ISEL

Languages and Managed Runtimes

2022

Week 9 – Sequences

Sequences

General Purpose Languages, para construir qualquer tipo de software.

GOAL: construir queries sobre objectos numa linguagem como o Kotlin, Java ou outra, de forma semelhante a como construimos queries SQL sobre dados relacionais.

Faz parte do grupo de linguagems DSL (Domain Specific Language)
Ex:
- SQL no domínio dos dados relacionais
- Compose na construção de GUI em Android
- Expressões Regulares para padrões parsing de texto
- Etc.

Sequences

- 1970 Lisp: `distinct( select(…., where(…, ) )`
  // ordem execução Esquerda -> direita

- 1990 – Small talk – suporte para operações sobre dados OO

Construção de uma **query** através de um **encadeamento** de operação que formam um ***Pipeline***

- 2003 E.g. C# `students.Where ( …).Select(… ).Distinct()`

- 2014 E.g. Java `students.stream().filter { …}.map{… }.distinct()`
- 2018 E.g. Kotlin `students.filter { …}.map{… }.distinct()`

E.g. SQL SELECT UNIQUE … WHERE ….

martinFowler.com

Press F11 to exit full screen

Refactoring Agile Architecture About

Collection Pipeline

Collection pipelines are a programming pattern where you organize some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next. (Common operations are filter, map, and reduce.) This pattern is common in functional programming, and also in object-oriented languages which have lambdas. This article describes the pattern with several examples of how to form pipelines, both to introduce the pattern to those unfamiliar with it, and to help people understand the core concepts so they can more easily take ideas from one language to another.

25 June 2015

CONTENTS

First encounters
Defining Collection Pipeline

Pipeline

martinFowler.com

Press F11 to exit full screen

Refactoring Agile Architecture About

Collection Pipeline

Collection pipelines are a programming pattern where you organize some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next. (Common operations are filter, map, and reduce.) This pattern is common in functional programming, and also in object-oriented languages which have lambdas. This article describes the pattern with several examples of how to form pipelines, both to introduce the pattern to those unfamiliar with it, and to help people understand the core concepts so they can more easily take ideas from one language to another.

25 June 2015

CONTENTS

First encounters
Defining Collection Pipeline


```
students.filter { …}.map{… }.distinct().count()
```

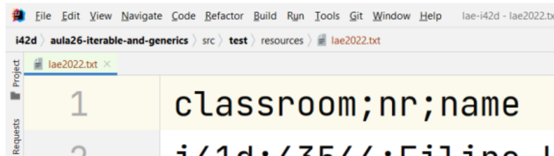
Data Source

Intermediate Operations

Terminal Operation

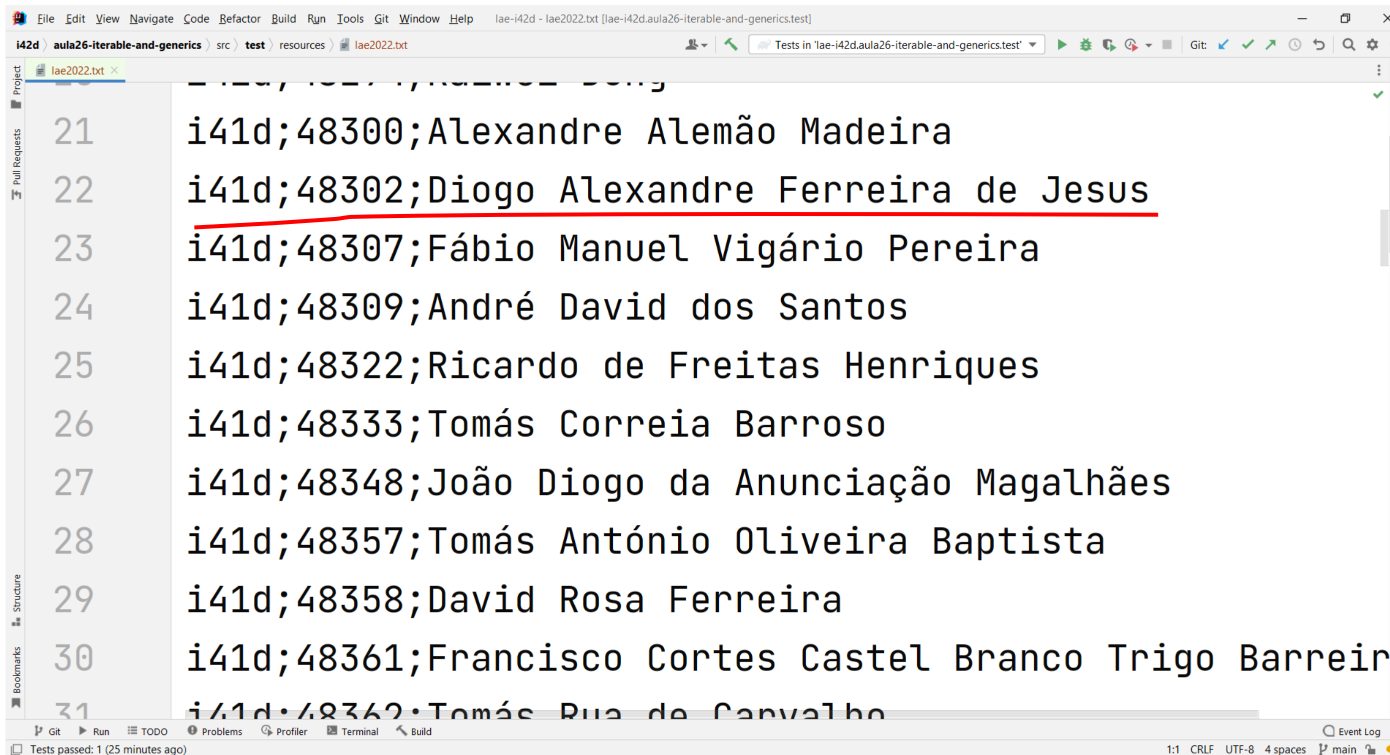
Exemplo

Ficheiro CSV lae2022.txt



URI

```
File(…)
  .readLines() // List<String>
  …
```



Query:
Seleccionar o primeiro apelido do aluno com numero maior que 47000 e a primeira letra a começar por A

Exemplo

Ficheiro CSV lae2022.txt

URI

```
1  classroom;nr;name
```

```
21  i41d;48300;Alexandre Ale
22  i41d;48302;Diogo Alexand
23  i41d;48307;Fábio Manuel
24  i41d;48309;André David d
25  i41d;48322;Ricardo de Fr
26  i41d;48333;Tomás Correia
27  i41d;48348;João Diogo da
28  i41d;48357;Tomás Antónic
29  i41d;48358;David Rosa Fe
30  i41d;48361;Francisco Cor
```

```
File(…)
    .readLines() // List<String>
    .parseCsv()  // List<Map<String, String>>
    .convertToStudent()          // List<Student>
    .whereNrGreaterThan(47000) // List <Student>
    .convertToSurname()          // List <String>
    .whereStartsWith("A")        // List <String>
    .iterator()
    .next()
```

Query:

*** Seleccionar o primeiro apelido**

*** … do aluno**

*** … com numero maior que 47000**

*** … e a primeira letra a começar por A**

Exemplo

Ficheiro CSV lae2022.txt

URI

```
1   classroom;nr;name
```

```
21  i41d;48300;Alexandre Ale
22  i41d;48302;Diogo Alexand
23  i41d;48307;Fábio Manuel
24  i41d;48309;André David d
25  i41d;48322;Ricardo de Fr
26  i41d;48333;Tomás Correia
27  i41d;48348;João Diogo da
28  i41d;48357;Tomás Antónic
29  i41d;48358;David Rosa Fe
30  i41d;48361;Francisco Cor
```

```
File(…)
   .readLines() // List<String>
   .parseCsv()  // List<Map<String, String>>
   .convertToStudent()          // List<Student>
   .whereNrGreaterThan(47000) // List <Student>
   .convertToSurname()          // List <String>
   .whereStartsWith(“A”)        // List <String>
   .iterator()
   .next()                      // 1st element
```

- EAGER processamento adiantado:
  Processa todos os elementos apesar de só seleccionarmos a 1ª ocorrencia.
- EAGER => Overhead no GC
  Constrói listas intermédias que são usadas apenas por cada operação intermédias
- **Repetição de Código entre convert… e entre where….**
- **Mistura entre Domínio <> Utilitário**

Repetição de Código

Parametro de Tipo (ou Genérico):

- **R – tipo de elementos produzido**
- **T – tipo de elementos da fonte**

```
fun ... .convertToStudents(): ... {
    val res = mutableListOf<Student>()
    for (item in this) {
        res.add(item.toStudent())
    }
    return res
}
```

```
fun ....convertToSurnames(): ... {
    val res = mutableListOf<String>()
    for (item in this) {
        res.add(item.name.split(" ").last())
    }
    return res
}
```

Parametro do tipo função:

- (Map<String, String>) -> Student
- Student -> String

(T) -> R

Repetição de Código

Parametro de Tipo (ou Genérico):

- **R – tipo de elementos produzido**
- **T – tipo de elementos da fonte**

```
fun <T, R> Iterable<T>.convert (transform: (T) -> R): Iterable<R> {
    val res = mutableListOf<R>()
    for (item in this) {
        res.add(transform(item))
    }
    return res
}
```

Parametro do tipo função:

- (Map<String, String>) -> Student
- Student -> String

(T) -> R

Repetição de Código

Parametro de Tipo (ou Genérico):

- **R – tipo de elementos produzido**
- **T – tipo de elementos da fonte**

```
fun <T, R> Iterable<T>.convert (transform: (T) -> R): Iterable<R> {
    val res = mutableListOf<R>()
    for (item in this) {
        res.add(transform(item))
    }
    return res
}
```

```
File(lae2022uri)
    .readLines()
    .parseCsv(';')
    .convert<Map<String, String>, Student> { it.toStudent() }
    .where<Student> { it.nr > 47000 }
    .convert<Student, String> { it.name.split(" ").last() }
    .where<String> { it.startsWith("A") }
    .iterator()
    .next()
```

Inferência de tipo

- Os **argumentos de tipo** são inferidos a partir dos **parametros actuais (e.g. a função do tipo T -> R)**

```
File(lae2022uri)
    .readLines()
    .parseCsv(';')
    .convert { it.toStudent() }
    .where { it.nr > 47000 }
    .convert { it.name.split(" ").last() }
    .where { it.startsWith("A") }
    .iterator()
    .next()
```

Semelhança entre convert() e o mapTo

Generics constraints -> E.g. < T: Comparable >

```
fun <T, R, C : MutableCollection<in R>> Iterable<T>.mapTo(destination: C, transform: (T) -> R): C {
    for (item in this)
        destination.add(transform(item))
    return destination
}
```

A única restrição de C é ter um método add(), ou seja,
Tem que ser compatível com MutableCollection

Reified type parameter (inline functions)

```
val actual = Files
    .lines(lae2022uri.toPath())
    ...
    .convert { it.toObject<Student>() }
    ...
```

```
val actual = Files
    .lines(lae2022uri.toPath())
    ...
    .convert { it.toObject (Student::class) }
    ...
```

Inlined

```
inline fun <reified T : Any> Map<String, String>.toObject() : T {
    return this.toObject(T::class)
}

/**
 * Creates an instance of T initialized with the values of the Map receiver object.
 *
 * @param Map<String, String> Name value pairs corresponding to T properties.
 */
fun <T : Any> Map<String, String>.toObject(destination: KClass<T>) : T {
  ....
```

- EAGER processamento adiantado:
  Processa todos os elementos apesar de só seleccionarmos a 1ª ocorrencia.

- EAGER => Overhead no GC
  Constrói listas intermédias que são usadas apenas por cada operação intermédias

- ~~Repetição de Código entre convert… e entre where….~~

- ~~Mistura entre Domínio <> Utilitário~~

Sequence Lazy

```
internal class TransformingSequence<T, R>
constructor(private val sequence: Sequence<T>, private val transformer: (T) -> R) : Sequence<R> {

    override fun iterator(): Iterator<R> = object : Iterator<R> {
        val iterator = sequence.iterator()
        override fun next(): R {
            return transformer(iterator.next())
        }

        override fun hasNext(): Boolean {
            return iterator.hasNext()
        }
    }
```

```kotlin
val actual: List<String> = File(lae2022uri)
    .readLines()
    .parseCsv(';')
    .convert { it.toStudent() }
    .where { it.nr > 47000 }
    .convert { it.name.split(" ").last() }
    .where { it.startsWith("A") }
```

```kotlin
val actual: List<String> = actual = File(lae2022uri)
    .readLines()
    .asSequence()
    .parseCsv(';')
    .convert { it.toStudent() }
    .where { it.nr > 47000 }
    .convert { it.name.split(" ").last() }
    .where { it.startsWith("A") }
    .toList()
```

yield

- Objectivo: permitir a implementação de Iteradores de forma **Lazy** com código tão simples como a implementação Eager.

```
fun <T, R> Sequence<T>.convert(transform: (T) -> R): Sequence<R> {
    return sequence {
        for (item in this@convert) {
            yield(transform(item))
        }
    }
}
```