react面试题总结一波，以备不时之需

React组件的构造函数有什么作用？它是必须的吗？

构造函数主要用于两个目的：

- 通过将对象分配给this.state来初始化本地状态
- 将事件处理程序方法绑定到实例上

所以，当在React class中需要设置state的初始值或者绑定事件时，需要加上构造函数，官方Demo：

javascript 复制代码

```
class LikeButton extends React.Component {
  constructor() {
    super();
    this.state = {
      liked: false
    };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState({liked: !this.state.liked});
  }
  render() {
    const text = this.state.liked ? 'liked' : 'haven\'t liked';
    return (
      <div onClick={this.handleClick}>
        You {text} this. Click to toggle.      </div>
    );
  }
}
ReactDOM.render(
  <LikeButton />,
  document.getElementById('example')
);
```

构造函数用来新建父类的this对象；子类必须在constructor方法中调用super方法；否则新建实例时会报错；因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法；子类就得不到this对象。

注意：

- constructor () 必须配上 super(), 如果要在constructor 内部使用 this.props 就要 传入 props , 否则不用
- JavaScript中的 bind 每次都会返回一个新的函数, 为了性能等考虑, 尽量在constructor中绑定事件

除了在构造函数中绑定 `this`，还有其它方式吗

你可以使用属性初始值设定项(property initializers)来正确绑定回调，create-react-app 也是默认支持的。在回调中你可以使用箭头函数，但问题是每次组件渲染时都会创建一个新的回调。

什么原因会促使你脱离 create-react-app 的依赖

当你想去配置 webpack 或 babel presets。

何为 action

Actions 是一个纯 javascript 对象，它们必须有一个 type 属性表明正在执行的 action 的类型。实质上，action 是将数据从应用程序发送到 store 的有效载荷。

diff算法如何比较?

- 只对同级比较，跨层级的dom不会进行复用
- 不同类型节点生成的dom树不同，此时会直接销毁老节点及子孙节点，并新建节点
- 可以通过key来对元素diff的过程提供复用的线索
- 单节点diff
- 单点diff有如下几种情况：
- key和type相同表示可以复用节点
- key不同直接标记删除节点，然后新建节点
- key相同type不同，标记删除该节点和兄弟节点，然后新创建节点

组件通信的方式有哪些

- **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
- **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
- **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

参考 前端进阶面试题详细解答

什么是受控组件和非受控组件

- 受控组件:

  没有维持自己的状态

  数据由付组件控制

  通过props获取当前值，然后通过回调函数通知更改

- 非受控组件

  保持这个自己的状态

  数据有DOM控制

  refs用于获取其当前值

React的虚拟DOM和Diff算法的内部实现

> 传统 diff 算法的时间复杂度是 O(n^3)，这在前端 render 中是不可接受的。为了降低时间复杂度，react 的 diff 算法做了一些妥协，放弃了最优解，最终将时间复杂度降低到

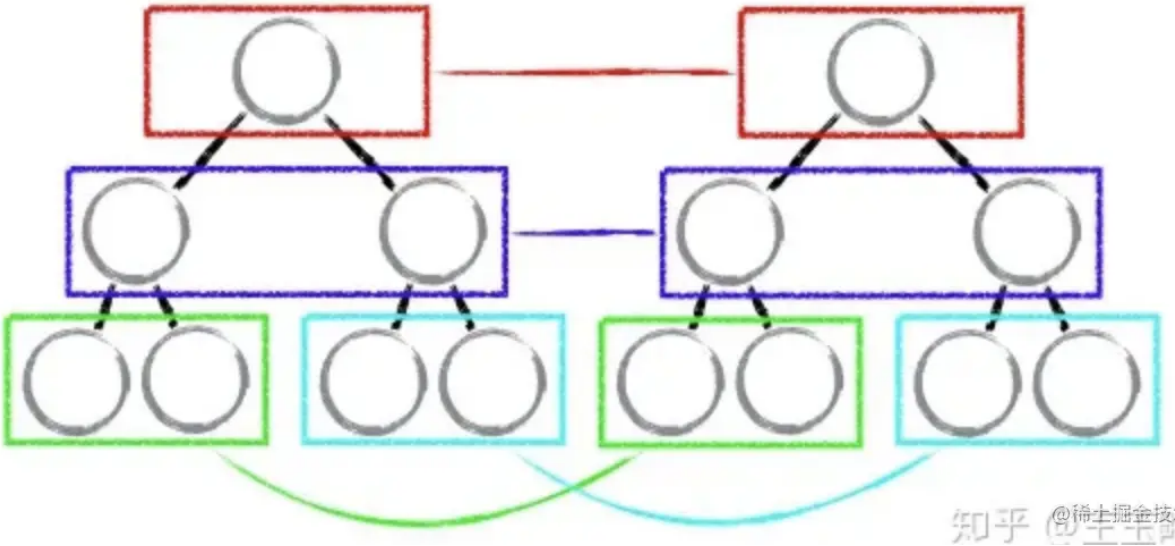
> 了 O(n)。

那么 react diff 算法做了哪些妥协呢？，参考如下：

1. tree diff：只对比同一层的 dom 节点，忽略 dom 节点的跨层级移动

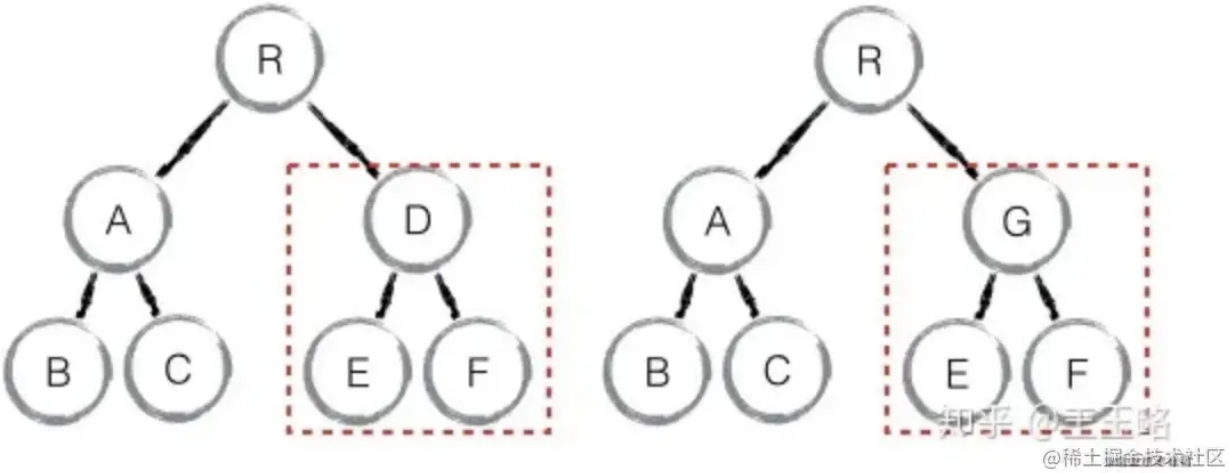
如下图，react 只会对相同颜色方框内的 DOM 节点进行比较，即同一个父节点下的所有子节点。当发现节点不存在时，则该节点及其子节点会被完全删除掉，不会用于进一步的比较。

这样只需要对树进行一次遍历，便能完成整个 DOM 树的比较。



这就意味着，如果 dom 节点发生了跨层级移动，react 会删除旧的节点，生成新的节点，而不会复用。

2. component diff：如果不是同一类型的组件，会删除旧的组件，创建新的组件



3. element diff：对于同一层级的一组子节点，需要通过唯一 id 进行来区分

- 如果没有 id 来进行区分，一旦有插入动作，会导致插入位置之后的列表全部重新渲染
- 这也是为什么渲染列表时为什么要使用唯一的 key。

React如何获取组件对应的DOM元素?

可以用ref来获取某个子节点的实例，然后通过当前class组件实例的一些特定属性来直接获取子节点实例。ref有三种实现方法:

- **字符串格式**：字符串格式，这是React16版本之前用得最多的，例如：`<p ref="info">span</p>`
- **函数格式**：ref对应一个方法，该方法有一个参数，也就是对应的节点实例，例如：`<p ref={ele => this.info = ele}></p>`
- **createRef方法**：React 16提供的一个API，使用React.createRef()来实现

如何配置 React-Router 实现路由切换

（1）使用 `<Route>` 组件

路由匹配是通过比较 `<Route>` 的 path 属性和当前地址的 pathname 来实现的。当一个 `<Route>` 匹配成功时，它将渲染其内容，当它不匹配时就会渲染 null。没有路径的 `<Route>` 将始终被匹配。

javascript 复制代码

```
// when location = { pathname: '/about' }
<Route path='/about' component={About}/> // renders <About/>
<Route path='/contact' component={Contact}/> // renders null
<Route component={Always}/> // renders <Always/>
```

（2）结合使用 `<Switch>` 组件和 `<Route>` 组件

`<Switch>` 用于将 `<Route>` 分组。

javascript 复制代码

```
<Switch>
    <Route exact path="/" component={Home} />
    <Route path="/about" component={About} />
```

```
    <Route path="/contact" component={Contact} />
</Switch>
```

`<Switch>` 不是分组 `<Route>` 所必须的，但他通常很有用。 一个 `<Switch>` 会遍历其所有的子 `<Route>` 元素，并仅渲染与当前地址匹配的第一个元素。

（3）使用 `<Link>`、 `<NavLink>`、`<Redirect>` 组件

`<Link>` 组件来在你的应用程序中创建链接。无论你在何处渲染一个 `<Link>` ，都会在应用程序的 HTML 中渲染锚（ `<a>` ）。

javascript 复制代码

```
<Link to="/">Home</Link>
// <a href='/'>Home</a>
```

是一种特殊类型的 当它的 to属性与当前地址匹配时，可以将其定义为"活跃的"。

javascript 复制代码

```
// location = { pathname: '/react' }
<NavLink to="/react" activeClassName="hurray">
    React
</NavLink>
// <a href='/react' className='hurray'>React</a>
```

当我们想强制导航时，可以渲染一个 `<Redirect>` ，当一个 `<Redirect>` 渲染时，它将使用它的to属性进行定向。

Redux Thunk 的作用是什么

Redux thunk 是一个允许你编写返回一个函数而不是一个 action 的 actions creators 的中间件。如果满足某个条件，thunk 则可以用来延迟 action 的派发(dispatch)，这可以处理异步 action 的派发(dispatch)。

React实现的移动应用中，如果出现卡顿，有哪些可以考虑的优化方案

- 增加 `shouldComponentUpdate` 钩子对新旧 `props` 进行比较，如果值相同则阻止更新，避免不必要的渲染，或者使用 `PureReactComponent` 替代 `Component` ，其内部已经封装了

`shouldComponentUpdate` 的浅比较逻辑

- 对于列表或其他结构相同的节点，为其中的每一项增加唯一 `key` 属性，以方便 `React` 的 `diff` 算法中对该节点的复用，减少节点的创建和删除操作
- `render` 函数中减少类似 `onClick={() => {doSomething()}}` 的写法，每次调用render函数时均会创建一个新的函数，即使内容没有发生任何变化，也会导致节点没必要的重渲染，建议将函数保存在组件的成员对象中，这样只会创建一次
- 组件的 `props` 如果需要经过一系列运算后才能拿到最终结果，则可以考虑使用 `reselect` 库对结果进行缓存，如果props值未发生变化，则结果直接从缓存中拿，避免高昂的运算代价
- `webpack-bundle-analyzer` 分析当前页面的依赖包，是否存在不合理性，如果存在，找到优化点并进行优化

Diff 的瓶颈以及 React 的应对

由于 diff 操作本身会带来性能上的损耗，在 React 文档中提到过，即使最先进的算法中，将前后两棵树完全比对的算法复杂度为 `O(n3)` ，其中 n 为树中元素的数量。

如果 React 使用了该算法，那么仅仅一千个元素的页面所需要执行的计算量就是十亿的量级，这无疑是无法接受的。

为了降低算法的复杂度，React 的 diff 会预设三个限制：

1. 只对同级元素进行 diff 比对。如果一个元素节点在前后两次更新中跨越了层级，那么 React 不会尝试复用它
2. 两个不同类型的元素会产生出不同的树。如果元素由 div 变成 p，React 会销毁 div 及其子孙节点，并新建 p 及其子孙节点
3. 开发者可以通过 key 来暗示哪些子元素在不同的渲染下能保持稳定

fetch封装

javascript 复制代码

```
npm install whatwg-fetch --save  // 适配其他浏览器
npm install es6-promise

export const handleResponse = (response) => {
  if (response.status === 403 || response.status === 401) {
    const oauthurl = response.headers.get('locationUrl');
    if (!_.isEmpty(oauthUrl)) {
      window.location.href = oauthurl;
      return;
    }
```

```
  }
  if (!response.ok) {
    return getErrorMessage(response).then(errorMessage => apiError(response.status, errorMessage));
  }
  if (isJson(response)) {
    return response.json();
  }
  if (isText(response)) {
    return response.text();
  }

  return response.blob();
};

const httpRequest = {
  request: ({
    method, headers, body, path, query,
  }) => {
    const options = {};
    let url = path;
    if (method) {
      options.method = method;
    }
    if (headers) {
      options.headers = {...options.headers,...headers};
    }
    if (body) {
      options.body = body;
    }
    if (query) {
      const params = Object.keys(query)
        .map(k => `${k}=${query[k]}`)
        .join('&');
      url = url.concat(`?${params}`);
    }
    return fetch(url, Object.assign({}, options, { credentials: 'same-origin' })).then(handleRespons
  },
};

export default httpRequest;
```

什么是上下文Context

Context 通过组件树提供了一个传递数据的方法，从而避免了在每一个层级手动的传递 props 属性。

- 用法：在父组件上定义getChildContext方法，返回一个对象，然后它的子组件就可以通过this.context属性来获取

javascript 复制代码

```
import React,{Component} from 'react';
import ReactDOM from 'react-dom';
import PropTypes from 'prop-types';
class Header extends Component{
    render() {
        return (
            <div>
                <Title/>
            </div>
        )
    }
}
class Title extends Component{
    static contextTypes={
        color:PropTypes.string
    }
    render() {
        return (
            <div style={{color:this.context.color}}>
                Title
            </div>
        )
    }
}
class Main extends Component{
    render() {
        return (
            <div>
                <Content>
                </Content>
            </div>
        )
    }
}
class Content extends Component{
    static contextTypes={
        color: PropTypes.string,
        changeColor:PropTypes.func
    }
    render() {
        return (
            <div style={{color:this.context.color}}>
                Content
                <button onClick={()=>this.context.changeColor('green')}>绿色</button>
                <button onClick={()=>this.context.changeColor('orange')}>橙色</button>
```

```
            </div>
        )
    }
}
class Page extends Component{
    constructor() {
        super();
        this.state={color:'red'};
    }
    static childContextTypes={
        color: PropTypes.string,
        changeColor:PropTypes.func
    }
    getChildContext() {
        return {
            color: this.state.color,
            changeColor:(color)=>{
                this.setState({color})
            }
        }
    }
    render() {
        return (
            <div>
                <Header/>
                <Main/>
            </div>
        )
    }
}
ReactDOM.render(<Page/>,document.querySelector('#root'));
```

何为 `Children`

在JSX表达式中，一个开始标签(比如 `<a>`)和一个关闭标签(比如 `</a>`)之间的内容会作为一个特殊的属性 `props.children` 被自动传递给包含着它的组件。

这个属性有许多可用的方法，包括 `React.Children.map` ， `React.Children.forEach` ， `React.Children.count` ， `React.Children.only` ， `React.Children.toArray` 。

componentWillReceiveProps调用时机

- 已经被废弃掉
- 当props改变的时候才调用，子组件第二次接收到props的时候

React 性能优化

- shouldCompoentUpdate
- pureComponent 自带shouldCompoentUpdate的浅比较优化
- 结合Immutable.js达到最优

说说你用react有什么坑点?

1. JSX做表达式判断时候，需要强转为boolean类型

> 如果不使用 `!!b` 进行强转数据类型，会在页面里面输出 `0` 。

javascriptx 复制代码

```
render() {
  const b = 0;
  return <div>
    {
      !!b && <div>这是一段文本</div>
    }
  </div>
}
```

2. 尽量不要在 `componentWillReviceProps` 里使用 setState，如果一定要使用，那么需要判断结束条件，不然会出现无限重渲染，导致页面崩溃

3. 给组件添加ref时候，尽量不要使用匿名函数，因为当组件更新的时候，匿名函数会被当做新的prop处理，让ref属性接受到新函数的时候，react内部会先清空ref，也就是会以null为回调参数先执行一次ref这个props，然后在以该组件的实例执行一次ref，所以用匿名函数做ref的时候，有的时候去ref赋值后的属性会取到null

4. 遍历子节点的时候，不要用 index 作为组件的 key 进行传入

React Hooks 解决了哪些问题?

React Hooks 主要解决了以下问题：

（1）在组件之间复用状态逻辑很难

React 没有提供将可复用性行为“附加”到组件的途径（例如，把组件连接到 store）解决此类问题可以使用 render props 和 高阶组件。但是这类方案需要重新组织组件结构，这可能会很麻烦，并且会使代码难以理解。由 providers，consumers，高阶组件，render props 等其他抽象层组成的组件会形成“嵌套地狱”。尽管可以在 DevTools 过滤掉它们，但这说明了一个更深层次的问题：React 需要为共享状态逻辑提供更好的原生途径。

可以使用 Hook 从组件中提取状态逻辑，使得这些逻辑可以单独测试并复用。Hook 使我们在无需修改组件结构的情况下复用状态逻辑。 这使得在组件间或社区内共享 Hook 变得更便捷。

（2）复杂组件变得难以理解

在组件中，每个生命周期常常包含一些不相关的逻辑。例如，组件常常在 componentDidMount 和 componentDidUpdate 中获取数据。但是，同一个 componentDidMount 中可能也包含很多其它的逻辑，如设置事件监听，而之后需在 componentWillUnmount 中清除。相互关联且需要对照修改的代码被进行了拆分，而完全不相关的代码却在同一个方法中组合在一起。如此很容易产生 bug，并且导致逻辑不一致。

在多数情况下，不可能将组件拆分为更小的粒度，因为状态逻辑无处不在。这也给测试带来了一定挑战。同时，这也是很多人将 React 与状态管理库结合使用的原因之一。但是，这往往会引入了很多抽象概念，需要你在不同的文件之间来回切换，使得复用变得更加困难。

为了解决这个问题，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），而并非强制按照生命周期划分。你还可以使用 reducer 来管理组件的内部状态，使其更加可预测。

（3）难以理解的 class

除了代码复用和代码管理会遇到困难外，class 是学习 React 的一大屏障。我们必须去理解 JavaScript 中 this 的工作方式，这与其他语言存在巨大差异。还不能忘记绑定事件处理器。没有稳定的语法提案，这些代码非常冗余。大家可以很好地理解 props，state 和自顶向下的数据流，但对 class 却一筹莫展。即便在有经验的 React 开发者之间，对于函数组件与 class 组件的差异也存在分歧，甚至还要区分两种组件的使用场景。

为了解决这些问题，Hook 使你在非 class 的情况下可以使用更多的 React 特性。 从概念上讲，React 组件一直更像是函数。而 Hook 则拥抱了函数，同时也没有牺牲 React 的精神原则。Hook 提供了问题的解决方案，无需学习复杂的函数式或响应式编程技术