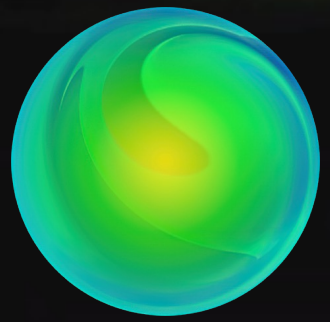
Измеряем sequences



Максим Сидоров,

Системные сервисы, Salute TV

SBER DEVICES

StarOS – операционная система для всех умных устройств Сбера

3 года назад

— 2019-2020 внедрение kotlin в «сбербанк-онлайн»

— Тогда sequence проигрывали коллекциям на небольших списках

— Спорное правило «**CouldBeSequence**» в detect

— Что то поменялось или я просто недостаточно хорошо их померил?

О чем этот доклад

Сравнение производительности работы sequence и обычных коллекций

— Рассмотрим разницу в подходах (последовательная vs ленивая)
— Как это работает внутри
— Измерим, стоит ли овчинка выделки?
— И немного допилим напильником

Сравнение способов обработки коллекций

Collection

— Каждое преобразование выполняется в своем отдельном цикле

— На каждое преобразование создается новая, промежуточная коллекция

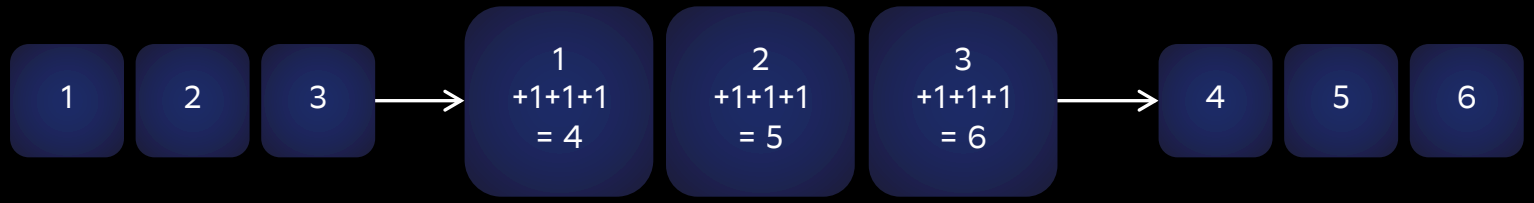
```
sourceCollection
    .map { it + 1 }
    .map { it + 1 }
    .map { it + 1 }
```

```
val result = ArrayList<Int>(sourceCollection.size)
sourceCollection.forEach { result.add(it + 1) }
return result
```

Sequence

— Все преобразования выполняются последовательно, за один проход цикла

— Нет выделения памяти под хранение промежуточных результатов



Что такое Sequence

Sequence – это просто интерфейс обертка для доступа к итератору

```
public interface Sequence<out T> {

    Returns an Iterator that returns the values from the sequence.

    Throws an exception if the sequence is constrained to be iterated once and iterator is invoked
    the second time.

    public operator fun iterator(): Iterator<T>
}
```

```
public fun <T> Iterable<T>.asSequence(): Sequence<T> {
    return Sequence { this.iterator() }
}
```

Создание sequence – это анонимный класс, содержащий ссылку на итератор коллекции

Да, я капитан очевидность...

Давайте вспомним **итератор**

(итераторы правят миром, по крайней мере миром sequence)

```
An iterator over a collection or another entity that can be represented as a sequence of elements.
Allows to sequentially access the elements.
public interface Iterator<out T> {

    Returns the next element in the iteration.

    public operator fun next(): T


    Returns true if the iteration has more elements.

    public operator fun hasNext(): Boolean
}
```

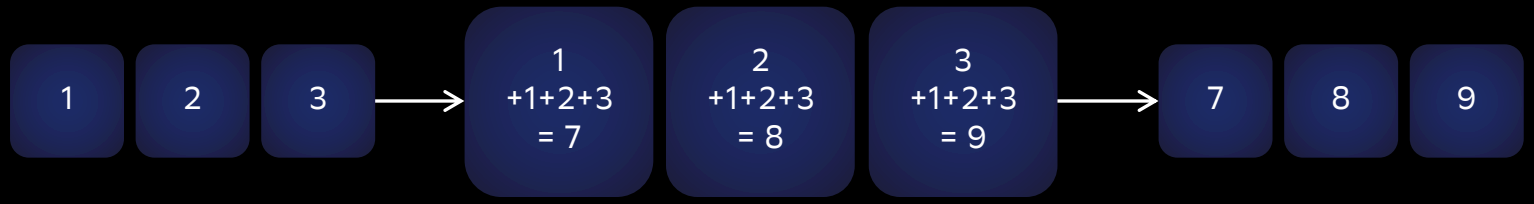
Как работает эта ленивая магия внутри

Каждое преобразование – это декоратор над итератором оригинальной коллекции

```
sourceCollection.asSequence()
    .map { it + 1 }
    .map { it + 2 }
    .map { it + 3 }
    .toList()
```

```
val resultIterator =
    MapIterator( { it + 3 },
        MapIterator( { it + 2 },
            MapIterator( { it + 1 },
                sourceCollection.iterator()
            )
        )
    )

val result = mutableListOf<Int>()
resultIterator.forEach { result.add(it) }
```

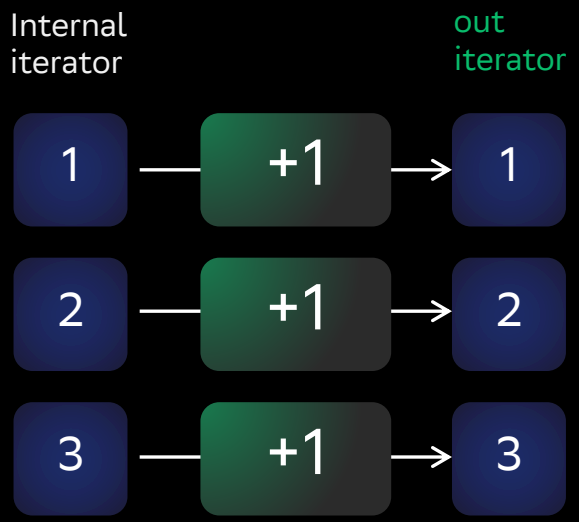


Устройство декоратора для map

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

    internal fun <E> flatten(iterator: (R) -> Iterator<E>): Sequence<E> {
        return FlatteningSequence<T, R, E>(sequence, transformer, iterator)
    }
}
```



Кажется сплошные плюсы?

— На первый взгляд в реализации нет подводных камней
— Внутри все до гениального просто
— Все построено на обычных итераторах

Надо бы померить

Жизненный опыт подсказывает, что где то есть подвох

Бесплатный сыр и все такое...

И опять же, наверняка «**все не так однозначно**»

Берем **Jetpack Benchmark** и за дело...

Измеряйте только на рутованных устройствах

— Замеры на не рутованных устройствах – это гадание на кофейной гуще!

— Используйте скрипт для установки фиксированной тактовой частоты (lockClock.sh)

— Правильней брать минимальные значения, так как по ним меньше разброс результатов

— Не управляйте порядком выполнения тестов через аннотации @Ignore, @OrderWith

Рутованное устройство

— Измерение на фиксированной тактовой частоте

— Разброс значений 3-7% по минимумам и 10-15% по медиане

Не рутованное устройство

— Частота процессора плавает во время теста

— Разброс значений 20-30% по минимумам и 30-60% по медиане

Измерения – 2 map { ... }

Тест на серию двух последовательных преобразований через **map**

```
fun map2_collection() {
    var transformedList: List<Int> = emptyList()
    benchmarkRule.measureRepeated { this: BenchmarkRule.Scope
        transformedList = sourceCollection
            .map { it + 1 }
            .map { it + 2 }
    }
    Assert.assertTrue(transformedList.isNotEmpty())
}
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	41 333	42 186	-2%
1 000	532 771	529 219	1%
10 000	5 726 766	5 603 207	2%
50 000	28 594 531	27 336 781	4%
100 000	57 318 875	54 838 584	4%

Измерения – 3 map { ... }

Тест на серию трех последовательных преобразований через **map**

```kotlin
@Test
fun map3_collection() {
    var transformedList: List<Int> = emptyList()
    benchmarkRule.measureRepeated { this: BenchmarkRule.Scope
        transformedList = sourceCollection
            .map { it + 1 }
            .map { it + 2 }
            .map { it + 3 }
    }
    Assert.assertTrue(transformedList.isNotEmpty())
}
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	62 094	52 029	16%
1 000	796 137	682 835	14%
10 000	8 625 083	7 072 713	18%
50 000	42 868 792	34 840 396	19%
100 000	86 425 959	69 078 709	20%

Измерения – 5 map { ... }

Тест на серию пяти последовательных преобразований через **map**

```
@Test
fun map5_collection() {
    var transformedList: List<Int> = emptyList()
    benchmarkRule.measureRepeated { this: BenchmarkRule.Scope
        transformedList = sourceCollection
            .map { it + 1 }
            .map { it + 2 }
            .map { it + 3 }
            .map { it + 4 }
            .map { it + 5 }
    }
    Assert.assertTrue(transformedList.isNotEmpty())
}
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	102 256	69 116	32%
1 000	1 327 987	968 823	27%
10 000	14 172 953	9 975 721	30%
50 000	71 466 459	49 866 708	30%
100 000	144 102 292	98 161 854	32%

Измерения – методика измерения сложных функций

— Преобразование из двух map дает равновесное состояние

— Заменяем одно преобразование на измеряемую функцию и измеряем разницу

Измерения – **2 map** { ... }

Размер списка	Collection (ns)	Sequence (ns)	%
100	41 333	42 186	-2%
1 000	532 771	529 219	1%
10 000	5 726 766	5 603 207	2%
50 000	28 594 531	27 336 781	4%
100 000	57 318 875	54 838 584	4%

Измерения – sort { ... }

Тест на сортировку промежуточных результатов

```
return sourceCollection.asSequence()
    .map { it + 1 }
    .sortedByDescending { it }
    .toList()
```

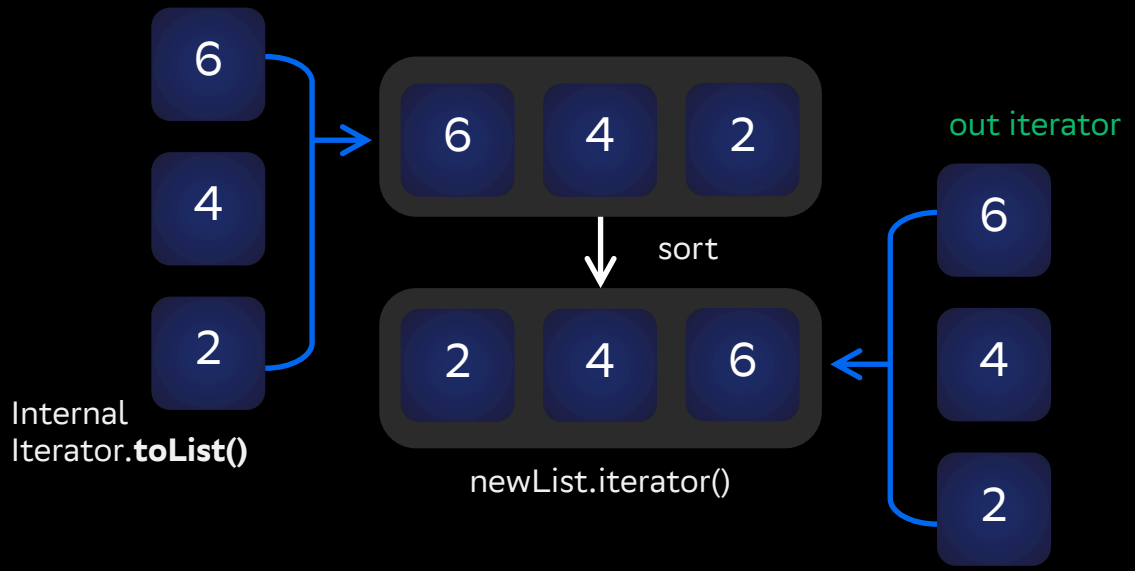
Размер списка	Collection (ns)	Sequence (ns)	%
100	149 580	180 167	-20%
1 000	3 214 330	3 563 508	-11%
10 000	45 126 896	49 879 000	-11%
50 000	289 200 750	315 998 501	-9%
100 000	662 256 355	711 287 875	-7%

Тест **2 map** { }

Размер списка	Collection (ns)	Sequence (ns)
100	41 333	42 186
1 000	532 771	529 219
10 000	5 726 766	5 603 207
50 000	28 594 531	27 336 781
100 000	57 318 875	54 838 584

Устройство декоратора для sortedBy

```kotlin
public fun <T> Sequence<T>.sortedWith(comparator: Comparator<in T>): Sequence<T> {
    return object : Sequence<T> {
        override fun iterator(): Iterator<T> {
            val sortedList = this@sortedWith.toMutableList()
            sortedList.sortWith(comparator)
            return sortedList.iterator()
        }
    }
}
```



Как работает sort

— Лениво выполняет все преобразования До и сохраняет результат в промежуточную коллекцию

— Сортирует полученную коллекцию

— И продолжает ленивую обработку уже на новой коллекции

```
sequenceOf( ...elements: 5, 4, 3, 2, 1)
    .map { it + 1 }
    .filter { it % 2 == 0 }
    .sortedBy { it }
    .map { it - 1 }
    .map { it + 1 }
```

```
val list = sequenceOf( ...elements: 5, 4, 3, 2, 1)
    .map { it + 1 }
    .filter { it % 2 == 0 }
    .toList()

return list.sortedBy { it }
    .map { it - 1 }
    .map { it + 1 }
```

Измерения – filter { ... }

Тест на filter, возвращающий 10% записей

```
return sourceCollection.asSequence()
    .map { it }
    .filter { it in 0 ≤ .. ≤ percent10 }
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	33 045	24 060	27%
1 000	394 953	292 209	26%
10 000	4 185 852	2 985 708	29%
50 000	20 955 136	15 083 892	28%
100 000	41 977 667	31 003 865	26%

Тест на filter, возвращающий 90% записей

```
return sourceCollection.asSequence()
    .map { it }
    .filter { it in 0 ≤ .. ≤ percent90 }
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	42 828	43 557	-2%
1 000	488 092	479 876	2%
10 000	5 014 653	4 862 738	3%
50 000	25 080 389	24 081 423	4%
100 000	50 261 272	48 016 813	4%

Устройство декоратора для filter

```
override fun iterator(): Iterator<T> = object : Iterator<T> {
    val iterator = sequence.iterator()
    var nextState: Int = -1 // -1 for unknown, 0 for done, 1 for contir
    var nextItem: T? = null

    private fun calcNext() {
        while (iterator.hasNext()) {
            val item = iterator.next()
            if (predicate(item) == sendWhen) {
                nextItem = item
                nextState = 1
                return
            }
        }
        nextState = 0
    }
```

```
    override fun next(): T {
        if (nextState == -1)
            calcNext()
        if (nextState == 0)
            throw NoSuchElementException()
        val result = nextItem
        nextItem = null
        nextState = -1
        @Suppress( ...names: "UNCHECKED_CAST")
        return result as T
    }

    override fun hasNext(): Boolean {
        if (nextState == -1)
            calcNext()
        return nextState == 1
    }
}
```

SBER DEVICES

Почему выигрыш зависит от количества возвращаемых записей?

```
override fun iterator(): Iterator<T> = object : Iterator<T> {
    val iterator = sequence.iterator()
    var nextState: Int = -1 // -1 for unknown, 0 for done, 1 for contin
    var nextItem: T? = null

    private fun calcNext() {
        while (iterator.hasNext()) {
            val item = iterator.next()
            if (predicate(item) == sendWhen) {
                1 nextItem = item
                2 nextState = 1
                return
            }
        }
     3 nextState = 0
    }

    override fun next(): T {
     4 if (nextState == -1)
            calcNext()
     5 if (nextState == 0)
            throw NoSuchElementException()
        val result = nextItem
     6 nextItem = null
     7 nextState = -1
        @Suppress( ...names: "UNCHECKED_CAST")
        return result as T
    }

    override fun hasNext(): Boolean {
     8 if (nextState == -1)
            calcNext()
        return nextState == 1
    }
}
```

Измерения – distinct { ... }

Оставляет 10% исходного списка

```
return sourceCollection.asSequence()
    .map { it * 10 / 100 }
    .distinctBy { it }
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	53 566	44 832	16%
1 000	538 987	451 603	16%
10 000	7 646 898	6 617 495	13%
50 000	43 246 521	39 631 886	8%
100 000	95 196 333	87 074 083	9%

Оставляет 90% исходного списка

```
return sourceCollection.asSequence()
    .map { it * 90 / 100 }
    .distinctBy { it }
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	83 113	100 762	-21%
1 000	1 005 813	1 169 785	-16%
10 000	10 851 783	12 856 475	-18%
50 000	60 652 896	69 893 855	-15%
100 000	129 569 604	146 594 000	-13%

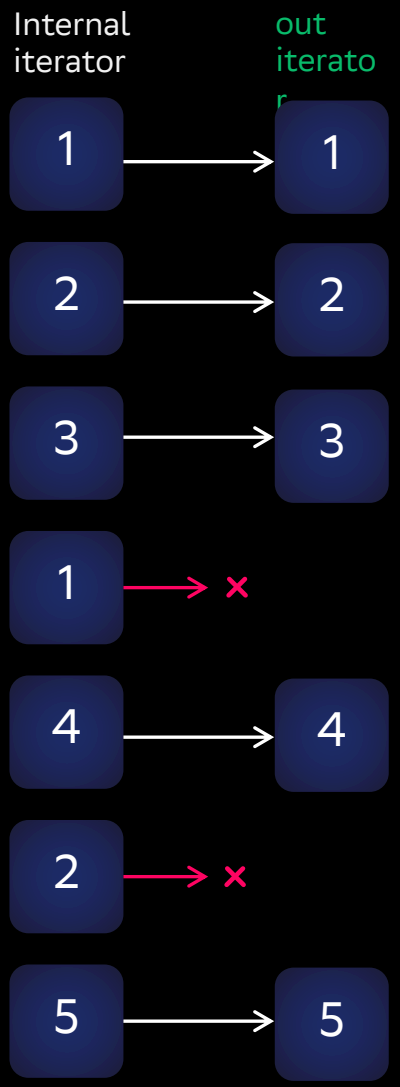
Устройство декоратора для distinct

```
private class DistinctIterator<T, K>(
    private val source: Iterator<T>,
    private val keySelector: (T) -> K
) : AbstractIterator<T>() {
    private val observed = HashSet<K>()

    override fun computeNext() {
        while (source.hasNext()) {
            val next = source.next()
            val key = keySelector(next)

            if (observed.add(key)) {
                setNext(next)
                return
            }
        }

        done()
    }
```



Измерения – take { ... }

Тест на take, возвращает 10% записей

```
return sourceCollection.asSequence()
    .map { it + 1 }
    .take( countPercent10 )
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	23 016	3 999	83%
1 000	275 794	38 106	86%
10 000	2 954 285	381 353	87%
50 000	14 981 667	1 969 455	87%
100 000	29 886 927	3 977 857	87%

Тест на take, возвращает 90% записей

```
return sourceCollection.asSequence()
    .map { it + 1 }
    .take( countPercent90 )
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	36 368	29 986	18%
1 000	421 734	342 892	19%
10 000	4 504 151	3 456 383	23%
50 000	22 658 014	17 092 458	25%
100 000	45 291 084	34 242 125	24%

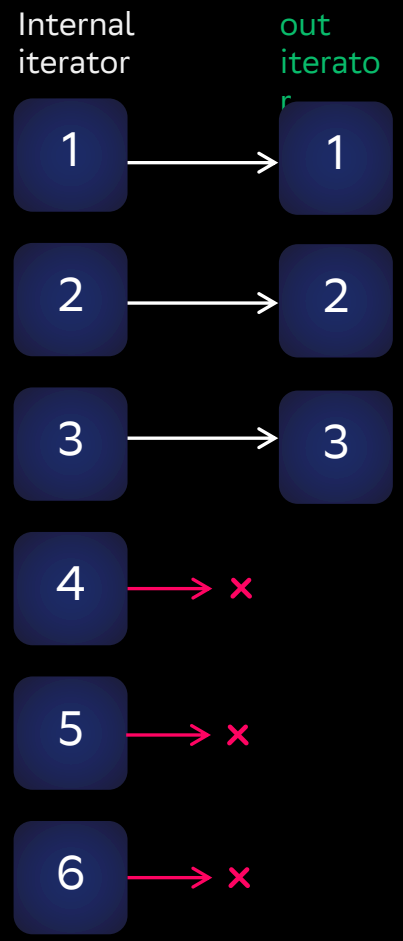
Устройство декоратора для take (взять n первых элементов)

```
override fun iterator(): Iterator<T> = object : Iterator<T> {
    var left = count
    val iterator = sequence.iterator()

    override fun next(): T {
        if (left == 0)
            throw NoSuchElementException()
        left--
        return iterator.next()
    }

    override fun hasNext(): Boolean {
        return left > 0 && iterator.hasNext()
    }
}
```

```
listOf(1,2,3,4,5,6)
    .take(3)
```



Измерения – drop { ... }

Тест на drop, удаляющий **10%** записей

```
return sourceCollection.asSequence()
    .map { it + 1 }
    .drop( countPercent10 )
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	33 701	31 629	6%
1 000	385 249	361 036	6%
10 000	4 232 115	3 681 223	13%
50 000	21 088 674	18 295 245	13%
100 000	42 020 209	36 519 083	13%

Тест на drop, удаляющий **90%** записей

```
return sourceCollection.asSequence()
    .map { it + 1 }
    .drop( countPercent90 )
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	23 993	19 886	17%
1 000	282 420	191 700	32%
10 000	3 052 168	2 032 454	33%
50 000	15 293 025	9 922 920	35%
100 000	30 413 583	19 863 292	35%

drop устроен точно также, как и take

Тогда откуда такая разница в результатах?

drop

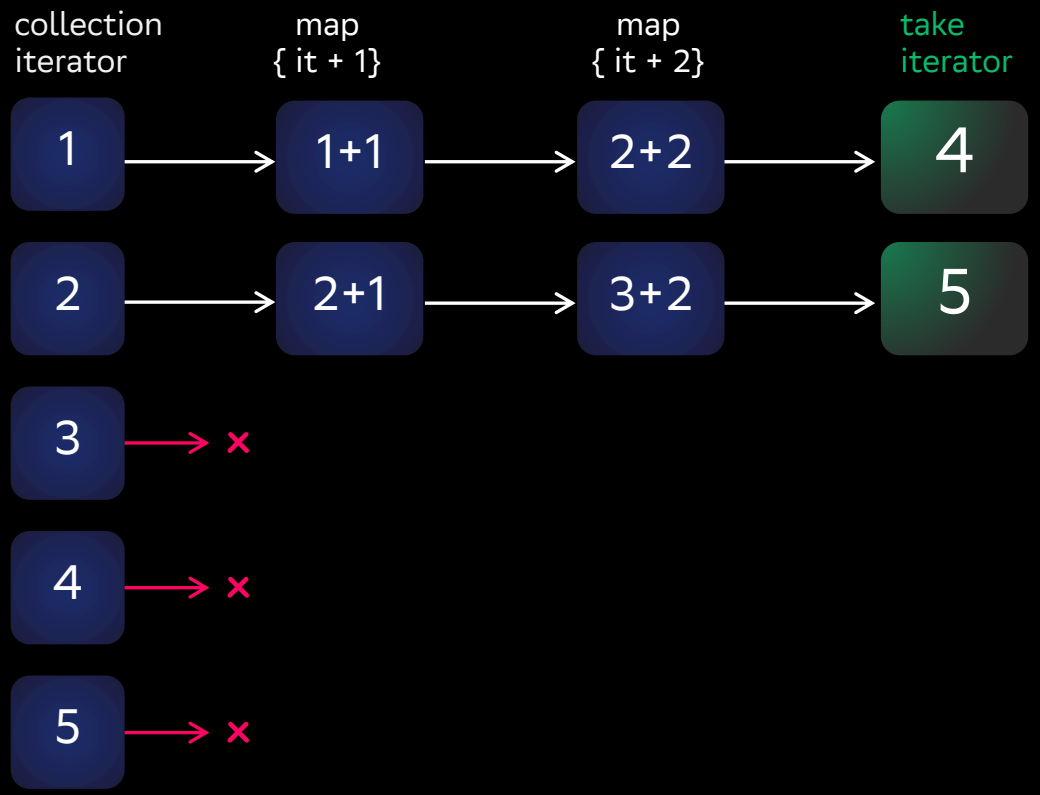
Размер списка	Collection (ns)	Sequence (ns)	%
100	23 993	19 886	17%
1 000	282 420	191 700	32%
10 000	3 052 168	2 032 454	33%
50 000	15 293 025	9 922 920	35%
100 000	30 413 583	19 863 292	35%

take

Размер списка	Collection (ns)	Sequence (ns)	%
100	23 813	4 020	83%
1 000	285 328	38 688	86%
10 000	3 105 021	387 201	88%
50 000	15 462 725	2 034 473	87%
100 000	30 792 854	4 101 179	87%

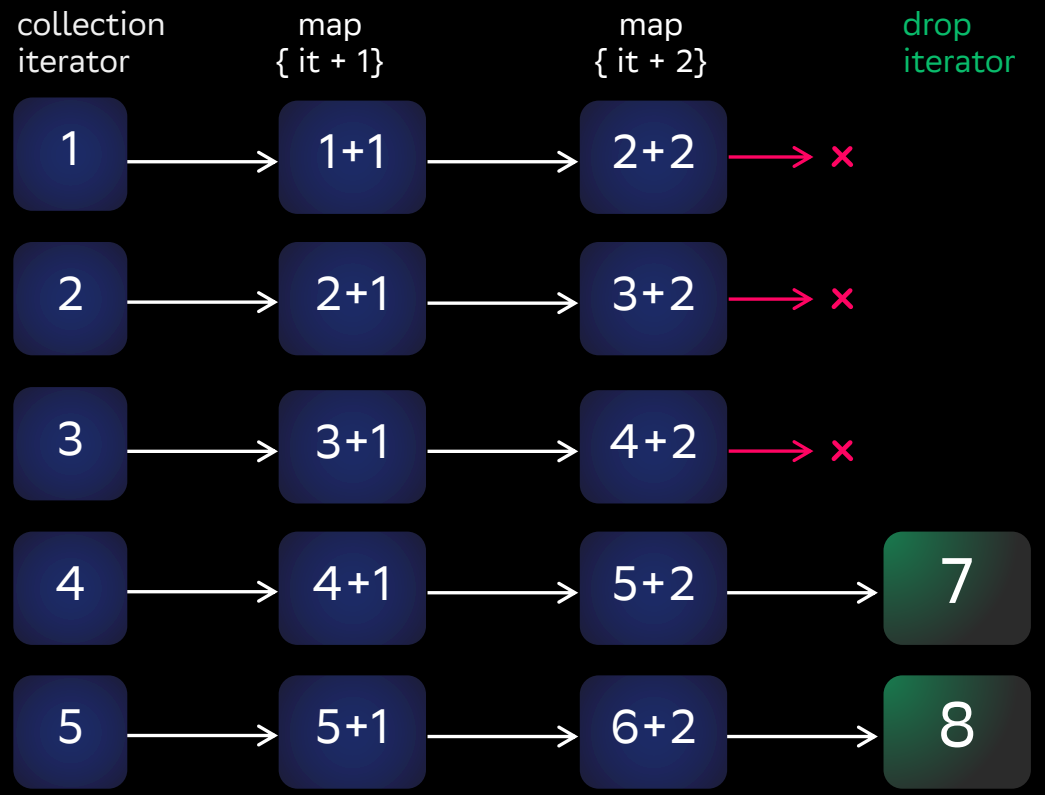
take

```
sequenceOf( ...elements: 1, 2, 3, 4, 5)
    .map { it + 1 }
    .map { it + 2 }
    .take( n: 2)
```



drop

```
sequenceOf( ...elements: 1, 2, 3, 4, 5)
    .map { it + 1 }
    .map { it + 2 }
    .drop( n: 3)
```



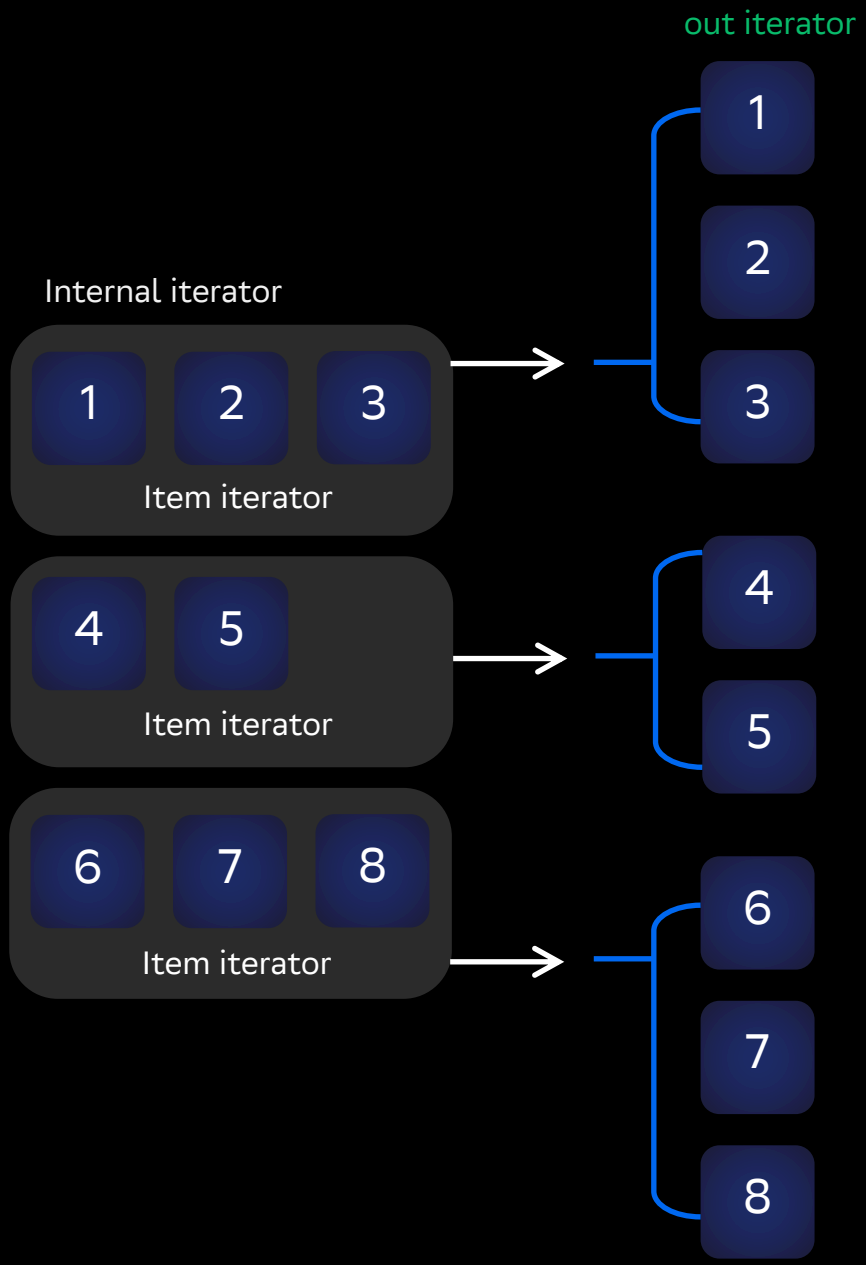
Измерения – flatten { ... }

Тест на разворачивание списка списков (внутренний список 10 элементов)

```
fun flatten_sequence(
    sourceCollection: List<List<Int>>
): List<Int> {
    return sourceCollection.asSequence() Sequence<List<Int>>
        .map { it }
        .flatten() Sequence<Int>
        .toList()
}
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	147 451	414 376	181%
1 000	1 512 504	4 167 416	176%
10 000	15 345 992	41 305 365	169%
50 000	75 917 917	205 262 897	170%
100 000	142 455 042	401 473 313	182%

Как работает flatten



Устройство функции flatten

(Разворачивает список списков в один линейный список)

```
internal class FlatteningSequence<T, R, E>
constructor(
    private val sequence: Sequence<T>,
    private val transformer: (T) -> R,
    private val iterator: (R) -> Iterator<E>
) : Sequence<E> {
    override fun iterator(): Iterator<E> = obj
        val iterator = sequence.iterator()
        var itemIterator: Iterator<E>? = null

        override fun next(): E {
            if (!ensureItemIterator())
                throw NoSuchElementException()
            return itemIterator!!.next()
        }

        override fun hasNext(): Boolean {
            return ensureItemIterator()
        }
```

```
private fun ensureItemIterator(): Boolean {
    if (itemIterator?.hasNext() == false)
        itemIterator = null

    while (itemIterator == null) {
        if (!iterator.hasNext()) {
            return false
        } else {
            val element = iterator.next()
            val nextItemIterator = iterator(transformer(element))
            if (nextItemIterator.hasNext()) {
                itemIterator = nextItemIterator
                return true
            }
        }
    }
    return true
}
```

SBER DEVICES

Измерения – plus { ... }

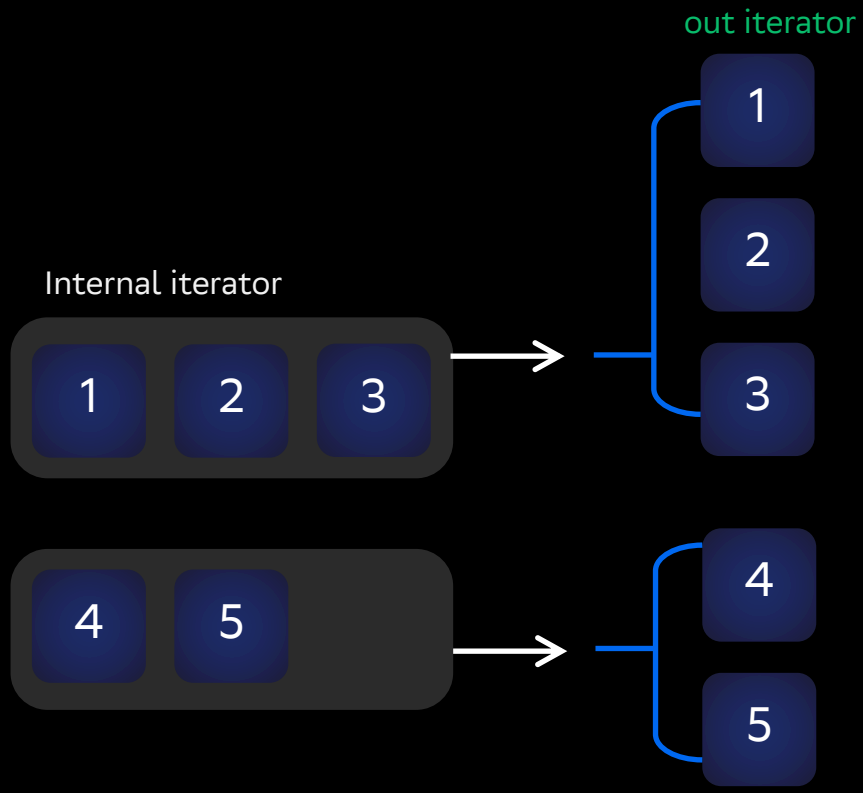
Тест на добавление элементов в коллекцию

```
return sourceCollection.asSequence()
    .map { it }
    .plus(sourceCollection)
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	28 685	84 539	-195%
1 000	325 547	902 038	-177%
10 000	3 400 506	8 951 631	-163%
50 000	16 999 468	43 911 542	-158%
100 000	33 640 031	88 627 167	-163%

Устройство декоратора для plus

```
listOf(1,2,3)
    .plus( listOf(4,5) )
```




```
public operator fun <T> Sequence<T>.plus(elements: Iterable<T>): Sequence<T> {
    return sequenceOf( ...elements: this, elements.asSequence()).flatten()
}
```

Измерения – minus { ... }

Удаляет **10%** исходного списка

```
return sourceCollection.asSequence()
    .map { it }
    .minus(minusCollection_10perc)
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	56 503	55 714	1%
1 000	623 336	615 714	1%
10 000	6 832 593	6 839 409	0%
50 000	37 601 979	39 143 896	-4%
100 000	80 364 834	86 419 125	-8%

Удаляет **90%** исходного списка

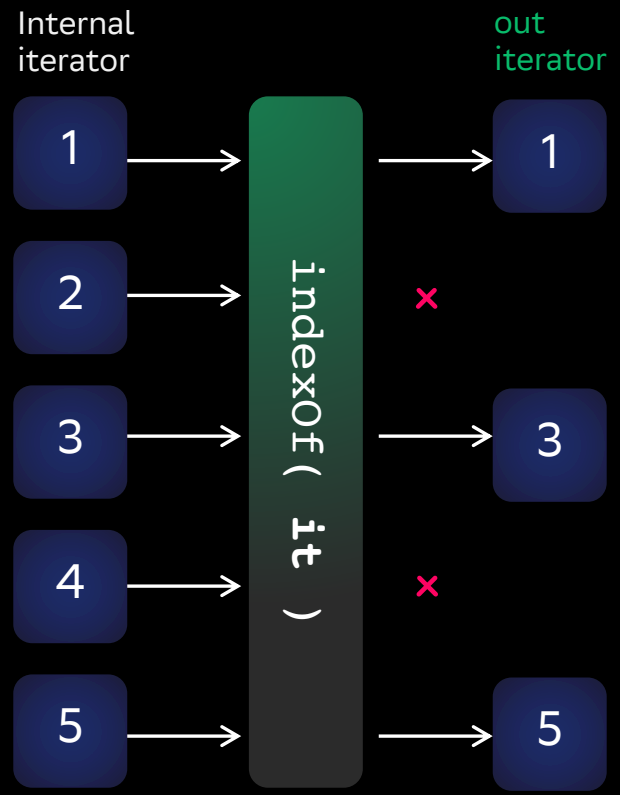
```
return sourceCollection.asSequence()
    .map { it }
    .minus(minusCollection_90perc)
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	47 017	37 997	19%
1 000	592 271	507 700	14%
10 000	8 578 409	7 801 112	9%
50 000	48 183 813	45 459 104	6%
100 000	101 262 667	94 718 000	6%

Устройство декоратора для minus (сделано на основе filter)

```
listOf(1,2,3,4,5)
   .minus( listOf(2,4) )
```

```
public operator fun <T> Sequence<T>.minus(elements: Iterable<T>): Sequence<T
    return object: Sequence<T> {
        override fun iterator(): Iterator<T> {
            val other = elements.convertToSetForSetOperation()
            if (other.isEmpty())
                return this@minus.iterator()
            else
                return this@minus.filterNot { it in other }.iterator()
        }
    }
}
```



listOf(2,4)

Измерения – zip { ... }

Тест на объединение двух источников

```
fun zip_sequence(sourceCollection: List<Int>): List<Int> {
    return sourceCollection.asSequence()
        .map { it + 1 }
        .zip(sourceCollection.asSequence()) { v1, v2 ->
            v1 + v2
        }
        .toList()
}
```

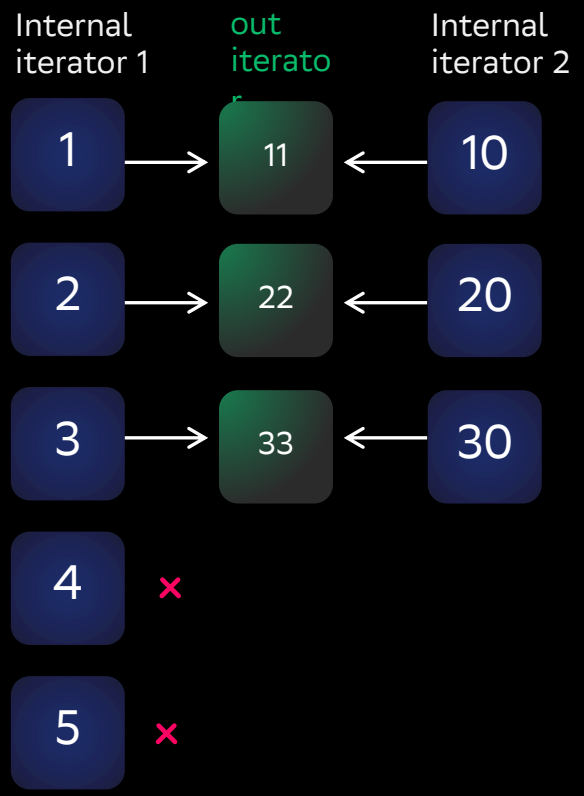
Размер списка	Collection (ns)	Sequence (ns)	%
100	53 215	52 888	1%
1 000	619 275	601 973	3%
10 000	6 765 979	6 306 892	7%
50 000	34 140 229	31 015 625	9%
100 000	68 545 855	60 928 042	11%

Устройство декоратора для merge (объединение 2 sequence)

```
internal class MergingSequence<T1, T2, V>
constructor(
    private val sequence1: Sequence<T1>,
    private val sequence2: Sequence<T2>,
    private val transform: (T1, T2) -> V
) : Sequence<V> {
    override fun iterator(): Iterator<V> = object : Iterator<V> {
        val iterator1 = sequence1.iterator()
        val iterator2 = sequence2.iterator()
        override fun next(): V {
            return transform(iterator1.next(), iterator2.next())
        }

        override fun hasNext(): Boolean {
            return iterator1.hasNext() && iterator2.hasNext()
        }
    }
}
```

```
sequenceOf(1,2,3,4,5)
   .zip(sequenceOf(10,20,30))
      { a, b -> a + b}
```



Измерения – groupBy { ... }

Оставляет **10%** исходного списка

```
return sourceCollection.asSequence()
    .map { it + 1 }
    .groupBy { it * 10 / 100 }
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	64 973	50 670	22%
1 000	716 454	579 001	19%
10 000	9 308 440	7 863 016	16%
50 000	56 283 417	51 490 709	9%
100 000	121 278 750	110 725 750	9%

Оставляет **90%** исходного списка

```
return sourceCollection.asSequence()
    .map { it + 1 }
    .groupBy { it * 90 / 100 }
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	138 715	126 220	9%
1 000	1 605 923	1 476 468	8%
10 000	17 802 682	16 516 791	7%
50 000	94 385 813	88 222 730	7%
100 000	196 906 730	185 750 396	6%

Устройство groupBy (сравниваем с коллекциями)

Sequence

```
inline fun <T, K> Sequence<T>.groupByTo(destination: MutableMap<K, MutableList<T>>, keySelector: (T) -> K
): Map<K, List<T>> {
    for (element in this) {
        val key = keySelector(element)
        val list = destination.getOrPut(key) { ArrayList() }
        list.add(element)
    }
    return destination
}
```

Найди 10 отличий

Collections

```
inline fun <T, K> Iterable<T>.groupByTo(destination: MutableMap<K, MutableList<T>>, keySelector: (T) -> K
): Map<K, List<T>> {
    for (element in this) {
        val key = keySelector(element)
        val list = destination.getOrPut(key) { ArrayList() }
        list.add(element)
    }
    return destination
}
```

Измерения – associateBy { ... }

Тест на преобразование списка в map

```
return sourceCollection.asSequence()
    .map { it + 1 }
    .associateBy { it * 10 / 100 }
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	52 987	33 198	37%
1 000	565 668	399 033	29%
10 000	7 383 587	5 577 064	24%
50 000	42 322 355	34 233 552	19%
100 000	92 500 625	75 998 125	18%

Устройство associateBy (сравниваем с коллекциями)

Sequence

```
inline fun <T, K> Sequence<T>.associateByTo(destination: MutableMap<K, T>, keySelector: (T) -> K): Map<K, T>
    for (element in this) {
        destination[keySelector(element)] = element
    }
    return destination
}
```

Найди 10 отличий

Collections

```
inline fun <T, K> Iterable<T>.associateByTo(destination: MutableMap<K, T>, keySelector: (T) -> K): Map<K, T>
    for (element in this) {
        destination[keySelector(element)] = element
    }
    return destination
}
```

Измерения – chunked { ... }

Тест на сворачивание списка в список списков
(возврат списка порциями по 100 элементов)

```
return sourceCollection.asSequence() Sequence<Int>
    .map { it + 1 }
    .chunked(count) Sequence<List<Int>>
    .toList()
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	46 775	74 951	-60%
1 000	391 686	446 668	-14%
10 000	3 909 134	3 959 575	-1%
50 000	19 408 088	19 955 328	-3%
100 000	38 675 740	39 789 334	-3%

Измерения – сложное преобразование { map + filter + map + distinct + map }

Оставляет **10%** исходного списка

```
return sourceCollection
    .map { if (it % percent10 == 0) null else :
    .filterNotNull() List<Int>
    .map { it * 10 / 100 }
    .distinctBy { it }
    .map { it + 1 }
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	88 536	80 143	9%
1 000	1 016 568	838 390	18%
10 000	12 818 476	11 594 958	10%
50 000	69 441 167	65 913 250	5%
100 000	146 355 438	138 842 709	5%

Оставляет **90%** исходного списка

```
return sourceCollection
    .map { if (it % percent10 == 0) null else :
    .filterNotNull() List<Int>
    .map { it * 90 / 100 }
    .distinctBy { it }
    .map { it + 1 }
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	140 646	146 642	-4%
1 000	1 678 383	1 838 271	-10%
10 000	17 951 730	19 954 936	-11%
50 000	96 481 604	105 065 167	-9%
100 000	200 193 313	216 806 292	-8%

Измерения – реальное преобразование { ... }

И наконец тест реального преобразования из кода нашего проекта

```
// return list of products with default photo Pair<productUid,
fun reality_collection(realData: DataFactory):
    List<Pair<String, String?>>
{
    return realData.sessionManager.productCategories List<Product
        .map { it.categoryName } List<String>
        .mapNotNull { realData.productRepository.getCategoryPr
        .flatten() List<Product>
        .distinctBy { it.productUid }
        .map { product ->
            product.productUid to product.photos.firstOrNull { it.isDefault }?.url
        }
}
```

Размер списка	Collection (ns)	Sequence (ns)	%
100	891 207	1 201 871	-35%
1 000	5 265 883	7 175 659	-36%
10 000	37 733 636	53 586 562	-42%
50 000	179 987 000	256 107 209	-42%
100 000	354 953 042	508 049 334	-43%

Вывод

— Выигрыш можно получить, если не использовать flatten, plus, sort, distinct
— Чем больше преобразований, тем больше выигрыш

Выигрыш

функция	К
map	⬆
filter	⬆
groupBy	⬆
associateBy	⬆
take	⬆
drop	⬆
zip	⬆
minus	⬆

Проигрыш

функция	К	примечание
sort	⬇⬇	Тяжелая операция и стабильный значительный проигрыш
flatten	⬇⬇	Огромный проигрыш и достаточно тяжелая операция
plus	⬇⬇	Огромный проигрыш (сделана на основе flatten)
distinct	⬆⬇	Если коллекция содержит много уникальных значений, то будет проигрыш
chunked	⬇	Легкая операция и небольшой проигрыш

Почему flatten проигрывает в разы?

В первую очередь
из за копирования
элементов по одному

Размер списка	Collection (ns)	Sequence (ns)	%
100	147 451	414 376	-181%
1 000	1 512 504	4 167 416	-176%
10 000	15 345 992	41 305 365	-169%
50 000	75 917 917	205 262 897	-170%
100 000	142 455 042	401 473 313	-182%

```
public fun <T> Iterable<Iterable<T>>.flatten(): List<T> {
    val result = ArrayList<T>()
    for (element in this) {
        result.addAll(element)
    }
    return result
}
```

Коллекции сразу расширяют массив
на нужное число элементов, sequence так
не могут(

Оптимизируем flatten (ну что здесь можно сделать???)

```
internal class FlatteningSequence<T, R, E>
constructor(
    private val sequence: Sequence<T>,
    private val transformer: (T) -> R,
    private val iterator: (R) -> Iterator<E>
) : Sequence<E> {
    override fun iterator(): Iterator<E> = obj
        val iterator = sequence.iterator()
        var itemIterator: Iterator<E>? = null

        override fun next(): E {
            if (!ensureItemIterator())
                throw NoSuchElementException()
            return itemIterator!!.next()
        }

        override fun hasNext(): Boolean {
            return ensureItemIterator()
        }
```

```
private fun ensureItemIterator(): Boolean {
    if (itemIterator?.hasNext() == false)
        itemIterator = null

    Iterator var10000 = this.itemIterator;
    if (var10000 != null) {
       if (!var10000.hasNext()) {
          this.itemIterator = null;
       }
    }
                                          ansformer(element))
                itemIterator = nextItemIterator
                return true
            }
        }
    }
    return true
}
```

SBER DEVICES

Оптимизируем flatten (избавляемся от проверок на null)

```
private object EmptyIterator: Iterator<Nothing> {
    override fun hasNext(): Boolean = false
    override fun next(): Nothing = throw NoSuchElementException()
}

override fun iterator(): Iterator<E> = object : Iterator<E> {
    private val iterator = sequence.iterator()
    private var itemIterator: Iterator<E> = EmptyIterator // optimization for exclude nullable variable
    private var state: Int = UNDEFINED_STATE

    override fun next(): E {
        if (state == UNDEFINED_STATE) { // optimized typical cause hasNext() + next()
            ensureItemIterator()
        }
        state = UNDEFINED_STATE
        return itemIterator.next()
    }
```

Оптимизируем flatten (избавляемся от лишнего вызова ensureItemIterator)

```
internal class FlatteningSequence<T, R, E>
constructor(
    private val sequence: Sequence<T>,
    private val transformer: (T) -> R,
    private val iterator: (R) -> Iterator<E>
) : Sequence<E> {
    override fun iterator(): Iterator<E> = obj
        val iterator = sequence.iterator()
        var itemIterator: Iterator<E>? = null

        override fun next(): E {
            if (!ensureItemIterator())
                throw NoSuchElementException()
            return itemIterator!!.next()
        }

        override fun hasNext(): Boolean {
            return ensureItemIterator()
        }
```

```
private fun ensureItemIterator(): Boolean {
    if (itemIterator?.hasNext() == false)
        itemIterator = null

    while (itemIterator == null) {
        if (!iterator.hasNext()) {
            return false
        } else {
            val element = iterator.next()
            val nextItemIterator = iterator(transformer(element))
            if (nextItemIterator.hasNext()) {
                itemIterator = nextItemIterator
                return true
            }
        }
    }
    return true
}
```

SBER DEVICES

Оптимизируем flatten (избавляемся от лишнего вызова insureItemIterator)

```
override fun iterator(): Iterator<E> = object : Iterator<E> {
    private val iterator = sequence.iterator()
    private var itemIterator: Iterator<E> = EmptyIterator // optimization for exclude nullable variable
    private var state: Int = UNDEFINED_STATE // { UNDEFINED_STATE, HAS_NEXT_ITEM, HAS_FINISHED }

    override fun next(): E {
        if (state == UNDEFINED_STATE) { // optimized typical cause hasNext() + next()
            ensureItemIterator()
        }
        state = UNDEFINED_STATE
        return itemIterator.next()
    }

    override fun hasNext(): Boolean {
        return when (state) { // optimized cause for multiple call hasNext()
            HAS_NEXT_ITEM -> true
            HAS_FINISHED -> false
            else -> ensureItemIterator()
        }
```

Оптимизируем flatten (убран nullable тип и введен state)

```
override fun iterator(): Iterator<E> = object : Iterator<E> {
    private val iterator = sequence.iterator()
    private var itemIterator: Iterator<E> = EmptyIterator // optimization for exclude nullable variable
    private var state: Int = UNDEFINED_STATE // { UNDEFINED_STATE, HAS_NEXT_ITEM, HAS_FINISHED }

    override fun next(): E {
        if (state == UNDEFINED_STATE) { // opt
            ensureItemIterator()
        }
        state = UNDEFINED_STATE
        return itemIterator.next()
    }

    override fun hasNext(): Boolean {
        return when (state) { // optimized cau
            HAS_NEXT_ITEM -> true
            HAS_FINISHED -> false
            else -> ensureItemIterator()
        }
    }
```

```
    private fun ensureItemIterator(): Boolean {
        if (itemIterator.hasNext()) {
            state = HAS_NEXT_ITEM
            return true
        } else {
            while (iterator.hasNext()) {
                val nextItemIterator = iterator(transformer(iterator.next()))
                if (nextItemIterator.hasNext()) {
                    itemIterator = nextItemIterator
                    state = HAS_NEXT_ITEM
                    return true
                }
            }
            state = HAS_FINISHED
            itemIterator = EmptyIterator
            return false
        }
    }
```

SBER DEVICES

Оптимизированный flatten

Размер списка	Collection (ns)	Sequence (ns)	%	Optimized Sequence	%	Ускорение
100	147 451	414 376	-181%	348 026	-136%	42%
1 000	1 512 504	4 167 416	-176%	3 568 417	-136%	38%
10 000	15 345 992	41 305 365	-169%	35 307 083	-130%	37%
50 000	75 917 917	205 262 897	-170%	175 828 229	-132%	36%
100 000	142 455 042	401 473 313	-182%	369 176 459	-159%	36%

А можно ли оптимизировать distinct?

Размер списка	Collection (ns)	Sequence (ns)	%
100	83 113	100 762	-21%
1 000	1 005 813	1 169 785	-16%
10 000	10 851 783	12 856 475	-18%
50 000	60 652 896	69 893 855	-15%
100 000	129 569 604	146 594 000	-13%

Устройство декоратора для distinct (сравниваем с коллекциями)

```kotlin
private class DistinctIterator<T, K>(
    private val source: Iterator<T>,
    private val keySelector: (T) -> K
) : AbstractIterator<T>() {
    private val observed = HashSet<K>()

    override fun computeNext() {
        while (source.hasNext()) {
            val next = source.next()
            val key = keySelector(next)

            if (observed.add(key)) {
                setNext(next)
                return
            }
        }

        done()
    }
```

Sequence

```kotlin
inline fun <T, K> Iterable<T>.distinctBy(
    selector: (T) -> K
): List<T> {
    val set = HashSet<K>()
    val list = ArrayList<T>()
    for (e in this) {
        val key = selector(e)
        if (set.add(key))
            list.add(e)
    }
    return list
}
```

Collections

Устройство декоратора для distinct (AbstractIterator)

```kotlin
abstract class AbstractIterator<T> : Iterator<T> {
    private var state = State.NotReady
    private var nextValue: T? = null

    override fun hasNext(): Boolean {
        require(state != State.Failed)
        return when (state) {
            State.Done -> false
            State.Ready -> true
            else -> tryToComputeNext()
        }
    }

    override fun next(): T {
        if (!hasNext()) throw NoSuchElementException()
        state = State.NotReady
        return nextValue as T
    }
```

```kotlin
    private fun tryToComputeNext(): Boolean {
        state = State.Failed
        computeNext()
        return state == State.Ready
    }

    abstract protected fun computeNext()

    protected fun setNext(value: T): Unit {
        nextValue = value
        state = State.Ready
    }

    protected fun done() {
        state = State.Done
    }
}
```

SBER DEVICES

Сравнение when, Enum vs Int

when с Enum

```
val result = when (value) {
    TestEnum.TEST_1 -> 0
    TestEnum.TEST_2 -> 1
    TestEnum.TEST_3 -> 2
    TestEnum.TEST_4 -> 3
    TestEnum.TEST_5 -> 4
}
```

when с Int

```
val result = when (value) {
    TEST_1 -> 0
    TEST_2 -> 1
    TEST_3 -> 2
    TEST_4 -> 3
    TEST_5 -> 4
    else -> value
}
```

when	Enum (ns)	Int (ns)	%
10 000	1 334 055	1 192 559	11%

Enum vs Int, 11% выигрыш. Как так то? Откуда?

```
override fun hasNext(): Boolean {
    require(state != State.Failed)
    return when (state) {
        State.Done -> false
        State.Ready -> true
        else -> tryToComputeNext()
    }
}
```

Это бессмысленная операция и об этом писал еще Джек Вартон несколько лет назад

Он обещал это исправить в R8, но видимо не успел(

R8 Optimization: Enum Switch Maps, 16 October 2019

```
ALOAD 0
GETFIELD com/example/benchmark/tst/AbstractIterator.state : Lcom/example/benchmark/tst/State;
GETSTATIC com/example/benchmark/tst/AbstractIterator$WhenMappings.$EnumSwitchMapping$0 : [I
SWAP
INVOKEVIRTUAL com/example/benchmark/tst/State.ordinal ()I
IALOAD
L7
TABLESWITCH
  1: L8
  2: L9
  default: L10
```

Оптимизируем distinct (убираем наследование и enum)

```
private class OptimizedDistinctIterator<T, K>(
    private val source: Iterator<T>, private val keySelector: (T) -> K
) : Iterator<T>{
    private val observed = HashSet<K>()
    // { UNDEFINED_STATE, HAS_NEXT_ITEM, HAS_FINISHED }
    private var nextState: Int = UNDEFINED_STATE
    private var nextItem: T? = null

    override fun hasNext(): Boolean {
        if (nextState == UNDEFINED_STATE)
            calcNext()
        return nextState == HAS_NEXT_ITEM
    }

    override fun next(): T {
        if (nextState == UNDEFINED_STATE)
            calcNext()
        if (nextState == HAS_FINISHED)
            throw NoSuchElementException()
        nextState = UNDEFINED_STATE
        return nextItem as T
    }
```

```
private fun calcNext() {
    while (source.hasNext()) {
        val next = source.next()
        val key = keySelector(next)

        if (observed.add(key)) {
            nextItem = next
            nextState = HAS_NEXT_ITEM // found next item
            return
        }
    }
    nextState = HAS_FINISHED // end of iterator
}
```

SBER DEVICES

Оптимизированный distinct, пропускающий 90% записей

Размер списка	Collection (ns)	Sequence (ns)	%	Optimized Sequence	%	Ускорение
100	83 113	100 762	-21%	80 538	3%	18%
1 000	1 005 813	1 169 785	-16%	984 303	2%	16%
10 000	10 851 783	12 856 475	-18%	11 111 379	2%	17%
50 000	60 652 896	69 893 855	-15%	61 164 896	1%	14%
100 000	129 569 604	146 594 000	-13%	129 624 667	0%	12%

Надеюсь я принес пользу миру)

— Отправил оптимизированные функции в JetBrains...
— При разработке подобных функций важна каждая мелочь, так как это сильно бьет по производительности кода
— Возможно это исследование станет чем то большим, чем просто доклад)

Накидывайте лайки на issue в Jet Brains KT-58588

Задача

Нужно загрузить в базу данных информацию о гео-точках из csv файла объемом более 10 Гб

Когда sequence действительно необходим

Пример кода ленивого чтения CSV файла на 10 Гб и сохранения его в базу данных

```
csvFile.useLines { linesSequence ->
    linesSequence
        .drop( n: 1)// skip fields header
        .map { line -> line.split( ...delimiters: ",") } // line to fieldList
        .filter { fields -> fields.find { it.isNullOrBlank() } == null } // skip error line
        .map { fields ->
            WifiPoint(
                id = fields[0].toLong(),
                pointId = fields[1].toLong(),
                latitude = fields[2].toDouble(),
                longitude = fields[3].toDouble(),
                customData = fields[4],
            )
        }
        .forEach { wifiPoint -> database.addPoint(wifiPoint) }
}
```

Когда sequence действительно необходим

Устройство декоратора для ленивого чтения файлов

```
public inline fun <T> File.useLines(charset: Charset = Charsets.UTF_8, block: (Sequence<String>) -> T): T =
    bufferedReader(charset).use { block(it.lineSequence()) }
```

```
private class LinesSequence(private val reader: BufferedReader) : Sequence<String> {
    override public fun iterator(): Iterator<String> {
        return object : Iterator<String> {
            private var nextValue: String? = null
            private var done = false

            override public fun hasNext(): Boolean {
                if (nextValue == null && !done) {
                    nextValue = reader.readLine()
                    if (nextValue == null) done = true
                }
                return nextValue != null
            }
```

```
override public fun next(): String {
    if (!hasNext()) {
        throw NoSuchElementException()
    }
    val answer = nextValue
    nextValue = null
    return answer!!
}
```

Вот теперь все

— Думаю sequence идеально подходит для решения именно таких задач
когда размер входящих данных неизвестен или имеет гигантский объем

— Sequence это удобный адаптер для оборачивания таких входящих данных в знакомый интерфейс обработки коллекций

— На простых преобразованиях sequence даст выигрыш, если не использовать тяжелые операции
flatten, plus, sort, distinct

Максим Сидоров
sidorov.m.vad@sberbank.ru, 2023
linkedIn: max-sidorov-maxssoft