ChatGPT

TypeScript型定義ファイルの互換性問題とモジュール解決の落とし穴

TypeScriptでライブラリの型定義ファイル（`*.d.ts`）を下流プロジェクトに提供（いわゆる「注入」）する際には、モジュールの定義方法や各種設定の違いによって**型解決の互換性問題**や思わぬ挙動の違いが発生します。本稿では、それらの問題点を整理し、公式仕様や実務知見を踏まえた対処法やベストプラクティスを解説します。

モジュール定義形式の違いと型解決への影響

TypeScriptの型定義では、モジュールのエクスポート形式によって記述方法が異なります。特に**CommonJS**形式と**ES Module**形式の違い、および`esModuleInterop`等のコンパイラオプションの影響に注意が必要です。

- **`export =` と CommonJS**: 実装が`module.exports = ...`で単一の値（関数・クラス・オブジェクトなど）をエクスポートする場合、型定義では`export =`構文を使います[1]。`export =`はモジュール自体がその値を表す形式で、**デフォルトエクスポートではありません**。この型定義を消費する側では、`import モジュール = require("...")`という形式か、`"esModuleInterop": true`（または`allowSyntheticDefaultImports`）を有効にした上でデフォルトインポートする必要があります。

- **`export default` と ES Modules**: 実装がESMの`export default`構文を使う場合、型定義でも`export default`で表現します。`export default`は1モジュールにつき1つの主要なエクスポートを示し、`import X from "mod"`でインポート可能です。ただし実装がCommonJSなのに型定義を`export default`にしてしまうと不整合が生じます（後述の`esModuleInterop`の項目参照）。

- **名前付き`export`**: 実装が`module.exports = { foo, bar, ... }`のように複数のプロパティを持つオブジェクトをエクスポートしている場合、型定義では個々のプロパティを`export`で公開する方法がとられます[2]。この場合、`import { foo } from "mod"`のように名前付きインポートで利用できます。CommonJSモジュールでもエクスポートがオブジェクトの場合は、このように**複数の名前付きエクスポート**として型定義する方が、`export =`でオブジェクト全体をエクスポートするより型の流用性が高いです。

- **アンビエント宣言（`declare module`）**: ライブラリ側が型定義を提供していない場合や、既存モジュールに追加の型補強をする場合に、開発者が自前で「アンビエントモジュール宣言」を書くことがあります。例えば`declare module 'legacy-lib' { ... }`のように**既存モジュール名を指定**して中に型定義を書くと、そのモジュールの型情報を補います。ただし、これを使う際は**定義の競合**に注意が必要です。例えば、ライブラリが独自に`index.d.ts`を同梱するようバージョンアップした後も手元に古い`declare module`による定義が残っていると、重複定義エラーになることがあります[3]。アンビエント宣言は通常、プロジェクト内の`global.d.ts`等に置き、**外部モジュールの型をパッチ**する用途に留める方が無難です。また、`declare module`で指定する文字列は実際のインポートパスと正確に一致させる必要があります（不一致だと型が当たりません）。

- **`esModuleInterop`と`allowSyntheticDefaultImports`**: これらのコンパイラオプションは、CommonJSモジュールをES Module的に**柔軟にインポートできるようにするための設定**です。`"esModuleInterop": true`の場合、`export =`で定義された型を持つモジュールでも

`import foo from 'foo'`というデフォルトインポート構文が許容されます（トランスパイル時に`__importDefault`ヘルパーが挿入され、`module.exports`を`default`プロパティとして参照します[4]）。一方で、この互換機能により**型チェック上はエラーにならないコードが、実行時にエラーになる**ケースもあります[5]。たとえば、型定義が存在しないCJSモジュールをデフォルトインポートするとコンパイルは通りますが（`allowSyntheticDefaultImports`によりエラーを無視）、実行時には実際には`default`プロパティがないため`undefined`となりエラーになります[6]。逆にCommonJSモジュールを`import * as foo from 'foo'`と名前空間インポートした場合、型的には問題なくても、トランスパイル後の`__importStar`処理によって**オブジェクトのプロトタイプが欠落**し、実行時にメソッドが存在しないといった不具合が発生することがあります[6]（例: CJSが関数オブジェクトをエクスポートし、その関数にメソッドが付与されていると、`__importStar`で包んだ結果メソッドが消失する[7]）。このように、`esModuleInterop`有効時は**TypeScriptの型システム上は安全に見えても、実行環境の挙動とずれが生じる**ことがあるため、ライブラリの型定義を作成・利用する際は実装側のエクスポート形態に忠実に型を記述し、必要に応じて利用側でこのフラグを有効にする/しないを検討する必要があります。

補足: DefinitelyTypedのガイドラインでも、実装がCommonJSの場合は「型定義では`export =`を使い、決して`export default`にしない」ことが推奨されています[1]。例えば実装が`module.exports = function()`であれば、型定義は以下のようになります。

```
// JS実装 (CommonJS)
module.exports = function blabla(x) { ... }

// 型定義 (d.ts)
declare function blabla(x: any): any;
export = blabla;
```

この場合、利用側では`import blabla = require('my-module')`でインポートするのが正規のスタイルです[8]（もしくは`esModuleInterop`を有効にした上で`import blabla from 'my-module'`）。仮にここで誤って`export default`を使って型定義してしまうと、`import blabla from 'my-module'`はコンパイル上は通りますが、`esModuleInterop`が`false`のプロジェクトでは「モジュールにデフォルトエクスポートがない」というエラーになり、逆に`true`の場合は実行時にundefinedになる恐れがあります。

NodeNext/Node16とClassicのmoduleResolutionによるモジュール探索の違い

TypeScript 4.7以降、`tsconfig.json`の`moduleResolution`オプションに**`node16`**や**`nodenext`**が追加され、Node.jsの新しいESM解決アルゴリズムに沿ったモジュール解決がサポートされました。従来の`node`（別名`node10`相当）や`classic`モードとは挙動が大きく異なるため、互換性問題の温床となっています。

- **`node16`/`nodenext`モード**: これらは**Node.js v12+のモジュール解決規則**（ESMとCJSのデュアルサポート）をTypeScriptに取り入れたモードです[9]。具体的には、Node.jsがパッケージ内の`package.json`の`exports`フィールドやファイル拡張子によって「どのファイルを読み込むか」を決定するルールを、TypeScriptの型解決でも再現します。`nodenext`は将来のNode仕様変更も含めて追従する「最新追随モード」で、現時点では`node16`と同一ですが今後の拡張に備えた名称です[10]。**重要な点**として、`moduleResolution`を`node16`/`nodenext`にする場合、`compilerOptions.module`も対応する`Node16`や`NodeNext`に合わせなければエラーになります[10]（例えば`moduleResolution: "node16"`で`module: "CommonJS"`は不可）。

- **パッケージの `exports` / `imports` 対応**: Node16/Nextモードでは、`package.json` 内の `"exports"` や `"imports"` フィールドを解釈して型解決します [11]。従来の `node` モードではこれらを無視して `main` フィールドやファイルパスで探索していましたが、新モードでは**対応する設定でないと `exports` による参照が解決できません** [11]。たとえば、パッケージが `"exports"` でサブパスを定義している場合、`moduleResolution: "node16"` または `"nodenext"` でなければTypeScriptはそのモジュールを見つけられません。「Nodeのパッケージ参照（imports/exports）は `node16` / `nodenext` でのみサポートされる」という注意が公式にも明記されています [11]。

- **ファイル拡張子の要求**: Node.jsのESMでは、`import` 文で拡張子（.jsなど）が必須です。`node16` / `nodenext` モードではTypeScriptの型解決時にも**拡張子の有無チェック**が行われ、足りないとエラーとなります [12]。例えば、`.mts` や `.mjs` ファイルに対応する型定義 `.d.ts` を参照するとき、`import { foo } from "./util"` と書くと「拡張子を付けろ」というエラー(TS2835)が発生します [12]。実例では、`.d.ts` ファイル内の相対インポートに対して**「'--moduleResolution' が 'node16'のときは相対インポートパスに拡張子が必要です。'./util.js'と書きましたか？」**というエラーメッセージが報告されています [12]。拡張子を付ければエラーは消えますが、開発時のソースコード（.ts）には通常拡張子を書かないため、これを強要されることで**VSCode上の補完や移行時の混乱**が生じるケースがあります。

- **CJSとESM混在への厳格なチェック**: Node16/Nextモードでは、**出力されるJSコードにおけるimportかrequireか**をシミュレートして型チェックします。例えば、プロジェクトがCommonJS (`"module": "CommonJS"`)で `moduleResolution: "node16"` の場合、`import pkg from 'esm-package'` のようにESMパッケージを `require` しようとするとエラーTS1479が発生します [13]。これは「現在のファイルはCommonJSモジュールで、'pkg'はESModuleとして解決されたので `require` で読み込めない」という趣旨のエラーです [13]。実際のNode実行時には、該当パッケージが例えば `.cjs` エントリを用意して `require` に対応している場合でも、TypeScriptはパッケージの `package.json` の `"type": "module"` 等を見て「ESM扱い」と判断するとエラーにします [13]。この挙動により「Nodeでは実行できる組み合わせなのにTypeScriptではコンパイルエラーになる」ケースが存在します [13]。後述の例のように、パッケージ側が `exports` でCommonJS用エントリを用意していても、TypeScript 4.7現在はそれを考慮できずエラーとしてしまう既知の問題があります（詳細は後述の事例参照）。

- **`classic` モードの非推奨**: ちなみに `moduleResolution: "classic"` は古いTypeScript(1.6以前)の解決方式であり、現代的なプロジェクトでは使用すべきでないとされています [14]。ClassicモードはNodeの仕組みを無視した独自探索で、意図しない型が引っかかったりするため、互換性問題の観点でも推奨されません。

- **`bundler` モードとの比較**: `moduleResolution: "bundler"` はNode16/Nextと同様に `exports` / `imports` を理解しつつ、**拡張子のチェックを行わない**モードです [15]。これはWebpackやViteなどのバンドラが拡張子なしのimportを許容する（ビルド時に解決する）文化に合わせた設定です。`node16` だとエラーになるコードでも、`bundler` ならコンパイルが通ります。その代わり、TypeScriptは**ランタイム上の安全性を一部犠牲**にします。公式ドキュメントでも「`bundler` モードは他の環境では動かないコードを許容する伝染的なモード」であり、逆に `nodenext` は「Node.js上で動作することを保証するためのチェックを行うモード」だと述べられています [16]。実際、前述の拡張子問題に悩んだ開発者がVSCodeでの快適さを優先して `moduleResolution: "bundler"` に切り替えるケースもあります [15]（bundlerモードでは拡張子エラーが出ないためエディタ警告が消える）。ただし、このまま出力したコードをNode上で実行すると（ビルドやバンドラを介さない場合）**モジュールが見つからず実行時エラー**になるリスクがある点には注意が必要です。

まとめ: ライブラリの型定義を提供する側としては、Node16/Nextモードではパッケージの `"exports"` / `"type"` 設定やファイル拡張子が厳密に考慮されることを踏まえ、適切に型定義ファイル名や

参照を設計する必要があります。逆に利用者側では、自分のプロジェクト設定に合ったパッケージを選ぶ（または設定を調整する）ことが重要です。例えばNode　ESM環境を前提としたライブラリをCommonJSのプロジェクトで使う場合、型解決で苦労する可能性が高くなります。その場合は`moduleResolution`の設定変更や、可能であればライブラリ側が提供するCJS向けエントリ（`.cjs`ファイル）と型定義を利用する、といった対策が必要です。

型定義ファイルの配置、`types`フィールドと`typesVersions`の使い分け

ライブラリが提供する型定義ファイルの**配置場所**や、`package.json`の`"types"`フィールド、さらにTypeScriptのバージョン差異に対応するための`"typesVersions"`設定も互換性に影響します。正しく設定しないと「型が見つからない」「特定のTypeScriptバージョンで型解決に失敗する」といった問題が発生します。

- **`index.d.ts`の配置と`types`/`typings`フィールド**: パッケージが型定義を同梱する場合、通常はエントリポイントに対応する`index.d.ts`を用意し、`package.json`の`"types"`フィールドでそのパスを指定します。例えば、`"main": "./lib/main.js"`に対し`"types": "./lib/main.d.ts"`のように指定すると、TypeScriptはそのパッケージをインポートした際に`main.d.ts`を参照します [17] [18]。このフィールドが正しく設定されていないと、型定義ファイルが存在していても**TSから発見されず**`Could not find a declaration file for module 'xxx'`というエラー(TS7016)になります [19]。また、パスの書き方にも注意が必要で、相対パスを記述する際は先頭に`./`を付けるべきです（付けないと意図しない場所を起点に探索する可能性があります）。実際に、`"types": "dist/index.d.ts"`と書いていて型が解決しなかったケースで、`"types": "./dist/index.d.ts"`に直すことで直った例もあります [20]。

- **`dependencies`と`devDependencies`の適切な指定**:　型定義内で他のパッケージの型を参照している場合（例: 自分の型定義で`import { Something } from "other-package";`など）、その参照先が**型定義付きのパッケージか**否かに応じて、自パッケージの`package.json`における依存関係の指定を適切に行う必要があります [21]。もし参照先が型定義を同梱していない（`@types/〜`に頼る）場合、型定義パッケージを自パッケージの`dependencies`に含めなければ、利用者側で型が見つからずエラーになります [21]。例えば、自ライブラリが`pg`というモジュールの型を使用しているとします。`pg`は型定義を持たないため`@types/pg`が必要ですが、これを自ライブラリ側で`devDependencies`に入れて開発していると、利用者がそのライブラリを使ったときに`@types/pg`がインストールされず、TS上で`pg`関連の型が`any`になったりエラーが発生したりします [22] [23]。これはVSCode上では型が見えている（エディタが自動的に@typesを取得して補完する機能が働く場合がある）ため気づきにくいですが、ビルドすると`noImplicitAny`違反や型不足エラーになる典型例です [22]。したがって、**自パッケージの公開APIが依存する他パッケージの型**は、そのパッケージまたは対応する`@types`を`dependencies`に含める必要があります [21]（ライブラリとして利用されることを想定する場合）。

- **`typesVersions`の活用**:　TypeScriptのバージョン間で型定義の記法や型シグネチャを変える必要がある場合、`package.json`の`"typesVersions"`フィールドで複数の型定義セットを用意できます [24] [25]。例えば、TypeScript　3.1以降では新しい型定義（フォルダ`ts3.1/*`内）を使い、それより古いバージョンでは従来の型定義を使わせる、といったことが可能です [26] [27]。TypeScriptはコンパイラのバージョンに応じて`typesVersions`内のセマンティックバージョン範囲をマッチし、該当すれば指定されたパスに置き換えて型ファイルを読み込みます [27]。もしどの範囲にもマッチしなければ、通常の`types`フィールドのパスにフォールバックします [28]。この機能により、新旧互換のために型定義を分けたり、一部の型のみ差し替えたりできます。

- **`typesVersions`設定ミスの危険**:　`typesVersions`は便利な反面、設定ミスによる思わぬ副作用があります。典型的な失敗パターンは「`typesVersions`で指定した先が型定義ではなくソースコードになっている」ケースです [29] [30]。例えば、あるライブラリで`typesVersions`の対象を`"./src/*"`（生の

TSソース）に向けてしまったために、利用者のコンパイラがライブラリ内部のTSコードを直接読みに行ってしまい、利用者の`tsconfig`の厳格設定と食い違ってエラーが頻発した事例があります[31]。これは、本来ならコンパイル済みの`.d.ts`を読むべきところを、誤って実装コードを読ませてしまったためです。実装コードはコンパイル対象外のはずですが、このような設定ミスにより**利用者側でライブラリ内部実装が型チェックされてしまい**、strictモードの違い等からエラーが生じました[31]。さらにこの状況では`skipLibCheck`オプションも効かず、利用者はエラーを抑制できません[32]。解決策は、`typesVersions`を正しく**`.d.ts`ファイルを指すよう修正すること**です[32]。先の事例でも、`"./src/..."`から`"./dist/... .d.ts"`に変更したところ問題が解決しています[33]。このように、`typesVersions`を使う際は各バージョン向けに**適切な型定義ファイル**を用意し、参照先を間違えないようにしなければなりません。

- **ディレクトリ構造とサブパス**: 型定義ファイルをどのように配置するかも重要です。単純なライブラリなら`index.d.ts`一つで済みますが、サブモジュール（`import { X } from "lib/sub"`のようなパス）をエクスポートする場合、対応する型定義ファイルを用意する必要があります。一般的にはソースコードのディレクトリ構造に合わせて`.d.ts`を配置し、`types`フィールドではエントリーポイントのみ指します。TypeScriptはデフォルトでパッケージ内の`types`指定フォルダを起点に、サブパスを補完的に探します[34]。例えば`"types": "./index.d.ts"`かつ`import "pkg/sub"`があれば、`node_modules/pkg/sub.d.ts`や`node_modules/pkg/sub/index.d.ts`がないか探索します[34]。しかし、Node16モードでは`exports`フィールドに定義されたサブパス以外は基本見ないため、この自動探索は働きません。したがって**Node16/Next環境下では、サブパスごとに`exports`で型定義への参照を明示するか、typesVersionsでフォローする必要があります**（詳細は次項で説明）。

`exports`フィールドと`types`の干渉による落とし穴

Node.jsの`package.json`における`"exports"`フィールドは、モジュールごとに使用可能なエントリポイントを制御します。これは従来の`"main"`や`"module"`フィールドに代わり、ESM/CJSや環境ごとに適切なファイルを提供するための仕組みですが、TypeScriptの型解決と組み合わせると**いくつかの注意点**があります。

- **`"exports"`と型定義の不整合**: パッケージが`"exports"`を使う場合、TypeScriptもNode16/Nextモードでその指定に従います。しかし`"exports"`には通常、`"require"`や`"import"`向けにJSファイルを指定します。**型定義ファイル(`.d.ts`)への参照**も`"exports"`内で指定することが可能で、条件キーとして`"types"`を使います[35]。ベストプラクティスとして「`"exports"`内では`"types"`条件を最優先で記述すること」とされています[35]。例えば:

```
"exports": {
 ".": {
  "types": "./dist/index.d.ts",
  "import": "./dist/index.mjs",
  "require": "./dist/index.cjs"
 },
 "./sub": {
  "types": "./dist/sub.d.ts",
  "import": "./dist/sub.mjs",
  "require": "./dist/sub.cjs"
 }
}
```

このようにしておけば、TypeScriptは`import "pkg"`や`import "pkg/sub"`に対してそれぞれ適切な`.d.ts`を読み込みます。**問題は**、これを怠った場合に生じます。もし`"./sub"`のエクスポートに`"types"`指定がなかったり、型ファイルを配置していないと、TypeScriptは型を解決できずエラーとなったり`any`として扱っ

たりします。また`"types"`条件を書かず単に`"types"`フィールド（グローバルな型エントリ）だけで済ませた場合、サブパスについては**運頼みの探索**になるため、NodeNextモードでは見つけられない可能性が高いです。

- **デュアルエントリ（CJS/ESM）の型定義**: `"exports"`ではCJSとESMで別のファイルを指定できますが、**型定義は通常1つ**しか用意しません。例えば`"import"`がESM版、`"require"`がCJS版の実装を指していても、型定義は共通にするケースが多いです。そのため、CJS側では本来`export =`的な使い方をすべき所を、型定義上はESMスタイル（export defaultやnamed exports）に寄せて書く、といったことがあります。この場合、**CJSとして利用した場合に型と実装が乖離**する恐れがあります。幸い、Node.js自体がESMからCJSを`require`すると自動的にデフォルトエクスポート扱いする（CJSモジュールをESMから`import`すると、デフォルトにその`module.exports`を割り当てる）仕様があるため、多くの場合は一つの型定義で両対応できます。しかし、TypeScriptコンパイラ上は厳密にチェックするため、例えば先述のTS1479のようなケースが起こり得ます。「パッケージ側ではCJSもESMも動く実装なのに、TypeScriptはそれを許容しない」というのは、**型定義と`exports`/`package.json`設定の噛み合わせによる齟齬**です[13]。

- **具体例: requireとimportの両立失敗**: あるパッケージが`"type": "module"`（全体はESM想定）だが、`"exports"`で`.cjs`ファイルを`require`向けに提供しているとします。このとき、そのパッケージをCommonJSのプロジェクトで`require('pkg')`するとNodeは`.cjs`をちゃんと読み込みます。しかしTypeScriptでは、`moduleResolution: "node16"`下だと`require('pkg')`相当のコード（=`import pkg = require("pkg")`や`import pkg from "pkg"` with esModuleInterop false）を書いてもエラーになります[13]。前述のTS1479エラーの例[13]では、「出力がCommonJS（= requireになる）コードから、ESModuleのファイルを参照しようとした」と判断されています。TypeScriptは`package.json`の`"type": "module"`や`"exports"`内の`"import"`エントリを見て「この参照はESMモジュールを指す」とみなし、CJSコードからは不正と判断したわけです。実際には`"require"`エントリもあって`.cjs`を提供しているのですが、現状のTSコンパイラは**片方（import側）しか見ていない**挙動のようです[13][36]。このように、**モジュール解決の条件分岐をTypeScriptが完全には評価しきれず**、利用側コードに制限が課される場合があります。

- **`main`/`module`/`types`の不整合**: 古くからある`"main"`（CommonJSエントリポイント）と、Webpack等が見る`"module"`（ESMビルドのエントリ）フィールド、そして型定義用の`"types"`フィールドが指すファイルがそれぞれ噛み合っていないと、やはり問題が発生します。例えば、`main`はCJS版の`index.js`を指し、`module`はESM版の`index.mjs`を指すが、`types`が単一の`index.d.ts`しか指していない場合を考えます。この`index.d.ts`がCJS的記述かESM的記述かによって、どちらかの利用形態で不都合が起こりえます。理想的には、CJSとESMで**公開するAPIは同一**であるべきなので、一つの型定義で両方カバーできるはずですが、実装上そうでないケースでは型定義を分離する必要があります（が、それはかなり困難です）。多くのライブラリではビルド時にCJS/ESM両対応のコードを出力し、型定義もそれに合わせて作ります。その際、**exportの形式（defaultかnamedか）を揃える**ことが重要です。例えばESM版では`export default`だがCJS版では`module.exports =`であるような場合、型定義上は`export default`で書いておき、CJSから使う場合には`esModuleInterop`を前提とする、といった戦略がとられます。このあたりはライブラリ作者のポリシーによりますが、不整合があると利用者が混乱したり型エラーに遭遇したりします。

- **`exports`でのサブモジュールの型**: すでに触れたように、`exports`でサブパスを区切っている場合、各サブパスに対応する型定義を提供しないと利用者は型情報を得られません。特に`NodeNext`環境では`exports`に書かれていないパスは**型解決できない**ため、`exports`に現れない内部ファイルへのインポートをユーザが記述するとエラーになります。例えば、一部の利用者が`import "pkg/lib/internal"`のように非公開の深いパスを直接インポートしていた場合、`exports`で隠されたそのパスはNode実行時にはエラー（Module Not Found）になりますし、TypeScript NodeNextでも同様にエラーになります。しかし古い環境やバンドラではファイルを見つけてしまうこともあり（隠蔽が効かない場合がある）、その結果「実行時は動くのに型チェックだけ通らない/またはその逆」のような状

況も起こり得ます。この点でも、**パッケージ作者は`exports`で隠す/公開するパスの整合性と、型定義の範囲をよく検討**する必要があります。

DefinitelyTypedでの互換性確保パターン

コミュニティ主導の型定義リポジトリであるDefinitelyTypedでは、さまざまなライブラリの型定義が管理されています。そこで蓄積されているノウハウとして、**実装の差異やTypeScriptバージョン差による互換性問題に対処するパターン**がいくつか存在します。

- **`export =`と`export as namespace`の併用**: UMDモジュール（Nodeでもブラウザでも使えるグローバルもエクスポートするようなライブラリ）では、型定義内で`export =`（CommonJS的エクスポート）と`export as namespace`（グローバル名前空間としても利用可能）を組み合わせる例があります[37]。これにより、モジュールインポートしても良し、`<script>`タグ等で読み込んでグローバル変数として使っても良し、という柔軟な型提供が可能になります。

- **`export =`と同時にデフォルトエクスポート対応**: 一部の型定義では、実装がCommonJSで`module.exports`を使っているが、利用者の利便性のため**デフォルトエクスポートも許容**するケースがあります。TypeScriptでは直接同じモジュールで`export = X`と`export default X`を併記することはできません。しかし、`esModuleInterop`を有効にしていれば`export = X`の型定義でも`import X from 'mod'`が通るため、通常はそれで足ります[8]。どうしても両方の宣言を明示的にしたい場合は、**別ファイルを経由するテクニック**もあります。とはいえ、DefinitelyTypedでは基本的に実装に忠実な形式を採用し、無用なエクスポートは避ける方針です[1]。利用側が`allowSyntheticDefaultImports`を有効にすれば良い、というスタンスで、あえて`export default`を嘘の型として書くことはしないよう注意されています[38]。

- **型のバージョン分岐**: DefinitelyTypedでは`ts3.5`や`ts4.1`のようなフォルダを設けて、特定のTypeScriptバージョン以前向け/以降向けに型定義を分ける手法がとられることがあります。これは前述の`typesVersions`を活用したものです。古いコンパイラでは解釈できない新構文を含む型定義や、TSのバグ回避のための差分実装などを、コンパイラバージョンで出し分けています[26][27]。たとえば、TS4.5で導入された`Awaited`型に対応するために、TS4.3では別実装を使う、などの対応が行われました。ライブラリ開発者も、自前提供の型定義で同様のことが必要なら`typesVersions`を使うべきでしょう。

- **シム（shim）型定義**: 何らかの環境差異や将来的の変更に備えて、型だけを定義する「シム」を用意することがあります。例えばNode.jsでは`import fs from 'node:fs'`という形式（`node:`プレフィックス）が登場しましたが、古い@types/nodeには当然そのようなdeclareはありませんでした。そこで新旧両対応のため、`node:fs`モジュールを`fs`モジュールと同一視するdeclareを追加する、といった対応が行われました。このような型の「埋め木（shim）」は、コンパイラのエラーメッセージを解消したり、複数バージョン間の互換を持たせるために有効です[39]。ただし、不用意なshimは型の二重定義を招く可能性もあるため、DTでは必要最低限に留める方針です。

- **テストとStrictモード**: DefinitelyTyped上の型定義は、様々なコンパイラオプション（`strict`モードやターゲットバージョン）で型チェックテストが走るようになっています。これにより、前述したような**誤ったtypesVersions設定**（実装コードを参照してしまうもの）やstrictモードでの欠陥がないか監視されています。例えば、ある型定義が`strictNullChecks`でエラーを出すようならIssueが上がり、対応（strictモードでも通るようUnion型にする等）がなされます[31]。

以上のようなパターンは、ライブラリ作者が自前で型定義を管理する際にも参考になります。つまり、「実装に忠実な型定義」を心がけつつ、利用者のTS設定や将来の変化も見据えて調整することが重要です。

バンドラとTypeScriptのモジュール解決差異による問題

現代のフロントエンド開発ではWebpackやVite、Rollupなどの**バンドラ**を通してモジュールを読み込むケースが多く、それらとTypeScriptコンパイラの挙動の違いも互換性問題を引き起こすことがあります。特に、**型チェック時に通るか否か**と**実行時に動くか否か**が逆転してしまうケースが要注意です。

- **拡張子の扱い**: 前述の通り、TypeScriptはNodeNextモードでは拡張子必須ですが、WebpackやViteはインポート時に拡張子を省略しても解決してくれます。そのため、プロジェクトをESM化して`type: "module"`にした場合に、VSCode上では「拡張子を付けろ」という警告(TS2835)が出る一方で、Viteでのビルドや実行自体は拡張子なしでも通る、といった状況になります [40] [15]。このギャップに対処するために、開発者は`moduleResolution: "bundler"`に切り替えて警告を消す選択をすることがあります [15]。しかしながら、そうすると**コードをNode単体で動かしたときには失敗する**可能性が残ります。つまり、バンドラ経由ならOKだがネイティブ環境ではダメ、というコードがTypeScript上は許容されてしまうのです [16]。

- **`require`/`import`のミスマッチ**: WebpackなどはCommonJSの`require`もESMの`import`もどちらも吸収してバンドルできます。そのため、コード中でCJSモジュールを`import default`しても、Webpackのビルドでは問題なく動作する（Webpackがモジュールラッパーを用意してdefaultエクスポートを生成する）ことがあります。しかしTypeScriptの型チェックでは、先述のようにstrictなモードではエラーになるため、そのエラーを回避するために`esModuleInterop`を有効にする、という対処が必要になります [6]。開発者がこの違いを理解せずにいると、「開発中は型エラーになるが実際ビルドすると動く」あるいは「型は通っているのに実行するとエラーになる」といった混乱を招きます。

- **バンドラ固有のエイリアスや拡張**: バンドラでは`import`文に特殊なパス（`@/components/〜`のようなエイリアス）を使えたり、JSONやCSSを直接インポートできたりします。これらはTypeScript側でも`paths`エイリアス設定や`resolveJsonModule`等を使って対応しないと、型解決エラーになります。例えばWebpackで`import data from './config.json'`が許されている場合、TSでは`"resolveJsonModule": true`か、型宣言として`declare module "*.json"`を用意しないと**型エラー**になります。同様に、バンドラが拡張子省略やインデックスファイル解決を独自にやっていると、TS側で`moduleResolution: "bundler"`等にしないと齟齬が出ます。このように、**バンドラ環境とTSコンパイラ環境をできるだけ合わせる**ことが重要です。最近では、TS 5.2+で`moduleResolution: "bundler"`が**デフォルト推奨**になりつつあり（Node.js用のプロジェクトでもBundler推奨の動き [41]）、ランタイムごとの差異を埋めようとしています。ただし完全ではないため、引き続き注意が必要です。

- **実行時にのみ発覚する問題**: TypeScriptの型チェックはあくまで静的なものなので、モジュール解決やエクスポート形式の不一致による**実行時エラー**を完全には防げません。例えば、前述したように`esModuleInterop`を有効にしたがゆえに生じる`undefined`プロパティの問題 [6]は、TSコンパイラは検知できませんでした [5]。また、複数の型定義ソースが競合して間違った型が選ばれてしまう（例えば同じクラスが2種類の定義で不一致のため、実行時型と食い違う）ケースもあります。このため、ライブラリ作者は可能な限り**テストを充実**させ、型が通るだけでなく実際に両モジュール形式で動作するか確認する必要があります。特にデュアルモジュールの場合、CJSとしてrequireした場合とESMとしてimportした場合の両方でユニットテストを行うことが望ましいでしょう。

実際に発生した互換性問題の事例

理論的な話だけでなく、実務やOSSで報告されている具体的なトラブル例をいくつか紹介します。

- **事例1: 型定義二重取り込みによる競合** – ライブラリがあるバージョンから自前の型定義を同梱し始めたにも関わらず、依然として `@types/` 経由の型定義もプロジェクトに存在していたために衝突が起きたケースです。例えば、テンプレートエンジンのHandlebarsはv4.1.0で公式に `handlebars.d.ts` を含めましたが、それ以前から@types/handlebarsが存在していました。この移行期に両方の型定義がインストールされてしまい、コンパイラに「同じモジュールの型が二箇所から提供されている」と判断されてエラーが出る問題が起きました [3] 。対応としては、DefinitelyTyped側で型定義を**deprecated（非推奨化）**し、新バージョンでは不要になるように案内したり、ユーザ側で明示的に片方を排除する（npm/yarnの `resolutions` 機能で@types側をバージョン固定して入れないようにする等 [42] ）ことになりました。これは**ライブラリの型定義提供タイミング**と**利用者側の依存解決**の問題ですが、しばしば起こる現象です。類似の例として、Reactがv18で `@types/react` を内部統合した際にも、古い@typesを参照してしまうプロジェクトで不整合が起きるなどがありました。解決策は各ライブラリのリリースノート等を確認し、不要な@typesパッケージを削除することです。

- **事例2: NodeNextモードでのCJS互換問題** – 前述したGitHub Issueの具体例です [43] 。あるパッケージ（ `package` ）が `"type": "module"` でありながら、 `"exports"` でCommonJS向けに `.cjs` ファイルを提供していました。しかし、それをCommonJSターゲットのプロジェクトから使おうとすると、TypeScriptが**ESMとして認識して輸入禁止エラー(TS1479)を出す**問題が発生しました [13] 。さらにそのパッケージが出力した `index.d.ts` 内に拡張子なしの相対インポートがあったため、TS2835エラー（拡張子付けろ）が発生しました [12] 。実行上はNode.jsは問題なくCJSとして扱えるのに、型定義上だけ問題になるという非常に厄介なケースです。このIssueの結論としては、**TypeScriptの現行仕様上は回避が難しく、パッケージ側で `"type": "commonjs"` に変更するか、利用側で `moduleResolution: "node"` にするなどの妥協策**しかない状況でした [44] 。このように、TypeScriptの解釈とNode実行時解釈のズレが利用者を悩ませた例です。

- **事例3: `esModuleInterop` による実行時エラー** – こちらも先述しましたが、サードパーティーのログツールで、 `esModuleInterop` フラグを有効にしてインポートしたところ**実行時に `undefined` プロパティアクセスでエラーが起きた**ケースです [6] 。原因は、CommonJSモジュールのエクスポートが関数オブジェクトで、そのプロトタイプメソッドにアクセスしようとした際、TypeScriptが挿入した `__importStar` ヘルパーがプロトタイプを無視してしまったことでした [7] 。型的にはanyなどが絡んでいてエラー検出されず、実行して初めて不具合に気付いた例です [5] 。このIssueではコンパイラが検知できるよう改善を提案する声もありましたが、根本的には**言語間の相互運用の限界**に触れる部分です。解決策として、ライブラリ作者がESM版を提供する、または利用側で `import * as` を避け `const apm = require('...')` を使うなどが考えられます。

- **事例4: 型定義ファイル参照先ミスによるビルドエラー** – あるUIライブラリでは、誤って実装TSファイルを `typesVersions` で参照させてしまった結果、利用プロジェクトでコンパイルエラーが多発しました [31] （前述のSMUIの例）。これはstrictオプション（ `strictNullChecks` や `noUncheckedIndexedAccess` など）やTSバージョンの違いにより、ライブラリ内部コードが利用者側でチェックされたことで露呈した問題です [31] 。解決にはライブラリ側の迅速な修正が必要でした [32] が、利用者から見ると突然外部ライブラリ由来のエラーが出始める恐れがある怖いケースです。常に `skipLibCheck` で黙殺できるわけでもないため、信頼性の高いライブラリ選定やバージョン固定も検討すべきでしょう。

- **事例5: エディタ補完不全** – 互換性問題とは少し異なりますが、型定義の不備によりエディタでの補完（IntelliSense）が効かなくなる例もあります。例えば、ライブラリの型定義が複雑すぎたり、ジェネリクスの条件分岐で型が推論できない場合、エディタ上で型が `any` に見えて補完が出ないことがあり

ます。これはTypeScriptの型システムの限界かバグによることも多いですが、発生すると開発体験を損ねます。OSSではそうした報告がIssueとして上がり、メンテナが型定義を簡素化したり補助的な型エイリアスを導入したりして改善することがあります。実務上は、**あまりに難解な型定義は避け、開発者体験とのバランスを取る**ことも重要と言えます。

以上、様々な事例を見てきましたが、共通して言えるのは「**型定義は実装と切り離せない**」ということです。実装がどのようにモジュールをエクスポートし、利用者がどの環境でそれを使うかによって、最適な型定義の形や配布方法が決まります。型定義の配布者はその点を深く理解し、利用者からのフィードバックも踏まえて改善を続ける必要があります。

まとめ

TypeScriptの型定義ファイルを巡る互換性問題や挙動の落とし穴について、主要な論点を解説しました。ポイントを整理します。

- **モジュール形式と型定義**: CommonJS vs ESMのエクスポート差異を正しく型定義に反映し、`export =`と`export default`を混同しないこと。 1 2
- **コンパイラ設定の影響**: `moduleResolution: node16/nodenext`ではNodeの挙動を厳密に再現するため、従来との違い（exports対応や拡張子要求）が多々ある。 12 11
  （プロジェクトに合わせて適切なモードを選択し、必要なら`bundler`モードも検討する。）
- **型定義の配置と参照**: `package.json`の`types`フィールドを正しく設定し、依存する他パッケージの型も漏れなく提供すること。 21
  TypeScriptバージョン差に応じた調整が必要なら`typesVersions`を活用するが、設定ミスに注意すること。 32
- **`exports`と型の連携**: パッケージが`exports`を使う場合、型定義への参照も適切に書く（`"types"`条件の活用）ことで、利用者の型解決エクスペリエンスを損なわない。 35
  ESM/CJS両対応ライブラリでは、可能な限り統一的な型APIを提供し、必要に応じてドキュメントで`esModuleInterop`設定について触れる。
- **DefinitelyTypedパターンの参考**: コミュニティ型定義に学び、実装に忠実かつ多様な利用形態を許容する型定義を目指す。古いTSとの互換にも配慮し、テストを十分行う。 45 26
- **バンドラ環境とのズレ対策**: プロジェクト全体でTSコンパイラと実行環境（Node/ブラウザ）のモジュール解釈を揃えるのが理想。そうでない場合は、その差異（例えば拡張子やエイリアス）を埋める設定を行い、型と実行時の不一致を早期に検出する。 15 16

最後に、型定義は「動かないコード」ではありますが、ソフトウェアの一部として扱うべきものです。型定義の不備は開発者体験を損ない、時に実行時バグにも繋がります。逆に型定義が適切であれば、下流プロジェクトでの安心感と生産性向上に大きく寄与します。今回取り上げた落とし穴を踏まえて、より堅牢で互換性の高い型定義の提供・運用を心掛けていきましょう。

---

1 DefinitelyTyped/README.md at master - GitHub
https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/README.md

2 The repository for high quality TypeScript type definitions. - GitHub
https://github.com/DefinitelyTyped/DefinitelyTyped

3 42 Conflicting TypeScript definitions between @types/handlebars and latest handlebars · Issue #1499 · handlebars-lang/handlebars.js · GitHub
https://github.com/handlebars-lang/handlebars.js/issues/1499

4 37 45 TypeScript: Documentation - Modules .d.ts
https://www.typescriptlang.org/docs/handbook/declaration-files/templates/module-d-ts.html

5 6 7 `esModuleInterop: true` cause runtime error · Issue #41898 · microsoft/TypeScript · GitHub
https://github.com/microsoft/TypeScript/issues/41898

8 typescript - Write a declaration file for a default export module - Stack Overflow
https://stackoverflow.com/questions/39109027/write-a-declaration-file-for-a-default-export-module

9 14 TypeScript: TSConfig Option: moduleResolution
https://www.typescriptlang.org/tsconfig/moduleResolution.html

10 16 TypeScript: Documentation - Modules - Theory
https://www.typescriptlang.org/docs/handbook/modules/theory.html

11 35 node.js - package.json `exports` field not working with TypeScript - Stack Overflow
https://stackoverflow.com/questions/58990498/package-json-exports-field-not-working-with-typescript

12 13 36 43 44 moduleResolution node16 conflicts with type "module" packages that export commonjs · Issue #53045 · microsoft/TypeScript · GitHub
https://github.com/microsoft/TypeScript/issues/53045

15 40 vue.js - VueJS, Typescript and VSCode - "Relative import paths need explicit file extensions in EcmaScript imports..." - Stack Overflow
https://stackoverflow.com/questions/76746153/vuejs-typescript-and-vscode-relative-import-paths-need-explicit-file-extensi

17 18 21 24 25 26 27 28 34 TypeScript: Documentation - Publishing
https://www.typescriptlang.org/docs/handbook/declaration-files/publishing.html

19 How to fix error TS7016: Could not find a declaration file for module ...
https://pjausovec.medium.com/how-to-fix-error-ts7016-could-not-find-a-declaration-file-for-module-xyz-has-an-any-type-ecab588800a8

20 22 23 Typescript not finding types from module's index.d.ts question - $50 bounty : r/typescript
https://www.reddit.com/r/typescript/comments/mbjy7o/typescript_not_finding_types_from_modules/

29 30 31 32 33 `typesVersions` in `package.json` incorrectly point to source code instead of compiled `.d.ts` files · Issue #532 · hperrin/svelte-material-ui · GitHub
https://github.com/hperrin/svelte-material-ui/issues/532

38 Many modules are missing default exports? · Issue #27468 · DefinitelyTyped/DefinitelyTyped · GitHub
https://github.com/DefinitelyTyped/DefinitelyTyped/issues/27468

39 Node Typing Typescript Compile error, TS2304: Cannot find name ...
https://github.com/DefinitelyTyped/DefinitelyTyped/issues/10919

41 Option 'module' must be set to 'NodeNext' | Total TypeScript

https://www.totaltypescript.com/concepts/option-module-must-be-set-to-nodenext-when-option-moduleresolution-is-set-to-nodenext