真传X

IT前沿技术在线大学

www.zhenchuanx.com

React.js

所谓的virtual diff

计算treeA转换成treeB的最少操作

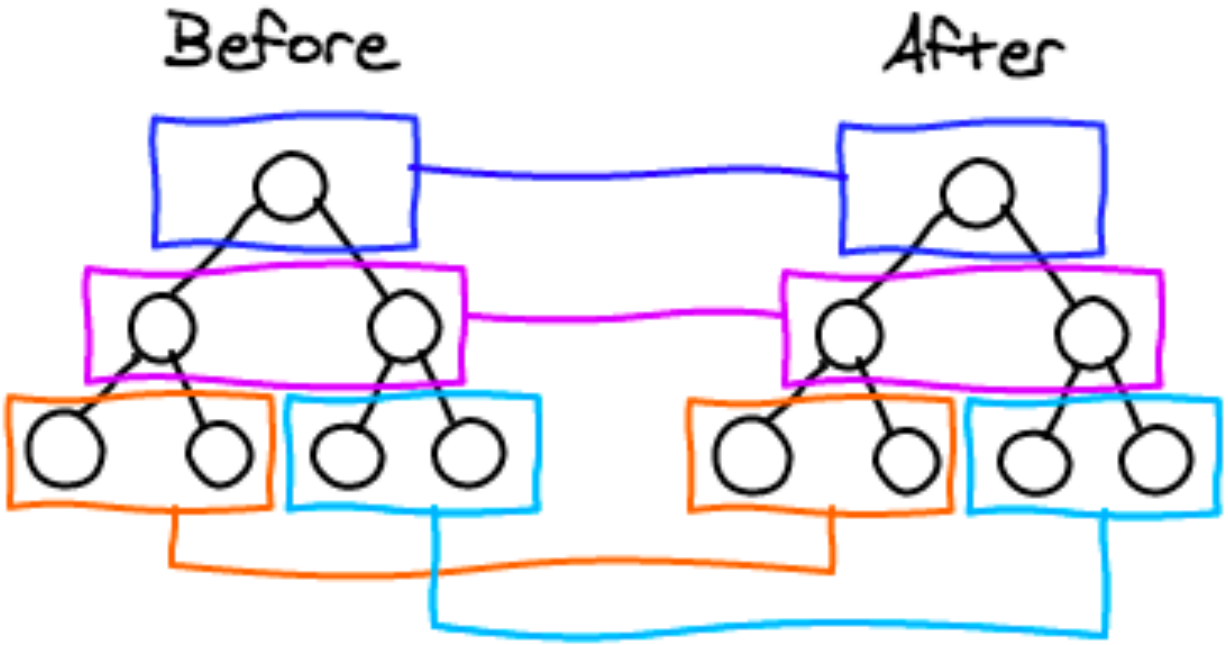
传统解法并不友好，算法复杂度：O(n^3)

https://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf

react是怎么做的？

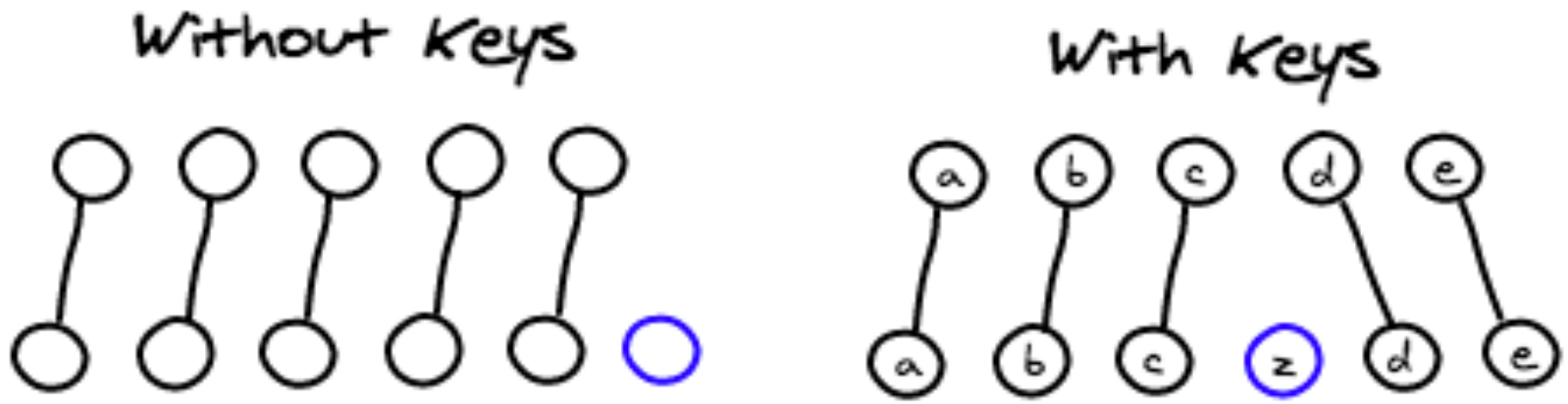
https://blog.csdn.net/zhulin2609/article/details/82263618

分层对比

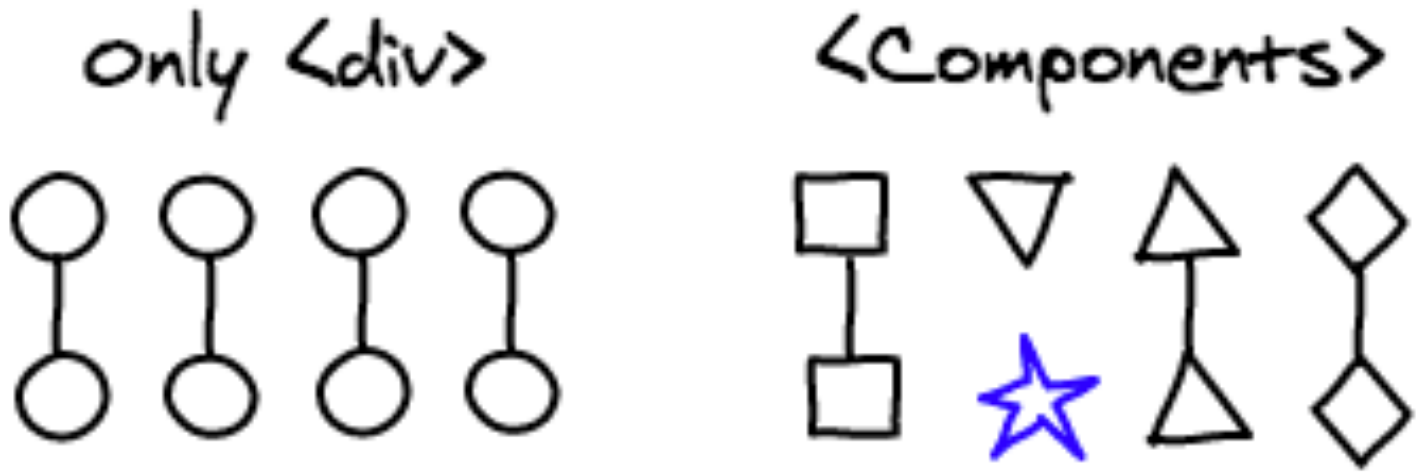


算法的复杂度接近O(n)

基于key匹配



基于自定义组件的优化



render和re-render

首次渲染

1. 创建虚拟DOM
2. 遍历虚拟DOM，并创建对应的真实DOM
3. 将创建好的真实DOM一次性append到容器中

再次渲染

1. 对比新旧两个虚拟DOM
2. 生成差异
3. 一次性执行所有差异

React中的key属性

https://reactjs.org/docs/lists-and-keys.html

旧的virtual dom

```
<ul>
    <li>first</li>
</ul>
```

新的virtual dom

```
<ul>
    <li>first</li>
    <li>second</li>
</ul>
```

1. insertNode(
second)

旧的virtual dom

```
<ul>
    <li>first</li>
    <li>second</li>
</ul>
```

新的virtual dom

```
<ul>
    <li>zero</li>
    <li>first</li>
    <li>second</li>
</ul>
```

1. replaceAttribute textContent ‘first’ to ‘zero’
2. replaceAttribute textContent ‘second’ to ‘first’
3. insertNode(
second)

https://en.wikipedia.org/wiki/Levenshtein_distance

算法复杂度：O(n2)

旧的virtual dom

```
<ul>
    <li key='1'>first</li>
    <li key='2'>second</li>
</ul>
```

新的virtual dom

```
<ul>
    <li key='0'>zero</li>
    <li key='1'>first</li>
    <li key='2'>second</li>
</ul>
```

1. insertNode(
zero)

利用hashTable存储key

算法复杂度O(n)

旧的virtual dom

```
<ul>
    <li key='0'>zero</li>
    <li key='1'>first</li>
    <li key='2'>second</li>
</ul>
```

新的virtual dom

```
<ul>
    <li key='0'>c</li>
    <li key='1'>b</li>
    <li key='2'>a</li>
</ul>
```

1. replaceAttribute textContent ‘zero’ to ‘c’
2. replaceAttribute textContent ‘first’ to ‘b’
3. replaceAttribute textContent ‘second’ to ‘a’

不会移动节点，只会替换内容

记住，key只需要在兄弟节点中唯一，而不是全局唯一

React中的组件

什么是组件?

组件能够将你的界面分割成独立且可复用的小块，而这些小组件都是互相独立的。

什么是组件?

但我认为：“对于To C的产品来说，相对于可复用性，组件化更大的价值在于提升可维护性。”

To B: https://github.com/alibaba/ice、https://ant.design/index-cn

简单组件就不用再跟大家说了

React中的PureComponent

https://reactjs.org/docs/react-api.html#reactpurecomponent

checkShouldComponentUpdate

```
225     if (ctor.prototype && ctor.prototype.isPureReactComponent) {
226       return (
227         !shallowEqual(oldProps, newProps) || !shallowEqual(oldState, newState)
228       );
229     }
```

https://github.com/facebook/react/blob/0f2f90bd9a9daf241d691bf4af3ea2e3a263c0e3/packages/react-reconciler/src/ReactFiberClassComponent.js#L225-L229

让我们来看看shallowEqual

```
function shallowEqual(objA: mixed, objB: mixed): boolean {
  if (is(objA, objB)) {
    return true;
  }

  if (typeof objA !== 'object' || objA === null ||
      typeof objB !== 'object' || objB === null) {
    return false;
  }

  const keysA = Object.keys(objA);
  const keysB = Object.keys(objB);

  if (keysA.length !== keysB.length) {
    return false;
  }

  // Test for A's keys different from B.
  for (let i = 0; i < keysA.length; i++) {
    if (
      !hasOwnProperty.call(objB, keysA[i]) ||
      !is(objA[keysA[i]], objB[keysA[i]])
    ) {
      return false;
    }
  }

  return true;
}
```

https://github.com/facebook/fbjs/blob/master/packages/fbjs/src/core/shallowEqual.js

```
202       const instance = workInProgress.stateNode;
203       const ctor = workInProgress.type;
204       if (typeof instance.shouldComponentUpdate === 'function') {
205         startPhaseTimer(workInProgress, 'shouldComponentUpdate');
206         const shouldUpdate = instance.shouldComponentUpdate(
207           newProps,
208           newState,
209           newContext,
210         );
211         stopPhaseTimer();
212
213         if (__DEV__) {
214           warning(
215             shouldUpdate !== undefined,
216             '%s.shouldComponentUpdate(): Returned undefined instead of a ' +
217               'boolean value. Make sure to return true or false.',
218             getComponentName(workInProgress) || 'Component',
219           );
220         }
221
222         return shouldUpdate;
223       }
224
225       if (ctor.prototype && ctor.prototype.isPureReactComponent) {
226         return (
227           !shallowEqual(oldProps, newProps) || !shallowEqual(oldState, newState)
228         );
229       }
230
231       return true;
232     }
```

当然，如果组件实例中定义了shouldComponentUpdate，以组件的结果为准

由于Pure Component的实现原理，跟不可变数据搭配味道更佳

如果不跟不可变数据(ImmutableJS)进行搭配会有啥问题

可能的问题

- 如果 props 和 state 虽然值没变，但引用变了，就会造成虚拟DOM计算的浪费
- 如果值改了，但引用没改，又会造成不渲染

```
{this.pureList.map((item) => {
    return <PureComponent data={item} />
})}
```

嫌ImmutableJS太大了？那就手动处理吧，记得用Object.assign或者setState(本质上也是用Object.assign产生新的state)

基于shouldComponentUpdate的优化

- https://reactjs.org/docs/optimizing-performance.html#shouldcomponentupdate-in-action

React中的Stateless Component

大家觉得stateless component的用途是啥?

stateless component

- 无逻辑的纯展示组件
- 用于Server端渲染，避免了DOM操作

Component vs Stateless Functional component

- Component包含内部state，而Stateless Functional Component所有数据都来自props，没有内部state
- Component 包含的一些生命周期函数，Stateless Functional Component都没有，因为Stateless Functional component没有shouldComponentUpdate,所以也无法控制组件的渲染，也即是说只要是收到新的props，Stateless Functional Component就会重新渲染。
- Stateless Functional Component 不支持refs

https://stackoverflow.com/questions/40703675/react-functional-stateless-component-purecomponent-component-what-are-the-dif

Container & Presentational Component

大家来对比一下~

Presentational组件

- 只关注展示，只关注自己的UI状态
- 可以包含其他的container和presentational组件
- 可以有自己的DOM和样式
- 不依赖app的其他部分（如：redux的action和store）
- 不指定数据的加载和变化
- 只通过props接受数据和回调函数

Container

- 处理交互和逻辑
- 可以包含其他的container和presentational组件
- 除了必要的div包裹，不需要有自己的DOM和样式
- 通常作为数据源，提供数据和行为给其他presentational组件（不一定是children）
- 会使用dispatch(action)和redux的connect()/Relay的createContainer/Flux Utils的Container.create
- 通常是高阶组件

React中的高阶组件（HOC）

https://reactjs.org/docs/higher-order-components.html

```
function ppHOC(WrappedComponent) {
    return class PP extends React.Component {
        constructor(props) {
            super(props);

            this.state = {
                name: ''
            };

            this.onNameChange = this.onNameChange.bind(this);
        }

        onNameChange(event) {
            this.setState({
                name: event.target.value
            })
        }

        render() {
            const { props, state } = this;
            const newProps = {
                name: {
                    value: state.name,
                    onChange: this.onNameChange
                }
            }

            return <WrappedComponent {...this.props} {...newProps}  />
        }
    }
}
```

```
export default class Example extends React.Component {
    render() {
        return <input name='name' {...this.props.name} />
    }
}
```

这样input就可以快速变成受控组件

```
function iiHOC(WrappedComponent) {
    return class Enhancer extends WrappedComponent {
        render() {
            if(this.props.flag) {
                return super.render()
            } else {
                return null
            }
        }
    }
}
```

渲染劫持

但HOC有一些坑

```
render() {
  // A new version of EnhancedComponent is created on every render
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // That causes the entire subtree to unmount/remount each time!
  return <EnhancedComponent />;
}
```

不要在render里写HOC

```
// Define a static method
WrappedComponent.staticMethod = function() {/*...*/}
// Now apply a HOC
const EnhancedComponent = enhance(WrappedComponent);

// The enhanced component has no static method
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

静态方法不会被继承

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

在HOC中先把静态方法复制一份，再return HOC

```
<HighOrderInput ref={(ele) => {
    this.hoc= ele
}}/>
```

```
console.log(this.hoc)
```

无法获得ref

```
getWarpIns() {
    return this.wrappedIns
}

render() {
    const { props, state } = this;
    const newProps = {
        name: {
            value: state.name,
            onChange: this.onNameChange
        }
    }

    // return <WrappedComponent {...this.props} {...newProps}  />
    return (
        <div style={{display: 'block'}}>
            <WrappedComponent {...this.props} {...newProps}
                ref={(ele) => {
                    this.wrappedIns = ele;
                }}  />
        </div>
    )
}
```

```
console.log(this.hoc.getWarpIns())
```

或者使用React.forwardRef API

IT前沿技术在线大学