型情報と相性のいいバリデーションライブラリ - Zod, Croma, Pydantic

本日の内容

WebAPIのリクエストボディのバリデーション等に利用するバリデーションライブラリに関して、型情報を上手く利用できるライブラリを複数の言語(TypeScript / Elixir / Python)に渡って見ていく。

昔のバリデーション

```
const validate = require("validate.js");

const constraints = {
  fieldA: { // required string field
    presence: true,
    type: "string",
  },
  fieldB: { // optional integer field
    type: "integer",
  },
};

const request = { fieldA: "a", fieldB: 10 };
const invalid = validate(request, constraints);
if (invalid) {
  throw new Error("invalid");
}
// この後はrequestはチェックされた値として扱う
```

現代で同じようなライブラリを使うと

- TypeScriptにより型を定義している場合、型情報とバリデーションの情報が重複してしまう。
- バリデーション後は正しい型として扱うためにキャストが必要。キャストを行うとバリデーションと型の記載を間違えてもチェックされない。

```
// 省略...
const request: any = { fieldA: "a", fieldB: 10 };
const invalid: any = validate(request, constraints);
if (invalid) {
  throw new Error("invalid");
}

type Request = {
  fieldA: string;   // 型情報とバリデーションの情報が重複
  fieldB?: number;
};

const validRequest: Request = request as Request; // キャストが必要。
```

Zod

- バリデーションのコードから型情報を生成するライブラリ。

```
import { z } from "zod";

const UserSchema = z.object({
  fieldA: z.string(),
  fieldB: z.number().int().optional(),
});

const user = UserSchema.parse({ fieldA: "a" });
// この時点で、userの型は以下のようになる
// {
//     fieldA: string;
//     fieldB?: number | undefined;
// }
user.fieldC = "c"; // `fieldC`は存在しないので「コンパイル時」にエラーになる
```

Zodの特徴

- 型情報をバリデーションのコードから「コンパイル時」に生成するため、型情報とバリデーションの情報が重複しない。
- バリデーション後の値は型情報がつくのでキャストが不要。結果の値に対して型にないオペレーションを書くとコンパイルエラーになる。
- `z.union` やobjectの `.pick` など、TypeScriptの型操作で使うような操作をサポートしている。

Zodの実現方法

- TypeScriptには非常に強力なGenericsがある。
- Genericsはコンパイル時に動作する。
- Genericsを使えば、引数の型情報を利用して、返り値の型情報を生成することが出来る。
- 原理的には理解できるが、zodのtype定義を見ればわかるとおり、かなり高度な型定義をしている。types.ts、型操作用のutil.ts。

```
const fieldB = z
    .number()   // ZodNumber型
    .optional()  // ZodOptional<ZodNumber>型

const UserSchema = z.object({
  fieldB: fieldB,
}); // ZodObject<{ fieldB: ZodOptional<ZodNumber> }>型

// parse()の関数定義として上記のGenerics型から型情報を返すGenericsが定義されている。
const user = UserSchema.parse({ fieldA: "a" });
```

変換

- バリデーション処理で文字列型を数値型に変換するなどをしたいことも多いが、Zodは型変換もサポートしている。
- リテラルの変換なら`z.coerce`が利用でき、カスタムの変換がしたいならば`z.preprocess`を利用する。

```
import { z } from "zod";

const UserSchema = z.object({
  fieldA: z.string(),
  fieldB: z.coerce.number().int().optional(), // coerceを使うと型が変換される
});

// fieldBはstring型だが、coerceによりnumber型に変換されるのでエラーにならない
const user = UserSchema.parse({ fieldA: "a", fieldB: "1" });
console.log(user);
```

Zodを利用したライブラリ

https://github.com/colinhacks/zod#ecosystem

- WebAPIでのバリデーション
- フォームでのバリデーション
- CLIの引数のバリデーション
  - ZodOpts
- 環境変数のバリデーション
- 他
  - LangChain JSのStructured Output Parserとしても利用されている。
    - https://js.langchain.com/docs/modules/model_io/output_parsers/#structured-output-parser-with-zod-schema

Croma

- Elixirのマクロユーティリティーライブラリ。
- `Croma.Struct`を使ってバリデーションのコードから型情報を生成できる。
  - https://github.com/skirino/croma#cromastruct
- 型情報の生成をマクロで実現している。

Cromaによるバリデーション

```elixir
defmodule User do
  use Croma.Struct, fields: [
    field_a: Croma.String,
    field_b: Croma.TypeGen.nilable(Croma.Integer),
  ]
end

case User.new(%{field_a: "a", field_b: 10}) do
  {:ok, user} ->
    IO.inspect(user.field_c) # `field_c`は存在しないので、dialyzerの実行時にエラーになる
  {:error, e} ->
    IO.puts("validation error: #{inspect(e)}")
end
```

Pydantic

- Pythonの型ヒントを利用したバリデーションライブラリ。
- ZodやCromaとは異なり、バリデーションのコードから型を作るのではなく、型ヒントを実行時のバリデーションに利用する。

Pydanticによるバリデーション

```python
from pydantic import BaseModel, ValidationError
from typing import Any


class User(BaseModel):
    field_a: str
    field_b: int | None = None


external_data: Any = {"field_a": "foo", "field_b": 1}

try:
    user = User(**external_data)

    # 型ヒントでUserのフィールドが定義されているため、
    # 下記のコードでは`field_c`が存在しないので、mypyでのチェック時にエラーが発生する。
    # error: "User" has no attribute "field_c"; maybe "field_a" or "field_b"?  [attr-defined]
    print(user.field_c)

except ValidationError as e:
    # 型ヒントに合わないデータが渡された場合にはValidationErrorが発生する。
    print(e)
```

Pydanticの実現方法

- Pythonの型ヒントを`__annotations__`を使って実行時に取得することが出来るので、Pydanticはそれを利用している。

```
class User:
    field_a: str
    field_b: int | None = None


print(User.__annotations__) # {'field_a': <class 'str'>, 'field_b': int | None}
```

Pydanticによる環境変数のバリデーション・変換

- 環境変数でプログラムの挙動を変更できるようにしたときに、`os.getenv('use_great_option')`のように呼び出すと、文字列からどのようにboolを判定するのかをコードで記述する必要がある。
- Pydanticの`BaseSettings`を利用すると、環境変数の値を型ヒントに応じて変換したり不正な値の場合にエラーを起こしてくれる。

```
from pydantic import ValidationError
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    use_great_option: bool = False

try:
    settings = Settings()
    print(settings.use_great_option)
    # use_great_option="true" -> true
except ValidationError as e:
    # use_great_option="aaa" -> ValidationError
    print(e)
```

Pydanticを利用したライブラリ

https://github.com/Kludex/awesome-pydantic

Pydanticは2系と1系が存在し、1系しかサポートしていないライブラリもあるので注意。

- FastAPI
  - pydanticを使ってリクエストボディのクラス定義を行うことで、リクエストボディのバリデーションを非常に簡潔に記載することが出来る。
  - https://fastapi.tiangolo.com/tutorial/body/#declare-it-as-a-parameter

まとめ

- Zod / Croma / Pydanticのようなライブラリを利用すると、型情報と重複したバリデーションのコードを書く必要がない。
- バリデーション完了後は型が付くので、型による安全性を保ちながら開発を進めることが出来る。
- 型とバリデーションをまとめてStruct / Classに定義して再利用可能なビルディングブロックとして組み合わせていくことで、明瞭なコードを書くことが出来る。