链滴

# 关于微前端实现原理与 ngx-planet(二)

作者：someone61489
原文链接：https://ld246.com/article/1611298351444
来源网站：链滴

<!--
 * @Author: ferried
 * @Email: harlancui@outlook.com
 * @Date: 2021-01-22 15:02:37
 * @LastEditTime: 2021-01-25 16:31:48
 * @LastEditors: ferried
 * @Description: Basic description
 * @FilePath: /undefined/Users/ferried/ngx-planet2.md

-->

# 道标

准备好源码，然后跟着文章去看代码，在每个代码块的第一行，我都把 filename 写上了，并且打开了 gitalk。

# 项目结构

```
- packages/planet
|--src
   |--application
      |--planet-application-loader.spec.ts
      |--planet-application-loader.ts # 应用加载器
      |--planet-application-ref.spec.ts
      |--planet-application-ref.ts # 应用的引用
      |--planet-application.service.spec.ts
      |--planet-application.service.ts # 应用逻辑处理Service
      |--portal-application.spec.ts
      |--portal-application.ts  # Portal应用
   |--component
      |--planet-component-loader.spec.ts
      |--planet-component-loader.ts # 组件加载器
      |--planet-component-ref.ts # 组件引用
      |--plant-component.config.ts # 组件配置
   |--empty
      |--empty.component.spec.ts
      |--empty.component.ts   # 空组件
   |--testing
      |--app1.module.ts # 测试用例
      |--app2.module.ts
      |--applications.ts
      |--index.ts
      |--utils.ts
   |--assets-loader.spec.ts
   |--assets-loader.ts    # 静态资源加载器
   |--global-event-dispatcher.spec.ts
   |--global-event-dispatcher.ts   # 全局事件调度器
   |--global-planet.spec.ts
   |--global-planet.ts     # 一些函数
   |--helper.spec.ts
   |--helper.ts    # 一些util函数
   |--module.spec.ts
```

```
    |--module.ts    # packager module
    |--planet.class.ts  # 注入配置和InjectToken
    |--planet.spec.ts
    |--planet.ts    # planet 对象，包含了注册，启动设置信息，实质为service
    |--public-api.ts  # 桶
    |--test.ts # 测试
|--karma.config.js  # test config
|--ng-package.json  # packager scheme
|--pakcage.json     # npm
|--tsconfig.lib.json    # compiler config
|--tsconfig.lib.prod.json # env=prod compiler config
|--tsconfig.spec.json   # test copiler config
|--tslint.json  # code lint rules
```

# 从 Portal 的 AppComponent 开始

```
// appcomponent.ts
    // 首先注入了 planet 对象
    constructor(
        private planet: Planet,
    ) {}

// appcomponent.ts
    // 初始化AppComponent中配置Portal和Applications
   ngOnInit() {
        ...
   }

// appcomponent.ts
// 设置PlanetApplicationLoader应用加载器的options
this.planet.setOptions({
  // switchMode
  switchMode: SwitchModes.coexist,
  // Application资源加载错误处理回调函数
  errorHandler: (error) => {
    // thy组件库的通知组件，理解成alert吧
    this.thyNotify.error(`错误`, "加载资源失败");
  },
});

// planet.class.ts
// SiwtchModes枚举类，切换子应用的模式，默认切换会销毁，设置 coexist 后只会隐藏
export enum SwitchModes {
  default = "default",
  coexist = "coexist",
}

// planet-application-loader.ts
    // Injectable直接注到模块里了(唯一),项目启动会通过Factory自行创建,所以不需要初始化
    @Injectable({
        providedIn: 'root'
    })
    export class PlanetApplicationLoader {
    private firstLoad = true;
```

```
    private startRouteChangeEvent: PlanetRouterEvent;

    // 这里是ApplicationLoader中的option
    private options: PlanetOptions

    ......

    }

// appcomponent.ts
// 向planet注册了两个应用
this.planet.registerApps([
  {
    // 子应用名
    name: "app1",
    // 应用渲染的容器元素, 指定子应用显示在哪个元素内部
    hostParent: "#app-host-container",
    // 宿主元素的 Class，也就是在子应用启动组件上追加的样式
    hostClass: appHostClass,
    // 子应用路由路径前缀，根据这个匹配应用
    routerPathPrefix: /\/app1|app4/,
    // 脚本和样式文件路径前缀，多个脚本可以避免重复写同样的前缀
    resourcePathPrefix: "/static/app1/",
    // 是否启用预加载，启动后刷新页面等当前页面的应用渲染完毕后预加载子应用
    preload: settings.app1.preload,
    // 切换子应用的模式，默认切换会销毁，设置 coexist 后只会隐藏
    switchMode: settings.app1.switchMode,
    // 是否串行加载脚本静态资源
    loadSerial: true,
    // 样式前缀
    stylePrefix: "app1",
    // 脚本资源文件
    scripts: ["main.js"],
    // 样式资源文件
    styles: ["styles.css"],
    // 应用程序打包后的脚本和样式文件替换
    manifest: "/static/app1/manifest.json",
    // 附加数据，主要应用于业务，比如图标，子应用的颜色，显示名等个性化配置
    extra: {
      name: "应用1",
      color: "#ffa415",
    },
  },
  .......
]);
```

在看 planet.registerApps之前先看一下 GlobalPlanet

```
// global-planet.ts
// 浏览器window对象
declare const window: any;

// interface
```

```
export interface GlobalPlanet {
  // 这里是一个Map,key是String类型的，Value为应用的引用
  apps: { [key: string]: PlanetApplicationRef };
  // 基座模式，只需要一个Portal
  portalApplication?: PlanetPortalApplication;
  // 应用加载器
  applicationLoader?: PlanetApplicationLoader;
  // service,函数
  applicationService?: PlanetApplicationService;
}

// 全局变量，如果window.planet不为空那么就拿window.planet的值，不然就初始化一个app的ma
出来
export const globalPlanet: GlobalPlanet = (window.planet = window.planet || {
  apps: {},
});

// 先看一眼，下面会在回来看
export function defineApplication(
  name: string,
  options: BootstrapAppModule | BootstrapOptions
) {
  if (globalPlanet.apps[name]) {
    throw new Error(`${name} application has exist.`);
  }
  if (isFunction(options)) {
    options = {
      template: "",
      bootstrap: options as BootstrapAppModule,
    };
  }
  const appRef = new PlanetApplicationRef(name, options as BootstrapOptions);
  globalPlanet.apps[name] = appRef;
}
```

为了继续看defineApplication先看一个用例,这个用例是app1的main.ts中的

```
// app1的 main.ts
// 调用defineApplication 传入
// 1. app1,
// 2.函数返回 ModuleRef 启动的模块
defineApplication('app1', {

    // template 为一个dom字符串
    template: `<app1-root class="app1-root"></app1-root>`,

    // bootstrap 应用的加载点
    bootstrap: (portalApp: PlanetPortalApplication) => {

    // platformBrowserDynamic 传入 Provider, Provider同于 app.module中的providers 提供服
用的，作者在这里插入了两个 值或对象
    >>>>  export declare const platformBrowserDynamic: (extraProviders?: StaticProvider[] | u
defined) => PlatformRef;
        return platformBrowserDynamic([
```

```
        // 其实提供服务也可以理解成有一个Context,把Value注入到Context中了，在项目中的Cont
xt中可以通过Inject方式来提取使用这个服务了
        {
            provide: PlanetPortalApplication,
            // 注意这里传入的是 defineApplication 函数 第二参数options中 BootstrapAppModul
中的回调参数
            useValue: portalApp
        },
        ...
    ])
    // 引导AppModule模块,创建@NgModule实例也就是创建一个AppModule出来
        .bootstrapModule(AppModule)
        .then(appModule => {
            // 最后返回Promise<NgModule>
            // 至此，第二参中的BootstrapAppModule结束
            return appModule;
        })
        .catch(error => {
            console.error(error);
            return null;
        });
  }
});
```

好，现在返回defineApplication中去看看用两个参数做了些什么呢

```
// planet-applicaiton.ref.ts

// 这是第二参的类型
export interface BootstrapOptions {
  template: string;
  bootstrap: BootstrapAppModule;
}

// 第二参数中 bootstrap 要传入portalApp然后返回NgModuldeRef
export type BootstrapAppModule = (
  portalApp?: PlanetPortalApplication
) => Promise<NgModuleRef<any>>;
```

上面已经讲过怎么用了,下面看当调用时，包内发生了什么?

```
// global-planet.ts
export function defineApplication(
  name: string,
  options: BootstrapAppModule | BootstrapOptions
) {
  // 首先，看看globalPlanet的app map中有没有加入过这个项目，也就是说，如果有10个应用，调
defineApplication时传入相同的第一参name,就会发生应用重复，无法分别当前需要加载的是哪个
用，作者在这里判断并抛出异常
  if (globalPlanet.apps[name]) {
    throw new Error(`${name} application has exist.`);
  }
  // 其次，看options第二参数，是带template的Bootstrap函数还是直接就是bootstrap函数
  if (isFunction(options)) {
```

```
    // 如果是函数，那么template为空，第二参数转化类型,options备用
    options = {
      template: "",
      bootstrap: options as BootstrapAppModule,
    };
  }
  // 这里比较关键了，通过name和options new了一个PlanetApplicationRef对象
  const appRef = new PlanetApplicationRef(name, options as BootstrapOptions);
  // 将appRef也就是子applicaiton的引用加入globalPlanet.apps的map中，也就是window.planet.
pps中,至于为什么，前面有提到
  globalPlanet.apps[name] = appRef;
}
```

看看如何 new 一个子应用引用对象吧~

```
// planet-application-ref.ts

export class PlanetApplicationRef{
    ...

private innerSelector: string;
// 这里 将 innerSelector调用方式改为了 app.selector因为是私有变量嘛
public get selector() {
    return this.innerSelector;
}

// 首先，构造函数两个参数由defineApplication传递过来
 constructor(name: string, options: BootstrapOptions) {
     // 传递一些值给当前对象的属性
        this.name = name;
        if (options) {
            this.template = options.template;
            // 如果template被传入了 getTagNameByTemplate,由于是util不在赘述，这里拿到了标签
称，稍后看一个应用demo
            this.innerSelector = this.template ? getTagNameByTemplate(this.template) : null;
            // 将Promise<NgModuleRef>也就是bootstrap Instance也传进来
            this.appModuleBootstrap = options.bootstrap;
    }
 }
```

作者每一个属性名和变量的词语都非常精准，所以导致看代码的时候非常好看,非常干净,这也是 clean ode 第一章所追求的,我所追求的代码风格，非常棒

看一个 selecor 应用 demo

```
// planet-application-loader.ts
// 首先通过name获取ApplicationRef引用
  private hideApp(planetApp: PlanetApplication) {
        const appRef = getPlanetApplicationRef(planetApp.name);
        // 获取插入的整体dom节点的document,通过selector
        const appRootElement = document.querySelector(appRef.selector || planetApp.selector);
        // 隐藏dom元素
        if (appRootElement) {
            appRootElement.setAttribute('style', 'display:none;');
        }
```

```
    }
```

好了，至此注册内容结束，总体来说就是把 name 和 template 放到 window 下的对象中的 map 里在提供一个 boostrap 和 angular 的 main.ts 结合起来，将 instance 也存入 map 中

# 再次回到 Portal 的 appcomponent,这次看细致一些

```
// portal appcomponent.ts
// 注册了planet
    constructor(
        private planet: Planet,
    ) {}
```

看看初始化 plant 的时候，plant 内部是什么样子的

```
// planet.ts
// injectable -> context
@Injectable({
    providedIn: 'root'
})
export class Planet {
    ...

// 看这里，当factory向context注入Planet时，发生了什么
    constructor(
        private injector: Injector,
        private router: Router,
        // angular 提供InjectToken方式注入值
        @Inject(PLANET_APPLICATIONS) @Optional() planetApplications: PlanetApplication[]
        // 为了解释，插入一段module.ts的代码
---NgxPlanetModule : module.ts
// 放出NgxPlanetModule
export class NgxPlanetModule {
    // 引入模块既可以直接引入也可以NgxPlanetModule.forRoot(apps)
    // apps为静态的PlanetApplication集合，其实就是appcomponent中register的那个集合
    // 标识为PLANET_APPLICATIONS理解成字符串就好了
    // Inject(PLANET_APPLICATIONS)也就是拿到外部模块引用forRoot函数中传入的结合
    static forRoot(apps: PlanetApplication[]): ModuleWithProviders<NgxPlanetModule> {
        return {
            ngModule: NgxPlanetModule,
            providers: [
                {
                    provide: PLANET_APPLICATIONS,
                    useValue: apps
                }
            ]
        };
    }
}
---
// 回到planet.ts
    ) {
        // 通过injector从context里提取出injectable过的service,注意，全局唯一service
        if (!this.planetApplicationService) {
```

```
            // 注意，这里service给了谁了,给了global下的applicationService,也就是window下的那个
象,这里可以点setApplicationService去看,前面看过globalPlanet了
            setApplicationService(this.injector.get(PlanetApplicationService));
        }
        // 如果forRoot传入了apps，那么注册
        if (planetApplications) {
            // 调用注册函数
            this.registerApps(planetApplications);
        }
    }


    // 由于construct中用injector获得了全局planetApplicationService实例，所以这里直接可以用了
    registerApps<TExtra>(apps: PlanetApplication<TExtra>[]) {
        // 调用service的注册，将apps在向下传一层
        this.planetApplicationService.register(apps);
    }

}
```

看一下 service 中的代码

```
// planet-application.service.ts
@Injectable({
  providedIn: "root",
})
export class PlanetApplicationService {
  // PlanetApplication集合
  private apps: PlanetApplication[] = [];
  // PlanetApplication map
  private appsMap: { [key: string]: PlanetApplication } = {};

  constructor(private http: HttpClient, private assetsLoader: AssetsLoader) {
    // 首先检测 global下的applicationService是不是已经有了，有了就抛异常，因为这里存的是app
列表，所以不能刷新他的引用使它变成新的对象,不然存储的注册application就没有了
    if (getApplicationService()) {
      throw new Error(
        "PlanetApplicationService has been injected in the portal, repeated injection is not allow
d"
      );
    }
  }

  // 注册
  register<TExtra>(
    appOrApps: PlanetApplication<TExtra> | PlanetApplication<TExtra>[]
  ) {
    // 单个application转数组,多个application直接返回数组
    const apps = coerceArray(appOrApps);
    apps.forEach((app) => {
      // 判断是否注册过了application了
      if (this.appsMap[app.name]) {
        throw new Error(`${app.name} has be registered.`);
      }
```

```
      // 将application加入apps数组和map中
      this.apps.push(app);
      this.appsMap[app.name] = app;
    });
  }
}
```

至此，模块启动实例(bootstrap|@NgModule)和注册从main.ts,portal 的appcomponent.ts解析完毕

无论是启动实例或是应用列表，都存在了 global 下面，前一种存到了apps下，后一个，存到了applicationService的数组和 map 中

# 第三次回到 portal 的 appcomponent

register 下一行，调用了 start

```
// appcomponent.ts
// 调用planet的start
this.planet.start();
```

看看如何写的,这里我加入了一些 console 通过输出的内容分析一下

```
// plant.ts
start() {
      this.subscription = this.router.events
          .pipe(
              filter(event => {
                  return event instanceof NavigationEnd;
              }),
              map((event: NavigationEnd) => {
                  return event.urlAfterRedirects || event.url;
              }),
              startWith(location.pathname),
              distinctUntilChanged(),
          )
          .subscribe((url: string) => {
              console.log(url)
              this.planetApplicationLoader.reroute({
                  url: url
              });
          });
    }
```

以下是输出内容

```
// construct是我在planetApplicationLoader构造函数里加入的console
// 通过顺序验证了，当factory创建好了planetApplicationLoader后会立即执行构造函数中的内容，以construct排第一
construct
// subscript真正订阅到的路由字符串
planet.ts:103 /about
```

然后使用planetApplicationLoader.reroute({})去改变了planetApplicationLoader中的private routeChange$ = new Subject<PlanetRouterEvent>();

由于 construct 先运行，所以先来看planetApplicationLoader的构造函数进行了什么操作

```
// planet-application-loader.ts


 constructor(
    // assetsLoader先不考虑，我猜大概是对应了angular静态资源目录进行操作的一个对象
        private assetsLoader: AssetsLoader,
        // planetApplicationService 前面看到的service
        // 里面有register和app列表
        private planetApplicationService: PlanetApplicationService,
        // ngZone https://hijiangtao.github.io/2020/01/17/Angular-Zone-Concepts/ 可以看这篇
子，主要是关于变更检测机制的一个东西
        // 简单来说 run runOutsideAngular两个函数，一个是进行变更检测，一个在zone之外运行不
发变更检测
        private ngZone: NgZone,
        // 路由
        router: Router,
        // injector
        injector: Injector,
        // 对页面上运行的angular应用程序的引用
        applicationRef: ApplicationRef
    ) {
        // 这里类似planetApplicationService的检测,检测global中是否已经存在了loader加载器
        // 那么ApplicationLoader什么时候被创建的呢，看看之前planetApplicationService在哪里创
的，它上面就是setApplicationLoader
        if (getApplicationLoader()) {
            throw new Error(
                'PlanetApplicationLoader has been injected in the portal, repeated injection is not al
owed'
            );
        }

        // 设置一些参数
        this.options = {
            // 应用是隐藏还是销毁
            switchMode: SwitchModes.default,
            // 错误处理回调函数
            errorHandler: (error: Error) => {
                console.error(error);
            }
        };
        // 将zone传递给portalApp
        this.portalApp.ngZone = ngZone;
        // 当前application的引用传递给portalApp
        this.portalApp.applicationRef = applicationRef;
        // 路由传递
        this.portalApp.router = router;
        // 这里我理解为把注入器传过来了，也就可以用injector来拿当前context的对象了
        // 可能理解有误，请在下方留言告知
        this.portalApp.injector = injector;
        // 一会儿在看如何共享事件调用的
        this.portalApp.globalEventDispatcher = injector.get(GlobalEventDispatcher);
        // 将portalApp给global对象了
```

```
    globalPlanet.portalApplication = this.portalApp;
    // 调用了一个关于路由的函数
    this.setupRouteChange();
  }
```

先看 this.portalApp 是什么

```
//  planet-application-loader.ts
  private portalApp = new PlanetPortalApplication();
```

new 了一个 portalApplication 对象对吧，这就是我们的基座 application 了，看看里面有什么

```
// portal-application.ts
export class PlanetPortalApplication<TData = any> {
  // 当前应用的引用
  applicationRef: ApplicationRef;
  // 注入器
  injector: Injector;
  // 路由
  router: Router;
  // 变更检测
  ngZone: NgZone;
  // 事件分发器
  globalEventDispatcher: GlobalEventDispatcher;
  // 额外数据?
  data: TData;

  // 跳转函数,返回了一个Promise<boolean>
  navigateByUrl(
    url: string | UrlTree,
    extras?: NavigationExtras
  ): Promise<boolean> {
    return this.ngZone.run(() => {
      return this.router.navigateByUrl(url, extras);
    });
  }

  // ngZone.run 在变更检测区域内运行函数，这里不知道为何不 object.ngZone.run而是要写一个
数来包装
  run<T>(fn: (...args: any[]) => T): T {
    return this.ngZone.run<T>(() => {
      return fn();
    });
  }

  // 全局性调用变化检测
  // applicationRef可以通过attachView()将视图包含到变化检测中
  // 可以用detachView()将视图移除变化检测
  tick() {
    this.applicationRef.tick();
  }
}
```

好了，接下来看加载应用的核心函数setupRouteChange这个函数有点长

```
// planet-application-loader.ts

 private setupRouteChange() {
        // 当路由变化时，先思考，application-loader比start调用先创建，所以start后，页面定到/ab
ut路由，所以这里可以接收到/about
        this.routeChange$
            .pipe(
                distinctUntilChanged((x, y) => {
                    return (x && x.url) === (y && y.url);
                }),
                // 和其他打平操作符的主要区别是它具有取消效果。在每次发出时，会取消前一个内部 ob
ervable (你所提供函数的结果) 的订阅，然后订阅一个新的 observable
                switchMap(event => {
                    return of(event).pipe(
                        // 卸载应用程序并返回应加载的应用程序
                        map(() => {
                            // 这里拿到{url:'/about'}
                            this.startRouteChangeEvent = event;
                            // 通过url找app ,PlanetApplication类型,用service中的app列表和 注册过的app的
outerPathPrefix和url进行匹配
                            const shouldLoadApps = this.planetApplicationService.getAppsByMatchedUr
(event.url);
                            // 这里拿到了需要卸载的应用,下面有讲
                            const shouldUnloadApps = this.getUnloadApps(shouldLoadApps);
                            // 设置加载和卸载的application,这里传入了加载应用和卸卸载应用，并且放出了o
对象，估计是作者提供hook函数，想提供给我们在loading时做一些操作的自定义函数的hook
                            // 使用的时候 planet.appsLoadingStart.subscribe就可以拿到了
                            this.appsLoadingStart$.next({
                                shouldLoadApps,
                                shouldUnloadApps
                            });
                            // 卸载,下面有讲
                            this.unloadApps(shouldUnloadApps, event);
                            return shouldLoadApps;
                        }),
                        // 加载静态资源
                        switchMap(shouldLoadApps => {
                            let hasAppsNeedLoadingAssets = false;
                            const loadApps$ = shouldLoadApps.map(app => {
                                // 获取当前app的状态
                                const appStatus = this.appsStatus.get(app);
                                //  如果没有加载过，或者曾经加载错了
                                // 设置需要加载状态后使用assetsLoader去加载静态资源,稍后讲assetsLoader
                                if (
                                    !appStatus ||
                                    appStatus === ApplicationStatus.assetsLoading ||
                                    appStatus === ApplicationStatus.loadError
                                ) {
                                    hasAppsNeedLoadingAssets = true;
                                    return this.ngZone.runOutsideAngular(() => {
                                        return this.startLoadAppAssets(app);
                                    });
                                } else {
                                    return of(app);
```

```
                }
            });
            if (hasAppsNeedLoadingAssets) {
                this.loadingDone = false;
            }
            return loadApps$.length > 0 ? forkJoin(loadApps$) : of([] as PlanetApplicati
n[]);
        }),
        // 引导启动或展示application
        map(apps => {
            const apps$: Observable<PlanetApplication>[] = [];
            apps.forEach(app => {
                // 获取app状态
                const appStatus = this.appsStatus.get(app);
                // 如果引导过了,也就是拿到过context了
                if (appStatus === ApplicationStatus.bootstrapped) {
                    apps$.push(
                        of(app).pipe(
                            tap(() => {
                                // 展示app
                                this.showApp(app);
                                // 获取app引用
                                const appRef = getPlanetApplicationRef(app.name);
                                // 路由跳转 router是从 applicationRef也就是子application中 injec
出来的router,所以路由表完备
                                appRef.navigateByUrl(event.url);
                                // 设置app状态为激活状态
                                this.setAppStatus(app, ApplicationStatus.active);
                                // 加载结束
                                this.setLoadingDone();
                            })
                        )
                    );
                } else if (appStatus === ApplicationStatus.assetsLoaded) {
                    // 如果appStatus状态为静态资源加载完毕
                    apps$.push(
                        of(app).pipe(
                            switchMap(() => {
                                // 引导app获取app context中的一些对象
                                return this.bootstrapApp(app).pipe(
                                    map(() => {
                                        // 设置app模式为激活状态
                                        this.setAppStatus(app, ApplicationStatus.active);
                                        // 加载完毕
                                        this.setLoadingDone();
                                        return app;
                                    })
                                );
                            })
                        )
                    );
                    // 如果应用未激活状态
                } else if (appStatus === ApplicationStatus.active) {
                    apps$.push(
```

```
                    of(app).pipe(
                        tap(() => {
                            // 获取引用
                            const appRef = getPlanetApplicationRef(app.name);
                            // 获取当前路径
                            const currentUrl = appRef.getCurrentRouterStateUrl? appRef.ge
CurrentRouterStateUrl(): '';
                            // 如果当前url和 事件url不一致
                            if (currentUrl !== event.url) {
                                // 路由跳转
                                appRef.navigateByUrl(event.url);
                            }
                        })
                    )
                );
            } else {
                throw new Error(
                    `app(${app.name})'s status is ${appStatus}, can't be show or bootstra
`
                );
            }
        });

        if (apps$.length > 0) {
            // 切换到应用后会有闪烁现象，所以使用 setTimeout 后启动应用
            setTimeout(() => {
                // 此处判断是因为如果静态资源加载完毕还未启动被取消，还是会启动之前
应用，虽然可能性比较小，但是无法排除这种可能性，所以只有当 Event 是最后一个才会启动
                if (this.startRouteChangeEvent === event) {
                    this.ngZone.runOutsideAngular(() => {
                        // 将apps中的ob们合并到一起执行后
                        forkJoin(apps$).subscribe(() => {
                            // 设置加载完成状态
                            this.setLoadingDone();
                            // 然后去搞预加载,注册时传入的参数preload=tue/false
                            this.ensurePreloadApps(apps);
                        });
                    });
                }
            });
        } else {
            // 设置
            this.ensurePreloadApps(apps);
            this.setLoadingDone();
        }
    }),
    // 使用自定义异常hook function
    catchError(error => {
        this.errorHandler(error);
        return [];
    })
);
})
)
```

```
            .subscribe();
    }
```

然后看unload,bootstrap,destory,preload,show,hide,卸载，加载，销毁，预加载，展示，隐藏相关这几个函数

获取卸载 app

```
// planet-applicaiton-loader.ts
// 传入激活的application列表
 private getUnloadApps(activeApps: PlanetApplication[]) {
        const unloadApps: PlanetApplication[] = [];
        this.appsStatus.forEach((value, app) => {
            // application状态为激活，但是激活列表中没有这个application
            if (value === ApplicationStatus.active && !activeApps.find(item => item.name === a
p.name)) {
                // 那么放入待卸载应用集合里
                unloadApps.push(app);
            }
        });
        return unloadApps;
    }
```

卸载

```
// planet-applicaiton-loader.ts
// 卸载applications,第一参为application集合,第二个参数类似{url:'/about'}
private unloadApps(shouldUnloadApps: PlanetApplication[], event: PlanetRouterEvent) {
        const hideApps: PlanetApplication[] = [];
        const destroyApps: PlanetApplication[] = [];
        // 需要卸载的app看看是隐藏模式还是销毁模式
        shouldUnloadApps.forEach(app => {
            // app.SwitchMode参数的两个值，在注册的时候设置的
            // coexist模式
            if (this.switchModeIsCoexist(app)) {
                hideApps.push(app);
                // 隐藏应用
                this.hideApp(app);
                // 设置状态
                this.setAppStatus(app, ApplicationStatus.bootstrapped);
                // default模式,销毁模式
            } else {
                destroyApps.push(app);
                // 销毁之前先隐藏，否则会出现闪烁，因为 destroy 是延迟执行的
                // 如果销毁不延迟执行，会出现切换到主应用的时候会有视图卡顿现象
                this.hideApp(app);
                //
    ---
    // application应用状态有5个
    // 资源加载状态分别是,我估计这里和预加载或者做判断有用到
>>  assetsLoading = 1,
    // 资源被加载状态
    assetsLoaded = 2,
    // 正在引导启动状态
    bootstrapping = 3,
```

```
    // 已经被引导启动状态
    bootstrapped = 4,
    // 激活状态
    active = 5,
    // 加载失败
    loadError = 10
    ---
                this.setAppStatus(app, ApplicationStatus.assetsLoaded);
            }
        });

        // 如果隐藏列表或销毁列表中有值
        if (hideApps.length > 0 || destroyApps.length > 0) {
            // 从其他应用切换到主应用的时候会有视图卡顿现象，所以先等主应用渲染完毕后再加载其
应用
            // 此处尝试使用 this.ngZone.onStable.pipe(take(1)) 应用之间的切换会出现闪烁
            setTimeout(() => {
                // 这里还不太理解
                hideApps.forEach(app => {
                    const appRef = getPlanetApplicationRef(app.name);
                    if (appRef) {
                        appRef.navigateByUrl(event.url);
                    }
                });
                // 销毁app
                destroyApps.forEach(app => {
                    this.destroyApp(app);
                });
            });
        }
    }
```

销毁

```
// planet-application-loader.ts
// 传入需要销毁的application
 private destroyApp(planetApp: PlanetApplication) {
     // 得到applicationRef的引用
        const appRef = getPlanetApplicationRef(planetApp.name);
        if (appRef) {
            // 销毁
            appRef.destroy();
        }
        // 删除 document节点
        const container = getHTMLElement(planetApp.hostParent);
        const appRootElement = container.querySelector((appRef && appRef.selector) || planet
pp.selector);
        if (appRootElement) {
            container.removeChild(appRootElement);
        }
    }
```

显示和隐藏

```
// planet-application-loader.ts
    private hideApp(planetApp: PlanetApplication) {
        const appRef = getPlanetApplicationRef(planetApp.name);
        // 拿到dom节点
        const appRootElement = document.querySelector(appRef.selector || planetApp.selector);
        if (appRootElement) {
            // css隐藏
            appRootElement.setAttribute('style', 'display:none;');
        }
    }

    private showApp(planetApp: PlanetApplication) {
        const appRef = getPlanetApplicationRef(planetApp.name);
        // 拿到dom节点
        const appRootElement = document.querySelector(appRef.selector || planetApp.selector);
        // 去除隐藏样式
        if (appRootElement) {
            appRootElement.setAttribute('style', '');
        }
    }
```

预加载

```
// planet-applicatin-loader.ts
 private ensurePreloadApps(activeApps?: PlanetApplication[]) {
        // 第一次加载的时候 对app进行预加载
        if (this.firstLoad) {
            this.preloadApps(activeApps);
            this.firstLoad = false;
        }
    }

private preloadApps(activeApps?: PlanetApplication[]) {
        setTimeout(() => {
            // 过滤preload为true的application
            const toPreloadApps = this.planetApplicationService.getAppsToPreload(activeApps ?
ctiveApps.map(item => item.name) : null);
            // 加载
            const loadApps$ = toPreloadApps.map(preloadApp => {
                return this.preloadInternal(preloadApp);
            });
            // 对加载过程使用hooks错误处理
            forkJoin(loadApps$).subscribe({
                error: error => this.errorHandler(error)
            });
        });
}


// 第一参数application,第二参是否直接加载
private preloadInternal(app: PlanetApplication, immediate?: boolean): Observable<PlanetAppl
cationRef> {
        // 获取app状态
        const status = this.appsStatus.get(app);
```

```
        // 如果没有状态或者加载错误
        if (!status || status === ApplicationStatus.loadError) {
            return this.startLoadAppAssets(app).pipe(
                switchMap(() => {
                    // 如果是直接加载
                    if (immediate) {
                        // 在隐藏模式下加载
                        return this.bootstrapApp(app, 'hidden');
                    } else {
                        // 如果不是直接加载
                        // 在状态监测之外
                        return this.ngZone.runOutsideAngular(() => {
                            // 加载applicaiton
                            return this.bootstrapApp(app, 'hidden');
                        });
                    }
                }),
                map(() => {
                    // 返回application的引用
                    return getPlanetApplicationRef(app.name);
                })
            );
            // 如果在application属于其他状态
        } else if (
            [ApplicationStatus.assetsLoading, ApplicationStatus.assetsLoaded, ApplicationStatus.b
otstrapping].includes(
                status
            )
        ) {
            return this.appStatusChange.pipe(
                filter(event => {
                    return event.app === app && event.status === ApplicationStatus.bootstrapped;
                }),
                take(1),
                map(() => {
                    // 返回引用
                    return getPlanetApplicationRef(app.name);
                })
            );
        } else {
            // 异常
            const appRef = getPlanetApplicationRef(app.name);
            if (!appRef) {
                throw new Error(`${app.name}'s status is ${ApplicationStatus[status]}, planetApplicat
onRef is null.`);
            }
            return of(appRef);
        }
    }
```

bootstrap

```
// planet-application-loader.ts
 private bootstrapApp(
```

```
        app: PlanetApplication,
        defaultStatus: 'hidden' | 'display' = 'display'
    ): Observable<PlanetApplicationRef> {
        // 设置状态正在加载中
        this.setAppStatus(app, ApplicationStatus.bootstrapping);
        // 获取app的引用
        const appRef = getPlanetApplicationRef(app.name);
        // 如果注册了且boostrap函数不为空
        if (appRef && appRef.bootstrap) {
            // 获取 宿主dom
            const container = getHTMLElement(app.hostParent);
            let appRootElement: HTMLElement;
            // 进行显示隐藏，加入前缀，宿主class的操作
            if (container) {
                appRootElement = container.querySelector(appRef.selector || app.selector);
                if (!appRootElement) {
                    if (appRef.template) {
                        appRootElement = createElementByTemplate(appRef.template);
                    } else {
                        appRootElement = document.createElement(app.selector);
                    }
                    appRootElement.setAttribute('style', 'display:none;');
                    if (app.hostClass) {
                        appRootElement.classList.add(...coerceArray(app.hostClass));
                    }
                    if (app.stylePrefix) {
                        appRootElement.classList.add(...coerceArray(app.stylePrefix));
                    }
                    container.appendChild(appRootElement);
                }
            }
            // 加载
            let result = appRef.bootstrap(this.portalApp);
            if (result['then']) {
                result = from(result) as Observable<PlanetApplicationRef>;
            }
            // 最后返回app引用
            return result.pipe(
                tap(() => {
                    this.setAppStatus(app, ApplicationStatus.bootstrapped);
                    if (defaultStatus === 'display' && appRootElement) {
                        appRootElement.removeAttribute('style');
                    }
                }),
                map(() => {
                    return appRef;
                })
            );
        } else {
            throw new Error(
                `[${app.name}] not found, make sure that the app has the correct name defined use
defineApplication(${app.name}) and runtimeChunk and vendorChunk are set to true, details s
e https://github.com/worktile/ngx-planet#throw-error-cannot-read-property-call-of-undefin
d-at-__webpack_require__-bootstrap79`
```

```
            );
        }
    }
```

app.Ref.bootstrap()

```
// planet-application-ref
// 加载应用获取application的引用后返回自身  PlanetApplicationRef类型
 bootstrap(app: PlanetPortalApplication): Observable<this> {
        if (!this.appModuleBootstrap) {
            throw new Error(`app(${this.name}) is not defined`);
        }
        this.portalApp = app;
        // 使用main.ts中传入的bootstrap来加载app,然后拿到模块引用
        return from(
            this.appModuleBootstrap(app).then(appModuleRef => {
                // 传递引用,然后复制一些自己想要的信息从appModuleRef中
                this.appModuleRef = appModuleRef;
                // 传递name
                this.appModuleRef.instance.appName = this.name;
                this.syncPortalRouteWhenNavigationEnd();
                // 返回
                return this;
            })
        );
    }
```

this.appModuleBootstrap是通过main.ts的defineApplication第二参数的bootstrap传入的,之前看一下

```
// global-planet.ts
export function defineApplication(
  name: string,
  options: BootstrapAppModule | BootstrapOptions
) {
  if (globalPlanet.apps[name]) {
    throw new Error(`${name} application has exist.`);
  }
  if (isFunction(options)) {
    options = {
      template: "",
      bootstrap: options as BootstrapAppModule,
    };
  }
  // 这里new了一个PlanetApplicationRef然后将引用传入了map中
  const appRef = new PlanetApplicationRef(name, options as BootstrapOptions);
  globalPlanet.apps[name] = appRef;
}
```

this.appModuleBootstrap

```
// planet-application-ref.ts
export class PlanetApplicationRef {
    ...
    // 这里是global-planet new的对象时候传入的
```

```
    private appModuleBootstrap: (app: PlanetPortalApplication) => Promise<NgModuleRef<a
y>>;
    ...
    constructor(name: string, options: BootstrapOptions) {
        this.name = name;
        if (options) {
            this.template = options.template;
            this.innerSelector = this.template ? getTagNameByTemplate(this.template) : null;
            // 具体传入
            this.appModuleBootstrap = options.bootstrap;
        }
    }
```

至此，bootstrap 结束

# assetsLoader

之前在核心的setupRouteChange走第三步之前，中间有一个加载 assets 的阶段，毕竟子应用的main ts应该是不能主动注册过来的,需要加载静态文件来获取，我是这么猜测的

```
// planet-application-loader.ts
switchMap((shouldLoadApps) => {
  let hasAppsNeedLoadingAssets = false;
  const loadApps$ = shouldLoadApps.map((app) => {
    const appStatus = this.appsStatus.get(app);
    if (
      !appStatus ||
      appStatus === ApplicationStatus.assetsLoading ||
      appStatus === ApplicationStatus.loadError
    ) {
      hasAppsNeedLoadingAssets = true;
      return this.ngZone.runOutsideAngular(() => {
          // 核心调用,开始加载application的assets
        return this.startLoadAppAssets(app);
      });
    } else {
      return of(app);
    }
  });
  if (hasAppsNeedLoadingAssets) {
    this.loadingDone = false;
  }
  return loadApps$.length > 0
    ? forkJoin(loadApps$)
    : of([] as PlanetApplication[]);
});


 private startLoadAppAssets(app: PlanetApplication) {
        if (this.inProgressAppAssetsLoads.get(app.name)) {
            return this.inProgressAppAssetsLoads.get(app.name);
        } else {
            // assetsLoader.loadAppAssets 核心函数,调用完毕后看是否放入map中
            const loadApp$ = this.assetsLoader.loadAppAssets(app).pipe(
```

```
            tap(() => {
                this.inProgressAppAssetsLoads.delete(app.name);
                // 设置加载完毕状态
                this.setAppStatus(app, ApplicationStatus.assetsLoaded);
            }),
            map(() => {
                return app;
            }),
            catchError(error => {
                this.inProgressAppAssetsLoads.delete(app.name);
                // 加载失败状态
                this.setAppStatus(app, ApplicationStatus.loadError);
                throw error;
            }),
            share()
        );
        // 放入map中
        this.inProgressAppAssetsLoads.set(app.name, loadApp$);
        // 设置状态为assetsLoading状态,这里要知道返回的ob所以写代码的时候Loaded在Loading
面，真正订阅subscribe的时候，顺序就对了
        this.setAppStatus(app, ApplicationStatus.assetsLoading);
        return loadApp$;
    }
  }
```

调用了构造器注入的 assetsLoader 看一眼 loader 里咋写的吧

关于 mainfest.json 看一看 https://jonny-huang.github.io/angular/training/19_pwa/ 有关于 PWA 方面的内容

```
{
  "main.js": "main-es2015.js",
  "main.js.map": "main-es2015.js.map",
  "polyfills-es5.js": "polyfills-es5-es2015.js",
  "polyfills-es5.js.map": "polyfills-es5-es2015.js.map",
  "polyfills.js": "polyfills-es2015.js",
  "polyfills.js.map": "polyfills-es2015.js.map",
  "styles.css": "styles.css",
  "styles.css.map": "styles.css.map"
}

// load结果的接口类型
export interface AssetsLoadResult {
  src: string;
  hashCode: number;
  loaded: boolean;
  status: string;
}

// context注入,全剧唯一
@Injectable({
  providedIn: "root",
})
export class AssetsLoader {
```

```
// 加载过的sources
private loadedSources: number[] = [];

// 获取http客户端
constructor(private http: HttpClient) {}

// 加载script
loadScript(src: string): Observable<AssetsLoadResult> {
  // hashCode
  const id = hashCode(src);
  // 通过hashCode判断是否加载过了这个js
  if (this.loadedSources.includes(id)) {
    return of({
      src: src,
      hashCode: id,
      loaded: true,
      status: "Loaded",
    });
  }
  // 然后构建<script type="text/javascript" src=""> 插入dom
  return new Observable((observer: Observer<AssetsLoadResult>) => {
    const script: HTMLScriptElement = document.createElement("script");
    script.type = "text/javascript";
    script.src = src;
    script.async = true;
    if (script["readyState"]) {
      // IE
      script["onreadystatechange"] = () => {
        if (
          script["readyState"] === "loaded" ||
          script["readyState"] === "complete"
        ) {
          script["onreadystatechange"] = null;
          observer.next({
            src: src,
            hashCode: id,
            loaded: true,
            status: "Loaded",
          });
          observer.complete();
          this.loadedSources.push(id);
        }
      };
    } else {
      // Others
      script.onload = () => {
        observer.next({
          src: src,
          hashCode: id,
          loaded: true,
          status: "Loaded",
        });
        observer.complete();
        this.loadedSources.push(id);
```

```
      };
    }
    script.onerror = (error) => {
      observer.error({
        src: src,
        hashCode: id,
        loaded: false,
        status: "Error",
        error: error,
      });
      observer.complete();
    };
    document.body.appendChild(script);
  });
}

// 加载style  和script一个道理
loadStyle(src: string): Observable<AssetsLoadResult> {
  const id = hashCode(src);
  if (this.loadedSources.includes(id)) {
    return of({
      src: src,
      hashCode: id,
      loaded: true,
      status: "Loaded",
    });
  }
  return new Observable((observer: Observer<AssetsLoadResult>) => {
    const head = document.getElementsByTagName("head")[0];
    const link = document.createElement("link");
    link.rel = "stylesheet";
    link.type = "text/css";
    link.href = src;
    link.media = "all";
    link.onload = () => {
      observer.next({
        src: src,
        hashCode: id,
        loaded: true,
        status: "Loaded",
      });
      observer.complete();
      this.loadedSources.push(id);
    };
    link.onerror = (error) => {
      observer.error({
        src: src,
        hashCode: id,
        loaded: true,
        status: "Loaded",
        error: error,
      });
      observer.complete();
    };
```

```
    head.appendChild(link);
  });
}

// 加载script集合
loadScripts(
  sources: string[],
  serial = false
): Observable<AssetsLoadResult[]> {
  if (isEmpty(sources)) {
    return of(null);
  }
  const observables = sources.map((src) => {
    return this.loadScript(src);
  });
  if (serial) {
    const a = concat(...observables).pipe(
      map((item) => {
        return of([item]);
      }),
      concatAll()
    );
    return a;
  } else {
    return forkJoin(observables).pipe();
  }
}

// 加载style集合
loadStyles(sources: string[]): Observable<AssetsLoadResult[]> {
  if (isEmpty(sources)) {
    return of(null);
  }
  return forkJoin(
    sources.map((src) => {
      return this.loadStyle(src);
    })
  );
}

// 双重加载
loadScriptsAndStyles(
  scripts: string[] = [],
  styles: string[] = [],
  serial = false
) {
  return forkJoin([
    this.loadScripts(scripts, serial),
    this.loadStyles(styles),
  ]);
}

// 加载static资源
loadAppAssets(app: PlanetApplication) {
```

```
    if (app.manifest) {
      // 通过httpClient找到json，通过json找到script路径，最后插入到dom中
      return this.loadManifest(
        `${app.manifest}?t=${new Date().getTime()}`
      ).pipe(
        switchMap((manifestResult) => {
          // 获取了css和js的全路径 app1/{json中的js和css}
          const { scripts, styles } = getScriptsAndStylesFullPaths(
            app,
            manifestResult
          );
          // 然后加载
          return this.loadScriptsAndStyles(scripts, styles, app.loadSerial);
        })
      );
    } else {
      const { scripts, styles } = getScriptsAndStylesFullPaths(app);
      return this.loadScriptsAndStyles(scripts, styles, app.loadSerial);
    }
  }

  // 加载mainfest
  loadManifest(url: string): Observable<{ [key: string]: string }> {
    return this.http.get(url).pipe(
      map((response: any) => {
        return response;
      })
    );
  }
}
```

到此，javascript 和 stylesheet 加载完毕

# 组件注册

直接看文件然后找个例子看看

```
// planet-component-ref.ts
// 引用
export class PlanetComponentRef<TComp = any> {
  // 包装元素
  wrapperElement: HTMLElement;
  // 组件类型
  componentInstance: TComp;
  // 组件引用
  componentRef: ComponentRef<TComp>;
  // 处理函数
  dispose: () => void;
}

// planet-component-config.ts
// 组件配置
export class PlantComponentConfig<TData = any> {
  // 目标容器
```

```
  container: HTMLElement | ElementRef<HTMLElement | any>;
  // 包装class
  wrapperClass?: string;
  // 初始化状态
  initialState?: TData | null = null;
}

const componentWrapperClass = 'planet-component-wrapper';

export interface PlanetComponent<T = any> {
    name: string;
    component: ComponentType<T>;
}

@Injectable({
    providedIn: 'root'
})
export class PlanetComponentLoader {
    private domPortalOutletCache = new WeakMap<any, DomPortalOutlet>();

    private get applicationLoader() {
        return getApplicationLoader();
    }

    private get applicationService() {
        return getApplicationService();
    }

    constructor(
        // 当前页面应用的引用
        private applicationRef: ApplicationRef,
        // 模块的引用
        private ngModuleRef: NgModuleRef<any>,
        // 变更检测
        private ngZone: NgZone,
        // document
        @Inject(DOCUMENT) private document: any
    ) {}

    // 获取Planet存储的Application的引用，通过name,返回一个ob<引用>对象
    private getPlantAppRef(name: string): Observable<PlanetApplicationRef> {
        if (globalPlanet.apps[name] && globalPlanet.apps[name].appModuleRef) {
            return of(globalPlanet.apps[name]);
        } else {
            const app = this.applicationService.getAppByName(name);
            return this.applicationLoader.preload(app, true).pipe(
                map(() => {
                    return globalPlanet.apps[name];
                })
            );
        }
    }

    // 传入模块和组件的引用,创建injector
```

```
    private createInjector<TData>(
        appModuleRef: NgModuleRef<any>,
        componentRef: PlanetComponentRef<TData>
    ): PortalInjector {
        const injectionTokens = new WeakMap<any, any>([[PlanetComponentRef, componentR
f]]);
        const defaultInjector = appModuleRef.injector;
        return new PortalInjector(defaultInjector, injectionTokens);
    }

    // 获取container内component的HTML元素
    private getContainerElement(config: PlantComponentConfig): HTMLElement {
        if (!config.container) {
            throw new Error(`config 'container' cannot be null`);
        } else {
            if ((config.container as ElementRef).nativeElement) {
                return (config.container as ElementRef).nativeElement;
            } else {
                return config.container as HTMLElement;
            }
        }
    }

    // 创建包装元素
    private createWrapperElement(config: PlantComponentConfig) {
        const container = this.getContainerElement(config);
        // 创建div
        const element = this.document.createElement('div');
        // 拿到应用PlantApplication
        const subApp = this.applicationService.getAppByName(this.ngModuleRef.instance.appN
me);
        // 加入 planet-component-wrapper 在classList中
        element.classList.add(componentWrapperClass);
        // 加入 attribute planet-inline
        element.setAttribute('planet-inline', '');
        // 如果设置中配置了wrapperClass
        if (config.wrapperClass) {
            // 加入
            element.classList.add(config.wrapperClass);
        }
        // 如果注册的时候加入了stylePrefix前缀
        if (subApp && subApp.stylePrefix) {
            // 加入到classList里
            element.classList.add(subApp.stylePrefix);
        }
        // container插入element
        container.appendChild(element);
        return element;
    }

    // 附加组件在页面上
    private attachComponent<TData>(
        plantComponent: PlanetComponent,
        appModuleRef: NgModuleRef<any>,
```

```
        config: PlantComponentConfig
    ): PlanetComponentRef<TData> {
        // 创建一个planetComponent引用
        // 注意里面有实例 componentInstance: TComp;
        // 有引用 componentRef: ComponentRef<TComp>;
        const plantComponentRef = new PlanetComponentRef();
        // 组件工厂解析器, 组件工厂解析器可以产生一个组件工厂(ComponentFactory)
        const componentFactoryResolver = appModuleRef.componentFactoryResolver;
        const appRef = this.applicationRef;
        // 创建一个自定义注入器,
        // 向入口组件提供自定义注入token时要使用的自定义注入器
        const injector = this.createInjector<TData>(appModuleRef, plantComponentRef);
        // 创建包装后的element
        const wrapper = this.createWrapperElement(config);
        // 如果有了相同的dom出口
        let portalOutlet = this.domPortalOutletCache.get(wrapper);
        if (portalOutlet) {
            // 那么卸载
            portalOutlet.detach();
        } else {
            // DomPortalOutlet 我理解成，容器？出口？站位dom元素。不知道对不对
            // 传递了element,组件工厂,模块引用,注入器
            portalOutlet = new DomPortalOutlet(wrapper, componentFactoryResolver, appRef, inj
ctor);
            // 存储此次添加的 dom出口
            this.domPortalOutletCache.set(wrapper, portalOutlet);
        }
        // 创建一个 填入出口 的 portalComponent 也就是去填占位符的ComponentPortal<自定义
件实例>
        const componentPortal = new ComponentPortal(plantComponent.component, null);
        // ComponentPortal<自定义组件实例> 填入出口中
        const componentRef = portalOutlet.attachComponentPortal<TData>(componentPortal);
        // 如果有初始化state
        if (config.initialState) {
            // 那么混入component的实例中去
            Object.assign(componentRef.instance, config.initialState);
        }
        // 传递一些引用
        plantComponentRef.componentInstance = componentRef.instance;
        plantComponentRef.componentRef = componentRef;
        plantComponentRef.wrapperElement = wrapper;
        // 注册卸载函数
        plantComponentRef.dispose = () => {
            // 删除缓存的出口element
            this.domPortalOutletCache.delete(wrapper);
            // 从dom中清除出口/占位符?
            portalOutlet.dispose();
        };
        // 返回 填上出口的 component引用 PlanetComponentRef 类型
        return plantComponentRef;
    }

    private registerComponentFactory(componentOrComponents: PlanetComponent | PlanetC
mponent[]) {
```

```
        // 获取app名称
        const app = this.ngModuleRef.instance.appName;
        // 拿到当前应用
        this.getPlantAppRef(app).subscribe(appRef => {
--- registerComponentFactory
// 传入匿名函数,函数两个参数:组件名称,config.返回一个PlanetComponentRef
>export type PlantComponentFactory = <TData, TComp>(
    componentName: string,
    config: PlantComponentConfig<TData>
) => PlanetComponentRef<TComp>;
---
                // 将 componentOrComponents转成数组
                const components = coerceArray(componentOrComponents);
                // 找到 与componentName名称一样的组件
                const component = components.find(item => item.name === componentName);
                // 如果存在
                if (component) {
                    // 在变更检测范围内
                    return this.ngZone.run(() => {
                        // 附加组件，然后返回PlanetComponentRef匿名函数结束
                        // 此时，planet-application-ref的factory为注册组件的factory
                        // 然后进行组件附加,传入component,模块引用，组件设置
                        const componentRef = this.attachComponent<any>(component, appRef.app
oduleRef, config);
                        return componentRef;
                    });
                } else {
                    throw Error(`unregistered component ${componentName} in app ${app}`);
                }
            });
        });
    }

    // 注册，
    register(components: PlanetComponent | PlanetComponent[]) {
        setTimeout(() => {
            this.registerComponentFactory(components);
        });
    }

    // 加载,传入app名称,组件名称,配置项
    load<TComp = unknown, TData = unknown>(
        app: string,
        componentName: string,
        config: PlantComponentConfig<TData>
    ): Observable<PlanetComponentRef<TComp>> {
        // 获取应用的引用
        const result = this.getPlantAppRef(app).pipe(
            //将源的发射延迟可观察到的时间间隔 由另一个 Observable 的发射确定。
            delayWhen(appRef => {
                if (appRef.getComponentFactory()) {
                    return of();
                } else {
                    // Because register use 'setTimeout',so timer 20
```

```
                return timer(20);
            }
        }),
        map(appRef => {
            // 拿到引用中的组件工厂
            const componentFactory = appRef.getComponentFactory();
            if (componentFactory) {
                // 这里传入了componentName和config,从而调用了入口出口填补组件的函数`attachC
mponent`，比较巧妙
                return componentFactory<TData, TComp>(componentName, config);
            } else {
                throw new Error(`${app}'s component(${componentName}) is not registered`);
            }
        }),
        finalize(() => {
            // 变更检测
            this.applicationRef.tick();
        }),
        shareReplay()
    );
    result.subscribe();
    return result;
  }
}
```

# 事件注册

简单过一下事件注册

```
// global-event-dispatcher.ts
export interface GlobalDispatcherEvent {
  name: string;
  payload: any;
}

const CUSTOM_EVENT_NAME = "PLANET_GLOBAL_EVENT_DISPATCHER";

@Injectable({
  providedIn: "root",
})
export class GlobalEventDispatcher {
  // ob
  private subject$: Subject<GlobalDispatcherEvent> = new Subject();
  // 是否加入了全局事件监听
  private hasAddGlobalEventListener = false;
  // 订阅总数
  private subscriptionCount = 0;

  private globalEventListener = (event: CustomEvent) => {
    this.subject$.next(event.detail);
  };

  // 加入全局事件监听
  private addGlobalEventListener() {
```

```
  this.hasAddGlobalEventListener = true;
  window.addEventListener(CUSTOM_EVENT_NAME, this.globalEventListener);
}

// 删除全局事件监听
private removeGlobalEventListener() {
  this.hasAddGlobalEventListener = false;
  window.removeEventListener(CUSTOM_EVENT_NAME, this.globalEventListener);
}

constructor(private ngZone: NgZone) {}

// 发射数据,你可以通过 name进行数据的区分,发射的应该是主动方
dispatch<TPayload>(name: string, payload?: TPayload) {
  window.dispatchEvent(
      // 通过CUSTOM_EVENT_NAME 找到了 globalEventListener函数
      // 然后将detail传入函数，所以现在subject中接收到值了
    new CustomEvent(CUSTOM_EVENT_NAME, {
      detail: {
        name: name,
        payload: payload,
      },
    })
  );
}

// 注册事件 返回ob对象, 注册的应该是被动方
register<T>(eventName: string): Observable<T> {
  return new Observable((observer) => {
    // 如果没有全局事件监听，那么先添加
    if (!this.hasAddGlobalEventListener) {
      this.addGlobalEventListener();
    }
    this.subscriptionCount++;
    // subscription 用来去掉订阅用的
    const subscription = this.subject$
      .pipe(
        filter((event) => {
          // 过滤出eventName一致的payload
          return event.name === eventName;
        }),
        map((event) => {
          // 返回payload
          return event.payload;
        })
      )
      .subscribe((payload) => {
        // 发射payload让订阅的地方得到值
        this.ngZone.run(() => {
          observer.next(payload);
        });
      });
    return () => {
      this.subscriptionCount--;
```

```
        subscription.unsubscribe();
        if (!this.subscriptionCount) {
          this.removeGlobalEventListener();
        }
      };
    });
  }

  getSubscriptionCount() {
    return this.subscriptionCount;
  }
}
```

看一个demo

```
// 被动方,等待name=openADetail的event发射值
        this.globalEventDispatcher.register('openADetail').subscribe(event => {
            this.thyDialog.open(ADetailComponent);
        });
//  name=openADetail 发射空值,此时上一个函数的 this.thyDialog.open(ADetailComponent);将调用
        this.globalEventDispatcher.dispatch('openADetail');
```

# END

结束,鄙人技术有限,有些地方不太准确(太不准确),不过按照这个思路将项目走一遍可以对 ngx-planet 一个比较清晰的认识，有问题可<a href="mailto:harlancui@outlook.com">邮件</a>联系，或在 g talk 中直接回复