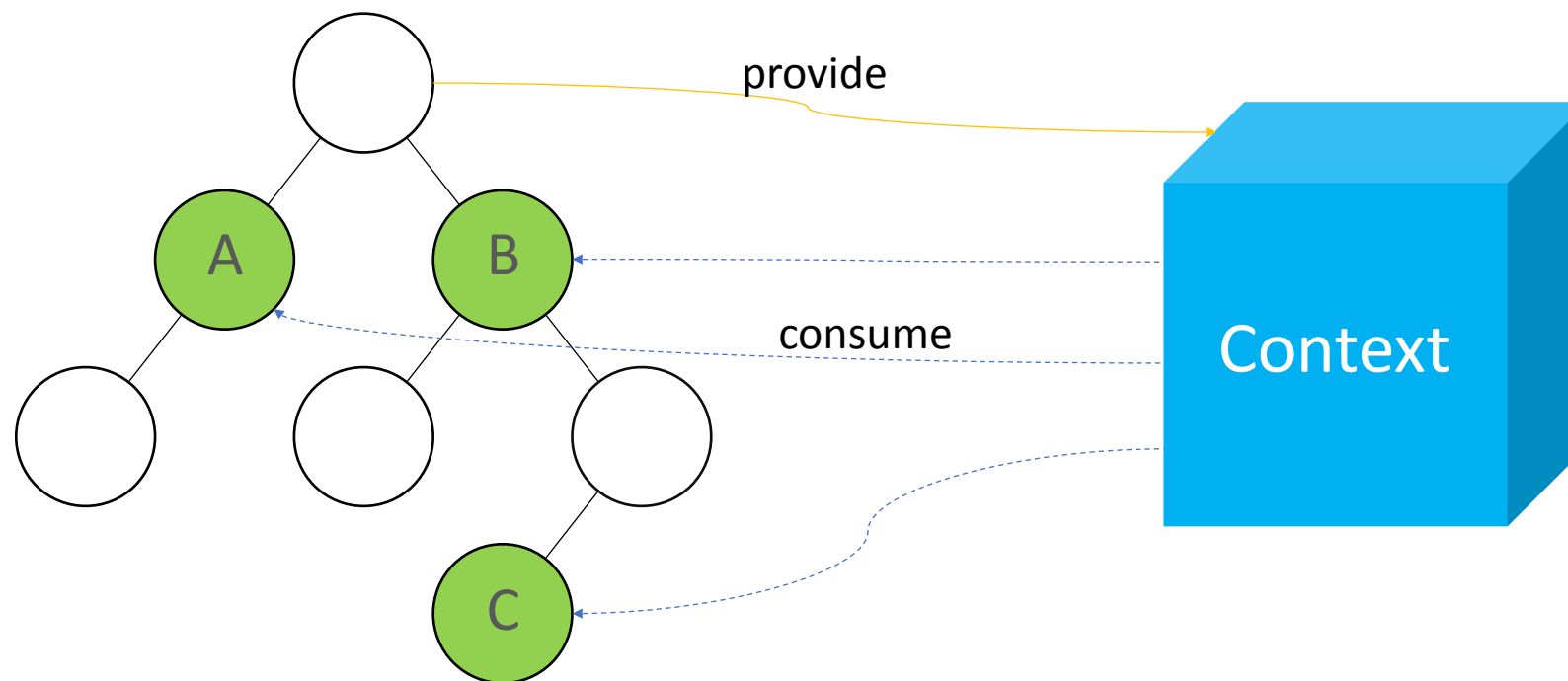
DEMO

# 小结

1. 高阶组件和函数子组件都是设计模式
2. 可以实现更多场景的组件复用

# 理解 Context API 的使用场景

# React 16.3 新特性：Context API



# React 16.3 新特性：Context API

```
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    return (
      <ThemeContext.Provider value="dark">
        <ThemedButton />
      </ThemeContext.Provider>
    );
  }
}

function ThemedButton(props) {
  return (
    <ThemeContext.Consumer>
      {theme => <Button {...props} theme={theme} />}
    </ThemeContext.Consumer>
  );
}
```

DEMO

# 小结

1. Conext API 的使用方法
2. 使用场景



使用脚手架工具创建 React 应用：  
Create React App, Codesandbox, Rekit

# 为什么需要脚手架工具



React



Redux



*BABEL*









webpack  
MODULE BUNDLER

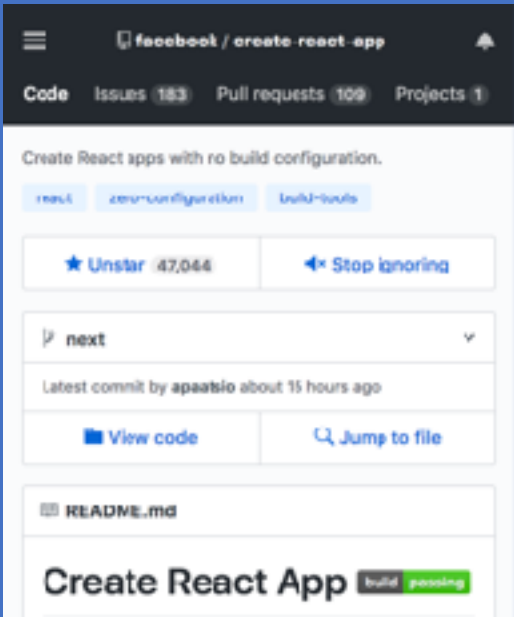


ESLint

# 为什么需要脚手架工具

<p>babel-polyfill isomorphic-fetch lodash react react-dom react-redux react-router react-router-redux redux</p> 	<p>babel-preset-es2015 babel-preset-react babel-preset-stage-0 babel-register chai css-loader enzyme eslint eslint-config-airbnb eslint-import-resolver-babel-module eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-react estraverse estraverse-fb express express-history-api-fallback file-loader jsdom</p> 	<p>less less-loader lodash-webpack-plugin mocha mocha-webpack nock node-sass npm-run nyc react-addons-test-utils react-hot-loader redux-mock-store sass-loader sinon url-loader webpack webpack-dev-middleware webpack-hot-middleware webpack-node-externals</p> 
<p>redux-logger redux-thunk style-loader argparse babel-core babel-eslint babel-loader babel-plugin-istanbul babel-plugin-lodash babel-plugin-module-resolver</p> 	<p>webpack</p> 	<p>ESLint</p> 

# create-react-app



The screenshot shows the GitHub repository for 'facebook / create-react-app'. It includes the repository name, navigation links for Code, Issues (183), Pull requests (109), and Projects (1). The main content area features the repository description: 'Create React apps with no build configuration.' Below this are tabs for 'react', 'zero-configuration', and 'build-tools'. There are also buttons for 'Unstar' (47,044) and 'Stop ignoring'. A commit by 'apaatis' is shown with a 'View code' button and a 'Jump to file' button. A 'README.md' file is visible, and a 'Create React App' button with a 'build passing' status is at the bottom.

Babel

Webpack Config

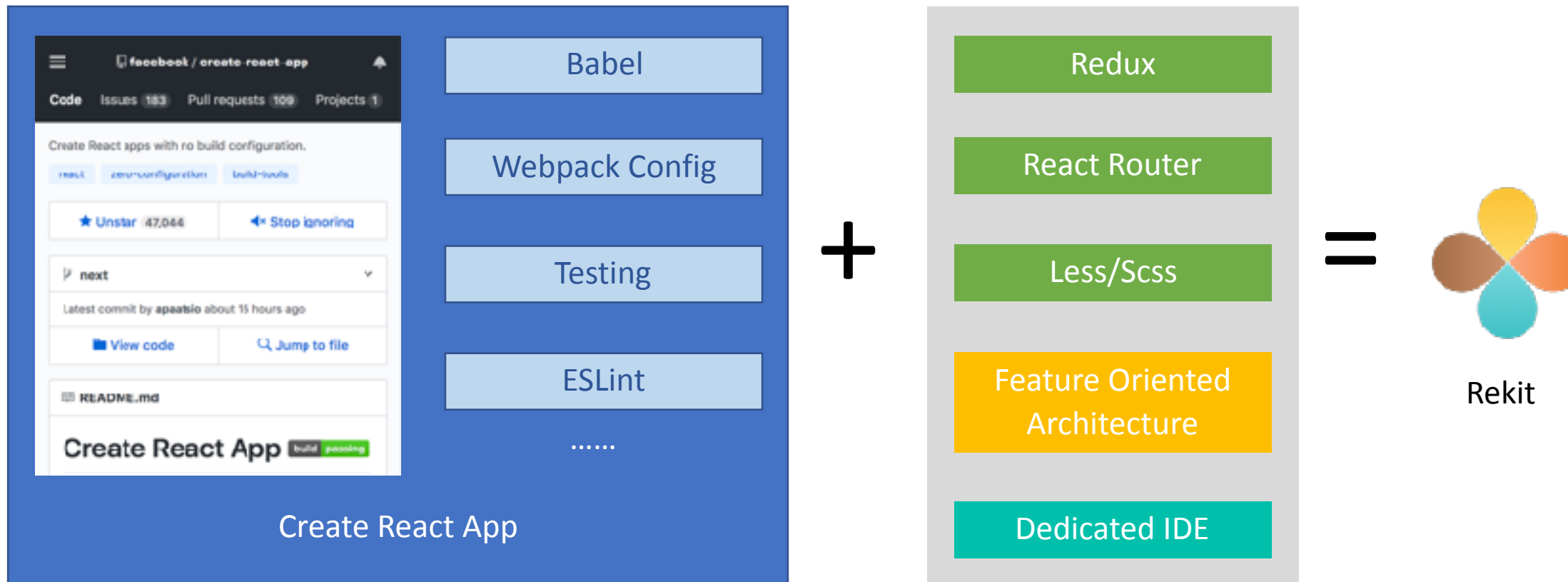
Testing

ESLint

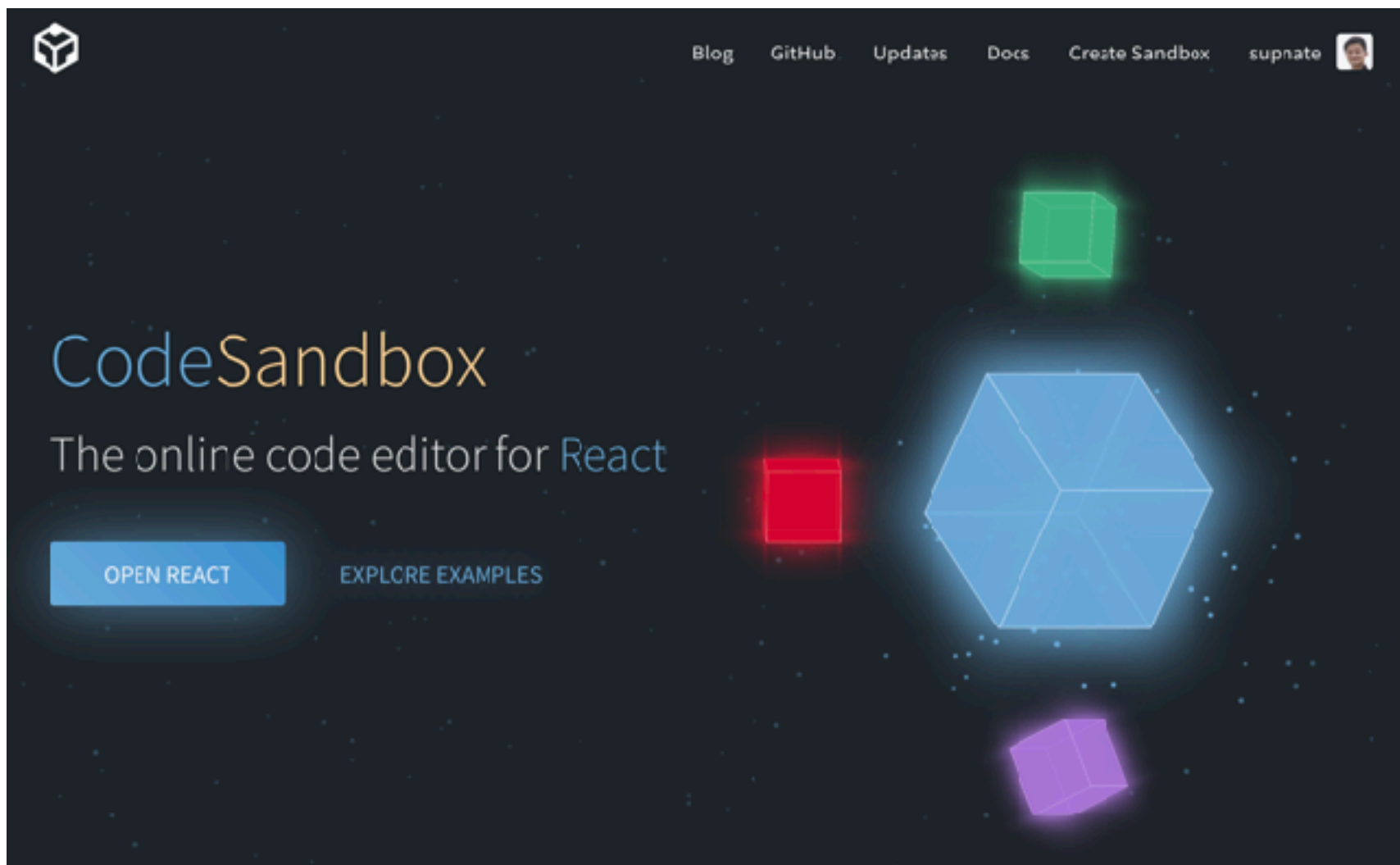
.....

Create React App

# Rekit



# Online: Codesandbox.io



# 小结

介绍了3种脚手架工具及它们的使用场景。

DEMO

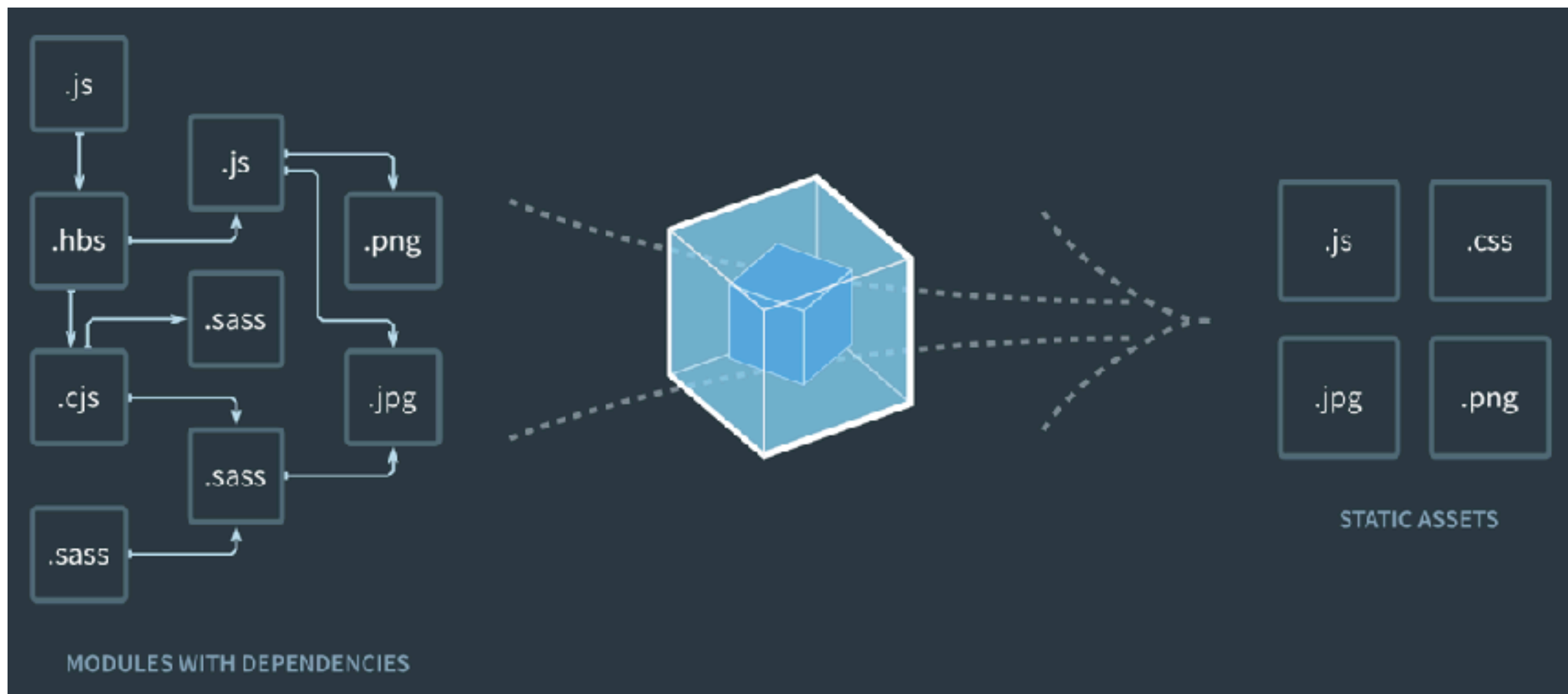


# 打包和部署

# 为什么需要打包？

1. 编译 ES6 语法特性，编译 JSX
2. 整合资源，例如图片，Less/Sass
3. 优化代码体积

# 使用 Webpack 进行打包



# 打包注意事项

1. 设置 nodejs 环境为 production
2. 禁用开发时专用代码，比如 logger
3. 设置应用根路径

DEMO

# 小结

1. 为什么需要打包
2. 如何进行打包
3. 打包和部署的注意事项

# Redux ( 1 ) : JS 状态管理框架



# Redux



@dan\_abramov

Interest over time

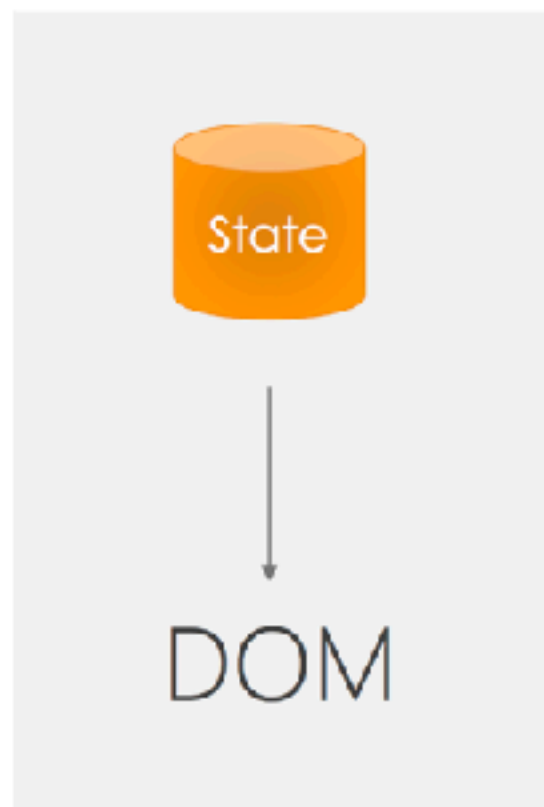
Google Trends

● react flux ● react redux

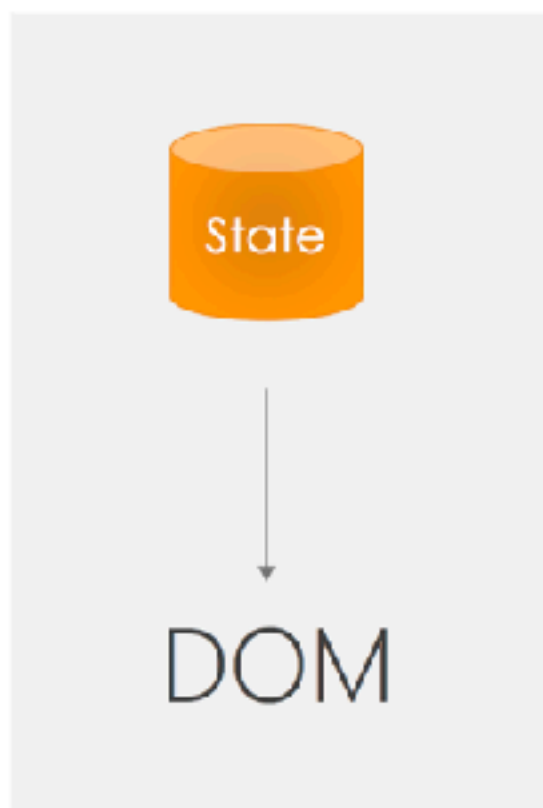




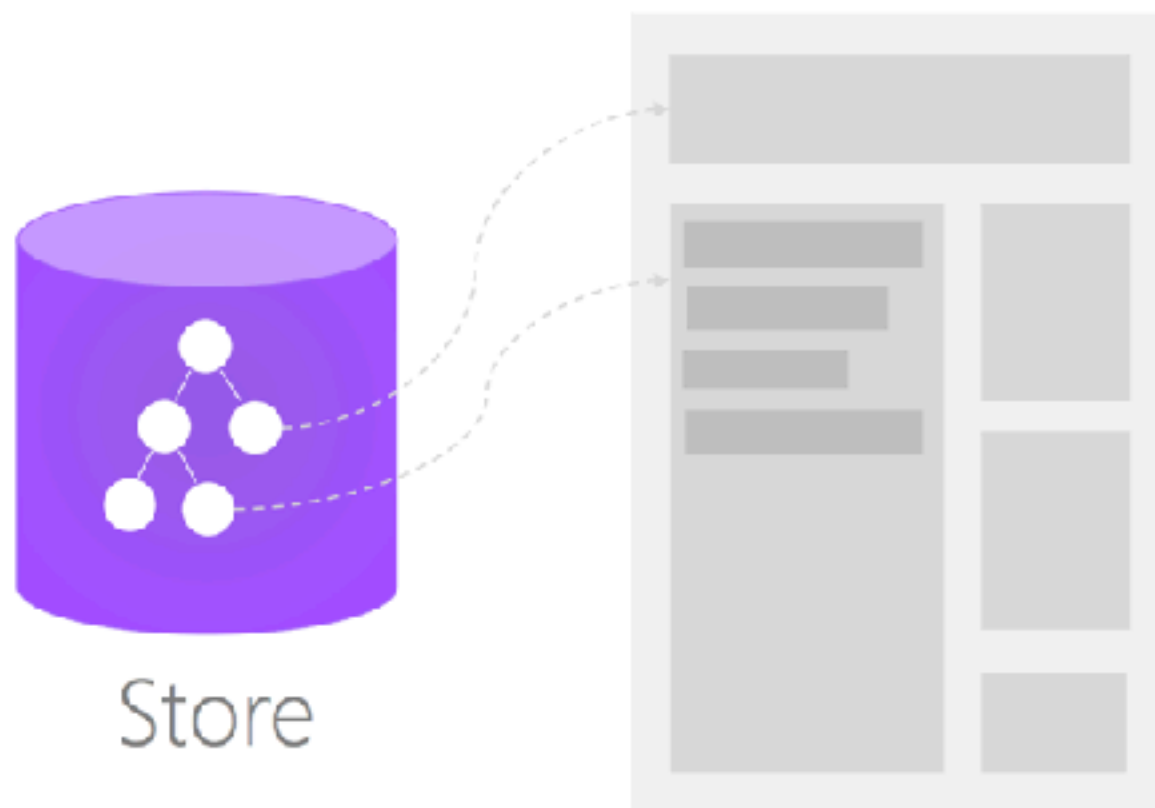
# React



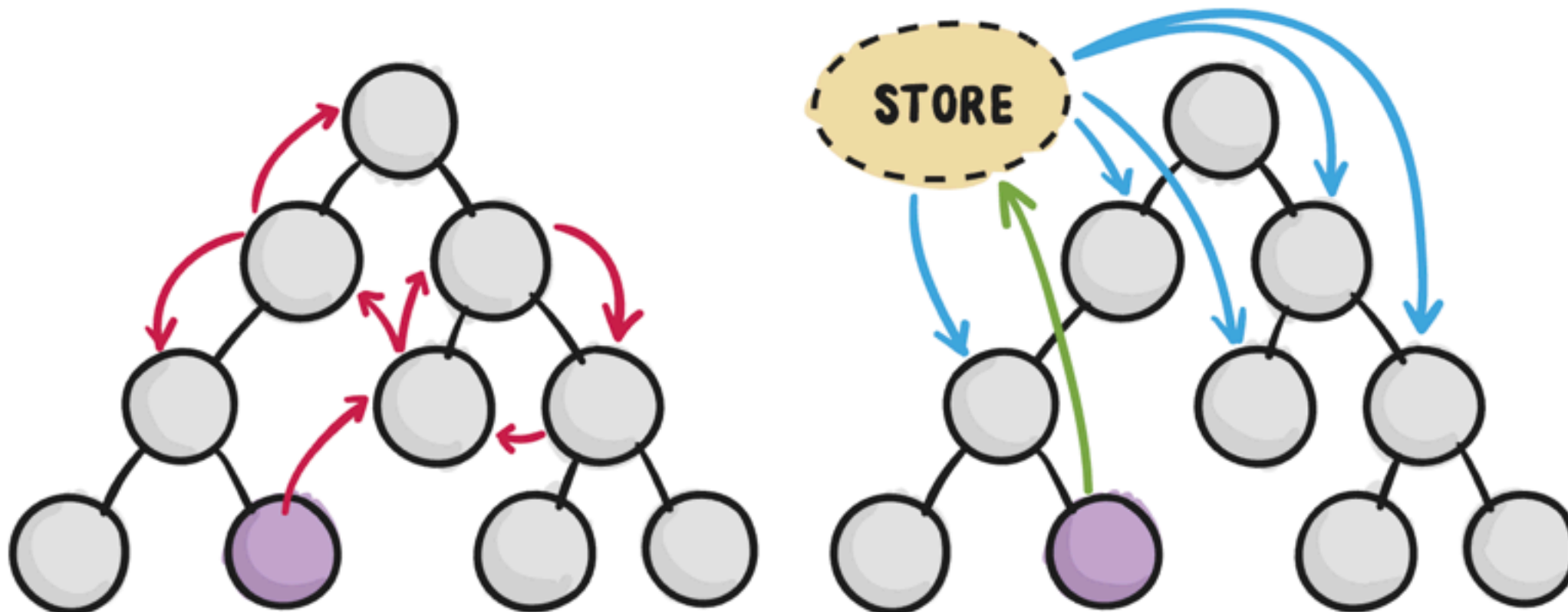
# React



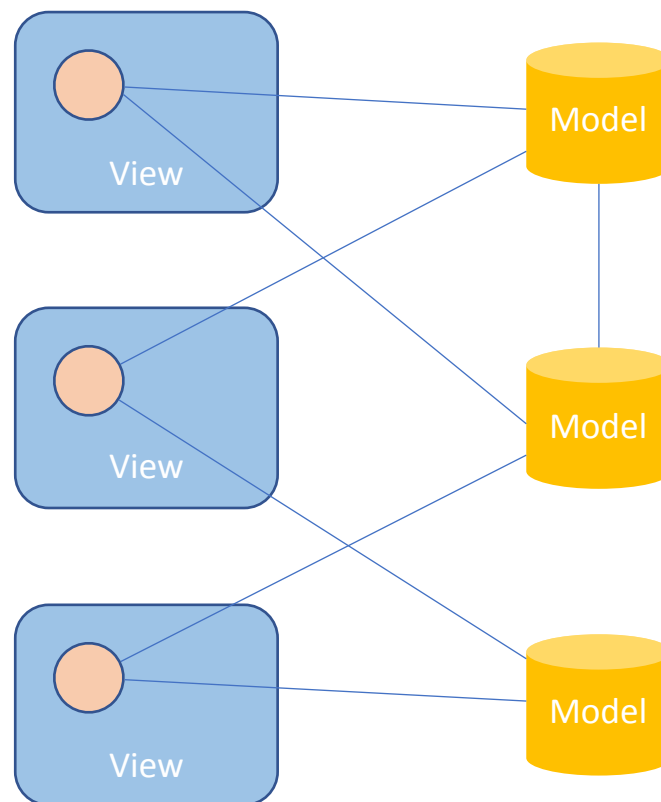
# Redux



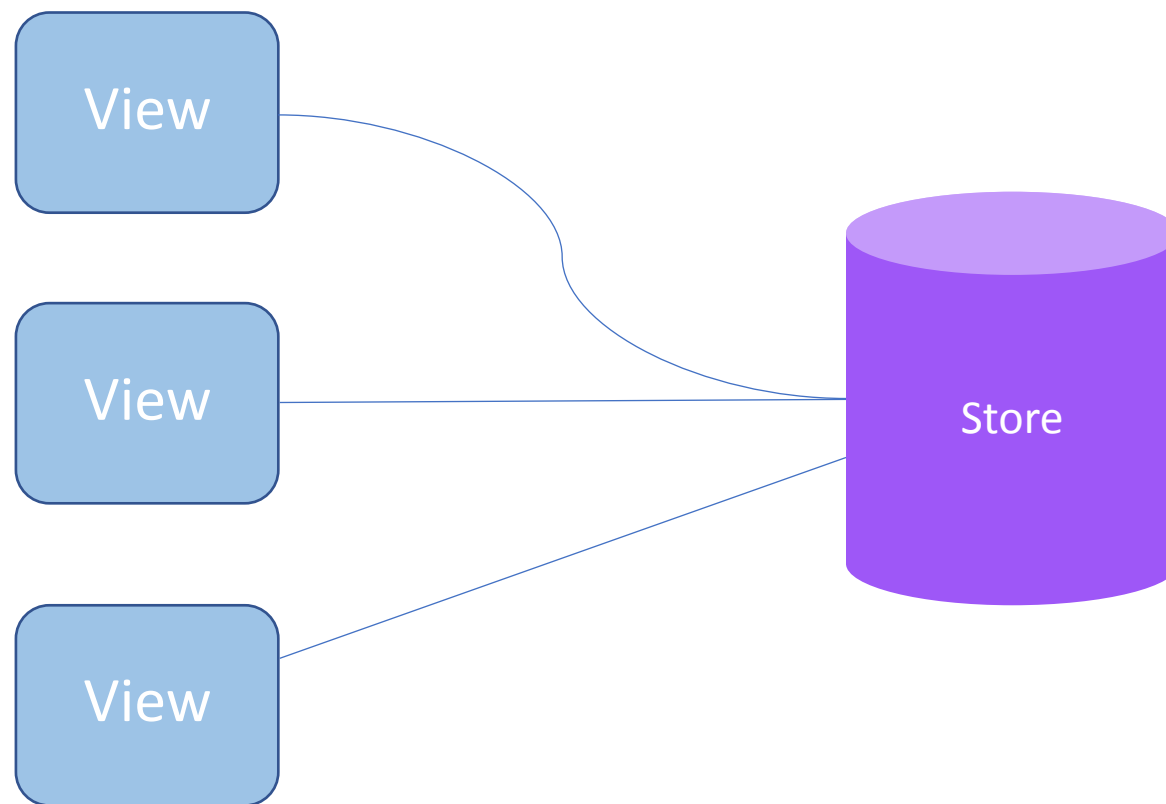
# Redux 让组件通信更加容易



# Redux 特性 : Single Source of Truth



# Redux 特性：Single Source of Truth



# Redux 特性：可预测性

state + action = new state

# Redux 特性：纯函数更新 Store

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map(
        (todo, index) =>
          action.index === index
            ? { text: todo.text, completed: !todo.completed }
            : todo
      )
    default:
      return state
  }
}
```

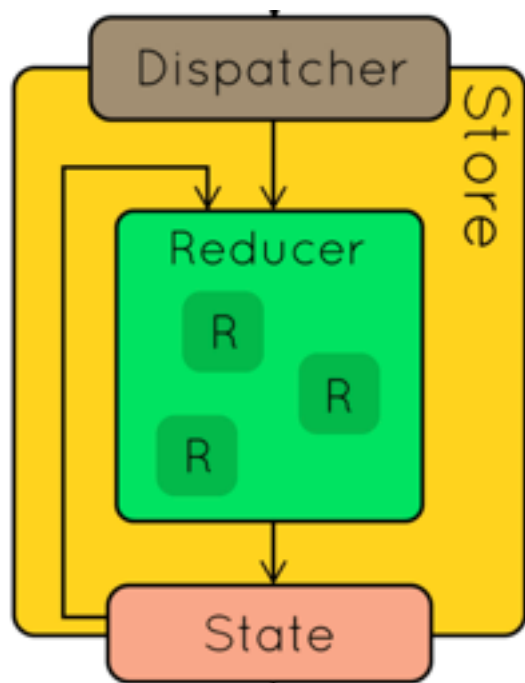
# 小结

1. 为什么需要 Redux
2. Redux 的三个特性



# Redux ( 2 ) : 深入理解 Store , Action , Reducer

# 理解 Store



```
const store = createStore(reducer)
```

1. getState()
2. dispatch(action)
3. subscribe(listener)

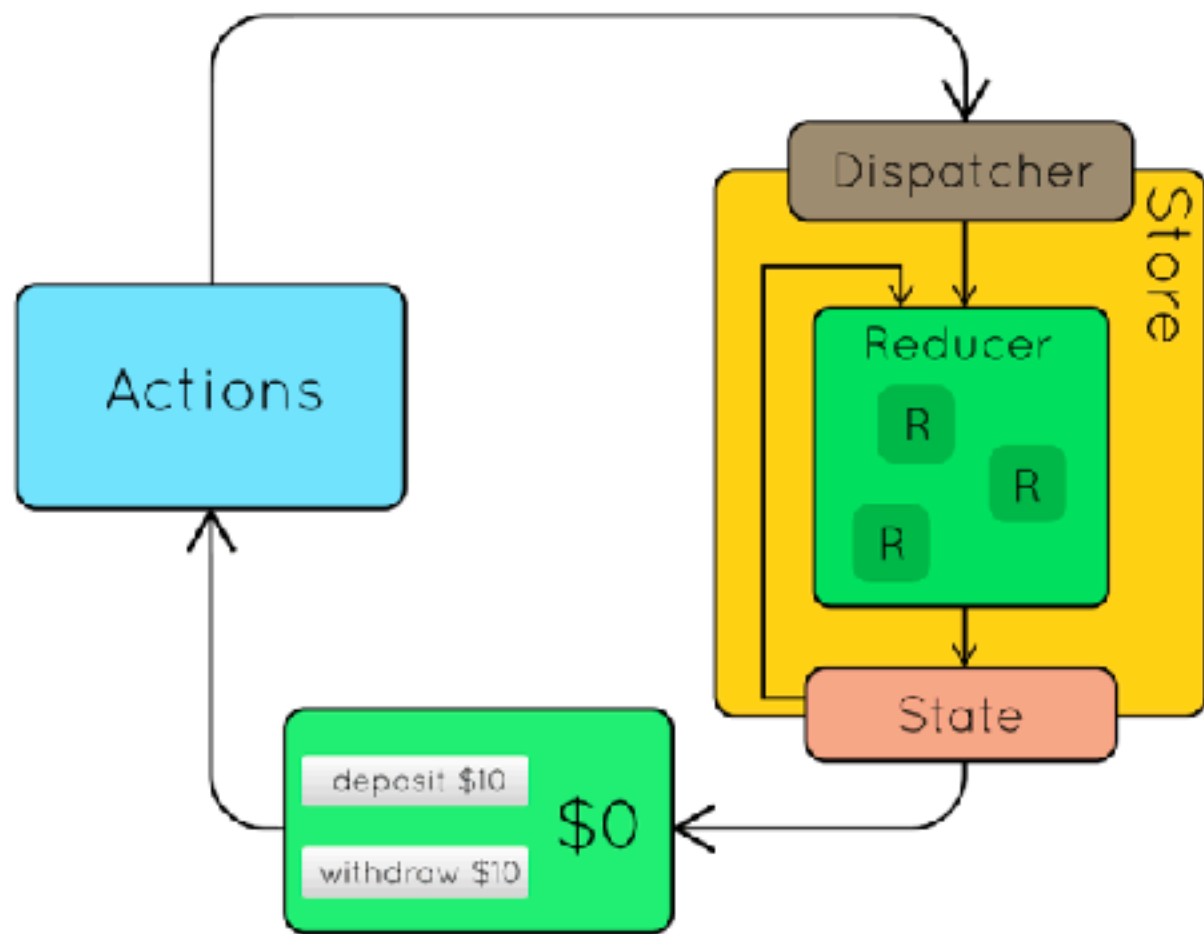
# 理解 action

```
{  
  type: ADD_TODO,  
  text: 'Build my first Redux app'  
}
```

# 理解 reducer

```
function todoApp(state = initialState, action) {  
  switch (action.type) {  
    case ADD_TODO:  
      return Object.assign({}, state, {  
        todos: [  
          ...state.todos,  
          {  
            text: action.text,  
            completed: false  
          }  
        ]  
      })  
    default:  
      return state  
  }  
}
```

# Redux



(state, action) => new state

- Store
- Actions
- Reducer
- View

# 理解 combineReducers

```
export default function todos(state = [], action) {  
  switch (action.type) {  
    case 'ADD_TODO':  
      return state.concat([action.text])  
    default:  
      return state  
  }  
}
```

```
export default function counter(state = 0, action) {  
  switch (action.type) {  
    case 'INCREMENT':  
      return state + 1  
    case 'DECREMENT':  
      return state - 1  
    default:  
      return state  
  }  
}
```

```
import { combineReducers } from 'redux'  
import todos from './todos'  
import counter from './counter'  
  
export default combineReducers({  
  todos,  
  counter  
})
```

# 理解 bindActionCreators

```
function addTodoWithDispatch(text) {  
  const action = {  
    type: ADD_TODO,  
    text  
  }  
  dispatch(action)  
}
```

```
dispatch(addTodo(text))  
dispatch(completeTodo(index))
```

```
const boundAddTodo = text => dispatch(addTodo(text))  
const boundCompleteTodo = index => dispatch(completeTodo(index))
```

# 理解 bindActionCreators

```
function bindActionCreators(actionCreator, dispatch) {
  return function() {
    return dispatch(actionCreator.apply(this, arguments))
  }
}

function bindActionCreators(actionCreators, dispatch) {
  const keys = Object.keys(actionCreators)
  const boundActionCreators = {}
  for (let i = 0; i < keys.length; i++) {
    const key = keys[i]
    const actionCreator = actionCreators[key]
    if (typeof actionCreator === 'function') {
      boundActionCreators[key] = bindActionCreators(actionCreator, dispatch)
    }
  }
  return boundActionCreators
}
```

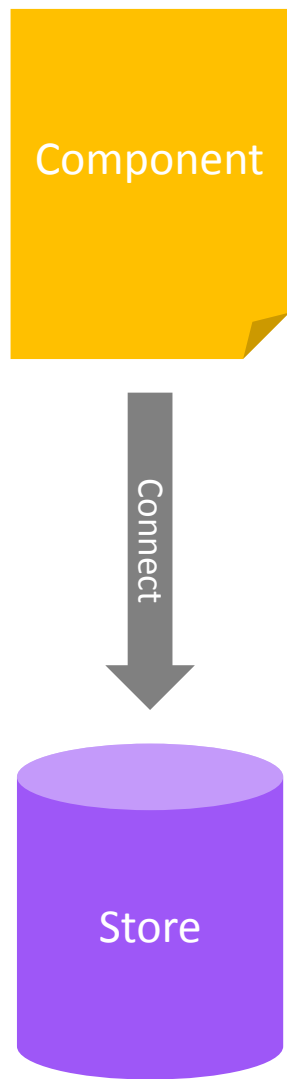


DEMO

# 小结

1. Redux 的基本概念
2. combineReducers
3. bindActionCreators

# Redux ( 3 ) : 在 React 中使用 Redux



```
import { connect } from 'react-redux';

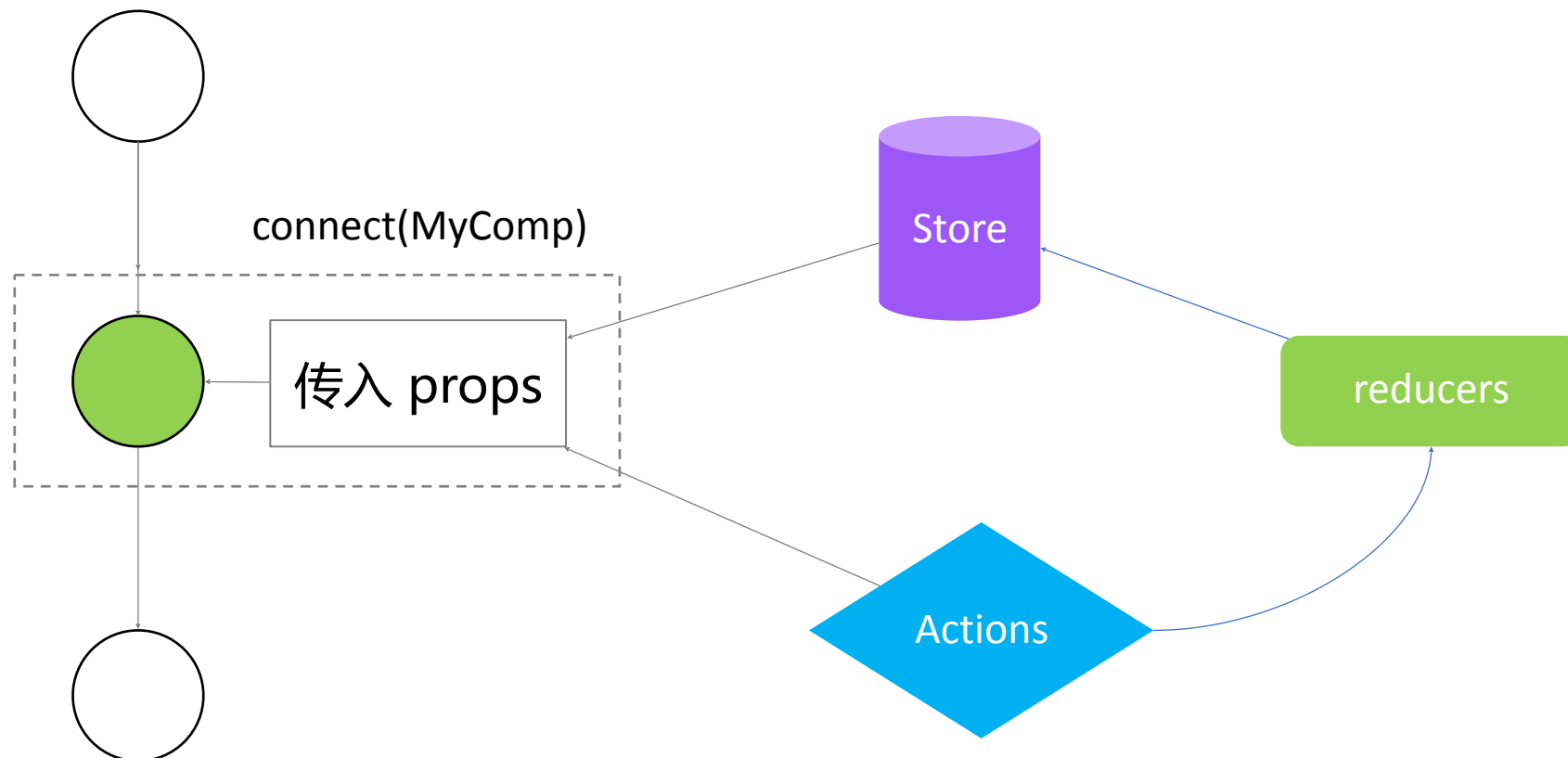
class SidePanel extends Component {
  // ...
}

function mapStateToProps(state) {
  return {
    nextgen: state.nextgen,
    router: state.router,
  };
}

function mapDispatchToProps(dispatch) {
  return {
    actions: bindActionCreators({ ...actions }, dispatch),
  };
}

export default connect(mapStateToProps, mapDispatchToProps)(SidePanel);
```

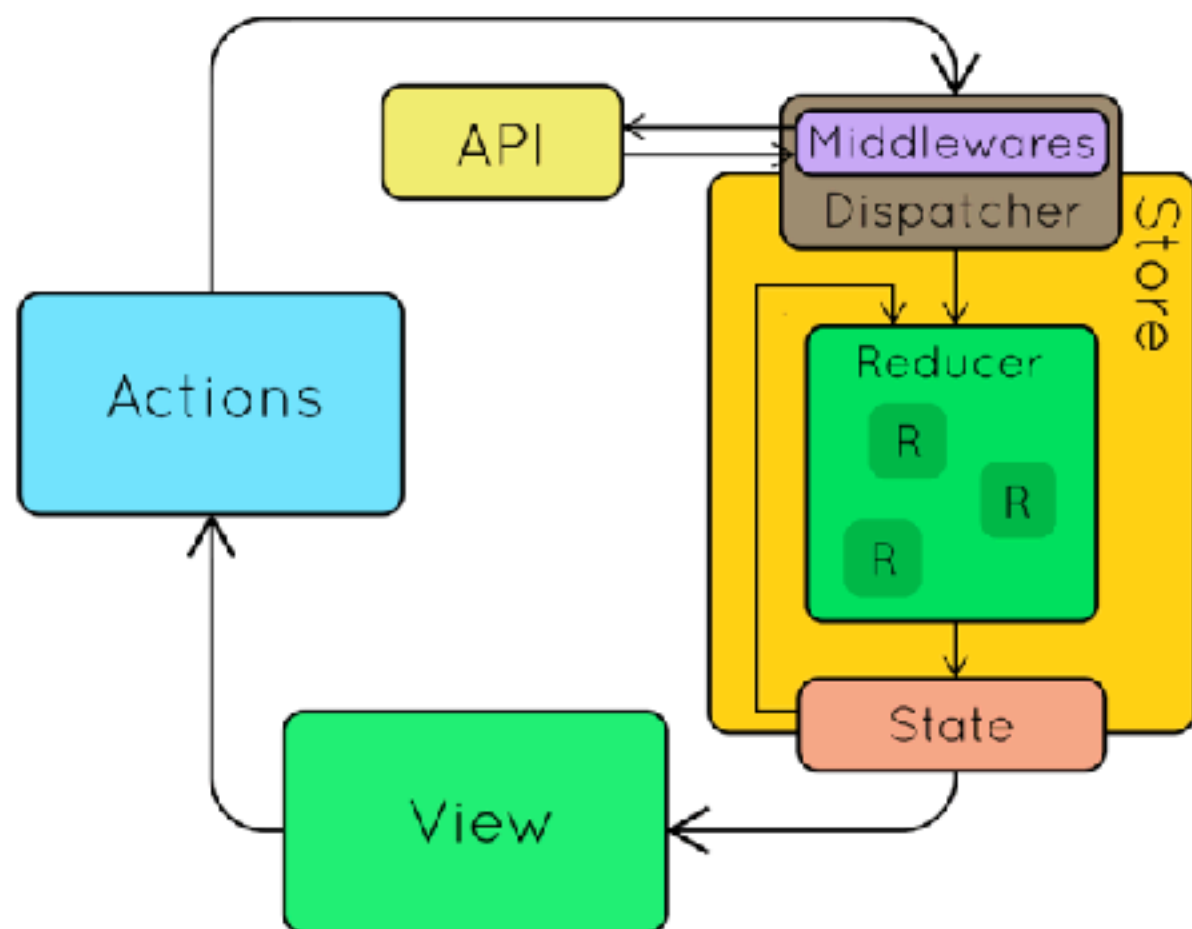
# connect 的工作原理：高阶组件



DEMO

# Redux ( 4 ) : 理解异步 Action , Redux 中间件

# Redux 异步请求



(state, action) => new state

- Store
- Actions
- Reducer
- View
- Middlewares



# Redux 中间件 ( Middleware )

1. 截获 action

2. 发出 action

DEMO

# 小结

1. 异步 action 不是特殊 action ,  
而是多个同步 action 的组合使用
2. 中间件在 dispatcher 中截获 action 做特殊处理

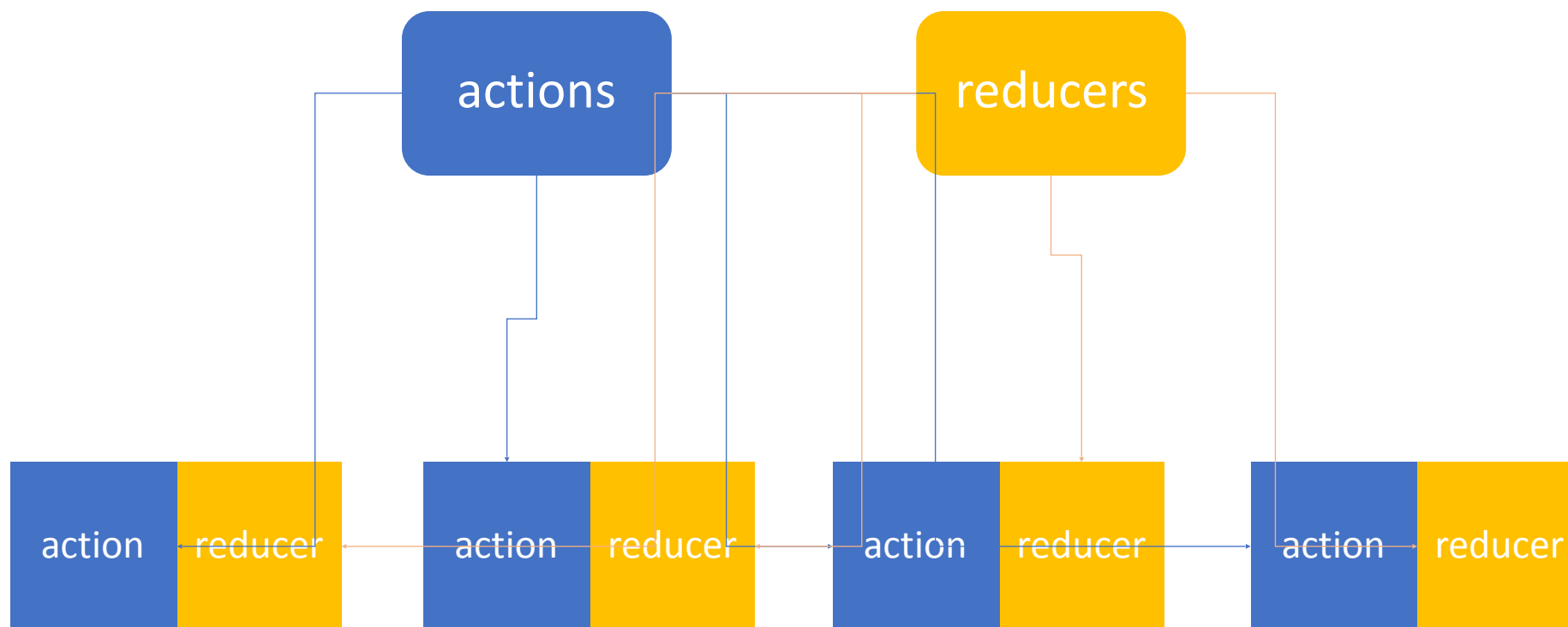
# Redux ( 5 ) : 如何组织 Action 和 Reducer

# “标准”形式 Redux Action 的问题

1. 所有 Action 放一个文件，会无限扩展
2. Action, Reducer 分开，实现业务逻辑时需要来回切换
3. 系统中有哪些 Action 不够直观

```
actions.js
1 import {
2   ADD_PRODUCT_BEGIN,
3   ADD_PRODUCT_SUCCESS,
4   ADD_PRODUCT_FAILURE,
5   ADD_PRODUCT_DISMISS_ERROR,
6
7   SAVE_PRODUCT_BEGIN,
8   SAVE_PRODUCT_SUCCESS,
9   SAVE_PRODUCT_FAILURE,
10  SAVE_PRODUCT_DISMISS_ERROR,
11
12  DELETE_PRODUCT_BEGIN,
13  DELETE_PRODUCT_SUCCESS,
14  DELETE_PRODUCT_FAILURE,
15  DELETE_PRODUCT_DISMISS_ERROR,
16
17  FETCH_PRODUCT_BEGIN,
18  FETCH_PRODUCT_SUCCESS,
19  FETCH_PRODUCT_FAILURE,
20  FETCH_PRODUCT_DISMISS_ERROR,
21
22  FETCH_PRODUCT_LIST_BEGIN,
23  FETCH_PRODUCT_LIST_SUCCESS,
24  FETCH_PRODUCT_LIST_FAILURE,
25  FETCH_PRODUCT_LIST_DISMISS_ERROR,
26
27  UPDATE_PRODUCT_PICTURE_BEGIN,
28  UPDATE_PRODUCT_PICTURE_SUCCESS,
29  UPDATE_PRODUCT_PICTURE_FAILURE,
30  UPDATE_PRODUCT_PICTURE_DISMISS_ERROR,
31
32  MOVE_PRODUCT_BEGIN,
33  MOVE_PRODUCT_SUCCESS,
34  MOVE_PRODUCT_FAILURE,
35  MOVE_PRODUCT_DISMISS_ERROR,
36
37  FETCH_HOT_PRODUCT_BEGIN,
38  FETCH_HOT_PRODUCT_SUCCESS,
39  FETCH_HOT_PRODUCT_FAILURE,
40  FETCH_HOT_PRODUCT_DISMISS_ERROR,
41
42  CHANGE_PRODUCT_OWNER_BEGIN,
43  CHANGE_PRODUCT_OWNER_SUCCESS,
44  CHANGE_PRODUCT_OWNER_FAILURE,
```

# 新的方式：单个 action 和 reducer 放在同一个文件



# 新的方式：每个文件一个 Action

1. 易于开发：不用在 action 和 reducer 文件间来回切换
2. 易于维护：每个 action 文件都很小，容易理解
3. 易于测试：每个业务逻辑只需对应一个测试文件
4. 易于理解：文件名就是 action 名字，文件列表就是 action 列表

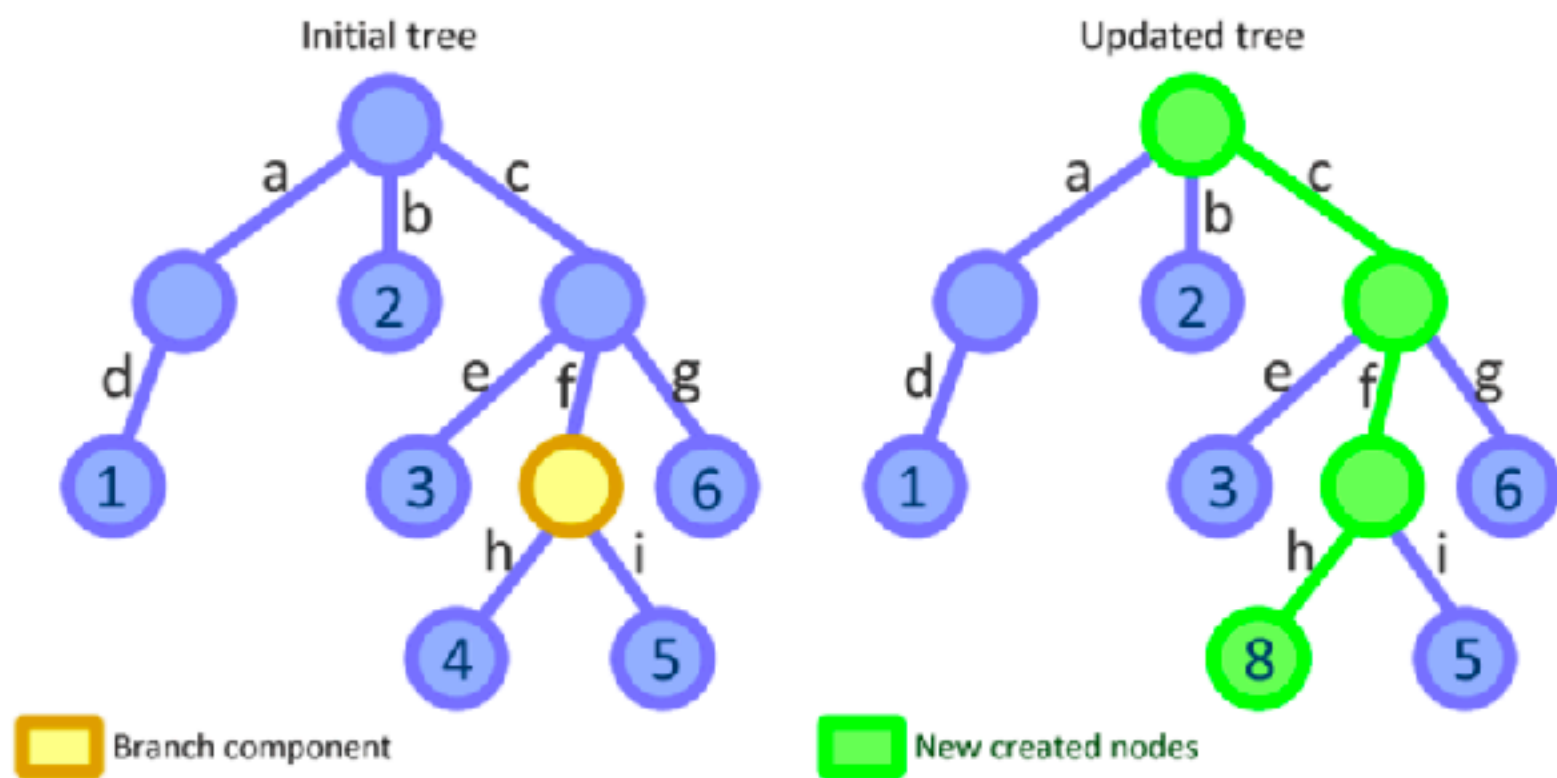
```
import {
  COUNTER_PLUS_ONE,
} from './constants';
export function counterPlusOne() {
  return {
    type: COUNTER_PLUS_ONE,
  };
}
export function reducer(state, action) {
  switch (action.type) {
    case COUNTER_PLUS_ONE:
      return {
        ...state,
        count: state.count + 1,
      };
    default:
      return state;
  }
}
```

DEMO



# Redux ( 6 ) : 理解 Redux 的运行基础 , 不可变数据 ( Immutability )

# 不可变数据 (immutable data)



# 为何需要不可变数据

1. 性能优化
2. 易于调试和跟踪
3. 易于推测

# 如何操作不可变数据

- 1.原生写法：`{ ... }`, `Object.assign`
- 2.`immutability-helper`
- 3.`immer`

# 原生写法：{ ... }, Object.assign

```
const state = { filter: 'completed', todos: [
  'Learn React'
]};
const newState = { ...state, todos: [
  ...state.todos,
  'Learn Redux'
]};
const newState2 = Object.assign({}, state, { todos: [
  ...state.todos,
  'Learn Redux'
]});
```

# immutability-helper

```
import update from 'immutability-helper';

const state = { filter: 'completed', todos: [
  'Learn React'
]};

const newState = update(state, { todos: {$push: ['Learn Redux']}});
```

<https://github.com/kolodny/immutability-helper>

# immer

```
import produce from 'immer';

const state = { filter: 'completed', todos: [
  'Learn React'
]};

const newState = produce(state, draftState => {
  draftState.todos.push('Learn Redux. ');
})
```

<https://github.com/mweststrate/immer>

# 小结

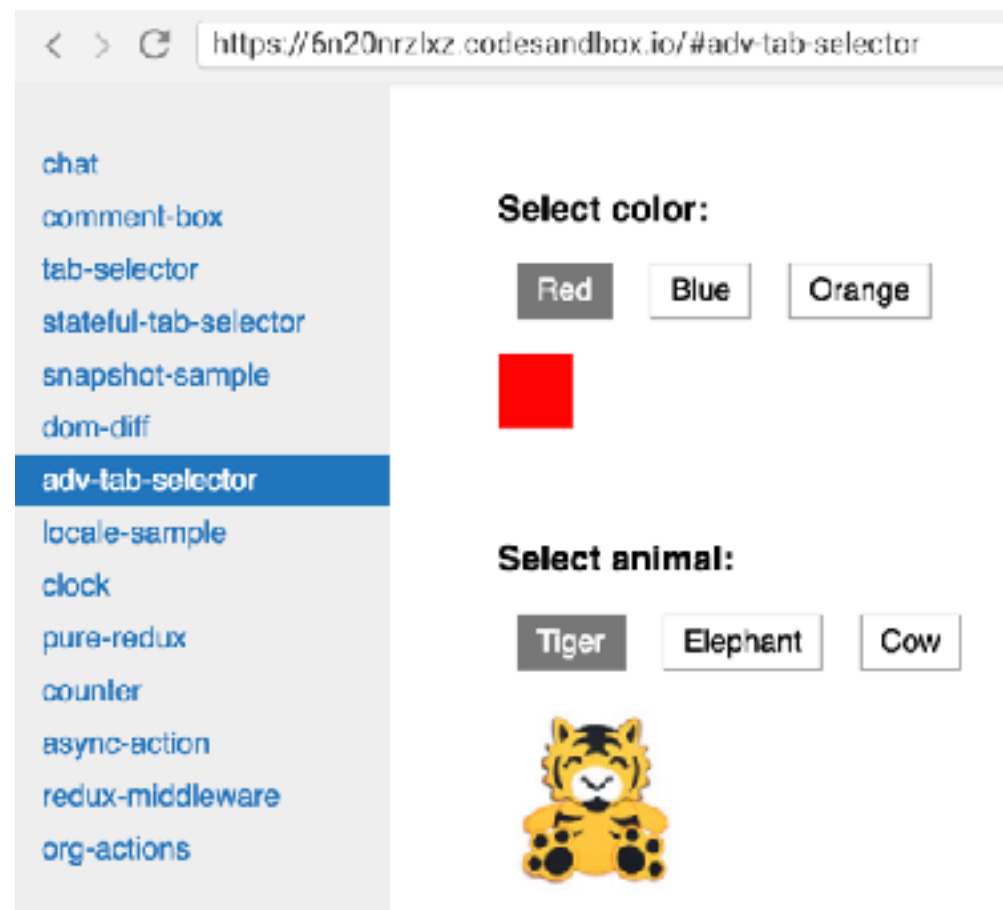
1. 不可变数据的含义
2. Redux 为什么使用不可变数据
3. 如何操作不可变数据



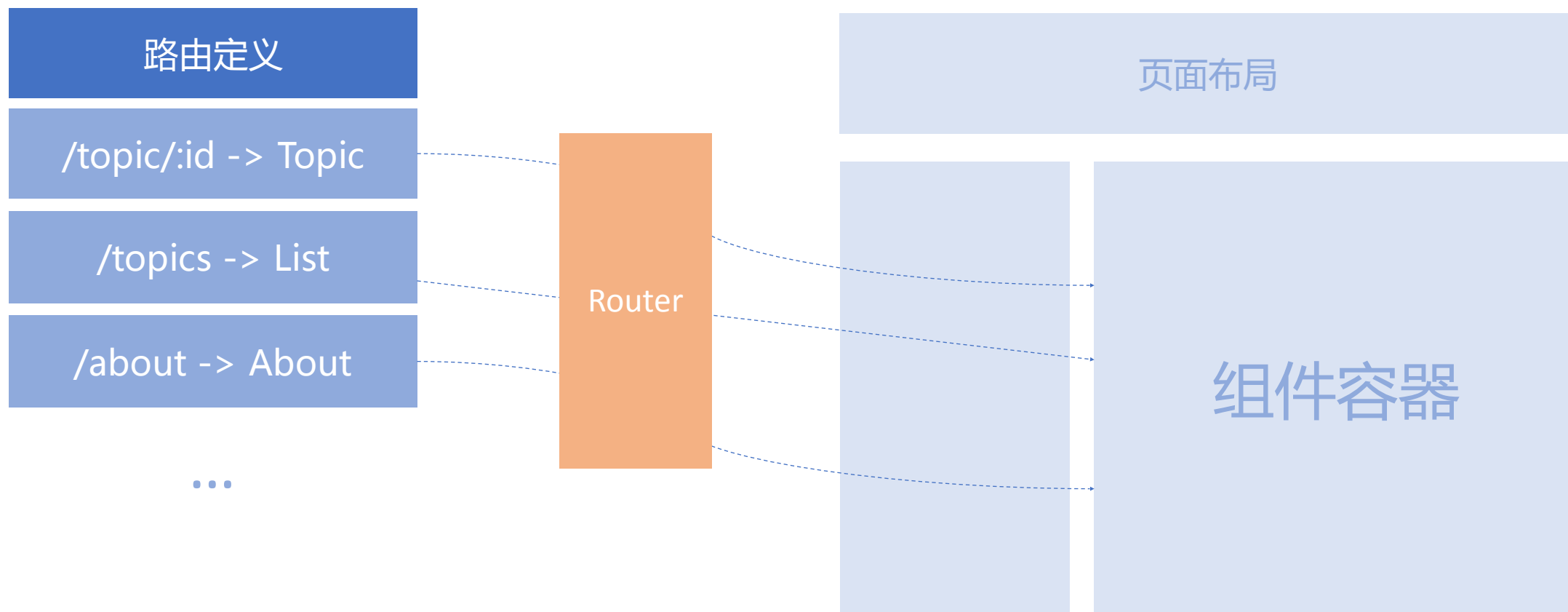
# React Router (1): 路由不只是页面切换，更是代码组织方式

# 为什么需要路由

1. 单页应用需要进行页面切换
2. 通过 URL 可以定位到页面
3. 更有语义的组织资源



# 路由实现的基本架构



# React Router 的实现

```
<Router>
  <div>
    <ul id="menu">
      <li><Link to="/home">Home</Link></li>
      <li><Link to="/hello">Hello</Link></li>
      <li><Link to="/about">About</Link></li>
    </ul>

    <div id="page-container">
      <Route path="/home" component={Home} />
      <Route path="/hello" component={Hello} />
      <Route path="/about" component={About} />
    </div>
  </div>
</Router>
```

# React Router 的特性

1. 声明式路由定义

2. 动态路由

```
const App = () => (  
  <div>  
    <nav>  
      <Link to="/dashboard">Dashboard</Link>  
    </nav>  
    <div>  
      <Route path="/dashboard" component={Dashboard}/>  
    </div>  
  </div>  
)
```

# 三种路由实现方式

1. URL 路径
2. hash 路由
3. 内存路由

# 基于路由配置进行资源组织

1. 实现业务逻辑的松耦合
2. 易于扩展，重构和维护
3. 路由层面实现 Lazy Load

# React Router API

1. `<Link>` : 普通链接，不会触发浏览器刷新
2. `<NavLink>` : 类似 Link 但是会添加当前选中状态
3. `<Prompt>` : 满足条件时提示用户是否离开当前页面
4. `<Redirect>` : 重定向当前页面，例如登录判断
5. `<Route>` : 路由配置的核心标记，路径匹配时显示对应组件
6. `<Switch>` : 只显示第一个匹配的路由



# <Link> : 普通链接，不会触发浏览器刷新

```
import { Link } from 'react-router-dom'  
  
<Link to="/about">About</Link>
```

<NavLink> : 类似 Link 但是会增加当前选中状态

```
<NavLink  
  to="/faq"  
  activeClassName="selected"  
>FAQs</NavLink>
```

<Prompt> : 满足条件时提示用户是否离开当前页面

```
import { Prompt } from 'react-router'

<Prompt
  when={formIsHalfFilledOut}
  message="Are you sure you want to leave?"
/>
```

<Redirect> : 重定向当前页面，例如登录判断

```
import { Route, Redirect } from 'react-router'

<Route exact path="/" render={() => (
  loggedIn ? (
    <Redirect to="/dashboard"/>
  ) : (
    <PublicHomePage/>
  )
)} />
```

# <Route> : 路径匹配时显示对应组件

```
import { BrowserRouter as Router, Route } from 'react-router-dom'  
  
<Router>  
  <div>  
    <Route exact path="/" component={Home}/>  
    <Route path="/news" component={NewsFeed}/>  
  </div>  
</Router>
```

# <Switch> : 只显示第一个匹配的路由

```
import { Switch, Route } from 'react-router'  
  
<Switch>  
  <Route exact path="/" component={Home}/>  
  <Route path="/about" component={About}/>  
  <Route path="/:user" component={User}/>  
  <Route component={NoMatch}/>  
</Switch>
```

# 小结

1. 前端路由是什么
2. React Router 如何实现路由
3. 基于路由思考资源的组织
4. React Router 核心 API

## React Router (2): 参数定义, 嵌套路由的使用场景



# 通过 URL 传递参数

1. 如何通过 URL 传递参数：`<Route path= "/topic/:id" ... />`
2. 如何获取参数：`this.props.match.params`
3. <https://github.com/pillarjs/path-to-regexp>

# 何时需要 URL 参数

## 页面状态尽量通过 URL 参数定义



The screenshot shows a web browser with the URL `https://app.guanxinyi.com/timing/list/201608/`. The page displays a sidebar menu on the left and a main content area titled "考勤信息" (Attendance Information). The main area includes a filter for "2016年8月" and "全部方案", and a table with columns: "员工号", "姓名", "考勤类型", "日期", "数量", and "备注". The table is currently empty, showing "暂无数据" (No data).

URL: `https://app.guanxinyi.com/timing/list/201608/`

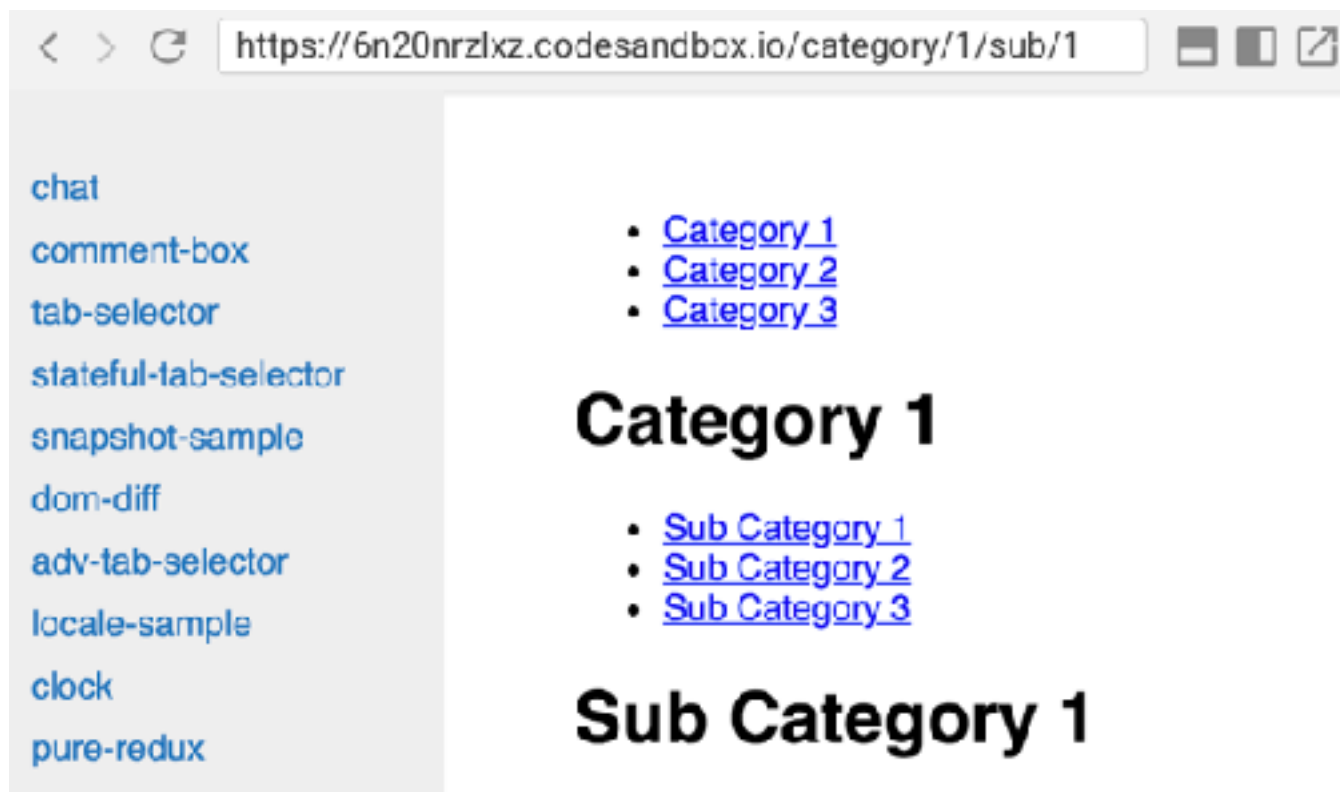
考勤信息

2016年8月 全部方案

员工号	姓名	考勤类型	日期	数量	备注
暂无数据					

# 嵌套路由

1. 每个 React 组件都可以是路由容器
2. React Router 的声明式语法可以方便的定义嵌套路由



# 小结

1. 路由参数的传递
2. 嵌套路由的实现

UI 组件库对比和介绍 : Ant.Design, Material UI, Semantic UI

# Ant Design

一个服务于企业级产品的设计体系，基于『确定』和『自然』的设计价值观和模块化的解决方案，让设计者专注于更好的用户体验。

[开始使用](#)[设计语言](#)[Star](#)

30231

Ant Design

[介绍](#)[设计价值观](#)[实践案例](#)

原则

[亲密性](#)[对齐](#)[对比](#)[重复](#)[直截了当](#)[简化交互](#)[足不出户](#)[提供邀请](#)[巧用过放](#)[即时反应](#)

## 足不出户

能在这个页面解决的问题，就不要去其它页面解决，因为任何页面刷新和跳转都会引起变化盲视（Change Blindness），导致用户心流（Flow）被打断。频繁的页面刷新和跳转，就像在看戏时，演员说完一行台词就安排一次谢幕一样。

**变换盲视（Change Blindness）**：指视觉场景中的某些变化并未被观察者注意到的心理现象。产生这种现象的原因包括场景中的障碍物、眼球运动、地点的变化，或者是缺乏注意力等。——摘自《维基百科》

**心流（Flow）**：也有别号以化境（Zone）表示，亦有人翻译为神游状态，定义是一种将个人精神力完全投注在某种活动上的感觉；心流产生时同时会有高度的兴奋及充实感。——摘自《维基百科》

## 覆盖层



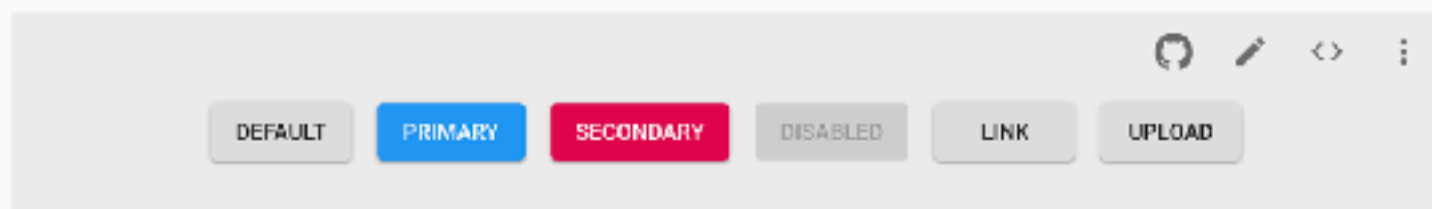


# MATERIAL-UI

React components that implement  
Google's Material Design.

## Contained Buttons

**Contained buttons** are high-emphasis, distinguished by their use of elevation and fill. They contain actions that are primary to your app.

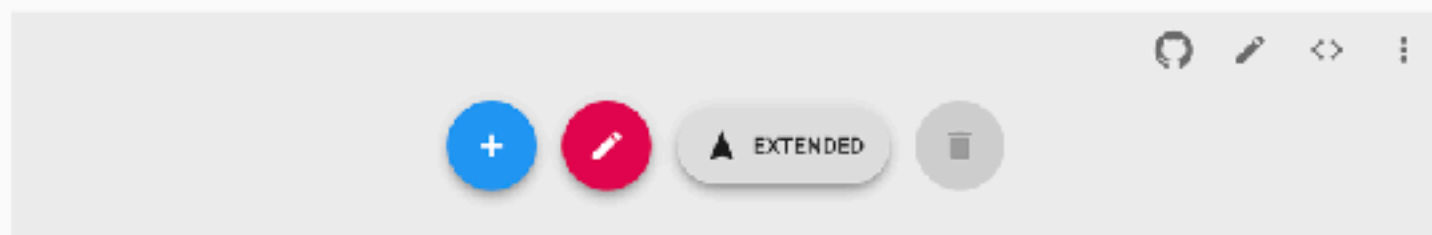


## Floating Action Buttons

A **floating action button** (FAB) performs the primary, or most common, action on a screen. It appears in front of all screen content, typically as a circular shape with an icon in its center. FABs come in three types: regular, mini, and extended.

Only use a FAB if it is the most suitable way to present a screen's primary action.

Only one floating action button is recommended per screen to represent the most common action.





☰ Menu

🐦 Tweet

★ Star 41,864

🌐 English

2.3.2

# Semantic UI

User Interface is the language of the web

Get Started

New in 2.3



# 选择 UI 库的考虑因素

1. 组件库是否齐全
2. 样式风格是否符合业务需求
3. API 设计是否便捷和灵活
4. 技术支持是否完善
5. 开发是否活跃

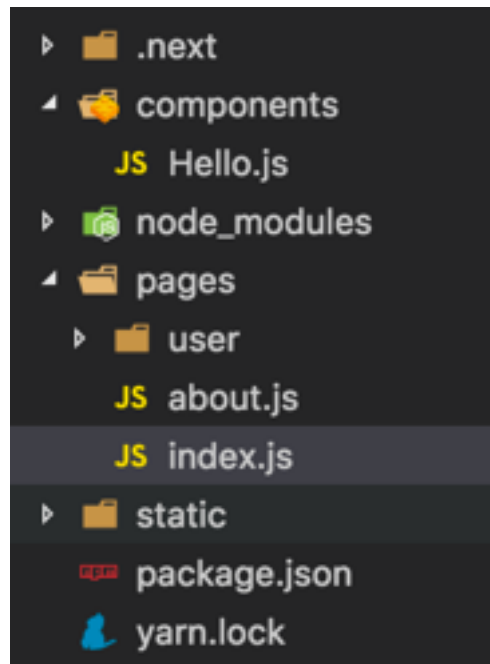
DEMO

# 使用 Next.js 创建 React 同构应用

# 什么是同构应用



# 创建页面



1. 页面就是 pages 目录下的一个组件
2. static 目录映射静态文件
3. page 具有特殊静态方法 `getInitialProps`

# 在页面中使用其它 React 组件

1. 页面也是标准的 node 模块，可使用其它 React 组件
2. 页面会针对性打包，仅包含其引入的组件

# 使用 Link 实现同构路由

```
import Link from 'next/link'

export default () =>
  <div>
    Click{' '}
    <Link href="/about">
      <a>here</a>
    </Link>{' '}
    to read more
  </div>
```

1. 使用 “next/link” 定义链接
2. 点击链接时页面不会刷新
3. 使用 prefetch 预加载目标资源
4. 使用 replace 属性替换 URL

# 动态加载页面

```
import dynamic from 'next/dynamic'

const DynamicComponentWithCustomLoading = dynamic(
  import('../components/hello2'),
  {
    loading: () => <p>...</p>
  }
)

export default () =>
  <div>
    <Header />
    <DynamicComponentWithCustomLoading />
    <p>HOME PAGE is here!</p>
  </div>
```



DEMO

# 小结

1. 同构应用的概念
2. Next.js 的基本用法

# 使用 Jest, Enzyme 等工具进行单元测试

# React 让前端单元测试变得容易

1. React 应用很少需要访问浏览器 API
2. 虚拟 DOM 可以在 NodeJS 环境运行和测试
3. Redux 隔离了状态管理，纯数据层单元测试

# 单元测试涉及的工具

1. Jest: Facebook 开源的 JS 单元测试框架
2. JS DOM: 浏览器环境的 NodeJS 模拟
3. Enzyme : React 组件渲染和测试
- 4.nock: 模拟 HTTP 请求
- 5.sinon: 函数模拟和调用跟踪
- 6.istanbul: 单元测试覆盖率

# Jest

## Delightful JavaScript Testing

```
1 const add = require('./add');
2 describe('add', () => {
3   it('should add two numbers', () => {
4     expect(add(1, 2)).toBe(3);
5   });
6 });
```

```
PASS ./add-test.js
  add
    ✓ should add two numbers (6ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:  0 total
Time:        0.903s
Ran all test suites.
```



jsdom

```
const JSDOM = require('jsdom').JSDOM;

global.window = new JSDOM('<!DOCTYPE html><div id="react-root"></div>').window;
global.document = window.document;
global.navigator = window.navigator;
global.HTMLElement = window.HTMLElement;
```



# Enzyme



```
import React from 'react';
import { shallow } from 'enzyme';
import { DefaultPage } from 'src/features/home/DefaultPage';

describe('home/DefaultPage', () => {
  it('renders node with correct class name', () => {
    const pageProps = {
      home: {},
      actions: {},
    };
    const renderedComponent = shallow(
      <DefaultPage {...pageProps} />
    );

    expect(
      renderedComponent.find('.home-default-page').getElement()
    ).to.exist;
  });
});
```

## Shallow Rendering

```
import { shallow } from 'enzyme';

const wrapper = shallow(<MyComponent />);
// ...
```

## Full Rendering

```
import { mount } from 'enzyme';

const wrapper = mount(<MyComponent />);
// ...
```

## Static Rendering

```
import { render } from 'enzyme';

const wrapper = render(<MyComponent />);
// ...
```



# Nock

```
it('handles fetchRedditReactjsList failure', () => {
  nock('http://www.reddit.com/')
    .get('/r/reactjs.json')
    .reply(500, null);
  const store = mockStore({ redditReactjsList: [] });

  return store.dispatch(fetchRedditReactjsList())
    .catch(() => {
      const actions = store.getActions();
      expect(actions[0]).to.have.property('type', HOME_FETCH_REDDIT_REACTJS_LIST_BEGIN);
      expect(actions[1]).to.have.property('type', HOME_FETCH_REDDIT_REACTJS_LIST_FAILURE);
      expect(actions[1]).to.have.nested.property('data.error').that.exist;
    });
});
```

# Sinon

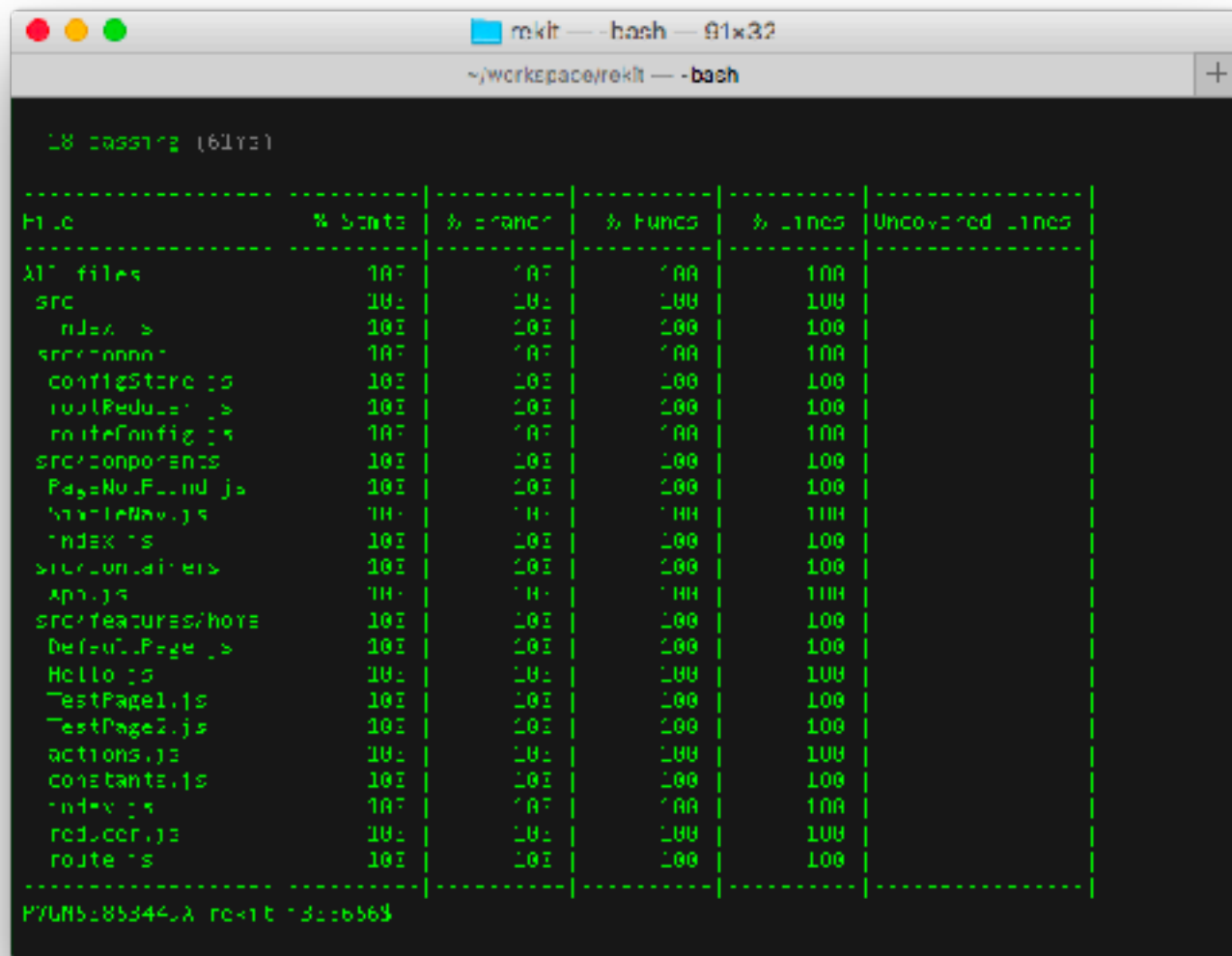
```
it('counter actions are called when buttons clicked', () => {
  const pageProps = {
    home: {},
    actions: {
      counterPlusOne: sinon.spy(),
      counterMinusOne: sinon.spy(),
      resetCounter: sinon.spy(),
      fetchRedditReactjsList: sinon.spy(),
    },
  };
  const renderedComponent = shallow(
    <DefaultPage {...pageProps} />
  );
  renderedComponent.find('.btn-plus-one').simulate('click');
  renderedComponent.find('.btn-minus-one').simulate('click');
  renderedComponent.find('.btn-reset-counter').simulate('click');
  renderedComponent.find('.btn-fetch-reddit').simulate('click');
  expect(pageProps.actions.counterPlusOne).to.have.property('callCount', 1);
  expect(pageProps.actions.counterMinusOne).to.have.property('callCount', 1);
  expect(pageProps.actions.resetCounter).to.have.property('callCount', 1);
  expect(pageProps.actions.fetchRedditReactjsList).to.have.property('callCount', 1);
});
```

# Istanbul

```
$ npm install -g istanbul
$ cd /path/to/your/source/root
$ istanbul cover test.js
```

```
function meaningOfLife() {
  return 42;
}

function meaningOfLife() {
  __cov_ECTNDGoq6USQlIaViK8Qyw.f['1']++;
  __cov_ECTNDGoq6USQlIaViK8Qyw.s['2']++;
  return 42;
}
```



Terminal window showing the output of the Istanbul coverage tool. The report is a table with columns: File, % Stmts, % Branch, % Funcs, % Lines, and Uncovered Lines. The report shows 100% coverage for all metrics across all files in the project.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
all files	100	100	100	100	
src	100	100	100	100	
index.js	100	100	100	100	
src/constants	100	100	100	100	
configStore.js	100	100	100	100	
routeReducer.js	100	100	100	100	
routeConfig.js	100	100	100	100	
src/components	100	100	100	100	
PageNotFound.js	100	100	100	100	
HomeNav.js	100	100	100	100	
index.js	100	100	100	100	
src/layouts	100	100	100	100	
App.js	100	100	100	100	
src/features/home	100	100	100	100	
DefaultPage.js	100	100	100	100	
Hello.js	100	100	100	100	
TestPage1.js	100	100	100	100	
TestPage2.js	100	100	100	100	
actions.js	100	100	100	100	
constants.js	100	100	100	100	
index.js	100	100	100	100	
reducer.js	100	100	100	100	
route.js	100	100	100	100	

DEMO

# 小结

1. React 让单元测试变得更容易
2. 单元测试所涉及的工具和技术

常用开发调试工具：ESLint, Prettier, React DevTool, Redux  
DevTool



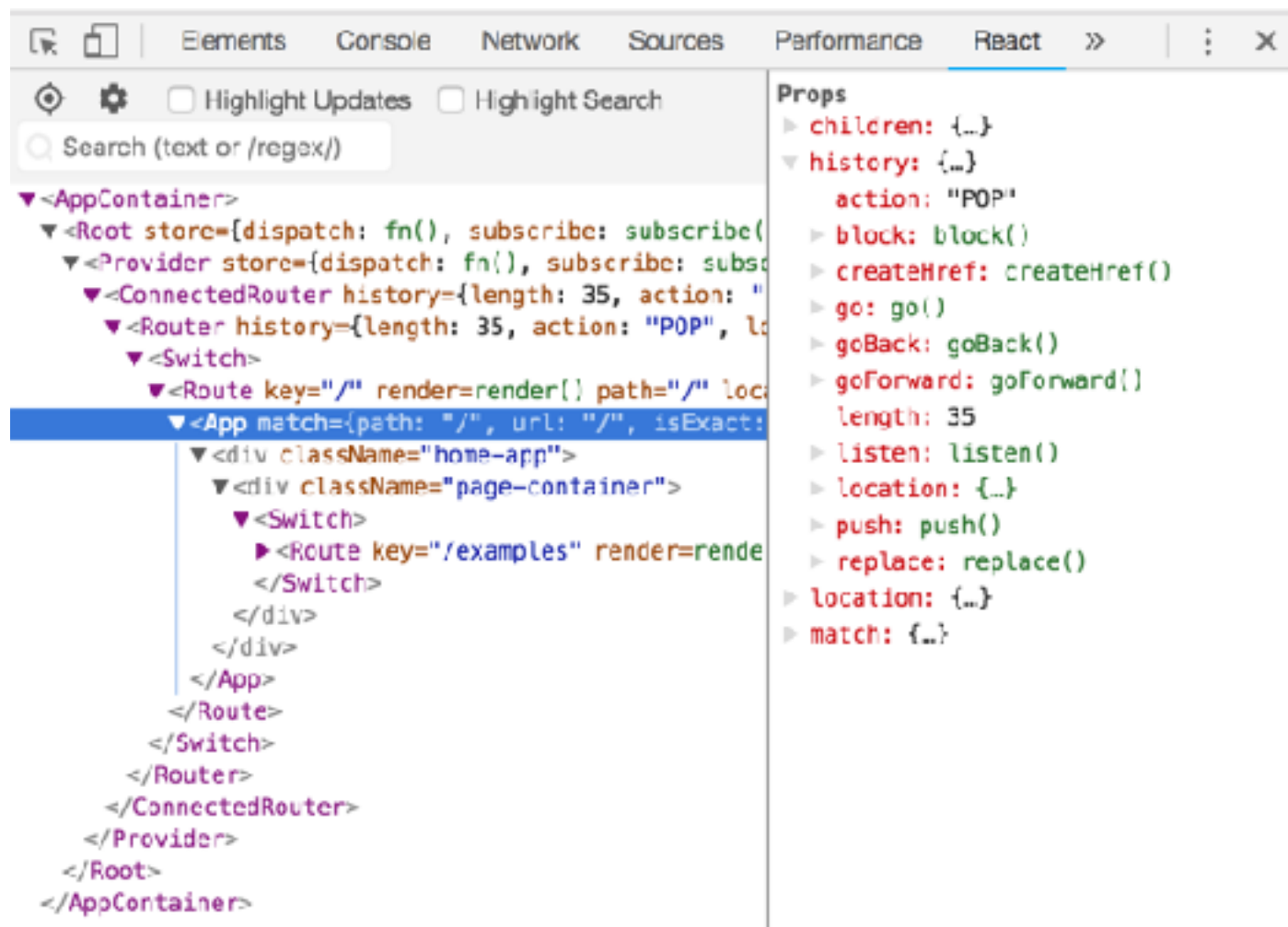
1. 使用 `.eslintrc` 进行规则的配置
2. 使用 `airbnb` 的 JavaScript 代码风格



1. 代码格式化的神器
2. 保证更容易写出风格一致的代码



# React Dev Tools



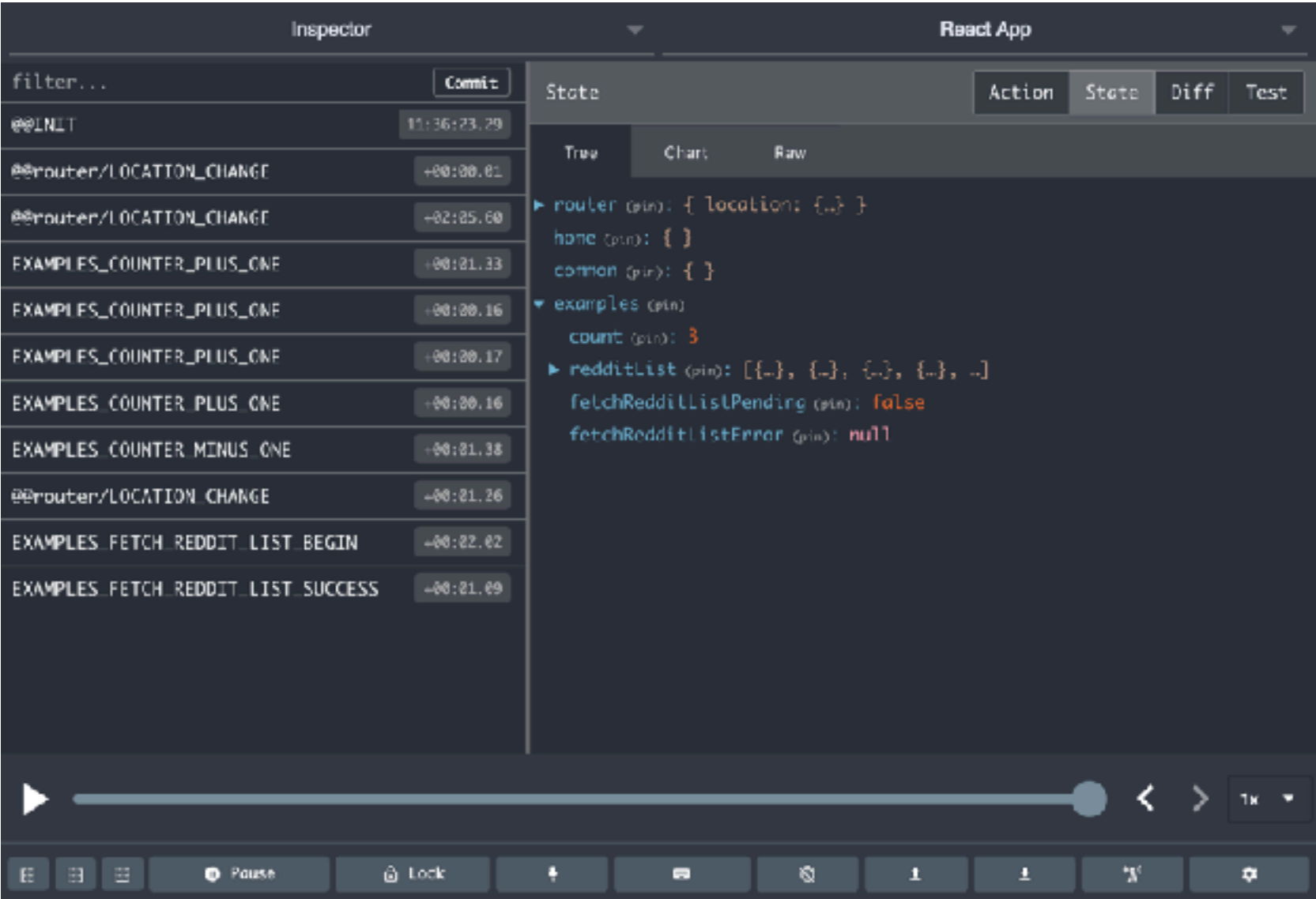
The screenshot displays the React Dev Tools interface. The top navigation bar includes tabs for Elements, Console, Network, Sources, Performance, and React. Below the navigation bar, there are options for 'Highlight Updates' and 'Highlight Search', along with a search input field labeled 'Search (text or /regex/)'. The main area is split into two panels. The left panel shows the component tree, with the following structure visible:

```
▼ <AppContainer>
  ▼ <Root store={dispatch: fn(), subscribe: subscribe()}
    ▼ <Provider store={dispatch: fn(), subscribe: subscribe()}
      ▼ <ConnectedRouter history={length: 35, action: "POP"}
        ▼ <Router history={length: 35, action: "POP", location: {}}
          ▼ <Switch>
            ▼ <Route key="/" render=render() path="/" location={}/>
              ▼ <App match={path: "/", url: "/", isExact: true}>
                ▼ <div className="home-app">
                  ▼ <div className="page-container">
                    ▼ <Switch>
                      ▶ <Route key="/examples" render=render() path="/examples" location={}/>
                    </Switch>
                  </div>
                </div>
              </App>
            </Route>
          </Switch>
        </Router>
      </ConnectedRouter>
    </Provider>
  </Root>
</AppContainer>
```

The right panel shows the props for the selected component, including:

```
Props
▶ children: {...}
▼ history: {...}
  ▶ action: "POP"
  ▶ block: block()
  ▶ createHref: createHref()
  ▶ go: go()
  ▶ goBack: goBack()
  ▶ goForward: goForward()
  ▶ length: 35
  ▶ listen: listen()
  ▶ location: {...}
  ▶ push: push()
  ▶ replace: replace()
▶ location: {...}
▶ match: {...}
```

# Redux Dev Tools

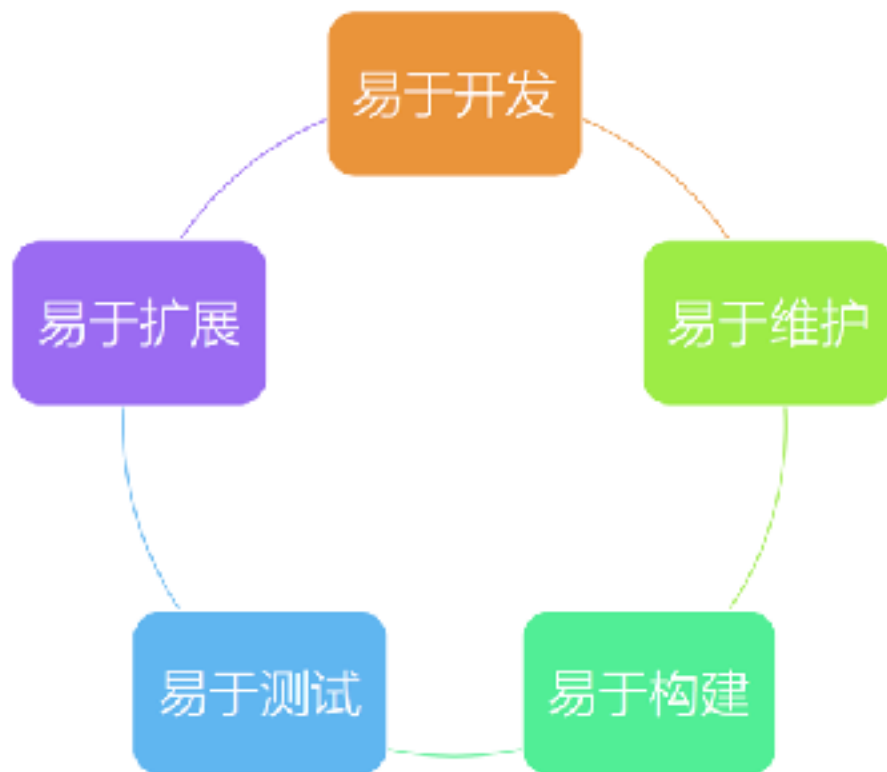


DEMO

前端项目的理想架构：

可维护，可扩展，可测试，易开发，易构建

# 理想架构



# 易于开发

1. 开发工具是否完善
2. 生态圈是否繁荣
3. 社区是否活跃

# 易于扩展

1. 增加新功能是否容易
2. 新功能是否会显著增加系统复杂度

# 易于维护

1. 代码是否容易理解
2. 文档是否健全



# 易于测试

1. 功能的分层是否清晰
2. 副作用少
3. 尽量使用纯函数

# 易于构建

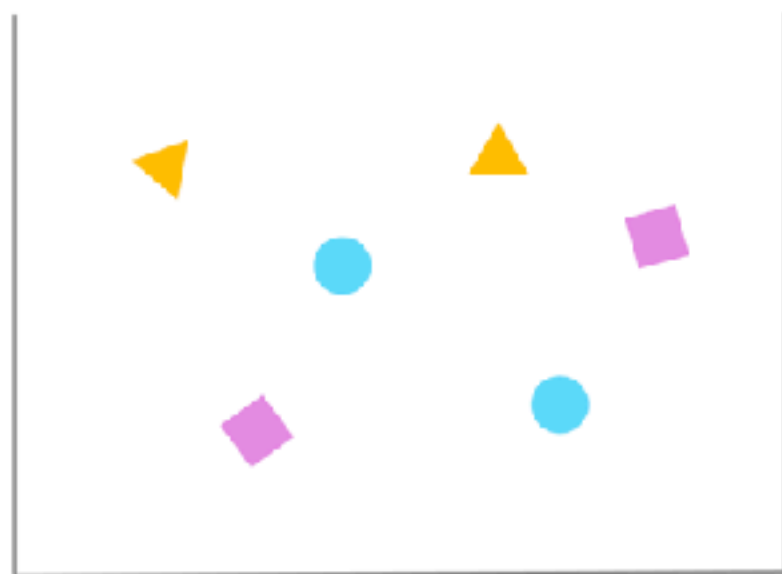
1. 使用通用技术和架构
2. 构建工具的选择

拆分复杂度（1）：

按领域模型（feature）组织代码，降低耦合度

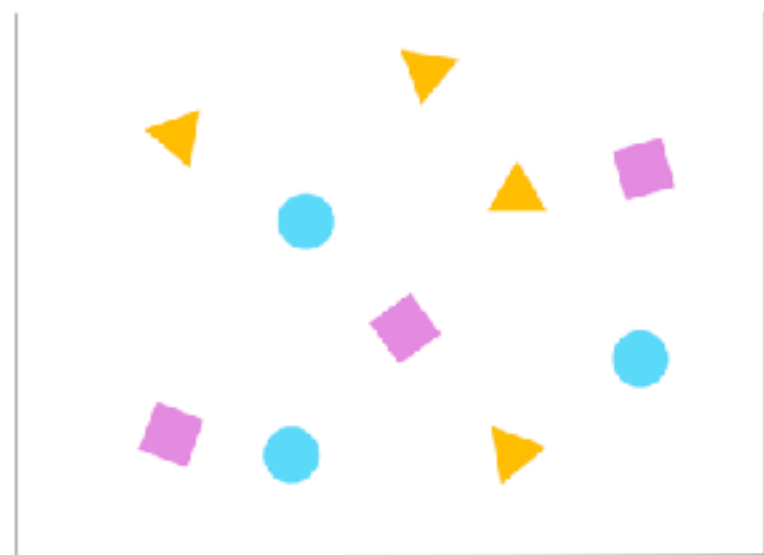
# 项目初期：规模小，模块关系清晰

- ▲ Reducer
- Action
- Component

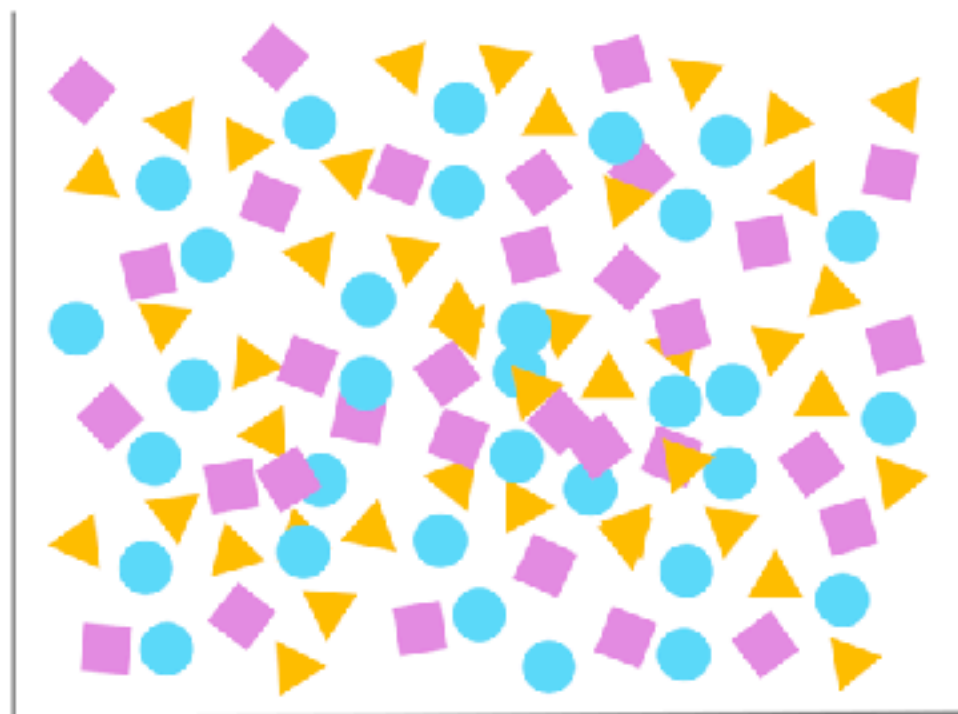


Application

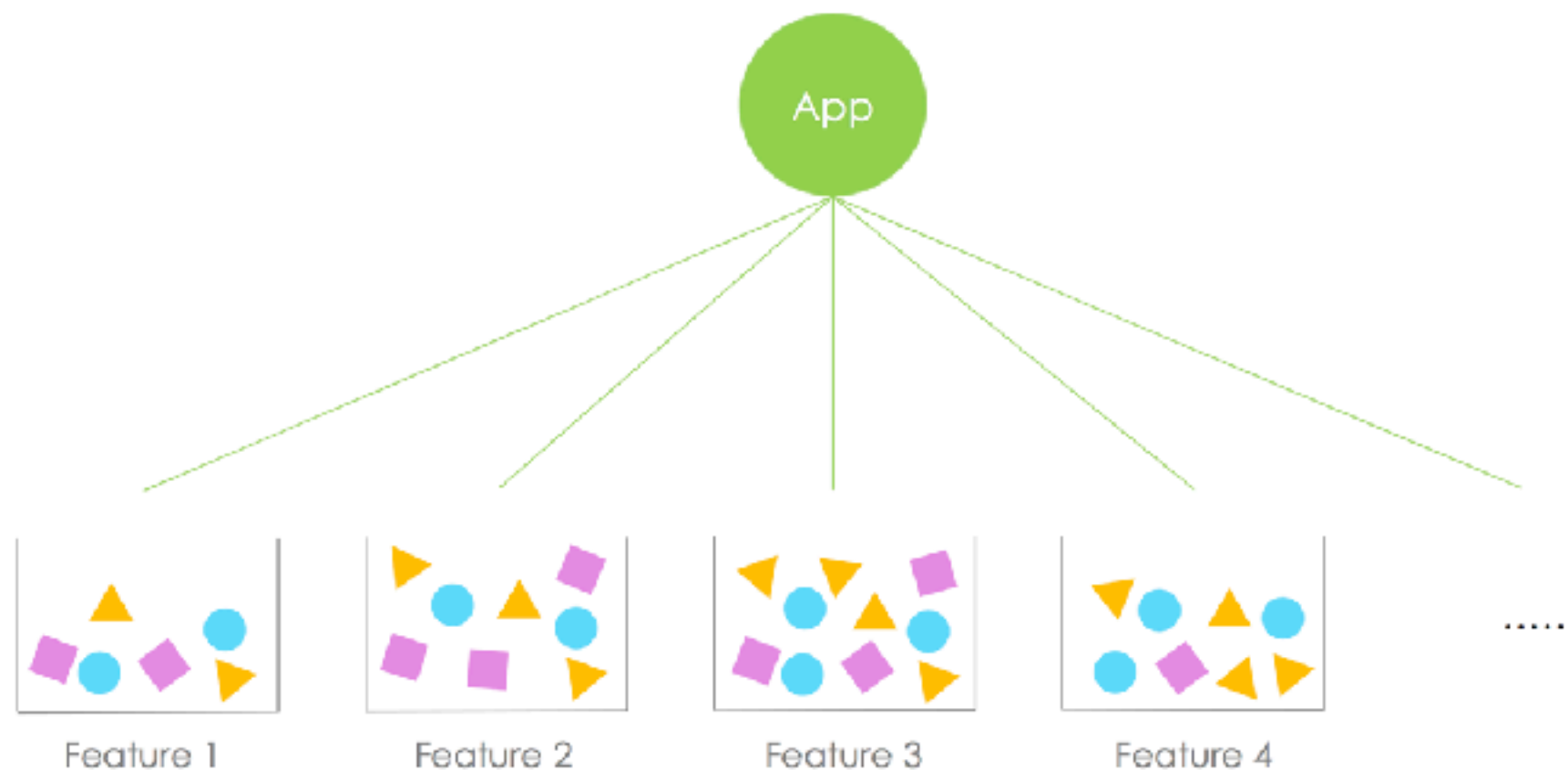
项目逐渐复杂，添加了更多组件和其他元素



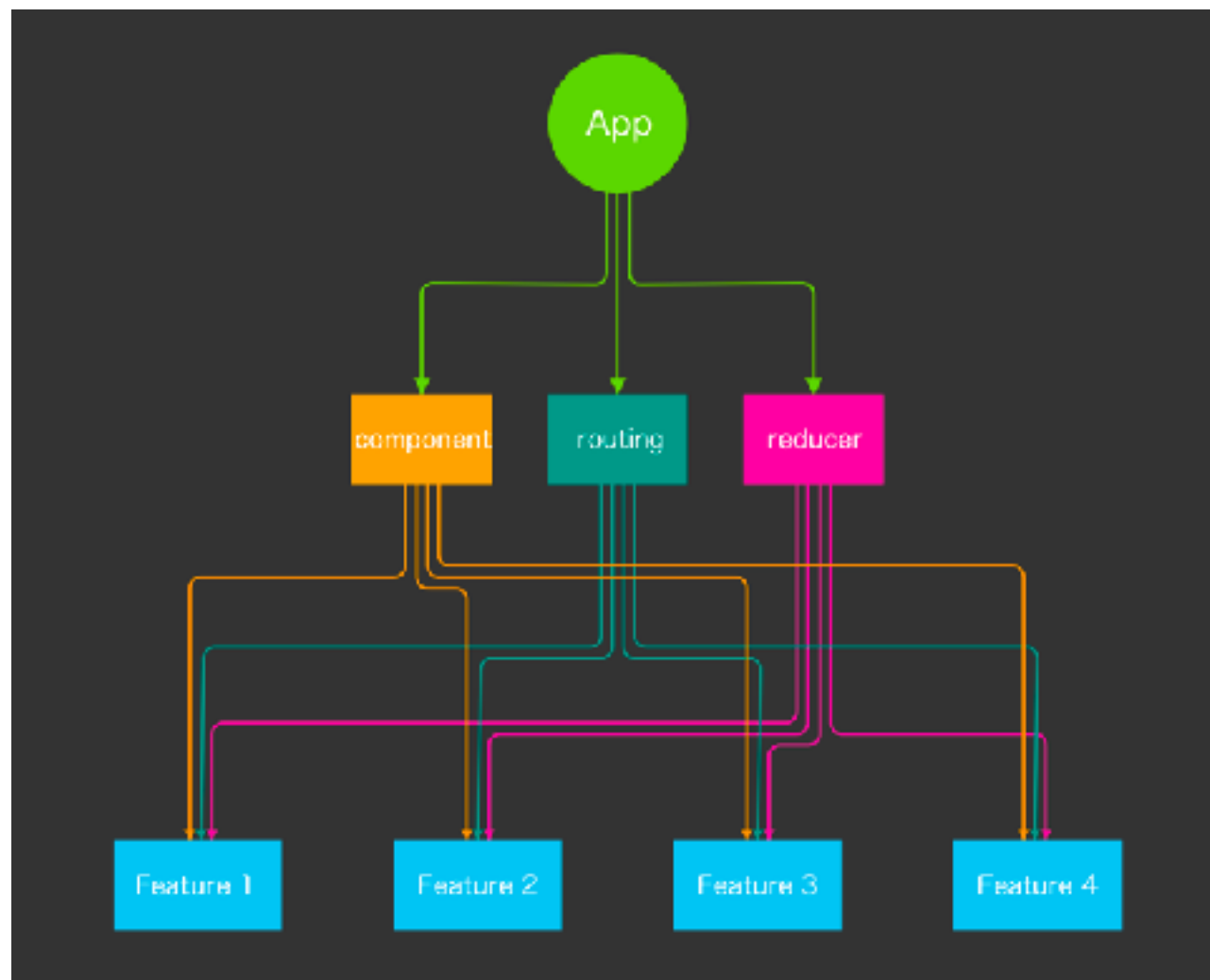
项目收尾：文件结构，模块依赖错综复杂



# 将业务逻辑拆分成高内聚松耦合的模块



# 通过 React 技术栈实现





# 小结

1. 大型前端应用需要拆分复杂度
2. 使用 React 技术栈如何实现

拆分复杂度（2）：

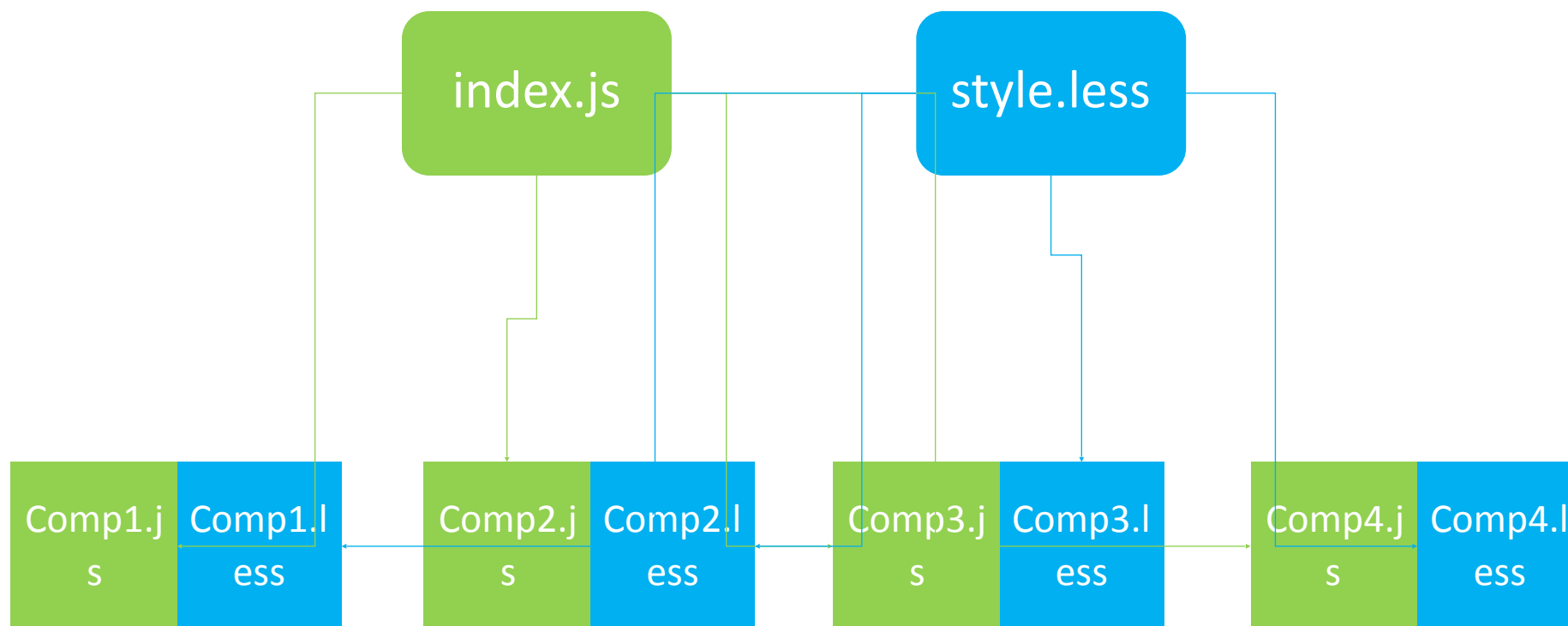
如何组织 component, action 和 reducer

# 文件夹结构

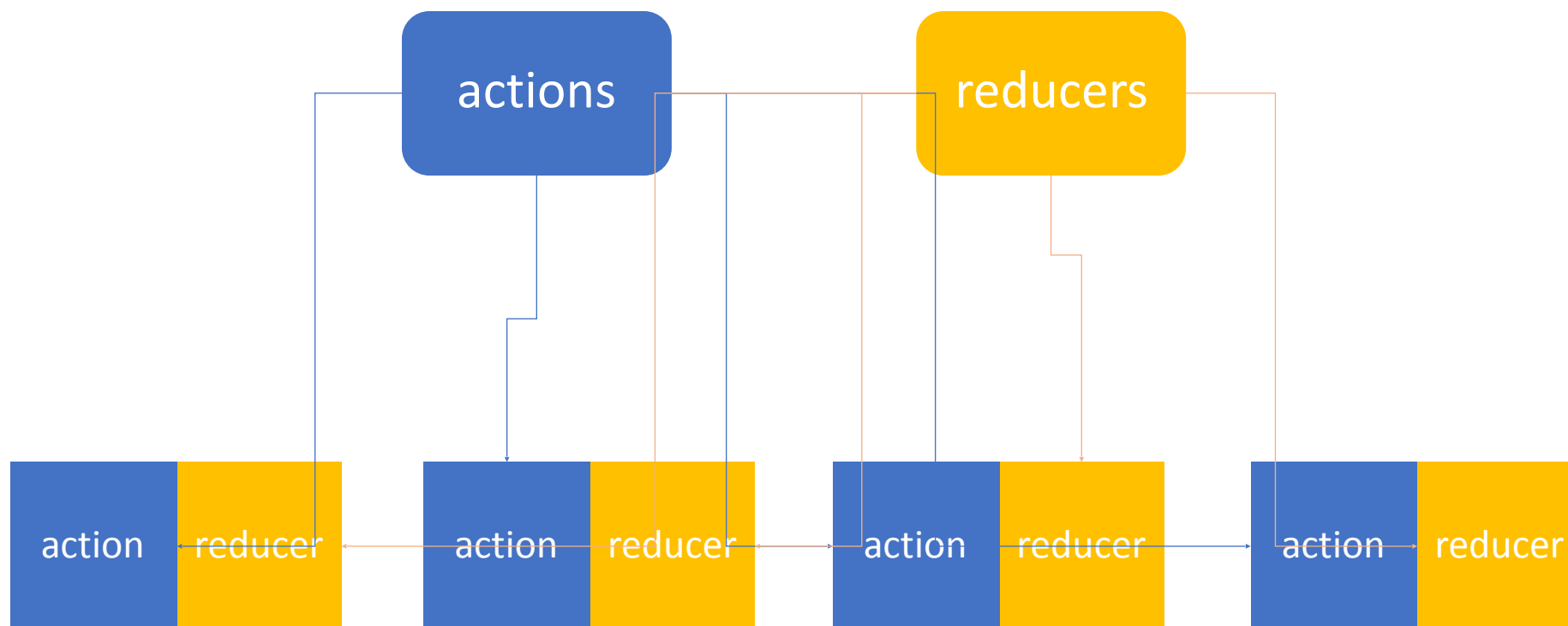
- 按 feature 组织源文件
- 组件和样式文件同一级
- Redux 单独文件夹
- 单元测试保持同样目录结构放在 tests 文件夹

```
src
├── common
│   ├── configStore.js
│   ├── history.js
│   ├── rootReducer.js
│   └── routeConfig.js
├── features
│   ├── common
│   └── examples
│       └── redux
│           ├── actions.js
│           ├── constants.js
│           ├── counterMinusOne.js
│           ├── counterPlusOne.js
│           ├── counterReset.js
│           ├── fetchRedditList.js
│           ├── initialState.js
│           ├── reducer.js
│           ├── CounterPage.js
│           ├── CounterPage.less
│           ├── index.js
│           ├── Layout.js
│           ├── Layout.less
│           ├── RedditListPage.js
│           └── RedditListPage.less
```

# 组件和样式



# 组织 Action 和 Reducer



# counterPlusOne.js

```
import { EXAMPLES_COUNTER_PLUS_ONE } from './constants';

export function counterPlusOne() {
  return {
    type: EXAMPLES_COUNTER_PLUS_ONE,
  };
}

export function reducer(state, action) {
  switch (action.type) {
    case EXAMPLES_COUNTER_PLUS_ONE:
      return {
        ...state,
        count: state.count + 1,
      };

    default:
      return state;
  }
}
```

# constants.js

```
export const EXAMPLES_COUNTER_PLUS_ONE = 'EXAMPLES_COUNTER_PLUS_ONE';
export const EXAMPLES_COUNTER_MINUS_ONE = 'EXAMPLES_COUNTER_MINUS_ONE';
export const EXAMPLES_COUNTER_RESET = 'EXAMPLES_COUNTER_RESET';
export const EXAMPLES_FETCH_REDDIT_LIST_BEGIN = 'EXAMPLES_FETCH_REDDIT_LIST_BEGIN';
export const EXAMPLES_FETCH_REDDIT_LIST_SUCCESS = 'EXAMPLES_FETCH_REDDIT_LIST_SUCCESS';
export const EXAMPLES_FETCH_REDDIT_LIST_FAILURE = 'EXAMPLES_FETCH_REDDIT_LIST_FAILURE';
export const EXAMPLES_FETCH_REDDIT_LIST_DISMISS_ERROR = 'EXAMPLES_FETCH_REDDIT_LIST_DISMISS_ERROR';
```

# reducer.js

```
import initialState from './initialState';
import { reducer as counterPlusOneReducer } from './counterPlusOne';
import { reducer as counterMinusOneReducer } from './counterMinusOne';
import { reducer as counterResetReducer } from './counterReset';
import { reducer as fetchRedditListReducer } from './fetchRedditList';

const reducers = [
  counterPlusOneReducer,
  counterMinusOneReducer,
  counterResetReducer,
  fetchRedditListReducer,
];

export default function reducer(state = initialState, action) {
  let newState;
  switch (action.type) {
    // Handle cross-topic actions here
    default:
      newState = state;
      break;
  }
  return reducers.reduce((s, r) => r(s, action), newState);
}
```



# actions.js

```
export { counterPlusOne } from './counterPlusOne';  
export { counterMinusOne } from './counterMinusOne';  
export { counterReset } from './counterReset';  
export { fetchRedditList, dismissFetchRedditListError } from './fetchRedditList';
```

# rootReducer.js

```
import { combineReducers } from 'redux';
import { routerReducer } from 'react-router-redux';
import homeReducer from '../features/home/redux/reducer';
import commonReducer from '../features/common/redux/reducer';
import examplesReducer from '../features/examples/redux/reducer';

const reducerMap = {
  router: routerReducer,
  home: homeReducer,
  common: commonReducer,
  examples: examplesReducer,
};

export default combineReducers(reducerMap);
```

DEMO

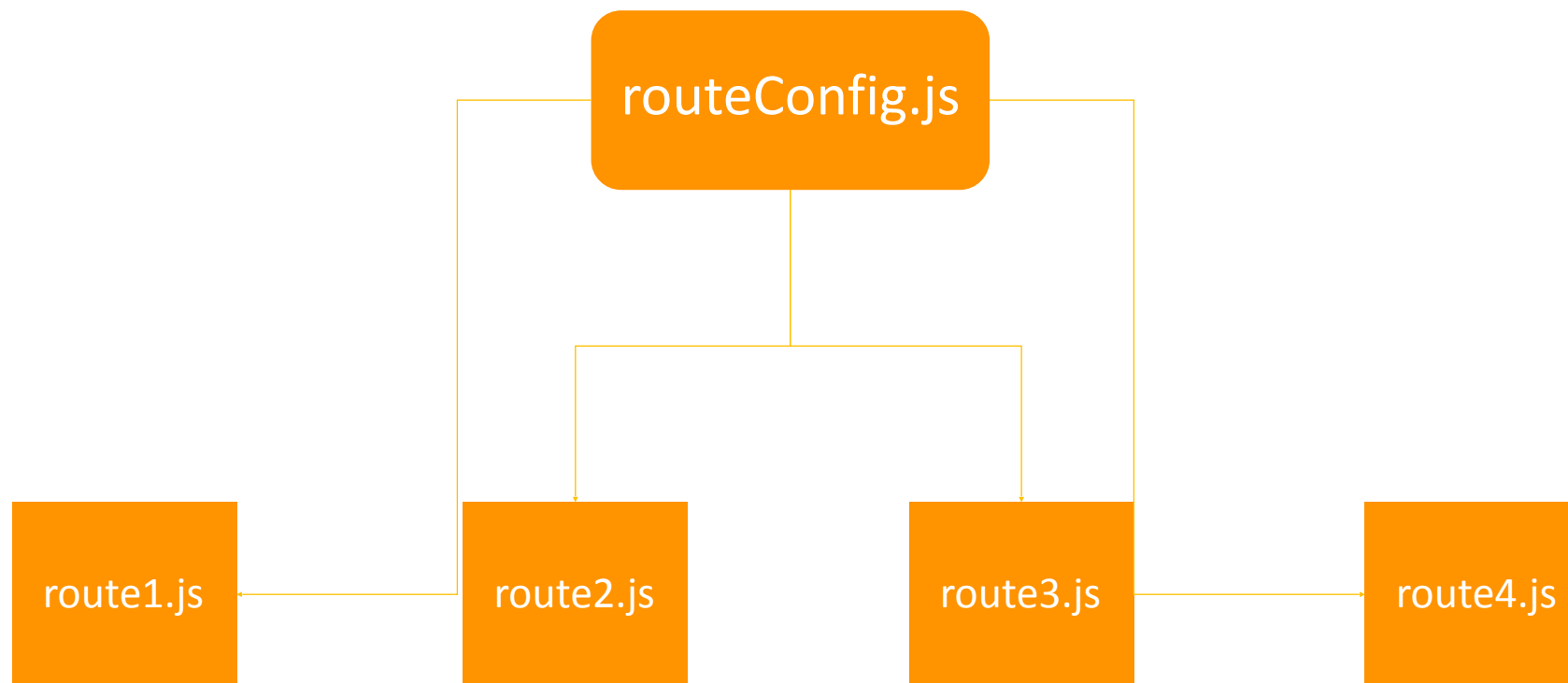
# 小结

1. 按 feature 组织组件，action 和 reducer
2. 使用 root loader 加载 feature 下的各个资源
3. 做到高内聚松耦合

拆分复杂度（3）：

如何组织 React Router 的路由配置

# 在每个 feature 中单独定义自己的路由



# 使用 JSON 定义顶层路由

```
import { WelcomePage, CounterPage, RedditListPage, Layout } from './';

export default {
  path: 'examples',
  name: 'Examples',
  component: Layout,
  childRoutes: [
    { path: '', name: 'Welcome page', component: WelcomePage },
    { path: 'counter', name: 'Counter page', component: CounterPage },
    { path: 'reddit', name: 'Reddit list page', protected: true, component: RedditListPage },
  ],
};
```

# 解析 JSON 路由到 React Router 语法

```
function renderRouteConfigV3(routes, contextPath) {
  // Resolve route config object in React Router v3.
  const children = []; // children component list

  const renderRoute = (item, routeContextPath) => {
    let newContextPath;
    if (/^\/?/.test(item.path)) {
      newContextPath = item.path;
    } else {
      newContextPath = `${routeContextPath}/${item.path}`;
    }
    newContextPath = newContextPath.replace(/\/+/g, '/');
    if (item.component && item.childRoutes) {
      const childRoutes = renderRouteConfigV3(item.childRoutes, newContextPath);
      children.push(
        <Route
          key={newContextPath}
          render={props => <item.component {...props}>{childRoutes}</item.component>}
          path={newContextPath}
        />
      );
    } else if (item.component) {
      children.push(<Route key={newContextPath} component={item.component} path={newContextPath} exact />);
    } else if (item.childRoutes) {
      item.childRoutes.forEach(r => renderRoute(r, newContextPath));
    }
  };
};
```



DEMO

# 小结

1. 每个 feature 都有自己的专属路由配置
2. 顶层路由使用 JSON 配置更易维护和理解
3. 如何解析 JSON 配置到 React Router 语法

# 使用 Rekit ( 1 ) : 创建项目, 代码生成和重构



# Rekit

React 专属 IDE 和工具集

Unwatch ▾

128

★ Unstar

3,209

Fork

189

# 背景

前端技术功能越来越强大

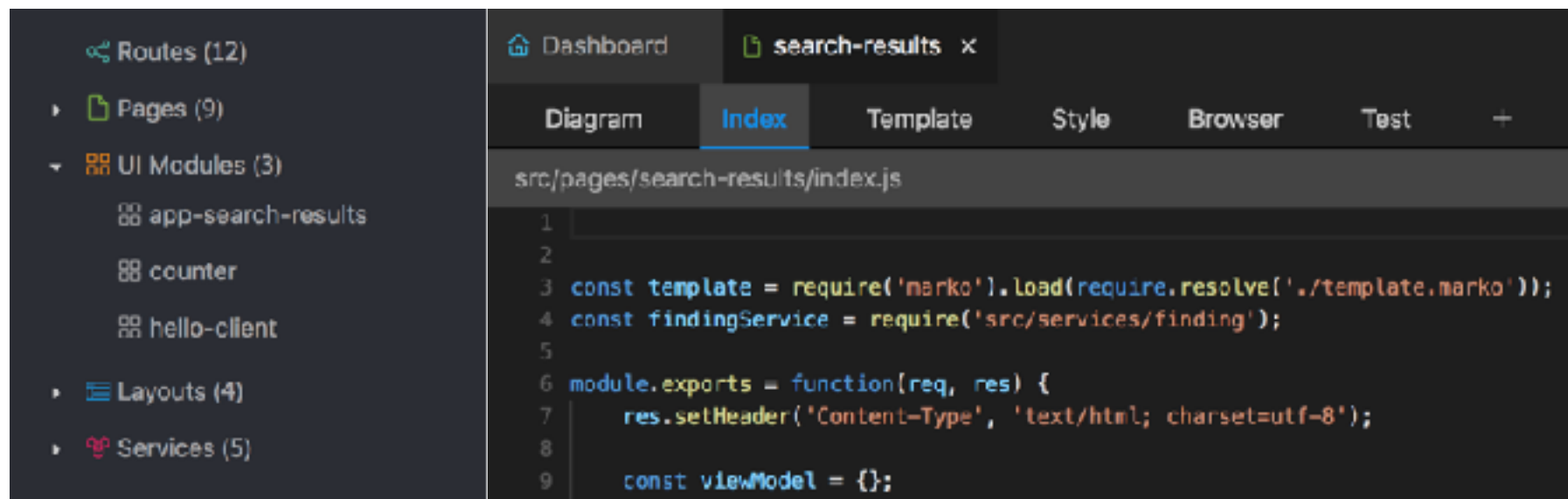


开发变得越来越复杂

- 一个独立功能通常需要多个文件组成
- 代码模板很复杂
- 重构极为困难
- 项目复杂后很难理解和维护

# Rekit: 更好的代码导航

1. 语义化的组织源代码文件
2. 使用子 Tab 来展示项目元素的各个部分
3. 直观的显示和导航某个功能的所有依赖

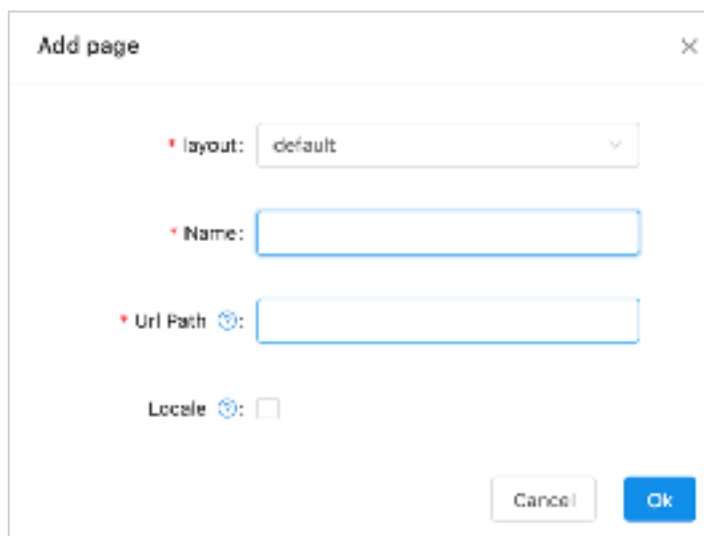
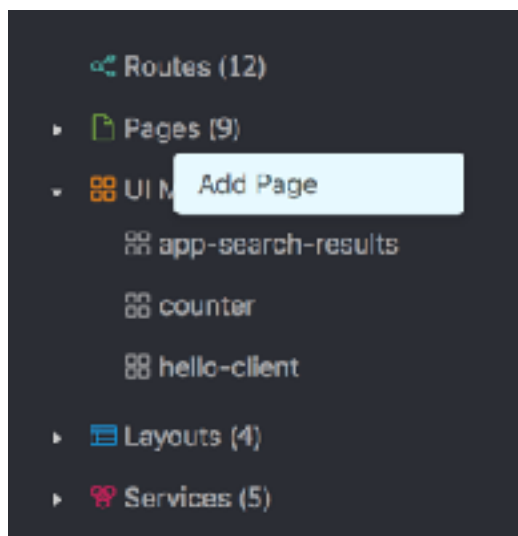


The screenshot shows the Rekit code navigation interface. On the left, a sidebar lists project components: Routes (12), Pages (9), UI Modules (3) including app-search-results, counter, and hello-client, Layouts (4), and Services (5). The main area displays a file explorer for 'src/pages/search-results/index.js' with tabs for Diagram, Index (selected), Template, Style, Browser, and Test. The code editor shows the following JavaScript code:

```
1  
2  
3 const template = require('marko').load(require.resolve('./template.marko'));  
4 const findingService = require('src/services/finding');  
5  
6 module.exports = function(req, res) {  
7   res.setHeader('Content-Type', 'text/html; charset=utf-8');  
8  
9   const viewModel = {};
```

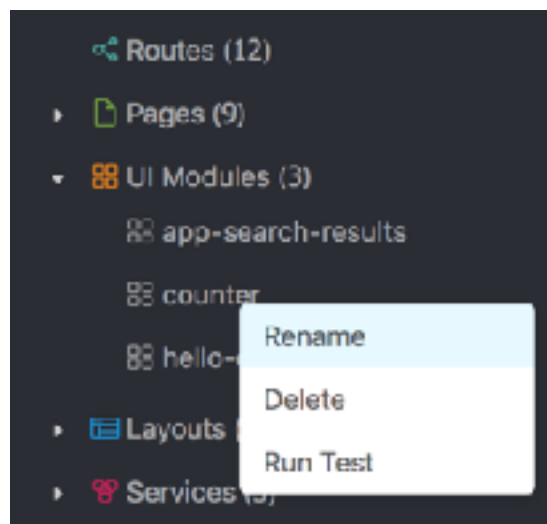
# Rekit: 一键生成项目元素

1. 直观的 UI 用于生成组件，action，reducer 等
2. 模板代码遵循最佳实践
3. 支持命令行方式创建项目元素



# Rekit: 重构非常容易

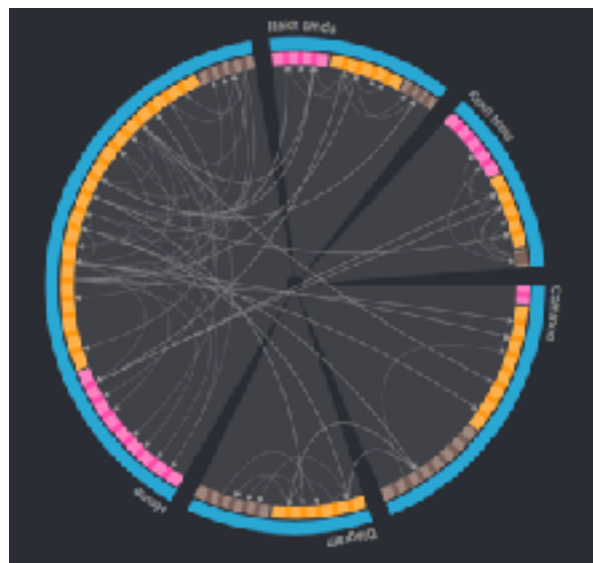
1. 右键菜单重命名或者删除某个项目元素
2. 所有相关代码都会一次性重构从而保证一致性
3. 详细的 log 信息显示重构的完整修改





# Rekit: 可视化的项目架构

1. 项目总体架构的可视化图表
2. 项目依赖关系的图表

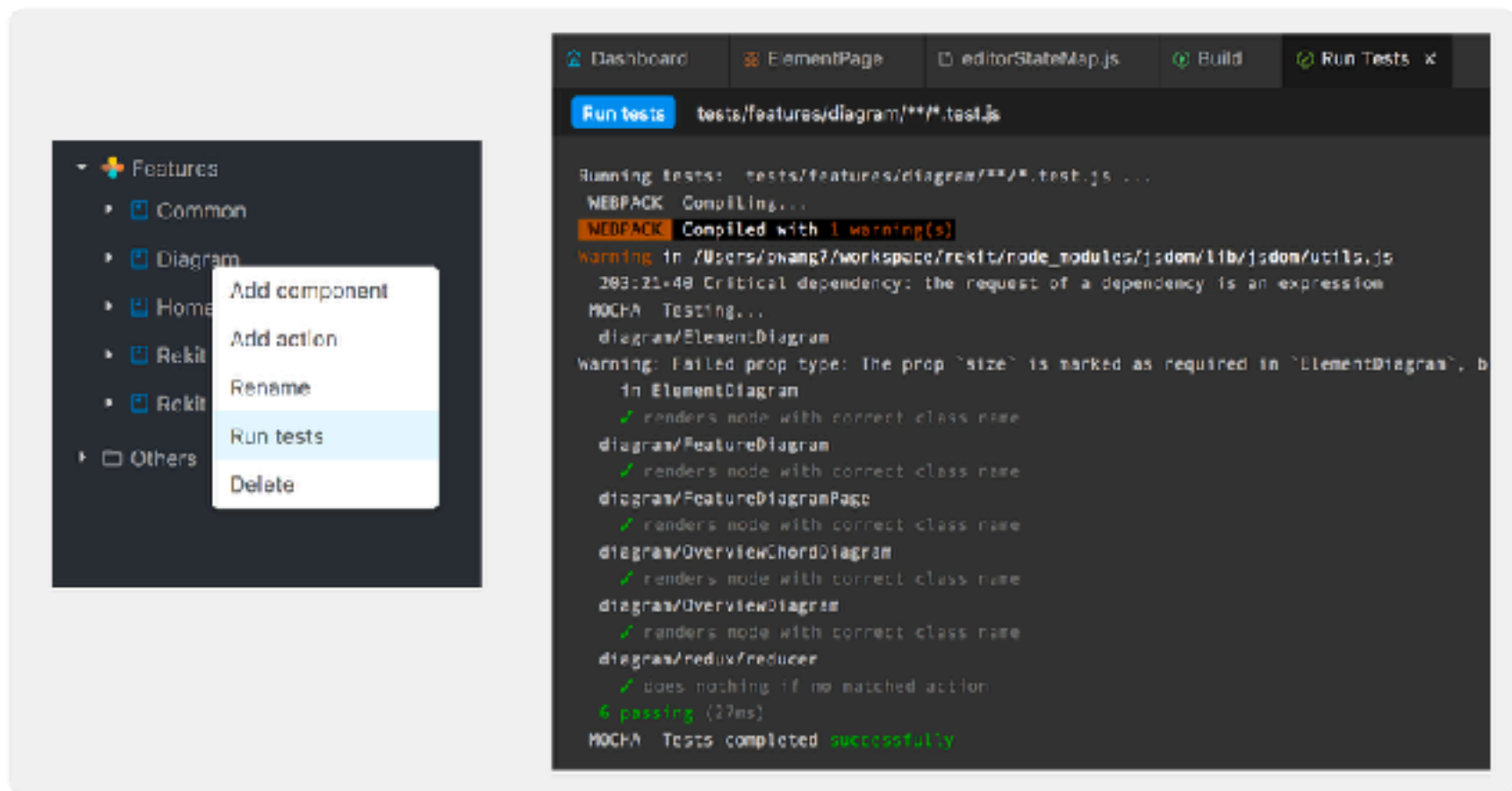


# Rekit 是如何工作的？

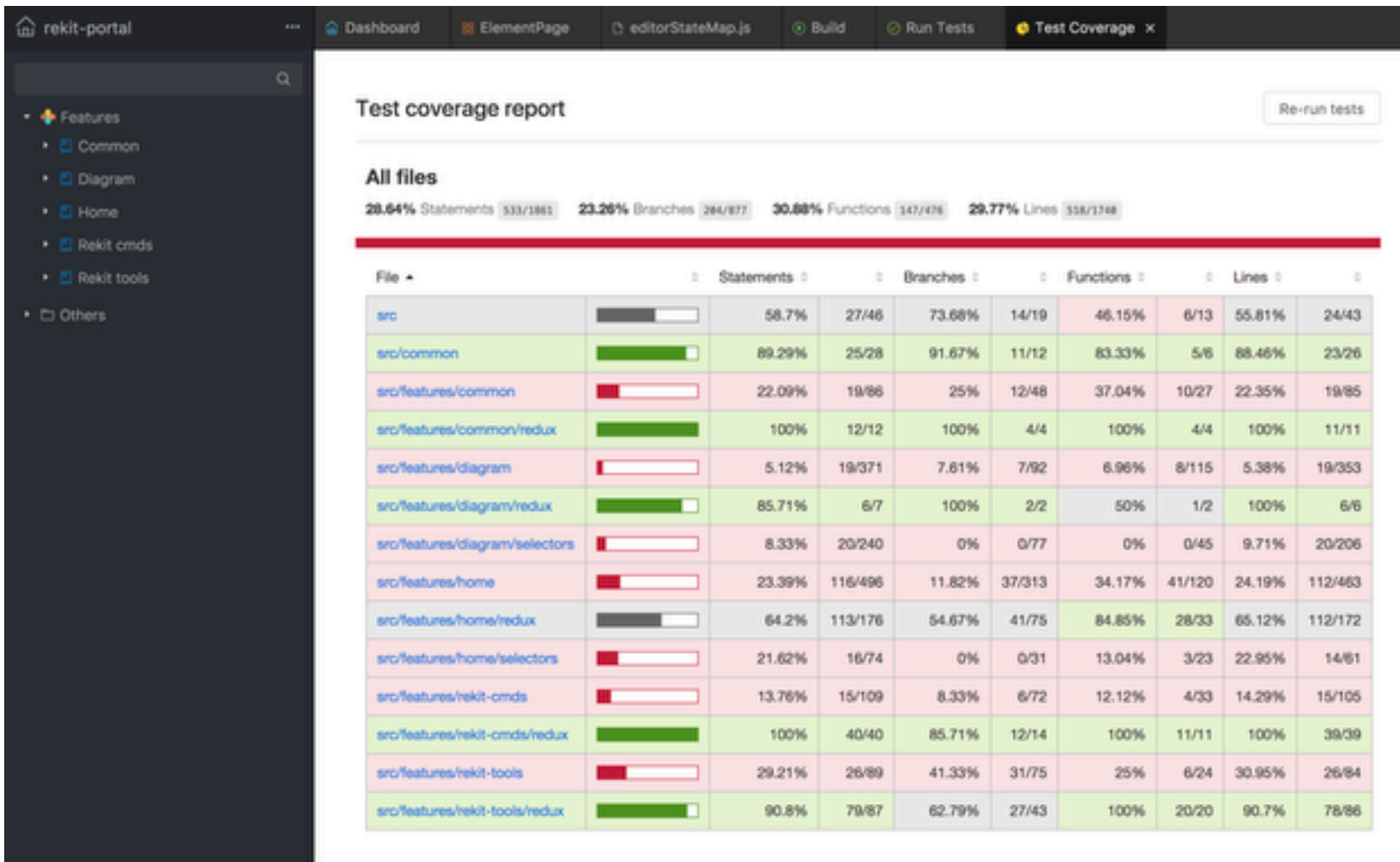
1. 定义了基于 feature 的可扩展文件夹结构
2. 基于最佳实践生成代码和管理项目元素
3. 提供工具和 IDE 确保代码和文件夹结构遵循最佳实践



# 集成单元测试



# 单元测试覆盖率



# 小结

1. 什么是 Rekit
2. 创建 Rekit 项目的背景
3. Rekit Studio 提供的基本功能

# 使用 Rekit ( 2 ) : 遵循最佳实践, 保持代码一致性

# 遵循最佳实践

1. 以 feature 方式组织代码
2. 拆分组件，action 和 reducer
3. 拆分路由配置

# 通过代码自动生成保持一致性

1. 文件夹结构一致性
2. 文件名一致性
3. 变量名一致性
4. 代码逻辑的一致性



DEMO

# 使用 React Router 管理登录和授权

# 使用 React Router 管理路由授权

1. 实现基础：React Router 的动态路由机制
2. 区分受保护路由和公开路由
3. 访问未授权路由时重定向到登录页面

DEMO

# 实现表单（1）：初始数据，提交和跳转

DEMO

实现表单（2）：错误处理，动态表单元素，内容动态加载

DEMO

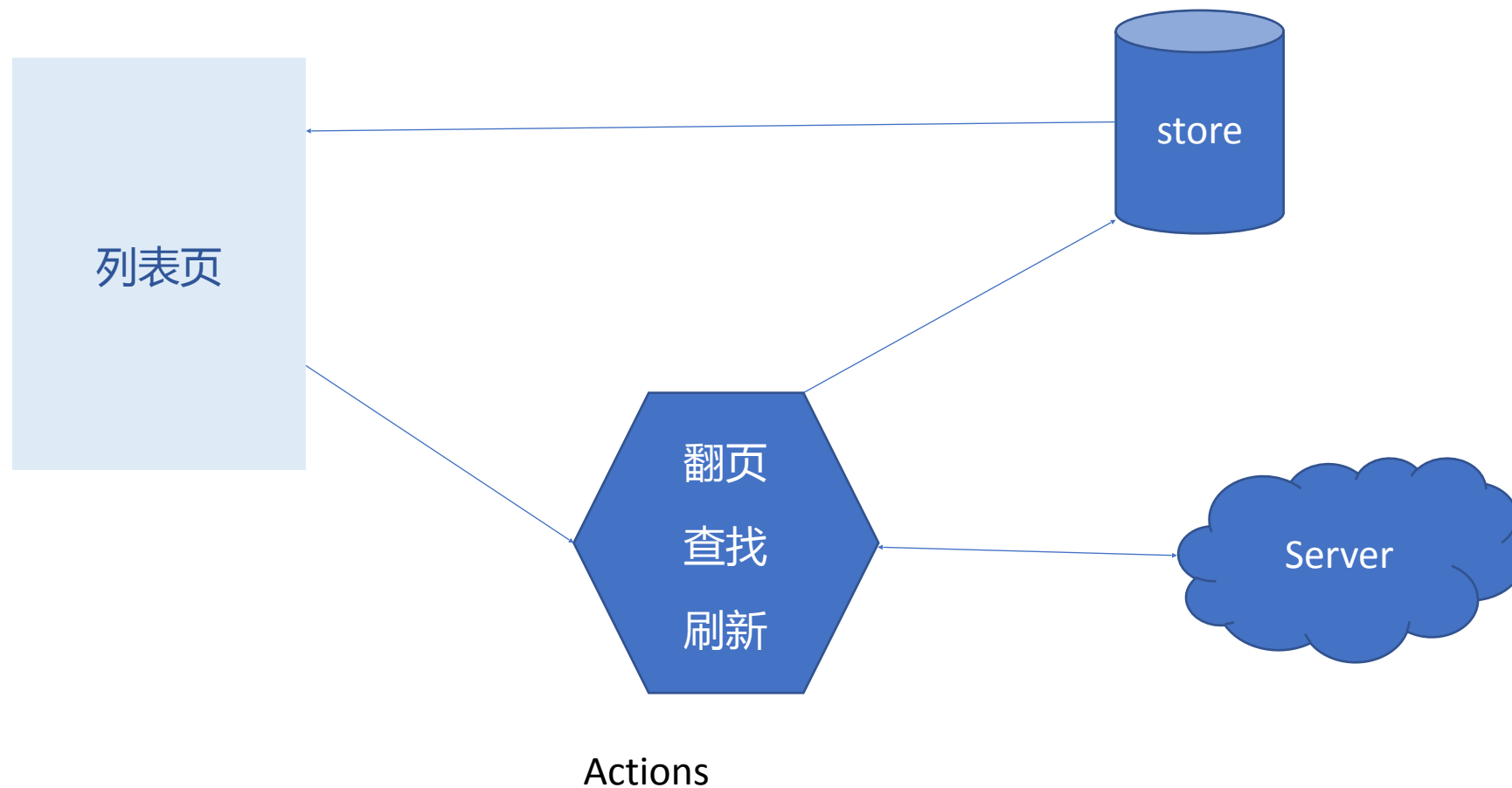


# 列表页(1)：搜索，数据缓存和分页

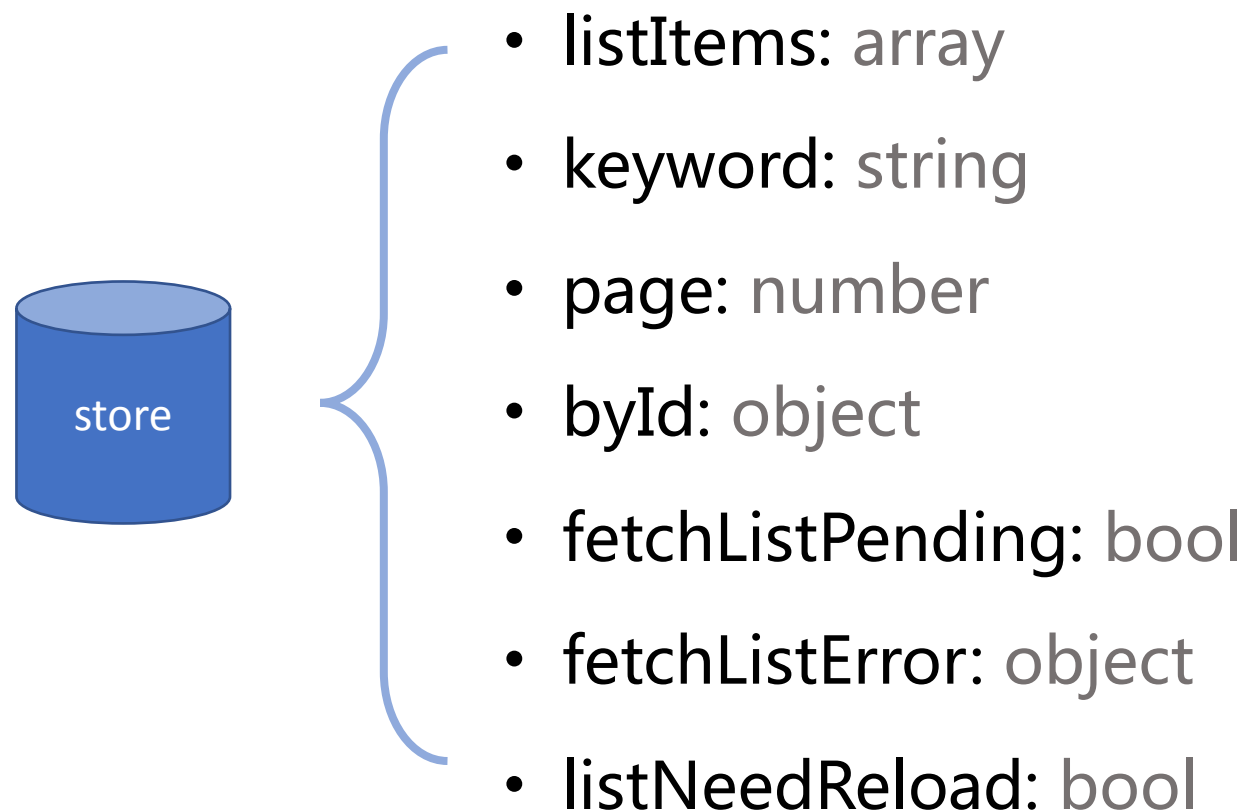
# 开发列表也要考虑的技术要点

1. 如何翻页
2. 如何进行内容搜索
3. 如何缓存数据
4. 何时进行页面刷新

# 列表页的数据模型



# 设计 Redux 的 Store 模型



# URL 设计以及和 Store 同步

/list/page?keyword=xxx



DEMO

# 小结

1. 页面数据逻辑
2. Redux Store 设计
3. URL 的设计
4. 翻页，搜索的功能的实现

## 列表页(2)：缓存更新，加载状态，错误处理



# 加载，错误信息也是 store 的状态



- listItems: array
- keyword: string
- page: number
- byId: object
- **fetchListPending**: bool
- **fetchListError**: object
- **listNeedReload**: bool

DEMO

# 页面数据需要来源多个请求的处理

# 页面数据来自多个请求

1. 请求之间无依赖关系，可以并发进行
2. 请求有依赖，需要依次进行
3. 请求完成之前，页面显示 Loading 状态

DEMO

# 内容页的加载和缓存

# 内容页和列表页的数据关系

1. 简单业务：列表页数据包含内容页的数据
2. 复杂业务：内容页数据需要额外获取
3. 内容页数据的缓存

DEMO



# 基于 React Router 实现分步操作

# 向导页面需要考虑的技术要点

1. 使用 URL 进行导航的好处
2. 表单内容存放的位置
3. 页面状态如何切换

DEMO

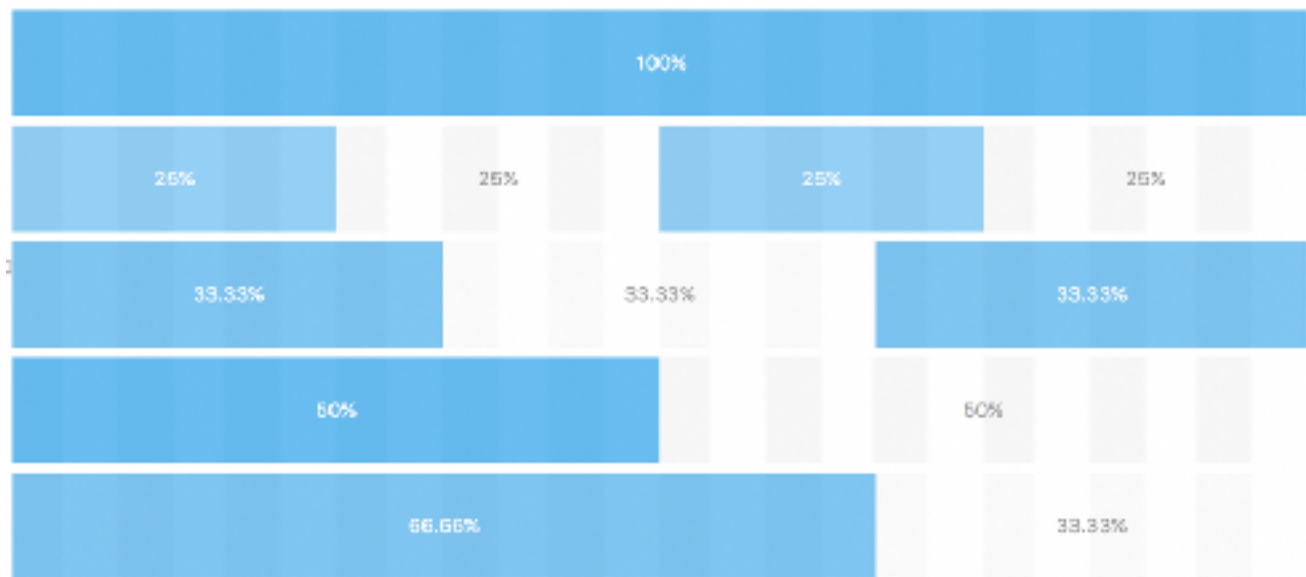
# 常见页面布局的实现

# 实现布局的几种方式

1. 从0开始用 CSS 实现
2. 使用 CSS Grid 系统
3. 使用组件库，例如 antd



Ant Design



```
<div>
  <Row>
    <Col span={12}>col-12</Col>
  </Row>
  <Row>
    <Col span={8}>col-8</Col>
    <Col span={8}>col-8</Col>
  </Row>
  <Row>
    <Col span={6}>col-6</Col>
    <Col span={6}>col-6</Col>
    <Col span={6}>col-6</Col>
  </Row>
</div>
```

# 布局常见场景：侧边栏宽度可调整

1. 手动实现拖放逻辑
2. 使用 local storage 存储宽度位置

DEMO



使用 React Portals 实现对话框，使用 antd 对话框

# React Portals

1. React 16.3 新引入的 API
2. 可以将虚拟 DOM 映射到任何真实 DOM 节点
3. 解决了漂浮层的问题，比如：Dialog，Tooltip 等

DEMO

# 集成第三方 JS 库：以 d3.js 为例

# 集成第三方 JS 库的技术要点

1. 使用 ref 获取原生 DOM 节点引用
2. 手动将组件状态更新到 DOM 节点
3. 组件销毁时移除原生节点 DOM 事件

DEMO

# 基于路由实现菜单导航

# 技术要点

1. 菜单导航只是改变 URL 状态
2. 根据当前 URL 显示菜单的当前状态



DEMO

# React 中拖放的实现

# 使用 React 实现拖放的技术要点

1. 如何使用 React 的鼠标事件系统
2. 如何判断拖放开始和拖放结束
3. 如何实现拖放元素的位置移动
4. 拖放状态在组件中如何维护

DEMO

性能永远是第一需求：时刻考虑性能问题

# 如何避免应用出现性能问题

1. 了解常见的性能问题场景
2. 时刻注意代码的潜在性能问题
3. 注重可重构的代码
4. 了解如何使用工具定位性能问题

DEMO

# 网络性能优化：自动化按需加载



# 如何在 React 中实现按需加载

1. 什么是按需加载
2. 使用 Webpack 的 import API
3. 使用 react-loadable 库实现 React 异步加载

DEMO

# 使用 reselect 避免重复计算

# Reselect:

## 创建自动缓存的数据处理流程

```
import { createSelector } from 'reselect'

const shopItemsSelector = state => state.shop.items
const taxPercentSelector = state => state.shop.taxPercent

const subtotalSelector = createSelector(
  shopItemsSelector,
  items => items.reduce((acc, item) => acc + item.value, 0)
)

const taxSelector = createSelector(
  subtotalSelector,
  taxPercentSelector,
  (subtotal, taxPercent) => subtotal * (taxPercent / 100)
)

export const totalSelector = createSelector(
  subtotalSelector,
  taxSelector,
  (subtotal, tax) => ({ total: subtotal + tax })
)

let exampleState = {
  shop: {
    taxPercent: 8,
    items: [
      { name: 'apple', value: 1.20 },
      { name: 'orange', value: 0.95 },
    ]
  }
}

console.log(subtotalSelector(exampleState)) // 2.15
console.log(taxSelector(exampleState)) // 0.172
console.log(totalSelector(exampleState)) // { total: 2.322 }
```

DEMO

# 下一代 React : 异步渲染

# 异步渲染的两个部分

## 时间分片 ( Time Slicing )

DOM 操作的优先级低于浏览器原生行为，例如键盘和鼠标输入，从而保证操作的流畅。

## 渲染挂起 ( Suspense )

虚拟 DOM 节点可以等待某个异步操作的完成，并指定 timeout，之后才完成真正的渲染。

# 时间分片

1. 虚拟 DOM 的 diff 操作可以分片进行
2. React 新 API : `unstable_deferredUpdates`
3. Chrome 新 API : `requestIdleCallback`



# 渲染挂起

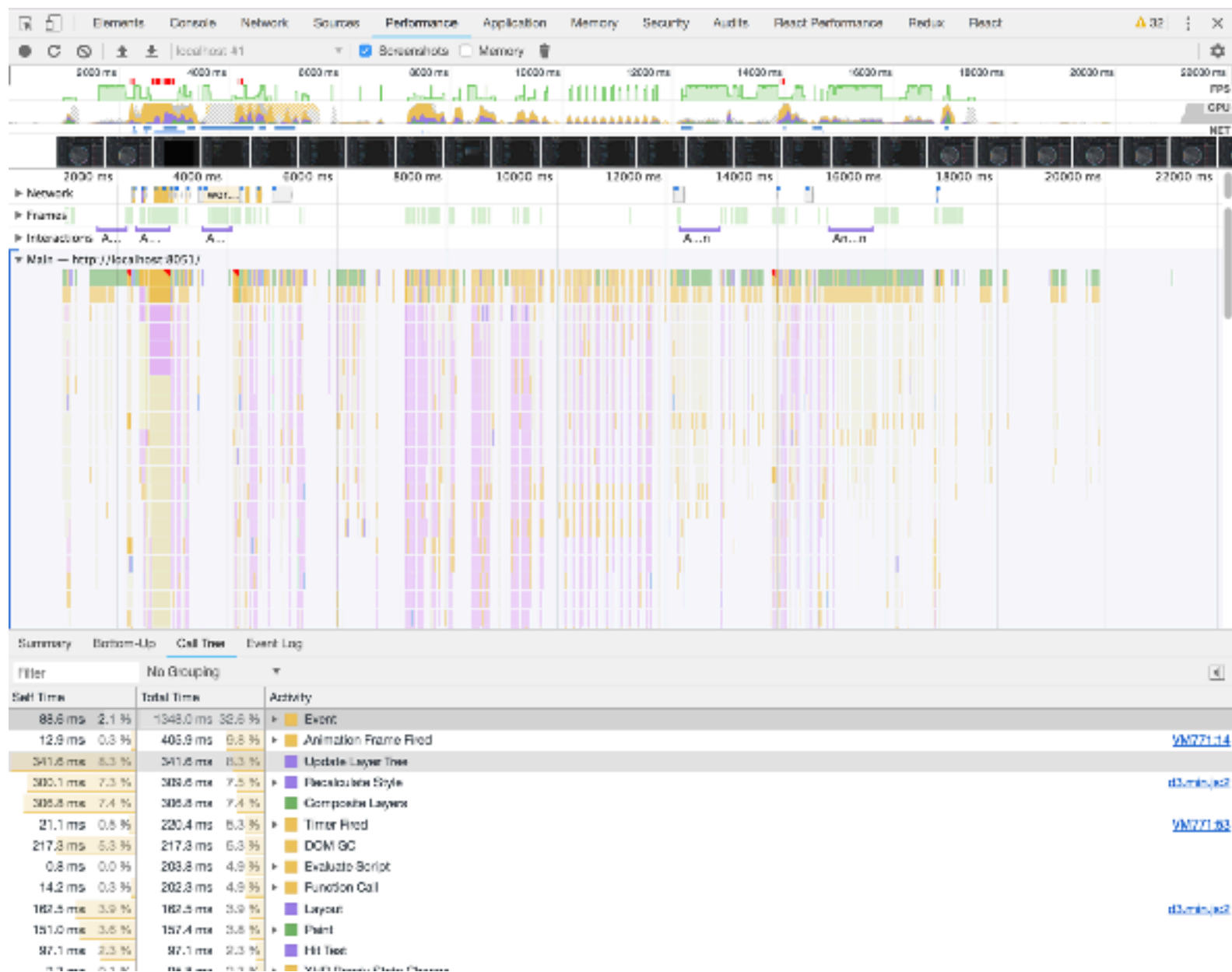
1. 新内置组件：Timeout
2. `unstable_deferUpdate`

DEMO

# 使用 Chrome DevTool 进行性能调优

# 借助工具发现性能问题

1. 使用 React DevTool 找到多余渲染
2. 使用 Chrome DevTool 定位性能瓶颈



DEMO