Scala Library I

Mikhail Mutcianko, Alexey Shcherbakov

СПБгУ, СП

11 марта 2021

Scala Library: Basic Collections

Option

`Option[A]` is a container for an optional value of type `A`

- If the value of type `A` is present, the `Option[A]` is an instance of `Some[A]`
- If the value is absent, the `Option[A]` is the object `None`

Option

monad

```
1 trait Monad[T] {
2   def flatMap[U](f: T => Monad[U]): Monad[U]
3 }
4
5 def unit[T](x: T): Monad[T]
```

Option

- create with `Some(...)` or `None`
- transform with `map` or `flatMap`
- test contents with `exists` or `contains`
- check kind with `isDefined` or `isEmpty`
- unwrap with `get` or `getOrElse`

Tuple

Scala tuple combines a fixed number of items together so that they can be passed around as a whole.

- one-indexed
- immutable
- unlike an array or list, a tuple can hold objects with different types
- tuples are a syntactic sugar

Tuple

Scala tuple combines a fixed number of items together so that they can be passed around as a whole.

- one-indexed
- immutable
- unlike an array or list, a tuple can hold objects with different types
- tuples are a syntactic sugar

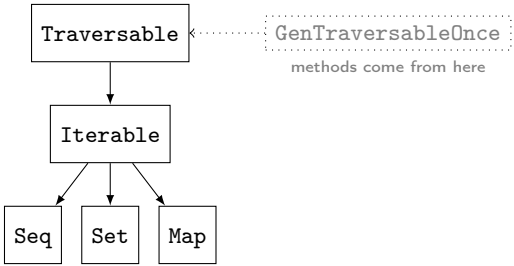
Tuple

```scala
val t = (1, "hello", Console)
val t = new Tuple3(1, "hello", Console) // same

val tuple = ("apple", "dog") // access by index
val fruit = tuple._1
val animal = tuple._2

val student = ("Sean Rogers", 21, 3.5)
val (name, age, gpa) = student // deconstruction

val tuple = ("apple", 3).swap // Tuple2 swap
```

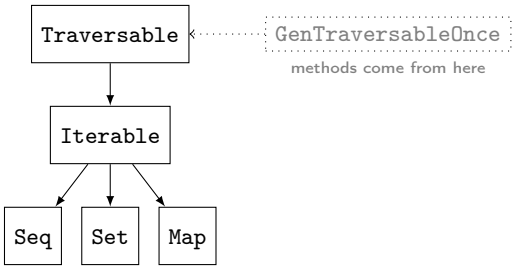
Mutable and Immutable

- Scala collections are immutable by default
- mutable collections have to be explicitly imported `import scala.collection.mutable.Map`
- all transforming operations return a modified copy
- mutable collections can be modified by methods with = suffix, e.g. ++=
- mind the variance: immutable[+T] $\leftrightarrow$ mutable[T]

Collections hierarchy



Collections hierarchy



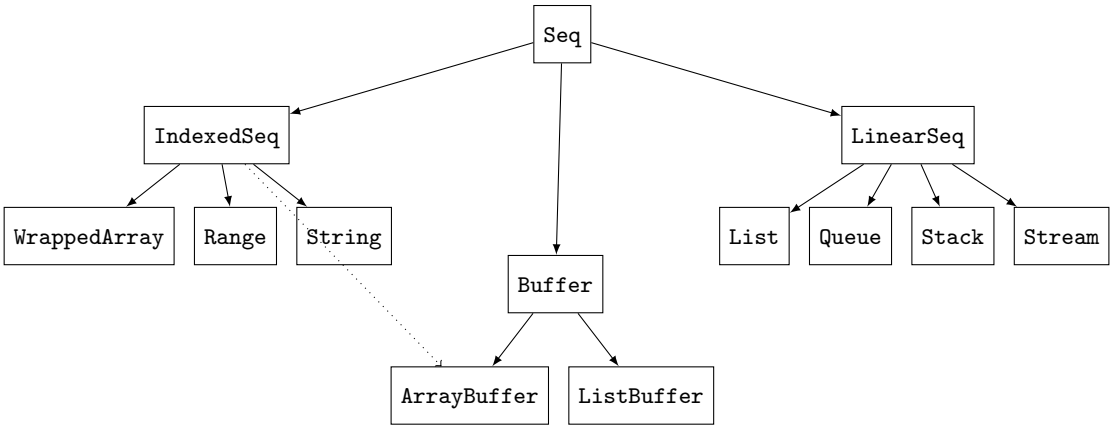
- `Traversable` trait lets you traverse an entire collection
- `Iterable` trait defines an iterator, which lets you loop through a collection's elements one at a time

Traversable

Methods

- concatenate collections with ++
- transform elements with `map` `flatMap` and `collect` ...
- convert to other collections with `toList` / `toMap` / `toSet` ...
- check size with `isEmpty` / `nonEmpty` / `size`
- access elements with `head` / `last` / `find` ...
- narrow down with `filter` / `collect` / `take` ...
- split with `partition` / `groupBy` / `span` ...
- fold with `foldLeft` / `reduce` / `aggregate` ...

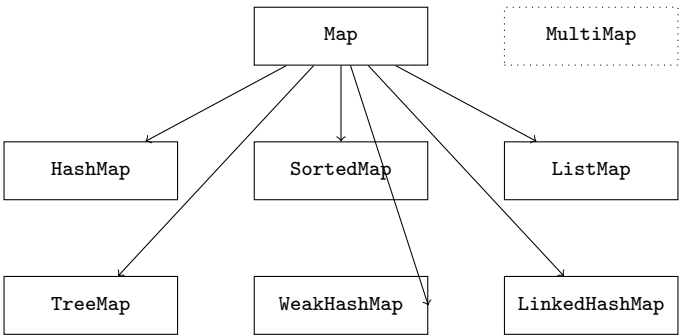
Sequences


Seq
IndexedSeq
LinearSeq
WrappedArray
Range
String
Buffer
List
Queue
Stack
Stream
ArrayBuffer
ListBuffer

Sequences

- `IndexedSeq` — indicates that random access of elements is efficient
- `LinearSeq` — can be efficiently split into head and tail components
- `Buffer` — efficient appending, prepending, or inserting new elements

Map


Map
MultiMap
HashMap
SortedMap
ListMap
TreeMap
WeakHashMap
LinkedHashMap

Map

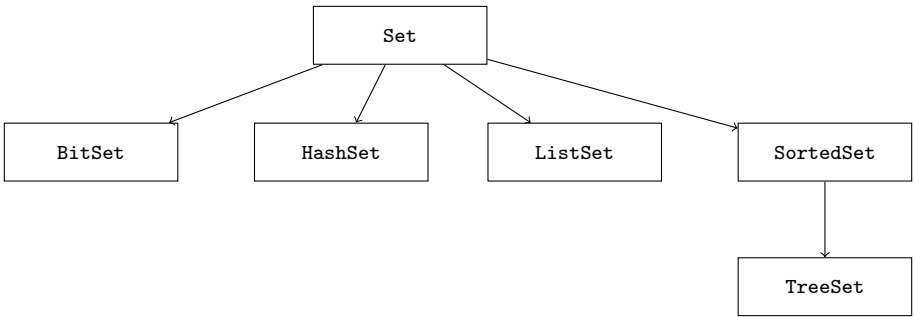
- is an `Iterable` consisting of pairs of keys and values
- can be constructed from a `Seq[Tuple2[_,_]]`:
  `Seq(1 -> "foo").toMap`
- `apply` throws exeption when key isn't found
- `get` returns an `Option`
- access subcollections via `keys` and `values`

Map

example

```
1 val m = Map(1 -> "a", 2 -> "b")
2 val newMap = m + (3 -> "foo")
3
4 m.contains(3) == false
5 m(3) // NoSuchElementException
6 m.get(3).isDefined == false
```

Set



Set

Sets are `Iterables` that contain no duplicate elements

- test: `contains` / `apply` / `subsetOf`
- add: `+` / `++`
- remove: `-` / `--`
- operations: `intersect` / `union` / `diff`

Predef

The `Predef` object provides definitions that are accessible in all Scala compilation units without explicit qualification

Predef

The `Predef` object provides definitions that are accessible in all Scala compilation units without explicit qualification

- methods: `??? assert print classOf ...`
- aliases: `Class Function Map Set String -> ...`
- implicits: `Long2long long2Long genericArrayOps intArrayOps wrapString ...`

Java collections

Scala vs Java vs Kotlin

- Java's collections are a part of JRE

Scala vs Java vs Kotlin

- Java's collections are a part of JRE
- Kotin library re-uses Java collections by adding extension methods

Scala vs Java vs Kotlin

- Java's collections are a part of JRE
- Kotin library re-uses Java collections by adding extension methods
- Scala implememnts own collections(except for `Array`)

Scala vs Java vs Kotlin

- Java's collections are a part of JRE
- Kotin library re-uses Java collections by adding extension methods
- Scala implememnts own collections(except for `Array`)
- Java $\leftrightarrow$ Scala ?

Conversion

Two-way

```
Iterator                <=>     java.util.Iterator
Iterator                <=>     java.util.Enumeration
Iterable                <=>     java.lang.Iterable
Iterable                <=>     java.util.Collection
mutable.Buffer          <=>     java.util.List
mutable.Set             <=>     java.util.Set
mutable.Map             <=>     java.util.Map
mutable.ConcurrentMap   <=>     java.util.concurrent.ConcurrentMap
```

Conversion

One-way

```
Seq           =>    java.util.List
mutable.Seq   =>    java.util.List
Set           =>    java.util.Set
Map           =>    java.util.Map
```

collection.JavaConverters

- `import collection.JavaConverters._` [2]
- conversions work by setting up a "wrapper" object that forwards all operations to the underlying collection object
- collections are never copied when converting between Java and Scala
- attempting to mutate immutabe collection from java will yield `UnsupportedOperationException`

collection.JavaConverters

Example

```scala
import collection.mutable._
import collection.JavaConverters._

val jul: java.util.List[Int] = ArrayBuffer(1, 2, 3).asJava
val buf: Seq[Int] = jul.asScala
val m: java.util.Map[String, Int] = HashMap("abc" -> 1, "hello" -> 2).asJava

val jul = List(1, 2, 3).asJava
jul.add(7) // throws UnsupportedOperationException
```

Scala Library: Best and Worst Practices

Guidelines

Basic usage rules for Scala collections [3]

Guidelines

Basic usage rules for Scala collections [3]

- prefer using immutable collections

Guidelines

Basic usage rules for Scala collections [3]

- prefer using immutable collections
- use the mutable namespace explicitly:

```
val set = mutable.Set()
```

Guidelines

Basic usage rules for Scala collections [3]

- prefer using immutable collections
- use the mutable namespace explicitly:

```
val set = mutable.Set()
```

- use the default constructor for the collection type:

```
val seq = Seq(1, 2, 3)
```

Guidelines

Basic usage rules for Scala collections [3]

- prefer using immutable collections
- use the mutable namespace explicitly:

```
val set = mutable.Set()
```

- use the default constructor for the collection type:

```
val seq = Seq(1, 2, 3)
```

- prefer default collections over specific ones

Sequences

- Prefer length to size for arrays

  `Array.size` calls are still implemented via implicit conversion

Sequences

- Prefer length to size for arrays

  `Array.size` calls are still implemented via implicit conversion

- Create empty collections explicitly:

```
1 Seq[T]()          ✗
2 Seq.empty[T]      ✓
```

Sequences

- Prefer length to size for arrays

  `Array.size` calls are still implemented via implicit conversion

- Create empty collections explicitly:

```
1 Seq[T]()          ✗
2 Seq.empty[T]      ✓
```

- Don't negate emptiness-related properties

```
1 !seq.isEmpty      ✗
2 seq.nonEmpty      ✓
```

Sequences

- Prefer length to size for arrays

  `Array.size` calls are still implemented via implicit conversion

- Create empty collections explicitly:

```
1 Seq[T]()          ✗
2 Seq.empty[T]      ✓
```

- Don't negate emptiness-related properties

```
1 !seq.isEmpty      ✗
2 seq.nonEmpty      ✓
```

- Don't compute length for emptiness check

```
1 seq.length > 0    ✗
2 seq.nonEmpty      ✓
```

Eqality

- Don't rely on == to compare array contents

```
1 array1 == array2                ✗
2 array1.sameElements(array2)     ✓
```

Eqality

- Don't rely on == to compare array contents

```
1 array1 == array2                ✗
2 array1.sameElements(array2)     ✓
```

- Don't check equality between collections in different categories

```
1 seq == set          ✗
2 seq.toSet == set    ✓
```

Indexing

- use `head` and `last` instead of index access

Indexing

- use `head` and `last` instead of index access
- use `headOption` and `lastOption` instead of bound checks

Indexing

- use `head` and `last` instead of index access
- use `headOption` and `lastOption` instead of bound checks
- use `indices` instead of manual `Range` construction

Existence

- Don't use equality predicate to check element presence

```
1 seq.exists(_ == x)   ✗
2 seq.contains(x)      ✓
```

Existence

- Don't use equality predicate to check element presence

```
1 seq.exists(_ == x)   ✗
2 seq.contains(x)      ✓
```

- Use exists for everything else

```
1 seq.count(p) > 0          ✗
2 seq.filter(p).nonEmpty    ✗
3 seq.find(p).isDefined     ✗
4
5 seq.exists(p)             ✓
```

Filtering

- Don't negate filter predicate

```
1 seq.filter(!p)      ✗
2 seq.filterNot(p)    ✓
```

Filtering

- Don't negate filter predicate

```
1 seq.filter(!p)      ✗
2 seq.filterNot(p)    ✓
```

- Don't resort to filtering to count elements

```
1 seq.filter(p).length  ✗
2 seq.count(p)          ✓
```

Filtering

- Don't negate filter predicate

```
1 seq.filter(!p)      ✗
2 seq.filterNot(p)    ✓
```

- Don't resort to filtering to count elements

```
1 seq.filter(p).length  ✗
2 seq.count(p)          ✓
```

- Don't use filtering to find first occurrence

```
1 seq.filter(p).headOption  ✗
2 seq.find(p)               ✓
```

Sorting

- Don't sort by a property manually

```
1 seq.sortWith(_.property <  _.property) ✗
2 seq.sortBy(_.property)                 ✓
```

Sorting

- Don't sort by a property manually

```
1 seq.sortWith(_.property <  _.property) ✗
2 seq.sortBy(_.property)                 ✓
```

- Perform reverse sorting in one step

```
1 seq.sorted.reverse                ✗
2 seq.sorted(Ordering[T].reverse)   ✓
```

Sorting

- Don't sort by a property manually

```
1 seq.sortWith(_.property <  _.property) ✗
2 seq.sortBy(_.property)                 ✓
```

- Perform reverse sorting in one step

```
1 seq.sorted.reverse                ✗
2 seq.sorted(Ordering[T].reverse)   ✓
```

- Don't use sorting to find the smallest or biggest element

```
1 seq.sorted.reverse.head       ✗
2 seq.sortBy(_.property).head   ✗
3 seq.max                       ✓
4 seq.minBy(_.property)         ✓
```

TBC

More on Paval Fatin's blog [1]

[1] https://pavelfatin.com/scala-collections-tips-and-tricks/

[2] https://docs.scala-lang.org/overviews/collections/conversions-between-java-and-scala-collections.html

[3] https://twitter.github.io/effectivescala/#Collections-Use