回到顶部

react diff算法

==传统diff==:

diff算法即差异查找算法，对于Html Dom结构即为tree的差异查找算法，而对于计算两棵树差异时间复杂度为O(n*3)，显然成本太高，react不可能采用这种传统算法；

==React Diff==:

React采用虚拟Dom技术实现对真实Dom的映射，即React Diff算法的差异查找实质是对两个JavaScript对象的差异查找；只有在React更新阶段才有Diff算法的应用；事实上，Diff算法只被调用于React更新阶段的Dom元素更新过程，因为如果为更新文本类型，内容不同就直接更新替换，并不会调用复杂的diff算法，在自定义组件中，自定义组件最后结合React Diff优化。策略（不同的组件具备不同的结构）

==Diff策略==:

DOM节点跨层级的操作特别少，可以忽略不计 拥有相同类的两个组件将会生成相似的树形结构，拥有不同类的两个组件将会生成不同的树形结构 同一层级的一组子节点，他们可以通过uuild进行区分

==对于Diff的开发建议：==

==基于tree diff：==
　　开发组件时，注意保持Dom结构的稳定；即，尽可能少的动态操作Dom结构，尤其是移动操作。当节点过大或者页面更新次数过多时，页面卡顿的现象会比较明显。 这时可以通过css隐藏或显示节点，而不是真的移除或添加DOM节点。

==基于component diff： ==　　基于shouleComponentUpdate()来减少组件不必要的更新 对于类似的结构应该尽量封装成组件，减少代码量，又能减少component Diff的性能消耗

==基于element diff：==

对于列表结构，尽量较少类似将最后一个节点移动到列表首部的操作，当节点数量过大或更 新操作过于频繁时，在一定程度上会影响React的渲染性能。

- 回到顶部

react 生命周期

==组件初始化阶段:==

这个阶段没有具体的生命周期函数，在类组件里面，继承了react component，才有了render函数，生命周期才可以使用，这就说明函数组件为什么不能使用这些方法的原因。

==挂载阶段：==

componentWillMount：在第一次渲染时的第一个运行的生命周期，可以修改state，无法判断传入参数，可以修改props，修改了props以后重新运行生命周期。

render：可以修改state和props，需要重新执行生命周期。

conponentDidMount：在第一次渲染时，渲染结束运行的生命周期，修改state很安全，可以修改props，重新运行生命周期。

==更新阶段：==

componentWillReceiveProps：在运行props时运行的生命周期，可以修改state或者props。如果修改state以后，之后的生命周期传入的state参数会更新。修改props在这个生命周期里面很危险，会重新运行一遍生命周期。

shouldComponentUpdate：在更新props或者state时运行的生命周期，返回true或者fase，一旦返回的是false，接下来的生命周期都不再运行。这里可以修改props或者state，但是很危险,都是重新运行一遍生命周期。

componentWillUpdate：在修改props或者state即将渲染时运行的生命周期。可以修改state或者props，但是很危险,都是重新运行一遍生命周期。

render

componentDidUpdate：在修改props或者state渲染完render运行的生命周期。可以修改state或者props，但是很危险,都是重新运行一遍生命周期。

==卸载阶段：==

componentWillUnmount：卸载

==新的生命周期：==

getDerivedStateFromProps:

无论什么情况下都是第一个运行的生命周期。这个生命周期是类的静态方法，这个函数返回一个对象(必须得是对象或者数组)来更新state的数据。更新当前的state和新的即将要更新的新state。不能访问实例，所以不能修改state和props。但是也可以给全局增加一个变量，在componentDidMount的时候赋值实例。这样子可以修改props和state，但是很危险。这个生命周期在第一次渲染时代替了componentWillMount。在修改state代替了componentWillUpdate。在修改props时代替了componentWillReceiveProps和componentWillUpdate生命周期。

getSnapshotBeforeUpdate：

在更新props或者state时，在rander渲染完之后在componentDidUpdate之前时运行这个生命周期。作用是返回一个数据，作为componentDidUpdate的第三个参数传入。可以修改state和props，但是很危险的操作，都是重新运行一遍生命周期。更新state或者props时，在render和componentDidUpdate之间加了这个生命周期。

componentDidCatch：抓捕错误的生命周期

- 回到顶部

为什么使用虚拟dom?

优点:

1. 保证性能下限: 虚拟DOM可以经过diff找出最小差异,然后批量进行patch,这种操作虽然比不上手动优化,但是比起粗暴的DOM操作性能要好很多,因此虚拟DOM可以保证性能下限

2. 无需手动操作DOM: 虚拟DOM的diff和patch都是在一次更新中自动进行的,我们无需手动操作DOM,极大提高开发效率 跨平台: 虚拟DOM本质上是JavaScript对象,而DOM与平台强相关,相比之下虚拟DOM可以进行更方便地跨平台操作,例如服务器渲染、移动端开发等等

缺点:

无法进行极致优化: 在一些性能要求极高的应用中虚拟DOM无法进行针对性的极致优化,比如VScode采用直接手动操作DOM的方式进行极端的性能优化

- 回到顶部

虚拟dom实现原理:

- 虚拟DOM本质上是JavaScript对象,是对真实DOM的抽象;
- 状态变更时，记录新树和旧树的差异
- 最后把差异更新到真正的dom中
- 回到顶部

react合成事件:

采用事件冒泡的形式冒泡到document上面，然后React将事件封装给正式的函数处理运行和处理。围绕浏览器原生事件充当跨浏览器包装器的对象。它们将不同浏览器的行为合并为一个 API。这样做是为了确保事件在不同浏览器中显示一致的属性。

- 回到顶部

什么是高阶组件（HOC）:

高阶组件是重用组件逻辑的高级方法，是一种源于 React 的组件模式。 HOC 是自定义组件，在它之内包含另一个组件。它们可以接受子组件提供的任何动态，但不会修改或复制其输入组件中的任何行为。你可以认为 HOC 是“纯（Pure）”组件。
可以用于:

- 代码重用，逻辑和引导抽象
- 渲染劫持
- 状态抽象和控制
- Props 控制
- 回到顶部

react中key的重要性:

key 用于识别唯一的 Virtual DOM 元素及其驱动 UI 的相应数据。它们通过回收 DOM 中当前所有的元素来帮助 React 优化渲染。在虚拟dom节点中赋予key值，会更加快速的拿到需要的目标节点，不会造成就地复用的情况，对于节点的把控更加精准。

- 回到顶部

react-hook

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。

```
import React ,{useState} from 'react'
```

useState: 函数组件里面没有state，所以我们使用useState，只接受一个参数，就是该state属性的初始值，它会返回一个数组，里面包含两个值，第一个值是初始值，第二个用来更改初始值。

useEffect: 函数式组件没有生命周期，使用useEffect来替代componentDidMount和componentDidUpdate。有两个参数，第一个是一个回调函数，第二个是空数组。如果第二个数组为空数组，就会在初始化执行完成之后，执行一次，相当于componentDidMount，数组有内容，会根据数组内容改变去执行前面的回调函数。相当于componentDidUpdate生命周期。
userContext：组件之间共享状 https://blog.csdn.net/weixin_43606158/article/details/100750602
useReducer：相当于简单的redux。接收两个参数，第一个参数是一个回调函数，里面接收一个state数据，以及action，通过dispatch来更改内容。第二个参数是一个初始值。

```
const initialState = {
  count1: 0,
  count2: 0,
};
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment1':
      return { ...state, count1: state.count1 + 1 };
    case 'decrement1':
      return { ...state, count1: state.count1 - 1 };
    case 'set1':
      return { ...state, count1: action.count };
    case 'increment2':
      return { ...state, count2: state.count2 + 1 };
    case 'decrement2':
      return { ...state, count2: state.count2 - 1 };
    case 'set2':
      return { ...state, count2: action.count };
    default:
      throw new Error('Unexpected action');
  }
```

```
const Example02 = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      <div>
        {state.count1}
        <button onClick={() => dispatch({ type: 'increment1' })}>+1</button>
        <button onClick={() => dispatch({ type: 'decrement1' })}>-1</button>
        <button onClick={() => dispatch({ type: 'set1', count: 0
})}>reset</button>
      </div>
      <div>
        {state.count2}
        <button onClick={() => dispatch({ type: 'increment2' })}>+1</button>
        <button onClick={() => dispatch({ type: 'decrement2' })}>-1</button>
        <button onClick={() => dispatch({ type: 'set2', count: 0
})}>reset</button>
      </div>
    </>
  );
```

==useCallback==：传入两个参数，第一个是回调函数，第二个是依赖，这个回调函数只在某个依赖改变时才会更新。
==useMemo==：可以优化用以优化每次渲染的耗时工作。
==useRef==：它可以用来获取组件实例对象或者是DOM对象，来跨越渲染周期存储数据，而且对它修改也不会引起组件渲染。

- 回到顶部

redux

state：组件内部状态
action：组件动作，相应的改变组件内部的状态值
dispatch：发出相应的动作
单一事件源，状态只读，使用纯函数进行更改

redux中提供createStore方法用于生成一个store对象，这个函数接收一个初始值state和一个reducer函数，当用户发出相应的action时，利用传入的reducer函数计算一个新的state值，并返回。当存在多个reducer，分别管理不同的state，需要将其合并成一个reducer，我们使用combinReducers函数。

react-redux提供了一个Provider组件，以及connect方法，Provider组件作为上层组件，需要将store作为参数注入到组件中，此后在子组件中都可以访问到store这个对象，connect方法接收两个参数：mapStateToProps，actionCreators，并返回处理后的组件，其中mapStateToProps可以将对应的state作为props注入对应的子组件，actionCreator可以将对应的actioncreator作为prop注入对应的子组件。

- 回到顶部

redux中间件

中间件提供第三方插件的模式，自定义拦截 action -> reducer 的过程。变为 action -> middlewares -> reducer 。这种机制可以让我们改变数据流，实现如异步 action ， action 过滤，日志输出，异常报告等功能。

- 为什么要用redux

在React中，数据在组件中是单向流动的，数据从一个方向父组件流向子组件（通过props）,所以，两个非父子组件之间通信就相对麻烦，redux的出现就是为了解决state里面的数据问题

- Redux设计理念

Redux是将整个应用状态存储到一个地方上称为store,里面保存着一个状态树store tree,组件可以派发(dispatch)行为(action)给store,而不是直接通知其他组件，组件内部通过订阅store中的状态state来刷新自己的视图。

- Redux三大原则

唯一数据源

保持只读状态

数据改变只能通过纯函数来执行

- Redux源码

```
let createStore = (reducer) => {
    let state;
    //获取状态对象
    //存放所有的监听函数
    let listeners = [];
```

```
    let getState = () => state;
    //提供一个方法供外部调用派发action
    let dispath = (action) => {
        //调用管理员reducer得到新的state
        state = reducer(state, action);
        //执行所有的监听函数
        listeners.forEach((l) => l())
    }
    //订阅状态变化事件，当状态改变发生之后执行监听函数
    let subscribe = (listener) => {
        listeners.push(listener);
    }
    dispath();
    return {
        getState,
        dispath,
        subscribe
    }
}
let combineReducers=(renducers)=>{
    //传入一个renducers管理组，返回的是一个renducer
    return function(state={},action={}){
        let newState={};
        for(var attr in renducers){
            newState[attr]=renducers[attr](state[attr],action)

        }
        return newState;
    }
}
export {createStore,combineReducers};
```

常见的中间件：

redux-logger：提供日志输出
redux-thunk：处理异步操作
redux-promise：处理异步操作 actionCreator的返回值是promise

- 回到顶部

redux优缺点：

1.一个组件所需要的数据，必须由父组件传过来，而不能像flux中直接从store取。
2.当一个组件相关数据更新时，即使父组件不需要用到这个组件，父组件还是会重新render，可能会有效率影响，或者需要写复杂的shouldComponentUpdate进行判断。

- 回到顶部

简述flux思想

Flux 的最大特点，就是数据的"单向流动"
1.用户访问 View
2.View 发出用户的 Action
3.Dispatcher 收到 Action，要求 Store 进行相应的更新
4.Store 更新后，发出一个"change"事件
5.View 收到"change"事件后，更新页面

- 回到顶部

redux-saga：

redux-saga的使用主要是添加监听，用于处理自定义的异步处理请求，在将结果调用put方法发起action，从而交给reducer方法处理，使得redux异步处理更灵活。

saga需要一个全局监听器（watcher saga），用于监听组件发出的action，将监听到的action转发给对应的接收器（worker saga），再由接收器执行具体任务，任务执行完后，再发出另一个action交由reducer修改state，，所以这里必须注意：watcher saga监听的action和对应worker saga中发出的action不能是同一个，否则造成死循环。在saga中，全局监听器和接收器都使用Generator函数和saga自身的一些辅助函数实现对整个流程的管控

- 回到顶部

redux-thunk：

thunk采用的是扩展action的方式：使得redux的store能dispatch的内容从普通对象扩展到函数

将普通的对象换成函数，在函数里处理异步逻辑，普通的对象有哪些属性，表明这个action的type,payload(携带的数据)

- 回到顶部

组件通信：

- 父组件向子组件通讯: 父组件可以向子组件通过传 props 的方式，向子组件进行通讯
- 子组件向父组件通讯: props+回调的方式,父组件向子组件传递props进行通讯，此props为作用域为父组件自身的函数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中。
- pubsub可以采用发布订阅的方式实现组件间的传值。

```
import Pubsub from 'pubsub-js'
Pubsub.publish('username',val)//发布
Pubsub.subscribe('username',(msg,data)=>{}//data是你要拿回来的数据。
```

- 兄弟组件通信: 找到这两个兄弟节点共同的父节点,结合上面两种方式由父节点转发信息进行通信
- 跨层级通信: Context设计目的是为了共享那些对于一个组件树而言是“全局”的数据，例如当前认证的用户、主题或首选语言,对于跨越多层的全局数据通过Context通信再适合不过。 context使用了Provider和Customer模式，和react-redux的模式非常像。在顶层的Provider中传入value， 在子孙级的Customer中获取该值
- 发布订阅模式: 发布者发布事件，订阅者监听事件并做出反应,我们可以通过引入event模块进行通信
- 全局状态管理工具: 借助Redux或者Mobx等全局状态管理工具进行通信,这种工具会维护一个全局状态中心Store,并根据不同的事件产生新的状态

- 回到顶部

React-router里面的hash-router和broswer-router的区别

hashrouter是以#号方式匹配路由，从url中可以看出来，这个地址对于后端来说，全部指向同一个地址而browserrouter不存在#的，不同的路由对于后端也是不同的地址 当你需要做同步渲染的时候，肯定是要用browserrouter的

- 回到顶部

为什么虚拟dom会提高性能:

虚拟dom相当于在js的真实dom中间加了一个缓存,利用dom diff算法避免了没有必要的dom操作,从而提高性能。

- 回到顶部

虚拟dom和真实dom区别:

1.虚拟dom不会进行排版与重绘操作
2.虚拟dom进行频繁修改,然后一次性比较并修改真实dom中需要改的部分,最后在真实dom中进行排版与重绘
3.真实dom频繁修改效率极低.
4.虚拟dom有效降低大面积的重绘和排版

- 回到顶部

mobx

MobX是响应式编程，实现状态的存储和管理。使用MobX将应用变成响应式可归纳为三部曲：

- 定义状态并使其可观察
- 创建视图以响应状态的变化
- 更改状态

1. observable是将类属性等进行标记，实现对其的观察。三部曲中的第一曲，就是通过Observable实现的。
2. 通过action改变state。三部曲中的第一曲通过action创建一个动作。action函数是对传入的function进行一次包装，使得function中的observable对象的变化能够被观察到，从而触发相应的衍生。 3.

mobx api

computed

。计算值(computed values)是可以根据现有的状态或其它计算值衍生出的值。简单理解为对可观察数据做出的反应，多个可观察属性进行处理，然后返回一个可观察属性
使用方式：1、作为普通函数，2、作为decorator

```
import {observable} form 'mobx'
class Store{
    @observable arr = [];
    @observable obj = {};
    @observable mao = new Map();

    @observable num = 1;
    @observable str = 'str';
    @observable bool = true;

    // 2. 作为decorator
    @computed get mixed(){
        return store.str + '/'+ store.num
    }

}

const store = new Store();

// 1. 作为普通函数
let foo = computed(function(){
    return store.str + '/'+ store.num
})
```

```
// computed 接收一个方法，里面可以使用被观察的属性

// 监控数据变化的回调，当foo里面的被观察属性变化的时候 都会调用这个方法
foo.observe(function(change){
    console.log(change) // 包含改变值foo改变前后的值
})
store.str = '1';
sotre.num = 2;
```

autorun

当我们使用decorator来使用computed,我们就无法得到改变前后的值了，这样我们就要使用autorun方法。

从方法名可以看出是“自动运行”。 所以我们要明确两点： 自动运行什么，怎么触发自动运行

自动运行传入autorun的参数 ， 修改传入的autorun的参数修改的时候会触发自动运行

```
import {observable,autorun} form 'mobx'
class Store{
    @observable arr = [];
    @observable obj = {};
    @observable mao = new Map();

    @observable num = 1;
    @observable str = 'str';
    @observable bool = true;

    // 2. 作为decorator
    @computed get mixed(){
        return store.str + '/'+ store.num
    }
}

const store = new Store();
autorun(() => {
    console.log(store.str + '/'+ store.num)
})

store.str = '1';
sotre.num = 2;
```

when

用法： when(predicate: () => boolean, effect?: () => void, options?)

when 观察并运行给定的 predicate，直到返回true。 一旦返回 true，给定的 effect 就会被执行，然后 autorunner(自动运行程序) 会被清理。
该函数返回一个清理器以提前取消自动运行程序。

```
import {observable,when} form 'mobx'
class Store{
    @observable arr = [];
    @observable obj = {};
    @observable mao = new Map();
```

```
    @observable num = 1;
    @observable str = 'str';
    @observable bool = false;

    // 2. 作为decorator
    @computed get mixed(){
        return store.str + '/'+ store.num
    }

}
const store = new Store();

when(() => store.bool,()=> {
    console.log('it's a  true)
})
store.bool = true;
```

when方法接收两个参数（两个方法），第一个参数根据可观察属性的值做出判断返回一个boolean值，当为true的时候，执行第二个参数。如果一开始就返回一个true,就会立即执行后面的方法。

reaction

```
用法：reaction(() => data, (data, reaction) => { sideEffect }, options?)

它接收两个函数参数，第一个(数据
函数)是用来追踪并返回数据作为第二个函数(效果 函数)的输入。 不同于
autorun的是当创建时效果函数不会直接运行(第二个参数不会立即执行，
autorun会立即执行传入的参数方法)，只有在数据表达式首次返回一个新值
后才会运行。 在执行 效果 函数时访问的任何 observable 都不会被追踪。

import {observable,reaction} form 'mobx'
class Store{
    @observable arr = [];
    @observable obj = {};
    @observable mao = new Map();

    @observable num = 1;
    @observable str = 'str';
    @observable bool = false;

    // 2. 作为decorator
    @computed get mixed(){
        return store.str + '/'+ store.num
    }

}
const store = new Store();

reaction(() => [store.str,store.num],(arr) => console.log(arr.join('\')))
store.str = '1';
sotre.num = 2;
当初始化的时候 程序会先执行reaction中的第一个参数方法，确定那些被观察数据被引用了，然后当被引用的数据被修改的时候，就会将执行第二个参数
```

action

在redux中，唯一可以更改state的途径便是dispatch一个action。这种约束性带来的一个好处是可维护性。整个state只要改变必定是通过action触发的，对此只要找到reducer中对应的action便能找到影响数据改变的原因。强约束性是好的，但是Redux要达到约束性的目的，似乎要写许多样板代码，虽说有许多库都在解决该问题，然而Mobx从根本上来说会更加优雅。 首先Mobx并不强制所有state的改变必须通过action来改变，这主要适用于一些较小的项目。对于较大型的，需要多人合作的项目来说，可以使用Mobx提供的api configure来强制。

observer

mobx-react的observer就将组件的render方法包装为autorun，所以当可观察属性的改变的时候，会执行render方法。

observable

observable是一种让数据的变化可以被观察的方法
observable(value) 是一个便捷的 API ，此 API 只有在它可以被制作成可观察的数据结构(数组、映射或 observable 对象)时才会成功。对于所有其他值，不会执行转换。

```
* [回到顶部](#666)

<h4 id="20">setstate</h4>

Setstate只在合成事件和钩子函数中是‘异步’的，在原生事件和setTimeout中是同步的。react内部的批量更新也是建立在异步之上，除了合成事件、钩子函数以外，其他情况都不会触发批量更新的，批量更新就是连续多次的setstate调用合成一次，同一个属性改变会取最后一次，不同属性改变进行合并。而setstate异步并不是说内部由异步代码实现，其实它本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用是在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形成了所谓的‘异步’。


* [回到顶部](#666)

<h4 id="21">Generator 函数</h4>

    Generator函数是ES6提供的一种异步编程解决方案，语法行为与传统函数完全不同
    Generator函数有多种理解角度：
语法上，Generator函数是一个状态机，封装了多个内部状态。执行Generator函数会返回一个遍历器对象，可以依次遍历Generator函数内部的每一个状态。
    形式上，Generator函数是一个普通函数，但是有两个特征。一是，function关键字与函数名之间有一个星号；二是，函数体内部使用yield表达式，定义不同的内部状态。
<h6>yield语句</h6>
    Generator函数返回的遍历器对象，yield语句暂停，调用next方法恢复执行，如果没遇到新的yeild，一直运行到return语句为止，return 后面表达式的值作为返回对象的value值，如果没有return语句，一直运行到结束，返回对象的value为undefined.
```