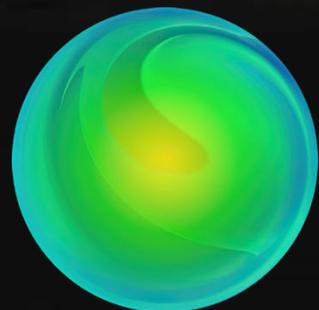


Нюансы работы Android Runtime в сравнении с HotSpot VM



Максим Сидоров, Максим Митюшкин

Системные сервисы, Salute TV

Доклад «Измеряя sequence»

- Исследование производительности sequence в сравнении с коллекциями
- Экспериментально доказал, что в мире Android sequence выигрывают у коллекций начиная уже с двух преобразований
- Абсолютное большинство функций sequence работают быстрее коллекций



Доклад «Непоследовательные последовательности»

- Внутреннее устройство Kotlin sequence
- Сравнение производительности работы Kotlin sequence и Java stream api
- Сравнение ленивого подхода с последовательным на стандартной HotSpot VM





Что может быть проще

Берем старый доклад «Измеряя sequence»

Добавляем щепотку
СТРИМОВ

И вуаля, готов **НОВЫЙ ДОКЛАД**



В HotSpot VM цифры упорно не хотели подчиняться логике и здравому смыслу

Результаты были прямо противоположны результатам Android

А все доступные мне эксперты говорили «Молись богу Шипилеву»

HotSpot VM

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)
}
```

HotSpot VM

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

HotSpot VM

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)
}
```

Измерения – map10 { ... }

HotSpot VM

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)
}
```

Измерения –map, динамика

HotSpot VM
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%

Парадокс {map} с ростом числа преобразований

11 / 61

HotSpot VM

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%

Парадокс {filter} с ростом числа преобразований

12 / 61

HotSpot VM

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%

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 сначала выигрывают, а потом начинают проигрывать?

Падение производительности в 10 раз

map80_collection

14 / 61

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

Берем JITWatch и начинаем профилировать.

Включаем логи компиляции

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

-XX:+UnlockDiagnosticVMOptions, разблокируем диагностические опции JVM

-XX:+LogCompilation, включаем логирование компиляций JVM

Смотрим JITWatch для метода map80_collection

TriView - Source, Bytecode, Assembly Viewer - JITWatch

Class: Member:

S... Byte... Ass... Mous...

Bytecode size	Native size	Compile time
8963	n/a	n/a

Assembly Labels

Not JIT-compiled

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 ...

map20_collection – map50_collection

18 / 61

Почему после 20 преобразований коллекции начинают замедляться?

И снова JITWatch нам в помощь!

Смотрим JITWatch для метода map50_collection

TriView - Source, Bytecode, Assembly Viewer - JITWatch

Class: `com.maxssoft.func.SequenceFunctionsKt` Member: `map50_collection(List)`

... By... As... Inlin... Mou...

Bytecode size	Native size	Compile time
5603	48336	146ms

Bytecode (double click for JVM spec)

```
1282: aload      6
1284: invokeinterface #142, 1// InterfaceMethod java/util/Iterator.next:()Ljava/lang/Object;
1289: astore      7
1291: aload      4
1293: aload      7
1295: checkcast   #41 // class java/lang/Integer
1298: astore      8
1300: astore     10
1302: iconst_0
1303: istore      9
1305: aload      8
1307: dup
1308: ifnull     1323
1311: invokevirtual #145 // Method java/lang/Integer.intValue:()I
```

Class: `java.util.ArrayList$Itr`
Method: `next`
Inlined: No, size > DesiredMethodLimit
Count: 540000
iicount: 19168
Bytes: 66
Prof factor: 1.000000

Ctrl-click to inspect this method
Backspace to return

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>

Смотрим JITWatch для метода map20_collection

The screenshot shows the JITWatch application interface. The title bar reads "TriView - Source, Bytecode, Assembly Viewer - JITWatch". The "Class" field contains "com.maxssoft.func.SequenceFunctionsKt" and the "Member" dropdown shows "map20_collection(List)". Below these fields are several checkboxes: "Byt..." (checked), "Ass...", "J...", "Inlin...", and "Mou..." (checked). To the right, there are three data fields: "Bytecode size" with a value of 2183 (highlighted in red), "Native size" with a value of 32352, and "Compile time" with a value of 278ms.

The main area displays the bytecode for the method, with a yellow header "Bytecode (double click for JVM spec)". The bytecode instructions are listed as follows:

```
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
```

A tooltip is visible over the "checkcast" instruction at line 1807. It contains the following information:

```
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 каждый узел – элементарная операция.

Когда количество узлов превышает 18000, инлайнинг останавливается.

**источник: Server Compiler Inlining Messages*

<https://wiki.openjdk.org/display/HotSpot/Server+Compiler+Inlining+Messages>

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

Жирный data class

```
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,
```

```
    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,
```

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`.



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 ...

Смотрим JITWatch для метода map10_sequence

TriView - Source, Bytecode, Assembly Viewer - JITWatch

Class: Member:

So... Bytec... Asse... Mouse...

Bytecode size	Native size	Compile time
19	1064	2ms

Bytecode (double click for JVM spec)

```
0: aload_0
1: getfield      #29  // Field transformer:Lkotlin/jvm/functions/Function1;
4: aload_0
5: getfield      #38  // Field iterator:Ljava/util/Iterator;
8: invokeinterface #48, 1// InterfaceMethod java/util/Iterator.next: ()Ljava/lang/Object;
13: invokeinterface #54, 2// InterfaceMethod kotlin/jvm/functions/Function1.invoke: (Ljava/lang/Object;)Ljava/lang/Object;
18: areturn
```

Class: null
Method: null
Inlined: No, virtual call
Count: 227839
Prof factor: 1.000000
Virtual call, not inlined

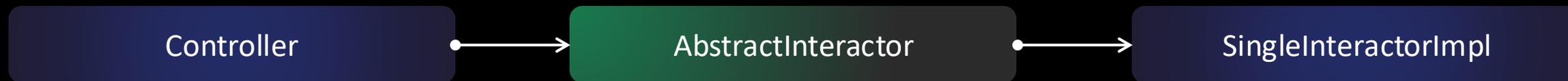
Почему `invokevirtual` не инлайнится?

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/>,

Через один `invokevirtual` вызывается только одна реализация



В этом случае JVM заинлайнит вызов `invokevirtual`.

Через один `invokevirtual` вызывается несколько реализаций



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

Давайте разберем интересный пример

```
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) }  
}
```

```
.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))
```

```
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

Ошибка инлайнинга `invokevirtual`

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

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

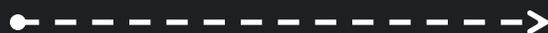
В случае преобразования через `SingleMapper` инлайнинг работает, а в случае стандартного преобразования – нет.

Давайте поставим коллекции в равные условия

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

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

```
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) }  
}
```



```
.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

Теперь Sequence обгоняют коллекции

Кол-во операций	Standalone			LambdaMapper		
	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%

Проблема мегаморфизма

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

Коллекции выигрывают, так как они инлайнятся.

Но проблема несколько шире и касается не только `Sequence`

Любое использование функционального типа в Kotlin блокирует возможность инлайнига из-за проблемы мегаморфизма

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

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

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
```

Результаты выполнения теста на Android в Jetpack Benchmark

Android
Jetpack Benchmark

Размер списка	Collection (ns)	Sequence (ns)	%
map 2	4 938 372	4 938 635	0%
map 3	7 276 791	6 768 868	8%
map 5	11 379 786	9 556 890	19%
map 10	17 615 430	22 019 028	24%

Различия в работе компилятора кода

HotSpot VM – использует только динамическую JIT компиляцию

Android RunTime – использует смешанную AOT и JIT компиляцию

Виды компиляции

JIT (just-In-time) компиляция

Код компилируется динамически, по мере его выполнения в среде исполнения

Компилируется только тот код, который исполняется

AOT (ahead-of-time) компиляция

Весь код компилируется заранее

Компиляция кода никак не связана с его исполнением

Компиляция кода в HotSpot VM

Как работает JIT компиляция

Сначала метод исполняется в режиме интерпретации байт-кода
(примерно в 10 раз медленней скомпилированного кода)

Затем, после некоторого числа вызовов метода система запускает его компиляцию

Главный критерий – число вызовов метода на исполнение

Если код будет исполнен всего один раз, то компиляция этого кода — пустая трата времени.

При последующих вызовах этого метода, будет исполняться уже скомпилированный машинный код

JIT - Оптимизации на основе статистики вызовов

Виртуальная машина накапливает статистику вызовов метода во время работы приложения

На основе накопленной статистики, виртуальная машина выполняет дополнительные оптимизации кода, невозможные при статической компиляции:

- девиртуализация методов
- более агрессивный инлайнинг для горячих методов
- некоторое количество дополнительных оптимизаций

Плюсы и минусы JIT компиляции

Возможность быстрого старта без долгой предварительной компиляции

Возможность более глубоких оптимизаций, недоступных при статической компиляции (выигрыш до 10-15%)

Экономия места на диске (объем скомпилированного кода в 3-4 раза больше объема байт-кода)

Замедленная работа приложения на начальном этапе

Прогрев: Для достижения стабильной скорости, приложение должно прогреться в течение нескольких минут или часов

AOT + JIT компиляция в Android

В Android используются сразу два вида компиляции:

Статическая AOT компиляция

Часть кода, который нужен при запуске, компилируется сразу в момент установки приложения (на основе Baseline профилей)

Это позволяет уйти от первого запуска в режиме интерпретации и сразу получить очень быстрый старт приложения

Динамическая JIT компиляция

Весь остальной код компилируется только в момент исполнения кода через JIT компиляцию

Плюсы комбинированной модели компиляции в Android (AOT + JIT)

Быстрая установка (компилируется только тот код, который нужен при старте)

Возможность задать участки кода для AOT компиляции через Baseline профили

Очень быстрый первый старт приложения

Возможность более глубоких оптимизаций кода через JIT компиляцию

Экономия места – компилируется только тот код, который реально используется

Нюансы компиляции в Android (AOT + JIT)

Для AOT и JIT компиляции используется одна имплементация компилятора

Внутри компилятора много ветвлений в зависимости от типа компиляции

Имплементация одна, но работает она по-разному для AOT и JIT режимов

Нет уровней компилятора C1-C2, более простой inliner

Как снять логи компиляции в Android?

```
> adb root  
> adb shell stop  
> adb shell setprop dalvik.vm.extra-opts -verbose:compiler,deopt,jit,oat  
> adb shell start
```



Включение логов JIT компилятора

Логи компиляции на Jetpack Benchmark

Запустили тест на Jetpack Benchmark

И ничего не увидели в логах компиляции....

Jetpack Benchmark запускает для тестов принудительную AOT компиляцию

В этом режиме логи инлайнера не пишутся

Тогда мы решили попробовать прогнать наши тесты на обычной релизной сборке, без использования Jetpack Benchmark

Результаты теста на обычной релизной сборке

Android release apk

Размер списка	Collection (ns)		Sequence (ns)		%
	First 1000	Last 1000 (JIT)	First 1000	Last 1000 (JIT)	
map 2	4 520 599	3 962 711	4 023 827	3 510 455	13%
map 3	6 641 629	5 721 141	5 169 879	4 725 174	21%
map 5	9 974 508	9 438 369	8 016 803	7 341 876	28%
map 10	19 892 175	18 935 519	14 976 344	14 295 514	32%

Прогрев теста выполнялся в течение 15 минут и затем брались усредненные показатели по последним 1000 прогонам

Отчетливо виден результат работы JIT компилятора (разница между первыми и последними 1000 прогонов)

И наконец то мы увидели реальные логи компиляции с инлайнингом

```
Starting pass: inliner
Try inlining call: kotlin.jvm.functions.Function1 kotlin.sequences.TransformingSequence.access$getTransformer$
Success: kotlin.jvm.functions.Function1 kotlin.sequences.TransformingSequence.access$getTransformer$p(kotlin.seq
Try inlining call: java.lang.Object java.util.Iterator.next()
Fail: Method java.lang.Object kotlin.sequences.TransformingSequence$iterator$1.next()
is not inlined because it has reached its polymorphic recursive call budget.
Fail: Call to java.lang.Object java.util.Iterator.next()
from inline cache is not inlined because none of its targets could be inlined
Try inlining call: java.lang.Object kotlin.jvm.functions.Function1.invoke(java.lang.Object)
Fail: Interface or virtual call to java.lang.Object kotlin.jvm.functions.Function1.invoke(java.lang.Object)
is megamorphic and not inlined
Starting pass: constant_folding$after_inlining
Starting pass: instruction_simplifier$after_inlining
Starting pass: dead_code_elimination$after_inlining
```

В Android чуда не произошло и Sequence точно также не инлайнятся из-за проблемы мегаморфизма

Почему Collection проигрывают в Android?

Sequence в Android не инлайнятся из-за проблемы мегаморфизма

Но при этом в Android Sequence выигрывают у Collection, а в HotSpot VM наоборот проигрывают

Возможно в Android проблема с инлайнингом Collection???

Давайте смотреть логи компиляции Collection

```
Try inlining call: boolean java.util.Collection.add(java.lang.Object)
  Try inlining call: void java.util.ArrayList.ensureCapacityInternal(int)
  Note: Calls in void java.util.ArrayList.ensureCapacityInternal(int)
  will not be inlined because the outer method has reached its instruction budget limit. 11
  Fail: Method void java.util.ArrayList.ensureCapacityInternal(int)
  is not inlined because the outer method has reached its instruction budget limit.
Success: boolean java.util.ArrayList.add(java.lang.Object)
Try inlining call: java.lang.Object java.util.Iterator.next()
Fail: Method java.lang.Object java.util.ArrayList$Itr.next()
is not inlined because its code item is too big: 48 > 32
Try inlining call: int java.lang.Integer.intValue()
Success: int java.lang.Integer.intValue()
```

Метод не смог заинлайниться, так как его размер больше 32 инструкций байт-кода

Причины проигрыша Collection в Android

54 / 61

Судя по логам, достаточно многие методы не укладываются в это ограничение в 32 инструкции байт-кода

На HotSpot VM таких проблем не возникает, там инлайнятся намного более объемные методы

Collection проигрывают в Android, потому что виртуальная машина Android просто не умеет инлайнить код также эффективно как HotSpot VM

Инлайнинг в Android устроен намного проще

В Android нет зависимости от частоты использования метода

В HotSpot VM более гибкие и расширенные лимиты бюджета инлайнинга

Android

Максимальный размер для инлайнинга

InlineMaxCodeUnits = 32

Максимальный суммарный размер метода

MaximumNumberOfTotalInstructions = 1 024

HotSpot VM

Максимальный размер для инлайнинга

MaxInlineSize = 35 **FreqInlineSize** = 325

Максимальный суммарный размер метода

DesiredMethodLimit = 8 000

HugeMethodLimit = 8 000

NodeCountInliningCutoff = 18 000

Мы попробовали провести эксперимент и увеличили параметры инлайнинга

InlineMaxCodeUnits = 32 -> **128**,

MaximumNumberOfTotalInstructions = 1024 -> **4096**

Размер списка	Collection			Sequence		
	Standard	Extended	%	Standard	Extended	%
map 2	4 515 748	3 898 300	16%	3 473 127	3 432 349	1%
map 3	6 611 315	5 716 844	16%	4 739 007	4 613 998	3%
map 5	10 532 289	9 391 219	12%	7 415 816	7 307 968	1%
map 10	21 828 025	18 749 598	16%	14 175 795	14 179 717	0%

Негативные эффекты

Объем скомпилированного кода вырос в **3-4 раза**

Сильно увеличилось время AOT компиляции, **примерно в 4-5 раз**

Кажется игра не стоит свеч

Android Runtime и HotSpot VM решают немного разные задачи

HotSpot VM заточен на максимальную оптимизацию долго живущих приложений

Android Runtime старается достичь максимально быстрого старта и экономит память

У приложений Android как правило очень короткий жизненный цикл

Может Android это и не нужно?

Мы попробовали провести еще один эксперимент

Очень уж нам хотелось ускорить Android

Мы решили попробовать внедрить в инлайнер Android наработки из HotSpot VM, расширяющие лимиты инлайнинга для горячих методов

Кажется что цена здесь не должна быть высокой, но это может существенно ускорить выполнение кода

Эксперимент с горячими методами

61 / 61

In processing...

Мы очень старались успеть к весеннему Mobius, но не успели

Мы расскажем о результатах этого эксперимента на [Mobius Autumn 2025](#)

Покажем как устроен инлайнер Android на уровне кода

И расскажем как мы его доработали

И кто знает, возможно нам все таки удастся ускорить следующий Android?



Мои статьи на Хабр



Мои статьи на
ProAndroidDev



linkedIn: [sidorov-max](#)

Спасибо за внимание

Максим Сидоров, Максим
Митюшкин

Системные сервисы, Salute TV

