

Angular Universal 服务端渲染

0.628 2018.11.14 11:01:01 字数 3703 阅读 869

Angular Universal

Angular在服务端渲染方面提供一套前后端同构解决方案，它就是 [Angular Universal](#)（统一平台），一项在服务端运行 Angular 应用的技术。

标准的 Angular 应用会执行在浏览器中，它会在 DOM 中渲染页面，以响应用户的操作。

而 Angular Universal 会在服务端通过一个被称为服务端渲染（server-side rendering - SSR）的过程生成静态的应用页面。

它可以生成这些页面，并在浏览器请求时直接用它们给出响应。它也可以把页面预先生成为 HTML 文件，然后把它们作为静态文件供服务端使用。

工作原理

要制作一个 Universal 应用，就要安装 `platform-server` 包。`platform-server` 包提供了服务端的 DOM 实现、XMLHttpRequest 和其它底层特性，但不再依赖浏览器。

你要使用 `platform-server` 模块而不是 `platform-browser` 模块来编译这个客户端应用，并且在一个 Web 服务器上运行这个 Universal 应用。

服务器（下面的示例中使用的是 Node Express 服务器）会把客户端对应用页面的请求传给 `renderModuleFactory` 函数。

`renderModuleFactory` 函数接受一个模板 HTML 页面（通常是 `index.html`）、一个包含组件的 Angular 模块和一个用于决定该显示哪些组件的路由作为输入。

该路由从客户端的请求中传给服务器。每次请求都会给出所请求路由的一个适当的视图。

`renderModuleFactory` 在模板中的 `<app>` 标记中渲染出哪个视图，并为客户端创建一个完成的 HTML 页面。

最后，服务器就会把渲染好的页面返回给客户端。

为什么要服务端渲染

三个主要原因：

1. 帮助网络爬虫（SEO）
2. 提升在手机和低功耗设备上的性能
3. 迅速显示出第首页

帮助网络爬虫（SEO）

Google、Bing、百度、Facebook、Twitter 和其它搜索引擎或社交媒体网站都依赖网络爬虫去索引你的应用内容，并且让它的内容可以通过网络搜索到。

这些网络爬虫可能不会像人类那样导航到你的具有高度交互性的 Angular 应用，并为其建立索引。



互联网编程

拥有 135 钻 (约 14.62 元)

关注

爬取抖音视频最新版代码-2019年8月

阅读 118

实时协同编辑的实现

阅读 258

精彩继续

34张深夜监控照，撕开了上亿成年人的伪装

阅读 25942



汤唯的口语，到底有多好？

阅读 5839

写下你的评论...

评论0 赞7 ...

启用网络爬虫通常被称为[搜索引擎优化 \(SEO\)](#)。

提升手机和低功耗设备上的性能

有些设备不支持 JavaScript 或 JavaScript 执行得很差，导致用户体验不可接受。对于这些情况，你可能会需要该应用的服务端渲染、无 JavaScript 的版本。虽然有一些限制，不过这个版本可能是那些完全没办法使用该应用的人的唯一选择。

快速显示首页

快速显示首页对于吸引用户是至关重要的。

如果页面加载超过了三秒中，那么 53% 的移动网站会被放弃。你的应用需要启动的更快一点，以便在用户决定做别的事情之前吸引他们的注意力。

使用 Angular Universal，你可以为应用生成“着陆页”，它们看起来就和完整的应用一样。这些着陆页是纯 HTML，并且即使 JavaScript 被禁用了也能显示。这些页面不会处理浏览器事件，不过它们可以用 routerLink 在这个网站中导航。

在实践中，你可能要使用一个着陆页的静态版本来保持用户的注意力。同时，你也会在幕后加载完整的 Angular 应用。用户会认为着陆页几乎是立即出现的，而当完整的应用加载完之后，又可以获得完全的交互体验。

示例解析

下面将基于我在GitHub上的示例项目 [angular-universal-starter](#) 来进行讲解。

这个项目与第一篇的示例项目一样，都是基于 Angular CLI进行开发构建的，因此它们的区别只在于服务端渲染所需的那些配置上。

安装工具

在开始之前，下列包是必须安装的（示例项目均已配置好，只需 `npm install` 即可）：

- `@angular/platform-server` - Universal 的服务端元件。
- `@nguniversal/module-map-ngfactory-loader` - 用于处理服务端渲染环境下的惰性加载。
- `@nguniversal/express-engine` - Universal 应用的 [Express 引擎](#)。
- `ts-loader` - 用于对服务端应用进行转译。
- `express` - Node Express 服务器

使用下列命令安装它们：

```
@angular/platform-server @nguniversal/module-map-ngfactory-loader ts-loader @nguniversal/express-engine express
```

项目配置

配置工作有：

1. 创建服务端应用模块：`src/app/app.server.module.ts`
2. 修改客户端应用模块：`src/app/app.module.ts`
3. 创建服务端应用的引导程序文件：`src/main.server.ts`

写下你的评论...

评论0 赞7 ...

7. 创建 Node Express 的服务程序： `server.ts`
8. 创建服务端预渲染的程序： `prerender.ts`
9. 创建 Webpack 的服务端配置： `webpack.server.config.js`

1、创建服务端应用模块： `src/app/app.server.module.ts`

```
1 import { NgModule } from '@angular/core';
2 import { ServerModule, ServerTransferStateModule } from '@angular/platform-server';
3 import { ModuleMapLoaderModule } from '@nguniversal/module-map-ngfactory-loader';
4
5 import { AppBrowserModule } from './app.module';
6 import { AppComponent } from './app.component';
7
8 // 可以注册那些在 Universal 环境下运行应用时特有的服务提供商
9 @NgModule({
10   imports: [
11     AppBrowserModule, // 客户端应用的 AppModule
12     ServerModule, // 服务端的 Angular 模块
13     ModuleMapLoaderModule, // 用于实现服务端的路由的惰性加载
14     ServerTransferStateModule, // 在服务端导入，用于实现将状态从服务器传输到客户端
15   ],
16   bootstrap: [AppComponent],
17 })
18 export class AppServerModule {
19 }
```

服务端应用模块（习惯上叫作 AppServerModule）是一个 Angular 模块，它包装了应用的根模块 AppModule，以便 Universal 可以在你的应用和服务器之间进行协调。AppServerModule 还会告诉 Angular 再把你的应用以 Universal 方式运行时，该如何引导它。

2、修改客户端应用模块： `src/app/app.module.ts`

```
import { BrowserModule, BrowserTransferStateModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { APP_ID, Inject, NgModule, PLATFORM_ID } from '@angular/core';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { TransferHttpCacheModule } from '@nguniversal/module-map-ngfactory-loader';
import { isPlatformBrowser } from '@angular/common';
import { AppRoutingModule } from './app.routes';

@NgModule({
  imports: [
    AppRoutingModule,
    BrowserModule.withServerTransition({appId: 'my-app'}),
    TransferHttpCacheModule, // 用于实现服务器到客户端的请求传输缓存，防止客户端重复请求服务端已完成的请求
    BrowserTransferStateModule, // 在客户端导入，用于实现将状态从服务器传输到客户端
    HttpClientModule
  ],
  declarations: [
    AppComponent,
    HomeComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppBrowserModule {
  constructor(@Inject(PLATFORM_ID) private platformId: Object,
    @Inject(APP_ID) private appId: string) {

    // 判断运行环境为客户端还是服务端
    const platform = isPlatformBrowser(platformId) ? 'in the browser' : 'on the server';
    console.log(`Running ${platform} with appId=${appId}`);
  }
}
```

将 `NgModule` 的元数据中 `BrowserModule` 的导入改成 `BrowserModule.withServerTransition({appId: 'my-app'})`

写下你的评论...

评论0 赞7 ...

此时，我们可以通过依赖注入（`@Inject(PLATFORM_ID)` 及 `@Inject(APP_ID)`）取得关于当前平台和 `appId` 的运行时信息：

```
1 constructor(@Inject(PLATFORM_ID) private platformId: Object,  
2             @Inject(APP_ID) private appId: string) {  
3  
4             // 判断运行环境为客户端还是服务端  
5             const platform = isPlatformBrowser(platformId) ? 'in the browser' : 'on the server';  
6             console.log(`Running ${platform} with appId=${appId}`);  
7         }
```

3、创建服务端应用的引导程序文件： `src/main.server.ts`

该文件导出服务端模块：

```
1 export { AppServerModule } from './app/app.server.module';
```

4、修改客户端应用的引导程序文件： `src/main.ts`

监听 `DOMContentLoaded` 事件，在发生 `DOMContentLoaded` 事件时运行我们的代码，以使 `TransferState` 正常工作

```
1 import { enableProdMode } from '@angular/core';  
2 import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  
3  
4 import { AppBrowserModule } from './app/app.module';  
5 import { environment } from './environments/environment';  
6  
7 if (environment.production) {  
8     enableProdMode();  
9 }  
10  
11 // 在 DOMContentLoaded 时运行我们的代码，以使 TransferState 正常工作  
12 document.addEventListener('DOMContentLoaded', () => {  
13     platformBrowserDynamic().bootstrapModule(AppBrowserModule);  
14 });
```

5、创建 TypeScript 的服务端配置： `src/tsconfig.server.json`

```
1 {  
2     "extends": "../tsconfig.json",  
3     "compilerOptions": {  
4         "outDir": "../out-tsc/app",  
5         "baseUrl": "./",  
6         "module": "commonjs",  
7         "types": [  
8             "node"  
9         ]  
10    },  
11    "exclude": [  
12        "test.ts",  
13        "**/*.spec.ts"  
14    ],  
15    "angularCompilerOptions": {  
16        "entryModule": "app/app.server.module#AppServerModule"  
17    }  
18 }
```

与 `tsconfig.app.json` 的差异在于：

- `module` 属性必须是 `commonjs`，这样它才能被 `require()` 方法导入你的服务端应用。
- `angularCompilerOptions` 部分有一些面向 AOT 编译器的选项：

在 `apps` 下添加:

```
1 {
2   "platform": "server",
3   "root": "src",
4   "outDir": "dist/server",
5   "assets": [
6     "assets",
7     "favicon.ico"
8   ],
9   "index": "index.html",
10  "main": "main.server.ts",
11  "test": "test.ts",
12  "tsconfig": "tsconfig.server.json",
13  "testTsconfig": "tsconfig.spec.json",
14  "prefix": "",
15  "styles": [
16    "styles.scss"
17  ],
18  "scripts": [],
19  "environmentSource": "environments/environment.ts",
20  "environments": {
21    "dev": "environments/environment.ts",
22    "prod": "environments/environment.prod.ts"
23  }
24 }
```

7、创建 Node Express 的服务程序: `server.ts`

```
1 import 'zone.js/dist/zone-node';
2 import 'reflect-metadata';
3 import { enableProdMode } from '@angular/core';
4
5 import * as express from 'express';
6 import { join } from 'path';
7 import { readFileSync } from 'fs';
8
9 // Faster server renders w/ Prod mode (dev mode never needed)
10 enableProdMode();
11
12 // Express server
13 const app = express();
14
15 const PORT = process.env.PORT || 4000;
16 const DIST_FOLDER = join(process.cwd(), 'dist');
17
18 // Our index.html we'll use as our template
19 const template = readFileSync(join(DIST_FOLDER, 'browser', 'index.html')).toString();
20
21 // * NOTE :: leave this as require() since this file is built Dynamically from webpack
22 const {AppServerModuleNgFactory, LAZY_MODULE_MAP} = require('./dist/server/main.bundle');
23
24 // Express Engine
25 import { ngExpressEngine } from '@nguniversal/express-engine';
26 // Import module map for lazy loading
27 import { provideModuleMap } from '@nguniversal/module-map-ngfactory-loader';
28
29 // Our Universal express-engine (found @ https://github.com/angular/universal/tree/master/modules
30 app.engine('html', ngExpressEngine({
31   bootstrap: AppServerModuleNgFactory,
32   providers: [
33     provideModuleMap(LAZY_MODULE_MAP)
34   ]
35 })));
36
37 app.set('view engine', 'html');
38 app.set('views', join(DIST_FOLDER, 'browser'));
39
40 /* - Example Express Rest API endpoints -
41 app.get('/api/**', (req, res) => { });
42 */
43
44 // Server static files from /browser
45 app.get('*..*', express.static(join(DIST_FOLDER, 'browser')), {
```

写下你的评论...

评论0 赞7 ...

```
51 res.render('index', {req});
52 });
53
54 // Start up the Node server
55 app.listen(PORT, () => {
56   console.log(`Node Express server listening on http://localhost:${PORT}`);
57 });
```

Universal 模板引擎

这个文件中最重要的部分是 ngExpressEngine 函数：

```
1 app.engine('html', ngExpressEngine({
2   bootstrap: AppServerModuleNgFactory,
3   providers: [
4     provideModuleMap(LAZY_MODULE_MAP)
5   ]
6 }));
```

ngExpressEngine 是对 Universal 的 renderModuleFactory 函数的封装。它会把客户端请求转换成服务端渲染的 HTML 页面。如果你使用不同于 Node 的服务端技术，你需要在该服务端的模板引擎中调用这个函数。

- 第一个参数是你以前写过的 AppServerModule。它是 Universal 服务端渲染器和你的应用之间的桥梁。
- 第二个参数是 extraProviders。它是在这个服务器上运行时才需要的一些可选的 Angular 依赖注入提供商。当你的应用需要那些只有当运行在服务器实例中才需要的信息时，就要提供 extraProviders 参数。

ngExpressEngine 函数返回了一个会解析成渲染好的页面的承诺（Promise）。

接下来你的引擎要决定拿这个页面做什么。现在这个引擎的回调函数中，把渲染好的页面返回给了 Web 服务器，然后服务器通过 HTTP 响应把它转发给了客户端。

8、创建服务端预渲染的程序：prerender.ts

```
1 // Load zone.js for the server.
2 import 'zone.js/dist/zone-node';
3 import 'reflect-metadata';
4 import { readFileSync, writeFileSync, existsSync, mkdirSync } from 'fs';
5 import { join } from 'path';
6
7 import { enableProdMode } from '@angular/core';
8 // Faster server renders w/ Prod mode (dev mode never needed)
9 enableProdMode();
10
11 // Import module map for lazy loading
12 import { provideModuleMap } from '@nguniversal/module-map-ngfactory-loader';
13 import { renderModuleFactory } from '@angular/platform-server';
14 import { ROUTES } from './static.paths';
15
16 // * NOTE :: leave this as require() since this file is built Dynamically from webpack
17 const {AppServerModuleNgFactory, LAZY_MODULE_MAP} = require('./dist/server/main.bundle');
18
19 const BROWSER_FOLDER = join(process.cwd(), 'browser');
20
21 // Load the index.html file containing references to your application bundle.
22 const index = readFileSync(join('browser', 'index.html'), 'utf8');
23
24 let previousRender = Promise.resolve();
25
26 // Iterate each route path
27 ROUTES.forEach(route => {
28   const fullPath = join(BROWSER_FOLDER, route);
```

写下你的评论...

 评论0  赞7 ...

```
35 // Writes rendered HTML to index.html, replacing the file if it already exists.
36 previousRender = previousRender.then(_ => renderModuleFactory(AppServerModuleNgFactory, {
37   document: index,
38   url: route,
39   extraProviders: [
40     provideModuleMap(LAZY_MODULE_MAP)
41   ]
42 })).then(html => writeFileSync(join(fullPath, 'index.html'), html));
43 });
```

9、创建 Webpack 的服务端配置：webpack.server.config.js

Universal 应用不需要任何额外的 Webpack 配置，Angular CLI 会帮我们处理它们。但是由于本例子的 Node Express 的服务程序是 TypeScript 应用（server.ts及prerender.ts），所以要使用 Webpack 来转译它。这里不讨论 Webpack 的配置，需要了解的移步 [Webpack官网](#)

```
1 // Work around for https://github.com/angular/angular-cli/issues/7200
2
3 const path = require('path');
4 const webpack = require('webpack');
5
6 module.exports = {
7   entry: {
8     server: './server.ts', // This is our Express server for Dynamic universal
9     prerender: './prerender.ts' // This is an example of Static prerendering (generative)
10  },
11  target: 'node',
12  resolve: {extensions: ['.ts', '.js']},
13  externals: [/!(node_modules|main\\.\\.\\.js)/], // Make sure we include all node_modules etc
14  output: {
15    path: path.join(__dirname, 'dist'), // Puts the output at the root of the dist folder
16    filename: '[name].js'
17  },
18  module: {
19    rules: [
20      {test: /\.ts$/, loader: 'ts-loader'}
21    ]
22  },
23  plugins: [
24    new webpack.ContextReplacementPlugin(
25      /(.)?angular(\\|\/)core(.+)?/, // fixes WARNING Critical dependency: the request of
26      path.join(__dirname, 'src'), // location of your src
27      {} // a map of your routes
28    ),
29    new webpack.ContextReplacementPlugin(
30      /(.)?express(\\|\/)(.+)?/, // fixes WARNING Critical dependency: the request of a de
31      path.join(__dirname, 'src'),
32      {}
33    )
34  ]
35 };
```

测试配置

通过上面的配置，我们就制作完成一个可在服务端渲染的 Angular Universal 应用。

在 package.json 的 scripts 区配置 build 和 serve 有关的命令：

```
1 {
2   "scripts": {
3     "ng": "ng",
4     "start": "ng serve -o",
5     "ssr": "npm run build:ssr && npm run serve:ssr",
6     "prerender": "npm run build:prerender && npm run serve:prerender",
7     "build": "ng build",
8     "build:client-and-server-bundles": "ng build --prod && ng build --prod --app 1 --output-f
9     "build:prerender": "npm run build:client-and-server-bundles && npm run webpack:server &&
10    "build:ssr": "npm run build:client-and-server-bundles && npm run webpack:server",
11    "generate:prerender": "cd dist && node prerender"
```

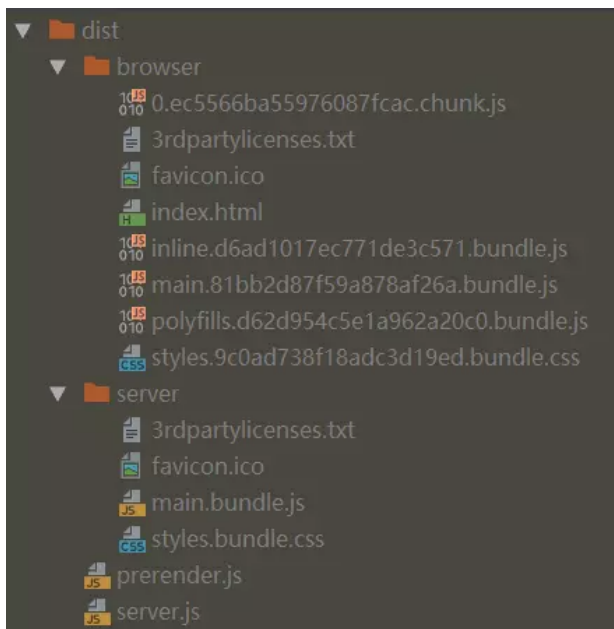
写下你的评论...

评论0 赞7 ...

开发只需运行 `npm run start`

执行 `npm run ssr` 编译应用程序，并启动一个Node Express来为应用程序提供服务 `http://localhost:4000`

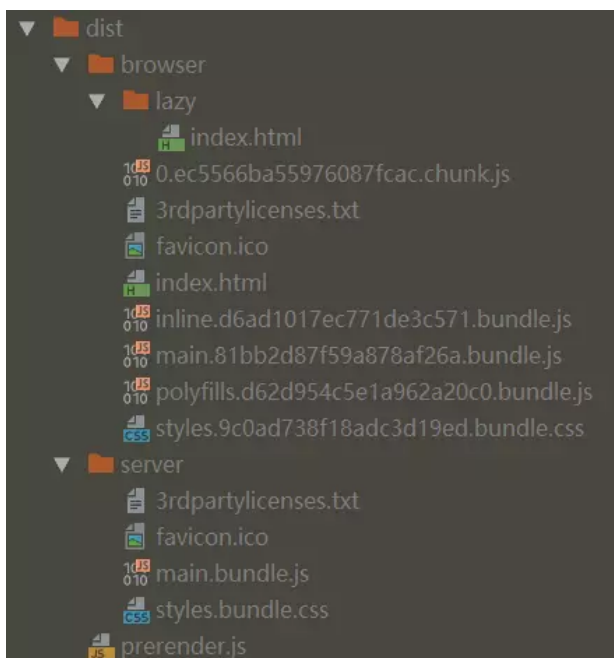
dist目录：



执行`npm run prerender` - 编译应用程序并预渲染应用程序文件，启动一个演示http服务器，以便您可以查看它 `http://localhost:8080`

注意：要将静态网站部署到静态托管平台，您必须部署dist/browser文件夹，而不是dist文件夹

dist目录：



写下你的评论...

评论0 赞7 ...


```
1 export const ROUTES = [  
2   '/',  
3   '/lazy'  
4 ];
```

因此，从dist目录可以看到，服务端预渲染会根据配置好的路由在 browser 生成对应的静态 index.html。如 / 对应 /index.html，/lazy 对应 /lazy/index.html。

服务端的模块懒加载

在前面的介绍中，我们在 `app.server.module.ts` 中导入了 `ModuleMapLoaderModule`，在 `app.module.ts`。

`ModuleMapLoaderModule` 模块可以使得懒加载的模块也可以在服务端进行渲染，而你要做也只是在 `app.server.module.ts` 中导入。

服务端到客户端的状态传输

在前面的介绍中，我们在 `app.server.module.ts` 中导入了 `ServerTransferStateModule`，在 `app.module.ts` 中导入了 `BrowserTransferStateModule` 和 `TransferHttpCacheModule`。

这三个模块都与服务端到客户端的状态传输有关：

- `ServerTransferStateModule`：在服务端导入，用于实现将状态从服务端传输到客户端
- `BrowserTransferStateModule`：在客户端导入，用于实现将状态从服务端传输到客户端
- `TransferHttpCacheModule`：用于实现服务端到客户端的请求传输缓存，防止客户端重复请求服务端已完成的请求

使用这几个模块，可以解决 **http请求在服务端和客户端分别请求一次** 的问题。






比如在 `home.component.ts` 中有如下代码：

```
1 import { Component, OnDestroy, OnInit } from '@angular/core';  
2 import { HttpClient } from '@angular/common/http';  
3 import { Observable } from 'rxjs/Observable';  
4  
5 @Component({  
6   selector: 'app-home',  
7   templateUrl: './home.component.html',  
8   styleUrls: ['./home.component.scss']  
9 })  
10 export class HomeComponent implements OnInit, OnDestroy {  
11   constructor(public http: HttpClient) {  
12   }  
13  
14   ngOnInit() {  
15     this.poiSearch(this.keyword, '北京市').subscribe((data: any) => {  
16       console.log(data);  
17     });  
18   }  
19  
20   ngOnDestroy() {  
21   }  
22  
23   poiSearch(text: string, city?: string): Observable<any> {  
24     return this.http.get(encodeURIComponent(`http://restapi.amap.com/v3/place/text?keywords=${text}&ci`));  
25   }  
26 }
```

代码运行之后，

```
[ { id: 'B0FFIOY46A',
  name: '肯德基',
  tag: [],
  type: '餐饮服务;快餐厅;肯德基',
  typecode: '050301',
  biz_type: 'diner',
  address: '西单北大街堂子胡同9号新一代商城地下一层',
  location: '116.375098,39.911168',
  tel: [],
  postcode: [],
  website: [],
  email: [],
  pcode: '110000',
```

客户端再一次请求并打印：

Name	Status
 polyfills.062d954c5e1a9b2azucv.bundle.js	200
 styles.9c0ad738f18adc3d19ed.bundle.css	200
 main.a3e6453ddcccd135c3c2.bundle.js	200
 text?keywords=%E8%82%AF%E5%BE%B7%E5%9F%BA&city=%E5...y=55f909211b9950837fba2c71d0488db9&extensi...	200
 ng-validate.js	200

7 requests | 8.6 KB transferred | Finish: 1.42 s | DOMContentLoaded: 1.08 s | Load: 1.46 s

Console

top Filter Default levels

Running in the browser with appId=my-app

(20) [{}], {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}]

方法1：使用 TransferHttpCacheModule
使用 TransferHttpCacheModule 很简单，代码不需要改动。在 app.module.ts 中导入之后，Angular会自动会将服务端请求缓存到客户端，换句话说就是服务端请求到数据会自动传输到客户端，客户端接收到数据之后就不会再发送请求了。

方法2：使用 BrowserTransferStateModule
该方法稍微复杂一些，需要改动一些代码。

调整 home.component.ts 代码如下：

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { makeStateKey, TransferState } from '@angular/platform-browser';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';

const KFCLIST_KEY = makeStateKey('kfcList');

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit, OnDestroy {
  constructor(public http: HttpClient,
    private state: TransferState) {
```

```
3 // 不用 ！ 你也不应该去服务器端去自己拿到数据， 你也不需要到数据就再去服务器端请求， 你也不需要到数据就再去服务器端请求。
4 const kfclist: any[] = this.state.get(KFCLIST_KEY, null as any);
5
6 if (!this.kfclist) {
7   this.poiSearch(this.keyword, '北京市').subscribe((data: any) => {
8     console.log(data);
9     this.state.set(KFCLIST_KEY, data as any); // 存储数据
10   });
11 }
12 }
13
14 ngOnDestroy() {
15   if (typeof window === 'object') {
16     this.state.set(KFCLIST_KEY, null as any); // 删除数据
17   }
18 }
19
20 poiSearch(text: string, city?: string): Observable<any> {
21   return this.http.get(encodeURIComponent(`http://restapi.amap.com/v3/place/text?keywords=${text}&city=${city}`));
22 }
```

}

使用 const KFCLIST_KEY = makeStateKey('kfclist') 创建储存传输数据的 StateKey

在 HomeComponent 的构造函数中注入 TransferState

在 ngOnInit 中根据 this.state.get(KFCLIST_KEY, null as any) 判断数据是否存在（不管是服务端还是客户端），存在就不再请求，不存在则请求数据并通过 this.state.set(KFCLIST_KEY, data as any) 存储传输数据

在 ngOnDestroy 中根据当前是否客户端来决定是否将存储的数据进行删除

客户端与服务端渲染对比

最后，我们分别通过这三个原因来进行对比：

帮助网络爬虫（SEO）

提升在手机和低功耗设备上的性能

迅速显示出首页

帮助网络爬虫（SEO）

客户端渲染：

```
<!--></head><body><app-root></app-root><script type="text/javascript" src="inline.bc54d499c18b29e4d2bb.bundle.js"></script>
返回的页面没有渲染数据
```

服务端渲染：

从上面可以看到，服务端提前将信息渲染到返回的页面上，这样网络爬虫就能直接获取到信息了（网络爬虫基本不会解析javascript的）。

提升在手机和低功耗设备上的性能

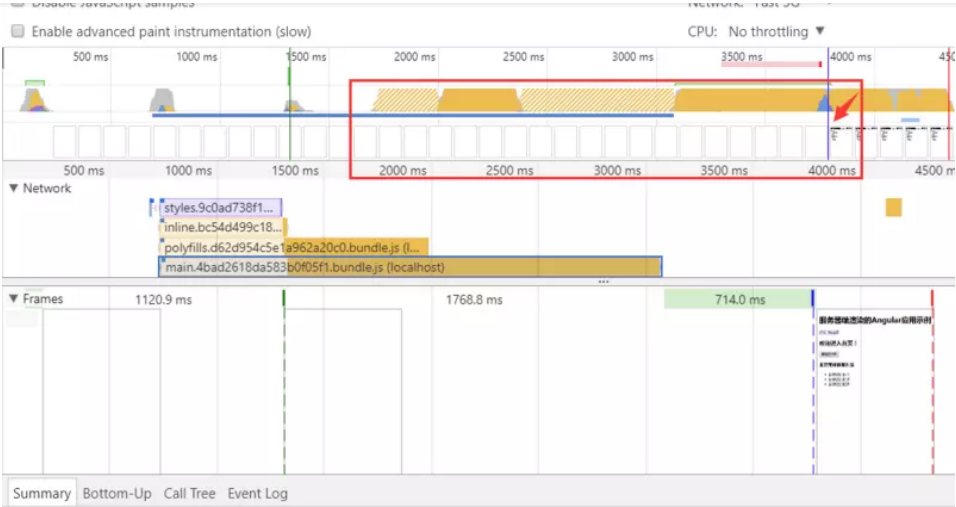
这个原因通过上面就可以看出，对于一些低端的设备，直接显示页面总比要解析javascript性能高的多。

迅速显示出首页

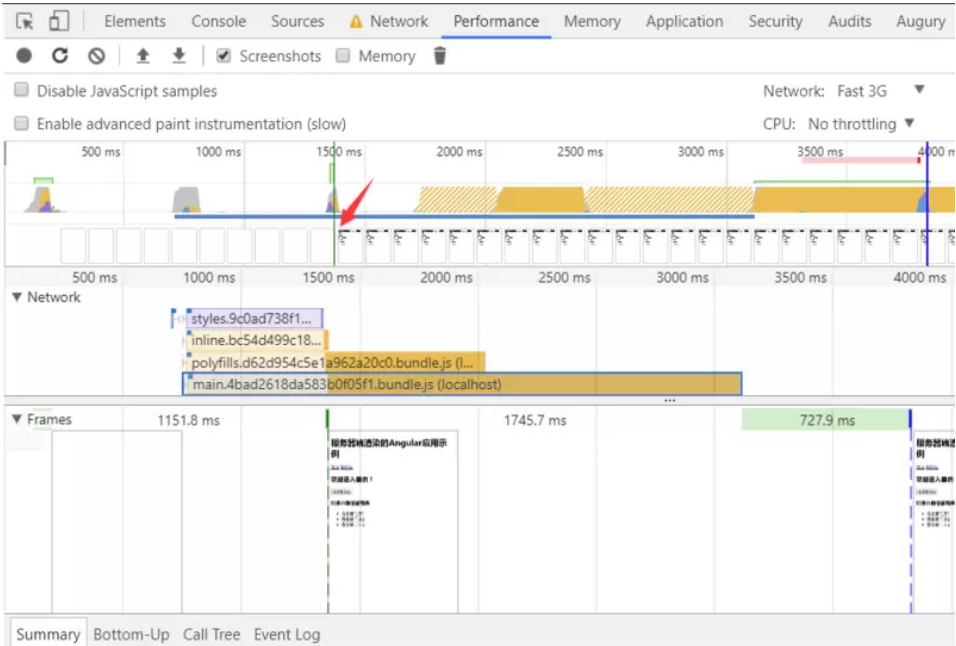
同样在 Fast 3G 网络条件下进行测试

客户端渲染：





服务端渲染：



Network request

URL [main.4bad261...bundle.js](#)

Duration 2.35 s

Request Method GET

Priority High

Mime Type application/javascript

Encoded Data 259 KB

Decoded Body 259 KB

首页渲染花费时间大概是：1s
此时加载的main.xxx.js依然是2.35s，但不影响首页的渲染

牢记几件事情

- 对于服务器软件包，您可能需要将第三方模块包含到 `nodeExternals` 白名单中
- `window` , `document` , `navigator` 以及其它的浏览器类型 - 不存在于服务端 - 如果你直接使用，在服务端将无法正常工作。以下几种方法可以让你的代码正常工作：
 - 可以通过 `PLATFORM_ID` 标记注入的 `Object` 来检查当前平台是浏览器还是服务器，然后使用浏览器端特有的类型



写下你的评论...

评论0 赞7 ...

```
6   ngOnInit() {
7     if (isPlatformBrowser(this.platformId)) {
8       // 仅运行在浏览器端的代码
9       ...
10    }
11    if (isPlatformServer(this.platformId)) {
12      // 仅运行在服务端的代码
13      ...
14    }
15  }
```

- 尽量**限制**或**避免**使用`setTimeout`。它会减慢服务器端的渲染过程。确保在组件的`ngOnDestroy`中删除它。
- 对于RxJs超时，请确保在成功时`取消`它们的流，因为它们也会降低渲染速度。


- 不要直接操作`NativeElement`，使用`Renderer2`，从而可以跨平台改变应用视图。


```
1  constructor(element: ElementRef, renderer: Renderer2) {
2    this.renderer.setStyle(element.nativeElement, 'font-size', 'x-large');
3  }
```


- 解决应用程序在服务器上运行XHR请求，并在客户端再次运行的问题
 - 使用从服务器传输到客户端的缓存（`TransferState`）
- 清楚了解与DOM相关的属性和属性之间的差异
- 尽量让指令无状态。对于有状态指令，您可能需要提供一个属性，以反映相应属性的初始字符串值，例如img标签中的url。对于我们的native元素，src属性被反映为元素类型`HTMLImageElement`的src属性

 7人点赞 >



 Angular






互联网编程
拥有135钻 (约14.62元)

关注

"拒绝赞赏！有钱任性！！！"

赞赏



写下你的评论...

全部评论 0

只看作者

按时间倒序 按时间正序

被以下专题收入，发现更多相似内容

 Angular...

推荐阅读

更多精彩内容 >

写下你的评论...

评论0

 赞7

