

АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 10

ПЛАН

- Async/await
- Flow
- Введение в Channel

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

```
1 // .....
2
3     println(result)
4     println(result.await())
5     println(result)
6     }
7 }
8 }
```

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

```
1 // .....
2
3 @OptIn(ExperimentalCoroutinesApi::class)
4 fun main() {
5     val data = listOf(1, 2)
6
7     runBlocking(Dispatchers.Default) {
8         launch {
9             val result = data.map { sleepAndTwice(it) }
10            println("result: $result")
11        }
12    }
13 }
```

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

```
1 // .....
2
3     launch {
4         val result = data.map { sleepAndTwice(it) }
5                               .map { it.await() }
6         println("result: $result")
7     }
8 }
9 }
```

AWAITALL

- Есть `awaitAll` для списка `Deferred`
- 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

```
1 // .....
2     launch {
3         val result = data.map {
4             sleepAndDiv(it)
5         }.map {
6             try {
7                 it.await()
8             } catch (exc: ArithmeticException) {
9                 // .....
```

ПРИМЕР ОБРАБОТКИ ИСКЛЮЧЕНИЯ

```
1 // .....
2         println(
3             it.getCompletionExceptionOrNull()
4         )
5         null
6     }
7 }
8     println("result: $result")
9 }
10 }
11 }
```

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

```
1 // .....
2
3     try {
4         println("start $v")
5         delay(random.nextLong(5000))
6         println("after delay $v")
7         5 / v
8     } catch (exc: ArithmeticException) {
9         null
10    }
11 }
12
13 // .....
```

ДРУГОЙ ПОДХОД

```
1 // .....
2
3     launch {
4         val result = data.map {
5             sleepAndDiv(it)
6         }.map {
7             it.await()
8         }
9         println("result: $result")
10    }
11 }
12 }
```


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 СВФ
- 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

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

ПРИМЕР

```
1 // .....
2
3 fun randomData(count: Int, delayMs: Long) = flow {
4     println("start flow")
5     repeat(count) {
6         delay(delayMs)
7         emit(random.nextInt(100))
8     }
9 }
```

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

```
1 // .....
2     }
3     }
4 }
5
6 fun randomData(count: Int, delayMs: Long) = flow {
7     println("start flow: " +
8         "${currentCoroutineContext().job}")
9     while (true) {
10         delay(delayMs)
11         emit(random.nextInt(100))
12     }
13 }
```

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

```
1 val random = Random(42)
2
3 fun main() {
4     runBlocking {
5         launch {
6             val flow = randomData(2, 200)
7
8             flow.map { it * 2 }
9                 .takeWhile { it < 180 }
10                .collect(::println)
11        }
12    }
13 }
14 // .....
```

ПРИМЕР

```
1 // .....
2
3 fun randomData(count: Int, delayMs: Long) = flow {
4     println("start flow: ${currentCoroutineContext().job}")
5     while (true) {
6         delay(delayMs)
7         emit(random.nextInt(100))
8     }
9 }
```

ПРИМЕР

```
1 fun main() {  
2     runBlocking(Dispatchers.Default) {  
3         flow {  
4             emit(1) ; delay(90)  
5             emit(2)  
6             delay(90)  
7             emit(3)  
8             delay(1010)  
9             emit(4)  
10            delay(1010)  
11            emit(5)  
12        }.debounce(1000).collect { println(it) }  
13    }  
14 }
```

НЕТЕРМИНАЛЬНЫЕ МЕТОДЫ

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

ПРИМЕР

```
1 suspend fun main() {  
2     runBlocking {  
3         flow {  
4             repeat(10) {  
5                 emit(it)  
6                 delay(160)  
7             }  
8         }.sample(200).collect(::println)  
9     }
```

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

```
1 // .....
2         .takeWhile { it.first < 200 }
3         .collect(::println)
4     }
5 }
6 }
7
8 fun randomData(count: Int, delayMs: Long) = flow {
9     println("flow: ${currentCoroutineContext().job}")
10    repeat(count) {
11        delay(delayMs)
12        emit(random.nextInt(100))
13    }
14 }
```

ТЕРМИНАЛЬНЫЕ МЕТОДЫ

- Базовый: `collect`
- Производный: `single` - первый элемент с проверкой, что нет других
- Производный `first` - первый, возможно, 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

```
1 // .....
2         .onEach { println("curr: " + it) }
3         .takeWhile { it.first < 2000 }
4         .onCompletion {
5             println("done takeWhile")
6         }
7         .onStart { println("start takeWhile") }
8         .drop(3)
9         .onStart { println("start drop") }
10        .onCompletion { println("done drop") }
11        .onEach { println("got-1: " + it) }
12        .onEach { println("got-2: " + it) }
13        .filter { it.first > 1000 }
14 // .....
```

ПРИМЕР

```
1 // .....
2         .onEmpty {
3             println("No")
4             emit(Pair(0, 0))
5         }
6         .collect {
7             println(it)
8         }
9     }
10 }
11 }
12 // .....
```

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

```
1 // .....
2 fun randomData(count: Int, delayMs: Long) = flow {
3     println("start flow: " +
4         "${currentCoroutineContext().job}")
5     repeat(count) {
6         println("before delay: " +
7             "${currentCoroutineContext().job} $it")
8         delay(random.nextLong(delayMs))
9         println("after delay: " +
10             "${currentCoroutineContext().job} $it")
11         emit(random.nextInt(100))
12     }
13 }
```

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

```
1 // .....
2         .onEach { println("curr: " + it) }
3         .takeWhile { it.first < 2000 }
4         .onCompletion {
5             println("done takeWhile")
6         }
7         .onStart { println("start takeWhile") }
8         .drop(3)
9         .onStart { println("start drop") }
10        .onCompletion { println("done drop") }
11        .onEach { println("got-1: " + it) }
12        .onEach { println("got-2: " + it) }
13        .filter { it.first > 1000 }
14 // .....
```

ПРИМЕР

```
1 // .....
2         .onEmpty {
3             println("No")
4             emit(Pair(0, 0))
5         }
6         .launchIn(this)
7
8         delay(1000)
9         job.cancel()
10    }
11 }
12 }
13 // .....
```

ПРИМЕР

```
1 // .....
2
3 fun randomData(count: Int, delayMs: Long) = flow {
4     println("start flow: ${currentCoroutineContext().job}")
5     repeat(count) {
6         delay(delayMs)
7         emit(random.nextInt(100))
8     }
9 }
```

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

```
1 // .....
2         repeat(100) { jobIndex ->
3             withTimeoutOrNull(200) {
4                 flow.collect {
5                     println("job #" + jobIndex)
6                     println(it)
7                 }
8             }
9         }
10    }
11 }
12 }
13 // .....
```

ПРИМЕР

```
1 // .....
2 fun randomData(count: Int, delayMs: Long) = flow {
3     println("start flow: " +
4         "${currentCoroutineContext().job}")
5     repeat(count) {
6         println("before delay: " +
7             "${currentCoroutineContext().job} $it")
8         delay(random.nextLong(delayMs))
9         println("after delay: " +
10             "${currentCoroutineContext().job} $it")
11         emit(random.nextInt(100))
12     }
13 }
```


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

```
1 fun createFlow(): Flow<Int> = flow {  
2     for (i in 1..10) {  
3         println("Emitting $i")  
4         emit(i)  
5     }  
6 }  
7 // .....
```

ПРИМЕР

```
1 // .....
2 fun main() = runBlocking(Dispatchers.Default) {
3     val flow = createFlow()
4     supervisorScope {
5         launch {
6             flow.onEach { value ->
7                 if (value == 3) cancel()
8                 println(value)
9             }.launchIn(this)
10        }
11    }
12    println(1111)
13    delay(1000)
14 }
```

ИГНОРИРУЕМ CANCEL

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

ИГНОРИРУЕМ CANCEL

```
1 // .....
2     supervisorScope {
3         launch {
4             flow.onEach { value ->
5                 if (value == 3) cancel()
6                 println(value)
7             }.launchIn(this)
8         }
9     }
10    println(1111)
11    delay(1000)
12 }
```

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

```
1 // .....
2     supervisorScope {
3         launch {
4             flow.onEach { value ->
5                 if (value == 3) cancel()
6                 println(value)
7             }.launchIn(this)
8         }
9     }
10    println(1111)
11    delay(1000)
12 }
```


CHANNEL

- В первом приближении - аналог блокирующей очереди
- Нет класса `Channel`
- Есть интерфейсы `SendChannel<T>` и `ReceiveChannel<T>`
- Есть наследующий обоих интерфейсов `Channel<T>`
- И есть `factory`-метод `Channel<T>` создающий нужную реализацию

ПРИМЕР

```
1 val channel = Channel<Int>()
2
3 fun main() = runBlocking(Dispatchers.Default) {
4     launch {
5         for (x in 1..5) {
6             println("before send")
7             channel.send(x * x)
8             println("after send")
9         }
10    }
11    // .....
```

ПРИМЕР

```
1 // .....
2     repeat(5) {
3         delay(1000)
4         println(channel.receive())
5     }
6 }
```

ОПИСАНИЕ

- По умолчанию используется режим рандеву
- При посылке ожидаем того, что будет читать
- Передаем из рук в руки
- Возможны варианты: конечный буфер, бесконечный буфер
- Поведение при записи в переполненный: suspend, перезапись старых, исключение
- Поведение при чтении из пустого : suspend, исключение

