Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Бибаев Виталий Игоревич

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

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
ООО ИнтеллиДжей Лабс, разработчик платформы IntellJ Ушаков Е. А.

Рецензент:
ООО ИнтеллиДжей Лабс, Руководитель проекта Scala plugin Тропин Н. В.

Санкт-Петербург
2017

SAINT-PETERSBURG ACADEMIC UNIVERSITY
Higher education centre

Department of Mathematics and Information Technology

Vitaliy Bibaev

Debugging of functional-style operations in Java in the IntelliJ IDEA IDE

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Alexander Omelchenko

Scientific supervisor:
Software Developer in IntelliJ Platform at IntelliJ labs Egor Ushakov

Reviewer:
Software Developer in Scala plugin at IntelliJ labs Nikolay Tropin

Saint-Petersburg
2017

Оглавление

Введение

В настоящее время объектно-ориентированные языки программирования заимствуют некоторые концепции функциональных языков. Обычно это позволяет расширить синтаксис языка, упростить написание некоторых операций а так же сделать код программ более выразительным. Одним из основных понятий функциональных языков является функция высшего порядка – функция, принимающая в качестве аргумента другие функции или возвращающая их в качестве результата своей работы. Примерами языков, активно использующих функции высшего порядка являются C#, Scala, Kotlin и другие. В восьмой версии Java появилась возможность использовать анонимные функции, и был добавлен пакет java.util.stream в стандартную библиотеку, этот пакет содержит набор классов и интерфейсов, позволяющих обрабатывать потоки объектов при помощи функций высшего порядка. Часто можно встретить альтернативное название данного пакета - Stream API. В данной работе также будем использовать русскоязычный термин – интерфейс потока объектов.

Отладка программы - это один из этапов разработки программного обеспечения, в ходе которого разработчик находит и исправляет ошибки. Для отладки программ существует много подходов. Один их них – использование отладчика – отдельной программы, позволяющей управлять процессом исполнения другой программы, просматривать и модифицировать её внутреннее состояние.

В случае java добавление анонимных функций привело к созданию нового способа написания кода для обработки последовательностей объектов. К сожалению, появление новых конструкций языка не привело к появлению новых средств для их отладки. Поэтому одним из наиболее заметных недостатков использования Stream API является сложность процесса поиска ошибок.

В рамках данной работы разработано расширение для среды разработки IntelliJ IDEA, позволяющее упростить процесс отладки кода с использованием функций высшего порядка. В главе 1 представлен обзор имеющихся возможностей по отладке кода на языке Java, разобраны особенности использования классов из пакета java.util.stream, а также перечислены трудности, возникающие при отладке программ, которые используют эти классы. Также в первой главе рассмотрены попытки решения проблемы для других языков. В главе 2 приводится обоснование предположений, сделанных о пользовательском коде, описывается основные идеи и алгоритмы, которые были использованы при решении задачи. В главе 3 перечисляются особенности реализации и используемые возможности среды разработки, объясняются принятые архитектурные решения, приводится обзор пользовательского интерфейса.

1. Обзор литературы

В процессе разработки программ программист неизбежно допускает ошибки. Для их обнаружения и исправления существует немало подходов: написание автоматических тестов, ручное тестирование, сохранение некоторой информации в лог, отладка и многие другие. Как правило, используется сразу несколько. В данной работе нас будет интересовать поиск ошибок при помощи отладчика.

Определение 1.1. Отладчик – компьютерная программа, предназначенная для поиска ошибок в других программах, ядрах операционных систем, SQL-запросах и других видах кода. Отладчик позволяет выполнять трассировку, отслеживать, устанавливать или изменять значения переменных в процессе выполнения кода, устанавливать и удалять контрольные точки или условия остановки и т.д.[16]

Отладчик позволяет просматривать состояние исполняемой программы:

- Текущую инструкцию
- Стек вызовов
- Значения локальных переменных
- Состояние памяти
- Потоки (состояние и стеки)

И выполнять действия:

- Добавление точки останова
- Переход к следующей инструкции/внутрь функции/месту вызова текущей функции
- Установка указателя на текущую инструкцию (IP)
- Модификация значений в памяти
- Вычисление выражений
- и другие

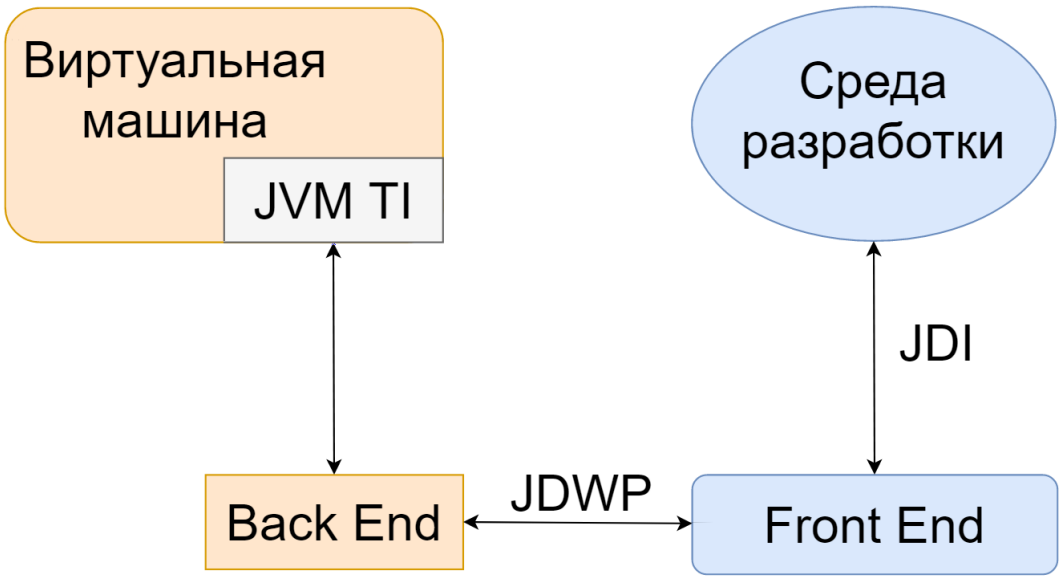
1.1. Отладка Java

Java исполняется на виртуальной машине JVM, поэтому средства для отладки предоставляет JVM.

Вместе с JDK (Java Development Kit) поставляется отладчик - jdb [6]. Это очень простой отладчик с интерфейсом командной строки, его цель – демонстрация части возможностей платформы Java по отладке программ (JPDA - Java Platform Debugger Architecture), поэтому для отладки реальных проектов он не подходит. Вместо него используются отладчики внутри сред разработки, которые так же используют JPDA.

1.1.1. Архитектура JPDA

Упрощенно схема работы JPDA выглядит следующим образом



В блоках ”Front End” и ”Back End” скрыты детали реализации по обмену сообщениями – очереди сообщений и потоки, обрабатывающие эти сообщения.

JPDA состоит из трех частей:

- JVM TI – Java VM Tool Interface - описывает сервисы для отладки, которые предоставляет виртуальная машина.
- JDWP – Java Debug Wire Protocol - протокол обмена сообщениями между отладчиком и виртуальной машиной. Описывает формат сообщений между отладчиком и JVM, создавая дополнительную абстракцию и позволяя использовать произвольные технологии для написания отладчиков.
- JDI – Java Debug Interface - высокоуровневый Java интерфейс для взаимодействия с виртуальной машиной. Поставляется вместе с JDK, входит в пакет com.sun.jdi. Позволяет получить доступ к состоянию программы, а также кон-

тролировать ход её исполнения. Содержит набор классов-посредников для сущностей отлаживаемой виртуальной машины (потоки, объекты, классы и т.д.). Эти классы позволяют вызывать методы, получать и модифицировать значения полей, получать уведомления о событиях, приостанавливать и возобновлять потоки исполнения.

1.2. Функции высших порядков

В 2014 году вышла восьмая версия языка программирования Java. Одним из нововведений обновления стало появление поддержки анонимных функций и библиотеки функций высшего порядка для обработки последовательностей элементов – пакет java.util.stream. Кроме добавления этого пакета изменения коснулись и ранее определённых классов – для некоторых объектов естественно представление в виде последовательности элементов: коллекции и поток ввода.

Поддержка анонимных функций сделало доступным следующий синтаксис.

```
final Predicate<Integer> isOdd = x -> x % 2 == 1;
```

После этого данную функцию можно вызвать:

```
isOdd.test(10);
```

С точки зрения пользователя, такое определение анонимной функции является более короткой версией следующего использования анонимного класса:

```
final Predicate<Integer> isOdd = new Predicate<Integer>() {
      @Override
      public boolean test(Integer x) {
        return x % 2 == 1;
      }
    };
```

Вместе с классами из пакета java.util.stream анонимные функции позволяют создавать следующие конструкции:

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

Данный вызов возвращает отсортированный по возрасту список имен людей, чей возраст меньше 18 лет. Последовательность операций над объектами Stream будем называть **цепочкой**. С точки зрения результата, такой вызов эквивалентен следующей последовательности операторов:

```
public List<String> cycles(List<Person> persons) {
  persons = new ArrayList<>(persons);
  final Iterator<Person> iterator = persons.iterator();
  while (iterator.hasNext()) {
    final Person person = iterator.next();
    if (person.age >= 18)
      iterator.remove();
  }

  persons.sort((p1, p2) -> Integer.compare(p1.age, p2.age));

  final ArrayList<String> names = new ArrayList<>();
  for (final Person person : persons) {
    names.add(person.name);
  }

  return names;
}
```

Сравнивая два способа, можно отметить, что Stream API избавил от деталей реализации, таких как использование итератора для обхода списка и использование промежуточных коллекций. Таким образом, использование Stream API избавило программиста от написания простого и часто повторяющегося кода, в котором тем не менее можно допустить ошибку.

Важно, что последовательность вызовов в цепочке нельзя воспринимать как трансформацию коллекций. Правильно трактовать подобные вызовы именно как поток объектов. Это значит, что из источника берется объект, и затем он последовательно проходит через все операции, пока это возможно, затем данные действия повторяются и для всех остальных объектов. Очевидно, возможны ситуации, когда все элементы из источника не понадобятся. Операции, для которых это верно, называются короткозамкнутыми. Как следствие, допускаются бесконечные источники, но тогда цепочка методов должна содержать хотя бы одну короткозамкнутую операцию, иначе такой вызов никогда не завершится.

Таким образом, пакет java.util.stream предоставляет возможность отказаться от обычных управляющих конструкций в пользу операций в функциональном стиле.

Обычно это приводит к краткости кода, скрывая от программиста детали реализации операций, оставляя лишь их семантику.

1.2.1. Части вызова Stream API

Типичное использование классов из java.util.stream состоит из нескольких этапов: сначала нужно инициализировать поток объектов, затем выполнить над объектами набор преобразований, после чего нужно агрегировать объекты в результирующее значение, поэтому операции можно разделить на два класса:

- **Промежуточная операция.** Операция, которая возвращает Stream. Может инициировать поток объектов из источника или преобразовать уже существующий (далее будем называть такой поток входным). После вызова таких операций создается новый объект Stream. Примеры:
  - `map` – заменяет каждый объект новым значением, сохраняя порядок;
  - `filter` – оставляет только те объекты, которые удовлетворяют некоторому условию;
  - `distinct` – оставляет только различные элементы (сравнение по *equals* [10]);
  - и многие другие (см [12]).

  Промежуточные операции делятся на две части согласно возможности хранить состояние:
  - Без состояния. Таких операций большинство, они выполняют локальные действия над одним объектом. Для данного вида операций гарантируется, что объекты обрабатываются независимо. Иными словами, чтобы произвести новый элемент потока, таким операциям не нужно вынимать более одного элемента из входного потока;
  - С состоянием. Такие операции обрабатывают поток не независимо. Они могут считать часть входного потока (возможно, прочитать его целиком), прежде чем вернуть значения после себя. В стандартной реализации таких операций две - sorted и distinct.
- **Завершающая операция.** Вызывается на заранее созданном объекте Stream. Запускает вычисления, преобразуя объекты из потока в некоторый результат – произвольный объект. Как правило, это коллекции, массивы, определенные значения из потока (максимум/минимум по какому-либо признаку) и другие. Примеры:
  - `toArray` – сохраняет все объекты из потока в массив;

– `max` – находит среди объектов в потоке максимум по заданному критерию сравнения;

– и многие другие (см [12]).

Кроме того, можно выделить объект - источник. Он может быть рассмотрен как последовательность объектов. В качестве источника могут выступать коллекции, массивы, функция-генератор, поток ввода.

```
Arrays.stream(new int[] {1, 2, 3});
```

Массив, переданный параметром методу Arrays.stream, является источником.

```
Stream.generate(() -> 0.);
```

Данный вызов создает бесконечный поток нулей. Анонимная функция-генератор является источником. Любой объект может быть рассмотрен как поток объектов (из одного объекта):

```
Stream.of(object)
```

1.2.2. Детали реализации Stream API

В своей реализации потоки объектов в java используют абстракцию Spliterator [11], так же введенную в восьмой версии java. Spliterator - это интерфейс, который усложняет понятие обычного итератора – объекта, позволяющего обойти некоторый набор объектов [9]. Spliterator хранит свойства перечисляемых объектов: их количество, упорядоченность, изменяемость, признак того, что данные уже отсортированы и другие. Эти свойства могут быть использованы при выполнении операций над потоком объектов. Так же в Spliterator добавлен новый метод split, который позволяет разделить множество объектов на две непересекающиеся части и обойти параллельно. На этой идее основывается реализация параллельных потоков.

Несмотря на то, что java является объектно-ориентированным языком, не все значения в java являются объектами. Исключения составляют примитивные типы, для которых хранение в виде объектов невыгодно с точки зрения производительности. По этим же причинам, реализация java.util.stream содержит отдельную специализацию для объектов примитивных типов – IntStream, LongStream, DoubleStream – для потоков целых и вещественных чисел.

1.2.3. Расширения стандартной библиотеки

Стандартная библиотека предоставляет довольно большой набор промежуточных и завершающих операций. Тем не менее существуют расширения базового интерфейса. Примерами таких расширений служат библиотеки - jOOL [22] и StreamEx [21].

Они полностью совместимы со стандартной реализацией и дают те же гарантии. Операции, добавленные в этих библиотеках, как правило, упрощают написание кода с использованием интерфейса потоков объектов. Хотя многие из новых операций можно выразить через уже имеющиеся.

1.2.4. Аналоги в других языках

Идея обрабатывать множества объектов, как последовательность элементов, в императивных языках не нова, – в 2007 году Microsoft представил LINQ (язык интегрированных запросов) для языка *C#* [19]. Он предоставляет очень схожие возможности с java.util.stream, но дизайн решений различен. В реализации LINQ используются возможности языка C#, отсутствующие в Java: методы-расширения [4], а так же ключевое слово yield [5], в то время как реализация java основана на абстракции Spliterator (см 1.2.2). Кроме того, в расширении от Microsoft, операции названы по аналогии с SQL, в java же операции имеют аналоги в функциональных языках, например, в Haskell.

Подобные возможности реализуют коллекции в языке Scala [3] и абстракция Sequence в языке Kotlin [1].

1.3. Отладка операций в функциональном стиле

После появления функций высшего порядка и расширения стандартной библиотеки, возможности JPDA остались прежними. Таким образом, с одной стороны, имеется способ записывать операции над множествами объектов в новом синтаксисе, с другой стороны, отсутствуют удобные средства их диагностики.

Рассмотрим возможности отладчиков трех крупнейших сред разработки (Eclipse, NetBeans, IntelliJ IDEA), которые могут быть полезны при отладке цепочек методов, оперирующих потоком объектов.

- Использование точек останова (с условием). Допускается, что точка останова будет внутри тела анонимной функции.
- Последовательные переходы по вызовам анонимных функций внутри цепочки.
- Вычисление выражений с анонимными функциями (доступно только в IDEA).

1.3.1. Недостатки имеющихся подходов

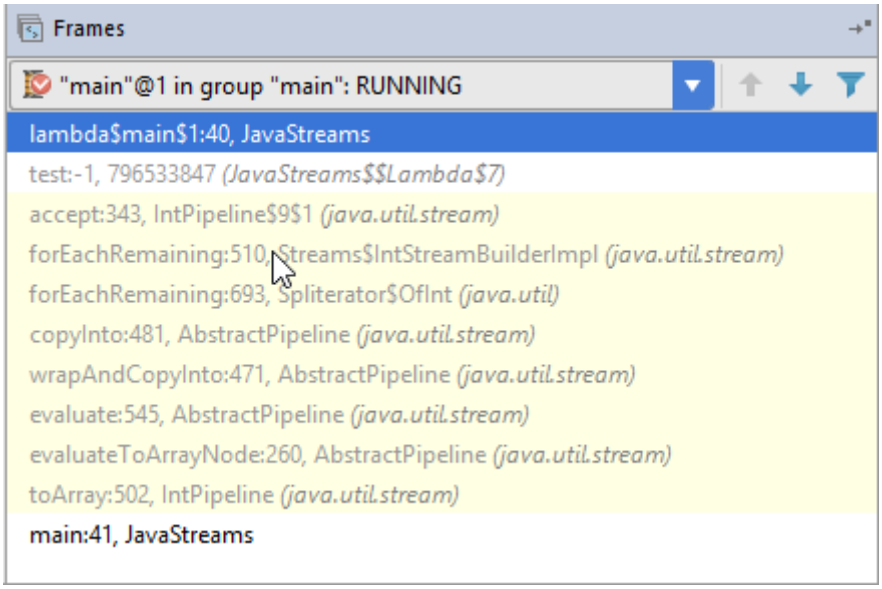
Описанные методы обладают недостатками:

- При использовании точек останова внутри анонимной функции у нас есть только информация о текущем вызове и объект. При этом нельзя понять, почему

именно этот объект появился в данном месте, и какие трансформации он претерпел. Логичное предположение, что эту информацию можно восстановить в одном из элементов стека вызовов, окажется неверным. Стек вызовов состоит из большого числа внутренних вызовов, и нет гарантий, что получится найти внутри него что-то полезное. Ниже приведен пример стека для достаточно простого вызова:

```
IntStream.of(1)
    .filter(x ->  {
        return x % 2 == 1; /* Breakpoint at this line */
    })
    .toArray();
```

И стек вызовов в точке останова внутри анонимной функции:



- При переходах по анонимным функциям внутри операций цепочки, возникает следующая проблема: для того, чтобы отладчик мог остановиться внутри тела анонимной функции, нужно чтобы эта функция была в исходном коде. Но не все вызовы предполагают передачу анонимной функции в качестве параметра. Поэтому такой подход применим не для всех цепочек. По этой же причине переходы по последовательным вызовам анонимных функций не всегда возможны.

  Пример кода, когда применение точек останова затруднительно:

  ```
  collection.stream().distinct().sorted().toArray();
  ```

- Частичное исполнение цепочки с сохранением в промежуточную коллекцию.

  Пример:

  ```
  collection.stream().distinct().sorted().toArray();
  ```

Дополнительно можно вычислить два выражения:

```
collection.stream().toArray();
```

```
collection.stream().distinct().toArray();
```

В результате у нас есть данные о промежуточных состоянии, которые могут помочь найти ошибку.

Недостатки:

- Не все вызовы Stream API допускают частичное исполнение. Бесконечные потоки с короткозамкнутыми операциями не позволяют исполнить их частично,– приведенный ниже вызов не завершится

  ```
  Stream.iterate(1, i -> i + 1).toArray();
  ```

  Для его завершения необходимо добавить дополнительные операции, такие как limit, подбирая необходимые параметры:

  ```
  Stream.iterate(1, i -> i + 1).limit(100/* 200, 300 , ? */).toArray();
  ```

- Не позволяет отследить историю трансформаций объекта;
- Доступно не во всех средах разработки и не для всех анонимных функций.

Кроме описанных способов можно использовать и другие, но они также имеют недостатки:

- Интерфейс Stream определяет специальный метод peek, который может упростить отладку. Назначение этого метода – извлечь каждый объект из потока (не изменив логику и последовательность работы потока), и совершить в ним какое-то действие, полезное для отладки. Примером такого действия может быть печать в лог.

  Недостатки:

  - Необходимо модифицировать код.
  - Не все объекты можно удобно представить в виде строки.
  - Вычисление потока происходит лениво, поэтому понять получившийся лог будет не так просто.

- Некоторые среды разработки позволяют автоматически преобразовать некоторые цепочки методов Stream API в эквивалентный код с циклами.

  Недостатки:

  - Не каждая цепочка может быть автоматически преобразована.

– Необходимо модифицировать код.

– Ошибка может пропасть после трансформации. Кроме того, могут появиться новые ошибки из-за неправильного преобразования.

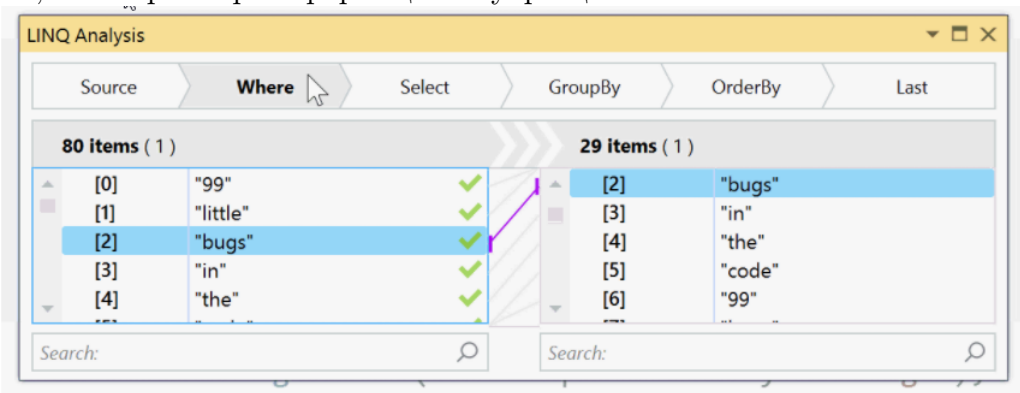
– Обратный переход от циклов к Stream API скорее всего невозможен (на данный момент среды разработки умеют конвертировать лишь наиболее тривиальные циклы в вызовы Stream API).

Таким образом, нет удобного способа, который бы позволил увидеть, как трансформируются объекты внутри потока.

1.3.2. Решение для C#. OzCode

В языке C# имеются аналогичные проблемы. Но существует продукт, упрощающий отладку цепочек LINQ методов. Он называется OzCode. Это платное расширение Visual Studio [20] с закрытым исходным кодом. В конце 2016 года в нём появилась новая функция - отладчик LINQ [14].

OzCode позволяет узнать объекты, которые проходят через каждый вызов в цепочке, и историю трансформаций внутри цепочки вызовов.



К сожалению, исходный код OzCode закрыт, расширение работает только с LINQ внутри отладчика Visual Studio. Поэтому нам не удалось понять как он работает.

1.4. Постановка задачи

Цель данной работы – расширить возможности встроенного в среду разработки IntelliJ IDEA отладчика для упрощения отладки операций в функциональном стиле с использованием Stream API.

Для того, чтобы достичь этой цели, поставим следующие задачи:

- Определить вызовы с использованием операций в функциональном стиле среди доступных из текущей позиции отладчика;
- Получить информацию о трансформациях каждого объекта в потоке объектов;

- Визуализировать результаты.

2. Подход к решению задачи

Прежде чем приступить к описанию решения, необходимо понять, какие гарантии предоставляет Stream API, а так же какие предположения можно сделать о пользовательском коде.

2.1. Гарантии Stream API

Использование интерфейса потоков позволяет абстрагироваться от деталей часто используемых операций, описывая не то, **как** нужно делать операцию, а то, **что** должно быть сделано. Такой подход открывает большую свободу для внутренней реализации этих операций. Тем не менее, снаружи это должно всегда выглядеть одинаково и работать ожидаемо. Для этого существуют гарантии, которые должны выполняться для любой реализации интерфейса Stream.

Основная гарантия предоставляемая Stream API, – промежуточные операции всегда ленивые. Это значит, что операция не будет запрашивать следующий элемент, если может корректно вычислиться без его использования.

- Это исключает ситуации, когда в промежуточных операциях из входного потока вынимались объекты, но затем не использовались.
- Но это не исключает, что некоторым промежуточным операциям потребуется прочитать более одного объекта, чтобы что-либо вычислить. (sorted, distinct)
- Следствие: нет вызова завершающей операции – нет вычислений.

2.2. Требования к коду пользователя

Кроме предоставления гарантий, Stream API делает предположения о коде пользователя. Соблюдение следующих требований снизит вероятность неожиданного поведения вызовов Stream API. Эти требования особенно критичны в случае использования параллельных потоков.

- Операции над объектами не могут модифицировать объект-источник.

  ```
  // во время исполнения произойдёт java.util.ConcurrentModificationException
  List<Integer> ints = new ArrayList<>(Arrays.asList(1,2,3));
  ints.stream().filter(x -> x % 2 == 1).forEach(i -> ints.add(i));
  System.out.println(ints);
  ```

  Исправленная версия:

```
List<Integer> ints = new ArrayList<>(Arrays.asList(1,2,3));
Collection<Integer> tmp = ints.stream()
    .filter(x -> x % 2 == 1)
    .collect(Collectors.toList());
ints.addAll(tmp);
System.out.println(ints);
```

- Функции над объектами в потоке не должны иметь состояния (когда это возможно).

```
List<Integer> ints = Arrays.asList(1, 2, 0, 3, 0/*, ...*/);
int[] zeroCount = new int[1];
final long total = ints.stream().peek(x -> {
  if (x == 0) {
    zeroCount[0]++;
  }
}).count();
System.out.println(zeroCount[0] + " zeros. Total count = " + total);
```

Исправленная версия:

```
List<Integer> ints = Arrays.asList(1, 2, 0, 3, 0/*, ...*/);
long zeroCount = ints.stream().filter(x -> x == 0).count();
System.out.println(zeroCount + " zeros. Total count = " + total);
```

- Однажды созданный объект Stream может вызвать лишь одну терминальную операцию. Если это требование нарушено, то произойдет исключение.

```
// Во время исполнения произойдет исключение IllegalStateException
public long repeatCount(Stream<?> stream) {
  final long total = stream.count();
  final long distinct = stream.distinct().count();

  return total - distinct;
}
```

Исправленная версия:

```
public long repeatCount(Stream<?> stream) {
  Collection<?> collection = stream.collect(Collectors.toList());
  return collection.size() - collection.stream().distinct().count();
}
```

2.3. Параллельные потоки объектов

Потоки объектов могут использовать параллелизм для оптимизации производительности. Для того, чтобы некоторый вызов исполнялся с использованием нескольких потоков, достаточно вызвать метод parallelStream у источника, либо метод `parallel` у уже имеющегося объекта `Stream`. Пример:

```
final Stream<Integer> sequenceStream = collection.stream();
final Stream<Integer> parallelStream1 = collection.parallelStream();
final Stream<Integer> parallelStream2 = sequenceStream.parallel();
```

При запуске вычислений в параллельном потоке, исходный поток разбивается на множество мелких задач, которые выполняются при помощи Fork/Join Pool [7], работающий по принципу work-stealing [18]. При этом гарантии могут нарушаться – промежуточные операции могут быть уже не всегда ленивыми. Это связано с желанием уменьшить необходимую синхронизацию между потоками, поэтому могут выполняться избыточные действия.

Мы не будем ставить задачу научиться отлаживать параллельные потоки. Как правило, при их использовании возникают совершенно другие проблемы и для их решения используются подходы, отличные от использования отладчика. Основная цель параллельных потоков заключается в том, чтобы логика, работающая корректно последовательно, работала быстрее. Поэтому разумно оставить возможность отлаживать такие вызовы как последовательные.

Для того, чтобы избавиться от параллелизма можно после каждого вызова, который делает поток параллельным, добавить вызов `sequential`, тогда поток будет всюду последовательным. Но в текущей реализации поток не может одновременно содержать однопоточные и многопоточные части, поэтому, чтобы сделать его полностью последовательным, будет достаточно добавить вызов `sequential` перед завершающим вызовом. Это обусловлено дизайном библиотеки, – если завершающая операция запускает все вычисления, то и она "принимает решение" сколько потоков использовать.

2.4. Определение подходящего вызова

При отладке программ внутри среды разработки пользователь видит текущее положение программы относительно исходного кода и текущее состояние программы (локальные переменные, объекты и их поля, потоки, стек и другие). Виртуальная машина Java исполняет инструкции байт-кода, заменяя их набором машинных команд. Можно считать, что существует способ отображения инструкций байт-кода на исходный код, реализуемый средой разработки, и благодаря которому, мы можем говорить о текущем положении программы не в терминах машинных комманд или java

байт-кода, а в терминах исходного java кода. Поэтому, под позицией отладчика будем понимать строку в исходном коде.

Прежде чем начать отладку цепочки Stream API, необходимо определить её границы и понять, достаточно ли данных для её вычисления.

В отладчике для каждого из потоков всегда определено текущее положение указателя инструкций. Определим все положения указателя инструкций относительно цепочки Stream API, когда все необходимые данные для вычисления этой цепочки уже заданы.

При вызове цепочки в ней могут участвовать объявленные ранее переменные и значения полей. Таким образом, отлаживать вызов, который находится значительно позже, чем текущая инструкция, не имеет смысла (возможно, имеющихся данных будет недостаточно). Текущая строка отладчика отмечена стрелкой "=>". Интересующий нас вызов – выражение, инициализирующее переменную `transformed`.

```
public static Collection<Object> example(Collection<Object> collection) {
=>  int skipBefore = 1;
    int limitValue = 10;

    final List<Object> transformed = collection.stream()
        .map(x -> doSomeStuff(x))
        .skip(skipBefore)
        .limit(limitValue)
        .collect(Collectors.toList());

    collection.addAll(transformed);
    return collection;
}
```

Если же все инструкции до вызова выполнены, значит все переменные и поля были инициализированы и можно начать его отладку.

```
public static Collection<Object> example(Collection<Object> collection) {
    int skipBefore = 1;
    int limitValue = 10;

=>  final List<Object> transformed = collection.stream()
        .map(x -> doSomeStuff(x))
        .skip(skipBefore)
        .limit(limitValue)
        .collect(Collectors.toList());
```

```
    collection.addAll(transformed);
    return collection;
}
```

Начинать отладку можно и после начала исполнения цепочки, т.к. мы предположили, что вызов не имеет побочных эффектов, которые изменят окружение.

```
public static Collection<Object> example(Collection<Object> collection) {
    int skipBefore = 1;
    int limitValue = 10;

    final List<Object> transformed = collection.stream()
        .map(x -> doSomeStuff(x))
        .skip(skipBefore)
        .limit(limitValue)
        .filter(x -> {
=>          return x.isValid();
        })
        .collect(Collectors.toList());

    collection.addAll(transformed);
    return collection;
}
```

После того, как вызов завершен, он больше не подходит для отладки, потому что окружение может измениться, изменив начальную семантику вызова. В примере ниже цепочка для инициализации переменной `transformed` больше не подходит для отладки – её повторное исполнение приведет к другому результату, потому что переменная `limitValue` получила новое значение.

```
public static Collection<Object> example(Collection<Object> collection) {
    int skipBefore = 1;
    int limitValue = 10;

    final List<Object> transformed = collection.stream()
        .map(x -> doSomeStuff(x))
        .skip(skipBefore)
        .limit(limitValue)
        .collect(Collectors.toList());
```

```
    limitValue++;
=>  collection.addAll(transformed);
    return collection;
}
```

Таким образом, отладка допустима, если текущее положение отладчика находится между непосредственным началом вызова потока и до его завершения.

2.4.1. Цепочки в других элементах стека вызовов

В процессе отладки пользователь может ставить точки останова внутри произвольных функций. После попадания на точку останова пользователь может изменять текущий элемент стека вызова. При переключении элемента стека изменяется и положение отладчика – он будет находиться на вызове какого-либо метода. Ничего не мешает этому вызову находиться внутри цепочки Stream API. Вероятно, пользователь может захотеть запустить эту цепочку для отладки. И это должно быть возможно, т.к. согласно 2.4 отладчик находится внутри цепочки (просто несколькими уровнями выше по стеку вызовов), следовательно, такая цепочка подходит для отладки.

```
private static int method(int x) {
=>  return x * x;
}

private static void nestedExample() {
**  IntStream.of(1, 2).map(JavaStreams::method).toArray();
}
```

В примере выше отладчик попал в точку останова внутри метода `method`. Среди элементов на стеке вызовов будет метод `nestedExample`. При переключении отладчика на этот элемент, позиция отладчика будет установлена внутри тела метода `nestedExample`. На этой же строке находится и цепочка. Т.к. положение отладчика сейчас внутри цепочки, то она доступна для отладки.

2.4.2. Неоднозначные вызовы

Из одного и того же положения отладчика может быть доступно сразу несколько подходящих вызовов Stream API:

- Внутри одного арифметического выражения;

```
=>  long total = IntStream.iterate(1, i -> i + 1).limit(100).sum() +
        IntStream.iterate(1, i -> i + 1).map(i -> i * i).limit(100).sum();
```

- Аргументы некоторого вызова;

```
    int[] values1 = ...;
    int[] values2 = ...;

=>  Math.max(Arrays.stream(values1).sum(), Arrays.stream(values2).sum());
```

- Параметр другого вызова Stream API;

```
=>  LongStream.of(IntStream.of(1,2,3).sum()).count();
```

- Вызов находится в объемлющем коде;

```
    long count = LongStream.of(1, 2, 3).map(x -> {
=>      return x * 2;
    }).count();
```

- В одной цепочке методов есть несколько последовательных цепочек с терминальной операцией;

```
=>  collection.stream().map(x -> x * x).collect(Collectors.toList())
        .stream().map(x -> x * x).collect(Collectors.toList());
```

Все такие цепочки подходят для отладки и их необходимо уметь обнаруживать.

2.4.3. Незавершенные цепочки

Согласно 2.1 цепочки методов Stream API без терминальной операции не запускают вычисление, значит и отлаживать их не нужно.

2.4.4. Присваивание объекта потока в переменную

Поскольку Stream API это просто набор классов, а не специальный синтаксис, то цепочки создают обычные объекты, с которыми можно обращаться как с любыми другими объектами. В этом контексте интерес представляет операция присваивания в переменную.

```
final Stream<Integer> even = collection.stream().filter(x -> x % 2 == 0);
final long size = even.count();
```

Ранее мы рассматривали цепочки методов Stream API, как нечто неделимое, но теперь видим, что это не так. Причем выражение, инициализирующее переменную `even`, не содержит завершающей операции, а значит, это выражение нельзя отлаживать (см 2.4.3).

С другой стороны, выражение `even.count()` нельзя запустить повторно.

Этот случай достаточно прост, и можно решить, что можно посмотреть, как инициализируется переменная `even`, затем объединить инициализацию с терминальной операцией, получив цепочку с завершающей операцией, не упустив промежуточные:

```
final long size = collection.stream().filter(x -> x % 2 == 0).count()
```

Но в общем случае это решение не сработает. Этот пример слишком прост, и на практике могут встретиться и более сложные случаи:

```
Stream<Integer> even = collection.stream();
even = filterEven ? even.filter(x -> x % 2 == 0) : even.filter(x -> x % 2 == 1);
final long size = even.count();
```

В этом случае тем же способом уже не справиться (хотя этот случай выглядит более близким к практике). Поэтому для таких вызовов будем находить лишь ту часть, которая относится только к последней части цепочки (с терминальной операцией). Заметим, что это инвалидирует поток в переменной even, поэтому лучше предупреждать об этом пользователя.

2.5. Построение состояний и переходов

В 2.2 описаны требования к коду пользователя, который использует Stream API. Эти требования позволяют сделать предположение, что вызов не имеет побочных эффектов и его можно повторить несколько раз, получив тот же самый результат.

Основная идея состоит в том, чтобы запустить вычисление модифицированного вызова Stream API, результат которого совпадет с исходным, при этом новая цепочка собирает полезную для отладки информацию.

2.5.1. Выражение для сбора отладочной информации

Для того, чтобы собрать информацию о том, какие объекты проходили через поток, можно использовать метод `peek`. Добавив такой вызов между каждой промежуточной операцией, мы сможем найти какие объекты были внутри потока.

Рассмотрим пример, показанный в 1.2:

```
persons.stream()
    .filter(person -> person.age < 18)
    .sorted(Comparator.comparing(x -> x.age))
    .map(person -> person.name)
    .collect(Collectors.toList());
```

При помощи метода `peek` можно наблюдать промежуточные состояния в потоке:

```
persons.stream()
  .peek(x -> /* action */)
  .filter(person -> person.age < 18)
  .peek(x -> /* action */)
  .sorted(Comparator.comparing(x -> x.age))
  .peek(x -> /* action */)
  .map(person -> person.name)
  .peek(x -> /* action */)
  .collect(Collectors.toList());
```

Если мы будем просто печатать элементы в методе peek (с указанием где именно), то получим результат, из которого можно извлечь некоторую информацию. Для списка из пяти человек, данный вызов напечатает следующее:

```
after stream: {id = 1, name = Vasily, age = 10}
after filter: {id = 1, name = Vasily, age = 10}
after stream: {id = 2, name = Petr, age = 15}
after filter: {id = 2, name = Petr, age = 15}
after stream: {id = 3, name = Dmitry, age = 14}
after filter: {id = 3, name = Dmitry, age = 14}
after stream: {id = 4, name = Nastya, age = 21}
after stream: {id = 5, name = Michael, age = 33}
after sort: {id = 1, name = Vasily, age = 10}
after map: Vasily
after sort: {id = 3, name = Dmitry, age = 14}
after map: Dmitry
after sort: {id = 2, name = Petr, age = 15}
after map: Petr
```

Заметим, что из такого вывода можно понять следующее:

- **Последовательность** прохождения объектов через цепочку вызовов. Напечатанные строки идут в порядке исполнения программы. Сначала из источника был взят объект с `id = 1`, затем он прошел фильтрацию и попал в sorted, и так далее.
- **Результат фильтрации**. Можно увидеть все значения, которые прошли фильтрацию – это ровно те значения, которые мы наблюдали после вызова `filter`.
- **Свойства вызовов.** Вызов `sorted` имеет состояние и требует выполнить весь поток до своего вызова. Поэтому, пока объекты в источнике не закончились, в потоке после вызова `sorted` объектов не было.

- **Множества** объектов до и после вызова. Обратим внимание, что мы печатаем положение метода peek, поэтому можно определить содержимое множеств объектов, которые были до и после каждого из вызовов.

- **Преобразования** объектов. На примере вызова map, видно, что в последовательных событиях содержится преобразование – извлечение имени человека: `{id = 1, name = Vasily, age = 10} => Vasily`

Приведенный лог плох тем, что он описывает поведение всего потока целиком, а не каждого вызова в отдельности. Этот недостаток легко исправить: для каждой операции в цепочке можем рассмотреть только те записи, которые сделаны непосредственно до и сразу после этой операции. Таким образом, получим два набора объектов. Объекты в этих наборах можно упорядочить по времени появления в логе. Будем называть эти упорядоченные множества **состояниями** потока до и после промежуточной операции.

Чтобы построить эти состояния, достаточно добавить метод `peek` между всеми операциями в потоке. Задача добавленных `peek`'ов – сохранить объект, который в него пришел. Кроме того, добавим глобальное время, которое будет увеличиваться, когда следующий элемент будет запрошен у одного из исходных промежуточных объектов Stream. Момент времени, в который наблюдается объект в методе peek – уникальный идентификатор для этого объекта в рамках данного запуска цепочки. Это позволит различать одинаковые объекты, которые могут встретиться в потоке.

```
int[] time = new int[]{0};

source.stream()
    ...
    .peek(x -> store("before call n", x, time))
    .call(/* ... */)
    .peek(x -> time[0]++)
    .peek(x -> store("after call n", x, time))
    ...
    .terminationCall();
```

Таким образом, для вызова `call` мы сможем составить состояние потока до него и после него:

$$L = \{(t_i, obj_i)\}, R = \{(t_j, obj_j)\}$$

Первый элемент пары – момент времени, в который наблюдается объект. Второй элемент пары – наблюдаемый объект. Такие пары, полученные до операции `call`, обозначим L, а после `call` – R

После этого нужно восстановить связи между элементами множеств L и R. Для удобства описания следующих алгоритмов введем проекции для элементов множеств L и R. Пусть $x = (t, obj)$ элемент одного из этих множеств, тогда проекции задаются правилами:

$$time(x) = t$$
$$object(x) = obj$$

Множеством **переходов** будем называть множество $T \subset L \times R$. Элемент этого множества (l, r), где $l \in L, r \in R$ несет следующую информацию: элемент потока r является результатом воздействия операции `call` на элемент l.

Для большинства промежуточных операций множеств L и R достаточно для построения переходов. Единственное исключение – промежуточная операция `distinct`. Решение для него предложим немного позже в 2.5.4.

Таким образом нам удалось локализовать задачу поиска переходов для объектов внутри потока. Вместо того, чтобы пытаться отследить путь объекта через весь поток, мы получили задачу восстановления переходов для промежуточных вызовов поотдельности. Эта задача имеет решение абсолютно для всех промежуточных вызовов, которые есть в стандартной реализации. Разрешив переходы для всех промежуточных операций, можно последовательно восстановить историю объекта уже внутри целого потока. Это даст удобный способ понять, как объект попал в любую часть потока, а значит, появится возможность быстрее понять, что именно работает не так, как ожидается.

2.5.2. Вычисление выражения

Выше мы получили формальную постановку задачи. Для решения этой задачи на практике нам нужно иметь возможность построить множества L и R. Для этого нам достаточно уметь вычислять модифицированную цепочку (после добавления методов `peek` и возможно каких-либо других методов, которые не влияют на итоговый результат).

При попадании на точку останова, отладчики сред разработки предоставляют возможность вычислять выражения на java.

Существует 2 способа вычислить выражение.

- JDI. Java Debug Interface позволяет взаимодействовать с объектами на удаленной виртуальной машине при помощи объектов-посредников (см 1.1.1). Эти объекты позволяют узнать тип объекта, получить значение полей, вызывать методы, создать новый экземпляр, передать в качестве параметра в другой метод и другие. Зачастую для вычисления выражений хватает интерфейса JDI. Но не всегда: в JDI отсутствует возможность определять новые классы, Это значит,

что в выражениях нельзя использовать синтаксис, приводящий к определению новых классов. Это ограничение запрещает использование анонимных классов и анонимных функций.

- Загрузка новых классов. Основным достоинством этого подхода является возможность определить новые классы, а значит – в выражениях можно использовать анонимные классы и функции. Недостатком является то, что прежде чем загрузить класс, его нужно скомпилировать и загрузить. С другой стороны, после того как код загружен, он сможет исполняться быстрее, чем с использованием JDI (нет накладных расходов на взаимодействие через объекты-посредники 1.1.1). Этот способ лучше подходит для вычислительно сложных выражений.

В нашем случае, большинство промежуточных операций принимают параметр, который может быть анонимным классом, либо анонимной функцией. Это не позволяет использовать первый способ, поэтому у нас нет выбора и будем использовать второй.

2.5.3. Интерпретация результата

Как мы сказали в 2.5.1, результатом выполнения выражения является набор множеств L и R для каждой промежуточной операции. Элементы этих множеств это пары $\{t_i, x_i\}$, где t_i – это целое число, x_i – это соответствующий объект-посредник для объекта, который находится на виртуальной машине, исполняющей отлаживаемую программу(см 1.1.1). Будем считать, что элементы внутри множеств L и R упорядочены по времени. Иными словами, $t_i < t_{i+1}$. Рассмотрим правила построения переходов для каждого из промежуточных вызовов стандартной реализации Stream API.

- `map` – промежуточная операция без состояния – она сразу же вернёт новый элемент потока. Таким образом для элемента $\{t_i, x_i\} = l \in L$, будет соответствовать в точности один элемент $\{t_j, y_j\} = r \in R$, такой, что $t_j = t_i + 1$. Таким образом, мы нашли переход (l, r). Всего таких переходов ровно $|L|$.

- `filter` – промежуточная операция без состояния. У применения предиката операции `filter` на элемент есть два исхода: он останется в потоке, либо будет отброшен. Если он останется, то сразу же вернется в поток. Это значит, что если объект $\{t_i, x_i\} = l \in L$ прошёл фильтрацию, следовательно найдется элемент $\{t_j, y_j\} = r \in R$, такой что $t_j = t_i + 1$, и элементы l и r образуют переход (l, r) для этой операции. Если же условие не выполнилось, то в следующий момент исключена ситуация, когда описанный элемент r будет в множестве R, т.к. операция `filter` не имеет состояния и не может выдать новый элемент, прежде чем прочитает следующий, а значит значение $t_j > t_i + 1$.

Очевидно, что количество переходов для операции `filter` не превышает $|L|$, т.к. для каждого элемента из L может быть не более одного перехода.

- `flatMap` – промежуточная операция без состояния. В результате применения `flatMap`, один объект в потоке заменяется на 0 или более новых объектов. По аналогии с решением для `filter`, воспользуемся тем, что `flatMap` не может считывать следующие значения, до тех пор, пока можно возвращать значения, соответствующие считанному в прошлый раз. Рассмотрим элемент $\{t_i, x_i\} = l \in L$, для которого мы хотим построить переходы. Далее возможны два случая.

  – Существует элемент $\{t_{i+1}, x_{i+1}\} \in L$. Это значит, что элементы из R, которые соответствуют l, не могли появится раньше момента t_i, и не позже t_{i+1}. Таким образом, получили переходы (l, r),
  $r \in \{\{t_j, x_j\} \in R : t_i < t_j < t_{i+1}\}$.

  – l – последний в L. Нужно построить переходы $(l, r),\ \forall r \in \{\{t_j, x_j\} \in R : t_i < t_j\}$

  Для каждого элемента из R может быть не более одного перехода. Поэтому количество переходов для этого вызова не более $|R|$.

- `sorted` – промежуточная операция с состоянием. В результате применения операции sorted в потоке остаются те же самые объекты, но изменяется их порядок. Множество переходов в этом случае $T = \{(l, r)\} = \{(\{t_i, obj\}, \{t_j, obj\})\}$. То есть пары, в которых совпадают объекты, но значения момента времени, в которые объект наблюдался в потоке, различны. В случае, если в потоке были идентичные объекты (для которых верно, что `obj1 == obj2`), нужно добавить ограничение, что каждый элемент из множества L входит ровно в один переход в T. Аналогичное ограничение верно и для элементов множества R.

  Количество переходов для этой операции в точности равно $|L| = |R|$

Выше перечислены методы, для которых построение переходов различается. Для остальных вызовов можно использовать эти же алгоритмы, т.к. они в смысле переходов являются частным случаев описанных выше операций.

- `limit(k)` – частный случай `filter` – фильтрацию прошли только первые k элементов;

- `skip(k)` – частный случай `filter` – первые k элементов не прошли фильтрацию, а все остальные прошли;

- `peek` – частный случай `filter` – все элементы прошли фильтрацию;

- `onClose` – частный случай peek;

- `flatMapToInt/flatMapToLong/flatMapToDouble` –частный случай `flatMap`;
- `mapToInt/mapToLong/mapToDouble/mapToObj` – частный случай `map`;
- `boxed` – частный случай `mapToObj`.

Заметим, что для всех рассмотренных операций размер множества переходов не превосходит $\max(|L|, |R|)$, поэтому при реализации нужно стремиться к линейной сложности (от размеров множеств L и R) алгоритмов построения переходов.

2.5.4. Решение для операции distinct

`distinct` – промежуточная операция с состоянием, поэтому прежде чем что-либо вернуть, ей может понадобиться прочитать весь входной поток. Результатом является поток объектов, для которого гарантируется, что все объекты попарно различны (в отношении `equals`[10]). Новых объектов при этом появиться не может.

На первый взгляд, эта операция является частным случаем операции `filter`: в ней так же часть элементов из потока будет отброшена. Но в отличие от `filter` объекты обрабатываются не независимо друг от друга, поэтому рассуждения, представленные в 2.5.3 могут лишь помочь увидеть, какие элементы прошли дальше, но узнать причины, по которым другие были отброшены, не удастся. Для установления этих причин необходимо узнать, какие объекты равны друг другу. Но имея лишь состояния потока до и после операции `distinct`, восстановить эту информацию невозможно. Проблема возникает, когда объект был отфильтрован в результате вызова. Всё, что мы можем сказать о данном объекте, – он равен ровно одному объекту из тех, которые были после вызова. Но какому именно, сказать нельзя. (Решение попарно сравнить все объекты нам не подходит, т.к. оно квадратичное, и приведет к вызову методов через объекты-обертки, а значит к дополнительным накладным расходам на передачу данных между виртуальными машинами 1.1.1).

Для решения описанной проблемы используем следующий подход. В java операция equals должна удовлетворять отношению эквивалентности. Тогда множество объектов L можно разбить на классы эквивалентности по `equals` для объектов $object(l), l \in L$ на непересекающиеся множества $L_1, L_2, .., L_k$. Для этого будем использовать `HashMap<Object, List<Ojbect>>`. У этой структуры данных есть свойство – все её ключи попарно различны. Ключами данного отображения будут объекты перед вызовом `distinct`, а значениями – списки объектов, равных между собой. То есть для каждого объекта $x \in \{object(l) : l \in L\}$ нужно получить соответвующий список (при помощи `computeIfAbsent(x, key -> new ArrayList())`), затем добавить `x` в конец списка. Сложность описанного алгоритма $O(|L|)$ в среднем.

Заметим, что объекты в множестве R – это, согласно семантике `distinct`, представители каждого из классов эквивалентности. Таким образом, для того, чтобы постро-

ить переходы, нужно взять объект $r \in R$, найти класс эквивалентности L_i, к которому он принадлежит, и добавить переходы $(l, r)\ \forall l \in L_i$.

2.5.5. Переходы для завершающих операций

Иногда может быть интересно узнать, почему в результате выполнения цепочки вызовов получился тот или иной результат. Для некоторых операций это можно понять при помощи переходов к результирующему значению, либо к его полям/содержимому. Поставим задачу найти такие переходы.

Для следующих завершающих операций можно получить дополнительную информацию при помощи переходов. Опишем для каждой из них способ нахождения множества переходов:

- `toArray` – наиболее простая завершающая операция. Объекты потока становятся элементами результирующего массива, сохраняя свой порядок.
- `min/max/findFirst/findAny` – вызовы, которые возвращают Optional [8]. Возможно сделать переход для объектов, которые равны содержимому Optional.
- `allMatch` – короткозамкнутая завершающая операция. Возвращает `true`, если все объекты в потоке удовлетворяют переданному предикату, иначе `false`. Если в результате получили значение `false`, то при отладке полезно знать, для каких объектов предикат не верен.

  Чтобы получить объекты, для которые предикат не верен, достаточно произвести описанную ниже трансформацию цепочки. Пусть исходная цепочка имеет вид:

  ```
  source.stream()./* ops */.allMatch(predicate);
  ```

  После трансформации она будет иметь следующий вид:

  ```
  source.stream()./* ops */
      .filter(x -> !predicate.test(x))
      .peek(x -> store(x))
      .allMatch(x -> false);
  ```

  Все объекты, попавшие в метод peek, не прошли проверку `predicate`, а значит, нарушают условие `allMatch`. Результат вызова новой цепочки совпадает с результатом исходной, потому что, если поток пуст, то вызов `allMatch(x -> false)` вернёт `true`. Заметим, что предикат вызовется по одному разу для каждого объекта, как и в исходной цепочке.

- `anyMatch` – Короткозамкнутая завершающая операция. Возвращает `true`, если хотя бы один объект удовлетворяет переданному предикату, иначе `false`. Если значение оказалось true, то для целей отладки может быть полезно знать для каких объектов выполнился предикат.

  Чтобы найти эти объекты, снова трансформируем цепочку, но уже немного иначе. Пусть исходная цепочка имеет следующий вид:

  ```
  source.stream()./* ops */.anyMatch(predicate);
  ```

  После трансформации она будет иметь следующий вид:

  ```
  source.stream()./* ops */
      .filter(predicate)
      .peek(x -> store(x))
      .anyMatch(x -> true);
  ```

  По аналогии с `allMatch`, можно понять, что такая цепочка имеет тот же результат, позволяет найти интересующие объекты и вызывает `predicate` ровно для тех же самых объектов, причем делает это не чаще одного раза для каждого из объектов. А значит, такая трансформация корректна.

- `noneMatch` – Короткозамкнутая завершающая операция. Возвращает `true`, если все объекты в потоке не удовлетворяют переданному предикату, а в случае, когда хотя бы один удовлетворяет, `false`.

  После выполнения этой операции полезно узнать, почему результат $false$. Это значит, что нужно узнать, для каких объектов предикат вернул $true$.

  Покажем соответствующую трансформацию цепочки. Пусть исходная цепочка имеет вид:

  ```
  source.stream()./* ops */.noneMatch(predicate);
  ```

  После трансформации она будет иметь следующий вид:

  ```
  source.stream()./* ops */
      .filter(predicate)
      .peek(x -> store(x))
      .noneMatch(x -> true);
  ```

  По аналогии с предыдущими операциями можно убедиться, что такая трансформация корректна.

На первый взгляд может показаться, что в операциях `allMatch`/`anyMatch`/`noneMatch` интересующий нас объект - это всегда последний элемент перед завершающей операцией. Но это не так, потому что короткозамкнутые операции не дают гарантий, что прочитают ровно столько элементов, сколько им достаточно – они могут прочитать больше. В текущей реализации это наблюдается, например, при использовании `flatMap` перед короткозамкнутой операцией.

```
Stream.of(1, 2)
    .flatMap(x -> Stream.of(1,2,3))
    .peek(x -> System.out.print(x))
    .anyMatch(x -> x == 1);
```

Данный вызов напечатает 123, хотя для завершения anyMatch достаточно только первого объекта. Описанный выше подход корректно справится с этой особенностью – он строит переход `(1, true)`. Наивный алгоритм предложит неверный переход `(3, true)`.

3. Реализация решения

В главе 2 было представлено описание решения, которое позволит получить промежуточные состояния потока объектов между вызовами операций Stream API. Чтобы применить решение на практике, реализуем расширение для среды разработки IntelliJ IDEA.

IntelliJ IDEA – среда для разработки, основанная на платформе IntelliJ, позволяет разрабатывать программы на нескольких языках программирования, в том числе на java. Функциональность платформы может быть расширена с помощью плагинов, которые могут быть установлены из официального репозитория [2], либо из других источников. Платформа предоставляет разработчикам плагинов API – классы и интерфейсы, с помощью которых компоненты плагина могут быть интегрированы в работу платформы.

3.1. Нахождение подходящего вызова

Для нахождения границ вызова будем использовать API, предоставляемый платформой для работы с исходным кодом. Исходный код представляется в виде *AST* (абстрактное синтаксическое дерево [17]). Обходя его, можно найти цепочки вызовов. Интерфейс платформы позволяет определить тип объекта, на котором вызывается метод и тип результата вызова. Используя следующие правила, мы сможем классифицировать вызовы внутри цепочки:

- Промежуточный вызов – возвращает объект, реализующий `Stream<T>`.
- Завершающий вызов - происходит на объекте, реализующем `Stream<T>`, возвращает объект произвольного типа, который не реализует `Stream<T>`.

Учитывая, что положение отладчика может быть отображено на некоторый элемент *AST*, у нас есть возможность обойти поддерево этого элемента и найти все подходящие цепочки. Кроме того, мы не привязываемся к именам методов для промежуточных операций, это снижает требование к потоку, который может быть отлажен (вызовы с неподдерживаемым операциями могут отлаживаться: для них будут построены состояния, но переходы восстановить не удастся). Таким образом, можно удовлетворить всем требованиям из 2.4.

Отдельно стоит рассмотреть редкий случай, когда результат завершающей операции наследник `Stream<T>`. Действуя неосторожно, можно перепутать такую завершающую операцию с промежуточной. Чтобы этого избежать, необходимо проверить, что имя промежуточной операции не совпадает с именами терминальных операций, которые могут вернуть `Stream<T>`. Таких операций немного – `collect` и `reduce`.

3.2. Построение выражения

Во второй главе был получен важный результат. Для того, чтобы восстановить все переходы, их нужно восстановить локально для каждого из промежуточных вызовов. При этом необходимая информация для построения переходов у разных операций может различаться. Поэтому будет удобно ввести абстракции, которые позволят каждому вызову модифицировать цепочку, чтобы собрать данные для восстановления переходов.

Для всех вызовов верно, что они требуют лишь локальной модификации цепочки: добавления методов до и после самого вызова.

Таким образом, достаточно ввести абстракции, которые позволят:

- Объявлять локальные переменные, нужные для сохранения информации о вычислении, в выражении для вычисления.
- Добавлять промежуточные вызовы в цепочку до и после самой операции.
- Преобразовывать собранную информацию в процессе запуска выражения к удобному представлению для дальнейшей интерпретации.

При этом нужно позаботиться, чтобы имена локальных переменных, которые используют разные вызовы были различны. Самый простой способ этого достичь – добавить к имени используемых переменных порядковый номер вызова в исходной цепочке.

Структура фрагмента кода для вычисления, с точки зрения одной операции, будет выглядеть следующим образом:

```
final Object trace = new Object[M + 2];

final int[] time = new time[] { 0 };

final VarType callVariable11 = ...;

// declarations for calls from 2 to k - 1

final VarType callVariableK1 = ...;
final VarType callVariableK2 = ...;

// declarations for calls from k + 1 to M - 1

final VarType callVariableM1 = ...;
final VarType callVariableM2 = ...;
```

```
final VarType callVariableM3 = ...;

Object streamResult = souce.stream().
        .peek(x -> time[0]++)
        .callBefore11(/* args */)
    .callNumber1()
        .peek(x -> time[0]++)
        .callArfter11(/* args */)
    /* calls from 2 to k - 1 */
        .callBeforeK1(/* args */)
        .callBeforeK2(/* args */)
    .callNumberK()
        .peek(x -> time[0]++)
        .callAfterK1(/* args */)
        .callAfterK2(/* args */)
    /* calls from k + 1 to M - 1  */
        .callBeforeM1(/* args */)
    .callNumberM()
        .peek(x -> time[0]++)
        .callArfterM1(/* args */)
        .callArfterM2(/* args */)
    .terminal();

trace[0] = prepare0(callVariable11);
/* saving of trace for calls from 2 to K - 1 */
trace[K] = prepareK(callVariableK1, callVariableK2)
/* saving of trace for calls from K + 1 to M - 1 */
trace[M - 1] = prepareM(callVariableM1, callVariableM2, callVariableM3)
trace[M] = streamResult;
```

В примере выше показано, как вызов, для которого мы хотим построить состояния и переходы, может воспринимать структуру генерируемого кода. Этот вызов является K-м по порядку в цепочке из M методов. Его основные особенности:

- Массив `trace` используется для сохранения результатов. Имеено он является результатом вычисления фрагмента этого фрагмента кода.
- В начале фрагмента происходит определение всех переменных, использующихся для сбора данных о объектах внутри потока. Различным вызовам может требоваться разное количество переменных.

- После определения переменных расположен вызов модифицированной цепочки. Каждая операция исходной цепочки дополнила её требуемыми вызовами до и после самой операции. Количество этих вызовов определяется потребностями вызова.

- После каждого вызова исходной цепочки добавлен вспомогательный вызов `peek`. Его цель – увеличить глобальное время (2.5.1) после появления нового объекта в цепочке.

- Затем последовательно заполняется массив `trace`. Исходные значения переменных могут быть неудобными для интерпретации, поэтому их стоит трансформировать так, чтобы на стороне IDE, этими данными было удобно оперировать (см 3.4). Обычно это означает конвертацию вспомогательных значений в массивы (возможно, вложенные).

Пример сгенерированного кода есть в приложении А.

3.3. Вычисление выражения

В 2.5.2 описаны подходы к вычислению произвольного кода внутри отладчика. Там же был сделан выбор в пользу стратегии загрузки и запуска новых классов. Данная возможность уже присутствует в среде разработки, поэтому не будем подробно останавливаться на деталях реализации этого процесса. У такого подхода есть недостаток – у загружаемых классов может не быть прав доступа к некоторым полям и методам других объектов. Это происходит из-за того, что новый класс компилируется как внутренний класс для класса, внутри которого находится отладчик при запуске операции вычисления выражения. Но класс, в котором находится отладчик уже загружен, и он не знает о новом внутреннем классе, поэтому не может предоставить доступ к приватным полям и методам.

Для решения этой проблемы используется внутренний класс JDK `MagicAccessorImpl` [13]. Это часть небезопасного API java. Наследование от этого класса является указанием для виртуальной машины java, что необходимо пропускать проверки доступа при вызове методов и обращении к полям из методов класса-наследника `MagicAccessorImpl`.

Кроме сгенерированного класса, анонимные функции тоже могут пытаться получить доступ к приватным методам и полям. В текущей версии платформы это приведет к исключению. Но это можно обойти, преобразовав все анонимные функции, которые встречаются в выражении, в анонимные классы, которые наследуются от `MagicAccessorImpl`. Для этого преобразования можно использовать существующие рефакторинги для преобразования анонимных функций в анонимные классы.

Таким образом, весь код внутри выражения для построения состояний и переходов не будет иметь проблем с доступом к методам и полям других классов.

3.4. Интерпретация результата вычисления выражения

В 2.5 был предложен способ восстановления переходов по информации, собранной на этапе вычисления выражения. Результатом вычисления является некий объект, в котором хранится вся собранная информация. Для упрощения обработки этой информации потребуем, чтобы она хранилась только в массивах. Это обусловлено тем, что JDI позволяет получить элементы массива без вызова дополнительных методов на отлаживаемой виртуальной машине. Заметим, что вспомогательные значения и объекты, которые использовались для сохранения информации при вычислении, всегда можно представить как массив объектов:

- Если этот объект был типа `Map<Key, Value>`, то его можно представить в виде двух массивов: массив ключей и массив значений. Сами ключи и значения, в свою очередь, так же могут быть массивами.
- Другие коллекции очевидным образом могут быть представлены в виде массивов.
- Произвольный объект `reference` может быть сохранен в виде массива: `new Object[] {reference}`.
- Значение примитивного типа, тоже может быть обернуто в массив. Например, так `new int[] {42}`.

Эти преобразования нужно выполнять только для вспомогательных структур данных. Для объектов из потока этого делать не нужно.

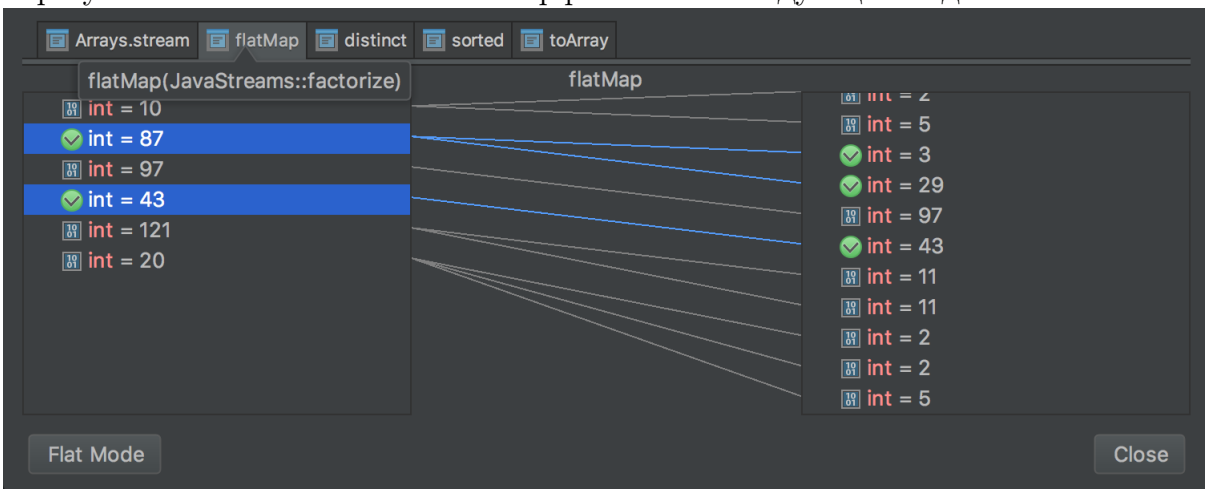
В результате интерпретации и разрешения переходов согласно правилам из 2.5.3, получим промежуточные состояния и переходы объектов между соседними состояниями.

3.5. Визуализация

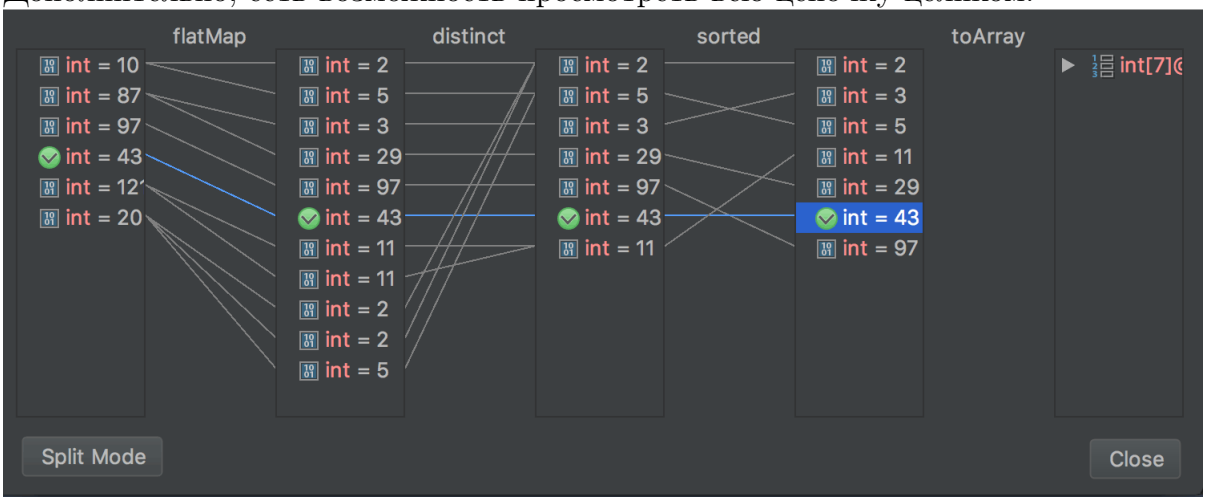
После завершения этапа интерпретации имеется набор промежуточных состояний и переходы между ними. Промежуточные состояния – это просто список объектов. Переходы – это связь между объектам в двух соседних состояниях. Наиболее естественно это визуализировать с помощью двух списков объектов и линий-переходов.

При выборе какого-либо значения в списке нужно визуально выделить все объекты из соседних состояний, с которыми он связан переходами, затем те, с которыми связаны они и так далее.

В результате пользовательский интерфейс имеет следующий вид:



Дополнительно, есть возможность просмотреть всю цепочку целиком:



Заключение

Настоящая работа посвящена проблемам отладки кода, использующего пакет java.util.stream стандартной библиотеки языка java. В работе исследованы имеющиеся способы нахождения ошибок в таком коде, а так же описаны их недостатки. Были сформулированы требования к решению, которое упростит процесс отладки. Был описан подход, который позволит получить информацию о последовательности исполнения вызова Stream API и данных о трансформациях каждого объекта внутри этого вызова. Описанный подход был реализован в качестве расширения для среды разработки IntelliJ IDEA, упрощающего отладку вызовов Stream API. Расширение доступно для пользователей [15].

В качестве возможных направлений развития следует отметить:

- Поддержка операций над потоками элементов, предоставляемых библиотеками StreamEx и jOOL.
- Обобщение решения для возможности отладки функциональных операций в других языках – Scala, Kotlin, C#.
- Исследование возможности сбора информации об объектах внутри потока при помощи точек останова.

Список литературы

[1] JetBrains. Kotlin Sequence // Koltin Reference. — 2017. — Access mode: https://goo.gl/dxtIhj (online; accessed: 28.05.2017).

[2] JetBrains. Plugin Repository // JetBrains Plugin Repository. — 2017. — Access mode: https://plugins.jetbrains.com/idea (online; accessed: 3.06.2017).

[3] Martin Odersky Lex Spoon. Views // The Scala 2.8 Collections API. — 2010. — Access mode: https://goo.gl/NuV4Kj (online; accessed: 28.05.2017).

[4] Microsoft. Extension Methods // C# Reference. — 2017. — Access mode: https://goo.gl/Sv4N3o (online; accessed: 25.04.2017).

[5] Microsoft. yield // C# Reference. — 2017. — Access mode: https://goo.gl/TgOYPx (online; accessed: 27.05.2017).

[6] Oracle. The Java Debugger // Java SE Documentation. — 2014. — Access mode: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html (online; accessed: 29.05.2017).

[7] Oracle. Fork/Join // The Java Documentation. — 2015. — Access mode: https://goo.gl/TUVlj9 (online; accessed: 15.04.2017).

[8] Oracle. Optional // The Java Documentation. — 2015. — Access mode: https://goo.gl/MqYuFC (online; accessed: 15.04.2017).

[9] Oracle. Iterator // The Java Documentation. — 2016. — Access mode: https://goo.gl/rX5m1j (online; accessed: 25.04.2017).

[10] Oracle. Method equals of class Object // The Java Documentation. — 2016. — Access mode: https://goo.gl/92O32c (online; accessed: 23.04.2017).

[11] Oracle. Spliterator // The Java Documentation. — 2016. — Access mode: https://goo.gl/fonBNg (online; accessed: 25.04.2017).

[12] Oracle. Stream interface // The Java Documentation. — 2016. — Access mode: https://goo.gl/TZp895 (online; accessed: 25.04.2017).

[13] Oracle. MagicAccessorImpl // OpenJDK Repository. — 2017. — Access mode: https://goo.gl/U70Sb2 (online; accessed: 2.06.2017).

[14] OzCode. OzCode LINQ Debugger // List of OzCode features. — 2017. — Access mode: https://goo.gl/DlGa7S (online; accessed: 29.05.2017).

[15] V. Bibaev. Java Stream Debugger // JetBrains Plugin Repository. — 2017. — Access mode: https://plugins.jetbrains.com/plugin/9696-java-stream-debugger (online; accessed: 4.06.2017).

[16] Wikipedia. Отладчик // Википедия, свободная энциклопедия. — 2016. — Режим доступа: https://goo.gl/OYnIZX (дата обращения: 31.05.2017).

[17] Wikipedia. Abstract syntax tree // Wikipedia, the free encyclopedia. — 2016. — Access mode: https://goo.gl/xMJiIx (online; accessed: 02.05.2017).

[18] Wikipedia. Work stealing // Wikipedia, the free encyclopedia. — 2016. — Access mode: https://goo.gl/xMJiIx (online; accessed: 21.04.2017).

[19] Wikipedia. LINQ // Википедия, свободная энциклопедия. — 2017. — Режим доступа: https://goo.gl/Eq3CmX (дата обращения: 25.04.2017).

[20] Wikipedia. Microsoft Visual Studio // Wikipedia, the free encyclopedia. — 2017. — Access mode: https://goo.gl/f8kzP3/ (online; accessed: 29.05.2017).

[21] amaembo. StreamEx // Github repository. — 2017. — Access mode: https://github.com/amaembo/streamex (online; accessed: 1.06.2017).

[22] jOOQ. jOOL // Github repository. — 2017. — Access mode: https://github.com/jOOQ/jOOL (online; accessed: 2.06.2017).

ПРИЛОЖЕНИЕ А

Пример сгенерированного фрагмента кода

```
Arrays.stream(new int[]{10, 87, 97, 43, 121, 20})
        .flatMap(JavaStreams::factorize)
        .distinct()
        .sorted()
        .toArray();
```

К сожалению, сгенерированный код слишком объемный, чтобы поместить его на эти страницы, но он доступен в интернете по адресу: https://goo.gl/cI2euu.

ПРИЛОЖЕНИЕ Б

Исходный код разработанного решения

Исходный код плагина расположен в репозитории по адресу https://github.com/Roenke/stream-debugger-plugin.