React 从入门到精通

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

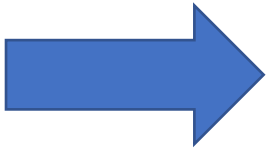
React：始终整体”刷新”页面

无需关心细节

局部刷新

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

{ text: 'message3' }

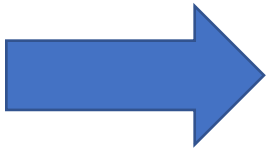


```
<ul>
  <li>message1</li>
  <li>message2</li>
</ul>
```

Append:
message3

React：整体刷新

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



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

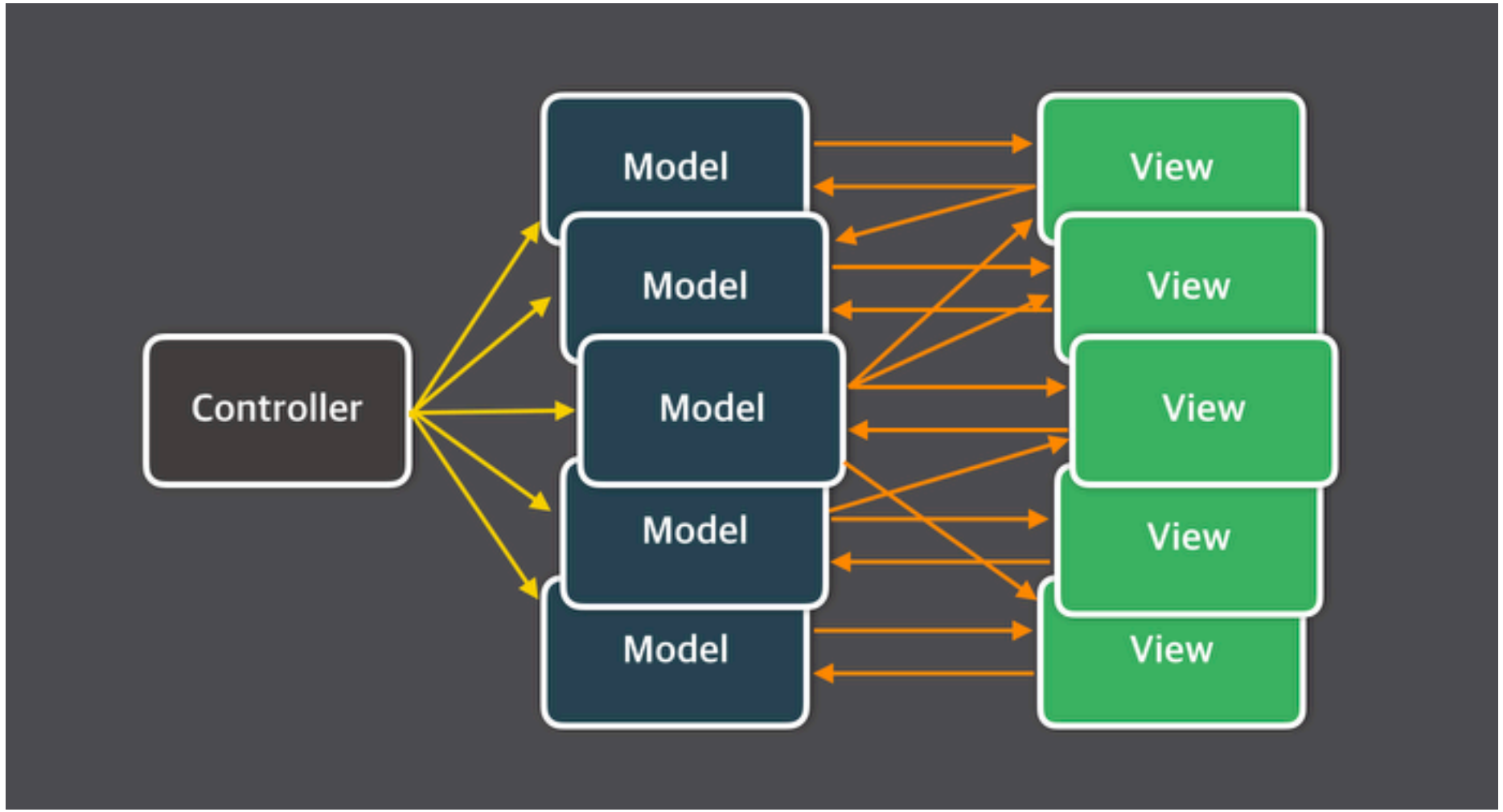
React 很简单

- 1个新概念
- 4个必须 API
- 单向数据流
- 完善的错误提示

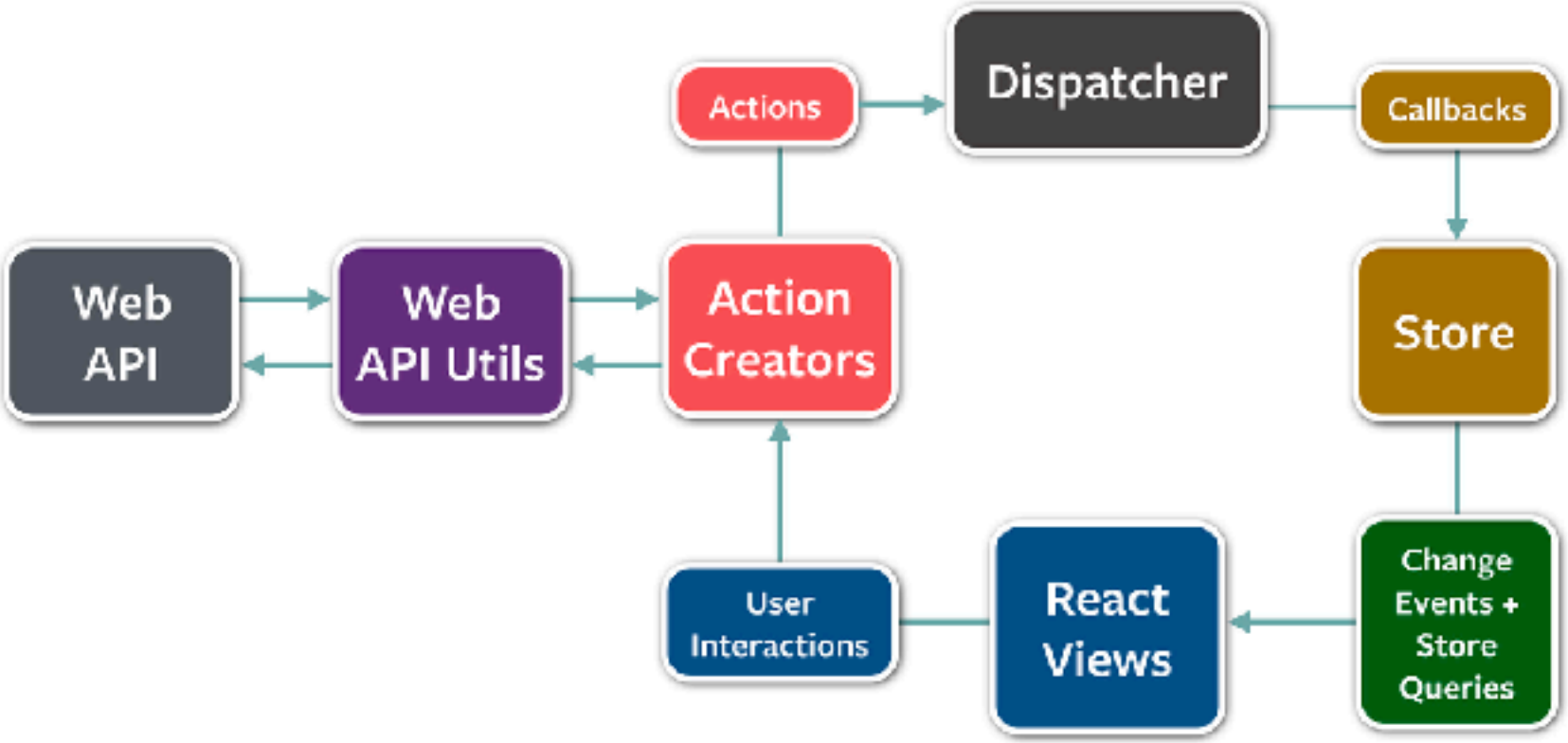
React 解决了 UI 细节问题

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

极客时间

Comments (3)

Nate

Hello React! This is a sample comment.

Kevin

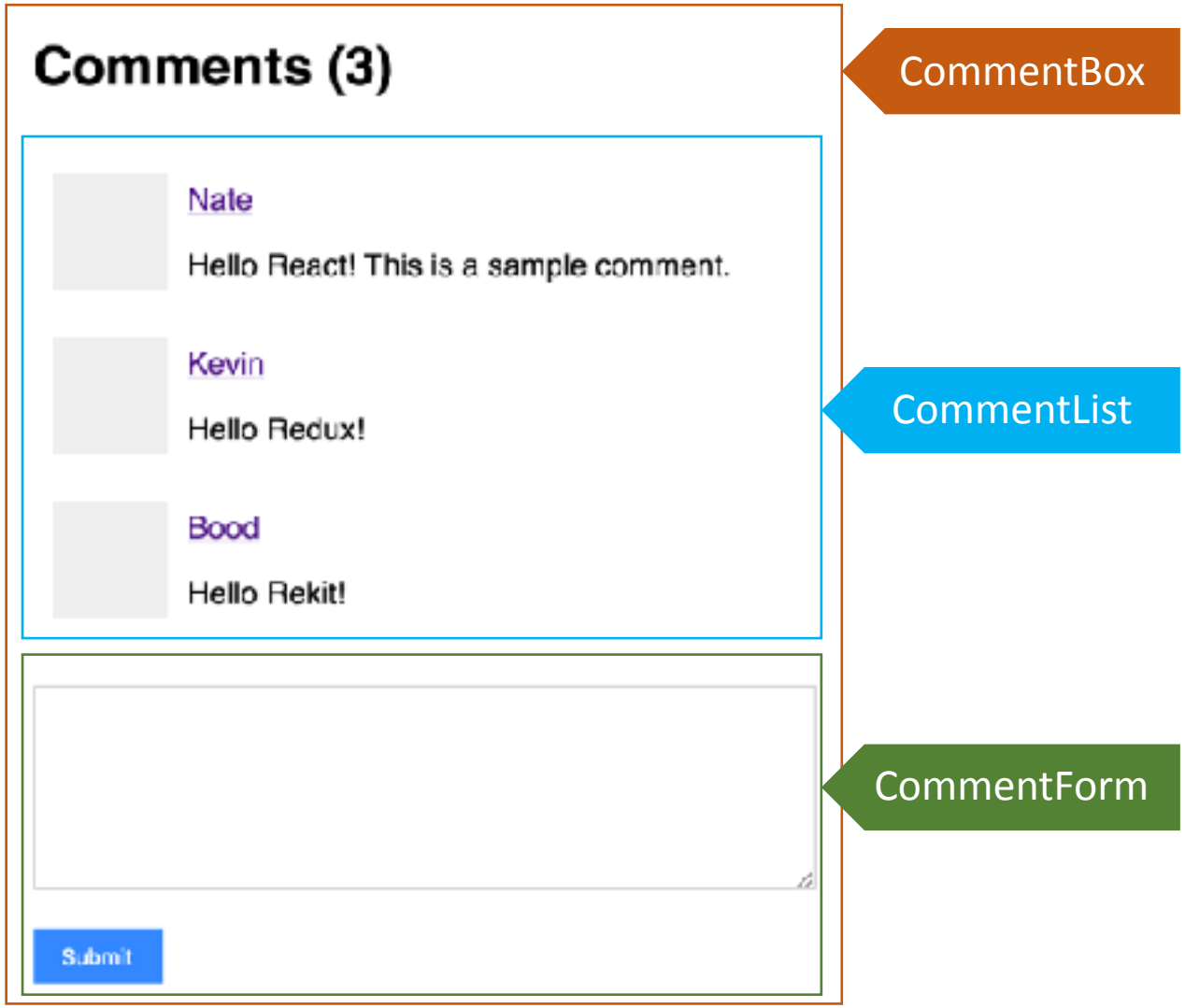
Hello Redux!

Bood

Hello Rekit!

Submit

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: Red Blue Orange

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

极客时间

```
<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

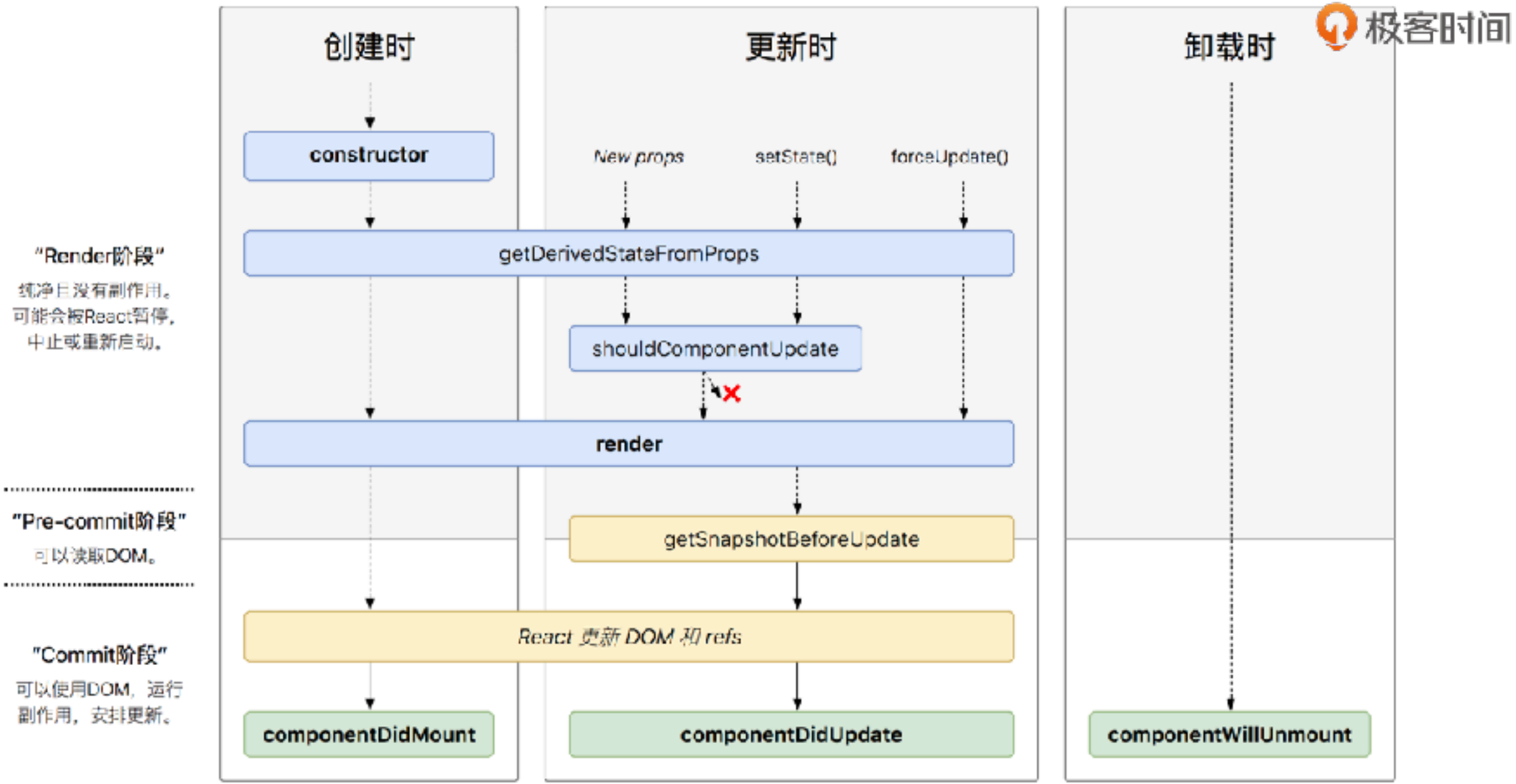
2.大写字母开头为自定义组件

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

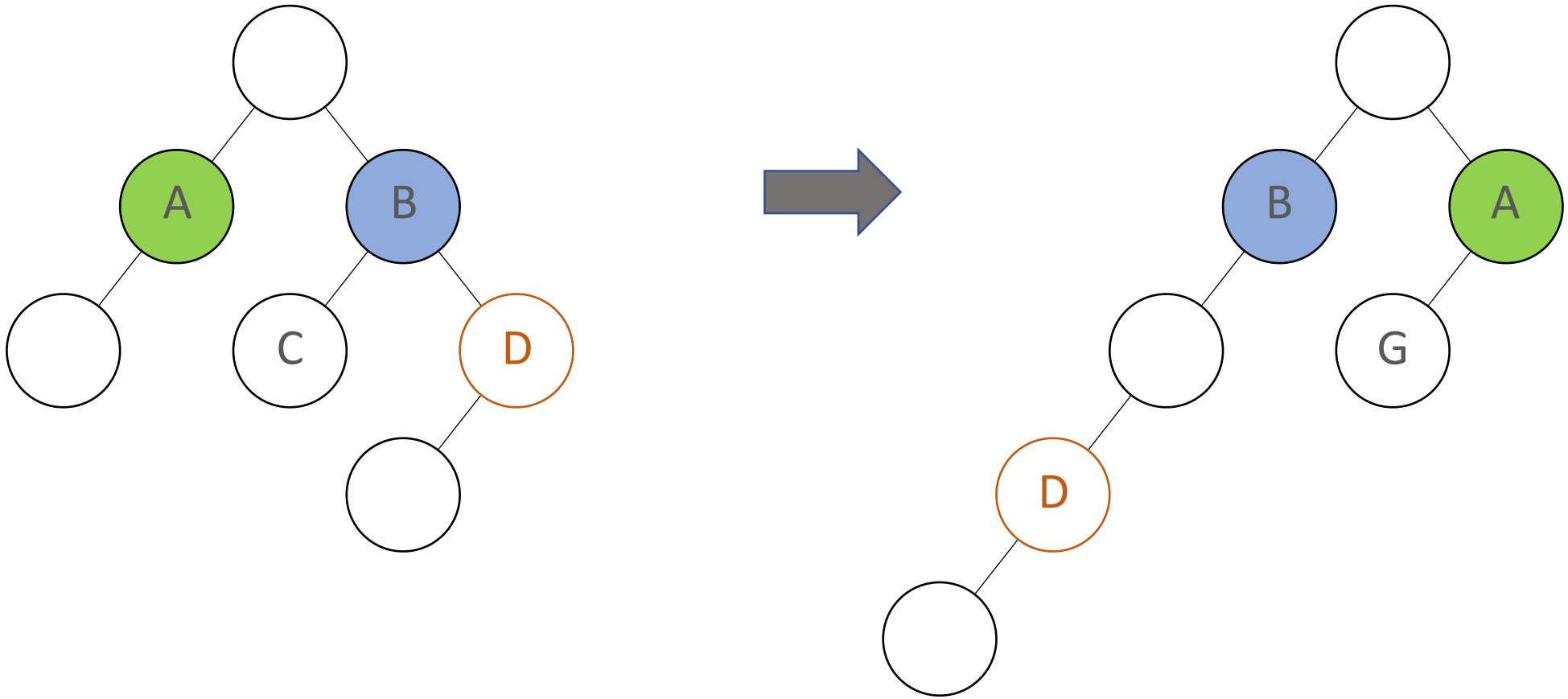
1. 理解 React 组件的生命周期方法
2. 理解生命周期的使用场景

理解 Virtual DOM 的工作原理，

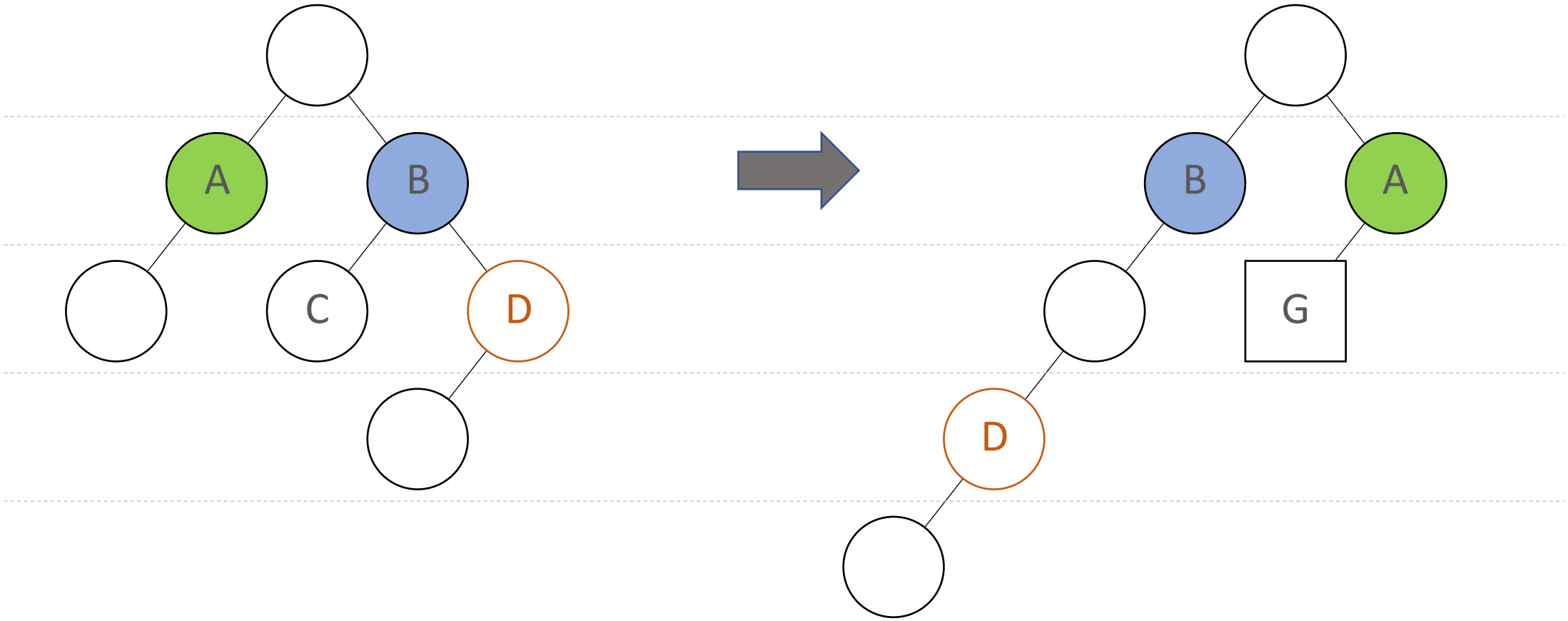
理解 key 属性的作用

JSX 的运行基础：Virtual DOM

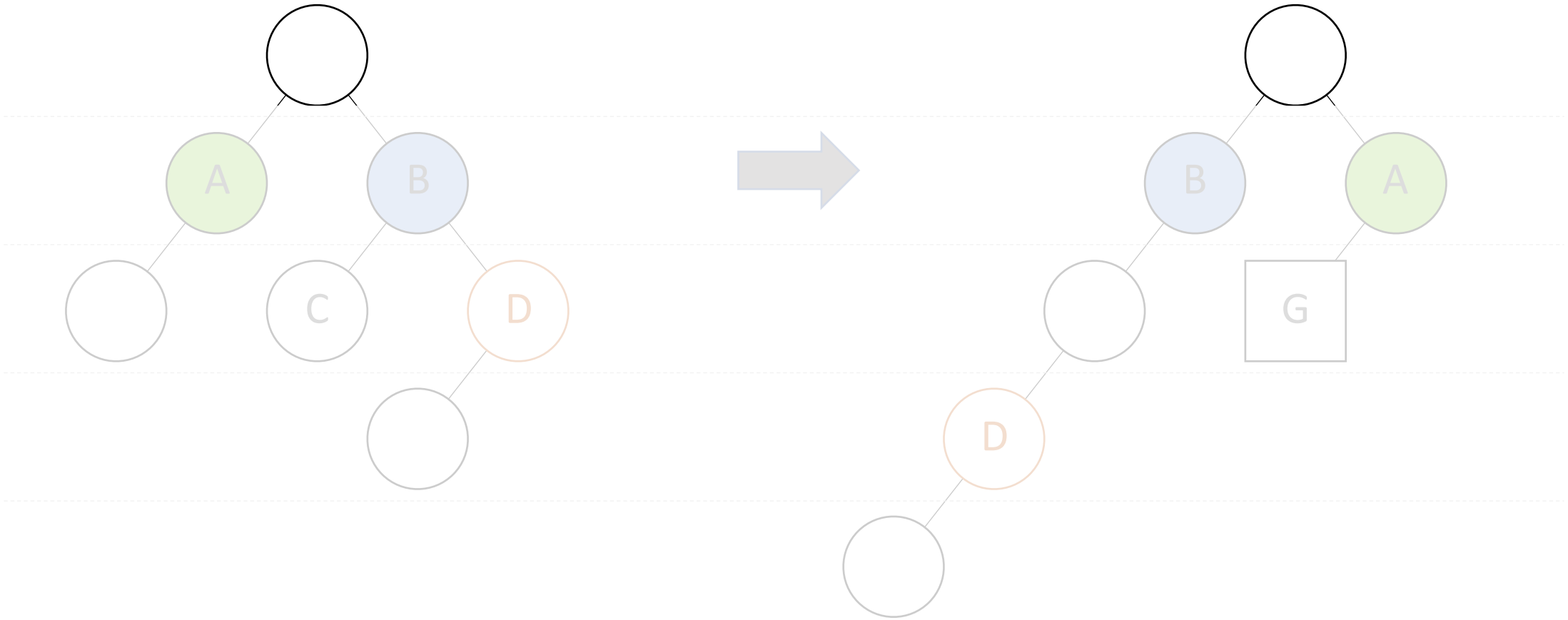
虚拟 DOM 是如何工作的



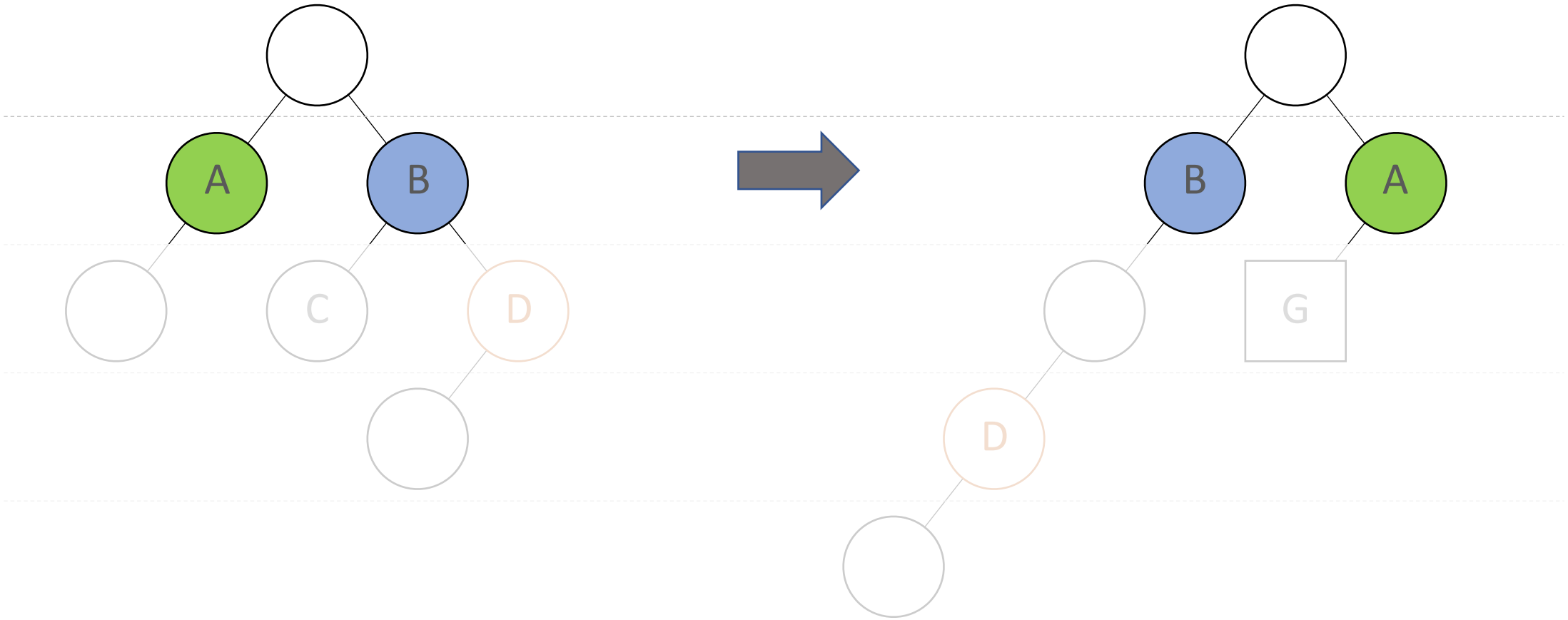
广度优先分层比较



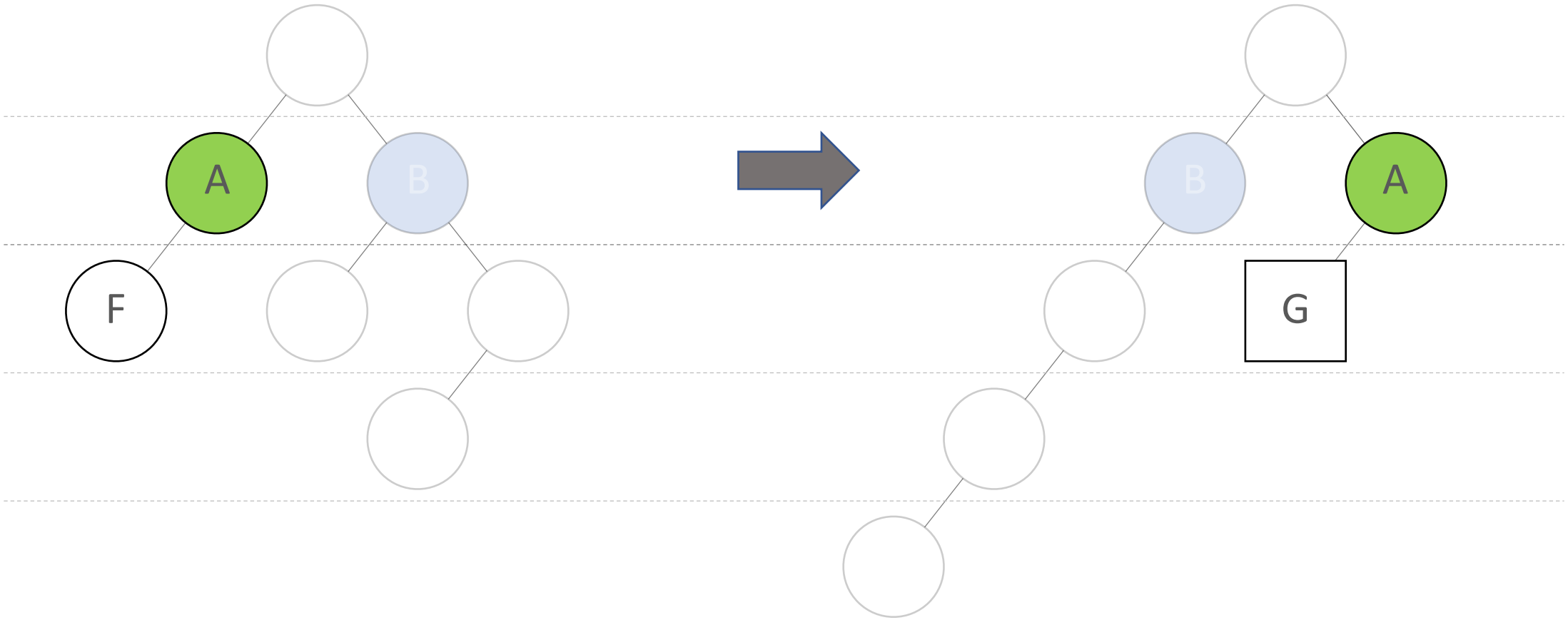
根节点开始比较



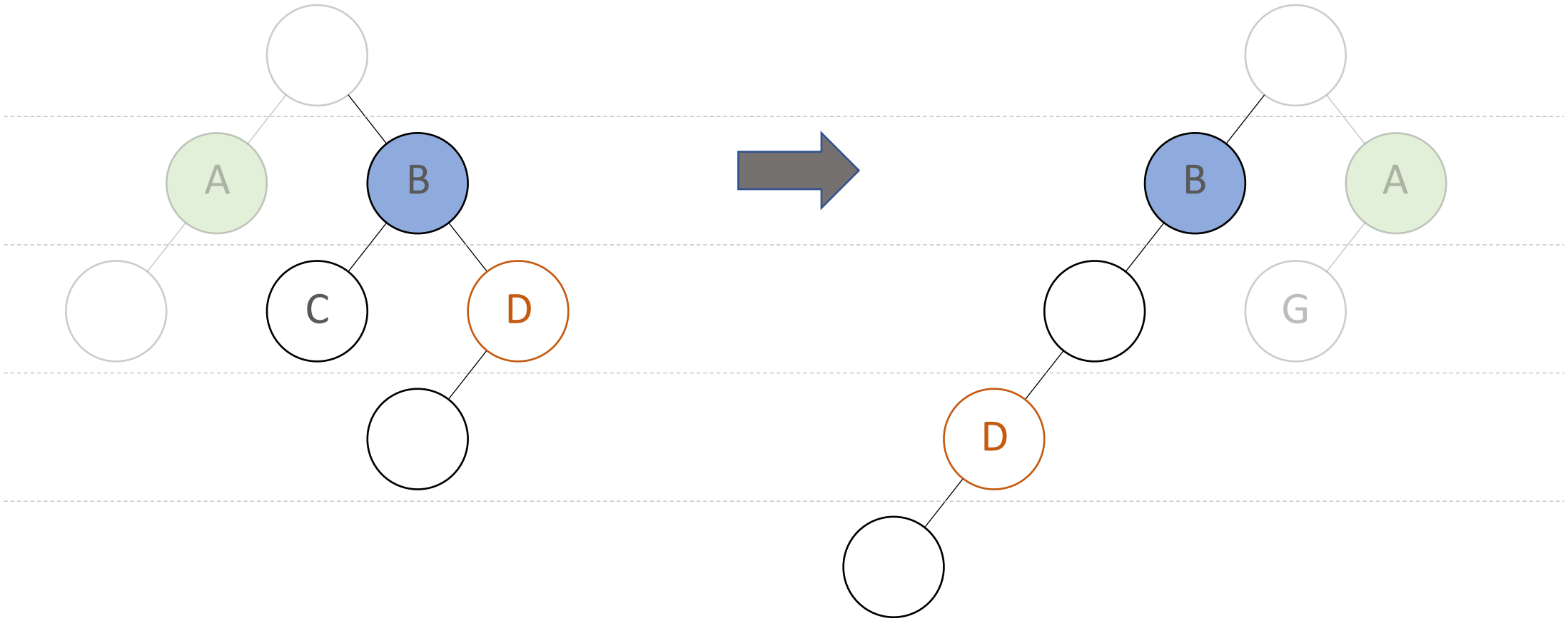
属性变化及顺序



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

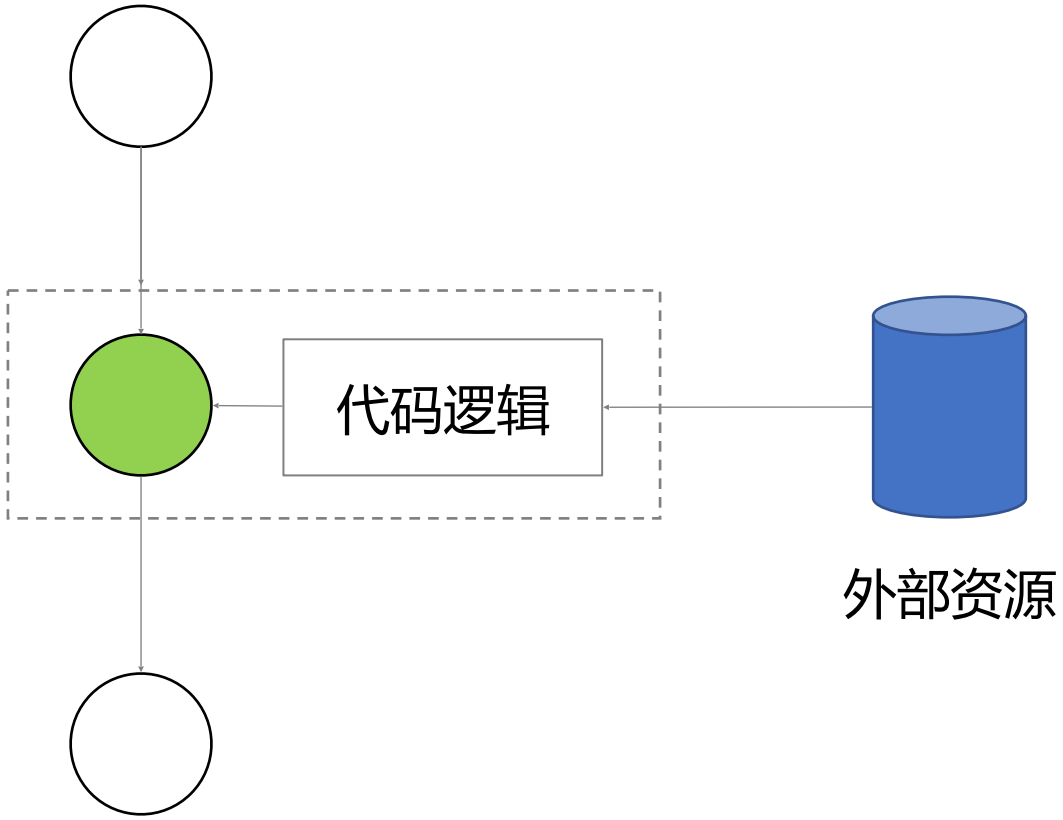
小结

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）



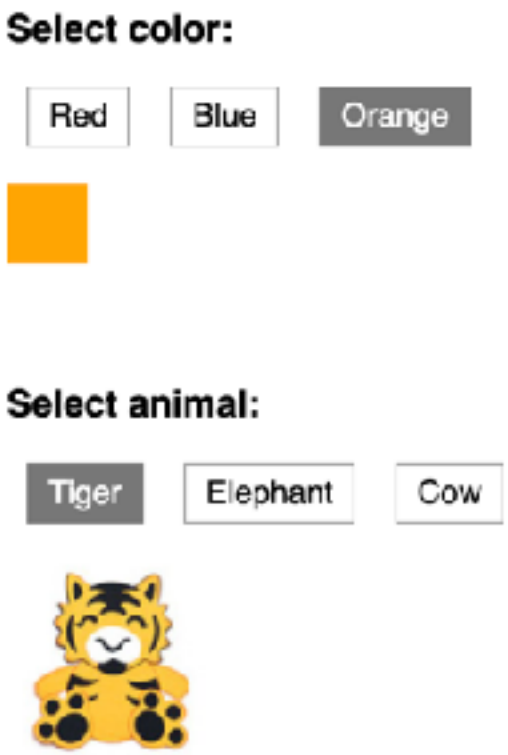

```
const EnhancedComponent =
      higherOrderComponent(WrappedComponent);
```

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>
```



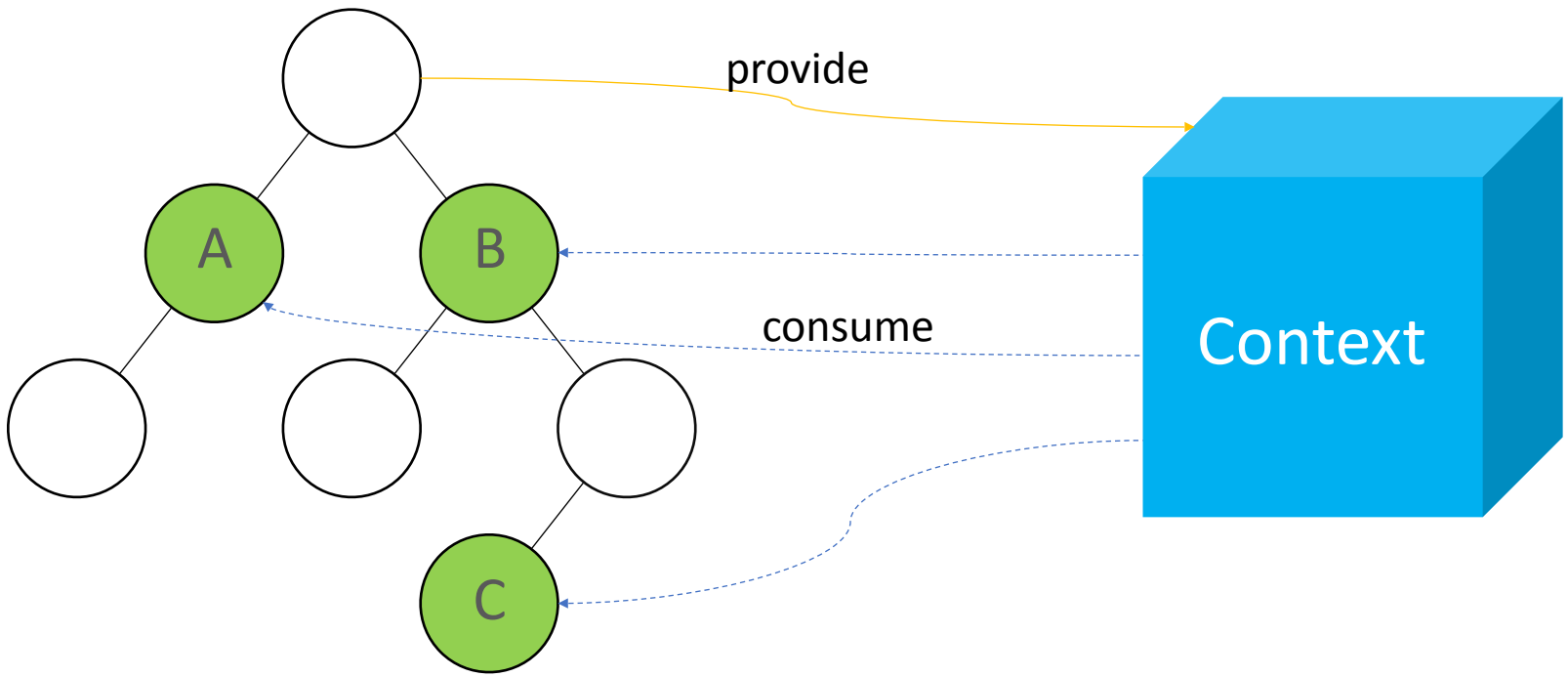
DEMO

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

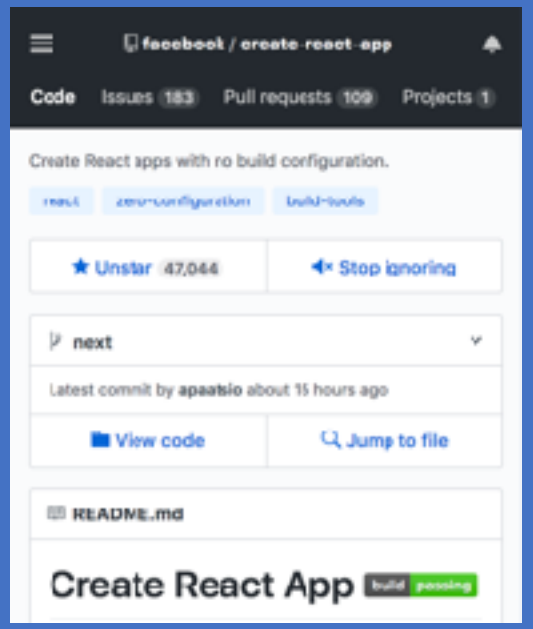
为什么需要脚手架工具

React	Redux	REACT/ROUTER
BABEL	webpack	ESLint

为什么需要脚手架工具

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

create-react-app



Babel

Webpack Config

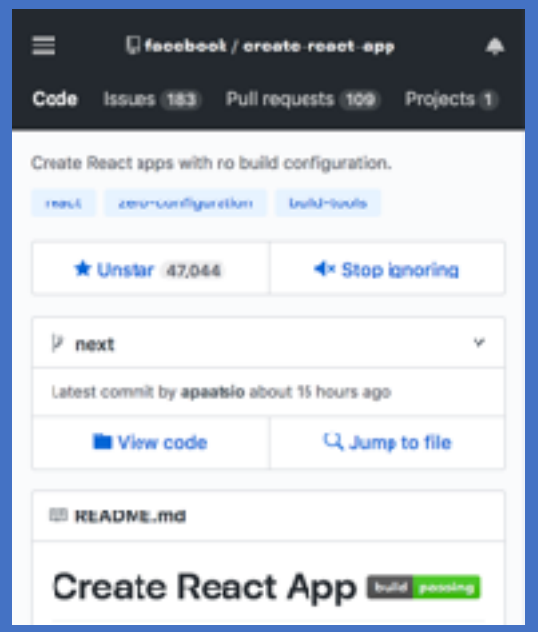
Testing

ESLint

……

Create React App

Rekit



Babel

Webpack Config

Testing

ESLint

.....

Create React App

+

Redux

React Router

Less/Scss

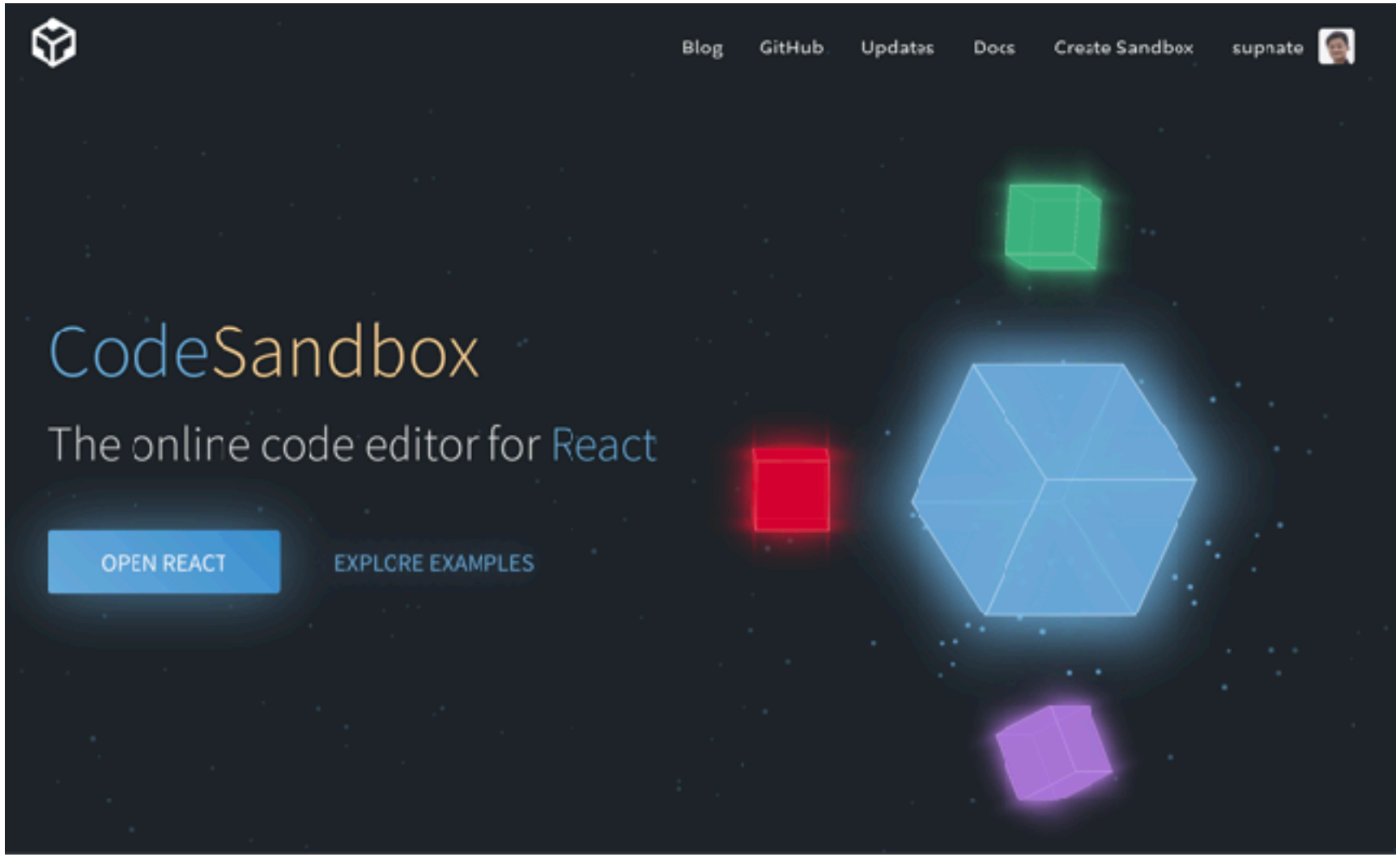
Feature Oriented Architecture

Dedicated IDE

=

Rekit

Online: Codesandbox.io



小结

介绍了3种脚手架工具及它们的使用场景。

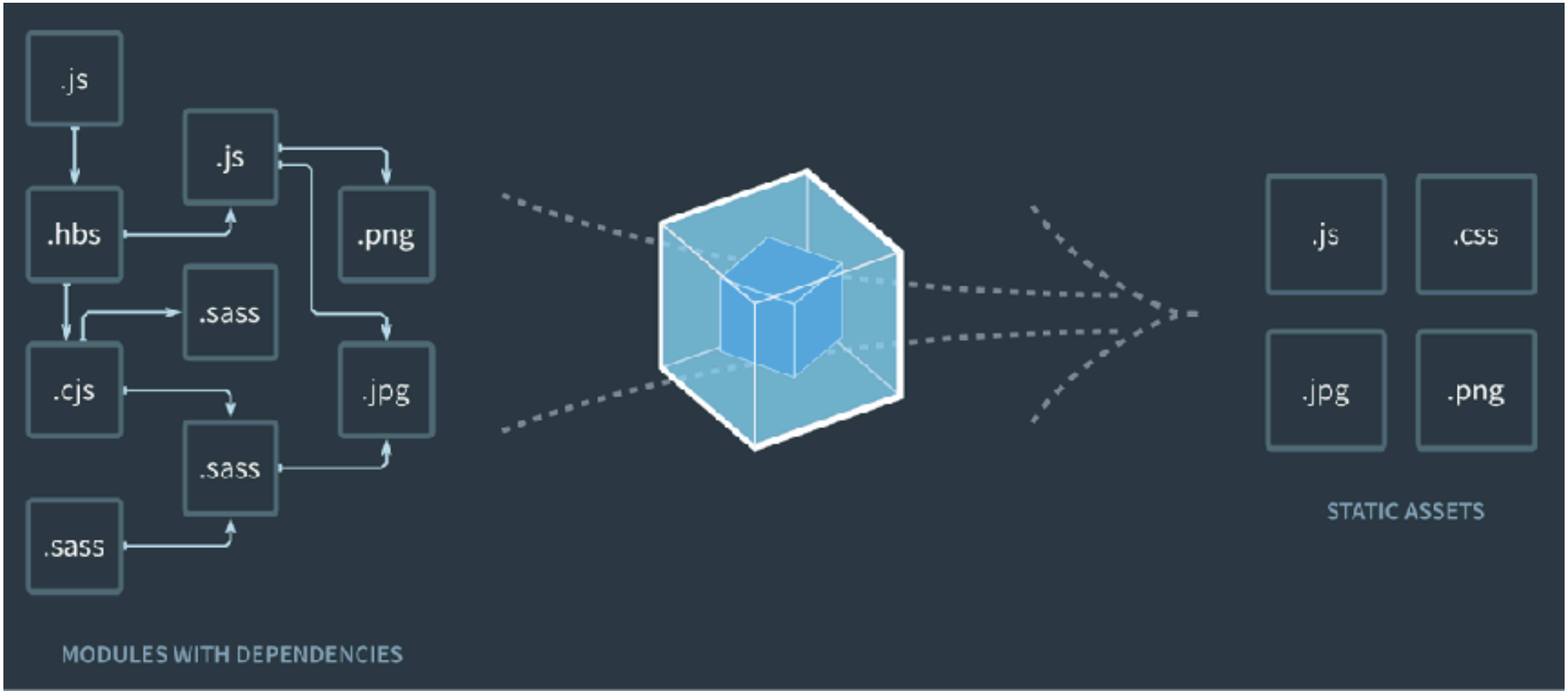
DEMO

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

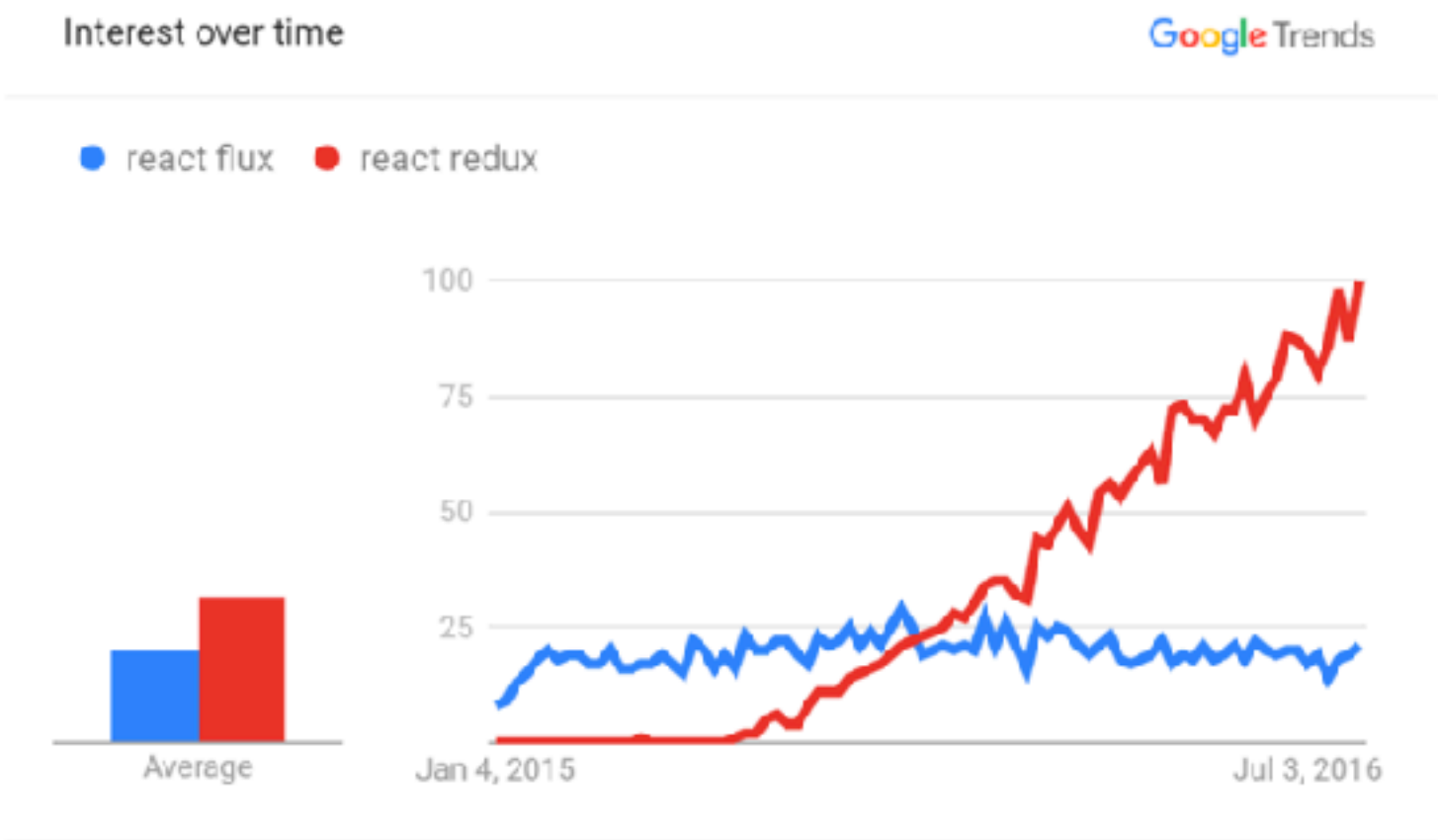
小结

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

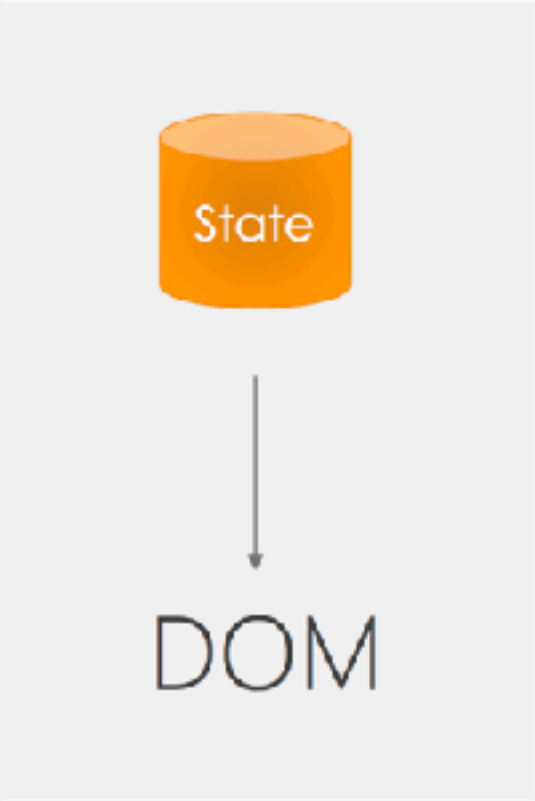
Redux（1）: JS 状态管理框架

Redux

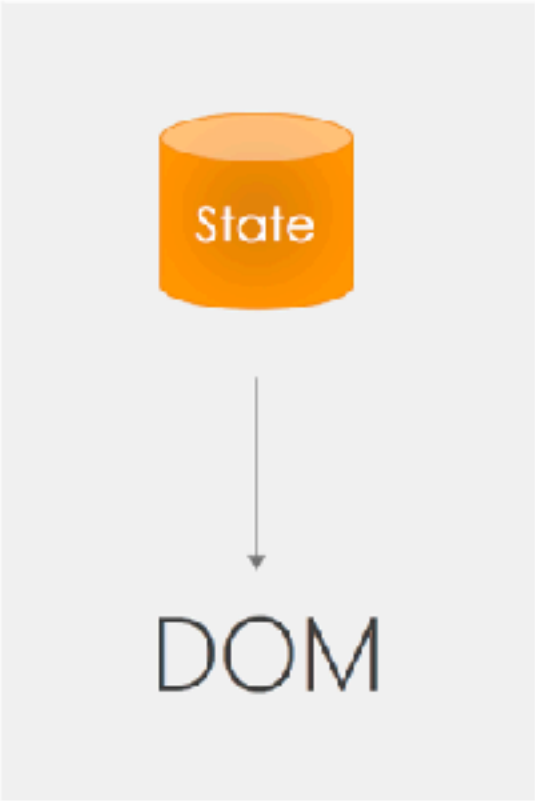
@dan_abramov



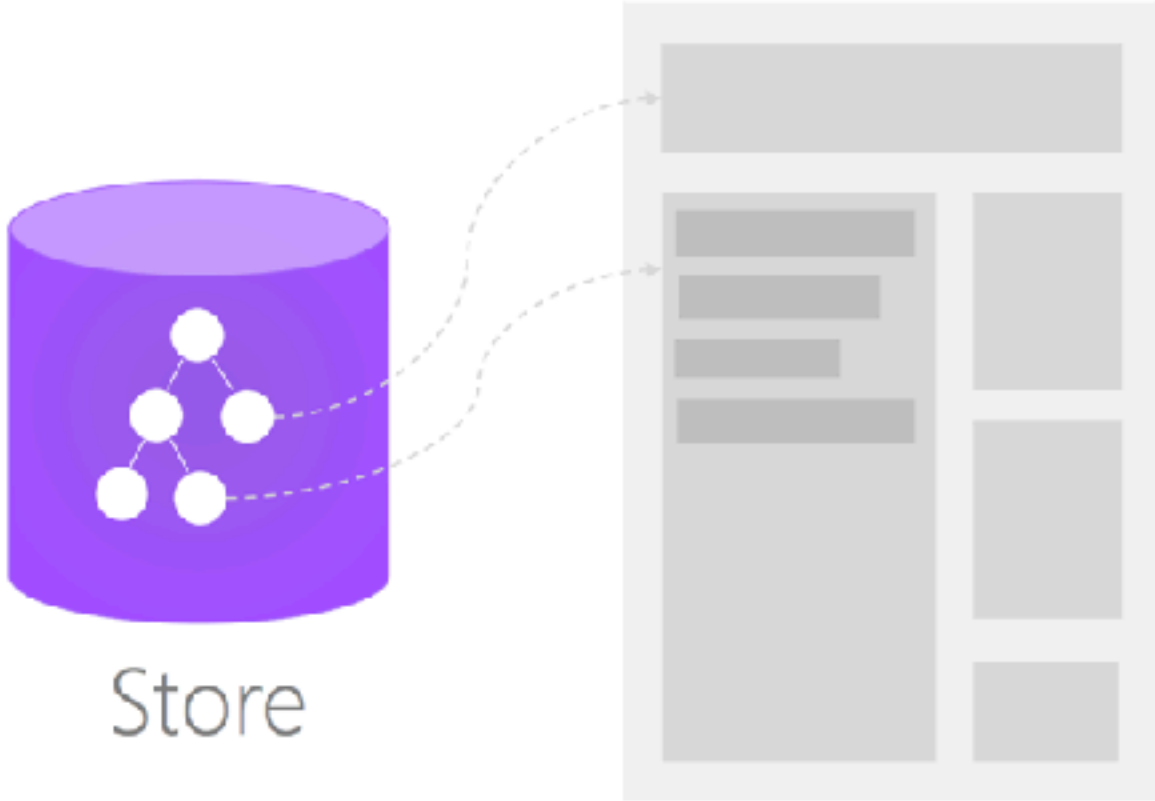
React



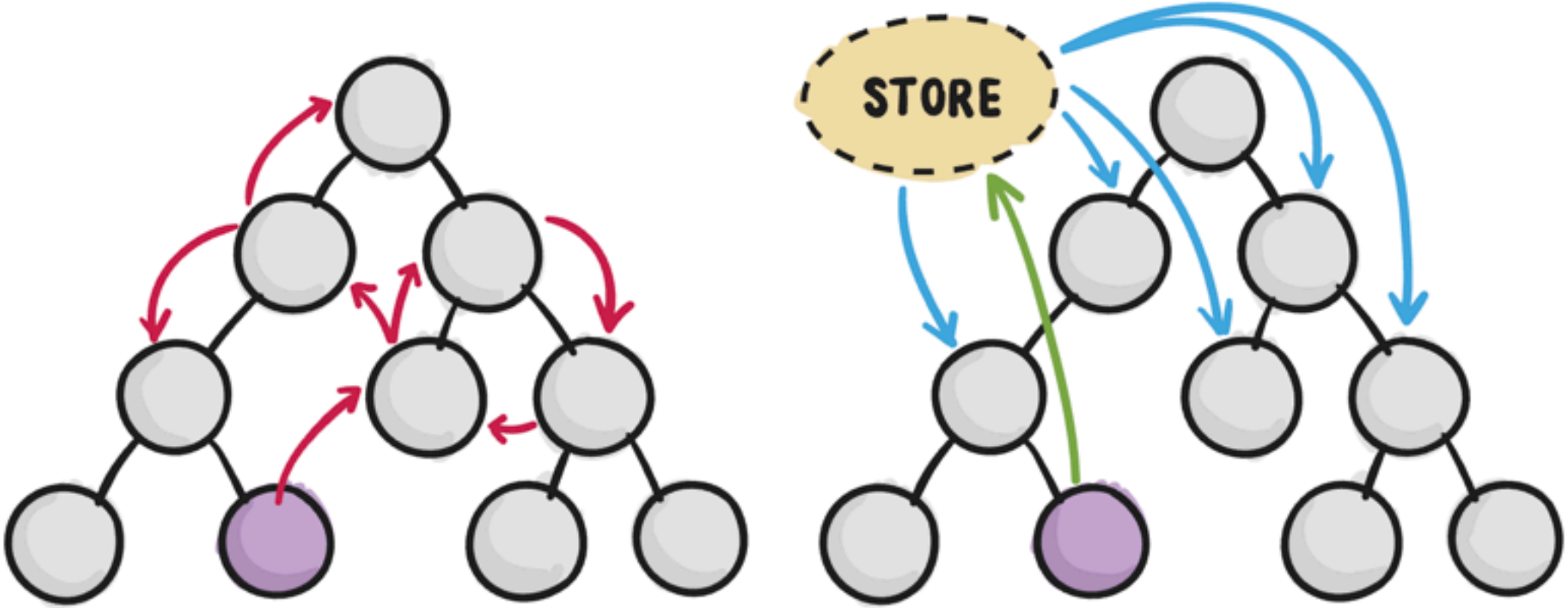
React



Redux

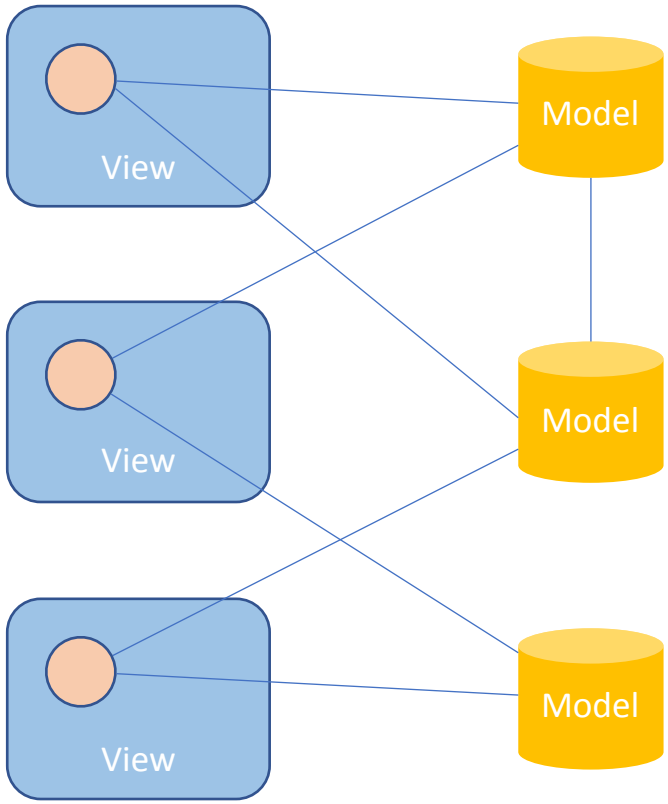


Redux 让组件通信更加容易

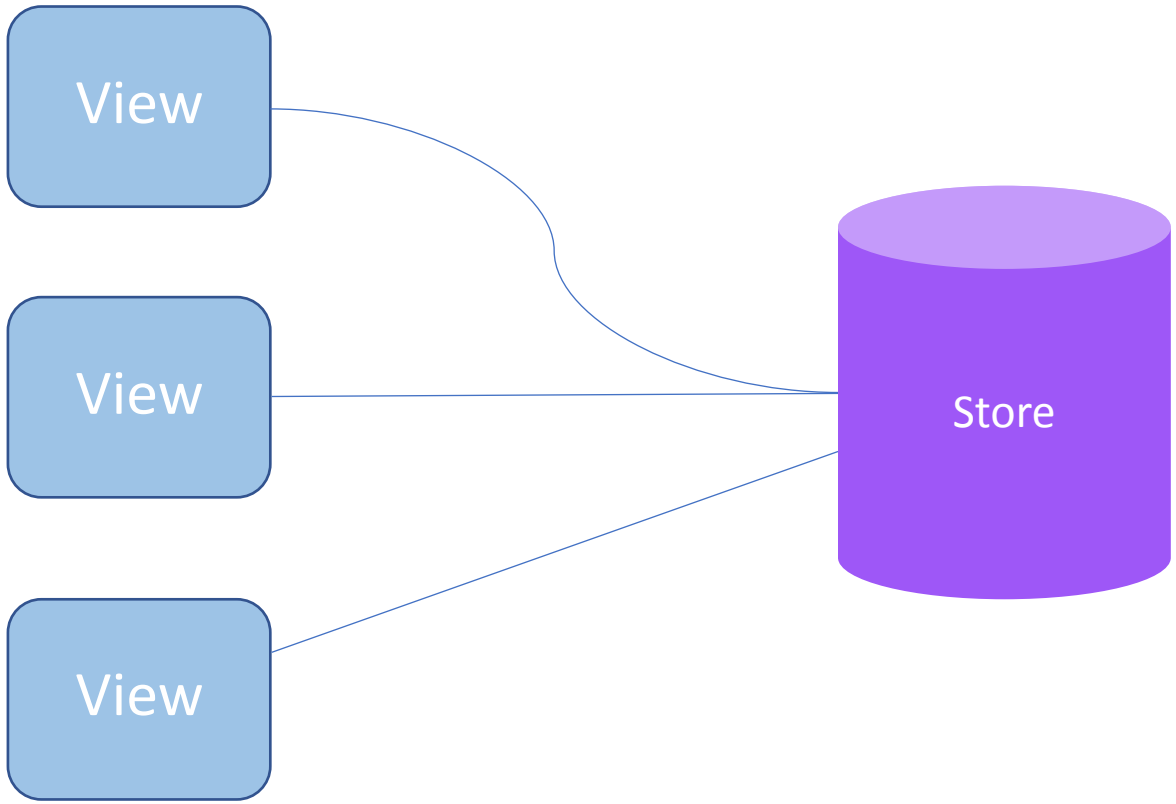


图片来源：https://css-tricks.com/learning-react-redux/

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

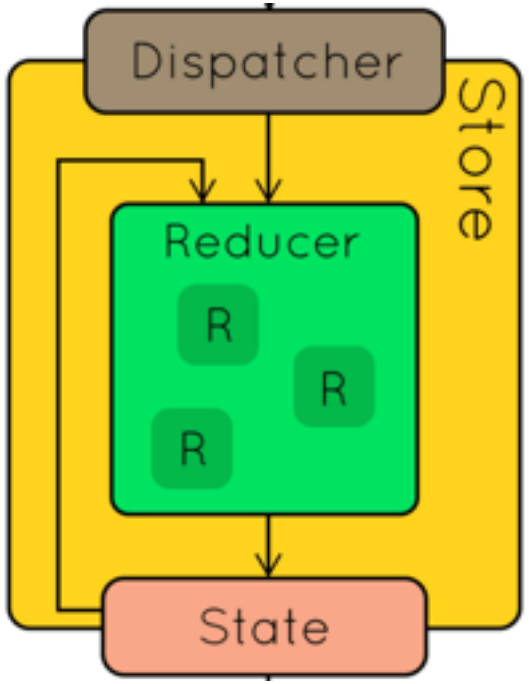
```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map(
        (todo, index) =>
          action.index === index
            ? { text: todo.text, completed: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

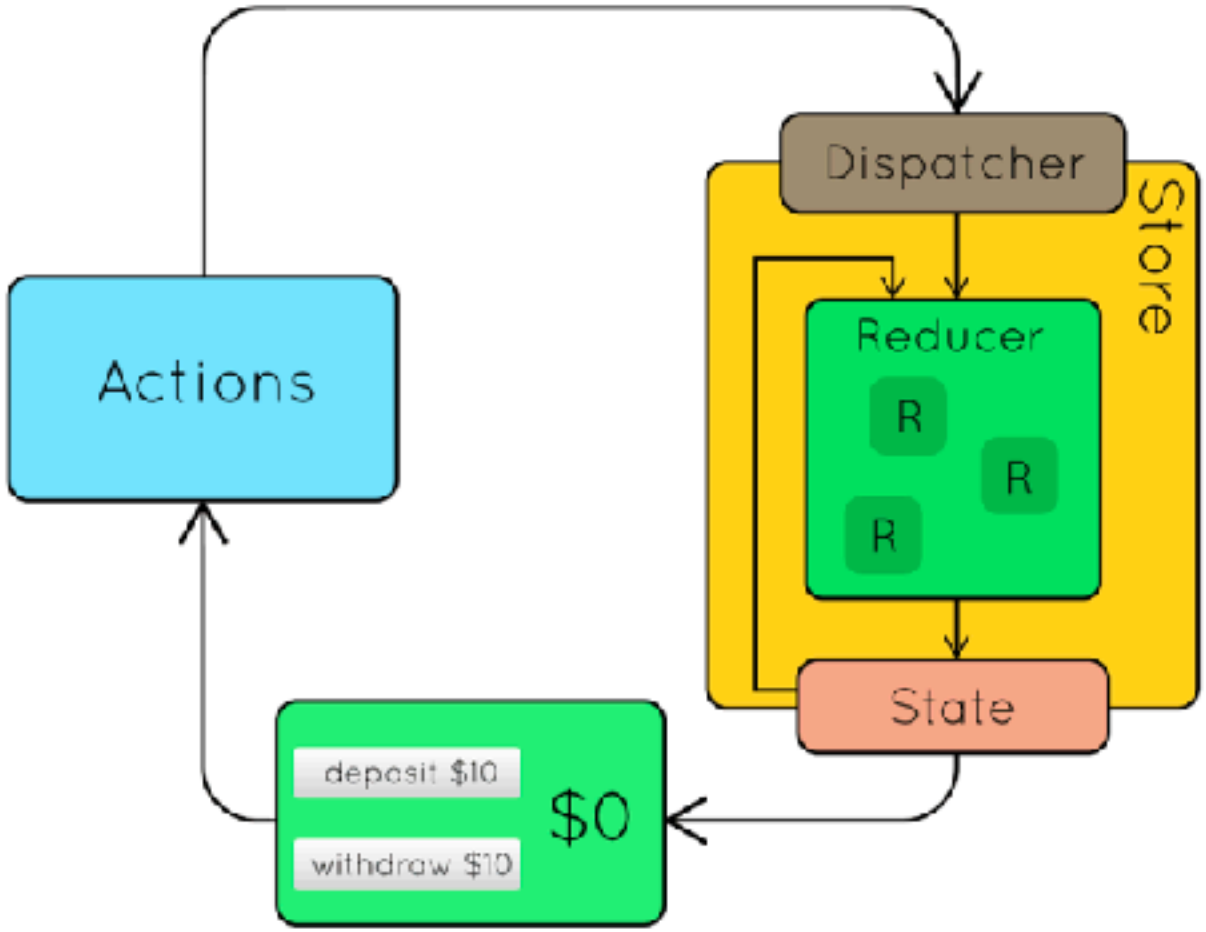
理解 action

```
{
  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



© http://slides.com/jenyaterpil/redux-from-twitter-hype-to-production

极客时间

(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

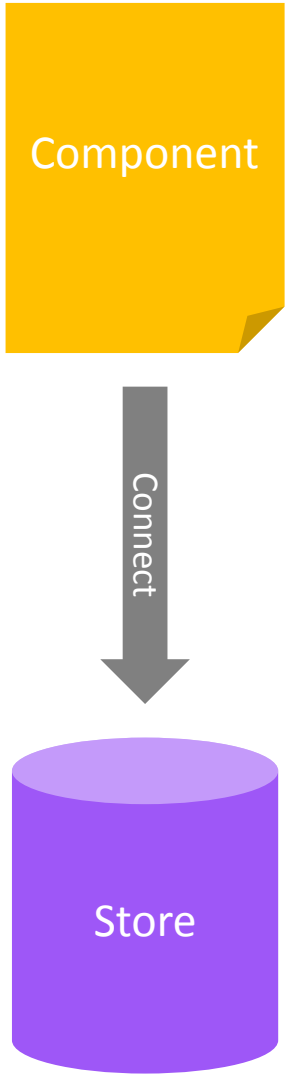
```
function bindActionCreator(actionCreator, dispatch) {
  return function() {
    return dispatch(actionCreator.apply(this, arguments))
  }
}

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] = bindActionCreator(actionCreator, dispatch)
    }
  }
  return boundActionCreators
}
```

DEMO

小结

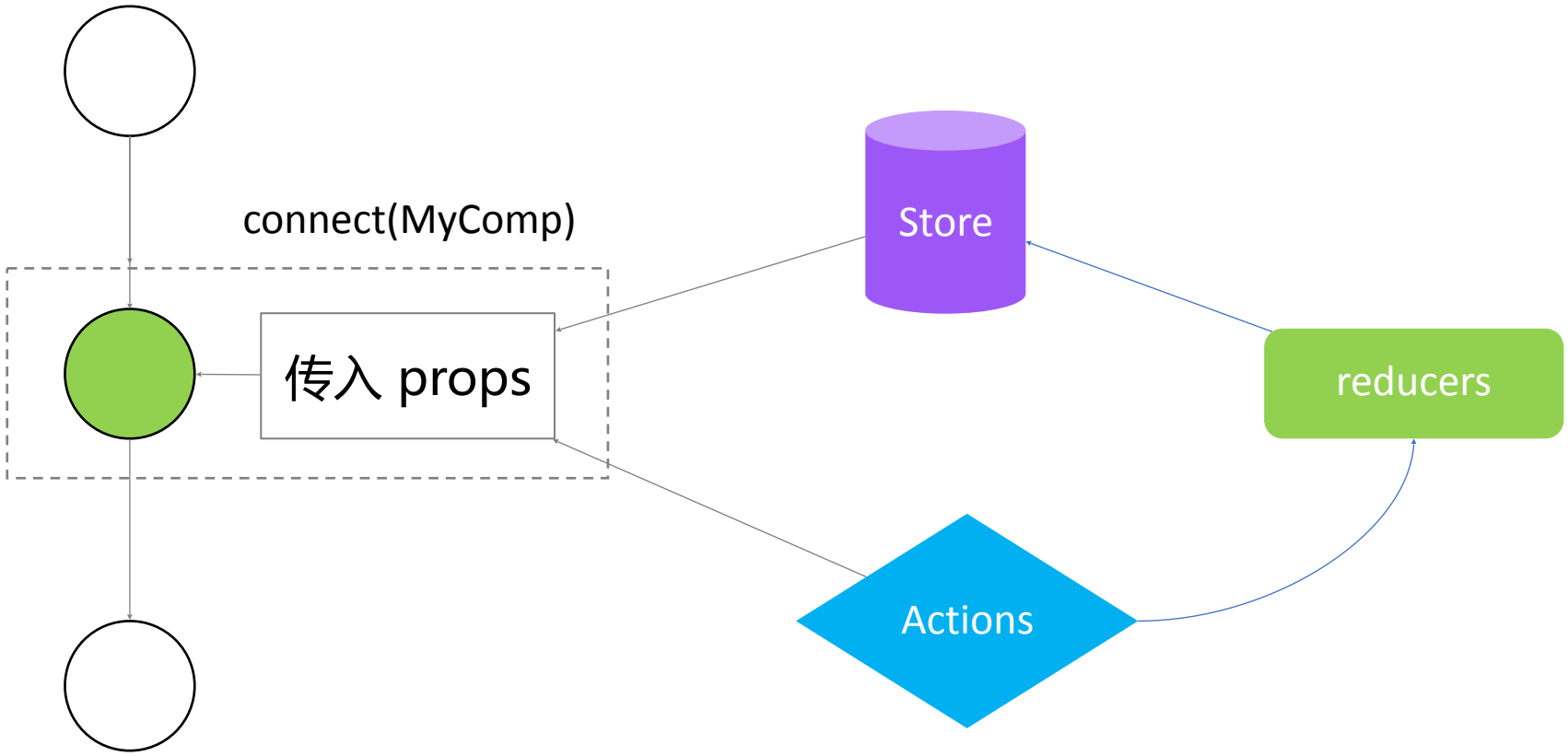
1. Redux 的基本概念
2. combineReducers
3. bindActionCreators

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);
```

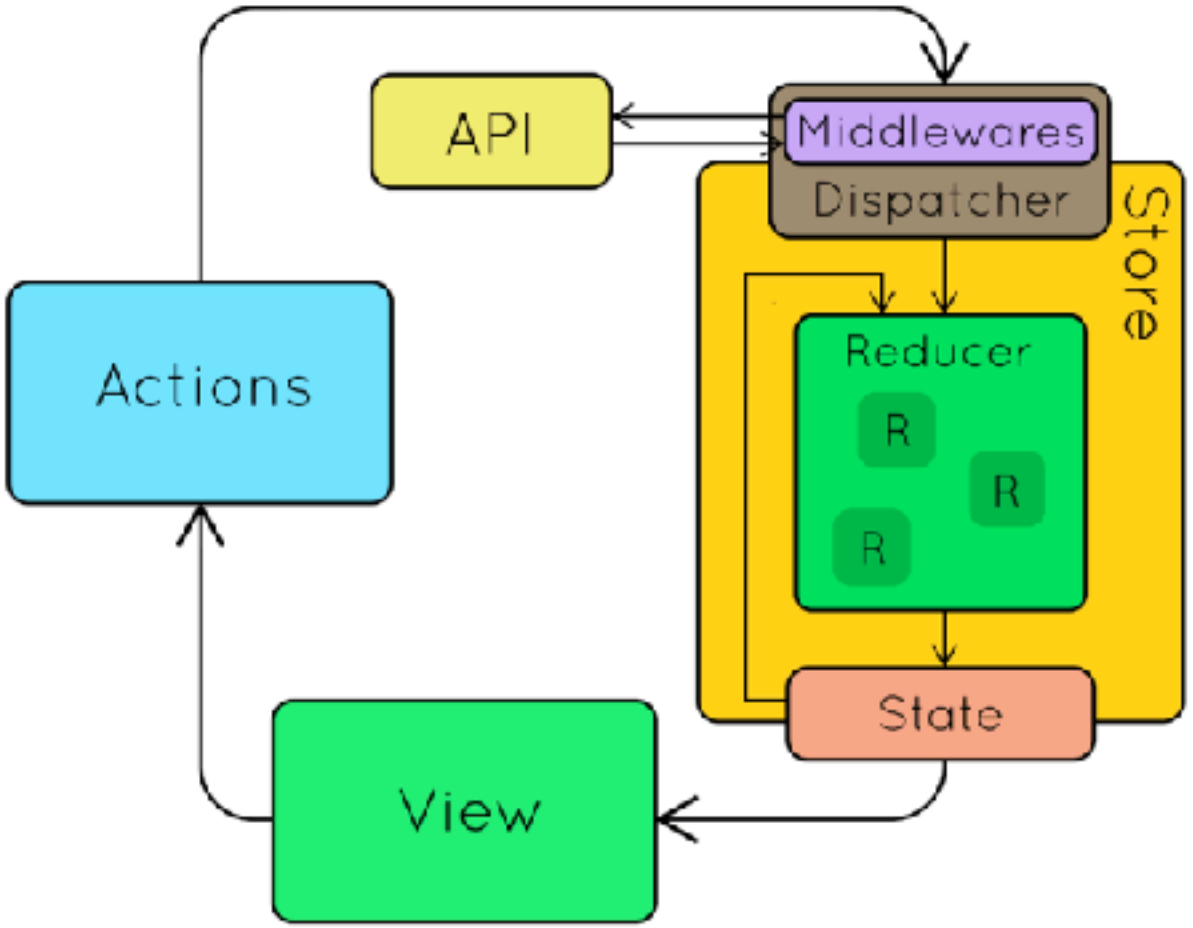
connect 的工作原理：高阶组件



DEMO

Redux（4）：理解异步 Action，Redux 中间件

Redux 异步请求



@ http://slices.com/jenyaterpil/redux-from-twitter-hype-to-production

极客时间

(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

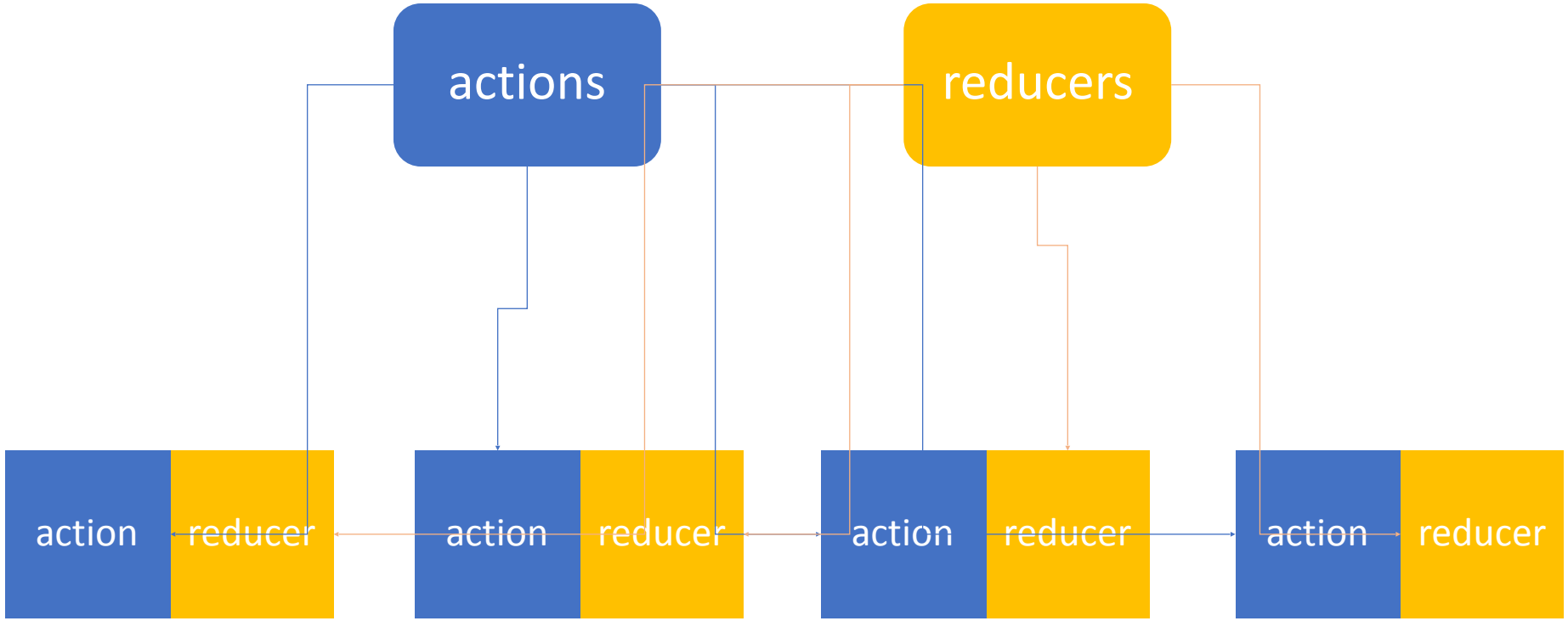
actions.js

```
import {
  ADD_PRODUCT_BEGIN,
  ADD_PRODUCT_SUCCESS,
  ADD_PRODUCT_FAILURE,
  ADD_PRODUCT_DISMISS_ERROR,

  SAVE_PRODUCT_BEGIN,
  SAVE_PRODUCT_SUCCESS,
  SAVE_PRODUCT_FAILURE,
  SAVE_PRODUCT_DISMISS_ERROR,

  DELETE_PRODUCT_BEGIN,
  DELETE_PRODUCT_SUCCESS,
  DELETE_PRODUCT_FAILURE,
  DELETE_PRODUCT_DISMISS_ERROR,

  FETCH_PRODUCT_BEGIN,
  FETCH_PRODUCT_SUCCESS,
  FETCH_PRODUCT_FAILURE,
  FETCH_PRODUCT_DISMISS_ERROR,

  FETCH_PRODUCT_LIST_BEGIN,
  FETCH_PRODUCT_LIST_SUCCESS,
  FETCH_PRODUCT_LIST_FAILURE,
  FETCH_PRODUCT_LIST_DISMISS_ERROR,

  UPDATE_PRODUCT_PICTURE_BEGIN,
  UPDATE_PRODUCT_PICTURE_SUCCESS,
  UPDATE_PRODUCT_PICTURE_FAILURE,
  UPDATE_PRODUCT_PICTURE_DISMISS_ERROR,

  MOVE_PRODUCT_BEGIN,
  MOVE_PRODUCT_SUCCESS,
  MOVE_PRODUCT_FAILURE,
  MOVE_PRODUCT_DISMISS_ERROR,

  FETCH_HOT_PRODUCT_BEGIN,
  FETCH_HOT_PRODUCT_SUCCESS,
  FETCH_HOT_PRODUCT_FAILURE,
  FETCH_HOT_PRODUCT_DISMISS_ERROR,

  CHANGE_PRODUCT_OWNER_BEGIN,
  CHANGE_PRODUCT_OWNER_SUCCESS,
  CHANGE_PRODUCT_OWNER_FAILURE,
```

“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

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

```
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;
  }
}
```

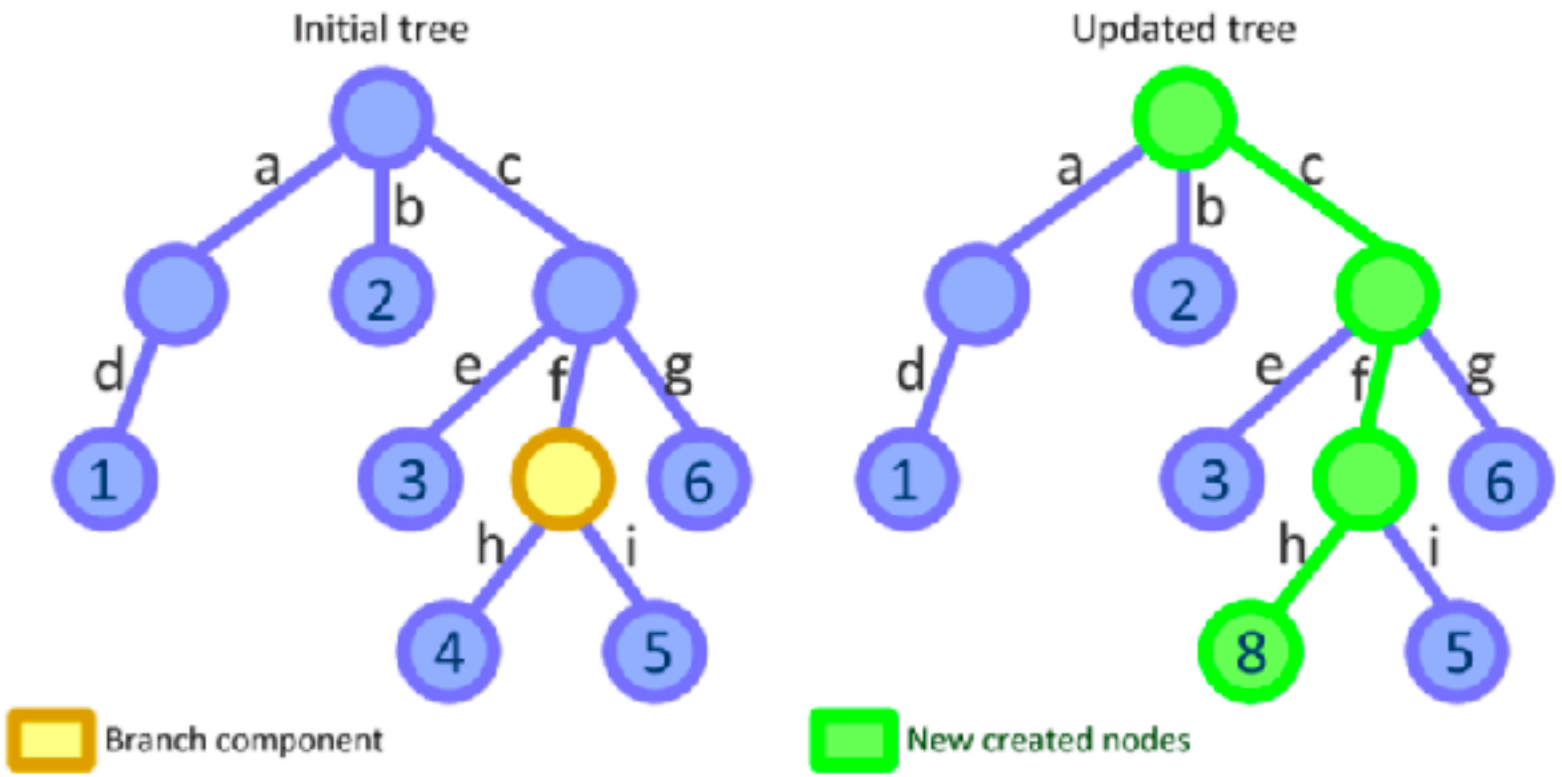
counterPlusOne.js

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

DEMO

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

不可变数据（immutable data)



© http://arqex.com/991/json-editor-react-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