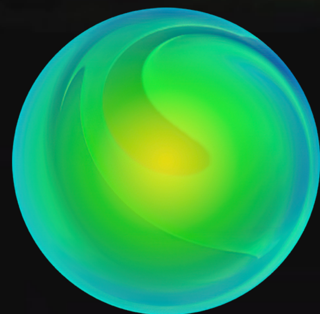
Как избавиться от рекурсии или как мой код попал в Google



Максим Сидоров,

Системные сервисы, Salute TV

SBER DEVICES

Mobius spring 2023

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

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

SBER DEVICES

Никогда такого не было и вот опять

На этот раз мне просто повезло)

Функция **ViewGroup.descendants** (расчет иерархии View) работает в сотни раз медленней чем могла бы

оптимизируем)

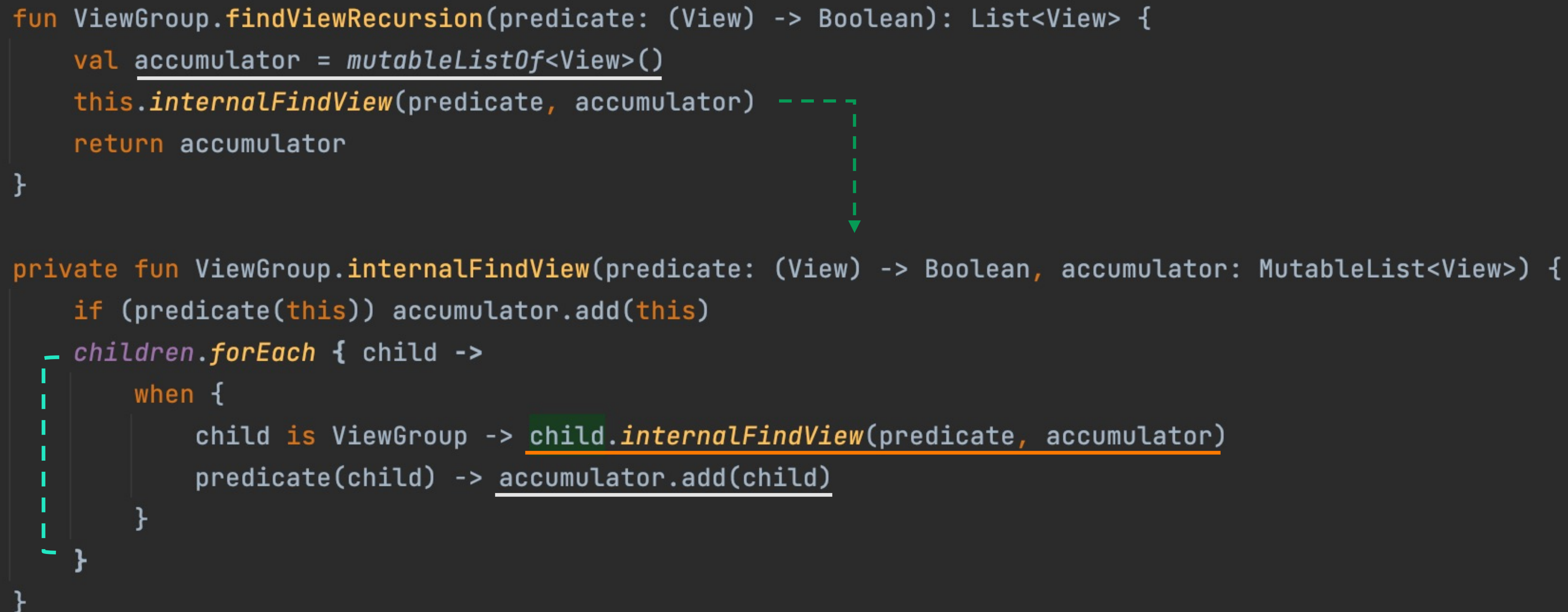
SBER DEVICES

Исследование различных способов оптимизации рекурсии

- Хвостовая рекурсия и **tailrec** функции
- Оптимизация через очередь
- Оптимизация через ленивые методы sequence.**yield** (засада была здесь)
- Оптимизация через **DeepRecursiveFunction**
- Моя оптимизация через ленивый TreeIterator (уже добавлена в androidx.core-ktx.core)

У меня есть задача, которую я люблю давать на собесах

Напиши функцию поиска вьюшек по предикату



SBER DEVICES

А как можно обработать иерархию View без использования рекурсии?

Магическое ключевое слово, оптимизирующее любую рекурсию

— Заклинание **tailrec**

Я не люблю магию в программировании, потому давайте разбираться...

SBER DEVICES

Хвостовая рекурсия

Хвостовая рекурсия - это частный случай рекурсии, при котором рекурсивный вызов является последней операцией перед возвратом из функции.

Если упрощенно, то функция должна вернуть либо конечный результат, либо вернуть результат своего рекурсивного вызова.

SBER DEVICES

Пример функции, вычисляющей последовательность Фибоначчи

```
tailrec fun fibonacci(n: Int, a: BigInteger, b: BigInteger): BigInteger {
    return if (n == 0) b else fibonacci(n - 1, a + b, a)
}
```

Декомпилированный байт-код

Рекурсия превратилась в цикл

```
public final BigInteger fibonacci(
        int n, BigInteger a, BigInteger b
) {
    while(true) {
        if (n == 0) {
            return b;
        }

        n = n - 1;
        BigInteger var10001 = a.add(b);
        b = a;
        a = var10001;
    }
}
```

SBER DEVICES

Немного поменяем функцию

```
tailrec fun fibonacciModified(n: Int, a: BigInteger, b: BigInteger): BigInteger {
    var result = if (n == 0) b else fibonacciModified( n: n - 1,  a: a + b, a)
    return result
}
```

Декомпилированный байт-код

И снова здравствуйте, рекурсия снова с нами

```
public final BigInteger fibonacciModified(
        int n, BigInteger a, BigInteger b
) {
    BigInteger var10000;
    if (n == 0) {
        var10000 = b;
    } else {
        var10000 = this.fibonacciModified( n: n - 1, a.add(b), a);
    }

    return var10000;
}
```

Tailrec и хвостовая рекурсия

Все что нам скажет на это kotlin, это всего лишь малозаметный warning

```
tailrec fun fibonacciModified(n: Int, a: BigInteger, b: BigInteger): BigInteger {
    ( n: n - 1, a: a + b, a)
}
```

A function is marked as tail-recursive but no tail calls are found

'tailrec' marks a function as tail-recursive (allowing the compiler to replace recursion with iteration)

Kotlin_demo.app.main

Хотя на мой взгляд это состояние ошибки

В данном случае мы потеряли явно указанную оптимизацию и можем получить в runtime ошибку StackOverflow

Я создал на это Issue в JetBrains (Tailrec)

Очень нужны ваши лайки, чтобы продвинуть Issue

SBER DEVICES

Чем плоха рекурсия

У нас есть стек, имеющий фиксированный размер (-**Xss** = 1Mb)

Каждый вызов рекурсии отъедает кусочек этого стека

При достаточной глубине рекурсии стек закончится и
мы получим StackOverflowException

SBER DEVICES

Типовая задача обхода иерархии через рекурсию

Как же решить эту задачу без рекурсии?

Напиши функцию поиска вьюшек по предикату

```
fun ViewGroup.findViewRecursion(predicate: (View) -> Boolean): List<View> {
    val accumulator = mutableListOf<View>()
    this.internalFindView(predicate, accumulator)
    return accumulator
}

private fun ViewGroup.internalFindView(predicate: (View) -> Boolean, accumulator: MutableList<View>) {
    if (predicate(this)) accumulator.add(this)
    children.forEach { child ->
        when {
            child is ViewGroup -> child.internalFindView(predicate, accumulator)
            predicate(child) -> accumulator.add(child)
        }
    }
}
```

SBER DEVICES

Стандартный способ избавления от рекурсии через очередь (Queue)

```
fun ViewGroup.findViewQueue(predicate: (View) -> Boolean): List<View> {
    val accumulator = mutableListOf<View>()
    val queue: Queue<View> = LinkedList()

    queue.add(this) // add self as a first element of queue
    while (queue.isNotEmpty()) {
        val view = queue.poll() // get and remove next item from queue

        if (predicate(view)) {
            accumulator.add(view)
        }

        if (view is ViewGroup) { // add to queue all child for current view
            view.children.forEach { queue.add(it) }
        }
    }

    return accumulator
}
```

SBER DEVICES

Оптимизация через Queue

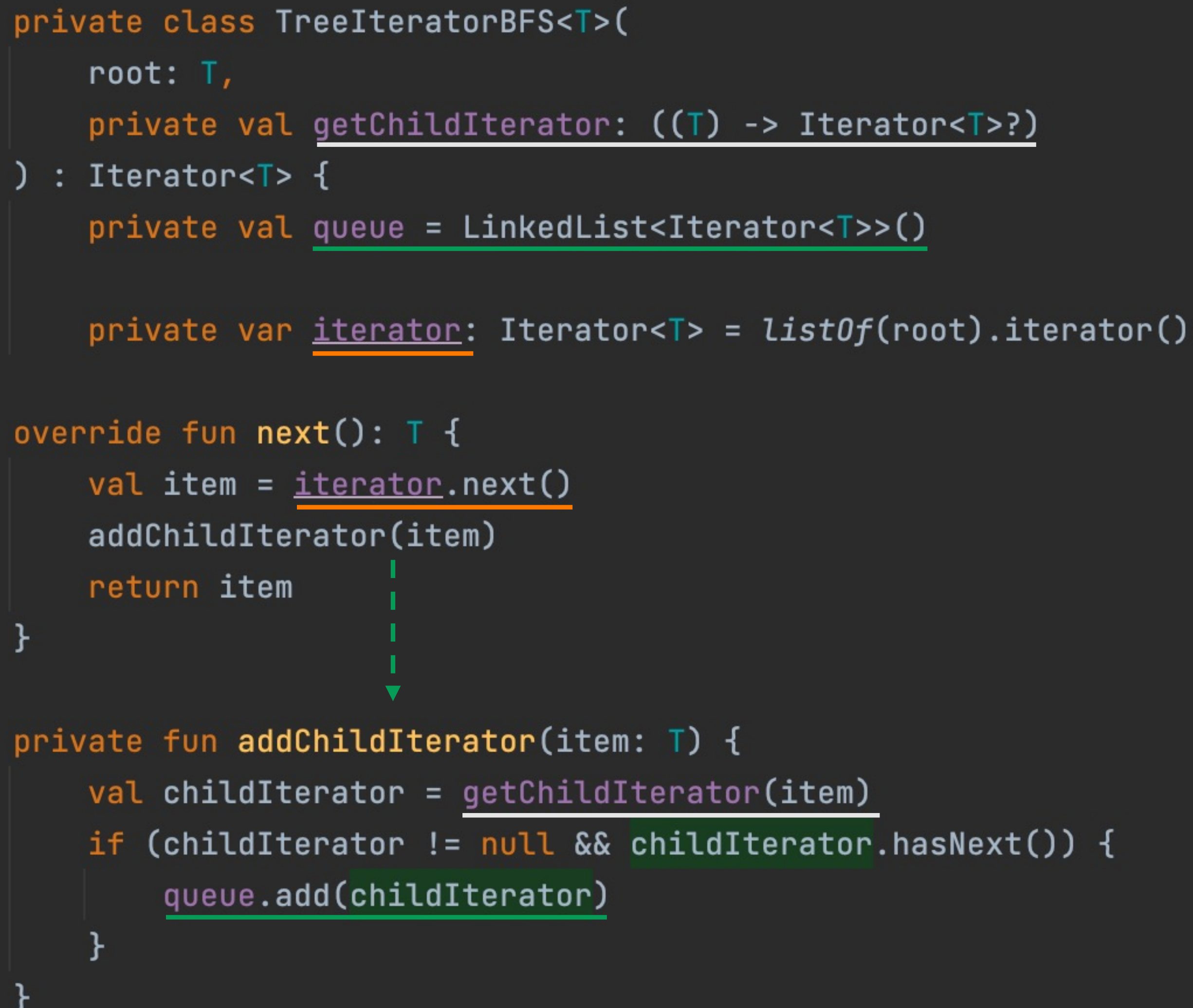
Функция	**Глубина иерархии**			
	1 000 (ns)	**3 000 (ns)**	**5 000 (ns)**	
Recursion	17 492	52 453	Error	
Queue	54 254	152 197	246 697	x3

Оптимизация через очередь работает в 3 раза медленней рекурсии

Но для рекурсии на глубине иерархии больше 3 000 мы получаем StackOverflowException (*зависит от параметра в VM*)

SBER DEVICES

TreeIterator – ленивая итерация по дереву (поиск в ширину)




```kotlin
override fun hasNext(): Boolean {
    val hasNext = when {
        iterator.hasNext() -> true
        queue.isNotEmpty() -> {
            iterator = queue.pollFirst()
            true
        }
        else -> false
    }
    return hasNext
}
```

SBER DEVICES

Пример реализации поиска вьюшек через TreeIterator

```
fun ViewGroup.findViewTreeIterator(predicate: (View) -> Boolean): Sequence<View> {
    return TreeIteratorBFS<View>( root: this) { view ->
        (view as? ViewGroup)?.children?.iterator()
    }
        .asSequence()
        .filter { predicate(it) }
}
```

Обработка иерархии будет происходить лениво, без рекурсии и выделения лишней памяти.

Вся логика итерации спрятана в итераторе, а логику обработки элементов можно выносить наружу через sequence

SBER DEVICES

Оптимизация через TreeIterator

Функция	Глубина иерархии			
	1 000 (ns)	**3 000 (ns)**	**5 000 (ns)**	
Recursion	17 492	52 453	Error	
Queue	54 254	152 197	246 697	x3
TreeIterator	33 261	96 104	170 995	x2

TreeIterator работает в 2 раза медленней чем рекурсия

TreeIterator более гибкий и легко адаптируется к стандартным преобразованиям sequence

SBER DEVICES

Issue в JetBrains (TreeIterator)

Я создал Issue в JetBrains с предложением добавить мой TreeIterator в публичное api sequence

- Добавил реализации для поиска в ширину и глубину и реализовал удобный билдер
- В планах добавить также реализацию для бинарного дерева

Issue в JetBrains (TreeIteratior)

Очень нужны ваши лайки, чтобы продвинуть Issue

gitHub репозиторий с исходниками

SBER DEVICES

DeepRecursiveFunction – рекурсия через suspend функции

```kotlin
fun ViewGroup.findViewDeepRecursive(predicate: (View) -> Boolean): List<View> {
    val result = mutableListOf<View>()
    val recursion = DeepRecursiveFunction<View, List<View>> { this: DeepRecursiveScope<View, List<View>> view ->
        if (predicate(view)) {
            result.add(view)
        }
        if (view is ViewGroup) {
            view.children.forEach { it: View
                callRecursive(it)
            }
        }

        result ^DeepRecursiveFunction
    }
    return recursion( value: this)
}
```

```kotlin
sealed class DeepRecursiveScope<T, R> {
    /**
     * Выполняет рекурсивный вызов функции,
     * помещая кадр активации вызова в кучу, а не в стек
     */
    abstract suspend fun callRecursive(value: T): R
}
```

SBER DEVICES

DeepRecursiveScope – как это работает под капотом

```kotlin
private class DeepRecursiveScopeImpl<T, R>(
    block: suspend DeepRecursiveScope<T, R>.(T) -> R,
    value: T
) : DeepRecursiveScope<T, R>(), Continuation<R> {
    // Active function block
    private var function: DeepRecursiveFunctionBlock = block as DeepRecursiveFunctionBlock

    // Value to call function with
    private var value: Any? = value

    // Continuation of the current call
    private var cont: Continuation<Any?>? = this as Continuation<Any?>

    // Completion result (completion of the whole call stack)
    private var result: Result<Any?> = UNDEFINED_RESULT
```

SBER DEVICES

DeepRecursiveScope

```
fun runCallLoop(): R {
    while (true) {
        // Note: cont is set to null in DeepRecursiveScopeImpl.resumeWith when the whole computation completes
        val result = this.result
        val cont = this.cont
            ?: return (result as Result<R>).getOrThrow() // done -- final result
        // call "function" with "value" using "cont" as completion
        val r = try {
            // This is block.startCoroutine(this, value, cont)
            function.startCoroutineUninterceptedOrReturn( receiver: this, value, cont)
        } catch (e: Throwable) {
            cont.resumeWithException(e)
            continue
        }
        // If the function returns without suspensio
        if (r !== COROUTINE_SUSPENDED)
            cont.resume(r as R)
    }
}
```

```
override suspend fun callRecursive(value: T): R = suspendCor
    // calling the same function that is currently active
    this.cont = cont as Continuation<Any?>
    this.value = value
    COROUTINE_SUSPENDED ^suspendCoroutineUninterceptedOrReturn
}
```

```
override fun resumeWith(result: Result<R>) {
    this.cont = null
    this.result = result
}
```

SBER DEVICES

DeepRecursiveFunction – рекурсия через suspend функции

```kotlin
fun ViewGroup.findViewDeepRecursive(predicate: (View) -> Boolean): List<View> {
    val result = mutableListOf<View>()
    val recursion = DeepRecursiveFunction<View, List<View>> { view ->
        if (predicate(view)) {
            result.add(view)
        }
        if (view is ViewGroup) {
            view.children.forEach {
                callRecursive(it)
            }
        }

        result
    }
    return recursion(this)
}
```

SBER DEVICES

DeepRecursiveFunction

Функция	Глубина иерархии			
	1 000 (ns)	3 000 (ns)	5 000 (ns)	
Recursion	17 492	52 453	Error	
Queue	54 254	152 197	246 697	x3
TreeIterator	33 261	96 104	170 995	x2
DeepRecursive	70 048	173 037	281 738	x4

DeepRecursiveFunction работает в 4 раза медленней чем стандартная рекурсия

Позволяет оптимизировать любую рекурсию

Отсутствует ленивость вычислений и не получится адаптировать к sequence

SBER DEVICES

Ленивое формирование sequence

ViewGroup.descendants

```kotlin
val ViewGroup.descendants: Sequence<View>
    get() = sequence { this: SequenceScope<View>
        forEach { child ->
            yield(child)
            if (child is ViewGroup) {
                yieldAll(child.descendants)
            }
        }
    }
```

```kotlin
/**
 * Возвращает значение итератору последовательности
 * и приостанавливает работу до тех пор,
 * пока не будет запрошено следующее значение.
 */
abstract suspend fun yield(value: T)
```

Теперь функция поиска вьюшек стала тривиальной

```kotlin
viewGroup.descendants.filter { it.isVisible }
```

SBER DEVICES

ViewGroup.descendants

Функция	Глубина иерархии		
	1 000 (ns)	3 000 (ns)	5 000 (ns)
Recursion	17 492	52 453	Error
Queue	54 254	152 197	246 697
TreeIterator	33 261	96 104	170 995
DeepRecursive	70 048	173 037	281 738
descendants	**16 806 854**	**195 829 303**	**558 457 428**

Yield работает в **1000** раз медленней рекурсии

Это ШОК,
Настолько медленно???

SBER DEVICES

Как работает yield под капотом

SequenceBuilderIterator

```
private const val State_NotReady: State = 0
private const val State_ManyNotReady: State = 1
private const val State_ManyReady: State = 2
private const val State_Ready: State = 3
private const val State_Done: State = 4
private const val State_Failed: State = 5

private class SequenceBuilderIterator<T> : SequenceScope<T>(), Iterator<T>, Continuation<Unit> {
    private var state = State_NotReady  // состояние вычисления hasNext
    private var nextValue: T? = null    // следующий элемент
    private var nextIterator: Iterator<T>? = null    // следующий итератор для получения элементов
    var nextStep: Continuation<Unit>? = null         // следующая suspend функция для получения элементов
```

SBER DEVICES

Как работает yield под капотом

```kotlin
override fun hasNext(): Boolean {
    while (true) {
        when (state) {
            State_NotReady -> {}    // Если данные берутся из Yield оператора, то ???
            State_ManyNotReady ->   // Если данные берутся из итератора, то возвращает nextIterator.hasNext()
                if (nextIterator!!.hasNext()) {
                    state = State_ManyReady
                    return true
                } else {
                    nextIterator = null
                }
            State_Done -> return false    // Если элементы закончились, то прекращает итерацию по sequence
            State_Ready, State_ManyReady -> return true      // Следующий элемент уже вычислен
            else -> throw exceptionalState()
        }

        state = State_Failed
        val step = nextStep!!  // Берет следующую suspend функцию yield, yieldAll
        nextStep = null
        step.resume(Unit)       // Вызывает вычисление следующего элемента через suspend функцию
    }
}
```

SBER DEVICES

Как работает yield под капотом

```
override suspend fun yield(value: T) {
    nextValue = value      // Запоминает значение, которое было передано в yield как следующее значение
    state = State_Ready    // Выставляет статус готовности следующего значения
    return suspendCoroutineUninterceptedOrReturn { c ->    // Приостанавливает выполнение suspend функции
        nextStep = c       // Запоминает точку восстановления suspend функции в nextStep
        COROUTINE_SUSPENDED ^suspendCoroutineUninterceptedOrReturn
    }
}

override suspend fun yieldAll(iterator: Iterator<T>) {
    if (!iterator.hasNext()) return
    nextIterator = iterator
    state = State_ManyReady
    return suspendCoroutineUninterceptedOrReturn { c ->
        nextStep = c
        COROUTINE_SUSPENDED ^suspendCoroutineUninterceptedOrReturn
    }
}
```

SBER DEVICES

Как работает yield под капотом

```
override fun hasNext(): Boolean {
    while (true) {
        when (state) {
            State_NotReady -> {}
            State_Ready, State_ManyReady -> return true
            else -> throw exceptionalState()
        }

        state = State_Failed
        val step = nextStep!!
        nextStep = null
        step.resume(Unit)
    }
}
```

Лямбда формирующая Sequence

```
yield(value: T) ---> state = State_Ready ---> suspend
```

SBER DEVICES

Как работает yield под капотом

```
return sequence { this: SequenceScope<Int>
    println("call yield(0), nextValue = 0")
    println("suspend ---------")
    yield( value: 0)
    // ----- suspending -------

    println("call yieldAll(1,2,3)")
    println("nextIterator = iterator(1,2,3)")
    println("suspend----------")
    yieldAll(listOf(1,2,3))
    // ----- suspending -------

    println("start for cycle")
    for (i in 4 ≤.. ≤ 8) {
        println("call yield($i), nextValue = $i")
        println("suspend----------")
        yield(i)
        // ----- suspending -------
    }
}
```

SBER DEVICES

Как работает yield под капотом

```kotlin
return sequence { this: SequenceScope<Int>
    println("call yield(0), nextValue = 0")
    println("suspend ---------")
    yield( value: 0)
    // ----- suspending -------

    println("call yieldAll(1,2,3)")
    println("nextIterator = iterator(1,2,3)")
    println("suspend----------")
    yieldAll(listOf(1,2,3))
    // ----- suspending -------

    println("start for cycle")
    for (i in 4 ≤ .. ≤ 8) {
        println("call yield($i), nextValue = $i")
        println("suspend----------")
        yield(i)
        // ----- suspending -------
    }
}
```

Вывод в Terminal

```
iterator.hasNext() -> call yield(0), nextValue = 0
suspend ---------

iterator.next() -> item: 0

iterator.hasNext() -> call yieldAll(1,2,3)
nextIterator = iterator(1,2,3)
suspend----------

iterator.next() -> item: 1
iterator.hasNext() -> iterator.next() -> item: 2
iterator.hasNext() -> iterator.next() -> item: 3

iterator.hasNext() -> start for cycle
call yield(4), nextValue = 4
suspend----------

iterator.next() -> item: 5
iterator.hasNext() -> call yield(6), nextValue = 6
suspend----------
```

SBER DEVICES

Почему же рекурсия через **ViewGroup.descendants** такая медленная?

```kotlin
val ViewGroup.descendants: Sequence<View>
    get() = sequence { this: SequenceScope<View>
        forEach { child ->
            yield(child)
            if (child is ViewGroup) {
                yieldAll(child.descendants)
            }
        }
    }
```

Функция	**Глубина иерархии 1000**
Recursion	17 492
TreeIterator	33 261
DeepRecursive	70 048
Yield	**16 806 854**

- Каждый уровень иерархии создает новую последовательность
- На 1000 уровней вложенности будет создано 1000 sequence, вложенных друг в друга
- Каждый вызов **next** – это вызов 1000 next с проверками состояний и копированием данных

SBER DEVICES

Issue в Google (ViewGroup.descendants)

Я создал Issue в Google с описанием проблемы и предложил свое решение, основанное на TreeIterator

Моя реализация ViewGroup.descendants на базе TreeIterator уже влита в репозиторий Google

core/core-ktx/src/main/java/androidx/core/view/ViewGroup.kt

[Issue в Google (ViewGroup.descendant)]

[gitHub репозиторий с исходниками]

SBER DEVICES

Рекурсия является самым быстрым решением, но имеет ограничение

Любой способ отказа от рекурсии имеет свою цену и работает в несколько раз медленней рекурсии

Функция	**Глубина иерархии**			
	1 000 (ns)	**3 000 (ns)**	**5 000 (ns)**	
Recursion	17 492	52 453	Error	
Queue	54 254	152 197	246 697	x3
TreeIterator	33 261	96 104	170 995	x2
DeepRecursive	70 048	173 037	281 738	x4

SBER DEVICES

Мои статьи на Хабр

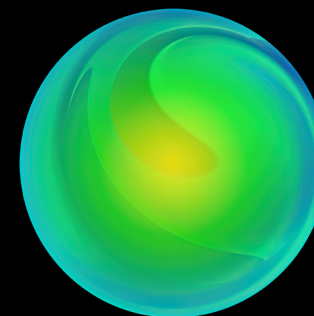
Спасибо за внимание

Мои статьи на ProAndroidDev

linkedIn: sidorov-max

Максим Сидоров,

Системные сервисы, Salute TV



SBER DEVICES

Важно понимать – здесь нельзя вызывать любые suspend функции

```
private suspend fun mySuspendFunc(value: Int) {
    println(value)
}

private fun createYieldData1(): Sequence<Int> {
    return sequence { this: SequenceScope<Int>
        println("start for cycle")
        for (i in 4 ≤ .. ≤ 8) {
            println("call yield($i), nextValue = $i")
            println("suspend----------")

            mySuspendFunc( value: 0)

            yield(i)
            // ----- suspending -------
        }
    }
}
```

Restricted suspending functions can only invoke member or extension suspending functions on their restricted coroutine scope

SBER DEVICES

Что такое ограниченный Scope

```
@RestrictsSuspension
@SinceKotlin( version: "1.7")
sealed class DeepRecursiveScope<T,
    new *
    abstract suspend fun callRecurs
    new *
    abstract suspend fun <U, S> Dee
}

new *
@RestrictsSuspension
@SinceKotlin( version: "1.3")
abstract class SequenceScope<in T> internal constructor() {
    new *
    abstract suspend fun yield(value: T)
    new *
    abstract suspend fun yieldAll(iterator: Iterator<T>)
}
```

```
/**
 * Classes and interfaces marked with this annotation
 * are restricted when used as receivers for extension `suspend` functions.
 * These `suspend` extensions can only invoke other member or extension `suspend`
 * functions on this particular receiver and are restricted
 * from calling arbitrary suspension functions.
 */
new *
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.BINARY)
annotation class RestrictsSuspension
```

SBER DEVICES