Отладка операций в функциональном стиле на языке Java в среде разработки IntelliJ IDEA

Бибаев Виталий Игоревич

Научный руководитель: Ушаков Егор Анатольевич

САНКТ-ПЕТЕРБУРГСКИЙ АКАДЕМИЧЕСКИЙ УНИВЕРСИТЕТ

vitaliy.bibaev@gmail.com

13.06.2017

Введение

Интерфейс потоков объектов

- *Java 8* - появление анонимных функций и *java.util.stream*
- *java.util.stream* (Stream API) - набор классов и интерфейсов для упрощения обработки последовательностей элементов с помощью функций высших порядков

```java
public static void main(String[] args) {
  final int[] result =
    Arrays.stream(new int[]{10, 87, 97, 43, 121, 20})
    .flatMap(JavaStreams::factorize)
    .distinct()
    .sorted()
    .toArray();
  // ...
}
```

- Ключевая особенность - ленивость
- Аналоги в других языках - LINQ в C#, Sequence в Kotlin

- Структура цепочки Stream API.
  - **Промежуточные операции**. Создают новый поток объектов (возможно, используя существующий).
    - Без состояния

      `flatMap(JavaStreams::factorize)`
    - С состоянием

      `distinct()`
      `sorted()`
  - **Завершающая операция**. Завершает цепочку, преобразуя поток объектов в результат.

    `toArray()`

Недостатки отладки Stream API в сравнении с обычными управляющими структурами:

- Нетривиальная последовательность исполнения
- Отсутствие знаний о промежуточных результатах вычислений
- Отсутствие информации о трансформации объектов
- Сложные стеки вызовов

```
factorize:10, JavaStreams
apply:-1, 1289479439 (JavaStreams$$Lambda$1)
accept:305, IntPipeline$7$1 (java.util.stream)
forEachRemaining:1032, Spliterators$IntArraySpliterator (java.util)
forEachRemaining:693, Spliterator$OfInt (java.util)
copyInto:481, AbstractPipeline (java.util.stream)
wrapAndCopyInto:471, AbstractPipeline (java.util.stream)
evaluate:545, AbstractPipeline (java.util.stream)
evaluateToArrayNode:260, AbstractPipeline (java.util.stream)
toArray:502, IntPipeline (java.util.stream)
main:27, JavaStreams
```

Введение

Пример использования интерфейса потоков в проекте с открытым кодом

```
final Map<String, VirtualFile> modules = StreamEx.
    of(PhpLibraryRoot.EP_NAME.getExtensions()).
    map(PhpLibraryRoot::getProvider).
    filter(PhpLibraryRootProvider::isRuntime).
    map(provider -> provider.getLibraryRoot(project)).
    flatMap(StreamEx::of).map(VirtualFile::getChildren).
    flatMap(Arrays::stream).filter(VirtualFile::isDirectory
    remove(module -> module.getName().startsWith(".")).
    collect(Collectors
      .toMap(VirtualFile::getName, identity(),
        (curr, next) -> next));

return modules.values().stream().
    filter(root -> root.findChild(".ignore") == null).
    collect(Collectors.toSet());
```

Существует расширение для отладчика Visual Studio для C# - OzCode. Одна из его функций - отладчик для LINQ. Особенности этого расширения:

- Работает с LINQ (язык интегрированных запросов). Его реализация отличается от *java.util.stream*
- Позволяет увидеть трансформации объектов и промежуточные результаты
- Закрытый исходный код
- Платный
- Только для отладчика Visual Studio

Цель и задачи

Цель: Расширить возможности отладчика для поиска ошибок при использовании библиотек с функциями высшего порядка.

Задачи:

1. Распознавание вызова Stream API возле текущей позиции отладчика.
2. Построение промежуточных состояний между вызовами в цепочке.
3. Нахождение переходов между состояниями.
4. Визуализация объектов внутри состояний и переходов для них.

Решение

Поиск вызова

Чтобы распознать вызов будем использовать AST файла. С учетом следующих особенностей:

- Цепочка может не иметь завершающей операции. Такие цепочки нам не интересны (в них нет вычислений).
- Может быть несколько подходящих вызовов. Необходимо найти их все.
- Цепочка может быть на других уровнях стека вызовов.
- Цепочка может быть в объемлющем коде.

Решение

Построение состояний и переходов

Для отладочных целей интерфейс Stream определяет метод peek. С помощью него можем запомнить объекты перед вызовом и после него, не изменив логику.

$$.peek(x \rightarrow store(x, time))$$
$$.call(...)$$
$$.peek(z \rightarrow \{time.increment()\})$$
$$.peek(y \rightarrow store(y, time))$$

В результате получим два множества:

$$Before = \{(t_i, x_i)\}, After = \{(t_i, y_i)\}$$

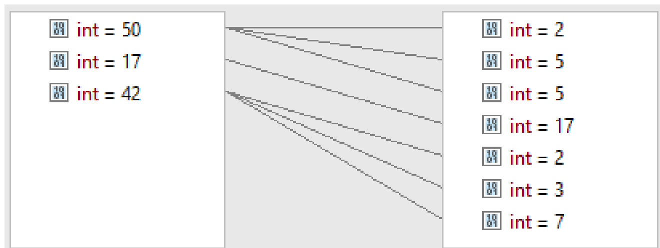
Они образуют состояния между вызовами.
Чтобы найти переходы достаточно построить **отображения**

$$(t_i, x_i) \rightarrow List[(t_j, y_j)],\ \forall (t_i, x_i) \in Before$$
$$(t_i, y_i) \rightarrow List[(t_j, x_j)],\ \forall (t_j, y_j) \in After$$

Решение
Пример

Рассмотрим в качестве примера вызов:

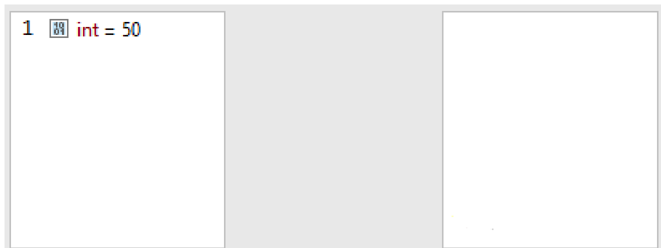
```
public static IntStream factorize(int n) {...}
flatMap(x -> factorize(x))
```



Решая такие же задачи и для остальных методов, получим для них отображение. Такой подход работает почти для всех методов. Исключение - операции с состоянием. Для них можно придумать отдельное решение.

Рассмотрим в качестве примера вызов:

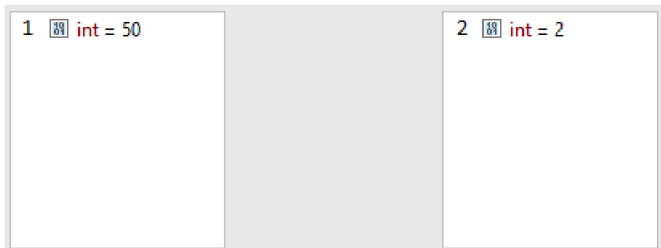
```
public static IntStream factorize(int n) {...}
flatMap(x -> factorize(x))
```



Решая такие же задачи и для остальных методов, получим для них отображение. Такой подход работает почти для всех методов. Исключение - операции с состоянием. Для них можно придумать отдельное решение.

Рассмотрим в качестве примера вызов:

```
public static IntStream factorize(int n) {...}
flatMap(x -> factorize(x))
```



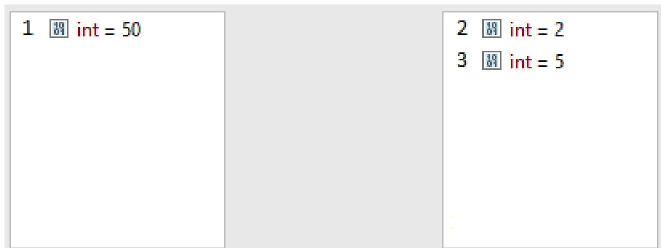
Решая такие же задачи и для остальных методов, получим для них отображение. Такой подход работает почти для всех методов. Исключение - операции с состоянием. Для них можно придумать отдельное решение.

Решение

Пример

Рассмотрим в качестве примера вызов:

```
public static IntStream factorize(int n) {...}
flatMap(x -> factorize(x))
```



Решая такие же задачи и для остальных методов, получим для них отображение. Такой подход работает почти для всех методов. Исключение - операции с состоянием. Для них можно придумать отдельное решение.

Решение

Пример

Рассмотрим в качестве примера вызов:

```
public static IntStream factorize(int n) {...}
flatMap(x -> factorize(x))
```

1 int = 50	2 int = 2
	3 int = 5
	4 int = 5

Решая такие же задачи и для остальных методов, получим для них отображение. Такой подход работает почти для всех методов. Исключение - операции с состоянием. Для них можно придумать отдельное решение.

Решение

Пример

Рассмотрим в качестве примера вызов:

```
public static IntStream factorize(int n) {...}
flatMap(x -> factorize(x))
```

1 int = 50	2 int = 2
5 int = 17	3 int = 5
	4 int = 5

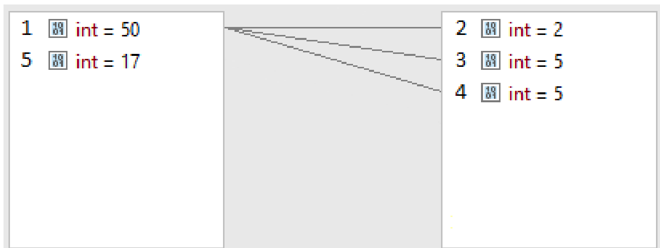
Решая такие же задачи и для остальных методов, получим для них отображение. Такой подход работает почти для всех методов. Исключение - операции с состоянием. Для них можно придумать отдельное решение.

Решение

Пример

Рассмотрим в качестве примера вызов:

```
public static IntStream factorize(int n) {...}
flatMap(x -> factorize(x))
```



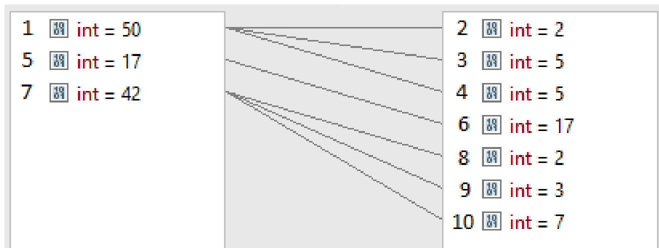
Решая такие же задачи и для остальных методов, получим для них отображение. Такой подход работает почти для всех методов. Исключение - операции с состоянием. Для них можно придумать отдельное решение.

Решение

Пример

Рассмотрим в качестве примера вызов:

```
public static IntStream factorize(int n) {...}
flatMap(x -> factorize(x))
```



Решая такие же задачи и для остальных методов, получим для них отображение. Такой подход работает почти для всех методов. Исключение - операции с состоянием. Для них можно придумать отдельное решение.

Решение

Вычисление

Пример сгенерированной цепочки для вычисления.

```
IntStream.of(1, 2).filter(x -> x % 2 == 0).sum();
```

Для данной цепочки нужно запустить следующий код:

```
// declare time, before, after
final int result = IntStream.of(1, 2)
      .peek(x -> time[0]++)
      .peek(x -> before.put(time[0], x))
    .filter(x -> x % 2 == 0)
      .peek(x -> time[0]++)
      .peek(x -> after.put(time[0], x))
    .sum();
// return new { new { result } , before, after };
}
```

Возможности для вычисления этого кода предоставляются средой разработки.

Результаты

- Разработан плагин, упрощающий отладку операций над потоками объектов.
  - Поддержаны все операции в стандартной реализации *java.util.stream*.
  - Список поддерживаемых методов может быть быстро расширен.
  - Плагин размещен в открытом доступе.

Результаты

https://plugins.jetbrains.com/plugin/9696-java-stream-debugger

Исходный код доступен по ссылке:
https://github.com/Roenke/stream-debugger-plugin

Архитектура отладчика

Выражение для вычисления

Чтобы вычислить выражение для отслеживания исполнения цепочки потоков объектов нужно определить класс, а так же учесть следующие особенности

- Поля и методы этого класса.
- Расположение класса: пакет, объемлющий класс.
- Доступ к полям и методам объемлющего класса.
- Доступ к приватным членам класса из лямбд и анонимных классов.
- Минимизация дальнейших обращений к виртуальной машине.

```
private static class Generated extends MagicAccessorImpl {
  public Object eval(Collection<Integer> context) {
    // Вычисление выражения
    // Подготовка результата
  }
}
```

Гарантии библиотеки

Поток объектов всегда ленивый. Это значит, что объекты не из источника будут браться только когда выполняется терминальная операция и ей нужен объект.

- Это исключает ситуации, когда из каких-то промежуточных операций требовались объекты, но затем не использовались
- Но это не исключает, что некоторым промежуточным операциям потребуется прочитать более одного объекта, чтобы вернуть один. (см sorted, distinct, и др)
- Следствие: нет вызова терминальной операции - нет вычислений

Требования к коду пользователя

"Streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed."

- Это значит, что объекты могут браться из источника только по необходимости.
- Но это не исключает, что некоторым промежуточным операциям потребуется прочитать всё объекты. (см sorted, distinct, и др)

Распознавание вызова

```
33    private static List<Long> ambiguous() {
34        return Stream.of(Stream.of(1, 2).count())
35            .collect(Collectors.toList());
36    }
37 }
38
```

Multiple chains found

- Stream.of -> count -> of -> collect
- Stream.of -> count

OzCode

```
var passed = StudentRepository    ≤ 1ms elapsed
                HelloLinq.Department
    .GetAllDepartments() 2/2
    .SelectMany(d => d.Students) 12/12
                HelloLinq.Stud...
    NullReferenceException will occur
    .Where(args => args.Info.Grade >= 60) 5/5
                "System.NullReferenceException"
    .Select(args => args.Name) ⊘/4
    .Count() 0 ;
```

12 › Where › 5

- [0] Name: "Barbara Montgomery" ✗
- [1] Name: "Ivory Benjamin" ✓
- [2] Name: "Chancellor Levine" ✗
- [3] Name: "Nicholas Barry" ✗
- [4] Name: "Carissa Ball" ✓
- [5] Name: "Rama Carney" ✗
- [6] Name: "Nadine Rollins" ✓
- [7] Name: "Cameron Pollard" ✗
- [8] Name: "Juliet Hopkins" ✓
- [9] Name: "Charissa Levine" ✗
- [10] Name: "Brett Bryan" ✗
- [11] Name: "Zahir Short" ⚠

Search:

Пример

Рассмотрим пример. Поставим задачу восстановить промежуточные состояния и переходы.

```
List<String> streamAPIExample(List<Person> persons) {
  return persons.stream()
      .filter(person -> person.age < 18)
      .sorted(Comparator.comparing(x -> x.age))
      .map(person -> person.name)
      .collect(Collectors.toList());
}
```

Пример

Интерфейс Stream определяет для отладочных целей метод peek. Добавим его между всеми вызовами методов Stream<T>.

```
List<String> streamAPIWithPeeks(List<Person> persons) {
  return persons.stream()
       .peek(x -> System.out.println(x))
       .filter(person -> person.age < 18)
       .peek(x -> System.out.println("after filter: " + x))
       .sorted(Comparator.comparing(x -> x.age))
       .peek(x -> System.out.println("after sort: " + x))
       .map(person -> person.name)
       .peek(x -> System.out.println("after map: " + x))
       .collect(Collectors.toList());
}
```

Пример

Информация, которую можно извлечь:

- Последовательность прохождения объектов через цепочку вызовов
- Результат фильтрации
- sorted имеет состояние и требует выполнить весь поток до своего вызова
- Преобразования объектов

Результат вызова:

```
1: 1 - Vasily [10]
2: after filter: 1 - Vasily [10
3: 2 - Petr [15]
4: after filter: 2 - Petr [15]
5: 3 - Dmitry [14]
6: after filter: 3 - Dmitry [14
7: 4 - Sasha [20]
8: 5 - Michael [33]
9: after sort: 1 - Vasily [10]
10: after map: Vasily
11: after sort: 3 - Dmitry [14]
12: after map: Dmitry
13: after sort: 2 - Petr [15]
14: after map: Petr
```