

2015-05-11

## Java8のStream APIの使い方（中間操作編① - filter, map）

Stream API の中間操作について一番使う機会の多い filter と map について使用方法などをまとめました。

### filter : 抽出

引数：Predicate<T> / 戻り値：Stream<T>

#### ☑ 関数型インターフェース

filterメソッドはPredicate<T>を引数に取り、Streamのオブジェクトから条件に一致する（Predicateがtrueを返す）オブジェクトを抽出します。Predicateは判定を行うための関数型インターフェースです。実装が必要なメソッドは `boolean test(T t)` で引数を1つ受け取り、booleanを返します。

#### ☑ filterの使い方

IntegerのStreamから数値が200以上のデータを抽出するサンプルです。

```
List<Integer> prices = Arrays.asList(100, 200, 300, 400, 500);
// ラムダ式
prices.stream().filter(pri -> pri > 200).forEach(System.out::println);

// 匿名クラス
prices.stream().filter(new Predicate<Integer>() {
    @Override
    public boolean test(Integer t) {
        return t > 200;
    }
}).forEach(System.out::println);
```

#### ☑ defaultメソッド

Predicateのデフォルトメソッドには `negate` `and` `or` `isEqual` の4種類があります。

`negate` メソッドは条件判定を否定形にします。 `!(t > 200)` と記述するのと同様です。

```
Predicate<Integer> pre = t -> t > 200;
prices.stream().filter(pre.negate()).forEach(System.out::println);
```

`and` メソッドは `&&` と、 `or` メソッドは `||` と同様です。

```
Predicate<Integer> pre1 = t -> t > 200;
Predicate<Integer> pre2 = t -> (t % 3) == 0;
prices.stream().filter(pre1.and(pre2)).forEach(System.out::println);
prices.stream().filter(pre1.or(pre2)).forEach(System.out::println);
```

`isEqual` メソッドは static メソッドですが、isEqual の引数で指定したオブジェクトと等しいか判定する関数を生成します。引数に null を渡した場合は、 `Object#isNull` が返されます。

```
Predicate<Integer> pre3 = Predicate.isEqual(200);
prices.stream().filter(pre3).forEach(System.out::println);
```

因みに記述してあるサンプルについては、今回はStreamの例なので組み込んでいますが、次のように関数型インターフェース単体で使うこともできます。

```
Predicate<Integer> pre3 = Predicate.isEqual(200);
if (pre3.test(200)) {
```

```
...  
}
```

## map：変換

引数：Function<T, R> / 戻り値：Stream<R>

### ☑ 関数型インターフェース

mapメソッドはFunction<T, R>を引数に取ります。Functionは値を変換するための関数型インターフェースです。実装が必要なメソッドは `R apply(T t)` でT型の引数を1つ受け取り、R型の戻り値を返します。TとRは同じ型でも大丈夫です。

### ☑ mapの使い方

IntegerのStreamをStringに変換した結果を返すサンプルです。

```
List<Integer> lists = Arrays.asList(100, 200, 300, 400, 500);  
// ラムダ式  
lists.stream().map(String::valueOf).forEach(System.out::println);  
  
// 匿名クラス  
lists.stream().map(new Function<Integer, String>() {  
    @Override  
    public String apply(Integer t) {  
        return String.valueOf(t);  
    }  
}).forEach(System.out::println);
```

### ☑ defaultメソッド

Functionのデフォルトメソッドには `compose` `andThen` `identity` の3種類があります。

`andThen` メソッドを使用すると1つ目のFunctionを処理して2つ目のFunctionに結果を渡して処理します。 `map(~).map(~)` と繋げて処理するのと同じです。

```
List<String> str = Arrays.asList("100", "200", "300", "400", "500");  
Function<String, Integer> f1 = s -> Integer.parseInt(s);  
Function<Integer, Integer> f2 = s -> s * s;  
str.stream().map(f1.andThen(f2)).forEach(System.out::println);  
  
// mapを繋げるのと同じ  
str.stream().map(f1).map(f2).forEach(System.out::println);
```

`compose` メソッドは `andThen` の逆で2つ目のFunctionを処理して1つ目のFunctionに渡します。

```
str.stream().map(f2.compose(f1)).forEach(System.out::println);
```

`identity` メソッドはstaticメソッドであり、同じ値を返す関数を生成します。 `t -> t` のみです。

```
Function<Integer, Integer> f = Function.identity();  
lists.stream().map(f).forEach(System.out::println);
```

### ☑ プリミティブ型のmapメソッド

Streamではプリミティブ型を扱えないため、プリミティブ型用のStream（IntStream・LongStream・DoubleStream）があります。mapを使用してプリミティブ型に変換したい場合は、次のmapを使用します。

メソッド	引数	戻り値
mapToInt	ToIntFunction	IntStream
mapToLong	ToLongFunction	LongStream
mapToDouble	ToDoubleFunction	DoubleStream