1. Redux

> React 是视图层框架。 Redux 是一个用来管理数据状态和UI状态的 JavaScript 应用工具。
> Redux 提供了一个叫 store 的统一仓储库，组件通过 dispatch 将 state 直接传入 store ， 不用通过其他的组件。并且组件通过 subscribe 从 store 获取到 state 的改变。使用了 Redux ， 所有的组件都可以从 store 中获取到所需的 state ， 他们也能从 store 获取到 state 的改变。这比组件之间互相传递数据清晰明朗的多。
> 主要解决的问题：
> 单纯的Redux只是一个状态机，是没有UI呈现的，react-redux作用是将Redux的状态机和React的UI呈现绑定在一起，当你dispatch action改变state的时候，会自动更新页面。

1.1 Redux原理

1.1.1 文件组成

- compose.js 提供从右到左进行函数式编程
- createStore.js 提供作为生成唯一 store 的函数
- combineReducers.js 提供合并多个 reducer 的函数，保证 store 的唯一性
- bindActionCreators.js 可以让开发者在不直接接触 dispacth 的前提下进行更改state的操作
- applyMiddleware.js 这个方法通过中间件来增强 dispatch 的功能

```
const actionTypes = {
    ADD: 'ADD',
    CHANGEINFO: 'CHANGEINFO',
}

const initState = {
    info: '初始化',
}

export default function initReducer(state=initState, action) {
    switch(action.type) {
        case actionTypes.CHANGEINFO:
            return {
                ...state,
                info: action.preload.info || '',
            }
        default:
            return { ...state };
    }
}

export default function createStore(reducer, initialState, middleFunc) {

    if (initialState && typeof initialState === 'function') {
        middleFunc = initialState;
        initialState = undefined;
    }

    let currentState = initialState;

    const listeners = [];

    if (middleFunc && typeof middleFunc === 'function') {
        // 封装dispatch
        return middleFunc(createStore)(reducer, initialState);
    }

    const getState = () => {
        return currentState;
    }

    const dispatch = (action) => {
        currentState = reducer(currentState, action);

        listeners.forEach(listener => {
            listener();
        })
    }

    const subscribe = (listener) => {
```

```
        listeners.push(listener);
    }

    return {
        getState,
        dispatch,
        subscribe
    }
}
```

1.1.2 工作流程

- `const store= createStore（fn）` 生成数据;
- `action: {type: Symble('action01), payload:'payload' }` 定义行为;
- `dispatch` 发起 `action`： `store.dispatch(doSomething('action001'));`
- `reducer`： 处理 `action`， 返回新的 `state`；

2. Hooks

> 函数组件真正地将数据和渲染绑定到了一起。函数组件是一个更加匹配其设计理念、也更有利于逻辑拆分与重用的组件表达形式。
> React-Hooks 是一套能够使函数组件更强大、更灵活的“钩子”.(弥补类组件拥有的优点，比如生命周期、对 state 的管理等)

hooks解决的问题

- 在组件之间复用状态逻辑很难（共享hook）
- 复杂组件变得难以理解 - 拆分更小粒度
- 难以理解的 `class`

使用限制

- Hook 不能在 class 组件中使用，只能在 React 的函数组件 / 自定义hook 中调用 Hook。
- 只能在函数最外层调用 Hook。不要在循环、条件判断或者子函数中调用。

> Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性

3. Virtual DOM

> 从本质上来说， `Virtual Dom` 是一个 `JavaScript` 对象，

> - 通过对象的方式来表示 DOM 结构。将页面的状态抽象为 JS 对象的形式，配合不同的渲染工具，使跨平台渲染成为可能。
> - 通过事务处理机制，将多次 DOM 修改的结果一次性的更新到页面上，从而有效的减少页面渲染的次数，减少修改DOM的重绘重排次数，提高渲染性能。

> 虚拟 DOM 是对 DOM 的抽象，这个对象是更加轻量级的对 DOM 的描述。
>
> - 它设计的最初目的，就是更好的跨平台，比如node.js就没有DOM，如果想实现SSR，那么一个方式就是借助虚拟dom，因为虚拟dom本身是js对象。 在代码渲染到页面之前，vue或者react会把代码转换成一个对象（虚拟DOM）。
> - 以对象的形式来描述真实dom结构，最终渲染到页面。在每次数据发生变化前，虚拟dom都会缓存一份，变化之时，现在的虚拟dom会与缓存的虚拟dom进行比较。在vue或者react内部封装了diff算法，通过这个算法来进行比较，渲染时修改改变的变化，原先没有发生改变的通过原先的数据进行渲染。

> 另外现代前端框架的一个基本要求就是无须手动操作DOM
>
> - 一方面是因为手动操作DOM无法保证程序性能，多人协作的项目中如果review不严格，可能会有开发者写出性能较低的代码
> - 另一方面更重要的是省略手动DOM操作可以大大提高开发效率

使用 Virtual DOM 的原因：

1. 保证性能下限，在不进行手动优化的情况下，提供过得去的性能
   下面对比一下修改DOM时真实DOM操作和Virtual DOM的过程，来看一下它们重排重绘的性能消耗：
   1. 真实DOM: 生成 HTML 字符串＋ 重建所有的DOM元素
   2. Virtual DOM：生成 vNode ＋ DOMDiff ＋必要的 DOM 更新
      Virtual DOM 的更新 DOM 的准备工作耗费更多的时间，也就是 JS 层面，相比于更多的 DOM 操作它的消费是极其便宜的。
2. 跨平台
   Virtual DOM 本质上是 JavaScript 的对象，它可以很方便的跨平台操作，比如服务端渲染、uniapp等。

4. React diff 算法的原理是什么?

> diff 算法探讨的就是虚拟 DOM 树发生变化后，生成 DOM 树更新补丁的方式。它通过对比新旧两株虚拟 DOM 树的变更差异，将更新补丁作用于真实 DOM ，以最小成本完成视图更新。
> 具体的流程如下：

- 真实的 DOM 首先会映射为虚拟 DOM ；

- 当虚拟 DOM 发生变化后，就会根据差距计算生成 patch ，这个 patch 是一个结构化的数据，内容包含了增加、更新、移除等；
- 根据 patch 去更新真实的 DOM ，反馈到用户的界面上。

diff算法可以总结为三个策略，分别从树、组件及元素三个层面进行复杂度的优化

1. 策略一：忽略节点跨层级操作场景，提升比对效率。（基于树进行对比）
2. 策略二：如果组件的 class 一致，则默认为相似的树结构，否则默认为不同的树结构。（基于组件进行对比）
3. 策略三：同一层级的子节点，可以通过标记 key 的方式进行列表对比。（基于节点进行对比）

> React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染此外， React 还需要借助 Key 值来判断元素与本地状态的关联关系。

> 注意事项：
>
> 1. key值一定要和具体的元素一一对应；
> 2. 尽量不要用数组的index去作为key；
> 3. 不要在render的时候用随机数或者其他操作给元素加上不稳定的key，这样造成的性能开销比不加key的情况下更糟糕。

Others

同时引用这三个库react.js、react-dom.js和babel.js它们都有什么作用?

- react：包含react所必须的核心代码
- react-dom：react渲染在不同平台所需要的核心代码
- babel：将jsx转换成React代码的工具

1. Motivation

- 高阶组件 =>“嵌套地狱” / render props => 代码难以理解/维护
  - => React需要为共享状态逻辑提供更好的原生途径(复用状态逻辑 无需修改组件结构)
- 复杂组件充斥状态逻辑和副作用，理解困难
  - => Hook将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据）
- class组件理解困难 vs React函数组件
  - => Hook 使你在非 class 的情况下使函数组件保留状态(保留函数特性)

2. 基础Hooks

> - Hook 不能在 class 组件中使用，只能在 React 的函数组件 / 自定义hook 中调用 Hook。
> - 只能在函数最外层调用 Hook。不要在循环、条件判断或者子函数中调用。

useState()

> State Hook
>
> - useState 通过在函数组件里调用它来给组件添加一些内部 state。
> - React 会在重复渲染时保留这个 state。useState 会返回一对值：*当前状态*和一个*更新它的函数*，可以在事件处理函数中或其他一些地方调用这个函数。
> - 一般来说，在函数退出后变量就会”消失”，而 state 中的变量会被 React 保留。
> - 类似 class 组件的 this.setState，但是它不会把新的 state 和旧的 state 进行合并。

useEffect()

> Effect Hook
>
> - useEffect 就是一个 Effect Hook，给函数组件增加了操作副作用的能力。
> - 功能类似于class组件中componentDidMount componentDidUpdate componentWillUnmount 生命周期函数
> - 通过调用useEffectHook，React 组件获知在渲染后需要执行某些操作。React 会保存传递的函数（ “effect”），并且在执行 DOM 更新之后调用它。
> - 将 useEffect 放在组件内部，可以方便我们在 “effect”直接访问state变量（或其他 props）。不需要特殊的 API 来读取它 —— 它已经保存在函数作用域中。Hook 使用了 JavaScript 的闭包机制，而不用在 JavaScript 已经提供了解决方案的情况下，还引入特定的 React API。
> - useEffect 在每次渲染后都会执行。 在第一次渲染之后和每次更新之后都会执行。React 保证了每次运行 effect 的同时，DOM 都已经更新完毕。

- 副作用需要清除：在effect中返回一个函数 （effect可选的清除机制）===> componentWillUnmount
  - 适用情形：副作用需要清除情况。例如订阅外部数据源。可以防止引起内存泄露

```
useEffect(() => {
  function handleStatusChange(status) {setIsOnline(status.isOnline);}
  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  }; // 清除函数
});
```

- useEffect 的第二个可选参数，该参数可以不传，每次更新组件都会调用，也可以传入依赖，在依赖变化的时候调用 ===>componentDidUpdate；
- 确保数组中包含了所有外部作用域中会随时间变化并且在 effect 中使用的变量 => 弊端：代码会引用到先前渲染中的旧变量
- 传递一个空数组（[]）作为第二个参数：执行只运行一次的 effect（仅在组件挂载和卸载时执行），effect 内部的 props 和 state 就会一直拥有其初始值 ===>componentDidMount
- 如果同时存在多个 useEffect，会按照出现次序执行。

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // 仅在 count 更改时更新
```