前端必会react面试题及答案

state 和 props 触发更新的生命周期分别有什么区别?

state 更新流程： 这个过程当中涉及的函数：

1. shouldComponentUpdate: 当组件的 state 或 props 发生改变时，都会首先触发这个生命周期函数。它会接收两个参数：nextProps, nextState——它们分别代表传入的新 props 和新的 state 值。拿到这两个值之后，我们就可以通过一些对比逻辑来决定是否有 re-render（重渲染）的必要了。如果该函数的返回值为 false，则生命周期终止，反之继续；

> 注意：此方法仅作为**性能优化的方式**而存在。不要企图依靠此方法来"阻止"渲染，因为这可能会产生 bug。应该**考虑使用内置的 PureComponent 组件**，而不是手动编写 `shouldComponentUpdate()`

2. componentWillUpdate：当组件的 state 或 props 发生改变时，会在渲染之前调用 componentWillUpdate。componentWillUpdate **是 React16 废弃的三个生命周期之一。**过去，我们可能希望能在这个阶段去收集一些必要的信息（比如更新前的 DOM 信息等等），现在我们完全可以在 React16 的 getSnapshotBeforeUpdate 中去做这些事；
3. componentDidUpdate：componentDidUpdate() 会在UI更新后会被立即调用。它接收 prevProps（上一次的 props 值）作为入参，也就是说在此处我们仍然可以进行 props 值对比（再次说明 componentWillUpdate 确实鸡肋哈）。

props 更新流程： 相对于 state 更新，props 更新后唯一的区别是增加了对 componentWillReceiveProps 的调用。关于 componentWillReceiveProps，需要知道这些事情：

- componentWillReceiveProps：它在Component接受到新的 props 时被触发。componentWillReceiveProps 会接收一个名为 nextProps 的参数（对应新的 props 值）。**该生命周期是 React16 废弃掉的三个生命周期之一。**在它被废弃前，可以用它来比

较 this.props 和 nextProps 来重新setState。在 React16 中，用一个类似的新生命周期 getDerivedStateFromProps 来代替它。

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

props 是什么

- react的核心思想是组件化，页面被分成很多个独立，可复用的组件
- 而组件就是一个函数，可以接受一个参数作为输入值，这个参数就是props，所以props就是从外部传入组件内部的数据
- 由于react的单向数据流模式，所以props是从父组件传入子组件的数据

react代理原生事件为什么?

通过冒泡实现，为了统一管理，对更多浏览器有兼容效果

合成事件原理

如果react事件绑定在了真实DOM节点上，一个节点同事有多个事件时，页面的响应和内存的占用会受到很大的影响。因此SyntheticEvent作为中间层出现了。

事件没有在目标对象上绑定，而是在document上监听所支持的所有事件，当事件发生并冒泡至document时，react将事件内容封装并叫由真正的处理函数运行。

为什么 React 要用 JSX?

JSX 是一个 JavaScript 的语法扩展，或者说是一个类似于 XML 的 ECMAScript 语法扩展。它本身没有太多的语法定义，也不期望引入更多的标准。

其实 React 本身并不强制使用 JSX。在没有 JSX 的时候，React 实现一个组件依赖于使用 React.createElement 函数。代码如下：

javascript 复制代码

```javascript
class Hello extends React.Component {
  render() {
    return React.createElement(
        'div',
        null,
        `Hello ${this.props.toWhat}`
      );
  }
}
ReactDOM.render(
  React.createElement(Hello, {toWhat: 'World'}, null),
  document.getElementById('root')
);
```

而 JSX 更像是一种语法糖，通过类似 XML 的描述方式，描写函数对象。在采用 JSX 之后，这段代码会这样写：

javascript 复制代码

```javascript
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}
ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);
```

通过对比，可以清晰地发现，代码变得更为简洁，而且代码结构层次更为清晰。

因为 React 需要将组件转化为虚拟 DOM 树，所以在编写代码时，实际上是在手写一棵结构树。而**XML 在树结构的描述上天生具有可读性强的优势。**

但这样可读性强的代码仅仅是给写程序的同学看的，实际上在运行的时候，会使用 Babel 插件将 JSX 语法的代码还原为 React.createElement 的代码。

总结： JSX 是一个 JavaScript 的语法扩展，结构类似 XML。JSX 主要用于声明 React 元素，但 React 中并不强制使用 JSX。即使使用了 JSX，也会在构建过程中，通过 Babel 插件编译为 React.createElement。所以 JSX 更像是 React.createElement 的一种语法糖。

React 团队并不想引入 JavaScript 本身以外的开发体系。而是希望通过合理的关注点分离保持组件开发的纯粹性。

react 实现一个全局的 dialog

javascript 复制代码

```
import React, { Component } from 'react';
import { is, fromJS } from 'immutable';
import ReactDOM from 'react-dom';
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
import './dialog.css';
let defaultState = {
  alertStatus:false,
  alertTip:"提示",
  closeDialog:function(){},
  childs:''
}
class Dialog extends Component{
  state = {
    ...defaultState
  };
  // css动画组件设置为目标组件
  FirstChild = props => {
    const childrenArray = React.Children.toArray(props.children);
    return childrenArray[0] || null;
  }
  //打开弹窗
  open =(options)=>{
    options = options || {};
    options.alertStatus = true;
    var props = options.props || {};
    var childs = this.renderChildren(props,options.childrens) || '';
    console.log(childs);
    this.setState({
      ...defaultState,
```

```
      ...options,
      childs
    })
  }
  //关闭弹窗
  close(){
    this.state.closeDialog();
    this.setState({
      ...defaultState
    })
  }
  renderChildren(props,childrens) {
    //遍历所有子组件
    var childs = [];
    childrens = childrens || [];
    var ps = {
        ...props,  //给子组件绑定props
        _close:this.close  //给子组件也绑定一个关闭弹窗的事件
       };
    childrens.forEach((currentItem,index) => {
        childs.push(React.createElement(
            currentItem,
            {
                ...ps,
                key:index
            }
        ));
    })
    return childs;
  }
  shouldComponentUpdate(nextProps, nextState){
    return !is(fromJS(this.props), fromJS(nextProps)) || !is(fromJS(this.state), fromJS(nextState))
  }

  render(){
    return (
      <ReactCSSTransitionGroup
        component={this.FirstChild}
        transitionName='hide'
        transitionEnterTimeout={300}
        transitionLeaveTimeout={300}>
        <div className="dialog-con" style={this.state.alertStatus? {display:'block'}:{display:'none'
            {this.state.childs}        </div>
      </ReactCSSTransitionGroup>
    );
  }
}
let div = document.createElement('div');
let props = {
```

```
};
document.body.appendChild(div);
let Box = ReactD
```

子类：

javascript 复制代码

```
//子类jsx
import React, { Component } from 'react';
class Child extends Component {
    constructor(props){
        super(props);
        this.state = {date: new Date()};
  }
  showValue=()=>{
    this.props.showValue && this.props.showValue()
  }
  render() {
    return (
      <div className="Child">
        <div className="content">
           Child           <button onClick={this.showValue}>调用父的方法</button>
        </div>
      </div>
    );
  }
}
export default Child;
```

CSS：

css 复制代码

```
.dialog-con{
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background: rgba(0, 0, 0, 0.3);
}
```

参考 前端进阶面试题详细解答

react-router4的核心

- 路由变成了组件
- 分散到各个页面，不需要配置 比如 `<link> <route></route>`

什么是虚拟DOM?

虚拟 DOM (VDOM)**是真实 DOM 在内存中的表示。UI 的表示形式保存在内存中，并与实际的 DOM 同步。这是一个发生在渲染函数被调用和元素在屏幕上显示之间的步骤，整个过程被称为**调和。

React.forwardRef有什么用

forwardRef

- 使用 `forwardRef` （ `forward` 在这里是「传递」的意思）后，就能跨组件传递 `ref` 。
- 在例子中，我们将 `inputRef` 从 `Form` 跨组件传递到 `MyInput` 中，并与 `input` 产生关联

javascript 复制代码

```
const MyInput = forwardRef((props, ref) => {
  return <input {...props} ref={ref} />;
});

function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <MyInput ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}
```

useImperativeHandle

> 除了「限制跨组件传递 `ref` 」外，还有一种「防止 `ref` 失控的措施」，那就是 `useImperativeHandle` ，他的逻辑是这样的：既然 「ref失控」 是由于「使用了不该被使用的DOM方法」（比如 `appendChild` ），那我可以限制「 `ref` 中只存在可以被使用的方法」。用 `useImperativeHandle` 修改我们的MyInput组件：

javascript 复制代码

```
const MyInput = forwardRef((props, ref) => {
  const realInputRef = useRef(null);
  useImperativeHandle(ref, () => ({
    focus() {
      realInputRef.current.focus();
    },
  }));
  return <input {...props} ref={realInputRef} />;
});
```

现在， `Form` 组件中通过 `inputRef.current` 只能取到如下数据结构：

javascript 复制代码

```
{
  focus() {
    realInputRef.current.focus();
  },
}
```

> 就杜绝了 「开发者通过ref取到DOM后，执行不该被使用的API，出现ref失控」 的情况

- 为了防止错用/滥用导致 `ref` 失控，React限制 「默认情况下，不能跨组件传递ref」
- 为了破除这种限制，可以使用 `forwardRef` 。
- 为了减少 `ref` 对 `DOM` 的滥用，可以使用 `useImperativeHandle` 限制 `ref` 传递的数据结构。

react 父子传值

父传子——在调用子组件上绑定，子组件中获取this.props
子传父——引用子组件的时候传过去一个方法，子组件通过this.props.methed()传过去参数

connection

React.Children.map和js的map有什么区别?

JavaScript中的map不会对为null或者undefined的数据进行处理，而React.Children.map中的map可以处理React.Children为null或者undefined的情况。

Redux内部原理 内部怎么实现dispstch一个函数的

> 以 redux-thunk 中间件作为例子，下面就是 thunkMiddleware 函数的代码

javascript 复制代码

```
// 部分转为ES5代码，运行middleware函数会返回一个新的函数，如下：
return ({ dispatch, getState }) => {
    // next实际就是传入的dispatch
    return function (next) {
        return function (action) {
            // redux-thunk核心
            if (typeof action === 'function') {
                return action(dispatch, getState, extraArgument);
            }
            return next(action);
        };
    };
}
```

> redux-thunk 库内部源码非常的简单，允许 action 是一个函数，同时支持参数传递，否则调用方法不变

- redux 创建 Store ：通过 combineReducers 函数合并 reducer 函数，返回一个新的函数 combination （这个函数负责循环遍历运行 reducer 函数，返回全部 state ）。将这个新函数作为参数传入 createStore 函数，函数内部通过dispatch，初始化运行传入的 combination ，state生成，返回store对象
- redux 中间件： applyMiddleware 函数中间件的主要目的就是修改 dispatch 函数，返回经过中间件处理的新的 dispatch 函数
- redux 使用：实际就是再次调用循环遍历调用 reducer 函数，更新 state

为什么 React 元素有一个 $$typeof 属性

> 目的是为了防止 XSS 攻击。因为 Synbol 无法被序列化，所以 React 可以通过有没有 $$typeof 属性来断出当前的 element 对象是从数据库来的还是自己生成的。

- 如果没有 $$typeof 这个属性，react 会拒绝处理该元素。
- 在 React 的古老版本中，下面的写法会出现 XSS 攻击：

javascript 复制代码

```
// 服务端允许用户存储 JSON
let expectedTextButGotJSON = {
  type: 'div',
  props: {
    dangerouslySetInnerHTML: {
      __html: '/* 把你想的搁着 */'
    },
  },
  // ...
};
let message = { text: expectedTextButGotJSON };

// React 0.13 中有风险
<p>
  {message.text}
</p>
```

React 组件中怎么做事件代理？它的原理是什么？

React基于Virtual DOM实现了一个SyntheticEvent层（合成事件层），定义的事件处理器会接收到一个合成事件对象的实例，它符合W3C标准，且与原生的浏览器事件拥有同样的接口，支持冒泡机制，所有的事件都自动绑定在最外层上。

在React底层，主要对合成事件做了两件事：

- **事件委派：** React会把所有的事件绑定到结构的最外层，使用统一的事件监听器，这个事件监听器上维持了一个映射来保存所有组件内部事件监听和处理函数。
- **自动绑定：** React组件中，每个方法的上下文都会指向该组件的实例，即自动绑定this为当前组件。

描述 Flux 与 MVC?

传统的 MVC 模式在分离数据(Model)、UI(View和逻辑(Controller)方面工作得很好，但是 MVC 架构经常遇到两个主要问题: **数据流不够清晰**:跨视图发生的级联更新常常会导致混乱的事件网络，难于调试。 **缺乏数据完整性**:模型数据可以在任何地方发生突变，从而在整个UI中产生不可预测的结果。 使用 Flux 模式的复杂用户界面不再遭受级联更新，任何给定的React 组件都能够根据 `store` 提供的数据重建其状态。Flux 模式还通过限制对共享数据的直接访问来加强数据完整性。

父子组件的通信方式?

父组件向子组件通信：父组件通过 props 向子组件传递需要的信息。

javascript 复制代码

```
// 子组件: Child
const Child = props =>{
  return <p>{props.name}</p>
}
// 父组件 Parent
const Parent = ()=>{
    return <Child name="react"></Child>
}
```

子组件向父组件通信：：props+回调的方式。

javascript 复制代码

```
// 子组件: Child
const Child = props =>{
  const cb = msg =>{
      return ()=>{
          props.callback(msg)
      }
  }
  return (
```

```
      <button onClick={cb("你好!")}>你好</button>
  )
}
// 父组件 Parent
class Parent extends Component {
    callback(msg){
        console.log(msg)
    }
    render(){
        return <Child callback={this.callback.bind(this)}></Child>
    }
}
```

对于store的理解

Store 就是把它们联系到一起的对象。Store 有以下职责：

- 维持应用的 state；
- 提供 getState() 方法获取 state；
- 提供 dispatch(action) 方法更新 state；
- 通过 subscribe(listener)注册监听器;
- 通过 subscribe(listener)返回的函数注销监听器

讲讲什么是 JSX？

当 **Facebook** 第一次发布 React 时，他们还引入了一种新的 JS 方言 `JSX`，将原始 HTML 模板嵌入到 JS 代码中。JSX 代码本身不能被浏览器读取，必须使用 `Babel` 和 `webpack` 等工具将其转换为传统的JS。很多开发人员就能无意识使用 JSX，因为它已经与 React 结合在一直了。

javascript 复制代码

```
class MyComponent extends React.Component {
  render() {
    let props = this.props;
    return (
      <div className="my-component">
        <a href={props.url}>{props.name}</a>
      </div>
    );
  }
}
```

在React中如何避免不必要的render?

React 基于虚拟 DOM 和高效 Diff 算法的完美配合，实现了对 DOM 最小粒度的更新。大多数情况下，React 对 DOM 的渲染效率足以业务日常。但在个别复杂业务场景下，性能问题依然会困扰我们。此时需要采取一些措施来提升运行性能，其很重要的一个方向，就是避免不必要的渲染（Render）。这里提下优化的点：

- **shouldComponentUpdate 和 PureComponent**

在 React 类组件中，可以利用 shouldComponentUpdate或者 PureComponent 来减少因父组件更新而触发子组件的 render，从而达到目的。shouldComponentUpdate 来决定是否组件是否重新渲染，如果不希望组件重新渲染，返回 false 即可。

- **利用高阶组件**

在函数组件中，并没有 shouldComponentUpdate 这个生命周期，可以利用高阶组件，封装一个类似 PureComponet 的功能

- **使用 React.memo**

React.memo 是 React 16.6 新的一个 API，用来缓存组件的渲染，避免不必要的更新，其实也是一个高阶组件，与 PureComponent 十分类似，但不同的是， React.memo只能用于函数组件。

对React SSR的理解

服务端渲染是数据与模版组成的html，即 HTML = 数据 + 模版。将组件或页面通过服务器生成html字符串，再发送到浏览器，最后将静态标记"混合"为客户端上完全交互的应用程序。页面没使用服务渲染，当请求页面时，返回的body里为空，之后执行js将html结构注入到body里，结合css显示出来;

SSR的优势：

- 对SEO友好
- 所有的模版、图片等资源都存在服务器端
- 一个html返回所有数据
- 减少HTTP请求
- 响应快、用户体验好、首屏渲染快

1）更利于SEO

不同爬虫工作原理类似，只会爬取源码，不会执行网站的任何脚本使用了React或者其它MVVM框架之后，页面大多数DOM元素都是在客户端根据js动态生成，可供爬虫抓取分析的内容大大减少。另外，浏览器爬虫不会等待我们的数据完成之后再去抓取页面数据。服务端渲染返回给客户端的是已经获取了异步数据并执行JavaScript脚本的最终HTML，网络爬中就可以抓取到完整页面的信息。

2）更利于首屏渲染

首屏的渲染是node发送过来的html字符串，并不依赖于js文件了，这就会使用户更快的看到页面的内容。尤其是针对大型单页应用，打包后文件体积比较大，普通客户端渲染加载所有所需文件时间较长，首页就会有一个很长的白屏等待时间。

SSR的局限：

1）服务端压力较大

本来是通过客户端完成渲染，现在统一到服务端node服务去做。尤其是高并发访问的情况，会大量占用服务端CPU资源;

2）开发条件受限

在服务端渲染中，只会执行到componentDidMount之前的生命周期钩子，因此项目引用的第三方的库也不可用其它生命周期钩子，这对引用库的选择产生了很大的限制;

3）学习成本相对较高 除了对webpack、MVVM框架要熟悉，还需要掌握node、 Koa2等相关技术。相对于客户端渲染，项目构建、部署过程更加复杂。

时间耗时比较：

1）数据请求

由服务端请求首屏数据，而不是客户端请求首屏数据，这是"快"的一个主要原因。服务端在内网进行请求，数据响应速度快。客户端在不同网络环境进行数据请求，且外网http请求开销大，导致时间差

- 客户端数据请求
- 服务端数据请求

2）html渲染 服务端渲染是先向后端服务器请求数据，然后生成完整首屏 html返回给浏览器；而客户端渲染是等js代码下载、加载、解析完成后再请求数据渲染，等待的过程页面是什么都没有的，就是用户看到的白屏。就是服务端渲染不需要等待js代码下载完成并请求数据，就可以返回一个已有完整数据的首屏页面。

- 非ssr html渲染
- ssr html渲染