Next.js

Next.js 是一个轻量级的 React 服务端渲染应用框架。有了它我们可以简单轻松的实现React的服务端渲染，从而加快首屏打开速度，也可以作SEO（收索引擎优化了）。在没有Next.js的时候，用React开发需要配置很多繁琐的参数，如Webpack配置，Router配置和服务器端配置等....。如果需要作SEO，要考虑的事情就更多了，怎么样服务端渲染和客户端渲染保持一致就是一件非常麻烦的事情，需要引入很多第三方库。但有了Next.js，这些问题都解决了，使开发人员可以将精力放在业务逻辑上，从繁琐的配置中解放出来。

**[1：Next.js简介和创建项目](https://jspang.com/detailed?id=51" \l "toc21)**

**[Next.js简介](https://jspang.com/detailed?id=51" \l "toc34)**

官网地址：<https://www.nextjs.cn/>

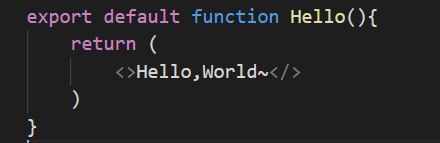
Next.js 是一个轻量级的 React 服务端渲染应用框架。

**关键特性和优势**

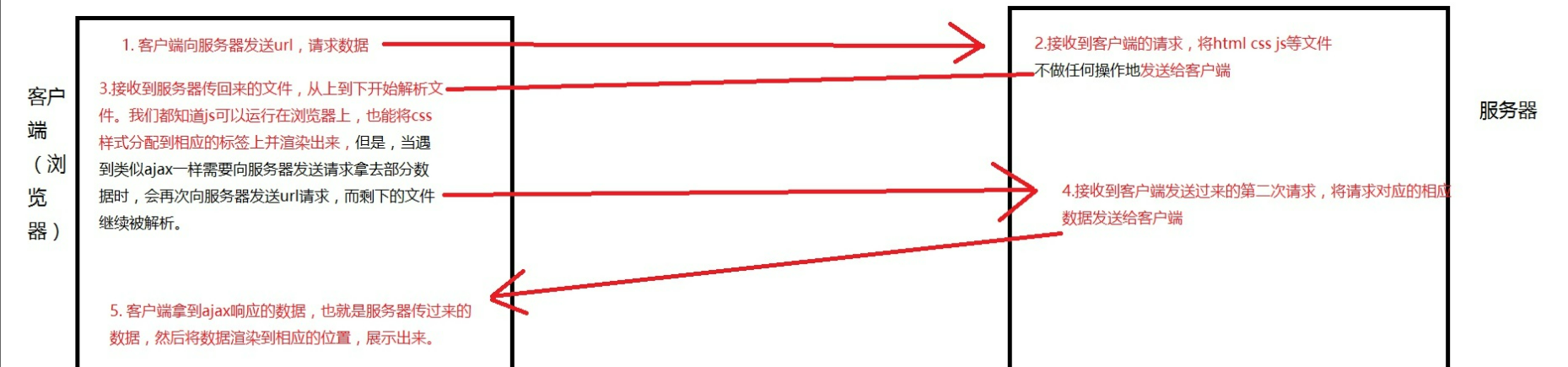




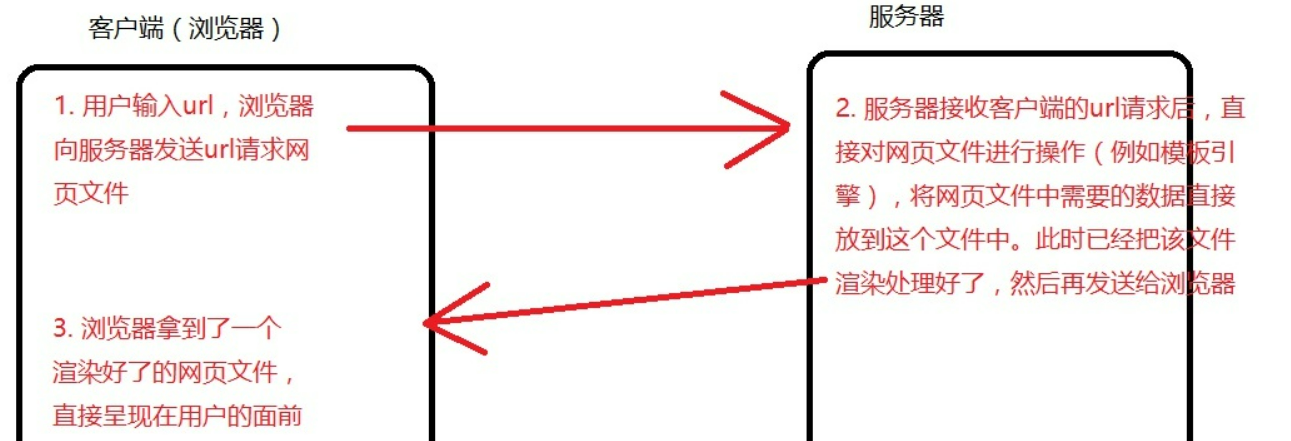
客户端和服务端渲染：（以Hello World为例）

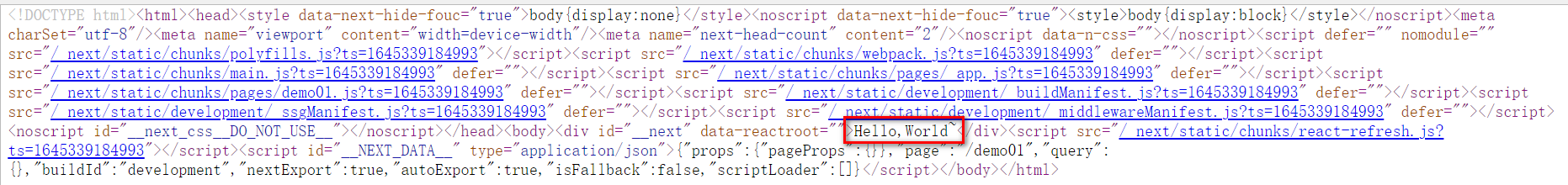


客户端渲染：





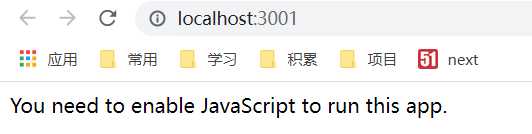




判断：

看源码；

关闭js:设置>安全和隐私设置>网站设置>Javascript



**用途**

如果你想使用React来开发需要SEO的应用，利于SEO 搜索引擎优化，即能被搜索引擎搜索到，能向用户展示你网页的东西，基本上就要使用Next.js。

**[手动创建Next.js项目](https://jspang.com/detailed?id=51" \l "toc35)**

**第一步： 建立文件夹**

创建一个Next.js项目，可以有两种方法进行，一种是手动创建，另一种是用create-next-app（脚手架）来创建。我们先来进行手动创建，这样虽然麻烦点，但是可以更容易让新手了解过程和原理。

先新建一个文件夹，名称可以自己起NextDemo文件夹。

mkdir NextDemo

npm init

这里的npm init 是用来把文件夹初始化成可管理的项目的，其实就是在根目录里给你添加了一个package.json的文件。

**第二步：安装所需要的依赖包**

接下来可以使用yarn来安装所需要的项目依赖包，先来安装下面三个react、react-dom和next。

yarn add react react-dom next

当让你也可以使用npm来进行安装，npm安装时记得要使用--save

npm install --save react react-dom next

安装完可以打开package.json文件查看一下dependencies的版本。

**第三步：增加快捷命令**

为了开发时简便的使用Next.js中的操作命令行工具，所以把常用的配置到package.json中，代码如下：

"scripts": {

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

"dev" : "next" ,

"build" : " next build",

"start" : "next start"

},

**第四步：创建pages文件夹和文件**

在根目录下，创建一个pages文件夹，这个文件夹是Next规定的，在这个文件夹下写入的文件，Next.js会自动创建对应的路由。有了文件夹以后，在文件下面创建一个index.js文件，这就是我们的首页了,然后用React Hooks的写法，写个最简单的Hello World。

function Index(){

return (

<div>Hello Next.js</div>

)

}

export default Index

写好后在终端中使用yarn dev来打开预览，在浏览器中可以看到输出了正确的结果.

**[2：creact-next-app快速创建Next.js项目](https://jspang.com/detailed?id=51" \l "toc26)**

create-next-app可以快速的创建Next.js项目，它就是一个脚手架，有了它只要一句命令就可以把项目需要的依赖包和基本目录都生成，工作中我基本不用手动的形式自己创建，全部使用create-next-app来创建。

**[安装create-next-app](https://jspang.com/detailed?id=51" \l "toc37)**

使用脚手架前，需要先进行全局安装。

npm install -g create-next-app

安装完成后，就可以通过create-next-app命令来创建一个Next.js的项目了。

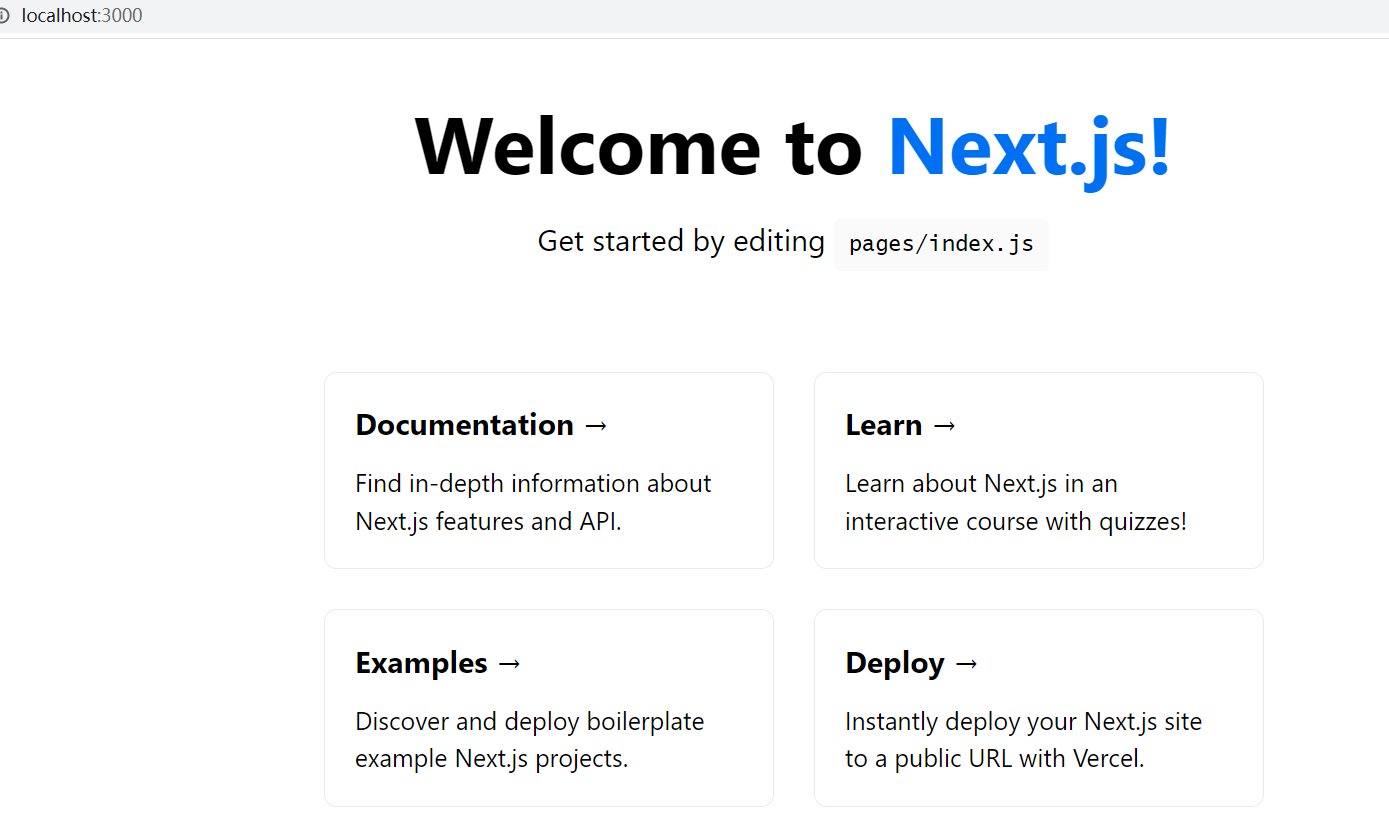
**[创建Next.js项目](https://jspang.com/detailed?id=51" \l "toc38)**

npx命令创建项目。

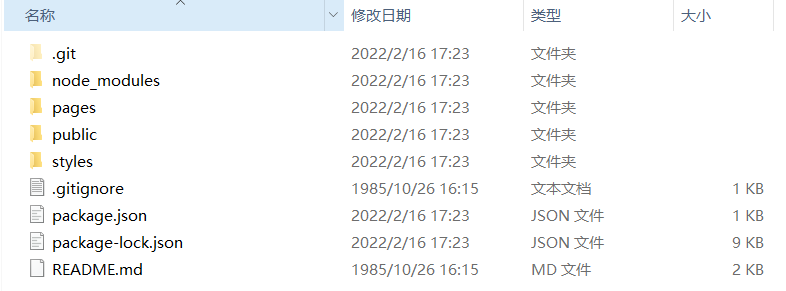
$ npx create-next-app next-create

输入后按回车，就会自动给我们进行安装项目需要的依赖。并且会给我们添加好命令。 稍等一会，全部安装完成后，可以进入项目母乳，执行npm run dev来测试项目。

在浏览器中输入http://localhost:3000/,看到下面的内容，说明项目生成成功.



**[项目结构介绍](https://jspang.com/detailed?id=51" \l "toc39)**



看到结果后，用VSCode打开目录，可以看到已经有了很多自动建立好的文件和文件夹，下面就简单的介绍一下这些它们的用处：

* node\_modules文件夹：Next项目的所有依赖包都在这里，一般我们不会修改和编辑这里的内容。
* pages文件夹：这里是放置页面的，这里边的内容会自动生成路由，并在服务器端渲染，渲染好后进行数据同步。
* public 文件夹：这里是放置项目需要的图片、图标等公共资源。
* styles 文件夹：这里是放置样式文件的。
* .gitignore文件： 这个主要是控制git提交和上传文件的，简称就是忽略提交。
* package.json文件：定义了项目所需要的文件和项目的配置信息。

**[3：Next.js的Page和Component的使用](https://jspang.com/detailed?id=51" \l "toc210)**

**[新建页面和访问路径](https://jspang.com/detailed?id=51" \l "toc311)**

直接在根目录下的pages文件夹下，新建一个hello.js页面。然后写入下面的代码：

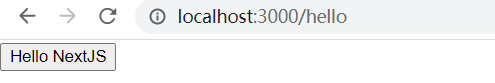
function Hello (){

return (<button>Hello NextJS </button>)

}

export default Hello;

只要写完上面的代码，Next框架就自动作好了路由，这个也算是Next的一个重要优点，给我们节省了大量的时间。

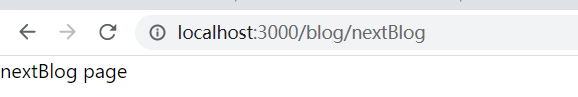


现在要作一个更深的页面，比如把有关博客的界面都放在这样的路径下http://localhost:3000/blog/nextBlog,其实只要在pages文件夹下再建立一个新的文件夹blog，然后进入blog文件夹，新建一个nextBlog.js文件，就可以实现了。

nextBlog.js文件内容,我们这里就用最简单的写法了

export default ()=><div>nextBlog page</div>

写完后，就可以直接在浏览器中访问了。



**[Component组件的制作](https://jspang.com/detailed?id=51" \l "toc312)**

制作组件也同样方便，比如要建立一个button组件，根目录下新建components目录，在components目录下建立一个文件button.js,然后写入下面代码:

export default ({children})=><button>{children}</button>

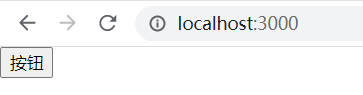
组件写完后需要先引入，比如我们在Index页面里进行引入：

import Button from '../components/button'

使用就非常简单了，直接写入标签就可以。

< Button >按钮</Button >

浏览器中查看



**[4\*：路由-基础和基本跳转](https://jspang.com/detailed?id=51" \l "toc213)**

路由（页面）

在 Next.js 中，存放在 pages 目录下的页面就是路由。

静态路由：/pages/about.js --- http://localhost:3000/about

动态路由：/pages/blog/[id].js--- http://localhost:3000/blog/1

获取所有路由：/pages/test/[…slug].js

--- [http://localhost:3000/test /1](http://localhost:3000/test%20/1)

[http://localhost:3000/test /1/2](http://localhost:3000/test%20/1/2)

http://localhost:3000/test /1/2/3

匹配规则：优先静态

路由跳转

页面跳转一般有两种形式，第一种是利用标签<Link>,第二种是用js编程的方式进行跳转，也就是利用Router组件。先来看一下标签的形式如何跳转。

**[标签式导航<Link>](https://jspang.com/detailed?id=51" \l "toc314)**

在编写代码之前，先删除index.js中的代码，保证代码的最小化。使用标签式导航需要先进行引入，代码如下:

import Link from 'next/link'

然后新建两个页面testA.js和testB.js，新建后写个最简单的页面，能标识出来A、B两个页面就好。

//testA.js

import Link from 'next/link'

export default ()=>(

<>

<div>Test-A page . </div>

<Link href="/"><a>返回首页</a></Link>

</>

)

写完A页面后，可以直接复制A页面的内容，然后修改一下就是B页面。

//testB.js

import Link from 'next/link'

export default ()=>(

<>

<div>Test-B page . </div>

<Link href="/"><a>返回首页</a></Link>

</>

)

有了两个页面后，可以编写首页的代码，实现跳转了。

//index.js

import React from 'react'

import Link from 'next/link'

const Home = () => (

<>

<div>我是首页</div>

<div><Link href="/testA"><a>去TestA页面</a></Link></div>

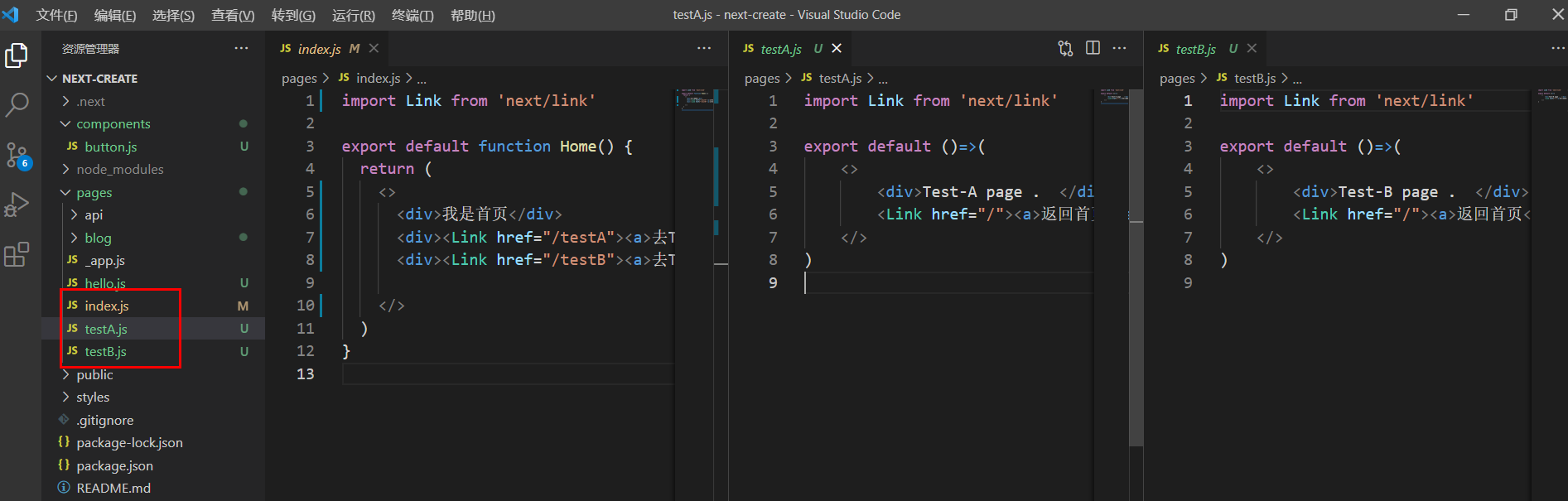
<div><Link href="/testB"><a>去TestB页面</a></Link></div>

</>

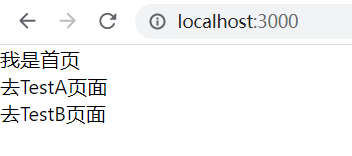
)

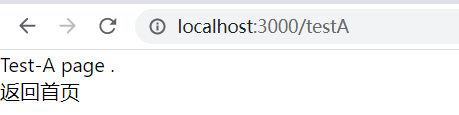
export default Home

当前项目截图：



浏览器中查看：





**[编程式跳转Router](https://jspang.com/detailed?id=51" \l "toc315)**

在Next框架中还可以使用Router模块进行编程式的跳转，使用前也需要我们引入Router，代码如下：

import Router from 'next/router'

然后在index.js页面中加入，直接使用Router进行跳转就可以了。

<div>

<button onClick={()=>{Router.push('/testA')}}>去TestA页面</button>

</div>

或

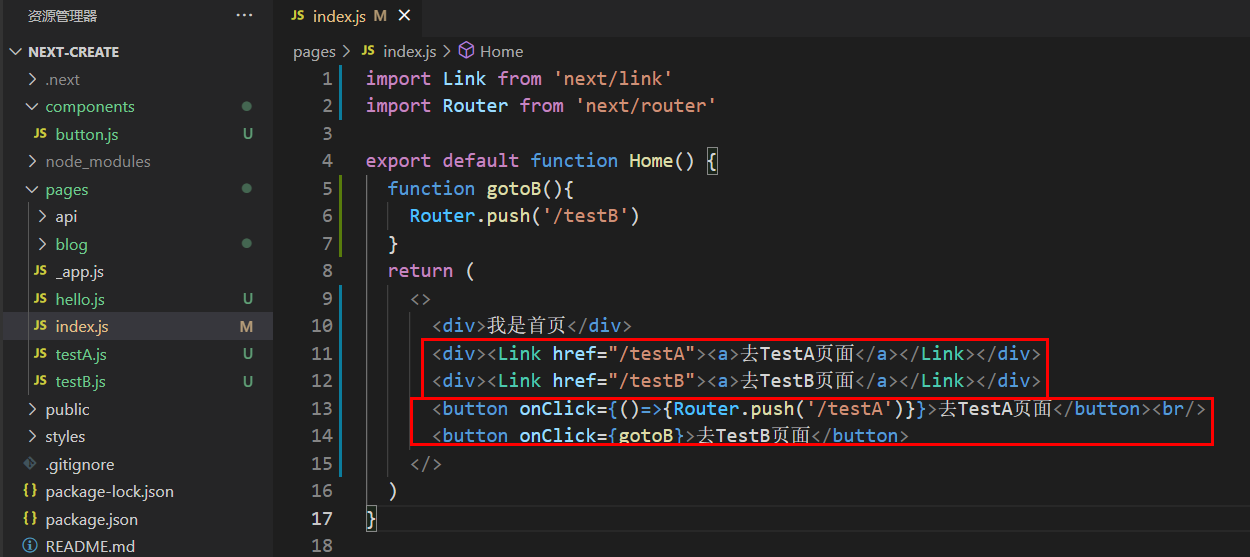
 function gotoB(){

    Router.push('/testB')

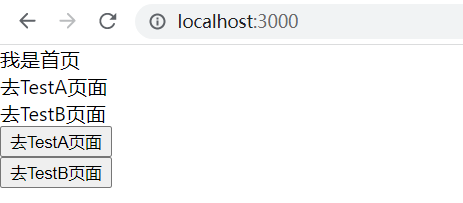
  }

<button onClick={gotoB}>去TestB页面</button>

此时项目截图：



浏览器中查看：



**[5：路由-跳转时用query传递和接受参数](https://jspang.com/detailed?id=51" \l "toc216)**

项目开发中一般都不是简单的静态跳转，而是需要动态跳转的。动态跳转就是跳转时需要带一个参数或几个参数过去，然后在到达的页面接受这个传递的参数，并根据参数不同显示不同的内容。比如新闻列表，然后点击一个要看的新闻就会跳转到具体内容。这些类似这样的需求都是通过传递参数实现的。

**[只能用query传递参数](https://jspang.com/detailed?id=51" \l "toc317)**

新闻的例子，通过这个例子来通俗易懂的讲解一下路由带参数的知识。在Next.js中只能通过通过query（?id=1）来传递参数。

现在我们改写一下pages文件夹下的index.js文件。

import Link from 'next/link'

export default function Home() {

  return (

    <>

      <h1>我是首页</h1>

      <div>

        <Link href="/news?id=1"><a>新闻1</a></Link><br/>

        <Link href="/news?id=2"><a>新闻2</a></Link>

      </div>

    </>

  )

}

这样编写query参数就可以进行传递过去了，接下来就是要接受参数了。

**[接收传递过来的参数](https://jspang.com/detailed?id=51" \l "toc318)**

现在还没有对应的页面，所以我们要创建news.js页面，并写下下面的代码。

import { withRouter} from 'next/router'

import Link from 'next/link'

const News = ({router})=>{

return (

<>

<div>id为{router.query.id}的新闻详情</div>

<Link href="/"><a>返回首页</a></Link>

</>

)

}

export default withRouter(News)

withRouter是Next.js框架的高级组件，用来处理路由用的，这里先学简单用法，以后还会学习的。通过这种方式就获得了参数，并显示在页面上了。

如果不使用withRouter,也可以使用useRouter接收参数

import { useRouter} from 'next/router'

import Link from 'next/link'

const News = ()=>{

    const router = useRouter();

    return (

        <>

            <div>id为{router.query.id}的新闻详情</div>

            <Link href="/"><a>返回首页</a></Link>

        </>

    )

}

export default News

**[编程式跳转传递参数](https://jspang.com/detailed?id=51" \l "toc319)**

编程式跳转也可以简单使用?加参数的形式，代码如下：

import Link from 'next/link'

import Router from 'next/router'

export default function Home() {

  function gotoNews1(){

    Router.push({

      pathname:'/news',

      query:{

        id:1

      }

    })

  }

  function gotoNews2(){

    Router.push('/news?id=2')

  }

  return (

    <>

      <h1>我是首页</h1>

      <div>

        <Link href="/news?id=1"><a>新闻1</a></Link><br/>

        <Link href="/news?id=2"><a>新闻2</a></Link><br/>

        <button onClick={gotoNews1}>新闻1</button><br/>

        <button onClick={gotoNews2}>新闻2</button>

      </div>

    </>

  )

}

**[6：路由-六个钩子事件的讲解](https://jspang.com/detailed?id=51" \l "toc220)**

路由的钩子事件，也就是当路由发生变化时，可以监听到这些变化事件，执行对应的函数。它一共有六个钩子事件，这节课就学习一下。

**[routerChangeStart路由发生变化时](https://jspang.com/detailed?id=51" \l "toc321)**

在监听路由发生变化时，我们需要用Router组件，然后用on方法来进行监听,在pages文件夹下的index.js，然后写入下面的监听事件，这里我们只打印一句话，就不作其他的事情了。代码如下：

Router.events.on('routeChangeStart',(...args)=>{

console.log('1.routeChangeStart->路由开始变化,参数为:',...args)

})

这个时路由发生变化时，第一时间被监听到，并执行了里边的方法。

**[routerChangeComplete路由结束变化时](https://jspang.com/detailed?id=51" \l "toc322)**

路由变化开始时可以监听到，那结束时也时可以监听到的，这时候监听的事件是routerChangeComplete。

Router.events.on('routeChangeComplete',(...args)=>{

console.log('routeChangeComplete->路由结束变化,参数为:',...args)

})

**[beforeHistoryChange浏览器history触发前](https://jspang.com/detailed?id=51" \l "toc323)**

history就是HTML中的API，如果这个不了解可以百度了解一下，Next.js路由变化默认都是通过history进行的，所以每次都会调用。 不适用history的话，也可以通过hash

Router.events.on('beforeHistoryChange',(...args)=>{

console.log('3,beforeHistoryChange->在改变浏览器 history之前触发,参数为:',...args)

})

**[routeChangeError路由跳转发生错误时](https://jspang.com/detailed?id=51" \l "toc324)**

Router.events.on('routeChangeError',(...args)=>{

console.log('4,routeChangeError->跳转发生错误,参数为:',...args)

})

需要注意的是404找不到路由页面不算错误。

**[转变成hash路由模式](https://jspang.com/detailed?id=51" \l "toc325)**

还有两种事件，都是针对hash的，所以现在要转变成hash模式。hash模式下的两个事件hashChangeStart和hashChangeComplete,就都在这里进行编写了。

Router.events.on('hashChangeStart',(...args)=>{

console.log('5,hashChangeStart->hash跳转开始时执行,参数为:',...args)

})

Router.events.on('hashChangeComplete',(...args)=>{

console.log('6,hashChangeComplete->hash跳转完成时,参数为:',...args)

})

在下面的jsx语法部分，再增加一个链接,使用hash来进行跳转，代码如下：

<div>

<Link href="#test"><a>测试</a></Link>

</div>

当前项目代码：index.js

import Link from 'next/link'

import Router from 'next/router'

export default function Home() {

  Router.events.on('routeChangeStart',(...args)=>{

    console.log('1.routeChangeStart->路由开始变化,参数为:',...args)

  })

  Router.events.on('routeChangeComplete',(...args)=>{

    console.log('2.routeChangeComplete->路由结束变化,参数为:',...args)

  })

  Router.events.on('beforeHistoryChange',(...args)=>{

    console.log('3,beforeHistoryChange->在改变浏览器 history之前触发,参数为:',...args)

  })

  Router.events.on('routeChangeError',(...args)=>{

    console.log('4,routeChangeError->跳转发生错误,参数为:',...args)

  })

  Router.events.on('hashChangeStart',(...args)=>{

    console.log('5,hashChangeStart->hash跳转开始时执行,参数为:',...args)

  })

  Router.events.on('hashChangeComplete',(...args)=>{

    console.log('6,hashChangeComplete->hash跳转完成时,参数为:',...args)

  })

  function gotoNews1(){

    Router.push({

      pathname:'/news',

      query:{

        id:1

      }

    })

  }

  function gotoNews2(){

    Router.push('/news?id=2')

  }

  return (

    <>

      <h1>我是首页</h1>

      <div>

        <Link href="/news?id=1"><a>新闻1</a></Link><br/>

        <Link href="/news?id=2"><a>新闻2</a></Link><br/>

        <button onClick={gotoNews1}>新闻1</button><br/>

        <button onClick={gotoNews2}>新闻2</button><br/>

        <Link href="#test"><a>测试</a></Link>

      </div>

      <div style={{width:'100%',height:'1000px'}}>

      </div>

      <div id='test' style={{width:'100%',height:'100px'}}>

        显示这里

     </div>

    </>

  )

}

路由的钩子事件，利用钩子事件是可以作很多事情的，比如转换时的加载动画，关掉页面的一些资源计数器.....。



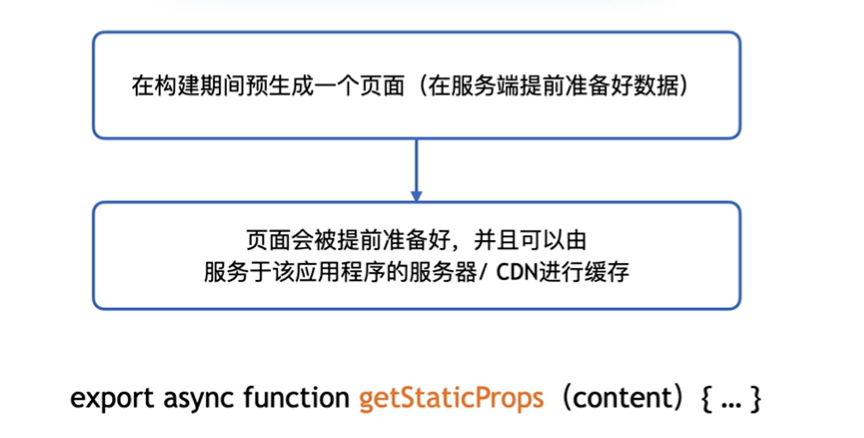




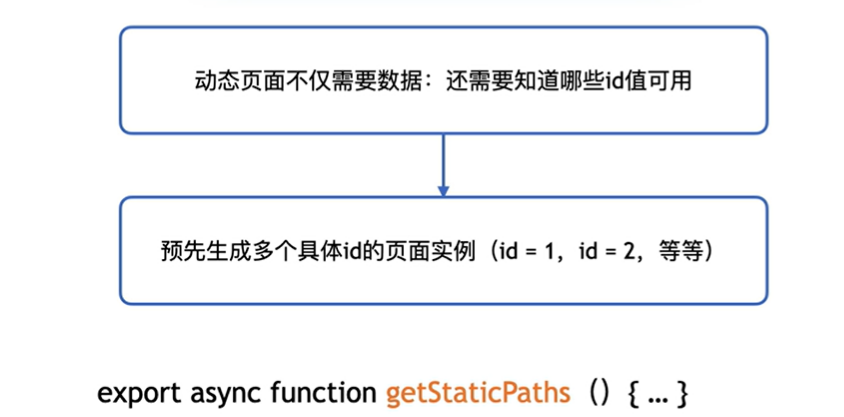
**[7\*：预渲染和数据获取](https://jspang.com/detailed?id=51" \l "toc226)**

在Next.js框架中提供了getInitialProps静态方法用来获取远端数据，这个是框架的约定，所以你也只能在这个方法里获取远端数据。

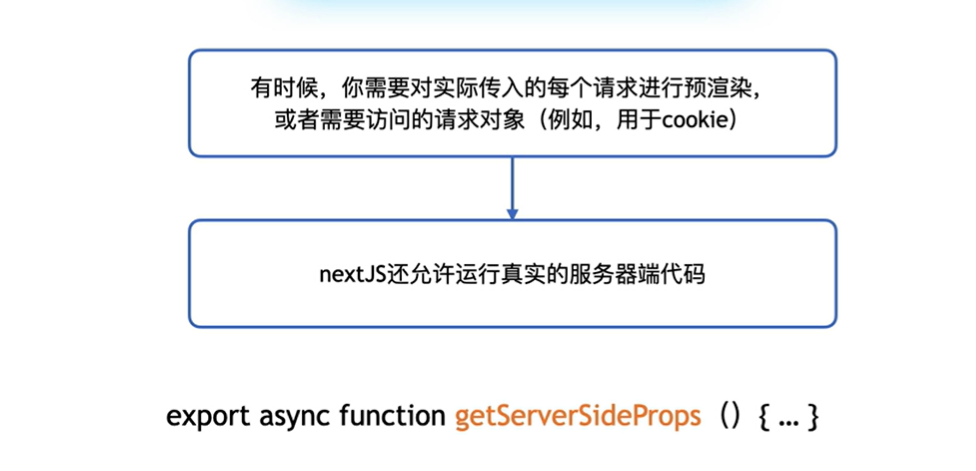
getStaticProps (Static Generation): Fetch data at build time.



getStaticPaths (Static Generation): Specify dynamic routes to pre-render pages based on data.



getServerSideProps (Server-side Rendering): Fetch data on each request.



**[静态渲染](https://jspang.com/detailed?id=51" \l "toc327)**

根目录下新建一个data文件夹，并新建product.json，用来模拟动态数据

{

    "data": [

        {

            "id": "1",

            "name": "产品1",

            "desc": "产品1的详细内容"

        },

        {

            "id": "2",

            "name": "产品2",

            "desc": "产品2的详细内容"

        },{

            "id": "3",

            "name": "产品3",

            "desc": "产品3的详细内容"

        }

    ]

}

在pages文件夹下新建product文件夹，文件夹中新建index.js，使用静态生成 getStaticProps获取数据，并进行显示；

import fs from 'fs/promises'

import path from 'path'

function ProductPage(props){

    const { products } = props;

    console.log(products);

    return (

        <>

<h1>产品列表页</h1>

        <ul>

            {

                products.map(product => {

                    return (<li key={product.id}>{product.name}</li>)

                })

            }

        </ul>

        </>

    )

}

// 静态生成 getStaticProps (Static Generation): Fetch data at build time.

export async function getStaticProps(){

    // process.cwd() 获取根目录

    const filePath = path.join(process.cwd(),'data','product.json')

    const jsonData = await fs.readFile(filePath);

    const products = JSON.parse(jsonData).data;

    return {

        props:{

            // products:[{id:1,name:'产品1'}]

            products

        }

    }

}

export default ProductPage

浏览器查看：



备注：增量静态生成以及其他参数

// 告诉服务器需要间隔多长时间重新渲染

        revalidate:10,

        // 无数据时，显示404

        notFound: false,

        // 重定向

        redirect:{

            destination:'/'

        }

渲染路径

当前我们只使用了静态生成渲染了列表页，接下来我们给列表页中的产品添加链接，点击查看详情。

pages/product/index.js 添加链接

import Link from 'next/link'

//跳转部分

return (<li key={product.id}><Link href={`product/${product.id}`}>{product.name}</Link></li>)

pages/product目录下新建文件[productId].js

import fs from 'fs/promises'

import path from 'path'

import Link from 'next/link'

function ProductDetailPage(props) {

    const { product } = props

    return (

        <>

            <h1>产品列表页</h1>

            <h2>{product.name}</h2>

            <p>{product.desc}</p>

            <br />

            <Link href='/product'>返回产品列表页</Link>

        </>

    )

}

// 从json文件读取数据

async function getData() {

    // process.cwd() 获取根目录

    const filePath = path.join(process.cwd(), 'data', 'product.json')

    const jsonData = await fs.readFile(filePath);

    const products = JSON.parse(jsonData).data;

    return products;

}

// 获取单个路径下页面使用的数据

export async function getStaticProps(context) {

    // 获取产品id

    const { params: { productId } } = context;

    const products = await getData();

    const product = products.find(pro => pro.id === productId);

    return {

        props: {

            // product:[{id:1,name:'产品1',desc:'产品1的详细内容'}]

            product

        },

        // 告诉服务器需要间隔多长时间重新渲染

        revalidate: 10

    }

}

// 获取所以路径

export async function getStaticPaths() {

    // 从json文件获取所有id

    const products = await getData();

    const ids = products.map(product => product.id);

    // 处理为paths需要的格式

    const pathWithParams = ids.map(id => ({ params: { productId: id } }));

    // [{params:{productId:'1'}},{params:{productId:'2'}}]

    return {

        paths: pathWithParams,

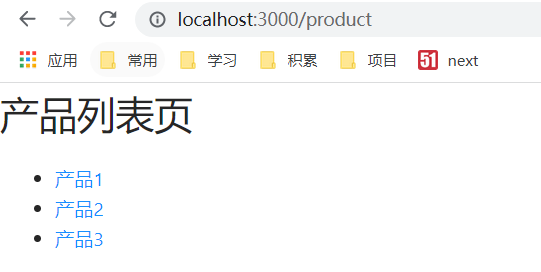
        fallback: false,

    }

}

export default ProductDetailPage

浏览器中查看





备注：

fallback:

false: 预渲染所有

true: 自定义预渲染，配合加载器使用(如电商平台产品多时)

‘blocking’:服务器端渲染

服务器端渲染

我们将使用一个用户个人信息展示的页面学习服务端渲染，因为个人信息等应该在用户请求时再返回，不能预渲染，预渲染会泄露信息。

page目录下新建user文件夹，index.js

function PersonInfoPage(props){

    const { name } =  props;

    return (

        <>

        用户名：{ name }

        </>

    )

}

// 用户每次发送请求，都会执行

export async function getServerSideProps(context){

  // 可以获取到的信息

  const { params,req, res } = context;

  return {

      props:{

          name: 'Wenda'

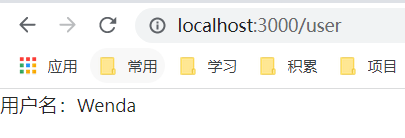
      }

  }

}

export default PersonInfoPage;

浏览器中查看



若是动态路由，则可以使用params去获取动态路由中的参数

page/user文件夹，[uuid].js

function PersonId(props){

    const {id} = props;

    return (

        <>

        用户Id:{id}

        </>

    )

}

// 用户每次发送请求，都会执行

export async function getServerSideProps(context){

    // 可以获取到的信息

    const { params,req, res } = context;

    const { uuid } = params;

    return {

        props:{

            id: uuid

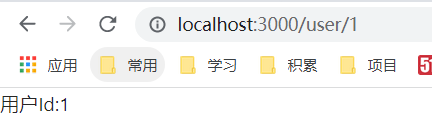
        }

    }

  }

export default PersonId

浏览器中查看：



备注：尽管生产环境三个方法的执行时机不同，但是再开发模式下，每次请求都会执行。

客户端渲染:

当页面的数据更新速度较快如股票信息，也就是说页面上动态加载的信息，就需要使用客户端渲染

我们准备实现一个显示用户信息的例子，来学习客户端渲染。

预先准备一个api:

page/api目录下新建users-info.js

export default (req, res) => {

  const data = [

    {

      "userId": 1,

      "id": 1,

      "name": "Wenda01",

      "desc": "Test Wenda01"

    },

    {

      "userId": 1,

      "id": 2,

      "name": "Wenda02",

      "desc": "Test Wenda02"

    },

  ];

  res.status(200).json(data)

}

安装swr，这是一个用于数据请求的 React Hooks 库, 使用 SWR，组件将会不断地、自动获得最新数据流。UI 也会一直保持快速响应。

npm install --save swr

page目录下，新建usersInfo.js

import React from "react";

import useSWR from "swr";

const fetcher = (url) => fetch(url).then((res) => res.json());

function UsersInfoPage() {

  const { data, error } = useSWR("/api/users-info", fetcher);

  if (error) return "An error has occurred.";

  if (!data) return "Loading...";

  return (

    <>

    <h1>最新用户信息</h1>

    <ul>

        {

          data.map(user=>{

             return (

                  <li key={user.id}>

                  <span>用户：{user.name}</span>

                  <span>详情：{user.desc}</span>

              </li>

             )

          })

        }

    </ul>

    </>

  );

}

export default UsersInfoPage

浏览器中查看



**[8：使用Style JSX编写页面的CSS样式](https://jspang.com/detailed?id=51" \l "toc229)**

在Next.js中引入一个CSS样式是不可以用的，如果想用，需要作额外的配置。因为框架为我们提供了一个style jsx特性，也就是把CSS用JSX的语法写出来。

**[初识Style JSX语法 把字体设成蓝色](https://jspang.com/detailed?id=51" \l "toc330)**

在pages文件夹下，新建一个test.js文件。然后写入下面的代码：

//test.js

function Test(){

return (

<>

<div>CSS文字颜色演示</div>

</>

)

}

export default Test

这个是一个最简单的页面，只在层中写了一句话。这时候我们想把页面中字的颜色变成蓝色，就可以使用Style JSX语法。直接在<></>之间写下如下的代码:

<style jsx>

{`

div{color:blue;}

`}

</style>

主要所有的css样式需要用{}进行包裹，否则就会报错。这时候你打开浏览器进行预览，字体的颜色就变成了蓝色。

**[自动加随机类名 不会污染全局CSS](https://jspang.com/detailed?id=51" \l "toc331)**

加入了Style jsx代码后，Next.js会自动加入一个随机类名，这样就防止了CSS的全局污染。比如我们把代码写成下面这样，然后在浏览器的控制台中进行查看，你会发现自动给我们加入了类名，类似jsx-xxxxxxxx。

function Test() {

    return (

        <>

            <div>CSS文字颜色演示</div>

            <style jsx>

                {`

                 div{color:blue;}

               `}

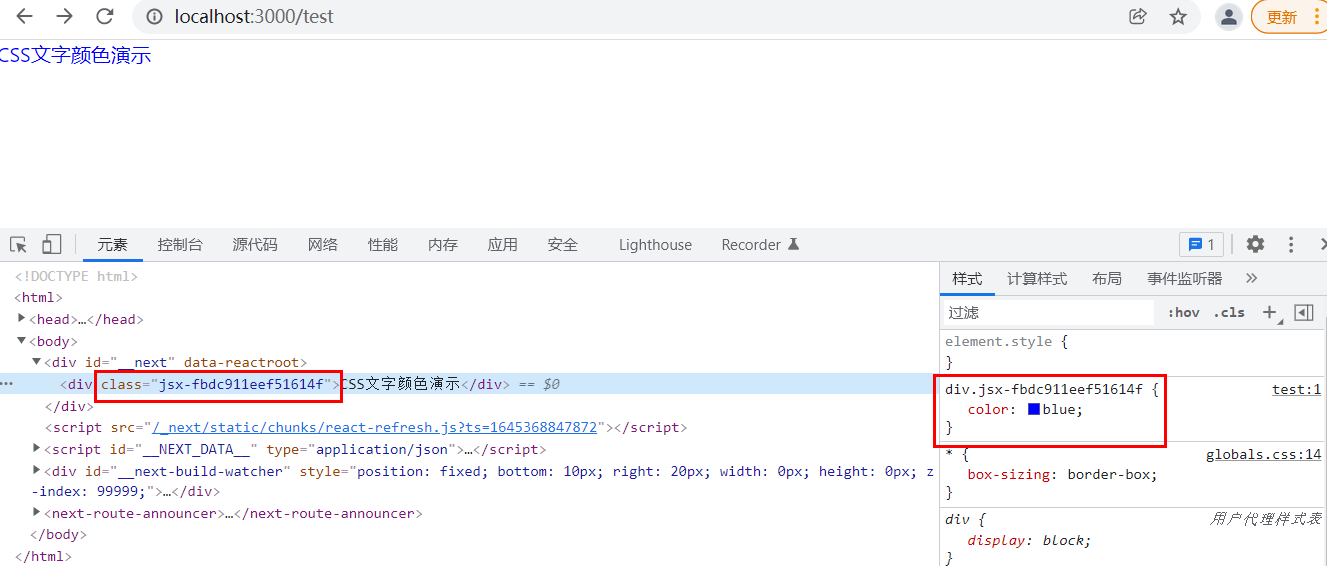
            </style>

        </>

    )

}

export default Test



**[动态显示样式](https://jspang.com/detailed?id=51" \l "toc332)**

Next.js使用了Style jsx,所以定义动态的CSS样式就非常简单，比如现在要作一个按钮，点击一下，字体颜色就由蓝色变成了红色。下面是实现代码。

import { useState } from "react"

function Test() {

    const [color, setColor] = useState('blue');

    function changeColor(){

        setColor(color === 'red' ? 'blue' : 'red')

    }

    return (

        <>

            <div>CSS文字颜色演示</div>

            <button onClick={changeColor}>点击改变文字颜色</button>

            <style jsx>

                {`

                 div{color:${color};}

               `}

            </style>

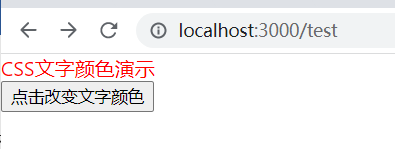
        </>

    )

}

export default Test

这样就完成了CSS的动态显示。



**[9：Lazy Loading实现模块懒加载](https://jspang.com/detailed?id=51" \l "toc233)**

当项目越来越大的时候，模块的加载是需要管理的，如果不管理会出现首次打开过慢，页面长时间没有反应一系列问题。这时候可用Next.js提供的LazyLoading来解决这类问题。让模块和组件只有在用到的时候在进行加载，一般我把这种东西叫做“懒加载”.它一般分为两种情况，一种是懒加载（或者说是异步加载）模块，另一种是异步加载组件。

**[懒加载模块](https://jspang.com/detailed?id=51" \l "toc334)**

这里使用一个在开发中常用的模块Moment.js，它是一个JavaScript日期处理类库。

npm install --save moment

然后在pages文件夹下，新建立一个time.js文件，并使用刚才的moment库来格式化时间，代码如下:

import React, {useState} from 'react'

import moment from 'moment'

function Time(){

const [nowTime,setTime] = useState(Date.now())

const changeTime=()=>{

setTime(moment(Date.now()).format())

}

return (

<>

<div>显示时间为:{nowTime}</div>

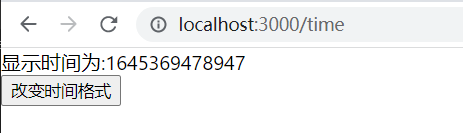
<div><button onClick={changeTime}>改变时间格式</button></div>

</>

)

}

export default Time



这个看起来很简单和清晰的案例，缺存在着一个潜在的风险，就是如何有半数以上页面使用了这个momnet的库，那它就会以公共库的形式进行打包发布，就算项目第一个页面不使用moment也会进行加载，这就是资源浪费，对于我这样有代码洁癖的良好程序员是绝对不允许的。下面我们就通过Lazy Loading来进行改造代码。

import React, {useState} from 'react'

//删除import moment

function Time(){

const [nowTime,setTime] = useState(Date.now())

const changeTime= async ()=>{ //把方法变成异步模式

const moment = await import('moment') //等待moment加载完成

setTime(moment.default(Date.now()).format()) //注意使用defalut

}

return (

<>

<div>显示时间为:{nowTime}</div>

<div><button onClick={changeTime}>改变时间格式</button></div>

</>

)

}

export default Time

这时候就就是懒加载了，可以在浏览器中按F12，看一下Network标签，当我们点击按钮时，才会加载它就是momnet.js的内容。



**[懒加载自定义组件](https://jspang.com/detailed?id=51" \l "toc335)**

在components文件夹下建立一个one.js文件，然后编写如下代码：

export default ()=><div>Lazy Loading Component</div>

有了自定义组件后，先要在懒加载这个组件的文件中引入dynamic,我们这个就在上边新建的time.js文件中编写了。

import dynamic from 'next/dynamic'

引入后就可以懒加载自定义模块了，代码如下：

import React, {useState} from 'react'

import dynamic from 'next/dynamic'

const One = dynamic(import('../components/one'))

function Time(){

    const [nowTime,setTime] = useState(Date.now())

    const changeTime= async ()=>{

        const moment = await import('moment');

        setTime(moment.default(Date.now()).format())

    }

    return (

        <>

            <One />

            <div>显示时间为:{nowTime}</div>

            <div><button onClick={changeTime}>改变时间格式</button></div>

        </>

    )

}

export default Time

写完代码后，可以看到自定义组件是懒加载的，只有在jsx里用到<One/>时，才会被加载进来，如果不使用就不会被加载。

当我们作的应用存在首页打开过慢和某个页面加载过慢时，就可以采用Lazy Loading的形式，用懒加载解决这些问题。

**[10：自定义Head 更加友好的SEO操作](https://jspang.com/detailed?id=51" \l "toc236)**

既然用了Next.js框架，你就是希望服务端渲染，进行SEO操作。那为了更好的进行SEO优化，可以自己定制<Head>标签，定义<Head>一般有两种方式。

**[方法1：在各个页面加上<Head>标签](https://jspang.com/detailed?id=51" \l "toc337)**

先在/pages文件夹下面建立一个header.js文件，然后写一个最简单的Hooks页面，代码如下:

function Header(){

return (<div>Test.com</div>)

}

export default Header

写完后到浏览器中预览一下，可以发现title部分并没有任何内容，显示的是localhost:3000/header,接下来就自定义下<Head>。自定义需要先进行引入next/head。

import Head from 'next/head'

全部代码如下:

import Head from 'next/head'

function Header(){

return (

<>

<Head>

<title>我是title </title>

<meta charSet='utf-8' />

</Head>

<div>Test.com</div>

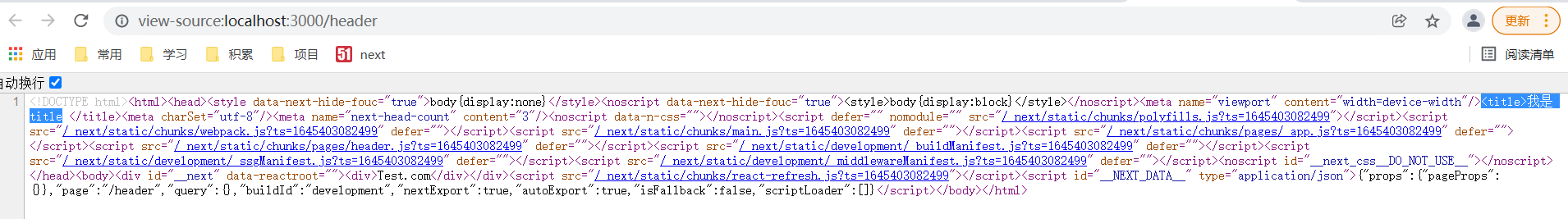
</>

)

}

export default Header

这时候再打开浏览器预览，你发现已经有了title。



**[方法2：定义全局的<Head>](https://jspang.com/detailed?id=51" \l "toc338)**

这种方法相当于自定义了一个组件，然后把<Head>在组件里定义好，以后每个页面都使用这个组件,其实这种方法用处不大，也不灵活。因为Next.js已经把<Head>封装好了，本身就是一个组件，我们再次封装的意义不大。

比如在components文件夹下面新建立一个myheader.js,然后写入下面的代码:

import Head from 'next/head'

const MyHeader = ()=>{

return (

<>

<Head>

<title> test.com </title>

</Head>

</>

)

}

export default MyHeader

这时候把刚才编写的header.js页面改写一下，引入自定义的myheader，在页面里进行使用，最后在浏览器中预览，也是可以得到title的。

// import Head from 'next/head'

import Myheader from '../components/myheader'

function Header(){

    return (

        <>

            {/\* <Head>

                <title>我是title </title>

                <meta charSet='utf-8' />

            </Head> \*/}

            <Myheader />

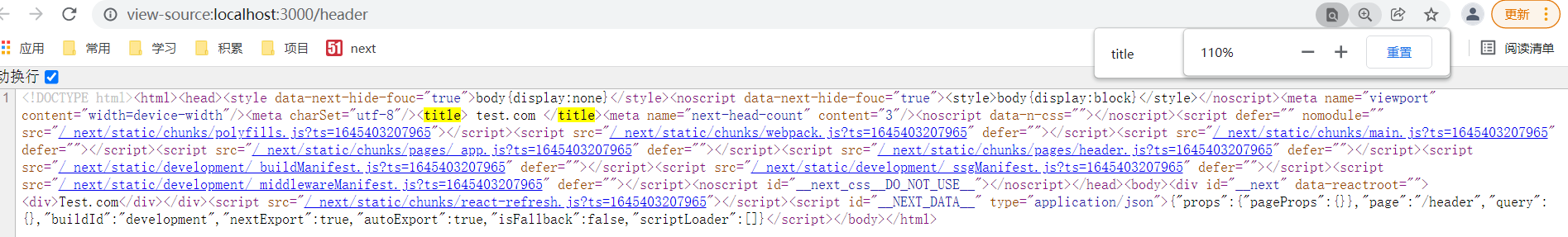
            <div>Test.com</div>

        </>

    )

}

export default Header



**[11：Next.js框架下使用Ant Design UI](https://jspang.com/detailed?id=51" \l "toc239)**

在Next.js框架下使用 Ant Design，Ant Design是一款阿里开源的前端组件库，用来搭建项目的中后台系统，比如后台的管理系统。从React的角度来讲，它就是一个组件库，里边封装了开发中最常用的一些组件，让我们可以通过简单的配置就可以使用他们。

**安装antd并使用**

npm install --save antd

安装成功后，pages文件夹下新建文件antdTest.js，引入<Button>组件，并进行使用，代码如下。

import Myheader from '../components/myheader'

import {Button} from 'antd'

function Header(){

    return (

        <>

            <Myheader />

            <div>Test.com</div>

            <div><Button>我是按钮</Button></div>

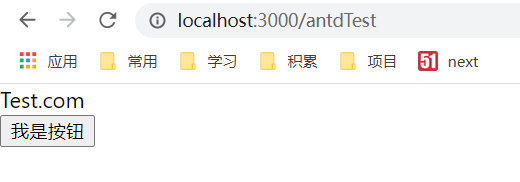
        </>

    )

}

export default Header

然后到浏览器中查看一下结果，这时候Ant Design的按钮虽然显示出来了，但是没有样式。



**[全局样式](https://jspang.com/detailed?id=51" \l "toc340)**

pages/ \_app.js文件里引入全局样式，打开浏览器，样式显示正常。

import '../styles/globals.css'

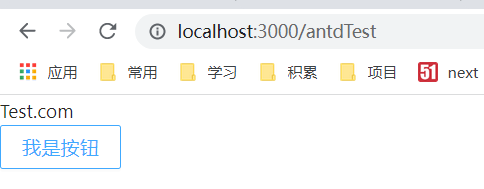
import 'antd/dist/antd.css'

function MyApp({ Component, pageProps }) {

  return <Component {...pageProps} />

}

export default MyApp



在生产环境中，所有 CSS 文件将自动合并为一个经过精简的 .css 文件

[**添加组件级 CSS**](https://www.nextjs.cn/docs/basic-features/built-in-css-support#%E6%B7%BB%E5%8A%A0%E7%BB%84%E4%BB%B6%E7%BA%A7-css)

Next.js 通过 [name].module.css 文件命名约定来支持 [CSS 模块](https://github.com/css-modules/css-modules) 。

CSS 模块通过自动创建唯一的类名从而将 CSS 限定在局部范围内。 这使您可以在不同文件中使用相同的 CSS 类名，而不必担心冲突。

此行为使 CSS 模块成为包含组件级 CSS 的理想方法。 CSS 模块文件 **可以导入（import）到应用程序中的任何位置**。

components中改变button.js

import styles from './button.module.css'

export default ({children})=><button className={styles.error}>{children}</button>

并新建并引入button.module.css

.error {

    border: 2px solid red;

    background-color: pink;

}

pages下新建buttonTest.js

import Button from '../components/button'

function TestButton(){

    return (

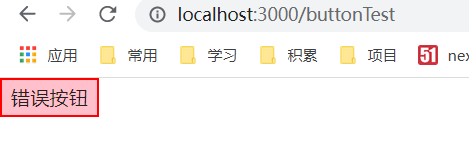
        <Button>错误按钮</Button>

    )

}

export default TestButton

浏览器中查看



**[12\*api路由](https://jspang.com/detailed?id=51" \l "toc242)**

**api路由**

pages/api 目录下的任何文件都将作为 API 端点映射到 /api/\*，而不是 page。这些文件只会增加服务端文件包的体积，而不会增加客户端文件包的大小。

一个通常的 RESTful 模式将设置以下路由：

* GET api/posts - 获取 post 列表，可能还有分页
* GET api/posts/12345 - 获取 id 为 12345 的 post

我们可以按照如下两种方式建模：

* 方案 1：
  + /api/posts.js
  + /api/posts/[postId].js
* 方案 2：
  + /api/posts/index.js
  + /api/posts/[postId].js

**静态api路由**

简单的路由：pages/api/users-info.js

export default (req, res) => {

  const data = [

    {

      "userId": 1,

      "id": 1,

      "name": "Wenda01",

      "desc": "Test Wenda01"

    },

    {

      "userId": 1,

      "id": 2,

      "name": "Wenda02",

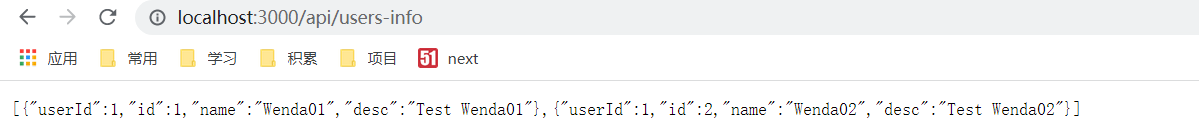
      "desc": "Test Wenda02"

    }

  ];

  res.status(200).json(data)

}



支持增删改查的路由：

pages/api/目录下新建task文件夹，新建index.js

import fs from 'fs';

import path from 'path';

import { sortBy } from 'lodash';

function getFilePath(fileName){

   return path.join(process.cwd(),'data',fileName);

}

export function readJson(fileName){

   const filePath = getFilePath(fileName);

   const fileData = fs.readFileSync(filePath);

   const data = JSON.parse(fileData);

   console.log(data)

   return data;

}

function wirteJson(fileName, data){

    const filePath = getFilePath(fileName);

    fs.writeFileSync(filePath,JSON.stringify(data))

}

function handle(req,res){

   const fileName = 'task.json';

   let status = 200;

   let data = readJson(fileName);

   let message = '操作成功';

   // 查询

    if (req.method === 'GET') {

        res.status(status).json({ data, message });

    } else if(req.method === 'POST'){

        // 增加

        let taskObj = req.body;

        taskObj.id = new Date().getTime().toString();

        data.push(taskObj);

        wirteJson(fileName,sortBy(data, 'id'));

        res.status(status).json({ taskObj, message });

    } else if(req.method === 'PUT'){

        // 修改

        let taskObj = req.body;

        const result = data.filter(task=>{

            return task.id !== taskObj.id;

        });

        result.push(taskObj);

        wirteJson(fileName,result);

        res.status(status).json({ taskObj, message });

    } else if(req.method === 'DELETE'){

        // 删除

        let taskObj = req.body;

        const result = data.filter(task=>{

            return task.id !== taskObj.id;

        });

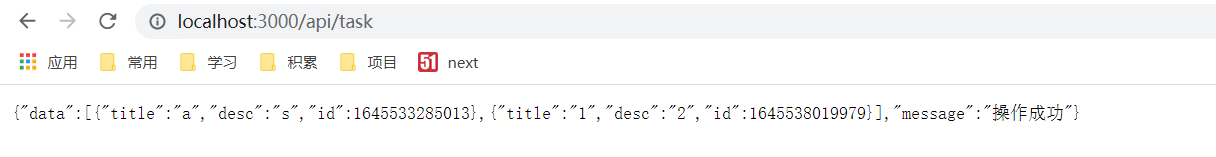
        wirteJson(fileName,result);

        res.status(status).json({ message });

    }

}

export default handle;



**动态api路由**

pages/api/task文件夹下，新建[taskId].js

import {readJson} from './index'

export default (req, res) => {

    // 获取api路由中的参数

    const id = req.query.taskId;

    const data = readJson('task.json');

    let task = {}

    const tasks  = data.filter(task=>{

        return task.id === id;

    })

    if(tasks.length>0){

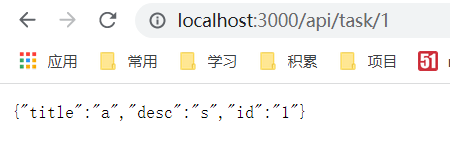
        task = tasks[0];

    }

    res.status(200).json(task)

  }

浏览器中查看



**实例**

预渲染

<http://localhost:3000/task/taskStatic>

客户端请求：

http://localhost:3000/task

**[13：Next.js生产环境打包](https://jspang.com/detailed?id=51" \l "toc242)**

再回头看一下我们的package.json

{

  "name": "next-create",

  "version": "0.1.0",

  "private": true,

  "scripts": {

    "dev": "next dev",

    "build": "next build",

    "start": "next start"

  },

  "dependencies": {

    "antd": "^4.18.7",

    "moment": "^2.29.1",

    "next": "12.1.0",

    "react": "17.0.2",

    "react-dom": "17.0.2"

  }

}

实际运行的就是build和start命令，我是将代码提交至GitHub，然后使用

Vercel进行部署，得到一个线上可以访问的环境