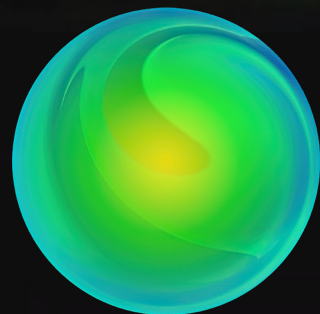
Непоследовательные последовательности: производительность sequence, stream и collection в JVM



Максим Сидоров, Максим Митюшкин

Системные сервисы, Salute TV

SBER DEVICES

Mobius spring 2023

Представил доклад «Измеряя sequence» о моем исследовании производительности sequence и сравнении их с коллекциями

- Рассказал и показал как работают под капотом все функции sequence
- Сравнил их с коллекциями и измерил результаты
- Экспериментально подтвердил правило «CouldBeSequence»

Используйте sequences, если в вашем преобразовании больше 3 функций преобразования

Абсолютное большинство функций sequence выигрывают у коллекций начиная с уже двух преобразований

SBER DEVICES

Оптимизации

В рамках исследования я предложил оптимизации некоторых функций sequence, которые существенно их ускорили

Компания JetBrains приняла предложенные мной оптимизации и они уже включены в состав релиза Kotlin 2.0

Программный комитет Joker предложил мне прочитать этот доклад на осенней конференции

SBER DEVICES

Готовим новый доклад

Что может быть проще

Берем старый доклад «Измеряя sequence»

Добавляем щепотку **стримов**

И вуаля, готов новый доклад

Но что-то пошло не так

Цифры упорно не хотели подчиняться логике и здравому смыслу

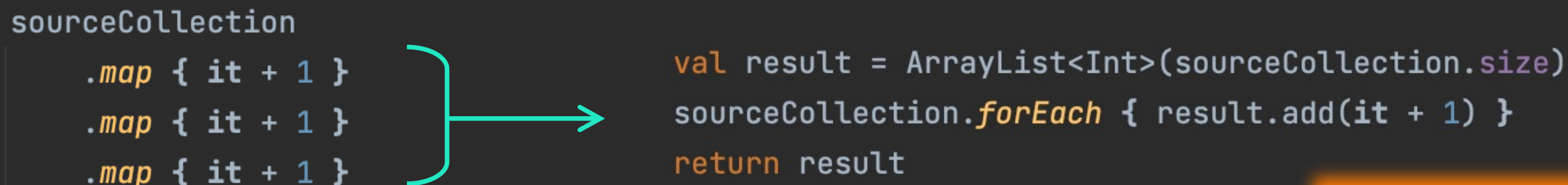
Результаты были прямо противоположны результатам Android

А все доступные мне эксперты говорили «Молись богу Шипилеву»

Сравнение способов обработки коллекций

Collection

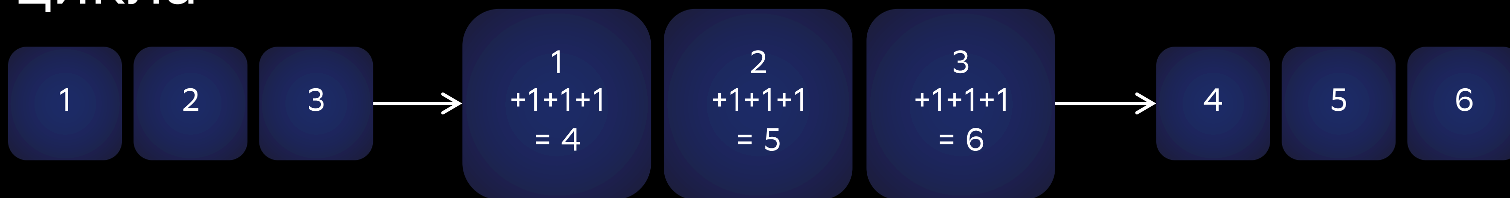
Каждое преобразование выполняется в своем отдельном цикле



На каждое преобразование создается новая коллекция

Sequence

Все преобразования выполняются последовательно, за один проход цикла



Нет выделения памяти под хранение промежуточных результатов

SBER DEVICES

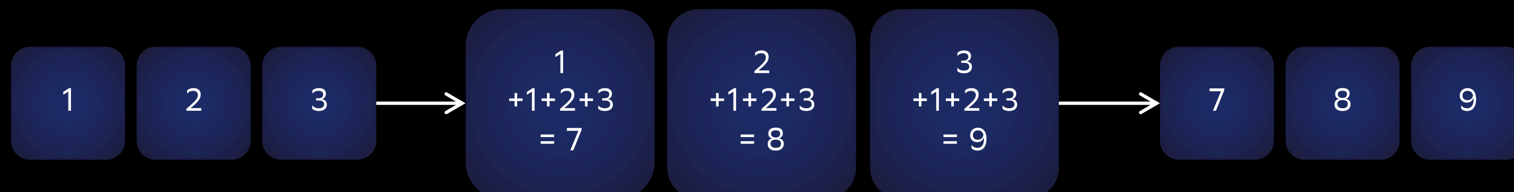
Как работает эта ленивая магия внутри

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) }
```



SBER DEVICES

Кажется, сплошные плюсы?

— На первый взгляд ленивая обработка должна выигрывать у коллекций
— Все построено на обычных итераторах и паттерне декоратора

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 **JMH** и за дело...

SBER DEVICES

Методика измерений

Kotlin Benchmark 0.4.10

—Каждый тест выполнялся на 25 прогонах

—JMH version 1.21

—JVM Oracle Open JDK 21

—Kotlin 1.9.10

Все тесты проводились на VPS:

— Linux, Debian 12

— Xeon E-2288G, 3.7GHz

— 4 Гб RAM

На MacBook и Windows машинах результаты тестов были похожими

SBER DEVICES

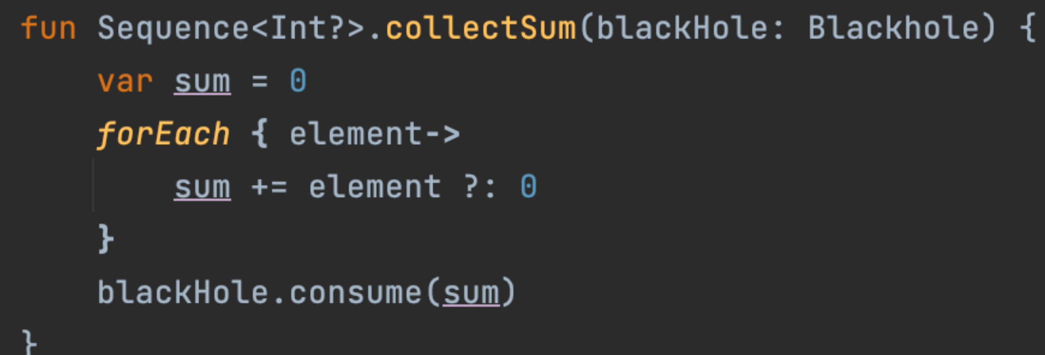
Генерация списков

- Используется последовательность целых значений <Int?> заданного размера, перемешанная случайным образом
- В последовательность замешивается явный null, чтобы исключить любые оптимизации с примитивами

```
fun createIntList(count: Int): List<Int?> =
    buildList { this: MutableList<Int?>
        addAll( elements: 0 ≤ .. ≤ count)
        add(null)
        shuffle()
    }
```

Пример кода теста

```
@Warmup(iterations = WARN_UP_ITERATIONS, time = WARN_UP_TIME,
class MapExample {
    private lateinit var originCollection: List<Int?>
    new *
    @Setup
    fun setup() {
        originCollection = createIntList( count: 100_000)
    }
    new *
    @Benchmark
    fun map5_100_000_rec_sequence(blackHole: Blackhole) {
        map5_sequence(originCollection).collectSum(blackHole)
    }
    new *
    @Benchmark
    fun map5_100_000_rec_collection(blackHole: Blackhole) {
        map5_collection(originCollection).collectSum(blackHole)
    }
```



SBER DEVICES

Пример функций для теста

```
fun map5_sequence(sourceCollection: List<Int?>): Sequence<Int?> {
    return sourceCollection.asSequence()
        .map { it?.plus( other: 1) }
        .map { it?.plus( other: 2) }
        .map { it?.plus( other: 3) }
        .map { it?.plus( other: 4) }
        .map { it?.plus( other: 5) }
}
new *
fun map5_stream(sourceCollection: List<Int?>): Stream<Int?> {
    return sourceCollection.stream() Stream<Int?>
        .map { it?.plus( other: 1) } Stream<Int?>!
        .map { it?.plus( other: 2) }
        .map { it?.plus( other: 3) }
        .map { it?.plus( other: 4) }
        .map { it?.plus( other: 5) }
}
```

SBER DEVICES

Измерения – filter { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	573	183	68%	145	75%	20%
1 000	5 415	1 803	67%	972	82%	46%
10 000	76 880	23 163	70%	10 473	86%	55%
50 000	425 684	148 039	65%	95 901	77%	35%
100 000	1 137 801	339 369	70%	239 720	79%	29%

```
@Benchmark
fun test(blackHole: Blackhole) {
    val percent10 = (originCollection.size * 0.1).toInt()
    originCollection.asSequence()
        .filter { it in 0 ≤ .. ≤ percent10 }
        .collectSum(blackHole)
}
```

SBER DEVICES

Имплементация filter

FilteringSequence

```kotlin
override fun iterator(): Iterator<T> = object .
    val iterator = sequence.iterator()
    var nextState: Int = -1 // -1 for unknown,
    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
    }
```

Stream:ReferencePipeLine.filter

```java
@Override
Sink<P_OUT> opWrapSink(int flags, Sink<P_OUT> sink) {
    return new Sink.ChainedReference<P_OUT, P_OUT>(sink) {
        @Override
        public void begin(long size) { downstream.begin( size: -1);

        @Override
        public void accept(P_OUT u) {
            if (predicate.test(u))
                downstream.accept(u);
        }
    };
}
```

SBER DEVICES

Измерения – filter { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	573	183	68%	145	75%	20%
1 000	5 415	1 803	67%	972	82%	46%
10 000	76 880	23 163	70%	10 473	86%	55%
50 000	425 684	148 039	65%	95 901	77%	35%
100 000	1 137 801	339 369	70%	239 720	79%	29%

— Внутренняя реализация примерно одинаковая

— Кажется, что stream имеет меньше накладных расходов на возврат одного элемента

SBER DEVICES

Измерения – distinct { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	1 918	2 173	-13%	2 376	10%	20%
1 000	22 479	25 304	-13%	28 582	13%	22%
10 000	327 575	366 272	-12%	439 876	7%	17%
50 000	1 888 549	1 913 547	-1%	3 143 350	14%	16%
100 000	7 443 358	8 306 638	-12%	7 134 284	-13%	-2%

```
@Benchmark
fun test(blackHole: Blackhole) {
    originCollection.asSequence()
        .distinctBy { it }
        .collectSum(blackHole)
}
```

SBER DEVICES

Имплементация distinct

DistinctSequence

```
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()
    }
```

Stream:DistinctOps

```
@Override
Sink<T> opWrapSink(int flags, Sink<T> sink) {
    Objects.requireNonNull(sink);

    if (StreamOpFlag.DISTINCT.isKnown(flags)) {
        return sink;
    } else if (StreamOpFlag.SORTED.isKnown(flags)) {
        // Более эффективный алгоритм distinct
        // для упорядоченной последовательности
    } else {
        public void begin(long size) {
            seen = new HashSet<>();
            downstream.begin( size: -1);
        }

        public void accept(T t) {
            if (seen.add(t)) {
                downstream.accept(t);
            }
        }
```

SBER DEVICES

Измерения – distinct { … }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	1 918	2 173	-13%	2 376	10%	20%
1 000	22 479	25 304	-13%	28 582	13%	22%
10 000	327 575	366 272	-12%	439 876	7%	17%
50 000	1 888 549	1 913 547	-1%	3 143 350	14%	16%
100 000	7 443 358	8 306 638	-12%	7 134 284	-13%	-2%

— Внутренняя реализация одинаковая

— Stream чуть быстрее, что подтверждает предположение по поводу накладных расходов на возврат одного элемента

Оптимизированный мной distinct, который вышел в Kotlin 2.0, немного обгоняет stream

SBER DEVICES

Измерения – sort { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	3 587	4 436	-24%	4 193	-17%	5%
1 000	99 870	107 220	-7%	105 528	-6%	2%
10 000	1 426 413	1 475 389	-3%	1 465 024	-3%	1%
50 000	8 930 319	9 252 358	-4%	9 079 944	-2%	2%
100 000	19 779 983	20 143 321	-2%	21 534 011	-9%	-7%

```
@Benchmark
fun test(blackHole: Blackhole) {
    originCollection.asSequence()
        .sortedByDescending { it }
        .collectSum(blackHole)
}
```

SBER DEVICES

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 }
```

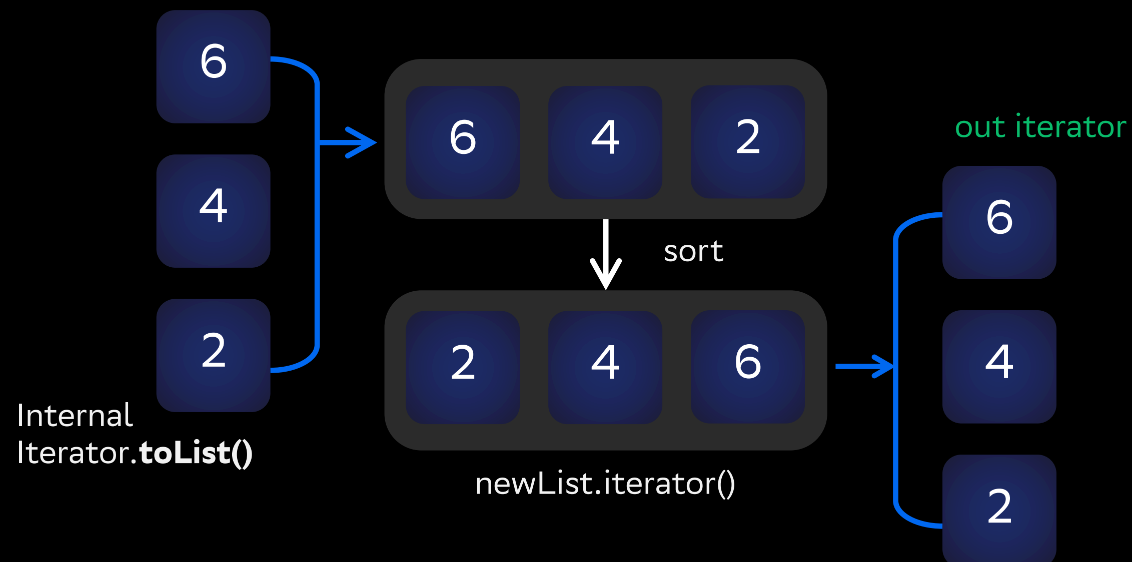
```
val list = sequenceOf( ...elements: 5, 4, 3, 2, 1)
    .map { it + 1 }
    .filter { it % 2 == 0 }
    .toList()

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

SBER DEVICES

Sequence: устройство декоратора для sortedBy


```
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()
        }
    }
}
```



SBER DEVICES

Stream: устройство операции RefSortingSink


```
private static final class RefSortingSink<T> extends AbstractRefSortingSink<T> {
    private ArrayList<T> list;

    @Override
    public void begin(long size) {
        if (size >= Nodes.MAX_ARRAY_SIZE)
            throw new IllegalArgumentException(Nodes.BAD_SIZE);
        list = (size >= 0) ? new ArrayList<>((int) size) : new ArrayList<>();
    }

    @Override
    public void end() {
        list.sort(comparator);
        downstream.begin(list.size());
        if (!cancellationRequestedCalled) {
            list.forEach(downstream::accept);
        }

    @Override
    public void accept(T t) { list.add(t); }
```

SBER DEVICES

Измерения – drop { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	339	228	33%	324	4%	-42%
1 000	3 469	2 118	39%	3 137	10%	-48%
10 000	33 463	23 860	29%	30 094	10%	-26%
50 000	177 926	122 661	31%	138 058	22%	-13%
100 000	392 923	367 697	6%	404 603	-3%	-10%

```
@Benchmark
fun test(blackHole: Blackhole) {
    val count = originCollection.size * 10 / 100
    originCollection.asSequence()
        .drop(count)
        .collectSum(blackHole)
}
```

SBER DEVICES

Измерения – take { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	75	23	70%	65	14%	-189%
1 000	626	100	84%	267	57%	-168%
10 000	6 421	1 383	78%	2 449	62%	-77%
50 000	31 751	8 015	75%	9 924	69%	-24%
100 000	69 095	21 674	69%	30 685	56%	-42%

```
@Benchmark
fun test(blackHole: Blackhole) {
    val count = originCollection.size * 10 / 100
    originCollection.asSequence()
        .take(count)
        .collectSum(blackHole)
}
```

SBER DEVICES

Sequence: Устройство декораторов

TakeSequence

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

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

DropSequence

```
private fun drop() {
    while (left > 0 && iterator.hasNext()) {
        iterator.next()
        left--
    }
}

override fun next(): T {
    drop()
    return iterator.next()
}

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

SBER DEVICES

Stream: Устройство операции SliceOps (limit)


```java
Sink<T> opWrapSink(int flags, Sink<T> sink) {
    return new Sink.ChainedReference<>(sink) {
        long n = skip;
        long m = normalizedLimit;

        @Override
        public void begin(long size) { downstream

        @Override
        public void accept(T t) {
            if (n == 0) {
                if (m > 0) {
                    m--;
                    downstream.accept(t);
                }
            }
            else {
                n--;
            }
        }
```

drop

take

take и drop реализованы в виде одной универсальной операции SliceOps

SBER DEVICES

Почему stream проигрывает?

take {}

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	75	23	70%	65	14%	-189%
1 000	626	100	84%	267	57%	-168%
10 000	6 421	1 383	78%	2 449	62%	-77%
50 000	31 751	8 015	75%	9 924	69%	-24%
100 000	69 095	21 674	69%	30 685	56%	-42%

drop {}

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	339	228	33%	324	4%	-42%
1 000	3 469	2 118	39%	3 137	10%	-48%
10 000	33 463	23 860	29%	30 094	10%	-26%
50 000	177 926	122 661	31%	138 058	22%	-13%
100 000	392 923	367 697	6%	404 603	-3%	-10%

SBER DEVICES

Stream: причина проигрыша


```
Sink<T> opWrapSink(int flags, Sink<T> sink) {
    return new Sink.ChainedReference<>(sink) {
        long n = skip;
        long m = normalizedLimit;

        @Override
        public void begin(long size) { downstream

        @Override
        public void accept(T t) {
            if (n == 0) {
                if (m > 0) {
                    m--;
                    downstream.accept(t);
                }
            }
            else {
                n--;
            }
        }
```

drop

take

Stream - сэкономили на имплементации, проиграли в производительности

Для take эта проверка лишняя

Для drop эти операции лишние

SBER DEVICES

Измерения – groupBy { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	1 088	1 106	-1%	2 338	-115%	-49%
1 000	12 079	12 482	-3%	22 572	-87%	-81%
10 000	177 866	174 809	2%	286 993	-61%	-64%
50 000	1 320 308	1 234 817	6%	2 096 832	-59%	-70%
100 000	4 459 051	4 093 430	8%	6 864 816	-54%	-68%

```
@Benchmark
fun test(blackHole: Blackhole) {
    originCollection.asSequence() Sequence<Int?>
        .groupBy { (it ?: 1) * 10 / 100 } Map<Int, List<Int?>>
        .keys.collectSum(blackHole)
}
```

Sequence: 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
}
```

SBER DEVICES

Stream: Устройство groupingBy

Сложно!!!

```
Collector<T, ?, M> groupingBy(Function<? super T, ? extends K> classifier,
                              Supplier<M> mapFactory,
                              Collector<? super T, A, D> downstream) {
    Supplier<A> downstreamSupplier = downstream.supplier();
    BiConsumer<A, ? super T> downstreamAccumulator = downstream.accumulator();
    BiConsumer<Map<K, A>, T> accumulator = (m, t) -> {
        K key = Objects.requireNonNull(classifier.apply(t), message: "element cannot be mappe
        A container = m.computeIfAbsent(key, k -> downstreamSupplier.get());
        downstreamAccumulator.accept(container, t);
    };
    BinaryOperator<Map<K, A>> merger = Collectors.<~>mapMerger(downstream.combiner());
    /unchecked/
    Supplier<Map<K, A>> mangledFactory = (Supplier<Map<K, A>>) mapFactory;

    if (downstream.characteristics().contains(Collector.Characteristics.IDENTITY_FINISH)) {
        return new CollectorImpl<>(mangledFactory, accumulator, merger, CH_ID);
    }
    else {
        /unchecked/
        Function<A, A> downstreamFinisher = (Function<A, A>) downstream.finisher();
        Function<Map<K, A>, M> finisher = intermediate -> {
```

SBER DEVICES

Измерения – groupBy { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	1 088	1 106	-1%	2 338	-115%	-49%
1 000	12 079	12 482	-3%	22 572	-87%	-81%
10 000	177 866	174 809	2%	286 993	-61%	-64%
50 000	1 320 308	1 234 817	6%	2 096 832	-59%	-70%
100 000	4 459 051	4 093 430	8%	6 864 816	-54%	-68%

```
@Benchmark
fun test(blackHole: Blackhole) {
    originCollection.asSequence() Sequence<Int?>
        .groupBy { (it ?: 1) * 10 / 100 } Map<Int, List<Int?>>
        .keys.collectSum(blackHole)
}
```

SBER DEVICES

Измерения – одиночные базовые операции на 10.000 элементов

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
filter	76 880	23 163	70%	10 473	86%	55%
distinct	327 575	366 272	-12%	439 876	7%	17%
sort	1 426 413	1 475 389	-3%	1 465 024	-3%	1%
drop	33 463	23 860	29%	30 094	10%	-26%
take	6 421	1 383	78%	2 449	62%	-77%
groupBy	177 866	174 809	2%	286 993	-61%	-64%

— В большинстве одиночных операций ленивые преобразования выигрывают у коллекций

— Кажется, что stream имеет меньше накладных расходов на возврат одного элемента, поэтому базово они должны быть чуть быстрее (но здесь очень тонкий баланс)

SBER DEVICES

Stream устроены намного сложнее Sequence

- Они используют более сложный **SplitIterator** (упорядоченность, уникальность, размер и т.д.)
- Есть отдельные имплементации операций для разных типов данных и типов потоков
- Операции в stream связаны друг с другом и умеют обмениваться информацией

На сложных операциях это иногда может давать Stream существенное преимущество

Но это как карта ляжет

И конечно киллер фича стримов - ParallelStream

SBER DEVICES

Но это все в теории на одиночных преобразованиях

— А что будет с более реальными преобразованиями???

Измерения – middle { 4 операции }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	3 848	4 053	-5%	4 305	-12%	-6%
1 000	53 353	55 348	-4%	52 150	2%	6%
10 000	641 115	650 517	-1%	598 693	7%	8%
50 000	3 173 258	3 178 359	0%	3 004 347	5%	5%
100 000	7 316 505	7 488 397	-2%	7 069 650	3%	6%

```
@Benchmark
fun test(blackHole: Blackhole) {
    val percent90 = (originCollection.size * 0.9).toInt()
    originCollection.asSequence()
        .map { it?.plus( other: 1) }
        .filter { it in 0 ≤.. ≤ percent90 }
        .map { it?.let { it % 3 } }
        .sortedBy { it }
        .collectSum(blackHole)
}
```

Измерения – crazy { 7 операций }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	6 206	7 537	-21%	9 933	-7%	12%
1000	109 291	130 131	-19%	150 570	-7%	10%
10 000	1 467 263	1 630 517	-11%	2 021 225	-3%	8%
50 000	8 899 964	9 955 203	-12%	12 359 273	-5%	7%
100 000	20 122 174	21 626 819	-7%	30 951 104	-1%	6%

```
@Benchmark
fun test(blackHole: Blackhole) {
    originCollection.asSequence()
        .map { it?.plus( other: 1) }
        .filter { it != null }
        .sortedBy { it }
        .map { it?.let { it % 3 } }
        .filter { (it ?: 0)  > 0 }
        .map { it?.plus( other: 1) }
        .distinctBy { it }
        .collectBlackHole(blackHole)
```

SBER DEVICES

Измерения – реальное преобразование { 5 тяжелых операций }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	32 159	37 181	-16%	32 463	-1%	13%
1000	256 134	280 339	-9%	364 733	-42%	-30%
10 000	1 906 587	2 344 094	-23%	3 239 264	-70%	-38%
50 000	10 734 587	11 621 365	-8%	16 747 520	-56%	-44%
100 000	23 707 730	23 713 471	0%	32 645 589	-38%	-38%

```
return realData.sessionManager.productCategories.asSequence() Sequence<ProductCategory>
    .map { it.categoryName } Sequence<String>
    .mapNotNull { realData.productRepository.getCategoryProducts(it) } Sequence<List<Product>>
    .flatten() Sequence<Product>
    .distinctBy { it.productUid }
    .map { product ->
        product.productUid to product.photos.firstOrNull { it.isDefault }?.url
    }
```

SBER DEVICES

А дальше начнется интересное...

SBER DEVICES

Измерения – map2 { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	1 321	666	50%	829	37%	-24%
1 000	14 056	7 384	47%	11 717	17%	-59%
10 000	140 482	65 977	53%	100 288	29%	-52%
50 000	895 977	351 568	61%	592 326	34%	-68%
100 000	1 681 014	792 065	53%	1 083 026	36%	-37%

```
@Benchmark
fun test(blackHole: Blackhole) {
    originCollection.asSequence()
        .map { it?.plus( other: 1) }
        .map { it?.plus( other: 2) }
        .collectSum(blackHole)
}
```

SBER DEVICES

Измерения – map3 { … }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	1 751	2 430	-39%	3 167	-81%	-30%
1 000	21 963	25 027	-14%	33 043	-50%	-32%
10 000	169 148	185 722	-10%	219 511	-30%	-16%
50 000	1 096 220	1 203 239	-10%	1 695 113	-55%	-41%
100 000	2 514 696	2 644 224	-5%	3 139 259	-25%	-19%

```
@Benchmark
fun test(blackHole: Blackhole) {
    originCollection.asSequence()
        .map { it?.plus( other: 1) }
        .map { it?.plus( other: 2) }
        .map { it?.plus( other: 3) }
        .collectSum(blackHole)
}
```

Правило CouldBeSequence отправляется в утиль

SBER DEVICES

Измерения – map5 { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	2 905	3 810	-31%	4 709	-62%	-24%
1 000	34 223	40 588	-19%	49 403	-44%	-22%
10 000	290 542	375 164	-29%	534 746	-84%	-43%
50 000	1 819 715	2 103 401	-16%	2 499 327	-37%	-19%
100 000	3 469 184	3 952 660	-14%	5 135 326	-48%	-30%

```
@Benchmark
fun test(blackHole: Blackhole) {
    originCollection.asSequence()
        .map { it?.plus( other: 1) }
        .map { it?.plus( other: 2) }
        // .......
        .map { it?.plus( other: 5) }
        .collectSum(blackHole)
}
```

SBER DEVICES

Измерения – map10 { ... }

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
100	6 192	8 480	-37%	11 011	-78%	-30%
1 000	75 302	91 474	-21%	107 022	-42%	-17%
10 000	614 538	843 246	-37%	1 151 171	-87%	-37%
50 000	3 427 219	4 403 515	-28%	5 887 390	-72%	-34%
100 000	7 465 281	9 815 935	-31%	11 558 141	-55%	-18%

```
@Benchmark
fun test(blackHole: Blackhole) {
    originCollection.asSequence()
        .map { it?.plus( other: 1) }
        .map { it?.plus( other: 2) }
        // .......
        .map { it?.plus( other: 10) }
        .collectSum(blackHole)
}
```

SBER DEVICES

Измерения –map, размер списка 10 000 элементов

Open JDK
Kotlin Benchmark

Размер списка	Collection (ns)	Sequence (ns)	%	Stream (ns)	%
map 2	140 482	65 977	53%	100 288	29%
map 3	204 405	235 032	-15%	213 769	-5%
map 5	290 542	375 164	-29%	534 746	-84%
map 10	614 538	843 246	-37%	1 151 171	-87%

Android
Jetpack Benchmark

Размер списка	Collection (ns)	Sequence (ns)	%
map 2	5 726 766	5 603 207	2%
map 3	8 625 083	7 072 713	18%
map 5	14 172 953	9 975 721	30%
map 10	25 770 447	17 523 903	32%

SBER DEVICES

Парадокс – парадокс {map} с ростом числа преобразований

Кол-во операций	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
map 2	114 158	56 694	45%	78 968	31%	-28%
map 3	169 148	185 722	-10%	219 511	-30%	-16%
map 5	269 135	323 280	-20%	393 282	-46%	-18%
map 10	578 930	683 528	-18%	838 439	-45%	-18%
map 20	1 416 756	1 501 005	-6%	1 178 739	17%	27%
map 30	2 989 817	2 356 094	21%	1 847 659	38%	28%
map 40	4 525 295	3 554 629	21%	2 610 452	42%	36%
map 50	6 085 048	4 942 998	19%	3 570 704	41%	38%
map 80	89 466 405	10 014 910	89%	6 849 965	92%	46%

SBER DEVICES

Парадокс – парадокс {filter} с ростом числа преобразований

Кол-во операций	Collection (ns)	Sequence (ns)	%	Stream (ns)	%	Stream vs Sequence
filter 2	175 267	196 165	-12%	79 589	55%	59%
filter 3	243 992	396 359	-62%	218 263	11%	45%
filter 5	430 436	663 841	-54%	387 407	10%	42%
filter 10	824 414	1 310 797	-59%	645 864	13%	45%
filter 20	2 601 810	2 811 450	-8%	1 483 218	43%	47%
filter 30	5 130 262	3 694 294	28%	2 071 303	60%	44%
filter 50	9 297 885	6 235 821	33%	3 762 281	60%	40%
filter 80	110 337 404	9 389 511	91%	6 213 364	94%	34%

SBER DEVICES

И тут мне потребовалась помощь друга...

SBER DEVICES

И тут мы начали копать...

Кол-во операций	Collection (ns)	Sequence (ns)	%	Stream (ns)	%
map 2	114 158	56 694	45%	78 968	31%
map 3	169 148	185 722	-10%	219 511	-30%
map 5	269 135	323 280	-20%	393 282	-46%
map 10	578 930	683 528	-18%	838 439	-45%
map 20	1 416 756	1 501 005	-6%	1 178 739	17%
map 30	2 989 817	2 356 094	21%	1 847 659	38%
map 40	4 525 295	3 554 629	21%	2 610 452	42%
map 50	6 085 048	4 942 998	19%	3 570 704	41%
map 80	89 466 405	10 014 910	89%	6 849 965	92%

Почему коллекции выигрывают, а потом начинают проигрывать?

Падение производительности в 10 раз

Почему на 80 преобразованиях map в коллекциях производительность падает в 10 раз?

Берем JITWatch и начинаем профилировать.

SBER DEVICES

Включаем логи компиляции

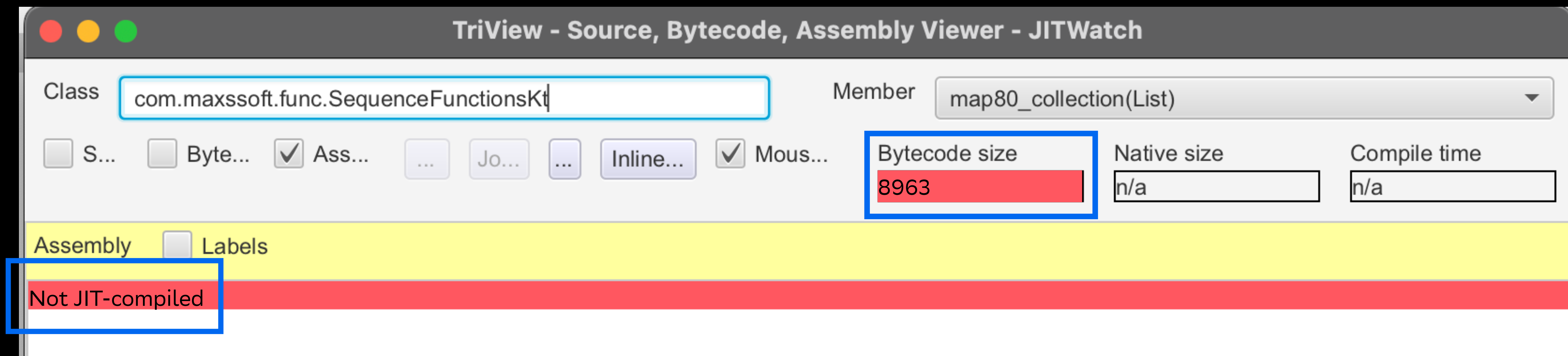
```
> java -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation –jar benchmark.jar
```

-XX:+UnlockDiagnosticVMOptions, разблокируем диагностические опции JVM

-XX:+LogCompilation, включаем логирование компиляций JVM

SBER DEVICES

Смотрим JITWatch для метода map80_collection



Метод не скомпилировался

-XX:HugeMethodLimit=8000 (don’t compile methods larger than this if)

Если метод содержит больше 8000 инструкций байт-кода, то он даже не будет компилироваться и будет работать в режиме интерпретации.

То есть очень, очень медленно ...

**источник: Java HotSpot VM Options*
https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html

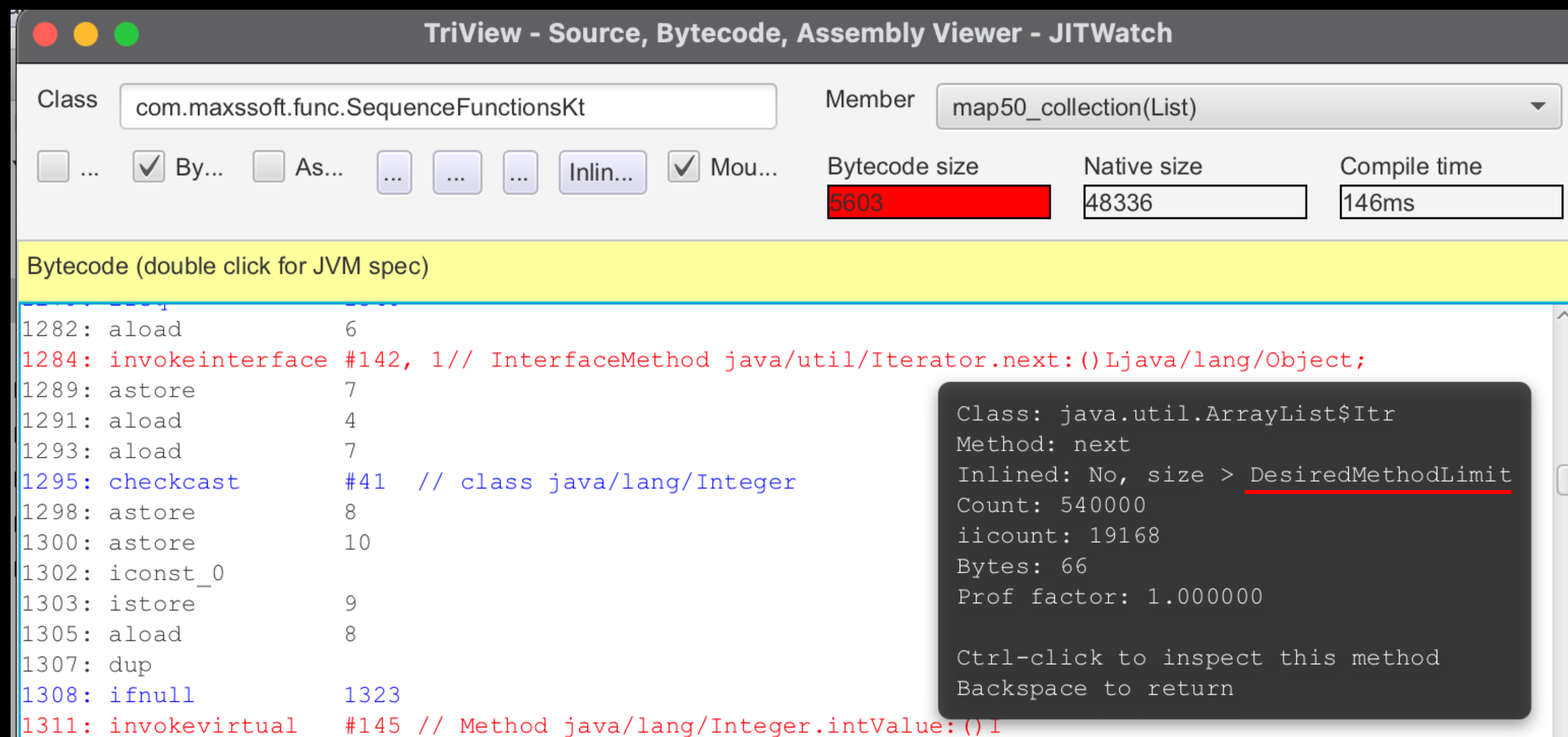
SBER DEVICES

Почему после 20 преобразований коллекции начинают замедляться?

JITWatch нам в помощь!

SBER DEVICES

Смотрим JITWatch для метода map50_collection



SBER DEVICES

DesiredMethodLimit

-XX:DesiredMethodLimit=8000 (maximum size in bytecodes of aggregate method after inlining.)

Общий размер метода после инлайнинга не может быть больше 8000 инструкций байт-кода.

map30_collection – такая же ошибка, DesiredMethodLimit

**источник: Java HotSpot VM Options*
https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html

SBER DEVICES

Смотрим JITWatch для метода map20_collection

TriView - Source, Bytecode, Assembly Viewer - JITWatch

Class: com.maxssoft.func.SequenceFunctionsKt
Member: map20_collection(List)

... Byt... Ass... ... J... ... Inlin... Mou...

Bytecode size: 2183
Native size: 32352
Compile time: 278ms

Bytecode (double click for JVM spec)

```
1791: ifeq            1849
1794: aload           6
1796: invokeinterface #142, 1// InterfaceMethod java/util/Iterator.next:()Ljava/lang/Object;
1801: astore
1803: aload
1805: aload
1807: checkcast
1810: astore
1812: astore
1814: iconst_0
1815: istore
1817: aload
1819: dup
1820: ifnull
```

```
Class: java.util.ArrayList$Itr
Method: next
Inlined: No, NodeCountInliningCutoff
Count: 740000
iicount: 21034
Bytes: 66
Prof factor: 1.000000

Ctrl-click to inspect this method
Backspace to return
```

NodeCountInliningCutoff

-XX:NodeCountInliningCutoff=18000 (if parser node generation exceeds limit stop inlining)

Перед компиляцией метода, JVM преобразовывает его в граф промежуточного представления (IR), где каждая вершина – элементарная операция.

Когда количество вершин превышает 18000, инлайнинг останавливается.

**источник: Server Compiler Inlining Messages*
https://wiki.openjdk.org/display/HotSpot/Server+Compiler+Inlining+Messages

SBER DEVICES

Как это бывает в реальной жизни

20 и больше преобразований – это конечно нереально

Но вот такое преобразование вполне реально

```
widgets
    .filter { favorites.contains(it) } List<AppWidget>
    .map { Pair(it.packageName, it.backdropInfo) } List<Pair<String?, BackdropInfo?>>
    .filter { it.second == backdropCriteria }
    .sortedBy { it.first }
```

Инлайнинг для такого преобразования также не будет работать из-за ограничения на размер метода:

AppWidget::equals fail: hot method too big

SBER DEVICES

Ошибки инлайнинга коллекций

Жирный data class

```kotlin
data class AppWidget(
    val providerId: String,
    val type: AppWidgetType,
    val weight: Int,
    val deepLink: String? = null,
    val applicationId: String? = null,
    val packageName: String? = null,
    val showDate: Date? = null,
    val startShowDate: Date? = null,
    val endShowDate: Date? = null,
    val shouldMoveToEndId: Boolean = false,
    val shouldHideInOtherSession: Boolean = false,
    val subCategoryProviderId: String? = null,
    val appPackageName: String = providerId,
    val mark: String? = null,
    val guid: String? = null,
    val isPromo: Boolean = false,
    val isAdvertisement: Boolean = false,
```

```kotlin
    val businessType: WidgetBusinessType =
        WidgetBusinessType.COMMON,
    val attributes: String? = null,
    val adsClickUrl: String? = null,
    val backgroundImage: ImageModel? = null,
    val additionalBackgroundImage: ImageModel? = null,
    val backdropInfo: BackdropInfo? = null,
    val leftTag: TagInfo? = null,
    val rightTag: TagInfo? = null,
    val rightAdditionalTag: TagInfo? = null,
    val icon: ImageModel? = null,
    val isOnAir: Boolean = false,
    val innerText: String? = null,
    val localeInnerText: String? = null,
    val progressValue: Int? = null,
    val title: String,
    val localeTitle: String? = null,
    val subtitle: String? = null,
    val localeSubtitle: String? = null,
    val disabledSurfacesTag: String? = null,
    val source: String? = null,
    val hiddenName: String = "",
)
```

SBER DEVICES

Проблемы инлайнинга

Даже небольшой рост количества выполняемых инструкций в методе может отключить ее инлайнинг и существенно снизить производительность.

- Ситуацию сложно заметить, так как большой объем кода может быть скрыт в имплементациях внутренних методов
- На отказ JVM от инлайнинга влияет не только длина метода, но и количества стека, расходуемого им.
- С ростом числа преобразований коллекции перестают инлайниться и начинают проигрывать Stream и Sequence.

SBER DEVICES

Почему коллекции в JVM обгоняют sequence?

Кол-во операций	Collection (ns)	Sequence (ns)	%	Stream (ns)	%
map 2	114 158	56 694	45%	78 968	31%
map 3	169 148	185 722	-10%	219 511	-30%
map 5	269 135	323 280	-20%	393 282	-46%
map 10	578 930	683 528	-18%	838 439	-45%
map 20	1 416 756	1 501 005	-6%	1 178 739	17%
map 30	2 989 817	2 356 094	21%	1 847 659	38%
map 40	4 525 295	3 554 629	21%	2 610 452	42%
map 50	6 085 048	4 942 998	19%	3 570 704	41%
map 80	89 466 405	10 014 910	89%	6 849 965	92%

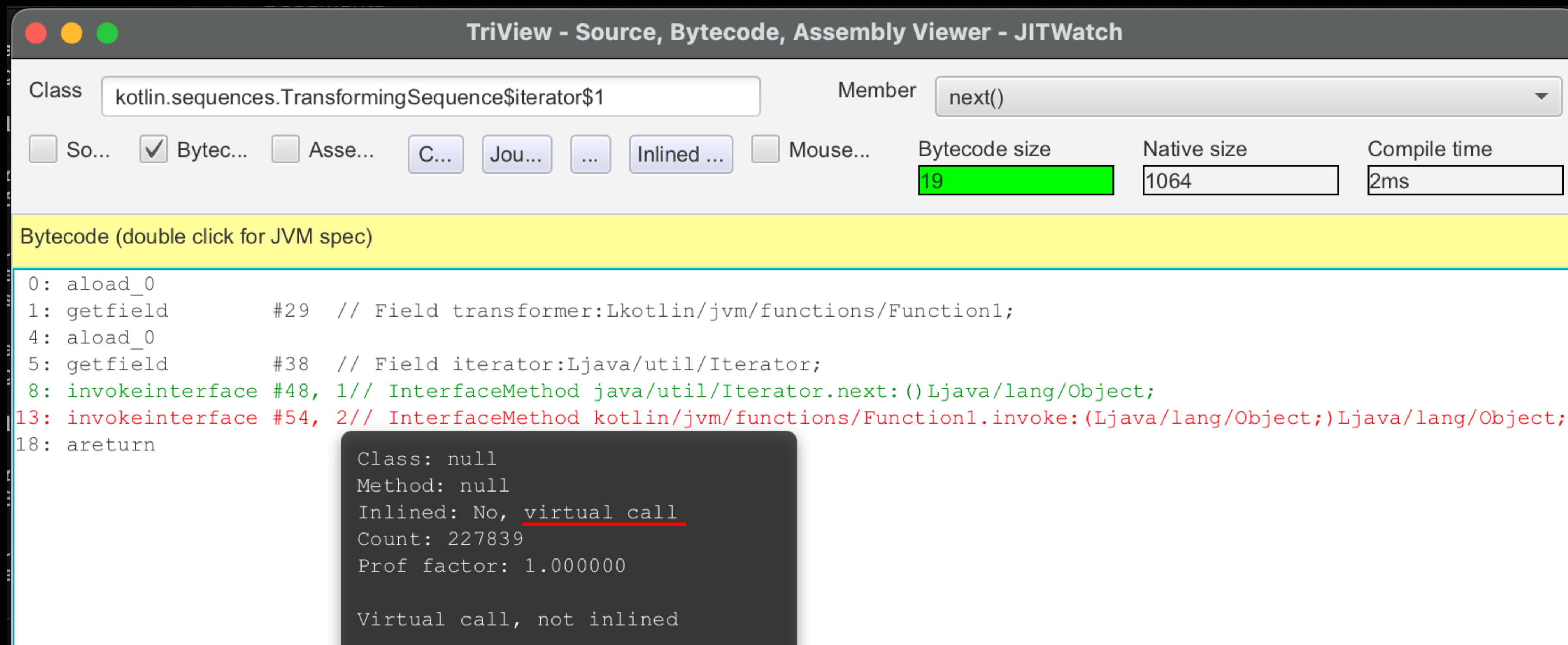
Проблемы у коллекций начинаются когда они перестают инлайниться.

А что с инлайнингом Sequence?

Берем JITWatch и начинаем разбираться …

SBER DEVICES

Смотрим JITWatch для метода map10_sequence



SBER DEVICES

Почему invokevirtual не инлайнится?

When the types are evenly distributed, we get a severe performance hit in dynamic_… cases. This is because HotSpot thinks the call site now has too many receiver types; in other words, the call site is megamorphic. Current C2 does not do megamorphic inlining at all.

Если через один и тот же invokevirtual вызывается много реализаций, то возникает проблема мегаморфизма, из-за чего JVM отказывается от инлайнинга в этом месте.

**источник: The Black Magic of (Java) Method Dispatch, https://shipilev.net/blog/2015/black-magic-method-dispatch/,*

SBER DEVICES

Мегаморфизм наглядно

Когда через один invokevirtual вызывается только одна реализация

В этом случае JVM заинлайнит вызов invokevirtual.

SBER DEVICES

Мегаморфизм наглядно

Когда через один invokevirtual вызывается несколько реализаций

JVM откажется инлайнить вызов invokevirtual, потому что заранее не знает какая реализация будет вызвана.

SBER DEVICES

Давайте разберем интересный пример

```kotlin
fun map10_sequence(sourceCollection: List<Int?>): Sequence<Int?> {
    return sourceCollection.asSequence()
        .map { it?.plus( other: 1) }
        .map { it?.plus( other: 2) }
        .map { it?.plus( other: 3) }
        .map { it?.plus( other: 4) }
        .map { it?.plus( other: 5) }
        .map { it?.plus( other: 6) }
        .map { it?.plus( other: 7) }
        .map { it?.plus( other: 8) }
        .map { it?.plus( other: 9) }
        .map { it?.plus( other: 10) }
}
```

```kotlin
.map(SingleMapper(1))
.map(SingleMapper(2))
.map(SingleMapper(3))
.map(SingleMapper(4))
.map(SingleMapper(5))
.map(SingleMapper(6))
.map(SingleMapper(7))
.map(SingleMapper(8))
.map(SingleMapper(9))
.map(SingleMapper(10))
```

```kotlin
class SingleMapper(
    private val increment: Int,
) : (Int?) -> Int? {
    override fun invoke(p1: Int?) =
        p1?.plus(increment)
}
```

Стандартное преобразование (много классов)

Все преобразования выполняются одним классом

SBER DEVICES

Ошибка инлайнинга invokevirtual

Стандартное преобразование (много классов) – **1 120 264 ns**

Преобразование с единственным классом – **386 946 ns**

В случае преобразования через SingleMapper инлайнинг работает, а в случае стандартного преобразования – нет.

SBER DEVICES

Давайте поставим коллекции в равные условия

```
class LambdaMapper(
    val action: (Int?) -> Int?,
) : (Int?) -> Int? {
    override fun invoke(p1: Int?): Int? =
        action(p1)
}
```

Класс LambdaMapper имитирует проблему мегаморфизма, вызывая лямбду через прокси-класс

SBER DEVICES

Давайте поставим коллекции в равные условия

```kotlin
fun map10_collection(sourceCollection: List<Int?>): List<Int?> {
    return sourceCollection
        .map { it?.plus( other: 1) }
        .map { it?.plus( other: 2) }
        .map { it?.plus( other: 3) }
        .map { it?.plus( other: 4) }
        .map { it?.plus( other: 5) }
        .map { it?.plus( other: 6) }
        .map { it?.plus( other: 7) }
        .map { it?.plus( other: 8) }
        .map { it?.plus( other: 9) }
        .map { it?.plus( other: 10) }
}
```

```kotlin
.map(LambdaMapper { it?.plus( other: 1) })
.map(LambdaMapper { it?.plus( other: 2) })
.map(LambdaMapper { it?.plus( other: 3) })
.map(LambdaMapper { it?.plus( other: 4) })
.map(LambdaMapper { it?.plus( other: 5) })
.map(LambdaMapper { it?.plus( other: 6) })
.map(LambdaMapper { it?.plus( other: 7) })
.map(LambdaMapper { it?.plus( other: 8) })
.map(LambdaMapper { it?.plus( other: 9) })
.map(LambdaMapper { it?.plus( other: 10) })
```

Стандартное преобразование, все преобразования с лямбдами инлайнятся

Все преобразования с лямбдами выполняются через прокси-класс LambdaMapper

SBER DEVICES

Теперь Sequence обгоняют коллекции

	Standalone			LambdaMapper		
Кол-во операций	**Collection (ns)**	**Sequence (ns)**	**%**	**Collection (ns)**	**Sequence (ns)**	**%**
map 2	114 158	56 694	45%	98 658	51 141	48%
map 3	169 148	185 722	-10%	207 014	186 442	10%
map 5	269 135	323 280	-20%	355 451	327 014	8%
map 10	578 930	683 528	-18%	714 716	628 950	12%
map 20	1 416 756	1 501 005	-6%	2 074 659	1 579 490	24%
map 30	2 989 817	2 356 094	21%	3 836 447	2 527 484	34%
map 40	4 525 295	3 554 629	21%	5 295 858	3 542 871	33%
map 50	6 085 048	4 942 998	19%	7 546 256	4 950 093	34%
map 80	89 466 405	10 014 910	89%	69 642 860	9 591 236	86%

SBER DEVICES

Проблема мегаморфизма

У Sequence и Stream нет шанса на инлайнинг из-за проблемы мегаморфизма.

Поэтому коллекции выигрывают, так как они инлайнятся.

SBER DEVICES

Попробовали GraalVM

На GraalVM Sequence отстали от коллекций еще больше.

Кол-во операций	Collection Hotspot (ns)	Sequence Hotspot (ns)	Hotspot %	Collection GraalVM (ns)	Sequence GraalVM (ns)	GraalVM %
map 2	114 158	56 694	45%	81 382	16 967	79%
map 3	169 148	185 722	-10%	137 238	104 852	24%
map 5	269 135	323 280	-20%	192 345	227 831	-18%
map 10	578 930	683 528	-18%	438 737	875 089	-99%
map 20	1 416 756	1 501 005	-6%	1 104 305	1 939 260	-76%
map 30	2 989 817	2 356 094	21%	1 662 111	3 597 031	-116%
map 40	4 525 295	3 554 629	21%	2 107 064	5 067 506	-141%
map 50	6 085 048	4 942 998	19%	3 144 642	6 642 493	-111%
map 80	89 466 405	10 014 910	89%	83 117 091	11 393 820	86%

SBER DEVICES

Попробовали GraalVM

Аналогично и со стримами

Кол-во операций	Collection Hotspot (ns)	Stream Hotspot (ns)	Hotspot %	Collection GraalVM (ns)	Stream GraalVM (ns)	GraalVM %
map 2	114 158	78 968	31%	81 382	7297	91%
map 3	169 148	219 511	-30%	137 238	7566	94%
map 5	269 135	393 282	-46%	192 345	190 859	1%
map 10	578 930	838 439	-45%	438 737	1 031 124	-135%
map 20	1 416 756	1 178 739	17%	1 104 305	2 136 441	-93%
map 30	2 989 817	1 847 659	38%	1 662 111	3 665 017	-121%
map 40	4 525 295	2 610 452	42%	2 107 064	4 982 196	-136%
map 50	6 085 048	3 570 704	41%	3 144 642	6 561 370	-109%
map 80	89 466 405	6 849 965	92%	83 117 091	10 912 167	87%

SBER DEVICES

Включаем режим интерпретации

```
> java -Xint –jar benchmark.jar
```

-Xint, переведем JVM в режим, в котором работает только интерпретатор, отключаются все оптимизации.

SBER DEVICES

Collection vs Sequence

Теперь результаты похожи на результаты Android.

Интересно, а что же происходит в Android?

Кол-во операций	Collection (ns)	Collection Xint (ns)	Sequence Xint (ns)	% Xint
map 2	114 158	13 909 218	10 345 754	25%
map 3	169 148	19 991 054	14 305 326	28%
map 5	269 135	31 218 064	21 859 476	29%
map 10	578 930	59 288 620	40 473 756	31%
map 20	1 416 756	117 015 602	76 987 486	34%
map 30	2 989 817	173 887 844	113 978 412	34%
map 40	4 525 295	231 430 172	147 062 656	36%
map 50	6 085 048	289 242 640	185 609 176	35%
map 80	89 466 405	458 105 642	315 507 030	31%

```
> java -Xint –jar benchmark.jar
```

SBER DEVICES

А что же происходит в Android?

А об этом мы расскажем на следующей конференции по мобильной разработке Mobius 2025

Это большая и интересная тема для отдельного доклада.

SBER DEVICES

Извечный спор: Что быстрее, collection, sequence или stream?

— Универсального ответа у нас нет

— На простых преобразованиях коллекции всегда будут быстрее, так как они инлайнятся

— Но только пока ваши преобразования укладываются в лимит инлайнинга. На сложных преобразованиях коллекции проиграют ленивым преобразованиям.

— Экономия на спичках, ваш профит в любом случае будет небольшим

SBER DEVICES

Задача

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

SBER DEVICES

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) }
}
```

SBER DEVICES

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!!
}
```

SBER DEVICES

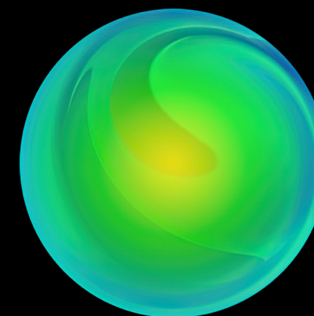
Спасибо за внимание

linkedIn:
sidorov-max

linkedIn:
kotlinovsky

Максим Сидоров,
Максим Митюшкин

Системные сервисы, Salute TV



SBER DEVICES