# Next+Koa2开发github

## 项目准备

### 安装

### 手动搭建

yarn add react react-dom next

根目录创建pages文件夹

#### Package.json文件：

{

"name": "next-github",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"dev": "next",

"build": "next build",

"start": "next start"

},

"author": "",

"license": "ISC",

"dependencies": {

"next": "^9.0.5",

"react": "^16.9.0",

"react-dom": "^16.9.0"

}

}

增加一下代码

"scripts": {

"dev": "next",

"build": "next build",

"start": "next start"

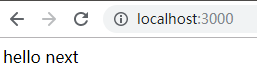
},

yarn dev 打开开发环境

#### pages目录下新建index.js文件

import React from 'react'

export default () => <span>hello next</span>



### 2. create-next-app

安装

npm i -g create-next-app

使用

npx create-next-app next-create 或者 yarn create next-app next-create 安装next项目

上面命令都创建了名为next-create的项目

### Next作为koa中间件使用

#### 安装koa

yarn add koa

根目录下创建server.js

const Koa = require('koa')

const next = require('next')

const dev = process.env.NODE\_ENV !== 'production'

const app = next({ dev })

const handle = app.getRequestHandler()

app.prepare().then(() => {

const server = new Koa()

server.use(async (ctx, next) => {

await handle(ctx.req, ctx.res)

ctx.respond = false

})

})

Package.json文件修改scripts

"dev": "node server.js",

yarn dev 启动

## koa的使用

### 基础用法

#### server.js

const Koa = require('koa')

const next = require('next')

const dev = process.env.NODE\_ENV !== 'production'

const app = next({ dev })

const handle = app.getRequestHandler()

const server = new Koa()

server.use(async (ctx, next) => {

const path = ctx.path

const method = ctx.method

ctx.body = `<span>Koa Render ${path} ${method}</span>`

await next()

})

server.use(async (ctx, next) => {

ctx.body = '<span>Koa Render2</span>'

})

server.listen(3000, () => {

console.log('koa server listening on 3000...')

})

### Koa-router

#### 安装koa-router

yarn add koa-router

#### koa-router简单使用

const Router = require('koa-router')

const router = new Router()

server.use(router.routes())

#### 示例：

const Koa = require('koa')

const Router = require('koa-router')

const next = require('next')

const dev = process.env.NODE\_ENV !== 'production'

const app = next({ dev })

const handle = app.getRequestHandler()

const server = new Koa()

const router = new Router()

router.get('/test', async (ctx) => {

ctx.body = `<p>this is /test </p>`

})

router.get('/test/:id', async (ctx) => {

ctx.body = `<p>this is /test ${ctx.params.id}</p>`

})

router.get('/json', async (ctx) => {

ctx.body = { success: true }

// ctx.set('Content-Type', 'application/json')

})

server.use(router.routes())

server.listen(3000, () => {

console.log('koa server listening on 3000...')

})

## Redis的安装使用

### 安装redise

cnpm i koa-generic-session koa-redis --save

### \* redis 远程连接方法

1、修改redis服务器的配置文件

vi redis.conf

注释以下绑定的主机地址

# bind 127.0.0.1

或

vim redis.conf

bind 0.0.0.0

protected-mode no

!!!!坑 修改配置文件启动时，记得指定配置文件

redis-server ./redis.conf

### 远程连接：

redis-cli -h 129.28.187.206 -p 6379

命令：

keys \* :反馈存储的所有key值

get key值 :获取指定key的数据

hget fix name :不经过sesson读取redis

### redis的基础使用

#### 连接redis

redis-cli

#### 获取一个值

get a

#### 设置一个值

set b 111

#### 配置密码

Redis.conf文件修改一下代码

requirepass foobared 修改密码为foobared

#### 连接：

Redis-cli –p 6379

#### 认证密码

auth foobared

#### 设置过期时间

setex c 10 1 设置一个a过期时间为10秒 值为1

#### 设置带前缀类的key

set session:sessionId 123

#### 获取

get session:sessionId

#### 查看所有key值

Keys \*

#### 删除某个key

del a

## nodejs连接redis数据库

### ioredis 模块

安装 yarn add ioredis

### 使用

#### test-redis.js

async function test() {

const Redis = require('ioredis')

const redis = new Redis({

port: 6379,

host: '129.28.187.206'

// password: '123456'

})

await redis.set('c', 123)

await redis.setex('d', 10 , 2345)

const keys = await redis.keys('\*')

console.log(keys, await redis.get('c'), await redis.get('d'))

setTimeout(async () => {

console.log(await redis.get('d'))

},13000)

}

test()

效果：



## 项目引入ant-design

### 安装

注意！ Nextjs 默认是不支持直接import引入css文件的， 需要第三方支持

yarn add @zeit/next-css

项目根目录新建文件

### next-config.js

const widthCss = require('@zeit/next-css')

if (typeof require !== undefined) {

require.extensions['.css'] = file => {}

}

module.exports = widthCss({})

### 安装antd

yarn add antd

yarn add babel-plugin-import

### 根目录新建文件 .babelrc

{

"presets": ["next/babel"],

"plugins": [

[

"import",

{

"libraryName": "antd"

}

]

]

}

引入以上代码后的效果是：

把

Import { Button } from ‘antd’

替换为

Import Button from ‘antd/lib/button’

这样做到按需加载

### 引入样式文件

Pages文件夹下新建一个文件\_app.js用于覆盖next默认的appjs

#### \_app.js

import App from 'next/app'

import 'antd/dist/antd.css'

export default App

#### .babelrc

{

"presets": ["next/babel"],

"plugins": [

[

"import",

{

"libraryName": "antd",

"style": "css" // 按需引入css目前可能会有问题（暂未发现）

}

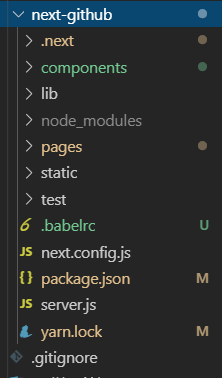
]

]

}

## 项目构建和相关技术

### 目录结构



### 路由跳转

#### Link组件

1. 必须包含可渲染的节点
2. 这个节点每层必须只能有唯一节点

##### Index.js

import { Button } from 'antd'

import Link from 'next/link'

export default () => (

<Link href='test/bbb'>

<Button>index</Button>

</Link>

)

或

<Link href='test/bbb'>

<div>

<Button>index</Button>

<Button>index</Button>

</div>

</Link>

#### Router模块

import { Button } from 'antd'

import Link from 'next/link'

import Router from 'next/router'

export default () => {

function gotoTestB() {

Router.push('/test/bbb')

}

return (

<>

<Link href='test/bbb'>

<Button>index</Button>

</Link>

<Button onClick={ gotoTestB }>

test b

</Button>

</>

)

}

#### 动态路由

注意Nextjs动态路由只能通过query形式

如 localhost:3000/test/b?a=1

Router.push('/test/bbb?id=2')

// 或

Router.push({

pathname: '/test/b',

query: {

id: 2

}

})

#### 路由映射

##### Link方式

<Link href='/test/bbb?id=1' as="/test/bbb/1">

<Button>index</Button>

</Link>

##### Router方式

Router.push({

pathname: '/test/bbb',

query: {

id: 2

}

}, '/test/bbb/2')

解决刷新页面路由映射出现404的方案

在koa的server.js里面配置如下内容

const router = new Router()

router.get('/test/bbb/:id', async (ctx) => {

const id = ctx.params.id

await handle(ctx.req, ctx.res, {

pathname: '/test/bbb',

query: { id }

})

ctx.respond = false

})

server.use(router.routes())

此时在进入test/bbb/1时刷新页面则也会正常显示页面了

#### 路由变化的钩子

const event = [

'routeChageStart',

'routeChangeComplete',

'routeChangeError',

'beforeHistoryChange',

'hashChangeStart',

'hashChangeComplete '

]

function makeEvent(type) {

return (...args) => {

console.log(type, ...args)

}

}

event.forEach(event => {

Router.events.on(event, makeEvent(event))

})

### Next数据获取的方式

#### Nextjs数据获取规范

在getInitialProps里面获取数据而不是通过compontentDidMount里面

#### 渲染同步数据

import { withRouter } from 'next/router'

import Link from 'next/link'

const B = ({ router, name }) => <Link href="#aaa"><a >bbba{ router.query.id } { name } </a></Link>

B.getInitialProps = () => {

return {

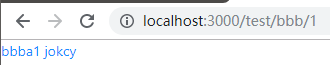
name: 'jokcy'

}

}

export default withRouter(B)

效果如下



#### 渲染异步数据

B.getInitialProps = async () => {

const promise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve({

name: 'jokcy'

})

}, 5000)

})

return await promise

}

### 自定义app

#### 作用：

##### 固定layout

###### Layout.jsx

import { Button } from 'antd'

import Link from 'next/link'

export default ({ children }) => (

<div>

<header>

<Link href='/test/bbb?id=1' as="/test/bbb/1">

<Button>index1</Button>

</Link>

<Link href='/test/bbb?id=2' as="/test/bbb/2">

<Button>index2</Button>

</Link>

</header>

{ children }

</div>

)

###### \_app.js

import App from 'next/app'

import Layout from '../components/layout'

import 'antd/dist/antd.css'

class MyApp extends App {

static async getInitialProps ({ Component }) {

let pageProps

if (Component.getInitialProps) {

pageProps = await Component.getInitialProps()

}

return {

pageProps

}

}

render() {

const { Component, pageProps } = this.props

return (

<Layout>

<Component { ...pageProps } />

</Layout>

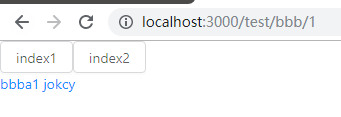
)

}

}

export default MyApp

###### 效果



##### 保持一些公用状态

##### 给页面传入一些自定义数据

##### 自定义错误处理

### 自定义document

只有在服务器渲染的时候才会被调用

用来修改服务端渲染的文档内容

一般用来配合第三方的css-in-js方案使用

Pages下新建\_document,js

import Document, { Html, Head, Main, NextScript } from 'next/document'

class MyDocument extends Document {

static async getInitialProps(ctx) { // 若要使用它需要先继承Document里面封装的getInitialProps

const props = await Document.getInitialProps(ctx)

return {

...props

}

}

render() {

return <Html>

<Head>

<title>MyApp</title> {/\* 不建议 \*/}

<style>{`.test { font-size: 35px }`}</style>

</Head>

<body className="test">

<Main />

<NextScript />

</body>

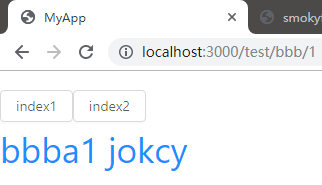
</Html>

}

}

export default MyDocument

此时重启项目后



### 自定义样式

#### css-in-jsx

const colors = 'green'

const B = ({ router, name }) => (

<>

<Link href="#aaa">

<a className="link">

bb

{ router.query.id } { name }

</a>

</Link>

1. 只在当前组件生效

<style jsx>{`

a {

color: blue

}

.link {

color: ${colors}

}

`}</style>

2. 样式全局生效

<style jsx global>{` ${/\* 注意这里的全局样式必须在页面渲染后才会生效 \*/''}

a {

color: ${colors}

}

`}</style>

</>

)

### Next集成styled-component

#### 安装

yarn add styled-components babel-plugin-styled-components

#### .babelrc

{

"presets": ["next/babel"],

"plugins": [

[

"import",

{

"libraryName": "antd",

"style": "css"

}

],

[

"styled-components", { "ssr": true }

]

]

}

#### \_document,js

import Document, { Html, Head, Main, NextScript } from 'next/document'

import { ServerStyleSheet } from 'styled-components'

function withLog(Comp) {

return (props) => {

console.log(props)

return <Comp {...props} />

}

}

class MyDocument extends Document {

static async getInitialProps(ctx) { // 若要使用它需要先继承Document里面封装的getInitialProps

const sheet = new ServerStyleSheet()

const originalRenderPage = ctx.renderPage

try {

ctx.renderPage = () => originalRenderPage({

enhanceApp: App => (props) => sheet.collectStyles(<App {...props} />) // \_app里面返回的MyApp

// enhanceComponent: Component => withLog(Component) // 每个页面对应的js文件

})

const props = await Document.getInitialProps(ctx)

return {

...props,

styles: <>{ props.styles }{ sheet.getStyleElement() }</>

}

} finally {

sheet.seal

}

}

render() {

return <Html>

<Head>

{/\* <title>MyApp</title> 不建议 \*/}

{/\* <style>{`.test { color: red }`}</style> \*/}

</Head>

<body className="test">

<Main />

<NextScript />

</body>

</Html>

}

}

export default MyDocument

#### bbb.js 相关代码

import styled from 'styled-components'

const Title = styled.h1`

color: yellow;

font-size: 40px

`

const B = ({ router, name }) => (

<div>

<Title>this is title</Title>

<Link href="#aaa">

<a className="link">

bb

{ router.query.id } { name }

</a>

</Link>

</div>

)

#### 此时渲染的页面



### Nextjs中异步模块和组件的加载(lazyLoading)

#### 异步加载模块

yarn add moment

##### bbb.js

当我们用到moment.js时才会被异步加载 相关代码：

B.getInitialProps = async () => {

const moment = await import ('moment')

return {

name: 'jokcy',

time: moment.default(Date.now() - 60 \* 1000).fromNow()

}

}

#### 异步加载组件

##### bbb.js

import dynamic from 'next/dynamic'

// import Comp from '../../components/comp'

const Comp = dynamic(import('../../components/comp')) // 异步加载组件

const B = ({ router, name, time }) => (

<div>

<Title>this is title { time }</Title>

<Comp />

</div>

)

### Nextjs中的配置项（next.config.js）

可配置项如下

const configs = {

// 编译文件的输出目录

distDir: 'dest',

// 是否给每个路由生成Etag: 用来缓存验证

generateEtags: true,

// 页面内容缓存配置

onDemandEntries: {

// 内容在内存中缓存的时常

maxInactiveAge: 25 \* 1000,

// 同时缓存多少个页面

pageBufferLength: 2,

},

// 在page目录下哪种后缀的文件会被认为是页面

pageExtensions: ['jsx', 'js'],

// 配置buildId

generateBuildId: async () => {

if (process.env.YOUR\_BUILD\_ID) {

return process.env.YOUR\_BUILD\_ID

}

// 返回null使用默认的unique id

return null

},

// 手动修改webpack config

webpack(config, options) {

return config

},

// 修改webpackDevMiddleware配置

webpackDevMiddleware: config => {

return config

},

// 可以在页面上通过process.env.customKey 获取 value

env: {

customKey: 'value',

},

// 下面两个要通过 'next/config' 来获取

// 只有服务端渲染时才会获取的配置

serverRuntimeConfig: {

mySecret: 'secret',

secondSecret: process.env.SECOND\_SECRET

},

// 在服务端渲染和客户端渲染都可以获取的配置

publicRuntimeConfig: {

staticFolder: '/static',

},

}

测试使用其中的三个

module.exports = widthCss({

env: {

customKey: 'value',

},

serverRuntimeConfig: {

mySecret: 'secret',

secondSecret: process.env.SECOND\_SECRET

},

// 在服务端渲染和客户端渲染都可以获取的配置

publicRuntimeConfig: {

staticFolder: '/static',

}

})

bbb.js()相关代码

import getConfig from 'next/config'

const { serverRuntimeConfig, publicRuntimeConfig } = getConfig()

const B = ({ router, name, time }) => {

console.log(serverRuntimeConfig, publicRuntimeConfig)

return (

<div>

<Title>this is title { time } { process.env.customKey }</Title>

</div>

)

}

完整代码

import { withRouter } from 'next/router'

import Head from 'next/head'

import getConfig from 'next/config'

import Link from 'next/link'

import styled from 'styled-components'

import dynamic from 'next/dynamic'

// import Comp from '../../components/comp'

const Comp = dynamic(import('../../components/comp'))

const { serverRuntimeConfig, publicRuntimeConfig } = getConfig()

const Title = styled.h1`

color: yellow;

font-size: 40px

`

const colors = 'green'

const B = ({ router, name, time }) => {

console.log(serverRuntimeConfig, publicRuntimeConfig)

return (

<div>

<Title>this is title { time } { process.env.customKey }</Title>

<Comp />

<Link href="#aaa">

<a className="link">

bb

{ router.query.id } { name }

</a>

</Link>

<style jsx>{`

a {

color: blue

}

.link {

color: ${colors}

}

`}</style>

<style jsx global>{` ${/\* 注意这里的全局样式必须在页面渲染后才会生效 \*/''}

a {

color: ${colors}

}

`}</style>

</div>

)

}

B.getInitialProps = async () => {

const moment = await import ('moment')

return {

name: 'jokcy',

time: moment.default(Date.now() - 60 \* 1000).fromNow()

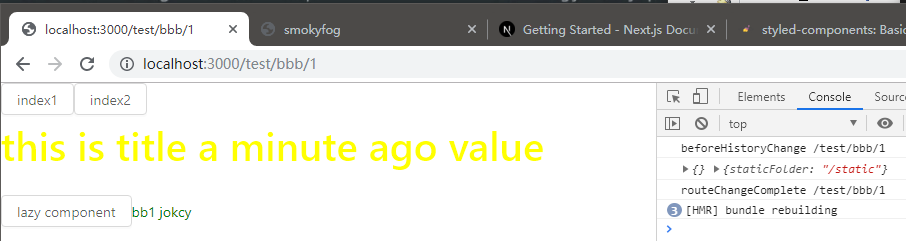
}

}

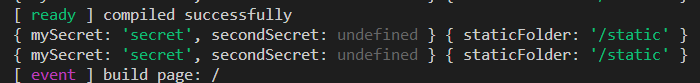
export default withRouter(B)

效果如下

页面



服务端



### 服务端渲染（SSR）流程

…

## HOOKS

### 基础用法

#### 常用类组件

import React from 'react'

class MyCount extends React.Component {

state = {

count: 0

}

componentDidMount() {

this.interval = setInterval(() => {

this.setState({ count: this.state.count + 1 })

}, 1000)

}

componentWillUnmount() {

if (this.interval) {

clearInterval(this.interval)

}

}

render () {

return <span>{ this.state.count }</span>

}

}

export default MyCount

#### 使用React Hooks写法

import React, { useState, useEffect } from 'react'

function MyCountFunc() {

const [ count, setCount ] = useState(0)

useEffect(() => {

const interval = setInterval(() => {

setCount(c => c + 1)

}, 1000)

return () => clearInterval(this.interval)

}, [])

return <span>{ count }</span>

}

export default MyCountFunc

这种写法的效果和上面类的方法是一致的

### State Hooks

#### useState

// setCount用法

// 1. setCount(1)

// 2. setCount((c) => c + 1 )

function MyCountFunc() {

const [ count, setCount ] = useState(0) // [a, b]

useEffect(() => {

const interval = setInterval(() => {

// setCount用法

// setCount(1)

// setCount((c) => c + 1 )

setCount(c => c + 1)

}, 1000)

return () => clearInterval(interval)

}, [])

return <span>{ count }</span>

}

export default MyCountFunc

#### useReducer

import React, { useState, useReducer, useEffect } from 'react'

function countReducer(state, action) {

switch (action.type) {

case 'add':

return state + 1

case 'minus':

return state - 1

default:

return state

}

}

function MyCountFunc() {

const [ count, dispatchCount ] = useReducer(countReducer, 0)

useEffect(() => {

const interval = setInterval(() => {

dispatchCount({ type: 'add' })

}, 1000)

return () => clearInterval(interval)

}, [])

return <span>{ count }</span>

}

export default MyCountFunc

### Effect Hooks

#### useEffect

import React, { useState, useReducer, useEffect } from 'react'

function MyCountFunc() {

const [ count, dispatchCount ] = useReducer(countReducer, 0)

const [ name, setName ] = useState('jock')

useEffect(() => {

console.log('effect invoked') // 后执行

return () => console.log('effect deteched') // 先执行

}, [count]) // 只有数组中的状态发生变化时才会先卸载return的值，在重新执行上面的部分， 若传入这个数组，则每次发生状态的变更都会执行卸载然后执行上面的内容

return (

<div>

<input value={ name } onChange={(e) => setName(e.target.value)} />

<button onClick={() => dispatchCount({type: 'add'})}>{ count }</button>

</div>

)

}

export default MyCountFunc

#### useLayoutEffect

function MyCountFunc() {

const [ count, dispatchCount ] = useReducer(countReducer, 0)

const [ name, setName ] = useState('jock')

// 会在更新dom节点树且被挂载到页面上之后执行

useEffect(() => {

console.log('effect invoked') // 后执行

return () => console.log('effect deteched') // 先执行

}, [count]) // 只有数组中的状态发生变化时才会先卸载return的值，在重新执行上面的部分， 若传入这个数组，则每次发生状态的变更都会执行卸载然后执行上面的内容

// 会在更新dom节点树而没有被挂载到页面上之前执行

useLayoutEffect(() => {

console.log('layout effect invoked') // 后执行

return () => console.log('layout effect deteched') // 先执行

}, [count])

return (

<div>

<input value={ name } onChange={(e) => setName(e.target.value)} />

<button onClick={() => dispatchCount({type: 'add'})}>{ count }</button>

</div>

)

}

export default MyCountFunc

useEffect : 会在更新dom节点树且被挂载到页面上之后执行

useLayoutEffect: 会在更新dom节点树而没有被挂载到页面上之前执行

为了用户体验 我们一般会优先使用useEffect

### Context Hook

Lib目录下新建文件

#### my-context

import React from 'react'

export default React.createContext('')

#### \_app.js

import App from 'next/app'

import Layout from '../components/layout'

import MyContext from '../lib/my-context'

import { Button } from 'antd'

import 'antd/dist/antd.css'

class MyApp extends App {

state = {

context: 'value'

}

static async getInitialProps ({ Component, ctx }) {

let pageProps

if (Component.getInitialProps) {

pageProps = await Component.getInitialProps(ctx)

}

return {

pageProps

}

}

render() {

const { Component, pageProps } = this.props

return (

<Layout>

<MyContext.Provider value={ this.state.context } >

<Component { ...pageProps } />

<Button onClick={() => this.setState({context: this.state.context + '111'})}>update context</Button>

</MyContext.Provider>

</Layout>

)

}

}

export default MyApp

#### ccc.js

import

React,

{ useState, useReducer, useEffect, useLayoutEffect, useContext }

from 'react'

import MyContext from '../../lib/my-context'

function MyCountFunc() {

const [ count, dispatchCount ] = useReducer(countReducer, 0)

const [ name, setName ] = useState('jock')

const context = useContext(MyContext)

useEffect(() => {

console.log('effect invoked') // 后执行

return () => console.log('effect deteched') // 先执行

}, [count])

useLayoutEffect(() => {

console.log('layout effect invoked') // 后执行

return () => console.log('layout effect deteched') // 先执行

}, [count])

return (

<div>

<input value={ name } onChange={(e) => setName(e.target.value)} />

<button onClick={() => dispatchCount({type: 'add'})}>{ count }</button>

<p>{ context }</p>

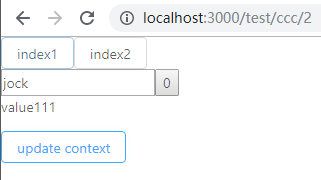
</div>

)

}

export default MyCountFunc

#### 此时渲染页面



### Ref Hook

#### Class类组件的使用方法

class MyCount extends React.Component {

constructor() {

super()

this.ref = React.createRef()

}

state = {

count: 0

}

componentDidMount() {

this.refs.current

this.interval = setInterval(() => {

this.setState({ count: this.state.count + 1 })

}, 1000)

}

componentWillUnmount() {

if (this.interval) {

clearInterval(this.interval)

}

}

render () {

return <span ref={this.ref}>{ this.state.count }</span>

}

}

#### Hooks使用

import

React,

{

useState,

useReducer,

useEffect,

useLayoutEffect,

useContext,

useRef

} from 'react'

function countReducer(state, action) {

switch (action.type) {

case 'add':

return state + 1

case 'minus':

return state - 1

default:

return state

}

}

function MyCountFunc() {

// const [ count, setCount ] = useState(0) // [a, b]

const [ count, dispatchCount ] = useReducer(countReducer, 0)

const [ name, setName ] = useState('jock')

const context = useContext(MyContext)

const inputRef = useRef()

useEffect(() => {

console.log(inputRef) //此时获取的即是input的dom节点

return () => console.log('effect deteched') // 先执行

}, [count])

return (

<div>

<input ref={ inputRef } value={ name } onChange={(e) => setName(e.target.value)} />

<button onClick={() => dispatchCount({type: 'add'})}>{ count }</button>

<p >{ context }</p>

</div>

)

}

export default MyCountFunc

### 优化Hook

#### b.js

import

React,

{

useState,

useReducer,

useEffect,

useLayoutEffect,

useContext,

useRef,

memo

} from 'react'

function countReducer(state, action) {

switch (action.type) {

case 'add':

return state + 1

case 'minus':

return state - 1

default:

return state

}

}

function MyCountFunc() {

const [ count, dispatchCount ] = useReducer(countReducer, 0)

const [ name, setName ] = useState('jock')

const config = {

text: `count is ${count}`,

color: count > 3 ? 'red' : 'blue',

}

return (

<div>

<input value={ name } onChange={(e) => setName(e.target.value)} />

<Child

config={config}

onButtonClick={() => dispatchCount({ type: 'add' })}

/>

</div>

)

}

const Child = memo(function Child({ onButtonClick, config }) {

console.log('child render')

return (

<button onClick={ onButtonClick } style={{color: config.color}}>

{config.text}

</button>

)

})

export default MyCountFunc

这时无论当我点击button按钮还是在输入框里填写文字，都会触发Child组件里面console这样就影响了性能，为此我们在Child组件外层套上一层memo但此时发现当在input输入内容时child的console.log依然会被调用，原因是当input框发生改变时调用了MyCountFun函数的setName，此时MyCountFun函数会被重新执行，函数内部的config也随之重新声明，而Child组件中传入了config此时发生了更新，因此Child组件会被重新执行

我们希望在count没有发生变化的时候， config还是原来的config, 此时我们使用uesMemo这个Hook,

此时我们把config进行包装

const config = useMemo(

() => ({

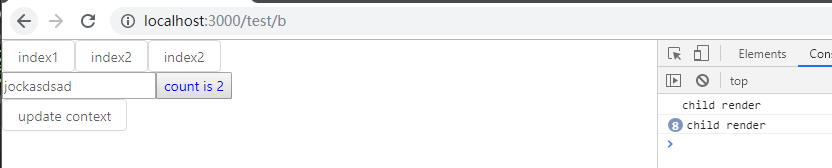
text: `count is ${count}`,

color: count > 3 ? 'red' : 'blue',

}

), [count])

此时我们再次在输入框里面修改发现还是会调用child里面的console方法



是由于

<Child

config={config}

onButtonClick={() => dispatchCount({ type: 'add' })}

/>

Child组件里面的onButtonClick调用的还是新声明的方法

此时我们又要用到一个新的hook useCallback

const handleButtonClick = useCallback(() => dispatchCount({ type: 'add' }), [])

return (

<div>

<input value={ name } onChange={(e) => setName(e.target.value)} />

<Child

config={config}

onButtonClick={handleButtonClick}

/>

</div>

)

此时我们再次输入内容时就不会再次重新调用Child了

#### b.js完整代码

import

React,

{

useState,

useReducer,

useEffect,

useLayoutEffect,

useContext,

useRef,

memo,

useMemo,

useCallback

} from 'react'

function countReducer(state, action) {

switch (action.type) {

case 'add':

return state + 1

case 'minus':

return state - 1

default:

return state

}

}

function MyCountFunc() {

const [ count, dispatchCount ] = useReducer(countReducer, 0)

const [ name, setName ] = useState('jock')

const config = useMemo(

() => ({

text: `count is ${count}`,

color: count > 3 ? 'red' : 'blue',

}

), [count])

const handleButtonClick = useCallback(

() => dispatchCount({ type: 'add' }),

[]

)

//这里我们也可以用useMemo来记忆方法,实际上useCallback就是useMemo的简化

// const handleButtonClick = useMemo(

// () => () => dispatchCount({ type: 'add' }),

// []

// )

return (

<div>

<input value={ name } onChange={(e) => setName(e.target.value)} />

<Child

config={config}

onButtonClick={handleButtonClick}

/>

</div>

)

}

const Child = memo(function Child({ onButtonClick, config }) {

console.log('child render')

return (

<button onClick={ onButtonClick } style={{color: config.color}}>

{config.text}

</button>

)

})

export default MyCountFunc

### 闭包陷阱

const handleAlertClick = function () {

setTimeout(() => {

alert(count)

}, 2000)

}

return (

<div>

<input value={ name } onChange={(e) => setName(e.target.value)} />

<Child

config={config}

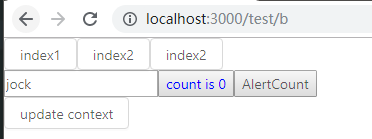
onButtonClick={handleButtonClick}

/>

<button onClick ={ handleAlertClick }>AlertCount</button>

</div>

)



此时我们先点击AlertCount这个按钮， 然后多次点击count is 0 按钮， 会发先alert弹窗依旧会显示为0

若想要显示的值为多次点击count is 0 按钮后的count值则

const countRef = useRef() // 因为useRef生成的是同一个对象，因此规避了这一闭包现象

countRef.current = count

const handleAlertClick = function () {

setTimeout(() => {

alert(countRef.current)

}, 2000)

}

return (

<div>

<input value={ name } onChange={(e) => setName(e.target.value)} />

<Child

config={config}

onButtonClick={handleButtonClick}

/>

<button onClick ={ handleAlertClick }>AlertCount</button>

</div>

)

}

#### b.js全部代码

import

React,

{

useState,

useReducer,

useEffect,

useLayoutEffect,

useContext,

useRef,

memo,

useMemo,

useCallback

} from 'react'

function countReducer(state, action) {

switch (action.type) {

case 'add':

return state + 1

case 'minus':

return state - 1

default:

return state

}

}

function MyCountFunc() {

const [ count, dispatchCount ] = useReducer(countReducer, 0)

const [ name, setName ] = useState('jock')

const countRef = useRef()

countRef.current = count

const config = useMemo(

() => ({

text: `count is ${count}`,

color: count > 3 ? 'red' : 'blue',

}

), [count])

const handleButtonClick = useCallback(

() => dispatchCount({ type: 'add' }),

[]

)

//这里我们也可以用useMemo来记忆方法,实际上useCallback就是useMemo的简化

// const handleButtonClick = useMemo(

// () => () => dispatchCount({ type: 'add' }),

// []

// )

const handleAlertClick = function () {

setTimeout(() => {

alert(countRef.current)

}, 2000)

}

return (

<div>

<input value={ name } onChange={(e) => setName(e.target.value)} />

<Child

config={config}

onButtonClick={handleButtonClick}

/>

<button onClick ={ handleAlertClick }>AlertCount</button>

</div>

)

}

const Child = memo(function Child({ onButtonClick, config }) {

console.log('child render')

return (

<button onClick={ onButtonClick } style={{color: config.color}}>

{config.text}

</button>

)

})

export default MyCountFunc

## Redux

Redux是一个单向数据流的状态管理工具

安装redux

yarn add redux

项目根目录下新建文件夹store

新建文件

### store.js

import { createStore } from 'redux'

const initialState = {

count: 0

}

const ADD = 'ADD'

function reducer(state = initialState, action) {

console.log(state, action)

switch (action.type) {

case ADD:

return { count: state.count + 1 }

default:

return state

}

}

const store = createStore(reducer, initialState)

console.log(store)

console.log(store.getState())

store.dispatch({ type: ADD })

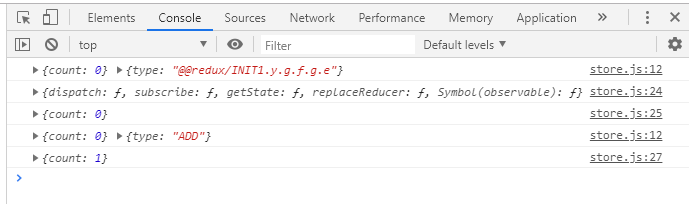
console.log(store.getState())

export default store

index.js

import store from '../store/store'

此时输出内容如下：



store.dispatch({ type: ADD })

store.subscribe(() => {

console.log(store.getState())

})

store.dispatch({ type: ADD })

