卒業論文

JavaScript ライブラリの型情報整備と TypeScript 移行を支援する技術に関する調査

立命館大学 情報理工学部 情報理工学科
学籍番号：26002202149
氏名：竹内 太一
指導教員：丸山　勝久

Contents

Chapter 1

はじめに

1.1 背景

JavaScript（JS）はWebフロントエンドを中心に広く利用されてきたが，大規模化・長期運用に伴い保守性や変更容易性が課題となる．TypeScript（TS）は静的型付けを導入することで，IDE補完やリファクタリング支援，型検査によるバグ抑止など開発生産性の向上を狙う言語であり，JS資産を段階的に移行できる点から実務で普及している [1].

一方で実際のTS導入では「プロジェクト内に型情報が不足する領域」が移行コストを押し上げる．典型例として,（i）外部依存ライブラリに宣言ファイル（`.d.ts`）が無い,（ii）依存先は型を持つがモジュール解決（`exports`，CommonJS/ESM差異等）との整合が取れない,（iii）既存の型が下流の利用コードと整合しない，といった状況がある．この結果，下流TSプロジェクトで`tsc`が失敗し，移行や導入の進捗が妨げられる．

1.2 課題

JSライブラリの型情報不足は，単に「型が無い」だけでなく，複数の技術的要因により問題が複雑化する．例えば，モジュール境界のexport/import形（default export，名前付きexport，namespace import）や，`package.json`における`types`・`exports`設定の差異により，型定義が存在しても参照できない場合がある．また，型が参照できたとしても，利用文脈（引数の多形性，オーバーロード，動的プロパティ）と整合しないことで型エラーが生じる．このように，移行の成立は局所的な型注釈の正しさだけではなく，プロジェクト全体の整合性に依存する．

1.3　研究目的

本研究の目的は，JS ライブラリの型情報整備を支援し，下流の TypeScript プロジェクトにおける `tsc` 成功率（エラー無し）を高めることである．そのために，本論文では TypeScript の型システム，宣言ファイルとモジュール解決，移行時に現れる典型的な型エラー，および型推論・宣言生成の研究動向を整理する．

1.4　本論文の構成

第 2 章では技術背景として，TypeScript の型システムと `tsc` の挙動，宣言ファイル（`.d.ts`）とその仕様，モジュール解決と `tsconfig.json`，型エラーの性質と移行の難しさ，関連研究の系譜と評価観点を述べる．第 3 章以降で，これらの背景を踏まえた提案手法と評価設計を示す．

Chapter 2

技術背景

本章では，JavaScript（JS）ライブラリの型情報不足が TypeScript（TS）利用に与える影響を理解するために必要な予備知識を整理する．具体的には，(1) TypeScript の型システムと型検査，(2) TypeScript コンパイラ（`tsc`）とプロジェクト型チェック，(3) 宣言ファイル（`.d.ts`）の役割と仕様，(4) モジュール解決と `tsconfig.json`，(5) 移行時に頻出する型エラーの性質，(6) 型推論・宣言生成に関する研究の系譜と評価観点，を扱う．

2.1 TypeScript の型システムの基礎

2.1.1 基本型と型注釈

TypeScript は JavaScript に型注釈を追加し，静的型検査を行う言語である．基本型として `number`，`string`，`boolean`，`null`，`undefined`，`object` などがあり，配列型（`T[]`），タプル型（`[T,U]`），関数型（`(x:T)=>U`）なども記述できる．型注釈は関数引数・戻り値・変数宣言・プロパティ等に付与できるが，すべてが必須ではなく推論により省略可能な場合も多い．この「省略できる」性質は導入容易性の利点である一方，境界 API（外部モジュール，I/O，イベント）では型不足が残りやすい．

2.1.2 型推論と型の絞り込み（Narrowing）

TypeScript は代入や式の構造から型を推論する．また実行時の分岐に基づき型を絞り込む（narrowing）機構を持つ．`typeof` や `in` 演算子，`instanceof`，判別可能 Union（discriminated union）などにより，条件分岐内での型がより具体化される．ただし，ライブラリ境界で `any` が混入すると，絞り込みが成立せず安全性が低下する．

2.1.3 構造的型付けと互換性

TypeScript は構造的型付け（structural typing）であるため，型名よりも構造（プロパティ集合）が互換性を決める．例えば同じプロパティを持つオブジェクト同士は代入可能な場合がある．一方で，構造的型付けでは「必要最小限の構造だけを満たせば代入できる」ため，宣言側が過度に広い型を与えると，利用側の誤用が型検査で捕捉されにくくなる．このため宣言ファイル設計では，安全性（誤用検出）と柔軟性（互換性）のバランスが問題となる．

2.1.4 Union/Intersection とオプショナル性

Union 型（`A|B`）は複数型のいずれかを表す．Intersection 型（`A&B`）は両方を満たす型を表す．またプロパティのオプショナル（`p?:T`）や `undefined` を含む Union は，未定義状態を許容するモデル化に使われる．実務では API の引数が「省略可能」「複数形態を許す」といった仕様を持つことが多く，これらの表現選択は型の実用性に直結する．

2.1.5 Generics と制約

Generics（型パラメータ）は再利用性を高めるが，適切な制約（`<T extends ...>`）を与えないと型が広がり過ぎ，利用側の型安全性が下がる場合がある．例えば配列操作 API や Promise 等は Generics を前提として型が設計されているため，宣言側で Generics を省略すると利用側に波及的な型エラーが発生しやすい．

2.1.6 Mapped/Conditional/Utility Types

TypeScript には Mapped Types（`[K in keyof T]: ...`）や Conditional Types（`T extends U ? X : Y`）がある．これらは型レベルプログラミングを可能にし，高精度な型表現を実現する一方で，理解・保守が難しい．また型定義の複雑化はコンパイル時間やエラーメッセージの難読化に繋がる場合がある．宣言ファイル設計では「表現力」と「簡潔さ」を両立させる必要がある．

2.1.7 any と unknown

`any` は型検査をほぼ無効化し，任意の操作を許す．一方 `unknown` は安全側のトップ型であり，利用には型の絞り込みが必要となる．型情報が不足する局面では `any` に

逃げることで型エラーを減らせるが，型の利点（誤用検出）が失われるため，移行後の品質には影響する．このため,「型エラーを減らす」ことと「型の情報量を確保する」ことは別問題として考える必要がある．

2.2 TypeScript コンパイラ（tsc）と型チェック

2.2.1 tsc の処理フロー

tsc は概ね（1）入力収集,（2）構文解析,（3）名前解決（binding）,（4）型検査（type checking）,（5）出力（emit）を行う．noEmit を用いると出力を行わず，型検査のみを実行できるため，評価や CI で頻繁に利用される．

2.2.2 プロジェクト型チェックと依存関係

tsc は tsconfig.json を起点に対象ファイル群を決め，import された依存モジュールの型情報（主に.d.ts）を探索する．依存の探索が失敗すると，型検査以前に「モジュール未発見」や「宣言不足」が表面化し，エラーが連鎖して増える場合がある．

2.2.3 インクリメンタル・ビルドと Project References

TypeScript は incremental や Project References（composite）により大規模コードベースのビルドを高速化できる．ただしプロジェクト分割により型の境界が増え，参照関係や.d.ts の配置が重要になる．宣言ファイルの不整合は，プロジェクト境界を跨いで影響することがある．

2.2.4 Language Service と IDE

TypeScript Language Service は IDE 補完やエラー表示を行う．tsc の結果と同じエンジンを用いるが，解決環境（エディタ設定）差で見え方が異なる場合がある．実務上は，tsc で通ることに加え，IDE 上での型解決が安定することも重要となる．

2.3 宣言ファイル（.d.ts）の仕様と提供形態

2.3.1 .d.ts の役割

.d.ts は実装を含まず型のみを記述し，ライブラリの公開 API を型として外部へ提示する．JS 実装のライブラリであっても .d.ts があれば，利用側は型付きで扱える．逆に.d.ts が無いと，利用側は暗黙 any やエラー（設定次第）に直面する．

2.3.2 declare module とモジュール指定子

ライブラリを型として提供する代表的形式が declare module である．下流の import で指定されるモジュール指定子（"pkg"や"pkg/sub"）と，declare module の文字列が一致しない場合，型定義は参照されない．

Listing 2.1: declare module による宣言例

```
declare module "mylib" {
  export function foo(x: string): number;
  export type Options = { verbose?: boolean };
  export default function main(opts?: Options): Promise<void>;
}
```

2.3.3 アンビエント宣言とグローバル汚染

declare global やグローバル interface の追加は，環境全体へ影響するため副作用が大きい．意図せず他の型と衝突すると，下流プロジェクト全体で型が崩れる場合がある．従って，宣言ファイルの設計ではスコープ（モジュール内/グローバル）を明確化する必要がある．

2.3.4 Declaration Merging（宣言のマージ）

TypeScript は同名 interface や namespace などがマージされ得る．これは拡張を容易にする一方で，複数の型定義が同一の識別子に作用する場合，意図しない型の合成や衝突が起こることがある．DefinitelyTyped 等の@types とライブラリ同梱型が重なると，挙動が複雑化する．

2.3.5 Module Augmentation（モジュール拡張）

既存モジュールに対して追加宣言を与える形式が Module Augmentation である．プラグインや拡張機構のあるライブラリでは，この仕組みが型表現に使われることがある．ただし拡張対象の指定子や宣言の位置を誤ると，拡張が届かない・衝突するなどの問題が起きる．

2.3.6 オーバーロード・可変長引数・this 型

JS ライブラリの API は，引数の型や個数により挙動が変わることがある．TypeScript ではオーバーロード宣言や可変長引数（`...args:T[]`）で表現できるが，実装の全パターンを型で表すと宣言が膨大になる場合がある．またメソッドチェーン等では `this` 型が重要となる．

2.3.7 JSDoc と `checkJs`

TypeScript は JSDoc 注釈（`@param`，`@returns` 等）を読み取り，JS ファイルに対しても型検査を行える（`checkJs`）．これは段階的移行に有用だが，複雑な Generics や条件型の記述は難しい場合がある．

2.4 モジュール解決と `tsconfig.json`

2.4.1 CommonJS/ESM 相互運用

`module.exports` と `export default` の対応，名前付き export の扱い，namespace import（`import * as ns`）などが絡み合い，型定義側の export 形が合わないと利用側にエラーが出る．この問題は「型推論」以前に，モジュール境界の表現差として理解する必要がある．

2.4.2 モジュール解決戦略

TypeScript には複数の解決戦略があり，`moduleResolution` で指定される．戦略により，拡張子の扱い（`.ts/.tsx/.d.ts`），`package.json` の解釈，`exports` 条件分岐の評価などが変わる．

2.4.3 package.json の設定

types（typings）は型定義の入口を指す．exports は入口を制限し，サブパス解決にも影響する．typesVersions により TypeScript バージョン別に型定義を出し分ける場合もあるため，比較では TS のバージョン固定が重要となる．

2.4.4 パスエイリアス

baseUrl/paths はモノレポで多用されるが，ビルドツール側の alias と不一致になると tsc で解決失敗しやすい．

2.4.5 主要オプション整理表

Table 2.1: 主要な tsconfig オプションと一般的影響

オプション	影響（概略）
strict	strict 系をまとめて有効化し，移行時のエラーが増えやすいが型安全性は高い
noImplicitAny	暗黙 any を禁止し，型欠落がエラーになりやすい
strictNullChecks	null/undefined の扱いを厳密化し，API 境界で影響が大きい
skipLibCheck	依存.d.ts の検査をスキップし，エラー数が大きく変動し得る
module	出力モジュール形式（ESM/CJS 等）を指定
moduleResolution	参照解決戦略を指定（NodeNext 等で挙動差）
esModuleInterop	CJS/ESM 相互運用に関係し，default import の扱いが変わる
typeRoots/types	型定義探索範囲を制御し，@types や追加型の読込に影響
baseUrl/paths	エイリアス解決に影響し，解決失敗の要因になり得る
noEmit	出力なしで型検査のみを行う

2.5 移行時に頻出する tsc エラーと失敗モード

2.5.1 層の異なる失敗

モジュール解決失敗は依存・設定・エントリの問題であり，API の型不一致は型設計・利用文脈の問題である．層が混在すると，根本原因の切り分けが難しくなる．

2.5.2 代表的なエラーコード

Table 2.2: 代表的な TypeScript エラーコードと原因例

コード	概要	原因例
TS2307	モジュールが見つからない	依存不足，`paths` 不一致，`exports` 制約
TS7016	宣言が無く暗黙 `any`	JS パッケージで `.d.ts` 無し
TS2305	export されていない	default/名前付きの不一致，サブパス違い
TS2613/TS2614	import/export 形の差	CJS/ESM 差，`esModuleInterop` 設定差
TS2339	プロパティ不存在	形状型の不足，Union の不足，定義の粗さ
TS2345	引数型不一致	overload 不足，Generics 不足，型の過小/過大一般化
TS2322	代入不一致	期待型と実型の不整合，`null` 扱い
TS2554	引数個数不一致	可変長/省略可能引数の表現不足

2.5.3 エラーの連鎖（Cascade）

解決失敗や export 形不一致は，その後の型検査を破綻させ，派生エラーとして多数出力され得る．例えば本質的には「型が無い」問題であるにも関わらず，多数の `TS2339` が出る場合があるため，エラー総数だけの比較は危険である．

2.6 JavaScript 特有の表現が型付けを難しくする理由

2.6.1 動的プロパティアクセス

`obj[key]` のような動的アクセスは，key 集合が静的に決まりにくく，型表現が難しい．`Record<string,T>`で近似できるが，精密なキー集合を表すには追加情報が必要となる．

2.6.2 可変形 API

引数の型や個数により挙動が変わる API は，型としてはオーバーロードや Union で表現する必要がある．しかし実装が実行時分岐に依存するため，宣言側の表現選択が難しい．

2.6.3 モンキーパッチ・ミックスイン

実行時にオブジェクトへ後からプロパティを追加する記述は静的解析では追跡が難しく，型定義が遅れて破綻しやすい．

2.6.4 再 export とバレル構造

`export * from ...` の多用は入口 API と実装対応を複雑化し，型定義の入口設計を難しくする．

2.6.5 条件付き依存

環境条件により依存が変わる場合，型定義側でも条件分岐が必要となり，`exports` 条件や `typesVersions` と絡んで複雑になる．

2.7 型推論・宣言生成研究の系譜と評価観点

2.7.1 方式の分類

型推論や宣言生成は，入力情報と出力形態により整理できる．静的解析（AST・制約），動的解析（実行トレース），機械学習（コーパス学習），生成＋検証（コンパ

イラを評価器に用いる）などがある．

2.7.2 学習ベース型推論の流れ

DeepTyper[3] 等は局所文脈を用いた型推論を検討した．LambdaNet[4] は依存関係をグラフとして扱うことで精度向上を狙う方向性を示した．一方で，局所精度が高くても統合後に型チェックが通るとは限らない．

2.7.3 Search を用いた整合性の向上

TypeWriter[5] は探索（Search）と検証を組み合わせ，候補を選別する方向性を示した．この流れは「予測＝最終解」ではなく,「予測＋検証」で整合性を高める観点を与える．

2.7.4 TypeWeaver：移行成功に近い評価の重視

TypeWeaver[6] は，モデルの出力が「型注釈として統合されたときに TypeScript が型チェックできるか」を重視する枠組みを提示した．ここで重要なのは，移行成功が単に「型が当たるか」ではなく，依存型情報の有無やモジュール境界整合など周辺要因によって左右され得る点である．本章では評価観点としての意義に留め，具体的手法設計は次章以降で扱う．

2.7.5 OpenTau：LLM とコンパイラ評価

OpenTau[7] では，LLM による型生成に加えて，分解・探索・評価を組み合わせる方向性が議論されている．この流れは，生成のみでなく検証信号を用いる重要性を示す．

2.7.6 評価指標：局所と統合

局所評価（Accuracy/Top-k）は比較が容易だが，型同値性や全体整合を反映しにくい．統合評価（type check success）は実運用に近いが，環境固定や基盤整備が必要となる．

2.7.7 型同値性と正規化の難しさ

Union 順序，別名型，オプショナル表現などにより，同じ意味でも表現が異なる場合がある．完全同値判定は難しく，目的に応じて評価設計を選ぶ必要がある．

2.8 再現性の観点：環境差の影響

2.8.1 バージョン固定

Node.js や TypeScript のバージョン差は `moduleResolution` や標準ライブラリ型に影響するため，比較では固定と記録が重要である．

2.8.2 依存解決と lockfile

依存解決結果が変わると型チェックが変化するため，lockfile や `npm ci` 等による再現性確保が重要である．

2.8.3 `skipLibCheck` 等の設定差

`skipLibCheck` は観測されるエラーを大きく変える可能性がある．比較条件として設定差を明示する必要がある．

2.9 本章のまとめ

本章では，TypeScript の型システム，`tsc` の挙動，宣言ファイルの仕様，モジュール解決と `tsconfig`，移行時に頻出するエラーと失敗モード，JavaScript 特有の難しさ，型推論・宣言生成研究の系譜と評価観点，再現性の観点を整理した．これらは次章以降で述べる提案手法・評価設計の前提となる．

Bibliography

[1] TypeScript Documentation, Handbook.
`https://www.typescriptlang.org/docs/`

[2] DefinitelyTyped.
`https://github.com/DefinitelyTyped/DefinitelyTyped`

[3] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep Learning Type Inference," ESEC/FSE 2018.

[4] J. Wei, Y. Zhang, J. Wang, R. Wang, and L. Song, "LambdaNet: Probabilistic Type Inference using Graph Neural Networks," ICLR 2020.

[5] M. Pradel, et al., "TypeWriter: Neural Type Prediction with Search-Based Validation," ICSE 2020.

[6] M. Yee and A. Guha, "Do Machine Learning Models Produce TypeScript Types That Type Check?" ECOOP 2023.

[7] F. Cassano, et al., "Type Prediction With Program Decomposition and Fill-in-the-Type Training," arXiv 2023.

[8] K. R. Jesse, et al., "ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference," MSR 2022.