05 | 数据是如何在 React 组件之间流动的？（下）

2020/10/26 修言

在上个课时，我们掌握了 React 数据流方案中风格相对“朴素”的 Props 单向数据流方案，以及通用性较强的“发布-订阅”模式。在本课时，我们将一起认识 React 天然具备的全局通信方式“Context API”，并对 Redux 的设计思想和编码形态进行初步的探索。

使用 Context API 维护全局状态

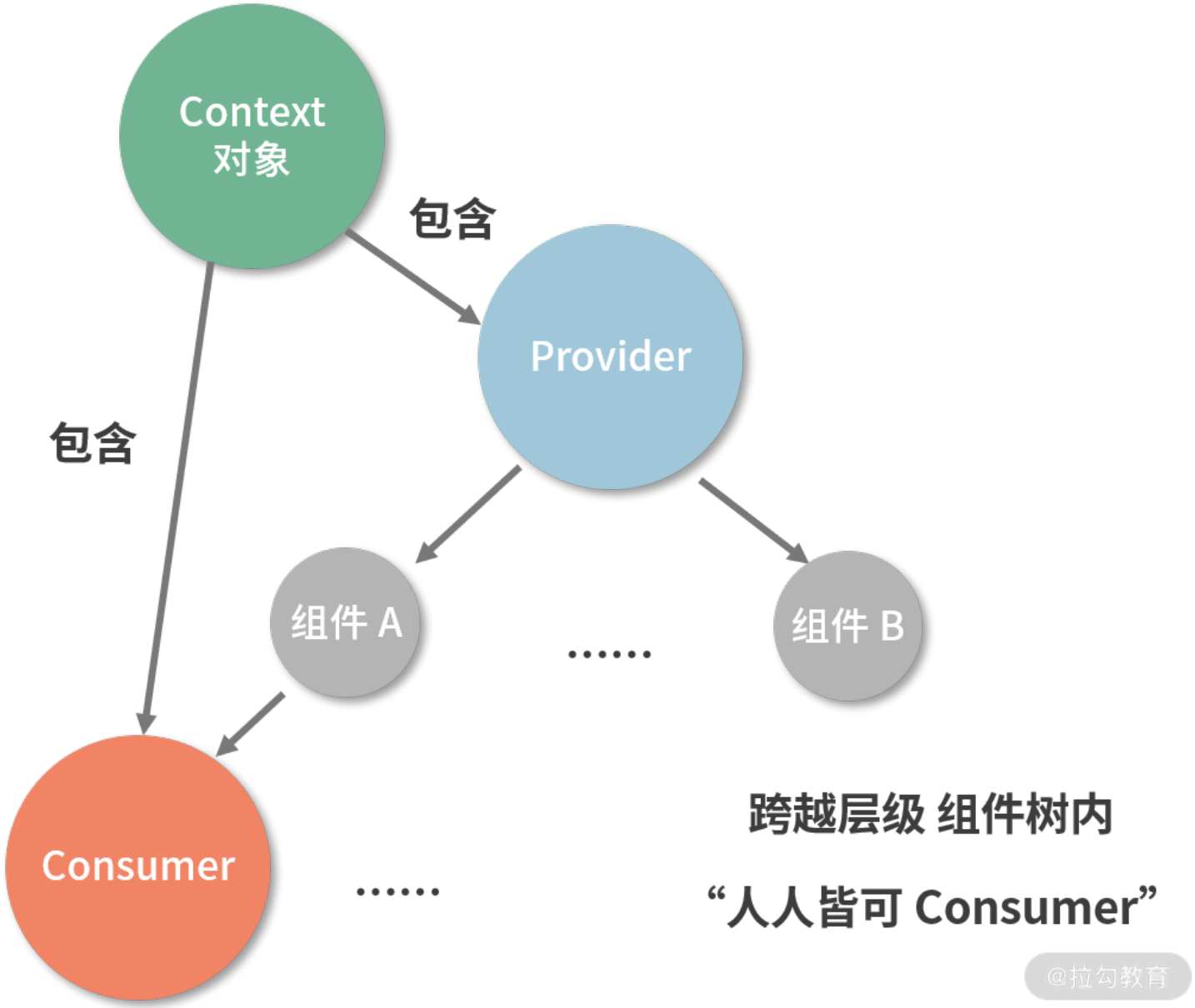
Context API 是 React 官方提供的一种组件树全局通信的方式。

在 React 16.3 之前，Context API 由于存在种种局限性，并不被 React 官方提倡使用，开发者更多的是把它作为一个概念来探讨。而从 v 16.3.0 开始，React 对 Context API 进行了改进，新的 Context API 具备更强的可用性。这里我们首先针对 React 16 下 Context API 的形态进行介绍。

图解 Context API 工作流

Context API 有 3 个关键的要素：React.createContext、Provider、Consumer。

我们通过调用 React.createContext，可以创建出一组 Provider。Provider 作为数据的提供方，可以将数据下发给自身组件树中任意层级的 Consumer，这三者之间的关系用一张图来表示：



注意：Cosumer 不仅能够读取到 Provider 下发的数据，**还能读取到这些数据后续的更新**。这意味着数据在生产者和消费者之间能够及时同步，这对 Context 这种模式来说至关重要。

从编码的角度认识“三要素”

- **React.createContext**，作用是创建一个 context 对象。下面是一个简单的用法示范：

```
1. const AppContext = React.createContext()
```

复制代码

注意，在创建的过程中，我们可以选择性地传入一个 defaultValue：

```
1. const AppContext = React.createContext(defaultValue)
```

复制代码

从创建出的 context 对象中，我们可以读取到 Provider 和 Consumer：

```
const { Provider, Consumer } = AppContext
```

复制代码

- **Provider**，可以理解为“数据的 Provider（提供者）”。

我们使用 Provider 对组件树中的根组件进行包裹，然后传入名为“value”的属性，这个 value 就是后续在组件树中流动的“数据”，它可以被 Consumer 消费。使用示例如下：

```
<Provider value={title: this.state.title, content: this.state.content}>
  <Title />
  <Content />
 </Provider>
```

复制代码

- **Consumer**，顾名思义就是“数据的消费者”，它可以读取 Provider 下发下来的数据。

其特点是需要接收一个函数作为子元素，这个函数需要返回一个组件。像这样：

```
<Consumer>
  {value => <div>{value.title}</div>}
</Consumer>
```

复制代码

注意，当 Consumer 没有对应的 Provider 时，value 参数会直接取创建 context 时传递给 createContext 的 defaultValue。

新的 Context API 解决了什么问题

想要知道新的 Context API 解决了什么问题，先要知道过时的 Context API 存在什么问题。

我们先从编码角度认识“过时的”Context API

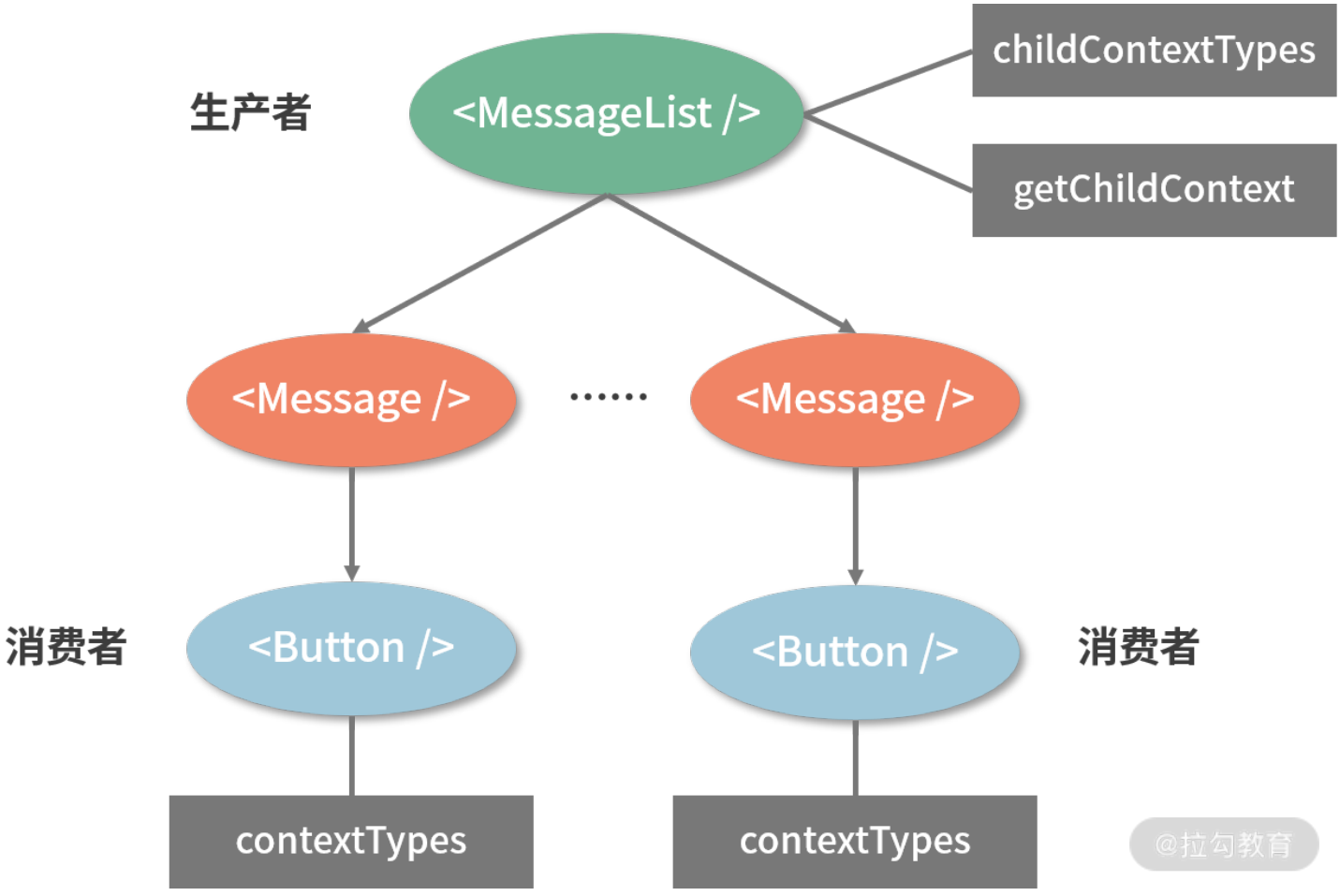
“过时的”是 React 官方对旧的 Context API 的描述，由于个人和团队在实际项目中都并不会考虑去使用旧 Context API 来解决问题，这里我直接引用过时的文档中的 Context API 使用示例：

```
import PropTypes from 'prop-types';
class Button extends React.Component {
  render() {
    return (
      <button style={{background: this.context.color}}>
        {this.props.children}
      </button>
    );
  }
```

复制代码

```
}
Button.contextTypes = {
  color: PropTypes.string
};
class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}
class MessageList extends React.Component {
  getChildContext() {
    return {color: "purple"};
  }
  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}
MessageList.childContextTypes = {
  color: PropTypes.string
};
```

为了方便你理解，我将上述代码对应的组织结构梳理到一张图里，如下图所示：



借着这张图，我们来理解旧的 Context API 的工作过程：

- 首先，通过给 MessageList 设置 childContextTypes 和 getChildContext，可以使其承担起 context 的生产者的角色；
- 然后，MessageList 的组件树内部所有层级的组件都可以通过定义 contextTypes 来成为数据的消费者，进而通过 this.context 访问到 MessageList 提供的数据。

现在回过头来，我们再从编码角度审视一遍“过时的” Context API 的用法。

首先映入眼帘的第一个问题是**代码不够优雅**：一眼望去，你很难迅速辨别出谁是 Provider、谁是 Consumer。同时这琐碎的属性设置和 API 编写过程，也足够我们写代码的时候“喝一壶了”。总而言之，从编码形态的角度来说，“过时的” Context API 和新 Context API 相去甚远。

不过，这还不是最要命的，最要命的弊端我们从编码层面暂时感知不出来，但是一旦你感知到它，麻烦就大了——前面我们特别提到过，“Cosumer 不仅能够读取到 Provider 下发的数据，**还能够读取到这些数据后续的更新”**。数据在生产者和消费者之间的及时同步，这一点对于 Context 这种模式来说是至关重要的，但旧的 Conext API 无法保证这一点：

> 如果组件提供的一个Context发生了变化，而中间父组件的 shouldComponentUpdate 返回 false，**那么使用到该值的后代组件不会进行更新**。使用了 Context 的组件则完全失控，所以基本上没有办法能够可靠的更新 Context。这篇博客文章很好地解释了为何会出现此类问题，以及你该如何规避它。
> ——React 官方

新的 Context API 改进了这一点：**即便组件的 shouldComponentUpdate 返回 false，它仍然可以“穿透”组件继续向后代组件进行传播，进而确保了数据生产者和数据消费者之间数据的一致性**。再加上更加“好看”的语义化的声明式写法，新版 Context API 终于顺利地摘掉了“试验性 API”的帽子，成了一种确实可行的 React 组件间通信解决方案。

理解了 Context API 的前世今生，接下来我们继续来串联 React 组件间通信的解决方案。

第三方数据流框架“课代表”：初探 Redux

对于简单的跨层级组件通信，我们可以使用发布-订阅模式或者 Context API 来搞定。但是随着应用的复杂度不断提升，需要维护的状态越来越多，组件间的关系也越来越难以处理的时候，我们就需要请出 Redux 来帮忙了。

什么是 Redux

我们先来看一下官方对 Redux 的描述：

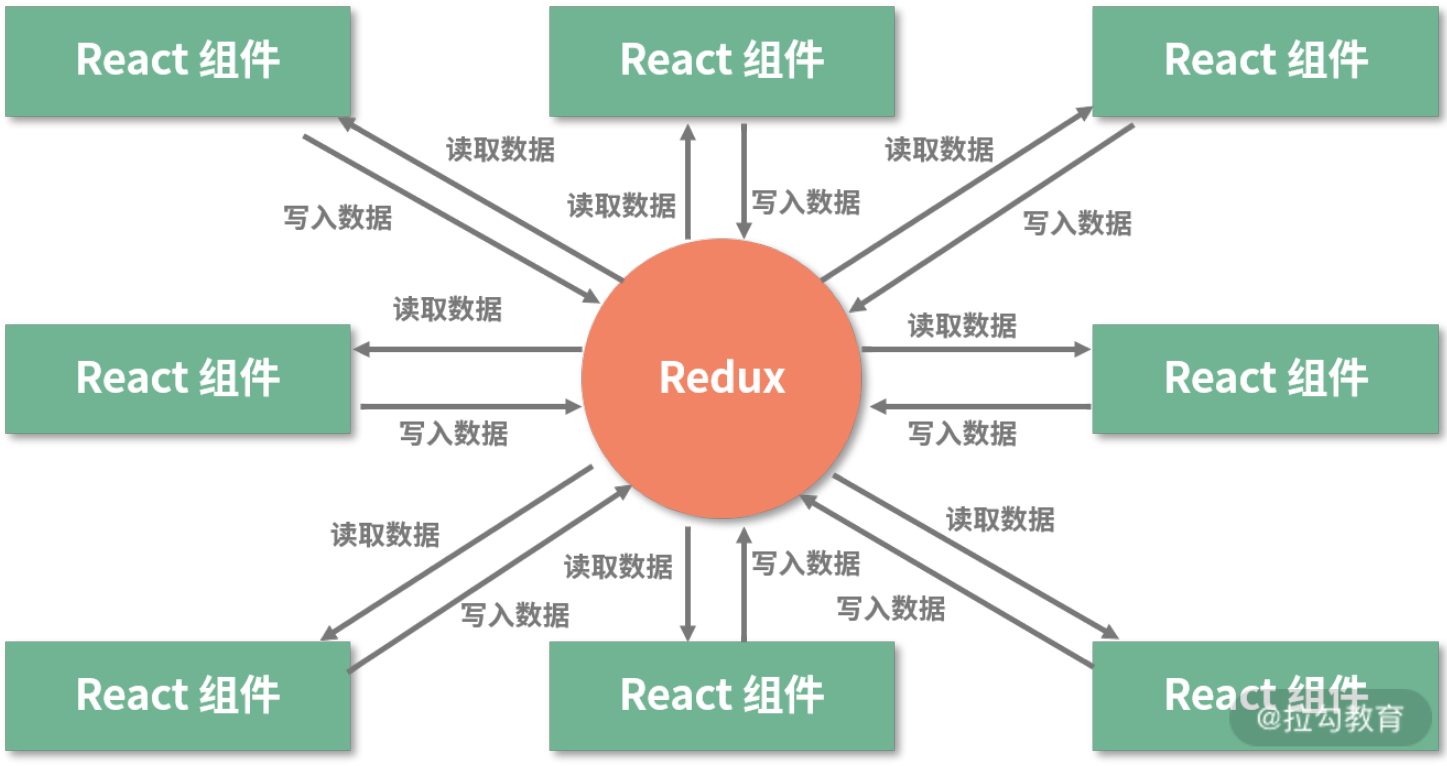
> Redux 是 JavaScript 状态容器，它提供可预测的状态管理。

我们一起品品这句话背后的深意：

- Redux 是为**JavaScript**应用而生的，也就是说它不是 React 的专利，React 可以用，Vue 可以用，原生 JavaScript 也可以用；
- Redux 是一个**状态容器**，什么是状态容器？这里我举个例子。

假如把一个 React 项目里面的所有组件拉进一个钉钉群，那么 Redux 就充当了这个群里的“群文件”角色，所有的组件都可以把需要在组件树里流动的数据存储在群文件里。当某个数据改变的时候，其他组件都能够通过下载最新的群文件来获取到数据的最新值。这就是“状态容器”的含义——存放公共数据的仓库。

读懂了这个比喻之后，你对 Redux、数据和 React 组件的关系想必已经形成了一个初步的认知。这里我帮你把这层关系总结进一张图里：



Redux 是如何帮助 React 管理数据的

Redux 主要由三部分组成：store、reducer 和 action。我们先来看看它们各自代表什么：

- store 就好比组件群里的“群文件”，它是一个**单一的数据源**，而且是只读的；

- action 人如其名，是“动作”的意思，它是**对变化的描述**。

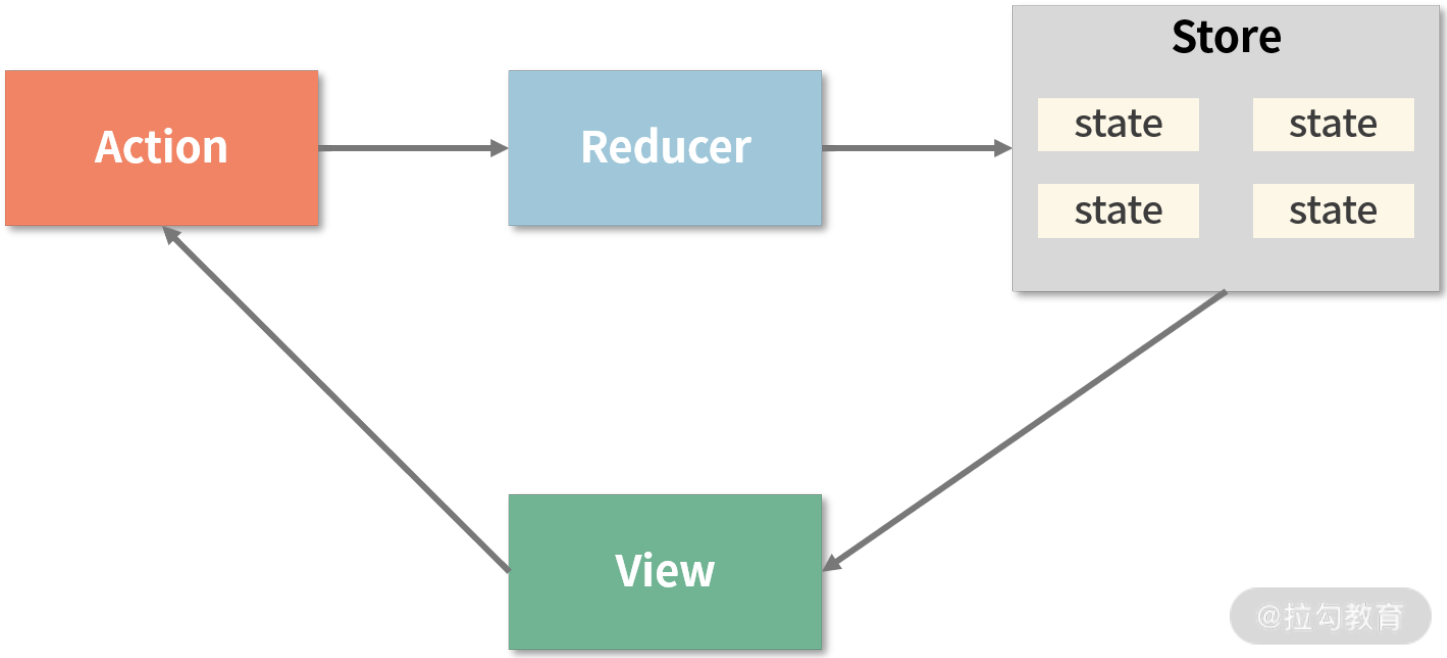
举个例子，下面这个对象就是一个 action：

```
const action = {
  type: "ADD_ITEM",
  payload: '<li>text</li>'
}
```

复制代码

- reducer 是一个函数，它负责**对变化进行分发和处理，** 最终将新的数据返回给 store。

store、action 和 reducer 三者紧密配合，便形成了 Redux 独树一帜的工作流：



从上图中，我们首先读出的是数据的流向规律：**在 Redux 的整个工作过程中，数据流是严格单向的。**这一点一定一定要背下来，面试的时候也一定一定要记得说——不管面试官问的是 Redux 的设计思想还是工作流还是别的什么概念性的知识，开局先放这句话，准没错。

接下来仍然是围绕上图，我们来一起看看 Redux 是如何帮助 React 管理数据流的。对于一个 React 应用来说，视图（View）层面的所有数据（state）都来自 store（再一次诠释了单一数据源的原则）。

如果你想对数据进行修改，只有一种途径：派发 action。action 会被 reducer 读取，进而根据 action 内容的不同对数据进行修改、生成新的 state（状态），这个新的 state 会更新到 store 对象里，进而驱动视图层面做出对应的改变。

对于组件来说，任何组件都可以通过约定的方式从 store 读取到全局的状态，任何组件也都可以通过合理地派发 action 来修改全局的状态。**Redux 通过提供一个统一的状态容器，使得数据能够自由而有序地在任意组件之间穿梭**，这就是 Redux 实现组件间通信的思路。

从编码的角度理解 Redux 工作流

到这里，你已经了解了 Redux 的设计思想和要素关系。接下来我们将站在编码的角度，探讨 Redux 的工作流，将上文中所提及的各个要素和流程具象化。

1. 使用 createStore 来完成 store 对象的创建

复制代码

```
// 引入 redux
import { createStore } from 'redux'
// 创建 store
const store = createStore(
    reducer,
    initial_state,
    applyMiddleware(middleware1, middleware2, ...)
);
```

createStore 方法是一切的开始，它接收三个入参：

- reducer；
- 初始状态内容；
- 指定中间件（这个你先不用管）。

这其中一般来说，只有 reducer 是你不得不传的。下面我们就看看 reducer 的编码形态是什么样的。

2. reducer 的作用是将新的 state 返回给 store

一个 reducer 一定是一个纯函数，它可以有各种各样的内在逻辑，但它最终一定要返回一个 state：

复制代码

```
const reducer = (state, action) => {
    // 此处是各种样的 state处理逻辑
    return new_state
}
```

当我们基于某个 reducer 去创建 store 的时候，其实就是给这个 store 指定了一套更新规则：

复制代码

```
// 更新规则全都写在 reducer 里
const store = createStore(reducer)
```

3. action 的作用是通知 reducer “让改变发生”

如何在浩如烟海的 store 状态库中，准确地命中某个我们希望它发生改变的 state 呢？reducer 内部的逻辑虽然不尽相同，但其本质工作都是“将 action 与和它对应的更新动作对应起来，并处理这个更新”。所以说**要想让 state 发生改变，就必须用正确的 action 来驱动这个改变**。

前面我们已经介绍过 action 的形态，这里再提点一下。首先，action 是一个大致长这样的对象：

```
const action = {
  type: "ADD_ITEM",
  payload: '<li>text</li>'
}
```

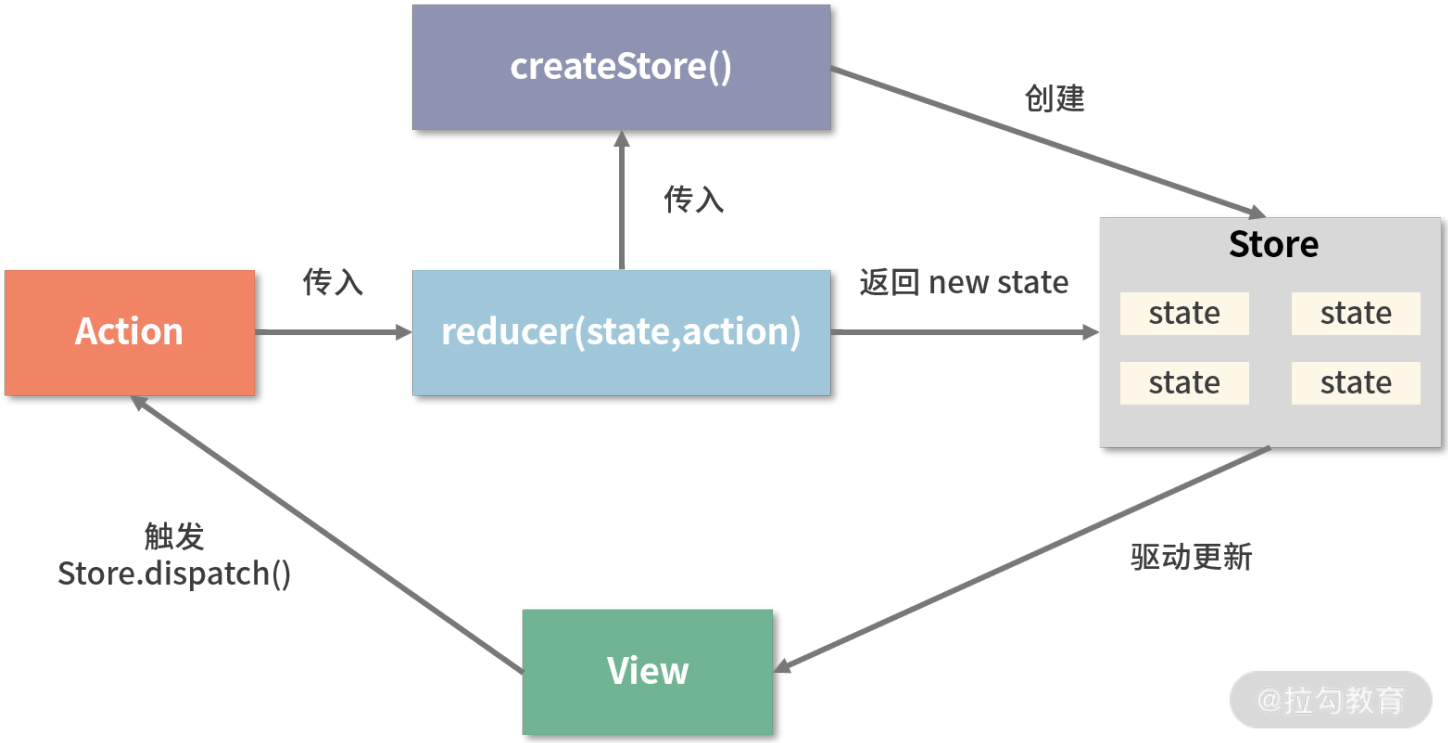
action 对象中允许传入的属性有多个，**但只有 type 是必传的**。type 是 action 的唯一标识，reducer 正是通过不同的 type 来识别出需要更新的不同的 state，由此才能够实现精准的“定向更新”。

4. 派发 action，靠的是 dispatch

action 本身只是一个对象，要想让 reducer 感知到 action，还需要“派发 action”这个动作，这个动作是由 store.dispatch 完成的。这里我简单地示范一下：

```
import { createStore } from 'redux'
// 创建 reducer
const reducer = (state, action) => {
    // 此处是各种样的 state处理逻辑
    return new_state
}
// 基于 reducer 创建 state
const store = createStore(reducer)
// 创建一个 action，这个 action 用 “ADD_ITEM” 来标识
const action = {
  type: "ADD_ITEM",
  payload: '<li>text</li>'
}
// 使用 dispatch 派发 action，action 会进入到 reducer 里触发对应的更新
store.dispatch(action)
```

以上这段代码，是从编码角度对 Redux 主要工作流的概括，这里我同样为你总结了一张对应的流程图：



注意：先别急着死磕。本课时并不要求你掌握 Redux 中涉及的所有概念和原理，只需要你跟着我的思路走，大致理解 Redux 中几个关键角色之间的关系，进而明白 Redux 是如何驱动数据在 React 组件间流动、如何帮助我们实现**灵活的组件间通信**的，这就够了。关于更多 Redux 的技术细节，我将在专栏的第三个大模块慢慢推敲。

总结

在 04 和 05 课时，我讲解的知识点覆盖面广、跨度大。面试场景下，考察此类问题的目的也主要是对候选人的知识广度进行检验。因此对于这两节的内容，你应抱着梳理“**知识地图**”的目的去学习，以**构建知识体系**为第一要务。完成第一要务后，再带着一个完整的上下文，去攻克某个具体的薄弱点。