```
45   }
```

当对象包含复杂的数据结构时，对象深层的数据已改变却没有触发 `render`

注意：在 `react` 中，是不建议使用深层次结构的数据

24.2.3. React.memo

`React.memo` 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

JSX | 复制代码

```jsx
import { memo } from 'react';

function Button(props) {
  // Component code
}

export default memo(Button);
```

如果需要深层次比较，这时候可以给 `memo` 第二个参数传递比较函数

JSX | 复制代码

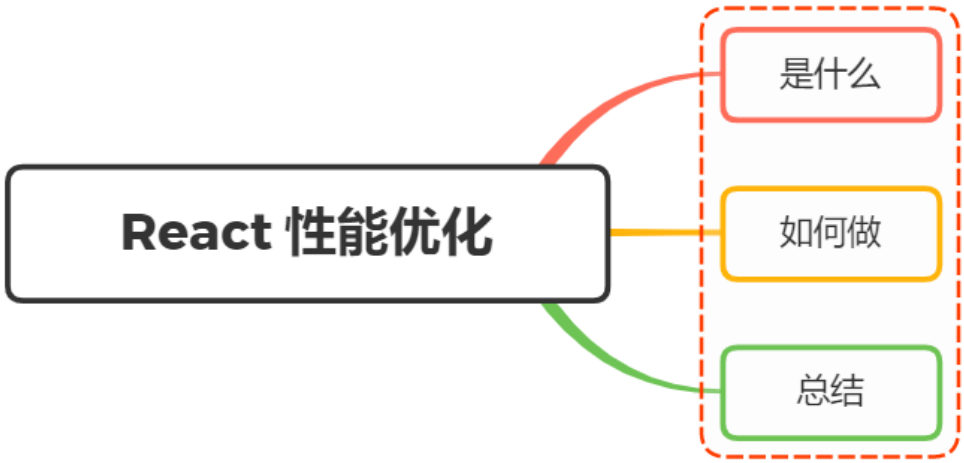
```jsx
function arePropsEqual(prevProps, nextProps) {
  // your code
  return prevProps === nextProps;
}

export default memo(Button, arePropsEqual);
```

24.3. 总结

在实际开发过程中，前端性能问题是一个必须考虑的问题，随着业务的复杂，遇到性能问题的概率也在增高

除此之外，建议将页面进行更小的颗粒化，如果一个过大，当状态发生修改的时候，就会导致整个大组件的渲染，而对组件进行拆分后，粒度变小了，也能够减少子组件不必要的渲染

25. 说说 React 性能优化的手段有哪些?

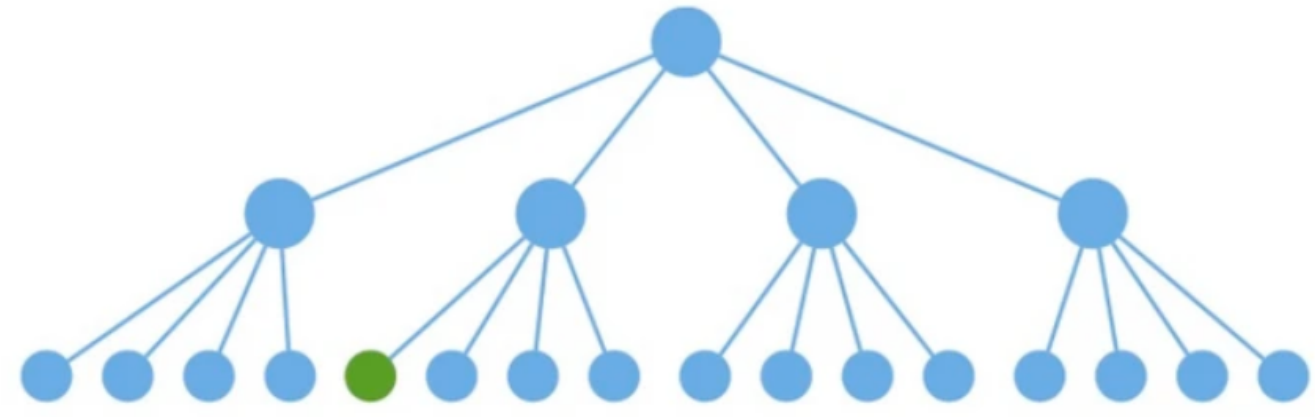


25.1. 是什么

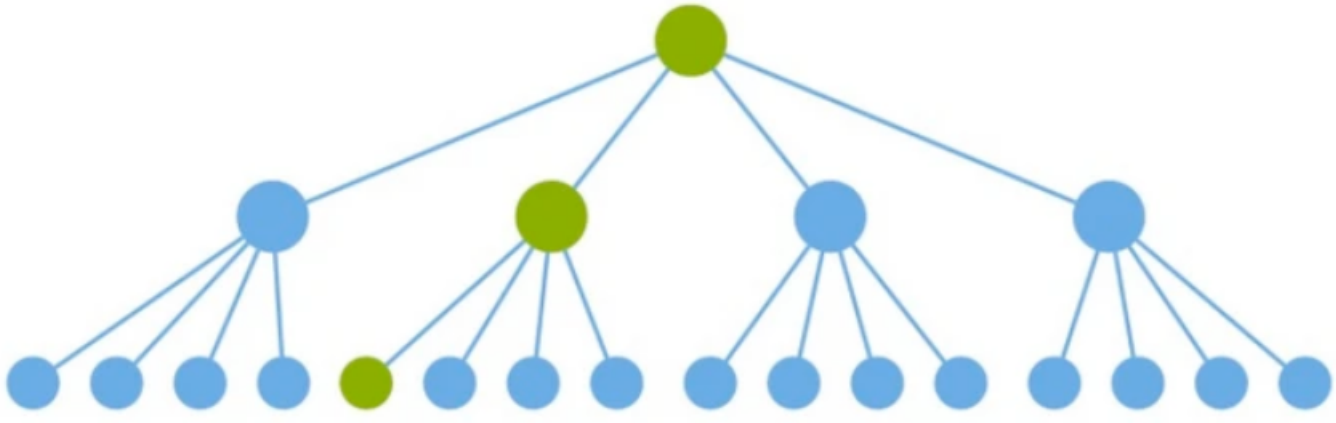
`React` 凭借 `virtual DOM` 和 `diff` 算法拥有高效的性能，但是某些情况下，性能明显可以进一步提高

在前面文章中，我们了解到类组件通过调用 `setState` 方法， 就会导致 `render` ，父组件一旦发生 `render` 渲染，子组件一定也会执行 `render` 渲染

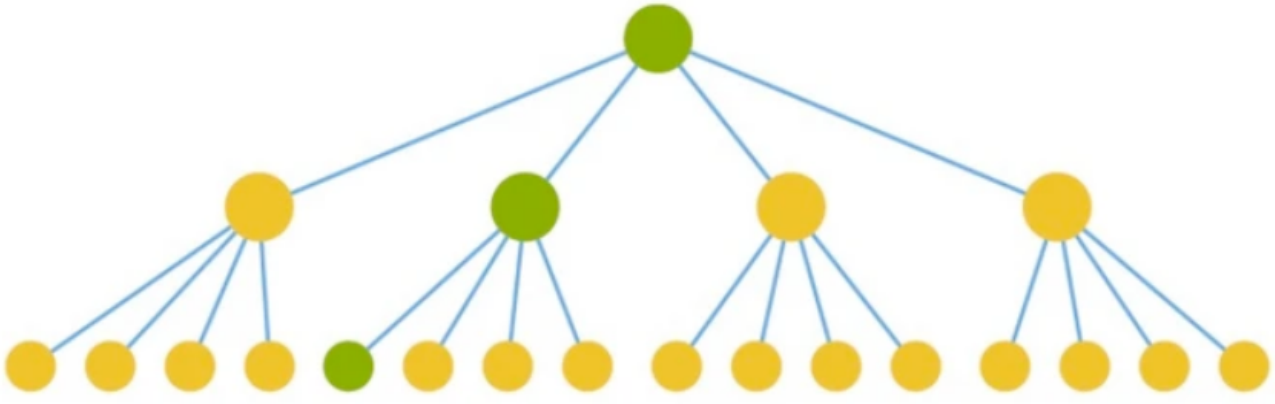
当我们想要更新一个子组件的时候，如下图绿色部分：



理想状态只调用该路径下的组件 `render` ：



但是 `react` 的默认做法是调用所有组件的 `render`，再对生成的虚拟 `DOM` 进行对比（黄色部分），如不变则不进行更新



从上图可见，黄色部分 `diff` 算法对比是明显的性能浪费的情况

25.2. 如何做

在React中如何避免不必要的render中，我们了解到如何避免不必要的 `render` 来应付上面的问题，主要手段是通过 `shouldComponentUpdate` 、 `PureComponent` 、 `React.memo` ，这三种形式这里就不再复述

除此之外， 常见性能优化常见的手段有如下：

- 避免使用内联函数
- 使用 React Fragments 避免额外标记
- 使用 Immutable
- 懒加载组件
- 事件绑定方式
- 服务端渲染

25.2.1. 避免使用内联函数

如果我们使用内联函数，则每次调用 `render` 函数时都会创建一个新的函数实例，如下：

JSX | 复制代码

```jsx
import React from "react";

export default class InlineFunctionComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>Welcome Guest</h1>
        <input type="button" onClick={(e) => { this.setState({inputValue: e.target.value}) }} value="Click For Inline Function" />
      </div>
    )
  }
}
```

我们应该在组件内部创建一个函数，并将事件绑定到该函数本身。这样每次调用 `render` 时就不会创建单独的函数实例，如下：

JSX | 复制代码

```jsx
import React from "react";

export default class InlineFunctionComponent extends React.Component {

  setNewStateData = (event) => {
    this.setState({
      inputValue: e.target.value
    })
  }

  render() {
    return (
      <div>
        <h1>Welcome Guest</h1>
        <input type="button" onClick={this.setNewStateData} value="Click For Inline Function" />
      </div>
    )
  }
}
```

25.2.2. 使用 React Fragments 避免额外标记

用户创建新组件时，每个组件应具有单个父标签。父级不能有两个标签，所以顶部要有一个公共标签，所以我们经常在组件顶部添加额外标签 `div`

这个额外标签除了充当父标签之外，并没有其他作用，这时候则可以使用 `fragement`

其不会向组件引入任何额外标记，但它可以作为父级标签的作用，如下所示：

JSX | 复制代码

```jsx
export default class NestedRoutingComponent extends React.Component {
    render() {
        return (
            <>
                <h1>This is the Header Component</h1>
                <h2>Welcome To Demo Page</h2>
            </>
        )
    }
}
```

25.2.3. 事件绑定方式

在事件绑定方式中，我们了解到四种事假绑定的方式

从性能方面考虑，在 `render` 方法中使用 `bind` 和 `render` 方法中使用箭头函数这两种形式在每次组件 `render` 的时候都会生成新的方法实例，性能欠缺

而 `constructor` 中 `bind` 事件与定义阶段使用箭头函数绑定这两种形式只会生成一个方法实例，性能方面会有所改善

25.2.4. 使用 Immutable

我们了解到使用 `Immutable` 可以给 `React` 应用带来性能的优化，主要体现在减少渲染的次数

在做 `react` 性能优化的时候，为了避免重复渲染，我们会在 `shouldComponentUpdate()` 中做对比，当返回 `true` 执行 `render` 方法

`Immutable` 通过 `is` 方法则可以完成对比，而无需像一样通过深度比较的方式比较

25.2.5. 懒加载组件

从工程方面考虑，`webpack`存在代码拆分能力，可以为应用创建多个包，并在运行时动态加载，减少初始包的大小

而在`react`中使用到了`Suspense`和`lazy`组件实现代码拆分功能，基本使用如下：

JSX 复制代码

```jsx
const johanComponent = React.lazy(() => import(/* webpackChunkName: "johanComponent" */ './myAwesome.component'));

export const johanAsyncComponent = props => (
  <React.Suspense fallback={<Spinner />}>
    <johanComponent {...props} />
  </React.Suspense>
);
```

25.2.6. 服务端渲染

采用服务端渲染端方式，可以使用户更快的看到渲染完成的页面

服务端渲染，需要起一个`node`服务，可以使用`express`、`koa`等，调用`react`的`renderToString`方法，将根组件渲染成字符串，再输出到响应中

例如：

JavaScript 复制代码

```javascript
import { renderToString } from "react-dom/server";
import MyPage from "./MyPage";
app.get("/", (req, res) => {
  res.write("<!DOCTYPE html><html><head><title>My Page</title></head><body>");
  res.write("<div id='content'>");
  res.write(renderToString(<MyPage/>));
  res.write("</div></body></html>");
  res.end();
});
```

客户端使用render方法来生成HTML

JSX 复制代码

```jsx
import ReactDOM from 'react-dom';
import MyPage from "./MyPage";
ReactDOM.render(<MyPage />, document.getElementById('app'));
```

25.2.7. 其他

除此之外，还存在的优化手段有组件拆分、合理使用 `hooks` 等性能优化手段...

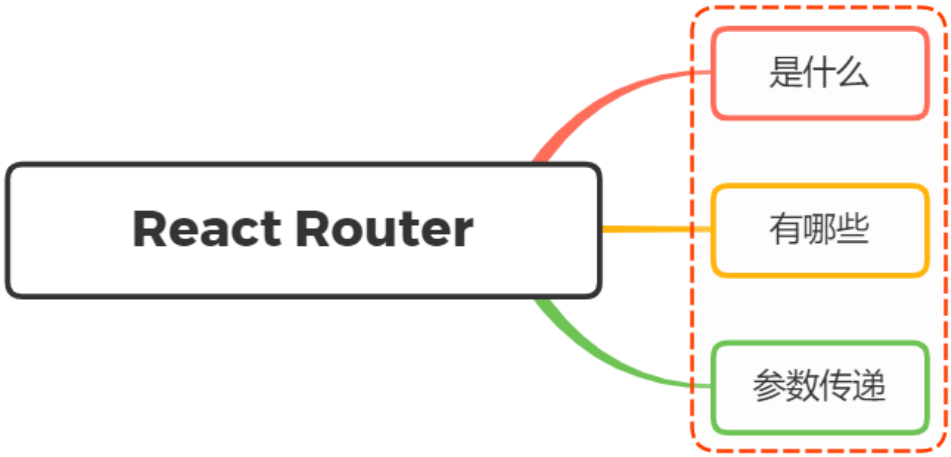
25.3. 总结

通过上面初步学习，我们了解到 `react` 常见的性能优化可以分成三个层面：

- 代码层面
- 工程层面
- 框架机制层面

通过这三个层面的优化结合，能够使基于 `react` 项目的性能更上一层楼

26. 说说你对React Router的理解？常用的Router组件有哪些？



26.1. 是什么

`react-router` 等前端路由的原理大致相同，可以实现无刷新的条件下切换显示不同的页面

路由的本质就是页面的 `URL` 发生改变时，页面的显示结果可以根据 `URL` 的变化而变化，但是页面不会刷新

因此，可以通过前端路由可以实现单页(SPA)应用

`react-router` 主要分成了几个不同的包：

- react-router: 实现了路由的核心功能
- react-router-dom： 基于 react-router，加入了在浏览器运行环境下的一些功能
- react-router-native：基于 react-router，加入了 react-native 运行环境下的一些功能
- react-router-config: 用于配置静态路由的工具库

26.2. 有哪些

这里主要讲述的是 `react-router-dom` 的常用 `API` ，主要是提供了一些组件：

- BrowserRouter、HashRouter
- Route
- Link、NavLink
- switch
- redirect

26.2.1. BrowserRouter、HashRouter

`Router` 中包含了对路径改变的监听，并且会将相应的路径传递给子组件

`BrowserRouter` 是 `history` 模式， `HashRouter` 模式

使用两者作为最顶层组件包裹其他组件

JSX | 复制代码

```jsx
import { BrowserRouter as Router } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <main>
        <nav>
          <ul>
            <li>
              < a href=" ">Home</ a>
            </li>
            <li>
              < a href="/about">About</ a>
            </li>
            <li>
              < a href="/contact">Contact</ a>
            </li>
          </ul>
        </nav>
      </main>
    </Router>
  );
}
```

26.2.2. Route

`Route` 用于路径的匹配，然后进行组件的渲染，对应的属性如下：

- path 属性：用于设置匹配到的路径
- component 属性：设置匹配到路径后，渲染的组件
- render 属性：设置匹配到路径后，渲染的内容
- exact 属性：开启精准匹配，只有精准匹配到完全一致的路径，才会渲染对应的组件

JSX | 复制代码

```jsx
import { BrowserRouter as Router, Route } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <main>
        <nav>
          <ul>
            <li>
              < a href="/">Home</ a>
            </li>
            <li>
              < a href="/about">About</ a>
            </li>
            <li>
              < a href="/contact">Contact</ a>
            </li>
          </ul>
        </nav>
        <Route path="/" render={() => <h1>Welcome!</h1>} />
      </main>
    </Router>
  );
}
```

26.2.3. Link、NavLink

通常路径的跳转是使用 `Link` 组件，最终会被渲染成 `a` 元素，其中属性 `to` 代替 `a` 标题的 `href` 属性

`NavLink` 是在 `Link` 基础之上增加了一些样式属性，例如组件被选中时，发生样式变化，则可以设置 `NavLink` 的一下属性：

- activeStyle：活跃时（匹配时）的样式
- activeClassName：活跃时添加的class

如下：

JavaScript | 复制代码

```jsx
<NavLink to="/" exact activeStyle={{color: "red"}}>首页</NavLink>
<NavLink to="/about" activeStyle={{color: "red"}}>关于</NavLink>
<NavLink to="/profile" activeStyle={{color: "red"}}>我的</NavLink>
```

如果需要实现 `js` 实现页面的跳转，那么可以通过下面的形式：

通过 `Route` 作为顶层组件包裹其他组件后,页面组件就可以接收到一些路由相关的东西，比如 `props.history`

JSX | 复制代码

```jsx
const Contact = ({ history }) => (
  <Fragment>
    <h1>Contact</h1>
    <button onClick={() => history.push("/")}>Go to home</button>
    <FakeText />
  </Fragment>
)
```

`props` 中接收到的 `history` 对象具有一些方便的方法，如 `goBack` ， `goForward` , `push`

26.2.4. redirect

用于路由的重定向，当这个组件出现时，就会执行跳转到对应的 `to` 路径中，如下例子：

JavaScript | 复制代码

```javascript
const About = ({
  match: {
    params: { name },
  },
}) => (
  // props.match.params.name
  <Fragment>
    {name !== "tom" ? <Redirect to="/" /> : null}
    <h1>About {name}</h1>
    <FakeText />
  </Fragment>
)
```

上述组件当接收到的路由参数 `name` 不等于 `tom` 的时候，将会自动重定向到首页

26.2.5. switch

`swich` 组件的作用适用于当匹配到第一个组件的时候，后面的组件就不应该继续匹配

如下例子：

JSX | 复制代码

```jsx
<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/about" component={About} />
  <Route path="/profile" component={Profile} />
  <Route path="/:userid" component={User} />
  <Route component={NoMatch} />
</Switch>
```

如果不使用 `switch` 组件进行包裹

除了一些路由相关的组件之外， `react-router` 还提供一些 `hooks` ，如下：

- useHistory
- useParams
- useLocation

26.2.6. useHistory

`useHistory` 可以让组件内部直接访问 `history` ，无须通过 `props` 获取

JavaScript | 复制代码

```javascript
import { useHistory } from "react-router-dom";

const Contact = () => {
  const history = useHistory();
  return (
    <Fragment>
      <h1>Contact</h1>
      <button onClick={() => history.push("/")}>Go to home</button>
    </Fragment>
  );
};
```

26.2.7. useParams

```jsx
const About = () => {
  const { name } = useParams();
  return (
    // props.match.params.name
    <Fragment>
      {name !== "John Doe" ? <Redirect to="/" /> : null}
      <h1>About {name}</h1>
      <Route component={Contact} />
    </Fragment>
  );
};
```

26.2.8. useLocation

`useLocation` 会返回当前 `URL` 的 `location` 对象

```jsx
import { useLocation } from "react-router-dom";

const Contact = () => {
  const { pathname } = useLocation();

  return (
    <Fragment>
      <h1>Contact</h1>
      <p>Current URL: {pathname}</p >
    </Fragment>
  );
};
```

26.3. 参数传递

这些路由传递参数主要分成了三种形式：

- 动态路由的方式
- search传递参数
- to传入对象

26.3.1. 动态路由

动态路由的概念指的是路由中的路径并不会固定

例如将 `path` 在 `Route` 匹配时写成 `/detail/:id`，那么 `/detail/abc` 、 `/detail/123` 都可以匹配到该 `Route`

JSX 复制代码

```jsx
<NavLink to="/detail/abc123">详情</NavLink>

<Switch>
    ... 其他Route
    <Route path="/detail/:id" component={Detail}/>
    <Route component={NoMatch} />
</Switch>
```

获取参数方式如下：

JSX 复制代码

```jsx
console.log(props.match.params.xxx)
```

26.3.2. search传递参数

在跳转的路径中添加了一些query参数；

JSX 复制代码

```jsx
<NavLink to="/detail2?name=why&age=18">详情2</NavLink>

<Switch>
  <Route path="/detail2" component={Detail2}/>
</Switch>
```

获取形式如下：

JavaScript 复制代码

```javascript
console.log(props.location.search)
```

26.3.3. to传入对象

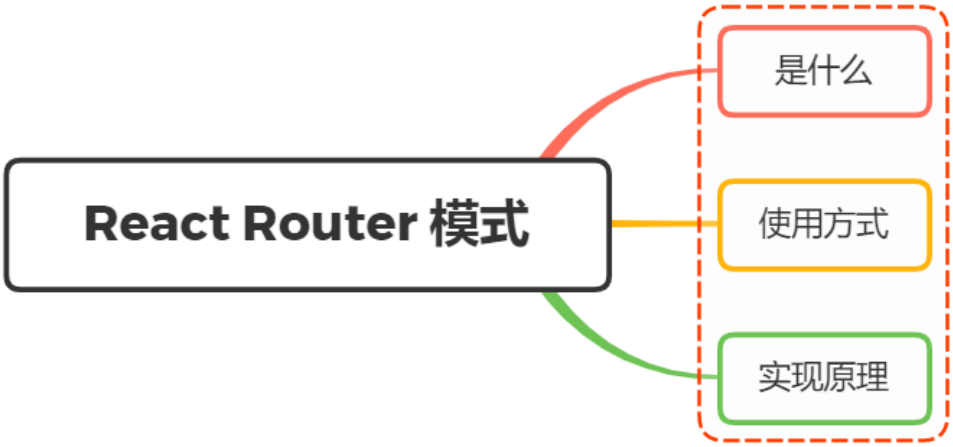
传递方式如下：

```jsx
<NavLink to={{
    pathname: "/detail2",
    query: {name: "kobe", age: 30},
    state: {height: 1.98, address: "洛杉矶"},
    search: "?apikey=123"
  }}>
  详情2
</NavLink>
```

获取参数的形式如下：

```javascript
console.log(props.location)
```

27. 说说React Router有几种模式？实现原理？



27.1. 是什么

在单页应用中，一个 `web` 项目只有一个 `html` 页面，一旦页面加载完成之后，就不用因为用户的操作而进行页面的重新加载或者跳转，其特性如下：

- 改变 url 且不让浏览器像服务器发送请求
- 在不刷新页面的前提下动态改变浏览器地址栏中的URL地址

其中主要分成了两种模式：

- hash 模式：在url后面加上#，如http://127.0.0.1:5500/home/#/page1

- history 模式：允许操作浏览器的曾经在标签页或者框架里访问的会话历史记录

27.2. 使用

`React Router` 对应的 `hash` 模式和 `history` 模式对应的组件为：

- HashRouter
- BrowserRouter

这两个组件的使用都十分的简单，作为最顶层组件包裹其他组件，如下所示

JSX | 复制代码

```jsx
// 1.import { BrowserRouter as Router } from "react-router-dom";
// 2.import { HashRouter as Router } from "react-router-dom";

import React from 'react';
import {
  BrowserRouter as Router,
  // HashRouter as Router
  Switch,
  Route,
} from "react-router-dom";
import Home from './pages/Home';
import Login from './pages/Login';
import Backend from './pages/Backend';
import Admin from './pages/Admin';


function App() {
  return (
    <Router>
        <Route path="/login" component={Login}/>
        <Route path="/backend" component={Backend}/>
        <Route path="/admin" component={Admin}/>
        <Route path="/" component={Home}/>
    </Router>
  );
}

export default App;
```

27.3. 实现原理

路由描述了 `URL` 与 `UI` 之间的映射关系，这种映射是单向的，即 URL 变化引起 UI 更新（无需刷新页面）

下面以 `hash` 模式为例子，改变 `hash` 值并不会导致浏览器向服务器发送请求，浏览器不发出请求，也就不会刷新页面

`hash` 值改变，触发全局 `window` 对象上的 `hashchange` 事件。所以 `hash` 模式路由就是利用 `hashchange` 事件监听 `URL` 的变化，从而进行 `DOM` 操作来模拟页面跳转

`react-router` 也是基于这个特性实现路由的跳转

下面以 `HashRouter` 组件分析进行展开：

27.4. HashRouter

`HashRouter` 包裹了整应用，

通过 `window.addEventListener('hashChange',callback)` 监听 `hash` 值的变化，并传递给其嵌套的组件

然后通过 `context` 将 `location` 数据往后代组件传递，如下：

JSX | 复制代码

```jsx
import React, { Component } from 'react';
import { Provider } from './context'
// 该组件下Api提供给子组件使用
class HashRouter extends Component {
  constructor() {
    super()
    this.state = {
      location: {
        pathname: window.location.hash.slice(1) || '/'
      }
    }
  }
  // url路径变化 改变location
  componentDidMount() {
    window.location.hash = window.location.hash || '/'
    window.addEventListener('hashchange', () => {
      this.setState({
        location: {
          ...this.state.location,
          pathname: window.location.hash.slice(1) || '/'
        }
      }, () => console.log(this.state.location))
    })
  }
  render() {
    let value = {
      location: this.state.location
    }
    return (
      <Provider value={value}>
        {
          this.props.children
        }
      </Provider>
    );
  }
}

export default HashRouter;
```

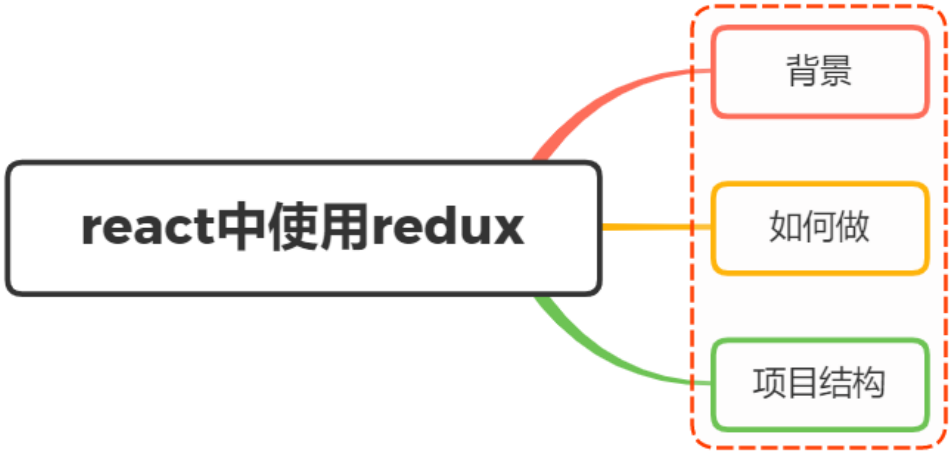
27.4.1. Router

`Router` 组件主要做的是通过 `BrowserRouter` 传过来的当前值，通过 `props` 传进来的 `path` 与 `context` 传进来的 `pathname` 进行匹配，然后决定是否执行渲染组件

JavaScript | 复制代码

```
import React, { Component } from 'react';
import { Consumer } from './context'
const { pathToRegexp } = require("path-to-regexp");
class Route extends Component {
  render() {
    return (
      <Consumer>
        {
          state => {
            console.log(state)
            let {path, component: Component} = this.props
            let pathname = state.location.pathname
            let reg = pathToRegexp(path, [], {end: false})
            // 判断当前path是否包含pathname
            if(pathname.match(reg)) {
              return <Component></Component>
            }
            return null
          }
        }
      </Consumer>
    );
  }
}
export default Route;
```

28. 你在React项目中是如何使用Redux的？项目结构是如何划分的？



28.1. 背景

在前面文章了解中，我们了解到 `redux` 是用于数据状态管理，而 `react` 是一个视图层面的库

如果将两者连接在一起，可以使用官方推荐 `react-redux` 库，其具有高效且灵活的特性

`react-redux` 将组件分成：

- 容器组件：存在逻辑处理
- UI 组件：只负责现显示和交互，内部不处理逻辑，状态由外部控制

通过 `redux` 将整个应用状态存储到 `store` 中，组件可以派发 `dispatch` 行为 `action` 给 `store`

其他组件通过订阅 `store` 中的状态 `state` 来更新自身的视图

28.2. 如何做

使用 `react-redux` 分成了两大核心：

- Provider
- connection

28.2.1. Provider

在 `redux` 中存在一个 `store` 用于存储 `state` ，如果将这个 `store` 存放在顶层元素中，其他组件都被包裹在顶层元素之上

那么所有的组件都能够受到 `redux` 的控制，都能够获取到 `redux` 中的数据

使用方式如下

```JavaScript
<Provider store = {store}>
    <App />
<Provider>
```

28.2.2. connection

`connect` 方法将 `store` 上的 `getState` 和 `dispatch` 包装成组件的 `props`

导入 `conect` 如下：

```JavaScript
import { connect } from "react-redux";
```

用法如下：

```JavaScript
connect(mapStateToProps, mapDispatchToProps)(MyComponent)
```

可以传递两个参数：

- mapStateToProps
- mapDispatchToProps

28.2.3. mapStateToProps

把 `redux` 中的数据映射到 `react` 中的 `props` 中去

如下

```JSX
const mapStateToProps = (state) => {
    return {
        // prop : state.xxx  | 意思是将state中的某个数据映射到props中
        foo: state.bar
    }
}
```

组件内部就能够通过 `props` 获取到 `store` 中的数据

```
class Foo extends Component {
    constructor(props){
        super(props);
    }
    render(){
        return(
         // 这样子渲染的其实就是state.bar的数据了
            <div>this.props.foo</div>
        )
    }
}
Foo = connect()(Foo)
export default Foo
```

28.2.4. mapDispatchToProps

将 `redux` 中的 `dispatch` 映射到组件内部的 `props` 中

```jsx
const mapDispatchToProps = (dispatch) => { // 默认传递参数就是dispatch
  return {
    onClick: () => {
      dispatch({
        type: 'increatment'
      });
    }
  };
}
```

JavaScript | 复制代码

```
class Foo extends Component {
    constructor(props){
        super(props);
    }
    render(){
        return(

            <button onClick = {this.props.onClick}>点击increase</button>
        )
    }
}
Foo = connect()(Foo);
export default Foo;
```

28.3. 小结

整体流程图大致如下所示：