Redux 是如何将 State 注入到 React 组件上去的？

1.React：用于构建用户界面的 JavaScript 库（负责组件的 UI 界面渲染的库）；

2.Redux：是 JavaScript 状态容器（负责管理数据的工具），也是一个独立的库，与 React 并没有直接关系，是 React-Redux 将它们俩联系起来的，Redux 的出现其实就是解决了复杂应用的状态管理问题，可以跨层级任意传递数据。

Redux 的原理：Redux 就是一个经典的发布订阅器。Redux 就是用一个变量存储所有的 State，并且提供了发布功能来修改数据，以及订阅（监听）功能来触发回调

Redux 核心源码：

```
/*
 * @param {Function} reducer     reducer
 * @param {any} preloadedState  初始化的 state，用的相对较少，一般在服务端渲染的时候使用
 * @param {Function} enhancer    中间件
 */
export default function createStore(reducer, preloadedState, enhancer) {
  // 实现第二个形参选填
  // 只有当第二参数传入的是中间件才会执行下面的代码
  if (typeof preloadedState === 'function' && typeof enhancer === 'undefined') {
    enhancer = preloadedState;
    preloadedState = undefined;
  }
  let currentReducer = reducer;
  let currentState = preloadedState; // 整个应用所有的 State 都存储在这个变量里
  let currentListeners = []; // 订阅传进来的的回调函数 <=>   Button.addEventListener('click', () => { ... })
  // 这是一个很重要的设计
  let nextListeners = currentListeners;
  function getState() {
    return currentState;
  }
  function subscribe(listener) {
    if (nextListeners === currentListeners) {
      // 浅复制  // 实际上 nextListeners 就是 currentListeners，避免直接操作 currentListeners，因为
      // 其他地方会用到 currentListeners，从而造成数据不一致。
      nextListeners = [...currentListeners];
    }
    nextListeners.push(listener);
    return function unsubscribe() {
        if (nextListeners === currentListeners) {
         // 浅复制
         nextListeners = [...currentListeners];
       const index = nextListeners.indexOf(listener);
       nextListeners.splice(index, 1);
    }
  }
// Button.addEventListener('click', () => { ... })
// Button.removeEventListener('click', () => { ... })
  function dispatch(action) {
    currentState = currentReducer(currentState, action); // 调用 reducer 来更新数据
```

```
    const listeners = (currentListeners = nextListeners); // 保证当前的 listeners 是最新的
    for (let i = 0; i < listeners.length; i++) {
      listeners[i](); // 依次执行回调函数
    }
    return action;
  }
  // 手动触发一次 dispatch，初始化
  dispatch({type: 'INIT'});
  return {
    getState,
    dispatch,
    subscribe,
  }
}
```

React-Redux 做了什么事情?

React-Redux 的作用就是订阅 Store 里数据的更新，他包含两个重要元素，Provider 和 connect 方法：

1.Provider 就是通过 React 的 Context API 把 Store 对象注入到 React 组件上去并把数据往下传。Provider 核心源码：

```
import React from 'react'
import PropTypes from 'prop-types'
export default class Provider extends React.Component {
  // context 往所有子组件，孙组件里传递数 // props 父组件往子组件里传递数据 // state 组件自身的数据
  // 声明一个 context 数据
  getChildContext() {
    return { store: this.store }
  }
  constructor(props, context) {
    super(props, context)
    this.store = props.store
  }
  render() {
    return React.Children.only(this.props.children)
  }
}
Provider.childContextTypes = {
  store: PropTypes.object
}
```

2.connect 就是一个高阶组件，接收 Provider 传递过来的 store 对象，并订阅 store 中的数据（的更新），如果 store 中的数据发生改变，就调用 setState 触发组件更新。connect 核心源码：

```
import React from 'react'
import PropTypes from 'prop-types'
// connect() => () => {} 函数的柯里化
// const sum = (a) => { return (b) => a + b }
// sum(1)(2) -> connect(mapStateToProps, mapDispatchToProps)(Comp)
// HOC 高阶组件
const connect = (mapStateToProps = state => state, mapDispatchToProps = {}) => (WrapComponent) => {
  return class ConnectComponent extends React.Component {
    static contextTypes = {
      store: PropTypes.object
```

```
    }
    constructor(props, context) {
      super(props, context)
      this.state = {
        props: {} // 声明了一个叫做 props 的 state
      }
    }
    componentDidMount() {
      const { store } = this.context  // 从 Context 中拿到 store 对象
      store.subscribe(() => this.update()) // 订阅 Redux 的数据更新
      this.update()
    }
    // 每次数据有更新的时候，就会调用这个方法
    update() {
      const { store } = this.context // 从 Context 中拿到 store 对象
      const stateProps = mapStateToProps(store.getState()) // 把 store 中的全部数据传到组件内部
      const dispatchProps = mapDispatchToProps(store.dispatch)//把 store.dispatch 传到组件内部
      // 调用 setState 触发组件更新
      // 将最新的 state 以及 dispatch 合并到当前组件的 props 上
      this.setState({
        props: {
          ...this.state.props,
          ...stateProps,
          ...dispatchProps
        }
      })
    }
    render() {
      // 传入 props
      return <WrapComponent {...this.state.props}></WrapComponent>
    }
  }
}
export default connect;
// const mapStateToProps = state => { Tips：这里注入进来的
//   return {
//     value: state,
//   }
// }
// const mapDispatchToProps = dispatch => {
//   return {
//     onIncrement: () => dispatch({ type: 'INCREMENT' }),
//     onDecrement: () => dispatch({ type: 'DECREMENT' }),
//   }
// }
```

考虑需不需要使用 Redux?

如果你 UI 层较简单，没有很多交互，Redux 就是不必要的，用了反而增加复杂性（reducer, action => this.setState()）。

另外，Redux 可以解决跨组件间数据传递的问题，并且修改数据很清晰。

使用 Redux 的痛点是什么？
修改一次数据，太麻烦，dispatch(action) -> 调用 reducer 计算 -> 触发回调 -> 更新数据。
Redux 在项目中使用，最大的弊端就是样板代码（action, reducer）太多了，修改数据的链路太长。

使用 Redux 有哪些比较好的实践方式？
可以通过一些手段减少模板代码，从而简化 Redux API：可以通过一些手段（工具）减少模板，从而简化 Redux API，如引入 Redux-Actions 来减少抒写固定不变的代码，以及使用 yeoman 来用命令自动生成样板文件以及代码。
1.使用 redux-actions ，在初始化 reducer 和 action 构造器时减少样板代码：
（1）减少创建 action 时写一堆固定的方法 () => ({ type: 'XXX' }) -> createAction('XXX', payload => payload)
（2）减少创建 reducer 时写一堆固定的 switch switch{} -> handleActions({})
2.使用 cli 工具，帮我们自动生成模板代码：
使用 yeoman 来帮我们用命令一键创建样板文件和样本代码

为什么 Redux 处理不了异步问题以及如何解决？
dispatch 默认只能接收一个 Object 类型的 action，因为 reducer 里面要接收 action.type 来处理不同的数据。
从 Redux 代码的原理出发：

```
dispatch(action)   // action = { type: 'XXX', payload: 'xxx' };
// ->reducer 是一个纯函数，无法处理其他类型的数据
// 所以 dispatch 默认接收的 action 不可以是其他类型的
dispatch((dispatch) => {
  setTimeout(() => {
    dispatch({ type: 'INCREMENT'})
  }, 3000);
});
```

Redux 异步问题可以通过中间件来解决：
1.使用 Redux-thunk 中间件，来解决异步 Action 的问题
2.使用 Redux-saga 中间件，让异步行为成为架构中独立的一层(称为 saga)

函数组件 vs 类组件

区别	函数组件(无状态组件)	类组件
是否有 this	没有	有
是否有生命周期	没有	有
是否有状态 state	没有	有

React Hooks 是 v16.8 版本引入了全新的 API，它算是一个颠覆性的变革：
所有的 React 组件都可以是函数组件，再也不需要写类组件了，再也不需要记住 React 有哪些生命周期函数了。
React Hooks 可以让我们的代码变得更加简洁，结构更清晰。
1.React Hooks 是一个新的 API，可以用函数来写所有的组件
2.可以让函数组件也可以拥有自己的状态管理（包括 state 和生命周期函数）

```
import { useState, useEffect } from 'react'; //v16.8
function Example() {
  const [count, setCount] = useState(0); // 类似于 this.state = { count: 0 }
  const [loading, setLoading] = useState(false); // this.state = {loading: false}
    // 声明的名称叫做 xxx, 那么必定有一个值叫做 setXxx() (固定的写法)
```

```
  // 类似于 componentDidMount 和 componentDidUpdate;
  useEffect(() => {
    // 改变 title
    document.title = `You clicked ${count} times`;
  });
    useEffect(() => {
    // 发送 ajax 请求
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

3.可以通过创建自定义的 Hooks 来抽离可复用的业务组件：

创建自定义 Hooks

```
// 注意 hooks 约定必须以 use 开头(useA, useB)
const useFetchData = (filmId) => {
  const [loading, setLoading] = useState(false); // 只有在第一次加载的时候才会 被 false 复制
  const [data, setData] = useState({});
  useEffect(() => {
    setLoading(true); // 1. 设置 loading 为 true
    fetch(`https://swapi.co/api/films/${filmId}`) // 2. 发送请求
      .then(data => {
        setData(data); // 3. 收到请求后，设置 data 为请求到的数据
        setLoading(false); // 4. 设置 loading 为 false
      });
  }, [filmId]); // filmId 变化的时候，才触发 useEffect
  return [loading, data];
};
```

使用自定义 Hooks

```
import useFetchData from './useFetchData';
function App({ filmId }) {
  const [loading, data] = useFetchData(filmId);
  if (loading === true) {
    return <p>Loading ...</p>;
  }
  return (
    <div>
      <p>电影名称: {data.title}</p>
      <p>导演: {data.producer}</p>
      <p>发布日期: {data.release_date}</p>
    </div>
  );
}
```