Flow操作符实验及理解

前置知识

参考文档： 异步流 - Kotlin 语言中文站

操作符类型

Intermediate operators 中间操作符，如 map, filter, take, zip

> 将上游flow转换成下游flow（**返回新的flow对象**）
>
> 操作符传入的transform function是suspend方法，因此**方法内部可以挂起**
>
> 流是冷的，当没有消费者时不会产生数据也不会执行中间操作符的方法
>
> 顺序消费，默认在同一个协程里挂起等待，如下示例代码

Kotlin

```
1
2  private suspend fun fetchUserName(userId: Int) : String {
3      delay(2000)
4      return "user$userId"
5  }
6
7  private suspend fun filterInvalidName(name: String) : Boolean {
8      delay(1000)
9      return name != "user1"
10 }
11
12 override fun onCreate() {
13     val testFlow = flowOf(1,2,3).map {
14         fetchUserName(it)
15     }.filter {
16         filterInvalidName(it)
17     }
18     lifecycleScope.launchWhenStarted {
19         testFlow.onStart {
20             Log.d("LOLM", "start")
21         }.collect {
22             Log.d("LOLM", "collect $it")
23         }
24     }
25 }
```

CSS

```
输出：
2021-10-08 21:29:08.415 D/LOLM: start
2021-10-08 21:29:14.420 D/LOLM: collect user2
2021-10-08 21:29:17.422 D/LOLM: collect user3
```

Terminal operators 末端操作符，如 collect, single, reduce, toList

> 相当于flow的消费者，会触发执行所有的操作
>
> 末端操作符的正常/异常完成取决于所有操作符的执行成功与否
>
> 不会再创建/返回flow对象
>
> 殊途同归,这些方法内部都调用collect方法

流是冷的

流的构造器及中间操作符中的方法只有在**末端操作符执行时**才会执行。

当然，**当流被collect时才会执行**的说法也是正确的，因为例如toList, toSet、single等末端操作符的实现里也都调用了collect方法。

下面这个例子生动的说明flow is cold

Kotlin

```kotlin
fun fibonacci(): Flow<BigInteger> = flow {
    var x = BigInteger.ZERO
    var y = BigInteger.ONE
    while (true) {
        emit(x)
        x = y.also {
            y += x
        }
    }
}

fibonacci().take(100).collect { println(it) }
print(fibonacci().take(100).toList().joinToString(","))
```

创建Flow

flow { ... }

在挂起函数闭包中发射数据

JavaScript

```
flow {
      delay(1000)
      emit(1) // Ok
      withContext(Dispatcher.IO) {
          emit(2) // Will fail with IllegalStateException
      }
}
```

不要在其中切换协程上下文，否则会抛异常

建议不要在其中捕获异常，上游flow中的异常用catch操作符捕获即可

flowOf(...)

将固定的可变长参数一一发射

Kotlin

```
public fun <T> flowOf(vararg elements: T): Flow<T> = flow {
    for (element in elements) {
        emit(element)
    }
}
```

asFlow()

将集合、序列转换成Flow，支持的类型包括： Iterable, Iterator, Sequence,Array, Range 以及函数/挂起函数类型

Kotlin

```kotlin
suspend fun test() : String {
    delay(3000)
    return "hehe"
}

fun main() {
    listOf(1,2,3).asFlow()
    setOf(1,2,3).asFlow()
    (1..3).asFlow()
    (::test).asFlow()
}
```

(挂起)函数类型比较局限，只针对没有入参的函数，应该也不常使用。

channelFlow{ ... }

flow默认生产者消费者之间通信是同步非阻塞的，即生产者和消费者是在同一个协程/线程。
channelFlow支持生产消费者之间**异步非阻塞**地通信，内部是通过Channel来实现。

在构造时，区别于普通flow，发射数据使用send

Kotlin

```kotlin
@ExperimentalCoroutinesApi
fun genChannelFlow() = channelFlow {
    (1..3).forEach {
        delay(1000)
        Log.d("LOLM", "Flow emit $it")
        send(it)
    }
}

override fun onCreate() {
.   val channelFlow = genChannelFlow()
    lifecycleScope.launchWhenStarted {
        Log.d("LOLM", "launchWhenStarted start")
        delay(2000)
        channelFlow.onStart {
            Log.d("LOLM", "collect Flow onStart ")
        }.collect {
            delay(2000)
            Log.d("LOLM", "collect Flow collect $it")
        }
    }
}
```

Output:

Apache

```
17:26:56 D/LOLM: launchWhenStarted start
17:26:58 D/LOLM: collect Flow onStart
17:26:59 D/LOLM: Flow emit 1
17:27:00 D/LOLM: Flow emit 2
17:27:01 D/LOLM: collect Flow collect 1
17:27:01 D/LOLM: Flow emit 3
17:27:03 D/LOLM: collect Flow collect 2
17:27:05 D/LOLM: collect Flow collect 3
```

上面示例能说明：

1. channelFlow依然是冷流，只有在被订阅时才会执行构造器代码
2. channelFlow的发送方不受接收方影响，两者可以并行执行
3. 如果仅仅为了实现发送接收方异步，指定生产者工作在另一个协程是否也可以

PHP

```
fun genFlowSwitchContext() = flow {
    (1..3).forEach {
        delay(1000)
        Log.d("LOLM", "Flow emit $it")
        emit(it)
    }
}.flowOn(Dispatchers.IO)

override fun onCreate() {
    lifecycleScope.launchWhenStarted {
        Log.d("LOLM", "launchWhenStarted start")
        delay(2000)
        genChannelFlow().onStart {
            Log.d("LOLM", "collect Flow onStart ")
        }.collect {
            delay(2000)
            Log.d("LOLM", "collect Flow collect $it")
        }
    }
}
```

输出和使用channelFlow一致，当然这里指定flowOn(Dispatchers.Main)也是一样，只要处于不同协程都可以实现异步，所以我对于channelFlow使用的兴趣不大

callbackFlow{ ... }

前面介绍flow { ... }时说过，不能在构造器闭包函数中切换协程，否则会抛异常。但实际开发中，总有些已经封装好的指定了线程/协程上下文并以设置回调的形式来调用的方法，这种情况使用callbackFlow可以解决。

Kotlin

```
interface MyCallback {
    fun onSuccess(res: String)
    fun onFailure(errCode: Int)
    fun onError(e: Throwable)
    fun onCompleted()
}

// 方法内部启动协程，调用方传回调方法
fun callTranslateApi(articleList: List<String>, callback: MyCallback) {
    lifecycleScope.launch(Dispatchers.IO) {
        try {
```

```
            articleList.forEach {
                delay(2000)
                val translation = "=${it}="
                if (translation.length > 10) {
                    callback.onFailure(-1)
                } else {
                    callback.onSuccess(translation)
                }
            }
        } catch (e : Throwable) {
            callback.onError(e)
        } finally {
            callback.onCompleted()
        }
    }
}


fun genCallbackFlow() = callbackFlow {
    listOf("hello", "helloworld").also {
        callTranslateApi(it, object : MyCallback {
            override fun onSuccess(res: String) {
                trySend(res)    // callbackflow中通过trySend异步发射数据
            }
            override fun onFailure(errCode: Int) {
                trySend(errCode)
            }

            override fun onError(e: Throwable) {
                trySend(e)
            }
            override fun onCompleted() {
                close()    // 任务完成后通过close方法
            }
        })
        awaitClose {    //当callbackFlow被close或者cancel时会执行
            Log.d("LOLM", "awaitClose")
        }
    }
}

override fun onCreate() {
    lifecycleScope.launchWhenStarted {
        genCallbackFlow().onCompletion {
            Log.d("LOLM", "onCompletion")
        }.flowOn(Dispatchers.IO)
            .collect {
                when (it) {
```

```
                    is Int -> {
                        Log.d("LOLM", "collect failure code: $it")
                    }
                    is String -> {
                        Log.d("LOLM", "collect success res: $it")
                    }
                    is Throwable -> {
                        Log.d("LOLM", "collect exception ex: $it")
                    }
                }
            }
        }
    }
```

Output：

```
15:30:44.930 D/LOLM: collect success res: =hello=
15:30:46.931 D/LOLM: awaitClose
15:30:46.935 D/LOLM: collect failure code: -1
15:30:46.937 D/LOLM: onCompletion
```

总结一下：

1. callbackFlow适合用于**将基于callback的api改写**成支持协程的flow
2. callbackFlow中发射数据使用trySend，关闭flow使用close，一般需要在awaitClose中unRegister callback

回调操作符

onStart

在flow被collect开始时会执行的动作，**可以在onStart中发射数据**，多次设置onStart都会执行。（不用担心上游flow中的onStart覆盖下游或被下游覆盖，其他回调操作符同理）

Kotlin

```kotlin
override fun initData() {
    lifecycleScope.launch {
        (2..4).asFlow().onStart {
            emit(0)
            Log.d("LOLM", "first OnStart")
        }.map {
            it*it
        }.onStart {
            emit(1)
            Log.d("LOLM", "second OnStart")
        }.collect {
            delay(1000)
            Log.d("LOLM", "collect $it")
        }
    }
}
```

输出：

Apache

```
17:45:24 D/LOLM: collect 1
17:45:24 D/LOLM: second OnStart
17:45:25 D/LOLM: collect 0
17:45:25 D/LOLM: first OnStart
17:45:26 D/LOLM: collect 4
17:45:27 D/LOLM: collect 9
17:45:28 D/LOLM: collect 16
```

onCompletion

在flow取消或结束时执行（onCompletion近似于finally,可以理解为前者为声明式写法，后者为命令式写法），可以发射数据。

可以收集到上游流中的异常，但是**不会捕获**

Kotlin

```kotlin
override fun initData() {
    lifecycleScope.launch {
        (2..4).asFlow().onCompletion {
            emit(0)
            Log.d("LOLM", "first onCompletion")
        }.map {
            it*it
        }.onCompletion {
            emit(1)
            Log.d("LOLM", "second onCompletion")
        }.collect {
            delay(1000)
            Log.d("LOLM", "collect $it")
        }
    }
}
```

输出：

Apache

```
17:49:57 D/LOLM: collect 4
17:49:58 D/LOLM: collect 9
17:49:59 D/LOLM: collect 16
17:50:00 D/LOLM: collect 0
17:50:00 D/LOLM: first onCompletion
17:50:01 D/LOLM: collect 1
17:50:01 D/LOLM: second onCompletion
```

onEmpty

当流完成时**没有发射任何数据时**回调，准确的说是**在上游流完成时**

Kotlin

```kotlin
override fun initData() {
    lifecycleScope.launch {
        (2..4).asFlow().onEmpty {
            Log.d("LOLM", "first onEmpty")
        }.filter {
            it < 1
        }.onEmpty {
            emit(-1)
            Log.d("LOLM", "second onEmpty")
        }.collect {
            delay(1000)
            Log.d("LOLM", "collect $it")
        }
    }
}
```

输出：

Apache

```
18:07:21 D/LOLM: collect -1
18:07:21 D/LOLM: second onEmpty
```

这里在filter操作符转换flow之前并非空流，所以第一个onEmpty不会执行。经过filter操作符转换后的flow为空流，所以执行第二个onEmpty并且可以发射数据。

onEmpty操作符可用作**请求数据为空时的兜底操作**

catch

捕获上游流中的异常，一般都是捕获发射的异常，如果想要捕获收集时的异常可以使用onEach操作符改写

Kotlin

```kotlin
override fun initData() {
    lifecycleScope.launch {
        flow {
            emit(1)
            throw Exception("ex when emit")
        }.catch { e ->
            Log.d("LOLM", "catch $e")
        }.collect {
            delay(1000)
            Log.d("LOLM", "collect $it")
        }
    }
}

输出:
18:55:00 D/LOLM: collect 1
18:55:00 D/LOLM: catch java.lang.Exception: ex when emit
```

使用onEach操作符，实现可以捕获收集时的异常

Kotlin

```kotlin
override fun initData() {
    lifecycleScope.launch {
        flow {
            emit(1)
        }.onEach {
            delay(1000)
            throw Exception("ex when collect")
            Log.d("LOLM", "onEach $it")
        }.catch { e ->
            Log.d("LOLM", "catch $e")
        }.collect()
    }
}

输出:
18:59:49 D/LOLM: catch java.lang.Exception: ex when collect
```

变换操作符

transform\transformLatest\transformWhile

transform

转换操作符，传入参数为挂起函数闭包，使用上游发送的每个值去执行该transform函数，可以在transform闭包中发送新值

Makefile

```
lifecycleScope.launch {
    flowOf(1, 2, 3).transform {
        if (it % 2 == 1) {
            emit(it * 2)
            emit(it * 3)
        }
    }.collect {
        Log.d("LOLM", "collect $it")
    }
}

输出：
D/LOLM: collect 2
D/LOLM: collect 3
D/LOLM: collect 6
D/LOLM: collect 9
```

transformLatest

同transform用作上游发射值转换，区别在于： **一旦上游发射新值，会取消当前正在执行的transform闭包**

Makefile

```
lifecycleScope.launch {
    flow {
        emit("a")
        delay(100)
        emit("b")
    }.transformLatest { value ->
        emit(value)
        delay(200)
        emit(value + "_last")
    }.collect {
        Log.d("LOLM", "collect $it")
    }
}

输出:
D/LOLM: collect a
D/LOLM: collect b
D/LOLM: collect b_last
```

应用场景举例：**防抖**：丢弃发射间隔过短的数据。**限流**：下一次请求过来cancel掉之前网络请求，避免短时间内的频繁变化导致的多次网络请求

transformWhile

相比于transform，多了一个cancelWhile的机制，transformWhile传入的函数闭包有Boolean返回值，当返回值为false时会cancel

Makefile

```
lifecycleScope.launch {
    (1..10).asFlow().transformWhile { value ->
        emit(value)
        emit("$value/2")
        value <= 2
    }.collect {
        Log.d("LOLM", "collect $it")
    }
}

输出
D/LOLM: collect 1
D/LOLM: collect 1/2
D/LOLM: collect 2
D/LOLM: collect 2/2
D/LOLM: collect 3
D/LOLM: collect 3/2
```

map\mapNotNull\mapLatest

map

简单看下实现，原理和transform相同，省去了在闭包中emit，直接以返回值的形式做1对1转换

Kotlin

```
public inline fun <T, R> Flow<T>.map(crossinline transform: suspend (value: T)
-> R): Flow<R> = transform { value ->
    return@transform emit(transform(value))
}
```

mapNotNull

相比map，mapNotNull中transform返回值可空，如果返回值为空则不发射

Kotlin

```kotlin
public inline fun <T, R: Any> Flow<T>.mapNotNull(crossinline transform: suspend (value: T) -> R?): Flow<R> = transform { value ->
    val transformed = transform(value) ?: return@transform
    return@transform emit(transformed)
}
```

mapLatest

同transformLatest，新发送的值会取消mapLatest中正在执行的动作以实现防抖

filter\filterNot\filterNotNull

filter

过滤掉不符合预期的值（false: 过滤）

Kotlin

```kotlin
public inline fun <T> Flow<T>.filter(crossinline predicate: suspend (T) -> Boolean): Flow<T> = transform { value ->
    if (predicate(value)) return@transform emit(value)
}
```

filterNot

过滤掉不符合预期的值（true: 过滤）

Kotlin

```kotlin
public inline fun <T> Flow<T>.filterNot(crossinline predicate: suspend (T) -> Boolean): Flow<T> = transform { value ->
    if (!predicate(value)) return@transform emit(value)
}
```

filterNotNull

过滤掉所有发射的null值，**保证流的元素非空**

Groovy

```
lifecycleScope.launch {
    listOf("HELLO","WORLD",null, null, "ZZZ").asFlow()
    .filterNotNull()
    .collect {
        Log.d("LOLM", "collect $it")
    }
}

输出:
D/LOLM: collect HELLO
D/LOLM: collect WORLD
D/LOLM: collect ZZZ
```

withIndex

将上游流包装成带index（从0开始）的对象流

Kotlin

```
public fun <T> Flow<T>.withIndex(): Flow<IndexedValue<T>> = flow {
    var index = 0
    collect { value ->
        emit(IndexedValue(checkIndexOverflow(index++), value))
    }
}

Sample:
lifecycleScope.launch {
    listOf("HELLO","WORLD").asFlow().withIndex().collect {
        Log.d("LOLM", "collect index:${it.index}, value: ${it.value}")
    }
}

Output:
D/LOLM: collect index:0, value: HELLO
D/LOLM: collect index:1, value: WORLD
```

take

take(count: Int) 限长操作符，取发射的前count个元素，当count个元素被消费后，flow会被cancel

Bash

```
lifecycleScope.launch {
    listOf("HELLO","WORLD").asFlow().take(1).collect {
        Log.d("LOLM", "collect $it")
    }
}

输出:
D/LOLM: collect HELLO
```

末端操作符

collect/collectIndexed/collectLatest

collect

最基本的消费发射元素的方法

collectIndexed

收集时带下标（从0开始）

Groovy

```
lifecycleScope.launch {
    (1..3).asFlow().collectIndexed { index, value ->
        Log.d("LOLM", "index: $index , value: $value")
    }
}

输出:
D/LOLM: index: 0 , value: 1
D/LOLM: index: 1 , value: 2
D/LOLM: index: 2 , value: 3
```

collectLatest

同transformLatest，在收集到发射的新值时会取消当前操作，**解决生产者生产速率大于消费者消费速率的背压问题**

launchIn

launchIn(scope: CoroutineScope) 是 scope.launch { flow.collect() } 的一种简写形式

```
Bash
```

```
(1..3).asFlow().onEach {
    Log.d("LOLM", "onEach $it")
}.launchIn(viewModelScope)
```

toList/toSet

将发射的值添加进List/Set

first/firstOrNull

first/last

返回发出的第一个/最后一个值，然后取消flow

如果没有发射任何值则会抛出NoSuchElementException异常

Groovy

```
lifecycleScope.launch {
    var firstElem = emptyFlow<String>().first()
}

crash日志:
java.util.NoSuchElementException: Expected at least one element


lifecycleScope.launch {
    val res = flow {
        emit(1)
        delay(1000)
        emit(2)
    }.onStart {
        Log.d("LOLM", "onStart")
    }.last()
    Log.d("LOLM", "res: $res")
}

输出:
17:46:12 D/LOLM: onStart
17:46:13 D/LOLM: res: 2
```

firstOrNull/lastOrNull

同first，如果没有发射任何值会返回Null，不会抛异常

Single/SingleOrNull

同first/firstOrNull，区别在于Single只要发射值不是一个（空流或者发射不止一个值）都会抛异常

fold

迭代计算，传入初始值和迭代函数，返回终值。

对于空流则会返回初始值

PHP

```
lifecycleScope.launch {
    var res = (1..5).asFlow().fold(1) { result, value ->
        result + value
    }
    Log.d("LOLM", "res: $res")
}

输出:
D/LOLM: res: 16
```

reduce

类似fold，将流中的元素累积计算。

相比fold，**没有初始值，空流会抛异常**

PHP

```
lifecycleScope.launch {
    var res = (1..5).asFlow().reduce { result, value ->
        result + value
    }
    Log.d("LOLM", "res: $res")
}

输出
D/LOLM: res: 15
```

组合操作符

zip

组合两个流中发射的值，组合后的流会在其中一个流结束时结束

Groovy

```
val flowA = (1..3).asFlow().withIndex().onEach {
    delay(it.index * 100L)
    Log.d("LOLM", "flowA emit: ${it.value}")
}
val flowB =
    listOf("hello", "world", "kotlin").asFlow().withIndex()
        .onEach {
        delay(300L - it.index * 100L)
        Log.d("LOLM", "flowB emit: ${it.value}")
    }
lifecycleScope.launch {
    flowA.zip(flowB) { aInt, bStr ->
        "${aInt.value + 100} with ${bStr.value}"
    }.collect {
        Log.d("LOLM", "collect: $it")
    }
}

输出:
14:47:36.579 D/LOLM: flowA emit: 1
14:47:36.880 D/LOLM: flowB emit: hello
14:47:36.881 D/LOLM: collect: 101 with hello
14:47:36.981 D/LOLM: flowA emit: 2
14:47:37.081 D/LOLM: flowB emit: world
14:47:37.082 D/LOLM: collect: 102 with world
14:47:37.183 D/LOLM: flowB emit: kotlin
14:47:37.282 D/LOLM: flowA emit: 3
14:47:37.283 D/LOLM: collect: 103 with kotlin
```

上面例子里，flowA/flowB没有指定协程上下文，都在同一个协程中发射、消费数据。

可以结合flowOn、buffer等操作符使两个流可以并发收集

Groovy

```
1
2  val flowA = (1..3).asFlow().withIndex().onEach {
3      delay(it.index * 100L)
4      Log.d("LOLM", "flowA emit: ${it.value}")
5  }.flowOn(Dispatchers.IO)
6  val flowB =
7      listOf("hello", "world", "kotlin").asFlow().withIndex()
8          .onEach {
9          delay(300L - it.index * 100L)
10         Log.d("LOLM", "flowB emit: ${it.value}")
11     }
12 lifecycleScope.launch {
13     flowA.zip(flowB) { aInt, bStr ->
14         "${aInt.value + 100} with ${bStr.value}"
15     }.collect {
16         Log.d("LOLM", "collect: $it")
17     }
18 }
19
20 输出:
21
22 14:56:19.169 D/LOLM: flowA emit: 1
23 14:56:19.271 D/LOLM: flowA emit: 2
24 14:56:19.470 D/LOLM: flowB emit: hello
25 14:56:19.471 D/LOLM: collect: 101 with hello
26 14:56:19.472 D/LOLM: flowA emit: 3
27 14:56:19.672 D/LOLM: flowB emit: world
28 14:56:19.673 D/LOLM: collect: 102 with world
29 14:56:19.775 D/LOLM: flowB emit: kotlin
30 14:56:19.776 D/LOLM: collect: 103 with kotlin
```

zip适用于**两个流严格一一对应组合**的情况，任何一个流发射的任何值只会被消费一次，只有两个流都发射值时才会触发zip后的新流的发射

combine

flowA.conbine(flowB){ ... } / combine(flowA, flowB) { ... }

结合两个流并生成一个新流，两个流的**任意一个流发射值都会结合另一个流当前的最新值执行传入的transform方法并将返回值发送至新流**

可以组合2/3/4/5个流

Groovy

```
val flowA = (1..3).asFlow().withIndex().onEach {
    delay(it.index * 100L)
    Log.d("LOLM", "flowA emit: ${it.value}")
}
val flowB =
    listOf("hello", "world", "kotlin").asFlow().withIndex()
        .onEach {
        delay(300L - it.index * 100L)
        Log.d("LOLM", "flowB emit: ${it.value}")
    }
lifecycleScope.launch {
    flowA.combine(flowB) { aInt, bStr ->
        "${aInt.value + 100} with ${bStr.value}"
    }.collect {
        Log.d("LOLM", "collect: $it")
    }
}

输出:
16:25:54.481 D/LOLM: flowA emit: 1
16:25:54.612 D/LOLM: flowA emit: 2
16:25:54.781 D/LOLM: flowB emit: hello
16:25:54.781 D/LOLM: collect: 102 with hello
16:25:54.814 D/LOLM: flowA emit: 3
16:25:54.814 D/LOLM: collect: 103 with hello
16:25:54.983 D/LOLM: flowB emit: world
16:25:54.984 D/LOLM: collect: 103 with world
16:25:55.086 D/LOLM: flowB emit: kotlin
16:25:55.086 D/LOLM: collect: 103 with kotlin
```

从上面的实验可以发现：

1. 通过combine组合的两个流即使不指定协程上下文也会**异步发射**
2. 当其中一个流未发射任何值时，即使另一个流发射值，组合后的新流也不会发射值。即组合后的流**只在两个流都有值**且其中任意一个流发射值时才会发射
3. combine适用于结合**表示最新状态/最近一次操作结果**的流，而**无需关心中间发射的值**，如页面ui状态，用户通过键盘输入的结果等

combineTransform

combine和combineTransform的区别可以理解成map和transform的区别，可以在combineTransform中transform方法里发射多个值

merge

合并两个/多个流，按照每个流各自的时间顺序（且在不同协程中）向新流中发射，**合并后的新流是一个channelFlow**，生产消费按照异步非阻塞模型

Kotlin

```kotlin
public fun <T> Iterable<Flow<T>>.merge(): Flow<T> {
    return ChannelLimitedFlowMerge(this)
}

internal class ChannelLimitedFlowMerge<T>(
    private val flows: Iterable<Flow<T>>,
    context: CoroutineContext = EmptyCoroutineContext,
    capacity: Int = Channel.BUFFERED,
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND
) : ChannelFlow<T>(context, capacity, onBufferOverflow)
```

Sample：

Groovy

```
val flowA = (1..3).asFlow().onEach {
    delay(1000L)
    Log.d("LOLM", "flowA emit: $it")
}
val flowB =
    listOf("hello", "world", "kotlin").asFlow()
        .onEach {
        delay(2000L)
        Log.d("LOLM", "flowB emit: $it")
    }
lifecycleScope.launch {
    listOf(flowA,flowB).merge().collect {
        delay(3000)
        Log.d("LOLM", "collect: $it")
    }
}

输出:
11:00:36 D/LOLM: flowA emit: 1
11:00:37 D/LOLM: flowB emit: hello
11:00:37 D/LOLM: flowA emit: 2
11:00:38 D/LOLM: flowA emit: 3
11:00:39 D/LOLM: flowB emit: world
11:00:39 D/LOLM: collect: 1
11:00:41 D/LOLM: flowB emit: kotlin
11:00:42 D/LOLM: collect: hello
11:00:45 D/LOLM: collect: 2
11:00:48 D/LOLM: collect: 3
11:00:51 D/LOLM: collect: world
11:00:54 D/LOLM: collect: kotlin
```

上面示例再次证明：1. merge后的流的收集在单独的协程中 2. 被merge的流独立发射，互不影响

flatten/flattenConcat/flattenMerge

flatten已废弃，被flattenConcat替代

flattenConcat

展开流，Flow<Flow<T>> => Flow<T>

Kotlin

```kotlin
public fun <T> Flow<Flow<T>>.flattenConcat(): Flow<T> = flow {
    collect { value -> emitAll(value) }
}
```

Sample:

Groovy

```groovy
val flowA = flow {
    emit((1..3).asFlow().onEach {
        delay(1000)
        Log.d("LOLM", "emit: $it")
    })
    emit((4..5).asFlow().onEach {
        delay(2000)
        Log.d("LOLM", "emit: $it")
    })
}
lifecycleScope.launch {
    flowA.flattenConcat().collect {
        delay(3000)
        Log.d("LOLM", "collect: $it")
    }
}

输出:
18:21:12 D/LOLM: emit: 1
18:21:15 D/LOLM: collect: 1
18:21:16 D/LOLM: emit: 2
18:21:19 D/LOLM: collect: 2
18:21:20 D/LOLM: emit: 3
18:21:23 D/LOLM: collect: 3
18:21:25 D/LOLM: emit: 4
18:21:28 D/LOLM: collect: 4
18:21:30 D/LOLM: emit: 5
18:21:33 D/LOLM: collect: 5
```

flattenMerge

展开流，同merge可以**并发收集流**，并且会产生一个**channelFlow**（异步消费）

入参concurrency用于控制**可并发收集的流的个数**

0: 等同于flattenConcat

默认值：systemProperty配置，默认16

Kotlin

```
public fun <T> Flow<Flow<T>>.flattenMerge(concurrency: Int = DEFAULT_CONCURRENCY): Flow<T> {
    require(concurrency > 0) { "Expected positive concurrency level, but had $concurrency" }
    return if (concurrency == 1) flattenConcat() else ChannelFlowMerge(this, concurrency)
}
```

Sample：

Groovy

```
val flowA = flow {
    emit((1..3).asFlow().onEach {
        delay(1000)
        Log.d("LOLM", "emit: $it")
    })
    emit((4..5).asFlow().onEach {
        delay(2000)
        Log.d("LOLM", "emit: $it")
    })
}
lifecycleScope.launch {
    flowA.flattenMerge().collect {
        delay(1000)
        Log.d("LOLM", "collect: $it")
    }
}

输出:
18:38:40 D/LOLM: emit: 1
18:38:41 D/LOLM: emit: 4
18:38:41 D/LOLM: collect: 1
18:38:41 D/LOLM: emit: 2
18:38:42 D/LOLM: collect: 4
18:38:42 D/LOLM: emit: 3
18:38:43 D/LOLM: emit: 5
18:38:43 D/LOLM: collect: 2
18:38:44 D/LOLM: collect: 3
18:38:45 D/LOLM: collect: 5
```

flatMap/flatMapConcat/flatMapMerge

flatMap已废弃，被flatMapConcat取代

flatMapConcat

先通过map转换成Flow<Flow<T>>， 再通过flattenConcat展开

等价于map(transform).flattenConcat()

Kotlin

```kotlin
public fun <T, R> Flow<T>.flatMapConcat(transform: suspend (value: T) -> Flow<R>): Flow<R> =
    map(transform).flattenConcat()
```

Sample：

Groovy

```groovy
lifecycleScope.launch {
    (1..3).asFlow().flatMapConcat {
        (0..it).asFlow()
    }.collect {
        Log.d("LOLM", "collect: $it")
    }
}

输出：
D/LOLM: collect: 0
D/LOLM: collect: 1
D/LOLM: collect: 0
D/LOLM: collect: 1
D/LOLM: collect: 2
D/LOLM: collect: 0
D/LOLM: collect: 1
D/LOLM: collect: 2
D/LOLM: collect: 3
```

flatMapMerge

等价于 map(transform).flattenMerge()

Kotlin

```kotlin
public fun <T, R> Flow<T>.flatMapMerge(
    concurrency: Int = DEFAULT_CONCURRENCY,
    transform: suspend (value: T) -> Flow<R>
): Flow<R> =
    map(transform).flattenMerge(concurrency)
```

其他操作符

背压相关

buffer

将上游flow通过一个特定容量的channel发射，并在另一个协程里收集

Kotlin

```kotlin
public fun <T> Flow<T>.buffer(
capacity: Int = BUFFERED,
onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND)
: Flow<T> {
            ...
    }
}
```

通过capacity和onBufferOverflow去指定channel的容量和即将溢出时的策略

buffer操作符默认capacity BUFFERED默认是64， 默认溢出策略是SUSPEND

Makefile

```
(1..3).asFlow().onStart {
    Log.d("LOLM", "onStart")
}.map {
    delay(100)
    it * it
}.buffer().onEach {
    delay(200)
    Log.d("LOLM", "onEach $it")
}.launchIn(lifecycleScope)

输出:
20:52:57.892 D/LOLM: onStart
----- ≈ 300ms  -----
20:52:58.220 D/LOLM: onEach 1
----- ≈ 200ms  -----
20:52:58.421 D/LOLM: onEach 4
----- ≈ 200ms  -----
20:52:58.622 D/LOLM: onEach 9
```

conflate

跳过中间发射的值，只取最新值

Kotlin

```
public fun <T> Flow<T>.conflate(): Flow<T> = buffer(CONFLATED)


public fun <T> Flow<T>.buffer(
capacity: Int = BUFFERED,
onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND)
: Flow<T> {

    ...
    if (capacity == CONFLATED) {
        capacity = 0
        onBufferOverflow = BufferOverflow.DROP_OLDEST
    }
    ...
    }
}
```

可以看出，conflate操作符就是快速进行**capacity为0，溢出策略为DROP_OLDEST**的buffer操作。

Makefile

```
(1..4).asFlow().onStart {
    Log.d("LOLM", "onStart")
}.map {
    delay(100)
    it * it
}.conflate().onEach {
    delay(500)
    Log.d("LOLM", "onEach $it")
}.launchIn(lifecycleScope)

输出:
D/LOLM: onStart
D/LOLM: onEach 1
D/LOLM: onEach 16
```

从输出不难理解conflate会保证缓冲区永远只有一个最新值，溢出策略除了SUSPEND、DROP_OLDEST之外还有DROP_LATEST：

Ada

```
(1..4).asFlow().onStart {
    Log.d("LOLM", "onStart")
}.map {
    delay(100)
    it * it
}.buffer(0, BufferOverflow.DROP_LATEST).onEach {
    delay(500)
    Log.d("LOLM", "onEach $it")
}.launchIn(lifecycleScope)

输出:
D/LOLM: onStart
D/LOLM: onEach 1
D/LOLM: onEach 4
```

retryWhen/retry

retryWhen

异常重试，当传入闭包中返回true时重试，否则不重试

Groovy

```
flow<Int> {
    Log.d("LOLM", "emit throw ex")
    throw IOException("")
}.retryWhen { cause, attempt ->
    if (attempt > 3) {
        return@retryWhen false
    }
    true
}.collect {
    Log.d("LOLM", "collect: $it")
}

输出:
D/LOLM: emit throw ex
D/LOLM: emit throw ex
D/LOLM: emit throw ex
D/LOLM: emit throw ex
D/LOLM: emit throw ex
```

cause： 抛出的异常 attempt： 重试次数，从0开始

retry

retry(times: Long)指定重试次数

PHP

```
lifecycleScope.launch {
    flow<Int> {
        Log.d("LOLM", "emit throw ex")
        throw IOException("")
    }.retry(2).collect {
        Log.d("LOLM", "collect: $it")
    }
}

输出
D/LOLM: emit throw ex
D/LOLM: emit throw ex
D/LOLM: emit throw ex
```

flowOn

改变flow执行的上下文，只会影响flowOn**操作符之前没有上下文的操作符**

flowOn操作符不会污染到下游，且拥有上下文的操作符会保持上下文

举例：

Kotlin

```
withContext(Dispatchers.Main) {
    val singleValue = intFlow // will be executed on IO if context wasn't specified before
        .map { ... } // Will be executed in IO
        .flowOn(Dispatchers.IO)
        .filter { ... } // Will be executed in Default
        .flowOn(Dispatchers.Default)
        .single() // Will be executed in the Main
}
```

debounce

防抖操作符，**把发射时间距离前一个值发射时间间隔小于timeout的值过滤掉**，但是**最后一个值必然会被发射**。因此如果上游流两个值的间隔时间都小于timeout的话，经过debounce之后只会发射最后一个值。

Groovy

```
lifecycleScope.launch {
    (1..4).asFlow().withIndex().onEach { delay(101L * it.index) }.debounce(300).collect {
        Log.d("LOLM", "collect: $it")
    }
}

输出:
D/LOLM: collect: IndexedValue(index=2, value=3)
D/LOLM: collect: IndexedValue(index=3, value=4)
```

cancellable

flow默认是不可取消的，想要取消flow只需要取消收集flow的协程即可，如果非要在flow内部控制取消，想要**确保取消后不再发射**，可以使用cancellable操作符

Kotlin

```kotlin
private class CancellableFlowImpl<T>(private val flow: Flow<T>) : CancellableFlow<T> {
    override suspend fun collect(collector: FlowCollector<T>) {
        flow.collect {
            currentCoroutineContext().ensureActive()
            collector.emit(it)
        }
    }
}
```

可以看出原理很简单，在flow的发出每个值之前先检查一下当前协程是否存活。

如果不用cancellable，在flow中调用cancel会怎么样?

实验一：

Groovy

```groovy
lifecycleScope.launch {
    (1..3).asFlow().collect {
        if (it == 2) {
            cancel()
        }
        Log.d("LOLM", "collect: $it")
    }
}

输出：
D/LOLM: collect: 1
D/LOLM: collect: 2
D/LOLM: collect: 3
```

使用IntRange.asFlow构建的flow，cancel后依然正常发射，即发射前没有检查协程状态

实验二：

PHP

```
lifecycleScope.launch {
    flow {
        (1..3).forEach {
            emit(it)
        }
    }.collect {
        if (it == 2) {
            cancel()
        }
        Log.d("LOLM", "collect: $it")
    }
}

输出
D/LOLM: collect: 1
D/LOLM: collect: 2
```

使用flow { ... }构建的流，cancel后不再发射新值

实验三：

Bash

```
lifecycleScope.launch {
    (1..3).asFlow().cancellable().collect {
        if (it == 2) {
            cancel()
        }
        Log.d("LOLM", "collect: $it")
    }
}
输出:
D/LOLM: collect: 1
D/LOLM: collect: 2
```

使用cancellable后，即使是使用IntRange.asFlow构建的flow也可以被准确及时取消

总结：

考虑到性能问题, 数据流的大多数操作符不会自己做额外的取消检查。因此**如果需要在flow内部控制外部协程的取消，需要加上cancellable操作符**。