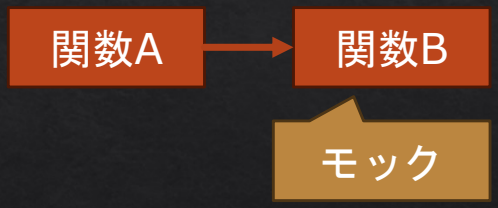
TypeScriptで書く
単体テストの考え方

自己紹介

- 神戸市在住のシステムエンジニア
- Web制作→ゲーム開発→業務オープン系→業務Web系→toC Web系
- 帳票・バッチ・サバクラ・Webフロント・Webバックエンド・インフラ・DevOps・RPA…

単体テストとは？

- 関数やクラスのメソッド単体の振る舞いを見るテスト
  - 関数やクラスのメソッドから、別の関数やクラスのメソッドを呼ぶ場合、呼ぶ対象の振る舞いは見ない
- 依存する振る舞いがある場合、その振る舞いはモックにする
  - 例えば、下図にある関数Aの単体テストを行うときは関数Bはモックする

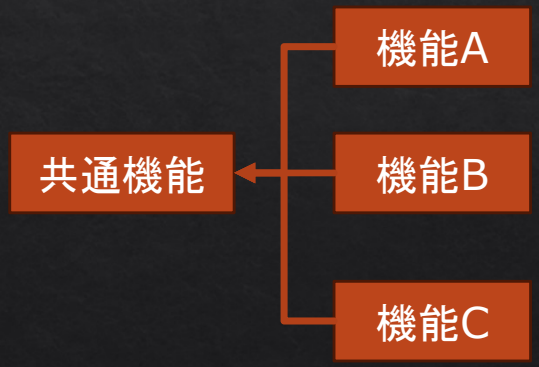


何のために単体テストをするか？

- テストコードを書いていると仕様の漏れに気づけることがある
  - 仕様や実装の矛盾に気づく切欠として有用だと思います
- 初回実装時でも予めテストコードを書いておくことで、「やっぱこう書こう」とかで直したときに不具合になることがあり、これを防ぐ効果がある
  - ちょろっとコードを直したら動かなくなった経験ありませんか？
- コードに改修がかかった時に不具合に気づきやすくなる
  - 仕様変更やリファクタリングなど

何故単体テストを書くか？

- 単体テストは結合テストやe2eテストに比べ実行速度が速い
- 単体テストは単体しか見ないので、改修の影響を受けづらく、保守コストが下がる
- 全て結合テストにしてしまうと、共通処理を毎回テストする必要があり、保守性が落ちる
- 単体テストは関数単体をテストするので、一つのテストの単位を小さくできる



単体テストを何で書くか？

- 任意のテストライブラリを使って書く
  - Jest, Vitest, Node.jsやDenoの組み込み機能など
- 個人的にはJestがオススメ
  - エコシステムが充実していてプラグインも多くあり、Linterもあるため

単体テスト向きの設計

- SOLID原則
  - S:単一責任の原則により関数の責務が小さくなり、関数が小さくなる
  - O:開放閉鎖の原則により、関数を呼ぶ関数自体が不変となり、テストの保守性が上がる
  - D: 依存逆転の法則により、関数内で呼び出す関数をテストオブジェクトに置換できる

```
1   type ReadFileSyncProps = {
2     openFile: (path: string) => number;
3     readFileSync: (fp: number) => Buffer;
4     closeFile: (fp: number) => void;
5     path: string;
6   }
7
8   const readFileSync = (props: ReadFileSyncProps) => {
9       const fp = props.openFile(props.path);
10      const data = props.readFileSync(fp);
11      props.closeFile(fp);
12      return data;
13  }
```

どのような単体テストを作るか

- 関数単体で完結するテスト
- 別の関数を呼び出す関数のテスト
- クラスのメソッドを呼ぶ単体テスト
- 外部の関数を呼び出すテスト
  - ライブラリとか
- テストが落ちることを確認するためのテスト

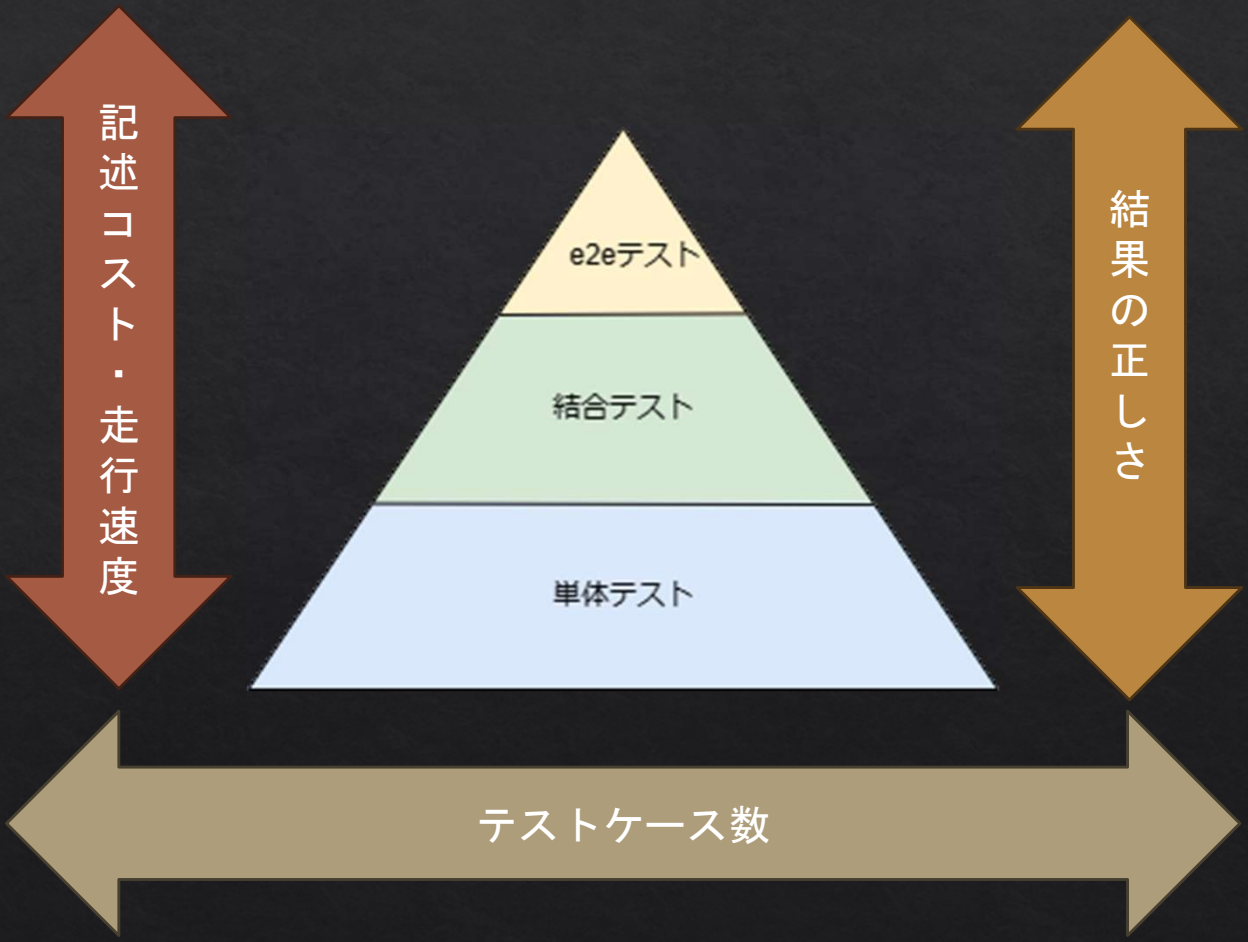
TypeScriptの特性を生かしたテストの作り方

- 型で保証できる部分は型で保証する
  - 型を信頼できない場所は全てテストで埋めるくらいの覚悟を持つ
  - 型を信頼できる場所はテストしない
  - anyとか{[key: string]: T}みたいな何でも入る実装を作るのは極力避ける
  - APIの戻り値など、外部からくる値については必ずバリデーションする

単体テストの欠点

- 信頼できない
  - 特に関数から関数を呼ぶ場所はモックであり、型での保証がなければ何も信じられない
- 広い意味での可読性の悪化
  - 責務を分けて関数に切り出すと疎結合になり、処理の全体像が掴みづらくなる
  - 全部一つにまとめた関数の方がパット見は分かりやすい

テストピラミッド



単体テストは銀の弾丸ではない

- 単体テストは不具合を減らすのに貢献するツールです
- 単体テストは関数やクラスのメソッドを単体で見た場合のふるまいを保証するためのものです
- 関数呼び出しの単体テストも書けますし、型で保障されていればそこそこ安全ですが絶対ではありません