

# Kotlin Flow上手指南（二）ChannelFlow



yuPFeG1819

2021年11月25日 15:00 · 阅读 2875

[关注](#)

继[上一篇](#)介绍了 `Flow` 的一些基本用法，足够满足大部分日常场景。

这是Kotlin协程系列的第三篇文章。

本篇就来看看 `Flow` 背后的影子兄弟——`Channel`，以及由此延伸出来的 `ChannelFlow`，并挖掘其背后实现原理

本篇内容有些还处于实验API阶段，后续可能会被修改。

Kotlin协程系列相关文章导航

[扒一扒Kotlin协程](#)

[Kotlin Flow上手指南（一）基础应用](#)

[Kotlin Flow上手指南（二）ChannelFlow（本篇）](#)

[Kotlin Flow上手指南（三）SharedFlow与StateFlow](#)

Kotlin 版本：1.5.31

Coroutine 版本： 1.5.2

以下正文。

## Channel

`Channel` 在概念上类似于 `BlockQueue`，[并发安全](#)的缓冲队列（先进先出），[实现多个协程间的通信](#)。

目前版本内部基于[无锁双向链表](#)结构，存在一个[永远不会删除的节点](#)作为将整个链表进行首尾相连的[根节点](#)。

- 添加新元素时，就会在根节点的左侧进行添加，即添加到整个队列的末尾。
- 取出元素时，会在先从根节点右侧开始移除并取出元素，即从队列顶部取出。
- 每个元素节点的左侧（前）、右侧（后）节点都为CAS引用类型。

`Channel` 内的发送数据和接收数据默认都是挂起函数。

对于同一个 `Channel` 对象，允许多个协程发送数据，也允许多个协程接收数据。

区别于 `Flow`，`Channel` 是一个**热流**，但其并不支持数据流操作。

即使没有订阅消费，生产端同样也会开始发送数据，并且始终处于运行状态。

```
● ● ●

public interface Channel<E> : SendChannel<E>, ReceiveChannel<E> {
    // 实际就是flow的buffer操作符中设置的缓存区容量类型
    public companion object Factory {
        public const val UNLIMITED: Int = Int.MAX_VALUE
        public const val RENDEZVOUS: Int = 0
        public const val CONFLATED: Int = -1
        public const val BUFFERED: Int = -2
        ...
        // 默认实现的缓冲区队列(默认容量64)
        internal val CHANNEL_DEFAULT_CAPACITY = systemProp(
            DEFAULT_BUFFER_PROPERTY_NAME,
            64, 1, UNLIMITED - 1
        )
    }
}
```

@稀土掘金技术社区

`Channel` 接口只是定义了一些常量，实际功能定义还在其继承的两个接口 `SendChannel` 与 `ReceiveChannel`。

## 「创建」

`Channel` 本身是个接口不能直接创建，但其有个同名函数用于创建 `Channel`，相当于一个工厂方法。

```
● ● ●

public fun <E> Channel(
    // 缓冲队列容量
    capacity: Int = RENDEZVOUS,
    // 背压策略
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND,
    // 在元素被发送但未交付给消费者时调用代码块
    onUndeliveredElement: ((E) → Unit)? = null
): Channel<E>
```

@稀土掘金技术社区

其中 `capacity` 参数为缓冲区容量，通常是以 `Channel` 中定义的常量为值。

- **RENDEZVOUS**

默认无锁、无缓冲区，只有消费端调用时，才会发送数据，否则挂起发送操作。

当缓存策略不为 `BufferOverflow.SUSPEND` 时，会创建缓冲区容量为1的 `ArrayChannel`。

- **CONFLATED**

队列容量仅为一，且 `onBufferOverflow` 参数只能为 `BufferOverflow.SUSPEND`。

缓冲区满时，永远用最新元素替代，之前的元素将被废弃。

创建实现类 `ConflatedChannel`

内部会使用 `ReentrantLock` 对发送与接收元素操作进行加锁，线程安全。

- **UNLIMITED**

无限制容量，缓冲队列满后，会直接扩容，直到OOM。

内部无锁，永远不会挂起

- **BUFFERED**

默认创建64位容量的缓冲队列，当缓存队列满后，会挂起发送数据，直到队列有空余。

创建实现类 `ArrayChannel`，内部会使用 `ReentrantLock` 对发送与接收元素操作进行加锁，线程安全。

- 自定义容量

当 `capacity` 容量为1，且 `onBufferOverflow` 为 `BufferOverflow.DROP_OLDEST` 时，由于与 `CONFLATED` 工作原理相同，会直接创建为实现类 `ConflatedChannel`。

其他情况都会创建为实现类 `ArrayChannel`。

## 「生产者」

`SendChannel` 是发送数据的生产者。

- **send**

挂起函数，向队列中添加元素，在缓冲队列满时，会挂起协程，暂停存入元素，直到队列容量满足存入需求，恢复协程。

```
public suspend fun send(element: E)
```

kotlin 复制代码

- **trySend**

尝试向队列中添加元素，返回 ChannelResult 表示操作结果。

```
public fun trySend(element: E): ChannelResult<Unit>
```

kotlin 复制代码

- **close** 关闭队列，**幂等操作**，后续操作都无效，只允许存在一个

```
public fun close(cause: Throwable? = null): Boolean
```

kotlin 复制代码

- **isClosedForSend**

实验性质API，为ture时表示 Channel 已经关闭，停止发送。

## » ProducerScope

SendChannel 还有个子接口 ProducerScope，表示允许发送数据的协程作用域。

还是处于实验性质的api，不推荐外部直接使用

```
@ExperimentalCoroutinesApi
public interface ProducerScope<in E> : CoroutineScope, SendChannel<E> {
    public val channel: SendChannel<E>
}
```

@稀土掘金技术社区

同时官方还提供了一个 CoroutineScope 的拓展函数 produce，用于快速启动生产者协程，并返回 ReceiveChannel。

其内部实际是在协程构建中创建 Channel，用以发送数据。

```
//Produce.kt
@ExperimentalCoroutinesApi
public fun <E> CoroutineScope.produce(
    context: CoroutineContext = EmptyCoroutineContext,
    //缓存区容量
    capacity: Int = 0,
    @BuilderInference block: suspend ProducerScope<E>.() -> Unit
): ReceiveChannel<E> = produce(
    context, //默认协程上下文为空实现，没有调度器
    capacity,
    BufferOverflow.SUSPEND, // 默认背压策略，缓存区满就挂起发送函数
    CoroutineStart.DEFAULT, //立即执行
    onCompletion = null, block = block
)
...
internal fun <E> CoroutineScope.produce(...): ReceiveChannel<E> {
    val channel = Channel<E>(capacity, onBufferOverflow)
    //创建新的协程上下文，在未指定协程调度器时，会默认使用Dispatchers.Default
    val newContext = newCoroutineContext(context)
    //协程体中包装上Channel，在结束协程时关闭Channel
    val coroutine = ProducerCoroutine(newContext, channel)
    if (onCompletion != null){
        coroutine.invokeOnCompletion(handler = onCompletion)
    }
    coroutine.start(start, coroutine, block)
    return coroutine
}
```

@稀土掘金技术社区

- [awaitClose](#)

**挂起函数**，`ProducerScope` 中的拓展函数，会挂起等待协程关闭，在关闭前执行操作，通常用于资源回收。

调用 `awaitClose` 后，需要外部手动调用 `SendChannel` 的 `close` 进行关闭，否则协程会一直**挂起**等待关闭，直到协程作用域被关闭。

```

@ExperimentalCoroutinesApi
public suspend fun ProducerScope<*>.awaitClose(
    block: () -> Unit = {}
) {
    ...
    try {
        suspendCancellableCoroutine<Unit> { cont ->
            invokeOnClose { cont.resume(Unit) }
        }
    } finally {
        block()
    }
}

```

@稀土掘金技术社区

## 「消费者」

`ReceiveChannel` 表示接收数据的消费者，其只有一个收集数据的作用

- `receive`

挂起函数，从缓冲队列中接收并移除元素，如果缓冲队列为空，则挂起协程。

如果在 `Channel` 被关闭后，调用 `receive` 去取值，会抛出 `ClosedReceiveChannelException` 异常。

`public suspend fun receive(): E`

kotlin 复制代码

- `receiveCatching`

挂起函数，功能与 `receive` 相同，只是防止在缓冲队列关闭时突然抛出异常导致程序崩溃，会返回 `ChannelResult` 包裹取出的元素值，同时表示当前操作的状态

`public suspend fun receiveCatching(): ChannelResult<E>`

kotlin 复制代码

- `tryReceive`

尝试从缓冲队列中拉取元素，返回 `ChannelResult` 包裹取出的元素，并表示操作结果。

`public fun tryReceive(): ChannelResult<E>`

kotlin 复制代码

- `cancel`

缓冲队列的接收端停止接收数据，会移除缓冲队列的所有元素，并停止 `SendChannel` 发送数据，内部会调用 `SendChannel` 的 `close` 函数。

**谨慎调用该函数**，通常 `Channel` 应该由发送端 `SendChannel` 来主导通道是否关闭。

毕竟很少会有老师还在【台上发言】，下面学生就已经说【我听完了】的场景。

- **iterator**

挂起函数，接收 `Channel` 时，允许使用 `for` 循环进行迭代

```
public operator fun iterator(): ChannelIterator<E>
```

kotlin 复制代码

`ReceiverChannel` 的迭代器是挂起函数，只能在协程中使用。

```
public interface ChannelIterator<out E> {
    public suspend operator fun hasNext(): Boolean
    ...
}
```

kotlin 复制代码

可以将缓冲队列中的元素依次取出。

- **actor与ActorScope**

以 `@ObsoleteCoroutinesApi` 注解的废弃api，与 `produce` 相对应的**消费者协程作用域**。

这两个API据说要重新设计，目前不要使用。[issues](#)

```
public interface ActorScope<E> : CoroutineScope, ReceiveChannel<E>
public fun <E> CoroutineScope.actor(...): SendChannel<E>
```

kotlin 复制代码

- **consume**

`ReceiveChannel` 的拓展函数，在 `Channel` 出现异常或结束后，调用 `cancel` 关闭 `Channel` 接收端。

```

public inline fun <E, R> ReceiveChannel<E>.consume(
    block: ReceiveChannel<E>.() -> R
): R {
    ...
    try {
        return block()
    } catch (e: Throwable) {
        cause = e
        throw e
    } finally {
        cancelConsumed(cause) // 内部调用cancel
    }
}

```

@稀土掘金技术社区

## 「 ChannelResult 」

`ChannelResult` 是一个内联类，仅用于表示 `Channel` 操作的结果，并携带元素。

```

@JvmInline
public value class ChannelResult<out T>{
    ...
    // 存入\取出是否操作成功
    public val isSuccess: Boolean
    public val isFailure: Boolean
    // 通道是否已关闭
    public val isClosed: Boolean
    // 获取取出的元素，没有则null
    public fun getOrNull(): T?
    // 获取取出的元素，没有则抛出异常
    public fun getOrThrow(): T
    ...
}

```

@稀土掘金技术社区

## 「 小结 」

`Channel` 目前版本仅作为生产者-消费者模型缓冲队列，多协程间通信的基础设施而存在。

在 `Flow` 中只要涉及到切换协程调度器与背压缓冲都少不了 `Channel` 参与的身影。

## 选择表达式

说到 `Channel` 就不得不提 `Kotlin Coroutine` 中的特殊机制——**选择表达式**。

在 `Kotlin Coroutine` 中存在一个特殊的挂起函数——`select`，其**允许同时等待多个挂起的结果，并且只取用其中最快完成的作为函数恢复的值**。



```

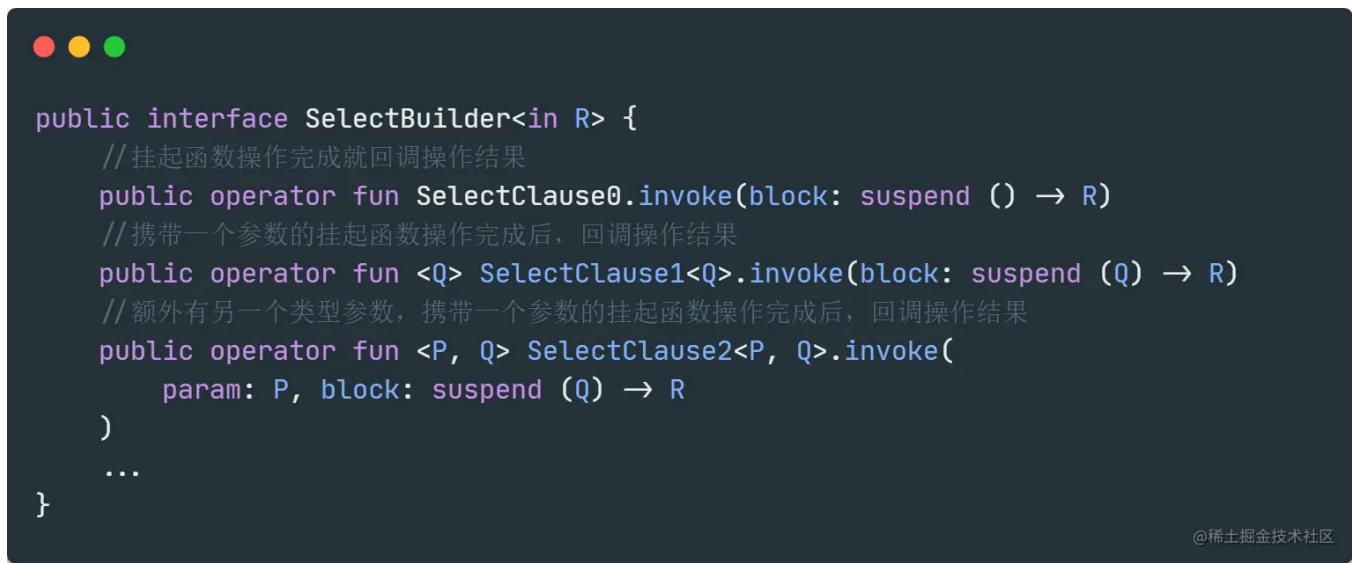
public suspend inline fun <R> select(
    crossinline builder: SelectBuilder<R>.() -> Unit
): R {
    ...
    return suspendCoroutineUninterceptedOrReturn { uCont ->
        val scope = SelectBuilderImpl(uCont)
        try {
            builder(scope)
        } catch (e: Throwable) {
            scope.handleBuilderException(e)
        }
        scope.getResult()
    }
}

```

@稀土掘金技术社区

初看这个函数实现，与 Kotlin 协程中回调 API 转协程的 `suspendCoroutine` 函数非常相似，同样是尝试获取结果，否则就挂起等待结果。

不过其参数 `builder` 却是以 `SelectBuilder` 作为接收者的函数类型。



```

public interface SelectBuilder<in R> {
    // 挂起函数操作完成就回调操作结果
    public operator fun SelectClause0.invoke(block: suspend () -> R)
    // 携带一个参数的挂起函数操作完成后，回调操作结果
    public operator fun <Q> SelectClause1<Q>.invoke(block: suspend (Q) -> R)
    // 额外有另一个类型参数，携带一个参数的挂起函数操作完成后，回调操作结果
    public operator fun <P, Q> SelectClause2<P, Q>.invoke(
        param: P, block: suspend (Q) -> R
    )
    ...
}

```

@稀土掘金技术社区

`select` 函数更像是一种 `Kotlin DSL` 的语法，在 `select` 的代码块并不是挂起函数，只能调用 `SelectBuilder` 中定义的表达子句。

允许从上述**子表达式**中，**选择其中一项执行**，同时 `select` 会将子表达式中最后语句的类型，作为自身的返回值类型。

- `select` 会**优先执行第一个表达式**，如果第一项无法执行，才会选择下一项，优先级依次类推。

- 如果需要完全公平的选择表达式，则使用 `selectUnbiased`。

而要使用选择表达式，就要使用返回值为 `SelectClause` 系列类型的函数作为子语句。

## 「Deferred选择」

在使用 `async` 启动协程的返回类型 `Deferred` 中，就定义了 `onAwait` 函数作为选择表达式的子表达式，以 `SelectClause1` 作为返回值类型。



```

public interface Deferred<out T> : Job {
    ...
    public val onAwait: SelectClause1<T>
}

```

@稀土掘金技术社区

其作用和 `await` 是一致的，只是当其在 `select` 语句中作为子语句时，就能同时等待多个协程返回值，并且选择其中最快执行完成的一个作为实际返回值。

kotlin 复制代码

```

fun testSelect() = runBlocking {
    val d1 = async {
        delay(60)
        1
    }
    val d2 = async {
        delay(50)
        2
    }
    val d3 = async {
        delay(70)
        3
    }

    val data = select<Int> {
        d3.onAwait{data->
            println("d3 first result $data")
            data
        }
        d1.onAwait{data->
            println("d1 first result :$data")
            data
        }
        d2.onAwait{i->
            println("d2 first result : $data")
            data
        }
    }
}

```

```

    }
    println("result : $data")
}

d2 first result : 2
result : 2

```

由于第2项 `Deferred` 是最先通过 `await` 获取到值的，所以 `select` 也是以其作为返回值。

## 「Channel选择」

同样的，在 `Channel` 中也定义了以 `SelectClause` 类型为返回值的函数。



```

public interface SendChannel<in E> {
    ...
    public val onSend: SelectClause2<E, SendChannel<E>>
    ...
}

public interface ReceiveChannel<out E> {
    ...
    public val onReceive: SelectClause1<E>
    ...
    public val onReceiveCatching: SelectClause1<ChannelResult<E>>
    ...
}

```

@稀土掘金技术社区

- **onSend**：等效于 `send`，参数作为需要发送的数据，并在被选择后，回调当前执行发送的 `Channel` 实例。
- **onReceive**：等效于 `receive`，回调从缓存队列中取出值的结果。
- **onReceiveCatching**：等效于 `receiveCatching`，回调从缓存队列中取出值的操作状态 `ChannelResult`。

`Channel` 使用选择表达式，通常用于多个 `Channel` 备份切换的场景。

```

fun testSelectChannel() = runBlocking {
    val slowChannel = Channel<Int>(
        capacity = 1, onBufferOverflow = BufferOverflow.SUSPEND
    )
    val fastChannel = Channel<Int>(
        capacity = 3, onBufferOverflow = BufferOverflow.SUSPEND
    )
    //生产者协程
    launch(Dispatchers.IO) {

```

SCSS 复制代码

```

for (i in 1..5){
    if (!isActive) break
    //选择表达式不需要返回值
    select<Unit> {
        //需要发送的数据
        slowChannel.onSend(i){channel->
            //lambda的参数是当前选中的channel
            println("slow channel selected $channel send $i")
            delay(50)
        }
        fastChannel.onSend(i){channel->
            println("fast channel selected $channel send $i")
        }
    }
    delay(300)
    //注意要关闭通道
    slowChannel.close()
    fastChannel.close()
}

//消费者协程
launch {
    while (isActive && !slowChannel.isClosedForSend && !fastChannel.isClosedForSend){
        //以ChannelResult携带的值作为选择表达式的值
        val result = select<Int> {
            slowChannel.onReceiveCatching{
                println("slowChannel is receive ${it.getOrNull()}")
                delay(100)
                it.getOrNull()?:-1
            }
            fastChannel.onReceiveCatching{
                println("fastChannel is receive ${it.getOrNull()}")
                it.getOrNull()?:-1
            }
        }
        println("receive result : $result")
    }
}
delay(500)
}

```

上述代码中，将 `fastChannel` 作为备份，在 `slowChannel` 无法发送数据时，选择使用 `fastChannel` 发送数据，接收端亦是同样的逻辑。程序运行结果：

sql 复制代码

```

slowChannel receive 1
slow channel selected ArrayChannel@1cc4b438{EmptyQueue}(buffer:capacity=1,size=1) send 1
slow channel selected ArrayChannel@1cc4b438{EmptyQueue}(buffer:capacity=1,size=1) send 2

```

```

receive result : 1
slowChannel receive 2
slow channel selected ArrayChannel@1cc4b438{EmptyQueue}(buffer:capacity=1,size=1) send 3
fast channel selected ArrayChannel@580f2a18{EmptyQueue}(buffer:capacity=2,size=1) send 4
fast channel selected ArrayChannel@580f2a18{EmptyQueue}(buffer:capacity=2,size=2) send 5
receive result : 2
slowChannel receive 3
receive result : 3
fastChannel receive 4
receive result : 4
fastChannel receive 5
receive result : 5
//Channel关闭后，取出元素时ChannelResult携带的元素为null
slowChannel receive null
receive result : -1

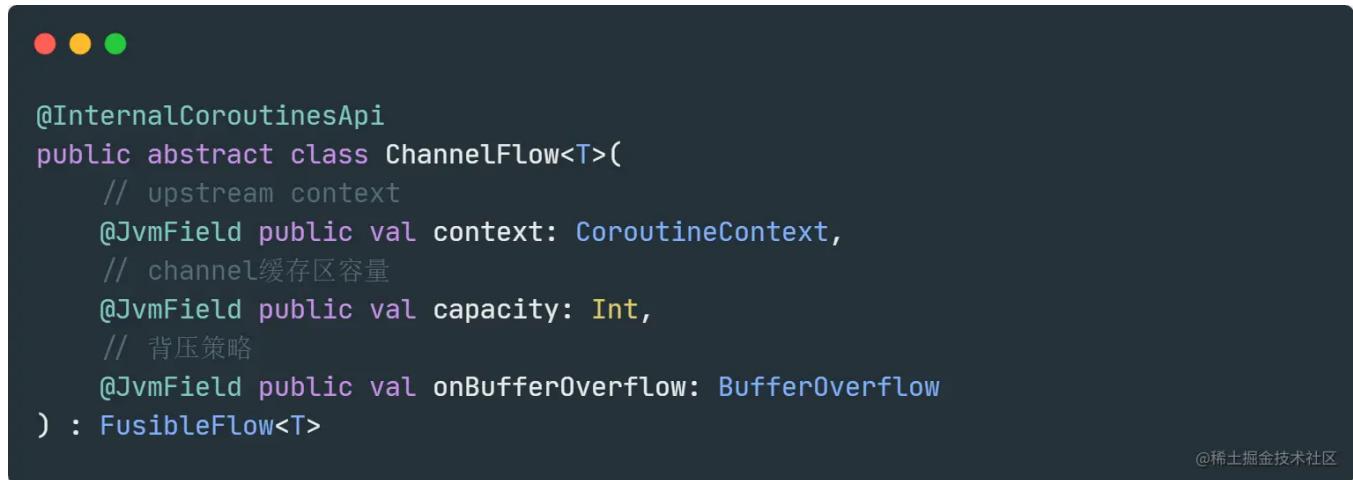
```

## ChannelFlow

虽然 `Channel` 能够在多个协程中线程安全的通信，但做不到复杂的数据流操作，使用上又比较繁琐。

而 `Kotlin Flow` 拥有灵活的数据流操作能力，却是线程不安全的，甚至不允许在发送时切换协程上下文。

那么将这两个组合在一起不就实现互补了吗？于是 `ChannelFlow` 就应运而生了。



不过 `ChannelFlow` 类本身是内部API，在外部是无法直接调用的。

## 「flowOn原理解析」

在上一篇介绍的 `buffer` 和 `flowOn` 操作符，其内部实现 `ChannelFlowOperator` 正是继承于 `ChannelFlow`。

这里就以 `flowOn` 为例，看看内部是如何实现切换 `CoroutineContext` 的。

```
public fun <T> Flow<T>.flowOn(context: CoroutineContext): Flow<T> {
    checkFlowContext(context)
    return when {
        context == EmptyCoroutineContext → this
        // 再次调用，创建ChannelFlowOperator对象，切换到指定上下文
        this is FusibleFlow → fuse(context = context)
        // 首次调用，创建ChannelFlowOperator的实现类
        else → ChannelFlowOperatorImpl(this, context = context)
    }
}
```

@稀土掘金技术社区

`ChannelFlowOperatorImpl` 是 `ChannelFlowOperator` 的实现类，本身只是实现了父类的 `flowCollect`，用于接收上游的数据。

而 `flowCollect` 又是在父类 `ChannelFlowOperator` 中的 `collectTo` 内调用。

同时其还重写了 `collect`，做些快速检测，避免不需要的 `Channel` 创建造成资源浪费，在需要修改协程调度器时，还是使用父类的 `collect` 实现。

其实 `buffer` 操作符内部也是同样创建 `ChannelFlowOperatorImpl`，只是 `capacity` 是允许外部设定的，而 `flowOn` 是固定为 `Channel.OPTIONAL_CHANNEL`。

```

internal class ChannelFlowOperatorImpl<T>(
    flow: Flow<T>,
    context: CoroutineContext = EmptyCoroutineContext,
    capacity: Int = Channel.OPTIONAL_CHANNEL, //flowOn操作符专用的channel容量
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND
) : ChannelFlowOperator<T, T>(flow, context, capacity, onBufferOverflow) {
    ...
    override suspend fun flowCollect(collector: FlowCollector<T>) = flow.collect(collector)
}

internal abstract class ChannelFlowOperator<S, T>(
    @JvmField protected val flow: Flow<S>,
    context: CoroutineContext,
    capacity: Int,
    onBufferOverflow: BufferOverflow
) : ChannelFlow<T>(context, capacity, onBufferOverflow) {
    ...
    protected override suspend fun collectTo(scope: ProducerScope<T>) =
        flowCollect(SendingCollector(scope)) //接收上游数据，再由ProducerScope发送数据

    override suspend fun collect(collector: FlowCollector<T>) {
        //针对flowOn操作符的快速校验通道，过滤不需要修改调度器的情况
        if (capacity == Channel.OPTIONAL_CHANNEL) {
            ...
        }
        // 切换调度器需要用Channel通道
        super.collect(collector)
    }
}
// SendingCollector.kt
@InternalCoroutinesApi
public class SendingCollector<T>(
    private val channel: SendChannel<T>
) : FlowCollector<T> {
    override suspend fun emit(value: T): Unit = channel.send(value) //添加接收到的数据到Channel
}

```

@稀土掘金技术社区

那么当前实现的 `collectTo` 函数中的 `ProducerScope` 类型参数又是从何而来呢？继续向父类 `ChannelFlow` 追溯。

```
①②③④  
@InternalCoroutinesApi  
public abstract class ChannelFlow<T>(...): FusibleFlow<T> {  
    ...  
    internal val collectToFun: suspend (ProducerScope<T>) → Unit  
        get() = { collectTo(it) }  
    //Channel缓冲区容量，flowOn使用默认的64容量  
    internal val produceCapacity: Int  
        get() = if (capacity == Channel.OPTIONAL_CHANNEL) Channel.BUFFERED  
            else capacity  
    ...  
    public open fun produceImpl(scope: CoroutineScope): ReceiveChannel<T> =  
        scope.produce( //创建channel生产者协程  
            context, produceCapacity, onBufferOverflow,  
            start = CoroutineStart.ATOMIC,  
            block = collectToFun  
        )  
  
    override suspend fun collect(collector: FlowCollector<T>): Unit =  
        coroutineScope { //创建channel消费者协程  
            collector.emitAll(produceImpl(this))  
        }  
}
```

@稀土掘金技术社区

其实在父类 `ChannelFlow` 的 `collect` 中也就只是在 [当前收集器所在协程上下文](#) 创建了一个新协程，通过 `emitAll` 发送数据。

很明显，子类实现的 `collectTo` 中的其实就是前面介绍过的 `produce` 里的lamda代码块。在 [新的协程上下文](#) 中创建一个 `Channel` 生产者协程，用于将上游数据添加到 `Channel` 缓冲队列。

不过这里的 `emitAll` 居然这么神奇，还能发送 `ReceiveChannel` 的？

当然这也只是 `FlowCollector` 的拓展函数而已，内部会一直循环取出 `Channel` 内的值，持续发送给下游。

```

public suspend fun <T> FlowCollector<T>.emitAll(
    channel: ReceiveChannel<T>
): Unit = emitAllImpl(channel, consume = true)

private suspend fun <T> FlowCollector<T>.emitAllImpl(
    channel: ReceiveChannel<T>, consume: Boolean
) {
    ensureActive() // 检查协程状态
    ...
    try{
        while (true) {
            //从缓存队列中取出元素
            //以ChannelResult为返回值，channel已关闭则退出循环
            val result = run { channel.receiveCatching() }
            if (result.isClosed) {
                result.exceptionOrNull()?.let { throw it }
                break
            }
            //向数据流下游发送从队列取出的元素
            //如果队列已关闭或取出失败，也会取出异常抛出给下游
            emit(result.getOrDefault())
        }
    }catch (e: Throwable) {
        cause = e
        throw e
    } finally {
        //协程结束后，channel关闭消费端
        if (consume) channel.cancelConsumed(cause)
    }
}

```

@稀土掘金技术社区

## » 小结

所以每次调用 `flowOn` 操作符会在内部创建一个新的 `Channel`，并在新设置的协程调度器，创建新协程，由 `Channel` 发送所有上游数据，添加到缓存队列中。

由默认的背压策略 `BufferOverflow.SUSPEND`，决定缓冲区队列的背压规则，**缓冲区满后挂起发送操作**。

而下游则是在**收集器所在协程调度器**内，新创建一个协程，作为消费者，循环接收 `Channel` 缓冲区队列的值，并发送数据给下游。

这也就是 `flowOn` 的线程调度**只对上游数据流生效**的原因。

## 「回调API转数据流」

而 `ChannelFlow` 的另一个实现 `ChannelFlowBuilder`，则提供了将回调API转化为 `Flow` 数据流（**冷流**）的功能。

官方这里提供了两个对外公开的函数。

- **channelFlow**

```
public fun <T> channelFlow(
    @BuilderInference block: suspend ProducerScope<T>.() -> Unit
): Flow<T> = ChannelFlowBuilder(block)
```

@稀土掘金技术社区

`block` 参数是以 `ProducerScope` 为接收者的函数类型。

这也就是前面提到的 `Channel` 的生产者协程作用域，能够利用 `send` 或者 `trySend` 来发送数据。

- **callbackFlow**

```
public fun <T> callbackFlow(
    @BuilderInference block: suspend ProducerScope<T>.() -> Unit
): Flow<T> = CallbackFlowBuilder(block)
```

@稀土掘金技术社区

实际上 `CallbackFlowBuilder` 就是 `ChannelFlowBuilder` 的子类，其唯一的区别就是在协程代码块结束时，**强制要求** 调用 `close` 或挂起函数 `awaitClose`，用以处理协程结束时的资源回收操作。

关闭 `Channel` 通道，不允许 `Channel` 继续发送元素。

准确的说是 `callbackFlow` 的lambda函数体执行完之前，必须确保调用 `close`，停止 `Channel` 通道发送元素，否则会抛出异常。

而调用 `awaitClose` 后，会**一直挂起**，不执行后续逻辑，一直等待 `Channel` 通道关闭，或者收集器所在协程被关闭。

更多是用于**反注册回调API**，等待注册的回调传递数据，避免内存泄漏，否则抛出异常。

使用 `channelFlow` 与 `callbackFlow` 创建数据流时，允许在生产端的使用 `withContext` 切换协程上下文，默认使用 `collect` 收集器所在协程的协程调度器。

```
fun testChannelFlow() = runBlocking {
    val flow = channelFlow<String> {
        send("11")
        println("send first on ${Thread.currentThread()}")
        withContext(Dispatchers.IO){
            send("22")
        }
    }
    flow.collect { value ->
        println("receive $value on ${Thread.currentThread()}")
    }
}
```

scss 复制代码

```

        println("send second on ${Thread.currentThread()}")
    }
    send("33")
    println("send third on ${Thread.currentThread()}")
    awaitClose {
        println("awaitClose")
    }
}
val job = launch {
    flow.collect {
        println("result : $it")
    }
}
delay(200)
job.cancel() //交由外部协程控制channel通道关闭
}

send first on Thread[Test worker @coroutine#3,5,main]
result : 11
send second on Thread[DefaultDispatcher-worker-1 @coroutine#3,5,main]
result : 22
send third on Thread[Test worker @coroutine#3,5,main]
result : 33
awaitClose

```

## » 拓展应用

假设有这样一段回调函数，要把它变成 **Flow** 数据流。

```

kotlin 复制代码

fun registerCallBack(callBack : (String)->Unit){
    for (i in 0..5){
        //do something
        callBack("data $i")
    }
}

suspend fun createCallBackFlow() = callbackFlow<String>{
    registerCallBack{result->
        send(result) //这里回调是普通函数，是无法调用send的
        trySendBlocking(result)
    }
    //可由外部collect的协程控制关闭
    awaitClose{ //一直挂起等待数据回调
        unRegisterCallBack() //反注册回调，回收资源
    }
}

```

但在这个回调的代码块中，由于并不是挂起函数，所以不能在这里调用 `send` 来发送数据。

此时除了 `trySend` 尝试发送数据，其实还有一种 `SendChannel` 的拓展函数 `trySendBlocking`，一直阻塞线程，等待发送数据结果，返回 `ChannelResult`，表示元素入队操作结果。

`Channel` 通道已经被关闭时，也会返回失败结果。

```
① ② ③

@Throws(InterruptedException::class)
public fun <E> SendChannel<E>.trySendBlocking(
    element: E
): ChannelResult<Unit> {
    trySend(element).onSuccess { return ChannelResult.success(Unit) }
    //阻塞线程的方式构造协程
    return runBlocking {
        val r = runCatching { send(element) }
        if (r.isSuccess) ChannelResult.success(Unit)
        else ChannelResult.closed(r.exceptionOrNull())
    }
}

@稀土掘金技术社区
```

不要在挂起函数或协程中调用该函数，仅推荐在普通回调函数内调用。

- 在线程阻塞时，如果线程被结束会抛出 `InterruptedException` 异常。

## » 原理解析

那么 `ChannelFlowBuilder` 内部究竟是如何将回调转成 `Flow` 数据流的呢？

```
① ② ③

private open class ChannelFlowBuilder<T>(
    private val block: suspend ProducerScope<T>.() -> Unit,
    context: CoroutineContext = EmptyCoroutineContext,
    capacity: Int = BUFFERED, //Channel缓存容量
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND //背压策略
) : ChannelFlow<T>(context, capacity, onBufferOverflow) {
    ...
    override suspend fun collectTo(scope: ProducerScope<T>) = block(scope)
}
```

@稀土掘金技术社区

`ChannelFlowBuilder` 并没有做什么，只是将挂起函数 `collectTo` 实现为了外部传入的函数类型。

与前面解析的 `flowOn` 操作符一样，只是将原本由接收上游数据流的数据，变为由外部手动控制 `Channel` 添加数据进入缓冲队列的逻辑。

那么 `callbackFlow` 创建的 `CallBackFlowBuilder` 又是如何对 `Channel` 的关闭进行强制检测呢？

```

private class CallBackFlowBuilder<T>(
    private val block: suspend ProducerScope<T>.() -> Unit,
    context: CoroutineContext = EmptyCoroutineContext,
    capacity: Int = BUFFERED,
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND
) : ChannelFlowBuilder<T>(block, context, capacity, onBufferOverflow) {

    override suspend fun collectTo(scope: ProducerScope<T>) {
        super.collectTo(scope) //外部lambda代码块
        if (!scope.isClosedForSend) { //lambda执行完后，还没有调用close，就会抛出异常
            throw IllegalStateException(...)
        }
    }
    ...
}

```

@稀土掘金技术社区

## » 小结

相比 `flow` 构建数据流， `channelFlow` 是基于 `Channel` 来发送数据的，是线程安全的，允许在发送数据时切换协程上下文，同时还能使用 `Flow` 的操作符。

如果回调API是通过类似注册的方式进行添加，需要在最后调用 `awaitClose` 函数中进行反注册，避免内存泄露，同时一直等待回调返回数据。

所以为了避免发送完后忘记 `close`，造成内存泄漏，更推荐使用 `callBackFlow`。

虽然 `channelFlow` 与 `callBackFlow` 这两个函数已经转正，但其内部还是有实验性质api，直接使用还是会警告。

需要添加 `@OptIn(ExperimentalCoroutinesApi::class)` 注解标记。

## 使用实验阶段API

使用Kotlin协程中的那些还处于实验阶段或者预览阶段的API时，IDE都会有警告，提示这是个不稳定API，后续可能会被修改。

让开发者在每个调用API的函数上都要添加 `@ExperimentalCoroutinesApi` 或者 `@FlowPreview` 注解。但这样难免有些麻烦，有没有什么一劳永逸的偷懒办法呢？

其实可以在需要调用这些API的模块（module）内的 `build.gradle` 文件中添加

```
//module build.gradle
```

arduino 复制代码

```
android {  
    ...  
    kotlinOptions {  
        jvmTarget = '1.8'  
        freeCompilerArgs += [  
            "-Xuse-experimental=kotlinx.coroutines.ExperimentalCoroutinesApi",  
            "-Xuse-experimental=kotlinx.coroutines.FlowPreview"  
            "-Xopt-in=kotlin.RequiresOptIn"  
        ]  
    }  
}
```

重新编译后，就能直接使用实验阶段与预览阶段API了。

如果后续版本升级后，API转为了 `@ObsoleteCoroutinesApi` 注解的废弃函数或类，依然会有提示，利用IDE的提示替换为等价操作即可。

## 总结

虽然 `Channel` 的出现，为协程间通信提供了相当方便的工具。随着 `Kotlin Flow` 问世之后，`Channel` 就迅速转为幕后，连旧版本中的操作符以及 `BroadcastChannel` 也都被废弃。

在 `Flow` 中的许多功能内部实现有不少 `Channel` 的身影，其本身的职责越发单一，仅作为 **协程间通信的并发安全的缓冲队列** 而存在。

对于大部分场景而言更多还是推荐使用 `Flow`，并**不推荐**直接使用 `Channel`。

## 参考资料

[kotlin 协程官方文档\(6\)Channel](#)

[select 表达式](#)

[使用协程和 Flow 简化 API 设计](#)