Chapter 3

提案手法（下流 TypeScript の typecheck 成立を目標とした `.d.ts` 注入）

3.1 何が難しいのか（問題点の再整理）

第1章・第2章で述べたとおり，TypeScript 移行の阻害要因は「型が無い」だけではない．現実には，
(i) 依存の型定義が存在しない／参照できない（型解決の失敗），
(ii) export/import 形が一致せず型が参照できても意味がズレる（モジュール境界の不整合），
(iii) 型が参照できても下流の利用実態と合致せず型エラーが残る（API 整合の失敗），
という複数層の失敗が混在する．

特に本研究の目的は「下流プロジェクトで `tsc` が通ること」であるため，
局所的に“それっぽい型”を出してもプロジェクト単位の成立に繋がらない場合がある．
さらに，生成された `.d.ts` が構文的に壊れると，`tsc` がパーサ段階で落ち，
本来観測したかったエラー（TS2339 等）が「出なくなる」ことで偽陽性が発生し得る．

したがって，本研究では「生成」そのものよりも，
1) 下流の失敗を層に分けて扱えること，
2) 介入（注入）が評価を壊さないこと（invalid 検知・フォールバック），
3) 回帰（特に TS2339/TS2554）を生む典型パターンを観測し，軽量な安全策で抑制できること，
を重視する．

3.2 アプローチ概要

本研究のアプローチは，下流プロジェクトの利用実態（usage evidence）から，
依存境界で必要な型情報のみを抽出し，依存側（外部モジュール）に対して
アンビエント宣言（`declare module '...' { ... }`）として `.d.ts` を生成・注入することで，
下流の `tsc` 失敗を減らすものである．

処理の流れは以下の通りである．

Algorithm 1: Phase3 runner の全体フロー（概略）
1. 対象 repo を clone / install
2. baseline で `tsc --noEmit` を実行し，診断（TSxxxx）を収集
3. Phase3 core エラーが出たファイル群から import を抽出
4. 注入用 `.d.ts` を生成（DTS_STUB または DTS_MODEL）
5. 注入用 `typeRoots/types` を設定した派生 tsconfig を作成し，再度 `tsc` を実行
6. baseline と injected の差分を集計し，invalid/timeout を除外した統計を出力

このアプローチの特徴は，(a) 下流を編集せず依存境界に介入する点，
(b) `tsc` を“評価器（oracle）”として使う点，
(c) 失敗を層（フェーズ）に分けて扱う点である．

3.2.1 入出力と成果物（再現性のための記録）

本手法は「実プロジェクトに対する介入」なので，再現性確保のために入出力を機械的に保存する．
Phase3 runner は 1 repo を 1 行の JSON として `results.jsonl` に追記し，
同じ情報を tsv に正規化した `summary.tsv` を出力する．

さらに DTS_MODEL では，アダプタ（Python）が返した生成物をキャッシュし，
後から「どの `.d.ts` が注入されたか」を追跡できるようにする．

・cache（例）: `evaluation/real/cache/typebert/<cache_key>.json`
  - `cache_key` は（入力 JSON + 実験設定 + adapter_version）から計算し，結果にも埋め込む
  - `meta` には，fallback 理由，missing 補完の有無，モデル設定などを含める

この追跡性は，卒論で「失敗要因（TS2339/TS2554 回帰）がどの生成に起因するか」を根拠付きで議論するた

3.3 フェーズ設計（層の異なる失敗を分解する）

本研究では，`tsc` のエラーコード分布を「意味の近い失敗層」に対応づけ，
評価と対策を段階化する．

Phase 1（型解決 / Type Resolution）
- 代表コード: TS2307, TS7016
- 失敗の意味: モジュール指定子に対して型定義が見つからない（型が参照できない）
- 介入: `declare module '...'` の最小スタブ注入

Phase 2（境界整合 / Module Boundary Alignment）
- 代表コード: TS2305, TS2613, TS2614
- 失敗の意味: export/import 形の不整合（default/named/namespace など）
- 介入: import 形の決定的変換や，境界側の export 形の整備

Phase 3（API 整合 / API Alignment）
- Phase3 core: TS2339, TS2345, TS2322, TS2554, TS2769, TS2353, TS2741, TS7053
- 失敗の意味: API 仕様と下流利用の不一致（プロパティ欠落，引数個数，代入不一致など）
- 介入: `.d.ts` を注入して API レベルの不整合を減らす（本章の主対象）

本研究の現状実装では，Phase3 の runner と集計が成立しており，
Phase3 の回帰要因（TS2339/TS2554）を抑える安全策の検討を進めている．

3.3.1 Phase3 core を採用する理由（卒論の指標設計）

Phase3 では多数のエラーが連鎖（cascade）し得るため，単純に「エラー総数」を指標にすると解釈が難しい
本研究では，API 整合の中心にあるエラー群（TS2339/TS2554 等）を Phase3 core として固定し，
注入によって core が減ったか（Reduced），0 になったか（Eliminated），コード別に増減したか，
を主要な分析単位とする．

また，注入 `.d.ts` が壊れると `tsc` がパース段階で落ち，
Phase3 core が「出なくなる」偽陽性が生まれるため，
Phase3 では invalid を除外した valid injection の母数を必ず併記する．

3.4 証拠抽出（usage evidence）

Phase3 は「API の整合」を扱うため，どのモジュールに対して何を宣言すべきかを，
下流のコードから抽出する必要がある．本研究では，Phase3 core を含む診断が出たファイルを起点とし，
そのファイル内の import を解析する（intra-file / best-effort）．

3.4.1 import 形の抽出
対象は ES import 文であり，以下を抽出する．
- default import: `import X from 'm'`
- namespace import: `import * as ns from 'm'`
- named import: `import { a, b as c } from 'm'`
- type-only import: `import type { T } from 'm'`

この結果，モジュール指定子 m ごとに
defaultImport / named（値） / typeNamed（型） を収集する．

3.4.2 メンバアクセスの抽出（TS2339 回帰抑制）
回帰の典型として，「import した識別子に対する `.foo` アクセス」が
型宣言側で欠落して TS2339 を爆増させるケースがある．
本研究では以下 2 種のアクセスを抽出する．

(A) namespace / default import の member access
例: `ns.foo`，`x.foo`
→ これらは module export の欠落として扱い，必要な export 名（foo）を追加する．

(B) named import の member access（静的メンバ参照）
例: `import { Foo } from 'm'` の後に `Foo.bar`
→ module export を増やしても解決しないため，

`export namespace Foo { export const bar: any }` の形で“宣言のマージ”を利用して吸収する

3.4.2.1 named import の member access が必要になる典型

実プロジェクトでは，クラスや関数の static メンバにアクセスする書き方や，
オブジェクトとして import した値に対するフィールド参照が頻出する．
これらは「モジュール export の欠落」ではなく，
「export された値の内部構造の欠落」として現れるため，
従来の `export const bar: any` を追加するだけでは TS2339 が抑えられない．

本研究では，Declaration Merging の性質（namespace を後付けでマージ可能）を利用して，
`export namespace Foo { ... }` を生成し，`Foo.bar` の形を満たす．
これは厳密な意味で Foo の型を推定したわけではないが，
「下流 `tsc` を成立させるための安全な緩和」として位置づける．

3.4.3 「外部モジュール」の判定と安全策（deps フィルタ）

大規模 repo では，`core/*` のような内部 alias を外部扱いして注入すると，
既存の型関係を壊して TS2339 が爆増することがある．
そこで本研究では，外部判定を 2 通り用意する．

- heuristic: 文字列パターンで外部っぽい指定子を採用（従来）
- deps: repo root の package.json を読み，依存に含まれるパッケージ名のみを外部とみなす

deps フィルタは「依存境界に介入する」という研究目的と整合し，
内部 alias 混入を避ける軽量安全策として有効である．

3.4.3.1 deps フィルタの具体

deps フィルタでは，モジュール指定子 `spec` から package 名を抽出し，
`package.json` の dependencies / devDependencies / peerDependencies / optionalDependencies の集合に
含まれる場合のみ「外部」と判定する．

Algorithm 2: deps フィルタの概略
1. `spec` が相対パス・node:・内部 alias（@/ など）なら除外
2. `spec` の先頭要素（@scope/pkg も含む）を package 名として取り出す
3. repo root `package.json` の依存集合 D に package 名が含まれる場合のみ採用

この判定により，巨大 repo に含まれる `core/*` のような内部パスや，
解決規則依存の alias を注入対象から外し，
TS2339 の“爆増”を抑えた状態で評価が可能になる．

3.4.4 収集スコープの調整（diag / repo）

メンバアクセスは，Phase3 core が出たファイル群（diagFiles）に限定して収集すると取りこぼしがある．
一方で repo 全体をスキャンすると計算量が増える．
本研究では，`--member-access-scope` により，
diag のみ／repo 全体（最大ファイル数・最大バイト数で上限）を切り替えられる設計とした．

3.5 `.d.ts` 生成（DTS_STUB と DTS_MODEL）

3.5.1 DTS_STUB（下限ベースライン）

DTS_STUB は「境界を any で埋めてどこまで通るか」を測るためのベースラインである．
抽出した default/named/typeNamed に対して，以下の形のアンビエント宣言を生成する．

```
declare module 'm' {
  const __default: any; export default __default;   // defaultImport がある場合
  export const __any: any;                          // namespace import のアンカー
  export const f: any;                              // named（値）
  export type T = any;                              // typeNamed（型）
  export namespace Foo { export const bar: any; }   // named import の member acc
```

ess（必要な場合）
}

3.5.2 DTS_MODEL（生成モデル＋サニタイズ）
DTS_MODEL は，抽出した modules 情報を JSON としてモデルアダプタに渡し，
モデルが生成した `.d.ts` を注入する方式である．
現状の実装では，アダプタは Python スクリプトとして実装し，
stdin JSON → stdout JSON の I/O 契約で runner と疎結合にしている．

重要なのは「壊れた `.d.ts` が評価を壊す」点である．
そのためアダプタ側では以下を行う．
- `declare module` ブロックのみを抽出（brace balancing）
- 既知の壊れパターンをサニタイズ（例: `export type Foo: any;` の修正）
- requested な export の欠落を `any` で補完（回帰抑制）
- それでも危険な構文が残る場合は DTS_STUB にフォールバック

加えて，生成物の追跡性のために cache_key と meta を保存し，
ケーススタディで「どの `.d.ts` が注入されたか」を後から辿れるようにする．

3.5.2.1 JSON I/O 契約（runner ↔ adapter）

アダプタの入出力は以下を想定する．（卒論では実装依存を最小化するため，形式のみ示す）

```
Input（stdin JSON）:
{
  "repo": {"url": "...", "slug": "..."},
  "modules": {
    "<specifier>": {
      "defaultImport": true|false,
      "named": ["Foo", "bar"],
      "typeNamed": ["T"],
      "members": { "Foo": ["baz"] }
    }
  }
}

Output（stdout JSON）:
{
  "ok": true,
  "backend": "hf_causal_lm"|"stub",
  "dts": "declare module '...' { ... }",
  "meta": { "adapter_version": "...", "fallback_reason": "...", ... },
  "cache_key": "..."
}
```

この疎結合により，将来モデルを差し替える場合でも，
runner 側は JSON 形式の維持だけで比較実験が可能になる．

3.5.2.2 欠落 export 補完（回帰抑制）

モデル出力は「必要な名前を出し忘れる」ことがあり，その場合 TS2339 が増える．
そこで本研究では requested exports に対して，
該当する `declare module` ブロック内に export が見つからない場合は `any` として補完する．

補完は「モデルを賢くする」のではなく，
「境界の欠落を保守的に埋めて下流の成立を優先する」戦略であり，
卒論では RQ2（any 濫用の懸念）と併せてトレードオフを議論する．

3.5.2.3 invalid の検知とフォールバック（評価の成立条件）

生成 `.d.ts` の構文エラーは，`tsc` のパース段階で検出され，
TS1127/TS1434 等として現れる．この場合，Phase3 core が観測できないため，

本研究では invalid を除外した valid injection のみを主要指標に用いる.

さらに，アダプタは“危険”と判定した場合に DTS_STUB にフォールバックし，
パイプラインが停止しないようにする（評価を回すための実務的要件）.

3.5.2.4 モデル設定と決定性（卒論での条件固定）

LLM による生成は確率的であり，条件が揺れると再現性が低下する.
卒論では以下を基本方針とする.

- temperature を 0.0 とし決定性を高める（do_sample を無効化）
- max_new_tokens を固定し，生成長の差による比較ブレを抑える
- device / dtype を記録し（例: mps + float16），環境差の影響を最小化する
- adapter_version を cache_key に含め，実装更新後に古いキャッシュが混ざらないようにする

このような条件固定は，TypeWeaver/OpenTau が指摘する「統合評価の再現性確保」に対応する実装上の工

3.5.3 追加の安全策：force-any（denylist）

実プロジェクトでは，一部の有名依存（例: lodash, zod など）や，巨大で可変な API を持つ依存は，
厳密な型を生成すると回帰（TS2339/TS2554）が増える場合がある.
本研究では，安定した比較を優先する場合に，特定モジュールを強制的に `any` スタブへ落とす
「force-any（denylist）」をオプションとして用意する.

この安全策は，
「モデルが当たるか」という議論ではなく，
「下流 `tsc` を成立させるために，どこを保守的に握るべきか」
という失敗要因分析（RQ3）と結びつけて説明できる.

3.5.4 生成物追跡（cache_key / meta）とケーススタディ

卒論で失敗要因を議論するには，「結果（TS2339爆増）→原因（注入`.d.ts`のどの宣言か）」を追える必要が
そこで本研究では，生成 `.d.ts` の全文を results に埋めずに，
cache_key と meta を残すことで，サイズを抑えつつ追跡性を確保する.

典型的なケーススタディ手順は以下の通りである.
1) 集計で「悪化上位」や「invalid」等の対象 repo を選ぶ
2) cache_key から注入 `.d.ts` を取得し，宣言（module specifier / export 形 / members）を
3) 原因カテゴリ（外部判定ミス・export欠落・静的メンバ参照・invalid等）に分類する
4) 軽量対策（deps フィルタ，欠落補完，namespace merge，denylist）を適用し，再実行で差分を見る

3.6 注入手法（typeRoots/types による確実な読み込み）

注入は下流プロジェクトの tsconfig を直接編集せず，
派生 tsconfig（例: `tsconfig.__phase3__.json`）を生成して `tsc -p` で参照する.

- `.evaluation-types/phase3/@types/__phase3_injected__/index.d.ts` に注入ファイルを配置
- `compilerOptions.typeRoots` に注入先と `node_modules/@types` を含める
- 既存の `compilerOptions.types` がある場合は `__phase3_injected__` を追加

この方式により，プロジェクト側の設定差を最小限にしつつ，
注入した `.d.ts` が確実に読み込まれるようにする.

3.6.1 `types` を上書きしているプロジェクトへの対応

現実のプロジェクトでは `compilerOptions.types` が設定されている場合があり，
その場合 `typeRoots` を追加しても読み込まれないことがある.
本研究では派生 tsconfig で `types` を拡張し，注入パッケージ名を追加することで
取りこぼしを避ける.

3.6.2 Gate A：tsconfig が repo root に無い場合

実プロジェクト評価では，monorepo 等の理由で `tsconfig.json` が repo root に存在しない場合があ
この場合，どの tsconfig を対象に `tsc` を走らせるべきかが自明ではない．
卒論の評価では「比較の成立」を優先し，repo root に tsconfig があるもののみを対象（Gate A）とし
それ以外は skip として母数から除外する．

この制約は外的妥当性の脅威となるため，評価章で skip 理由の内訳を必ず併記する．

3.7 評価の信頼性（invalid/timeout の扱い）

本研究では，生成 `.d.ts` が構文的に壊れた場合に偽陽性が起きる問題を避けるため，
注入後の `tsc` 出力から TS1005/TS1127/TS1434 等を検知し，
`phase3InjectedDtsInvalid=true` として記録・除外集計する．

また，モデル推論は時間がかかるため timeout が発生し得る．
その場合も invalid と同様に「評価不能」として扱い，valid injection の母数を明示した上で
Reduced/Eliminated 等の指標を算出する．

3.7.1 timeout 低減の方向性（卒論での運用）

卒論の評価では，母数を無理に増やすより，
「valid injection を増やし，解釈可能な比較を行う」ことを優先する．
本研究では以下の運用を想定する．

- `--max-stub-modules` により巨大入力を除外（または後回し）
- `--model-timeout-ms` の調整
- deps フィルタにより注入対象モジュール数を減らし，推論負荷を下げる

これにより，少数（例: 30件）でも，母数・invalid/timeout を併記した上で
説得力のある比較が可能になる．

3.7.2 集計の仕方（valid のみで議論する）

卒論の表・図では，次をセットで示す．
- 総数（repos_total）
- valid injection 数（repos_valid_injection）
- invalid / timeout / skip の内訳
- valid のみで算出した Reduced / Eliminated
- Phase3 core 合計差分と，TS2339/TS2554の差分

これにより，「改善が invalid による偽陽性ではない」こと，
「改善が一部に偏っていないか（母数の明示）」を担保できる．

3.7.3 条件比較の例（heuristic vs depsfilter）

本研究では，外部判定の違いが回帰（特にTS2339）に影響するという仮説を検証するため，
同一の対象集合（ranked30）に対して 2 条件の比較を行った．

- 条件A: heuristic（外部っぽい指定子をヒューリスティックに採用）
- 条件B: depsfilter（package.json 依存に含まれるパッケージのみを外部として採用）

この比較の集計結果（表・repo別差分）は，
`evaluation/real/report-ranked30-compare.md` および
`evaluation/real/report-ranked30-compare.tsv` に保存している．

卒論の評価章では，(i) valid injection 母数，(ii) Phase3 core合計差分，
(iii) TS2339/TS2554 の差分，(iv) ケーススタディ（改善例・悪化例），
をこの比較を起点として議論する．

3.8 まとめ（提案手法の位置づけ）

本章では，下流 `tsc` の成立を目標とした `.d.ts` 注入の提案手法を示した．
重要点は，(i) 失敗をフェーズに分解して扱うこと，
(ii) usage evidence から必要最小限の宣言を構成すること，
(iii) invalid/回帰を前提に，評価を壊さない安全策（deps フィルタ等）と追跡性を備えることである．

次章（評価）では，実プロジェクト集合に対して本手法を適用し，
Phase3 core の削減・回帰の傾向・valid injection 母数を中心に議論する．

3.8.1 本手法の限界（評価章へのつなぎ）

本章の提案手法は，下流`tsc`の成立を目的とするため，必ずしも厳密で最小な型を生成するわけではない．
特に，(i) dynamic property access（obj[key]）のような JS 特有の表現，
(ii) 実行時に変化する API（条件付き export，モンキーパッチ），
(iii) monorepo 構成や複数 tsconfig の存在，
は本手法の best-effort 解析では完全に扱えない．

また，deps フィルタは内部 alias 混入を抑える一方で，
package.json に現れない依存（ワークスペース内パッケージや生成コードの import）を除外し得る．
卒論では，このトレードオフを「valid injection の母数・skip 理由の内訳」と合わせて議論し，
本手法がどの条件で有効／無効になるかを明確化する．