Asynchronous Flow

코루틴 → 계속해서 emit value 하게 한다. 다른 코루틴과 연결되어 정보를 받을 수 도 있다.

Observer pattern 과 비슷하다.

매우 유용한데 ...

1. async~ flow 가 무엇인지
2. 이를 어떻게 적용할 수 있는지
3. 버퍼링이 무엇인가
4. flow 들을 섞는 것 ?
5. 예외 처리하기

Acynchronous Flow

- 비동기적으로 연산되는 값들의 스트림이다.
- Flow는 value 를 emit 하고 코루틴을 이를 받아서 연산한다.

Flow 만들기

```
* flow{ //builder 이다.
  ...
}

* emit(value) //transmits a value

* collect{ ... } //receive the values
```

```
fun sendPrimes(): Flow<Int> = flow {
      val primesList = listOf(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
      primesList.forEach {
            delay(it * 100L)
            emit(it)
      }
}
```

- 소수의 리스트, delay 를 한 후에 값들을 emit 한다.

```
runBlocking {
      println("Receiving prime numbers")
      sendPrimes().collect {
            println("Received prime number $it")
      }
      println("Finished receiving numbers")
}
```

- 코루틴 안에서 값들을 collect 한 후에 연산을 진행한다.

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.runBlocking

fun main() {

    runBlocking {
        println("Receiving Prime Numbers")
        sendPrimes().collect{
            //값들을 받는다.
            println("Received Prime Number $it")
        }
        println("Finished")
    }
}

fun sendPrimes() : Flow<Int> = flow{
    val primesList = listOf(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
    primesList.forEach{
        delay(it * 100L)
        emit(it)
    }
}
```

Creating Flows

- 각각의 값들을 독립적으로 emit 하면서 flow 를 만들 수 있다.

```
fun sendNumbers() = flow {
      for (i in 1..10)
            emit(i)
}
```

```
import kotlinx.coroutines.flow.asFlow
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.flowOf
import kotlinx.coroutines.runBlocking

fun main() {

    runBlocking {
        sendNumbers().collect{
            println("Received $it")
        }
    }
}

//Flow Builder 를 통해서 flow 를 생성한다.
fun sendNumbers() = flow {
    for (i in 1..10){
        emit(i)
    }
}

//리스트를 직접적으로 Flow 로 Convert 한다.
fun sendNumbers2()
    = listOf(1, 2, 3).asFlow()

//vararg 를 통해서 만들 수 도 있다.
fun sendNumbers3() = flowOf(
    1, 2, 3, 4, 5
)
```

Flow Properties

Flows 는 Cold 하다. 코드는 collect 함수가 실행되기 전까지 (consumer 가 있기 전까지) 실행되지 않는다.

- **collect 가 호출되기 전까지 숫자 1, 2, 3 은 emit 되지 않는다.**

```
fun sendNumbers(): Flow<Int> = flow {
      val list = listOf(1, 2, 3)
      list.forEach { emit(it) }
}

runBlocking {
      val numbersFlow = sendNumbers()
      println("Flow hasn't started yet")
      println("Starting flow now:")
      numbersFlow.collect { println(it) }
}
```

Flow 는 자신이 자신을 cancel 할 수 없다.

- 값을 받는 코루틴이 cancel 해야 한다.
- 코루틴이 1초안에 cancel 된다 → 이에 따라 flow 도 cancel 된다.

```
fun sendNumbers(): Flow<Int> = flow {
      val list = listOf(1, 2, 3)
      list.forEach {
            delay(400)
            emit(it)
      }
}

runBlocking {
      val numbersFlow = sendNumbers()
      println("Flow hasn't started yet")
      println("Starting flow now:")
      withTimeoutOrNull(1000) {
            numbersFlow.collect { println(it) }
      }
}
```

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.runBlocking
import kotlinx.coroutines.withTimeoutOrNull

fun main() {

    runBlocking {
        val numbersFlow = sendNumbers4()
        //cold 하므로 변수에 할당하는 순간에는 값을 emit 하지 않는다.
        println("Flow has not started yet")
        println("starting flow now")
        numbersFlow.collect{
            println(it)
        }
        println("flows are cold!")
    }

    runBlocking {
        val numbersFlow2 = sendNumbers5()
        withTimeoutOrNull(1000L){
            numbersFlow2.collect{
                println(it) //세번째 숫자는 출력되지 않는다.
            }
        }
    }
}

fun sendNumbers4() = flow{
    listOf(1, 2, 3).forEach{
        emit(it)
    }
}

fun sendNumbers5() = flow{
    listOf(1, 2, 3).forEach{
        delay(400L)
        emit(it)
    }
}
```

Operators

Flow 를 어떻게 변형할 수 있는가?

- input flow → output flow 를 반환한다.
- operators are cold : flow 에 적용한다고 해서 flow 가 active 한 것이 아니라, 연산자가 레지스터 되었다고 봐야한다.

- 반환되는 플로우는 동기적이다. return 바로 하는 것이고, 또 다른 코루틴이 아니다.

Map Operator : Map a flow to another flow

```
(1..10).asFlow()
    .map {
        delay(500)
        "mapping $it"
    }
    .collect {
        println(it)
    }
```

→ flow 에다가 map 을 적용하고, collect 를 호출한다.

Filter Operator

- 짝수일 경우에 print 하는것

```
(1..10).asFlow()
    .filter {
        it % 2 == 0
    }
    .collect {
        println(it)
    }
```

Transform operator

- 어느 시점에나 값을 emit 할 수 있다.

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

fun main() {
```

```
    runBlocking {
        mapOperator() //suspend function
        filterOperator()
        transformOperator()
    }
}

suspend fun mapOperator(){
    (1..10).asFlow()
        .map{
            delay(500L)
            "mapping $it"
        }.collect{
            println(it)
        }
}

suspend fun filterOperator(){
    (1..10).asFlow()
        .filter{
            it % 2 == 0
        }.collect{
            println(it)
        }
}

suspend fun transformOperator(){
    (1..10).asFlow()
        .transform{
            emit("emitting stirng value $it")
            emit(it)
        }.collect{
            println(it)
        }
}
```

take operator

- use only a number of values, disregard the rest.

```
suspend fun takeOperator(){
    (1..10).asFlow()
        .take(2) //두개의값만 출력된다.
        .collect{
            println(it)
        }
}
```

terminal flow operator

- flow 를 collection 으로 변형할 수 있게 해준다.
  - **collect (계속 썼던것)**
  - toList : 리스트로 변형
  - toSet : 셋으로 변형, 중복된 값은 포함되지 않는다.
  - reduce

- reduce : 누적하는 기능이 있다.

```
val size = 10
val factorial = (1..size).asFlow()
      .reduce { accumulator, value ->
            accumulator * value
      }
println("Factorial of $size is $factorial")
```

저장된 값이 accumulator → 들어오는 값과 곱해져서 결과가 반환된다.

```
suspend fun reduceOperator(){
    val size = 10
    val factorial = (1..size).asFlow()
        .reduce{accumulator, value ->
            accumulator * value
        } //terminal operator 이므로 collect 의 기능을 포함한다.
    println("Factorial of $size is $factorial")
}
```

flowOn operator

- change the context on which the flow is emitted
- Main 에서 IO 로 디스패처를 바꾸고 싶은 경우 사용된다.

```
suspend fun flowOnOperator(){
    (1..10).asFlow()
        .flowOn(Dispatchers.IO)
        .collect{
            println(it)
```

```
        }
}
```

Buffering

Flow 에서 값을 내는 데 시간이 많이 걸리면, 나중에 프로세싱 될 수 있는 값들을 누적한다. (버퍼링함)

만약 값들을 버퍼링 할 수 없다면 타임 디레이가 일어날 것이다.

- 매 100밀리세컨마다 값을 내는 flow 가 있는데, 값을 프로세싱 하는데 300밀리세컨이 걸린다면, flow가 빨리 내보내도, 프로세싱에 디레이가 생긴다.
- 이 때 버퍼링이 적용된다.

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.buffer
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.runBlocking
import kotlin.system.measureTimeMillis

fun main() {
    runBlocking {
        val time = measureTimeMillis {
            generate()
                .buffer() //버퍼를 넣으면 시간이 줄어든다.
                .collect{
                    delay(300L)
                    println(it)
                }
        }
        println("Collected in $time")
    }
}

fun generate() = flow{
    for(i in 1..3){
        delay(100L)
        emit(i)
    }
}
```

Composing Flows

한 개 이상의 flow 가 들어와서, 그 둘을 섞을 경우가 있을 것이다.

다른 flow 들을 어떻게 compose 할 수 있는지 ?

Zip : 두 개의 flow 에서 상응하는 값들을 매칭한다.

- 영어와, 프랑스어가 같이 들어오는 경우.

```
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) }
```

Combine : 가장 최근의 플로우의 값을 다른 플로우의 가장 최근의 값과 결합시킨다.

- latest value in both of the flows.

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

fun main() {
    runBlocking {
        zip()
        combine()
    }
}

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 is matched to $b"
    }.collect{
        println(it)
    }
}
```

```
suspend fun zip(){

    val english = flowOf("One", "Two", "Three")
    val french = flowOf("Un", "Deux", "Troix")

    english.zip(french){
        a, b -> "'$a' in French is '$b'"
    }.collect{
        println(it)
    }
}
```

Exception Handling

- 예외가 일어날 때 Flow 와 어떻게 처리해야 하는가?
- 세 가지 방법

1. **try, catch**

   Flow ... .collect 하는 부분을 try 에 넣고 예외를 catch 블럭에서 처리한다.

2. **코틀린의 catch 연산자를 사용한다.**

   catch 이후의 코드는 실행되지 않는다.

3. **onCompletion()**

   finally 블럭과 유사함.

```
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

fun main() {
    runBlocking {
//        tryCatch()
        onCompletion()
    }
}

suspend fun onCompletion(){
//finally 와 비슷한 역할을 하는 onCompletion 함수를 사용한다.
    (1..3).asFlow()
```

```
        //.onEach{ check(it != 2)} //블럭을 없애면 예외처리가 된다.
        .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)
        }
}

suspend fun catch(){
    //catch operator 를 사용해서 처리하기
    (1..3).asFlow()
        .onEach{ check(it != 2)}
        .catch{
            e -> println("Caught Exception $e")
        }.collect{
            println(it)
        }
}

suspend fun tryCatch(){
    try {
        (1..3).asFlow()
            .onEach{check(it != 2)}
            .collect{ println(it)}
    }catch (e : Exception){
        println("Caught exception ${e.localizedMessage}")
    }
}
```