YAVIの紹介

Toshiaki Maki (@making), https://ik.am
JSUG勉強会 2021-07-02

質問はTwitter(#jsug)にお願いします 🙇‍♂️

YAVI?

Yet ***A***nother ***V***al***i***dation (ヤヴァイ)

要はBean ValidationではないJavaのValidationライブラリ
ラムダベースでタイプセーフ

https://github.com/making/yavi
https://yavi.ik.am (Reference Document)

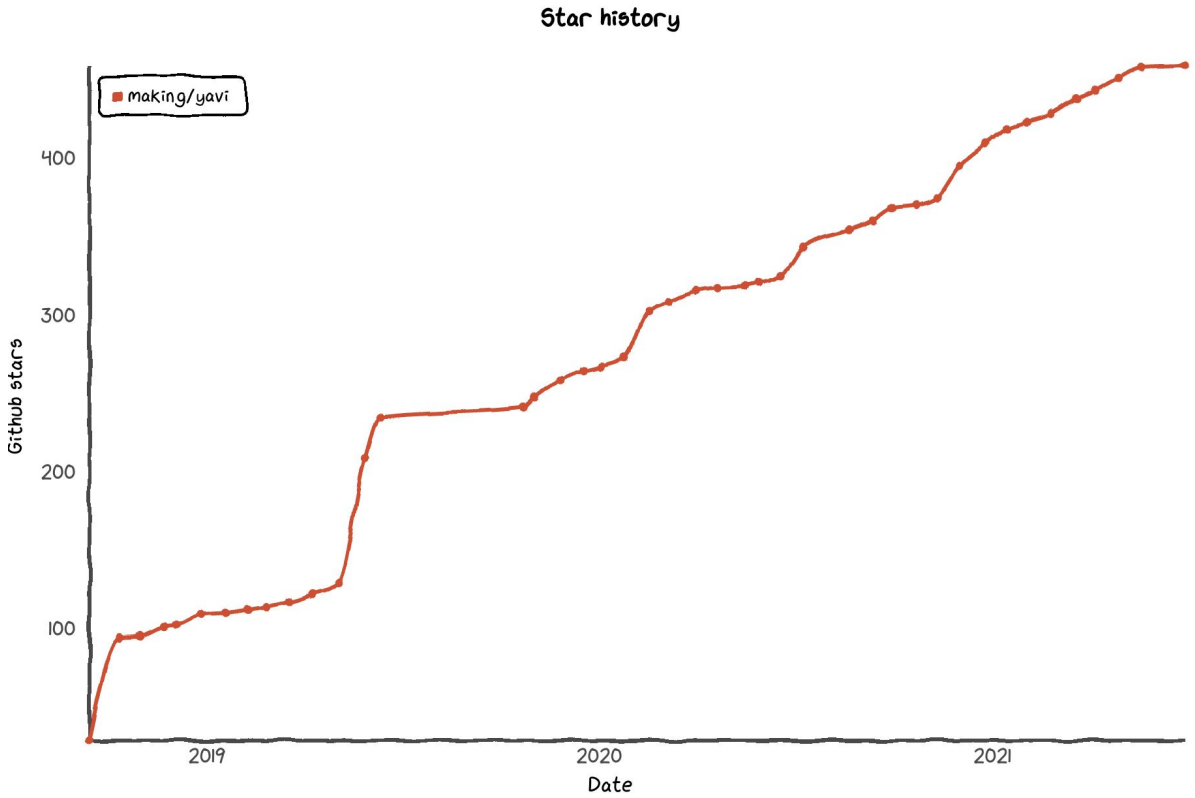
なぜYAVIを作った？

当時 … Functionを使ったWebフレームワーク(Spring WebFlux.fnなど)にBean Validationは使いづらいので、フィットするValidationライブラリが欲しかった

今 … 最高のValidationライブラリを作りたい

意外と⭐がついている

https://github.com/making/yavi スターください！



https://star-history.t9t.io/#making/yavi

YAVIのコンセプト (a.k.a YAVI 4原則)

1. リフレクションは使用しない
2. (Runtimeで)アノテーションは使用しない
3. Java Beansを前提にしない
4. 依存ライブラリを持たない

YAVIのウリ

- タイプセーフな制約 (対応していない型には制約をつけられない)
- Fluentで直感的なAPI
- どんなオブジェクトに対してもValidation可能
  - Java Beans, Records, Protocol Buffers, Immutables, その他何でもござれ
- たくさんの強力な組み込み制約ルール
- 簡単に実装できるカスタム制約
- グループに対するValidationや条件付きValidationのような高度なValidationが可能
- オブジェクトを生成する前に引数に対してValidationが可能
- 関数型プログラミングの概念を活用して、Validation結果の合成やValidatorの合成が可能

YAVIの使い方

YAVIの使い方

```
<dependency>
    <groupId>am.ik.yavi</groupId>
    <artifactId>yavi</artifactId>
    <version>0.8.2</version>
</dependency>
```

Validatorの定義

```
Validator<User> validator = ValidatorBuilder.<User> of() // or ValidatorBuilder.of(User.class)
    .constraint(User::getName, "name", c -> c.notBlank().lessThanOrEqual(20))
    .constraint(User::getEmail, "email", c -> c.notBlank().greaterThanOrEqual(5).lessThanOrEqual(50).email())
    .constraint(User::getAge, "age", c -> c.notNull().greaterThanOrEqual(0).lessThanOrEqual(200))
    .build();

ConstraintViolations violations = validator.validate(user);
boolean isValid = violations.isValid();
violations.forEach(x -> System.out.println(x.message()));
```

Validatorの定義 (型明示版)

```
Validator<User> validator = ValidatorBuilder.<User> of() // or ValidatorBuilder.of(User.class)
    ._string(User::getName, "name", c -> c.notBlank().lessThanOrEqual(20))
    ._string(User::getEmail, "email", c -> c.notBlank().greaterThanOrEqual(5).lessThanOrEqual(50).email())
    ._integer(User::getAge, "age", c -> c.notNull().greaterThanOrEqual(0).lessThanOrEqual(200))
    .build();

ConstraintViolations v   tions = validator.validate(user);
boolean isValid = violatio   lid();
violations.forEach(x -> System.   tln(x.message()));
```

オーバーロードのconstraintメソッドだと、ラムダ式によっては型推論がうまくできない場合があるので、制約対象の型を使用することもできる。

Recordsを使う場合

```
public record User(String name, String email, int age) {
}

Validator<User> validator = ValidatorBuilder.<User> of()
    .constraint(User::name, "name", c -> c.notBlank().lessThanOrEqual(20))
    .constraint(User::email, "email", c -> c.notBlank().greaterThanOrEqual(5).lessThanOrEqual(50).email())
    .constraint(User::age, "age", c -> c.notNull().greaterThanOrEqual(0).lessThanOrEqual(200))
    .build();
```

フィールドを公開したくない場合

```
public class User {
    private final String name;

    public final Validator<User> validator = ValidatorBuilder.<User> of()
        ._string(x -> x.name, "name", c -> c.notNull().lessThanOrEqual(20))
        .build();

    public User(String name) {
        this.name = name;
    }
    // Omits others
}
```

Nameをgetterで公開したくないので、Private Fieldに対して制約を課す

組み込み制約ルール

https://yavi.ik.am/#built-in-constraints

- notNull
- notEmpty
- notBlank
- fixedSize
- greaterThan
- greaterThanOrEqual
- lessThan
- lessThanOrEqual
- contains
- pattern
- email
- password
- Ipv4
- Ipv6
- url
- luhn
- codePoints
- etc...

Nestした制約 (委譲)

```
public class Address {
    private Country country;
    private City city;
    // Omits other fields and getters
}

Validator<Country> countryValidator = ValidatorBuilder.<Country> of()
    .constraint(Country::getName, "name", c -> c.notBlank().lessThanOrEqual(20))
    .build();
Validator<City> cityValidator = ValidatorBuilder.<City> of()
    .constraint(City::getName, "name", c -> c.notBlank().lessThanOrEqual(100))
    .build();

Validator<Address> validator = ValidatorBuilder.<Address> of()
    .nest(Address::getCountry, "country", countryValidator)
    .nest(Address::getCity, "city", cityValidator)
    .build();

// or country, cityがnullableな場合
Validator<Address> validator = ValidatorBuilder.<Address> of()
    .nestIfPresent(Address::getCountry, "country", countryValidator)
    .nestIfPresent(Address::getCity, "city", cityValidator)
    .build();
```

Nestした制約 (インライン)

```
Validator<Address> validator = ValidatorBuilder.<Address> of()
    .nest(Address::getCountry, "country",
        b -> b.constraint(Country::getName, "name", c -> c.notBlank().lessThanOrEqual(20)))
    .nest(Address::getCity, "city",
        b -> b.constraint(City::getName, "name", c -> c.notBlank().lessThanOrEqual(100)))
    .build();
```

条件付き制約

```
Validator<User> validator = ValidatorBuilder.<User> of()
    .constraintOnCondition((user, constraintGroup) -> !user.getName().isEmpty(),
        b -> b.constraint(User::getEmail, "email", c -> c.email().notEmpty()))
    .build();
```

制約を適用する条件

“getName()が空でない場合”の条件下でのみ適用される制約

制約のグループ

```
enum Group implements ConstraintGroup {
    CREATE, UPDATE, DELETE
}

Validator<User> validator = ValidatorBuilder.<User> of()
    .constraintOnCondition(Group.CREATE.toCondition(), b -> b.constraint(User::getId, "id", c -> c.isNull()))
    .constraintOnCondition(Group.UPDATE.toCondition(), b -> b.constraint(User::getId, "id", c -> c.notNull()))
    .build();
```

```
ConstraintViolations violations = validator.validate(user, Group.CREATE);
```

CREATEグループに対して
はidはisNullを

UPDATEグループに対して
はidはnotNullを

CREATEグループに対して
validate

制約のグループ (短縮版)

```
enum Group implements ConstraintGroup {
    CREATE, UPDATE, DELETE
}

Validator<User> validator = ValidatorBuilder.<User> of()
    .constraintOnGroup(Group.CREATE, b -> b.constraint(User::getId, "id", c -> c.isNull()))
    .constraintOnGroup(Group.UPDATE, b -> b.constraint(User::getId, "id", c -> c.notNull()))
    .build();
```

```
ConstraintViolations violations = validator.validate(user, Group.CREATE);
```

CREATEグループに対してはidはisNullを

UPDATEグループに対してはidはnotNullを

CREATEグループに対してvalidate

カスタム制約 (定義)

```
public class IsbnConstraint implements CustomConstraint<String> {

    @Override
    public boolean test(String s) {
        // Delegate processing to another method
        return ISBNValidator.isISBN13(s);
    }

    @Override
    public String messageKey() {
        return "string.isbn13";
    }

    @Override
    public String defaultMessageFormat() {
        return "\"{0}\" must be ISBN13 format";
    }
}
```

カスタム制約 (使用)

```
IsbnConstraint isbn = new IsbnConstraint();
Validator<Book> book = ValidatorBuilder.<Book> of()
    .constraint(Book::getTitle, "title", c -> c.notBlank().lessThanOrEqual(64))
    .constraint(Book::getIsbn, "isbn", c -> c.notBlank().predicate(isbn))
    .build();
```

カスタム制約 (インライン版)

```
Validator<Book> book = ValidatorBuilder.<Book> of()
    .constraint(Book::getTitle, "title", c -> c.notBlank().lessThanOrEqual(64))
    .constraint(Book::getIsbn, "isbn", c -> c.notBlank()
        .predicate(s -> ISBNValidator.isISBN13(s),
                   "string.isbn13", "\"{0}\" must be ISBN13 format"))
    .build();
```

相関チェック (定義)

```
public class RangeConstraint implements CustomConstraint<Range> {
    @Override
    public String defaultMessageFormat() {
        return "\"to\" must be greater than \"from\"";
    }

    @Override
    public String messageKey() {
        return "to.isGreaterThanFrom";
    }

    @Override
    public boolean test(Range range) {
        return range.getTo() > range.getFrom();
    }
}
```

相関チェック (使用)

```
RangeConstraint range = new RangeConstraint();
Validator<Range> validator = ValidatorBuilder.<Range> of()
        .constraintOnTarget(range, "to")
        .build();
```

相関チェック (インライン)

```
Validator<Range> validator = ValidatorBuilder.<Range> of()
    .constraint(range::getFrom, "from", c -> c.greaterThan(0))
    .constraint(range::getTo, "to", c -> c.greaterThan(0))
    .constraintOnTarget(range -> range.getTo() > range.getFrom(),
                        "to", "to.isGreaterThanFrom", "\"to\" must be greater than \"from\"")
    .build();
```

Kotlin版

```
val validator: Validator<User> = ValidatorBuilder.of<User>()
    .konstraint(User::name) {
        notNull()
            .lessThanOrEqual(20)
    }
    .konstraint(User::email) {
        notNull()
            .greaterThanOrEqual(5)
            .lessThanOrEqual(50)
            .email()
    }
    .konstraint(User::age) {
        notNull()
            .greaterThanOrEqual(0)
            .lessThanOrEqual(200)
    }
    .build()
```

Kotlin DSLの提供を検討中
https://github.com/making/yavi/pull/152

Validation結果の合成

Applicative Functorを使ったValidation

関数型プログラミングのApplicative Functorをサポート。

合成したValidation結果をショートカットすることなく集約可能。

Validation結果はValidated型で表現。

```
Validator<User> validator = ValidatorBuilder.<User> of()
    .constraint(User::getName, "name", c -> c.notNull().lessThanOrEqual(20))
    // ...
    .build();

ApplicativeValidator<User> applicativeValidator = validator.applicative();
Validated<User> userValidated = applicativeValidator.validate(user);

if (userValidated.isValid()) {
    User user = userValidated.value();
} else {
    ConstraintViolations violations = userValidated.errors();
}
// or
User user = userValidated.orElseThrow(violations ->
                                          new ConstraintViolationsException(violations));
// or
HttpStatus status = userValidated.fold(violations -> BAD_REQUEST, user -> OK);
```

Validatedオブジェクトの合成

```
Validator<Email> emailValidator = ValidatorBuilder.<Email> of()
    .constraint(Email::value, "email", c -> c.notBlank().email())
    .build();
Validator<PhoneNumber> phoneNumberValidator = ValidatorBuilder.<PhoneNumber> of()
    .constraint(PhoneNumber::value, "phoneNumber", c -> c.notBlank().pattern("[0-9\\-]+"))
    .build();

Validated<Email> emailValidated = emailValidator.applicative()
    .validate(email);
Validated<PhoneNumber> phoneNumberValidated = phoneNumberValidator.applicative()
    .validate(phoneNumber);

Validated<ContactInfo> contactInfoValidated = emailValidated.combine(phoneNumberValidated)
    .apply((em, ph) -> new ContactInfo(em, ph));
// or
Validated<ContactInfo> contactInfoValidated = Validations
    .combine(emailValidated, phoneNumberValidated)
    .apply((em, ph) -> new ContactInfo(em, ph));
```

EmailとPhoneNumberのValidationがpassしてContactInfoが作成される

Validatedオブジェクトの合成

```
Validated<Email> emailValidated = emailValidator.applicative()
    .validate(new Email(" "));
Validated<PhoneNumber> phoneNumberValidated = phoneNumberValidator.applicative()
    .validate(new PhoneNumber("a"));


Validated<ContactInfo> contactInfoValidated = Validations
    .combine(emailValidated, phoneNumberValidated)
    .apply((em, ph) -> new ContactInfo(em, ph));

ConstraintViolations violations = contactInfoValidated.errors();
```

```
* "email" must not be blank
* "email" must be a valid email address
* "phoneNumber" must match [0-9\-]+
エラーが集約される
```

引数に対するValidation

普通のValidation

```
Validator<User> validator = ValidatorBuilder.<User> of() // or ValidatorBuilder.of(User.class)
    .constraint(User::getName, "name", c -> c.notBlank().lessThanOrEqual(20))
    .constraint(User::getEmail, "email", c -> c.notBlank().greaterThanOrEqual(5).lessThanOrEqual(50).email())
    .constraint(User::getAge, "age", c -> c.notNull().greaterThanOrEqual(0).lessThanOrEqual(200))
    .build();

ConstraintViolations violations = validator.validate(new User("  ", "aa ", -1));
```

不正なオブジェクトの作成を許容した上での
Validation実行

Arguments Validator

引数に対してValidationを実行してからオブジェクトを作成する。

```
public class User {
    public User(String name, String email, Integer age) {
        // ...
    }

  static Arguments3Validator<String, String, Integer, User> validator = ArgumentsValidatorBuilder
        .of(User::new)
        .builder(b -> b
            ._string(Arguments1::arg1, "name", c -> c.notBlank().lessThanOrEqual(100))
            ._string(Arguments2::arg2, "email", c -> c.notBlank().lessThanOrEqual(100).email())
            ._integer(Arguments3::arg3, "age", c -> c.greaterThanOrEqual(0).lessThan(200)))
        .build();
}
```

```
Validated<User> userValidated = User.validator.validate("Jone Doe", "jdoe@example.com", 30);
```

引数は16までサポート

User::newの引数に対するValidationが成功したらUser::newを実行

Arguments Validator

引数に対してValidationを実行してからオブジェクトを作成する。

```
public class User {
    public User(String name, String email, Integer age) {
        // ...
    }

  static Arguments3Validator<String, String, Integer, User> validator = ArgumentsValidatorBuilder
        .of(User::new)
        .builder(b -> b
            ._string(Arguments1::arg1, "name", c -> c.notBlank().lessThanOrEqual(100))
            ._string(Arguments2::arg2, "email", c -> c.notBlank().lessThanOrEqual(100).email())
            ._integer(Arguments3::arg3, "age", c -> c.greaterThanOrEqual(0).lessThan(200)))
        .build();
}
```

ちょっと定義が大変？

```
Validated<User> userValidated = User.validator.validate("Jone Doe", "jdoe@example.com", 30);
```

引数は16までサポート

User::newの引数に対するValidationが成功したらUser::newを実行

引数が1つの場合の定義のショートカット

```
StringValidator<String> nameValidator = StringValidatorBuilder
    .of("name", c -> c.notBlank().lessThanOrEqual(100))
    .build();  // -> extends Arguments1Validator<String, String>

StringValidator<String> emailValidator = StringValidatorBuilder
    .of("email", c -> c.notBlank().lessThanOrEqual(100).email())
    .build();  // -> extends Arguments1Validator<String, String>

IntegerValidator<Integer> ageValidator = IntegerValidatorBuilder
    .of("age", c -> c.greaterThanOrEqual(0).lessThan(200))
    .build();  // -> extends Arguments1Validator<Integer, Integer>

Validated<String> nameValidated = nameValidator.validate("Jone Doe");
Validated<String> emailValidated = nameValidator.validate("jdoe@example.com");
Validated<Integer> ageValidated = nameValidator.validate(30);
```

Stringを検証してStringを返すValidator

Integerを検証してIntegerを返すValidator

引数が1つの場合の定義のショートカット

```
StringValidator<Name> nameValidator = StringValidatorBuilder
    .of("name", c -> c.notBlank().lessThanOrEqual(100))
    .build()
    .andThen(name -> new Name(name)); // -> extends Arguments1Validator<String, Name>

StringValidator<Email> emailValidator = StringValidatorBuilder
    .of("email", c -> c.notBlank().lessThanOrEqual(100).email())
    .build()
    .andThen(email -> new Email(email)); // -> extends Arguments1Validator<String, Email>

IntegerValidator<Age> ageValidator = IntegerValidatorBuilder
    .of("age", c -> c.greaterThanOrEqual(0).lessThan(200))
    .build()
    .andThen(age -> new Age(age)); // -> extends Arguments1Validator<Integer, Age>

Validated<Name> nameValidated = nameValidator.validate("Jone Doe");
Validated<Email> emailValidated = nameValidator.validate("jdoe@example.com");
Validated<Age> ageValidated = nameValidator.validate(30);

Validated<List<Email>> emailsValidated = emailValidator.liftList()
    .validate(List.of("foo@example.com", "bar@example.com"));
```

Stringを検証してNameを返すValidator

Integerを検証してAgeを返すValidator

List<String>に対するValidatorに変換

Validatorの合成

Arguments1Validatorの合成

ArgumentsNValidatorはN個のArguments1Validatorに分割して結合可能

```
public class User {
    public User(String name, String email, Integer age) {
        // ...
    }
}

StringValidator<String> nameValidator = /* 前の例と同じ */;
StringValidator<String> emailValidator = /* 前の例と同じ */;
IntegerValidator<Integer> ageValidator = /* 前の例と同じ */;

Arguments3Validator<String, String, Integer, User> userValidator = ArgumentsValidators
    .split(nameValidator, emailValidator, ageValidator)
    .apply(User::new);
// or
Arguments3Validator<String, String, Integer, User> userValidator = nameValidator
    .split(emailValidator)
    .split(ageValidator)
    .apply(User::new);

Validated<User> userValidated = User.validator.validate("Jone Doe", "jdoe@example.com", 30);
```

User::newを3つのArguments1Validatorにsplit

Validatorの合成 (Value Object)

```
public class User {
    public User(Name name, Email email, Age age) {
        // ...
    }
}

StringValidator<Name> nameValidator = /* see examples above */;
StringValidator<Email> emailValidator = /* see examples above */;
IntegerValidator<Age> ageValidator = /* see examples above */;

Arguments3Validator<String, String, Integer, User> UserValidator = ArgumentsValidators
    .split(nameValidator, emailValidator, ageValidator)
    .apply(User::new);
// or
Arguments3Validator<String, String, Integer, User> UserValidator = nameValidator
    .split(emailValidator)
    .split(ageValidator)
    .apply(User::new);

Validated<User> userValidated = User.validator.validate("Jone Doe", "jdoe@example.com", 30);
```

ValueObject単位でValidatorを持ち、合成することで再利用性が向上

ValueValidator<S, T>

Sクラスのオブジェクト(Source)を検証してTクラスのオブジェクト(Target)を返すValidatorインタフェース。
`ArgumentsNValidator`は`ValueValidator`へsplit可能。

```
public interface Arguments1Validator<S, T> extends ValueValidator<S, T> {
        // ...
}

public interface ApplicativeValidator<T> extends ValueValidator<T, T> {
        // ...
}
```

ApplicativeValidatorの合成

ArgumentsNValidatorをApplicativeValidatorにsplitすることもできる

```
Validator<Email> emailValidator = /* 前の例と同じ */;
Validator<PhoneNumber> phoneNumberValidator = /* 前の例と同じ */;

Arguments2Validator<Email, PhoneNumber, ContactInfo> contactInfoValidator = ArgumentsValidators
    .split(emailValidator.applicative(), phoneNumberValidator.applicative())
    .apply(ContactInfo::new);

Validated<ContactInfo> contactInfoValidated = contactInfoValidator
    .validate(new Email("yavi@example.com"), new PhoneNumber("090-123-4567"));
```

前の例はValidation結果を合成したが、今回の例はValidatorを合成した。

Sourceオブジェクトの変換

ValueValidator<S,T>をcomposeメソッドでValueValidator<S2,T>に変換し、ソースを変更できる

```
Argument1Validator<HttpServletRequest, Name> requestNameValidator = nameValidator
    .compose(req -> req.getParameter("name"));
Argument1Validator<HttpServletRequest, Email> requestEmailValidator = emailValidator
    .compose(req -> req.getParameter("email"));
Argument1Validator<HttpServletRequest, Age> requestAgeValidator = ageValidator
    .compose(req -> Integer.valueOf(req.getParameter("age")));

HttpServletRequest request = ...;
Validated<Name> nameValidated = requestNameValidator.validate(request);
Validated<Email> emailValidated = requestEmailValidator.validate(request);
Validated<Age> ageValidated = requestAgeValidator.validate(request);
```

Servletアプリの場合、Validation対象のStringやIntegerは通常HttpServletRequestから取得するので、ソースをHttpServletRequestに変更

Sourceオブジェクトの変換

HttpServletRequestを検証して、Name、Email, Ageを返すValidatorを結合し
Userを返すValidatorを合成

```
Arguments1Validator<HttpServletRequest, User> requestUserValidator = ArgumentsValidators
    .combine(requestNameValidator, requestEmailValidator, requestAgeValidator)
    .apply(User::new);

HttpServletRequest request = ...;
Validated<User> userValidated = requestUserValidator.validate(request);
```

HttpServletRequestに対するValidatorを合成することで、
HttpServletRequestを直接Validatorに渡して、検証済みの対象
オブジェクトを取得できる。

Enterprise-Readyな文字長チェック

見た目の長さとString#lengthの結果が異なる文字種

https://yavi.ik.am/#advanced-character-length-check

- サロゲートペア
- 結合文字
- 異字体セレクタ
- Emoji

YAVIは見た目の長さでの文字数チェックをサポート

サロゲートペア

```
System.out.println("𠮷野屋".length()); // 4 (\uD842\uDFB7野屋)
System.out.println("𠮷野屋".codePointCount(0, "𠮷野屋".length())); // 3
```

```
Validator<Message> validator = ValidatorBuilder.<Message> of()
    .constraint(Message::getText, "text", c -> c.lessThanOrEqual(3))
    .build();

ConstraintViolations = validator.validate(new Message("𠮷野屋")); // ✅ Valid
```

YAVIはString#lengthではなくString#codePointCountで文字長を取得

結合文字

```
System.out.println("モジ".length()); // 3 (モシ\u3099)
System.out.println("モジ".codePointCount(0, "モジ".length())); // 3
```

```
Validator<Message> validator = ValidatorBuilder.<Message> of()
    .constraint(Message::getText, "text", c -> c.lessThanOrEqual(2))
    .build();

ConstraintViolations = validator.validate(new Message("モジ")); // ✅ Valid
```

デフォルトでjava.text.Normalizer.Form#NFCを使って文字列を正規化した後にlengthを取得している

異字体セレクタ

```
System.out.println("𠮟".length()); // 4 (\uD842\uDF9F\uDB40\uDD00)
System.out.println("𠮟".codePointCount(0, "𠮟".length())); // 2
```

```
Validator<Message> validator = ValidatorBuilder.<Message> of()
    .constraint(Message::getText, "text", c -> c.lessThanOrEqual(1))
    .build();

ConstraintViolations = validator.validate(new Message("𠮟")); // ❌ Invalid
// The size of "text" must be less than or equal to 1. The given size is 2
```

異字体セレクタのコードポイントを正規表現で無視できるが、利用頻度に対してパフォーマンス低下が無視できないので、デフォルトではサポートしない。

異字体セレクタ

```
System.out.println("𠮟".length()); // 4 (\uD842\uDF9F\uDB40\uDD00)
System.out.println("𠮟".codePointCount(0, "𠮟".length())); // 2
```

```
Validator<Message> validator = ValidatorBuilder.<Message> of()
    .constraint(Message::getText, "text", c -> c
          .variant(opts -> opts.ivs(IdeographicVariationSequence.IGNORE))
          .lessThanOrEqual(1))
    .build();

ConstraintViolations = validator.validate(new Message("𠮟")); // ✅ Valid
```

IVS(0xE0100-0xE01EF)の範囲を明示的に無視する設定をすることでサポート。
SVS (Standardized Variation Sequence)、FVS (Mongolian Free Variation Selector)も同様。

Emoji

```
System.out.println("❤️".length()); // 2
System.out.println("👸".length()); // 4
System.out.println("👩‍👦".length()); // 5
System.out.println("👨‍👨‍👧‍👧".length()); // 12
System.out.println("🏴‍".length()); // 14 (WTH!)
```

Emojiの範囲を手当たり次第に正規表現で置換。
パフォーマンスが低下するのでデフォルトでは無効。

```
Validator<Message> validator = ValidatorBuilder.<Message> of()
    .constraint(Message::getText, "text", c -> c.emoji().lessThanOrEqual(1))
    .build();
ConstraintViolations = validator.validate(new Message("❤️")); // ✅ Valid
ConstraintViolations = validator.validate(new Message("👸")); // ✅ Valid
ConstraintViolations = validator.validate(new Message("👩‍👦")); // ✅ Valid
ConstraintViolations = validator.validate(new Message("👨‍👨‍👧‍👧")); // ✅ Valid
ConstraintViolations = validator.validate(new Message("🏴‍")); // ✅ Valid
```

バイト配列長でのチェック

見た目の文字長とバイト長が違いすぎるので、見た目上のの長さチェックに加えてバイト長のサイズのチェックを実施することを推奨

見た目のサイズは3文字以下、バイト長は16以下

```
Validator<Message> validator = ValidatorBuilder.<Message> of()
    .constraint(Message::getText, "text", c -> c.emoji().lessThanOrEqual(3)
            .asByteArray().lessThanOrEqual(16))
    .build();

ConstraintViolations = validator.validate(new Message("I❤️☕")); // ✅ Valid
ConstraintViolations = validator.validate(new Message("❤️❤️❤️")); // ❌ Invalid
// The byte size of "text" must be less than or equal to 16. The given size is 24
```

Spring Frameworkとの連携

Spring MVCで使う (Viewがある場合)

YAVIは依存ライブラリを持たないが、メソッド参照を活用してSpringにはマジカルフィットする

```
final Validator<UserForm> validator = ...;

@PostMapping(path = "users")
public String createUser(Model model, UserForm userForm, BindingResult bindingResult) {
    ConstraintViolations violations = validator.validate(userForm);
    if (!violations.isValid()) {
        violations.apply(bindingResult::rejectValue);
        return "userForm";
    }
    // ...
    return "redirect:/";
}
```

ConstraintViolations::applyの引数として
BindingResult::rejectValueがうまくフィットする

Spring MVCで使う (Viewがある場合)

より関数型っぽく書きたい場合

```
final Validator<UserForm> validator = ...;

@PostMapping(path = "users")
public String createUser(Model model, UserForm userForm, BindingResult bindingResult) {
    return validator.applicative()
        .validate(userForm)
        .fold(violations -> {
            ConstraintViolations.of(violations).apply(bindingResult::rejectValue);
            return "userForm";
        }, form -> {
            // ...
            return "redirect:/";
        });
}
```

https://github.com/making/demo-spring-mvc-yavi

Spring MVCで使う (REST)

```
final Validator<UserCreateRequest> validator = ...;

@PostMapping(path = "users")
public ResponseEntity<?> createUser(@RequestBody UserCreateRequest request) {
    ConstraintViolations violations = validator.validate(request);
    if (violations.isValid()) {
        User created = userService.create(request.toUser());
        return ResponseEntity.ok(created);
    } else {
        return ResponseEntity.badRequest().body(violations.details());
    }
}
```

Jacksonでシリアライズ可能な`ViolationDetail`に変換。
GraalVMのnative imageもサポート。

Spring MVCで使う (REST)

より関数型っぽく書きたい場合

```
final Validator<UserCreateRequest> validator = ...;

@PostMapping(path = "users")
public ResponseEntity<?> createUser(@RequestBody UserCreateRequest request) {
    return validator.applicative()
        .validate(request)
        .map(req -> userService.create(req.toUser()))
        .mapError(ConstraintViolation::detail)
        .fold(details -> ResponseEntity.badRequest().body(details),
            created -> ResponseEntity.ok(created));
}
```

Spring WebFlux.fnで使う

元々はこれがやりたかった。

```
final Validator<UserCreateRequest> validator = ...;

public RouterFunction<ServerResponse> routes() {
    return RouterFunctions.route()
        .POST("/users", request -> request.bodyToMono(UserCreateRequest.class)
            .flatMap(body -> validator.applicative()
                .validate(body)
                .map(req -> userService.create(req.toUser()))
                .mapError(ConstraintViolation::detail)
                .fold(details -> ServerResponse.badRequest().bodyValue(details),
                    created -> ServerResponse.ok().bodyValue(created))))
        .build();
}
```

SpringのMessageSourceを使ってエラーメッセージ作成

これも依存ライブラリなしでマジカルフィット

```
@RestController
public class OrderController {
    private final Validator<CartItem> validator;

    public OrderController(MessageSource messageSource) {
        MessageFormatter messageFormatter =
                                        new MessageSourceMessageFormatter(messageSource::getMessage);
        this.validator = ValidatorBuilder.<CartItem> of()
            .constraints(...)
            .messageFormatter(messageFormatter)
            .build();
    }
}
```

Thanks!

https://github.com/making/yavi

スターお願いします