常见react面试题

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:

```javascript 复制代码
export default React.createClass({  displayName: 'TodoApp',  // ...})
```

React推荐的方法:

```javascript 复制代码
export default class TodoApp extends React.Component {  // ...}
```

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 :

```复制代码
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：

```
javascript 复制代码
```

  - 维护应用状态并支持访问状态(`getState()`);
  - 支持监听action的分发，更新状态(`dispatch(action)`);
  - 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

shouldComponentUpdate有什么用？为什么它很重要?

组件状态数据或者属性数据发生更新的时候，组件会进入存在期，视图会渲染更新。在生命周期方法 should ComponentUpdate中，允许选择退出某些组件（和它们的子组件）的和解过程。 和解的最终目标是根据新的状态，以最有效的方式更新用户界面。如果我们知道用户界面的某一部分不会改变，那么没有理由让 React弄清楚它是否应该更新渲染。通过在shouldComponentUpdate方法中返回 false, React将让当前组件及其所有子组件保持与当前组件状态相同。

对React SSR的理解

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

SSR的优势：

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

1）更利于SEO

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

2）更利于首屏渲染

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

SSR的局限：

1）服务端压力较大

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

2）开发条件受限

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

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

时间耗时比较：

1）数据请求

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

- 客户端数据请求
- 服务端数据请求

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

- 非ssr html渲染
- ssr html渲染

React 的工作原理

React 会创建一个虚拟 DOM(virtual DOM)。当一个组件中的状态改变时，React 首先会通过 "diffing" 算法来标记虚拟 DOM 中的改变，第二步是调节(reconciliation)，会用 diff 的结果来更新 DOM。

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

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

2.使用this.props.history获取历史对象

```javascript 复制代码
let history = this.props.history;
```

参考 前端进阶面试题详细解答

React的虚拟DOM和Diff算法的内部实现

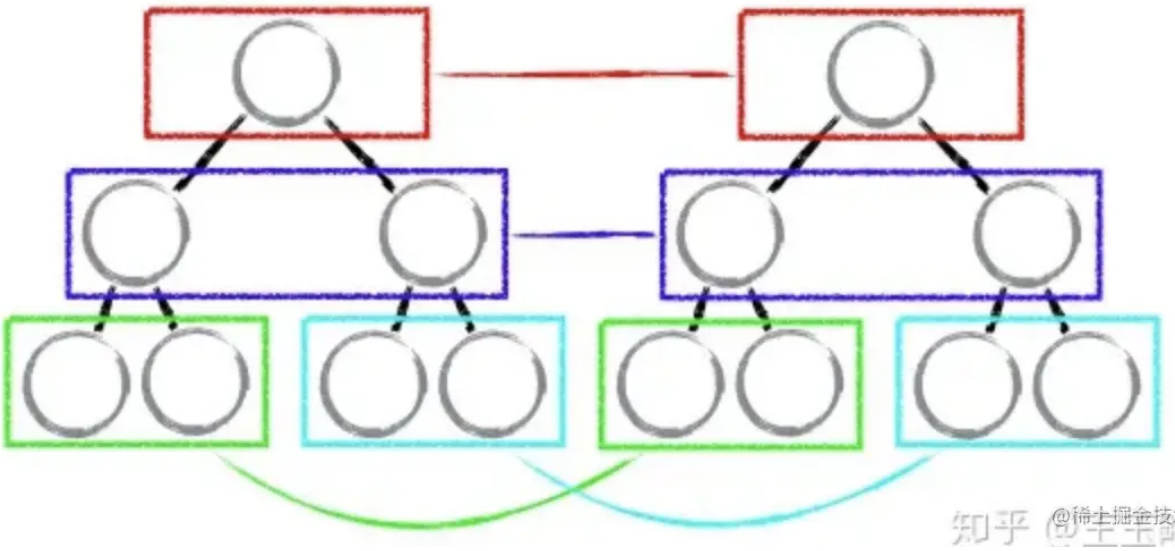
> 传统 diff 算法的时间复杂度是 O(n^3)，这在前端 render 中是不可接受的。为了降低时间复杂度，react 的 diff 算法做了一些妥协，放弃了最优解，最终将时间复杂度降低到了 O(n)。

那么 react diff 算法做了哪些妥协呢？，参考如下：

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

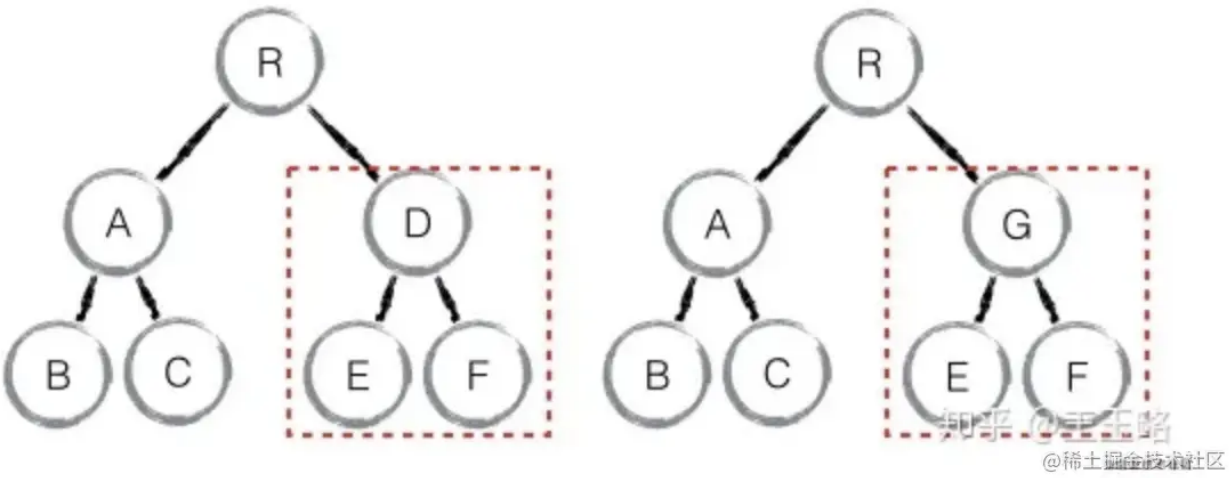
如下图，react 只会对相同颜色方框内的 DOM 节点进行比较，即同一个父节点下的所有子节点。当发现节点不存在时，则该节点及其子节点会被完全删除掉，不会用于进一步的比较。

这样只需要对树进行一次遍历，便能完成整个 DOM 树的比较。



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

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



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

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

如何使用4.0版本的 React Router?

React Router 4.0版本中对 hashHistory做了迁移，执行包安装命令 npm install react-router-dom后，按照如下代码进行使用即可。

```javascript 复制代码
import { HashRouter, Route, Redirect, Switch } from " react-router-dom";
class App extends Component {
  render() {
    return (
      <div>
        <Switch>
          <Route path="/list" componen t={List}></Route>
          <Route path="/detail/: id" component={Detail}>
            {" "}
          </Route>
          <Redirect from="/ " to="/list">
            {" "}
          </Redirect>
        </Switch>
      </div>
    );
  }
}
const routes = (
  <HashRouter>
    <App> </App>
  </HashRouter>
);
render(routes, ickt);
```

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：

```javascript 复制代码
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

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。

类组件和函数组件有何不同?

解答

在 React 16.8版本（引入钩子）之前，使用基于类的组件来创建需要维护内部状态或利用生命周期方法的组件（即 `componentDidMount` 和 `shouldComponentUpdate` ）。基于类的组件是 ES6 类，它扩展了 React 的 Component 类，并且至少实现了 `render()` 方法。

类组件：

```javascript 复制代码
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

函数组件是无状态的（同样，小于 React 16.8版本），并返回要呈现的输出。它们渲染 UI 的首选只依赖于属性，因为它们比基于类的组件更简单、更具性能。

函数组件：

```javascript 复制代码
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

注意：在 React 16.8版本中引入钩子意味着这些区别不再适用（请参阅14和15题）。

setState 是同步的还是异步的

有时表现出同步，有时表现出异步

1. setState 只有在 React 自身的合成事件和钩子函数中是异步的，在原生事件和 setTimeout 中都是同步的
2. setState 的异步并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数中没法立马拿到更新后的值，形成了所谓的异步。当然可以通过 setState 的第二个参数中的 callback 拿到更新后的结果
3. setState 的批量更新优化也是建立在异步（合成事件、钩子函数）之上的，在原生事件和 setTimeout 中不会批量更新，在异步中如果对同一个值进行多次 setState，setState 的批量更新策略会对其进行覆盖，去最后一次的执行，如果是同时 setState 多个不同的值，在更新时会对其进行合并批量更新

- 合成事件中是异步
- 钩子函数中的是异步
- 原生事件中是同步
- setTimeout中是同步

如何有条件地向 React 组件添加属性?

对于某些属性，React 非常聪明，如果传递给它的值是虚值，可以省略该属性。例如：

```javascript
var InputComponent = React.createClass({
  render: function () {
    var required = true;
    var disabled = false;
    return <input type="text" disabled={disabled} required={required} />;
  },
});
```

渲染结果：

```javascript
<input type="text" required>
```

另一种可能的方法是：

```javascript
var condition = true;
var component = <div value="foo" {...(condition && { disabled: true })} />;
```

Hooks可以取代 `render props` 和高阶组件吗?

通常， `render props` 和高阶组件仅渲染一个子组件。React团队认为，Hooks 是服务此用例的更简单方法。 这两种模式仍然有一席之地(例如，一个虚拟的 `scroller` 组件可能有一个 `renderItem prop` ，或者一个可视化的容器组件可能有它自己的 DOM 结构)。但在大多数情况下，Hooks 就足够了，可以帮助减少树中的嵌套。

react router

```javascript
import React from 'react'
import { render } from 'react-dom'
```

```
import { browserHistory, Router, Route, IndexRoute } from 'react-router'

import App from '../components/App'
import Home from '../components/Home'
import About from '../components/About'
import Features from '../components/Features'

render(
  <Router history={browserHistory}>  // history 路由
    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      <Route path='about' component={About} />
      <Route path='features' component={Features} />
    </Route>
  </Router>,
  document.getElementById('app')
)
render(
  <Router history={browserHistory} routes={routes} />,
  document.getElementById('app')
)
```

React Router 提供一个routerWillLeave生命周期钩子，这使得 React组件可以拦截正在发生的跳转，或在离开route前提示用户。routerWillLeave返回值有以下两种：

`return false` 取消此次跳转
`return` 返回提示信息，在离开 route 前提示用户进行确认。

对于store的理解

Store 就是把它们联系到一起的对象。Store 有以下职责：

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

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。

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：

```javascript 复制代码
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：

javascript 复制代码

```
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。

javascript 复制代码

```
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>
  );
}
```

简述flux 思想

> `Flux` 的最大特点，就是数据的"单向流动"。

- 用户访问 `View`
- `View` 发出用户的 `Action`
- `Dispatcher` 收到 `Action`，要求 `Store` 进行相应的更新
- `Store` 更新后，发出一个 `"change"` 事件

- View 收到 "change" 事件后，更新页面

react 实现一个全局的 dialog

javascript 复制代码

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

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

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

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

子类:

javascript 复制代码

```
//子类jsx
import React, { Component } from 'react';
class Child extends Component {
    constructor(props){
```

```
        super(props);
        this.state = {date: new Date()};
    }
    showValue=()=>{
      this.props.showValue && this.props.showValue()
    }
    render() {
      return (
        <div className="Child">
          <div className="content">
             Child           <button onClick={this.showValue}>调用父的方法</button>
          </div>
        </div>
      );
    }
  }
  export default Child;
```

CSS：

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