1. React 面试专题

1.1. React.js是 MVVM 框架吗?

React就是Facebook的一个开源JS框架，专注的层面为View层，不包括数据访问层或者那种Hash路由（不过React 有插件支持），与Angularjs，Emberjs等大而全的框架不同，React专注的中心是Component，即组件。React认为一切页面元 素都可以抽象成组件，比如一个表单，或者表单中的某一项。

React可以作为MVVM中第二个V，也就是View，但是并不是MVVM框架。MVVM一个最显著的特征：双向绑定。React没有这个，它是单向数据绑定的。React是一个单向数据流的库，状态驱动视图。react整体是函数式的思想，把组件设计成纯组件，状态和逻辑通过参数传入，所以在react中，是单向数据流，推崇结合immutable来实现数据不可变。

1.2. hooks用过吗？聊聊react中class组件和函数组件的区别

类组件是使用ES6 的 class 来定义的组件。 函数组件是接收一个单一的 `props` 对象并返回一个React元素。

关于React的两套API（类（class）API 和基于函数的钩子（hooks） API）。官方推荐使用钩子（函数），而不是类。因为钩子更简洁，代码量少，用起来比较"轻"，而类比较"重"。而且，钩子是函数，更符合 React 函数式的本质。

函数一般来说，只应该做一件事，就是返回一个值。 如果你有多个操作，每个操作应该写成一个单独的函数。而且，数据的状态应该与操作方法分离。根据函数这种理念，React 的函数组件只应该做一件事情：返回组件的 HTML 代码，而没有其他的功能。函数的返回结果只依赖于它的参数。不改变函数体外部数据、函数执行过程里面没有副作用。

类（class）是数据和逻辑的封装。 也就是说，组件的状态和操作方法是封装在一起的。如果选择了类的写法，就应该把相关的数据和操作，都写在同一个 class 里面。

类组件的缺点：

大型组件很难拆分和重构，也很难测试。\

业务逻辑分散在组件的各个方法之中，导致重复逻辑或关联逻辑。\

组件类引入了复杂的编程模式，比如 render props 和高阶组件。\

难以理解的 class，理解 JavaScript 中 `this` 的工作方式。

区别:

函数组件的性能比类组件的性能要高，因为类组件使用的时候要实例化，而函数组件直接执行函数取返回结果即可。

1.状态的有无\

hooks出现之前，函数组件`没有实例`，`没有生命周期`，`没有state`，`没有this`，所以我们称函数组件为无状态组件。 hooks出现之前，react中的函数组件通常只考虑负责UI的渲染，没有自身的状态没有业务逻辑代码，是一个纯函数。它的输出只由参数props决定，不受其他任何因素影响。

2.调用方式的不同\

函数组件重新渲染，将重新调用组件方法返回新的react元素。类组件重新渲染将new一个新的组件实例，然后调用render类方法返回react元素，这也说明为什么类组件中this是可变的。

3.因为调用方式不同，在函数组件使用中会出现问题\

在操作中改变状态值，类组件可以获取最新的状态值，而函数组件则会按照顺序返回状态值

React Hooks（钩子的作用）

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。

React Hooks的几个常用钩子:

1. `useState()` //状态钩子
2. `useContext()` //共享状态钩子
3. `useReducer()` //action 钩子
4. `useEffect()` //副作用钩子

还有几个不常见的大概的说下，后续会专门写篇文章描述下

- 1.useCallback 记忆函数 一般把**函数式组件理解为class组件render函数的语法糖**，所以每次重新渲染的时候，函数式组件内部所有的代码都会重新执行一遍。而有了useCallback 就不一样了，你可以通过 useCallback 获得一个记忆后的函数。

```
function App() {
  const memoizedHandleClick = useCallback(() => {
    console.log('Click happened')
  }, []); // 空数组代表无论什么情况下该函数都不会发生改变
  return <SomeComponent onClick={memoizedHandleClick}>Click
Me</SomeComponent>;
}
```

第二个参数传入一个数组，数组中的每一项一旦值或者引用发生改变，useCallback 就会重新返回一个新的记忆函数提供给后面进行渲染。

- 2.useMemo 记忆组件 useCallback 的功能完全可以由 useMemo 所取代，如果你想通过使用 useMemo 返回一个记忆函数也是完全可以的。 唯一的区别是：**useCallback 不会执行第一个参数函数，而是将它返回给你，而 useMemo 会执行第一个函数并且将函数执行结果返回给你**。\
  所以 useCallback 常用记忆事件函数，生成记忆后的事件函数并传递给子组件使用。而 useMemo 更适合经过函数计算得到一个确定的值，比如记忆组件。
- 3.useRef 保存引用值

  useRef 跟 createRef 类似，都可以用来生成对 DOM 对象的引用。useRef 返回的值传递给组件或者 DOM 的 ref 属性，就可以通过 ref.current 值**访问组件或真实的 DOM 节点，重点是组件也是可以访问到的**，从而可以对 DOM 进行一些操作，比如监听事件等等。
- 4.useImperativeHandle 穿透 Ref

  通过 useImperativeHandle 用于让父组件获取子组件内的索引
- 5.useLayoutEffect 同步执行副作用

  大部分情况下，使用 useEffect 就可以帮我们处理组件的副作用，但是如果想要同步调用一些副作用，比如对 DOM 的操作，就需要使用 useLayoutEffect，useLayoutEffect 中的副作用会在 DOM 更新之后同步执行。

  useEffect和useLayoutEffect有什么区别：简单来说就是调用时机不同，useLayoutEffect和原来componentDidMount&componentDidUpdate一致，在react完成DOM更新后马上同步调用的代码，会阻塞页面渲染。而useEffect是会在整个页面渲染完才会调用的代码。`官方建议优先使用useEffect`

1.3. React 组件通信方式

react组件间通信常见的几种情况:

- 1. 父组件向子组件通信
- 2. 子组件向父组件通信
- 3. 跨级组件通信
- 4. 非嵌套关系的组件通信

1.3.1. 1）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。父传子是在父组件中直接绑定一个正常的属性，这个属性就是指具体的值，在子组件中，用props就可以获取到这个值

```
// 子组件：Child
const Child = props =>{
  return <p>{props.name}</p>
}

// 父组件 Parent
const Parent = ()=>{
    return <Child name="京程一灯"></Child>
}
```

1.3.2. 2）子组件向父组件通信

props+回调的方式，使用公共组件进行状态提升。子传父是先在父组件上绑定属性设置为一个函数，当子组件需要给父组件传值的时候，则通过props调用该函数将参数传入到该函数当中，此时就可以在父组件中的函数中接收到该参数了，这个参数则为子组件传过来的值

```
// 子组件：Child
const Child = props =>{
  const cb = msg =>{
      return ()=>{
          props.callback(msg)
      }
  }
  return (
      <button onClick={cb("京程一灯欢迎你！")}>京程一灯欢迎你</button>
  )
}

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

1.3.3. 3）跨级组件通信

即父组件向子组件的子组件通信，向更深层子组件通信。

- 使用props，利用中间组件层层传递,但是如果父组件结构较深，那么中间每一层组件都要去传递props，增加了复杂度，并且这些props并不是中间组件自己需要的。
- 使用context，context相当于一个大容器，我们可以把要通信的内容放在这个容器中，这样不管嵌套多深，都可以随意取用，对于跨越多层的全局数据可以使用context实现。

```
// context方式实现跨级组件通信
// 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

const BatteryContext = createContext();

//  子组件的子组件
class GrandChild extends Component {
    render(){
        return (
            <BatteryContext.Consumer>
                {
                    color => <h1 style={{"color":color}}>我是红色的:{color}</h1>
                }
            </BatteryContext.Consumer>
        )
    }
}

//  子组件
const Child = () =>{
    return (
        <GrandChild/>
    )
}
// 父组件
class Parent extends Component {
      state = {
          color:"red"
      }
      render(){
          const {color} = this.state
          return (
          <BatteryContext.Provider value={color}>
              <Child></Child>
          </BatteryContext.Provider>
          )
      }
}
```

1.3.4. 4）非嵌套关系的组件通信

即没有任何包含关系的组件，包括兄弟组件以及不在同一个父级中的非兄弟组件。

- 1. 可以使用自定义事件通信（发布订阅模式），使用pubsub-js
- 2. 可以通过redux等进行全局状态管理
- 3. 如果是兄弟组件通信，可以找到这两个兄弟节点共同的父节点，结合父子间通信方式进行通信。
- 4. 也可以new一个 Vue 的 EventBus,进行事件监听，一边执行监听，一边执行新增VUE的eventBus 就是发布订阅模式，是可以在React中使用的;

1.4. setState 既存在异步情况也存在同步情况

1.异步情况 在`React事件当中是异步操作`

2.同步情况 如果是在`setTimeout事件或者自定义的dom事件`中，都是同步的

```
//setTimeout事件
import React,{ Component } from "react";
class Count extends Component{
    constructor(props){
        super(props);
        this.state = {
            count:0
        }
    }

    render(){
        return (
            <>
                <p>count:{this.state.count}</p>
                <button onClick={this.btnAction}>增加</button>
            </>
        )
    }

    btnAction = ()=>{
        //不能直接修改state，需要通过setState进行修改
        //同步
        setTimeout(()=>{
            this.setState({
                count: this.state.count + 1
            });
            console.log(this.state.count);
        })
```

```
    }
}

export default Count;
```

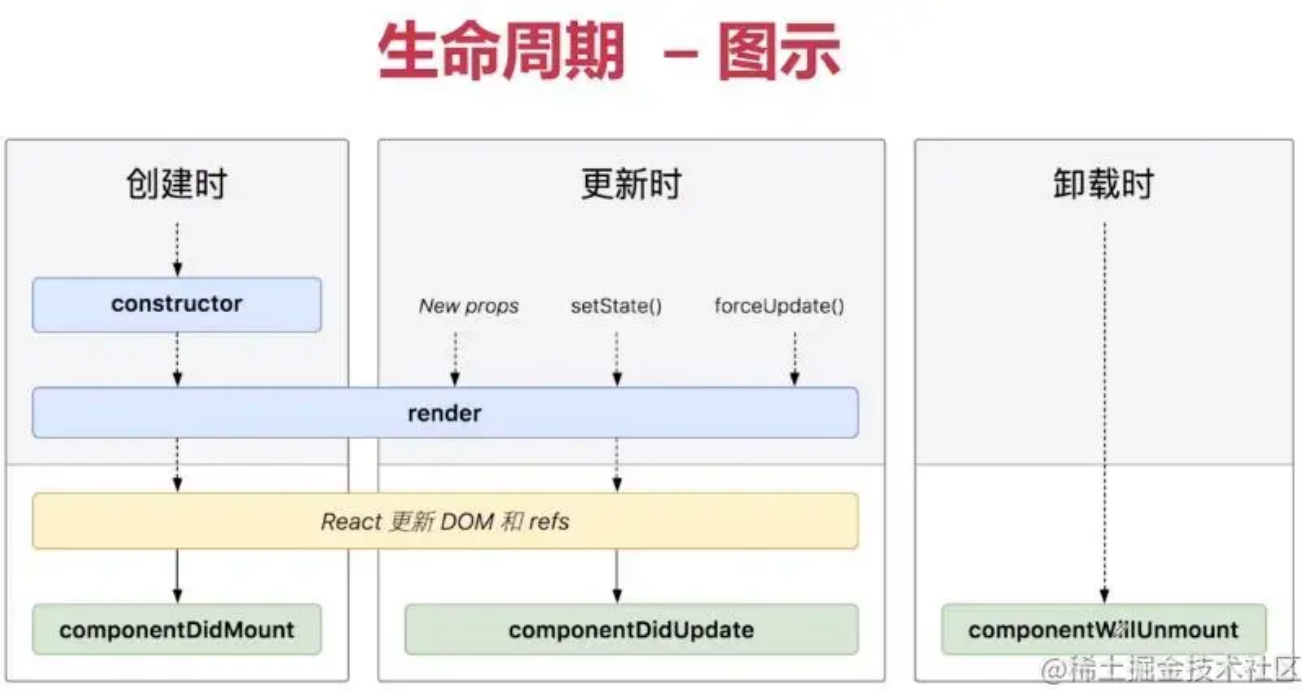
```
//自定义dom事件
import React,{ Component } from "react";
class Count extends Component{
    constructor(props){
        super(props);
        this.state = {
            count:0
        }
    }

    render(){
        return (
            <>
                <p>count:{this.state.count}</p>
                <button id="btn">绑定点击事件</button>
            </>
        )
    }

    componentDidMount(){
        //自定义dom事件，也是同步修改
        document.querySelector('#btn').addEventListener('click',()=>{
            this.setState({
                count: this.state.count + 1
            });
            console.log(this.state.count);
        });
    }
}

export default Count;
```

1.5. 生命周期



安装
当组件的实例被创建并插入到 DOM 中时，这些方法按以下顺序调用：

```
constructor()
static getDerivedStateFromProps()
render()
componentDidMount()
```

更新中
更新可能由道具或状态的更改引起。当重新渲染组件时，这些方法按以下顺序调用：

```
static getDerivedStateFromProps()
shouldComponentUpdate()
render()
getSnapshotBeforeUpdate()
componentDidUpdate()
```

卸载
当组件从 DOM 中移除时调用此方法：

```
componentWillUnmount()
```

1.6. 说一下 react-fiber

1.6.1. 1）背景

react-fiber 产生的根本原因，是`大量的同步计算任务阻塞了浏览器的 UI 渲染`。默认情况下，JS 运算、页面布局和页面绘制都是运行在浏览器的主线程当中，他们之间是互斥的关系。如果 JS 运算持续占用主线程，页面就没法得到及时的更新。当我们调用`setState`更新页面的时候，React 会遍历应用的所有节点，计算出差异，然后再更新 UI。如果页面元素很多，整个过程占用的时机就可能超过 16 毫秒，就容易出现掉帧的现象。

1.6.2. 2）实现原理

- react内部运转分三层：
    - Virtual DOM 层，描述页面长什么样。
    - Reconciler 层，负责调用组件生命周期方法，进行 Diff 运算等。
    - Renderer 层，根据不同的平台，渲染出相应的页面，比较常见的是 ReactDOM 和 ReactNative。

`Fiber 其实指的是一种数据结构，它可以用一个纯 JS 对象来表示`：

```
const fiber = {
    stateNode,    // 节点实例
    child,        // 子节点
    sibling,      // 兄弟节点
    return,       // 父节点
}
```

- 为了实现不卡顿，就需要有一个调度器 (Scheduler) 来进行任务分配。优先级高的任务（如键盘输入）可以打断优先级低的任务（如Diff）的执行，从而更快的生效。任务的优先级有六种：
    - synchronous，与之前的Stack Reconciler操作一样，同步执行
    - task，在next tick之前执行
    - animation，下一帧之前执行
    - high，在不久的将来立即执行
    - low，稍微延迟执行也没关系
    - offscreen，下一次render时或scroll时才执行
- Fiber Reconciler（react ）执行过程分为2个阶段：

    - 阶段一，生成 Fiber 树，得出需要更新的节点信息。这一步是一个渐进的过程，可以被打断。阶段一可被打断的特性，让优先级更高的任务先执行，从框架层面大大降低了页面掉帧的概率。
    - 阶段二，将需要更新的节点一次过批量更新，这个过程不能被打断。
- Fiber树：React 在 render 第一次渲染时，会通过 React.createElement 创建一颗 Element 树，可以称之为 Virtual DOM Tree，由于要记录上下文信息，加入了 Fiber，每一个 Element 会对应一个 Fiber Node，将 Fiber Node 链接起来的结构成为 Fiber Tree。Fiber Tree 一个重要的特点是链表结构，将递归遍历编程循环遍历，然后配合 requestIdleCallback API, 实现任务拆分、中断与恢复。

从Stack Reconciler到Fiber Reconciler，源码层面其实就是干了一件递归改循环的事情
传送门 ☞# 深入了解 Fiber

1.7. Portals

Portals 提供了一种一流的方式来将子组件渲染到存在于父组件的 DOM 层次结构之外的 DOM 节点中。结构不受外界的控制的情况下就可以使用portals进行创建

1.8. 何时要使用异步组件？如和使用异步组件

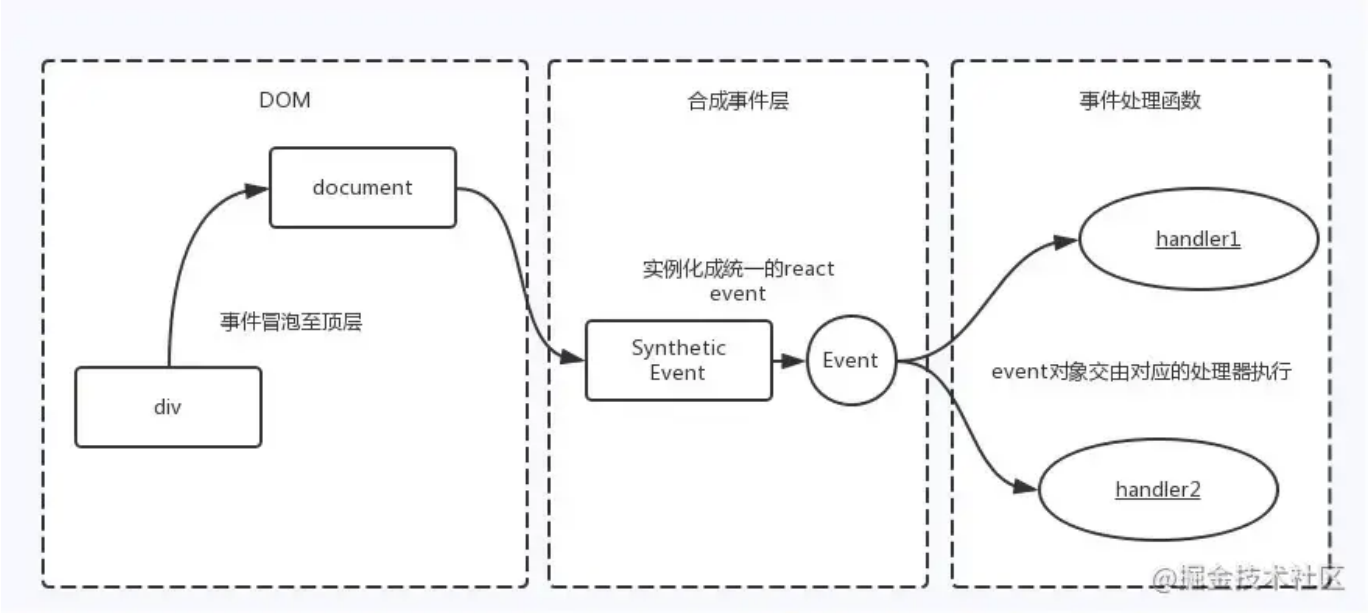
- 加载大组件的时候
- 路由异步加载的时候

react 中要配合 Suspense 使用

```
// 异步懒加载
const Box = lazy(()=>import('./components/Box'));
// 使用组件的时候要用suspense进行包裹
<Suspense fallback={<div>loading...</div>}>
    {show && <Box/>}
</Suspense>
```

1.9. React 事件绑定原理

React并不是将click事件绑在该div的真实DOM上，而是`在document处监听所有支持的事件`，当事件发生并冒泡至document处时，React将事件内容封装并交由真正的处理函数运行。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。\
另外冒泡到 document 上的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent）。因此我们如果不想要事件冒泡的话，调用 event.stopPropagation 是无效的，而应该调用 `event.preventDefault` 。



1.10. React.lazy() 实现的原理

React的懒加载示例：

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

React.lazy 原理

以下 React 源码基于 16.8.0 版本

React.lazy 的源码实现如下：

```
export function lazy<T, R>(ctor: () => Thenable<T, R>): LazyComponent {
  let lazyType = {
    $$typeof: REACT_LAZY_TYPE,
    _ctor: ctor,
    // React uses these fields to store the result.
    _status: -1,
    _result: null,
  };
  return lazyType;
}
```

可以看到其返回了一个 LazyComponent 对象。

而对于 LazyComponent 对象的解析：

```
case LazyComponent: {
  const elementType = workInProgress.elementType;
  return mountLazyComponent(
    current,
    workInProgress,
    elementType,
    updateExpirationTime,
    renderExpirationTime,
  );
}
```

```
function mountLazyComponent(
  _current,
  workInProgress,
  elementType,
  updateExpirationTime,
  renderExpirationTime,
) {
  ...
  let Component = readLazyComponentType(elementType);
  ...
}
```

```
// Pending = 0, Resolved = 1, Rejected = 2
export function readLazyComponentType<T>(lazyComponent: LazyComponent<T>): T {
  const status = lazyComponent._status;
  const result = lazyComponent._result;
  switch (status) {
```

```
    case Resolved: {
      const Component: T = result;
      return Component;
    }
    case Rejected: {
      const error: mixed = result;
      throw error;
    }
    case Pending: {
      const thenable: Thenable<T, mixed> = result;
      throw thenable;
    }
    default: { // lazyComponent 首次被渲染
      lazyComponent._status = Pending;
      const ctor = lazyComponent._ctor;
      const thenable = ctor();
      thenable.then(
        moduleObject => {
          if (lazyComponent._status === Pending) {
            const defaultExport = moduleObject.default;
            lazyComponent._status = Resolved;
            lazyComponent._result = defaultExport;
          }
        },
        error => {
          if (lazyComponent._status === Pending) {
            lazyComponent._status = Rejected;
            lazyComponent._result = error;
          }
        },
      );
      // Handle synchronous thenables.
      switch (lazyComponent._status) {
        case Resolved:
          return lazyComponent._result;
        case Rejected:
          throw lazyComponent._result;
      }
      lazyComponent._result = thenable;
      throw thenable;
    }
  }
}
```

注：如果 readLazyComponentType 函数多次处理同一个 lazyComponent，则可能进入 Pending、Rejected等 case 中。

从上述代码中可以看出，对于最初 React.lazy() 所返回的 LazyComponent 对象，其 _status 默认是 -1，所以首次渲染时，会进入 readLazyComponentType 函数中的 default 的逻辑，这里才会真正异步执行 import(url)操作，由于并未等待，随后会检查模块是否 Resolved，如果已经Resolved了（已经加载完毕）则直接返回 moduleObject.default（动态加载的模块的默认导出），否则将通过 throw 将 thenable 抛出到上层。

为什么要 throw 它？这就要涉及到 Suspense 的工作原理，我们接着往下分析。

Suspense 原理

由于 React 捕获异常并处理的代码逻辑比较多，这里就不贴源码，感兴趣可以去看 throwException 中的逻辑，其中就包含了如何处理捕获的异常。简单描述一下处理过程，React 捕获到异常之后，会判断异常是不是一个 thenable，如果是则会找到 SuspenseComponent ，如果 thenable 处于 pending 状态，则会将其 children 都渲染成 fallback 的值，一旦 thenable 被 resolve 则 SuspenseComponent 的子组件会重新渲染一次。

为了便于理解，我们也可以用 componentDidCatch 实现一个自己的 Suspense 组件，如下：

```
class Suspense extends React.Component {
  state = {
    promise: null
  }

  componentDidCatch(err) {
    // 判断 err 是否是 thenable
    if (err !== null && typeof err === 'object' && typeof err.then ===
'function') {
      this.setState({ promise: err }, () => {
        err.then(() => {
          this.setState({
            promise: null
          })
        })
      })
    }
  }

  render() {
    const { fallback, children } = this.props
    const { promise } = this.state
    return <>{ promise ? fallback : children }</>
  }
```

```
}
```

至此，我们分析完了 React 的懒加载原理。简单来说，React利用 React.lazy与import() 实现了渲染时的动态加载 ，并利用Suspense来处理异步加载资源时页面应该如何显示的问题。

参考传送门☞ React Lazy 的实现原理