難しくない実用型パズル入門

～タグ付きユニオンを最大限活用する～

somnicat

2026-06-08

自己紹介

名前: somnicat (@somnicattus)

職業: Web アプリケーションエンジニア

使う技術: フルスタック TypeScript

タグ付きユニオン（判別可能なユニオン）

- 公式ドキュメントにも書かれている**必須級の型パターン**
- `type` などの共通のキーに識別用のリテラルを持たせる
- TypeScript でオブジェクトのユニオン型を扱うときのベストプラクティス

タグ付きユニオンのコード例

```
interface InsertCommand {
  type: "insert"; // 識別リテラル
  values: Document;
}
interface UpdateCommand {
  type: "update"; // 識別リテラル
  id: string;
  values: Document;
}
interface DeleteCommand {
  type: "delete"; // 識別リテラル
  id: string;
}
type Command = InsertCommand | UpdateCommand | DeleteCommand;
```

タグ付きユニオンの分岐処理と網羅性チェック

- `switch` 文などの条件でユニオン型の識別リテラルを絞り込む
- パターンマッチング風にタグごとに異なる処理をする
- `satisfies never` で 網羅性を確認する

タグによる分岐処理のコード例

```
switch (command.type) {
  case "insert":
    await insertDocument(command); // command の型は InsertCommand
    break;
  case "update":
    await updateDocument(command); // command の型は UpdateCommand
    break;
  case "delete":
    await deleteDocument(command); // command の型は DeleteCommand
    break;
  default:
    command satisfies never; // 網羅性を確認（到達不能型は漏れがあったらエラーになる）
    throw new TypeError(`Unexpected command ${command.type}`);
}
```

タグ付きユニオンの特徴

- パターンマッチ風に記述できる
  - JavaScript の言語機能不足を補完
- 自然に型の絞り込み (Type Narrowing) が行われる
  - 複雑なオブジェクト型よりシンプル・確実・安全
- クラスより分岐の取り回しがいい
  - `instanceof` は `switch` で使えない
  - `.constructor` へのアクセスは型推論が効かない

タグ付きユニオンは素晴らしい！

本題：型パズルをしよう

型パズルでタグ付きユニオンを使いやすくする

解消したい課題

- もとのユニオン型から部分的な型を取り出しづらい
  - `Extract` を使う？
  - 面倒なので別のユニオン型として列挙する？

```
type InsertOrUpdateCommand = Extract<Command, { type: "insert" | "update" }>;

type InsertOrUpdateCommand = InsertCommand | UpdateCommand;
```

選択可能なユニオン（独自の呼称？）

- タグ付きユニオンをジェネリクス化する
- 型引数で取り出したい型の識別リテラルを指定する
- 好きな識別リテラルの組み合わせで簡単に型を取り出せる

選択可能なユニオンのコード例

```
// 識別リテラルをプロパティ名にしたインターフェースにまとめる
interface CommandRegistry {
  insert: InsertCommand;
  update: UpdateCommand;
  delete: DeleteCommand;
}
type CommandType = keyof CommandRegistry;

// Lookup Type を使ってジェネリクスにする
type Command<T extends CommandType = CommandType> = CommandRegistry[T];
```

選択可能なユニオンの使用例

```
type All = Command;
// -> InsertCommand | UpdateCommand | DeleteCommand

type Single = Command<"insert">;
// -> InsertCommand

type Multiple = Command<"insert" | "update">;
// -> InsertCommand | UpdateCommand
```

- 好きな組み合わせのユニオン型を簡単に取り出せる
- デフォルトの型は全パターンのユニオン

仕組み 1: Lookup Type で型を取り出す

- オブジェクト型のキーを指定して対応する値の型を取り出す
- レジストリのキーにタグを使うとタグに対応したバリアント型が返る

```
type Insert = CommandRegistry["insert"];
// -> InsertCommand

type Update = CommandRegistry["update"];
// -> UpdateCommand
```

仕組み 2: ユニオン型のキーは分配される

- Lookup のキーにユニオンを渡すと結果もユニオンになる
- `CommandType` は `CommandRegistry` のキーを列挙したユニオン型

```
type Some = CommandRegistry["insert" | "update"];
// -> InsertCommand | UpdateCommand

type All = CommandRegistry[CommandType];
// -> InsertCommand | UpdateCommand | DeleteCommand
```

仕組み 3: ジェネリクスで窓口を１つに

- 型引数 `T` を Lookup のキーに使うだけ
- `T` のデフォルトを全タグにすると、デフォルトの型が全体ユニオンに

```
type Command<T extends CommandType = CommandType> = CommandRegistry[T];
//                                   ^^^^^^^^^^^^^^^ 省略時は全タグ

Command; //                          = CommandRegistry[CommandType]          (全体)
Command<"insert">; //                = CommandRegistry["insert"]             (単一)
Command<"insert" | "update">; // = CommandRegistry["insert" | "update"]  (部分)
```

選択可能なユニオンの嬉しいポイント

- 単一の型を通してすべての組み合わせのユニオン型にアクセスできる
  - 個別の型を `export` / `import` しなくていい
- 型引数に指定できる識別リテラルの候補が IDE で表示される
  - `Extract` を使う方法では候補が表示されない
- (おまけ) interface 宣言マージでユニオンを後から拡張できる
  - `declare module` でレジストリを拡張するとユニオンも拡張される
  - 拡張性の高いライブラリなど使われる Type Registry 型設計パターン

まとめ

- タグ付きユニオン型でユニオン型を安全に扱う
  - オブジェクトに識別リテラルを持たせる
  - 条件分岐で型の絞り込み（`switch` 文も使える）
  - `satisfies never` で網羅性チェック
- 選択可能なユニオンで取り回しをよくする
  - レジストリから Lookup で取り出すジェネリクス
  - 任意の組み合わせでユニオン型を取り出せる
  - 指定できるリテラルが型機能で制限されるので `Extract` より安全

以上、難しくない実用型パズル入門でした

Thank You