高级前端一面常考react面试题总结

调和阶段 setState内部干了什么

- 当调用 setState 时，React会做的第一件事情是将传递给 setState 的对象合并到组件的当前状态
- 这将启动一个称为和解（ `reconciliation` ）的过程。和解（ `reconciliation` ）的最终目标是以最有效的方式，根据这个新的状态来更新 `UI` 。 为此， `React` 将构建一个新的 `React` 元素树（您可以将其视为 `UI` 的对象表示）
- 一旦有了这个树，为了弄清 UI 如何响应新的状态而改变，React 会将这个新树与上一个元素树相比较（ diff ）

> 通过这样做， React 将会知道发生的确切变化，并且通过了解发生什么变化，只需在绝对必要的情况下进行更新即可最小化 UI 的占用空间

React Hooks在平时开发中需要注意的问题和原因

（1）不要在循环，条件或嵌套函数中调用Hook，必须始终在 React函数的顶层使用Hook

这是因为React需要利用调用顺序来正确更新相应的状态，以及调用相应的钩子函数。一旦在循环或条件分支语句中调用Hook，就容易导致调用顺序的不一致性，从而产生难以预料到的后果。

（2）使用useState时候，使用push，pop，splice等直接更改数组对象的坑

使用push直接更改数组无法获取到新值，应该采用析构方式，但是在class里面不会有这个问题。代码示例：

javascript 复制代码

```
function Indicatorfilter() {
  let [num,setNums] = useState([0,1,2,3])
  const test = () => {
    // 这里坑是直接采用push去更新num
    // setNums(num)是无法更新num的
    // 必须使用num = [...num ,1]
    num.push(1)
```

```
    // num = [...num ,1]
    setNums(num)
  }
return (
    <div className='filter'>
      <div onClick={test}>测试</div>
        <div>
          {num.map((item,index) => (              <div key={index}>{item}</div>
          ))}      </div>
    </div>
  )
}

class Indicatorfilter extends React.Component<any,any>{
  constructor(props:any){
      super(props)
      this.state = {
          nums:[1,2,3]
      }
      this.test = this.test.bind(this)
  }

  test(){
      // class采用同样的方式是没有问题的
      this.state.nums.push(1)
      this.setState({
          nums: this.state.nums
      })
  }

  render(){
      let {nums} = this.state
      return(
          <div>
              <div onClick={this.test}>测试</div>
                  <div>
                      {nums.map((item:any,index:number) => (                          <div key={inde
                      ))}                  </div>
          </div>

      )
  }
}
```

（3）useState设置状态的时候，只有第一次生效，后期需要更新状态，必须通过useEffect

TableDeail是一个公共组件，在调用它的父组件里面，我们通过set改变columns的值，以为传递给TableDeail 的 columns是最新的值，所以tabColumn每次也是最新的值，但是实际tabColumn是最开始的值，不会随着columns的更新而更新：

javascript 复制代码

```
const TableDeail = ({    columns,}:TableData) => {
    const [tabColumn, setTabColumn] = useState(columns)
}

// 正确的做法是通过useEffect改变这个值
const TableDeail = ({    columns,}:TableData) => {
    const [tabColumn, setTabColumn] = useState(columns)
    useEffect(() =>{setTabColumn(columns)},[columns])
}
```

（4）善用useCallback

父组件传递给子组件事件句柄时，如果我们没有任何参数变动可能会选用useMemo。但是每一次父组件渲染子组件即使没变化也会跟着渲染一次。

（5）不要滥用useContext

可以使用基于 useContext 封装的状态管理工具。

React中发起网络请求应该在哪个生命周期中进行？为什么？

对于异步请求，最好放在componentDidMount中去操作，对于同步的状态改变，可以放在componentWillMount中，一般用的比较少。

如果认为在componentWillMount里发起请求能提早获得结果，这种想法其实是错误的，通常componentWillMount比componentDidMount早不了多少微秒，网络上任何一点延迟，这一点差异都可忽略不计。

react的生命周期： constructor() -> componentWillMount() -> render() -> componentDidMount()

上面这些方法的调用是有次序的，由上而下依次调用。

- constructor被调用是在组件准备要挂载的最开始，此时组件尚未挂载到网页上。

- componentWillMount方法的调用在constructor之后，在render之前，在这方法里的代码调用setState方法不会触发重新render，所以它一般不会用来作加载数据之用。
- componentDidMount方法中的代码，是在组件已经完全挂载到网页上才会调用被执行，所以可以保证数据的加载。此外，在这方法中调用setState方法，会触发重新渲染。所以，官方设计这个方法就是用来加载外部数据用的，或处理其他的副作用代码。与组件上的数据无关的加载，也可以在constructor里做，但constructor是做组件state初始化工作，并不是做加载数据这工作的，constructor里也不能setState，还有加载的时间太长或者出错，页面就无法加载出来。所以有副作用的代码都会集中在componentDidMount方法里。

总结：

- 跟服务器端渲染（同构）有关系，如果在componentWillMount里面获取数据，fetch data会执行两次，一次在服务器端一次在客户端。在componentDidMount中可以解决这个问题，componentWillMount同样也会render两次。
- 在componentWillMount中fetch data，数据一定在render后才能到达，如果忘记了设置初始状态，用户体验不好。
- react16.0以后，componentWillMount可能会被执行多次。

对React中Fragment的理解，它的使用场景是什么?

在React中，组件返回的元素只能有一个根元素。为了不添加多余的DOM节点，我们可以使用Fragment标签来包裹所有的元素，Fragment标签不会渲染出任何元素。React官方对Fragment的解释：

> React 中的一个常见模式是一个组件返回多个元素。Fragments 允许你将子列表分组，而无需向 DOM 添加额外节点。


```javascript
import React, { Component, Fragment } from 'react'

// 一般形式
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

```
// 也可以写成以下形式
render() {
  return (
    <>
      <ChildA />
      <ChildB />
      <ChildC />
    </>
  );
}
```

对React-Fiber的理解，它解决了什么问题?

React V15 在渲染时，会递归比对 VirtualDOM 树，找出需要变动的节点，然后同步更新它们， 一气呵成。这个过程期间， React 会占据浏览器资源，这会导致用户触发的事件得不到响应，并且会导致掉帧，**导致用户感觉到卡顿。**

为了给用户制造一种应用很快的"假象"，不能让一个任务长期霸占着资源。 可以将浏览器的渲染、布局、绘制、资源加载(例如 HTML 解析)、事件响应、脚本执行视作操作系统的"进程"，需要通过某些调度策略合理地分配 CPU 资源，从而提高浏览器的用户响应速率, 同时兼顾任务执行效率。

所以 React 通过Fiber 架构，让这个执行过程变成可被中断。"适时"地让出 CPU 执行权，除了可以让浏览器及时地响应用户的交互，还有其他好处:

- 分批延时对DOM进行操作，避免一次性操作大量 DOM 节点，可以得到更好的用户体验；
- 给浏览器一点喘息的机会，它会对代码进行编译优化（JIT）及进行热代码优化，或者对 reflow 进行修正。

核心思想: Fiber 也称协程或者纤程。它和线程并不一样，协程本身是没有并发或者并行能力的（需要配合线程），它只是一种控制流程的让出机制。让出 CPU 的执行权，让 CPU 能在这段时间执行其他的操作。渲染的过程可以被中断，可以将控制权交回浏览器，让位给高优先级的任务，浏览器空闲后再恢复渲染。

Redux 请求中间件如何处理并发

使用redux-Saga redux-saga是一个管理redux应用异步操作的中间件，用于代替 redux-thunk 的。它通过创建 Sagas 将所有异步操作逻辑存放在一个地方进行集中处理，以此将react中的同步操作与异步操作区分开来，以便于后期的管理与维护。 redux-saga如何处理并发：

- **takeEvery**

可以让多个 saga 任务并行被 fork 执行。

javascript 复制代码

```
import {
    fork,
    take
} from "redux-saga/effects"

const takeEvery = (pattern, saga, ...args) => fork(function*() {
    while (true) {
        const action = yield take(pattern)
        yield fork(saga, ...args.concat(action))
    }
})
```

- **takeLatest**

takeLatest 不允许多个 saga 任务并行地执行。一旦接收到新的发起的 action，它就会取消前面所有 fork 过的任务（如果这些任务还在执行的话）。 在处理 AJAX 请求的时候，如果只希望获取最后那个请求的响应， takeLatest 就会非常有用。

javascript 复制代码

```
import {
    cancel,
    fork,
    take
} from "redux-saga/effects"

const takeLatest = (pattern, saga, ...args) => fork(function*() {
    let lastTask
    while (true) {
        const action = yield take(pattern)
        if (lastTask) {
            yield cancel(lastTask) // 如果任务已经结束，则 cancel 为空操作
        }
        lastTask = yield fork(saga, ...args.concat(action))
    }
})
```

参考 前端进阶面试题详细解答

Component, Element, Instance 之间有什么区别和联系?

- **元素：** 一个元素 `element` 是一个普通对象(plain object)，描述了对于一个DOM节点或者其他组件 `component` ，你想让它在屏幕上呈现成什么样子。元素 `element` 可以在它的属性 `props` 中包含其他元素(译注:用于形成元素树)。创建一个React元素 `element` 成本很低。元素 `element` 创建之后是不可变的。
- **组件：** 一个组件 `component` 可以通过多种方式声明。可以是带有一个 `render()` 方法的类，简单点也可以定义为一个函数。这两种情况下，它都把属性 `props` 作为输入，把返回的一棵元素树作为输出。
- **实例：** 一个实例 `instance` 是你在所写的组件类 `component class` 中使用关键字 `this` 所指向的东西(译注:组件实例)。它用来存储本地状态和响应生命周期事件很有用。

函数式组件(`Functional component`)根本没有实例 `instance` 。类组件(`Class component`)有实例 `instance` ，但是永远也不需要直接创建一个组件的实例，因为React帮我们做了这些。

React 16中新生命周期有哪些

关于 React16 开始应用的新生命周期： 可以看出，React16 自上而下地对生命周期做了另一种维度的解读：

- **Render 阶段**：用于计算一些必要的状态信息。这个阶段可能会被 React 暂停，这一点和 React16 引入的 Fiber 架构（我们后面会重点讲解）是有关的；
- **Pre-commit阶段**：所谓“commit”，这里指的是“更新真正的 DOM 节点”这个动作。所谓 Pre-commit，就是说我在这个阶段其实还并没有去更新真实的 DOM，不过 DOM 信息已经是可以读取的了；
- **Commit 阶段**：在这一步，React 会完成真实 DOM 的更新工作。Commit 阶段，我们可以拿到真实 DOM（包括 refs）。

与此同时，新的生命周期在流程方面，仍然遵循“挂载”、“更新”、“卸载”这三个广义的划分方式。它们分别对应到：

- 挂载过程：
  - **constructor**
  - **getDerivedStateFromProps**
  - **render**
  - **componentDidMount**
- 更新过程：
  - **getDerivedStateFromProps**

    - **shouldComponentUpdate**
    - **render**
    - **getSnapshotBeforeUpdate**
    - **componentDidUpdate**
- 卸载过程：
    - **componentWillUnmount**

Redux 中异步的请求怎么处理

可以在 componentDidmount 中直接进行请求无须借助redux。但是在一定规模的项目中,上述方法很难进行异步流的管理,通常情况下我们会借助redux的异步中间件进行异步处理。redux异步流中间件其实有很多，当下主流的异步中间件有两种redux-thunk、redux-saga。

（1）使用react-thunk中间件

redux-thunk优点:

- 体积小: redux-thunk的实现方式很简单,只有不到20行代码
- 使用简单: redux-thunk没有引入像redux-saga或者redux-observable额外的范式,上手简单

redux-thunk缺陷:

- 样板代码过多: 与redux本身一样,通常一个请求需要大量的代码,而且很多都是重复性质的
- 耦合严重: 异步操作与redux的action偶合在一起,不方便管理
- 功能孱弱: 有一些实际开发中常用的功能需要自己进行封装

使用步骤：

- 配置中间件，在store的创建中配置

```javascript 复制代码
import {createStore, applyMiddleware, compose} from 'redux';
import reducer from './reducer';
import thunk from 'redux-thunk'

// 设置调试工具
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ? window.__REDUX_DEVTOOLS_EXTEN
// 设置中间件
const enhancer = composeEnhancers(
  applyMiddleware(thunk)
```

```
);

const store = createStore(reducer, enhancer);

export default store;
```

- 添加一个返回函数的actionCreator，将异步请求逻辑放在里面

```javascript
/**  发送get请求，并生成相应action，更新store的函数  @param url {string} 请求地址  @param func {functio
// dispatch为自动接收的store.dispatch函数
export const getHttpAction = (url, func) => (dispatch) => {
    axios.get(url).then(function(res){
        const action = func(res.data)
        dispatch(action)
    })
}
```

- 生成action，并发送action

```javascript
componentDidMount(){
    var action = getHttpAction('/getData', getInitTodoItemAction)
    // 发送函数类型的action时，该action的函数体会自动执行
    store.dispatch(action)
}
```

(2) 使用redux-saga中间件

redux-saga优点:

- 异步解耦: 异步操作被被转移到单独 saga.js 中，不再是掺杂在 action.js 或 component.js 中
- action摆脱thunk function: dispatch 的参数依然是一个纯粹的 action (FSA)，而不是充满 “黑魔法” thunk function
- 异常处理: 受益于 generator function 的 saga 实现，代码异常/请求失败 都可以直接通过 try/catch 语法直接捕获处理
- 功能强大: redux-saga提供了大量的Saga 辅助函数和Effect 创建器供开发者使用,开发者无须封装或者简单封装即可使用
- 灵活: redux-saga可以将多个Saga可以串行/并行组合起来,形成一个非常实用的异步flow

- 易测试，提供了各种case的测试方案，包括mock task，分支覆盖等等

redux-saga缺陷:

- 额外的学习成本: redux-saga不仅在使用难以理解的 generator function,而且有数十个API,学习成本远超redux-thunk,最重要的是你的额外学习成本是只服务于这个库的,与redux-observable不同,redux-observable虽然也有额外学习成本但是背后是rxjs和一整套思想
- 体积庞大: 体积略大,代码近2000行，min版25KB左右
- 功能过剩: 实际上并发控制等功能很难用到,但是我们依然需要引入这些代码
- ts支持不友好: yield无法返回TS类型

redux-saga可以捕获action，然后执行一个函数，那么可以把异步代码放在这个函数中，使用步骤如下：

- 配置中间件

```javascript 复制代码
import {createStore, applyMiddleware, compose} from 'redux';
import reducer from './reducer';
import createSagaMiddleware from 'redux-saga'
import TodoListSaga from './sagas'

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ? window.__REDUX_DEVTOOLS_EXTEN
const sagaMiddleware = createSagaMiddleware()

const enhancer = composeEnhancers(
  applyMiddleware(sagaMiddleware)
);

const store = createStore(reducer, enhancer);
sagaMiddleware.run(TodoListSaga)

export default store;
```

- 将异步请求放在sagas.js中

```javascript 复制代码
import {takeEvery, put} from 'redux-saga/effects'
import {initTodoList} from './actionCreator'
import {GET_INIT_ITEM} from './actionTypes'
import axios from 'axios'

function* func(){
```

```
    try{
        // 可以获取异步返回数据
        const res = yield axios.get('/getData')
        const action = initTodoList(res.data)
        // 将action发送到reducer
        yield put(action)
    }catch(e){
        console.log('网络请求失败')
    }
}

function* mySaga(){
    // 自动捕获GET_INIT_ITEM类型的action，并执行func
    yield takeEvery(GET_INIT_ITEM, func)
}

export default mySaga
```

- 发送action

javascript 复制代码

```
componentDidMount(){
  const action = getInitTodoItemAction()
  store.dispatch(action)
}
```

对componentWillReceiveProps 的理解

该方法当 `props` 发生变化时执行，初始化 `render` 时不执行，在这个回调函数里面，你可以根据属性的变化，通过调用 `this.setState()` 来更新你的组件状态，旧的属性还是可以通过 `this.props` 来获取,这里调用更新状态是安全的，并不会触发额外的 `render` 调用。

使用好处： 在这个生命周期中，可以在子组件的render函数执行前获取新的props，从而更新子组件自己的state。 可以将数据请求放在这里进行执行，需要传的参数则从componentWillReceiveProps(nextProps)中获取。而不必将所有的请求都放在父组件中。于是该请求只会在该组件渲染时才会发出，从而减轻请求负担。

componentWillReceiveProps在初始化render的时候不会执行，它会在Component接受到新的状态(Props)时被触发，一般用于父组件状态更新时子组件的重新渲染。

React中的setState和replaceState的区别是什么?

（1）setState() setState()用于设置状态对象，其语法如下：

```javascript
setState(object nextState[, function callback])
```

- nextState，将要设置的新状态，该状态会和当前的state合并
- callback，可选参数，回调函数。该函数会在setState设置成功，且组件重新渲染后调用。

合并nextState和当前state，并重新渲染组件。setState是React事件处理函数中和请求回调函数中触发UI更新的主要方法。

（2）replaceState() replaceState()方法与setState()类似，但是方法只会保留nextState中状态，原state不在nextState中的状态都会被删除。其语法如下：

```javascript
replaceState(object nextState[, function callback])
```

- nextState，将要设置的新状态，该状态会替换当前的state。
- callback，可选参数，回调函数。该函数会在replaceState设置成功，且组件重新渲染后调用。

总结： setState 是修改其中的部分状态，相当于 Object.assign，只是覆盖，不会减少原来的状态。而replaceState 是完全替换原来的状态，相当于赋值，将原来的 state 替换为另一个对象，如果新状态属性减少，那么 state 中就没有这个状态了。

何为 JSX

JSX 是 JavaScript 语法的一种语法扩展，并拥有 JavaScript 的全部功能。JSX 生产 React "元素"，你可以将任何的 JavaScript 表达式封装在花括号里，然后将其嵌入到 JSX 中。在编译完成之后，JSX 表达式就变成了常规的 JavaScript 对象，这意味着你可以在 `if` 语句和 `for` 循环内部使用 JSX，将它赋值给变量，接受它作为参数，并从函数中返回它。

react 强制刷新

component.forceUpdate() 一个不常用的生命周期方法, 它的作用就是强制刷新

官网解释如下

默认情况下，当组件的 state 或 props 发生变化时，组件将重新渲染。如果 render() 方法依赖于其他数据，则可以调用 forceUpdate() 强制让组件重新渲染。

调用 forceUpdate() 将致使组件调用 render() 方法，此操作会跳过该组件的 shouldComponentUpdate()。但其子组件会触发正常的生命周期方法，包括 shouldComponentUpdate() 方法。如果标记发生变化，React 仍将只更新 DOM。

通常你应该避免使用 forceUpdate()，尽量在 render() 中使用 this.props 和 this.state。

shouldComponentUpdate 在初始化 和 forceUpdate 不会执行

在构造函数调用 super 并将 props 作为参数传入的作用

在调用 super() 方法之前，子类构造函数无法使用this引用，ES6 子类也是如此。
将 props 参数传递给 super() 调用的主要原因是在子构造函数中能够通过this.props来获取传入的 props

传递了props

javascript 复制代码

```javascript
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    console.log(this.props); // { name: 'sudheer',age: 30 }
  }
}
```

没传递 props

javascript 复制代码

```javascript
class MyComponent extends React.Component {
  constructor(props) {
    super();
    console.log(this.props); // undefined
    // 但是 Props 参数仍然可用
    console.log(props); // Prints { name: 'sudheer',age: 30 }
  }
  render() {
    // 构造函数外部不受影响
    console.log(this.props); // { name: 'sudheer',age: 30 }
  }
}
```

React中的setState批量更新的过程是什么?

调用 `setState` 时，组件的 `state` 并不会立即改变， `setState` 只是把要修改的 `state` 放入一个队列， `React` 会优化真正的执行时机，并出于性能原因，会将 `React` 事件处理程序中的多次 `React` 事件处理程序中的多次 `setState` 的状态修改合并成一次状态修改。 最终更新只产生一次组件及其子组件的重新渲染，这对于大型应用程序中的性能提升至关重要。

javascript 复制代码

```
this.setState({
  count: this.state.count + 1    ===>    入队，[count+1的任务]
});
this.setState({
  count: this.state.count + 1    ===>    入队，[count+1的任务，count+1的任务]
});
                                         ↓
                                       合并 state，[count+1的任务]
                                         ↓
                                       执行 count+1的任务
```

需要注意的是，只要同步代码还在执行，“攒起来”这个动作就不会停止。（注：这里之所以多次 +1 最终只有一次生效，是因为在同一个方法中多次 setState 的合并动作不是单纯地将更新累加。比如这里对于相同属性的设置，React 只会为其保留最后一次的更新）。

(在构造函数中)调用 super(props) 的目的是什么

在 `super()` 被调用之前，子类是不能使用 `this` 的，在 ES2015 中，子类必须在 `constructor` 中调用 `super()` 。传递 `props` 给 `super()` 的原因则是便于(在子类中)能在 `constructor` 访问 `this.props` 。

React中的状态是什么？它是如何使用的

状态是 React 组件的核心，是数据的来源，必须尽可能简单。基本上状态是确定组件呈现和行为的对象。与props 不同，它们是可变的，并创建动态和交互式组件。可以通过 `this.state()` 访问它们。

react组件的划分业务组件技术组件?

- 根据组件的职责通常把组件分为UI组件和容器组件。

- UI 组件负责 UI 的呈现，容器组件负责管理数据和逻辑。
- 两者通过 `React-Redux` 提供 `connect` 方法联系起来

React如何进行组件/逻辑复用?

> 抛开已经被官方弃用的Mixin,组件抽象的技术目前有三种比较主流:

- 高阶组件:
  - 属性代理
  - 反向继承
- 渲染属性
- react-hooks

React中props.children和React.Children的区别

在React中，当涉及组件嵌套，在父组件中使用 `props.children` 把所有子组件显示出来。如下:

```javascript 复制代码
function ParentComponent(props){
    return (
        <div>
            {props.children}        </div>
    )
}
```

如果想把父组件中的属性传给所有的子组件，需要使用 `React.Children` 方法。

比如，把几个Radio组合起来，合成一个RadioGroup，这就要求所有的Radio具有同样的name属性值。可以这样：把Radio看做子组件，RadioGroup看做父组件，name的属性值在RadioGroup这个父组件中设置。

首先是子组件:

```javascript 复制代码
//子组件
function RadioOption(props) {
  return (
```

```
    <label>
      <input type="radio" value={props.value} name={props.name} />
      {props.label}    </label>
  )
}
```

然后是父组件，不仅需要把它所有的子组件显示出来，还需要为每个子组件赋上name属性和值：

javascript 复制代码

```javascript
//父组件用,props是指父组件的props
function renderChildren(props) {

  //遍历所有子组件
  return React.Children.map(props.children, child => {
    if (child.type === RadioOption)
      return React.cloneElement(child, {
        //把父组件的props.name赋值给每个子组件
        name: props.name
      })
    else
      return child
  })
}
//父组件
function RadioGroup(props) {
  return (
    <div>
      {renderChildren(props)}    </div>
  )
}
function App() {
  return (
    <RadioGroup name="hello">
      <RadioOption label="选项一" value="1" />
      <RadioOption label="选项二" value="2" />
      <RadioOption label="选项三" value="3" />
    </RadioGroup>
  )
}
export default App;
```

以上， `React.Children.map` 让我们对父组件的所有子组件又更灵活的控制。