モジュール

目次

(サンプルコードディレクトリ：4_PM)

1.モジュールとは

モジュール化とは

機能単位などでコードを分割し、管理しやすいサイズでファイル管理していくこと。基本的に 1 モジュール 1 ファイルで管理する。開発規模が大きくなった際に有効。

使い方概要

- 外部(別ファイル)に提供したい要素を`export`する
- 別ファイルから参照する場合`import`する

モジュール化しないデメリット

- 1 ファイルのコード量が多くなり、コードの見通しが悪くなる
- 変数の競合リスクが高まる
- メソッドの競合リスクが高まる

モジュール化するメリット

- ファイルがコンパクトになり、コードの見通しがよくなる
- 変数、メソッドの競合リスクが軽減される

モジュールの基本文法

- エクスポート
    - ファイルの中の変数、関数、クラスをエクスポートすると、他のファイルからそれらが利用可能となる。 エクスポートを行う方法は`export`キーワードをそれぞれの要素の前に付与する。
- インポート
    - エクスポートしたものは`import`で取り込む。

モジュールのコード例

- Main.ts(Main モジュールを定義)

```
const TITLE: string = "TypeScript";

export function helloMessage(): void {
  console.log(`Hello,${TITLE}!`);
}

export class TsVersion {
  static getVersion(): string {
    return "4.9.5";
  }
}
```

- module_basic.ts

```
helloMessage(); //結果：hello,TypeScript!
console.log(TsVersion.getVersion()); //結果：4.9.5
```

2.モジュールの使い方-import 時に別名を付与

import 時にas句を利用することで、モジュール配下の個々のメンバーに別名を付与 エクスポートされた名前のままだと、ファイル内で名前が重複しそうな場合など別名をつける。

コード例

- module_name.ts

```
//Mainモジュールをインポート
import { helloMessage as message, TsVersion as version } from "./Main";
message(); //結果：hello,TypeScript!
console.log(version.getVersion()); //結果：4.9.5
```

3.モジュールの使い方-モジュール配下のメンバーを全てインポート

*を使い、モジュール配下のメンバーをまとめてインポート

コード例

- module_every.ts

```
//Mainモジュールをインポート
import * as every from "./Main";
every.helloMessage(); //結果：hello,TypeScript!
console.log(every.TsVersion.getVersion()); //結果：4.9.5
```

4.モジュールの使い方-default エクスポートとインポート

モジュール配下のひとつのメンバーに対しての場合、defaultエクスポートを設定することも可能

コード例

- Main2.ts

```
export default class {
  static getVersion(): string {
    return "2.0.0";
  }
}
```

- module_default.ts

```
import main from "./Main2";
console.log(main.getVersion()); //結果：2.0.0
```

> defaultのエクスポートとdefault以外のエクスポートは両立できる。

5.モジュールの使い方-動的インポート

`import`/`export`は、コードの実行開始時に**全ての必要な情報へのアクセスが可能である**という前提で処理される。 動的インポートとは読み込みタイミングをコントロールすることを指す。巨大な EC サイトなどで、特定のページのみ使用するスクリプトを後から読み込み、初期ロード時間を削減する時などに使用する。

動的インポートのポイント

- `import`関数の戻り値は`promise`オブジェクト(非同期呼び出し)
- インポートの後続処理は`then`メソッドを利用

コード例

- `module_then.ts`

```
import("./Main").then((main) => main.helloMessage()); //結果：
Hello,TypeScript!
```

> then メソッド配下では main 経由でモジュールの機能にアクセスできる

`async`関数の配下であれば、以下のような呼び出しも可能

```
async function app() {
  let main = await import("./Main");
  main.showMessage();
}
```

6.モジュールの使い方-リネイムして export

単一ファイルにおける export

asを使用し、エクスポート時にリネイムし、エクスポートすることができる。

```
class TsVersion{...}
//そのままエクスポート
export {TsVersion};

//リネイムしてエクスポート
export {TsVersion as TsInfo}
```

複数ファイル内容をまとめて export

TypeScript で大規模なアプリを作成する場合、1 ファイルで全て実装することはない。アプリケーションから読み込まれるエントリーポイントとなるスクリプトを 1 つ書き、外部に公開したい要素をそこから再エクスポートすることにより、他の各ファイルに書かれた要素を集約することができる。

- module_export.ts

```
export { helloMessage, TsVersion } from "./Main";
export { default as main } from "./Main2";
```

- module_import.ts

```
import * as all from "./module_export";

all.helloMessage();
console.log(all.TsVersion.getVersion());

console.log(all.main.getVersion());
```

7.モジュールの使い方-モジュールの検索方法

相対インポート

- 「/」、「./」、「../」で始まるモジュール参照。
- 現在のコードが/root/src/app.ts/である場合はimport {...} from "./Hoge"は以下の順番でHogeモジュールを検索

```
1. /root/src/Hoge.ts 、Hoge.d.ts
2. /root/src/Hoge/package.json(package.jsonにtypesプロパティが指定されている場合)
3. /root/src/Hoge/index.ts 、index.d.ts
```

非相対インポート

- 「/」、「./」、「../」以外で始まるモジュール参照
- 現在のコードが/root/src/app.ts/である場合はimport {...} from "Hoge"は以下の順番でHogeモジュールを検索

```
1. /root/src/node_modules/Hoge.ts 、Hoge.d.ts
2. /root/src/node_modules/Hoge/package.json(package.jsonにtypesプロパティが指定されている場合)
3. /root/src/node_modules/Hoge/index.ts 、index.d.ts
4. /root/node_modules/Hoge.ts 、Hoge.d.ts
5. /root/node_modules/Hoge/package.json(package.jsonにtypesプロパティが指定されている場合)
6. /root/node_modules/Hoge/index.ts、index.d.ts
7. /node_modules/Hoge.ts、Hoge.d.ts
8. /node_modules/Hoge/package.json(package.jsonにtypesプロパティが指定されている場合)
9. /node_modules/Hoge/index.ts、index.d.ts
```

8.モジュールの使い方-名前空間

ひとつのファイルの中でスコープを分離する。`namespace`を使うと、同じファイルの中で階層化された名前空間を作ることができる。

- `module_ms.ts`

```
//NetWork名前空間を定義
namespace NetWork {
  export class Http {}
  export function https() {}
}

//名前空間配下のクラス/関数の呼び出す
let http = new NetWork.Http();
NetWork.https();
```

- ポイント
  - `namespace`の中で定義したクラスなどはデフォルトではその外(名前空間配下外)からは見えないようになっている
  - 外からアクセスする場合は定義の前に`export`をつけて公開する
  - 外から呼び出す場合、「名前空間.クラス名()」のような完全な名前(完全修飾名)で表す

> 名前空間はコードをカプセルかするために望ましい方法ではない。名前空間を使用すべきかモジュールを使用すべきか確信が持てない場合はモジュールを選ぶとよい。(モジュールの分離、明示的な依存関係、読みやすさの観点からモジュールがよい)

9.ジェネリック型

ジェネリック型の概要

- 共通化するための手段
- 汎用的なクラス/メソッドに対して特定の型を紐づける。
- 型を抽象化し、同じデータ構造をもっているものに使う
- 関数の引数と似ており、定義したタイミングでは、どんな型かは決まっておらず実際に使用する際(呼び出すとき)に型定義し、決まる。

ジェネリック型の書き方

関数名やクラス名の直後に`<T>`のような型引数を付与する

> 型引数はあくまであとから型を受けるための仮引数なので、名前は自由に決められる。慣例的には`Type`や`Element`などを意味する`T`や`E`を用いる

ジェネリック型を使わない書き方(string)

- コード例

```
const stringBond = (ary1: string[], val: string): string[] => {
  let sum = val;
  return ary1.map((val) => sum + val);
};
console.log("stringBond:", stringBond(["abc", "def"], "text:"));
```

- ログ結果

```
stringBond: ["text:abc", "text:def"];
```

ジェネリック型を使わない書き方(number)

- コード例

```
const numberBond = (ary1: number[], val: number): number[] => {
  let sum = val;
  return ary1.map((val) => sum + val);
};
console.log("numberBond:", numberBond([100, 200], 300));
```

- ログ結果

```
numberBond: [400, 500];
```

ジェネリック型を使い型を抽象化する

- コード例

```
type Bond<T> = { (ary1: T[], val: T): T[] };

const generitStringBond: Bond<string> = (ary1, val) => {
  let sum = val;
  return ary1.map((val) => sum + val);
};
console.log("generitStringBond:", generitStringBond(["abc", "def"],
"text:"));

const generitNumberBond: Bond<number> = (ary1, val) => {
  let sum = val;
  return ary1.map((val) => sum + val);
};
console.log("generitNumberBond:", generitNumberBond([100, 200], 300));
```

- ログ結果

```
generitStringBond: ["text:abc", "text:def"];
generitNumberBond: [400, 500];
```

- サンプルコード：4_am_samplecode_3.ts

> リファクタリング時に使うイメージ。最初から使おうとしなくてもよい。