React知识点

React特点

React思想：组件化思想，函数式编程

1. 声明式: 复杂过程封装成方法

- 开发模式:
    1. 命令式
    2. 声明式

2. 组件化编码: 复用html/css/js

- 区别于模块化: 只能复用js

3. 一次学习,随处编写（支持多端）

- 支持native的方案: react native: js->安卓/ios flutter:新语言,性能好,google开发

4. 高效
    1. 虚拟DOM,不总是直接操作DOM
    2. DOM diff算法（减少重排重绘，性能好）
5. 数据驱动视图

- React严格上只针对MVC的view层，区别于vue的MVVM模式
- 数据单向流，区别vue实现了数据双向绑定
    - 更容易追踪数据变化排查问题

JSX语法

```
// React中HTML的写法成为JSX
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>

// 被Babel编译成
```

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

- 参考文档：JSX的理解
- 查看JSX会被转化成什么？在线Babel编译器

jsx语法

- 作用:简化创建虚拟DOM对象
  - JSX => React.createElement() => 虚拟DOM (JS对象) => 真实DOM 也就是说JSX就是React.createElement('component',props,...children)的语法糖
- 语法规则:
  1. 以<开头, 会当做html标签解析, 如果是同名标签或首字母小写就被解析同名元素, 如果不是或首字母大写会当成组件解析
  2. 以{开头, 里面代码会当做js代码解析、变量:{}
  3. babel编译: 将jsx代码编译为原生的react语法

Babel会将JSX转换为 `React.createElement()` 函数的调用

react事件处理

React中使用**合成事件**，并结合jsx语法传入一个函数作为事件处理函数

语法上与DOM事件区分

1. 通过onXxx属性指定组件的合成事件处理函数(注意小驼峰命名法)

```
<button onClick="this.push('/test2')">push test2</button>
```


2. 阻止浏览器默认行为不一样 **实现源码:** react使用自定义(合成)事件，而不是使用原生DOM事件
   - react中事件是通过事件委托方式处理的(委托给组件最外层的元素)
   - 参考： blog.csdn.net/me_never/ar...

Virtual DOM 的优势

原因： DOM操作很慢，操作JS会快很多，所以可以使用JS对象模拟DOM，渲染DOM，最小化页面重排重绘，减少重排重绘的次数,收集变化,一次性重排重绘。 **为什么DOM操作很慢?** 执行JS需要JS引擎，执行DOM需要渲染引擎，使用JS操作DOM，就需要两个线程的引擎进行通信，如果频繁的操作，则两个线程需要频繁的通信，操作DOM还会造成重绘和重排，也会造成性能损耗。

难点在于：如何对比出新旧JS对象的最小差异并且局部更新DOM

对比出新旧JS对象最小差异的算法：

DOM是多叉树结构如果完整对比两棵树的差异，需要的复杂度很高，react团队优化了对比算法 就是只对比同层节点，而不是跨层对比，因为在实际业务中很少进行跨层移动DOM元素。

算法分为三种：

- tree Diff DOM是多叉树结构
    1. 如果根节点为不同类型的元素，则拆卸整个节点，建立新的节点
    2. 如果为同一类型的元素，则继续对比元素的属性和子节点
- component Diff
    1. 如果组件不同则拆卸组件，建立新的组件节点
    2. 如果组件相同则对比细节（属性，子节点）
- Element Diff 正常的对比都是走tree diff但是这种消耗较大，如果是一个数组结构，给每个子元素添加一个key，只对比key，则降低了消耗 **key有什么作用?**
    1. 能让diff 算法性能更好
    2. 什么时候用id/index 1）id是通用,能用就用 2）Index只适用于给列表末尾添加的数据,如果顺序改变或者删除元素使用index作为key值性能就不好了
- 提高性能
    - 作为一个兼容层，让我们还能对接非Web端的系统，实现跨端开发
    - 同样的，通过Virtual DOM我们可以渲染到其他的平台，比如实现SSR，同构渲染等。
    - 实现组件的高度抽象化

dangerouslySetInnerHTML

- 用法 `<div className="tab-introduction" dangerouslySetInnerHTML={{ __html: a }} />`

- 作用是：react中DOM元素的innerHTML替换方案，等同于获取元素设置元素的innerHTML
- 使用场景：富文本框内容直接渲染到div中，一般从后端获取的富文本内容都是字符串类型的html标签，例如：`'<p>134</p>'`，这种字符串直接渲染只能使用富文本框组件；如果需要在div中直接渲染，只能通过innerHTML方式渲染，在react就可以直接使用dangerouslySetInnerHTML属性

React生命周期

谈到React生命周期，必须谈到react v16版本引入的fiber机制，这个机制影响了部分生命周期的调用，且引入两个新的生命周期API来解决问题。

fiber

为了解决：当组件嵌套层级很深，修改了父组件的state，这样调度栈就会很长，如果中间有复杂的操作，会造成长时间阻塞主线程，带来不来的用户体验。fiber之前叫做stack reconciler，相当于一个递归调用所有的父组件到子组件找到虚拟dom的不同。

fiber在不影响用户体验的基础上，将这些同步渲染改成**异步渲染，去分段根据优先级计算更新。**

异步渲染有两个阶段：reconciliation和commit阶段：

reconciliation阶段是可以被打断的，这一阶段任务是完成虚拟DOM的对比。因此这阶段的生命周期可能会被多次调用

commit是不可以被打断的，它会一直执行下去，任务是完成页面更新

reconciliation阶段：（完成虚拟DOM对比）

- componentWillMount
- componentWillReceiveProps => getDerivedStateFromProps
- shouldComponentUpdate
- componentWillUpdate => getShapshotBeforeUpdate

commit阶段：（完成更新页面）

- ComponentDidMount
- ComponentDidUpdate
- ComponentWillUnMount

reconciliation渲染阶段可以被打断，所以这个阶段的生命周期可能被调用多次，而引发问题，所以应该避免去使用，fiber也引入了新的API来替代这些API，除了ComponentShouldUpdate用来进行性能优化使用。

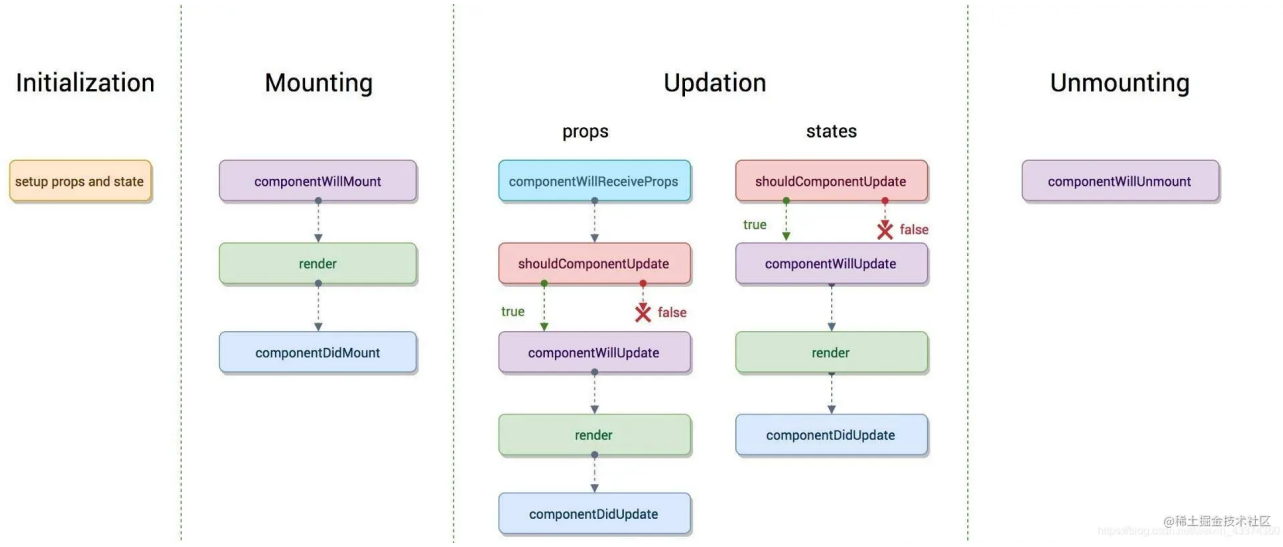
- **getDerivedStateFromProps(nextPorps,preState)**替代了componentWillReceiveProps:
  - 会在组件初始化和更新的时候被调用，render之前调用(也就是每次组件渲染都会被调用，而区别于componentWillReceiveProps只有在父组件更新时才会被调用)
  - 它应返回一个对象来更新 state，如果返回 null 则不更新任何内容
  - 只有当state完全取值与props时使用
- **getShapshopBeforeUpdate**替代了componentWillUpdate： 会在组件更新之后，DOM更新之前被调用，用于读取最新的DOM

16版本以前的生命周期

- 初始阶段：constructor，render
- 更新阶段：
  - componentWillreceiveProps(nextProps)子组件props被修改了会被调用
  - shouldConponentUpdate,返回true会调用componentWillUpdate，render和componentDidUpdate，
  - componentDidUpdate(prevProps, prevState)
- 卸载阶段：componentWillUnMount

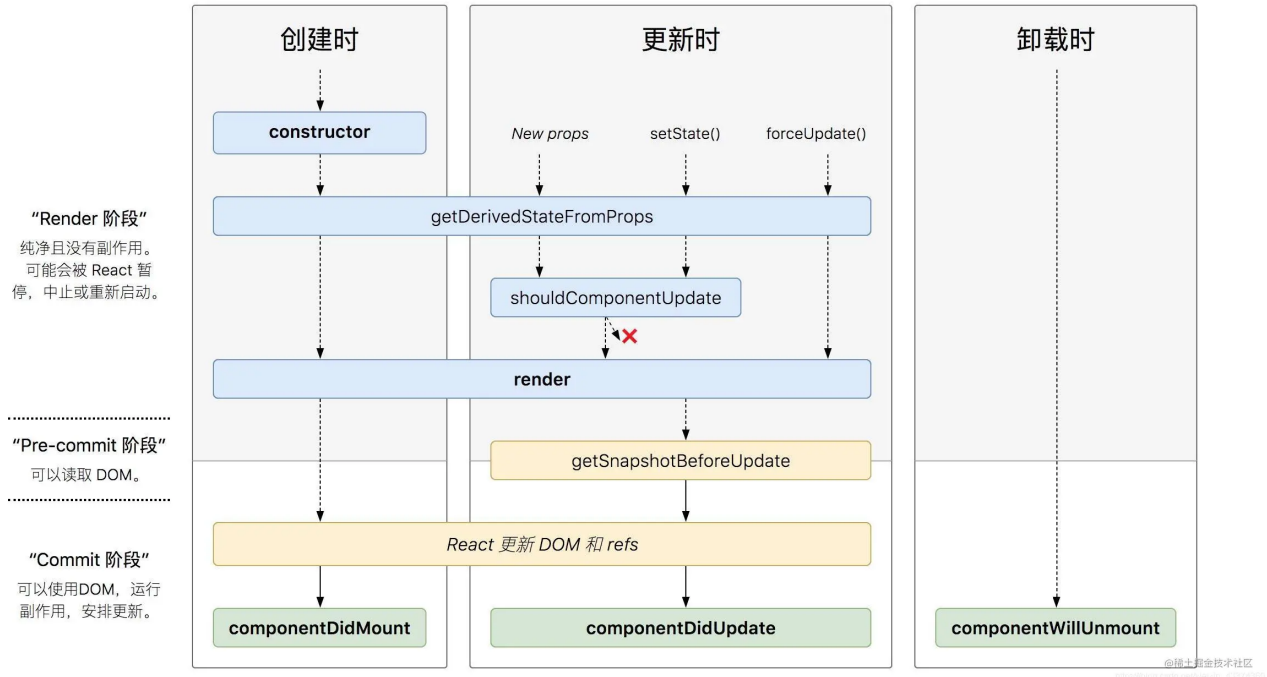
componentDidUpdate用法

```js
componentDidUpdate(prevProps) {
  // 典型用法（不要忘记比较 props）：
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```



新的生命周期：

- 新增了getDerivedStateFromProps和getSnapshotBeforeUpdate替代了，去除了componentWillMount和componentWillUpdate
- 初始阶段：constructor，getDerivedStateFromProps，render
- 更新阶段：getDerivedStateFromProps，shouldComponentUpdate，返回true则触发render，getSnapshotBeforeUpdate，componentDidUpdate
- 卸载阶段：componentWillUnmount



componentDidCatch: 用于ErrorBoundary（生命周期内部等抛出错误处理）

错误边界，包裹App组件 涉及到**componentDidCatch**生命周期：后代组件抛出错误后调用；

1. 两个专门处理错误边界的生命周期函数（当渲染过程，生命周期，或子组件的构造函数抛出错误）
2. **static getDerivedStateFromError(error)**
3. **componentDidCatch(info, error)** 用法：componentDidCatch

javascript 复制代码

```
ReactDOM.render(
  <ErrorBoundary>
    <App />
  </ErrorBoundary>,
  document.getElementById('root')
);
```

setState

setState是异步的，也是同步的 setState什么时候是异步的： setState什么时候是同步的：

- 两种使用方法: this.setState({})；普通修改 this.setState((state)=>{})；需要依赖当前state进行修改state时 this.setState({},()=>{})；state变更后需要马上获取其值进行使用时

异步&同步

- 在react相关的事件中，合成事件和生命周期中是异步的
- 在异步和原生DOM事件中是同步的

props

props.children

React原文：react组合和继承思想篇

理解： 为props的默认属性，将react组件**Title**作为JSX标签时，标签里的所有内容，将会作为{props.children}传递给**Title**组件，传递给让其能够编写嵌套和组合结构被传递的这些子组件都会出现在输出结果中。 **Title**中可以预留几个'自定义的内容'，或使用其他props属性，组合props.children使用，实现组件的**重用和组合**。

使用场景： **Siderbar**(侧边栏)和**Dialog**(对话框)，这些组件无法提前知晓它们子组件的具体内容。等展现通用容器（box）的组件中特别容易遇到这种情况

props.children的数据类型：

- 如果没有内容，则为undefined
- 有一个标签内容，则为Object
- 有多个标签内容，则为Array 原理是：将我们写的JSX 这样能够更加灵活的使用嵌套结构

javascript 复制代码

```
function Title({ text, children }) {
  return (
    <h1>
      { text }
      <br />
      { children }
    </h1>
  );
}
function App() {
  return (
    <Title text='hello world'>
      <span>
        community
      </span>
      </Title>
  );
}
```

javascript 复制代码

```
function Title({ text, children }) {
  return (
    <h1>
      { text }
      { children[1] }
    </h1>
  );
}
function App() {
  return (
    <Title text='hello world'>
      <span>community</span>
```

```
      <span>JavaScript</span>
    </Title>
  );
}
```

- react代码复用的方法
- React推荐使用组合模式复用代码而不是继承 参考react官网:reactjs.bootcss.com/docs/compos...

props.children来复用任何标签内的代码

- 每个组件都可以获取到props.children
- 它包含组件的开始标签和结束标签之间的内容
- **目的： 为了解决React中代码复用的问题**
  - 参考： 使用组合模式实现代码复用
    - reactjs.bootcss.com/docs/compos...

简单使用:

```
<Welcome>Hello world!</Welcome>
```

对于 class 组件，请使用 this.props.children 来获取：

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

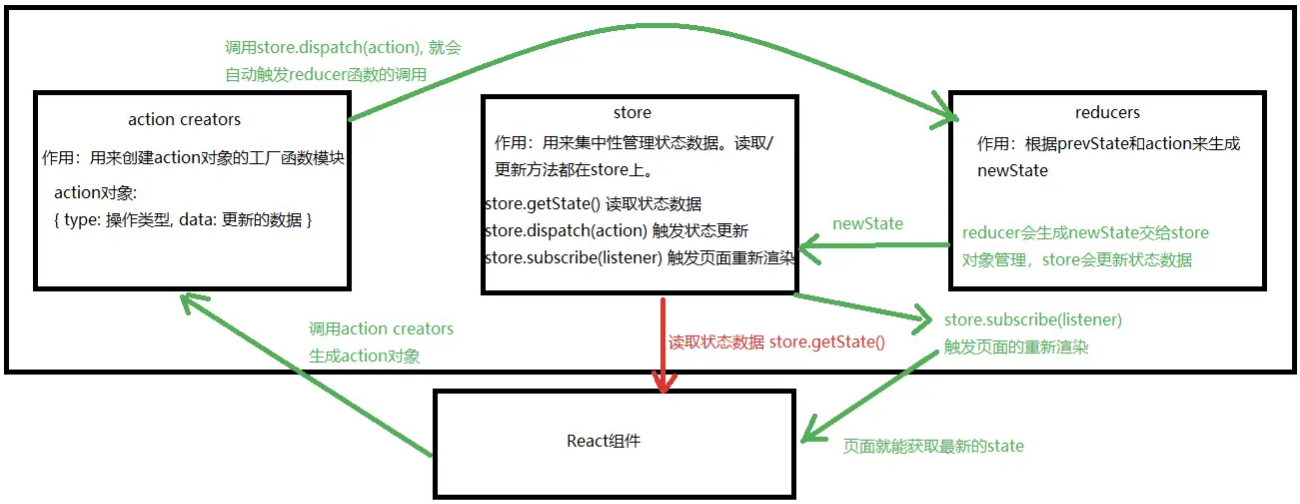
组件分类

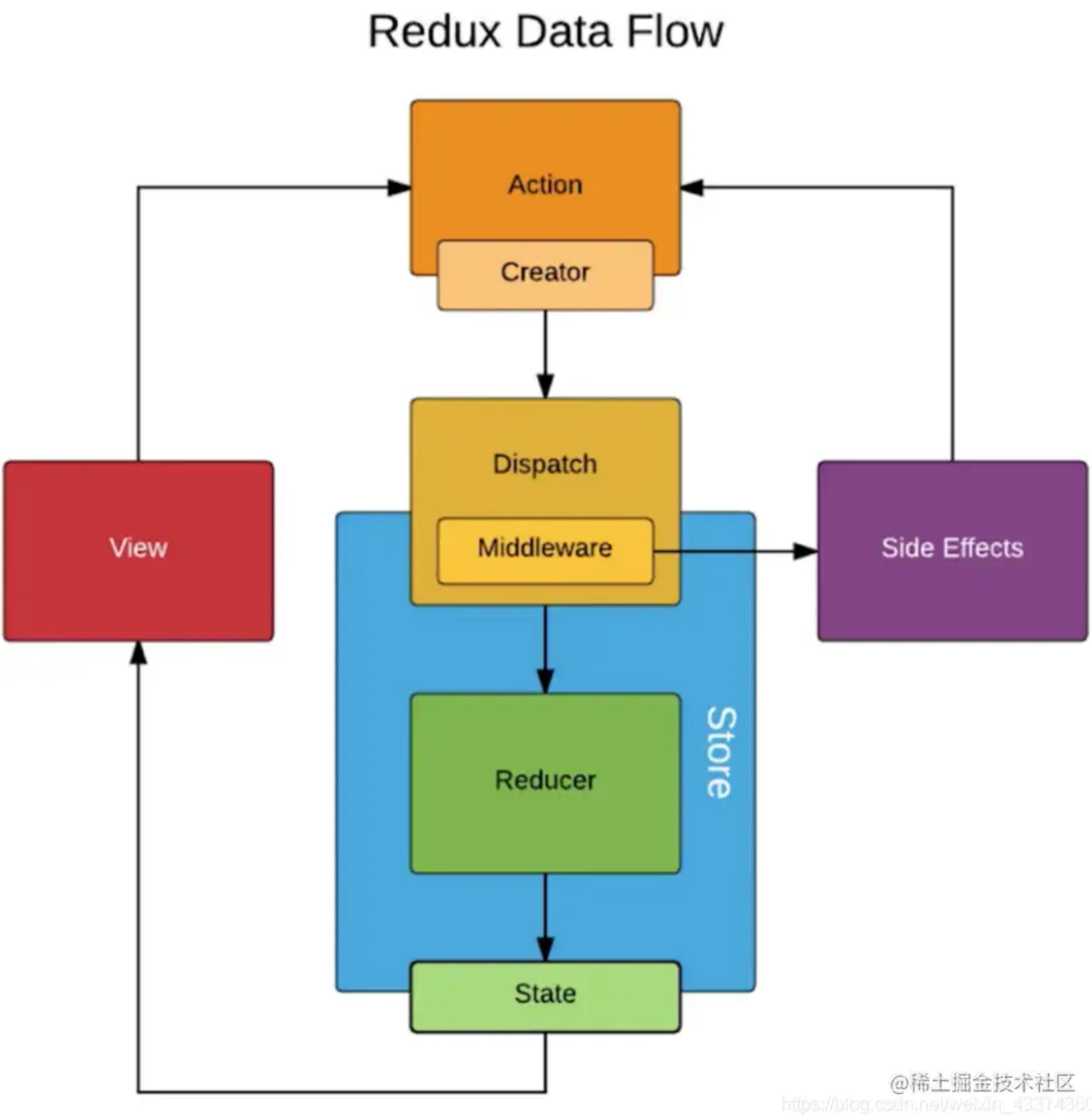
- 受控组件:根据收集form表单数据的方式分类：

1. 非受控组件(通过ref获取数据)收集表单数据
2. 受控组件(state, onChange)收集表单数据
   - 收集表单数据的时候:不使用全局变量,使用state和onChange事件收集表单数据

React组件通信

Redux





聊聊 Redux 和 Vuex 的设计思想

文章

两个以上组件通信redux

- redux是什么?
    - 是一个状态机
- redux作用:

    - 集中管理状态

- 组件使用redux

    - 读取redux中状态
    - 更新redux中状态

- 三大模块:

```
                                                                                    css 复制代码
action creators--------------------------->store--------------------------------->reducers
作用:                                  作用:
  用来创建action                         用来集中性的管理状态数据
  的工厂函数模式模块                      身上有读取/更新状态的方法
{
  type: 状态名称                                                - store.getState()
  data:更新的数据                        - store.dispatch(action)                    交给store对
}                                                  - store.subscribe(listener)

                                组件:  store.subscribe(listener)        store更新状态数据
```

- 理解流程:

    - 见redux流程图

- 组件从redux中读取状态: 调用action函数生成action store.getState()

- 组件从redux中更新数据

    1. action-creators创建action函数
        - 用来创建action对象的工厂函数模块
            - 分为同步和异步:
                - 同步:返回action对象
                - 异步:返回一个函数

```yaml
action对象:
{
  type: 操作类型,
  data: 操作数据
}
```

2. 定义store对象 `const store = redux.createStore()`
3. 定义reducers函数
4. 调用action creator创建新的state对象（组件）（组件）
5. 调用store.dispatch(action)更新数据（组件）
6. 自动调用reducers,reducer根据(preState,aciton)，生成newState并传入store（redux）
7. store中数据自动更新（redux）
8. 调用store.subscribe(()=>{})方法拿到store中的数据重新渲染数据到页面（组件）

- redux组件使用
  - 定义action-creators/store/reducers
  - 组件
    - 引入store，会默认调用reducer，reducer的返回值就是状态的初始值: reducer中的preState
      - 所以需要在reducer中做状态初始化，不然默认就是undefined
    - 引入action creators生成action对象，并传入新的状态数据
      - const action = actionFn(newStateVal) //返回action对象
    - 调用store.dispatch(action)触发更新
      - （reducer会根据action中的状态数据生成新的newState给Store）
  - 调用sunscribe方法重新渲染组件
    - 在最外面的index.js定义的渲染组件的位置引入store对象

```javascript
store.subscribe(()=>{
  //一旦store对象状态发生改变,就会触发当前函数
  //触发当前函数,重新渲染组件
  ReactDOM.render(<App />,document.getElementById('root'))
})
```

Object.fromEntries新增Api

Dva

类似于Redux，不过很多项目都是使用Dva进行复杂数据的存储。使得结构更加清晰明朗。大公司有的会自己基于Dva进行封装自己的组件通信的库，比如kos。

通过Context通信

同react.createContext

react-router

SPA技术： 局部更新，网址变

路由两种模式

1. hash
    1. 缺点1：路径比较丑
    2. 缺点2：会导致锚点失效，锚点到#后面找到，但是hash模式#后面没有
    3. 优点：出现比history早，兼容性比history好
2. history

路由的分类

后端路由 key function 前端路由 key component

ReactDOM

ReactDOM.createPotral

- Portal 提供了一种将子节点渲染到存在于父组件以外的 DOM 节点的优秀的方案。
- ReactDOM.createPortal(child, container)
- 一般用于定义Modal组件，将Modal插入到页面一个容器中

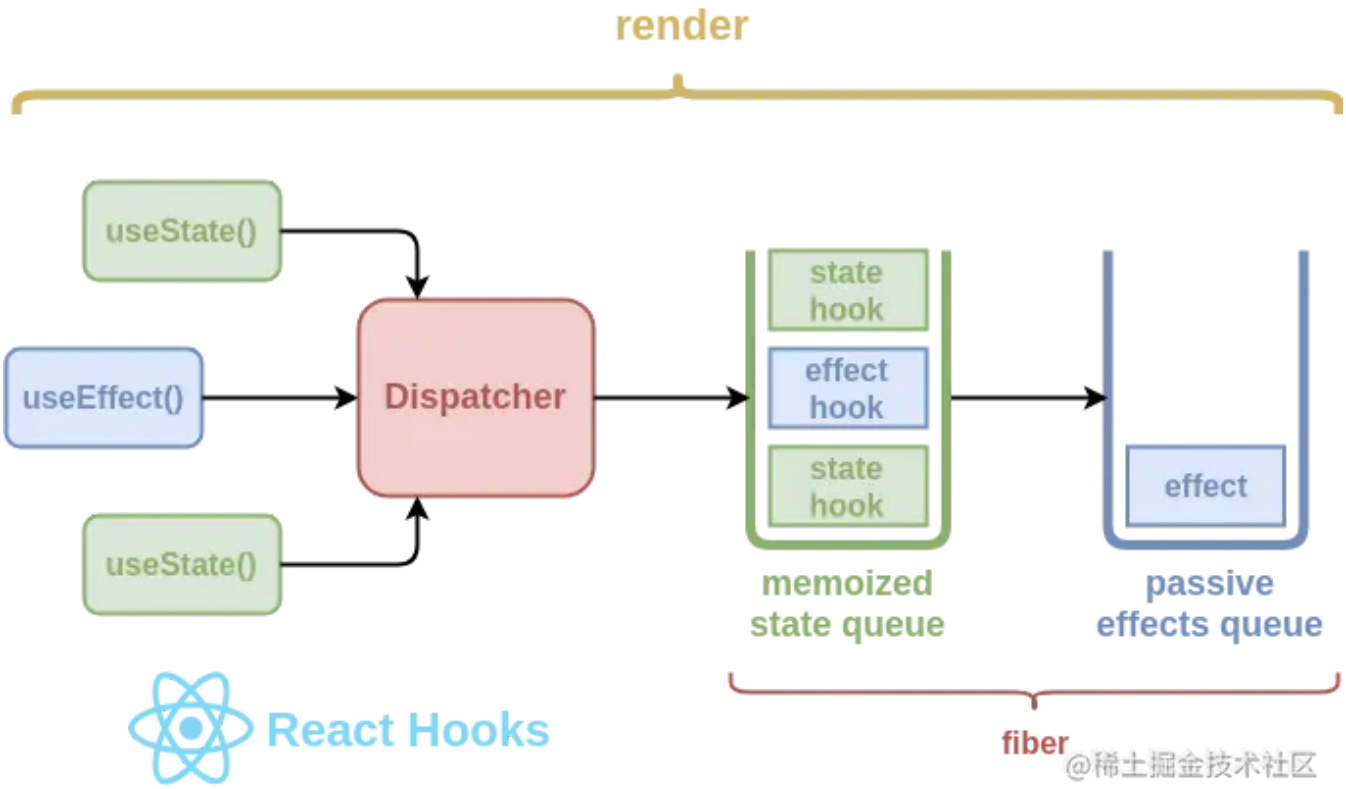
用法：ReactDOM.createPotral(child,reactNode) 将子元素child插入指定的DOM节点中 示例：将有条件的将不同的child插入指定的元素

javascript 复制代码

```
if (mode == 'single' && !!editable) {
  return ReactDOM.createPortal(this.renderEditFooter(), this.element);
} else if (mode == 'single' && !editable) {
  return ReactDOM.createPortal(this.renderDetailFooter(), this.element);
} else if (mode === 'multiple-footer') {
  return ReactDOM.createPortal(this.renderMultilFooter(), this.element);
}
```

React hook

Hook 是 React 16.8 的新增特性。



使用hook的原因

- 庞大的组件嵌套问题：（是因为逻辑分散到了不同的生命周期中）
- 解决：高阶函数/渲染属性render Props
- 造成组件嵌套地狱
- 逻辑复用，生命周期中的代码重复和冗余
- class组件对学习的难度有要求，并且对机器的要求也比较高
- class使可靠的热加载变得困难
- class组件使编译器优化变得困难
- hard for humans
- hard for mechines

hook的使用优势：

- 需要使用小型，简单，更轻量的方法来添加state或生命周期替代class； Hook是react提供的函数，代码更加偏平；

- 传统class类组件，进行一个操作，初始化需要在componentDidMount中写一遍，更新的时候需要在componentDidUpdate中写一遍，使用hook只需要使用useEffect即可。

使用hook的注意事项：

- 不能在组件的条件判断中使用hook，只能在组件的顶部写hook
  - 如果在函数组件中，遇到需要有条件的渲染 ，当value.length < 2 时渲染让a=3进行渲染，不能直接在useEffect中使用useState进行修改，而是需要将hook放在一个方法中单独定义和修改，避免有条件的渲染，ts编译也会报错`

0.useState

- 只有一个参数
  - 可以传递一个函数，如果修改的state依赖原先的state的时候可以传递一个函数，参数为原先的state，修改后的执为参数返回值
  - 传递需要修改的值
- 优点：如果修改的state和原先的state相同(通过Object.is方法比较)则不会渲染子组件和调用useEffect方法

1.useEffect

在初始化渲染和每次更新都会执行；操作具有一致性；

返回一个函数操作一些清除组件的时候做的事情，等同于componentUnmount；

分离代码不是基于生命周期，而是基于这段代码是做什么的，所以可以使用不同的useEffect去操作；

每个hook都是独立的，因为我们依赖hook调用的顺序

Custom hook：更灵活的创建抽象函数的功能。

---

2. useContext

1. Context Hook可以在函数组件中读取到保存在Context对象中的value值数据 简化了祖孙组件通信
2. 语法: a. 创建Context容器对象: const MyContext = React.createContext(defaultValue) b. 通过Provider提供value: <MyContext.Provider value={count}> c. 后代组件读取value: const value = React.useContext(MyContext)

3.useCallback

```ini 复制代码
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

- 返回：一个被缓存的回调函数
- useCallback依赖项数组不会作为参数传递给回调函数
- 场景：执行副作用，
- 功能：类似shouldComponentUpdate，避免不必要的渲染，根据依赖项去决定是否渲染
  - useCallback(fn, deps) 相当于 useMemo(() => fn, deps)。

4.useMemo

```scss 复制代码
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- 返回一个被缓存的值
- 功能：同useCallback，是性能优化的hook
- 场景：执行高消耗的计算，返回被缓存的值
- 不同于useCallback hook的是，useMemo依赖项数组是作为参数传递给回调函数的。

5. useRef

- 函数组件使用useRef获取DOM节点
  - useRef获取node节点
    - ref是一个对象时并不会获取到变化，只能获取当前从状态
  - callback ref

    - 可以接收一个变化

```ini
function App(){
        const InputRef = useRef();
        retunr (<input ref={InputRef}>)
}
```

- class组件使用React.createRef获取DOM节点

```scala
class App extends React.Component {
        const this.InputRef = React.createRef();
        render(){
                return <input ref={this.InputRef}/>
        }
}
```

6.useImperativeHandle

- 与forwardRef一起使用
- class组件使用React.createRef获取DOM节点
- 函数组件使用useRef获取DOM节点
    - 父组件使用useRef拿到子组件的DOM节点
    - 子组件使用forwardRef和useImperativeHandle方法给子组件自己的ref上添加方法和命令，让父组件可以通过获取的ref直接使用。

计数组件，父组件重置，子组件增加数值

```javascript
//父组件
function App(){
        const counterRef = useRef();
        const handleReset = ()=>{
                //可以从ref上拿到子组件的handleReset方法了；
                counterRef.current.handleReset();
        }
        return (
                <button onClick={handleReset}>重置</button>
                <Counter ref={counterRef} />
```

```
 )
}
```

scss 复制代码

```
//Counter子组件
function Counter (props,ref){
        const [num,setNum]=useState(0);
        const handleReset = ()=>{setNum(0);}
        useImperativeHandle(ref,()=>({
                handleReset:handleReset,
        }))
        return (
                {num}
                <button onClick={()=>(num)=>num+1}>+</button>
        );
};

export default forwardRef(Counter);
```

7. 自定义hooks

自定义封装一个useState返回数组的第二个设置函数的元素；以‘set’开头，将公共逻辑抽取出来。类似于class组件使用render props或高阶函数一样。

8. Hook 规则

复制代码

```
1). 只在函数组件的最顶层使用 Hook
2). 不要在循环，条件或嵌套函数中调用 Hook
3). 必须保证使用hook的顺序与个数总是不变的
```

面试题

- 有没有使用过 react hooks，它带来了那些便利?
  - React hooks 主要解决了状态以及副作用难以复用的场景，除此之外，他对我最大的好处就是在 Console 中不会看到重重叠叠相同名字的组件了(HOC)。

React API方法

React.createContext

作用订阅了context对象的组件会从最近的Provider中获取context值； 默认value值：React.createContext(defaultValue) 自定义value值：Provider 组件传入自定义value值；value值发生改变，销售组件就会被渲染，不受shouldComponentUpdate限制

javascript 复制代码

```
import {createContext} from 'react';
export default createContext("test")  // ./context 传入defaultValue
import TestContext from './context';
<TestContext.Provider value={/* 某个值 */}>
```

forwardRef

功能：创建一个组件,将接收的ref属性转发到组件树下的组件，将ref向下传递。 作用：复用代码 参考：forwardRef

React组件

两种绑定事件的写法

- 如果在元素上绑定事件使用普通函数,则需要指定this
- 如果在元素上绑定箭头函数，则不需要指定this

javascript 复制代码

```
<Button onClick={this.handleClick1.bind(this)}>需要指定</Button>
<Button onClick={this.handleClick2}>不需要指定</Button>

handleClick1(){};
const handleClick1 = ()=>{}
```

event

javascript 复制代码

```
<Button onClick={this.handleClick1.bind(this, "1", "2")}>
    普通函数，需要指定this，第三个参数为event
```

```
</Button>
<Button
    onClick={(event) => {
        this.handleClick2(1, 3, event);
     }}
 >
     通过箭头函数传参，获取event
 </Button>

handleClick1(id1,id2,event){
 console.log(id1,id2,event)//最后一个参数为event
};
handleClick2(id1,id2,event){
 console.log(id1,id2,event)//最后一个参数为event
};
```

HOC高阶组件

- 概念：高阶组件就是一个函数，且该函数接受一个组件作为参数，并返回一个新的组件
- 目的: 能让其他组件复用相同的逻辑
- 当你发现两个组件有重复逻辑时，就使用HOC来解决。 相当于函数式编程，实际就是一个函数，接收一个组件作为函数参数，函数体对传入组件

结合new Map对组件进行复用

javascript 复制代码

```
/*
   Map：存储键值对
   键可以是任意值
   性能：在频繁增删键值对的场景下表现更好。
    */

   /*
   // 使用常规的Map构造函数可以将一个二维键值对数组转换成一个Map对象
  let kvArray = [["key1","value1"],["key2","value2"]]
   let myMap = new Map(kvArray);
   myMap.get("key1")
    */

   /*
   可以进行数组合并并去重
   let myMap1 =  new Map([[1:"one"],[2:"two"]])
   let myMap1 =  new Map([[2:"two2"],[3:"three"]])
   let myMap = new Map([...myMap1,...myMap2])
    */
```

```
    // const { get } = _;
    const Mymap = new Map([
      ['modal', {
        // Component: ShopSelector,
        // key: (label: any) => `${label.value}`,
        initialValue: {},
        initProps: { editable: false },
      }],
      ['input', {
        // Component: ShopSelector,
        // key: (label: any) => `${label.value}`,
        initialValue: {},
        initProps: { editable: false },
      }],
      ['div', {
        // Component: ShopSelector,
        // key: (label: any) => `${label.value}`,
        initialValue: {},
        initProps: { editable: false },
      }],
    ]);
    const myObject = { a: "a", b: "b", c: "c" };
    for (const key in myObject) {
      if (myObject.hasOwnProperty(key)) {
        const element = myObject[key];
        console.log(element);
      }
    }
    console.log('Mymap', Mymap);
    const b = Mymap.get('modal');
    console.log('get', b);
```

react.createElement创建复合型组件:复用组件

```
import Atest1 from ‘./components/Atest1’;
import Btest1 from ‘./components/Btest1’;
import Ctest2 from ‘./components/Ctest2’;
import Dtest2 from ‘./components/Dtest2’;
import Dtest2 from ‘./components/Dtest2’;
import Etest3 from ‘./components/Etest3’;

const FormMap: any = {
  ’A-test1’: Atest1,
  ‘B-test1’: Btest1,
  ‘C-test2’: Ctest2,
  ‘D-test2’: Dtest2,
  ‘D-test2’: Dtest2,
  ‘E-test3’: Etest3,
};
```

```
render(){
    return (
      <TextContext.Provider value={{ ref: this.Ref }}>
          {React.createElement(FormMap[voucherType], {
                value=this.props.value
          })}
      </TextContext.Provider>
    )
}
```

---

ajax请求

1. axios库
    1. 用法
        1. get请求 `axios.get('/user', { params: { ID: 12345 } }) .then(function (response) { console.log(response); }) .catch(function (error) { console.log(error); }) .finally(function () { // always executed });`
        2. post请求 `axios.post('/user', { firstName: 'Fred', lastName: 'Flintstone' }) .then(function (response) { console.log(response); }) .catch(function (error) { console.log(error); });`
2. fetch:

- 原生
- 没有跨域问题
- 有兼容性问题:

React扩展

Fragment组件： 充当根标签，渲染时不会渲染任何东西 error-boundary报错处理:(生产环境有效) 处理生命周期中的报错 报错的组件报错;其他组件正常加载 代码分割:(生产环境有效) 功能组件分割

- 因为只有两个主文件,所以所有功能组件都在这两个文件中,并请求的时候加载全部
- 代码分割: 按需加载组件 import关键字 懒加载:两种方法 React.lazy(新) Loadable:(旧) 支持服务端渲染

- 应用项目: 只对路由组件进行懒加载 有两个js文件公共代码会抽取出来一个新的js文件 提升首页加载速度 React性能优化: 只能对比一层状态 PureCompent

Object.is方法,解决NaN不与NaN相等 解决0 和 -0相等的问题 项目构建: 优化打包

跨域: 服务器代码方式 请求发送到3000代理服务器

- 代理服务器(3000)将请求转发到目标服务器(5000)上 package.json加上proxy,则启动服务器 请求地址改成3000

反向代理: nginx服务器代理

react动态加载

React性能优化

preformance optimization

- 性能优化主要解决什么问题?
    - 避免不必要的组件渲染和重拍重绘
    - react会产生多余的组件渲染
    - 父组件传递给子组件是对象， 即使对象没有发生改变，父组件被渲染了，子组件也会被渲染，造成不必要的渲染。原因是：对象字面量父组件每次都会重新创建，对于子组件而言每次的props就都是不一样的所以会被更新。
- 是哪些代码导致react组件被重新渲染了?

参考：React性能问题之多余的组件渲染及检查工具

- 性能优化的方法?
    - 主要集中在shouldComponentUpdate这个钩子上：对前后的state进行浅比较，不建议深比较，因为组件渲染可能比较频繁，深比较对性能开销很大；
    - 使用immutable或immer库生成不可变对象，既能完整对比当前state和之前state是否相同，并且不影响性能；

使用-生命周期-进行性能优化

shouldComponentUpdate生命周期:

- 子组件需要优化的时候才会使用
- 控制当前props和nextProps的变化才更新

javascript 复制代码

```
public shouldComponentUpdate(nextProps: any) {
  if (this.props.id !== nextProps.id ) {
    return true;
  }
  return false;
}
```

componentWillUnmount

当组件从DOM中移除的时候方法会被调用。移除的时候将大数据量的属性删除。使用了setTimeout和addEventListeners之后需要在该生命周期进行对应的销毁工作。

javascript 复制代码

```
public componentWillUnmount() {
  for (const key in window['__test']) {
    delete window['__test'][key];
  }
}
```

componentWillReceiveProps

- 在16的版本中被废弃了，使用getDerivedStateFromProps生命周期替代，因为该生命周期会被重复调用多次。
- 在使用中任然可以用作对比props，如果相等则不更新组件。如果props不一致做一些副作用的操作。
- 同shouldComponentUpdate，在SCU之前，且SCU做不到时，就阻止掉更新或操作副作用。

javascript 复制代码

```
componentWillReceiveProps(nextProps: any) {
    if (_.isEqual(nextProps.value, this.props.value)) return;
    setTimeout(() => {
      this.initValues(nextProps);
```

```
    }, delayCount);
  }
```

useCallback

useCallback(fn, deps)等同于useMemo(() => fn, deps)

javascript 复制代码

```javascript
useMemo(()=>{
//...
},a===b)
```

React.memo(Component,Fn)

- 作用：相当于React.PureComponent，对组件的props和state进行浅比较，如果相同则不做无意义的子组件渲染。
- 进行性能优化,不做多余的渲染
- 适用于函数组件
- 有两个参数，第一个为组件，第二个为对比函数

javascript 复制代码

```javascript
React.memo(
  (props: any) => {},
)
```

案例

typescript 复制代码

```typescript
//子组件
const BigListView = (props: any) => {
    const { items } = props;
    console.log("BigListView render on " + Date.now());
    return (
      <ul>
        {items.map((item: string) => (
          <li key={item}>{item}</li>
        ))}
      </ul>
    );
  };
 export default BigListView;
```

javascript 复制代码

```javascript
//父组件
import BigListView from "./components/big-list-view";

// 虚构的列表数据
const initItems = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

function App() {
  const [items, setItems] = useState(initItems);
  const onUpdateList = () => {
    setItems([...items, Math.random() * 100]); // 每次增加一个随机的列表数据
  };

  const [counter, setCounter] = useState(0);
  const onUpdateCounter = () => {
    setCounter(counter + 1); // 给计数器加 1
  };

  return (
    <div>
      {/* counter改变BigListView也会被渲染？？  */}
      <button onClick={onUpdateCounter}>Update Counter</button>
      <div>{counter}</div>

      <br />

      <button onClick={onUpdateList}>Update List</button>
      <BigListView items={items} />
    </div>
  );
}
```

- 问题是：counter变化的时候不需要BigListVie被渲染，但是每次更新都会被渲染。如果是复杂功能组件，会导致性能问题。
- 解决使用React.memo(组件,对比函数)优化

typescript 复制代码

```typescript
//使用React的memo后，counter改变BigListView组件不会被渲染。
const BigListView = (props: any) => {
    const { items } = props;
    console.log("BigListView render on " + Date.now());
    return (
      <ul>
        {items.map((item: string) => (
          <li key={item}>{item}</li>
        ))}
      </ul>
    );
```

```
};
export default React.memo(BigListView);
```

合理使用异步组件

React.Lazy React.Suspense

webpack层面优化

前端通用是能优化，如图片懒加载

使用SSR

- ReactDOMServer.renderToString(element) 将React元素渲染HTML字符串
- next框架 -- create-next-app

> 以上所有内容都可以去 React官网 查阅，

其他开发优化细节

- 不要使用深拷贝进行操作，会大大降低性能

大列表渲染:React-window

MVVM和MVC

MVVM的精髓是：通过ViewModal将试图中的状态和用户行为分离出一个抽象。 MVC：用户输入时，通过控制器更新Modal数据库，并通知更新试图。这样控制器的责任太大，无法适用复杂场景，不利于维护。

React组件渲染那些事

React不会渲染的问题；

- 1: 当调用setState时，react不会渲染 原因一： 错误代码 const {period} = this.state; let cloneValue = period; ....操作了cloneValue this.setState({period:cloneValue}) 当period是对象，或者数组的时候，因为只是改变了数据的元素并没有修改其地址值，所以react认为没有修改会导致不会渲染

UI 组件库

- material-ui.com(国外)
- ant-desigin(国内)
- 引入组件和样式

其他知识点

window

1. 将需要通信的大数据存在window。以key和value的形式存储
2. 不要对大数据进行cloneDeep和遍历等等消耗性能的操作
3. 对大数据进行一次处理和传递（求最低价格的时候，在子组件中计算好了之后传递给父组件，父组件避免再次计算和传递）
4. 给每个大数据一个id/key进行过滤，检测等操作，减少对每个大数据的操作次数和频率
5. 方法：使用lodash get/set往对象中存储和获取数据

前端路由原理

前端路由的原理? 监听URL的变化，然后匹配路由规则，显示相应的页面，并且无需刷新页面。 **前端路由两种实现方式?**

- **Hash模式**
    - 比如这个URL：www.abc.com/#/hello, hash 的值为#/hello。
    - 只识别hash前面的内容，只有第一次才会发送请求，之后hash变化不会发送请求
    - 因为hash虽然出现在url中，但是不会包含在http请求中，对后端没有影响，所以hash改变不会造成页面刷新

- 且后端也不需要对路由进行全覆盖也不会返回404错误

- **History模式**

  - 利用了HTML5中新增的history.pushState/history.replaceState两个API
  - 对历史记录进行修改的功能，当执行修改时，虽然发生了url改变，但是不会立马向后盾发送请求
  - 所以前端的url必须和后端发起请求的url一致，www.book.com/book/id,如果后...
  - 问题是:手动刷新或通过url进入页面服务端无法识别这个url
  - 因为是单页面应用只有一个url，服务端在识别其他url的时候会出现404
  - 所以需要在服务端配置，如果url匹配不到任何静态资源，应该返回单页面应用的html文件

组件封装

组件封装

```
|---  package.json
|---  lib                   // 组件编译存放目录
|---  document              // 组件源码
    |---  demos             // demo存放目录
        |---  base.md       // demo示例
    |---  api.md            // api文档
|---  src                   // 组件源码
    |---  assets            // 组件依赖的图片资源
    |---  style             // 组件样式存放目录
    |---  index.jsx         // 组件入口文件
    |---  logo.png          // 组件缩略图
```

- api.md（api文档） title: SearchInput subtitle: 搜索输入框 cols: 2
- base.md (组件demo) order: 1 title: 基本 demoType: '' demoUrl: ''