

31 | 图解Channel：如何理解它的CSP通信模型？

2022-04-04 朱涛

《朱涛·Kotlin编程第一课》

[课程介绍 >](#)



讲述：朱涛

时长 13:09 大小 12.05M

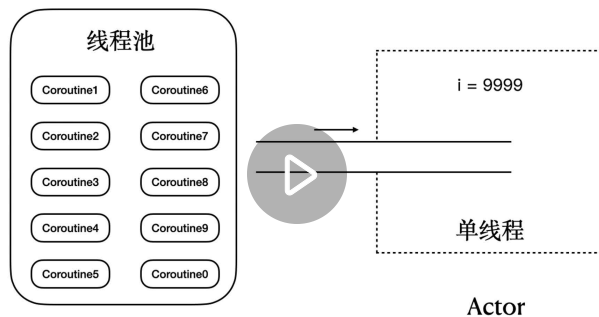


你好，我是朱涛。今天我们来分析 Channel 的源码。

Kotlin 的 Channel 是一个非常重要的组件，在它出现之前，协程之间很难进行通信，有了它以后，协程之间的通信就轻而易举了。在 [第 22 讲](#) 当中，我们甚至还借助 Channel 实现的 Actor 做到了并发安全。

那么总的来说，Channel 是热的，同时它还是一个**线程安全的数据管道**。而由于 Channel 具有线程安全的特性，因此，它最常见的用法，就是建立 CSP 通信模型（Communicating Sequential Processes）。

不过你可能会觉得，CSP 太抽象了不好理解，但其实，这个通信模型我们在第 22 讲里就接触过了。当时我们虽然是通过 Actor 来实现的，但却是把它当作 CSP 在用，它们两者的差异其实很小。



关于 **CSP 的理论**，它的精确定义其实比较复杂，不过它的核心理念用一句话就可以概括：**不要共享内存来通信；而是要用通信来共享内存**（Don't communicate by sharing memory; share memory by communicating）。

可是，我们为什么可以通过 Channel 实现 CSP 通信模型呢？这背后的技术细节，则需要我们通过源码来发掘了。

Channel 背后的数据结构

为了研究 Channel 的源代码，我们仍然是以一个简单的 Demo 为例，来跟踪它的代码执行流程。

[复制代码](#)

```
1 // 代码段1
2
3 fun main() {
4     val scope = CoroutineScope(Job() + mySingleDispatcher)
5     // 1, 创建管道
6     val channel = Channel<Int>()
7
8     scope.launch {
9         // 2, 在一个单独的协程当中发送管道消息
10        repeat(3) {
11            channel.send(it)
12            println("Send: $it")
13        }
14
15        channel.close()
16    }
17
18    scope.launch {
19        // 3, 在一个单独的协程当中接收管道消息
20        repeat(3) {
21            val result = channel.receive()
22            println("Receive ${result}")
```

```

23     }
24 }
25
26 println("end")
27 Thread.sleep(20000000L)
28 }
29
30 /*
31 输出结果:
32 end
33 Receive 0
34 Send: 0
35 Send: 1
36 Receive 1
37 Receive 2
38 Send: 2
39 */

```

以上代码主要分为三个部分，分别是：**Channel 创建**、**发送数据**、**接收数据**。

我们先来分析注释 1 处的 **Channel** 创建逻辑。我们都知道 **Channel** 其实是一个接口，它是通过组合 **SendChannel**、**ReceiveChannel** 得来的。而注释 1 处调用的 **Channel()**，其实是一个普通的顶层函数，只是**它发挥的作用是构造函数，因此它的首字母是大写的**，这跟我们上节课分析的 **CoroutineScope**、**Job** 也是类似的。

 复制代码

```

1 // 代码段2
2
3 public interface Channel<E> : SendChannel<E>, ReceiveChannel<E> {
4
5     public fun <E> Channel(
6         capacity: Int = RENDEZVOUS,
7         onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND,
8         onUndeliveredElement: ((E) -> Unit)? = null
9     ): Channel<E> =
10         when (capacity) {
11             RENDEZVOUS -> {
12                 if (onBufferOverflow == BufferOverflow.SUSPEND)
13                     RendezvousChannel(onUndeliveredElement)
14                 else
15                     ArrayChannel(1, onBufferOverflow, onUndeliveredElement)
16             }
17             CONFLATED -> {
18                 ConflatedChannel(onUndeliveredElement)
19             }
20             UNLIMITED -> LinkedListChannel(onUndeliveredElement)
21             BUFFERED -> ArrayChannel(

```

```

22         if (onBufferOverflow == BufferOverflow.SUSPEND) CHANNEL_DEFAULT_CAP
23             onBufferOverflow, onUndeliveredElement
24     )
25     else -> {
26         if (capacity == 1 && onBufferOverflow == BufferOverflow.DROP_OLDEST
27             ConflatedChannel(onUndeliveredElement)
28         else
29             ArrayChannel(capacity, onBufferOverflow, onUndeliveredElement)
30     }
31 }

```

然后，从上面的代码里，我们可以看到，**Channel()** 方法的核心逻辑就是一个 **when** 表达式，它根据传入的参数，会创建不同类型的 Channel 实例，包括了：**RendezvousChannel**、**ArrayChannel**、**ConflatedChannel**、**LinkedListChannel**。而这些实现类都有一个共同的父类：**AbstractChannel**。

 复制代码

```

1 // 代码段3
2
3 internal abstract class AbstractSendChannel<E>(
4     @JvmField protected val onUndeliveredElement: OnUndeliveredElement<E>?
5 ) : SendChannel<E> {
6
7     protected val queue = LockFreeLinkedListHead()
8
9     // 省略
10
11     internal abstract class AbstractChannel<E>(
12         onUndeliveredElement: OnUndeliveredElement<E>?
13     ) : AbstractSendChannel<E>(onUndeliveredElement), Channel<E> {}
14 }

```

可以看到，**AbstractChannel** 其实是 **AbstractSendChannel** 的内部类，同时它也是 **AbstractSendChannel** 的子类。而 **Channel** 当中的核心逻辑，都是依靠 **AbstractSendChannel** 当中的 **LockFreeLinkedListHead** 实现的。我们接着来看下它的源代码：

 复制代码

```

1 // 代码段4
2
3 public actual open class LockFreeLinkedListHead : LockFreeLinkedListNode() {
4     public actual val isEmpty: Boolean get() = next === this
5 }

```

```

6 public actual open class LockFreeLinkedListNode {
7     // 1
8     private val _next = atomic<Any>(this)
9     private val _prev = atomic(this)
10    private val _removedRef = atomic<Removed?>(null)
11 }
12

```

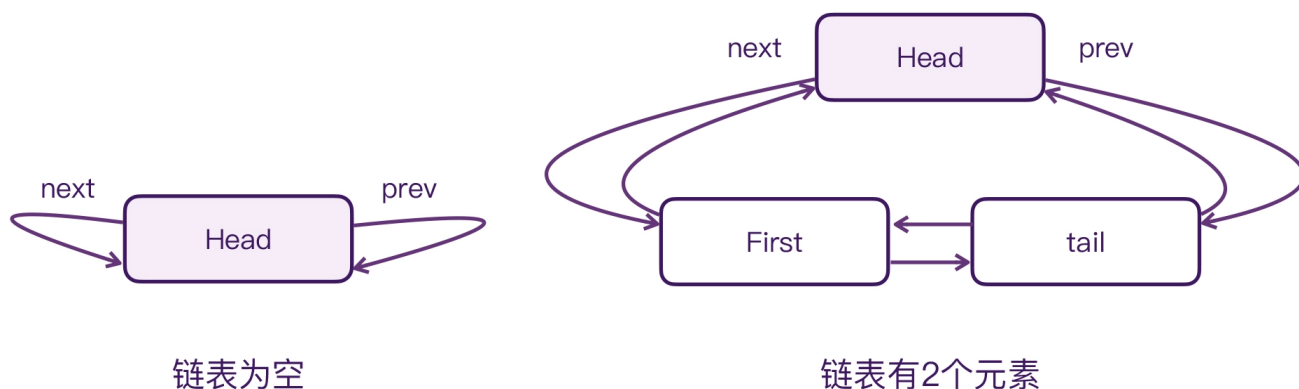
可见，LockFreeLinkedListHead 其实继承自 **LockFreeLinkedListNode**，而 LockFreeLinkedListNode 则是实现 Channel 核心功能的关键数据结构。整个数据结构的核心思想，来自于 2004 年的一篇论文：[《Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap》](#)。如果你对其中的原理感兴趣，可以去看看这篇论文。这里，为了不偏离主题，我们只分析它的核心思想。

LockFreeLinkedListNode，我们可以将其区分开来看待，即 LockFree 和 LinkedList。

第一个部分：**LockFree**，它是通过 [CAS](#)（Compare And Swap）的思想来实现的，比如 JDK 提供的 `java.util.concurrent.atomic`。这一点，我们从上面注释 1 的 `atomic` 也可以看出来。

第二个部分：**LinkedList**，这说明 LockFreeLinkedList 本质上还是一个**链表**。简单来说，它其实是一个循环双向链表，而 LockFreeLinkedListHead 其实是一个**哨兵节点**，如果你熟悉链表这个数据结构，也可以将其看作是链表当中的 [虚拟头结点](#)，这个节点本身不会用于存储任何数据，它的 `next` 指针会指向整个链表的**头节点**，而它的 `prev` 指针会指向整个链表的**尾节点**。

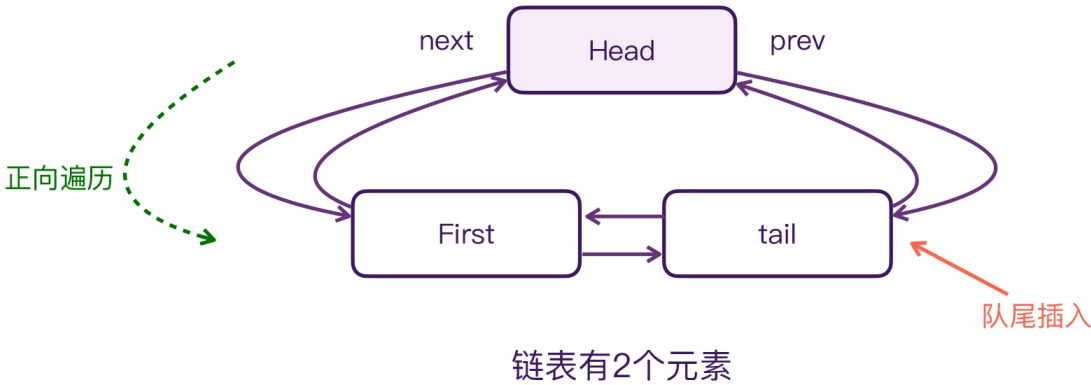
为了方便你理解，我画了一张图描述这个链表的结构：



请看图片左边的部分，**当链表为空的时候**，LockFreeLinkedListHead 的 next 指针和 prev 指针，都是指向自身的。这也就意味着，这个 Head 节点是不会存储数据，同时，也是不会被删除的。

然后再看图片右边的部分，**当链表有 2 个元素的时候**，这时 LockFreeLinkedListHead 节点的 next 指针才是第一个节点，而 Head 的 prev 指针则是指向尾结点。

实际上，寻常的循环双向链表是可以在首尾添加元素的，同时也支持“正向遍历、逆向遍历”的。但 Channel 内部的这个数据结构只能在末尾添加，而它遍历的顺序则是从队首开始的。这样的设计，就让它的行为在变成了先进先出**单向队列**的同时，还实现了队尾添加操作，只需要 $O(1)$ 的时间复杂度。



可以说，正是因为 LockFreeLinkedList 这个数据结构，我们才能使用 Channel 实现 CSP 通信模型。

好，在弄清楚 LockFreeLinkedList 这个数据结构以后，Channel 后续的源码分析就很简单了。让我们来分别分析一下 Channel 的 send()、receive() 的流程。

发送和接收的流程

我们回过头来看代码段 1 当中的逻辑，我们分别启动了两个协程，在这两个协程中，我们分别发送了三次数据，也接收了三次数据。程序首先会执行 send()，由于 Channel 在默认情况下容量是 0，所以，send() 首先会被挂起。让我们来看看这部分的逻辑：

复制代码

```
1 // 代码段5
2
```



```

3 public final override suspend fun send(element: E) {
4     // 1
5     if (offerInternal(element) === OFFER_SUCCESS) return
6     // 2
7     return sendSuspend(element)
8 }
9
10 protected open fun offerInternal(element: E): Any {
11     while (true) {
12         // 3
13         val receive = takeFirstReceiveOrPeekClosed() ?: return OFFER_FAIL
14         // 省略
15     }
16 }
17
18 private suspend fun sendSuspend(element: E): Unit = suspendCancellableCoroutine
19     loop@ while (true) {
20         if (isFullImpl) {
21             // 4
22             val send = if (onUndeliveredElement == null)
23                 SendElement(element, cont) else
24                 SendElementWithUndeliveredHandler(element, cont, onUndeliveredE
25             val enqueueResult = enqueueSend(send)
26             when {
27                 enqueueResult == null -> {
28                     // 5
29                     cont.removeOnCancellation(send)
30                     return@sc
31                 }
32                 enqueueResult is Closed<*> -> {
33                 }
34                 enqueueResult === ENQUEUE_FAILED -> {}
35                 enqueueResult is Receive<*> -> {}
36                 else -> error("enqueueSend returned $enqueueResult")
37             }
38         }
39         // 省略
40     }
41 }

```

上面的挂起函数 `send()` 分为两个部分：

- 注释 1，尝试向 Channel 发送数据，如果这时候 Channel 已经有了消费者，那么 if 就会为 true，`send()` 方法就会 return。不过，按照代码段 1 的逻辑，首次调用 `send()` 的时候，Channel 还不存在消费者，因此在注释 3 处，尝试从 `LockFreeLinkedList` 取出消费者是不可能的。所以，程序会继续执行注释 2 处的逻辑。

- 注释 2，会调用挂起函数 `sendSuspend()`，它是由高阶函数 `suspendCancellableCoroutineReusable{}` 实现的。我们看它的名字就能知道，它跟 `suspendCancellableCoroutine{}` 是类似的（如果你有些忘了，可以回过头去看看 [🍷加餐五](#)）。另外，请留意下这个方法的注释 4，它会将发送的元素封装成 `SendElement` 对象，然后调用 `enqueueSend()` 方法，将其添加到 `LockFreeLinkedList` 这个队列的末尾。如果 `enqueueSend()` 执行成功了，就会执行注释 5，注册一个回调，用于将 `SendElement` 从队列中移除掉。

如果你足够细心的话，你会发现这整个流程并没有涉及到 `resume` 的调用，因此，这也意味着 `sendSuspend()` 会一直被挂起，而这就意味着 `send()` 会一直被挂起！那么，问题来了，**`send()` 会在什么时候被恢复？**

答案当然是：**`receive()` 被调用的时候！**

 复制代码

```
1 // 代码段6
2
3 public final override suspend fun receive(): E {
4     // 1
5     val result = pollInternal()
6
7     @Suppress("UNCHECKED_CAST")
8     if (result !== POLL_FAILED && result !is Closed<*>) return result as E
9     // 2
10    return receiveSuspend(RECEIVE_THROWS_ON_CLOSE)
11 }
12
13 protected open fun pollInternal(): Any? {
14     while (true) {
15         // 3
16         val send = takeFirstSendOrPeekClosed() ?: return POLL_FAILED
17         val token = send.tryResumeSend(null)
18         if (token != null) {
19             assert { token === RESUME_TOKEN }
20             //4
21             send.completeResumeSend()
22             return send.pollResult
23         }
24
25         send.undeliveredElement()
26     }
27 }
28
29 // CancellableContinuationImpl
30 private fun dispatchResume(mode: Int) {
```



```

31     if (tryResume()) return
32     // 5
33     dispatch(mode)
34 }
35
36 internal fun <T> DispatchedTask<T>.dispatch(mode: Int) {
37     // 省略
38     if (!undispatched && delegate is DispatchedContinuation<*> && mode.isCancel
39
40         val dispatcher = delegate.dispatcher
41         val context = delegate.context
42         if (dispatcher.isDispatchNeeded(context)) {
43             // 6
44             dispatcher.dispatch(context, this)
45         } else {
46             resumeUnconfined()
47         }
48     } else {
49         // 省略
50     }
51 }

```

可以看到，挂起函数 `receive()` 的逻辑，跟代码段 5 当中的 `send()` 是类似的。

- 注释 1，尝试从 `LockFree` 队列当中找出是否有正在被挂起的**发送方**。具体的逻辑在注释 3 处，它会从队首开始遍历，寻找 `Send` 节点。
- 接着上面的代码段 1 的案例分析，此时我们一定是可以从队列中找到一个 `Send` 节点的，因此程序会继续执行注释 4 处的代码。
- 注释 4，`completeResumeSend()`，它最终会调用注释 5 处的 `dispatch(mode)`，而 `dispatch(mode)` 其实就是 `DispatchedTask` 的 `dispatch()`，是不是觉得很熟悉？这个 `DispatchedTask` 其实就是我们在 [第 29 讲](#)当中分析过的 `DispatchedTask`，这里的 `dispatch()` 就是协程体当中的代码在线程执行的时机。最终，它会执行在 Java 的 `Executor` 之上。至此，我们之前被挂起的 `send()` 方法，其实就算是恢复了。

另外，你可以再留意上面的注释 2，当 `LockFree` 队列当中没有正在挂起的发送方时，它会执行 `receiveSuspend()`，而 `receiveSuspend()` 也同样会被挂起：

 复制代码

```

1 private suspend fun <R> receiveSuspend(receiveMode: Int): R = suspendCancellable
2     val receive = if (onUndeliveredElement == null)
3         ReceiveElement(cont as CancellableContinuation<Any?>, receiveMode) else
4         ReceiveElementWithUndeliveredHandler(cont as CancellableContinuation<An

```

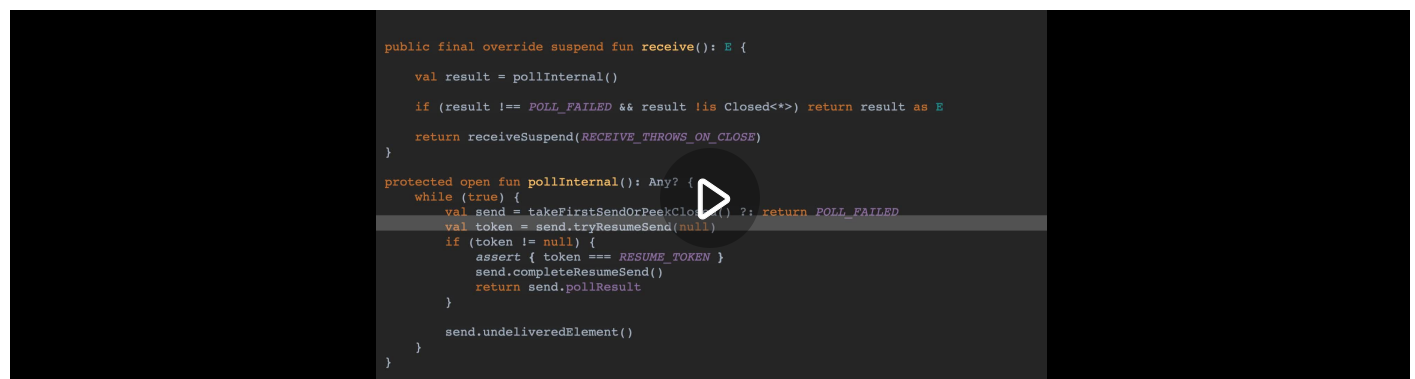
```

5     while (true) {
6         if (enqueueReceive(receive)) {
7             removeReceiveOnCancel(cont, receive)
8             return@sc
9         }
10
11        val result = pollInternal()
12        if (result is Closed<*>) {
13            receive.resumeReceiveClosed(result)
14            return@sc
15        }
16        if (result != POLL_FAILED) {
17            cont.resume(receive.resumeValue(result as E), receive.resumeOnCancel)
18            return@sc
19        }
20    }
21 }

```

所以，这里的逻辑其实跟之前的 `sendSuspend()` 是类似的。首先，它会封装一个 `ReceiveElement` 对象，并且将其添加到 `LockFree` 队列的末尾，如果添加成功的话，这个 `receiveSuspend` 就会继续挂起，这就意味着 `receive()` 也会被挂起。而 `receive()` 被恢复的时机，其实就对应了代码段 5 当中注释 1 的代码：`offerInternal(element)`。

至此，`Channel` 的发送和接收流程，我们就都已经分析完了。按照惯例，我们还是通过一个视频来回顾代码的整体执行流程：



小结

通过这节课，我们知道，`Channel` 其实是一个线程安全的管道。它最常见的用法，就是实现 CSP 通信模型。它的核心理念是：**不要共享内存来通信；而是要用通信来共享内存**。而 `Channel` 之所以可以用来实现 CSP 通信模型，主要还是因为它底层用到的数据结构：`LockFreeLinkedList`。

LockFreeLinkedList 虽然是一个循环双向链表，但在 Channel 的源码中，它会被当做**先进先出**的单向队列，它只在队列末尾插入节点，而遍历则只正向遍历。

还有 Channel 的 `send()`，它会分为两种情况，一种是当前的 LockFree 队列当中已经有被挂起的**接收方**，这时候，`send()` 会恢复 Receive 节点的执行，并且将数据发送给对方。第二种情况是：当前队列当中没有被挂起的接收方，这时候 `send()` 就会被挂起，而被发送的数据会被封装成 `SendElement` 对象插入到队列的末尾，等待被下次的 `receive()` 恢复执行。

而 Channel 的 `receive()`，也是分为两种情况，一种是当前的 LockFree 队列当中已经存在被挂起的**发送方**，这时候 `receive()` 会恢复 Send 节点的执行，并且取出 Send 节点当中带过来的数据。第二种情况是：当前队列没有被挂起的发送方，这时候 `receive()` 就会被挂起，同时它也会被封装成一个 `ReceiveElement` 对象插入到队列的末尾，等待被下次的 `send()` 恢复执行。

其实，Kotlin 推崇 CSP 模型进行并发的原因还有很多，比如门槛低、可读性高、扩展性好，还有一点是会被很多人提到的：不容易发生死锁。

不过，这里需要特别注意的是，CSP 场景下的并发模型，并非不可能发生死锁，在一些特殊场景下，它也是可能发生死锁的，比如：通信死锁（Communication Deadlock）。因此，CSP 也并不是解决所有并发问题的万能解药，我们还是要具体问题具体分析。

思考题

在课程的开头，我们分析了 Channel 一共有四种实现方式：RendezvousChannel、ArrayChannel、ConflatedChannel、LinkedListChannel，请问你能结合今天学习的知识，分析 LinkedListChannel 的原理吗？

 复制代码

```
1 internal open class LinkedListChannel<E>(onUndeliveredElement: OnUndeliveredEle
2     protected final override val isBufferAlwaysEmpty: Boolean get() = true
3     protected final override val isBufferEmpty: Boolean get() = true
4     protected final override val isBufferAlwaysFull: Boolean get() = false
5     protected final override val isBufferFull: Boolean get() = false
6
7     protected override fun offerInternal(element: E): Any {
8         while (true) {
9             val result = super.offerInternal(element)
10            when {
11                result === OFFER_SUCCESS -> return OFFER_SUCCESS
```


```

12         result === OFFER_FAILED -> { // try to buffer
13             when (val sendResult = sendBuffered(element)) {
14                 null -> return OFFER_SUCCESS
15                 is Closed<*> -> return sendResult
16             }
17             // otherwise there was receiver in queue, retry super.offer
18         }
19         result is Closed<*> -> return result
20         else -> error("Invalid offerInternal result $result")
21     }
22 }
23 }
24
25 protected override fun offerSelectInternal(element: E, select: SelectInstan
26     while (true) {
27         val result = if (hasReceiveOrClosed)
28             super.offerSelectInternal(element, select) else
29             (select.performAtomicTrySelect(describeSendBuffered(element)) ?
30         when {
31             result === ALREADY_SELECTED -> return ALREADY_SELECTED
32             result === OFFER_SUCCESS -> return OFFER_SUCCESS
33             result === OFFER_FAILED -> {} // retry
34             result === RETRY_ATOMIC -> {} // retry
35             result is Closed<*> -> return result
36             else -> error("Invalid result $result")
37         }
38     }
39 }
40 }

```

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | CoroutineScope是如何管理协程的？

下一篇 32 | 图解Flow：原来你是只纸老虎？



Paul Shan

2022-04-04

思考题: `LinkedListChannel.offerInternal`调用`AbstractSendChannel.offerInternal`失败的时候, 会把发送的内容持续放到队列中, 这样即使接受方没准备好或者不存在, 发送方也不会等待, 而持续进入可以接收数据并发送的状态。`LinkedListChannel.offerSelectInternal`调用`AbstractSendChannel.offerSelectInternal`失败的时候, 还是会继续尝试调用这个方法, 因为`LinkedListChannel`只要内存允许, 会时刻处于接受数据的状态。

作者回复: 很棒的答案~推荐给大家。

