ChatGPT

TypeScript型定義ファイル自動生成・補完技術の研究調査

はじめに

JavaScriptの人気ライブラリをTypeScriptで利用する際、対応する型定義ファイル（`.d.ts`）の有無が問題になります [1]。型定義ファイルはモジュールの形状（公開APIの型情報）を記述し、開発者が型安全に利用できるようにするものです。しかし多くの型定義は手作業でDefinitelyTypedリポジトリ等にメンテナンスされており、**作成・更新は時間がかかりエラーも起こりやすい**と指摘されています [2]。型定義と実装が食い違ってもTypeScriptコンパイラは検出できず（型定義を「鵜呑みにする」ため）、IDEで誤った補完候補が出たり、実行時に想定外の挙動・クラッシュを招く恐れがあります [2]。このように手動管理の型定義には限界があり、**JavaScriptプロジェクトから型定義ファイルを自動生成・補完する技術**が近年注目されています。

本調査では、TypeScriptの型定義ファイル（`.d.ts`）生成・補完に関する研究の系譜と現状を整理します。まずアプローチ手法を**静的解析・動的解析・自然言語情報の活用・大規模言語モデル(LLM)の活用**に分類し、それぞれ代表的な研究事例（例：**dts-generate**, **TypeWeaver**, **OpenTau**, **DeepTyper**, **dtsmake**, **QuickType**など）の手法を概説します。次に、自動生成された型定義の**品質上の課題**（型の不健全さ、過剰あるいは過小な近似、IDE補完や型チェックへの影響）について議論します。さらに、生成した型定義をプロジェクトに導入した際の**下流への影響**（型整合性、型エラーの発生状況、移行成功率など）や、評価指標の比較（単純な予測精度だけでなく、**コンパイル時の型チェック通過率**やプログラム実行可能性など）にも触れます。最後に、既存の関連ツール（オープンソースを含む）の比較と限界についてまとめ、今後の展望を述べます。

型定義ファイル生成のアプローチ分類

静的解析アプローチ

ソースコードの静的解析に基づき型情報を推論する手法です。プログラムのAST（抽象構文木）や構文解析結果を用い、変数や関数の使用状況から型を推定します。例えば**dtsmake**はJavaScriptコードから`.d.ts`を生成するOSSツールで、JavaScript補完エンジンであるTern.jsを内部で利用することで型推論を行います [3]。dtsmakeは入力JSファイルを解析し、できるだけ具体的な型を出力しようと試みます（安易に`any`型ばかり出さない） [4]。また、複数のサンプル使用コードをオプションで指定すると推論精度が向上する設計になっています [4] [5]。このように静的手法では、**コード構造や型推論エンジン**を駆使して型定義を生成します。

他にも、TypeScriptコンパイラ自体が持つ型推論機能を活用する方法もあります。TypeScriptはJSDocコメントや簡易的な型推論である程度の型情報を得られますが、複雑な型や外部モジュールについては多くの場合`any`に落ち着いてしまいます [6] [7]。例えばVisual Studio CodeのIntelliSenseは、未型注釈のJS関数の引数型を全て`any`と表示するため、適切な引数を推測する助けになりません [8] [6]。この限界を補うべく、静的解析に機械学習を組み合わせた研究も登場しています（後述）。

静的解析アプローチの利点は、ソースコードさえあれば実行せず型候補を得られる点です。動的手法と異なり、実行に伴う副作用や環境構築を避けられます。またコンパイラの型検出ロジックをそのまま利用できる場合もあります。しかし**課題**として、JavaScriptの動的な言語仕様ゆえに**静的解析のみで正確な型を推論するのは難しい**ことが挙げられます [9] [10]。Duck Typingや`eval`の存在、動的プロパティ追加などにより、サウンド（完全に網羅的かつ正確）な解析は極めて困難です [11] [12]。そのため静的手法では、型を安全側に過

剰一般化（over-approximation）してしまい`any`だらけになる、あるいは特定のケースに絞り過ぎて不足（under-approximation）し、実際の利用シナリオをカバーできない、といった問題が発生しがちです。実用上は、静的解析結果を手動で補完・修正するコストが残る点も限界と言えます。

動的解析アプローチ

動的解析では、対象ライブラリを実際に**実行**し、ランタイムの振る舞いから型情報を収集します。代表例である**dts-generate** [13]は、NPMパッケージのドキュメント中に掲載された使用例コードを自動抽出し、そのコードを実行して得た実行トレースから型情報を取得する手法を提案しました [14] [15]。具体的には、対象ライブラリをインストールし、READMEなどに含まれるサンプルコードを解析・抽出して実行環境下で動かします。実行時にはインストゥルメンテーション（Jalangiフレームワーク [16]など）によって関数呼び出しやプロパティアクセス時の実際の型（値の型）を記録します。その収集データに基づき、ライブラリが持つ関数の引数や戻り値、オブジェクトのプロパティ型などを**実際に観測された具体的な型**として`.d.ts`に書き出します [14]。このアプローチでは静的解析に頼らないため、最新のJS構文変更に追従する実装保守の負担が小さいメリットがあります [11]。

dts-generateの成果物を例に取ると、NPMパッケージ「abs」（パス文字列を絶対パスに変換する関数）のドキュメント例を実行した結果、`export = abs; declare function abs(input: string): string;`という型定義が生成されました [13] [17]。これは当時DefinitelyTypedに登録されていた手書き型定義と概ね一致するものです。ただし、後になって人間のメンテナが引数`input`をオプショナルに修正しており、生成結果にはその情報が欠けていました [17]。このケースではドキュメントの例に`abs()`（引数なし呼び出し）が無かったため、optionalである事実を検知できなかったのです [18]。開発者がそのようなケースを網羅する追加例を与えれば、生成結果も完全に正しくできたことが示唆されています [19]。

以上のように、動的解析アプローチは**ライブラリの実行例次第で精度が決まる**という特徴があります。十分多様な入力に対する例があれば型定義精度は高まりますが、ドキュメントが不十分だと見逃しが生じます。Cristianiらの評価では、DefinitelyTypedに型定義が存在する6029個のパッケージのうち、ドキュメントにコード例が含まれるものは249個のみだったと報告されています [20]。さらに実行可能なコード抽出の段階で多くのパッケージが脱落し、最終的にdts-generateで型定義を生成できたのは249件中の一部に留まります [21] [22]。それでも**動的解析で得た型定義の有用性**は一定示されており、生成された型定義の多くは手書きのものと大きく乖離しない品質でした [23] [24]。動的手法の利点は、**実行時の実データに即した具体的な型**が得られるため、静的解析では不明瞭な型も明らかにできる点です。例えば関数の戻り値オブジェクトの構造などは、実行して観測すれば精密な型（具体的なプロパティ名や型）として書けます。一方で弱点は、**網羅性の欠如（未観測経路の型は反映されない）**や、実行環境依存（セットアップや副作用の問題）があること、加えて**実行用の入力スクリプトを用意する必要**があることです [19]。実運用ではテストスイートやサンプルコードから自動で例を収集する工夫が鍵となります。

自然言語記述・コメントの活用アプローチ

ソースコード中の**識別子名やコメント（ドキュメンテーション）**といった自然言語情報から型を推定するアプローチも研究されています。JavaScriptではJSDoc形式で型をコメント記述する習慣もあり、これをパースして型定義を生成するツール（例えば**tsd-jsdoc**など）が実用化されています。TypeScriptコンパイラも`@param`や`@returns`タグの型記述を読み取って補完に用いることができます [25] [26]。しかし全てのプロジェクトが完備なJSDocを持つわけではなく、自然言語から暗黙の型情報を引き出す必要があります。

この方向で代表的な研究が**NL2Type**です。NL2Typeは関数名・引数名やコメント文から、その関数のパラメータ型および戻り値型を推定する機械学習モデルを提案しました [27]。開発者が記述する関数の説明や引数説明にはしばしば「〇〇を表すnumber型の引数length」といった情報が含まれます。NL2Typeはそれらテキストとコード構造をLSTMベースのモデルに入力し、最も可能性の高い型を分類問題として予測します [28]。結果として、トップ1予測の正解率はそれまでの静的解析ベース手法（JSNice等）より向上し、トップ

5では約90%のリコール（候補内に正解を含む率）を達成しました [29] 。さらに、既存のJSDoc型注釈と矛盾するケースを検出し、誤った型注釈の是正に繋がる例も示しています [30] 。このように**自然言語中のヒントを活用**することで、コード単体からは得られない文脈的な型推論が可能となります。

ただしコメントや名前に依存する手法には注意点もあります。コメントが古かったり不正確な場合、誤った型推論につながるリスクがあります。また名前やテキストからの推論は確率的であり、「似た名前の変数は似た型になりやすい」といったデータバイアスに影響されます。そのため、このアプローチ単独ではなく他の情報源と組み合わせ、**総合的に型を推定**する流れが主流です。実際、多くの機械学習型推論モデルはコード構文情報と識別子名情報の両方を特徴量として取り込み、モデルを学習させています [31] [32] 。

機械学習・大規模言語モデル(LLM)アプローチ

近年注目を集めるのが、**機械学習とくにディープラーニング**を用いて型定義生成・補完を行う手法です。2018年頃から、既存の大量の型付きコード（TypeScriptやFlowで書かれたコード）を学習データとして、未注釈コードに型を予測する研究が盛んになりました [33] [34] 。特にTransformerを用いた大規模言語モデル(LLM)の台頭により、コード片から欠損した型を生成する**コード補完的なアプローチ**が現れています。

初期の例として**DeepTyper**（Hellendoornら， 2018）が挙げられます。DeepTyperはJavaScript/TypeScriptコード中の識別子に対し、リカレントニューラルネットワーク(RNN)で適切な型ラベルを推定するモデルです [35] 。学習には型付きコードを用い、シーケンスモデルにより「文脈上自然な型」を当てることを目指しました。DeepTyperはPrecision 80%超・Recall 50%程度で型予測が可能と報告されており、型をある程度絞り込む効果が示されています [36] 。また、RNNの長距離依存性保持が弱いため一貫性に課題があることに対し、型ベクトルの工夫やファイル全体での整合性向上策を提案しています [37] [38] 。さらに、確率モデルの予測結果を従来の静的型推論エンジン（例えばコンパイラのany推論やJSNiceの型推定）と**統合（シンエルジー）**させることで、双方の長所を活かす試みも報告されています [39] 。このようにDeepTyperは「機械学習で型予測精度を上げられる」可能性を示しつつ、**予測の不整合やカバー範囲**など課題も浮き彫りにしました。

続いて2020年前後には**グラフニューラルネット(GNN)**を用いた型推論も登場しました。例えばLambdaNet（Weiら， 2020）はプログラムを型依存グラフとして表現し、GNNで各ノードの型を推定する手法です [40] [41] 。シーケンスモデルに比べ、プログラム構造上の制約を扱いやすい利点があります。LambdaNetはDeepTyperより高精度を報告しましたが、それでも**単独の精度評価**（例えば「予測型が正解と一致した割合」）に留まり、生成された型でコード全体がコンパイル通るかまでは検証されていませんでした。この点に着目したのが**TypeWeaver**（Yeeら, 2023）です。

TypeWeaverは「機械学習モデルで予測した型をコードに織り込み、**実際にTypeScriptコンパイルが通るコードにできるか**」を評価するためのフレームワークです [42] [43] 。TypeWeaver自体は型予測モデルではなく、任意のモデル（著者らはDeepTyper、LambdaNet、InCoderという3種を使用）の出力を受け取り、以下の処理を自動化します [44] [45] :

- **依存パッケージの型情報導入**: プロジェクトの依存ライブラリにも型定義が必要なため、再帰的に依存先を型付けするか、既存の `.d.ts` を導入する [46] 。
- **モジュール形式の変換**: CommonJSモジュールはそのままでは型チェックに不利なため、必要に応じてES6モジュール構文に書き換える [47] （CommonJSのままだとモジュール間の型共有が困難なため [47] ）。
- **型の織り込み（Type Weaving）**: 予測モデルが出力した型を、元のJavaScriptコードの適切な位置に挿入し、TypeScriptの構文に組み込む [48] 。関数宣言や変数宣言に対し推定された型注釈を付与していく。
- **非型トークンの除去**: 大規模言語モデル(LLM)由来の予測では稀に無効な文字列（型として文法誤りのもの）が含まれるため、それらを検出して除去・修正する [45] 。

これらの工程を経て自動生成された型付きコードをTypeScriptコンパイラでチェックし、**エラーなく通過するか（型移行成功か）**を評価指標としました [49]。従来は個々の予測精度（例えば「正解の型ラベルを当てた率」）が重視されていましたが、TypeWeaverの著者らは「精度が高くてもコード全体が動かなければ無意味」であり、**型チェックを通るコード生成こそ指標にすべき**と論じています [50] [51]。

TypeWeaverの評価では、選択した513個の未型付けnpmパッケージに対し、自動型付け後に**パッケージ単位で21%**のみがエラーなくコンパイル成功しました（最良モデルInCoder使用時） [52]。ファイル単位では**69%**が型チェック通過しており、一部ファイルだけエラーが残るケースが多かったことを示しています [52]。また、型注釈の**「自明な型」**（＝`any`など誰でも付けられる曖昧な型）の割合も分析され、モデルによっては多数の`any`を出力する傾向も確認されました [53] [54]。これは予測モデルが不確実な箇所で`any`を乱用しがちなこと、またはTypeWeaver側でコンパイルエラー回避のため`any`にフォールバックした可能性を示唆します。加えて、**典型的な型エラーのパターン**も分類されました [54]（後述）。CommonJS→ESM変換の有無では、変換することで型チェック成功率が向上する場合が多く、モジュール間依存の型解決に効果があると確認されています [55]。

LLMを活用した最新の取り組みとしては**OpenTau**（Cassanoら， 2023）があります。OpenTauは大規模言語モデルを用いた型補完で直面する課題（FIM注釈の困難さ、長大な文脈、生成型がコンパイルエラーを起こす問題など）に対処するため、**プログラム分解と探索**を組み合わせた手法です [56] [57]。具体的には、プログラムを再帰的に小さな部分に分解し（関数単位など）、各部分に対してLLMによる型補完を行います [58]。分割統治で文脈長の制約を緩和しつつ、局所的にはTypeScriptコンパイラの型推論（決定的な型推論ルール）も活用して矛盾を減らしています [59]。さらに、LLMに型補完タスクへ特化した**Fill-in-the-Type (FIT)**というファインチューニングを施し、与えられたコード中の穴埋めとして型を生成する性能を高めました [60] [61]。OpenTauは生成した型の質評価として、新たに「プログラムの型適用度（program typedness)」という指標も導入しています [57]。これはコード中のどれだけのシンボルに型が付与されたか、その程度を測るものです [57]。もっとも重要な評価は**型チェック通過率**で、OpenTauは独自のTypeScriptベンチマーク上で**47.4%**のファイルが完全に型チェックを通過し、従来比14.5ポイントの改善を達成したと報告しています [62]。残りのファイルも平均3.3件の型エラーまで減少しており、従来の汎用コード生成モデルより整合性が向上しました [62]。この結果は、LLMの強力な生成能力に探索・検証工程を組み合わせることで、より実用的な型定義補完が可能になることを示しています。

最後に、機械学習モデルを**開発支援ツールに組み込む**動きも紹介します。**FlexType**（Vorugantiら， 2022）は、Visual Studio Code上で動作するプラグイン型のフレームワークです [63] [64]。FlexTypeは任意のタイプ推論モデル（シーケンス系でもグラフ系でも）を背後で動かし、エディタ上で変数や関数にホバーした際に**候補の型一覧を提示**します [65] [66]。ユーザはその中から適切そうな型を選べば、AST操作によってコードに型注釈が正しく挿入されます [65] [66]。このように研究レベルのモデルをプラグイン化し、開発者がインタラクティブに利用できるようにした点がFlexTypeの貢献です [63] [64]。著者らの調査によれば、実務のJavaScript/TypeScriptコードではコンパイラ推論だけでは**63.46%の識別子が`any`のまま**（＝型ヒント不足）だったとのことで [7] [67]、モデルが提案する文脈に沿った型情報がIDE上で得られる意義は大きいと述べています。もっとも、最新の高精度モデルは巨大でローカルPCでは扱えないため、FlexTypeでは軽量モデルをローカル動作させる工夫もされています [68] [69]。FlexTypeは研究プロトタイプですが、将来的に開発支援IDEへの機械学習型補完の組み込み可能性を示すものでしょう。

型定義ファイルの品質と課題

自動生成された型定義（`.d.ts`）の品質には様々な観点があります。まず**正確性（サウンドネス）**の問題です。前述の通り、TypeScriptの型定義はあくまで開発時の仮想的な契約であり、実装と食い違ってもコンパイルエラーにはなりません [2]。したがって誤った型定義を配布すると、利用者は気付かずに誤使用し、実行時

にバグが顕在化する恐れがあります [70]。自動生成手法では、この**型の不健全さ**をいかに抑えるかが重要です。典型的には次の二つのズレが問題になります。

- **過剰な型の近似（over-approximation）**：型定義が実際より広すぎる場合です。例えば本当は戻り値が数値型の関数に対し、型定義を `any` や `object` としてしまうケースです。これは型チェック上はエラーを生じにくい（どんな使い方も許してしまう）ため、一見問題ないように見えます。しかしIDE補完では具体的な情報が得られず、誤った使い方も通してしまうため、**本来検出できたであろうバグを見逃す（偽陰性）**結果になります。過剰一般化の代表が `any` の乱用で、DeepTyperなど学習モデルも不確実な予測箇所では `any` を提案しがちです [53]。dts-genのような単純ツールも、型不明な箇所はひとまず `any` で埋める実装でした。そのため生成直後の型定義を人手で精査し、`any` を適切な型に置き換える作業が依然必要になることがあります。

- **過少な型の近似（under-approximation）**： 型定義が実際より狭すぎる場合です。例えば先述のabs関数の例で、本当は引数省略可能なのに `(input: string) => string` と必須にしてしまったケースです [18]。このように**実装が受け入れる有効な呼び出しを型定義が拒否する**と、利用側のTypeScriptコードで不要なエラー（偽陽性エラー）が発生します。ユーザは正しいコードを書いているのに型エラーが出るため、混乱や不信感を与えます [71] [72]。過少近似は動的解析系で観測不足により起こりやすく、dts-generateもドキュメント例外のケースでは「解決可能な差分(solvable difference)」として報告しています [73]。静的解析でも、保守的に特定の型だけ許容した結果、本来許される他型を排除してしまう場合があります。

理想的には実装の型集合と型定義の型集合が一致（**完全但し近似**）していることですが、動的言語では厳密な一致は難しく、上記どちらかに偏りがちです。研究ごとに「安全側に広めにとるか（エラーは出さないが型の厳密さは犠牲）」「厳しめに絞るか（役立つが誤検知リスク）」の設計判断が分かれます。TypeWeaverではコンパイルエラーを避けるため**unknown型の使用は最悪**で `any` の方がマシだと指摘しています（unknownは使用時に必ず型ガードや型アサートが必要なため、放置すると即エラーになる） [74]。一方、`any` はエラーにならないので一見成功率を上げますが型情報としての価値は低下します。このトレードオフは**精度（Accuracy）**と**型チェック通過率**のトレードオフとも言えます [75]。

次に**IDEでの補完品質**も重要です。型定義ファイルは開発者の手元で関数の引数ヒントやドキュメンテーション表示に使われます。誤った型や不完全な型では、IDEが的外れな補完を出したり、ドキュメントコメントが欠落する可能性があります [70]。例えばプロパティが存在しない型定義において、そのプロパティにアクセスするコードを書くとIDEは即座にエラーを表示します（過少近似の弊害）。逆に実装にないプロパティを型定義が含んでいると、IDEは存在しないメンバーを補完候補に出してしまうでしょう。したがって**IDE支援の面からも型定義の正確さ・完全さが要求**されます。

最後に**メンテナンス性**の課題です。自動生成した型定義をそのままプロジェクトに導入した後、ライブラリ本体がバージョンアップすると型定義も更新が必要になります。動的解析を再度走らせれば更新可能ですが、その精度が常に保証されるとは限りません。Cristianiらはライブラリの進化に伴う非互換を検出する**型定義比較ツール**も開発し、以前の型定義と新生成の差分から問題を発見できるようにしています [76] [77]。このような仕組みで継続的に型定義を管理することも今後は検討されます。また、LLM等の生成は確率的で同じ入力でも結果が変わることがあるため、再現性の確保や人的確認のコストも無視できません。

型定義自動生成の下流影響と評価手法

自動生成した型定義ファイルや注釈を実際のプロジェクトに適用すると、どのような影響が現れるでしょうか。本節では研究で報告された**下流タスクへの効果**と、評価指標の比較について述べます。

まず型定義導入の直接的な効果は、**型エラーの数や種類**に表れます。TypeWeaverの実験では、自動型付け後のコードをコンパイルした際に出現した型エラーを詳細に分析しています。典型例として多かったのは以下のようなエラーです [78]。

- プロパティ不存在エラー: オブジェクト型に本来あるはずのプロパティ定義が欠如しており、アクセス時に `Property X does not exist on type Y` といったエラーが発生する。
- 型の不一致エラー: 関数の引数型や戻り値型が不正確で、呼び出し側との型整合性が取れない（例：number型を期待する箇所にstring型を返すと定義してしまった）。
- 依存の型未解決エラー: 外部モジュールやグローバル変数の型が提供されず、`Cannot find name X` や `Cannot find module Y` といったエラーになる。
- unknownに起因するエラー: 予測で `unknown` 型を付与した結果、使用時に明示的な型ガード等が無くコンパイルエラーとなる。
- モジュール形式の不一致: CommonJS vs ESModuleの差異から、デフォルトエクスポートの扱いなどでエラーが出る（例： `import = require` が許容されない設定での不適切な書き方）。

TypeWeaverではエラーの上位10種をコード別に集計し、何がボトルネックかを明らかにしています [78]。その結果、プロパティ不存在や型不一致に関連するエラーコード（TS2339やTS2322など）が多くを占めていたようです。また、プロジェクト内のエラー分布を見ると「大半のエラーは一部のファイルに集中している」傾向も報告され、逆に言えば**多くのファイルは少数のエラー修正でコンパイルが通る**状況だったとのことです [78]。CommonJSからES6への変換を行うとモジュール解決関連のエラーが大きく減ることも示され、モジュールシステムの違いが無視できない要因であるとわかります [78]。

Downstreamへのポジティブな効果としては、型定義を追加することで**エディタ補助が充実しバグ検出能力が上がる**ことが期待されます。Vorugantiらは、多数の `any` が付いた状態から適切な型注釈が得られれば、IDEがより具体的な補完や警告を出せるようになると指摘しています [79]。特にAPIクライアントコードなどでは、レスポンスオブジェクトの型をquicktypeのようなツールで定義すれば、以降の開発でプロパティ名のタイプミスなどが即座に検出できるようになります。実行時に初めて現れるバグを事前に防ぐ効果は大きなメリットです。

評価指標については、従来の**型予測精度（Accuracy）**だけでは不十分であることが認識されつつあります [50]。Accuracyは与えられた変数の正解型を当てた割合ですが、これが高くともコード全体の一貫性は保証しません。そこで**型チェック通過率（Type-Check Success Rate）**が重要な指標として提案されています [50]。これは生成した型付きコードがエラーなくコンパイルできるかの率で、TypeWeaverやOpenTauが重視したものです。プロジェクト単位かファイル単位かで測定されますが、プロジェクト単位では一つでもエラーがあると失敗とみなすため非常に厳しい基準になります [52]。一方ファイル単位では部分的成功度合いも把握できます [52]。OpenTauはさらに**ファイルあたりの型エラー数**も報告しており、未解決エラーの程度を示す指標となっています [62]。

その他の評価指標には、**型適用度（Typedness）** [57]、**any率** [53]、**正解型との合致率（Accuracy や F1スコア）**、**実行時テストのパス率**などがあります。Typednessはコード中のシンボルの何割に型が付いたかを連続値で測る指標で、型付けの進行度合いを評価できます [57]。any率は生成された型定義中どの程度 `any` が使われているかで、実質的な型情報の欠落具合を見るものです [53]。正解型との合致は、既存の人筆による型定義がある場合に限りますが、モデルのトップN精度や精密度/再現率を見る指標として従来から使われています [29] [36]。最後に実行時のテストスイートがあれば、型付け後にテストが全て通るかを見るのも有効です。型を追加したことでコードを書き換えてはいないので、本来実行結果は変わらないはずですが、もし型定義の誤りから誤用が発覚したり、あるいは型エラー回避のためにコードを変更した結果バグを埋め込んでしまう可能性もゼロではありません。現状、そのような観点（**動的セマンティクスの維持**）まで評価した研究はほとんどありませんが、将来的には「自動型付けでバグ削減につながったか」といった評価も行われるかもしれません。

既存ツールの比較と限界

上述した研究・手法に関連して、実際に使えるOSSやサービスもいくつか存在します。それらの特徴と限界を簡単に整理します。

- **dts-gen**（Microsoft / DefinitelyTypedコミュニティ）：与えられたJSモジュールから雛形の型定義ファイルを生成するコマンドラインツールです。Node.jsで対象モジュールを`require`し、得られたオブジェクトのプロパティ構造を走査して型定義に落とし込む方式です。しかし詳細な型推論は行わず、多くの箇所を`any`や`unknown`で埋めるため、**粗いスケッチ**としての利用に留まります [80] [81]。「最低限これをベースに手動で編集してください」といった位置付けで、新規ライブラリの型定義を書く出発点にはなりますが、そのままでは品質が不十分です。

- **dtsmake**（ConquestArrow氏）：前述の通りTern.jsベースの静的解析ツールです。サンプルコードを与えることで`any`頼りを減らす工夫がされています [4] [5]。Qiita記事によれば「とりあえず使いたい場面で役立つことを目標」としており、**完全な網羅は目指していない**ようです [82] [83]。実際アルファ版クオリティで様々なJSコードでの検証が不十分とされています [84]。また2019年以降更新が無いようで、最新のJS/TS事情（ESモジュールや新構文）への対応も不明です。従って、**研究段階の試作に近い**状態であり、汎用性には課題があります。

- **tsd-jsdoc**：Google開発者らによるJSDocコメントからの`.d.ts`生成ツールです。JSDocコメントが充実しているプロジェクト（例えばGoogleのライブラリなど）では高品質な型定義を抽出できます。TypeScript公式ハンドブックのガイドラインに沿った生成を行うよう努めており、エクスポートの扱いなども適切です。とはいえ**コメント頼り**であるため、コメントがいい加減な場合は不正確な型になったり、コメントのない部分は結局`any`になります。

- **quicktype**：こちらは趣旨が少し異なり、JSONやGraphQLスキーマなど**データの例や模式**から対応する型定義（TypeScriptのinterfaceや型エイリアスなど）を生成するツールです [85]。たとえばJSONレスポンスのサンプルを与えると、その構造に一致するTypeScriptの型を出力します [86] [87]。また生成コードにはシリアライズ/デシリアライズ関数やランタイム型チェックコードも含めることが可能で、**実行時の安全性**までフォローします [88]。quicktypeは多言語対応しており、各言語の型定義やクラスを「ゴージャスで型安全なコード」として吐き出すことを謳っています [89]。TypeScriptに関して言えば、APIクライアントや設定ファイル読み込みなどで強力な道具ですが、**汎用のJSコードからAPIの意図を推測する用途には使えません**。すなわちquicktypeは**データ駆動**型の型定義生成であり、本調査の文脈（関数やライブラリAPIの型推論）とは適用範囲が異なります。しかし、たとえば動的型付きのREST APIクライアントをTypeScriptに移行する場合などには、レスポンス例から型を起こしてコード全体に適用するという意味で、型移行の重要な一部を担うツールと言えます。

- **Flow→TypeScript移行ツール**：FacebookのFlowからTypeScriptへの移行では、型定義の差異を埋めるツール（flowgenやtypescriptifyなど）が使われます。これらはFlowの型注釈やlibdefs（Flow版の型定義）を解析しTypeScript形式に変換するものですが、完全自動化は難しく手作業の微調整が必要です。動的型言語からではなく**他の型付き言語から**型定義を持ってくるアプローチですが、互換性の違い（例えばexact typeやutility typesの差）に起因する問題があります。従って特殊ケースでの利用に留まります。

- **学術プロトタイプ**：上記以外にも、研究プロジェクトとして実装が公開されているものがあります。TypeWeaverはGammaTauプロジェクトの一環でOSS公開されています [90]。OpenTauもGitHub上でコードとモデルを公開しており、再現実験や応用研究が可能です [90]。LambdaNetやTypilus、NL2Typeといったモデルも論文付属のコードが提供されています。ただ、これらを直接日常の開発フローに組み込むには専門知識が必要で、**一般開発者向けには敷居が高い**のが現状です。FlexTypeのようにIDE拡張として包装する試みはありますが、現時点では研究段階にとどまります [91] [63]。

以上のように、多くのツールや手法が提案・試作されていますが、「決定版」と言える万能な解決策はまだ存在しません。それぞれ**前提条件や得意不得意**があり、状況に応じて使い分ける必要があります。例えば、小規模なユーティリティ関数集ならdts-genやdtsmakeで雛形を生成し手で磨くのが早いかもしれません。大型のクライアントSDKならquicktypeでまずデータモデル型を生成し、残りは手作業、という手もあります。既存コードベース全体をTypeScriptに移行するには、TypeWeaverやOpenTauのような**高度な自動型付けフレームワーク**を使いつつ、人間がエラー部分を修正していくハイブリッドなアプローチが現実的でしょう [52] [62]。現存する限界として、**完全自動で高品質な型定義生成はまだ難しく、人手の介在や検証が不可欠**という点があります。しかし研究の進展は著しく、LLMの活用などにより徐々にそのギャップは埋まりつつあります [62]。

おわりに

本稿では、TypeScriptの型定義ファイル生成・補完技術について包括的にレビューしました。静的解析・動的解析・機械学習といった様々なアプローチが試みられ、それぞれ一長一短があることが分かりました。静的手法は即時性と軽量さで優れますが精度に課題があり、動的手法は具体性が高いものの網羅性に欠けます。機械学習は豊富なコード知識を活かせる一方で、結果の妥当性保証に追加の工夫（検証工程）が必要でした。

特に近年の研究は「型予測精度」より「型チェック通過率」や「移行成功率」に重きを置き、現実に使える型定義を自動生成することにフォーカスしています [50]。TypeWeaverやOpenTauによって、最新モデルでもプロジェクト全体を完全に型付けできるケースはまだ半数以下である実情が示されました [52] [62]。しかし同時に、ファイルレベルでは大部分が型チェックを通過するところまで来ており、残る課題は局所的なエラー修正や型の洗練であるとも言えます [52]。今後は、人間と自動化ツールの協調によりこの最後のギャップを埋める研究が進むでしょう。たとえば、生成ツールがエラー箇所を説明したり修正候補を提案する、あるいはIDE上で対話的に型定義を磨き上げるといった方向性です。

また、**生成型定義の品質評価**についても更なる検討が必要です。現状は型チェックという静的基準が主ですが、将来的には「実際にバグ削減につながったか」「開発生産性が向上したか」といった動的・人的観点での評価も視野に入るでしょう。型定義はあくまで手段であり、最終目的は**安全で効率的なソフトウェア開発**です。その意味で、型定義自動生成技術はまだ成熟途上ながらも、動的言語の開発スタイルに大きな変革をもたらす可能性を秘めています。TypeScriptを始めとする逐次型付け言語の普及に伴い、本分野の研究開発は今後も加速していくと考えられます。その動向に注視しつつ、実務への橋渡しとなるツール群の登場にも期待したいところです。

参考文献（一部抜粋）：

- Cristiani, F., & Thiemann, P. (2021). Generation of TypeScript Declaration Files from JavaScript Code. MPLR 2021. [2] [13] 他.
- Yee, M. et al. (2023). Do Machine Learning Models Produce TypeScript Types that Type Check? ECOOP 2023. [92] [52] 他.
- Cassano, F. et al. (2023). Type Prediction With Program Decomposition and Fill-in-the-Type Training. arXiv:2305.17145. [62] [60] 他.
- Hellendoorn, V. J. et al. (2018). Deep Learning Type Inference. ESEC/FSE 2018. [36] 他.
- Voruganti, S. et al. (2022). FlexType: A Plug-and-Play Framework for Type Inference Models. ASE 2022 (short paper). [65] [7] 他.
- Zhang, K. et al. (2019). NL2Type: Inferring JavaScript Function Types from Natural Language Information. ICSE 2019. [29] [27] 他.
- Qiita記事: **「TypeScript型定義ファイルのコツと生成ツール dtsmake」** (2015) [4] [3]. (dtsmake作者による解説).

1 3 4 5 82 83 84 TypeScript型定義ファイルのコツと生成ツール dtsmake #Tern - Qiita
http://qiita.com/ConquestArrow/items/450f961c3d54bc932cf3

2 11 12 13 14 15 16 17 18 19 20 21 22 23 24 70 71 72 73 76 77 Generation of TypeScript Declaration Files from JavaScript Code
https://arxiv.org/pdf/2108.08027

6 7 8 63 64 65 66 67 68 69 79 91 FlexType: A Plug-and-Play Framework for Type Inference Models
https://www.kevinrjesse.com/pdfs/FlexType_ASE22_preprint.pdf

9 10 31 32 35 36 37 38 39 Deep Learning Type Inference
https://vhellendoorn.github.io/fse2018-j2t.pdf

25 26 27 28 29 30 software-lab.org
https://www.software-lab.org/publications/icse2019_NL2Type.pdf

33 34 40 41 42 43 52 62 92 Publications and Other Works · Ming-Ho Yee
https://mhyee.com/publications/

44 45 46 47 48 49 50 51 53 54 55 75 78 TypeWeaver_ Learning Type Annotations from Examples 論文調査報告.pdf
file://file-6R3JyfdjD5GzHmPvMquMmZ

56 57 58 59 60 61 90 [2305.17145] Type Prediction With Program Decomposition and Fill-in-the-Type Training
https://ar5iv.labs.arxiv.org/html/2305.17145

74 [PDF] Predicting TypeScript Type Annotations and Definitions with ...
https://mhyee.com/publications/2024-phd-thesis-defense-slides-notes.pdf

80 81 Use dtsGen to generate declaration files for imported modules with no declarations · Issue #25746 · microsoft/TypeScript · GitHub
https://github.com/microsoft/TypeScript/issues/25746

85 86 87 88 89 Convert JSON to Swift, C#, TypeScript, Objective-C, Go, Java, C++ and more • quicktype
https://quicktype.io/