АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 9

ПЛАН

- Корутины и контексты
- Async/await
- Flow

ДИСПЕТЧЕРИЗАЦИЯ

- Корутины кооперативны
- Suspention point - место, где корутина может уступить место другим
- Может возобновиться на другой нити
- На какой именно - решает контекст

КОНТЕКСТ

- Каждая корутина запускается в некотором контексте
- Контекст представлен интерфейсом CoroutineContext
- CoroutineContext - это своего рода словарь
- Со своеобразным интерфейсом

КОНТЕКСТ

```
1 public interface CoroutineContext {
2     /**
3      * Returns the element with the given
4      * [key] from this context or `null`.
5      */
6     public operator fun <E : Element>
7     get(key: Key<E>): E?
8 // ..............
```

КОНТЕКСТ

```
// ..............
    /**
     * Accumulates entries of this context starting with
     * [initial] value and applying [operation]
     * from left to right to current accumulator value
     * and each element of this context.
     */
    public fun <R> fold(
        initial: R, operation: (R, Element) -> R
    ): R
// ..............
```

КОНТЕКСТ

```
// ..............
    /**
     * Returns a context containing elements from
     * this context and elements from  other [context].
     * The elements from this context with the same key
     * as in the other one are dropped.
     */
    public operator fun plus(
        context: CoroutineContext
    ): CoroutineContext =
          /* .......... */
// ..............
```

КОНТЕКСТ

```
1  // ...............
2      /**
3       * Returns a context containing elements from
4       * this context, but without an element with
5       * the specified [key].
6       */
7      public fun minusKey(key: Key<*>): CoroutineContext
8
9      /**
10      * Key for the elements of [CoroutineContext].
11      * [E] is a type of element with this key.
12      */
13     public interface Key<E : Element>
14 // ...............
```

КОНТЕКСТ

```
// ..............
    /**
     * An element of the [CoroutineContext].
     * An element of the coroutine context
     * is a singleton context by itself.
     */
    public interface Element : CoroutineContext {
        /**
         * A key of this coroutine context element.
         */
        public val key: Key<*>
// ..............
```

КОНТЕКСТ

```
// ..............
        public override operator fun <E> : Element
        get(key: Key<E>): E? =
            @Suppress("UNCHECKED_CAST")
            if (this.key == key) this as E else null

        public override fun <R> fold(
            initial: R, operation: (R, Element) -> R
        ): R =
            operation(initial, this)
// ..............
```

КОНТЕКСТ

```
// ................
        public override fun minusKey(
            key: Key<*>
        ): CoroutineContext =
            if (this.key == key) EmptyCoroutineContext
            else this
    }
```

ВИД СНАРУЖИ

```
1 fun main() {
2     runBlocking {
3         coroutineContext.fold(Unit) { a, b ->
4             println("key class" + b.key.javaClass)
5             println("value class" + b[b.key]?.javaClass)
6             println("key" + b.key)
7             println("value" + b[b.key])
8         }
9 // ..................
```

ВИД СНАРУЖИ

```
// ...................
        println()

        println(coroutineContext[Job.Key])
        println(
            coroutineContext[ContinuationInterceptor.Key]
        )

        println(coroutineContext[Job])
        println(
            coroutineContext[ContinuationInterceptor]
        )
    }
}
```

РАЗБЕРЕМ

- Контекст может включать несколько элементов
- В нашем случае - два
- Один описывает детали Job-а
- Другой - способ диспетчеризации

КОНТЕКСТ ЗАПУСКАЕМОЙ КОРУТИНЫ

- В первом приближении - job-часть определяется типом корутины
- Часть дипетчеризации - тем, что было передано параметром в launch или аналог
- По умолчанию - передается CoroutineContext.EMPTY
- Что означает наследование способа диспетчеризации

ПРИМЕР

```
fun main() {
    println(Thread.currentThread())
    runBlocking {
        println(coroutineContext[ContinuationInterceptor])
        repeat(5) {
            launch {
                println(
                    "start: ${Thread.currentThread()}"
                )
// ...................
```

ПРИМЕР

```
// ..................
                Thread.sleep(3000)
                println(
                    "finish: ${Thread.currentThread()}"
                )
            }
            val t = Thread.currentThread()
            println(
                "launched $it from $t"
            )
            // delay(1)
        }
    }
}
```

ПРИМЕР

```
fun current() = Thread.currentThread()

fun main() {
    println(current())
    runBlocking(Dispatchers.Default) {
        println(coroutineContext[ContinuationInterceptor])
        repeat(100) { i ->
            launch {
// ...............
```

ПРИМЕР

```
// ...................
                println("start $i: ${current()}")
                Thread.sleep(3000)
                println("finish $i: ${current()}")
            }
            println("launched $i from ${current()}")
            // delay(1)
        }
    }
}
```

ПРИМЕР С ОГРАНИЧЕНИЕМ ПАРАЛЛЕЛИЗМА

```
fun current() = Thread.currentThread()

fun main() {
    println(current())
    runBlocking(
        Dispatchers.Default.limitedParallelism(3)
    ) {
        println(
            coroutineContext[ContinuationInterceptor]
        )
// ....................
```

ПРИМЕР С ОГРАНИЧЕНИЕМ ПАРАЛЛЕЛИЗМА

```
// ....................
        repeat(100) { i ->
            launch {
                println("start $i: ${current()}")
                delay(3000)
                println("finish $i: ${current()}")
            }
            println("launched $i from ${current()}")
            // delay(1)
        }
    }
}
```

ОБРАБОТКА ОШИБОК В ДЕТЯХ

- Можем сделать родителя супервизором
- Ему будут приходить извещения о проблемах детей
- Супервизор что-то с этим делает
- Может игнорировать, может перезапускать

ПРИМЕР

```
fun main() {
    runBlocking(Dispatchers.Default) {
        launch {
            launch {
                val scope = CoroutineScope(
                    SupervisorJob()
                )
                scope.launch {
                    println("one")
                    delay(1000)
                    println("two")
                    Thread.sleep(1000)
// ...................
```

ПРИМЕР

```
// ...................
                    println("going to throw....")
                    throw IOException()
                    delay(1000)
                    println("three")
                }
                println("wait 10s")
                println(
                    currentCoroutineContext().job
                                             .children
                                             .toList()
                )
// ...................
```

ПРИМЕР

```
// ..................
                println(
                    scope.coroutineContext
                         .job
                         .children
                         .toList()
                )
                delay(10000)
// ..................
```

ПРИМЕР

```
// ..................
                println(
                    currentCoroutineContext().job
                                             .children
                                             .toList()
                )
// ..................
```

ПРИМЕР

```
// ..................
                println(
                    scope.coroutineContext.job
                                          .children
                                          .toList()
                )
                println("wait 10s done")
            }
        }
    }
}
```

ПРИМЕР С ОБРАБОТКОЙ

```
fun main() {
    val action: suspend CoroutineScope.() -> Unit = {
        println("one")
        delay(1000)
        println("two")
        Thread.sleep(1000)
        println("going to throw....")
        throw IOException()
        delay(1000)
        println("three")
    }
// ..................
```

ПРИМЕР С ОБРАБОТКОЙ

```
// ..................

    runBlocking(Dispatchers.Default) {
        launch {
            launch {
                val scope = CoroutineScope(SupervisorJob())
                val job1 = scope.launch(block = action)
                val job2 = scope.launch(block = action)
// ..................
```

ПРИМЕР С ОБРАБОТКОЙ

```
// ..................
                job1.invokeOnCompletion {
                    println("complete: $it")
                    it?.let {
                        scope.launch {
                            println("once again")
                            action(scope)
                        }
                    }
                }
// ..................
```

ПРИМЕР С ОБРАБОТКОЙ

```
// ..................
                println("wait 10s")
                println(currentCoroutineContext().job
                                        .children.toList())
                println(scope.coroutineContext.job
                                        .children.toList())
                delay(10000)
// ..................
```

ПРИМЕР С ОБРАБОТКОЙ

```
// ..................
                println(currentCoroutineContext()
                                        .job.children.toList())
                println(scope.coroutineContext
                                        .job.children.toList())
                println("wait 10s done")
            }
        }
    }
}
```

ИДЕМ ДАЛЬШЕ

- suspend-функция сама по себе - не корутина
- launch порождает Job/корутину
- Внутри могут вызываться suspend-функции
- Их значения можем использовать при вызове других
- Но launch ничего не возвращает

ASYNC/DEFERED

- Defered<T> - развитие Job
- Корутина, возвращающая значение
- async - аналог launch для конструирования Defered
- У Defered есть suspend-метод await - ожидает результат

ПРИМЕР

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferred: Deferred<Int> = async {
        loadData()
    }
    println("waiting...")
    println(deferred.await())
}
// ...............
```

ПРИМЕР

```
// ..............

suspend fun loadData(): Int {
    println("loading...")
    delay(1000L)
    println("loaded!")
    return 42
}
```

ПРИМЕР

```
1  val random = Random(42)
2
3  fun main() {
4      runBlocking(Dispatchers.Default) {
5          launch {
6              val result = async {
7                  delay(random.nextLong(3000))
8                  println("after delay")
9                  123
10             }
11 // ....................
```

ПРИМЕР

```
// .....................

            println(result)
            println(result.await())
            println(result)
        }
    }
}
```

ОТЛИЧИЕ ОТ SUSPEND-ФУНКЦИИ

- suspend-функции выполняются по очереди
- Пусть suspend-функция что-то скачивает
- И мы хотим скачать что-то из двух мест и сагреггировать
- Два вызова suspend-функции внутри корутины - два последовательных скачивания
- А две async-корутины - это два параллельных скачивания

ПРИМЕР

```
1 suspend fun sleepAndTwice(v: Int): Int {
2     println("start")
3     delay(random.nextLong(3000))
4     println("after delay")
5     return v * 2
6 }
7
8 // ..........................
```

ПРИМЕР

```
// .........................

@OptIn(ExperimentalCoroutinesApi::class)
fun main() {
    val data = listOf(1, 2)

    runBlocking(Dispatchers.Default) {
        launch {
            val result = data.map { sleepAndTwice(it) }
            println("result: $result")
        }
    }
}
```

ПРИМЕР

```
1  val random = Random(42)
2
3  @OptIn(ExperimentalCoroutinesApi::class)
4  fun main() {
5      val data = listOf(1, 2)
6
7      runBlocking(Dispatchers.Default) {
8          fun sleepAndTwice(v: Int) = async {
9              println("start")
10             delay(random.nextLong(5000))
11             println("after delay")
12             v * 2
13         }
14 // ....................
```

ПРИМЕР

```
// ....................

        launch {
            val result = data.map { sleepAndTwice(it) }
                              .map { it.await() }
            println("result: $result")
        }
    }
}
```

AWAITALL

- Есть `awaitAll` для списка `Defered`
- Но он привередлив в плане исключений
- Одно падение - падение всего сразу
- Если это не устраивает - надо отдельно обрабатывать

ПРИМЕР ОБРАБОТКИ ИСКЛЮЧЕНИЯ

```
1  val random = Random(42)
2
3  @OptIn(ExperimentalCoroutinesApi::class)
4  fun main() {
5      val data = listOf(1, 2, 5, 0)
6      runBlocking(Dispatchers.Default) {
7          fun sleepAndDiv(v: Int) =
8              async(SupervisorJob()) {
9                  println("start $v")
10                 delay(random.nextLong(5000))
11                 println("after delay $v")
12                 5 / v
13             }
14 // ..............................
```

ПРИМЕР ОБРАБОТКИ ИСКЛЮЧЕНИЯ

```
// ...............................
        launch {
            val result = data.map {
                sleepAndDiv(it)
            }.map {
                try {
                    it.await()
                } catch (exc: ArithmeticException) {
// ...............................
```

ПРИМЕР ОБРАБОТКИ ИСКЛЮЧЕНИЯ

```
// ..............................
                    println(
                        it.getCompletionExceptionOrNull()
                    )
                    null
                }
            }
            println("result: $result")
        }
    }
}
```

ДРУГОЙ ПОДХОД

```
1  val random = Random(42)
2
3  @OptIn(ExperimentalCoroutinesApi::class)
4  fun main() {
5      val data = listOf(1, 2, 5, 0)
6
7      runBlocking(Dispatchers.Default) {
8          fun sleepAndDiv(v: Int) =
9              async(SupervisorJob()) {
10 // ......................
```

ДРУГОЙ ПОДХОД

```
// ....................

                try {
                    println("start $v")
                    delay(random.nextLong(5000))
                    println("after delay $v")
                    5 / v
                } catch (exc: ArithmeticException) {
                    null
                }
            }

// ....................
```

ДРУГОЙ ПОДХОД

```
// ......................

        launch {
            val result = data.map {
                sleepAndDiv(it)
            }.map {
                    it.await()
            }
            println("result: $result")
        }
    }
}
```

FLOW: МОТИВИРУЮЩИЙ ПРИМЕР

- Есть сервис работы с пользовательскими сообщениями
- Есть микросервис, отвечающий за историю сообщений
- Можно отправить REST-запрос и получить фрагмент истории полученных пользователем сообщений

FLOW: МОТИВИРУЮЩИЙ ПРИМЕР

- Параметры запроса - id пользователя и границы "окна"
- Как offset/limit или как временной диапазон
- Можно указать id отправителя

FLOW: МОТИВИРУЮЩИЙ ПРИМЕР

- Хотим для заданного пользователя найти отправителя, от которого пришло 3 сообщения, помеченных тегом jvm
- Или понять, что такого нет
- А также хотим что-то оптимизировать, если такие типовые запросы приходят часто
- Меняется только количество вхождений и ключевые слова

FLOW: МОТИВИРУЮЩИЙ ПРИМЕР

- Для базовой задачи надо идти по истории всех полученных сообщений
- С таким шагом, чтобы в память влезал ответ от микросервиса
- Группировать сообщения по id отправителя и агрегировать по count

FLOW: МОТИВИРУЮЩИЙ ПРИМЕР

- Как только кто-то достигли 3 останавливаемся и возвращаем результат
- Если всех перебрали и не нашли - возвращаем другой результат

FLOW: МОТИВИРУЮЩИЙ ПРИМЕР

- Оптимизация: заведем Counting Bloom Filter
- Для каждого пользователя на каждое ключевое слово
- (Считаем, что их ограниченное количество)
- Их надо посчитать и обновлять

FLOW: МОТИВИРУЮЩИЙ ПРИМЕР

- При запросе идти до точки обновления
- А после точки обновления - с учетом CBF
- Возможно придется пройтись и до конца, но реже
- Хотим использовать корутинную асинхронность для всего этого

FLOW: МОТИВИРУЮЩИЙ ПРИМЕР

- Вариант: suspend-метод для извлечения порций данных из микросервиса
- И состояние для каждого отдельного случая
- В нашем примере получим 3-4 структурно похожих класса
- Вместо этого предлагается механизм `Flow`

ПРИМЕР

```kotlin
val random = Random(42)

@OptIn(ExperimentalCoroutinesApi::class)
fun main() {
    runBlocking {
        launch {
            val flow = randomData(10, 200)
            println("before collect")
            flow.collect { println(it) }
        }
    }
}
// ......................
```

ПРИМЕР

```
// ...............

fun randomData(count: Int, delayMs: Long) = flow {
    println("start flow")
    repeat(count) {
        delay(delayMs)
        emit(random.nextInt(100))
    }
}
```

FLOW

- `flow` - конструктор
- Возвращает реализацию интерфейса `Flow` - с suspend-методом `collect`
- Параметр билдера - блок, исполняющийся для извлечения элемента
- `emit` - suspend-метод
- Можно увидеть, что в примере `emit` и `collect` работают в одной корутине

ПРИМЕР

```
1  val random = Random(42)
2
3  fun main() {
4      runBlocking {
5          launch {
6              val flow = randomData(10, 200)
7              println("before collect: " +
8                      "${currentCoroutineContext().job}")
9              flow.collect { println(it) }
10         }
11     }
12 }
13 // ..............................
```

ПРИМЕР

```
1 // .............................
2 fun randomData(count: Int, delayMs: Long) = flow {
3     println("start flow: ${currentCoroutineContext().job}")
4     repeat(count) {
5         delay(delayMs)
6         emit(random.nextInt(100))
7     }
8 }
```

СТАТУС FLOW

- `Flow` - не корутина
- Это реактивная структура
- Она умеет создавать `Continuation` над переданным блоком кода
- И передавать управление на него из разных корутин

СТАТУС FLOW

- Это похоже на Python-генератор
- Только более ленивый и более ограниченный в логике итерирования
- Python-генератор сразу создает объект
- А потом умеет по запросу идти до следующего `yield`

СТАТУС FLOW

- `Flow` оттягивает момент создания объекта до начала итерирования
- И то, что возвращает `flow-builder` - это нечто, умеющее порождать итератор
- Реально итератор создается при вызове `collect` или одного из производных методов
- В случае collect итератор работает, пока код, переданный в конструктор `Flow`, не завершится

МЕТОД SINGLE

- Есть метод `first`
- Возвращает то, что породил первый `emit`
- Но следующий вызов `first` снова создаст новый итератор
- Тут `Flow` похож на Java-stream с терминальными методами

ПРИМЕР

```
1  val random = Random(42)
2  fun main() {
3      runBlocking {
4          launch {
5              val flow = randomData(2, 200)
6              println("before: " +
7                      "${currentCoroutineContext().job}")
8              println(flow.first())
9              println("between: " +
10                     "${currentCoroutineContext().job}")
11             println(flow.first())
12             println("after: " +
13                     "${currentCoroutineContext().job}")
14 // ..........................
```

ПРИМЕР

```
// ..........................
        }
    }
}

fun randomData(count: Int, delayMs: Long) = flow {
    println("start flow: " +
            "${currentCoroutineContext().job}")
    while (true) {
        delay(delayMs)
        emit(random.nextInt(100))
    }
}
```

НЕТЕРМИНАЛЬНЫЕ МЕТОДЫ

- Стандартный набор: `map, filter, flatMap, take, dropWhile` и т.п.
- Реактивно-специфичные: `debounce, sample`
- `debounce` получает параметром временной интервал
- Если между двумя `emit`-ами времени прошло меньше интервала, новое значение вытестняет старое

ПРИМЕР

```kotlin
val random = Random(42)

fun main() {
    runBlocking {
        launch {
            val flow = randomData(2, 200)

            flow.map { it * 2 }
                .takeWhile { it < 180 }
                .collect(::println)
        }
    }
}
// ....................
```

ПРИМЕР

```
// ....................

fun randomData(count: Int, delayMs: Long) = flow {
    println("start flow: ${currentCoroutineContext().job}")
    while (true) {
        delay(delayMs)
        emit(random.nextInt(100))
    }
}
```

ПРИМЕР

```
fun main() {
    runBlocking(Dispatchers.Default) {
        flow {
            emit(1) ; delay(90)
            emit(2)
            delay(90)
            emit(3)
            delay(1010)
            emit(4)
            delay(1010)
            emit(5)
        }.debounce(1000).collect { println(it) }
    }
}
```

НЕТЕРМИНАЛЬНЫЕ МЕТОДЫ

- `sample` делит время на интервалы
- Если в интевал попало больше одного, то в поток попадет последнее
- `timeout` - бросает исключение, если ничего не породилось в течение заданного таймаута
- (Если работа завершена - то для timeout это ok)

ПРИМЕР

```
suspend fun main() {
    runBlocking {
        flow {
            repeat(10) {
                emit(it)
                delay(160)
        }
    }.sample(200).collect(::println)
}
```

ПРИМЕР

```
1  fun main() {
2      runBlocking(Dispatchers.Default) {
3          flow {
4              repeat(10) {
5                  emit(it)
6                  val c = it.toLong()
7                  delay(50 * c)
8                  emit(it + 100)
9              }
10         }.timeout(200.milliseconds).collect {
11             println(it)
12         }
13     }
14 }
```

НЕТЕРМИНАЛЬНЫЕ МЕТОДЫ

- Накопительные `fold`/`reduce`: `runningFold`/`runningReduce`
- Получаем поток промежуточных значений
- Может пригодиться в мотивирующем примере
- Сделать накопительную свертку с количеством по ключевым словам
- И над ней вызвать `firstOrNull`

ПРИМЕР

```
1  val random = Random(42)
2
3  fun main() {
4      runBlocking {
5          launch {
6              val flow = randomData(20, 200)
7              flow.withIndex()
8                  .runningFold(Pair(0, -1)) { acc, curr ->
9                      Pair(acc.first + curr.value,
10                         curr.index)
11                 }
12 // ....................
```

ПРИМЕР

```
// ....................
                .takeWhile { it.first < 200 }
                .collect(::println)
        }
    }
}

fun randomData(count: Int, delayMs: Long) = flow {
    println("flow: ${currentCoroutineContext().job}")
    repeat(count) {
        delay(delayMs)
        emit(random.nextInt(100))
    }
}
```

ТЕРМИНАЛЬНЫЕ МЕТОДЫ

- Базовый: `collect`
- Производный: `single` - первый элемент с проверкой, что нет других
- Производный `first` - первый, возможножно, что из многих
- `singleOrNull, firstOrNull`
- `toList, toSet` - могут зависнуть на бесконечном `Flow`

CALLBACK-СТИЛЬ

- `onEach` - нетерминальный метод, описывающий действия над элементом
- `onCompletion` - действия по завершении
- `onStart, onEmpty` - понятно
- В `on`-методах можно делать `emit`

ПРИМЕР

```
1  val random = Random(42)
2
3  fun main() {
4      runBlocking {
5          launch {
6              val flow = randomData(20, 200)
7
8              flow.onStart { println("start") }
9                  .withIndex()
10                 .runningFold(Pair(0, -1)) { acc, curr ->
11                     Pair(acc.first + curr.value,
12                          curr.index)
13                 }
14 // ..........................
```

ПРИМЕР

```
// ..........................
                .onEach { println("curr: " + it) }
                .takeWhile { it.first < 2000 }
                .onCompletion {
                    println("done takeWhile")
                }
                .onStart { println("start takeWhile") }
                .drop(3)
                .onStart { println("start drop") }
                .onCompletion { println("done drop") }
                .onEach { println("got-1: " + it) }
                .onEach { println("got-2: " + it) }
                .filter { it.first > 1000 }
// ..........................
```

ПРИМЕР

```
// ..........................
                .onEmpty {
                    println("No")
                    emit(Pair(0, 0))
                }
                .collect {
                    println(it)
                }
        }
    }
}
// ..........................
```

ПРИМЕР

```
1 // ..........................
2 fun randomData(count: Int, delayMs: Long) = flow {
3     println("start flow: ${currentCoroutineContext().job}")
4     repeat(count) {
5         delay(delayMs)
6         emit(random.nextInt(100))
7     }
8 }
```

ОДИН FLOW В РАЗНЫХ КОРУТИНАХ

- Никаких проблем - можно из разных корутин вызывать методы одного экземпляра одного `Flow`
- Когда дойдет до терминального - для каждой будет создан свой итератор
- И в каждом будет свое состояние
- А после терминального следующий терминальный породит новый итератор и новое состояние

ПРИМЕР

```
1  val random = Random(42)
2
3  fun main() {
4      runBlocking(Dispatchers.Default) {
5          launch {
6              val flow = randomData(5, 200)
7              repeat(100) {
8                  launch { println(flow.toList()) }
9              }
10         }
11     }
12 }
13 // ..............................
```

ПРИМЕР

```
// ................................
fun randomData(count: Int, delayMs: Long) = flow {
    println("start flow: " +
            "${currentCoroutineContext().job}")
    repeat(count) {
        println("before delay: " +
                "${currentCoroutineContext().job} $it")
        delay(random.nextLong(delayMs))
        println("after delay: " +
                "${currentCoroutineContext().job} $it")
        emit(random.nextInt(100))
    }
}
```

`Flow` В СВОЕЙ КОРУТИНЕ

- Может быть удобным запустить `Flow` в своей корутине
- Хотя бы для того, чтобы остановить итератор по `cancel`
- Например, по таймауту
- И по сложному критерию - который через цепочку нетерминальных сложно выразить

LAUNCHIN

- Есть отдельный терминальный метод `launchIn`
- Принимает параметром контекст
- В этом контексте запускает корутину
- В теле корутины запускается `collect`
- `launchIn` возвращает `Job`

ПРИМЕР

```
1  val random = Random(42)
2
3  fun main() {
4      runBlocking {
5          launch {
6              val flow = randomData(20, 200)
7
8              val job =flow.onStart { println("start") }
9                  .withIndex()
10                 .runningFold(Pair(0, -1)) { acc, curr ->
11                     Pair(acc.first + curr.value,
12                          curr.index)
13                 }
14 // ..............................
```

ПРИМЕР

```
// ..............................
                .onEach { println("curr: " + it) }
                .takeWhile { it.first < 2000 }
                .onCompletion {
                    println("done takeWhile")
                }
                .onStart { println("start takeWhile") }
                .drop(3)
                .onStart { println("start drop") }
                .onCompletion { println("done drop") }
                .onEach { println("got-1: " + it) }
                .onEach { println("got-2: " + it) }
                .filter { it.first > 1000 }
// ..............................
```

ПРИМЕР

```
// ..............................
                .onEmpty {
                    println("No")
                    emit(Pair(0, 0))
                }
                .launchIn(this)

            delay(1000)
            job.cancel()
        }
    }
}
// ..............................
```

ПРИМЕР

```
// .............................

fun randomData(count: Int, delayMs: Long) = flow {
    println("start flow: ${currentCoroutineContext().job}")
    repeat(count) {
        delay(delayMs)
        emit(random.nextInt(100))
    }
}
```

RUNWITHTIMEOUT

- Есть универсальный способ вызвать suspend-функцию с таймаутом
- Функция `runWithTimeout`
- Внутри себя создает корутину, в ней вызывает функцию
- По истечении таймаута вызывает `cancel`

ПРИМЕР

```
1 val random = Random(42)
2
3 fun main() {
4     runBlocking(Dispatchers.Default) {
5         launch {
6             val flow = randomData(5, 200)
7 // ..........................
```

ПРИМЕР

```
// ........................
            repeat(100) { jobIndex ->
                withTimeoutOrNull(200) {
                    flow.collect {
                        println("job #" + jobIndex)
                        println(it)
                    }
                }
            }
        }
    }
}
// ........................
```

ПРИМЕР

```
// ........................
fun randomData(count: Int, delayMs: Long) = flow {
    println("start flow: " +
            "${currentCoroutineContext().job}")
    repeat(count) {
        println("before delay: " +
                "${currentCoroutineContext().job} $it")
        delay(random.nextLong(delayMs))
        println("after delay: " +
                "${currentCoroutineContext().job} $it")
        emit(random.nextInt(100))
    }
}
```

ЕЩЕ ПРОБЛЕМА

- Указанные подходы решают две проблемы
  - Изолировать итерирование по `Flow` в корутину
  - Не уронить исключением весь контекст
- Но есть еще одна: чтобы пользовательский код `Flow` смог услышать призыв остановить работу

ЕЩЕ ПРОБЛЕМА

- В типовых `Flow` такой проблемы не будет
- Но бывают специальные `Flow` - получающиеся из обычных коллекций
- На них могут работать вычислительные задачи
- И они не услышат `cancel`
- Можно вызвать `cancelable()` над `Flow`
- И тогда проверка выполнится при заборе одного из значений

ПРИМЕР

```
fun createFlow(): Flow<Int> = flow {
    for (i in 1..10) {
        println("Emitting $i")
        emit(i)
    }
}
// ......................
```

ПРИМЕР

```
// .....................
fun main() = runBlocking(Dispatchers.Default) {
    val flow = createFlow()
    supervisorScope {
        launch {
            flow.onEach { value ->
                if (value == 3) cancel()
                println(value)
            }.launchIn(this)
        }
    }
    println(1111)
    delay(1000)
}
```

ИГНОРИРУЕМ CANCEL

```
1 fun createFlow(): Flow<Int> = (1..10).asFlow()
2
3 fun main() = runBlocking(Dispatchers.Default) {
4     val flow = createFlow()
5 // ..............
```

ИГНОРИРУЕМ CANCEL

```
// ...............
    supervisorScope {
        launch {
            flow.onEach { value ->
                if (value == 3) cancel()
                println(value)
            }.launchIn(this)
        }
    }
    println(1111)
    delay(1000)
}
```

НЕ ИГНОРИРУЕМ

```
1 fun createFlow(): Flow<Int> = (1..10).asFlow()
2
3 fun main() = runBlocking(Dispatchers.Default) {
4     val flow = createFlow().cancellable()
5 // ..................
```

НЕ ИГНОРИРУЕМ

```
// ................
    supervisorScope {
        launch {
            flow.onEach { value ->
                if (value == 3) cancel()
                println(value)
            }.launchIn(this)
        }
    }
    println(1111)
    delay(1000)
}
```