

Coroutines 2 asynchrónnosť

Peter Borovanský
KAI, I-18

MS-Teams: [2sf3ph4](#), [List](#), [github](#)

borovan 'at' ii.fmph.uniba.sk

AsyncTask
Retrofit
RoomDB

Coroutines

- **channel**
- **flow**
- **Shared state**
 - **Atomická premenná**
 - **Prepínanie kontextov**
 - **mutex**

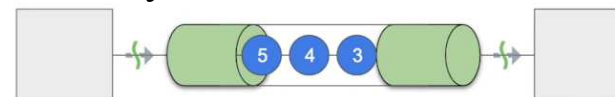
Flow

Flow je leniná/cold pipe-line:

```
val numbers : Flow<Int> = flow {  
    listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).forEach{  
        emit(it) ← yield  
        delay(it*100L)  
    }  
} // flow zaniká
```

```
runBlocking {  
    numbers.collect {  
        println(it)  
    }  
}
```

```
runBlocking {  
    numbers  
        .buffer  
        .collect {  
            println(it)  
        }  
}
```



```
listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).asFlow()
```

```
flowOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```



Flow

Flow je niečo ako generátor v Pythone, lazy v Haskell
Flow je typovaný, teda `Flow<T>`, resp. `Flow<Int>`

```
fun main() {  
    val numbers : Flow<Int> = flow {  
        listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).forEach {  
            emit(it)  
            delay(it*100L)  
        }  
    } // flow zanika  
    runBlocking {  
        numbers.collect {  
            println(it)  
        }  
    }  
}
```

←

← yield

←

```
18:57:57.132 1  
18:57:57.244 2  
18:57:57.444 3  
18:57:57.756 4  
18:57:58.165 5  
18:57:58.672 6  
18:57:59.284 7  
18:57:59.994 8  
18:58:00.796 9  
18:58:01.704 10
```



Flow

Flow je niečo ako generátor v Pythone, lazy v Haskell
Flow je typovaný, teda `Flow<T>`, resp. `Flow<Int>`

```
fun main() {  
    val numbers : Flow<Int> = flow {  
        listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).forEach {  
            emit(it)                ← yield  
            delay(it*100L)  
        }  
    } // flow zanika  
    runBlocking {  
        numbers.collect { ←  
            println(it)  
        }  
    }  
}
```

```
18:57:57.132 1  
18:57:57.244 2  
18:57:57.444 3  
18:57:57.756 4  
18:57:58.165 5  
18:57:58.672 6  
18:57:59.284 7  
18:57:59.994 8  
18:58:00.796 9  
18:58:01.704 10
```



Flow

konštruktory

`.emit` sám nič neurobí, kým si niekto nepýta hodnoty z `Flow`
`.collect`

```
fun main() {  
    runBlocking {  
        postupnost().collect {  
            println("$it")  
        }  
    }  
}  
fun postupnost()  
    = flowOf("Jeden", "Dva", "Tri", "Styri")  
    = listOf(1, 2, 3).asFlow()  
    = flow {  
        for (i in 1..10)  
            emit(i)  
    }
```



Flow

take, takeWhile, map, filter, ...

Flow môže byť konečný alebo nekonečný (potenciálne)

```
Fun main() {  
    runBlocking {  
        println("pred")  
        aritmeticka(1,3).collect {  
            println(it)  
        }  
        println("po")  
    }  
}
```

```
aritmeticka(1,3)  
    .take(5)  
    .takeWhile {it < 10}  
    .collect {  
        println(it)  
    }
```

```
fun aritmeticka(a :Int, delta :Int): Flow<Int> = flow {  
    (0..9).forEach {  
        delay(it * 100L)  
        emit(a + it*delta)  
    }  
}
```

```
var x = a  
while (true) {  
    delay(x * 100L)  
    emit(x)  
    x += delta  
}
```



Flow

withTimeoutOrNull

```
fun main() {  
    runBlocking {  
        val flow = geometricka(1,2)  
        println("pred")  
        withTimeoutOrNull(1000L) {  
            flow.collect { println(it) }  
        }  
        println("po")  
    }  
}  
fun geometricka(a :Int, q :Int) = flow {  
    var x = 1  
    (0..9).forEach {  
        delay(400L)  
        emit(a*x)  
        x *= q  
    }  
}
```





Flow

`.onEach, .map, .filter, .reduce, .take, .zip, .combine, .flowOn`

```
suspend fun combine() {  
    val numbers = (1..5).asFlow().onEach { delay(300L) }  
    val values = flowOf("One", "Two", "Three", "Four", "Five")  
        .onEach { delay(400L) }  
    numbers.combine(values) { a, b ->  
        "$a - $b"  
    }.collect { println(it) }  
}  
  
suspend fun zip() {  
    val english = flowOf("One", "Two", "Three")  
    val french = flowOf("Un", "Deux", "Trois")  
    english.zip(french) { a, b ->  
        "'$a' in French is '$b'"  
    }.collect {  
        println(it)  
    }  
}
```




Flow

.buffer

```
fun main() {  
    runBlocking {  
        val time = measureTimeMillis {  
            mocniny()  
                .buffer() ←  
                .collect {  
                    delay(300L)  
                    println(it)  
                }  
            }  
        println("Collected in $time ms")  
    }  
}  
  
fun mocniny() = flow {  
    (0..10).forEach {  
        delay(100L)  
        emit(1 shl it)  
    }  
}
```



Flow

.catch

```
suspend fun onCompletion() {  
    (1..3).asFlow()  
        .onEach { check(it != 2) }  
        → .onCompletion { e ->  
            if(e != null)  
                println("Flow completed with exception $e")  
            else  
                println("Flow completed successfully")  
        }  
        → .catch { e -> println("Caught exception $e") }  
        .collect { println(it) }  
}
```

```
for(i in 1..5)
/,      println(channel.receive())
```

Kanály

už objavili v jazyku Go

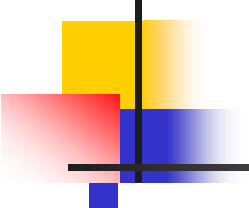
```
val channel = Channel<Int>()
runBlocking {
    launch {
        for (x in 1..10)
            channel.send(x * x)
        channel.close()
    }
}
```

```
for(i in 1..5) println(channel.receive())
```

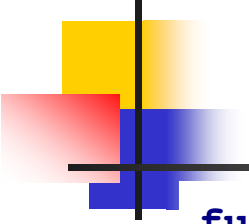
```
for (i in channel)
    println(i)
```

```
channel.consumeEach { println(it) }
}
```

```
1
4
9
16
25
36
49
64
81
100
```



```
fun main() {  
    runBlocking {  
        val numbers = naturals(1)  
        val squares = square(numbers)  
        while (true) {  
            val rec = squares.receive()  
            if (rec > 1000000) break  
            println(rec)  
        }  
        coroutineContext.cancelChildren()  
    }  
}  
  
fun CoroutineScope.naturals(start : Int) = produce {  
    var n = start  
    while (true) send(n++)  
}  
  
fun CoroutineScope.square(numbers: ReceiveChannel<Int>) = produce {  
    for (x in numbers) send(x * x)  
}
```

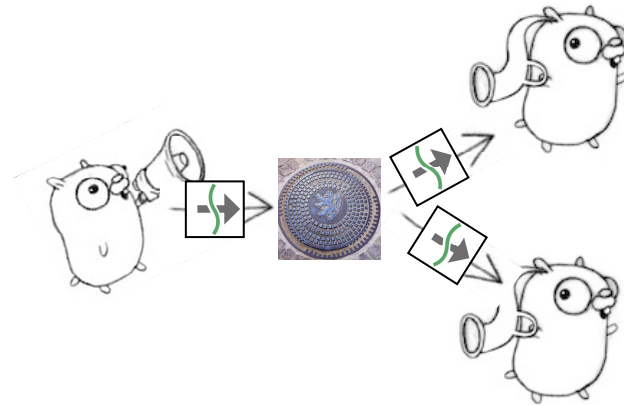


```
fun main() {  
    runBlocking {  
        val channel = Channel<String>()  
        launch { sendString(channel, 200L, "cor1") }  
        launch { sendString(channel, 500L, "cor2") }  
        launch { sendString(channel, 300L, "cor3") }  
        repeat(70) {  
            println(channel.receive())  
        }  
        coroutineContext.cancelChildren()  
    }  
}  
  
suspend fun sendString(channel: SendChannel<String>,  
                        time: Long, name: String) {  
    for (i in 1..30) {  
        delay(time)  
        channel.send("$name:$i")  
    }  
}
```

Kanály

už objavili v jazyku Go

```
val channel = Channel<Int>()
GlobalScope.launch {
    for (x in channel) {
        println("a:$x")
    }
}
GlobalScope.launch {
    for (x in channel) {
        println("b:$x")
    }
}
runBlocking{
    listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) .forEach{
        println(" :$it")
        channel.send(it)
    }
    delay(1000)
}
```



```
:1
:2
:3
a:2
b:1
a:3
:4
:5
:6
b:4
b:6
:7
b:7
:8
b:8
:9
b:9
:10
b:10
a:5
```



Primes

```
fun CoroutineScope.generator(i : Int) = produce<Int> {  
    var n = i  
    while (true) send(n++)  
}  
fun CoroutineScope.sieve(p : Int, f : ReceiveChannel<Int>) = produce<Int> {  
    for (x in f) if (x % p > 0) send(x)  
}  
fun CoroutineScope.eratosten() = produce<Int> {  
    var ch = generator(2)  
    while(true) {  
        val prvocislo = ch.receive()  
        send(prvocislo)  
        ch = sieve(prvocislo, ch)  
    }  
}  
fun main() {  
    runBlocking {  
        //for(p in eratosten()) println(p)  
        val ch = eratosten()  
        repeat(100) {  
            println(ch.receive())  
        }  
    }  
}
```



Share memory

atomic variable

```
runBlocking {  
    var state = 0  
    //var state = AtomicInteger(0)  
    withContext(Dispatchers.Default) {  
        val time = measureTimeMillis {  
            coroutineScope {  
                (1..1000).forEach {  
                    launch {  
                        (1..1000).forEach {  
                            state++  
                            //state.getAndIncrement()  
                        }  
                    }  
                }  
            }  
        }  
        println("elapsed time: $time ms")  
    }  
    println("State = $state")  
}
```

“
Do not communicate by
sharing memory; instead,
share memory by communicating.

— Effective Go

”



Same context

```
runBlocking {  
    var state = 0  
    val stateContext = newSingleThreadContext("stateContext")  
    val time = measureTimeMillis {  
        coroutineScope {  
            (1..1000).forEach {  
                launch {  
                    (1..1000).forEach { // switch context  
                        withContext(stateContext) { ←  
                            state++  
                        }  
                    }  
                }  
            }  
        }  
    }  
    println("elapsed time: $time ms")  
    println("State = $state")  
}
```



Mutex

```
runBlocking {  
    var state = 0  
    val mutex = Mutex() ←  
    withContext(Dispatchers.Default) {  
        val time = measureTimeMillis {  
            coroutineScope {  
                (1..1000).forEach {  
                    launch {  
                        (1..1000).forEach {  
                            mutex.withLock { ←  
                                state++  
                            }  
                        }  
                    }  
                }  
            }  
        }  
        println("elapsed time: $time ms")  
    }  
    println("State = $state")  
}
```