

# 関数型ドメインモデリングのすすめ

somnicat

September 21, 2026

# 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 (Vue/Node.js)



# 今日話すこと

---

1. ドメインモデリングって何？
2. ドメインモデリングとアーキテクチャの関係
3. 関数型って何？
4. 関数型ドメインモデリングって何？何が嬉しいの？

ドメインモデリングとは

# ドメインモデリングとは ①

---

## 業務の言葉でシステムを表現する

- データの持ち方やデータフローではなく、業務の専門家と同じ言葉をユビキタス言語として使う
- 会話・設計・コードで、同じものを同じ名前と呼ぶ

「申請」「承認者」「差し戻し」

—— 業務で使われている言葉が、そのままソースコード上の名前になる

# ドメインモデリングとは ② (重要！)

---

## 常に守るルールを洗い出して守る

- 「注文は確定したら書き換えない」のような規則を整理する
- 契約と検証でシステム全体がルールを満たすようにする
  - 契約: 信頼できる相手とのやり取りの約束 (破られるとバグ)
  - 検証: 信頼できないやり取りの防御 (破られても想定内のエラー)

「承認済みの申請は取り下げられない」

—— 運用ルールに書かれたこの一文を、設計とコードに落とし込む

# ドメインモデリングとは ③

---

## 文脈ごとに関心を区切る

- 実際の業務分担に沿って関心の範囲を区切る
- 境界付けられたコンテキストの範囲ごとにモデリングを行う

同じ「申請」でも、勤怠にとっては休む日数、経理にとっては手当の計算対象  
—— 1つの巨大なモデルで全部を背負わない

# ドメインモデリングとアーキテクチャ

# ドメインモデリングとアーキテクチャ ①

---

## ドメインモデルを **中心** に据える

- 業務のルールを表すコードを、アプリケーションの一番内側に置く
- 依存の向きを外側から内側へ。ドメインモデルは何にも依存しない
  - DI (依存性の注入) などを使って依存を追い出す

「承認済みの申請は取り下げられない」は、DB のテーブル定義も HTTP のリクエスト形式も知らずに書く

# ドメインモデリングとアーキテクチャ ② (重要！)

---

## 技術的な都合を 外側 に追い出す

- DB・Web・外部 API は「どう実現するか」の話であって、ドメインモデルが表す「業務が何であるか」とは別
- 使う技術に関係なく純粋なロジックをテストできるように隔離する

LeaveRequest の定義に、テーブル名もカラム属性も出てこない  
—— 自動採番の ID は業務の概念ではなく、保存方法の都合

# ドメインモデリングとアーキテクチャ ③

---

## 外側との出入りは 境界で変換する

- レコード行や API レスポンスは、境界でドメインモデルに詰め替える
- 変換の責任は外側が持つ。モデルは外の形式を知らない

ORM のエンティティをそのままモデルとして使うと、DB の都合がモデルに侵入する

—— 「全カラム nullable」に引きずられて、業務のルールが型から消える

# ドメインモデリングのまとめ

---

## ドメインモデリングとは

- 業務の言葉でシステムを表現する
- 常に守るルールを洗い出して守る
- 文脈ごとに関心を区切る

## ドメインモデリングとアーキテクチャ

- ドメインモデルを 中心 に据える
- 技術的な都合を 外側 に追い出す
- 外側との出入りは 境界で変換する

**Thank You!**

...関数型の話がまだですね

# 関数型プログラミングとは

# 関数型プログラミングとは ① (重要！)

---

## 純粋関数で処理を書く

- 同じ入力には必ず同じ出力を返す
- 外の状態を書き換ええない（副作用がない）
- 結果が予測しやすく、テストや並列化が簡単になる

関数の中身を読まなくても、引数と戻り値の型だけで何が起きるか分かる  
—— 隠れた入力も、隠れた出力もない

# 関数型プログラミングとは ② (重要！)

---

## データを 不変 に保つ

- データを書き換えず、変更が必要なら新しい値を作る
- 「いつ誰が変えたか」を追う必要がなくなる

渡した先で書き換えられている心配がないので、コピーして渡す必要もない  
—— 値は「そのとき何であったか」を変えない

# 関数型プログラミングとは ③

---

## 関数を 値として扱う

- 関数を引数に渡したり、戻り値として返したりできる
- 小さな関数を組み合わせて大きな処理を作る（`map` / `filter` / `reduce`）

「繰り返す」「選ぶ」「畳み込む」といった処理の骨格を、部品として使い回す  
—— 変えたいところだけを関数として差し込む

# 命令型と関数型

---

「リストの偶数を 2 乗して合計する」

```
// 命令型: ループを回し、合計用の変数を少しずつ書き換える
let sum = 0;
for (const x of xs) {
  if (isEven(x)) sum += square(x);
}
```

```
// 関数型: 変換をつなげる
const sum = xs.filter(isEven).map(square).reduce(add, 0);
```

# 関数型プログラミングのまとめ

---

## 関数型プログラミングとは

- 純粋関数で処理を書く
- データを 不変 に保つ
- 関数を 値として扱う

## 一言でまとめると

- 「何をどう変えるか」ではなく「何が何であるか」を記述するスタイル
- 副作用はゼロにできない。だからプログラムの端に押しやって、中心部を純粋に保つ
  - ドメインモデリングで「中心」に据えたものと、同じ場所の話

関数型ドメインモデリングとは

# 関数型ドメインモデリングとは ①

---

## あり得る状態だけを型で表現して束ねる

- 型で表現できる形と、業務で起こりうる形を一致させる
  - Make illegal states unrepresentable
- 状態ごとに別の型を用意し、**直和型**でまとめる
- optional の掛け算で膨らんだ「ありえない組み合わせ」が消える

OOP では、 `status` と optional フラグの組み合わせをコードに散らばった `if` が守っている

—— その `if` を、型の定義 1 か所に引き上げる

# OOP: 状態をフラグで持つ

```
// 休暇申請
class LeaveRequest {
  status: "draft" | "submitted" | "approved" | "rejected";
  period: DateRange;
  submittedAt?: Date;
  approver?: User;
  reason?: string;
}
```

status が 4 値、optional が 3 つ。この型が表現できる形は  $4 \times 2^3 = 32$  通り  
業務上ありうるのは **4 通り**。残り 28 通り（例:「draft なのに approver がいる」）を  
防いでいるのは、散らばった if とレビュアーの記憶

# FP: 状態ごとに型を分ける

```
type LeaveRequest =  
  | { kind: "draft"; period: DateRange }  
  | { kind: "submitted"; period: DateRange; submittedAt: Date }  
  | { kind: "approved"; period: DateRange; submittedAt: Date; approver: User }  
  | {  
    kind: "rejected";  
    period: DateRange;  
    submittedAt: Date;  
    approver: User;  
    reason: string;  
  };
```

32 通りが 4 通りになり、その 4 通りがそのまま 4 行として目に見えている

# 関数型ドメインモデリングとは ②

## 関数のシグネチャでデータの流を絞る

- 未検証 → 検証済み → 承認済み。関数のシグネチャが業務の流れそのもの
- `if` 文で「何を許すか」を検証する代わりに、型が「何ができるか」を契約する

```
submit: (r: DraftRequest) => SubmittedRequest;  
approve: (r: SubmittedRequest, by: User) => ApprovedRequest;  
reject: (r: SubmittedRequest, by: User, reason: string) => RejectedRequest;
```

approve の引数は SubmittedRequest だけ。ApprovedRequest を渡す口がない  
—— 「承認済みの申請は再承認できない」がシグネチャに書いてある

# なぜ「関数型」なのに型駆動のような話？

---

関数型の思想でデータを 不変 に保つと、自然とこの形になる

- 値を書き換えないので、状態の変化は新しい値を作る
- 値を書き換えないので、状態ごとに別の型を作れる  
-> 束ねると直和型になる
- 変化させる処理は「型 → 型」の関数として書かれる

# 関数型ドメインモデリングのまとめ

---

原則は 1 つ: 表現できる形を、ありうる形だけにする

- あり得るデータを絞る（直和型）
- あり得る操作を絞る（型の変換パイプライン）

一言でまとめると

- ドメインモデリングで見つけた常に守るルールを、型として書き下す
- レビューと記憶で守っていたものを、コンパイラに守らせる

# 何が嬉しいのか

---

## 常に守るルールの所在が明確になる

- 「承認済みの申請には承認者が設定される」がユニオン型の 1 か所にある
- 「承認済みの申請は取り下げられない」が関数シグネチャの 1 か所にある
- コードを読み込まなくても、どんなルールがあるか型を見れば分かる

## 純粋性が依存の向きを強制する

- 純粋関数は DB も HTTP も呼べない。内側から外側への依存が書けない
- 心がけや規約ではなく、純粋であることが中心を守る

関数型ドメインモデリングを始めたい！

# 必要な言語機能

---

1. 直和型 —— 判別可能 union / enum with data / sealed interface
2. 網羅性チェック —— 分岐漏れをコンパイラに見つけさせる
3. データの不変 —— 遷移が「新しい値を返す」で成立するため

これらが備わった言語の例:

F#, Scala, TypeScript, Kotlin, Java, C#, Rust

# 最初の一步

---

## `optional` が 3 つ並んだ型を探す

- そこはたいてい、直和型で書き直せる場所
- データについての業務ルールを型に移してみる

## `if` で書かれた入力値チェックを探す

- 入力値の制約を引数の型にしてみる
- ロジックに渡す前に検証済みであることを型で保証する
- 検証済みの入力は書き換えられないので、新しい検証済みの値を返す

Thank You! (本物)