TypeScript エラーハンドリング つらい [検索🔍]

> VRChat.ts 4/27

ありあな

自己紹介

ありあな(Alliana)

- 普段は TypeScript をメインに書いている
- 静的型付け言語が大好き
- AI エージェントと協働するための仕組みづくりに興味がある

GitHub: Allianaab2m

X: @Alliana_VRC

イントロ

TypeScript エラーハンドリング つらい [検索🔍]

Zenn
https://zenn.dev › pepabo › articles

try-catch に疲れたので、TypeScript で別の書き方を考えた

2025/12/18 — JS/TSを長く触れていると、無意識に受け入れてしまっている不便さがあります。その一つが、**TypeScript**における非同期処理の**エラーハンドリング**です。

Qiita
https://qiita.com › TypeScript

TypeScriptのエラーハンドリングはResult型を使うのが良さ ...

2022/08/18 — **Result型を使うと例外スローの苦しみから開放される**・型安全でない（型から例外がスローされるかわからない）・扱いづらい（スローされた例外は、catchして ...

note・ダイニー公式
40 件以上の高評価・1 年前

noren.ts #1「TypeScriptのエラーハンドリングを極める」～ ダイ ...

ともかくコミュニティのデフォルトとして Error を throw というのが**辛い**。 ので、勉強会で**エラーハンドリング**について語って JS コミュニティを一 ...

Zenn
https://zenn.dev › dragon1208 › articles

try / catch 地獄から抜け出すためにResult 型を採用した話

2025/12/12 — try / catch が辛い理由・1. **制御フローが壊れる**・2. UI 側に例外が漏れやすい・3. 型安全ではない.

Reddit・r/typescript
130 件以上のコメント・2 年前

なんでTypeScriptの設定はこんなに難しいの？

プロジェクトでランタイム**エラー**をうまく**処理**するために、また他の**TypeScript**の機能を使うために**TypeScript**を使い始めたかったんだ。

結構ヒットする，みんなつらそう

そもそも何がつらいんだっけ？

throw / try-catch の良いところ

- **手軽** 複雑なことを考えなくても書ける
- フレームワークが拾ってくれるのなら気軽に `throw` できる
  - ex): Express の error middleware, Hono の `onError`, NestJS の Exception Filter

...でも，さすがにつらい場面が多い

でもつらい！

スコープの問題

```
let result; // <= try スコープ内の結果を使いたいとき，let で宣言しないといけない
try {
  result = throwableFn();
} catch (e) {
  // ...
}

// ここで result 変数を使う
```

- `try` は **式ではない** ので結果を受け取ることができない
- 即時関数で包むワークアラウンドもあるが，見た目がとっても微妙

でもつらい！

型から何も読み取れない

```
const foo = (): number => { ... };
```

- `throw` するかどうか型に**出てこない**
- 呼ぶ側で `try-catch` が必要かわからない
- `catch` を忘れてもコンパイラは何も言ってくれない

でもつらい！

catch した e の型は `unknown`

```
try {
  // ...
} catch (e) {
  // e は unknown
  if (e instanceof Error) {
    if (e instanceof MyCustomError) {
      // ...
    }
  }
}
```

- `instanceof` でエラークラスかどうか判定して分岐させなければならない

そもそも「エラー」は一括りでいいのか？

=> そんなことない

エラーは2種類ある

例外 (Exception)

想定していない事態

- DB 接続できない
- メモリリーク
- etc...

→ `throw` でいい
(`panic` 的な使い方)

アプリケーションエラー

想定している失敗

- リソースが見つからない
- 権限がない
- バリデーション失敗

→ **ちゃんと向き合うべきはこっち**

ここで主張

アプリケーションエラーを

例外の仕組みで扱うのはちょっと違くない？

そらつらいわけです

アプリケーションエラーをどう扱うか

候補はいろいろ

- 値として扱う（生のユニオン型）
- **タグ付きユニオン (Discriminated Union)**
- `Effect System`

値として扱う

エラーも `return` で返す

```
const maybeFail = (): number | Error => {
  if (Math.random() > 0.5) {
    return new Error();
  }
  return 42;
};
```

戻り値の型が `number | Error` になる

値として扱う

呼ぶ側

```
const main = () => {
  const value = maybeFail(); // number | Error

  if (value instanceof Error) {
    console.error(value);
    return;
  }

  console.log(value); // number ✨ ナローイングが効く
};
```

メリデメ

✅ 型にエラーが現れる

✅ ハンドリングの漏れに気づきやすい

❌ 早期 `return` しないと型が絞れない

❌ `Error` を継承しない値（文字列やリテラル型）をエラーにすると `instanceof` で判別できない

タグ付きユニオン (Discriminated Union)

Result 型 — タグで成功/失敗を区別する型

```
import { ok, err, Result } from "neverthrow";

class DivByZero extends Error {}

const divide = (a: number, b: number): Result<number, DivByZero> =>
  b === 0 ? err(new DivByZero()) : ok(a / b);
```

- `ok(value)` / `err(error)` でラップ
- 戻り値の型に**成功値とエラー値の両方**が現れる
- `Result<T, E>` の中身は

  `{ ok: true; value: T } | { ok: false; error: E }`

タグ付きユニオン (Discriminated Union)

使う側

```ts
const result = divide(10, 2);

if (result.isErr()) {
  console.error(result.error);
  return;
}

console.log(result.value); // number ✨ ナローイングが効く
```

- `isErr()` / `isOk()` で綺麗に分岐できる
- ハンドリング漏れがあるとそもそも `value` にアクセスできない
- `map` / `mapErr` / `andThen` で**合成も簡単**

非同期処理はどうする？

```
const fetchUser = (id: string): Promise<User> => { ... };
```

- `Promise<T, E>` じゃなくて `Promise<T>`
- **どんなエラーが飛んでくるか型から分からない**
- `Promise.catch((e) => {})` の `e` も結局 `unknown`

Result 型を非同期処理で使うには

`neverthrow` には `ResultAsync<T, E>` という非同期版もある：

```
import { ResultAsync } from "neverthrow";

const fetchUser = (id: string): ResultAsync<User, FetchError> =>
  ResultAsync.fromPromise(
    fetch(`/users/${id}`).then((r) => r.json()),
    (e) => new FetchError(),
  );
```

- `Promise<T>` を `ResultAsync<T, E>` に変換
- `map` / `andThen` で同期処理と同じ感覚で繋げられる

Result で合成も書ける

```
const main = fetchUser("123")
  .andThen((user) => fetchPosts(user.id))
  .map((posts) => posts.length)
  .mapErr((e) => console.error(e));
```

- `await` も `if (!result.ok)` も書かなくていい
- 同期の `Result` と**まったく同じ感覚**で繋げる
- エラー型は自動的にユニオンとして `E` に合成される

Result 型を入れると

アプリケーションエラーが

型に乗ってきて安心 ✨

- どこでエラーが起きうるか型から分かる
- ハンドリング漏れした場合はそもそも正しいコードが書けない
- 非同期も同じ見た目で書ける

もう一歩進めたいなら: Effect-TS

```
Effect<A, E, R>;
//     ^  ^  ^
//     |  |  +-- 必要な依存（DI）
//     |  +------ 起こりうるエラー型
//     +--------- 成功値の型
```

- Result 型の発展形．**エラーも依存も全部型に乗る**
- `pipe` で合成，`catchTag` でタグ別ハンドリング
- 並行処理 / リトライ / タイムアウト / DI まで全部入り

Effect-TS のコード例

```ts
import { Effect, Data } from "effect";

class DivByZero extends Data.TaggedError("DivByZero")<{}> {}

const divide = (a: number, b: number) =>
  b === 0 ? Effect.fail(new DivByZero()) : Effect.succeed(a / b);

const program = divide(10, 2).pipe(
  Effect.map((n) => n * 3),
  Effect.catchTag("DivByZero", () => Effect.succeed(0)),
);
```

- 全部入り，関数型的な書き方を強要するので**学習コストはそれなり**
- 「Result 型だけで十分」なら neverthrow で OK

個人的な使い分け

シーン	おすすめ
小さいツール / スクリプト	`throw` で十分
アプリケーションエラーを ちゃんと扱いたい	**neverthrow**
エラーも依存も全部型に乗せたい	Effect-TS

=> Result 型を入れるだけでも世界が変わる

- アプリケーションエラーが**型に乗る安心感**
- ハンドリング漏れに**コンパイル時に気づける**
- 非同期も**同じ流儀**で書ける

エラーハンドリング，~~つらい~~ もうつらくないかも 🔍

質問 / 感想 お待ちしています！

GitHub: Allianaab2m

X: @Alliana_VRC