同构应用中你所忽略的细节

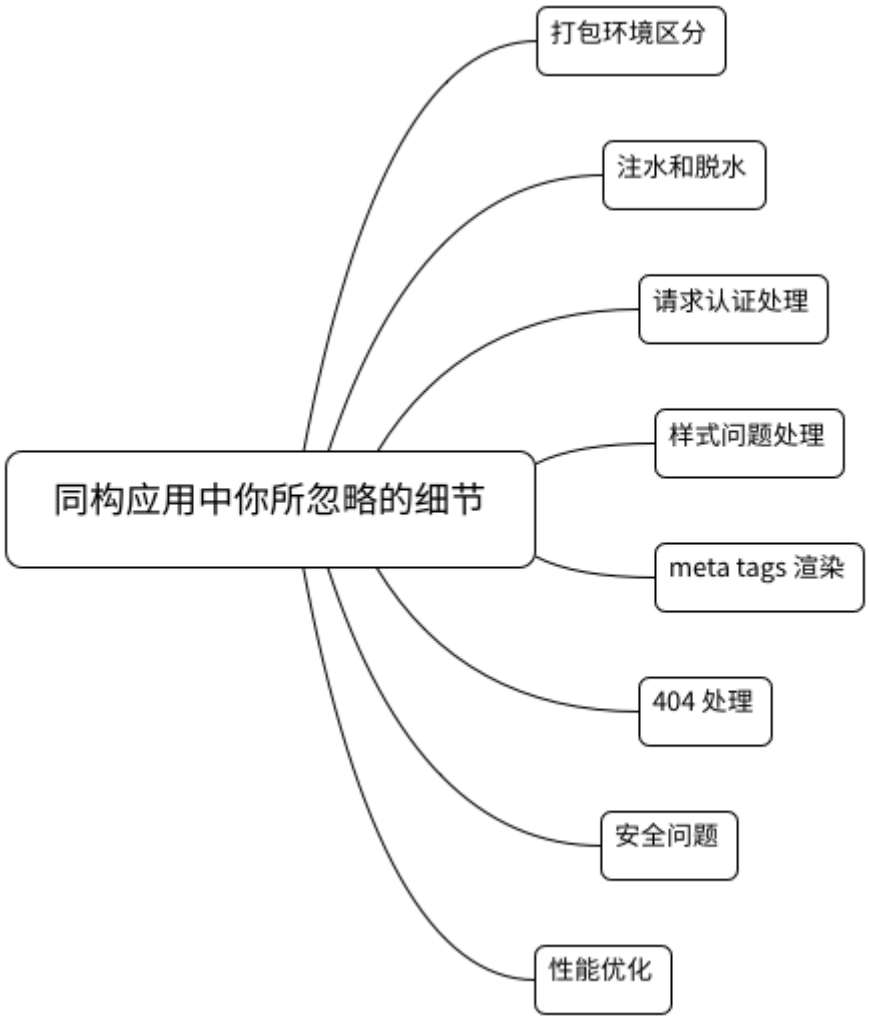
不管是服务端渲染还是服务端渲染衍生出的同构应用，现在来看已经并不新鲜了，实现起来也并不困难。可是有的开发者认为：同构应用不就是调用一个 renderToString（React 中）类似的 API 吗？

讲道理确实是这样的，但是讲道理你也许并没有真正在实战中领会同构应用的精髓。

同构应用能够完成的本质条件是虚拟 DOM，基于虚拟 DOM 我们可以生成真实的 DOM，并由浏览器渲染；也可以调用不同框架的不同 APIs，将虚拟 DOM 生成字符串，由服务端传输给客户端。

但是同构应用也不只是这么简单。拿面试来说，同构应用的考察点不是「纸上谈兵」的理论，而是实际实施时的细节。这一讲我们就来聊一聊「同构应用中往往被忽略的细节」，需要读者提前了解服务端渲染和同构应用的概念。

相关知识点如下：



打包环境区分

第一个细节：我们知道同构应用实现了客户端代码和服务端代码的基本统一，我们只需要编写一种组件，就能生成适用于服务端和客户端的组件案例。可是你是否知道，服务端代码和客户端代码大多数情况下还是需要单独处理？比如：

> 路由代码差别：服务端需要根据请求路径，匹配页面组件；客户端需要通过浏览器中的地址，匹配页面组件。

客户端代码：

```
const App = () => {
 return (
```

```
 )
}
ReactDom.render(, document.querySelector('#root'))
```

BrowserRouter 组件根据 window.location 以及 history API 实现页面切换，而服务端肯定是无法获取 window.location 的，服务端代码如下：

```
const App = () => {
 return




}
Return ReactDom.renderToString()
```

需要使用 StaticRouter 组件，并将请求地址和上下文信息作为 location 和 context 这两个 props 传入 StaticRouter 中。

打包差别：服务端运行的代码如果需要依赖 Node 核心模块或者第三方模块，就不再需要把这些模块代码打包到最终代码中了，因为环境已经安装这些依赖，可

以直接引用。这样一来，就需要我们在 webpack 中配置：target：node，并借助 webpack-node-externals 插件，解决第三方依赖打包的问题。

对于图片等静态资源，url-loader 会在服务端代码和客户端代码打包过程中分别被引用，因此会在资源目录中生成了重复的文件。当然后打包出来的因为重名，会覆盖前一次打包出来的结果，并不影响使用，但是整个构建过程并不优雅。

由于路由在服务端和客户端的差别，因此 webpack 配置文件的 entry 会不相同：

```
{
   entry: './src/client/index.js',
}

{
   entry: './src/server/index.js',
}
```

注水和脱水

什么叫做注水和脱水呢？这个和同构应用中数据的获取有关：在服务器端渲染时，首先服务端请求接口拿到数据，并处理准备好数据状态（如果使用 Redux，就是进行 store 的更新），为了减少客户端的请求，我们需要保留住这个状态。一般做法是在服务器端返回 HTML 字符串的时候，将数据 JSON.stringify 一并返回，这个过程，叫做脱水（dehydrate）；在客户端，就不再需要进行数据的请求了，可以直接使用服务端下发下来的数据，这个过程叫注水（hydrate）。用代码来表示：

服务端：

```
ctx.body = `
```

```
        // ...
```

`

客户端：

```
export const getClientStore = () => {
 const defaultState = JSON.parse(window.context.state)
 return createStore(reducer, defaultState,
applyMiddleware(thunk))
}
```

这一系列过程非常典型，但是也会有几个细节值得探讨：**在服务端渲染时，服务端如何能够请求所有的 APIs，保障数据全部已经请求呢?**

一般有两种方法：

> react-router 的解决方案是配置路由 route-config，结合 matchRoutes，找到页面上相关组件所需的请求接口的方法并执行请求。这就要求开发者通过路由配置信息，显式地告知服务端请求内容。

我们首先配置路由：

```
const routes = [
 {
   path: "/",
   component: Root,
   loadData: () => getSomeData()
 }
 // etc.
]

import { routes } from "./routes"

function App() {
 return (

     {routes.map(route => (

     ))}

 )
}
```

在服务端代码中：

```
import { matchPath } from "react-router-dom"

const promises = []
routes.some(route => {
 const match = matchPath(req.path, route)
 if (match) promises.push(route.loadData(match))
 return match
})

Promise.all(promises).then(data => {
 putTheDataSomewhereTheClientCanFindIt(data)
})
```

另外一种思路类似 Next.js，我们需要在 React 组件上定义静态方法。比如定义静态 loadData 方法，在服务端渲染时，我们可以遍历所有组件的 loadData，获取需要请求的接口。这样的方式借鉴了早期 React-apollo 的解决方案，我个人很喜欢这种设计。这里贴出我为 Facebook 团队 react-apollo 开源项目贡献的改动代码，其目的就是遍历组件，获取请求接口：

```
function getPromisesFromTree({
 rootElement,
 rootContext = {},
}: PromiseTreeArgument): PromiseTreeResult[] {
 const promises: PromiseTreeResult[] = [];

 walkTree(rootElement, rootContext, (_, instance, context,
childContext) => {
   if (instance && hasFetchDataFunction(instance)) {
     const promise = instance.fetchData();
     if (isPromise(promise)) {
       promises.push({ promise, context: childContext ||
context, instance });
       return false;
     }
   }
 });

 return promises;
}

// Recurse a React Element tree, running visitor on each
element.
// If visitor returns `false`, don't call the element's
render function
// or recurse into its child elements.
export function walkTree(
 element: React.ReactNode,
 context: Context,
 visitor: (
```

```
    instance: React.Component | null,
    context: Context,
    childContext?: Context,
  ) => boolean | void,
) {
  if (Array.isArray(element)) {
    element.forEach(item => walkTree(item, context,
visitor));
    return;
  }

  if (!element) {
    return;
  }

  // A stateless functional component or a class
  if (isReactElement(element)) {
    if (typeof element.type === 'function') {
      const Comp = element.type;
      const props = Object.assign({}, Comp.defaultProps,
getProps(element));
      let childContext = context;
      let child;

      // Are we are a react class?
      if (isComponentClass(Comp)) {
        const instance = new Comp(props, context);
        // In case the user doesn't pass these to super in
the constructor.
        // Note: `Component.props` are now readonly in
`@types/react`, so
        // we're using `defineProperty` as a workaround
(for now).
        Object.defineProperty(instance, 'props', {
          value: instance.props || props,
        });
```

```
        // Set the instance state to null (not undefined)
if not set, to match React behaviour
        instance.state = instance.state || null;

        // Override setState to just change the state, not
queue up an update
        // (we can't do the default React thing as we
aren't mounted
        // "properly", however we don't need to re-render
as we only support
        // setState in componentWillMount, which happens
*before* render).
        instance.setState = newState => {
          if (typeof newState === 'function') {
            // React's TS type definitions don't contain
context as a third parameter for
            // setState's updater function.
            // Remove this cast to `any` when that is
fixed.
            newState = (newState as any)(instance.state,
instance.props, instance.context);
          }
          instance.state = Object.assign({},
instance.state, newState);
        };

        if (Comp.getDerivedStateFromProps) {
          const result =
Comp.getDerivedStateFromProps(instance.props,
instance.state);
          if (result !== null) {
            instance.state = Object.assign({},
instance.state, result);
          }
        } else if (instance.UNSAFE_componentWillMount) {
```

```
      } else if (instance.componentWillMount) {
        instance.componentWillMount();
      }

      if (providesChildContext(instance)) {
        childContext = Object.assign({}, context,
instance.getChildContext());
      }

      if (visitor(element, instance, context,
childContext) === false) {
        return;
      }

      child = instance.render();
    } else {
      // Just a stateless functional
      if (visitor(element, null, context) === false) {
        return;
      }

      child = Comp(props, context);
    }

    if (child) {
      if (Array.isArray(child)) {
        child.forEach(item => walkTree(item,
childContext, visitor));
      } else {
        walkTree(child, childContext, visitor);
      }
    }
  } else if ((element.type as any)._context ||
(element.type as any).Consumer) {
    // A React context provider or consumer
    if (visitor(element, null, context) === false) {
```

```
    }

    let child;
    if ((element.type as any)._context) {
      // A provider - sets the context value before
rendering children
      ((element.type as any)._context as
any)._currentValue = element.props.value;
      child = element.props.children;
    } else {
      // A consumer
      child = element.props.children((element.type as
any)._currentValue);
    }

    if (child) {
      if (Array.isArray(child)) {
        child.forEach(item => walkTree(item, context,
visitor));
      } else {
        walkTree(child, context, visitor);
      }
    }
  } else {
    // A basic string or dom element, just get children
    if (visitor(element, null, context) === false) {
      return;
    }

    if (element.props && element.props.children) {
      React.Children.forEach(element.props.children,
(child: any) => {
        if (child) {
          walkTree(child, context, visitor);
        }
      });
```

```
    }
  } else if (typeof element === 'string' || typeof element
=== 'number') {
    // Just visit these, they are leaves so we don't keep
traversing.
    visitor(element, null, context);
  }
}
```

注水和脱水，是同构应用最为核心和关键的细节点。

请求认证处理

上面讲到服务端预先请求数据，那么思考这样的场景：某个请求依赖 cookie 表明的用户信息，比如请求「我的学习计划列表」。这种情况下服务端请求是不同于客户端的，不会有浏览器添加 cookie 以及不含有其他相关的 header 信息。这个请求在服务端发送时，一定不会拿到预期的结果。

为了解决这个问题，我们来看看 React-apollo 的解决方法：

```
import { ApolloProvider } from 'react-apollo'
import { ApolloClient } from 'apollo-client'
import { createHttpLink } from 'apollo-link-http'
import Express from 'express'
import { StaticRouter } from 'react-router'
import { InMemoryCache } from "apollo-cache-inmemory"

import Layout from './routes/Layout'

// Note you don't have to use any particular http server,
but
// we're using Express in this example
const app = new Express();
app.use((req, res) => {
```

```
   // Remember that this is the interface the SSR server
will use to connect to the
   // API server, so we need to ensure it isn't
firewalled, etc
   link: createHttpLink({
     uri: 'http://localhost:3010',
     credentials: 'same-origin',
     headers: {
       cookie: req.header('Cookie'),
     },
   }),
   cache: new InMemoryCache(),
 });

 const context = {}

 // The client-side App will instead use
 const App = (




 );

 // rendering code (see below)
})
```

这个做法也非常简单，原理是：服务端请求时需要保留客户端页面请求的信息，并在 API 请求时携带并透传这个信息。上述代码中，createHttpLink 方法调用时：

```
headers: {
   cookie: req.header('Cookie'),
},
```

这个配置项就是关键，它使得服务端的请求完整地还原了客户端信息，因此验证类接口也不再会有问题。

事实上，很多早期 React 完成服务端渲染的轮子都借鉴了 React-apollo 众多优秀思想，对这个话题感兴趣的读者可以抽空去了解 React-apollo。

样式问题处理

同构应用的样式处理容易被开发者所忽视，而一旦忽略，就会掉到坑里。比如，正常的服务端渲染只是返回了 HTML 字符串，样式需要浏览器加载完 CSS 后才会加上，这个样式添加的过程就会造成页面的闪动。

再比如，我们不能再使用 style-loader 了，因为这个 webpack loader 会在编译时将样式模块载入到 HTML header 中。但是在服务端渲染环境下，没有 window 对象，style-loader 进而会报错。一般我们换用 isomorphic-style-loader 来实现：

```
{
   test: /\.css$/,
   use: [
        'isomorphic-style-loader',
        'css-loader',
        'postcss-loader'
   ],
}
```

同时 isomorphic-style-loader 也会解决页面样式闪动的问题。它的原理也不难理解：在服务器端输出 html 字符串的同时，也将样式插入到 html 字符串当中，将结果一同传送到客户端。

isomorphic-style-loader 的原理是什么呢?

我们知道对于 webpack 来说，所有的资源都是模块，webpack loader 在编译过程中可以将导入的 CSS 文件转换成对象，拿到样式信息。因此 isomorphic-style-loader 可以获取页面中所有组件样式。为了实现的更

加通用化，isomorphic-style-loader 利用 context API，在渲染页面组件时获取所有 React 组件的样式信息，最终插入到 HTML 字符串中。

在服务端渲染时，我们需要加入这样的逻辑：

```
import express from 'express'
import React from 'react'
import ReactDOM from 'react-dom'
import StyleContext from 'isomorphic-style-
loader/StyleContext'
import App from './App.js'

const server = express()
const port = process.env.PORT || 3000

// Server-side rendering of the React app
server.get('*', (req, res, next) => {

 const css = new Set() // CSS for all rendered React
components

 const insertCss = (...styles) => styles.forEach(style =>
css.add(style._getCss()))

 const body = ReactDOM.renderToString(



 )
 const html = `
```

```
${body}


    `
 res.status(200).send(html)
})

server.listen(port, () => {
 console.log(`Node.js app is running at
http://localhost:${port}/`)
})
```

我们定义了 css Set 类型来存储页面所有的样式，并定义了 insertCss 方法，该方法通过 context 传给每个 React 组件，这样每个组件就可以调用 insertCss 方法。该方法调用时，会将组件样式加入到 css Set 当中。

最后我们用 [...css].join('') 就可以获取页面的所有样式字符串。

强调一下，isomorphic-style-loader 的源码目前已经更新，采用了最新的 React hooks API，我推荐给 React 开发者阅读，相信一定收获很多！

meta tags 渲染

React 应用中，骨架往往类似：

```
const App = () => {
 return (
```

```
}
ReactDom.render(, document.querySelector('#root'))
```

App 组件嵌入到 document.querySelector('#root') 节点当中，一般是不包含 head 标签的。但是单页应用在切换路由时，可能也会需要动态修改 head 标签信息，比如 title 内容。也就是说：在单页面应用切换页面，不会经过服务端渲染，但是我们仍然需要更改 document 的 title 内容。

那么服务端如何渲染 meta tags head 标签就是一个常被忽略但是至关重要的话题，我们往往使用 React-helmet 库来解决问题。

Home 组件：

```
import Helmet from "react-helmet";
```

Home component

Users 组件：

React-helmet 这个库会在 Home 组件和 Users 组件渲染时，检测到 Helmet，并自动执行副作用逻辑。

当服务端渲染时，我们还需要留心对 404 的情况进行处理，有 layout.js 文件如下：

当访问：/home 时，会得到一个空白页面，浏览器也没有得到 404 的状态码。为了处理这种情况，我们加入：

并创建 NotFound.js 文件：

```
import React from 'react'

export default function NotFound({ staticContext }) {
   if (staticContext) {
       staticContext.notFound = true
   }
   return (


Not found

   )
}
```

注意，在访问一个不存在的地址时，我们要返回 404 状态码。一般 React router 类库已经帮我们进行了较好的封装，Static Router 会注入一个 context prop，并将 context.notFound 赋值为 true，在 server/index.js 加入：

```
const context = {}
const html = renderer(data, req.path, context);
if (context.notFound) {
   res.status(404)
}
res.send(html)
```

即可。这一系列处理过程没有什么难点，但是这种处理意识，还是需要具备的。

安全问题

安全问题非常关键，尤其是涉及到服务端渲染，开发者要格外小心。这里提出一个点：我们前面提到了注水和脱水过程，其中的代码：

```
ctx.body = `

        // ...

`
```

非常容易遭受 XSS 攻击，JSON.stringify 可能会造成 script 注入。因此，我们需要严格清洗 JSON 字符串中的 HTML 标签和其他危险的字符。我习惯使用 serialize-javascript 库进行处理，这也是同构应用中最容易

这里给大家留一个思考题，React dangerouslySetInnerHTML API 也有类似风险，React 是怎么处理这个安全隐患的呢?

性能优化

我们将数据请求移到了服务端，但是依然要格外重视性能优化。目前针对于此，业界普遍做法包括以下几点。

使用缓存：服务端优化一个最重要的手段就是缓存，不同于传统服务端缓存措施，我们甚至可以实现组件级缓存，业界 walmartlabs 在这方面的实践非常多，且收获了较大的性能提升。感兴趣的读者可以找到相关技术信息。

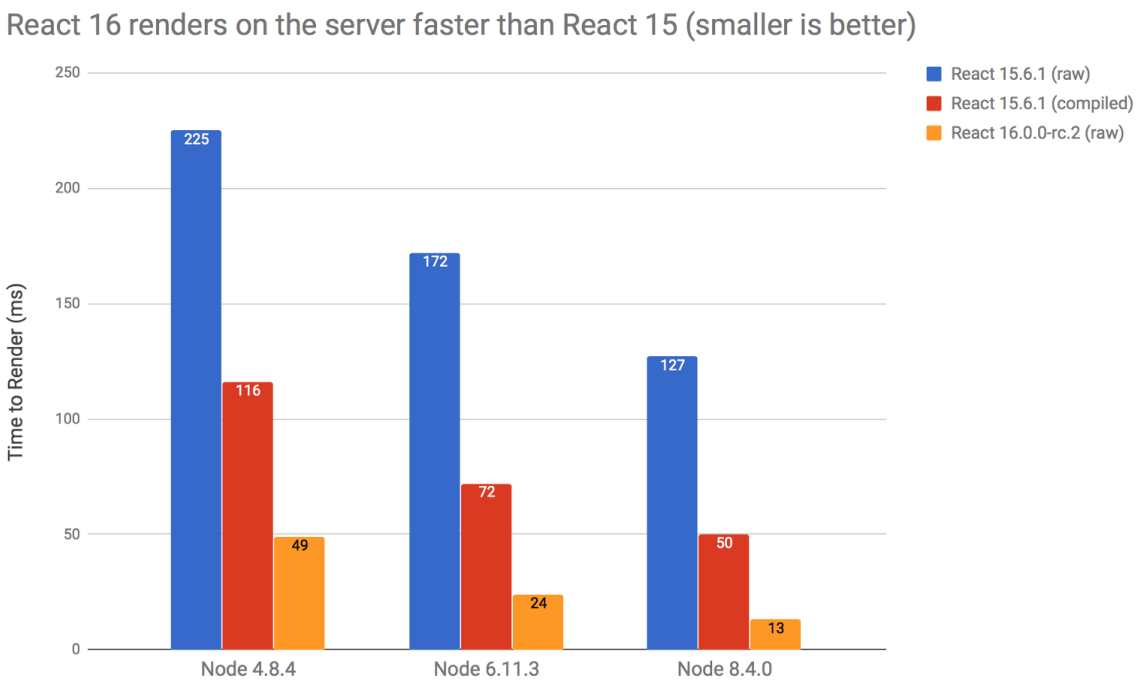
采用 HSF 代替 HTTP，HSF 是 High-Speed Service Framework 的缩写，译为分布式的远程服务调用框架，对外提供服务上，HSF 性能远超过 HTTP。

对于服务端压力过大的场景，动态切换为客户端渲染。

NodeJS 升级。

React 升级。

如图所示，React 16 在服务端渲染上的性能对比提升：



备注：图片来自 hacker noon

总结

本讲没有「手把手」教你实现服务端渲染的同构应用，因为这些知识并不困难，社区上资料也很多。我们从更高的角度出发，剖析同构应用中那些关键的细节点和疑难问题的解决方案，这些经验来源于真刀真枪的线上案例，如果读者没有开发过同构应用，也能从中全方位地了解关键信息，一旦掌握了这些细节，同构应用的实现就会更稳、更可靠。

同构应用其实远比理论复杂，绝对不是几个 APIs 和几台服务器就能完成的，希望大家多思考、多动手，一定会更有体会。

另外，同构应用各种细节也不止于此，坑也不止于此，欢迎大家和我讨论。

分享交流

阅读文章过程中有任何疑问可随时跟其他小伙伴讨论，或者直接向作者 LucasHC 提问（作者看到后抽空回复）。**你的分享不仅帮助他人，更会提升自**

你也可以说说自己最想了解的主题，课程内容会根据部分读者的意见和建议迭代和完善。

此外，我们为本课程付费读者创建了《前端开发核心知识进阶》微信交流群，以方便更有针对性地讨论课程相关问题（入群请到第 1-2 课末尾添加 GitChat 小助手伽利略的微信，并注明「前端核心」，谢谢~）

点击查看下一节

从框架和类库，我们该学到什么

点击查看下一节

从框架和类库，我们该学到什么