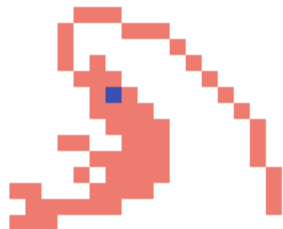


複雑なビジネスルールに挑む：  
**正確性** と **効率性** を両立する  
📎 fp-tsのチーム活用術

@kosui



## 自己紹介 (1分)

kosui (岩佐 幸翠)

株式会社カケハシにて共通基盤を開発

X @kosui\_me


@iwasa-kosui

<https://kosui.me>

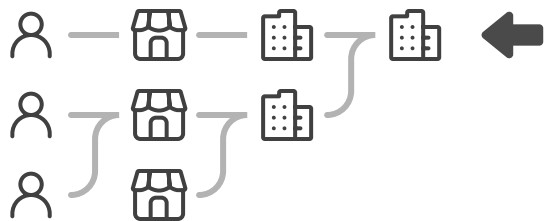
# この発表のねらい

## fp-ts で複雑性と闘う

複雑なビジネスロジックでも **正確性** と **利便性** を両立するために  
エラーや関数を合成できる **fp-ts** が便利！

実際に  一括入稿機能の開発で遭遇した悩みと解決策を紹介

### 組織管理システム



### 一括入稿機能

ユーザー		店舗		部門	
名前	所属	店名	割当先	部門名	親部門
飯塚	横浜店	横浜店	神奈川	神奈川	関東
角田	渋谷店	渋谷店	東京	東京	関東
豊本	渋谷店	中野店	東京	関東	東日本

## fp-ts をチームで使いこなすために

関数型プログラミング初学者の多いチームをどのようにスケールさせるか

# 目次

## 1. 背景

SaaSのエンタープライズ対応と  Excel一括入稿機能の重要性

## 2. 課題

 Excel一括入稿機能で 正確 かつ 効率的 に検証したい

## 3. fp-tsによるエラー合成

## 4. fp-tsのチーム活用

背景

# SaaSのエンタープライズ対応と ファイル一括入稿機能の重要性



# SaaS (Software as a Service) の発展と変化

SaaS業界が発展するにつれ、要求される機能や品質も変化している

## 個人・中小企業向け

- 差別化につながる先進的な機能・品質
- 安心して長期利用できるプライシング

## 大企業・行政向け

- 誰でも使える明快さ
- 上場企業も安心して使えるコンプライアンス
- 組織管理

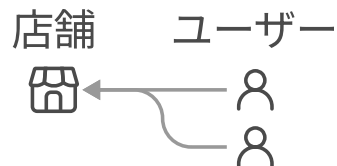
大規模で階層的な組織を

効率的かつ柔軟に管理できる

# SaaSの組織管理

## 個人・中小企業向け

データは **少量** で構造も **シンプル**

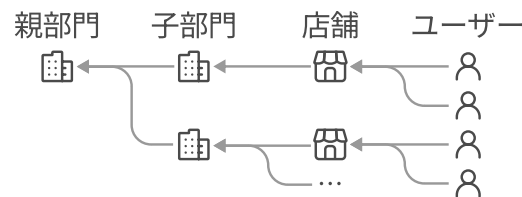


**GUI** から登録したい

ユーザー	店舗
名前 田中	店名 新宿店
生年月日 97/06/01	割当先 東京 ▼
所属 新宿店 ▼	

## 大企業・行政向け

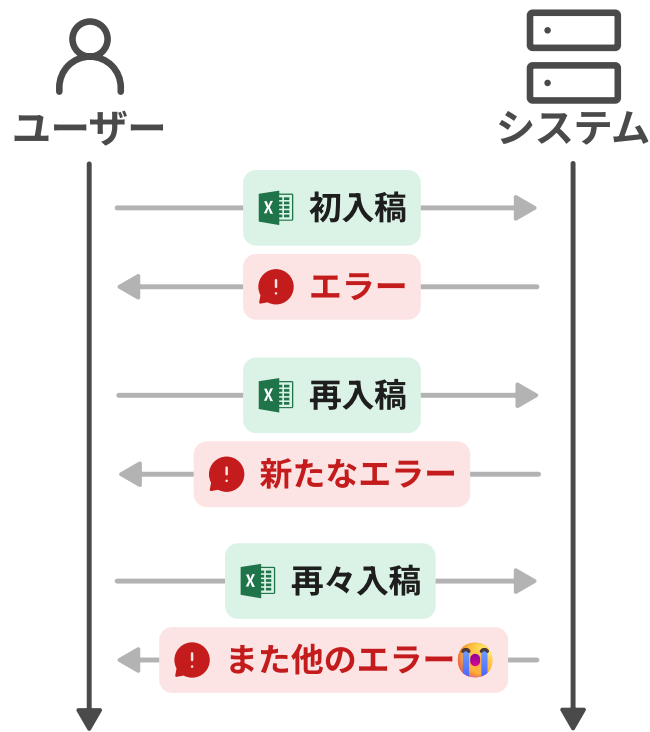
データが **大量** で **階層的**



**ファイル** から一括入稿したい

ユーザー			店舗		部門	
名前	生年月日	所属	店名	割当先	部門名	親部門
山田	97/06/01	新宿店	新宿店	東京	東京	関東
田中	54/10/12	渋谷店	渋谷店	東京	神奈川	関東
山本	70/10/01	中野店	中野店	東京	関東	東日本

# ユーザー から見た ファイル一括入稿のつらさ



## 不明瞭なフィードバック

エラーの原因が判別しにくい

➡ 修正作業は多大な時間と労力を伴う

- 修正すべき箇所(セル)が不明瞭
- 修正すべき理由が不明瞭

データ型？ 使用不能文字？ 値の重複？

他のシートと依存関係がある場合は特に難しい

## 繰り返される再入稿

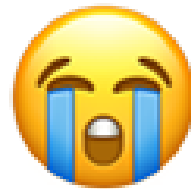
全てのエラーが一度に返却されない

➡ 何度も修正と再入稿を繰り返す

課題

表形式データの検証

正確性と効率性を阻む 2つの壁



# 表形式データの検証の3ステップ

## セルをパース

データ型やフォーマットを検証

例) 不正な日付、絵文字の混入



## 行を検証

行の重複や参照関係を検証

例) IDの重複チェック



## 表を検証

他の表との依存関係などを検証

例) 店舗IDの存在チェック

2	田中	54/10/12	66
ID	名前	生年月日	店舗ID
1	山田	97/06/01	22
2	田中	54/10/12	66
3	山本	70/10/01	66

ユーザー				店舗			部門	
ID	名前	生年月日	店舗ID	ID	店名	割当先	部門名	親
1	山田	97/06/01	22	22	新宿店	東京	東京	関
2	田中	54/10/12	66	44	渋谷店	東京	神奈川	関
3	山本	70/10/01	66	66	横浜店	神奈川	関東	東

## 表形式データ検証の壁①

# 表形式データのエラーハンドリング

### セルをパース

1行目がエラー  
不正な日付セル

2	山田	24/05/ <b>32</b>	22
2	田中	97/06/01	66
3	山本	70/10/01	999

### 行を検証

2行目がエラー  
IDが1行目と重複

ID	名前	生年月日	店舗ID
<b>2</b>	山田	24/05/32	22
<b>2</b>	田中	97/06/01	66
3	山本	70/10/01	999

### シートを検証

3行目がエラー  
店舗IDが店舗シートに存在しない

ユーザー				店舗	
ID	名前	生年月日	店舗ID	ID	...
2	山田	24/05/32	22	<b>22</b>	
2	田中	97/06/01	66	<b>44</b>	
3	山本	70/10/01	<b>999</b>	<b>66</b>	

## 理想

2・3行目の検証は続行

3行目の検証は続行

1・2・3行目のエラーを一括返却

## ナイーブな実装

例外を発生させ終了



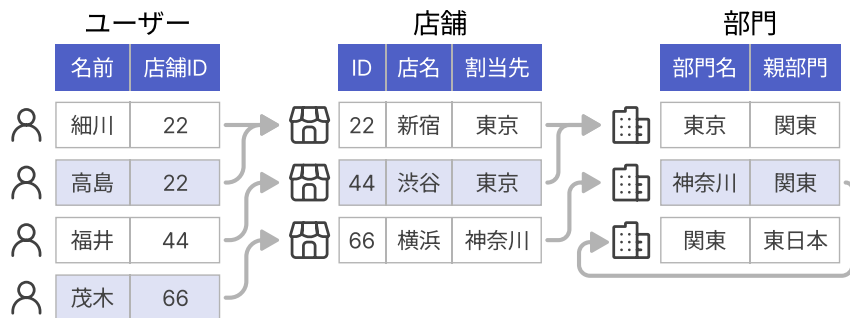
ユーザーは再入稿しないと他のエラーに気付けない

## 表形式データ検証の壁②

# 依存関係の解決

組織の階層構造を表現するために、ある行が他シートの行を参照する

例) ユーザーの所属を表現するため店舗シートを参照



😱 検証済みの行を参照したいのに  
未検証の行を参照してしまった

➡ 型検査で未然に防ぎたい

解決策

fp-tsによるエラー合成



# TypeScriptのエラーハンドリング

まずはよく知られたエラーハンドリングの手法をおさらい

## ユーザー定義エラーをthrowする

```
class MyError extends Error {  
  constructor(message) {  
    super(message);  
    this.name = "MyError";  
  }  
}
```

## Either型(Result型)を返す

```
type Either<E, A> = Left<E> | Right<A>;  
  
type Left<T> = Readonly<{  
  tag: 'Left';  
  left: T;  
  
type Right<A> = Readonly<{  
  tag: 'Right';  
  right: A;  

```

## TypeScriptのエラーハンドリング①

# ユーザー定義エラーをthrowする

```
class ParseError extends Error {  
  constructor(readonly row: number, msg?: string) {  
    //          ^^^ 行番号を持たせる  
    super(msg);  
    this.name = "ParseError";  
  }  
}
```

```
throw new ParseError(1);
```

```
if (err instanceof ParseError) {  
  console.error(err.row)  
}
```

**Error** を拡張したクラスを定義する

- エラー原因の情報を追加できる
- 例外としてthrowする
- **instanceof** 演算子によりエラーを識別できる

## 😭 例外をthrowする場合の悩み

複数のエラーを同時に伝搬しづらい

```
const parseUserRow = (cells: string[]) => ({
  id: parseUserId(cells[0]),          // 💥 throw err
  name: parseUsername(cells[1]),      // 🙋 bye err
  birthday: parseBirthday(cells[2]), // 🙋 bye err
});
```

```
const errors: Error = [];
let id: string;
try {
  id = parseUserId(cells[0]);
} catch(e) {
  errors.push(e);
}
// ...
```

`parseUserId` が例外を投げてしまうと  
他のセルの検証エラーをクライアントへ  
返せないまま処理が中断

try...catch文で頑張れば出来なくはない

しかし、流石にこれはつらい

## TypeScriptのエラーハンドリング②

# Either型(Result型)を返す

```
type Either<E, A> = Left<E> | Right<A>;  
  
type Left<T> = Readonly<{ tag: 'Left'; left: T; }>;  
  
type Right<A> = Readonly<{ tag: 'Right'; right: A; }>;
```

```
declare const either: Either<ParseError, string>;  
if (either.tag === 'Right') {  
  console.log(either.right);  
} else {  
  // OK  
  console.error(either.left);  
  // Property 'right' does not exist...  
  console.log(either.right);  
}
```

Discriminated

判別可能なUnion型を用いて  
成功した場合と失敗した場合を表現

- 失敗する可能性を型で表現できる
- **tag** プロパティを見れば  
型の絞り込みができる

# Either型を用いたエラーの伝搬

```
declare const parseUserId:  
  (v: string) => Either<ParseError, string>;  
  
declare const parseUsername:  
  (v: string) => Either<ParseError, string>;
```

それぞれのセルのパース関数が

`Either<ParseError, string>` を返す

```
const parseUserRow = (cells: string[]) => ({  
  id: parseUserId(cells[0]),      // Left  
  name: parseUsername(cells[1]), // Left  
});
```

IDと名前の両方のセルで  
パースに失敗した場合も...

```
console.log(parseUserRow(['bad', 'bad']));  
// {  
//   id: { left: ... },
```

👏 両方のセルのエラーを伝搬できる

```
// name: { left: ... },  
// }
```

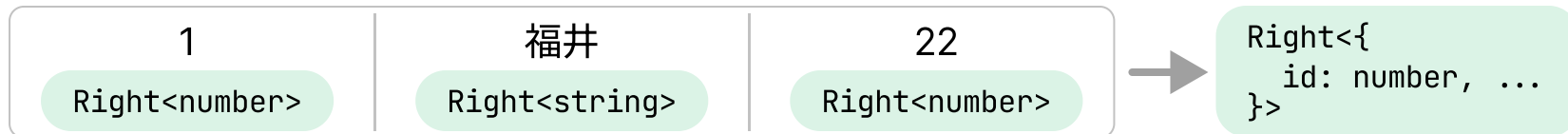
それぞれのセルを行へ合成したい

ID	名前	店舗ID
----	----	------

エラーとなるセルが含まれる場合



エラーとなるセルが無い場合



# それぞれのセルを行へ合成したい

エラーとなるセルが含まれる場合

```
{  
  id: left([new ParseError(1, 'ID')]),  
  name: left([new ParseError(1, '名前')]),  
  storeId: right(22),  
};
```



```
left([  
  new ParseError(1, 'ID'),  
  new ParseError(1, '名前'),  
]);
```

エラーとなるセルが無い場合

```
{  
  id: right(1),  
  name: right('田中'),  
  storeId: right(22),  
};
```



```
right({  
  id: right(1),  
  name: right('田中'),  
  storeId: right(22),  
});
```

# それぞれのセルを行へ合成したい

```
const errs: ParseError[] = [];  
if (isLeft(id)) {  
  errs.push(id.left);  
}  
if (isLeft(name)) {  
  errs.push(name.left);  
}  
  
if (errs) {  
  return left(errs);  
}  
  
assert(isRight(id));  
assert(isRight(name));  
return right({  
  id: id.right,  
  name: name.right;  
})
```

## 自前実装

いずれかのセルが **Left** の場合

エラーを取り出して配列に詰め

**Left<ParseError[]>** として返す

全てのセルが **Right** の場合

**Right<{ id: number, ... }>** として返す

## ライブラリに頼りたい

😓 自分でやりたくはない

それぞれのセルを行へ合成したい

## fp-tsによるオブジェクトのエラー合成

```
const cells = {  
  id: left([new ParseError(1)]),  
  name: left([new ParseError(1)]),  
};
```



```
type Row = Either<ParseError[], {  
  id: string,  
  name: string,  
}>;
```

```
import * as AP from 'fp-ts/Apply';  
import * as A from 'fp-ts/Array';  
import * as E from 'fp-ts/Either';  
const ap = E.getApplicativeValidation(  
  A.getSemigroup<string>(),  
);
```

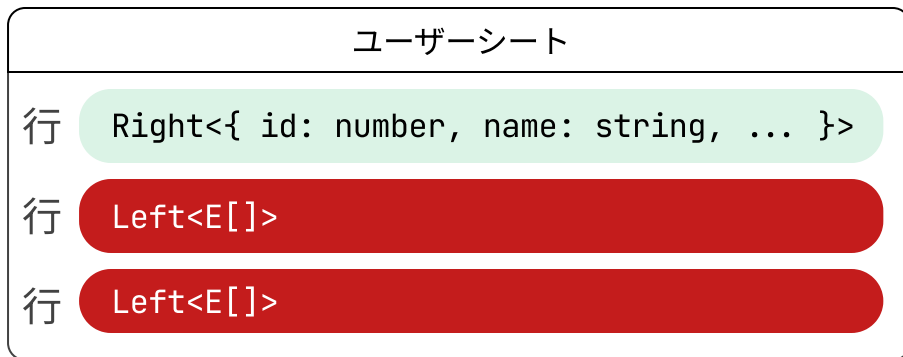
1. `Left` の `ParseError[]` を  
結合する関数 `ap` を定義

```
// 行へ合成  
const row: Row = AP.sequenceS(ap)(cells);
```

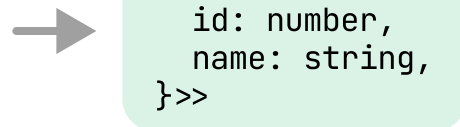
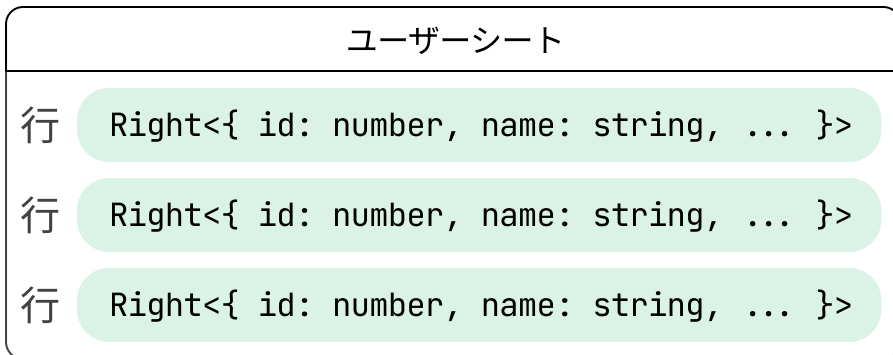
2. 先ほど定義した `ap` を用いて  
`Record<string, Either<>>` を合成できる

# それぞれの行をシートへ合成したい

エラーとなる行を含む場合



エラーとなる行が無い場合



fp-tsによるエラー合成

## 配列のエラー合成

やりたいこと

```
const rows = [  
  left([new ParseError(1)]),  
  right({  
    id: 2,  
    name: '田中',  
  }),  
  left([new ParseError(3)]),  
];
```



```
const sheet = left([  
  new ParseError(1),  
  new ParseError(3),  
]);
```

実現方法

```
// 行へ合成  
const sheet: Row = A.sequence(ap)(rows);
```

先ほど定義した **ap** を用いて

**Either** の配列を合成できる 🎉

fp-tsによるエラー合成

## セル→行→シートまで一気通貫で合成

```
import { pipe } from 'fp-ts/function';
pipe(
  [
    {
      id: left([new ParseError(1)]),
      name: left([new ParseError(1)]),
    },
    {
      id: left([new ParseError(1)]),
      name: left([new ParseError(1)]),
    },
  ],
  A.map(AP.sequenceS(ap)), // 1. 行へ合成
  A.sequence(ap), // 2. シートへ合成
);
```

1. 各セルで構成されるオブジェクト ➡ 行へ
2. 行の配列 ➡ シートへ

合成処理を簡潔に表現できる 🤖

顧客の価値に直結

冒頭でも述べた通り表形式データの検証は  
極力 一度に 全てのエラーを返すことが  
顧客体験のために重要

# fp-tsのチーム活用



# fp-tsの難しさ

## コンセプト

関数型プログラミングのためのユーティリティ

## 難点

- 非常に抽象度が高い
  - ドキュメントも抽象度が高い
- 日本語の情報が少ない
- 業務での実用例の解説が少ない

チームでfp-tsを利用するために①

## WIP プロジェクトで利用するものを限定する

- pipe/flow
- Either/Option
- Task/TaskEither

WIP

チームでfp-tsを利用するために②

## WIP ペアプロ・モブプロの活用

WIP

チームでfp-tsを利用するために③

## WIP 社内向けレシピ集

WIP