

UmiJS基础介绍

主讲人: 邵飞波 (322827)





基于ReactJSX语法, Babel, htm库, 挂载

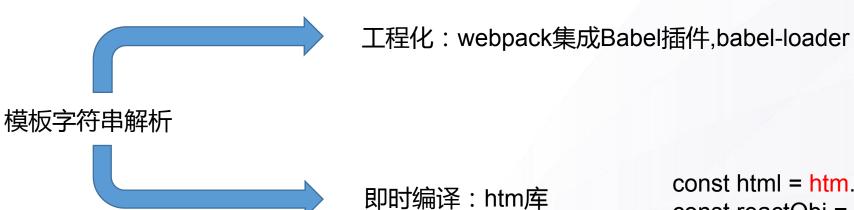
02 Umi _{目录结构}, 路由, dva TITLE

React









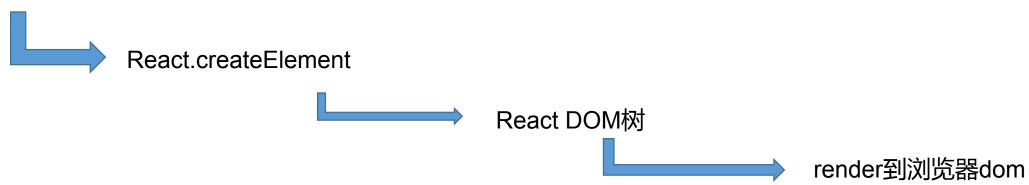
const html = htm.bind(React.createElement);
const reactObj = html`<div></div>`

挂载:

```
<div id="root"></div>
<script>
    ReactDOM.render(ReactObj, document.getElementById('root'));
</script>
```

流程:

字符串解析



响应式:

通过setState异步收集所有要更新的属性diff之前的reactDom后来触发嵌套的CC的render或执行FC

CC钩子函数:

- 1.static getDerivedStateFromProps // render或return时收集当前props和state;
- 2.shouldComponentUpdate// 返回bool去决定是否执行render,可优化
- 3.componentDidUpdate // 更新后保存之前的props和state

性能相关(减少render或return):

CC:

1.PureComponent (慎用)

特点:自带shouldComponentUpdate钩子,浅比较决定render

(例:state:{x:{y:1}} setState({x:{y:1}}仍会触发render)

(例:const arr; arr.push('xx');setState({x:arr}不会触发render)

FC:

1.React.memo

特点:根据传入Props,浅比较决定是否执行函数

2.React.useMemo

特点:仅能FC内使用;判断条件内使用也会导致错乱

useMemo(()=>CC||FC,[...args])

本质:浅比较args每一项来决定是否更新闭包单例,每次返回此单例

TITLE

Umi







```
// 默认的 build 输出目录
— dist/
                          // mock 文件所在目录,基于 express
 - mock/
— config/
                          // umi 配置,同 .umirc.js,二选一
  — config.js
L src/
                          // 源码目录,可选
   layouts/index.js
                          // 全局布局
                          // 页面目录,里面的文件即路由
   — pages/
                          // dev 临时目录,需添加到 .gitignore
      - .umi/
      _____.umi-production/
                          // build 临时目录,会自动删除
      — document.ejs
                          // HTML 模板
                    // 404 页面
      — 404.js
                          // 页面 1,仟意命名,导出 react 组件
      — page1.js
      — page1.test.js
                          // 用例文件, umi test 会匹配所有 .test.js 和 .e2e.js 结尾的文件
      └─ page2.js
                          // 页面 2,任意命名
                          // 约定的全局样式文件,自动引入,也可以用 global.less
    - global.css
    — global.js
                          // 可以在这里加入 polyfill
                          // 运行时配置文件
   — app.js
  .umirc.js
                          // umi 配置,同 config/config.js,二选一
                          // 环境变量
  package.json
```

基础布局layouts可通过props.location.pathname条件渲染,比如管理后台布局,来监听/admin // const withRouter = C => props => <C location={window.location} {...props} /> 目前组件应是都默认都有包一层不用写





跳转:

js: Router.push("/whr");

css: <Link to="/whr"/>

种类:

```
约定路由:
```

ReactDOM.render(<BasicLayout children={import('@/pages/index.js')} />, root);

/warehouse

ReactDOM.render(<BasicLayout children={import('@/pages/warehouse/index.js')} />, root);

/warehouse/creat

ReactDOM.render(<BasicLayout children={import('@/pages/warehouse/creat.js')} />, root);

配置路由:

配置式路由

如果你倾向于使用配置式的路由,可以配置 .umirc.(ts|js) 或者 config/config.(ts|js) 配置文件 中的 routes 属性,此配置项存在时则不会对 src/pages 目录做约定式的解析。

比如:

注意:

1. component 是相对于 src/pages 目录的

Routes可用作权限校验,比如FC: const Auth = props => isLogin? <View {...props}/>: <NotAuthorized/>

dva



数据层解决方案

种类: src/models全局model, pages目录下的为分级model

查找方式:类似node_modules逐级往上

分类:层次清晰;

```
<script type="text/babel">
   const { Component } = React;
   class Son1 extends Component {
       render() {
           console.log("Son1 cry");
           return <div>Father > Son1: newA</div>;
   class Son2 extends Component {
       state = { x: "newA" };
       update() {
           this.setState({ x: "newB" });
       render() {
           console.log("Son2 cry");
           return (
                   <button onClick={() => {this.update();}}>
                       Son2 beat self
                   Father > Son2: {this.props.x}
                   Son2 > Son2: {this.state.x}
```

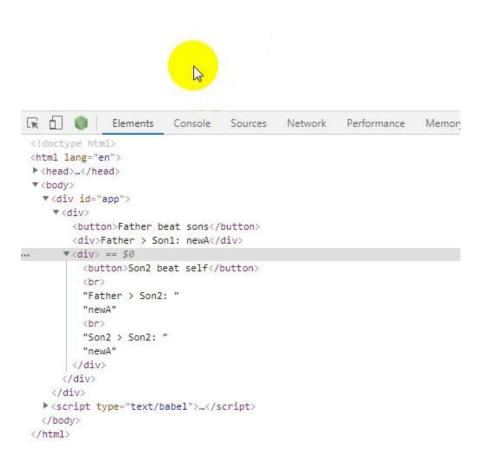
```
class Father extends Component {
       state = { x: "newA" };
       update() {
            this.setState({ x: "newB" });
       render() {
            return (
                <div>
                    <button onClick={() => {this.update();}}>
                        Father beat sons
                    </button>
                    <Son1 {...this.state} />
                   <Son2 {...this.state} />
                </div>
            );
   ReactDOM.render(<Father />, app);
</script>
```

Father beat sons

Father > Son1: newA

Son2 beat self

Father > Son2: newA Son2 > Son2: newA



结论:

不一定重复渲染,但可能重复render生成新ReactDom

dva



```
import React from "react";
import dva, { connect } from "dva";
const app = dva();
const Main = props => {
    return (
        <div>
           <Test />
        </div>
    );
app.model({
   namespace: "test",
    state: {
        test: "test"
    },
   reducers: {}
});
const Test = connect()(props => {
    console.log(props.dispatch === app[" store"].dispatch);
   return <div>test</div>;
3);
app.router(props => {
    console.log(props);
   return <Main {...props} />;
});
// ReactDOM.render(<Main />, document.getElementById("app"));
app.start("#app");
```

```
▼ {app: {...}, history: {...}} []
 ▶ app: {_models: Array(2), _store: {...}, _plugin: Plugin, use: f, model: f, ...}
 ▶ history: {length: 1, action: "POP", location: {...}, createHref: f, push: f, ...}
 ▶ proto : Object
▼ app:
 ▶ model: f ()
 ▶ replaceModel: f ()
 ▶ router: f router(router)
 ▶ start: f start(container)
 ▶ unmodel: f ()
 ▶ use: f ()
 ▶ getProvider: f ()
 ▶ getSaga: f ()
 ▶ history: {length: 1, action: "POP", location: {...}, createHref: f, push: f, ...}
 ▼_models: Array(2)
  ▶ 0: {namespace: "@@dva", state: 0, reducers: {...}}
   ▶ 1: {namespace: "test", state: {...}, reducers: {...}}
   length: 2
   ▶ proto : Array(0)
 ▶ plugin: Plugin { handleActions: null, hooks: {...}}
 ▶ router: props => {...}
 ▶_store: {dispatch: f, subscribe: f, getState: f, replaceReducer: f, runSaga: f, ...}
▼ history:
   action: "POP"
  ▶ block: f block(prompt)
  ▶ createHref: f createHref(location)
  ▶ go: f go(n)
  ▶ goBack: f goBack()
  ▶ goForward: f goForward()
  length: 1
  ▶ listen: f (callback)
  ▶ location: {pathname: "/", search: "", hash: "", state: undefined}
  ▶ push: f push(path, state)
  ▶ replace: f replace(path, state)
```

connect

const connect = (f: model.states => someStates) => (Com) => props => <Com {...props} {...someStates} />

model

```
app.model({
 namespace: 'count',
 state: {
   record: 0,
   current: 0,
 reducers: {
                                 触发connect此namespace的render或return
   add(state) {
     const newCurrent = state.current + 1;
     return { ...state,
      record: newCurrent > state.record ? newCurrent : state.record,
       current: newCurrent,
   minus(state) {
    return { ...state, current: state.current - 1};
                                  异步请求等,支持类似promise.all,race等异步次序
 effects: {
   *add(action, { call, put }) {
    yield call(delay, 1000);
    yield put({ type: 'minus' });
                                                触发reducer, 更新state
 subscriptions: {
   keyboardWatcher({ dispatch }) {
     key('\+up, ctrl+up', () => { dispatch({type:'add'}) });
                                  订阅模式eventListener, 可用来监听事件
```

