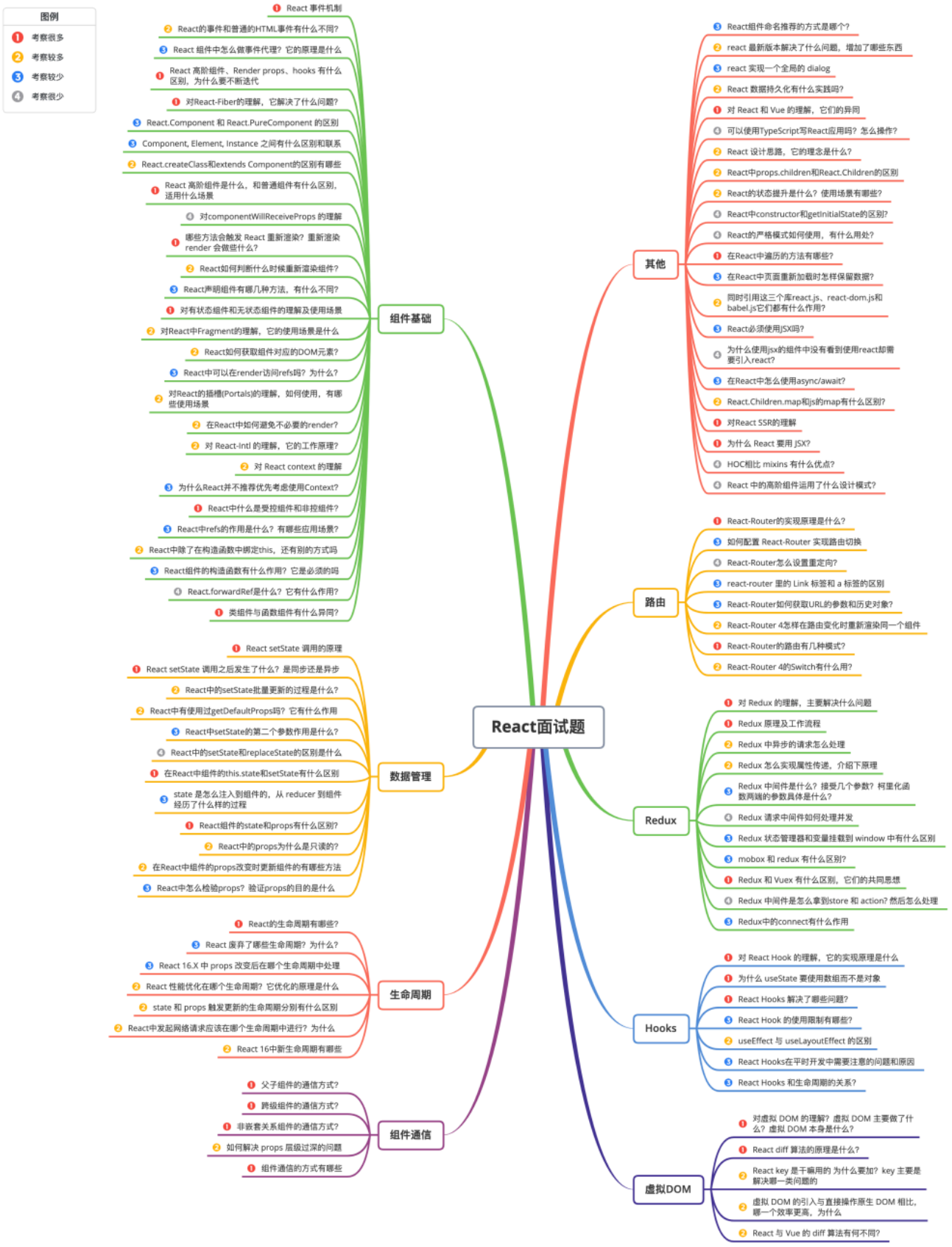


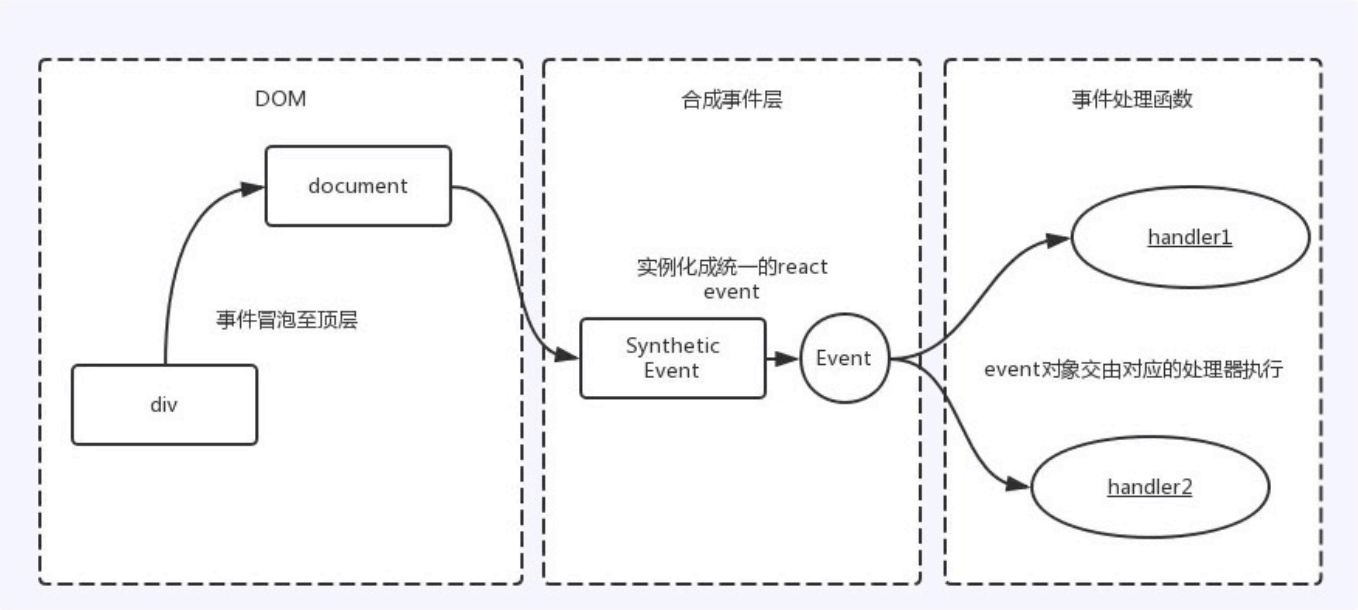
一、组件基础

1. React 事件机制

```
<div onClick={this.handleClick.bind(this)}>点我</div>
```

React并不是将click事件绑定到了div的真实DOM上，而是在document处监听了所有的事件，当事件发生并且冒泡到document处的时候，React将事件内容封装并交由真正的处理函数运行。这样的方式不仅仅减少了内存的消耗，还能在组件挂载销毁时统一订阅和移除事件。

除此之外，冒泡到document上的事件也不是原生的浏览器事件，而是由react自己实现的合成事件（SyntheticEvent）。因此如果不想要是事件冒泡的话应该调用event.preventDefault()方法，而不是调用event.stopPropgation()方法。



JSX 上写的事件并没有绑定在对应的真实 DOM 上，而是通过事件代理的方式，将所有的事件都统一绑定在了 `document` 上。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`。

实现合成事件的目的如下：

- 合成事件首先抹平了浏览器之间的兼容问题，另外这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力；
- 对于原生浏览器事件来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，那么就需要分配很多的事件对象，造成高额的内存分配问题。但是对于合成事件来说，有一个事件池专门来管理它们的创建和销毁，当事件需要被使用时，就会从池子中复用对象，事件回调结束后，就会销毁事件对象上的属性，从而便于下次复用事件对象。

2. React的事件和普通的HTML事件有什么不同?

区别：

- 对于事件名称命名方式，原生事件为全小写，react 事件采用小驼峰；
- 对于事件函数处理语法，原生事件为字符串，react 事件为函数；

- react 事件不能采用 return false 的方式来阻止浏览器的默认行为，而必须要地明确地调用`preventDefault()`来阻止默认行为。

合成事件是 react 模拟原生 DOM 事件所有能力的一个事件对象，其优点如下：

- 兼容所有浏览器，更好的跨平台；
- 将事件统一存放在一个数组，避免频繁的新增与删除（垃圾回收）。
- 方便 react 统一管理和事务机制。

事件的执行顺序为原生事件先执行，合成事件后执行，合成事件会冒泡绑定到 document 上，所以尽量避免原生事件与合成事件混用，如果原生事件阻止冒泡，可能会导致合成事件不执行，因为需要冒泡到document 上合成事件才会执行。

3. 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。

4. React 高阶组件、Render props、hooks 有什么区别，为什么要不断迭代

这三者是目前react解决代码复用的主要方式：

- 高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。具体而言，高阶组件是参数为组件，返回值为新组件的函数。
- render props是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术，更具体的说，render prop 是一个用于告知组件需要渲染什么内容的函数 prop。
- 通常，render props 和高阶组件只渲染一个子节点。让 Hook 来服务这个使用场景更加简单。这两种模式仍有用武之地，（例如，一个虚拟滚动条组件或许会有一个 renderItem 属性，或是一个可见的容器组件或许会有它自己的 DOM 结构）。但在大部分场景下，Hook 足够了，并且能够帮助减少嵌套。

（1）HOC

官方解释:

> 高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

简言之，HOC是一种组件的设计模式，HOC接受一个组件和额外的参数（如果需要），返回一个新的组件。HOC 是纯函数，没有副作用。

```
// hoc的定义
function withSubscription(WrappedComponent, selectData) {
  return class extends React.Component {
```

```
    constructor(props) {
      super(props);
      this.state = {
        data: selectData(DataSource, props)
      };
    }
    // 一些通用的逻辑处理
    render() {
      // ... 并使用新数据渲染被包装的组件!
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };

// 使用
const BlogPostWithSubscription = withSubscription(BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id));
```

HOC的优缺点:

- 优点: 逻辑复用、不影响被包裹组件的内部逻辑。
- 缺点: hoc传递给被包裹组件的props容易和被包裹后的组件重名，进而被覆盖

** (2) **Render props**

官方解释:

> "render prop"是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术

具有render prop 的组件接受一个返回React元素的函数，将render的渲染逻辑注入到组件内部。在这里，"render"的命名可以是任何其他有效的标识符。

```
// DataProvider组件内部的渲染逻辑如下
class DataProvider extends React.Components {
     state = {
    name: 'Tom'
  }

    render() {
    return (
        <div>
          <p>共享数据组件自己内部的渲染逻辑</p>
          { this.props.render(this.state) }
      </div>
    );
  }
}

// 调用方式
<DataProvider render={data => (
  <h1>Hello {data.name}</h1>
)}/>
```

由此可以看到，render props的优缺点也很明显:

- 优点：数据共享、代码复用，将组件内的state作为props传递给调用者，将渲染逻辑交给调用者。
- 缺点：无法在 return 语句外访问数据、嵌套写法不够优雅

** （3）**Hooks**

官方解释:

> 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。通过自定义hook，可以复用代码逻辑。

```
// 自定义一个获取订阅数据的hook
function useSubscription() {
  const data = DataSource.getComments();
  return [data];
}
//
function CommentList(props) {
  const {data} = props;
  const [subData] = useSubscription();
    ...
}
// 使用
<CommentList data='hello' />
```

以上可以看出，hook解决了hoc的prop覆盖的问题，同时使用的方式解决了render props的嵌套地狱的问题。hook的优点如下:

- 使用直观；
- 解决hoc的prop 重名问题；
- 解决render props 因共享数据 而出现嵌套地狱的问题；
- 能在return之外使用数据的问题。

需要注意的是：hook只能在组件顶层使用，不可在分支语句中使用。

总结:

Hoc、render props和hook都是为了解决代码复用的问题，但是hoc和render props都有特定的使用场景和明显的缺点。hook是react16.8更新的新的API，让组件逻辑复用更简洁明了，同时也解决了hoc和render props的一些缺点。

5. 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。

6. React.Component 和 React.PureComponent 的区别

PureComponent表示一个纯组件，可以用来优化React程序，减少render函数执行的次数，从而提高组件的性能。

在React中，当prop或者state发生变化时，可以通过在shouldComponentUpdate生命周期函数中执行return false来阻止页面的更新，从而减少不必要的render执行。React.PureComponent会自动执行shouldComponentUpdate。

不过，pureComponent中的 shouldComponentUpdate() 进行的是**浅比较**，也就是说如果是引用数据类型的数据，只会比较不是同一个地址，而不会比较这个地址里面的数据是否一致。浅比较会忽略属性和或状态突变情况，其实也就是数据引用指针没有变化，而数据发生改变的时候render是不会执行的。如果需要重新渲染那么就需要重新开辟空间引用数据。PureComponent一般会用在一些纯展示组件上。

使用pureComponent的**好处**：当组件更新时，如果组件的props或者state都没有改变，render函数就不会触发。省去虚拟DOM的生成和对比过程，达到提升性能的目的。这是因为react自动做了一层浅比较。

7. 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。

8. React.createClass和extends Component的区别有哪些?

React.createClass和extends Component的bai区别主要在于：

（1）语法区别

- createClass本质上是一个工厂函数，extends的方式更加接近最新的ES6规范的class写法。两种方式在语法上的差别主要体现在方法的定义和静态属性的声明上。

- createClass方式的方法定义使用逗号，隔开，因为creatClass本质上是一个函数，传递给它的是一个Object；而class的方式定义方法时务必谨记不要使用逗号隔开，这是ES6 class的语法规范。

（2）propType 和 getDefaultProps

- React.createClass：通过proTypes对象和getDefaultProps()方法来设置和获取props.
- React.Component：通过设置两个属性propTypes和defaultProps

（3）状态的区别

- React.createClass：通过getInitialState()方法返回一个包含初始值的对象
- React.Component：通过constructor设置初始状态

（4）this区别

- React.createClass：会正确绑定this
- React.Component：由于使用了 ES6，这里会有些微不同，属性并不会自动绑定到 React 类的实例上。

（5）Mixins

- React.createClass：使用 React.createClass 的话，可以在创建组件时添加一个叫做 mixins 的属性，并将可供混合的类的集合以数组的形式赋给 mixins。
- 如果使用 ES6 的方式来创建组件，那么 `React mixins` 的特性将不能被使用了。

9. React 高阶组件是什么，和普通组件有什么区别，适用什么场景

官方解释:

> 高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

高阶组件（HOC）就是一个函数，且该函数接受一个组件作为参数，并返回一个新的组件，它只是一种组件的设计模式，这种设计模式是由react自身的组合性质必然产生的。我们将它们称为纯组件，因为它们可以接受任何动态提供的子组件，但它们不会修改或复制其输入组件中的任何行为。

```
// hoc的定义
function withSubscription(WrappedComponent, selectData) {
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        data: selectData(DataSource, props)
      };
    }
    // 一些通用的逻辑处理
    render() {
      // ... 并使用新数据渲染被包装的组件!
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };

// 使用
```

```
const BlogPostWithSubscription = withSubscription(BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id));
```

1）HOC的优缺点

- 优点: 逻辑服用、不影响被包裹组件的内部逻辑。
- 缺点:hoc传递给被包裹组件的props容易和被包裹后的组件重名，进而被覆盖

2）适用场景

- 代码复用，逻辑抽象
- 渲染劫持
- State 抽象和更改
- Props 更改

3）具体应用例子

- **权限控制：**利用高阶组件的 **条件渲染** 特性可以对页面进行权限控制，权限控制一般分为两个维度：页面级别和 页面元素级别

```
// HOC.js
function withAdminAuth(WrappedComponent) {
    return class extends React.Component {
        state = {
            isAdmin: false,
        }
        async UNSAFE_componentWillMount() {
            const currentRole = await getCurrentUserRole();
            this.setState({
                isAdmin: currentRole === 'Admin',
            });
        }
        render() {
            if (this.state.isAdmin) {
                return <WrappedComponent {...this.props} />;
            } else {
                return (<div>您没有权限查看该页面，请联系管理员！</div>);
            }
        }
    };
}

// pages/page-a.js
class PageA extends React.Component {
    constructor(props) {
        super(props);
        // something here...
    }
    UNSAFE_componentWillMount() {
        // fetching data
    }
```

```
    render() {
        // render page with data
    }
}
export default withAdminAuth(PageA);


// pages/page-b.js
class PageB extends React.Component {
    constructor(props) {
        super(props);
    // something here...
        }
    UNSAFE_componentWillMount() {
    // fetching data
    }
    render() {
    // render page with data
    }
}
export default withAdminAuth(PageB);
```

- **组件渲染性能追踪：**借助父组件子组件生命周期规则捕获子组件的生命周期，可以方便的对某个组件的渲染时间进行记录:

```
class Home extends React.Component {
        render() {
            return (<h1>Hello World.</h1>);
        }
    }
    function withTiming(WrappedComponent) {
        return class extends WrappedComponent {
            constructor(props) {
                super(props);
                this.start = 0;
                this.end = 0;
            }
            UNSAFE_componentWillMount() {
                super.componentWillMount && super.componentWillMount();
                this.start = Date.now();
            }
            componentDidMount() {
                super.componentDidMount && super.componentDidMount();
                this.end = Date.now();
                console.log(`${WrappedComponent.name} 组件渲染时间为 ${this.end -
this.start} ms`);
            }
            render() {
                return super.render();
            }
        };
```

```
    }

    export default withTiming(Home);
```

注意：withTiming 是利用 反向继承 实现的一个高阶组件，功能是计算被包裹组件（这里是 Home 组件）的渲染时间。

- **页面复用**

```
const withFetching = fetching => WrappedComponent => {
    return class extends React.Component {
        state = {
            data: [],
        }
        async UNSAFE_componentWillMount() {
            const data = await fetching();
            this.setState({
                data,
            });
        }
        render() {
            return <WrappedComponent data={this.state.data} {...this.props} />;
        }
    }
}

// pages/page-a.js
export default withFetching(fetching('science-fiction'))(MovieList);
// pages/page-b.js
export default withFetching(fetching('action'))(MovieList);
// pages/page-other.js
export default withFetching(fetching('some-other-type'))(MovieList);
```

10. 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。

11. 哪些方法会触发 React 重新渲染？重新渲染 render 会做些什么？

（1）哪些方法会触发 react 重新渲染?

- **setState（）方法被调用**

setState 是 React 中最常用的命令，通常情况下，执行 setState 会触发 render。但是这里有个点值得关注，执行 setState 的时候不一定会重新渲染。当 setState 传入 null 时，并不会触发 render。

```
class App extends React.Component {
  state = {
    a: 1
  };

  render() {
    console.log("render");
    return (
      <React.Fragement>
        <p>{this.state.a}</p>
        <button
          onClick={() => {
            this.setState({ a: 1 }); // 这里并没有改变 a 的值
          }}
        >
          Click me
        </button>
        <button onClick={() => this.setState(null)}>setState null</button>
        <Child />
      </React.Fragement>
    );
  }
}
```

- **父组件重新渲染**

只要父组件重新渲染了，即使传入子组件的 props 未发生变化，那么子组件也会重新渲染，进而触发 render

（2）重新渲染 render 会做些什么?

- 会对新旧 VNode 进行对比，也就是我们所说的Diff算法。
- 对新旧两棵树进行一个深度优先遍历，这样每一个节点都会一个标记，在到深度遍历的时候，每遍历到一和个节点，就把该节点和新的节点树进行对比，如果有差异就放到一个对象里面
- 遍历差异对象，根据差异的类型，根据对应对规则更新VNode

React 的处理 render 的基本思维模式是每次一有变动就会去重新渲染整个应用。在 Virtual DOM 没有出现之前，最简单的方法就是直接调用 innerHTML。Virtual DOM厉害的地方并不是说它比直接操作 DOM 快，而是说不管数据怎么变，都会尽量以最小的代价去更新 DOM。React 将 render 函数返回的虚拟 DOM 树与老的进行比较，从而确定 DOM 要不要更新、怎么更新。当 DOM 树很大时，遍历两棵树进行各种比对还是相当耗性能的，特别是在顶层 setState 一个微小的修改，默认会去遍历整棵树。尽管 React 使用高度优化的 Diff 算法，但是这个过程仍然会损耗性能.

（3）forceUpdate

12. React如何判断什么时候重新渲染组件?

组件状态的改变可以因为`props`的改变，或者直接通过`setState`方法改变。组件获得新的状态，然后React决定是否应该重新渲染组件。只要组件的state发生变化，React就会对组件进行重新渲染。这是因为React中的`shouldComponentUpdate`方法默认返回`true`，这就是导致每次更新都重新渲染的原因。

当React将要渲染组件时会执行`shouldComponentUpdate`方法来看它是否返回`true`（组件应该更新，也就是重新渲染）。所以需要重写`shouldComponentUpdate`方法让它根据情况返回`true`或者`false`来告诉React什么时候重新渲染什么时候跳过重新渲染。

13. React声明组件有哪几种方法，有什么不同?

React 声明组件的三种方式：

- 函数式定义的无状态组件
- ES5原生方式`React.createClass`定义的组件
- ES6形式的`extends React.Component`定义的组件

（1）无状态函数式组件

它是为了创建纯展示组件，这种组件只负责根据传入的props来展示，不涉及到state状态的操作

组件不会被实例化，整体渲染性能得到提升，不能访问this对象，不能访问生命周期的方法

（2）ES5 原生方式 React.createClass // RFC

React.createClass会自绑定函数方法，导致不必要的性能开销，增加代码过时的可能性。

（3）E6继承形式 React.Component // RCC

目前极为推荐的创建有状态组件的方式，最终会取代React.createClass形式；相对于 React.createClass可以更好实现代码复用。

无状态组件相对于于后者的区别：

与无状态组件相比，React.createClass和React.Component都是创建有状态的组件，这些组件是要被实例化的，并且可以访问组件的生命周期方法。

React.createClass与React.Component区别：

① 函数this自绑定

- React.createClass创建的组件，其每一个成员函数的this都有React自动绑定，函数中的this会被正确设置。
- React.Component创建的组件，其成员函数不会自动绑定this，需要开发者手动绑定，否则this不能获取当前组件实例对象。

② 组件属性类型propTypes及其默认props属性defaultProps配置不同

- React.createClass在创建组件时，有关组件props的属性类型及组件默认的属性会作为组件实例的属性来配置，其中defaultProps是使用getDefaultProps的方法来获取默认组件属性的
- React.Component在创建组件时配置这两个对应信息时，他们是作为组件类的属性，不是组件实例的属性，也就是所谓的类的静态属性来配置的。

③ 组件初始状态state的配置不同

- React.createClass创建的组件，其状态state是通过getInitialState方法来配置组件相关的状态；
- React.Component创建的组件，其状态state是在constructor中像初始化组件属性一样声明的。

14. 对有状态组件和无状态组件的理解及使用场景

（1）有状态组件

特点：

- 是类组件
- 有继承
- 可以使用this
- 可以使用react的生命周期
- 使用较多，容易频繁触发生命周期钩子函数，影响性能
- 内部使用 state，维护自身状态的变化，有状态组件根据外部组件传入的 props 和自身的 state进行渲染。

使用场景：

- 需要使用到状态的。
- 需要使用状态操作组件的（无状态组件的也可以实现新版本react hooks也可实现）

总结：

类组件可以维护自身的状态变量，即组件的 state ，类组件还有不同的生命周期方法，可以让开发者能够在组件的不同阶段（挂载、更新、卸载），对组件做更多的控制。类组件则既可以充当无状态组件，也可以充当有状态组件。当一个类组件不需要管理自身状态时，也可称为无状态组件。

（2）无状态组件

特点：

- 不依赖自身的状态state
- 可以是类组件或者函数组件。
- 可以完全避免使用 this 关键字。（由于使用的是箭头函数事件无需绑定）
- 有更高的性能。当不需要使用生命周期钩子时，应该首先使用无状态函数组件
- 组件内部不维护 state ，只根据外部组件传入的 props 进行渲染的组件，当 props 改变时，组件重新渲染。

使用场景：

- 组件不需要管理 state，纯展示

优点：

- 简化代码、专注于 render
- 组件不需要被实例化，无生命周期，提升性能。 输出（渲染）只取决于输入（属性），无副作用
- 视图和数据的解耦分离

缺点：

- 无法使用 ref

- 无生命周期方法
- 无法控制组件的重渲染，因为无法使用shouldComponentUpdate 方法，当组件接受到新的属性时则会重渲染

总结：

组件内部状态且与外部无关的组件，可以考虑用状态组件，这样状态树就不会过于复杂，易于理解和管理。当一个组件不需要管理自身状态时，也就是无状态组件，应该优先设计为函数组件。比如自定义的 `<Button/>`、`<Input />` 等组件。

15. 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。

```
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 />
    </>
  );
}
```

16. 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:

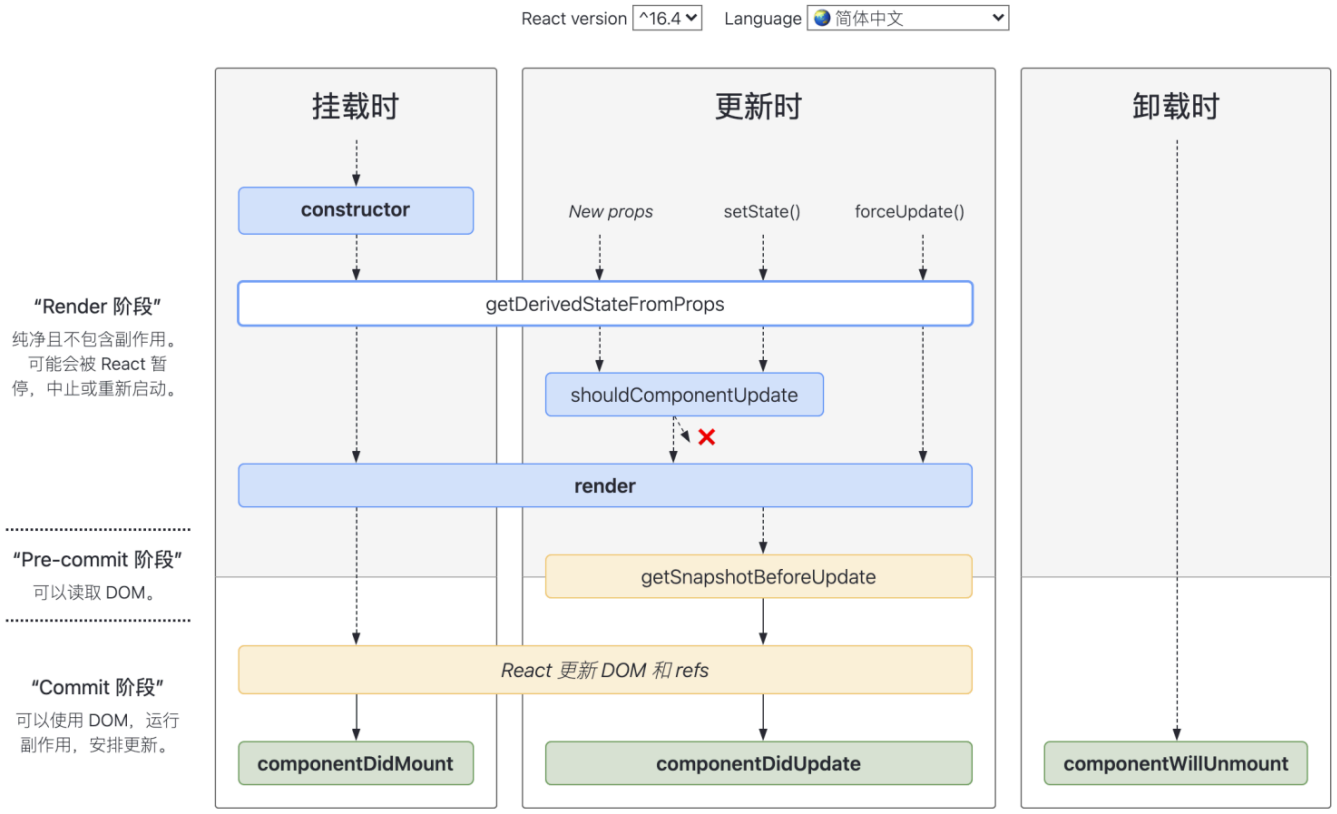
- **字符串格式**：字符串格式，这是React16版本之前用得最多的，例如：`<p ref="info">span</p>`
- **函数格式**：ref对应一个方法，该方法有一个参数，也就是对应的节点实例，例如：`<p ref={ele => this.info = ele}></p>`
- **createRef方法**：React 16提供的一个API，使用React.createRef()来实现

- **函数组件：** useRef

17. React中可以在render访问refs吗？为什么？

```
<>
  <span id="name" ref={this.spanRef}>{this.state.title}</span>
  <span>{
     this.spanRef.current ? '有值' : '无值'
  }</span>
</>
```

不可以，render 阶段 DOM 还没有生成，无法获取 DOM。DOM 的获取需要在 pre-commit 阶段和 commit 阶段：



18. 对React的插槽(Portals)的理解，如何使用，有哪些使用场景

React 官方对 Portals 的定义：

> Portal 提供了一种将子节点渲染到存在于父组件以外的 DOM 节点的优秀的方案

Portals 是React 16提供的官方解决方案，使得组件可以脱离父组件层级挂载在DOM树的任何位置。通俗来讲，就是我们 render 一个组件，但这个组件的 DOM 结构并不在本组件内。

Portals语法如下：

```
ReactDOM.createPortal(child, container);
```

- 第一个参数 child 是可渲染的 React 子项，比如元素，字符串或者片段等;
- 第二个参数 container 是一个 DOM 元素。

一般情况下，组件的render函数返回的元素会被挂载在它的父级组件上：

```
import DemoComponent from './DemoComponent';
render() {
  // DemoComponent元素会被挂载在id为parent的div的元素上
  return (
    <div id="parent">
        <DemoComponent />
    </div>
  );
}
```

然而，有些元素需要被挂载在更高层级的位置。最典型的应用场景：当父组件具有overflow: hidden或者z-index的样式设置时，组件有可能被其他元素遮挡，这时就可以考虑要不要使用Portal使组件的挂载脱离父组件。例如：对话框，模态窗。

```
import DemoComponent from './DemoComponent';
render() {
  // react会将DemoComponent组件直接挂载在真实的 dom 节点 domNode 上，生命周期还和16版本之前相同。
  return ReactDOM.createPortal(
    <DemoComponent />,
    domNode,
  );
}
```

19. 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。

20. 对 React-Intl 的理解，它的工作原理?

React-intl是雅虎的语言国际化开源项目FormatJS的一部分，通过其提供的组件和API可以与ReactJS绑定。

React-intl提供了两种使用方法，一种是引用React组件，另一种是直接调取API，官方更加推荐在React项目中使用前者，只有在无法使用React组件的地方，才应该调用框架提供的API。它提供了一系列的React组件，包括数字格式化、字符串格式化、日期格式化等。

在React-intl中，可以配置不同的语言包，他的工作原理就是根据需要，在语言包之间进行切换。

21. 对 React context 的理解

在React中，数据传递一般使用props传递数据，维持单向数据流，这样可以让组件之间的关系变得简单且可预测，但是单项数据流在某些场景中并不适用。单纯一对的父子组件传递并无问题，但要是组件之间层层依赖深入，props就需要层层传递显然，这样做太繁琐了。

Context 提供了一种在组件之间共享此类值的方式，而不必显式地通过组件树的逐层传递 props。

可以把context当做是特定一个组件树内共享的store，用来做数据传递。**简单说就是，当你不想在组件树中通过逐层传递props或者state的方式来传递数据时，可以使用Context来实现跨层级的组件数据传递。**

JS的代码块在执行期间，会创建一个相应的作用域链，这个作用域链记录着运行时JS代码块执行期间所能访问的活动对象，包括变量和函数，JS程序通过作用域链访问到代码块内部或者外部的变量和函数。

假如以JS的作用域链作为类比，React组件提供的Context对象其实就好比一个提供给子组件访问的作用域，而Context对象的属性可以看成作用域上的活动对象。由于组件 的 Context 由其父节点链上所有组件通 过 getChildContext（）返回的Context对象组合而成，所以，组件通过Context是可以访问到其父组件链上所有节点组件提供的Context的属性。

22. 为什么React并不推荐优先考虑使用Context?

- Context目前还处于实验阶段，可能会在后面的发行版本中有很大的变化，事实上这种情况已经发生了，所以为了避免给今后升级带来大的影响和麻烦，不建议在app中使用context。
- 尽管不建议在app中使用context，但是独有组件而言，由于影响范围小于app，如果可以做到高内聚，不破坏组件树之间的依赖关系，可以考虑使用context
- 对于组件之间的数据通信或者状态管理，有效使用props或者state解决，然后再考虑使用第三方的成熟库进行解决，以上的方法都不是最佳的方案的时候，在考虑context。
- context的更新需要通过setState()触发，但是这并不是很可靠的，Context支持跨组件的访问，但是如果中间的子组件通过一些方法不影响更新，比如 shouldComponentUpdate() 返回false 那么不能保证Context的更新一定可以使用Context的子组件，因此，Context的可靠性需要关注

23. React中什么是受控组件和非控组件?

（1）受控组件

在使用表单来收集用户输入时，例如 `<input><select><textearea>` 等元素都要绑定一个change事件，当表单的状态发生变化，就会触发onChange事件，更新组件的state。这种组件在React中被称为**受控组件**，在受控

组件中，组件渲染出的状态与它的value或checked属性相对应，react通过这种方式消除了组件的局部状态，使整个状态可控。react官方推荐使用受控表单组件。

受控组件更新state的流程：

- 可以通过初始state中设置表单的默认值
- 每当表单的值发生变化时，调用onChange事件处理器
- 事件处理器通过事件对象e拿到改变后的状态，并更新组件的state
- 一旦通过setState方法更新state，就会触发视图的重新渲染，完成表单组件的更新

受控组件缺陷：

表单元素的值都是由React组件进行管理，当有多个输入框，或者多个这种组件时，如果想同时获取到全部的值就必须每个都要编写事件处理函数，这会让代码看着很臃肿，所以为了解决这种情况，出现了非受控组件。

（2）非受控组件

如果一个表单组件没有value props（单选和复选按钮对应的是checked props）时，就可以称为非受控组件。在非受控组件中，可以使用一个ref来从DOM获得表单值。而不是为每个状态更新编写一个事件处理程序。

React官方的解释：

> 要编写一个非受控组件，而不是为每个状态更新都编写数据处理函数，你可以使用 ref来从 DOM 节点中获取表单数据。
>
> 因为非受控组件将真实数据储存在 DOM 节点中，所以在使用非受控组件时，有时候反而更容易同时集成 React 和非 React 代码。如果你不介意代码美观性，并且希望快速编写代码，使用非受控组件往往可以减少你的代码量。否则，你应该使用受控组件。

例如，下面的代码在非受控组件中接收单个属性：

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.value);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={(input) => this.input = input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
```

```
    }
  }
```

总结：页面中所有输入类的DOM如果是现用现取的称为非受控组件，而通过setState将输入的值维护到了state中，需要时再从state中取出，这里的数据就受到了state的控制，称为受控组件。

24. React中refs的作用是什么？有哪些应用场景？

Refs 提供了一种方式，用于访问在 render 方法中创建的 React 元素或 DOM 节点。Refs 应该谨慎使用，如下场景使用 Refs 比较适合：

- 处理焦点、文本选择或者媒体的控制
- 触发必要的动画
- 集成第三方 DOM 库

Refs 是使用 `React.createRef()` 方法创建的，他通过 `ref` 属性附加到 React 元素上。要在整个组件中使用 Refs，需要将 `ref` 在构造函数中分配给其实例属性：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.myRef = React.createRef()
  }
  render() {
    return <div ref={this.myRef} />
  }
}
```

由于函数组件没有实例，因此不能在函数组件上直接使用 `ref`：

```
function MyFunctionalComponent() {
  return <input />;
}
class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  render() {
    // 这将不会工作!
    return (
      <MyFunctionalComponent ref={this.textInput} />
    );
  }
}
```

但可以通过闭合的帮助在函数组件内部进行使用 Refs：

```
function CustomTextInput(props) {
  // 这里必须声明 textInput，这样 ref 回调才可以引用它
  let textInput = null;
  function handleClick() {
    textInput.focus();
  }
  return (
    <div>
      <input
        type="text"
        ref={(input) => { textInput = input; }} />
      <input
        type="button"
        value="Focus the text input"
        onClick={handleClick}
      />
    </div>
  );
}
```

注意：

- 不应该过度的使用 Refs
- `ref` 的返回值取决于节点的类型：
- - 当 `ref` 属性被用于一个普通的 HTML 元素时，`React.createRef()` 将接收底层 DOM 元素作为他的 `current` 属性以创建 `ref`。
  - 当 `ref` 属性被用于一个自定义的类组件时，`ref` 对象将接收该组件已挂载的实例作为他的 `current`。
- 当在父组件中需要访问子组件中的 `ref` 时可使用传递 Refs 或回调 Refs。

25. React中除了在构造函数中绑定this，还有别的方式吗？

- 在构造函数中绑定this

```
constructor(props){
      super(props);
       this.state={
           msg:'hello world',
       }
       this.getMsg = this.getMsg.bind(this)
   }
```

- 函数定义的时候使用箭头函数

```
constructor(props){
    super(props);
    this.state={
           msg:'hello world',
    }
    render(){
      <button onClcik={()=>{alert(this.state.msg)}}>点我</button>
    }
}
```

- 函数调用是使用bind绑定this

```
 <button onClick={this.getMsg.bind(this)}>点我</button>
```

26. 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：

```
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

27. React.forwardRef是什么？它有什么作用？

React.forwardRef 会创建一个React组件，这个组件能够将其接受的 ref 属性转发到其组件树下的另一个组件中。这种技术并不常见，但在以下两种场景中特别有用：

- 转发 refs 到 DOM 组件
- 在高阶组件中转发 refs

28. 类组件与函数组件有什么异同？

相同点：

组件是 React 可复用的最小代码片段，它们会返回要在页面中渲染的 React 元素。也正因为组件是 React 的最小编码单位，所以无论是函数组件还是类组件，在使用方式和最终呈现效果上都是完全一致的。

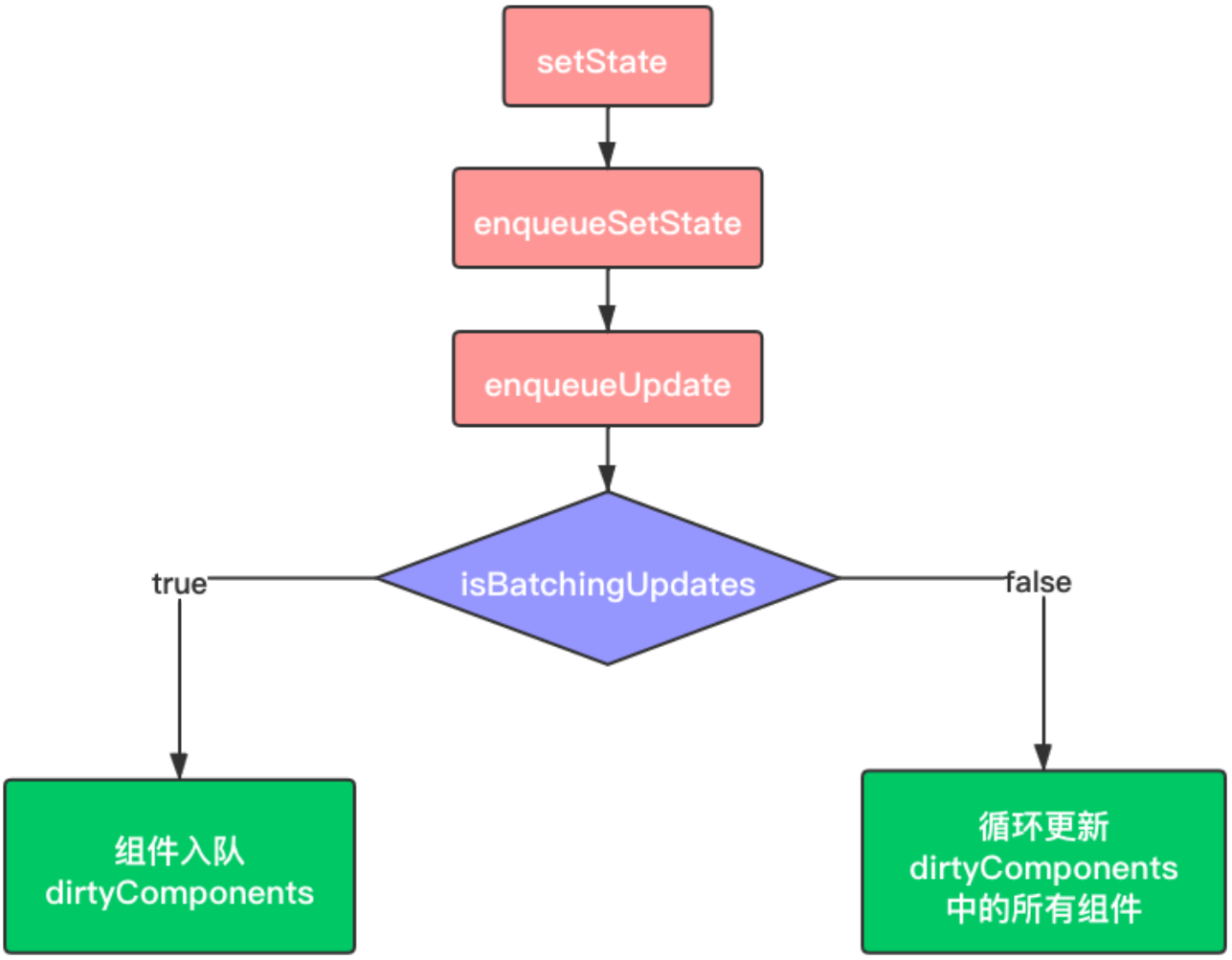
我们甚至可以将一个类组件改写成函数组件，或者把函数组件改写成一个类组件（虽然并不推荐这种重构行为）。从使用者的角度而言，很难从使用体验上区分两者，而且在现代浏览器中，闭包和类的性能只在极端场景下才会有明显的差别。所以，基本可认为两者作为组件是完全一致的。

不同点：

- 它们在开发时的心智模型上却存在巨大的差异。类组件是基于面向对象编程的，它主打的是继承、生命周期等核心概念；而函数组件内核是函数式编程，主打的是 immutable、没有副作用、引用透明等特点。
- 之前，在使用场景上，如果存在需要使用生命周期的组件，那么主推类组件；设计模式上，如果需要使用继承，那么主推类组件。但现在由于 React Hooks 的推出，生命周期概念的淡出，函数组件可以完全取代类组件。其次继承并不是组件最佳的设计模式，官方更推崇"组合优于继承"的设计概念，所以类组件在这方面的优势也在淡出。
- 性能优化上，类组件主要依靠 shouldComponentUpdate 阻断渲染来提升性能，而函数组件依靠 React.memo 缓存渲染结果来提升性能。
- 从上手程度而言，类组件更容易上手，从未来趋势上看，由于React Hooks 的推出，函数组件成了社区未来主推的方案。
- 类组件在未来时间切片与并发模式中，由于生命周期带来的复杂度，并不易于优化。而函数组件本身轻量简单，且在 Hooks 的基础上提供了比原先更细粒度的逻辑组织与复用，更能适应 React 的未来发展。

二、数据管理

1. React setState 调用的原理



具体的执行过程如下（源码级解析）：

- 首先调用了`setState`入口函数，入口函数在这里就是充当一个分发器的角色，根据入参的不同，将其分发到不同的功能函数中去；

```
ReactComponent.prototype.setState = function (partialState, callback) {
  this.updater.enqueueSetState(this, partialState);
  if (callback) {
    this.updater.enqueueCallback(this, callback, 'setState');
  }
};
```

- `enqueueSetState`方法将新的`state`放进组件的状态队列里，并调用`enqueueUpdate`来处理将要更新的实例对象；

```
enqueueSetState: function (publicInstance, partialState) {
  // 根据 this 拿到对应的组件实例
  var internalInstance = getInternalInstanceReadyForUpdate(publicInstance,
'setState');
  // 这个 queue 对应的就是一个组件实例的 state 数组
  var queue = internalInstance._pendingStateQueue ||
```

```
  (internalInstance._pendingStateQueue = []);
    queue.push(partialState);
    //  enqueueUpdate 用来处理当前的组件实例
    enqueueUpdate(internalInstance);
  }
```

- 在 enqueueUpdate 方法中引出了一个关键的对象——batchingStrategy，该对象所具备的 isBatchingUpdates 属性直接决定了当下是要走更新流程，还是应该排队等待；如果轮到执行，就调用 batchedUpdates 方法来直接发起更新流程。由此可以推测，batchingStrategy 或许正是 React 内部专门用于管控批量更新的对象。

```
function enqueueUpdate(component) {
  ensureInjected();
  // 注意这一句是问题的关键，isBatchingUpdates标识着当前是否处于批量创建/更新组件的阶段
  if (!batchingStrategy.isBatchingUpdates) {
    // 若当前没有处于批量创建/更新组件的阶段，则立即更新组件
    batchingStrategy.batchedUpdates(enqueueUpdate, component);
    return;
  }
  // 否则，先把组件塞入 dirtyComponents 队列里，让它“再等等”
  dirtyComponents.push(component);
  if (component._updateBatchNumber == null) {
    component._updateBatchNumber = updateBatchNumber + 1;
  }
}
```

注意：batchingStrategy 对象可以理解为“锁管理器”。这里的“锁”，是指 React 全局唯一的 isBatchingUpdates 变量，isBatchingUpdates 的初始值是 false，意味着“当前并未进行任何批量更新操作”。每当 React 调用 batchedUpdate 去执行更新动作时，会先把这个锁给“锁上”（置为 true），表明“现在正处于批量更新过程中”。当锁被“锁上”的时候，任何需要更新的组件都只能暂时进入 dirtyComponents 里排队等候下一次的批量更新，而不能随意“插队”。此处体现的“任务锁”的思想，是 React 面对大量状态仍然能够实现有序分批处理的基石。

2. React setState 调用之后发生了什么？是同步还是异步？

（1）React中setState后发生了什么

在代码中调用setState函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发调和过程(Reconciliation)。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个UI界面。

在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

如果在短时间内频繁setState。React会将state的改变压入栈中，在合适的时机，批量更新state和视图，达到提高性能的效果。

（2）setState 是同步还是异步的

假如所有setState是同步的，意味着每执行一次setState时（有可能一个同步代码中，多次setState），都重新vnode diff + dom修改，这对性能来说是极为不好的。如果是异步，则可以把一个同步代码中的多个setState合并成一次组件更新。所以默认是异步的，但是在一些情况下是同步的。

setState 并不是单纯同步/异步的，它的表现会因调用场景的不同而不同。在源码中，通过 isBatchingUpdates 来判断setState 是先存进 state 队列还是直接更新，如果值为 true 则执行异步操作，为 false 则直接更新。

- **异步：**在 React 可以控制的地方，就为 true，比如在 React 生命周期事件和合成事件中，都会走合并操作，延迟更新的策略。
- **同步：**在 React 无法控制的地方，比如原生事件，具体就是在 addEventListener 、setTimeout、setInterval 等事件中，就只能同步更新。

一般认为，做异步设计是为了性能优化、减少渲染次数：

- `setState`设计为异步，可以显著的提升性能。如果每次调用 `setState`都进行一次更新，那么意味着`render`函数会被频繁调用，界面重新渲染，这样效率是很低的；最好的办法应该是获取到多个更新，之后进行批量更新；
- 如果同步更新了`state`，但是还没有执行`render`函数，那么`state`和`props`不能保持同步。`state`和`props`不能保持一致性，会在开发中产生很多的问题；

3. 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。

```
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）。

4. React中有使用过getDefaultProps吗？它有什么作用?

通过实现组件的getDefaultProps，对属性设置默认值（ES5的写法）：

```
var ShowTitle = React.createClass({
  getDefaultProps:function(){
```

```
    return{
      title : "React"
    }
  },
  render : function(){
    return <h1>{this.props.title}</h1>
  }
});
```

5. React中setState的第二个参数作用是什么?

`setState` 的第二个参数是一个可选的回调函数。这个回调函数将在组件重新渲染后执行。等价于在 `componentDidUpdate` 生命周期内执行。通常建议使用 `componentDidUpdate` 来代替此方式。在这个回调函数中你可以拿到更新后 `state` 的值:

```
this.setState({
    key1: newState1,
    key2: newState2,
    ...
}, callback) // 第二个参数是 state 更新完成后的回调函数
```

6. 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:

```
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:

```
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。

7. 在React中组件的this.state和setState有什么区别?

this.state通常是用来初始化state的，this.setState是用来修改state值的。如果初始化了state之后再使用this.state，之前的state会被覆盖掉，如果使用this.setState，只会替换掉相应的state值。所以，如果想要修改state的值，就需要使用setState，而不能直接修改state，直接修改state之后页面是不会更新的。

8. state 是怎么注入到组件的，从 reducer 到组件经历了什么样的过程

通过connect和mapStateToProps将state注入到组件中：

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '@/reducers/Todo/actions'
import Link from '@/containers/Todo/components/Link'

const mapStateToProps = (state, ownProps) => ({
    active: ownProps.filter === state.visibilityFilter
})

const mapDispatchToProps = (dispatch, ownProps) => ({
    setFilter: () => {
        dispatch(setVisibilityFilter(ownProps.filter))
    }
})

export default connect(
    mapStateToProps,
    mapDispatchToProps
)(Link)
```

上面代码中，active就是注入到Link组件中的状态。 mapStateToProps（state，ownProps）中带有两个参数，含义是:

- state-store管理的全局状态对象，所有都组件状态数据都存储在该对象中。
- ownProps 组件通过props传入的参数。

reducer 到组件经历的过程：

- reducer对action对象处理，更新组件状态，并将新的状态值返回store。
- 通过connect（mapStateToProps，mapDispatchToProps）（Component）对组件 Component进行升级，此时将状态值从store取出并作为props参数传递到组件。

高阶组件实现源码:

```
import React from 'react'
import PropTypes from 'prop-types'
```

```
// 高阶组件 contect
export const connect = (mapStateToProps, mapDispatchToProps) => (WrappedComponent)
=> {
    class Connect extends React.Component {
        // 通过对context调用获取store
        static contextTypes = {
            store: PropTypes.object
        }

        constructor() {
            super()
            this.state = {
                allProps: {}
            }
        }

        // 第一遍需初始化所有组件初始状态
        componentWillMount() {
            const store = this.context.store
            this._updateProps()
            store.subscribe(() => this._updateProps()); // 加入_updateProps()至
store里的监听事件列表
        }

        // 执行action后更新props，使组件可以更新至最新状态（类似于setState）
        _updateProps() {
            const store = this.context.store;
            let stateProps = mapStateToProps ?
                mapStateToProps(store.getState(), this.props) : {} // 防止
mapStateToProps 没有传入
            let dispatchProps = mapDispatchToProps ?
                mapDispatchToProps(store.dispatch, this.props) : {
                                    dispatch: store.dispatch
                                } // 防止 mapDispatchToProps 没有传入
            this.setState({
                allProps: {
                    ...stateProps,
                    ...dispatchProps,
                    ...this.props
                }
            })
        }

        render() {
            return <WrappedComponent {...this.state.allProps} />
        }
    }
    return Connect
}
```

9. React组件的state和props有什么区别?

（1）props

props是一个从外部传进组件的参数，主要作为就是从父组件向子组件传递数据，它具有可读性和不变性，只能通过外部组件主动传入新的props来重新渲染子组件，否则子组件的props以及展现形式不会改变。

（2）state

state的主要作用是用于组件保存、控制以及修改自己的状态，它只能在constructor中初始化，它算是组件的私有属性，不可通过外部访问和修改，只能通过组件内部的this.setState来修改，修改state属性会导致组件的重新渲染。

（3）区别

- props 是传递给组件的（类似于函数的形参），而state 是在组件内被组件自己管理的（类似于在一个函数内声明的变量）。
- props 是不可修改的，所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。
- state 是在组件中创建的，一般在 constructor中初始化 state。state 是多变的、可以修改，每次setState都异步更新的。

10. React中的props为什么是只读的?

`this.props`是组件之间沟通的一个接口，原则上来讲，它只能从父组件流向子组件。React具有浓重的函数式编程的思想。

提到函数式编程就要提一个概念：纯函数。它有几个特点：

- 给定相同的输入，总是返回相同的输出。
- 过程没有副作用。
- 不依赖外部状态。

`this.props`就是汲取了纯函数的思想。props的不可以变性就保证的相同的输入，页面显示的内容是一样的，并且不会产生副作用

11. 在React中组件的props改变时更新组件的有哪些方法?

在一个组件传入的props更新时重新渲染该组件常用的方法是在`componentWillReceiveProps`中将新的props更新到组件的state中（这种state被成为派生状态（Derived State）），从而实现重新渲染。React 16.3中还引入了一个新的钩子函数`getDerivedStateFromProps`来专门实现这一需求。

** （1）**componentWillReceiveProps（已废弃）**

在react的componentWillReceiveProps(nextProps)生命周期中，可以在子组件的render函数执行前，通过this.props获取旧的属性，通过nextProps获取新的props，对比两次props是否相同，从而更新子组件自己的state。

这样的好处是，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。

** （2）**getDerivedStateFromProps（16.3引入）**

这个生命周期函数是为了替代`componentWillReceiveProps`存在的，所以在需要使用`componentWillReceiveProps`时，就可以考虑使用`getDerivedStateFromProps`来进行替代。

两者的参数是不相同的，而`getDerivedStateFromProps`是一个静态函数，也就是这个函数不能通过this访问到class的属性，也并不推荐直接访问属性。而是应该通过参数提供的nextProps以及prevState来进行判断，根据新传入的props来映射到state。

需要注意的是，**如果props传入的内容不需要影响到你的state，那么就需要返回一个null**，这个返回值是必须的，所以尽量将其写到函数的末尾：

```
static getDerivedStateFromProps(nextProps, prevState) {
    const {type} = nextProps;
    // 当传入的type发生变化的时候，更新state
    if (type !== prevState.type) {
        return {
            type,
        };
    }
    // 否则，对于state不进行任何操作
    return null;
}
```

12. React中怎么检验props？验证props的目的是什么？

React为我们提供了**PropTypes**以供验证使用。当我们向**Props**传入的数据无效（向Props传入的数据类型和验证的数据类型不符）就会在控制台发出警告信息。它可以避免随着应用越来越复杂从而出现的问题。并且，它还可以让程序变得更易读。

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

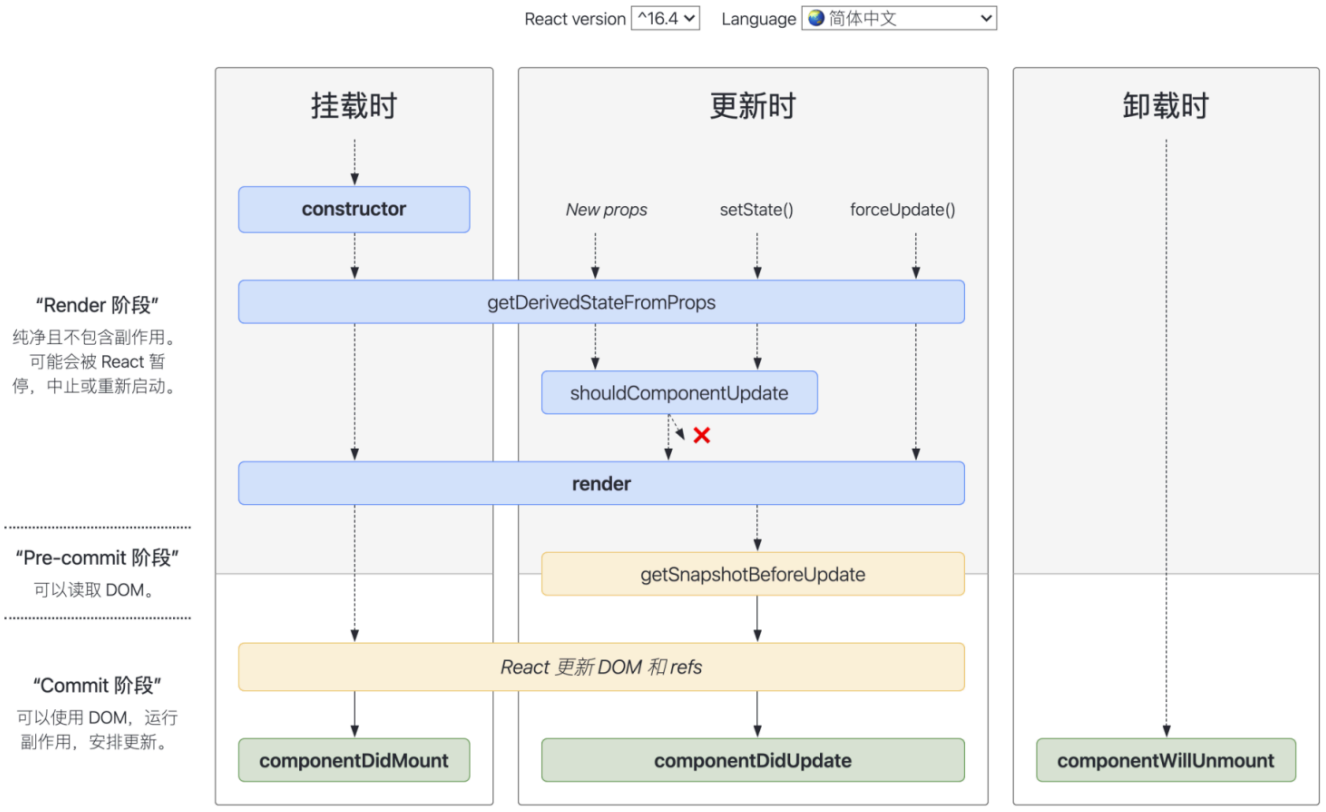
当然，如果项目汇中使用了TypeScript，那么就可以不用PropTypes来校验，而使用TypeScript定义接口来校验props。

三、生命周期

1. React的生命周期有哪些？

React 通常将组件生命周期分为三个阶段：

- 装载阶段（Mount），组件第一次在DOM树中被渲染的过程；
- 更新过程（Update），组件状态发生变化，重新更新渲染的过程；
- 卸载过程（Unmount），组件从DOM树中被移除的过程；



1）组件挂载阶段

挂载阶段组件被创建，然后组件实例插入到 DOM 中，完成组件的第一次渲染，该过程只会发生一次，在此阶段会依次调用以下这些方法：

- constructor
- getDerivedStateFromProps
- render
- componentDidMount

（1）constructor

组件的构造函数，第一个被执行，若没有显式定义它，会有一个默认的构造函数，但是若显式定义了构造函数，我们必须在构造函数中执行 `super(props)`，否则无法在构造函数中拿到this。

如果不初始化 state 或不进行方法绑定，则不需要为 React 组件实现构造函数**Constructor**。

constructor中通常只做两件事：

- 初始化组件的 state
- 给事件处理方法绑定 this

```
constructor(props) {
  super(props);
  // 不要在构造函数中调用 setState，可以直接给 state 设置初始值
  this.state = { counter: 0 }
  this.handleClick = this.handleClick.bind(this)
}
```

(2) getDerivedStateFromProps

```
static getDerivedStateFromProps(props, state)
```

这是个静态方法，所以不能在这个函数里使用 `this`，有两个参数 `props` 和 `state`，分别指接收到的新参数和当前组件的 `state` 对象，这个函数会返回一个对象用来更新当前的 `state` 对象，如果不需要更新可以返回 `null`。

该函数会在装载时，接收到新的 `props` 或者调用了 `setState` 和 `forceUpdate` 时被调用。如当接收到新的属性想修改 `state` ，就可以使用。

```
// 当 props.counter 变化时，赋值给 state
class App extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      counter: 0
    }
  }
  static getDerivedStateFromProps(props, state) {
    if (props.counter !== state.counter) {
      return {
        counter: props.counter
      }
    }
    return null
  }

  handleClick = () => {
    this.setState({
      counter: this.state.counter + 1
    })
  }
  render() {
    return (
      <div>
        <h1 onClick={this.handleClick}>Hello, world!{this.state.counter}</h1>
      </div>
    )
```

```
    }
  }
```

现在可以显式传入 `counter` ，但是这里有个问题，如果想要通过点击实现 `state.counter` 的增加，但这时会发现值不会发生任何变化，一直保持 `props` 传进来的值。这是由于在 React 16.4^ 的版本中 `setState` 和 `forceUpdate` 也会触发这个生命周期，所以当组件内部 `state` 变化后，就会重新走这个方法，同时会把 `state` 值赋值为 `props` 的值。因此需要多加一个字段来记录之前的 `props` 值，这样就会解决上述问题。具体如下：

```
// 这里只列出需要变化的地方
class App extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      // 增加一个 preCounter 来记录之前的 props 传来的值
      preCounter: 0,
      counter: 0
    }
  }
  static getDerivedStateFromProps(props, state) {
    // 跟 state.preCounter 进行比较
    if (props.counter !== state.preCounter) {
      return {
        counter: props.counter,
        preCounter: props.counter
      }
    }
    return null
  }
  handleClick = () => {
    this.setState({
      counter: this.state.counter + 1
    })
  }
  render() {
    return (
      <div>
        <h1 onClick={this.handleClick}>Hello, world!{this.state.counter}</h1>
      </div>
    )
  }
}
```

（3）render

render是React 中最核心的方法，一个组件中必须要有这个方法，它会根据状态 `state` 和属性 `props` 渲染组件。这个函数只做一件事，就是返回需要渲染的内容，所以不要在这个函数内做其他业务逻辑，通常调用该方法会返回以下类型中一个：

- **React 元素**：这里包括原生的 DOM 以及 React 组件；
- **数组和 Fragment（片段）**：可以返回多个元素；
- **Portals（插槽）**：可以将子元素渲染到不同的 DOM 子树种；
- **字符串和数字**：被渲染成 DOM 中的 text 节点；
- **布尔值或 null**：不渲染任何内容。

（4）componentDidMount()

componentDidMount()会在组件挂载后（插入 DOM 树中）立即调。该阶段通常进行以下操作：

- 执行依赖于DOM的操作；
- 发送网络请求；（官方建议）
- 添加订阅消息（会在componentWillUnmount取消订阅）；

如果在 `componentDidMount` 中调用 `setState`，就会触发一次额外的渲染，多调用了一次 `render` 函数，由于它是在浏览器刷新屏幕前执行的，所以用户对此是没有感知的，但是我应当避免这样使用，这样会带来一定的性能问题，尽量是在 `constructor` 中初始化 `state` 对象。

在组件装载之后，将计数数字变为1：

```
class App extends React.Component  {
  constructor(props) {
    super(props)
    this.state = {
      counter: 0
    }
  }
  componentDidMount () {
    this.setState({
      counter: 1
    })
  }
  render ()  {
    return (
      <div className="counter">
        counter值: { this.state.counter }
      </div>
    )
  }
}
```

2）组件更新阶段

当组件的 `props` 改变了，或组件内部调用了 `setState/forceUpdate`，会触发更新重新渲染，这个过程可能会发生多次。这个阶段会依次调用下面这些方法：

- getDerivedStateFromProps
- shouldComponentUpdate
- render

- getSnapshotBeforeUpdate
- componentDidUpdate

（1）shouldComponentUpdate

```
shouldComponentUpdate(nextProps, nextState)
```

在说这个生命周期函数之前，来看两个问题：

- **setState 函数在任何情况下都会导致组件重新渲染吗？例如下面这种情况：**

```
this.setState({number: this.state.number})
```

- **如果没有调用 setState，props 值也没有变化，是不是组件就不会重新渲染？**

第一个问题答案是 **会** ，第二个问题如果是父组件重新渲染时，不管传入的 props 有没有变化，都会引起子组件的重新渲染。

那么有没有什么方法解决在这两个场景下不让组件重新渲染进而提升性能呢？这个时候 `shouldComponentUpdate` 登场了，这个生命周期函数是用来提升速度的，它是在重新渲染组件开始前触发的，默认返回 `true`，可以比较 `this.props` 和 `nextProps` ， `this.state` 和 `nextState` 值是否变化，来确认返回 true 或者 `false`。当返回 `false` 时，组件的更新过程停止，后续的 `render`、`componentDidUpdate` 也不会被调用。

注意：添加 `shouldComponentUpdate` 方法时，不建议使用深度相等检查（如使用 `JSON.stringify()`），因为深比较效率很低，可能会比重新渲染组件效率还低。而且该方法维护比较困难，建议使用该方法会产生明显的性能提升时使用。

（2）getSnapshotBeforeUpdate

```
getSnapshotBeforeUpdate(prevProps, prevState)
```

这个方法在 `render` 之后，`componentDidUpdate` 之前调用，有两个参数 `prevProps` 和 `prevState`，表示更新之前的 `props` 和 `state`，这个函数必须要和 `componentDidUpdate` 一起使用，并且要有一个返回值，默认是 `null`，这个返回值作为第三个参数传给 `componentDidUpdate`。

（3）componentDidUpdate

componentDidUpdate() 会在更新后会被立即调用，首次渲染不会执行此方法。 该阶段通常进行以下操作：

- 当组件更新后，对 DOM 进行操作；
- 如果你对更新前后的 props 进行了比较，也可以选择在此处进行网络请求；（例如，当 props 未发生变化时，则不会执行网络请求）。

```
componentDidUpdate(prevProps, prevState, snapshot){}
```

该方法有三个参数：

- prevProps: 更新前的props
- prevState: 更新前的state
- snapshot: getSnapshotBeforeUpdate()生命周期的返回值

3）组件卸载阶段

卸载阶段只有一个生命周期函数，componentWillUnmount() 会在组件卸载及销毁之前直接调用。在此方法中执行必要的清理操作：

- 清除 timer，取消网络请求或清除
- 取消在 componentDidMount() 中创建的订阅等；

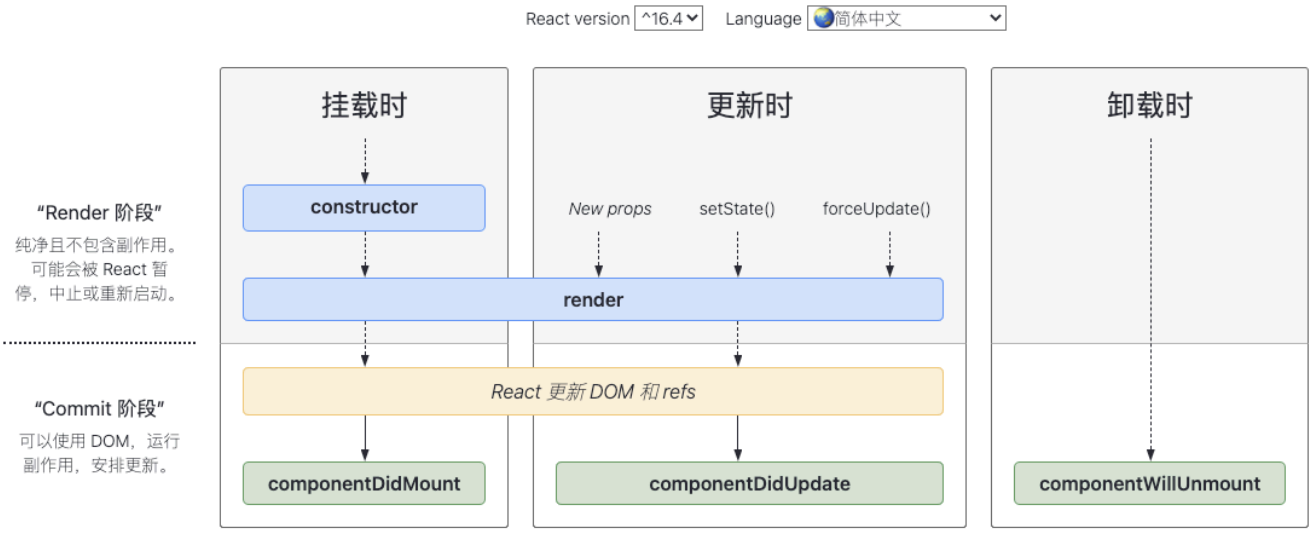
这个生命周期在一个组件被卸载和销毁之前被调用，因此你不应该再这个方法中使用 `setState`，因为组件一旦被卸载，就不会再装载，也就不会重新渲染。

4）错误处理阶段

componentDidCatch(error, info)，此生命周期在后代组件抛出错误后被调用。 它接收两个参数:

- error：抛出的错误。
- info：带有 componentStack key 的对象，其中包含有关组件引发错误的栈信息

React常见的生命周期如下：



React常见生命周期的过程大致如下：

- 挂载阶段，首先执行constructor构造方法，来创建组件
- 创建完成之后，就会执行render方法，该方法会返回需要渲染的内容
- 随后，React会将需要渲染的内容挂载到DOM树上
- **挂载完成之后就会执行****componentDidMount生命周期函数**

- 如果我们给组件创建一个props（用于组件通信）、调用setState（更改state中的数据）、调用forceUpdate（强制更新组件）时，都会重新调用render函数
- render函数重新执行之后，就会重新进行DOM树的挂载
- **挂载完成之后就会执行****componentDidUpdate生命周期函数**
- **当移除组件时，就会执行****componentWillUnmount生命周期函数**

React主要生命周期总结：

1. **getDefaultProps**：这个函数会在组件创建之前被调用一次（有且仅有一次），它被用来初始化组件的 Props；
2. **getInitialState**：用于初始化组件的 state 值；
3. **componentWillMount**：在组件创建后、render 之前，会走到 componentWillMount 阶段。这个阶段我个人一直没用过、非常鸡肋。后来React 官方已经不推荐大家在 componentWillMount 里做任何事情、到现在 **React16 直接废弃了这个生命周期**，足见其鸡肋程度了；
4. **render**：这是所有生命周期中唯一一个你必须要实现的方法。一般来说需要返回一个 jsx 元素，这时 React 会根据 props 和 state 来把组件渲染到界面上；不过有时，你可能不想渲染任何东西，这种情况下让它返回 null 或者 false 即可；
5. **componentDidMount**：会在组件挂载后（插入 DOM 树中后）立即调用，标志着组件挂载完成。一些操作如果依赖获取到 DOM 节点信息，我们就会放在这个阶段来做。此外，这还是 React 官方推荐的发起 ajax 请求的时机。该方法和 componentWillMount 一样，有且仅有一次调用。

2. React 废弃了哪些生命周期？为什么？

被废弃的三个函数都是在render之前，因为fber的出现，很可能因为高优先级任务的出现而打断现有任务导致它们会被执行多次。另外的一个原因则是，React想约束使用者，好的框架能够让人不得已写出容易维护和扩展的代码，这一点又是从何谈起，可以从新增加以及即将废弃的生命周期分析入手

1) componentWillMount

首先这个函数的功能完全可以使用componentDidMount和 constructor来代替，异步获取的数据的情况上面已经说明了，而如果抛去异步获取数据，其余的即是初始化而已，这些功能都可以在constructor中执行，除此之外，如果在 willMount 中订阅事件，但在服务端这并不会执行 willUnMount事件，也就是说服务端会导致内存泄漏所以componentWillMount完全可以不使用，但使用者有时候难免因为各 种各样的情况在componentWilMount中做一些操作，那么React为了约束开发者，干脆就抛掉了这个API

2) componentWillReceiveProps

在老版本的 React 中，如果组件自身的某个 state 跟其 props 密切相关的话，一直都没有一种很优雅的处理方式去更新 state，而是需要在 componentWilReceiveProps 中判断前后两个 props 是否相同，如果不同再将新的 props更新到相应的 state 上去。这样做一来会破坏 state 数据的单一数据源，导致组件状态变得不可预测，另一方面也会增加组件的重绘次数。类似的业务需求也有很多，如一个可以横向滑动的列表，当前高亮的 Tab 显然隶属于列表自身的时，根据传入的某个值，直接定位到某个 Tab。为了解决这些问题，React引入了第一个新的生命周期：getDerivedStateFromProps。它有以下的优点:

- getDSFP是静态方法，在这里不能使用this，也就是一个纯函数，开发者不能写出副作用的代码
- 开发者只能通过prevState而不是prevProps来做对比，保证了state和props之间的简单关系以及不需要处理第一次渲染时prevProps为空的情况
- 基于第一点，将状态变化（setState）和昂贵操作（tabChange）区分开，更加便于 render 和 commit 阶段操作或者说优化。

3) componentWillUpdate

与 componentWillReceiveProps 类似，许多开发者也会在 componentWillUpdate 中根据 props 的变化去触发一些回调 。 但不论是 componentWilReceiveProps 还 是 componentWilUpdate，都有可能在一次更新中被调用多次，也就是说写在这里的回调函数也有可能会被调用多次，这显然是不可取的。与 componentDidMount 类 似， componentDidUpdate 也不存在这样的问题，一次更新中 componentDidUpdate 只会被调用一次，所以将原先写在 componentWillUpdate 中 的 回 调 迁 移 至 componentDidUpdate 就可以解决这个问题。

另外一种情况则是需要获取DOM元素状态，但是由于在fber中，render可打断，可能在wilMount中获取到的元素状态很可能与实际需要的不同，这个通常可以使用第二个新增的生命函数的解决 getSnapshotBeforeUpdate(prevProps, prevState)

4) getSnapshotBeforeUpdate(prevProps, prevState)

返回的值作为componentDidUpdate的第三个参数。与willMount不同的是，getSnapshotBeforeUpdate会在最终确定的render执行之前执行，也就是能保证其获取到的元素状态与didUpdate中获取到的元素状态相同。官方参考代码：

```
class ScrollingList extends React.Component {
  constructor(props) {
    super(props);
    this.listRef = React.createRef();
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
    // 我们是否在 list 中添加新的 items ？
    // 捕获滚动位置以便我们稍后调整滚动位置。
    if (prevProps.list.length < this.props.list.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    // 如果我们 snapshot 有值，说明我们刚刚添加了新的 items，
    // 调整滚动位置使得这些新 items 不会将旧的 items 推出视图。
    // (这里的 snapshot 是 getSnapshotBeforeUpdate 的返回值)
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }

  render() {
    return (
      <div ref={this.listRef}>{/* ...contents... */}</div>
    );
  }
}
```

3. React 16.X 中 props 改变后在哪个生命周期中处理

在getDerivedStateFromProps中进行处理。

这个生命周期函数是为了替代`componentWillReceiveProps`存在的，所以在需要使用`componentWillReceiveProps`时，就可以考虑使用`getDerivedStateFromProps`来进行替代。

两者的参数是不相同的，而`getDerivedStateFromProps`是一个静态函数，也就是这个函数不能通过this访问到class的属性，也并不推荐直接访问属性。而是应该通过参数提供的nextProps以及prevState来进行判断，根据新传入的props来映射到state。

需要注意的是，**如果props传入的内容不需要影响到你的state，那么就需要返回一个null**，这个返回值是必须的，所以尽量将其写到函数的末尾：

```
static getDerivedStateFromProps(nextProps, prevState) {
    const {type} = nextProps;
    // 当传入的type发生变化的时候，更新state
    if (type !== prevState.type) {
        return {
            type,
        };
    }
    // 否则，对于state不进行任何操作
    return null;
}
```

4. React 性能优化在哪个生命周期？它优化的原理是什么？

react的父级组件的render函数重新渲染会引起子组件的render方法的重新渲染。但是，有的时候子组件的接受父组件的数据没有变动。子组件render的执行会影响性能，这时就可以使用shouldComponentUpdate来解决这个问题。

使用方法如下：

```
shouldComponentUpdate(nexrProps) {
    if (this.props.num === nexrProps.num) {
        return false
    }
    return true;
}
```

shouldComponentUpdate提供了两个参数nextProps和nextState，表示下一次props和一次state的值，当函数返回false时候，render()方法不执行，组件也就不会渲染，返回true时，组件照常重渲染。此方法就是拿当前props中值和下一次props中的值进行对比，数据相等时，返回false，反之返回true。

需要注意，在进行新旧对比的时候，是**浅对比，**也就是说如果比较的数据时引用数据类型，只要数据的引用的地址没变，即使内容变了，也会被判定为true。

面对这个问题，可以使用如下方法进行解决：

（1）使用setState改变数据之前，先采用ES6中assgin进行拷贝，但是assgin只深拷贝的数据的第一层，所以说不是最完美的解决办法：

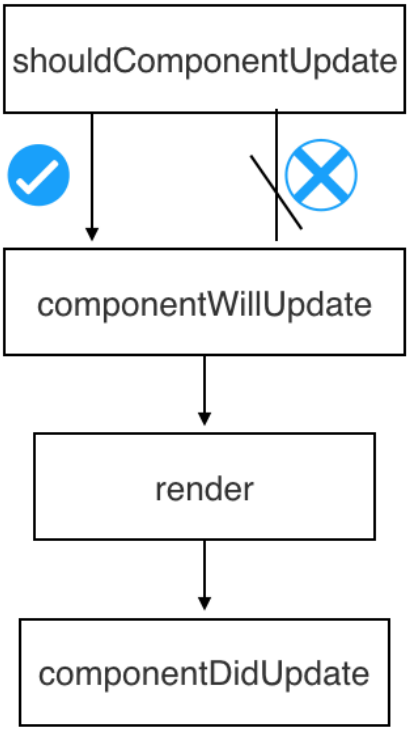
```
const o2 = Object.assign({},this.state.obj)
    o2.student.count = '00000';
    this.setState({
        obj: o2,
    })
```

（2）使用JSON.parse(JSON.stringfy())进行深拷贝，但是遇到数据为undefined和函数时就会错。

```
const o2 = JSON.parse(JSON.stringify(this.state.obj))
    o2.student.count = '00000';
    this.setState({
        obj: o2,
    })
```

5. state 和 props 触发更新的生命周期分别有什么区别?

state 更新流程：



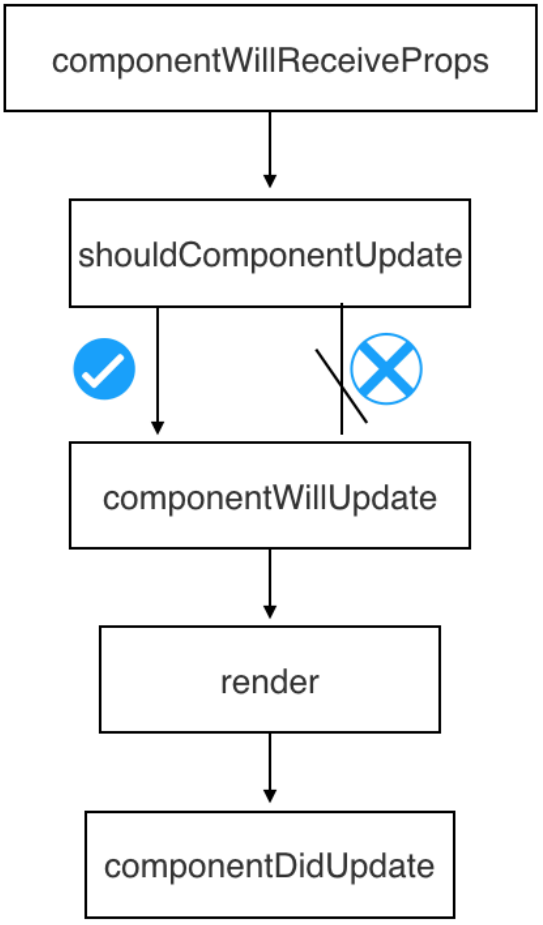
这个过程当中涉及的函数：

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()`

1. 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；
2. 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。

6. 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。

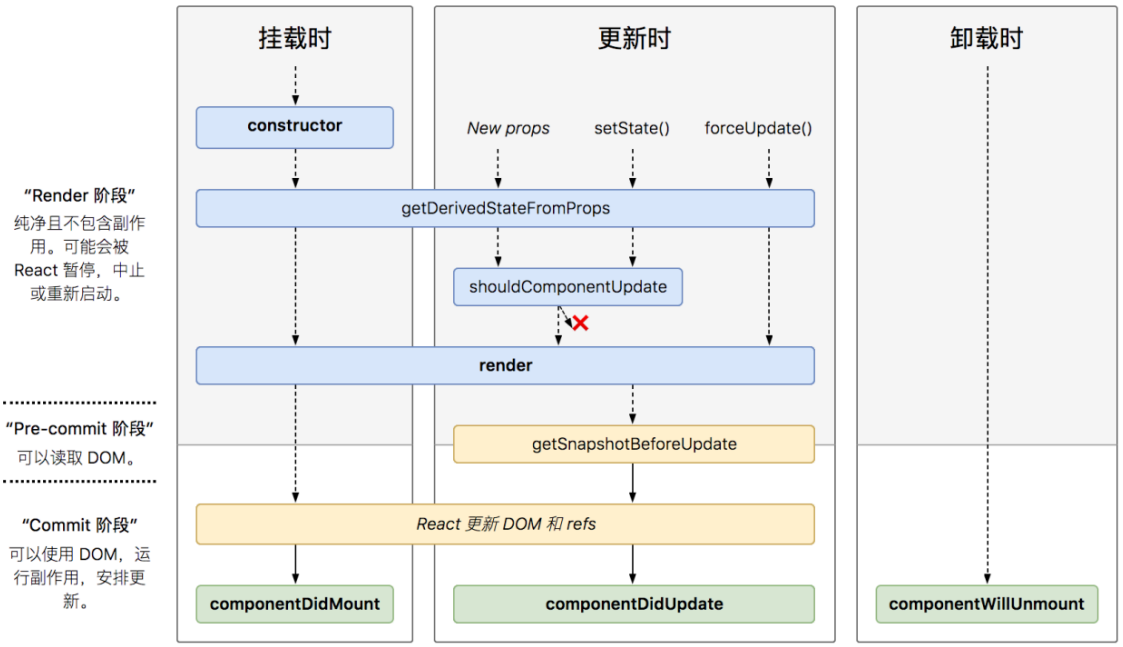
- constructor被调用是在组件准备要挂载的最开始，此时组件尚未挂载到网页上。
- componentWillMount方法的调用在constructor之后，在render之前，在这方法里的代码调用setState方法不会触发重新render，所以它一般不会用来作加载数据之用。
- componentDidMount方法中的代码，是在组件已经完全挂载到网页上才会调用被执行，所以可以保证数据的加载。此外，在这方法中调用setState方法，会触发重新渲染。所以，官方设计这个方法就是用来加载外部数据用的，或处理其他的副作用代码。与组件上的数据无关的加载，也可以在constructor里做，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。

7. 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**

四、组件通信

React组件间通信常见的几种情况:

- 父组件向子组件通信
- 子组件向父组件通信
- 跨级组件通信
- 非嵌套关系的组件通信

1. 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。

```
// 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。

```
// 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>
    }
}
```

2. 跨级组件的通信方式?

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，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>
          )
      }
}
```

3. 非嵌套关系组件的通信方式?

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。

- 可以使用自定义事件通信（发布订阅模式）
- 可以通过redux等进行全局状态管理
- 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. 如何解决 props 层级过深的问题

- 使用Context API：提供一种组件之间的状态共享，而不必通过显式组件树逐层传递props；
- 使用Redux等状态库。

5. 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

五、路由

1. React-Router的实现原理是什么?

客户端路由实现的思想：

- 基于 hash 的路由：通过监听`hashchange`事件，感知 hash 的变化
-
  - 改变 hash 可以直接通过 location.hash=xxx
- 基于 H5 history 路由：
-
  - 改变 url 可以通过 history.pushState 和 resplaceState 等，会将URL压入堆栈，同时能够应用 `history.go()` 等 API
  - 监听 url 的变化可以通过自定义事件触发实现

react-router 实现的思想：

- 基于 `history` 库来实现上述不同的客户端路由实现思想，并且能够保存历史记录等，磨平浏览器差异，上层无感知
- 通过维护的列表，在每次 URL 发生变化的回收，通过配置的 路由路径，匹配到对应的 Component，并且 render

2. 如何配置 React-Router 实现路由切换

（1）使用 组件

路由匹配是通过比较 的 path 属性和当前地址的 pathname 来实现的。当一个 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 将始终被匹配。

```
// 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 组件和 组件

用于将 分组。

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

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

（3）使用 、 、 组件

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 ，都会在应用程序的 HTML 中渲染锚（）。

```
<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"。

```
// 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，当一个渲染时，它将使用它的to属性进行定向。

3. React-Router怎么设置重定向?

使用组件实现路由的重定向：

```
<Switch>
  <Redirect from='/users/:id' to='/users/profile/:id'/>
  <Route path='/users/profile/:id' component={Profile}/>
</Switch>
```

当请求 `/users/:id` 被重定向去 `'/users/profile/:id'`：

- 属性 `from: string`：需要匹配的将要被重定向路径。
- 属性 `to: string`：重定向的 URL 字符串
- 属性 `to: object`：重定向的 location 对象
- 属性 `push: bool`：若为真，重定向操作将会把新地址加入到访问历史记录里面，并且无法回退到前面的页面。

4. react-router 里的 Link 标签和 a 标签的区别

从最终渲染的 DOM 来看，这两者都是链接，都是 标签，区别是:

是react-router 里实现路由跳转的链接，一般配合 使用，react-router接管了其默认的链接跳转行为，区别于传统的页面跳转， 的“跳转”行为只会触发相匹配的对应的页面内容更新，而不会刷新整个页面。 做了3件事情:

- 有onclick那就执行onclick
- click的时候阻止a标签默认事件
- 根据跳转href(即是to)，用history (web前端路由两种方式之一，history & hash)跳转，此时只是链接变了，并没有刷新页面而标签就是普通的超链接了，用于从当前页面跳转到href指向的另一 个页面(非锚点情况)。

a标签默认事件禁掉之后做了什么才实现了跳转?

```
let domArr = document.getElementsByTagName('a')
[...domArr].forEach(item=>{
    item.addEventListener('click',function () {
        location.href = this.href
    })
})
```

5. React-Router如何获取URL的参数和历史对象?

（1）获取URL的参数

- **get传值**

路由配置还是普通的配置，如：`'admin'`，传参方式如：`'admin?id='1111''`。通过`this.props.location.search`获取url获取到一个字符串`'?id='1111'`

可以用url，qs，querystring，浏览器提供的api URLSearchParams对象或者自己封装的方法去解析出id的值。

- **动态路由传值**

路由需要配置成动态路由：如`path='/admin/:id'`，传参方式，如`'admin/111'`。通过`this.props.match.params.id` 取得url中的动态路由id部分的值，除此之外还可以通过`useParams (Hooks)`来获取

- **通过query或state传值**

传参方式如：在Link组件的to属性中可以传递对象`{pathname:'/admin',query:'111',state:'111'};`。通过`this.props.location.state`或`this.props.location.query`来获取即可，传递的参数可以是对象、数组等，但是存在缺点就是只要刷新页面，参数就会丢失。

（2）获取历史对象

- 如果React >= 16.8 时可以使用 React Router中提供的Hooks

```
import { useHistory } from "react-router-dom";
let history = useHistory();
```

2.使用this.props.history获取历史对象

```
let history = this.props.history;
```

6. React-Router 4怎样在路由变化时重新渲染同一个组件?

当路由变化时，即组件的props发生了变化，会调用componentWillReceiveProps等生命周期钩子。那需要做的只是： 当路由改变时，根据路由，也去请求数据：

```
class NewsList extends Component {
  componentDidMount () {
     this.fetchData(this.props.location);
  }

  fetchData(location) {
    const type = location.pathname.replace('/', '') || 'top'
    this.props.dispatch(fetchListData(type))
  }
  componentWillReceiveProps(nextProps) {
     if (nextProps.location.pathname != this.props.location.pathname) {
         this.fetchData(nextProps.location);
     }
  }
  render () {
    ...
  }
}
```

利用生命周期componentWillReceiveProps，进行重新render的预处理操作。

7. React-Router的路由有几种模式?

React-Router 支持使用 hash（对应 HashRouter）和 browser（对应 BrowserRouter） 两种路由规则， react-router-dom 提供了 BrowserRouter 和 HashRouter 两个组件来实现应用的 UI 和 URL 同步：

- BrowserRouter 创建的 URL 格式：http://xxx.com/path
- HashRouter 创建的 URL 格式：http://xxx.com/#/path

（1）BrowserRouter

它使用 HTML5 提供的 history API（pushState、replaceState 和 popstate 事件）来保持 UI 和 URL 的同步。由此可以看出，**BrowserRouter 是使用 HTML 5 的 history API 来控制路由跳转的：**

```
<BrowserRouter
    basename={string}
    forceRefresh={bool}
    getUserConfirmation={func}
```

```
    keyLength={number}
/>
```

其中的属性如下：

- basename 所有路由的基准 URL。basename 的正确格式是前面有一个前导斜杠，但不能有尾部斜杠；

```
<BrowserRouter basename="/calendar">
    <Link to="/today" />
</BrowserRouter>
```

等同于

```
<a href="/calendar/today" />
```

- forceRefresh 如果为 true，在导航的过程中整个页面将会刷新。一般情况下，只有在不支持 HTML5 history API 的浏览器中使用此功能；
- getUserConfirmation 用于确认导航的函数，默认使用 window.confirm。例如，当从 /a 导航至 /b 时，会使用默认的 confirm 函数弹出一个提示，用户点击确定后才进行导航，否则不做任何处理；

```
// 这是默认的确认函数
const getConfirmation = (message, callback) => {
  const allowTransition = window.confirm(message);
  callback(allowTransition);
}
<BrowserRouter getUserConfirmation={getConfirmation} />
```

> 需要配合`<Prompt>`一起使用。

- KeyLength 用来设置 Location.Key 的长度。

（2）HashRouter

使用 URL 的 hash 部分（即 window.location.hash）来保持 UI 和 URL 的同步。由此可以看出，**HashRouter 是通过 URL 的 hash 属性来控制路由跳转的：**

```
<HashRouter
    basename={string}
    getUserConfirmation={func}
    hashType={string}
/>
```

其中的参数如下：

- basename, getUserConfirmation 和 `BrowserRouter` 功能一样；
- hashType window.location.hash 使用的 hash 类型，有如下几种：
-
    - slash - 后面跟一个斜杠，例如 #/ 和 #/sunshine/lollipops；
    - noslash - 后面没有斜杠，例如 # 和 #sunshine/lollipops；
    - hashbang - Google 风格的 ajax crawlable，例如 #!/ 和 #!/sunshine/lollipops。

8. React-Router 4的Switch有什么用?

Switch 通常被用来包裹 Route，用于渲染与路径匹配的第一个子 `<Route>` 或 `<Redirect>`，它里面不能放其他元素。

假如不加 `<Switch>`：

```
import { Route } from 'react-router-dom'

<Route path="/" component={Home}></Route>
<Route path="/login" component={Login}></Route>
```

Route 组件的 path 属性用于匹配路径，因为需要匹配 `/` 到 `Home`，匹配 `/login` 到 `Login`，所以需要两个 Route，但是不能这么写。这样写的话，当 URL 的 path 为 “/login” 时，`<Route path="/" />`和`<Route path="/login" />` 都会被匹配，因此页面会展示 Home 和 Login 两个组件。这时就需要借助 `<Switch>` 来做到只显示一个匹配组件：

```
import { Switch, Route} from 'react-router-dom'

<Switch>
    <Route path="/" component={Home}></Route>
    <Route path="/login" component={Login}></Route>
</Switch>
```

此时，再访问 “/login” 路径时，却只显示了 Home 组件。这是就用到了exact属性，它的作用就是精确匹配路径，经常与`<Switch>` 联合使用。只有当 URL 和该 `<Route>` 的 path 属性完全一致的情况下才能匹配上：

```
import { Switch, Route} from 'react-router-dom'

<Switch>
   <Route exact path="/" component={Home}></Route>
   <Route exact path="/login" component={Login}></Route>
</Switch>
```

六、Redux

1. 对 Redux 的理解，主要解决什么问题

React是视图层框架。Redux是一个用来管理数据状态和UI状态的JavaScript应用工具。随着JavaScript单页应用（SPA）开发日趋复杂， JavaScript需要管理比任何时候都要多的state（状态）， Redux就是降低管理难度的。（Redux支持React、Angular、jQuery甚至纯JavaScript）。

在 React 中，UI 以组件的形式来搭建，组件之间可以嵌套组合。但 React 中组件间通信的数据流是单向的，顶层组件可以通过 props 属性向下层组件传递数据，而下层组件不能向上层组件传递数据，兄弟组件之间同样不能。这样简单的单向数据流支撑起了 React 中的数据可控性。

当项目越来越大的时候，管理数据的事件或回调函数将越来越多，也将越来越不好管理。管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个 model 的变化，依次地，可能会引起另一个 view 的变化。直至你搞不清楚到底发生了什么。state 在什么时候，由于什么原因，如何变化已然不受控制。 当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。如果这还不够糟糕，考虑一些来自前端开发领域的新需求，如更新调优、服务端渲染、路由跳转前请求数据等。state 的管理在大项目中相当复杂。

Redux 提供了一个叫 store 的统一仓储库，组件通过 dispatch 将 state 直接传入store，不用通过其他的组件。并且组件通过 subscribe 从 store获取到 state 的改变。使用了 Redux，所有的组件都可以从 store 中获取到所需的 state，他们也能从store 获取到 state 的改变。这比组件之间互相传递数据清晰明朗的多。

主要解决的问题：

单纯的Redux只是一个状态机，是没有UI呈现的，react- redux作用是将Redux的状态机和React的UI呈现绑定在一起，当你dispatch action改变state的时候，会自动更新页面。

2. Redux 原理及工作流程

（1）原理

Redux源码主要分为以下几个模块文件

- compose.js 提供从右到左进行函数式编程
- createStore.js 提供作为生成唯一store的函数
- combineReducers.js 提供合并多个reducer的函数，保证store的唯一性
- bindActionCreators.js 可以让开发者在不直接接触dispacth的前提下进行更改state的操作
- applyMiddleware.js 这个方法通过中间件来增强dispatch的功能

```
const actionTypes = {
    ADD: 'ADD',
    CHANGEINFO: 'CHANGEINFO',
}

const initState = {
    info: '初始化',
}

export default function initReducer(state=initState, action) {
    switch(action.type) {
        case actionTypes.CHANGEINFO:
            return {
                ...state,
                info: action.preload.info || '',
```

```
            }
        default:
            return { ...state };
    }
}

export default function createStore(reducer, initialState, middleFunc) {

    if (initialState && typeof initialState === 'function') {
        middleFunc = initialState;
        initialState = undefined;
    }

    let currentState = initialState;

    const listeners = [];

    if (middleFunc && typeof middleFunc === 'function') {
        // 封装dispatch
        return middleFunc(createStore)(reducer, initialState);
    }

    const getState = () => {
        return currentState;
    }

    const dispatch = (action) => {
        currentState = reducer(currentState, action);

        listeners.forEach(listener => {
            listener();
        })
    }

    const subscribe = (listener) => {
        listeners.push(listener);
    }

    return {
        getState,
        dispatch,
        subscribe
    }
}
```

（2）工作流程

- const store= createStore（fn）生成数据;
- action: {type: Symble('action01), payload:'payload' }定义行为;
- dispatch发起action：store.dispatch(doSomething('action001'));
- reducer：处理action，返回新的state;

通俗点解释：

- 首先，用户（通过View）发出Action，发出方式就用到了dispatch方法
- 然后，Store自动调用Reducer，并且传入两个参数：当前State和收到的Action，Reducer会返回新的State
- State一旦有变化，Store就会调用监听函数，来更新View

以 store 为核心，可以把它看成数据存储中心，但是他要更改数据的时候不能直接修改，数据修改更新的角色由Reducers来担任，store只做存储，中间人，当Reducers的更新完成以后会通过store的订阅来通知react component，组件把新的状态重新获取渲染，组件中也能主动发送action，创建action后这个动作是不会执行的，所以要dispatch这个action，让store通过reducers去做更新React Component 就是react的每个组件。

3. 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

```
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_EXTENSION_COMPOSE__({}) : compose;
// 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

```
/**
  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 {function} 真正需要生成的action对应的actionCreator
  @return {function}
*/
// 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

```
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

```
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_EXTENSION_COMPOSE__({}) : compose;
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

```
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

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

4. Redux 怎么实现属性传递，介绍下原理

react-redux 数据传输: view-->action-->reducer-->store-->view。看下点击事件的数据是如何通过redux传到view上：

- view 上的AddClick 事件通过mapDispatchToProps 把数据传到action ---> click:()=>dispatch(ADD)
- action 的ADD 传到reducer上
- reducer传到store上 const store = createStore(reducer);
- store再通过 mapStateToProps 映射穿到view上text:State.text

代码示例:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux';
import { Provider, connect } from 'react-redux';
class App extends React.Component{
    render(){
        let { text, click, clickR } = this.props;
        return(
            <div>
                <div>数据:已有人{text}</div>
                <div onClick={click}>加人</div>
                <div onClick={clickR}>减人</div>
            </div>
        )
    }
}
const initialState = {
    text:5
}
const reducer = function(state,action){
    switch(action.type){
        case 'ADD':
            return {text:state.text+1}
        case 'REMOVE':
            return {text:state.text-1}
        default:
            return initialState;
    }
}

let ADD = {
    type:'ADD'
}
```

```
let Remove = {
    type:'REMOVE'
}

const store = createStore(reducer);

let mapStateToProps = function (state){
    return{
        text:state.text
    }
}

let mapDispatchToProps = function(dispatch){
    return{
        click:()=>dispatch(ADD),
        clickR:()=>dispatch(Remove)
    }
}

const App1 = connect(mapStateToProps,mapDispatchToProps)(App);

ReactDOM.render(
    <Provider store = {store}>
        <App1></App1>
    </Provider>,document.getElementById('root')
)
```

5. Redux 中间件是什么？接受几个参数？柯里化函数两端的参数具体是什么？

Redux 的中间件提供的是位于 action 被发起之后，到达 reducer 之前的扩展点，换而言之，原本 view -→> action -> reducer -> store 的数据流加上中间件后变成了 view -> action -> middleware -> reducer -> store，在这一环节可以做一些"副作用"的操作，如异步请求、打印日志等。

applyMiddleware源码：

```
export default function applyMiddleware(...middlewares) {
    return createStore => (...args) => {
        // 利用传入的createStore和reducer和创建一个store
        const store = createStore(...args)
        let dispatch = () => {
            throw new Error()
        }
        const middlewareAPI = {
            getState: store.getState,
            dispatch: (...args) => dispatch(...args)
        }
        // 让每个 middleware 带着 middlewareAPI 这个参数分别执行一遍
        const chain = middlewares.map(middleware => middleware(middlewareAPI))
        // 接着 compose 将 chain 中的所有匿名函数，组装成一个新的函数，即新的 dispatch
        dispatch = compose(...chain)(store.dispatch)
        return {
```

```
            ...store,
            dispatch
        }
    }
}
```

从applyMiddleware中可以看出:

- redux中间件接受一个对象作为参数，对象的参数上有两个字段 dispatch 和 getState，分别代表着 Redux Store 上的两个同名函数。
- 柯里化函数两端一个是 middewares，一个是store.dispatch

6. 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。

```
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。

```
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))
    }
})
```

7. Redux 状态管理器和变量挂载到 window 中有什么区别

两者都是存储数据以供后期使用。但是Redux状态更改可回溯——Time travel，数据多了的时候可以很清晰的知道改动在哪里发生，完整的提供了一套状态管理模式。

随着 JavaScript 单页应用开发日趋复杂，JavaScript 需要管理比任何时候都要多的 state （状态）。 这些 state 可能包括服务器响应、缓存数据、本地生成尚未持久化到服务器的数据，也包括 UI状态，如激活的路由，被选中的标签，是否显示加载动效或者分页器等等。

管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个model 的变化，依次地，可能会引起另一个 view 的变化。直至你搞不清楚到底发生了什么。state 在什么时候，由于什么原因，如何变化已然不受控制。 当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。

如果这还不够糟糕，考虑一些来自前端开发领域的新需求，如更新调优、服务端渲染、路由跳转前请求数据等等。前端开发者正在经受前所未有的复杂性，难道就这么放弃了吗?当然不是。

这里的复杂性很大程度上来自于：我们总是将两个难以理清的概念混淆在一起：变化和异步。 可以称它们为曼妥思和可乐。如果把二者分开，能做的很好，但混到一起，就变得一团糟。一些库如 React 视图在视图层禁止异步和直接操作 DOM来解决这个问题。美中不足的是，React 依旧把处理 state 中数据的问题留给了你。Redux就是为了帮你解决这个问题。

8. mobox 和 redux 有什么区别?

（1）共同点

- 为了解决状态管理混乱，无法有效同步的问题统一维护管理应用状态;
- 某一状态只有一个可信数据来源（通常命名为store，指状态容器）；
- 操作更新状态方式统一，并且可控（通常以action方式提供更新状态的途径）；
- 支持将store与React组件连接，如react-redux，mobx- react;

（2）区别

Redux更多的是遵循Flux模式的一种实现，是一个 JavaScript库，它关注点主要是以下几方面:

- Action: 一个JavaScript对象，描述动作相关信息，主要包含type属性和payload属性:

o type: action 类型;

o payload: 负载数据;

- Reducer: 定义应用状态如何响应不同动作（action），如何更新状态;
- Store: 管理action和reducer及其关系的对象，主要提供以下功能:

o 维护应用状态并支持访问状态(getState());

o 支持监听action的分发，更新状态(dispatch(action));

o 支持订阅store的变更(subscribe(listener));

- 异步流: 由于Redux所有对store状态的变更，都应该通过action触发，异步任务（通常都是业务或获取数据任务）也不例外，而为了不将业务或数据相关的任务混入React组件中，就需要使用其他框架配合管理异步任务流程，如redux-thunk，redux-saga等;

Mobx是一个透明函数响应式编程的状态管理库，它使得状态管理简单可伸缩:

- Action:定义改变状态的动作函数，包括如何变更状态;
- Store: 集中管理模块状态（State）和动作(action)
- Derivation（衍生）: 从应用状态中派生而出，且没有任何其他影响的数据

对比总结:

- redux将数据保存在单一的store中，mobx将数据保存在分散的多个store中
- redux使用plain object保存数据，需要手动处理变化后的操作;mobx适用observable保存数据，数据变化后自动处理响应的操作
- redux使用不可变状态，这意味着状态是只读的，不能直接去修改它，而是应该返回一个新的状态，同时使用纯函数;mobx中的状态是可变的，可以直接对其进行修改
- mobx相对来说比较简单，在其中有很多的抽象，mobx更多的使用面向对象的编程思维;redux会比较复杂，因为其中的函数式编程思想掌握起来不是那么容易，同时需要借助一系列的中间件来处理异步和副作用
- mobx中有更多的抽象和封装，调试会比较困难，同时结果也难以预测;而redux提供能够进行时间回溯的开发工具，同时其纯函数以及更少的抽象，让调试变得更加的容易

9. Redux 和 Vuex 有什么区别，它们的共同思想

（1）Redux 和 Vuex区别

- Vuex改进了Redux中的Action和Reducer函数，以mutations变化函数取代Reducer，无需switch，只需在对应的mutation函数里改变state值即可
- Vuex由于Vue自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的State即可
- Vuex数据流的顺序是:View调用store.commit提交对应的请求到Store中对应的mutation函数->store改变（vue检测到数据变化自动渲染）

通俗点理解就是，vuex 弱化 dispatch，通过commit进行 store状态的一次更变；取消了action概念，不必传入特定的 action形式进行指定变更；弱化reducer，基于commit参数直接对数据进行转变，使得框架更加简易;

（2）共同思想

- 单—的数据源
- 变化可以预测

本质上: redux与vuex都是对mvvm思想的服务，将数据从视图中抽离的一种方案。

10. Redux 中间件是怎么拿到store 和 action? 然后怎么处理?

redux中间件本质就是一个函数柯里化。redux applyMiddleware Api 源码中每个middleware 接受2个参数，Store 的getState 函数和dispatch 函数，分别获得store和action，最终返回一个函数。该函数会被传入 next 的下一个 middleware 的 dispatch 方法，并返回一个接收 action 的新函数，这个函数可以直接调用 next（action），或者在其他需要的时刻调用，甚至根本不去调用它。调用链中最后一个 middleware 会接受真实的 store的 dispatch 方法作为 next 参数，并借此结束调用链。所以，middleware 的函数签名是（{ getState，dispatch })=> next => action。

11. Redux中的connect有什么作用

connect负责连接React和Redux

（1）获取state

connect 通过 context获取 Provider 中的 store，通过 `store.getState()` 获取整个store tree 上所有state

（2）包装原组件

将state和action通过props的方式传入到原组件内部 wrapWithConnect 返回一个 ReactComponent 对 象 Connect，Connect 重 新 render 外部传入的原组件 WrappedComponent ，并把 connect 中传入的 mapStateToProps，mapDispatchToProps与组件上原有的 props合并后，通过属性的方式传给 WrappedComponent

（3）监听store tree变化

connect缓存了store tree中state的状态，通过当前state状态 和变更前 state 状态进行比较，从而确定是否调用 `this.setState()`方法触发Connect及其子组件的重新渲染

七、Hooks

1. 对 React Hook 的理解，它的实现原理是什么

React-Hooks 是 React 团队在 React 组件开发实践中，逐渐认知到的一个改进点，这背后其实涉及对**类组件**和**函数组****件**两种组件形式的思考和侧重。

（1）类组件：所谓类组件，就是基于 ES6 Class 这种写法，通过继承 React.Component 得来的 React 组件。以下是一个类组件：

```
class DemoClass extends React.Component {
  state = {
    text: ""
  };
  componentDidMount() {
    //...
  }
  changeText = (newText) => {
    this.setState({
      text: newText
    });
  };
```

```
  render() {
    return (
      <div className="demoClass">
        <p>{this.state.text}</p>
        <button onClick={this.changeText}>修改</button>
      </div>
    );
  }
}
```

可以看出，React 类组件内部预置了相当多的“现成的东西”等着我们去调度/定制，state 和生命周期就是这些“现成东西”中的典型。要想得到这些东西，难度也不大，只需要继承一个 React.Component 即可。

当然，这也是类组件的一个不便，它太繁杂了，对于解决许多问题来说，编写一个类组件实在是一个过于复杂的姿势。复杂的姿势必然带来高昂的理解成本，这也是我们所不想看到的。除此之外，由于开发者编写的逻辑在封装后是和组件粘在一起的，这就使得**类组件内部的逻辑难以实现拆分和复用。**

（2）函数组件：函数组件就是以函数的形态存在的 React 组件。早期并没有 React-Hooks，函数组件内部无法定义和维护 state，因此它还有一个别名叫“无状态组件”。以下是一个函数组件：

```
function DemoFunction(props) {
  const { text } = props
  return (
    <div className="demoFunction">
      <p>{`函数组件接收的内容：[${text}]`}</p>
    </div>
  );
}
```

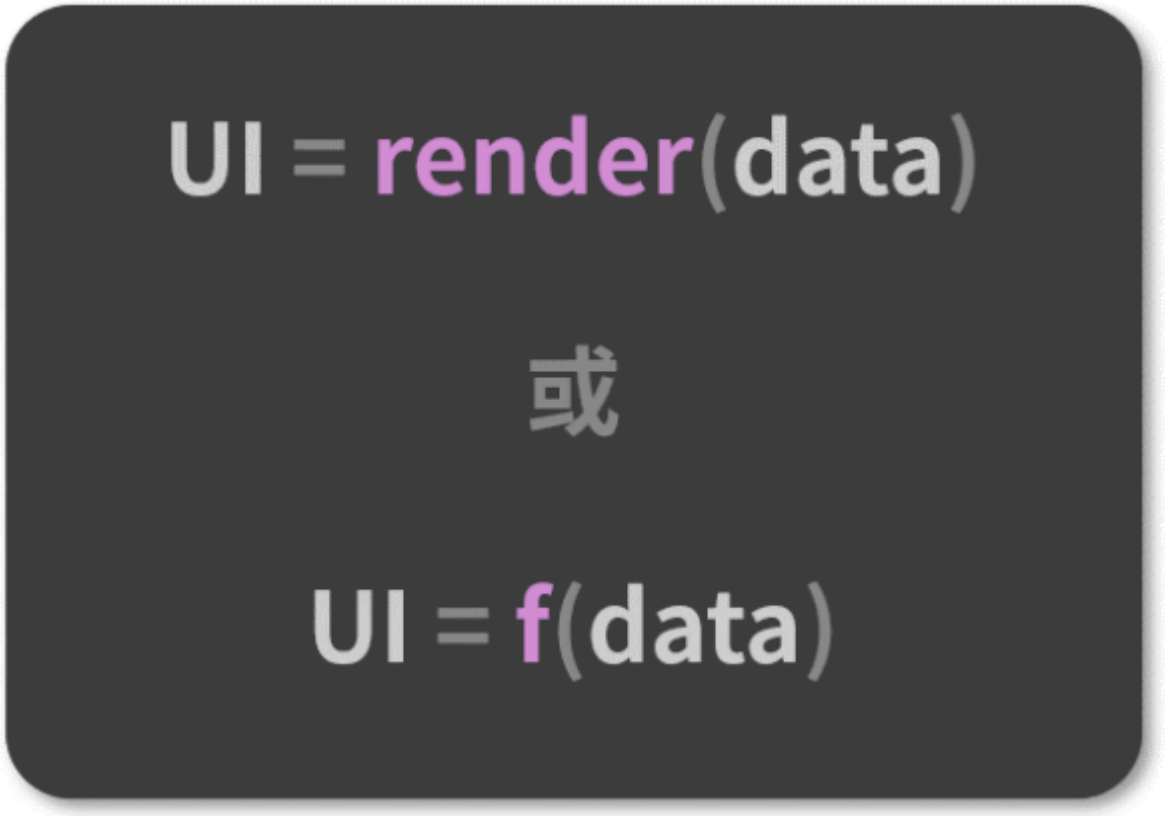
相比于类组件，函数组件肉眼可见的特质自然包括轻量、灵活、易于组织和维护、较低的学习成本等。

通过对比，从形态上可以对两种组件做区分，它们之间的区别如下：

- 类组件需要继承 class，函数组件不需要；
- 类组件可以访问生命周期方法，函数组件不能；
- 类组件中可以获取到实例化后的 this，并基于这个 this 做各种各样的事情，而函数组件不可以；
- 类组件中可以定义并维护 state（状态），而函数组件不可以；

除此之外，还有一些其他的不同。通过上面的区别，我们不能说谁好谁坏，它们各有自己的优势。在 React-Hooks 出现之前，**类组件的能力边界明显强于函数组件。**

实际上，类组件和函数组件之间，是面向对象和函数式编程这两套不同的设计思想之间的差异。而函数组件更加契合 React 框架的设计理念：



React 组件本身的定位就是函数，一个输入数据、输出 UI 的函数。作为开发者，我们编写的是声明式的代码，而 React 框架的主要工作，就是及时地把声明式的代码转换为命令式的 DOM 操作，把数据层面的描述映射到用户可见的 UI 变化中去。这就意味着从原则上来讲，React 的数据应该总是紧紧地和渲染绑定在一起的，而类组件做不到这一点。**函数组件就真正地将数据和渲染绑定到了一起。**__函数组件是一个更加匹配其设计理念、也更有利于逻辑拆分与重用的组件表达形式。__

为了能让开发者更好的的去编写函数式组件。于是，React-Hooks 便应运而生。

React-Hooks 是一套能够使函数组件更强大、更灵活的“钩子”。

函数组件比起类组件少了很多东西，比如生命周期、对 state 的管理等。这就给函数组件的使用带来了非常多的局限性，导致我们并不能使用函数这种形式，写出一个真正的全功能的组件。而React-Hooks 的出现，就是为了帮助函数组件补齐这些（相对于类组件来说）缺失的能力。

如果说函数组件是一台轻巧的快艇，那么 React-Hooks 就是一个内容丰富的零部件箱。“重装战舰”所预置的那些设备，这个箱子里基本全都有，同时它还不强制你全都要，而是允许你自由地选择和使用你需要的那些能力，然后将这些能力以 Hook（钩子）的形式“钩”进你的组件里，从而定制出一个最适合你的“专属战舰”。

2. 为什么 useState 要使用数组而不是对象

useState 的用法：

```
const [count, setCount] = useState(0)
```

可以看到 useState 返回的是一个数组，那么为什么是返回数组而不是返回对象呢？

这里用到了解构赋值，所以先来看一下ES6 的解构赋值：

数组的解构赋值

```
const foo = [1, 2, 3];
const [one, two, three] = foo;
console.log(one);   // 1
console.log(two);   // 2
console.log(three); // 3
```

对象的解构赋值

```
const user = {
  id: 888,
  name: "xiaoxin"
};
const { id, name } = user;
console.log(id);    // 888
console.log(name);  // "xiaoxin"
```

看完这两个例子，答案应该就出来了：

- 如果 useState 返回的是数组，那么使用者可以对数组中的元素命名，代码看起来也比较干净
- 如果 useState 返回的是对象，在解构对象的时候必须要和 useState 内部实现返回的对象同名，想要使用多次的话，必须得设置别名才能使用返回值

下面来看看如果 useState 返回对象的情况：

```
// 第一次使用
const { state, setState } = useState(false);
// 第二次使用
const { state: counter, setState: setCounter } = useState(0)
```

这里可以看到，返回对象的使用方式还是挺麻烦的，更何况实际项目中会使用的更频繁。

总结：useState 返回的是 array 而不是 object 的原因就是为了降低使用的复杂度，返回数组的话可以直接根据顺序解构，而返回对象的话要想使用多次就需要定义别名了。

3. 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

4. React Hook 的使用限制有哪些?

React Hooks 的限制主要有两条：

- 不要在循环、条件或嵌套函数中调用 Hook；
- 在 React 的函数组件中调用 Hook。

那为什么会有这样的限制呢？Hooks 的设计初衷是为了改进 React 组件的开发模式。在旧有的开发模式下遇到了三个问题。

- 组件之间难以复用状态逻辑。过去常见的解决方案是高阶组件、render props 及状态管理框架。
- 复杂的组件变得难以理解。生命周期函数与业务逻辑耦合太深，导致关联部分难以拆分。
- 人和机器都很容易混淆类。常见的有 this 的问题，但在 React 团队中还有类难以优化的问题，希望在编译优化层面做出一些改进。

这三个问题在一定程度上阻碍了 React 的后续发展，所以为了解决这三个问题，Hooks **基于函数组件**开始设计。然而第三个问题决定了 Hooks 只支持函数组件。

那为什么不要在循环、条件或嵌套函数中调用 Hook 呢？因为 Hooks 的设计是基于数组实现。在调用时按顺序加入数组中，如果使用循环、条件或嵌套函数很有可能导致数组取值错位，执行错误的 Hook。当然，实质上 React 的源码里不是数组，是链表。

这些限制会在编码上造成一定程度的心智负担，新手可能会写错，为了避免这样的情况，可以引入 ESLint 的 Hooks 检查插件进行预防。

5. useEffect 与 useLayoutEffect 的区别

（1）共同点

- **运用效果：**useEffect 与 useLayoutEffect 两者都是用于处理副作用，这些副作用包括改变 DOM、设置订阅、操作定时器等。在函数组件内部操作副作用是不被允许的，所以需要使用这两个函数去处理。
- **使用方式：**useEffect 与 useLayoutEffect 两者底层的函数签名是完全一致的，都是调用的mountEffectImpl方法，在使用上也没什么差异，基本可以直接替换。

（2）不同点

- **使用场景：**useEffect 在 React 的渲染过程中是被异步调用的，用于绝大多数场景；而 useLayoutEffect 会在所有的 DOM 变更之后同步调用，主要用于处理 DOM 操作、调整样式、避免页面闪烁等问题。也正因为是同步处理，所以需要避免在 useLayoutEffect 做计算量较大的耗时任务从而造成阻塞。
- **使用效果：**useEffect是按照顺序执行代码的，改变屏幕像素之后执行（先渲染，后改变DOM），当改变屏幕内容时可能会产生闪烁；useLayoutEffect是改变屏幕像素之前就执行了（会推迟页面显示的事件，先改变DOM后渲染），不会产生闪烁。**useLayoutEffect总是比useEffect先执行。**

在未来的趋势上，两个 API 是会长期共存的，暂时没有删减合并的计划，需要开发者根据场景去自行选择。React 团队的建议非常实用，如果实在分不清，先用 useEffect，一般问题不大；如果页面有异常，再直接替换为 useLayoutEffect 即可。

6. 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：

```
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={index}>{item}</div>
                      ))}
                  </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：

```
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。

7. React Hooks 和生命周期的关系?

函数组件 的本质是函数，没有 state 的概念的，因此**不存在生命周期**一说，仅仅是一个 **render 函数**而已。

但是引入 **Hooks** 之后就变得不同了，它能让组件在不使用 class 的情况下拥有 state，所以就有了生命周期的概念，所谓的生命周期其实就是 `useState`、 `useEffect()` 和 `useLayoutEffect()` 。

即：**Hooks 组件（使用了Hooks的函数组件）有生命周期，而函数组件（未使用Hooks的函数组件）是没有生命周期的。**

下面是具体的 class 与 Hooks 的**生命周期对应关系**：

- `constructor`：函数组件不需要构造函数，可以通过调用 `**useState** **来初始化 state**`。如果计算的代价比较昂贵，也可以传一个函数给 `useState`。

```
const [num, UpdateNum] = useState(0)
```

- `getDerivedStateFromProps`：一般情况下，我们不需要使用它，可以在**渲染过程中更新 state**，以达到实现 `getDerivedStateFromProps` 的目的。

```
function ScrollView({row}) {
  let [isScrollingDown, setIsScrollingDown] = useState(false);
  let [prevRow, setPrevRow] = useState(null);
  if (row !== prevRow) {
    // Row 自上次渲染以来发生过改变。更新 isScrollingDown。
    setIsScrollingDown(prevRow !== null && row > prevRow);
    setPrevRow(row);
  }
  return `Scrolling down: ${isScrollingDown}`;
}
```

React 会立即退出第一次渲染并用更新后的 state 重新运行组件以避免耗费太多性能。

- shouldComponentUpdate：可以用 **React.memo** 包裹一个组件来对它的 props 进行浅比较

```
const Button = React.memo((props) => {
  // 具体的组件
});
```

注意：**React.memo** **等效于** ``**PureComponent**，它只浅比较 props。这里也可以使用 useMemo 优化每一个节点。

- render：这是函数组件体本身。
- componentDidMount, componentDidUpdate： useLayoutEffect 与它们两的调用阶段是一样的。但是，我们推荐你**一开始先用 useEffect**，只有当它出问题的时候再尝试使用 useLayoutEffect。useEffect 可以表达所有这些的组合。

```
// componentDidMount
useEffect(()=>{
  // 需要在 componentDidMount 执行的内容
}, [])
useEffect(() => {
  // 在 componentDidMount，以及 count 更改时 componentDidUpdate 执行的内容
  document.title = `You clicked ${count} times`;
  return () => {
    // 需要在 count 更改时 componentDidUpdate（先于 document.title = ... 执行，遵守先清理后更新）
    // 以及 componentWillUnmount 执行的内容
  } // 当函数中 Cleanup 函数会按照在代码中定义的顺序先后执行，与函数本身的特性无关
}, [count]); // 仅在 count 更改时更新
```

请记得 React 会等待浏览器完成画面渲染之后才会延迟调用，因此会使得额外操作很方便

- componentWillUnmount：相当于 useEffect 里面返回的 cleanup 函数

```
// componentDidMount/componentWillUnmount
useEffect(()=>{
  // 需要在 componentDidMount 执行的内容
  return function cleanup() {
    // 需要在 componentWillUnmount 执行的内容
  }
}, [])
```

- componentDidCatch and getDerivedStateFromError：目前**还没有**这些方法的 Hook 等价写法，但很快会加上。

class 组件	Hooks 组件
constructor	useState

class 组件	Hooks 组件
getDerivedStateFromProps	useState 里面 update 函数
shouldComponentUpdate	useMemo
render	函数本身
componentDidMount	useEffect
componentDidUpdate	useEffect
componentWillUnmount	useEffect 里面返回的函数
componentDidCatch	无
getDerivedStateFromError	无

八、虚拟DOM

1. 对虚拟 DOM 的理解？虚拟 DOM 主要做了什么？虚拟 DOM 本身是什么？

从本质上来说，Virtual Dom是一个JavaScript对象，通过对象的方式来表示DOM结构。将页面的状态抽象为JS对象的形式，配合不同的渲染工具，使跨平台渲染成为可能。通过事务处理机制，将多次DOM修改的结果一次性的更新到页面上，从而有效的减少页面渲染的次数，减少修改DOM的重绘重排次数，提高渲染性能。

虚拟DOM是对DOM的抽象，这个对象是更加轻量级的对DOM的描述。它设计的最初目的，就是更好的跨平台，比如node.js就没有DOM，如果想实现SSR，那么一个方式就是借助虚拟dom，因为虚拟dom本身是js对象。 在代码渲染到页面之前，vue或者react会把代码转换成一个对象（虚拟DOM）。以对象的形式来描述真实dom结构，最终渲染到页面。在每次数据发生变化前，虚拟dom都会缓存一份，变化之时，现在的虚拟dom会与缓存的虚拟dom进行比较。在vue或者react内部封装了diff算法，通过这个算法来进行比较，渲染时修改改变的变化，原先没有发生改变的通过原先的数据进行渲染。

另外现代前端框架的一个基本要求就是无须手动操作DOM，一方面是因为手动操作DOM无法保证程序性能，多人协作的项目中如果review不严格，可能会有开发者写出性能较低的代码，另一方面更重要的是省略手动DOM操作可以大大提高开发效率。

为什么要用 Virtual DOM：

（1）保证性能下限，在不进行手动优化的情况下，提供过得去的性能

下面对比一下修改DOM时真实DOM操作和Virtual DOM的过程，来看一下它们重排重绘的性能消耗:

- 真实DOM: 生成HTML字符串 + 重建所有的DOM元素
- Virtual DOM: 生成vNode + DOMDiff + 必要的DOM更新

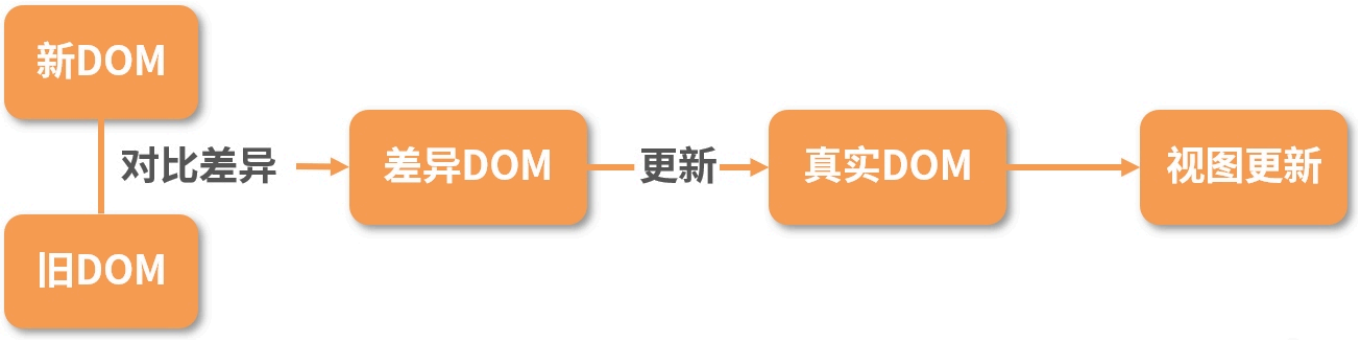
Virtual DOM的更新DOM的准备工作耗费更多的时间，也就是JS层面，相比于更多的DOM操作它的消费是极其便宜的。尤雨溪在社区论坛中说道: 框架给你的保证是，你不需要手动优化的情况下，我依然可以给你提供过得去的性能。

（2）跨平台

Virtual DOM本质上是JavaScript的对象，它可以很方便的跨平台操作，比如服务端渲染、uniapp等。

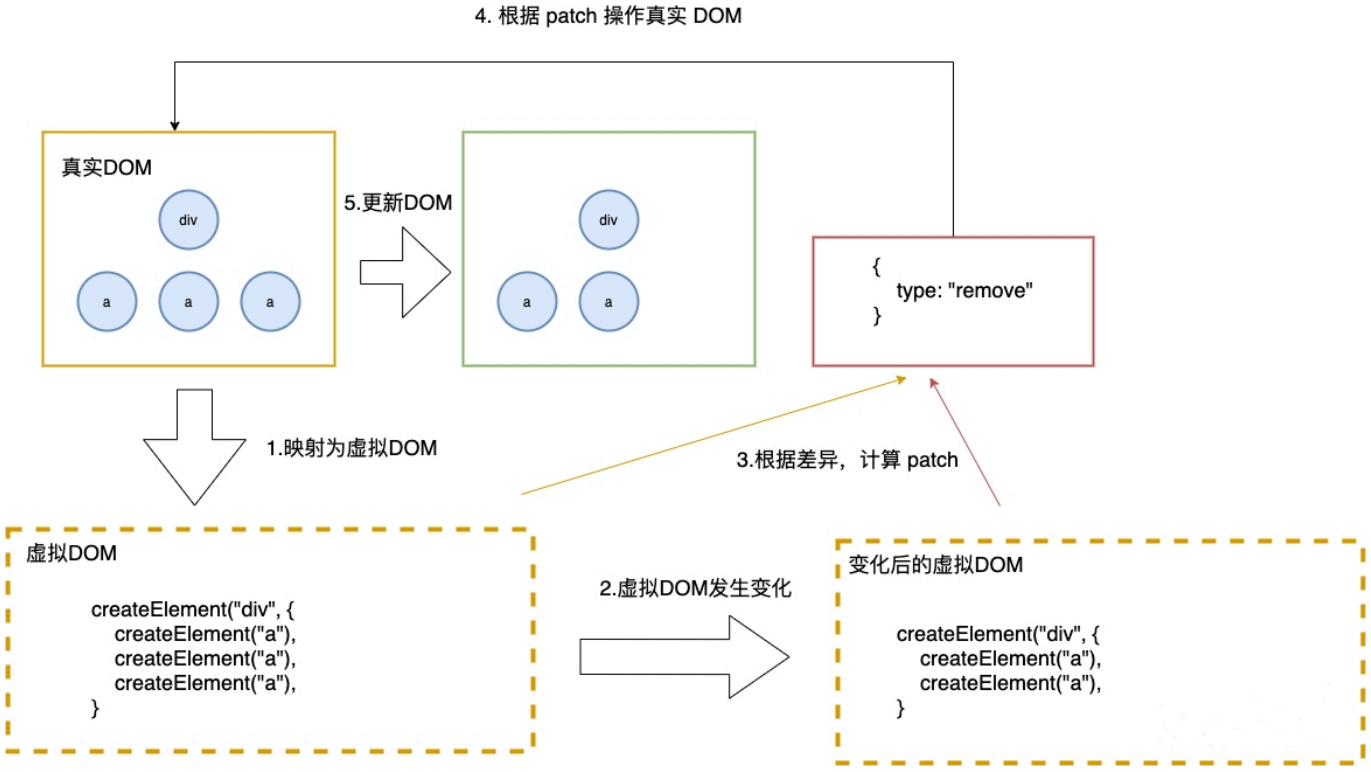
2. React diff 算法的原理是什么?

实际上，diff 算法探讨的就是虚拟 DOM 树发生变化后，生成 DOM 树更新补丁的方式。它通过对比新旧两株虚拟 DOM 树的变更差异，将更新补丁作用于真实 DOM，以最小成本完成视图更新。



具体的流程如下：

- 真实的 DOM 首先会映射为虚拟 DOM；
- 当虚拟 DOM 发生变化后，就会根据差距计算生成 patch，这个 patch 是一个结构化的数据，内容包含了增加、更新、移除等；
- 根据 patch 去更新真实的 DOM，反馈到用户的界面上。



一个简单的例子：

```
import React from 'react'
export default class ExampleComponent extends React.Component {
  render() {
    if(this.props.isVisible) {
       return <div className="visible">visbile</div>;
    }
     return <div className="hidden">hidden</div>;
```

```
    }
  }
```

这里，首先假定 ExampleComponent 可见，然后再改变它的状态，让它不可见 。映射为真实的 DOM 操作是这样的，React 会创建一个 div 节点。

```
<div class="visible">visbile</div>
```

当把 visbile 的值变为 false 时，就会替换 class 属性为 hidden，并重写内部的 innerText 为 hidden。**这样一个生成补丁、更新差异的过程统称为 diff 算法。**

diff算法可以总结为三个策略，分别从树、组件及元素三个层面进行复杂度的优化：

策略一：忽略节点跨层级操作场景，提升比对效率。（基于树进行对比）

这一策略需要进行树比对，即对树进行分层比较。树比对的处理手法是非常“暴力”的，即两棵树只对同一层次的节点进行比较，如果发现节点已经不存在了，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，这就提升了比对效率。

策略二：如果组件的 class 一致，则默认为相似的树结构，否则默认为不同的树结构。**（基于组件进行对比）**

在组件比对的过程中：

- 如果组件是同一类型则进行树比对；
- 如果不是则直接放入补丁中。

只要父组件类型不同，就会被重新渲染。这也就是为什么 shouldComponentUpdate、PureComponent 及 React.memo 可以提高性能的原因。

策略三：同一层级的子节点，可以通过标记 key 的方式进行列表对比。**（基于节点进行对比）**

元素比对主要发生在同层级中，通过标记节点操作生成补丁。节点操作包含了插入、移动、删除等。其中节点重新排序同时涉及插入、移动、删除三个操作，所以效率消耗最大，此时策略三起到了至关重要的作用。通过标记 key 的方式，React 可以直接移动 DOM 节点，降低内耗。

3. React key 是干嘛用的 为什么要加？key 主要是解决哪一类问题的

Keys 是 React 用于追踪哪些列表中元素被修改、被添加或者被移除的辅助标识。在开发过程中，我们需要保证某个元素的 key 在其同级元素中具有唯一性。

在 React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染此外，React 还需要借助 Key 值来判断元素与本地状态的关联关系。

注意事项：

- key值一定要和具体的元素—一对应；
- 尽量不要用数组的index去作为key；

- 不要在render的时候用随机数或者其他操作给元素加上不稳定的key，这样造成的性能开销比不加key的情况下更糟糕。

4. 虚拟 DOM 的引入与直接操作原生 DOM 相比，哪一个效率更高，为什么

虚拟DOM相对原生的DOM不一定是效率更高，如果只修改一个按钮的文案，那么虚拟 DOM 的操作无论如何都不可能比真实的 DOM 操作更快。在首次渲染大量DOM时，由于多了一层虚拟DOM的计算，虚拟DOM也会比innerHTML插入慢。它能保证性能下限，在真实DOM操作的时候进行针对性的优化时，还是更快的。所以要根据具体的场景进行探讨。

在整个 DOM 操作的演化过程中，其实主要矛盾并不在于性能，而在于开发者写得爽不爽，在于研发体验/研发效率。虚拟 DOM 不是别的，正是前端开发们为了追求更好的研发体验和研发效率而创造出来的高阶产物。虚拟 DOM 并不一定会带来更好的性能，React 官方也从来没有把虚拟 DOM 作为性能层面的卖点对外输出过。**虚拟 DOM 的优越之处在于，它能够在提供更爽、更高效的研发模式（也就是函数式的 UI 编程方式）的同时，仍然保持一个还不错的性能。**

5. React 与 Vue 的 diff 算法有何不同?

diff 算法是指生成更新补丁的方式，主要应用于虚拟 DOM 树变化后，更新真实 DOM。所以 diff 算法一定存在这样一个过程：触发更新 → 生成补丁 → 应用补丁。

React 的 diff 算法，触发更新的时机主要在 state 变化与 hooks 调用之后。此时触发虚拟 DOM 树变更遍历，采用了深度优先遍历算法。但传统的遍历方式，效率较低。为了优化效率，使用了分治的方式。将单一节点比对转化为了 3 种类型节点的比对，分别是树、组件及元素，以此提升效率。

- 树比对：由于网页视图中较少有跨层级节点移动，两株虚拟 DOM 树只对同一层次的节点进行比较。
- 组件比对：如果组件是同一类型，则进行树比对，如果不是，则直接放入到补丁中。
- 元素比对：主要发生在同层级中，通过标记节点操作生成补丁，节点操作对应真实的 DOM 剪裁操作。

以上是经典的 React diff 算法内容。自 React 16 起，引入了 Fiber 架构。为了使整个更新过程可随时暂停恢复，节点与树分别采用了 FiberNode 与 FiberTree 进行重构。fiberNode 使用了双链表的结构，可以直接找到兄弟节点与子节点。整个更新过程由 current 与 workInProgress 两株树双缓冲完成。workInProgress 更新完成后，再通过修改 current 相关指针指向新节点。

Vue 的整体 diff 策略与 React 对齐，虽然缺乏时间切片能力，但这并不意味着 Vue 的性能更差，因为在 Vue 3 初期引入过，后期因为收益不高移除掉了。除了高帧率动画，在 Vue 中其他的场景几乎都可以使用防抖和节流去提高响应性能。

九、其他

1. React组件命名推荐的方式是哪个?

通过引用而不是使用来命名组件displayName。

使用displayName命名组件：

```
export default React.createClass({
  displayName: 'TodoApp',
```

```
  // ...
})
```

React推荐的方法：

```
export default class TodoApp extends React.Component {
  // ...
}
```

2. react 最新版本解决了什么问题，增加了哪些东西

React 16.x的三大新特性 Time Slicing、Suspense、 hooks

- **Time Slicing（解决CPU速度问题**）使得在执行任务的期间可以随时暂停，跑去干别的事情，这个特性使得react能在性能极其差的机器跑时，仍然保持有良好的性能
- **Suspense （解决网络IO问题）**和lazy配合，实现异步加载组件。 能暂停当前组件的渲染， 当完成某件事以后再继续渲染，解决从react出生到现在都存在的「异步副作用」的问题，而且解决得非的优雅，使用的是 T异步但是同步的写法，这是最好的解决异步问题的方式
- 提供了一个**内置函数componentDidCatch**，当有错误发生时，可以友好地展示 fallback 组件; 可以捕捉到它的子元素（包括嵌套子元素）抛出的异常; 可以复用错误组件。

（1）React16.8

加入hooks，让React函数式组件更加灵活，hooks之前，React存在很多问题：

- 在组件间复用状态逻辑很难
- 复杂组件变得难以理解，高阶组件和函数组件的嵌套过深。
- class组件的this指向问题
- 难以记忆的生命周期

hooks很好的解决了上述问题，hooks提供了很多方法

- useState 返回有状态值，以及更新这个状态值的函数
- useEffect 接受包含命令式，可能有副作用代码的函数。
- useContext 接受上下文对象（从 React.createContext返回的值）并返回当前上下文值，
- useReducer useState 的替代方案。接受类型为 （state，action）=> newState的reducer，并返回与dispatch方法配对的当前状态。
- useCalLback 返回一个回忆的memoized版本，该版本仅在其中一个输入发生更改时才会更改。纯函数的输入输出确定性 o useMemo 纯的一个记忆函数 o useRef 返回一个可变的ref对象，其Current 属性被初始化为传递的参数，返回的 ref 对象在组件的整个生命周期内保持不变。
- useImperativeMethods 自定义使用ref时公开给父组件的实例值
- useMutationEffect 更新兄弟组件之前，它在React执行其DOM改变的同一阶段同步触发
- useLayoutEffect DOM改变后同步触发。使用它来从DOM读取布局并同步重新渲染

（2）React16.9

- 重命名 Unsafe 的生命周期方法。新的 UNSAFE_前缀将有助于在代码 review 和 debug 期间，使这些有问题的字样更突出

- 废弃 javascrip:形式的 URL。以javascript:开头的URL 非常容易遭受攻击，造成安全漏洞。
- 废弃"Factory"组件。 工厂组件会导致 React 变大且变慢。
- act（）也支持异步函数，并且你可以在调用它时使用 await。
- 使用 <React.ProfiLer> 进行性能评估。在较大的应用中追踪性能回归可能会很方便

（3）React16.13.0

- 支持在渲染期间调用setState，但仅适用于同一组件
- 可检测冲突的样式规则并记录警告
- 废弃 unstable_createPortal，使用CreatePortal
- 将组件堆栈添加到其开发警告中，使开发人员能够隔离bug并调试其程序，这可以清楚地说明问题所在，并更快地定位和修复错误。

3. react 实现一个全局的 dialog

```
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:

```
//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：

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

4. React 数据持久化有什么实践吗?

封装数据持久化组件：

```
】let storage={
    // 增加
    set(key, value){
        localStorage.setItem(key, JSON.stringify(value));
    },
    // 获取
    get(key){
        return JSON.parse(localStorage.getItem(key));
    },
    // 删除
```

```
    remove(key){
        localStorage.removeItem(key);
    }
};
export default Storage;
```

在React项目中，通过redux存储全局数据时，会有一个问题，如果用户刷新了网页，那么通过redux存储的全局数据就会被全部清空，比如登录信息等。这时就会有全局数据持久化存储的需求。首先想到的就是localStorage，localStorage是没有时间限制的数据存储，可以通过它来实现数据的持久化存储。

但是在已经使用redux来管理和存储全局数据的基础上，再去使用localStorage来读写数据，这样不仅是工作量巨大，还容易出错。那么有没有结合redux来达到持久数据存储功能的框架呢？当然，它就是**redux-persist**。redux-persist会将redux的store中的数据缓存到浏览器的localStorage中。其使用步骤如下：

（1）首先要安装redux-persist：

```
npm i redux-persist
```

（2）对于reducer和action的处理不变，只需修改store的生成代码，修改如下：

```
import {createStore} from 'redux'
import reducers from '../reducers/index'
import {persistStore, persistReducer} from 'redux-persist';
import storage from 'redux-persist/lib/storage';
import autoMergeLevel2 from 'redux-persist/lib/stateReconciler/autoMergeLevel2';
const persistConfig = {
    key: 'root',
    storage: storage,
    stateReconciler: autoMergeLevel2 // 查看 'Merge Process' 部分的具体情况
};
const myPersistReducer = persistReducer(persistConfig, reducers)
const store = createStore(myPersistReducer)
export const persistor = persistStore(store)
export default store
```

（3）在index.js中，将PersistGate标签作为网页内容的父标签：

```
import React from 'react';
import ReactDOM from 'react-dom';
import {Provider} from 'react-redux'
import store from './redux/store/store'
import {persistor} from './redux/store/store'
import {PersistGate} from 'redux-persist/lib/integration/react';
ReactDOM.render(<Provider store={store}>
            <PersistGate loading={null} persistor={persistor}>
                {/*网页内容*/}
```

```
            </PersistGate>
        </Provider>, document.getElementById('root'));
```

这就完成了通过redux-persist实现React持久化本地数据存储的简单应用。

5. 对 React 和 Vue 的理解，它们的异同

相似之处：

- 都将注意力集中保持在核心库，而将其他功能如路由和全局状态管理交给相关的库
- 都有自己的构建工具，能让你得到一个根据最佳实践设置的项目模板。
- 都使用了Virtual DOM（虚拟DOM）提高重绘性能
- 都有props的概念，允许组件间的数据传递
- 都鼓励组件化应用，将应用分拆成一个个功能明确的模块，提高复用性

不同之处：

1）数据流

Vue默认支持数据双向绑定，而React一直提倡单向数据流

2）虚拟DOM

Vue2.x开始引入"Virtual DOM"，消除了和React在这方面的差异，但是在具体的细节还是有各自的特点。

- Vue宣称可以更快地计算出Virtual DOM的差异，这是由于它在渲染过程中，会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。
- 对于React而言，每当应用的状态被改变时，全部子组件都会重新渲染。当然，这可以通过PureComponent/shouldComponentUpdate这个生命周期方法来进行控制，但Vue将此视为默认的优化。

3）组件化

React与Vue最大的不同是模板的编写。

- Vue鼓励写近似常规HTML的模板。写起来很接近标准 HTML元素，只是多了一些属性。
- React推荐你所有的模板通用JavaScript的语法扩展——JSX书写。

具体来讲：React中render函数是支持闭包特性的，所以我们import的组件在render中可以直接调用。但是在Vue中，由于模板中使用的数据都必须挂在 this 上进行一次中转，所以 import 完组件之后，还需要在 components 中再声明下。

4）监听数据变化的实现原理不同

- Vue 通过 getter/setter 以及一些函数的劫持，能精确知道数据变化，不需要特别的优化就能达到很好的性能
- React 默认是通过比较引用的方式进行的，如果不优化（PureComponent/shouldComponentUpdate）可能导致大量不必要的vDOM的重新渲染。这是因为 Vue 使用的是可变数据，而React更强调数据的不可变。

5）高阶组件

react可以通过高阶组件（Higher Order Components-- HOC）来扩展，而vue需要通过mixins来扩展。

原因高阶组件就是高阶函数，而React的组件本身就是纯粹的函数，所以高阶函数对React来说易如反掌。相反Vue.js使用HTML模板创建视图组件，这时模板无法有效的编译，因此Vue不采用HOC来实现。

6）构建工具

两者都有自己的构建工具

- React ==> Create React APP
- Vue ==> vue-cli

7）跨平台

- React ==> React Native
- Vue ==> Weex

6. 可以使用TypeScript写React应用吗？怎么操作？

（1）如果还未创建 Create React App 项目

- 直接创建一个具有 typescript 的 Create React App 项目：

```
npx create-react-app demo --typescript
```

（2）如果已经创建了 Create React App 项目，需要将 typescript 引入到已有项目中

- 通过命令将 typescript 引入项目：

```
npm install --save typescript @types/node @types/react @types/react-dom
@types/jest
```

- 将项目中任何 后缀名为 ‘.js’ 的 JavaScript 文件重命名为 TypeScript 文件即后缀名为 ‘.tsx’（例如 src/index.js 重命名为 src/index.tsx ）

7. React 设计思路，它的理念是什么？

（1）编写简单直观的代码

React最大的价值不是高性能的虚拟DOM、封装的事件机制、服务器端渲染，而是声明式的直观的编码方式。react文档第一条就是声明式，React 使创建交互式 UI 变得轻而易举。为应用的每一个状态设计简洁的视图，当数据改变时 React 能有效地更新并正确地渲染组件。 以声明式编写 UI，可以让代码更加可靠，且方便调试。

（2）简化可复用的组件

React框架里面使用了简化的组件模型，但更彻底地使用了组件化的概念。React将整个UI上的每一个功能模块定义成组件，然后将小的组件通过组合或者嵌套的方式构成更大的组件。React的组件具有如下的特性:

- 可组合：简单组件可以组合为复杂的组件

- 可重用：每个组件都是独立的，可以被多个组件使用
- 可维护：和组件相关的逻辑和UI都封装在了组件的内部，方便维护
- 可测试：因为组件的独立性，测试组件就变得方便很多。

(3) Virtual DOM

真实页面对应一个 DOM 树。在传统页面的开发模式中，每次需要更新页面时，都要手动操作 DOM 来进行更新。 DOM 操作非常昂贵。在前端开发中，性能消耗最大的就是 DOM 操作，而且这部分代码会让整体项目的代码变得难 以维护。React 把真实 DOM 树转换成 JavaScript 对象树，也就是 Virtual DOM，每次数据更新后，重新计算 Virtual DOM，并和上一次生成的 Virtual DOM 做对比，对发生变化的部分做批量更新。React 也提供了直观的 shouldComponentUpdate 生命周期回调，来减少数据变化后不必要的 Virtual DOM 对比过程，以保证性能。

（4）函数式编程

React 把过去不断重复构建 UI 的过程抽象成了组件，且在给定参数的情况下约定渲染对应的 UI 界面。React 能充分利用很多函数式方法去减少冗余代码。此外，由于它本身就是简单函数，所以易于测试。

（5）一次学习，随处编写

无论现在正在使用什么技术栈，都可以随时引入 React来开发新特性，而不需要重写现有代码。

React 还可以使用 Node 进行服务器渲染，或使用 React Native 开发原生移动应用。因为 React 组件可以映射为对应的原生控件。在输出的时候，是输出 Web DOM，还是 Android 控件，还是 iOS 控件，就由平台本身决定了。所以，react很方便和其他平台集成

8. 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：

```
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：

```
//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：

```
//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。

9. React的状态提升是什么？使用场景有哪些？

React的状态提升就是用户对子组件操作，子组件不改变自己的状态，通过自己的props把这个操作改变的数据传递给父组件，改变父组件的状态，从而改变受父组件控制的所有子组件的状态，这也是React单项数据流的特性决定的。官方的原话是：共享 state(状态) 是通过将其移动到需要它的组件的最接近的共同祖先组件来实现的。 这被称为“状态提升(Lifting State Up)”。

概括来说就是**将多个组件需要共享的状态提升到它们最近的父组件上，在父组件上改变这个状态然后通过props分发给子组件。**

一个简单的例子，父组件中有两个input子组件，如果想在第一个输入框输入数据，来改变第二个输入框的值，这就需要用到状态提升。

```
class Father extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            Value1: '',
            Value2: ''
        }
    }
    value1Change(aa) {
        this.setState({
            Value1: aa
        })
    }
    value2Change(bb) {
        this.setState({
            Value2: bb
        })
    }
    render() {
        return (
            <div style={{ padding: "100px" }}>
                <Child1 value1={this.state.Value1} onvalue1Change=
{this.value1Change.bind(this)} />
                <br />
                <Child2 value2={this.state.Value1} />
            </div>
        )
    }
}
class Child1 extends React.Component {
    constructor(props) {
        super(props)
    }
    changeValue(e) {
        this.props.onvalue1Change(e.target.value)
    }
    render() {
        return (
            <input value={this.props.Value1} onChange=
{this.changeValue.bind(this)} />
        )
    }
}
class Child2 extends React.Component {
    constructor(props) {
        super(props)
```

```
    }
    render() {
        return (
            <input value={this.props.value2} />
        )
    }
}

ReactDOM.render(
    <Father />,
    document.getElementById('root')
)
```

10. React中constructor和getInitialState的区别?

两者都是用来初始化state的。前者是ES6中的语法，后者是ES5中的语法，新版本的React中已经废弃了该方法。

getInitialState是ES5中的方法，如果使用createClass方法创建一个Component组件，可以自动调用它的getInitialState方法来获取初始化的State对象，

```
var APP = React.creatClass ({
  getInitialState() {
    return {
        userName: 'hi',
        userId: 0
     };
  }
})
```

React在ES6的实现中去掉了getInitialState这个hook函数，规定state在constructor中实现，如下：

```
Class App extends React.Component{
    constructor(props){
      super(props);
      this.state={};
    }
  }
```

11. React的严格模式如何使用，有什么用处?

`StrictMode` 是一个用来突出显示应用程序中潜在问题的工具。与 `Fragment` 一样，`StrictMode` 不会渲染任何可见的 UI。它为其后代元素触发额外的检查和警告。

可以为应用程序的任何部分启用严格模式。例如：

```
import React from 'react';
function ExampleApplication() {
  return (
    <div>
      <Header />
      <React.StrictMode>
        <div>
          <ComponentOne />
          <ComponentTwo />
        </div>
      </React.StrictMode>
      <Footer />
    </div>
  );
}
```

在上述的示例中，*不*会对 `Header` 和 `Footer` 组件运行严格模式检查。但是，`ComponentOne` 和 `ComponentTwo` 以及它们的所有后代元素都将进行检查。

`StrictMode` 目前有助于：

- 识别不安全的生命周期
- 关于使用过时字符串 ref API 的警告
- 关于使用废弃的 findDOMNode 方法的警告
- 检测意外的副作用
- 检测过时的 context API

12. 在React中遍历的方法有哪些?

（1）遍历数组：map && forEach

```
import React from 'react';

class App extends React.Component {
  render() {
    let arr = ['a', 'b', 'c', 'd'];
    return (
      <ul>
        {
          arr.map((item, index) => {
            return <li key={index}>{item}</li>
          })
        }
      </ul>
    )
  }
}

class App extends React.Component {
  render() {
```

```
    let arr = ['a', 'b', 'c', 'd'];
    return (
      <ul>
        {
          arr.forEach((item, index) => {
            return <li key={index}>{item}</li>
          })
        }
      </ul>
    )
  }
}
```

(2) 遍历对象：map && for in

```
class App extends React.Component {
  render() {
    let obj = {
      a: 1,
      b: 2,
      c: 3
    }
    return (
      <ul>
        {
          (() => {
            let domArr = [];
            for(const key in obj) {
              if(obj.hasOwnProperty(key)) {
                const value = obj[key]
                domArr.push(<li key={key}>{value}</li>)
              }
            }
            return domArr;
          })()
        }
      </ul>
    )
  }
}

// Object.entries() 把对象转换成数组
class App extends React.Component {
  render() {
    let obj = {
      a: 1,
      b: 2,
      c: 3
    }
    return (
      <ul>
        {
```

```
          Object.entries(obj).map(([key, value], index) => {   // item是一个数组，
把item解构，写法是[key, value]
            return <li key={key}>{value}</li>
          })
        }
      </ul>
    )
  }
}
```

13. 在React中页面重新加载时怎样保留数据?

这个问题就设计到了**数据持久化，**主要的实现方式有以下几种：

- **Redux：**将页面的数据存储在redux中，在重新加载页面时，获取Redux中的数据；
- **data.js：**使用webpack构建的项目，可以建一个文件，data.js，将数据保存data.js中，跳转页面后获取；
- **sessionStorge：**在进入选择地址页面之前，componentWillUnMount的时候，将数据存储到sessionStorage中，每次进入页面判断sessionStorage中有没有存储的那个值，有，则读取渲染数据；没有，则说明数据是初始化的状态。返回或进入除了选择地址以外的页面，清掉存储的sessionStorage，保证下次进入是初始化的数据
- **history API：**History API 的 `pushState` 函数可以给历史记录关联一个任意的可序列化 `state`，所以可以在路由 `push` 的时候将当前页面的一些信息存到 `state` 中，下次返回到这个页面的时候就能从 `state` 里面取出离开前的数据重新渲染。react-router 直接可以支持。这个方法适合一些需要临时存储的场景。

14. 同时引用这三个库react.js、react-dom.js和babel.js它们都有什么作用?

- react：包含react所必须的核心代码
- react-dom：react渲染在不同平台所需要的核心代码
- babel：将jsx转换成React代码的工具

15. React必须使用JSX吗?

React 并不强制要求使用 JSX。当不想在构建环境中配置有关 JSX 编译时，不在 React 中使用 JSX 会更加方便。

每个 JSX 元素只是调用 `React.createElement(component, props, ...children)` 的语法糖。因此，使用 JSX 可以完成的任何事情都可以通过纯 JavaScript 完成。

例如，用 JSX 编写的代码：

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

```
  document.getElementById('root')
);
```

可以编写为不使用 JSX 的代码：

```
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')
);
```

16. 为什么使用jsx的组件中没有看到使用react却需要引入react?

本质上来说JSX是`React.createElement(component, props, ...children)`方法的语法糖。在React 17之前，如果使用了JSX，其实就是在使用React， `babel` 会把组件转换为 `CreateElement` 形式。在React 17之后，就不再需要引入，因为 `babel` 已经可以帮我们自动引入react。

17. 在React中怎么使用async/await?

async/await是ES7标准中的新特性。如果是使用React官方的脚手架创建的项目，就可以直接使用。如果是在自己搭建的webpack配置的项目中使用，可能会遇到 **regeneratorRuntime is not defined** 的异常错误。那么我们就需要引入babel，并在babel中配置使用async/await。可以利用babel的 transform-async-to-module-method 插件来转换其成为浏览器支持的语法，虽然没有性能的提升，但对于代码编写体验要更好。

18. 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。

19. 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

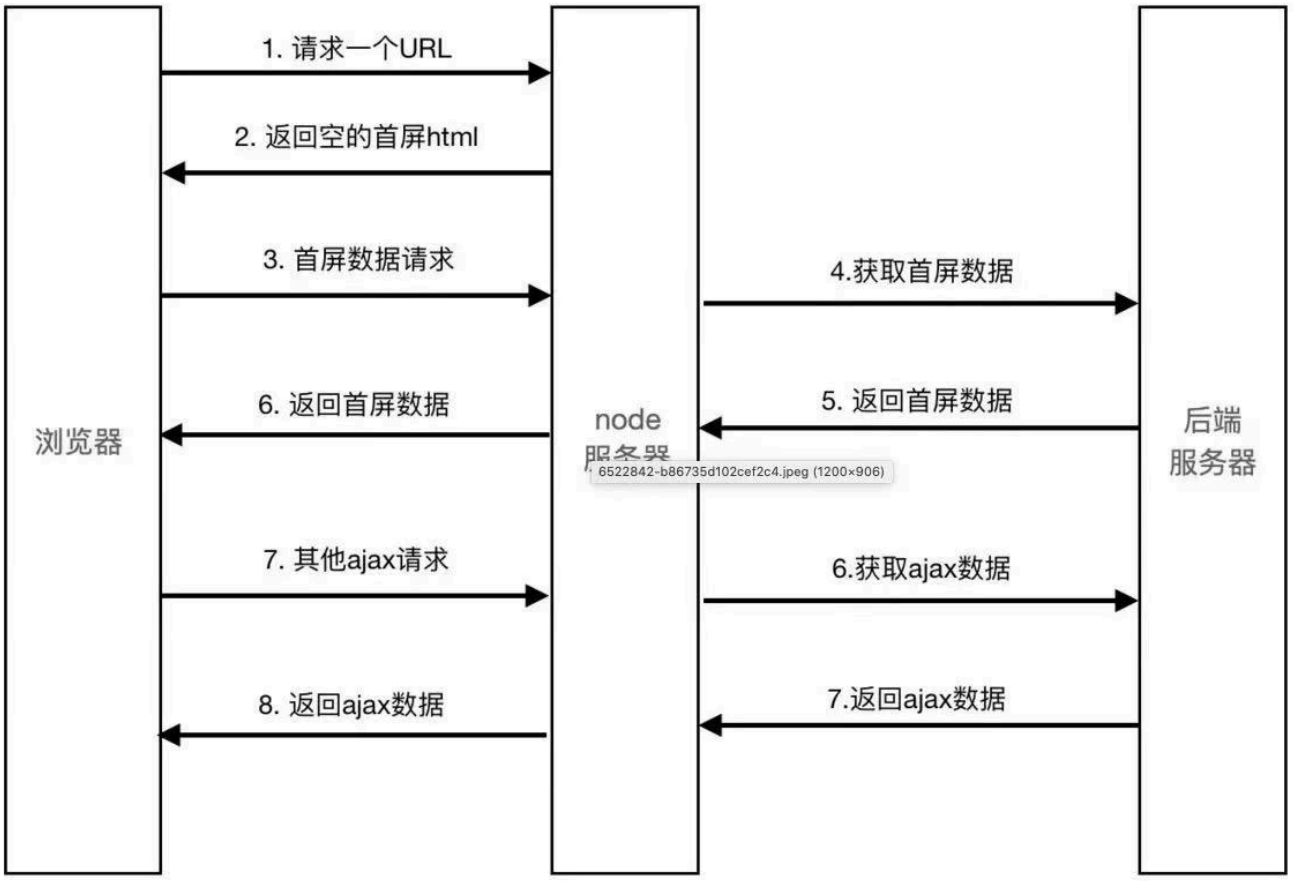
除了对webpack、MVVM框架要熟悉，还需要掌握node、 Koa2等相关技术。相对于客户端渲染，项目构建、部署过程更加复杂。

时间耗时比较：

1）数据请求

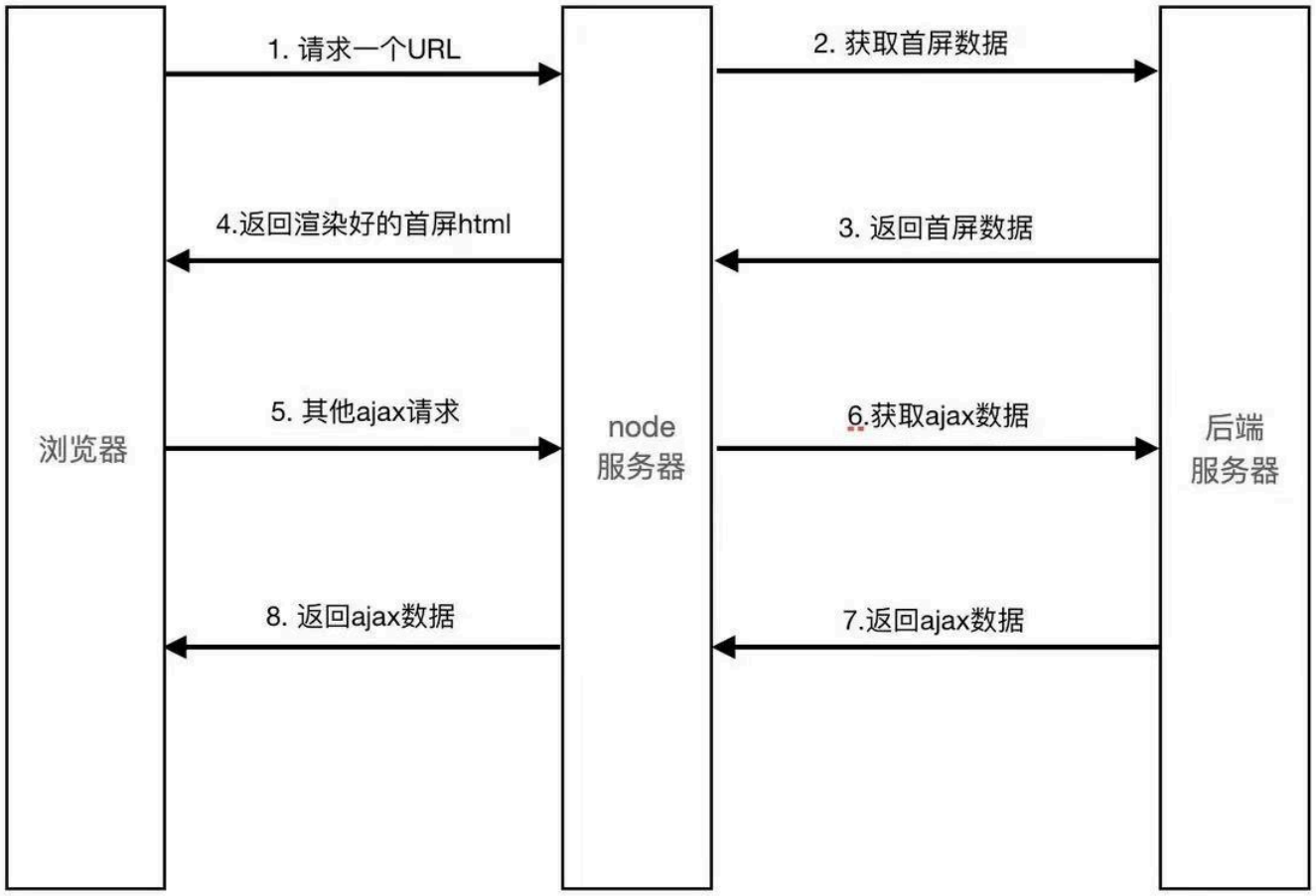
由服务端请求首屏数据，而不是客户端请求首屏数据，这是"快"的一个主要原因。服务端在内网进行请求，数据响应速度快。客户端在不同网络环境进行数据请求，且外网http请求开销大，导致时间差

- 客户端数据请求



客户端渲染路线

- 服务端数据请求

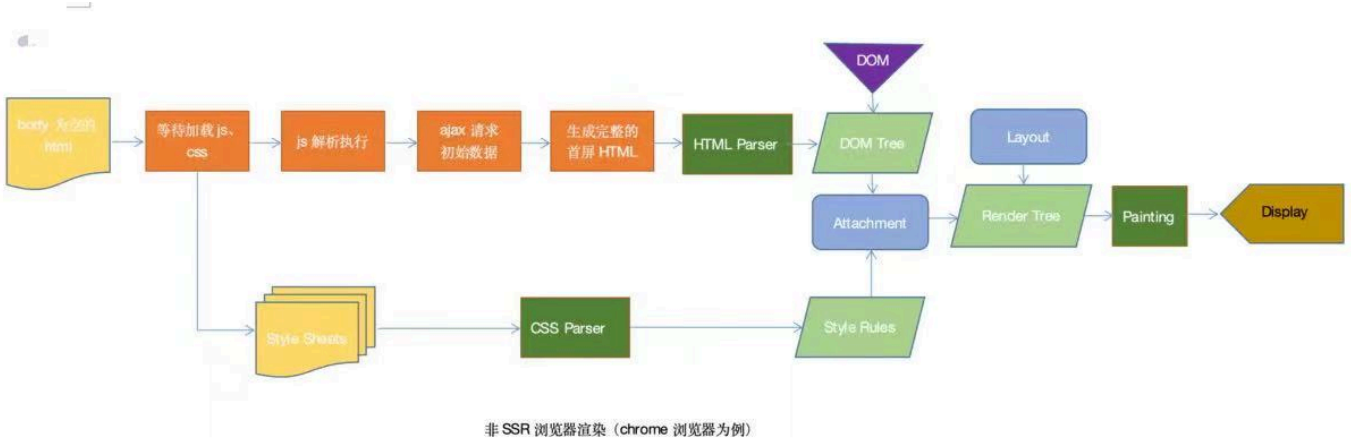


服务端渲染路线

2）html渲染

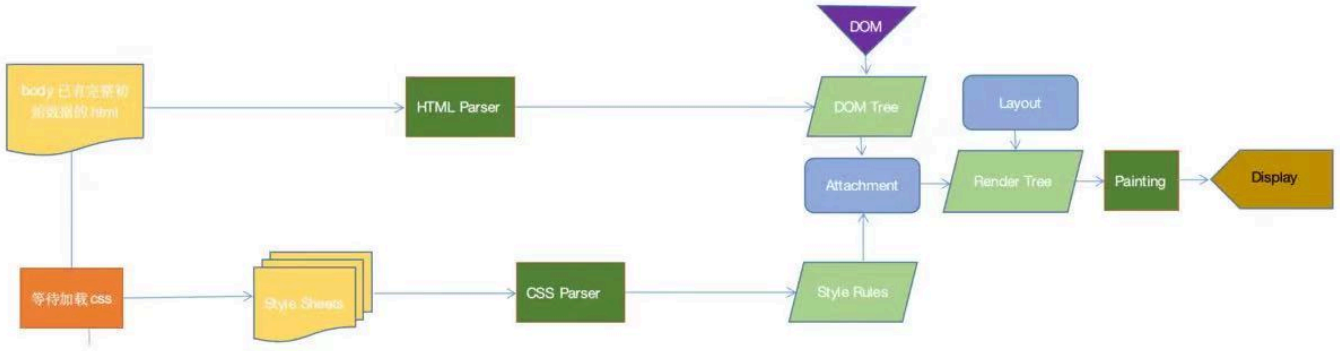
服务端渲染是先向后端服务器请求数据，然后生成完整首屏 html返回给浏览器；而客户端渲染是等js代码下载、加载、解析完成后再请求数据渲染，等待的过程页面是什么都没有的，就是用户看到的白屏。就是服务端渲染不需要等待js代码下载完成并请求数据，就可以返回一个已有完整数据的首屏页面。

- 非ssr html渲染



非 SSR 浏览器渲染（chrome 浏览器为例）

- ssr html渲染



20. 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：

```
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：

```
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。

21. HOC相比 mixins 有什么优点?

HOC 和 Vue 中的 mixins 作用是一致的，并且在早期 React 也是使用 mixins 的方式。但是在使用 class 的方式创建组件以后，mixins 的方式就不能使用了，并且其实 mixins 也是存在一些问题的，比如：

- 隐含了一些依赖，比如我在组件中写了某个 `state` 并且在 `mixin` 中使用了，就这存在了一个依赖关系。万一下次别人要移除它，就得去 `mixin` 中查找依赖
- 多个 `mixin` 中可能存在相同命名的函数，同时代码组件中也不能出现相同命名的函数，否则就是重写了，其实我一直觉得命名真的是一件麻烦事。。
- 雪球效应，虽然我一个组件还是使用着同一个 `mixin`，但是一个 `mixin` 会被多个组件使用，可能会存在需求使得 `mixin` 修改原本的函数或者新增更多的函数，这样可能就会产生一个维护成本

HOC 解决了这些问题，并且它们达成的效果也是一致的，同时也更加的政治正确（毕竟更加函数式了）。

22. React 中的高阶组件运用了什么设计模式?

使用了装饰模式，高阶组件的运用：

```
function withWindowWidth(BaseComponent) {
  class DerivedClass extends React.Component {
    state = {
      windowWidth: window.innerWidth,
    }
    onResize = () => {
      this.setState({
        windowWidth: window.innerWidth,
      })
    }
    componentDidMount() {
      window.addEventListener('resize', this.onResize)
    }
    componentWillUnmount() {
      window.removeEventListener('resize', this.onResize);
    }
    render() {
      return <BaseComponent {...this.props} {...this.state}/>
```

```
    }
  }
  return DerivedClass;
}
const MyComponent = (props) => {
  return <div>Window width is: {props.windowWidth}</div>
};
export default withWindowWidth(MyComponent);
```

装饰模式的特点是不需要改变 被装饰对象 本身，而只是在外面套一个外壳接口。JavaScript 目前已经有了原生装饰器的提案，其用法如下：

```
@testable
   class MyTestableClass {
}
```