/\*\* https://github.com/SAP/spartacus

\* 全局配置注入令牌，可用于将配置注入到应用程序的任何部分

\*/

export const Config = new InjectionToken('Configuration');

/\*\*

\* 配置块令牌，可用来提供配置块并贡献给全局配置对象。

\* 不应该直接使用，使用' provideConfig '或导入' ConfigModule。withConfig”代替。

\*/

export const ConfigChunk = new InjectionToken('ConfigurationChunk');

/\*\*

\* 配置块令牌，可用于提供配置块和贡献默认配置。

\* 一般规则是，库中提供的所有配置都应该作为默认配置提供。

\*/

export const DefaultConfigChunk = new InjectionToken(

'DefaultConfigurationChunk'

);

/\*\*

\* 使用ConfigChunk令牌提供配置块的帮助函数

\* 要在库中提供默认配置，应该使用defaultconfig。

\* @param config Config object to merge with the global configuration

\*/

export function provideConfig(

config: any = {},

defaultConfig = false

): Provider {

return {

provide: defaultConfig ? DefaultConfigChunk : ConfigChunk,

useValue: config,

multi: true,

};

}

/\*\*

\* 帮助函数使用工厂函数提供配置，使用ConfigChunk令牌

\* 要在提供的库中提供默认配置，应该使用defaultconfigfactory。

\* @param configFactory Factory Function that will generate config object

\* @param deps Optional dependencies to a factory function

\*/

export function provideConfigFactory(

configFactory: Function,

deps?: any[],

defaultConfig = false

): Provider {

return {

provide: defaultConfig ? DefaultConfigChunk : ConfigChunk,

useFactory: configFactory,

multi: true,

deps,

};

}

export function provideDefaultConfig(config: any = {}): Provider {

return {

provide: DefaultConfigChunk,

useValue: config,

multi: true,

};

}

/\*\*

\* 使用DefaultConfigChunk令牌为工厂函数提供默认配置

\* @param configFactory Factory Function that will generate config object

\* @param deps Optional dependencies to a factory function

\*/

export function provideDefaultConfigFactory(

configFactory: Function,

deps?: any[]

): Provider {

return {

provide: DefaultConfigChunk,

useFactory: configFactory,

multi: true,

deps,

};

}

/\*\*

\* 合并所有配置块的工厂函数。没有明确的理由不能直接使用。

\*/

export function configurationFactory(

configChunks: any[] = [],

defaultConfigChunks: any[] = []

) {

const config = deepMerge(

{},

...(defaultConfigChunks ? defaultConfigChunks : [] ),

...(configChunks ? configChunks : []),

// ...(defaultConfigChunks ?? []),

// ...(configChunks ?? [])

);

return config;

}

//

export const appConfig: AppConfig = {

rootUrl: env.appApi.baseUrl,

};

export const APP\_CONFIG = new InjectionToken<AppConfig>("app.config");

@NgModule({

imports: [CommonModule],

declarations: [],

})

export class ConfigModule {

/\*\*

\* 导入ConfigModule并将config贡献给全局配置

\* 要在库中提供默认配置，应该使用defaultconfig。

\* @param config Config object to merge with the global configuration

\*/

static withConfig(config: object): ModuleWithProviders<ConfigModule> {

return {

ngModule: ConfigModule,

providers: [provideConfig(config)],

};

}

/\*\*

\* 导入ConfigModule并使用工厂函数将config贡献给全局配置

\* @param configFactory Factory function that will generate configuration

\* @param deps Optional dependencies to factory function

\*/

static withConfigFactory(

configFactory: Function,

deps?: any[]

): ModuleWithProviders<ConfigModule> {

return {

ngModule: ConfigModule,

providers: [provideConfigFactory(configFactory, deps)],

};

}

/\*\*

\* 模块与提供者，应该只导入一次，如果可能的话，在应用程序的根。

\*/

static forRoot(config: any = {}): ModuleWithProviders<ConfigModule> {

return {

ngModule: ConfigModule,

providers: [ provideConfig(config),

{ provide: Config, useFactory: configurationFactory,

deps: [ [new Optional(), ConfigChunk], [new Optional(), DefaultConfigChunk], ],

},

{ provide: APP\_CONFIG, useValue: appConfig }

],

};

}

}

/\*

ConfigInitializerModule

\*/

import { ConfigInitializerService } from './config-initializer.service';

import { CONFIG\_INITIALIZER, CONFIG\_INITIALIZER\_FORROOT\_GUARD, ConfigInitializer,

} from './config-initializer';

export function configInitializerFactory(

configInitializer: ConfigInitializerService,

initializers: ConfigInitializer[]

) {

const isReady = () => configInitializer.initialize(initializers);

return isReady;

}

@NgModule({})

export class ConfigInitializerModule {

static forRoot(): ModuleWithProviders<ConfigInitializerModule> {

return {

ngModule: ConfigInitializerModule,

providers: [

{ provide: CONFIG\_INITIALIZER\_FORROOT\_GUARD, useValue: true, },

{

provide: APP\_INITIALIZER, multi: true, useFactory: configInitializerFactory,

deps: [ ConfigInitializerService, [new Optional(), CONFIG\_INITIALIZER], ],

},

],

};

}

}

/\*

ConfigValidatorModule

\*/

import { CONFIG\_INITIALIZER\_FORROOT\_GUARD, ConfigInitializer,

} from './config-initializer';

import { Config } from '../config.module';

import { deepMerge } from 'zyapp/core/config/utils/deep-merge';

/\*\*

\* 提供对CONFIG\_INITIALIZERS的支持

\*/

@Injectable({

providedIn: 'root',

})

export class ConfigInitializerService {

protected ongoingScopes$ = new BehaviorSubject<string[]>(undefined);

/\*\*

\* Returns true if config is stable, i.e. all CONFIG\_INITIALIZERS resolved correctly

\* 如果配置是稳定的，则返回true，即所有的CONFIG\_INITIALIZERS都被正确解析了

\*/

get isStable(): boolean {

return (

!this.initializerGuard ||

(this.ongoingScopes$.value && this.ongoingScopes$.value.length === 0)

);

}

constructor(

@Inject(Config) protected config: any,

@Optional()

@Inject(CONFIG\_INITIALIZER\_FORROOT\_GUARD)

protected initializerGuard

) {}

/\*\*

\* 在app完成初始化(APP\_INITIALIZERS, selected service构造函数)之前运行代码配置的推荐方法

\*

\* 在没有参数的情况下使用，等待整个配置变得稳定

\* 'siteContext', 'siteContext.language', etc.

\* 参数允许使用描述配置部分的字符串来描述配置的哪一部分应该是稳定的，

\* 例如:'siteContext'， 'siteContext。语言”等。

\*

\* @param scopes String describing parts of the config we want to be sure are stable

\* 参数范围字符串描述的配置部分，我们要确保是稳定的

\*/

async getStableConfig(...scopes: string[]): Promise<any> {

if (this.isStable) {

return this.config;

}

return this.ongoingScopes$

.pipe(

filter(

ongoingScopes =>

ongoingScopes && this.areReady(scopes, ongoingScopes)

),

take(1),

mapTo(this.config)

)

.toPromise();

}

/\*\*

\* 从当前正在运行的作用域中移除所提供的作用域

\*/

protected finishScopes(scopes: string[]) {

const newScopes = [...this.ongoingScopes$.value];

for (const scope of scopes) {

newScopes.splice(newScopes.indexOf(scope), 1);

}

this.ongoingScopes$.next(newScopes);

}

/\*\*

\* 如果提供的作用域不是ongoingscope的一部分，则返回true

\*/

protected areReady(scopes: string[], ongoingScopes: string[]): boolean {

if (!scopes.length) {

return !ongoingScopes.length;

}

for (const scope of scopes) {

for (const ongoingScope of ongoingScopes) {

if (this.scopesOverlap(scope, ongoingScope)) {

return false;

}

}

}

return true;

}

/\*\*

\* 检查两个范围是否重叠

\*范围重叠的例子:

\*

\* 范围不重叠的例子:

\* @param a ScopeA

\* @param b ScopeB

\*/

protected scopesOverlap(a: string, b: string): boolean {

if (b.length > a.length) {

[a, b] = [b, a];

}

return a.startsWith(b) && (a[b.length] || '.') === '.';

}

/\*\*

\* @internal

\* 不是公共API的一部分，APP\_INITIALIZER使用它来初始化所有提供的config\_initialalizer

\*/

async initialize(initializers?: ConfigInitializer[]) {

if (this.ongoingScopes$.value) {

// guard for double initialization

return;

}

const ongoingScopes: string[] = [];

const asyncConfigs: Promise<void>[] = [];

for (const initializer of initializers || []) {

if (!initializer) {

continue;

}

if (!initializer.scopes || !initializer.scopes.length) {

throw new Error('CONFIG\_INITIALIZER should provide scope!');

}

if (isDevMode() && !this.areReady(initializer.scopes, ongoingScopes)) {

console.warn(

'More than one CONFIG\_INITIALIZER is initializing the same config scope.'

);

}

ongoingScopes.push(...initializer.scopes);

asyncConfigs.push(

(async () => {

deepMerge(this.config, await initializer.configFactory());

this.finishScopes(initializer.scopes);

})()

);

}

this.ongoingScopes$.next(ongoingScopes);

if (asyncConfigs.length) {

await Promise.all(asyncConfigs);

}

}

}

export const CONFIG\_INITIALIZER = new InjectionToken('ConfigInitializer');

/\*\*

\* Used to provide asynchronous config during app initialization

\*/

export interface ConfigInitializer {

/\*\*

\* List of config parts that are resolved by configFactory, e.g.:

\* 'context.baseSite', 'context.language'

\*/

scopes: string[];

/\*\*

\* Promise that returns config chunk

\*/

configFactory: () => Promise<any>;

}

export const CONFIG\_INITIALIZER\_FORROOT\_GUARD = new InjectionToken<void>(

'CONFIG\_INITIALIZER\_FORROOT\_GUARD'

);

import { ConfigInitializerService } from 'zyapp/core/config/config-initializer/config-initializer.service';

import { ConfigValidator, ConfigValidatorToken, validateConfig,} from './config-validator';

export function configValidatorFactory(

configInitializer: ConfigInitializerService,

validators: ConfigValidator[]

) {

return () => {

if (isDevMode()) {

configInitializer

.getStableConfig()

.then(config => validateConfig(config, validators || []));

}

};

}

@NgModule()

export class ConfigValidatorModule {

static forRoot(): ModuleWithProviders<ConfigValidatorModule> {

return {

ngModule: ConfigValidatorModule,

providers: [

{

provide: APP\_INITIALIZER,

multi: true,

useFactory: configValidatorFactory,

deps: [

ConfigInitializerService,

[new Optional(), ConfigValidatorToken],

],

},

],

};

}

}

export function isObject(item: any): boolean {

return item && typeof item === 'object' && !Array.isArray(item);

}

export function deepMerge(target = {}, ...sources: any[]): any {

if (!sources.length) {

return target;

}

const source = sources.shift() || {};

if (isObject(target) && isObject(source)) {

for (const key in source) {

if (source[key] instanceof Date) {

Object.assign(target, { [key]: source[key] });

} else if (isObject(source[key])) {

if (!target[key]) {

Object.assign(target, { [key]: {} });

}

deepMerge(target[key], source[key]);

} else {

Object.assign(target, { [key]: source[key] });

}

}

}

return deepMerge(target, ...sources);

}