# 主题一：理解React

官网：https://reactjs.bootcss.com/docs/getting-started.html

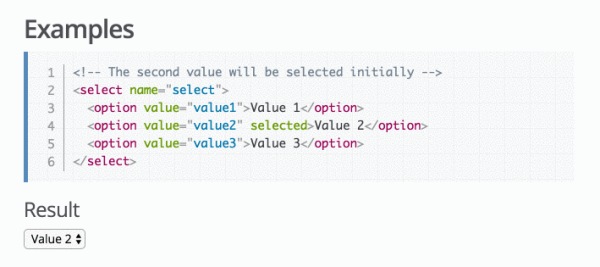
不是说我一篇文章就能能让你摇身一变成为精通React的大神，但如果你打算或者正在学习React的话，我至少能帮你理清思路。现在揭晓这五大核心概念：

1. 组件
2. JSX
3. Props & State
4. 组件API
5. 组件类型

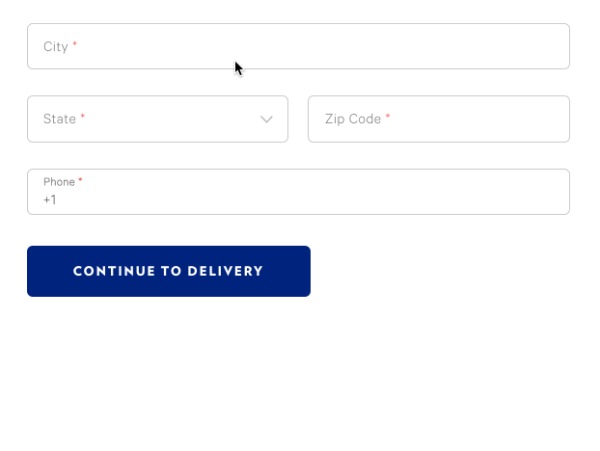
## 1、React组件的作用

你需要了解有关React的第一要点就是——组件。你编写的所有React代码基本上就是一个包含许多小组件在内的大组件。

那么到底什么是组件呢？我们可以拿HTML标签 <select> 来举一个很恰当的例子。原生的下拉框标签不止包括边框、文本、下拉箭头，它还掌控着自身打开关闭的逻辑。



现在来设想一下你需要构建一个你自定义样式和行为逻辑的<select>：



这其实就是React能够帮你做到的。React组件能够像原生的HTML标签一样输出特定的界面元素，并且也能包括一些元素相关逻辑功能的代码。

现在我们一般会用ES6的Class语法来声明一个React组件，它包含一个能够返回HTML的render方法。（当然也可以用函数声明，我们在之后会聊到）

**class MyComponent extends React.Component {**

**render() {**

**return <p>Hello World!<p>;**

**}**

**}**

## 2、JSX是什么玩意儿？

是的你没看错，按照上面React组件的示例代码，React的意思就是让我们把HTML和JS代码全都写在一起。React是通过一种叫做JSX的语法扩展（X代表XML）来实现的。

JSX乍看起来可能很奇怪，不过你慢慢会习惯的。

是的我知道，按照我们以往的传统，应该尽量把HTML和JavaScript的代码分开才是。不过看样子现在忘记这教条才是提高你前端开发效率的正道。

我们还是来举几个JSX实际应用的例子吧，比如你可以通过{}大括号来在JSX中显示JS变量：

**class MyComponent extends React.Component {**

**render() {**

**return <p>Today is: {new Date()}</p>;**

**}**

**}**

你不再需要什么前端模板标签之类的东西了，你可以直接在JSX中使用三元运算符一类的逻辑：

**class MyComponent extends React.Component {**

**render() {**

**return <p>Hello {this.props.someVar ? 'World' : 'Kitty'}</p>;**

**}**

**}**

顺便提一句，可能你现在对ES6还不是特别的了解，那么我推荐你去看看阮老师的[ECMAScript 6 入门](https://es6.ruanyifeng.com/" \t "_blank)。

## 3、Props & State又是啥？

你可能会疑惑上个例子里的this.props.someVar是从哪里冒出来的。

只要你对HTML有所了解，应该能够理解<a>标签的href属性是什么意思。延伸到React当中，属性就被称作props（properties的缩写）。组件之间可以通过Props进行交互。

**class ParentComponent extends React.Component {**

**render() {**

**return <ChildComponent message="Hello World"/>;**

**}**

**}**

**class ChildComponent extends React.Component {**

**render() {**

**return <p>And then I said, “{this.props.message}”</p>;**

**}**

**}**

也正因如此，React当中的数据流是单向的：数据只能从父组件传向子组件，反过来则不行。

可是组件不可能只接受从父组件传来的数据（例如还有用户在input当中的输入），这时state就派上了用场。

原文的例子实在是不接地气，在这里我情不自禁想要引用一句古诗来解释props和state之间的区别：

人有悲欢离合，月有阴晴圆缺。

我们可以把一个人的基因性别名字（我知道某些东西其实是可变的但请不要钻牛角尖好吗），月亮的大小重量理解为props，而随时可变的情感和圆缺则可以被理解为state。

要注意，组件的state同样也能被传入到子组件中作为子组件prop的值。你需要明确的就是在React当中整个数据流都是向下传递的，包括路由、数据层、各个组件等等，从整个应用的state中来并汇聚到一起。

在组建中，我们可以通过一个叫setState的方法来修改state，一般我们都会在事件处理的方法中调用它：

**class MyComponent extends React.Component {**

**handleClick = (e) => {**

**this.setState({clicked: true});**

**}**

**render() {**

**return <a href="#" onClick={this.handleClick}>Click me</a>;**

**}**

**}**

一般React应用当中的绝大多数数据都是prop，只有当用户输入内容时才会使用state来处理。

注意在上述的代码中，我们使用了自动绑定的语法，如果你想了解更多可以阅读官方文档[Handling Events](https://facebook.github.io/react/docs/handling-events.html)。

## 4、组件API

在之前的内容当中我们已经提及了render和setState两个方法，他们都包含在组件API方法之中。还有一个比较有用的方法[constructor](https://facebook.github.io/react/docs/react-component.html" \l "constructor" \t "_blank)，我们一般会在其中初始化state并做一些方法的绑定。

除了这三个方法之外，React还提供了一些列按照特定次序触发的[生命周期函数](https://facebook.github.io/react/docs/state-and-lifecycle.html" \t "_blank)。不过先不需要担心，只有当你深入一些了解React之后才有机会使用到它们。

我们并不会在这里展开篇幅讲解React的API，因为学习React更主要的目的是学习如何编程和它的构建理念，而不是死记硬背一些无聊的API方法。

## 5、组件类型

我们在React当中一般按照如下的方法定义一个组件：

**class MyComponent extends React.Component {**

**render() {**

**return <p>Hello World!<p>;**

**}**

**}**

在Class中我们还可以申明一个组件的许多其他方法，而在更多的情况下我们可以写一种函数式组件。

类似于自定义一个模板标签一样，函数式组件接收一个props参数并返回特定的HTML内容，不过你当然仍可以在其中调用一些JS代码：

**const myComponent = props => {**

**return <p>Hello {props.name}! Today is {new Date()}.</p>**

**}**

因为通常你的组件可能并不需要多么复杂的交互，也不需要多余的其他方法，用函数式写法可以让你的代码更加简洁。

当然在这样的组件当中你也没有办法使用setState方法，也即是说函数式组件没有state，所以也可以被称作是无状态组件。

当然，如果你接触React比较早，可能也见过下面这种写法：

**var Greeting = React.createClass({**

**render: function() {**

**return <h1>Hello, {this.props.name}</h1>;**

**}**

**});**

不同的组件类型也就延伸出了组件角色的概念，人们在实践过程中开始将组件分为两种角色，一种关注UI逻辑，用来展示或隐藏内容；另一种关注数据交互，例如加载服务器端的数据。

这两种组件被称作容器组件和展示组件。分别用来处理不同的业务逻辑：

**//presentational component**

**class CommentList extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return <ul> {this.props.comments.map(renderComment)} </ul>;**

**}**

**renderComment({body, author}) {**

**return <li>{body}—{author}</li>;**

**}**

**}**

**//container component**

**class CommentListContainer extends React.Component {**

**constructor() {**

**super();**

**this.state = { comments: [] }**

**}**

**componentDidMount() {**

**$.ajax({**

**url: "/my-comments.json",**

**dataType: 'json',**

**success: function(comments) {**

**this.setState({comments: comments});**

**}.bind(this)**

**});**

**}**

**render() {**

**return <CommentList comments={this.state.comments} />;**

**}**

**}**

这就又有点类似于view/controller的概念了。不过说来说去只是构建代码的不同方式而已，区分逻辑当然有其好处（例如分离业务逻辑，更好的代码复用），当然你也可以完全不吃这一套。

## 6、高阶组件

最后我们再稍微涉猎一些[高阶组件](https://facebook.github.io/react/docs/higher-order-components.html" \t "_blank)的概念（higher-order components通常缩写为HOCs）。

其实可以把它理解为一个工厂方法，你可以传入一个组件并得到一个HOC返回的附加了更多功能的新组件。HOC不能直接在render方法中调用。如果你想了解更多，可以去看react-router当中实际应用的[例子](https://github.com/ReactTraining/react-router/blob/master/upgrade-guides/v2.4.0.md" \l "withrouter-hoc-higher-order-component" \t "_blank)。

这里还有一篇[深入理解 React 高阶组件](https://zhuanlan.zhihu.com/p/24776678" \t "_blank)。

## 7、总结

1. React的代码是由一个个的组件构成的。
2. 组件采用了JSX语法扩展的写法。
3. 数据流总是从父组件到子组件，除state可以通过调用方法修改之外。
4. 组件包含一些特定方法和生命周期函数。
5. 你也完全可以用函数声明只有render方法的无状态组件。
6. 区分处理UI和数据逻辑的组件是一种很好的开发实践。
7. 高阶组件函数可以传入一个组件并为其赋予更多功能。

信不信由你，目前我们介绍的内容已经涵盖了React开发者在日常工作中应用的90%的知识。听起来可能有些晦涩抽象，单React当中涉及的内容总能被简化为functions和props。

# 主题二：理解Redux和React Redux

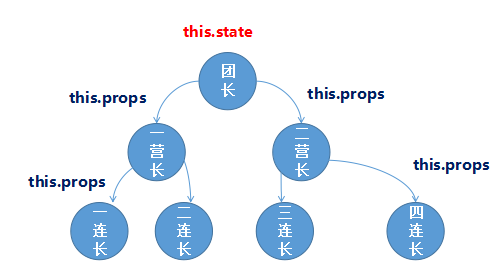
## 1、React组件数据的传递方式

一开始接触redux时对于它的数据state和react本身的数据state还有点容易打结（它们是完全不同的两个东西，后面会讲到）现在搞清楚一点了，来小结一下。

在此之前，还是先讲一下redux是干嘛的，也就是说它是解决什么需求的，这里举一个小例子。

先来看看React里面的数据是怎样传递的（**下图中并不是说this.state只能在“团长”处，这里只是为了简便，所以才这样写，其实营长，连长都可以有自己的this.state**）：

**React数据流：**



**1.**可以看到，一般都是团长等高级干部商量好对策之后（this.state设置好）

**2**. 把命令一层层的传递下去执行（this.props渲染）

**3**. 要是哪里发生了什么新的敌情了（想更新页面内容了）

**4**.就得一层层往上报告（通过回调一层层把数据传上去），此过程也必须一层层往上传递，不能越级，下面有解释

**5**. 团长做出决断后再发布命令传下去执行（通过setState修改数据并重新渲染）

**存在原问题：**

**这个过程是不能越级的**，比如说你一连长发现了敌情就不能直接报告团长，而必须先报告给一营长，一营长再报告给团长，这里层级还不是特别深，你再来个排长，班长什么的，按照这种模式，等团长知道敌情，敌人都打到家门口啦，又或者说一连长想找三连长喝个酒，还得先把这个消息传给营长，再传给团长，再传给二营长再传给三连长，这……

## 2、 **Redux横空出世**

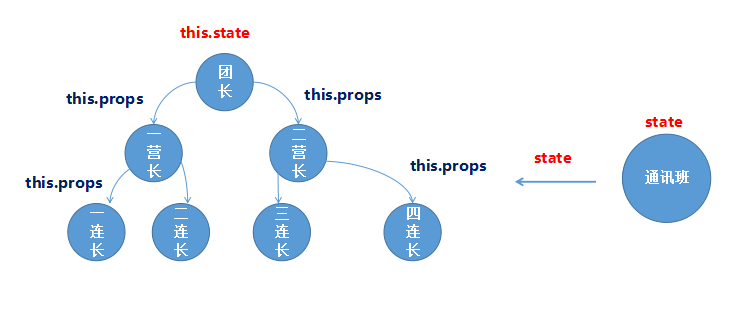
所以说层级不深，组价之间也没有什么公用数据的时候，用React自身的setState其实也可以了，可是嵌套一深，或者组件有一些公用数据时就比较麻烦了，于是就有了redux（其实之前还有个flux），回到这个例子就是，他们在整个团之外又建立了一个**通讯班（Redux）**：

说明：Redux 是 JavaScript 状态容器，提供可预测化的状态管理。可以让你构建一致化的应用，运行于不同的环境（客户端、服务器、原生应用），并且易于测试。Redux 除了和 React 一起用外，还支持其它界面库。

**分清两个state**

在说Redux之前，我想说一下自己学的时候遇到的一个小坑，就是**Redux中的state和React中的state完全不是一回事**，React中的state是组件内部自己的状态信息，而Redux中的state是Redux自己的数据，然后React就拿Redux中的数据来用，其实Redux也可以在其他框架下使用，并不是非要跟React一起使用。

简单画个图看起来就像下面这样：



这里只是一个很简单的示意图，实际使用并非如此，这个图示为了让大家理解React中的数据和Redux中的数据是独立的，并没有半毛钱关系。

**Redux**

现在让我们抛开战斗编制（React），单看通讯班（Redux）。

其实网上讲Redux的教程非常多，这里就简单说一下：

### 2.1 store

store就是redux的一个数据中心，简单的理解就是我们所有的数据都会存放在里面，然后在界面上使用时，从中取出对应的数据。

store:首先要创建一个对象store，这个对象有各种方法，用来让外界获取Redux的数据（store.getState），或者让外界来修改Redux中的数据（store.dispatch）

**import { createStore } from 'redux';**

**const store = createStore(reducer);**

### 2.2 action

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

action:描述我要干啥，一般是一个对象的形式，其中有一个type字段是必须要有的，比如：{ type：‘请求增援’ }，还可以带点数据{ type：‘请求增援’，gun:"100" }

### 2.3 reducer

Reducers 指定了应用状态的变化如何响应 actions 并发送到 store 的，记住 actions 只是描述了有事情发生了这一事实，并没有描述应用如何更新 state。Reducer 是一个函数，它接受 Action 和当前 State 作为参数，返回一个新的 State 。

reducer:撸开袖子真刀实枪的就去干了，比如一连长要求增援，增援要求是100杆枪，团长马上就给你加了100杆枪送了过去。

**const defaultState = 0;**

**const reducer = (state = defaultState, action) => {**

**switch (action.type) {**

**case '请求增援':**

**return state + action.gun;**

**default:**

**return state;**

**}**

**};**

 action和reducer也可以想象成产品经理和程序员的关系。

产品经理：“我要一个按钮，圆角的”

程序员：“嗯，做好了”

产品经理：“换个颜色吧，红色”

程序员：“ok，换好了”

差不多就是这样，产品经理并不操心具体怎么实现的，他只说他想要干什么（type），然后再提点实现的要求（各种其他的数据），程序员就去具体实现（reducer，修改state，然后返回一个**全新的state,**这里注意我们并没有直接返回原来的state，我们返回的是一个全新的state对象，因为reducer函数是一个没有副作用的纯函数）

那怎么去触发这个操作（action）呢，就好比说我一连长发现敌情了，我怎么报告给通讯班，让通讯班来处理呢？

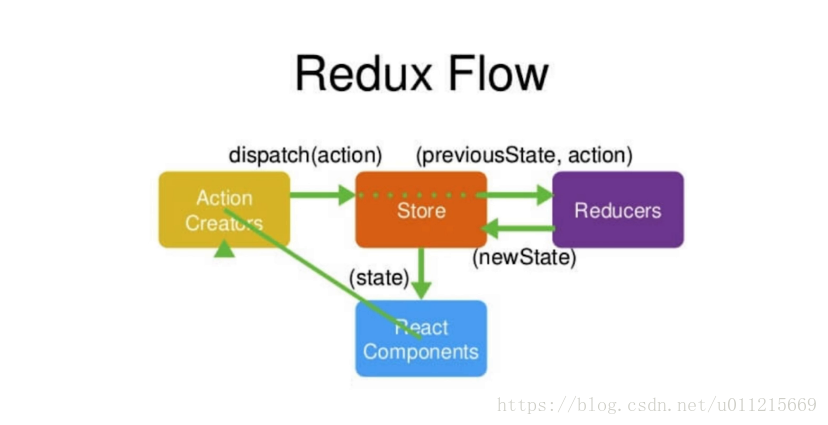
就是上面提到过的store.dispatch，不过还要加一个参数，那就是action

**store.dispatch({ type：‘请求增援’，gun:"100" })**

 这样就可以触发action，执行reducer，得到一个全新的state。

### 2.4 Redux的工作流程

本文中我将用通俗易懂的方式讲述Redux的原理和工作流程



React组件（或其他使用Redux的组件，这里用react组件来进行举例）和Redux的关系

各部分的身份

我将会把这个过程比拟成图书馆的一个流程来帮助理解。

* **Action Creator（具体借书的表达）**

想借书的人向图书馆管理员说明要借的书的那句话。

* **Store（图书馆管理员）**

负责整个图书馆的管理。是Redux的核心

* **Reducers（图书馆管理员的小本本）**

管理员需要借助Reducer（图书馆管理员的小本本）来记录。

* **React Component（借书的人 ）**

需要借书的人

* **工作流程**

借书的人（ReactComponent）说了一句话（Action Creator）向图书馆管理员（Store）借一本书，可是图书馆管理员年纪大了啊记不住啊，便掏出了自己的小本本（Reducers）。看了看知道了那本书有没有，在哪，怎么样。这样一来管理员就拿到了这本书，再把这本书交给了借书人。

翻译过来就是：

组件想要获取State， 用ActionCreator创建了一个请求交给Store,Store借助Reducer确认了该State的状态，Reducer返回给Store一个结果，Store再把这个State转给组件。

## 3、Redux 和 React

到此为止，Redux自己就折腾完了，那么Redux自己的数据并没有用，它要把数据交给React用才行，接下来讲一讲怎么把数据交给React来用。

上面我们创建了一个对象store，我们要把这个store对象作为props传给React，那React就可以用了。

这个store只能有一个，也就只能创建一次，也就是说你必须在最顶层处创建一个store对象，然后再一层层的传递下去，才能让所有的组件都能获得这个store对象，调用它的方法。

### 3.1 获取Redux中的数据

比如说我要在render函数中显示Redux的数据，那么我就可以先获取它的数据：

**store.getState()**

然后把这个数据当做props渲染到组件中去就行了。

### 3.2 更新Redux中的数据

如果你要修改它的数据，那就在JSX中调用

**store.dispatch( { type：‘请求增援’，gun:"100" } )**

### 3.3 响应Redux中的变化

那么这里问题又来了，你调用了store.dispatch之后Redux中的数据确实改变了，可是React并没有什么变化啊。也就是说React中的render函数并没有被触发呀是不是，就好像React中你直接修改React中的state是没有用的而必须调用React的setState才能重新渲染。

因此，为了让Redux的数据一改变我们就重新渲染，Redux自己提供了一个方法叫做

**store.subscribe(render)**

这个函数可以**监听Redux中state的变化**，一旦Redux中的state发生了变化，render函数就会被调用，页面就会被重新渲染。

 上面这个过程就是手动调用的过程，但这样调用有点麻烦，因为要让所有的子组件都能应用store中的数据，那么**所有的组件就都要把store当做props传进来**，这也太麻烦了。

**存在原问题：**

还是那之前那个例子来说，你一连长发现了敌情，是不用层层上报了，可以直接报告给通讯班，通讯班再生成新的命令，可问题是，你从下往上打报告是简单了，可是从上往下发布命令仍然是一层层的传递的呀。

就好比说，一连长发现了敌情，报告上去，通讯班做出决定让三连长带人去打，通讯班还是得通过团长--营长--连长这条路去一层层发布命令，能不能让通讯班直接就通知三连长呢，当然是有的，这就是我们的React-redux库

## 4、React-redux

这个是需要你自己去用npm额外安装的。

**使用这个方法之后，我们就不需要一层层往下发布命令了**

在React-redux中有两个比较关键的概念：Provider和connect方法。

### 4.1 Provider组件

一般我们都将顶层组件包裹在Provider组件之中，这样的话，所有组件就都可以在react-redux的控制之下了,**但是store必须作为参数放到Provider组件中去。**

**<Provider store = {store}>**

**<团长/>**

**<Provider>**

这个组件的目的是让所有组件都能够访问到Redux中的数据。

这个比较简单，我们主要讲connect方法。

### 4.2 connect方法

**connect()是react-redux中的核心方法之一，它将react组件预redux中的Store真正连接在一起。**

**connect(mapStateToProps, mapDispatchToProps)(MyComponent)**

其实connect方法一共有4个参数，这里主要讲前两个。

* **mapStateToProps**

字面含义是把state映射到props中去，意思就是把Redux中的数据映射到React中的props中去。也就是说你React想把Redux中的哪些数据拿过来用。

**特别注意：**任何时候只要 Redux store发生改变，mapStateToProp 函数就会被调用,该回调函数必须返回一个纯对象，该对象会与相应展示组件的 props 合并

比如这里二连这个组件想要渲染自己枪支的数量。就可以直接在二连这个组件中把Redux中的gunOfErlian拿过来用。

**const mapStateToProps = (state) => {**

**return {**

**gun: state.gunOfErlian**

**}**

**}**

然后渲染的时候就可以直接使用this.props.gun

**class Erlian extends Component {**

**constructor(props){**

**super(props);**

**}**

**render(){**

**return(**

**<div>this.props.gun</div>**

**)**

**}**

**}**

**Erlian = connect()(Erlian);**

**export default Erlian;**

那么这样就可以实现渲染，就是把Redux中的state变成React中的props。

* **mapDispatchToProps**

通过上面的分析，相信这个函数也很好理解，就是把各种dispatch也变成了props让你可以直接使用。然后就到了我们这里最重要的一点了。

**const mapDispatchToProps = (dispatch) => {**

**return {**

**onClick: () => {**

**dispatch({**

**type: '请求增援',  
　　　　 gun : 100**

**});**

**}**

**};**

**}**

更改一下上面的Erlian组件

**class Erlian extends Component {**

**constructor(props){**

**super(props);**

**}**

**render(){**

**return(**

**<div>this.props.gun</div>**

**<button onClick = {this.props.onClick}>请求增援</button>**

**)**

**}**

**}**

**Erlian = connect()(Erlian);**

**export default Erlian;**

当我点击请求增援按钮后，Erlian组件的枪支数量会自动的更新，而**不需要**我们手动的去用store.subscribe订阅render函数以达到更新页面的目的。

**这样一来我们就不需要一层层的传递store对象了。**

这种随处都可以使用、修改Redux中的数据的方式确实很方便，但Redux推荐的最佳实践还是在尽可能少的地方使用connect，把逻辑，数据相关的都放到容器组件中去处理，其他的组件都由容器组件所生成的props一层层传递下去然后渲染（傻瓜组件），这里就不多说了。

# 主题三：理解Dva

官网：https://dvajs.com/guide/

## 1、dva 是什么

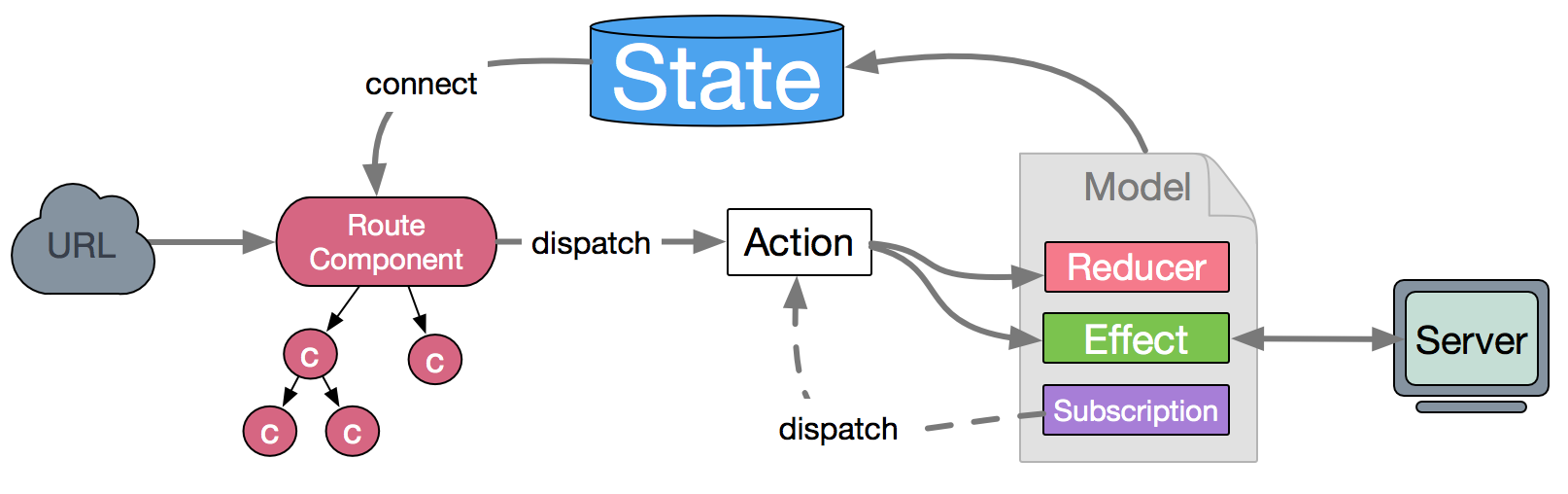
dva 是体验技术部开发的 React 应用框架，将React-Router、Redux和Redux-saga三个 React 工具库包装在一起，简化了 API，让开发 React 应用更加方便和快捷。

dva = React-Router + Redux + Redux-saga

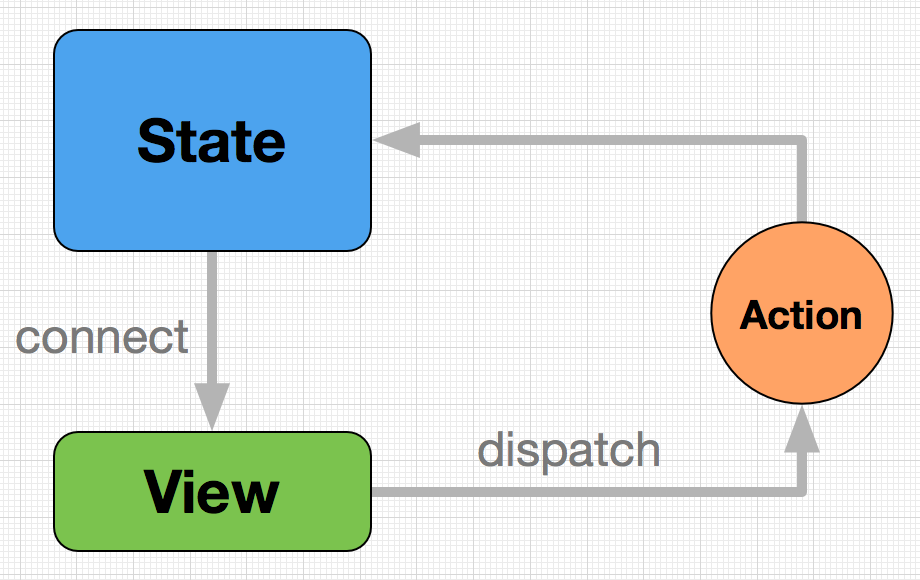
* 路由： React-Router
* 架构： Redux
* 异步操作： Redux-saga

## 2、数据流向

数据的改变发生通常是通过用户交互行为或者浏览器行为（如路由跳转等）触发的，当此类行为会改变数据的时候可以通过 dispatch 发起一个 action，如果是同步行为会直接通过 Reducers 改变 State ，如果是异步行为（副作用）会先触发 Effects 然后流向 Reducers 最终改变 State，所以在 dva 中，数据流向非常清晰简明，并且思路基本跟开源社区保持一致（也是来自于开源社区）。



简化的数据流图：



核心概念

* State：一个对象，保存整个应用状态
* View：React 组件构成的视图层
* Action：一个对象，描述事件
* connect 方法：一个函数，绑定 State 到 View
* dispatch 方法：一个函数，发送 Action 到 State

## 3、领域模型Model

model 是 dva 中最重要的概念，Model 非 MVC 中的 M，而是领域模型，用于把数据相关的逻辑聚合到一起，几乎所有的数据，逻辑都在这边进行处理分发。

### 3.1 State

type State = any

State表示 Model的状态数据，通常表现为一个javascript 对象（当然它可以是任何值）；操作的时候每次都要当作不可变数据（immutable data）来对待，保证每次都是全新对象，没有引用关系，这样才能保证 State 的独立性，便于测试和追踪变化。

### 3.2 Action

type AsyncAction = any

Action 是一个普通 javascript 对象，它是改变 State 的唯一途径。无论是从 UI 事件、网络回调，还是 WebSocket 等数据源所获得的数据，最终都会通过 dispatch 函数调用一个 action，从而改变对应的数据。action 必须带有 type 属性指明具体的行为，其它字段可以自定义，如果要发起一个 action 需要使用 dispatch 函数；需要注意的是 dispatch 是在组件 connect Models以后，通过 props 传入的。

### 3.3 dispatch 函数

type dispatch = (a: Action) => Action

dispatching function 是一个用于触发 action 的函数，action 是改变 State 的唯一途径，但是它只描述了一个行为，而 dipatch 可以看作是触发这个行为的方式，而 Reducer 则是描述如何改变数据的。

在 dva 中，connect Model 的组件通过 props 可以访问到 dispatch，可以调用 Model 中的 Reducer 或者 Effects。

### 3.4 Reducer

type Reducer<S, A> = (state: S, action: A) => S

Reducer（也称为 reducing function）函数接受两个参数：之前已经累积运算的结果和当前要被累积的值，返回的是一个新的累积结果。该函数把一个集合归并成一个单值。

### 3.5 Effect

Effect 被称为副作用，在我们的应用中，最常见的就是异步操作。它来自于函数编程的概念，之所以叫副作用是因为它使得我们的函数变得不纯，同样的输入不一定获得同样的输出。

dva 为了控制副作用的操作，底层引入了redux-sagas做异步流程控制，由于采用了generator的相关概念，所以将异步转成同步写法，从而将effects转为纯函数。

### 3.6 Subscription

Subscriptions 是一种从 源 获取数据的方法，它来自于 elm。

Subscription 语义是订阅，用于订阅一个数据源，然后根据条件 dispatch 需要的 action。数据源可以是当前的时间、服务器的 websocket 连接、keyboard 输入、geolocation 变化、history 路由变化等等。

## 4、Router

这里的路由通常指的是前端路由，由于我们的应用现在通常是单页应用，所以需要前端代码来控制路由逻辑，通过浏览器提供的 History API 可以监听浏览器url的变化，从而控制路由相关操作。

dva 实例提供了 router 方法来控制路由，使用的是react-router。

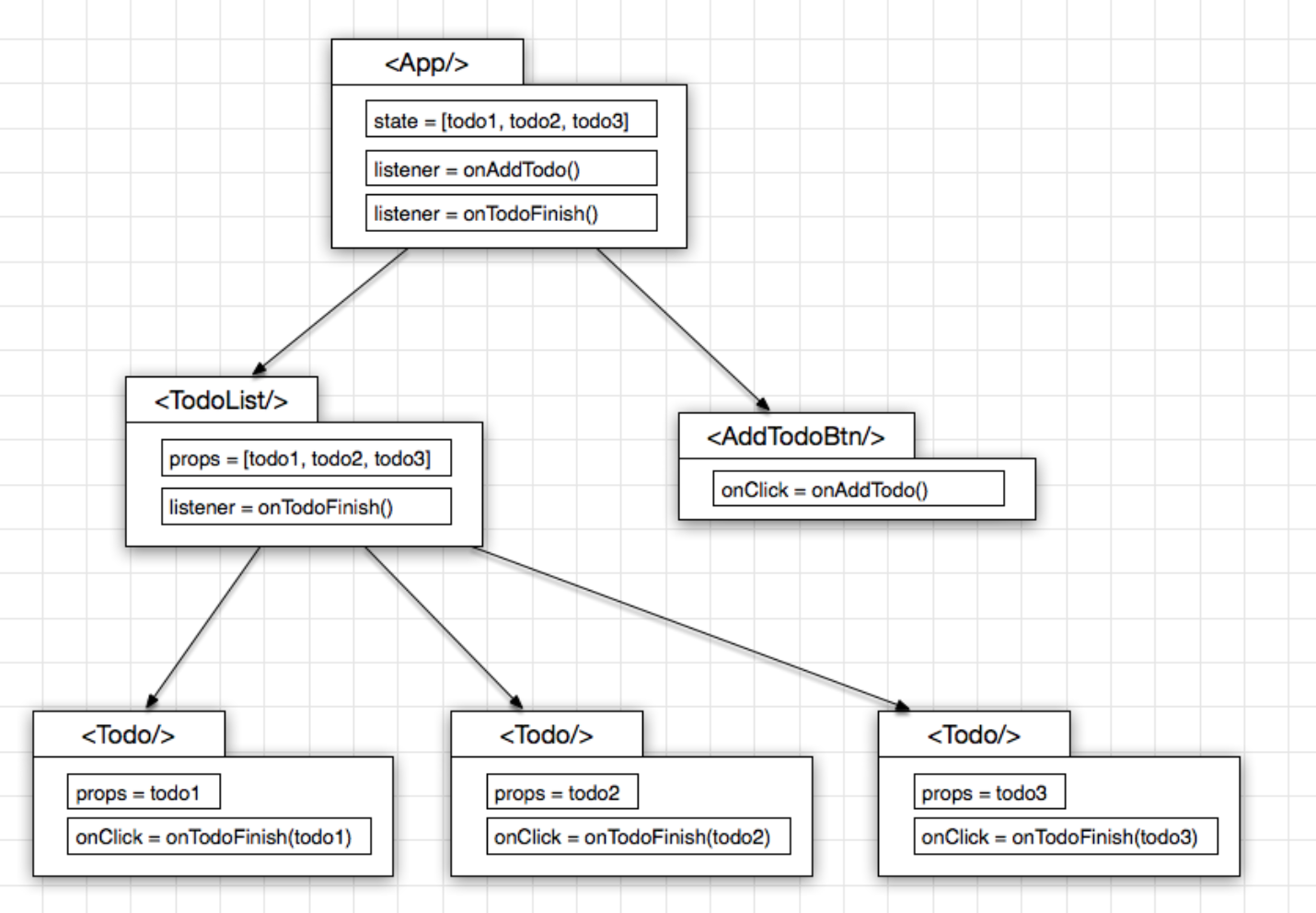
## 5、Route Components

在组件设计方法中，我们提到过Container Components，在dva中我们通常将其约束为 Route Components，因为在dva中我们通常以页面维度来设计Container Components。

所以在dva中，通常需要connect Model的组件都是Route Components，组织在/routes/目录下，而/components/目录下则是纯组件（Presentational Components）。

## 6、图解DVA

### 图解一: React 表示法

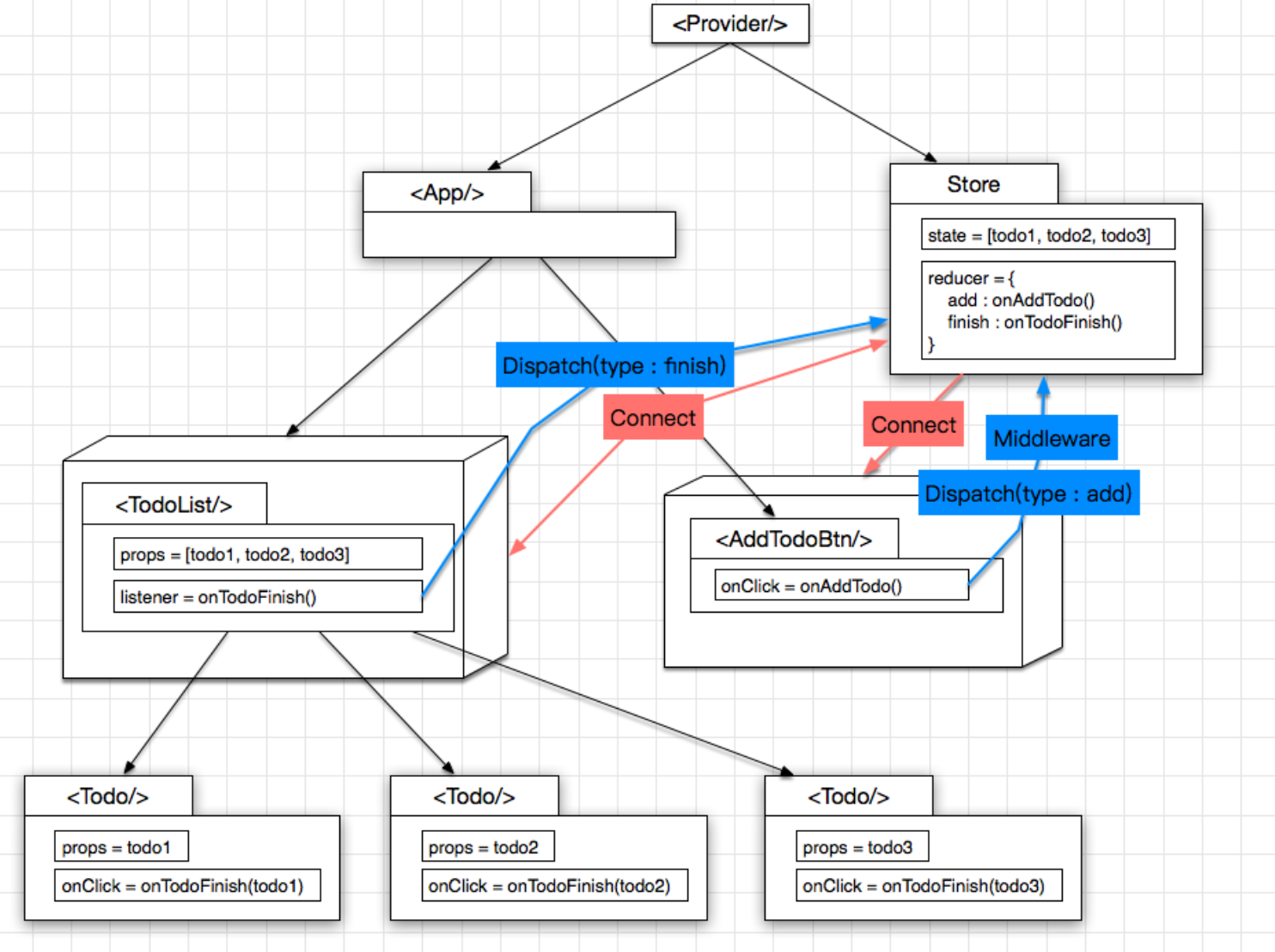


按照 React 官方指导意见, 如果多个 Component 之间要发生交互, 那么状态(即: 数据)就维护在这些 Component 的最小公约父节点上, 也即是 <App/>

<TodoList/> <Todo/> 以及<AddTodoBtn/> 本身不维持任何 state, 完全由父节点<App/> 传入 props 以决定其展现, 是一个纯函数的存在形式, 即: Pure Component

### 图解二: Redux 表示法

React 只负责页面渲染, 而不负责页面逻辑, 页面逻辑可以从中单独抽取出来, 变成 store



与图一相比, 几个明显的改进点:

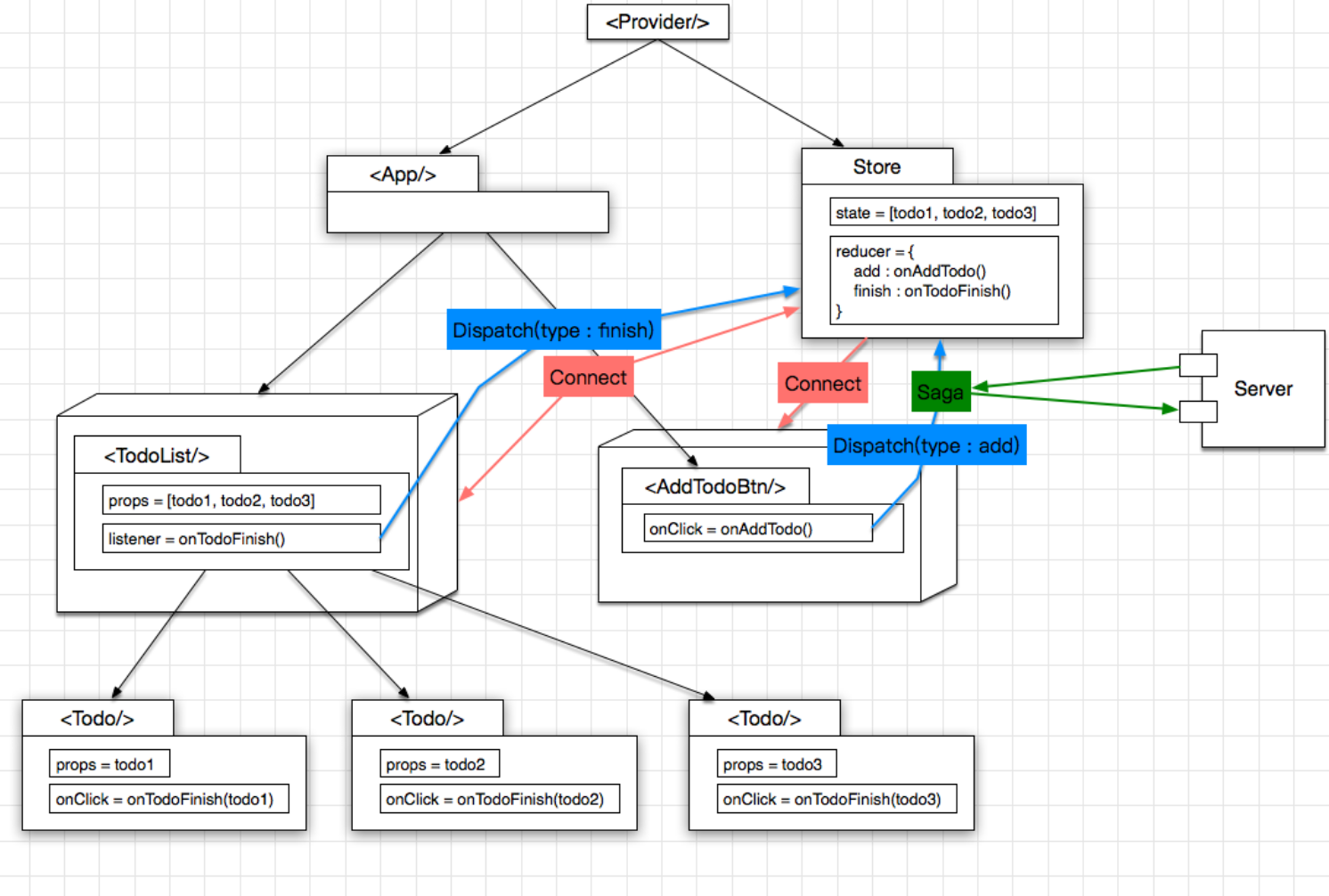
1. 状态及页面逻辑从 <App/>里面抽取出来, 成为独立的 store, 页面逻辑就是 reducer

2. <TodoList/> 及<AddTodoBtn/>都是 Pure Component, 通过 connect 方法可以很方便地给它俩加一层 wrapper 从而建立起与 store 的联系: 可以通过 dispatch 向 store 注入 action, 促使 store 的状态进行变化, 同时又订阅了 store 的状态变化, 一旦状态有变, 被 connect 的组件也随之刷新

3. 使用 dispatch 往 store 发送 action 的这个过程是可以被拦截的, 自然而然地就可以在这里增加各种 Middleware, 实现各种自定义功能, eg: logging

这样一来, 各个部分各司其职, 耦合度更低, 复用度更高, 扩展性更好

### 图解三: 加入 Saga

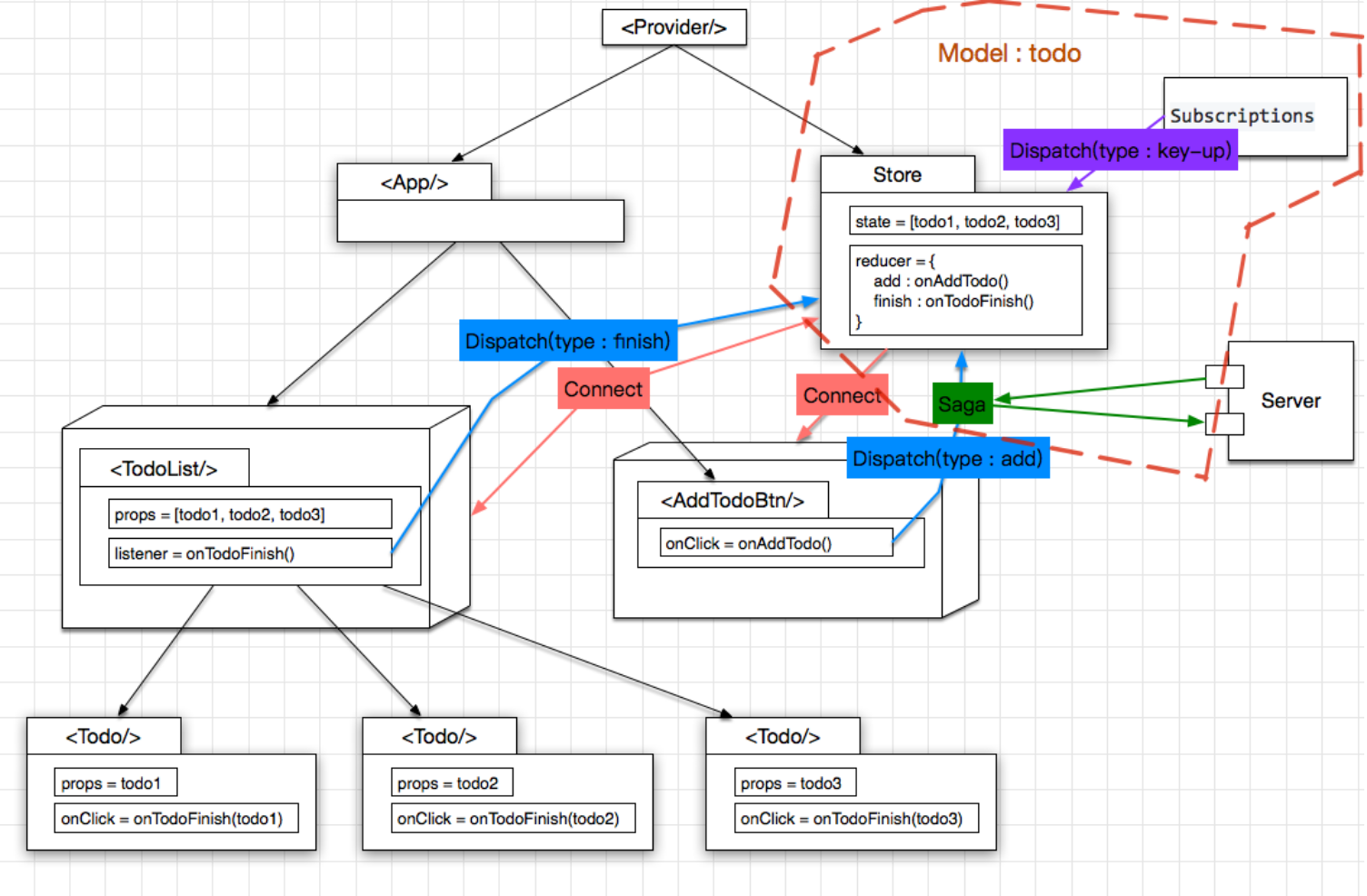


上面说了, 可以使用 Middleware 拦截 action, 这样一来异步的网络操作也就很方便了, 做成一个 Middleware 就行了, 这里使用 redux-saga 这个类库, 举个栗子:

1. 点击创建 Todo 的按钮, 发起一个 type == addTodo 的 action

2. saga 拦截这个 action, 发起 http 请求, 如果请求成功, 则继续向 reducer 发一个 type == addTodoSucc 的 action, 提示创建成功, 反之则发送 type == addTodoFail 的 action 即可

### 图解四: Dva 表示法



有了前面的三步铺垫, Dva 的出现也就水到渠成了, 正如 Dva 官网所言, Dva 是基于 React + Redux + Saga 的最佳实践沉淀, 做了 3 件很重要的事情, 大大提升了编码体验:

1. 把 store 及 saga 统一为一个 model 的概念, 写在一个 js 文件里面

2. 增加了一个 Subscriptions, 用于收集其他来源的 action, eg: 键盘操作

3. model 写法很简约, 类似于 DSL 或者 RoR, coding 快得飞起✈️

# 主题四：理解Umi

## 1、什么是Umi.js?

umi，中文可发音为乌米，是一个可插拔的企业级 react 应用框架。你可以将它简单的理解为一个专注性能的类 next.js 前端框架，并通过约定、自动生成和解析代码等方式来辅助开发，减少我们开发者的代码量。

## 2、为什么使用Umi.js?

我们做react开发的时候会不会遇到以下问题？：

1.项目做大的时候，开发调试的启动和热更新时间会变得很长。

2.大应用下，网站打开很慢，有没有办法基于路由做到按需加载。

3.dva的model每次都要手写载入，能否一开始就同项目初始化好？

使用乌米，即可解决以上问题，并且还能提供如下优势：

* 开箱即用，内置 react、react-router 等
* 类 next.js 且功能完备的路由约定，同时支持配置的路由方式
* 完善的插件体系，覆盖从源码到构建产物的每个生命周期
* 一键兼容到 IE9
* 完善的 TypeScript 支持
* 与 dva 数据流的深入融合

# 主题五：React.Component组件

React 的组件可以定义为 class 或函数的形式。如需定义 class 组件，需要继承 React.Component：

**class Welcome extends React.Component {**

**render() {**

**return <h1>Hello, {this.props.name}</h1>;**

**}**

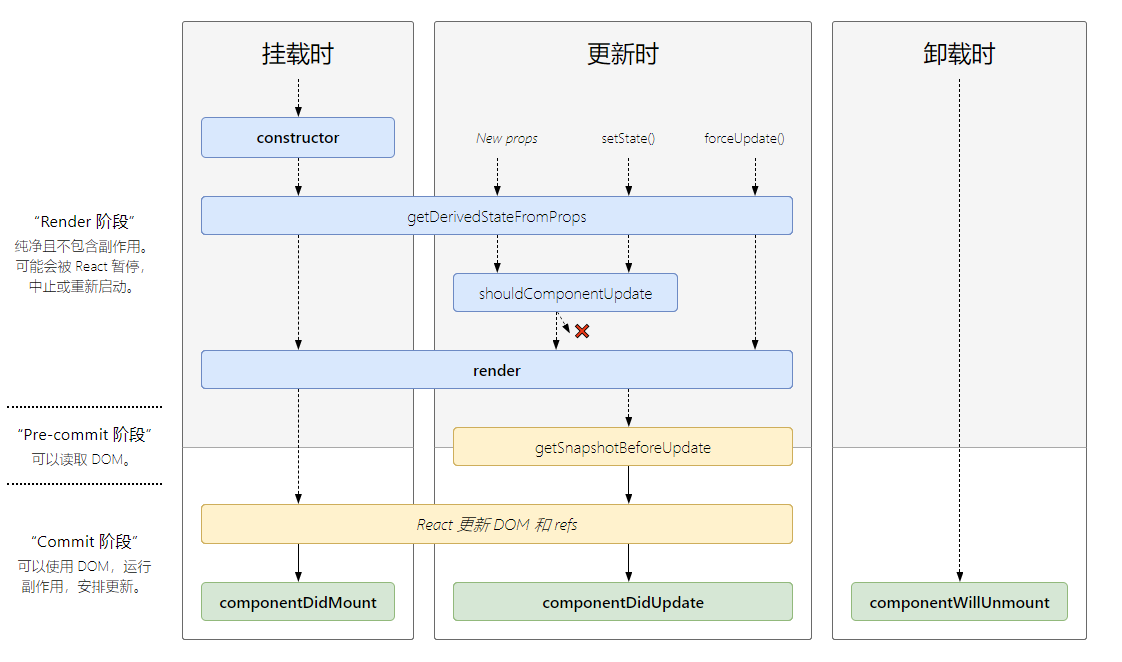
**}**

在 React.Component 的子类中有个必须定义的 render() 函数。

**我们强烈建议你不要创建自己的组件基类。**

## 1、React组件的生命周期

React组件的生命周期函数，只需要掌握constructor、componentWillMount、render、componentDidMount、componentWillUnmount这五个函数：



* **当组件实例被创建并插入DOM 中时：**
* constructor()
* render()
* componentDidMount()
* **当组件的 props 或 state 发生变化时会触发更新：**
* shouldComponentUpdate()
* render()
* componentDidUpdate()
* **当组件从 DOM 中移除时：**
* componentWillUnmount()

## 2、主要生命周期函数说明

### 2.1 constructor()

**形式：**

**constructor(props)**

如果不初始化 state 或不进行方法绑定，则不需要为 React 组件实现构造函数。

在 React 组件挂载之前，会调用它的构造函数。在为React.Component子类实现构造函数时，应在其他语句之前前调用super(props)。否则，this.props 在构造函数中可能会出现未定义的 bug。

通常，在 React 中，构造函数仅用于以下两种情况：

* 通过给 this.state 赋值对象来初始化内部 state。
* 为事件处理函数绑定实例

**特别说明：**这是JavaScript中，如果传递一个函数名给一个变量，然后通过在变量后加括号()来调用这个方法，此时方法内部的this的指向就会丢失。

**实例：**

**constructor(props) {**

**super(props);**

**this.state = { counter: 0 };**

**this.handleClick = this.handleClick.bind(this);**

**}**

### 2.2 render()

**形式：**

**render()**

render() 方法是 class 组件中唯一必须实现的方法。

render() 函数应该为纯函数，这意味着在不修改组件 state 的情况下，每次调用时都返回相同的结果，并且它不会直接与浏览器交互。

### 2.3 componentDidMount()

**形式：**

**componentDidMount()**

componentDidMount() 会在组件挂载后（插入 DOM 树中）立即调用。依赖于 DOM 节点的初始化应该放在这里。如需通过网络请求获取数据，此处是实例化请求的好地方。

这个方法是比较适合添加订阅的地方。如果添加了订阅，请不要忘记在 componentWillUnmount() 里取消订阅

### 2.4 componentDidUpdate()

**形式：**

**componentDidUpdate(prevProps, prevState, snapshot)**

componentDidUpdate() 会在更新后会被立即调用。首次渲染不会执行此方法。

当组件更新后，可以在此处对 DOM 进行操作。如果你对更新前后的 props 进行了比较，也可以选择在此处进行网络请求。（例如，当 props 未发生变化时，则不会执行网络请求）。

**实例：**

**componentDidUpdate(prevProps) {**

**// 典型用法（不要忘记比较 props）：**

**if (this.props.userID !== prevProps.userID) {**

**this.fetchData(this.props.userID);**

**}**

**}**

### 2.5 componentWillUnmount()

**形式：**

**componentWillUnmount()**

componentWillUnmount()会在组件卸载及销毁之前直接调用。在此方法中执行必要的清理操作，例如，清除timer，取消网络请求或清除在componentDidMount()中创建的订阅等。

## 3、组件&Props

组件，从概念上类似于 JavaScript 函数。它接受任意的入参（即 “props”），并返回用于描述页面展示内容的 React 元素。

### 3.1 函数组件与 class 组件

定义组件最简单的方式就是编写 JavaScript 函数：

**实例：**

**function Welcome(props) {**

**return <h1>Hello, {props.name}</h1>;**

**}**

该函数是一个有效的 React 组件，因为它接收唯一带有数据的 “props”（代表属性）对象与并返回一个 React 元素。这类组件被称为“函数组件”，因为它本质上就是 JavaScript 函数。

你同时还可以使用 ES6 的 class 来定义组件：

**实例：**

**class Welcome extends React.Component {**

**render() {**

**return <h1>Hello, {this.props.name}</h1>;**

**}**

**}**

上述两个组件在 React 里是等效的。

### 3.2 渲染组件

当 React 元素为用户自定义组件时，它会将 JSX 所接收的属性（attributes）以及子组件（children）转换为单个对象传递给组件，这个对象被称之为 “props”。

例如，这段代码会在页面上渲染 “Hello, Sara”：

**实例：**

**function Welcome(props) {**

**return <h1>Hello, {props.name}</h1>;**

**}**

**const element = <Welcome name="Sara" />;**

**ReactDOM.render(**

**element,**

**document.getElementById('root');**

**);**

让我们来回顾一下这个例子中发生了什么：

* 调用ReactDOM.render()函数，并传入<Welcome name="Sara"/>作为参数。
* React调用Welcome组件，**并将{name: 'Sara'}作为props传入**。
* Welcome组件将<h1>Hello, Sara</h1>元素作为返回值。
* React DOM将DOM高效地更新为<h1>Hello, Sara</h1>。

## 4、State&生命周期

State与props类似，但是state是私有的，并且完全受控于当前组件。

组件中的state包含了随时可能发生变化的数据。state由用户自定义，它是一个普通 JavaScript对象。

### 4.1 向class组件中添加局部的state

在构造函数中，增加this.state 赋初值：

**class Clock extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {date: new Date()};**

**}**

**render() {**

**return (**

**<div>**

**<h1>Hello, world!</h1>**

**<h2>It is {this.state.date.toLocaleTimeString()}.</h2>**

**</div>**

**);**

**}**

**}**

### 4.2 使用setState()修改state的值

setState()将对组件state的更改排入队列，并通知React需要使用更新后的state重新渲染此组件及其子组件。这是用于更新用户界面以响应事件处理器和处理服务器数据的主要方式

将setState()视为请求而不是立即更新组件的命令。为了更好的感知性能，React会延迟调用它，然后通过一次传递更新多个组件。React并不会保证state的变更会立即生效。

setState()并不总是立即更新组件。它会批量推迟更新。这使得在调用setState()后立即读取this.state成为了隐患。为了消除隐患，请使用componentDidUpdate或者setState 的回调函数（setState(updater, callback)），这两种方式都可以保证在应用更新后触发。

**调用形式一：**

**setState(stateChange[, callback])**

stateChange 会将传入的对象浅层合并到新的 state 中。例如，调整购物车商品数：

**this.setState({quantity: 2})**

这种形式的 setState() 也是异步的，并且在同一周期内会对多个 setState 进行批处理。例如，如果在同一周期内多次设置商品数量增加，则相当于：

**Object.assign(**

**previousState,**

**{quantity: state.quantity + 1},**

**{quantity: state.quantity + 1},**

**...**

**)**

后调用的 setState() 将覆盖同一周期内先调用 setState 的值，因此商品数仅增加一次。如果后续状态取决于当前状态，我们建议使用 updater 函数的形式代替：

**this.setState((state) => {**

**return {quantity: state.quantity + 1};**

**});**

**调用形式二：**

**setState(updater, [callback])**

* **参数一为带有形式参数的 updater 函数**

**(state, props) => stateChange**

state 是对应用变化时组件状态的引用。当然，它不应直接被修改。你应该使用基于 state 和 props 构建的新对象来表示变化。例如，假设我们想根据 props.step 来增加 state：

**this.setState((state, props) => {**

**return {counter: state.counter + props.step};**

**});**

updater 函数中接收的 state 和 props 都保证为最新。updater 的返回值会与 state 进行浅合并。

* **参数二为可选的回调函数**

它将在 setState 完成合并并重新渲染组件后执行。通常，我们建议使用 componentDidUpdate() 来代替此方式。

## 5、一个完整的例子

**import React, { Component } from "react";**

**class App extends Component {**

**//构造函数：参数为属性集props，初始化本组件的局部状态数据state**

**constructor(props) {**

**super(props);**

**this.state = { count: 1};**

**}**

**//组件挂载后的生命周期函数：启动定时器，每秒刷新一次记数**

**componentDidMount() {**

**const { count } = this.state;**

**document.title = "componentDidMount" + count;**

**this.timer = setInterval(() => {**

**this.setState(({ count }) => ({**

**count: count + 1**

**}));**

**}, 1000);**

**}**

**//组件更新后的生命周期函数**

**componentDidUpdate() {**

**const { count } = this.state;**

**document.title = "componentDidMount" + count;**

**}**

**//组件卸载前的生命周期函数**

**componentWillUnmount() {**

**document.title = "componentWillUnmount";**

**clearInterval(this.timer);**

**}**

**render() {**

**const { count } = this.state;**

**return (**

**<div>**

**Count:{count}**

**<button onClick={() => clearInterval(this.timer)}>clear</button>**

**</div>**

**);**

**}**

**}**

如果需要在codepen.io网站上运行，要做如下两件事：

1、在html部分增加：

<div id="app"></div>

2、在JS的最后部分增加：

ReactDOM.render(<App/>, document.getElementById('app'));

# 主题六：React.FC组件

## 1、函数组件

如果组件只包含一个render方法，并且不包含state，那么使用函数组件就会更简单。我们不需要定义一个继承于React.Component的类，我们可以定义一个函数，这个函数接收props作为参数，然后返回需要渲染的元素

**function Square(props) {**

**return (**

**<button className="square" onClick={props.onClick}>**

**{props.value}**

**</button>**

**);**

**}**

React.FC<>的在typescript使用的一个泛型，FC就是FunctionComponent的缩写，是函数组件，在这个泛型里面可以使用useState，个人觉得useState挺好用的，例子如下：

**//React.FC<>为typescript使用的泛型**

**const SampleModel: React.FC<{}> = () =>{**

**const [createModalVisible, handleModalVisible] = useState<boolean>(false);**

**return {**

**<Button onClick={()=>handleModalVisible(true)} >样例</Button>**

**}**

**}**

一个React函数组件代表一个函数，它接受props作为参数并最终返回一个React元素。所以一个组件的渲染过程其实就是一次函数调用。 组件状态的更新导致了组件的重新渲染，触发了函数调用。

## 2、useState函数

在函数组件中， React 通过 useState 来帮我们保存组件的状态。

**形式：**

**const [state, setState] = useState(initialState);**

返回一个 state，以及更新 state 的函数。

在初始渲染期间，返回的状态 (state) 与传入的第一个参数 (initialState) 值相同。

setState 函数用于更新 state。它接收一个新的 state 值并将组件的一次重新渲染加入队列。

**更新形式一：新值更新**

**setState(newState);**

在后续的重新渲染中，useState 返回的第一个值将始终是更新后最新的 state。

**更新形式二：函数式更新**

**setState(prevState => 计算表达式 );**

如果新的 state 需要通过使用先前的 state 计算得出，那么可以将函数传递给 setState。该函数将接收先前的 state，并返回一个更新后的值。

**function Counter({initialCount}) {**

**const [count, setCount] = useState(initialCount);**

**return (**

**<>**

**Count: {count}**

**<button onClick={() => setCount(initialCount)}>Reset</button>**

**<button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>**

**<button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>**

**</>**

**);**

**}**

## 3、useEffect函数

**形式：**

**useEffect(didUpdate);**

在函数组件主体内（这里指在 React 渲染阶段）改变DOM、添加订阅、设置定时器、记录日志以及执行其他包含副作用的操作都是不被允许的，因为这可能会产生莫名其妙的bug并破坏UI的一致性。

使用useEffect完成副作用操作。**赋值给useEffect的函数会在组件渲染到屏幕之后执行**。你可以把 effect 看作从 React 的纯函数式世界通往命令式世界的逃生通道。

* **清除 effect**

通常，组件卸载时需要清除effect创建的诸如订阅或计时器ID等资源。要实现这一点，**useEffect 函数需返回一个清除函数**。以下就是一个创建订阅的例子：

**useEffect(() => {**

**const subscription = props.source.subscribe();**

**return () => {**

**// 清除订阅**

**subscription.unsubscribe();**

**};**

**});**

为防止内存泄漏，清除函数会在组件卸载前执行。另外，如果组件多次渲染（通常如此），则在执行下一个effect之前，上一个effect就已被清除。

* **effect 的执行时机**

与componentDidMount、componentDidUpdate不同的是，**传给useEffect的函数会在浏览器完成布局与绘制之后，在一个延迟事件中被调用**。这使得它适用于许多常见的副作用场景，比如设置订阅和事件处理等情况，因为绝大多数操作不应阻塞浏览器对屏幕的更新。

虽然useEffect会在浏览器绘制后延迟执行，但会保证在任何新的渲染前执行。在开始新的更新前，React总会先清除上一轮渲染的effect。

* **effect 的条件执行**

默认情况下，effect 会在每轮组件渲染完成后执行。这样的话，一旦 effect 的依赖发生变化，它就会被重新创建。

然而，在某些场景下这么做可能会矫枉过正。比如，我们不需要在每次组件更新时都创建新的订阅，而是仅需要在属性props或状态state改变时重新创建。

要实现这一点，可以给 **useEffect 传递第二个参数，它是 effect 所依赖的值数组**。更新后的示例如下：

**useEffect(**

**() => {**

**const subscription = props.source.subscribe();**

**return () => {**

**subscription.unsubscribe();**

**};**

**},**

**[props.source],**

**);**

此时，只有当 props.source 改变后才会重新创建订阅。

**特别说明：**如果想执行只运行一次的 effect（仅在组件挂载和卸载时执行），可以传递一个空数组（[]）作为第二个参数。这就告诉 React 你的 effect 不依赖于 props 或 state 中的任何值，所以它永远都不需要重复执行。这并不属于特殊情况 —— 它依然遵循输入数组的工作方式。

**useEffect(() => {**

**timer = setInterval(() => {**

**setCount(prevCount => prevCount + 1);**

**}, 1000);**

**return () => {**

**clearInterval(timer);**

**};**

**}, []);**

## 4、connect函数

**connect()是react-redux中的核心方法之一，它将react组件预redux中的Store真正连接在一起。**

**connect(mapStateToProps, mapDispatchToProps)(MyComponent)**

* **mapStateToProps**

字面含义是**把state映射到props中去**，意思就是把Redux中的数据映射到React中的props中去。也就是说你React想把Redux中的哪些数据拿过来用。

**特别注意：**任何时候只要 Redux store发生改变，mapStateToProp 函数就会被调用,该回调函数必须返回一个纯对象，该对象会与相应展示组件的 props 合并

比如这里二连这个组件想要渲染自己枪支的数量。就可以直接在二连这个组件中把Redux中的gunOfErlian拿过来用。

**const mapStateToProps = (state) => {**

**return {**

**gun: state.gunOfErlian**

**}**

**}**

然后渲染的时候就可以直接使用this.props.gun

**class Erlian extends Component {**

**constructor(props){**

**super(props);**

**}**

**render(){**

**return(**

**<div>this.props.gun</div>**

**)**

**}**

**}**

**export default connect()(Erlian);**

那么这样就可以实现渲染，就是把Redux中的state变成React中的props。

# react框架学习笔记

刚刚学习react框架，记下来一些终于弄清楚的点，希望能给其他刚刚接触react全家桶，也被弄晕了的同学一些概念上的启发。

## 1、react全家桶解决了什么问题

1、我感觉，react最主要的贡献是解决了变量变化后dom需要自动刷新的问题

比如说原生js想实现计数器，点页面上一个按钮，然后一个数值不断增加。你需要绑定一个监听器，然后通过回调函数修改本地变量的值，最后操纵dom把这个值显示出来。

现在呢，还需要干前两部，第三部由于生成那个节点的时候，那个计数器变量就是被注入进入的，所以react帮你把它变了。所以用了react，你就不用写一堆getElementById了

2、有了react管理渲染问题，还得有人帮你维护一堆本地变量啊。redux就管这个事。redux的思路是，有一个全局对象store，存着所有(被验证的)信息和状态，比如用户id啦，还有一页显示多少条标题什么的。然后再把信息或者状态与对应的ui组件连起来，把store里的state注入到组件property里面，这样你只要改变store里的信息，组件就可以自动刷新了。

3、路由用react-router。其实原来用路由，主要是原来网站上不同的页面存储在不同的文件夹里，不同的路由对应不同的存储地址。现在用react了，打包后就一个js文件，一开始客户端访问服务器时服务器就把全部页面都传给客户端了，根本就不需要多次访问服务器刷新页面。现在的路由模块可能是为了写程序的时候，方便把程序分块放在不同文件，项目比较好组织

4、ajax这块主要就是用的原生javascript promise那一套，没什么变化。redux提供了一些中间件对promise提供了一些支持，不过没有本质区别

5、ui库。react最棒的就是，它把dom中一个节点看做是由几个变量决定的有限状态机，你把这几个变量提取出来，它就相当于一个函数，所以react比较容易开发组件库。

我自己用的是ant-design，阿里出的，常用的组件基本都有了。

## 2、我看教程的顺序

1、先复习一下mozilla的官方js教程(有中文版)。es6引入了class，extend。还有es5引入的什么object.assign(), ...array, [a,b] =一堆的。class建议先去看看文档对应章节，其他的可以不用提前看。

2、学react最好的还是官方文档，非常清楚明白(有中文版)。先看基础，然后一定要看看hook，最后再看看高级里面的高阶组件，ref什么的，一开始没必要全看完

3、然后学react-router，文档很短，例子也挺清楚的，很快就刷过去了

简书有一篇介绍也不错:

https://www.jianshu.com/p/a3db4772bbb8

​

4、然后看redux，同样最好的就是官方文档，基本和进阶都得看，如果你想用的ui组件库是hook写的话，还得看看redux-react 的hook部分

5、ui组件库。我用的是ant-design 阿里的文档例子是挺多的，但是可惜缺乏一个总览式的教程。前端经验丰富的开发人员应该能知道阿里这堆组件是干啥的，像我这种小白只能一个一个找哪个组件是我想要的效果。阿里语雀上配了一套教程，可是起手难度也不低。

总之，我对阿里组件库的感觉就是，东西确实是好东西，但是缺乏一个清晰的学习路线图，得自己看阿里提供的样例和代码琢磨。风格非常像中国教科书，又大又全，全是干货，不太好懂。做不到像外国官方文档那种从入门到精通的感觉。

还有，阿里的好多组件都用了hook，不学hook不太容易看懂他们的代码

## 3、用全家桶以后调用各部分的顺序

比如我们想填个表单，把表单内容发给服务器，服务器返回信息，最后跳转页面，那一个可行的顺序就是

hook组件-onchange回调函数-fetch-服务器－then-dispatch(action)发送给redux-reduce更新store的state-相关组件利用useState这个hook监听store内数据变化，如果有变化，更新相关组件－如果需要页面跳转，渲染router里的<redirect>组件，更新路径－router里的<switch>组件根据路径渲染对应的<route>组件，<route>组件里面包装的对应ui组件被渲染，完成页面跳转

# 简单易懂的 React useState() Hook 指南

状态是隐藏在组件中的信息，组件可以在父组件不知道的情况下修改其状态。我更偏爱函数组件，因为它们足够简单，要使函数组件具有状态管理，可以useState() Hook。

本文会逐步讲解如何使用useState() Hook。此外，还会介绍一些常见useState()坑。

## 1、使用 `useState()` 进行状态管理

无状态的函数组件没有状态，如下所示(部分代码)：



可以找 codesandbox 尝试一下。

运行效果：



这时，要如何添加一个按钮来打开/关闭灯泡呢？为此，咱们需要具有状态的函数组件，也就是有状态函数组件。

useState()是实现灯泡开关状态的 Hoook，将状态添加到函数组件需要4个步骤:启用状态、初始化、读取和更新。

### 1.1 启用状态

要将<Bulbs> 转换为有状态组件，需要告诉 React：从'react'包中导入useState钩子，然后在组件函数的顶部调用useState()。

大致如下所示：



在Bulbs函数的第一行调用useState()（暂时不要考Hook的参数和返回值）。重要的是，在组件内部调用 Hook 会使该函数成为有状态的函数组件。

启用状态后，下一步是初始化它。

### 1.2初始化状态

始时，灯泡关闭，对应到状态应使用false初始化 Hook：



useState(false)用false初始化状态。

启用和初始化状态之后，如何读取它?来看看useState(false)返回什么。

### 1.3 读取状态

当 hook useState(initialState)被调用时，它返回一个数组，该数组的第一项是状态值

const stateArray = useState(false);

stateArray[0]; // => 状态值

咱们读取组件的状态

function Bulbs() {

const stateArray = useState(false);

return <div className={stateArray[0] ? 'bulb-on' : 'bulb-off'} />;

}

<Bulbs>组件状态初始化为false，可以打开 codesandbox 看看效果。

useState(false)返回一个数组，第一项包含状态值，该值当前为false(因为状态已用false初始化)。

咱们可以使用数组解构来将状态值提取到变量on上：

import React, { useState } from 'react';

function Bulbs() {

const [on] = useState(false);

return <div className={on ? 'bulb-on' : 'bulb-off'} />;

}

on状态变量保存状态值。

状态已经启用并初始化，现在可以读取它了。但是如何更新呢?再来看看useState(initialState)返回什么。

### 1.4 更新状态

#### 1.4.1用值更新状态

咱们已经知道，useState(initialState)返回一个数组，其中第一项是状态值，第二项是一个更新状态的函数。

const [state, setState] = useState(initialState);

// 将状态更改为 'newState' 并触发重新渲染

setState(newState);

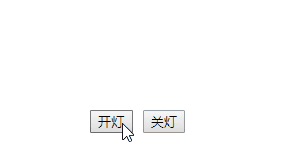
// 重新渲染`state`后的值为`newState`

要更新组件的状态，请使用新状态调用更新器函数setState(newState)。组件重新渲染后，状态接收新值newState。

当点击开灯按钮时将灯泡开关状态更新为true，点击关灯时更新为 false。



打开 codesandbox 自行尝试一下。



单击开灯按钮时，lightOn()函数将on更新为true: setOn(true)。单击关灯时也会发生相同的情况，只是状态更新为false。

状态一旦改变，React 就会重新渲染组件，on变量获取新的状态值。

状态更新作为对提供一些新信息的事件的响应。这些事件包括按钮单击、HTTP 请求完成等，确保在事件回调或其他 Hook 回调中调用状态更新函数。

#### 1.4.2使用回调更新状态

当使用前一个状态计算新状态时，可以使用回调更新该状态:

const [state, setState] = useState(initialState);

...

setState(prevState => nextState);

...

下面是一些事例：



接着，通过这种方式重新实现上面电灯的示例：



setOn(on => !on)使用函数更新状态。

### 1.5 小结一波

1. 调用useState() Hook 来启用函数组件中的状态。
2. useState(initialValue)的第一个参数initialValue是状态的初始值。
3. [state, setState] = useState(initialValue)返回一个包含2个元素的数组:状态值和状态更新函数。
4. 使用新值调用状态更新器函数setState(newState)更新状态。或者，可以使用一个回调setState(prev => next)来调用状态更新器，该回调将返回基于先前状态的新状态。
5. 调用状态更新器后，React 确保重新渲染组件，以使新状态变为当前状态。

## 2. 多种状态

通过多次调用useState()，一个函数组件可以拥有多个状态。

function MyComponent() {

const [state1, setState1] = useState(initial1);

const [state2, setState2] = useState(initial2);

const [state3, setState3] = useState(initial3);

// ...

}

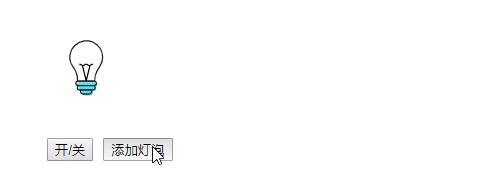
需要注意的，要确保对useState()的多次调用在渲染之间始终保持相同的顺序(后面会讲)。

我们添加一个按钮添加灯泡，并添加一个新状态来保存灯泡数量，单击该按钮时，将添加一个新灯泡。

新的状态count 包含灯泡的数量，初始值为1：



打开演示，然后单击添加灯泡按钮：灯泡数量增加，单击开/关按钮可打开/关闭灯泡。



[on, setOn] = useState(false) 管理开/关状态

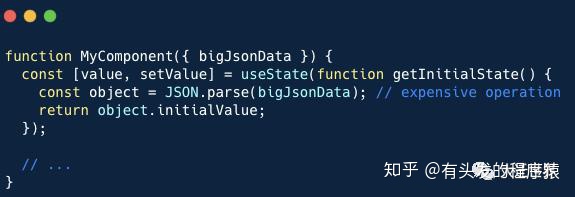
[count, setCount] = useState(1)管理灯泡数量。

多个状态可以在一个组件中正确工作。

## 3.状态的延迟初始化

每当 React 重新渲染组件时，都会执行useState(initialState)。如果初始状态是原始值（数字，布尔值等），则不会有性能问题。

当初始状态需要昂贵的性能方面的操作时，可以通过为useState(computeInitialState)提供一个函数来使用状态的延迟初始化，如下所示：



getInitialState()仅在初始渲染时执行一次，以获得初始状态。在以后的组件渲染中，不会再调用getInitialState()，从而跳过昂贵的操作。

## 4. useState() 中的坑

现在咱们基本已经初步掌握了如何使用useState()，尽管如此，咱们必须注意在使用useState()时可能遇到的常见问题。

### 4.1 在哪里调用 `useState()`

在使用useState() Hook 时，必须遵循 Hook 的规则

1. 仅顶层调用 Hook ：不能在循环，条件，嵌套函数等中调用useState()。在多个useState()调用中，渲染之间的调用顺序必须相同。
2. 仅从React 函数调用 Hook:必须仅在函数组件或自定义钩子内部调用useState()。

来看看useState()的正确用法和错误用法的例子。

**有效调用useState()**

useState()在函数组件的顶层被正确调用

function Bulbs() {

// Good

const [on, setOn] = useState(false);

// ...

}

以相同的顺序正确地调用多个useState()调用:

function Bulbs() {

// Good

const [on, setOn] = useState(false);

const [count, setCount] = useState(1);

// ...

useState()在自定义钩子的顶层被正确调用

function toggleHook(initial) {

// Good

const [on, setOn] = useState(initial);

return [on, () => setOn(!on)];

}

function Bulbs() {

const [on, toggle] = toggleHook(false);

// ...

}

**useState() 的无效调用**

在条件中调用useState()是不正确的：

function Switch({ isSwitchEnabled }) {

if (isSwitchEnabled) {

// Bad

const [on, setOn] = useState(false);

}

// ...

}

在嵌套函数中调用useState()也是不对的



### 4.2 过时状态

闭包是一个从外部作用域捕获变量的函数。

闭包（例如事件处理程序，回调）可能会从函数组件作用域中捕获状态变量。由于状态变量在渲染之间变化，因此闭包应捕获具有最新状态值的变量。否则，如果闭包捕获了过时的状态值，则可能会遇到过时的状态问题。

来看看一个过时的状态是如何表现出来的。组件<DelayedCount>延迟3秒计数按钮点击的次数。



打开演示，快速多次点击按钮。count 变量不能正确记录实际点击次数，有些点击被吃掉。

delay() 是一个过时的闭包，它从初始渲染（使用0初始化时）中捕获了过时的count变量。

为了解决这个问题，使用函数方法来更新count状态：



现在etCount(count => count + 1)在delay()中正确更新计数状态。React 确保将最新状态值作为参数提供给更新状态函数，过时闭包的问题解决了。

打开演示，快速单击按钮。延迟过去后，count 能正确表示点击次数。

### 4.3 复杂状态管理

useState()用于管理简单状态。对于复杂的状态管理，可以使用useReducer()hook。它为需要多个状态操作的状态提供了更好的支持。

假设需要编写一个最喜欢的电影列表。用户可以添加电影，也可以删除已有的电影，实现方式大致如下：



尝试演示：添加和删除自己喜欢的电影。



状态列表需要几个操作:添加和删除电影，状态管理细节使组件混乱。

更好的解决方案是将复杂的状态管理提取到reducer中：



reducer管理电影的状态，有两种操作类型：

1. "add"将新电影插入列表
2. "remove"从列表中按索引删除电影

尝试演示并注意组件功能没有改变。但是这个版本的<FavoriteMovies>更容易理解，因为状态管理已经被提取到reducer中。

还有一个好处:可以将reducer 提取到一个单独的模块中，并在其他组件中重用它。另外，即使没有组件，也可以对reducer 进行单元测试。

这就是关注点分离的威力:组件渲染UI并响应事件，而reducer 执行状态操作。

### 4.4 状态 vs 引用

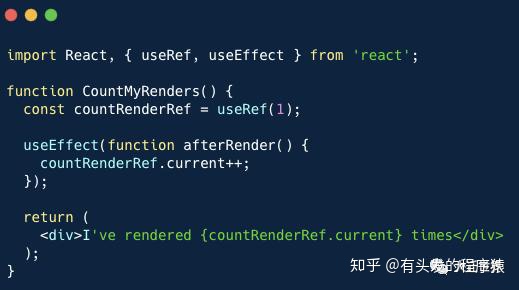
考虑这样一个场景:咱们想要计算组件渲染的次数。

一种简单的实现方法是初始化countRender状态，并在每次渲染时更新它(使用useEffect() hook)



useEffect()在每次渲染后调用afterRender()回调。但是一旦countRender状态更新，组件就会重新渲染。这将触发另一个状态更新和另一个重新渲染，依此类推。

可变引用useRef()保存可变数据，这些数据在更改时不会触发重新渲染，使用可变的引用改造一下<CountMyRenders> ：



打开演示并单击几次按钮来触发重新渲染。



每次渲染组件时，countRenderRef可变引用的值都会使countRenderRef.current ++递增。重要的是，更改不会触发组件重新渲染。

## 5. 总结

要使函数组件有状态，请在组件的函数体中调用useState()。

useState(initialState)的第一个参数是初始状态。返回的数组有两项:当前状态和状态更新函数。

const [state, setState] = useState(initialState);

使用 setState(newState)来更新状态值。另外，如果需要根据先前的状态更新状态，可以使用回调函数setState(prevState => newState)。

在单个组件中可以有多个状态:调用多次useState()。

当初始状态开销很大时，延迟初始化很方便。使用计算初始状态的回调调用useState(computeInitialState)，并且此回调仅在初始渲染时执行一次。

必须确保使用useState()遵循 Hook 规则。

当闭包捕获过时的状态变量时，就会出现过时状态的问题。可以通过使用一个回调来更新状态来解决这个问题，这个回调会根据先前的状态来计算新的状态。

最后，您将使用useState()来管理一个简单的状态。为了处理更复杂的状态，一个更好的的选择是使用useReducer() hook。

# 十个案例学会 React Hooks

## 1、前言

在 React 的世界中，有容器组件和 UI 组件之分，在 React Hooks 出现之前，UI 组件我们可以使用函数，无状态组件来展示 UI。而对于容器组件，函数组件就显得无能为力，我们依赖于类组件来获取数据，处理数据，并向下传递参数给 UI 组件进行渲染。在我看来，使用 React Hooks 相比于从前的类组件有以下几点好处：

1. 代码可读性更强，原本同一块功能的代码逻辑被拆分在了不同的生命周期函数中，容易使开发者不利于维护和迭代，通过 React Hooks 可以将功能代码聚合，方便阅读维护
2. 组件树层级变浅，在原本的代码中，我们经常使用 HOC/render props 等方式来复用组件的状态，增强功能等，无疑增加了组件树层数及渲染，而在 React Hooks 中，这些功能都可以通过强大的自定义的 Hooks 来实现

React 在 v16.8 的版本中推出了 React Hooks 新特性，虽然社区还没有最佳实践如何基于 React Hooks 来打造复杂应用(至少我还没有)，凭借着阅读社区中大量的关于这方面的文章，下面我将通过十个案例来帮助你认识理解并可以熟练运用 React Hooks 大部分特性。

## 2、useState 保存组件状态

在类组件中，我们使用 this.state 来保存组件状态，并对其修改触发组件重新渲染。比如下面这个简单的计数器组件，很好诠释了类组件如何运行：[在线 Demo](https://codesandbox.io/s/ojk1oloq9z)

import React from "react";

class App extends React.Component {

constructor(props) {

super(props);

**this.state** = {

count: 0,

name: "alife"

};

}

render() {

const { count } = this.state;

return (

<div>

Count: {count}

<button onClick={() => **this.setState**({ count: count + 1 })}>+</button>

<button onClick={() => **this.setState**({ count: count - 1 })}>-</button>

</div>

);

}

}

一个简单的计数器组件就完成了，而在函数组件中，由于没有 this 这个黑魔法，React 通过 useState 来帮我们保存组件的状态。[在线 Demo](https://codesandbox.io/s/mj3j955m6x)

import React, { useState } from "react";

function App() {

const [obj, setObject] = useState({

count: 0,

name: "alife"

});

return (

<div className="App">

Count: {obj.count}

<button onClick={() => setObject({ ...obj, count: obj.count + 1 })}>+</button>

<button onClick={() => setObject({ ...obj, count: obj.count - 1 })}>-</button>

</div>

);

}

通过传入 useState 参数后返回一个带有默认状态和改变状态函数的数组。通过传入新状态给函数来改变原本的状态值。**值得注意的是 useState 不帮助你处理状态，相较于 setState 非覆盖式更新状态，useState 覆盖式更新状态，需要开发者自己处理逻辑。(代码如上)**

似乎有个 useState 后，函数组件也可以拥有自己的状态了，但仅仅是这样完全不够。

## 3、useEffect 处理副作用

函数组件能保存状态，但是对于异步请求，副作用的操作还是无能为力，所以 React 提供了 useEffect 来帮助开发者处理函数组件的副作用，在介绍新 API 之前，我们先来看看类组件是怎么做的：[在线 Demo](https://codesandbox.io/s/y29rwpk3z1)

import React, { Component } from "react";

class App extends Component {

state = {

count: 1

};

componentDidMount() {

const { count } = this.state;

document.title = "componentDidMount" + count;

this.timer = setInterval(() => {

this.setState(({ count }) => ({

count: count + 1

}));

}, 1000);

}

componentDidUpdate() {

const { count } = this.state;

document.title = "componentDidMount" + count;

}

componentWillUnmount() {

document.title = "componentWillUnmount";

clearInterval(this.timer);

}

render() {

const { count } = this.state;

return (

<div>

Count:{count}

<button onClick={() => clearInterval(this.timer)}>clear</button>

</div>

);

}

}

在例子中，组件每隔一秒更新组件状态，并且每次触发更新都会触发 document.title 的更新(副作用)，而在组件卸载时修改 document.title（类似于清除）

从例子中可以看到，一些重复的功能开发者需要在 componentDidMount 和 componentDidUpdate 重复编写，而如果使用 useEffect 则完全不一样。[在线 Demo](https://codesandbox.io/s/0x1l4kk6lv)

import React, { useState, useEffect } from "react";

let timer = null;

function App() {

const [count, setCount] = useState(0);

useEffect(() => {

document.title = "componentDidMount" + count;

},[count]);

useEffect(() => {

timer = setInterval(() => {

setCount(prevCount => prevCount + 1);

}, 1000);

return () => {

document.title = "componentWillUnmount";

clearInterval(timer);

};

}, []);

return (

<div>

Count: {count}

<button onClick={() => clearInterval(timer)}>clear</button>

</div>

);

}

我们使用 useEffect 重写了上面的例子，useEffect 第一个参数传递函数，可以用来做一些副作用比如异步请求，修改外部参数等行为，而第二个参数是个数组，如果数组中的值才会触发 useEffect 第一个参数中的函数。返回值(如果有)则在组件销毁或者调用函数前调用。

1. 比如第一个 useEffect 中，理解起来就是一旦 count 值发生改变，则修改 documen.title 值
2. 而第二个 useEffect 中数组没有传值，代表不监听任何参数变化，即只有在组件初始化或销毁的时候才会触发，用来代替 componentDidMount 和 componentWillUnmount

基于这个强大 Hooks，我们可以模拟封装出其他生命周期函数，比如 componentDidUpdate 代码十分简单

function useUpdate(fn) {

// useRef 创建一个引用

const mounting = useRef(true);

useEffect(() => {

if (mounting.current) {

mounting.current = false;

} else {

fn();

}

});

}

现在我们有了 useState 管理状态，useEffect 处理副作用，异步逻辑，学会这两招足以应对大部分类组件的使用场景。

## 4、useContext 减少组件层级

上面介绍了 useState、useEffect 这两个最基本的 API，接下来介绍的 useContext 是 React 帮你封装好的，用来处理多层级传递数据的方式，在以前组件树种，跨层级祖先组件想要给孙子组件传递数据的时候，除了一层层 props 往下透传之外，我们还可以使用 React Context API 来帮我们做这件事，举个简单的例子：[在线 Demo](https://codesandbox.io/s/94p24or414)

const { Provider, Consumer } = React.createContext(null);

function Bar() {

return <Consumer>{color => <div>{color}</div>}</Consumer>;

}

function Foo() {

return <Bar />;

}

function App() {

return (

<Provider value={"grey"}>

<Foo />

</Provider>

);

}

通过 React createContext 的语法，在 APP 组件中可以跨过 Foo 组件给 Bar 传递数据。而在 React Hooks 中，我们可以使用 useContext 进行改造。[在线 Demo](https://github.com/happylindz/blog/issues/19)

const colorContext = React.createContext("gray");

function Bar() {

const color = useContext(colorContext);

return <div>{color}</div>;

}

function Foo() {

return <Bar />;

}

function App() {

return (

<colorContext.Provider value={"red"}>

<Foo />

</colorContext.Provider>

);

}

传递给 useContext 的是 context 而不是 consumer，返回值即是想要透传的数据了。用法很简单，使用 useContext 可以解决 Consumer 多状态嵌套的问题。[参考例子](https://daveceddia.com/usecontext-hook/)

function HeaderBar() {

return (

<CurrentUser.Consumer>

{user =>

<Notifications.Consumer>

{notifications =>

<header>

Welcome back, {user.name}!

You have {notifications.length} notifications.

</header>

}

}

</CurrentUser.Consumer>

);

}

而使用 useContext 则变得十分简洁，可读性更强且不会增加组件树深度。

function HeaderBar() {

const user = useContext(CurrentUser);

const notifications = useContext(Notifications);

return (

<header>

Welcome back, {user.name}!

You have {notifications.length} notifications.

</header>

);

}

## 5、useReducer

useReducer 这个 Hooks 在使用上几乎跟 Redux/React-Redux 一模一样，唯一缺少的就是无法使用 redux 提供的中间件。我们将上述的计时器组件改写为 useReducer，[在线 Demo](https://codesandbox.io/s/727o0kr4yx)

import React, { useReducer } from "react";

const initialState = {

count: 0

};

function reducer(state, action) {

switch (action.type) {

case "increment":

return { count: state.count + action.payload };

case "decrement":

return { count: state.count - action.payload };

default:

throw new Error();

}

}

function App() {

const [state, dispatch] = useReducer(reducer, initialState);

return (

<>

Count: {state.count}

<button onClick={() => dispatch({ type: "increment", payload: 5 })}>

+

</button>

<button onClick={() => dispatch({ type: "decrement", payload: 5 })}>

-

</button>

</>

);

}

用法跟 Redux 基本上是一致的，用法也很简单，算是提供一个 mini 的 Redux 版本。

## 6、useCallback 记忆函数

在类组件中，我们经常犯下面这样的错误：

class App {

render() {

return <div>

<SomeComponent style={{ fontSize: 14 }} doSomething={ () => { console.log('do something'); }} />

</div>;

}

}

这样写有什么坏处呢？一旦 App 组件的 props 或者状态改变了就会触发重渲染，即使跟 SomeComponent 组件不相关，由于每次 render 都会产生新的 style 和 doSomething，所以会导致 SomeComponent 重新渲染，倘若 SomeComponent 是一个大型的组件树，这样的 Virtual Dom 的比较显然是很浪费的，解决的办法也很简单，将参数抽离成变量。

const fontSizeStyle = { fontSize: 14 };

class App {

doSomething = () => {

console.log('do something');

}

render() {

return <div>

<SomeComponent style={fontSizeStyle} doSomething={ this.doSomething } />

</div>;

}

}

在类组件中，我们还可以通过 this 这个对象来存储函数，而在函数组件中没办法进行挂载了。所以函数组件在每次渲染的时候如果有传递函数的话都会重渲染子组件。

function App() {

const handleClick = () => {

console.log('Click happened');

}

return <SomeComponent onClick={handleClick}>Click Me</SomeComponent>;

}

而有了 useCallback 就不一样了，你可以通过 useCallback 获得一个记忆后的函数。

function App() {

const memoizedHandleClick = useCallback(() => {

console.log('Click happened')

}, []); // 空数组代表无论什么情况下该函数都不会发生改变

return <SomeComponent onClick={memoizedHandleClick}>Click Me</SomeComponent>;

}

老规矩，第二个参数传入一个数组，数组中的每一项一旦值或者引用发生改变，useCallback 就会重新返回一个新的记忆函数提供给后面进行渲染。

这样只要子组件继承了 PureComponent 或者使用 React.memo 就可以有效避免不必要的 VDOM 渲染。

## 7、useMemo 记忆组件

useCallback 的功能完全可以由 useMemo 所取代，如果你想通过使用 useMemo 返回一个记忆函数也是完全可以的。

useCallback(fn, inputs) is equivalent to useMemo(() => fn, inputs).

所以前面使用 useCallback 的例子可以使用 useMemo 进行改写：

function App() {

const memoizedHandleClick = useMemo(() => () => {

console.log('Click happened')

}, []); // 空数组代表无论什么情况下该函数都不会发生改变

return <SomeComponent onClick={memoizedHandleClick}>Click Me</SomeComponent>;

}

唯一的区别是：\*\*useCallback 不会执行第一个参数函数，而是将它返回给你，而 useMemo 会执行第一个函数并且将函数执行结果返回给你。\*\*所以在前面的例子中，可以返回 handleClick 来达到存储函数的目的。

所以 useCallback 常用记忆事件函数，生成记忆后的事件函数并传递给子组件使用。而 useMemo 更适合经过函数计算得到一个确定的值，比如记忆组件。

function Parent({ a, b }) {

// Only re-rendered if `a` changes:

const child1 = useMemo(() => <Child1 a={a} />, [a]);

// Only re-rendered if `b` changes:

const child2 = useMemo(() => <Child2 b={b} />, [b]);

return (

<>

{child1}

{child2}

</>

)

}

当 a/b 改变时，child1/child2 才会重新渲染。从例子可以看出来，只有在第二个参数数组的值发生变化时，才会触发子组件的更新。

## 8、useRef 保存引用值

useRef 跟 createRef 类似，都可以用来生成对 DOM 对象的引用，看个简单的例子：[在线 Demo](https://codesandbox.io/s/v6948pww5y?from-embed)

import React, { useState, useRef } from "react";

function App() {

let [name, setName] = useState("Nate");

let nameRef = useRef();

const submitButton = () => {

setName(nameRef.current.value);

};

return (

<div className="App">

<p>{name}</p>

<div>

<input ref={nameRef} type="text" />

<button type="button" onClick={submitButton}>

Submit

</button>

</div>

</div>

);

}

useRef 返回的值传递给组件或者 DOM 的 ref 属性，就可以通过 ref.current 值访问组件或真实的 DOM 节点，从而可以对 DOM 进行一些操作，比如监听事件等等。

当然 useRef 远比你想象中的功能更加强大，useRef 的功能有点像类属性，或者说您想要在组件中记录一些值，并且这些值在稍后可以更改。

利用 useRef 就可以绕过 Capture Value 的特性。可以认为 ref 在所有 Render 过程中保持着唯一引用，因此所有对 ref 的赋值或取值，拿到的都只有一个最终状态，而不会在每个 Render 间存在隔离。参考例子：[精读《Function VS Class 组件》](https://juejin.im/post/5c8eec1bf265da67cb619e79)

React Hooks 中存在 Capture Value 的特性：[在线 Demo](https://codesandbox.io/s/93m5mz9w24)

function MessageThread() {

const [message, setMessage] = useState("");

const showMessage = () => {

alert("You said: " + message);

};

const handleSendClick = () => {

setTimeout(showMessage, 3000);

};

const handleMessageChange = e => {

setMessage(e.target.value);

};

return (

<>

<input value={message} onChange={handleMessageChange} />

<button onClick={handleSendClick}>Send</button>

</>

);

}

在点击 Send 按钮后，再次修改输入框的值，3 秒后的输出依然是点击前输入框的值。这就是所谓的 capture value 的特性。而在类组件中 3 秒后输出的就是修改后的值，因为这时候 message 是挂载在 this 变量上，它保留的是一个引用值，对 this 属性的访问都会获取到最新的值。讲到这里你应该就明白了，useRef 创建一个引用，就可以有效规避 React Hooks 中 Capture Value 特性。

function MessageThread() {

const latestMessage = useRef("");

const showMessage = () => {

alert("You said: " + latestMessage.current);

};

const handleSendClick = () => {

setTimeout(showMessage, 3000);

};

const handleMessageChange = e => {

latestMessage.current = e.target.value;

};

}

只要将赋值与取值的对象变成 useRef，而不是 useState，就可以躲过 capture value 特性，在 3 秒后得到最新的值。

## 9、useImperativeHandle 透传 Ref

通过 useImperativeHandle 用于让父组件获取子组件内的索引 [在线 Demo](https://codesandbox.io/s/m7wxjz5j3j)

import React, { useRef, useEffect, useImperativeHandle, forwardRef } from "react";

function ChildInputComponent(props, ref) {

const inputRef = useRef(null);

useImperativeHandle(ref, () => inputRef.current);

return <input type="text" name="child input" ref={inputRef} />;

}

const ChildInput = forwardRef(ChildInputComponent);

function App() {

const inputRef = useRef(null);

useEffect(() => {

inputRef.current.focus();

}, []);

return (

<div>

<ChildInput ref={inputRef} />

</div>

);

}

通过这种方式，App 组件可以获得子组件的 input 的 DOM 节点。

## 10、useLayoutEffect 同步执行副作用

大部分情况下，使用 useEffect 就可以帮我们处理组件的副作用，但是如果想要同步调用一些副作用，比如对 DOM 的操作，就需要使用 useLayoutEffect，useLayoutEffect 中的副作用会在 DOM 更新之后同步执行。[在线 Demo](https://codesandbox.io/s/74myo7w94q)

function App() {

const [width, setWidth] = useState(0);

useLayoutEffect(() => {

const title = document.querySelector("#title");

const titleWidth = title.getBoundingClientRect().width;

console.log("useLayoutEffect");

if (width !== titleWidth) {

setWidth(titleWidth);

}

});

useEffect(() => {

console.log("useEffect");

});

return (

<div>

<h1 id="title">hello</h1>

<h2>{width}</h2>

</div>

);

}

在上面的例子中，useLayoutEffect 会在 render，DOM 更新之后同步触发函数，会优于 useEffect 异步触发函数。

## 11、React Hooks 不足

尽管我们通过上面的例子看到 React Hooks 的强大之处，似乎类组件完全都可以使用 React Hooks 重写。但是当下 v16.8 的版本中，还无法实现 getSnapshotBeforeUpdate 和 componentDidCatch 这两个在类组件中的生命周期函数。官方也计划在不久的将来在 React Hooks 进行实现。