

モジュール

目次

1. [モジュールとは](#)
2. [モジュールの使い方-import 時に別名を付与](#)
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](#)
4. [モジュールの使い方-default エクスポートとインポート](#)
5. [モジュールの使い方-動的インポート](#)
6. [モジュールの使い方-リネームして export](#)
7. [モジュールの使い方-モジュールの検索方法](#)
8. [モジュールの使い方-名前空間](#)
9. [ジェネリック\(総称型\)](#)

(サンプルコードディレクトリ : [4_AM_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-リネームして 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ームし、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};  
  
//リネー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化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。

様々なジェネリック記法

呼び出しシグネチャの記法とジェネリック型の割り当て範囲によって書き方が変わる

- 完全な呼び出しシグネチャ(シグネチャ全体にジェネリック型を割り当てる)

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

- 完全な呼び出しシグネチャ(個々のシグネチャにジェネリック型を割り当てる)

```
type Bond2 = {  
  <T>(ary1: T[], val: T): T[];  
  <U>(ary1: U[], val: U): U[];  
};
```

- 呼び出しシグネチャの省略記法

```
type Bond3<T> = (ary1: T[], val: T) => T[];  
type Bond4 = <T>(ary1: T[], val: T) => T[];
```

型引数に制約をつける

パラメーターに対して制約をつけたい場合は`extends`を使用する

`class` を継承する際に使用する`extends`とは意味が少し異なる。拡張というような意味ではなくどちらかということインターフェイスを実装する`implements`(狭める)という意味に近い

- 関数に型引数の制約をつける

```
function Username<T extends { name: string }>(value: T): T {  
  return value;  
}  
console.log(Username({ name: "Lick" }).name); //Lick
```

- `class` に型引数の制約をつける

```
class Mygeneric<T extends string | number> {  
  private data: T[] = [];  
  add(item: T) {  
    this.data.push(item);  
  }  
  get() {  
    return this.data;  
  }  
}  
const mygeneric1 = new Mygeneric<string>();  
mygeneric1.add("apple");  
mygeneric1.add("orange");  
console.log(mygeneric1.get());  
  
const mygeneric2 = new Mygeneric<number>();  
mygeneric2.add(1);  
mygeneric2.add(2);  
console.log(mygeneric2.get());
```

- インターフェイスにジェネリクスを使用する

```
interface UserGroup<T> {  
  id: number;  
  data: T[];  
}  
  
const userinfo1: UserGroup<string> = {  
  id: 1,  
  data: ["Lick", "Hinoko"],  
};
```

- タイプエイリアスにジェネリクスを使用する

```
type UserGroup<T> = {  
  id: number;  
  data: T[];  
};  
  
const userinfo1: UserGroup<string> = {  
  id: 1,  
  data: ["Lick", "Hinoko"],  
};
```

Utility 型

import をしなくても使用できる TypeScript に内蔵されている型のライブラリ。

- **Partial**: 全てオプションプロパティで返してくれるもの
- **Readonly**: 引数をすべて読み取り専用にしてくれる

```
interface Info {
  id: number;
  firstName: string;
  lastName: string;
}
type userInfo = Partial<Info>;
type ReaduserInfo = Readonly<Info>;

const promisRes: Promise<string> = new Promise((resolve) => {
  setTimeout(() => {
    resolve("hello");
  }, 5000);
});

promisRes.then((data) => {
  data.toUpperCase();
});

const fruits: Array<string> = ["apple", "banana", "peach"];
```

型引数に初期値を設定する

型引数にイコールをつけることで初期値を設定することができる

```
interface ResData<T extends { [labelName: string]: string } = any> {
  data: T;
  status: number;
}
let res1: ResData;
```