Идиомы функционального программирования в Java 8

В этой статье даётся краткий обзор нескольких основных понятий функционального программирования, а также их использование в Java 8.

Обозначения

Прежде всего, введём условное обозначение, принятое в разговорах о функциях. Запись

$$f : T \to R$$

будет означать "функция f, принимающая аргумент типа T и возвращающая результат типа R". В Java такая функция могла бы быть реализована функциональным интерфейсом `Function<T, R>`. Функциональный интерфейс, соответствующий этой функции, мог бы иметь, например, такой абстрактный метод:

```
R apply(T arg);
```

Обозначение

$$f : T \times U \to R$$

означает функцию от двух аргументов типа T и U, возвращающую результат типа R.

Чистые функции

Основным строительным блоком функционального программирования являются *чистые функции* (pure functions). Функция является чистой, если выполняются два условия:

1. Функция является *детерминированной*, то есть всегда возвращает один и тот же результат для одних и тех же аргументов. Это означает, что функция не должна зависеть ни от какого изменяемого внешнего состояния, в том числе от изменяемых объектов и внешних по отношению к программе ресурсов (таких, как потоки ввода-вывода).
2. Функция не имеет *побочных эффектов* (side effects), то есть не изменяет состояние никаких объектов и внешних ресурсов.

Вот примеры чистых функций:

- Обычные арифметические операции, например, лямбда-выражение `(int x) -> x + 1`. Несмотря

на то, что возвращаемый результат не равен `x`, сам `x` при этом не меняется.
- Большинство методов-геттеров, например, `List.get` и `List.size`.
- Методы неизменяемых объектов, чьей единственной операцией является создание нового объекта на основе существующего объекта, например, `String.toCharArray`, `BigDecimal.add` и `LocalDateTime.atZone`.

Вот примеры функций, которые не являются чистыми:

- Методы вывода, в том числе `println`. Они детерминированы, но имеют побочные эффекты.
- Методы ввода, в том числе `BufferedReader.readLine`. Они не имеют побочных эффектов, но не детерминированы, потому что их результат зависит от данных в потоке ввода.
- Методы-мутаторы для изменяемого объекта, например, `List.set` и `List.add`.

Подход, основанный на чистых функциях, естественным образом приводит к архитектуре, в которой большинство объектов являются неизменяемыми или де факто неизменяемыми, а вычисления над ними производятся путём составления конвейеров из чистых функций, которые возвращают новые объекты, созданные на основе существующих.

Все промежуточные операции Stream API, в том числе `map` и `filter`, являются чистыми функциями при правильном использовании. Оконечные операции, например, `forEach`, не могут быть чистыми уже потому, что после вызова оконечной операции поток исчерпывается и более не может быть переиспользован. Кроме того, операции Stream API требуют, чтобы их аргументы по крайней мере не меняли элементы самого потока. Желательно, чтобы передаваемые потоку пользовательские функции были чистыми, потому что в этом случае операции над потоком будут эффективно параллелизуемыми и потокобезопасными (в смысле других потоков, тех, которые thread). Например, следующий код *не идиоматичен*:

```
IntStream ints = ...;
AtomicInteger sumOfSquares = new AtomicInteger(0);
ints.forEach(x -> sumOfSquares.addAndGet(x * x));
```

Это будет работать, но идиоматичный код с чистыми функциями проще для понимания и, скорее всего, будет эффективнее и для последовательного, и для параллельного потока:

```
IntStream ints = ...;
int sumOfSquares = ints.map(x -> x * x).sum();
```

Ленивые вычисления

С чистыми функциями тесно связана идея *ленивых вычислений* (lazy evaluation). Если у нас есть конвейер из чистых функций, то мы можем оптимизировать процесс вычислений, выполняя все вычисления только по требованию, когда клиентский код запросит их результат. Мы видим это в Stream API, где все промежуточные операции только определяют конвейер вычислений (и возвращают новый поток, являющийся обёрткой над нижележащим потоком), а вызов оконечной операции является сигналом, приводящим конвейер в движение.

Ленивые вычисления просто необходимы для корректной работы бесконечных потоков, потому что в противном случае их вычисление никогда не завершилось бы.

Например, следующий код находит первую степень двойки, большую 1000 (то есть 1024). Обратите внимание, что метод `iterate` производит бесконечный поток, но код тем не менее выполняется за конечное время именно потому, что метод `findFirst` обрывает вычисление элементов потока, как только будет вычислен первый.

```
IntStream.iterate(1, x -> x * 2)
         .filter(x -> x > 1000)
         .findFirst();
```

Ещё один пример — добавленные в класс `java.util.logging.Logger` в Java 8 методы, принимающие `Supplier<String>` вместо `String`. Поскольку построение строк логирования зачастую является относительно дорогой операцией, часто бывает желательно создавать эти строки только тогда, когда соответствующий уровень логирования включён. Вот так бы мы написали в Java 7 и ниже:

```
if (log.isLoggable(Level.DEBUG)) {
    log.debug("Server configuration: " + buildConfigParamsString());
}
```

В Java 8 можно просто обернуть этот код в лямбда-выражение, и тогда логгер вычислит строку, записываемую в лог, только при необходимости:

```
log.debug(() -> "Server configuration: " + buildConfigParamsString());
```

Функции высшего порядка

Функции высшего порядка — это функции, оперирующие функциями. Другими словами, они принимают функции, или возвращают функции, или и то и другое сразу.

Распространённые функции высшего порядка в функциональном программировании — это *композиция*, *частичное применение* функции и *каррирование*.

Композиция функций

Композиция — это применение одной функции к результату другой функции.

Композицией функций $f : T \to U$ и $g : U \to R$ по определению является функция $f \circ g : T \to R$, такая, что

$$(f \circ g)(x) = g(f(x)).$$

То есть сначала применяется `f`, а потом `g`.

В Java, если функции `f` и `g` являются объектами типа `Function`, то мы можем получить результирующую функцию лямбда-выражением `x -> g(f(x))` или вызовом метода `f.andThen(g)`. А вот метод `f.compose(g)` возвращает $g \circ f$, то есть применяет к аргументу сначала `g`, а потом `f`.

Частичное применение функции

Пусть у нас есть функция $f(x_1, ..., x_N)$ от N аргументов. Тогда, зафиксировав значения M из этих аргументов, где $M < N$, мы получим новую функцию $g(x_{M+1}, ..., x_N)$ от оставшихся $M - N$ аргументов.

Операция *частичного применения*, таким образом, является функцией высшего порядка, которая принимает функцию и фиксируемые аргументы и возвращает новую функцию от меньшего числа аргументов:

$$\text{partial}(f(x_1, ..., x_N), C_1, ..., C_M) = g,$$
$$\text{где } g(x_{M+1}, ..., x_N) = f(C_1, ..., C_M, x_{M+1}, ..., x_N).$$

Пример — прибавление к числу фиксированного аргумента:

```
// Частичное применение операции +
```

```
IntUnaryOperator plusOne = x -> x + 1;
IntStream.of(1, 2, 3).map(plusOne).collect(Collectors.toList());  // 2, 3, 4
```

Вот методы, превращающие любую функцию с двумя аргументами в функцию от первого (или второго) аргумента с фиксированным вторым (или, соответственно, первым):

```
<T, U, R> Function<T, R> bindSecond(BiFunction<T, U, R> func, U second) {
    return first -> func.apply(first, second);
}

<T, U, R> Function<U, R> bindFirst(BiFunction<T, U, R> func, T first) {
    return second -> func.apply(first, second);
}
```

Каррирование

Каррирование (currying) — пожалуй, самая концептуально сложная для понимания функция высшего порядка из представленных. Она тоже принимает функцию от нескольких аргументов и возвращает новую функцию от одного аргумента. Возвращённая функция сама является функцией высшего порядка: она принимает первый аргумент и частично применяет его, возвращая функцию от всех остальных аргументов. Функцию, возвращённую операцией каррирования, можно таким образом рассматривать как *фабрику* частично применённых функций.

$$\text{curry}(f(x_1, \ldots, x_N)) = g,$$
$$\text{где } g(x_1) = h,$$
$$h(x_2, \ldots, x_N) = f(x_1, \ldots, x_N).$$

Зачем нужна такая сложная операция? Последовательно применяя каррирование, мы можем любую функцию от N аргументов переделать в последовательность вызовов N функций от одного аргумента. Это важно в контекстах, где функции могут принимать только один аргумент. В языке Haskell, например, все функции имеют каррированную форму, то есть с точки зрения синтаксиса языка вызов функции, например, с тремя аргументами состоит из вызова функции с первым аргументом, результат этого вызова (тоже функция) применяется ко второму аргументу, в результате чего получается функция, которая применяется, наконец, к третьему аргументу.

В Java каррирование менее важно, и мне неизвестны примеры его применения на практике. Тем не менее вот пример его реализации для произвольной `BiFunction`:

```
<T, U, R> Function<T, Function<U, R>> curry(BiFunction<T, U, R> func) {
    return t -> u -> func(t, u);
}
```

Кстати, операция каррирования и язык Haskell названы в честь одного и того же человека: математика Хаскелла Карри (Haskell Curry).

Функции высшего порядка на контейнерах

Теперь перейдём к рассмотрению типов, которые могут содержать в себе некоторые значения и при этом поддерживают операции из функционального программирования. Все они концептуально являются обобщёнными типами (generic), поскольку их тип зависит от типа элементов. Не всегда это означает, что они являются обобщёнными именно с точки зрения синтаксиса языка Java: из-за ограничений, не позволяющих сделать примитивные типы параметрами обобщённых, примитивные типы из соображений эффективности имеют свои собственные реализации `Stream` и `Optional`. Таковы `IntStream`,

`LongStream` и `DoubleStream`, а также `OptionalInt`, `OptionalLong` и `OptionalDouble`. Концептуально, однако, эти типы можно рассматривать наряду с обобщёнными типами `Stream<T>` и `Optional<T>`.

Операция reduce

Операция свёртки (reduce) имеет смысл только для контейнеров, которые могут содержать более одного элемента. Поэтому из новых типов Java 8 её предоставляет только семейство Stream.

Пусть у нас имеется обобщённый тип `G<T>` и функция *accum* : $T \times T \to T$, называемая *аккумулятором* (accumulator, накопитель). Тогда функция высшего порядка

$$\text{reduce} : G\langle T\rangle \times T \times (accum : T \times T \to T) \to T$$

берёт контейнер, *начальное значение* типа T и функцию, которую она последовательно применяет к каждому элементу, получая на выходе некоторое значение, основанное на всех элементах потока.

Например, сумма и произведение и являются частными случаями операции reduce:

```
// Сумма (но лучше использовать метод sum)
IntStream ints1 = ...;
int sum = ints1.reduce(0, (x, y) -> x + y);

// Произведение
IntStream ints2 = ...;
int product = ints2.reduce(1, (x, y) -> x * y);
```

Есть ещё два варианта reduce. Один из них позволяет опустить начальное значение, и тогда в качестве такового выступит первый элемент потока. Эта версия возвращает `Optional`: в случае, если поток пуст, возвращаемый `Optional` тоже будет пустым, в противном случае он будет содержать результат операции reduce. Наконец, последний вариант позволяет вернуть значение типа, отличного от типа элементов потока, но ему нужно передать ещё одну функцию, которая будет отвечать за слияние двух значений этого нового типа:

```
// Подсчёт числа элементов потока
// (но лучше используйте метод count!)
Stream<String> strings = ...;
int elementCount = strings.reduce(0,
    // аккумулятор
    (count, str) -> count + 1,
    // функция комбинации
    (count1, count2) -> count1 + count2);
```

Функция комбинации используется только в параллельных потоках. Она нужна затем, чтобы слить результаты выполнения reduce для разных частей потока, к которым операция reduce будет применяться параллельно и независимо друг от друга.

Функторы и map

Обобщённый тип `G<T>` называется *функтором* (functor), если для него определена функция высшего порядка `map`, которая принимает функцию $f : T \to R$ и возвращает новый объект `G<R>`, состоящий из результатов применения переданной функции к каждому элементу исходного объекта.

$$\text{map} : G\langle T\rangle \times (f : T \to R) \to G\langle R\rangle$$

В Java функторами являются типы `Stream`, `Optional` и `CompletableFuture`. При этом `Stream`

применяет функцию, переданную в `map`, к каждому элементу, а `Optional` применяет её к единственному аргументу, если он есть (и возвращает пустой `Optional`, если и исходный был пустым). Важно, что, в соответствии с принципами использования чистых функций, *map не изменяет исходный объект*, а возвращает новый.

Для типа `CompletableFuture` эквивалентом `map` является метод `thenApply`, который применяет указанную функцию к значению, как только это значение будет получено.

```
Stream<LocalDate> dates = ...;
Stream<String> dateStrings = dates.map(Object::toString);

Optional<LocalDate> maybeADate = ...;
Optional<String> maybeAString = maybeADate.map(Object::toString);

CompletableFuture<LocalDate> eventuallyADate = ...;
CompletableFuture<String> eventuallyAString =
    eventuallyADate.thenApply(Object::toString);
```

Монады и flatMap

Обобщённый тип `G<T>` называется *монадой* (monad), если для него определена функция высшего порядка `flatMap`, которая принимает функцию $f : T \to G\langle R\rangle$ и возвращает новый объект `G<R>`, состоящий из объединения исходной монады и монад, возвращённых применением переданной функции к каждому элементу исходного объекта.

$$\text{flatMap} : G\langle T\rangle \times (f : T \to G\langle R\rangle) \to G\langle R\rangle$$

Обратите внимание, что функция, которую мы передаём во `flatMap`, возвращает новую *монаду*, а не обычное значение! В этом отличие `flatMap` от `map`. При этом та монада, которую возвращает сам метод `flatMap`, каким-то образом комбинирует исходную монаду со всеми теми, которые вернули вызовы переданной функции.

В традиционной терминологии функционального программирования вместо `flatMap` используется термин "связывание" (bind). Типы `Stream`, `Optional` и `CompletableFuture` являются не только функторами, но и монадами, а смысл связывания зависит от конкретной монады:

- `Stream` склеивает подряд все потоки, в которые превращаются его элементы.
- `Optional` применяет переданную функцию к аргументу, если он не пуст, и возвращает полученный `Optional` (который может быть пустым или непустым). Если же `Optional` пуст, то вызывать переданную функцию он не будет и просто вернёт пустой `Optional`.
- Для `CompletableFuture` эквивалентом `flatMap` является метод `thenCompose`. Когда в `CompletableFuture` появляется значение, запускается обработчик, переданный в `thenCompose`, который возвращает новый `CompletableFuture`. При этом сам метод `thenCompose` возвращает `CompletableFuture`, оборачивающий оба асинхронных вычисления последовательно, и методы `thenXXX` у этой обёртки вызовут свои обработчики тогда, когда завершится сначала первое вычисление, а потом второе.

```
Stream<String> strings = Stream.of("Hello", "World");
Stream<Character> chars = strings.flatMap(String::chars);
// 'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd'

// CompletableFuture
// Первый асинхронный запрос к серверу: CompletableFuture<User>
```

```
server.login("username", "password")
    // Второй асинхронный запрос к серверу: CompletableFuture<List<Post>>
    .thenCompose(user -> server.getLastPosts(user, 10))
    .thenAccept(posts -> {
        for (Post post: Posts) {
            showPost(post);
        }
    });
```

Кроме `flatMap`, у монады должен быть ещё метод, позволяющий сразу завернуть ("поднять") в монаду обычное значение. Такой метод в обычной терминологии называется unit. В Java такие методы действительно есть: это статические методы `Stream.of`, `Optional.of` и `CompletableFuture.completedFuture`.

Ценность монад состоит в том, что они предоставляют абстракцию *цепочки последовательных вычислений*, при этом инкапсулируя "состояние мира" на каждом шаге вычисления. Это особенно ярко видно на примере `CompletableFuture`, где аргумент каждого метода `thenXXX` выполняется после завершения соответствующего этапа асинхронных вычислений. При этом монада сама знает, как "развернуть" полученный аргумент и передать его обработчику, который работает именно с обычным значением, а не с монадой. В Haskell, в котором в принципе нет функций с побочными эффектами, монада под названием `IO` используется для описания обычных императивных последовательных вычислений, которые в императивном языке наподобие Java были бы разделены точками с запятой. Чтобы понять, о чём идёт речь, представим себе, как мог бы выглядеть подобный синтаксис в Java:

```
public class IO<T> {
    <R> IO<R> thenCompose(Function<T, IO<R>> func);
}

IO<String> readLine();
IO<Void> writeLine(String str);
```

Тогда обычная императивная программа:

```
public static void main(String[] args) {
    System.out.println("Как вас зовут?");
    String name = new Scanner(System.in).nextLine();
    System.out.println("Привет, " + name);
}
```

перепишется с использованием монады `IO` вот так:

```
public static IO<?> main(String[] args) {
    return writeLine("Как вас зовут?")
        .thenCompose(x -> readLine())
        .thenCompose(name -> writeLine("Привет," + name));
}
```

Иными словами, в Haskell функция `main` (да, она там есть) сама, строго говоря, не запускает никаких вычислений. С помощью монады `IO` она *возвращает описание* процесса вычислений, который Haskell должен применить к внешней среде (включая возможности ввода-вывода), в которой выполняется программа.

Ковариантность и контравариантность

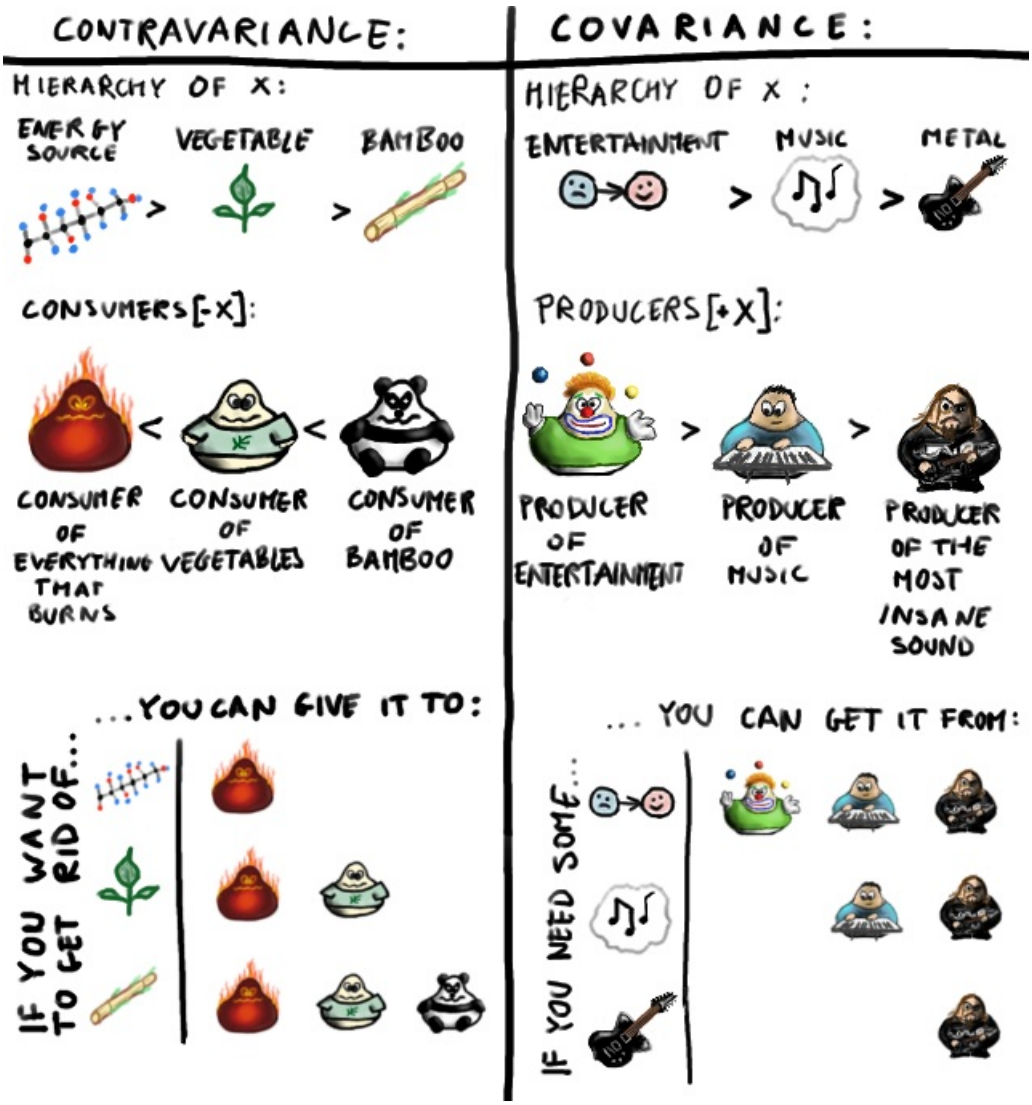
Пусть тип `Sub` — это подтип типа `Super`. Тогда:

- Обобщённый тип `G<T>` называется *ковариантным* по типу `T`, если логически каждый `G<Sub>` является подтипом (частным случаем) `G<Super>`.
- Обобщённый тип `G<T>` называется *контравариантным* по типу `T`, если логически каждый `G<Super>` является подтипом (частным случаем) `G<Sub>`.

Ключевое слово здесь — логически, а не на уровне синтаксиса Java. Например, `Stream` логически ковариантен, то есть любой `Stream<String>` можно было бы использовать там, где нужен `Stream<Object>`. Но в Java обобщённые типы *инвариантны*, то есть с точки зрения языка Java для *любых* типов `T1` и `T2` любой обобщённый тип `G<T1>` никогда не является ни подтипом, ни супертипом `G<T2>`.

Как мы знаем из Effective Java, в параметрах типа можно обойти это ограничение, если использовать `<? extends T>` для ковариантных типов и `<? super T>` для контравариантных.

Обязательная картинка в тему:



Отсюда видно, что *производители* значений (producers) всегда ковариантны, а *потребители* значений (consumers) всегда контравариантны. Отсюда проистекает правило из Effective Java, описывающее, когда мы должны использовать `extends` и `super` в объявлениях методов:

PECS значит производитель — `extends`, потребитель — `super`
(producer — `extends`, consumer — `super`).

Разберёмся, как этот принцип применяется в функциональном программировании на Java. Замечательное свойство чистых функций состоит в том, что из-за отсутствия побочных эффектов (чистые функции не имеют права изменять ни свои аргументы, ни какие-либо внешние объекты) для них свойства производителя и потребителя всегда чётко определены.

Чистые функции всегда являются потребителями своих аргументов и производителями своих возвращаемых значений.

Из этого прямо следует, что:

Объявление метода, использующее чистые функции, должно объявлять их как контравариантные (`super`) по типам принимаемых аргументов и как ковариантные (`extends`) по типу возвращаемого значения.

Теперь вы и сами можете понять, почему методы `forEach`, `map` и `sort` интерфейса `Stream<T>` объявлены именно так:

```
void forEach(Consumer<? super T>)
<R> Stream<R> map(Function<? super T, ? extends R>)
Stream<T> sorted(Comparator<? super T>)
```