Потоки данных

Итерирование коллекции

Типичная операция – обойти коллекцию, применяя итерацию к каждому элементу

```
List <Student>
allStudents;

. . .

int count=0;

for (Student
s:allStudents){

  if (s.isFrom(“Sfedu”))

       count++;

}
```

```
List <Student> allStudents;

. . .

int count=0;

Iterator<Student> it =
allStudents.iterator();

while(it.hasNext()){

Student   s = it.next();

  if (s.isFrom(“Sfedu”) )

       count++;

}
```

Проблемы

- Много стереотипного кода
- Невозможно распараллелить
- При вложенности циклов с большим количеством кода затемняется смысл

Внутреннее итерирование

```
long   count = allStudents.stream()
               .filter(st ->st.isFrom(“Sfedu”))
               .count();
```

Для параллельного выполнения

```
long   count = allStudents.parallelStream()
               .filter(st ->st.isFrom(“Sfedu”))
               .count();
```

Поток данных

- Средство конструирования сложных операций над коллекциями (массивами, генераторами или итераторами) с применением функционального подхода
- Не сохраняет свои элементы, они или хранятся в основной коллекции или формируются по требованию
- Потоковые операции не изменят их источник, а формируют новые потоки или выдают результат

Поток данных

- Потоки
  - Создаются из источника данных
  - Элементы потока подвергаются серии промежуточных операций (***конвейер***)
  - Процесс обработки потока должен завершаться ***терминальной операцией***
- Поток, выполнивший терминальную операцию, считается завершенным
- Поток обрабатывает лишь столько данных, сколько нужно для перехода в терминальное состояние
- Поток можно использовать только один раз

Создание потока

- Интерфейс `Stream<T>`

```
Stream <String> words = Stream.of("one","word", "next","Word");
```

- Для любой коллекции
  - Методом `stream()` или `parallelStream()`

```
List<Integer> list = new ArrayList<>();
Collections.addAll(list, 1, 5, 6, 11, 3, 15, 7, 8);
list.stream()
    .forEach(value-> System.out.println(value));
//list.stream().forEach(System.out::println);
```

Создание потока

- Для массива
  - метод `Stream.of(array);`
  - метод `Arrays.stream(array,from,to);`
- Пустой поток

```
Stream<String> s = Stream.empty();
```

Создание потока

- Бесконечный поток
  - Статические методы интерфейса `Stream`
  - Используя объект функционального интерфейса `Supplier<T>`

```
Stream<String> echo =Stream.generate(
          () -> "Echo");
echo.forEach(System.out::println);
Stream<Double> randoms =Stream.generate(
          Math::random);
```

Создание потока

- Бесконечный поток
  - Используя объект функционального интерфейса `UnaryOperator <T>`

```
Stream <BigInteger> integers = Stream.iterate(
  BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

Создание потока

- Ограничить поток
  - Используя предикатную функцию

```
BigInteger limit = new BigInteger("10000000000");
Stream <BigInteger> integers = Stream.iterate(
  BigInteger.ZERO,
  n -> n.compareTo(limit)<0,
  n -> n.add(BigInteger.ONE));
```

Создание потока

- Ограничить поток

```
Stream.iterate(LocalDate.now(), ld->ld.plusDays(1L))
   .limit(10)
   .forEach(System.out::println);
```

Создание потока

- Статические методы других классов

```
String str = ...;
String regex = . . .;// “\\PL+” строку на слова
. . .
Stream <String> words =Pattern.compile(regex)
                    . splitAsStream(str);

try( Stream<String> lines =  Files.lines(path)){
// строки из файла
 . . .
}
```

Методы потоков

- Преобразуют поток в другой поток
  - `filter()` формирует поток с данными, удовлетворяющими условию.
  - Условие – объект типа `Predicate<T>`

```
List <String> words =…;
Stream<String> longW = words.stream()
                       .filter(w->w.length()>12);
```

Методы потоков

```
List<Integer> list = new ArrayList<>();
Collections.addAll(list, 1, 5, 6, 11, 3, 15, 7, 8);
list.stream()
    .filter(i -> i%2==0)
    .forEach(System.out::println);
```

Методы потоков

- Преобразуют поток в другой поток
  - `map()` формирует поток с преобразованием

```
List <String> words =…;
Stream<String>
    lowercaseW   = words.stream()
                    .map(String::toLowerCase);
Stream<String>
    firstLetters =  words.stream()
                     .map(s -> s.substring(0,1));
```

Методы потоков

- Ограничение потока
  - `limit(n)`
  - `skip (n)`
  - `takeWhile( предикат )`
  - `dropWhile( предикат )`

```
Stream <Double> randoms = Stream.generate(Math::random)
                                .limit(100);
```

Методы потоков

- Соединение двух потоков
  - `concat(поток1, поток 2)`
- Другие преобразования
  - `reversed()` обратный порядок
  - `distinct()` подавление дубликатов в потоке
  - `sorted( компаратор)` сортировка

```
Stream<String>  lengthSort    =
                  words.stream()
                .sorted(Comparator.comparing(String::length));
```

Методы потоков

```
Stream<String>  lengthSort   =
                words.stream()
               .sorted(Comparator.comparing(String::length));
```

```
List<String> list = new ArrayList<>();
Collections.addAll(list,"a1","a2","back", "a2", "qwerty", "a2");
list.stream()
    .distinct()
    .forEach(System.out::println);
```

Методы потоков

- Выполнение для каждого элемента потока
  - `peek( функция)`

```
Stream<String>
    lowercaseW   = words.stream()
                .peek(System.out::println)
                .map(String::toLowerCase)
                .peek(System.out::println);
```

Методы сведения

- Выполнят операции, сводя поток к непотоковому значению
  - `count()`
  - `anyMatch()`
  - `allMatch()`
  - `noneMatch()`
- возвращающие значение Optional<T>
  - `max()`
  - `min()`
  - `findFirst()`
  - `findAny()`

Методы сведения

```
List<String> list = new ArrayList<>();
Collections.addAll(list, "разные", "слова",...);
System.out.println(
    list.stream()
        .filter(w -> w.length() == 5)
        .count());
```

Методы сведения

```
Optional<String> startWithQ  =
     words.stream()
        .parallel()
        .filter(s -> s.startsWith("Q")).
        .findAny();

boolean aWordStartWithQ  =
     words.stream()
        .parallel()
        .anyMatch (s -> s.startsWith("Q"));
```

Накопление результатов

- просмотр результатов

```
stream.forEach(System.out::println);
stream.forEachOrdered(System.out::println);
```

- сохранение в структуре данных

```
String [ ] result = stream.toArray(String[ ]::new);
List <String> result = stream.collect (
                Collectors.toList());
Set <String> result = stream.collect (
                Collectors.toSet());
```

Примеры

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");


myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Примеры

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Примеры

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });
```

Примеры

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Примеры

В предыдущем примере

1. Поставить `filter` перед `map`
2. Добавить перед фильтром сортировку

```
.sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
```

3. Поменять сортировку и фильтр местами

Дополнительно

Как уйти от обобщенного типа в интерфейсе

```
public class Letters extends AbstractCollection<Character>{
//implements Collection<Character>{
 // @Override
    public boolean contains(Character o) {
        return collect.indexOf(o)>=0;
 //throw new UnsupportedOperationException("Not supported yet.");
//To change body of generated methods, choose Tools | Templates.
    }
```