Автоматический рефакторинг Java-кода с использованием Stream API

Р. А. Иванов¹, Т. Ф. Валеев²

¹ Новосибирский государственный университет
Новосибирск, Россия

² Институт систем информатики им. А. П. Ершова СО РАН
Новосибирск, Россия

Аннотация

Долгое время функциональное программирование на Java было невозможно. Однако в 8-й версии Java появились лямбда-выражения. Благодаря поддержке стандартных библиотечных классов (Stream, Optional и т. д.) на Java стало возможно описывать преобразования над данными в функциональном стиле.

Java – достаточно старый язык, на нем написано большое количество императивного кода. Для того чтобы воспользоваться преимуществами нового подхода, требуется выполнить нетривиальный рефакторинг, что в случае осуществления человеком может быть весьма утомительным, легко совершить ошибку. К счастью, для достаточно большого количества ситуаций данный рефакторинг можно безопасно осуществить автоматически.

На основе IntelliJ Idea был разработан программный инструмент, который позволяет обнаружить места, где возможно автоматическое преобразование императивного кода в эквивалентный с использованием Stream API, а также автоматическое исправление, которое позволяет произвести замену. Рефакторинг пользуется средствами IntelliJ Idea для анализа Java-кода, а также интегрируется в саму IDE. Одним из основных критериев корректности работы алгоритма является безопасность данного преобразования. Пользователь не может доверять инструменту, если преобразование может изменять семантику кода. В данной статье рассматриваются различные ограничения, которые накладываются на шаблоны кода для того, чтобы преобразование без искажения семантики было возможно.

Данный рефакторинг был протестирован на различных библиотеках для проверки сохранения семантики путем проверки результатов тестирования до и после применения рефакторинга.

В статье не будет обсуждаться влияние использования Stream API на производительность приложения.

Ключевые слова

Stream API, refactoring, рефакторинг, преобразование кода, IntelliJ, IDE

Для цитирования

Иванов Р. А., Валеев Т. Ф. Автоматический рефакторинг Java-кода с использованием Stream API // Вестник НГУ. Серия: Информационные технологии. 2019. Т. 17, № 2. С. 49–60. DOI 10.25205/1818-7900-2019-17-2-49-60

Automatic Refactoring of Java Code Using the Stream API

R. A. Ivanov¹, T. F. Valeev²

¹ Novosibirsk State University
Novosibirsk, Russian Federation

² A. P. Ershov Institute of Informatics Systems SB RAS
Novosibirsk, Russian Federation

Abstract

For a long time, functional Java programming was not possible. However, lambda expressions appeared in version 8 of the Java language. Due to the support of standard library classes (Stream, Optional, etc.) in Java, it became possible to describe transformations over data in a functional style.

Java is a rather old language with a large imperative code base written in it. In order to take advantage of the new approach, it is necessary to perform a non-trivial refactoring, which can be very tedious and error prone when applied manually.

Fortunately, for a sufficiently large number of situations, this refactoring can be safely performed automatically.

Using IntelliJ Idea as a platform software tool was developed that allows you to find places where it is possible to automatically convert an imperative code to an equivalent code that uses Stream API, as well as an automatic fix that allows you to make a replacement. Refactoring uses IntelliJ Idea framework to analyze Java code, and integrates into the IDE itself.

One of the main criteria for the correct operation of the algorithm is the security of this transformation. The user cannot trust the tool if the transformation can change the semantics of the code.

This article discusses various constraints that are imposed on code patterns so that transformation without distortion of semantics is possible.

Refactoring has been tested in various libraries to verify the semantics are preserved by checking the test results before and after applying refactoring.

This article will not describe the impact of using the Stream API on the performance of the application.

Keywords

refactoring, IDE, code transformation, IntelliJ, Stream API

For citation

Ivanov R. A., Valeev T. F. Automatic Refactoring of Java Code Using the Stream API. *Vestnik NSU. Series: Information Technologies*, 2019, vol. 17, no. 2, p. 49–60. (in Russ.) DOI 10.25205/1818-7900-2019-17-2-49-60

Введение

Целью данной работы было создание инструмента для автоматического преобразования кода на Java, написанного в императивном стиле, в код на функциональном стиле с использованием цепочки Stream API, а также интеграция этого инструмента в среду разработки IntelliJ Idea.

Использование Stream API вместо традиционных циклов имеет весьма значительные преимущества: возможна автоматическая параллелизация, которая может дать прирост производительности, часто такой код намного более короткий и лаконичный, а также более четко выражает намерение автора¹.

Для среды разработки крайне важно предлагать различные варианты преобразования кода, так как каждый из них может обладать своими преимуществами и недостатками, и программист может сам решать, какой из них лучше в данной ситуации.

Благодаря инфраструктуре IDE данные автоматические преобразования могут производиться в одно нажатие кнопки.

Несмотря на то, что в IDE NetBeans данный рефакторинг уже присутствует, в нем было обнаружено значительное количество недостатков и возможностей улучшения, которые учтены при разработке. Данный рефакторинг было решено разрабатывать для IDE IntelliJ Idea, поскольку она является самой распространенной IDE для Java. Использовать разработки других IDE не представлялось возможным из-за больших различий в семантической модели Java, а также из-за того, что модель рефакторинга там значительно более проста.

Операции Stream API

Stream API² предоставляет возможность обработки совокупности однотипных данных. Сценарий обработки записывается декларативно посредством цепочки вызовов. Типичный пример использования Stream API выглядит следующим образом:

¹ Rahad K., Cao Z., Cheon Y. A Thought on Refactoring Java Loops Using Java 8 Streams, 2017.

² <https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>

```
List<String> result = listOfStrings.stream()
    .map(s -> s.trim())
    .filter(s -> !s.isEmpty())
    .limit(10)
    .collect(Collectors.toList());
```

Здесь у каждой строки из списка `listOfStrings` удаляются начальные и конечные пробелы с помощью операции `map`; затем пустые строки отфильтровываются с помощью `filter`; после этого результат ограничивается десятью строками с помощью `limit`, и, наконец, выполняется сборка в результирующий список с помощью `collect`. Здесь имеются операции различных типов.

- Источник потока – метод `stream` (входит в интерфейс `List`), создает поток из списка. Имеются другие виды источников, которые, например, создают поток из массива, из строк файла, из целых чисел в заданном диапазоне и т. д. Источник возвращает объект типа `Stream`.

- Промежуточные (intermediate) операции – `map`, `filter` и `limit`. Эти операции также возвращают объект типа `Stream`, позволяя объединять в цепочку несколько промежуточных операций. Важная особенность заключается в том, что промежуточные операции всегда «ленивы»: они не производят вычислений, а только запоминают действия, которые необходимо выполнить. Имеются и другие промежуточные операции: `distinct` (удалить повторяющиеся элементы), `skip` (пропустить заданное количество первых элементов) и т. д.

- Концевая (terminal) операция – `collect`. Именно при выполнении концевой операции производится вся цепочка вычислений и сбор результатов. Операция `collect` принимает так называемый коллектор – объект, специфицирующий способ объединения результатов. В данном случае используется коллектор `toList`, который собирает результаты в список. Другие коллекторы могут собирать в множество, ассоциативный массив, в строку с заданным разделителем и т. д. Помимо `collect` имеются и другие концевые операции. Например, `count` просто подсчитывает количество результатов, а `anyMatch` проверяет, имеется ли хоть один результат, удовлетворяющий заданному условию.

Промежуточные операции делятся на операции без состояния (`map` и `filter`) и операции с состоянием (`limit` или `distinct`). Операции без состояния значительно проще по своей сути: они единообразно обрабатывают каждый элемент, при этом порядок их обработки неважен. Обычно такие операции легко выражаются в традиционном цикле. Например, операции `map` соответствует присваивание или объявление переменной, а операции `filter` – условный оператор `if`. Операции с состоянием сложнее: состояние необходимо хранить в дополнительной переменной либо получать неявно. В традиционных циклах соответствующая семантика выражается разными способами, и их поддержка в процессе преобразования цикла в цепочку `Stream API` представляет определенную трудность. Ниже приводится два способа выразить тот же алгоритм с помощью традиционного цикла:

```
List<String> result = new ArrayList<>();
int count = 0;
for (String s : listOfStrings) {
    s = s.trim();
    if (!s.isEmpty()) {
        result.add(s);
        if (++count == 10) break;
    }
}

List<String> result = new ArrayList<>();
for (String s : listOfStrings) {
    s = s.trim();
    if (!s.isEmpty()) {
        result.add(s);
        if (result.size() == 10) break;
    }
}
```

В первом варианте для подсчета количества добавленных элементов используется дополнительная переменная-счетчик `count`. Во втором способе мы пользуемся размером результирующего списка, который изначально пуст, и на каждом шаге туда добавляется ровно одна запись. Второй способ короче, но менее универсален. К примеру, он не сработает, если результаты требуется собрать не в список, а в множество или в строку с разделителем. Оба приведенных варианта IntelliJ Idea поддерживает, но возможны и другие варианты: некоторые предпочитают проверять счетчик перед добавлением элемента, считать в обратном порядке (от 10 до 0) и т. д.

У операции `limit` есть другая важная особенность: это так называемая короткозамкнутая операция. Она может завершить весь процесс до того, как закончатся входные данные. Короткозамкнутыми могут быть как промежуточные, так и конечные операции. Пример конечной короткозамкнутой операции – `anyMatch`. Эта операция возвращает булеву «истину», если хоть один элемент потока удовлетворяет условию, переданному в виде лямбда-выражения параметром `anyMatch`. При этом, естественно, обработка прекращается.

Поток элементов объектного типа T имеет тип `java.util.stream.Stream<T>`. Так как в языке Java примитивный тип не может являться аргументом обобщенного типа, невозможно создать, к примеру, поток элементов типа `float`. Однако для удобства и повышенного быстродействия три примитивных типа – `int`, `long` и `double` – поддерживаются специально: для них имеются потоковые типы `IntStream`, `LongStream` и `DoubleStream`. Далее под «поддерживаемым типом» мы будем иметь в виду любой тип, который может быть элементом потока: либо любой объектный тип, либо `int`, `long` или `double`. Другие примитивные типы (`boolean`, `short` и т. д.) назовем неподдерживаемыми типами. Также для простоты под `Stream<T>` мы будем иметь в виду поток элементов любого поддерживаемого типа (если T соответствует типу `int`, то вместо `Stream<int>` подразумевается `IntStream`).

Преобразование фрагментов кода в лямбда-выражение

Будем называть фрагментом Java-выражение (expression глава 15 JLS), Java-утверждение (statement, JLS §14.5) либо совокупность из нескольких подряд идущих Java-утверждений в пределах одного блока (JLS §14.2). Возможен также пустой фрагмент, не содержащий никаких утверждений и выражений. Фрагмент можно преобразовать в тело лямбда-выражения для потоковой операции при соблюдении трех условий.

У1. Фрагмент не должен ссылаться на локальные переменные и параметры, которые определены за пределами этого фрагмента и не являются финальными и эффективно-финальными.

У2. Фрагмент не должен бросать проверяемых исключений.

У3. Фрагмент не должен содержать операторов, передающих управление за пределы данного фрагмента за исключением оператора `throw`.

Условие У1 необходимо для любого лямбда-выражения Java в соответствии с § 15.27.2 спецификации языка Java 8 (далее JLS)³. Понятие финальной и эффективно-финальной переменной приводится в JLS § 4.12.4. Основное требование заключается в том, чтобы значение присваивалось переменной ровно один раз на каждом пути управления.

Условие У2 не требуется для любых лямбда-выражений. Лямбда-выражение может бросать проверяемые исключения, если соответствующий абстрактный метод функционального интерфейса (JLS § 9.8) их объявляет. Однако ни один из функциональных интерфейсов, используемых в Stream API (к примеру, `java.util.function.Predicate`), не объявляет проверяемых исключений.

Условие У3 также требуется для любых лямбда-выражений в соответствии с их семантикой в языке Java в отличие, например, от языка Kotlin, где в ряде случаев возможна нелокальная передача управления из лямбда-выражения. Оператор `throw`, бросающий исключе-

³ <https://docs.oracle.com/javase/specs/>

ние, допустим, так как, по спецификации Stream API, любое исключение, выброшенное из лямбда-выражения, используемого в цепочке операций, будет выброшено наружу. Разумеется, в соответствии с У2, допустимы только непроверяемые исключения.

Условие У2 можно проверить для всего цикла: если в цикле есть хоть одна операция, бросающая проверяемое исключение, то преобразование цикла в Stream API невозможно. С другой стороны, условие У1 для всего цикла проверять неоправданно. Во-первых, изменение переменной может не попасть в лямбда-выражения, а стать результатом цепочки вызовов Stream API, как, к примеру, в следующем листинге:

```
static long countNonEmptyStrings(List<String> strings) {  
    long count = 0;  
    for(String s : strings) {  
        if(!s.isEmpty()) {  
            count++;  
        }  
    }  
    return count;  
}
```

Здесь переменная count не является эффективно-финальной (модифицируется в цикле) и объявлена за пределами цикла. Однако этот код легко трансформировать с использованием Stream API:

```
static long countNonEmptyStrings(List<String> strings) {  
    long count = strings.stream().filter(s -> !s.isEmpty()).count();  
    return count;  
}
```

Поэтому каждый фрагмент, который необходимо превратить в лямбда-выражение, следует проверять отдельно. В данном примере лямбда-выражением становится только условие !s.isEmpty(), для которого У1 выполняется.

Также возможна ситуация, когда переменная, не являющаяся эффективно-финальной в исходном цикле, станет таковой после преобразования. Тогда преобразование все еще возможно. К примеру, рассмотрим такой цикл:

```
for(int i=0; i<10; i++) {  
    System.out.println(i);  
}
```

Здесь требуется преобразовать в лямбда-выражение вызов метода System.out.println(i), который ссылается на изменяемую переменную i, определенную за пределами выражения. Однако после преобразования цикла i становится неизменяемой:

```
IntStream.range(0, 10).forEach(i -> System.out.println(i));
```

Условие У3 также не стоит проверять для всего цикла сразу. Действительно, лямбда-выражение не может содержать операторов типа break (если только само не содержит цикл), а оператор return завершит только само лямбда-выражение, но не окружающий его метод. Однако многие циклы, содержащие операторы управления потоком, все-таки можно преобразовать в цепочку Stream API. Операторы break и return иногда можно трансформировать в короткозамкнутую операцию. Оператор continue, переходящий на следующую итерацию цикла, допустим в качестве тела условного оператора if. К примеру, следующие два фрагмента семантически эквивалентны:

```

List<String> result = new ArrayList<>();
for (String s : listOfStrings) {
    if (!s.isEmpty()) {
        result.add(s);
    }
}

List<String> result = new ArrayList<>();
for (String s : listOfStrings) {
    if (s.isEmpty()) continue;
    result.add(s);
}

```

Модель Stream API

Для обсуждения алгоритма преобразования циклов в вызовы Stream API нам потребуется несколько определений.

Для заданной переменной Java-кода V будем обозначать через T_V ее тип.

Подфрагментом F' фрагмента F назовем фрагмент, целиком содержащийся во фрагменте F (являющийся его частью) и не совпадающий с F . Обозначим $F' \subset F$.

Генератор $G(V, S, F)$ – это совокупность выходной переменной V поддерживаемого типа, Java-выражения S , имеющего тип $\text{Stream}<T_V>$, и фрагмента F .

Представление генератора $G(V, S, F)$ есть следующий Java-цикл:

```

for(Iterator<T> it = S.iterator(); it.hasNext(); ) {
     $T_V$   $V$  = it.next();
     $F$ 
}

```

Имеются простые и составные генераторы. Простой генератор создается непосредственно по Java-циклу L таким образом, что представление этого генератора семантически эквивалентно исходному циклу L . Определим несколько видов простых генераторов, каждый из них соответствует определенному виду циклов (табл. 1).

Таблица 1

Определения генераторов

Table 1

Generator Definitions

Название	Вид цикла	Операция
Обход коллекции	for(T_V V : EXPR) F EXPR имеет тип <code>java.util.Collection</code>	$G_{\text{coll}} = (V, \text{"(EXPR).stream()", } F)$
Обход массива	for(T_V V : EXPR) F EXPR имеет тип массива	$G_{\text{arr}} = (V, \text{"Ar- rays.stream(EXPR)", } F)$
Диапазон чисел	for(T_V V = START ; V < BOUND ; $V++$) F T_V – либо <code>int</code> , либо <code>long</code>	$G_{\text{range}} = (V, \text{"IntStream / LongStream.range(START, BOUND)", } F)$
Диапазон чисел (закрытый)	for(T_V V = START ; V <= BOUND ; $V++$) F T_V – либо <code>int</code> , либо <code>long</code>	$G_{\text{closed}} = (V, \text{"IntStream / LongStream.rangeClosed(START, BOUND)", } F)$
Строки BufferedReader	<code>String</code> V ; while($(V = \text{BR.readLine()}) \neq \text{null}$) F BR имеет тип <code>java.io.BufferedReader</code>	$G_{\text{br}} = (V, \text{"BR.lines()", } F)$

Преобразование $X(E)$ – функция от Java-выражения типа `Stream`, которая возвращает новое Java-выражение типа `Stream` (возможно, с другим типом элементов). В большинстве случаев преобразование добавляет новый вызов к существующей цепочке вызовов. К примеру, фильтрующее преобразование может выглядеть так:

$$\text{Filt}_{V, P}(E) := E.\text{filter}(V \rightarrow P),$$

где V – переменная, E – Java-выражение, имеющее тип `Stream<TV>`, а P – Java-выражение, представляющее собой предикат от переменной V (условие фильтрации) и удовлетворяющее условиям У1–У3. Результирующее выражение также имеет тип `Stream<TV>`.

Операция $O(V_{in}, F_{in})$ – это функция от входной переменной V_{in} и фрагмента, которая возвращает тройку (V_{out}, X, F_{out}) , состоящую из выходной переменной V_{out} , преобразования X и выходного фрагмента F_{out} , который является подфрагментом F_{in} . Операции определены только для некоторых фрагментов. К примеру, операция фильтрации определена для фрагментов вида `if(P) F` и возвращает тройку $(V_{in}, \text{Filt}_{V_{in}, P}, F)$.

Всякая операция O обладает следующим свойством: для любого генератора $G(V, S, F)$, для которого $O(V, F)$ определена и возвращает тройку (V_{out}, X, F_{out}) , представление генератора G семантически эквивалентно представлению генератора $G'(V_{out}, X(S), F_{out})$. Генератор G' назовем составным генератором.

Мы определили следующие операции:

- *операция фильтрации* определена для фрагментов вида `if(P) F`. Возвращает тройку $(V_{in}, E \rightarrow E.\text{filter}(V_{in} \rightarrow P), F)$;
- *операция обратной фильтрации* определена для фрагментов вида `if(P) continue; F`. Возвращает тройку $(V_{in}, E \rightarrow E.\text{filter}(V_{in} \rightarrow !(P)), F)$;
- *операция объявления переменной* определена для фрагментов вида `TV V = R; F`, где V – объявленная переменная поддерживаемого типа T_V , причем V_{in} используется только в выражении R и не используется в F . Возвращает тройку $(V, E \rightarrow E.\text{map}(V_{in} \rightarrow R), F)$. Если тип входной или выходной переменной примитивный, вместо `map` может использоваться другой метод `Stream API` (например, `mapToInt`, `boxed`, `asLongStream` и т. д.);
- *операция переписывания переменной* определена для фрагментов вида `Vin = R; F`. Возвращает тройку $(V_{in}, E \rightarrow E.\text{map}(V_{in} \rightarrow R), F)$;
- *операция вложенного цикла* определена для фрагментов, являющихся циклами, для которых существует простой генератор $G(V_{out}, S, F)$, причем V_{in} не используется в F . Возвращает тройку $(V_{out}, E \rightarrow E.\text{flatMap}(V_{in} \rightarrow S), F)$. Если тип входной или выходной переменной примитивный, могут использоваться конструкции вида `E.flatMapToInt(Vin -> S)` или `E.mapToObj(Vin -> S).flatMap(Function.identity())`.

Расширенное преобразование назовем функцией $EX(Fb; E; Fa)$, которая возвращает новое Java выражение.

В отличие от преобразования, помимо непосредственно выражения типа `Stream`, принимает еще и 2 фрагмента, которые расположены непосредственно до внешнего цикла и после внешнего цикла.

Назовем терминалом функцию $T(G, Fb, Fl, Fa)$, где $G(V, S, F)$, которая для S возвращает расширенное преобразование X . Так же, как и операция, может быть определена не для всех фрагментов.

Определим следующие терминалы (табл. 2). В табл. 2 для краткости префикс `Fb; E; Fa ->` был опущен. Данный список не является полным, рефакторинг содержит значительно больше вариантов преобразований, в табл. 2 приведены примеры из каждого класса терминалов.

Таблица 2

Определения терминалов

Table 2

Terminal Definitions

Название терминала	Требования к фрагментам	Расширенное преобразование
Нахождение первого элемента	Fb вида <code>Optional<T> x = Optional.empty();</code> Fl вида <code>if (P) result = R; break;</code>	<code>Optional<T> x = E.findFirst(V -> P);</code> Fa;
Редукция элементов	Fb вида <code>T a = I;</code> Fl вида <code>for (T' x: c) a # x,</code> где # – одна из операций {+, *, , &&, , &, ^}	<code>T a = I; A # E.reduce(I, (a, b) -> a # b);</code> Fa, где I – значение идентичности (identity value) для данной операции, а переменные a и b – уникальные в данной области видимости
Нахождение минимального / максимального элемента	Fb вида <code>T a = I;</code> Fl вида <code>for (T' x: c) if (x # a) a = x,</code> где # обозначает > или <	<code>T a = op(I, E.max());</code> Fa, где op – max или min, в зависимости от #
Сохранение в список	Fb вида <code>List<T> l = new ArrayList<>()</code> Fl вида <code>for (T' x: c) l.add(x)</code>	<code>E.collect(Collectors.toList());</code> Fa
Склейка строк	Fb вида: <code>StringBuilder sb = new StringBuilder(prefix);</code> Fl вида: <code>boolean isTail = false;</code> <code>for (T e: collection)</code> <code>if (isTail) sb.append(delimiter);</code> <code>else isTail = true;</code> <code>sb.append(F);</code> Fa вида: <code>sb.append(suffix).toString();</code> где F содержит только переменную e или другие финальные переменные	<code>E.map(F).collect(Collectors.joining(delimiter, prefix, suffix))</code>
Применение операции для каждого элемента	Отсутствуют	Fb; <code>E.forEach(V -> Fl);</code> Fa

Рассмотрим подробнее алгоритм преобразования императивного кода в функциональный. Далее приводится псевдокод данного алгоритма:

```
extractStream(Fb, Fl, Fa) {
    G = findGeneratorFor(Fl)
    if (G == null) return null
    return extractTail(G, Fb, Fa)
}

extractTail(G, Fb, Fa) {
    O = findOperationFor(G.F, G.V)
    (Vout, X, Fout) = O(G.v, G.F)
    G' = G(Vout, X(G.s), Fout)
    if (G' != null) {
        return extractTail(G', Fb, Fa)
    } else {
        return extractTerminal(G, Fc)(G, Fb, G.F, Fa)
    }
}
```

Полагаем, что Fb – фрагмент контекста до цикла, Fl – фрагмент цикла, Fa – фрагмента контекста после цикла, а G – базовый генератор для текущего шага рекурсии.

```
extractTerminal(G, Fb, Fa) {
    for (matcher in terminalMatchers) {
        T = matcher.match(G.F, G.V, Fb, Fa)
        if (O != null) return T
    }
    return null
}

findOperationFor(F, V) {
    for (matcher in operationMatchers) {
        O = matcher.match(F, V)
        if (O != null) return O
    }
    return null
}
```

Следует пояснить, что operationMatchers и terminalMatchers соответствуют описанию операций и терминалов, которые были рассмотрены ранее.

Нетрудно заметить, что подход основан на преобразовании некоторых шаблонов кода, из которых и строится цикл. В статьях [1–3] были исследованы различные шаблоны кода в открытых кодовых базах, большая их часть была поддержана в данном рефакторинге.

Сравнение с конкурирующими реализациями

Данная реализация рефакторинга не была первой, аналоги есть в других IDE для Java: Eclipse и NetBeans. Далее мы приведем их сравнение с рефакторингом в IntelliJ Idea.

LambdiFicator

Аналогичная возможность реализована в рамках проекта LambdiFicator [4; 5] для IDE NetBeans 8.1. По сравнению с разработкой, представленной в данной статье, эта реализация обладает рядом недостатков.

1. Из промежуточных операций поддерживаются только операции `filter` и `map`. Операции `flatMap`, `distinct`, `limit`, `sorted` не поддерживаются ни в каком виде.

2. Из конечных операций поддерживаются только `reduce`, `forEach`, `anyMatch` и `noneMatch`. Не поддерживаются такие важные способы завершения потока, как `collect(toList())`; `collect(joining())`; `toArray()`; `findFirst()`.

3. Прimitives типы отождествляются с соответствующими им объектными типами (к примеру, тип `int` с типом `java.lang.Integer`), в то время как Stream API обеспечивает специальную поддержку примитивных типов `int`, `long`, `double` с помощью интерфейсов `IntStream`, `LongStream` и `DoubleStream` соответственно. Такое отождествление не только ухудшает производительность результирующего кода, но и в некоторых случаях может опасным образом изменить семантику кода. Рассмотрим, к примеру, следующий листинг:

```
import java.util.*;

public class Test {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3));
        List<Integer> input = Arrays.asList(0, 1);
        for (int idx : input) {
            if (idx > 0) {
                list.remove(idx);
            }
        }
        System.out.println(list);
    }
}
```

Данная Java-программа при выполнении выдает `[1, 3]`, так как в цикле вызывается метод удаления элемента списка по индексу: `remove(int)`. IDE NetBeans заменяет цикл следующим образом:

```
input.stream().filter((idx) -> (idx > 0)).forEachOrdered((idx) -> {
    list.remove(idx);
});
```

При этом тип переменной `idx` меняется на `java.lang.Integer`, и уже вызывается метод удаления элемента списка, эквивалентного заданному: `remove(Integer)`. В результате изменяется семантика программы: при выполнении она выдает `[2, 3]`. Для сравнения, IntelliJ Idea предлагает заменить тот же самый цикл следующим образом:

```
input.stream().mapToInt(idx -> idx)
    .filter(idx -> idx > 0).forEach(list::remove);
```

В данном случае благодаря применению операции `mapToInt` `Stream<Integer>` превращается в `IntStream`, и изменения семантики не происходит.

Eclipse plugin

В IDE Eclipse данная функциональность планировалась к реализации с использованием плагина `Convert-For-Each-Loop-to-Lambda-Expression-Eclipse-Plugin` [6]. Плагин не был доведен до рабочего состояния (на момент 19 декабря 2018 г.), даже базовая функциональность не поддерживается.

Доступность разработки

Код проекта может быть найден на GitHub в репозитории IntelliJ Idea ⁴.

Точка входа – класс `StreamApiMigrationInspection`, он предназначен для обнаружения шаблонов кода, подходящих для преобразования. После того как подходящий кусок кода найден, будет предложен вариант миграции – объект, класс которого `BaseStreamApiMigration`.

Непосредственно преобразованием кода занимается класс `MigrateToStreamFix`. Здесь же находятся классы `StreamSource` и `Operation`, наследники которых соответствуют определению простого генератора и операции соответственно из статьи.

Классу `TerminalBlock` напрямую не соответствует терминал из модели, он обозначает фрагмент, который еще не был обработан и, возможно, содержит терминал.

Данный автоматический рефакторинг активирован только для версий Java больше 7.

Заключение

В данной статье была выстроена модель Stream API, на основании нее реализован рефакторинг для преобразования циклов в цепочки вызовов Stream API и внедрен в платформу IntelliJ Idea. Сейчас это преобразование доступно к использованию у миллионов пользователей IDE. Данный рефакторинг успешно распознает большую часть распространенных способов преобразования данных. Корректность обеспечивается сотнями интеграционных тестов и проверками на реальном программном обеспечении, в частности самом коде IntelliJ Idea.

Список литературы / References

1. Allamanis M., Barr E. T., Bird C., Devanbu P., Marron M., Sutton C. Mining Semantic Loop Idioms. *IEEE Transactions on Software Engineering*, Jul. 2018, vol. 44, no. 7, p. 651–668. DOI 10.1109/TSE.2018.2832048
2. Barua A., Cheon Y. A Systematic Derivation of Loop Specifications Using Patterns, Technical Report 15-90, Department of Computer Science, University of Texas at El Paso, El Paso, TX, December 2015.
3. Barua A., Cheon Y. Finding Specifications of While Statements Using Patterns. *New Trends in Networking, Computing, E-learning, Systems Sciences, and Engineering*, Nov. 2014, p. 581–588. DOI 10.1109/FOSE.2007.27
4. Franklin L., Gyori A., Lahoda J., Dig D. LambdaFicator: From imperative to functional programming through automated refactoring. In: 35th International Conference on Software Engineering (ICSE), May 2013. DOI 10.1109/ICSE.2013.6606699
5. Gyori A., Franklin L., Dig D., Lahoda J. Crossing the gap from imperative to functional programming through refactoring. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering – ESEC/FSE, 2013. DOI 10.1145/2491411.2491461
6. Arefin M., Khatchadourian R. Porting the NetBeans Java 8 enhanced for loop lambda expression refactoring to eclipse. In: Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity – SPLASH Companion, 2015. DOI 10.1145/2814189.2817277.

Материал поступил в редколлегию
Received
04.03.2019

⁴ <https://github.com/JetBrains/intellij-community/tree/master/java/java-impl/src/com/intellij/codeInspection/streamMigration>

Сведения об авторах / Information about the Authors

Иванов Роман Андреевич, магистрант факультета информационных технологий Новосибирского государственного университета (ул. Пирогова, 1, Новосибирск, 630090, Россия)

Roman A. Ivanov, Master's Student, Faculty of Information Technologies, Novosibirsk State University (1 Pirogov Str., Novosibirsk, 630090, Russian Federation)

roman.ivanov@jetbrains.com

Валеев Тагир Фаридович, кандидат физико-математических наук, старший научный сотрудник, Институт систем информатики имени А. П. Ершова СО РАН (пр. Академика Лаврентьева, 6, Новосибирск, 630090, Россия)

Tagir F. Valeev, Candidate of Physical and Mathematical Sciences, senior research associate of the A. P. Ershov Institute of Informatics Systems of the Siberian Branch of the Russian Academy of Sciences (6 Academician Lavrentiev Ave., Novosibirsk, 630090, Russian Federation)

amaembo@gmail.com