The Source type is parameterized with two types:

- the first one is the type of element that this source emits
- the second one may signal that running the source produces some auxiliary value (e.g. a network source may provide information about the bound port or the peers address).
  Where no auxiliary information is produced, the type akka.NotUsed is used

the Source is just a description of what you want to run
this is just a description of what we want to have computed once we run the stream.

The Materializer is a factory for stream execution engines,
it is the thing that makes streams run ---- you don t need to worry about any of the details just now apart from that you need one for calling any of the run methods on a Source.

This stream is run by attaching a file as the receiver of the data. In the terminology of Akka Streams this is called a Sink.

```
def lineSink(filename: String): Sink[String, Future[IOResult]] =
  Flow[String]
    .map(s    6 0 ByteString(s + "\n"))
    .toMat(FileIO.toPath(Paths.get(filename)))(Keep.right)
```

Sink[String, Future[IOResult]] means that it accepts strings as its input and when materialized it will create auxiliary information of type Future[IOResult].
when chaining operations on a Source or Flow, the type of the auxiliary information is given by the leftmost starting point; since we want to retain what the FileIO.toPath sink has to offer, we need to say Keep.right

IOResult is a type that IO operations return in Akka Streams in order to tell you how many bytes or elements were consumed and whether the stream terminated normally or exceptionally.

Akka Stream 背后的设计原则

- all features are explicit in the API, no magic
- supreme(最高的) compositionality: combined pieces retain the function of each part
- exhaustive(详尽的) model of the domain of distributed bounded stream processing

核心概念：

Akka Streams is a library to process and transfer a sequence of elements using bounded buffer space.

Akka Stream 的几个术语：

Stream：An active process that involves moving and transforming data.

Element

An element is the processing unit(处理单元) of streams.

All operations transform and transfer elements from upstream to downstream.

Buffer sizes are always expressed as number of elements independently from the actual size of the elements.

Graph：流处理拓扑的描述，定义当 stream running 时 element 的流向的路径。

一、定义和运行 stream

1、Akka Stream 常见的构建组件

- Source: something with exactly one output stream
- Sink: something with exactly one input stream
- Flow: something with exactly one input and one output stream
- BidiFlow: something with exactly two input streams and two output streams，在概念的行为上就像两个方向相反的 Flow。
- Graph: a packaged stream processing topology that exposes a certain set of input and output ports, characterized by an object of type Shape.

在 akka stream 中使用如下核心抽象描述“线性处理管道”：

- Source：只有一个输出的处理状态，无论下游处理状态是否做好接收准备，(Source)都会发射数据元素。
- Sink：只有一个输入的处理状态，请求和接收元素可能减缓上流生产者生产元素
- Flow：只有一个输入和一个输出的处理状态，通过它转化数据元素将连接它的上游和下游。
- RunnableGraph：两端分别连接 Source 和 Sink 的 Flow，并且准备被调用 run()方法.

It is possible to attach a **Flow** to a Source resulting in a **composite source**,

and it is also possible to prepend a **Flow** to a **Sink** to get a **new sink**.

After a stream is properly terminated by having both a **source** and a **sink**, it will be represented by the **RunnableGraph** type, indicating(表明) that it is ready to be executed.

记住即使通过连接所有的 soure、sink 以及不同的处理状态来构造 RunnableGraph，在它被物化之前是不会有数据流过它的，这一点是很重要的。

```
val source = Source(1 to 10)
val sink = Sink.fold[Int, Int](0)(_ + _)
// connect the Source to the Sink, obtaining a RunnableGraph
val runnable: RunnableGraph[Future[Int]] = source.toMat(sink)(Keep.right)
// materialize the flow and get the value of the FoldSink
val sum: Future[Int] = runnable.run()
```

当运行(物化)RunnableGraph 之后，我们得到了一个 T 类型的物化值。

每一个 stream processing stage 都会提供一个物化的值，并且它是用户的职责来组合它们得到一个新的类型。

在上面的例子中，我们使用 toMat 方法来表明我们想要转换 source 和 sink 的物化值，并且用 Keep.right 函数来说明我们仅仅对 sink 的物化值感兴趣。

通常来说，一个 stream 会暴露多个物化值，但是相当常见的情况是我们仅仅对 source 或者 sink 的物化值感兴趣。
因为这个原因，Akka Stream 提供了一个非常方便的方法：runWith()，可以适用于 sink、source、flow。
runWith 既会物化 stream，同时也会返回给定 sink 或者 source 的物化值(不像上面的例子，需要执行 toMat 和 run)。

```
val source = Source(1 to 10)
val sink = Sink.fold[Int, Int](0)(_ + _)
// materialize the flow, getting the Sinks materialized value
val sum: Future[Int] = source.runWith(sink)
```

需要指出的是：因为 processing stages 是不可变的，连接它们将返回一个新的 processing stage，而不是修改它们。所以，当构造一个较长的 flow 时，一定要记得分配一个新的值给一个变量或者运行它。

```
val source = Source(1 to 10)
source.map(_ ⇒ 0)                  // 原来的 source 没有任何影响，因为它是不可变的
source.runWith(Sink.fold(0)(_ + _)) // 55
val zeroes = source.map(_ ⇒ 0)     // returns new Source[Int], with `map()` appended
zeroes.runWith(Sink.fold(0)(_ + _)) // 0
```

既然一个 stream 能被物化多次，物化值也会在每次物化时重新计算，这将导致每次返回不同的值。

```
// connect the Source to the Sink, obtaining a RunnableGraph
val sink = Sink.fold[Int, Int](0)(_ + _)
val runnable: RunnableGraph[Future[Int]] = Source(1 to 10).toMat(sink)(Keep.right)
// get the materialized value of the FoldSink
val sum1: Future[Int] = runnable.run()
val sum2: Future[Int] = runnable.run()
// sum1 and sum2 are different Futures!
```

2、定义 Source、Sink、Flow　　*

单例对象 Source 和 Sink 提供了多种方式来创建 source 和 sink。下面展示了一个最有用的构造方式：

(1) 创建 Source 的内建方式

方法	作用	代码示例	何时完成
fromIterator	1. 发射的元素来自一个 Iterable。 2. 在每次物化时都会重新创建一个新的 Iterable。这也就是为什么这个方法接受一个函数而不是直接接受一个 Iterable 的原因。		当 Iterable 中元素发射完时结束。
apply	发射的元素来自一个 immutable.Seq	Source(List(1, 2, 3))	当 seq 中的所有元素发射完时，Stream 会 complete。
single	这个 Source 只会发射一个元素	Source.single("only one element")	一旦这个元素发射完，这个 Source 就 complete 了。
repeat	重复地发射一个单独的元素		从来不会 complete
tick	定期重复地发射一个任意元素。 第一次 tick 的延迟和下一次 tick 的间隔被单独地指定。		从来不会 complete

fromFuture	当 Future complete 时，发射一个来自 Future 的单独的值。	Source.fromFuture(Future.successful("Hello Streams!"))	当 Future complete 时，这个 Source 也 complete
fromCompletionStage	当 CompletionStage complete 时，发射一个来自 CompletionStage 的单独的值。		当 Future complete 时，这个 Source 也 complete
fromFutureSource	发射的元素来自给定的 Future， 一旦这个 Future 成功的 complete 了，则发射 Future 中的值；如果 Future 失败了，则 Stream 也会失败。		当 Future source complete 时，这个 Source 也 complete
fromSourceCompletionStage	Streams the elements of an asynchronous source once its given completion stage completes. If the completion fails the stream is failed with that exception.		当异步的 source complete 时，这个 source 也会 complete
empty	创建的 Source 不会发射任何元素	Source.empty	不发射任何元素，直接 complete
maybe	物化一个 Promise[Option[T]]： 如果它以一个 Some[T]完成了，则发射元素 T，并且 complete 这个 stream； 如果它以 None 完成了，则直接 complete 这个 stream。		
failed	Fail directly with a user specified exception.		
lazlly	延迟一个 Source 的创建和物化。 直到下游有请求时。		
actorPublisher	包装一个继承自 ActorPublisher 的 actor 作为 Source。 发射的元素依赖 actor 的实现。		当 actor 终止(stop)时，complete 这个 stream。
actorRef	具体看“与 Actor 集成”一节		
range	发射在一个范围内的整型元素。 有一个可选参数，控制发射的步长		当到达范围的结尾时，则 complete stream。
combine	Combine several sources, using a given strategy such as merge or concat, into one source.		当所有的 source 都已经 complete 时，则 complete 这个 Stream。
queue	具体看“与 Actor 集成”一节		当下游 complete 时，complete 这个 Stream。
zipN	Combine the elements of multiple streams into a stream of sequences.		当任何一个上游 complete 时，则 complete 这个 Stream。
zipWithN	Combine the elements of multiple streams into a stream of sequences using a combiner function.		当任何一个上游 complete 时，则 complete 这个 Stream。

(2) 创建 Sink 的内建方式

方法	作用	代码示例	背压 backpressure
head	返回的物化值是一个 Future， Future 包装的值是第一个到达 sink 的元素， 然后这个 Stream 被 canceld。 如果没有元素发射，则返回的 Future 是 failed。	Sink.head	没有背压
headOption	返回的物化值是一个 Future[Option[T]]， 第一个到达 sink 的元素将会被包装在 Some 中， 如果没有元素发射，则 Future 包装的是 None		没有背压
last	返回的物化值是 Future， 最后一个到达 sink 的元素将会被包装在 Future 中， 如果没有元素发射，则这个 Future 是 failed。		没有背压
lastOption	返回的物化值是一个 Future[Option[T]]， 最后一个到达 sink 的元素将会被包装在 Some 中， 如果没有元素发射，则 Future 包装的是 None		
ignore	接收 stream 的所有元素，但是丢弃它们。	Sink.ignore	没有背压
cancelled	立即 cancel 这个 Stream		
seq	将 stream 发射的元素收集到一个集合中。 注意：这个集合的大小是 Int.MaxValue， 如果元素数量超过了，则这个 sink 会 cancel 这个 Stream。		
foreach	对每一个收到的元素执行一个程序。 返回的物化值是一个 Future[Option[Done]]	Sink.foreach[String](println(_))	当 foreach 给定的程序还未执行完时，会产生背压
foreachParallel	和 foreach 类似，但是运行并行调用给定的程序。		当 foreachParallel 给定的程序还未执行完时，会产生背压
onComplete	当 Stream 正常 complete，或者发生 failed 时，执行一个回调函数。		没有背压
fold	folds over the stream and returns a Future of the final result as its materialized value	Sink.fold[Int, Int](0)(_ + _)	当 fold 提供的函数还未执行完时，会产生背压
combine	使用用户指定的策略，将多个 sinks 合并成一个		依赖于使用的策略
actorRef	将 Stream 的元素发射给一个 ActorRef。 当 actor 终止时，会 cancel 这个 Stream		没有背压
actorRefWithAck	看“与 Actor 集成”一节		

(3) Source、Sink 和 InputStream、OutputStream 集成

也可以将 Source、Sink 和 java.io.InputStream、java.io.OutputStream 集成在一起。
因为 InputStream 和 OutputStream 的 API 都是阻塞的，所以相应的 source 和 sink 应该运行在单独的 dispatcher 上。
可以通过 akka.stream.blocking-io-dispatcher 进行配置。

fromOutputStream
创建一个包装了 OutputStream 的 Sink。
fronOutputStream 接收一个产生 OutputStream 的函数，当 sink 被物化时，这个函数会被调用。
发送到 sink 的字节将会写到 OutputStream 中。

Sink 返回的物化值是一个包装了 IOResult 的 Future。

当 Sink 所在 Stream 完成时，OutputStream 将会被 closed。

fromInputStream
创建一个包装了 InputStream 的 Source。
fromInputStream 接收一个产生 InputStream 的函数，当 Source 被物化时，这个函数会被调用
来自 InputStream 的字节将会发射给 Stream。

Source 返回的物化值是一个包装了 IOResult 的 Future。

当 Source 被它的下游取消时，InputStream 将会被 closed。
当 InputStream 的数据读取结束时，将会 complete Source。

(4) File IO 类型的 Source、Sink

Sources and sinks for reading and writing files can be found on FileIO。

fromPath
创建一个 Source，将一个 file 中的内容当做 ByteString 进行发射。
返回的物化值是一个包装了 IOResult 的 Future。

toPath
创建一个 Sink，将接收到的 ByteString 写到一个给定的 file path。

(5) 通过组合 Source 和 Sink 来创建 Flow

Flow.fromSinkAndSource

从一个 Source 和 Sink 来创建一个 Flow，Flow 的输入来自 Source，Flow 的输出将会发送到 Sink。

Flow.fromSinkAndSourceCoupled

3、有多种方式可以连接一个 stream 的不同部分 *

(1) 首先看一些示例

```
// 显示地创建并且连接 Source、Sink 和 Flow
// By default, the materialized value of the leftmost(最左边) stage is preserved
Source(1 to 6).via(Flow[Int].map(_ * 2)).to(Sink.foreach(println(_)))

// Starting from a Source
val source = Source(1 to 6).map(_ * 2)      // Source 支持 map 操作
source.to(Sink.foreach(println(_)))

// Starting from a Sink
val sink: Sink[Int, NotUsed] = Flow[Int].map(_ * 2).to(Sink.foreach(println(_)))
Source(1 to 6).to(sink)

// Broadcast to a sink inline
val otherSink: Sink[Int, NotUsed] = Flow[Int].alsoTo(Sink.foreach(println(_))).to(Sink.ignore)
Source(1 to 6).to(otherSink)
```

(2) 简单的 processing stages

方法	作用	背压	何时完成
alsoTo	将指定的 Sink 附着在 Flow 上。 意味着流过 Flow 的元素也会被发送到 Sink。	当下游或者 Sink 产生背压时	当上游 complete 时
map	将一个函数映射到 Stream 发射的每一个元素上。 函数返回的新结果将会被传递到下游	当下游产生背压时	当上游 complete 时
mapConcat	将一个元素转换成 0 个或多个元素。 转换后的多个元素将会单独地传递到下游。	当下游产生背压，或者上次计算的集合中仍有可用的元素时	当上游 complete，并且所有剩余的元素已经被发射时
statefulMapConcat			
filter	用一个 predicate 对收到的元素进行过滤。 如果 predicate 返回 true，则元素可以传递到下游； 若返回 false，则元素被丢弃。		当上游 complete 时
filterNot	用一个 predicate 对收到的元素进行过滤。 如果 predicate 返回 false，则元素可以传递到下游； 若返回 true，则元素被丢弃。		当上游 complete 时
collect	将一个偏函数应用到每一个收到的元素， if the partial function is defined for a value the returned value is passed downstream.		当上游 complete 时
grouped	积累收到的元素直到到达了指定的数量， 然后传递这个元素集合到下游		当上游 complete 时
sliding (滑动)	提供一个滑动窗口，每次将滑动窗口内的元素传递到下游		当上游 complete 时
scan			
fold			
reduce			

drop	丢弃 n 个元素，然后将后面的元素传递到下游		
dropWhile	Drop elements as long as a predicate function return true for the element		
take	将前面的 n 个元素传递到下游，然后 complete		
takeWhile	Pass elements downstream as long as a predicate function return true for the element include the element when the predicate first return false and then complete.		
recover	看“Stream 的错误处理”一节		
recoverWithRetries	看“Stream 的错误处理”一节		
mapError			
detach	从下游需求中分离出上游需求，而不会影响 stream 速率。		
throttle	限制每个时间单元内指定元素数量的吞吐量， 必须提供一个函数来计算每个元素的单独消耗。		
intersperse	Intersperse stream with provided element similar to **List.mkStrin**g. It can inject start and end marker elements to stream.		
limit	使用 max 参数限制来自上游的元素数量		
log	用日志记录 stream 发射的元素，以及 stream 正常完成或者 error 的情况。 默认情况下，stream 发射的元素和 stream 的正常完成状态只会以 debug 模式记录日志， stream 的 error 状态以 error 模式记录日志。 可以通过 Attributes.logLevels(...)来改变这种行为。		

(3) 异步的 processing stages

这些异步 processing stages 封装了一个异步的计算。通过关心异步操作返回的 Future 来合理地处理 backpressure。

mapAsync

将接收到的元素传递给一个函数，那个函数返回一个 Future 结果。当 Future 到达时，它的结果将会被传递到下游。

允许能并发地处理 n 个元素。

不管这并发处理的 n 个元素每个的完成时间是怎样的，它们传递到下游的顺序与它们被发射的顺序是一样的，没有发生改变。

如果 Future 失败了，这个 Stream 也会失败。

具体使用案例：可以看“与 Actor 集成”一节。

mapAsyncUnordered

与 mapAsync 类似，但是 Future 的结果传递到下游的顺序与相应元素被发射的顺序是一样的。

(4) 时间感知的 stages

initialTimeout

在过了一段指定的超时时间之后，Stream 的第一个元素还未发射，则这个 Stream 将会以一个 TimeoutException 失败。

completionTimeout

在过了一段指定的超时时间之后，Stream 还未 complete，则这个 Stream 将会以一个 TimeoutException 失败。

idleTimeout

如果两个处理的元素的时间差超过了给定的超时时间，则这个 Stream 将会以一个 TimeoutException 失败。

backpressureTimeout

如果一个元素的发射和下游请求之间的时间差超过了给定的超时时间，则这个 Stream 将会以一个 TimeoutException 失败。

keepAlive

如果在过了一段指定的时间，上游还未发射元素，则插入额外的元素(代码配置的)。

(5) Watching status stages

watchTermination

Materializes to a Future that will be completed with Done or failed depending whether the upstream of the stage has been completed or failed.

monitor

Materializes to a FlowMonitor that monitors messages flowing through or completion of the stage.

The stage otherwise passes through elements unchanged.

Note that the FlowMonitor inserts a memory barrier every time it processes an event, and may therefore affect performance.

非法的 stream elements

按照 Reactive Streams 规范(第 2.13 条)，akka stream 不允许 null 作为元素在流中传递，如果你要定义没有值的模型，使用 scala.Option 或者 scala.util.Either。

二、Back-pressure 的解释

Akka Streams implement an asynchronous non-blocking back-pressure protocol standardised by the Reactive Streams specification。

Akka Stream propagate(传播) back-pressure information from stream Sinks (Subscribers) to their Sources (Publishers).

It is not an optional feature, and is enabled at all times.

Akka Stream 的使用者不需要显示地写任何关于 back-pressure 的代码，这是 Akka stream 内置并且会自动处理的机制。

三、stream 的物化

当使用 Akka Stream 构造 flows 和 graphs 时，把它们想象成一个蓝图，一个执行计划。
stream 的物化是这样一个过程：接收一个 stream 描述(Graph)，分配所有需要的资源去运行用 Graph 描述的计算。
在 Akka Stream 中，这通常意味启动有处理能力的 Actor，但并不仅仅限制于此，也有可能意味着打开文件或者 socket 连接，主要取决于 stream 需要什么。

stream 的物化被触发：通过调用 Source 或者 Flow 定义的 run()和 runWith()方法，以及少数特殊的语法糖 runForeach(el => ...) (等价于 runWith(Sink.foreach(el => ...)))。

Materialization is performed synchronously on the materializing thread.
实际的 stream processing 是由在 stream 物化期间启动的 **actor** 发起的。

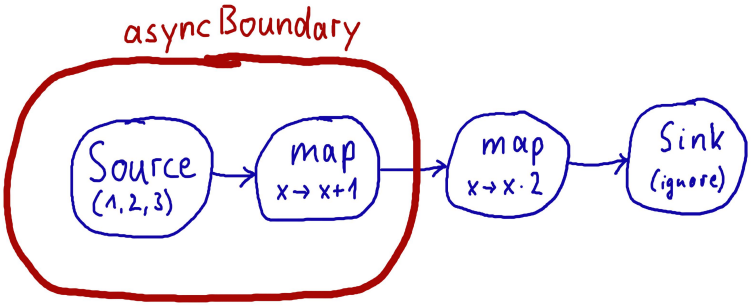
1、操作符融合　#

默认情况下，Akka Stream 将会融合 stream 的操作符。这意味着一个 flow 或者 graph 的 processing steps 将在一个相同的 actor 中执行，并且导致两个后果：
1. 将元素从一个处理阶段传递到下一个处理阶段的时间要比融合阶段快得多，因为避免了异步消息的开销。
2. 融合的流处理阶段并不会相互并行执行，这意味着每个融合部分只能使用最多一个 CPU 核心。

为了允许并行处理，你将不得不手动地插入异步边界到你的 flows 或者 graphs 中，方式是使用 Source、Sink、Flow 的 async 方法添加 Attributes.asyncBoundary，以异步的方式和 graph 的剩余部分进行通信。
例如：

```
Source(List(1, 2, 3))
  .map(_ + 1).async
  .map(_ * 2)
  .to(Sink.ignore)
```

在这个例子中，我们创建了流中的两个区域，这将在每个 actor 上执行。假如加法和乘法是代价很高的操作，由于使用两个 cpu 并行的执行这些任务将实现性能的提升。



上图表示所有在红色框中的 processing 都会在一个 actor 中执行，而在红框之外的所有 processing 在另一个 actor 中执行。

可以通过设置：**akka.stream.materializer.auto-fusing=off** 来禁用新的融合行为。

**2、组合物化值　*　**

因为在 Akka Stream 的每一个 processing stage 都会在物化之后产生一个物化值，以某种方式说明这些物化值应该如何组合成一个最终的值时非常必要的。针对这一点，很多组合方法有变体，接收一个额外的参数用来组合结果值。

```
// An source that can be signalled explicitly from the outside
val source: Source[Int, Promise[Option[Int]]] = Source.maybe[Int]

// A flow that internally throttles elements to 1/second, and returns a Cancellable which can be used to shut down the stream
val flow: Flow[Int, Int, Cancellable] = throttler

// A sink that returns the first element of a stream in the returned Future
val sink: Sink[Int, Future[Int]] = Sink.head[Int]

// By default, the materialized value of the leftmost(最左边) stage is preserved
val r1: RunnableGraph[Promise[Option[Int]]] = source.via(flow).to(sink)

// Simple selection of materialized values by using Keep.right
val r2: RunnableGraph[Cancellable] = source.viaMat(flow)(Keep.right).to(sink)
val r3: RunnableGraph[Future[Int]] = source.via(flow).toMat(sink)(Keep.right)

// Using runWith will always give the materialized values of the stages added by runWith() itself
val r4: Future[Int] = source.via(flow).runWith(sink)
val r5: Promise[Option[Int]] = flow.to(sink).runWith(source)
val r6: (Promise[Option[Int]], Future[Int]) = flow.runWith(source, sink)

// Using more complex combinations
val r7: RunnableGraph[(Promise[Option[Int]], Cancellable)] = source.viaMat(flow)(Keep.both).to(sink)
val r8: RunnableGraph[(Promise[Option[Int]], Future[Int])] = source.via(flow).toMat(sink)(Keep.both)
val r9: RunnableGraph[((Promise[Option[Int]], Cancellable), Future[Int])] = source.viaMat(flow)(Keep.both).toMat(sink)(Keep.both)
val r10: RunnableGraph[(Cancellable, Future[Int])] = source.viaMat(flow)(Keep.right).toMat(sink)(Keep.both)

// It is also possible to map over the materialized values. In r9 we had a doubly nested pair, but we want to flatten it out
val r11: RunnableGraph[(Promise[Option[Int]], Cancellable, Future[Int])] =
  r9.mapMaterializedValue {
    case ((promise, cancellable), future) ⇒ (promise, cancellable, future)
  }

// Now we can use pattern matching to get the resulting materialized values
val (promise, cancellable, future) = r11.run()

// Type inference works as expected
promise.success(None)
cancellable.cancel()
future.map(_ + 3)

//  上面 r11 的结果也可以通过使用 GraphAPI 来实现
val r12: RunnableGraph[(Promise[Option[Int]], Cancellable, Future[Int])] =
  RunnableGraph.fromGraph(GraphDSL.create(source, flow, sink)((_, _, _)) { implicit builder ⇒ (src, f, dst) ⇒
```

```
    import GraphDSL.Implicits._
    src ~> f ~> dst
    ClosedShape
  })
```

除了使用 Keep.left、Keep.right、Keep.both 来访问物化值，我们也可以使用自定义的函数来访问多个 processing stage 的物化值，并产生另一个最终的物化值。

示例：

```
case class MyClass(private val p: Promise[Option[Int]], conn: OutgoingConnection) {
  def close() = p.trySuccess(None)
}

def f(p: Promise[Option[Int]], rest: (Future[OutgoingConnection], Future[String])): Future[MyClass] = {
  val connFuture = rest._1
  connFuture.map(MyClass(p, _))
}

// Materializes to Promise[Option[Int]]                                 (red)
val source: Source[Int, Promise[Option[Int]]] = Source.maybe[Int]
// Materializes to NotUsed                                              (black)
val flow1: Flow[Int, Int, NotUsed] = Flow[Int].take(100)
// Materializes to Promise[Int]                                         (red)
val nestedSource: Source[Int, Promise[Option[Int]]] = source.viaMat(flow1)(Keep.left).named("nestedSource")

// Materializes to NotUsed                                              (orange)
val flow2: Flow[Int, ByteString, NotUsed] = Flow[Int].map { i ⇒ ByteString(i.toString) }
// Materializes to Future[OutgoingConnection]                           (yellow)
val flow3: Flow[ByteString, ByteString, Future[OutgoingConnection]] = Tcp().outgoingConnection("localhost", 8080)
// Materializes to Future[OutgoingConnection]                           (yellow)
val nestedFlow: Flow[Int, ByteString, Future[OutgoingConnection]] = flow2.viaMat(flow3)(Keep.right).named("nestedFlow")

// Materializes to Future[String]                                       (green)
val sink: Sink[ByteString, Future[String]] = Sink.fold("")(_ + _.utf8String)
// Materializes to (Future[OutgoingConnection], Future[String])         (blue)
val nestedSink: Sink[Int, (Future[OutgoingConnection], Future[String])] = nestedFlow.toMat(sink)(Keep.both)

// Materializes to Future[MyClass]                                      (purple)
val runnableGraph: RunnableGraph[Future[MyClass]] = nestedSource.toMat(nestedSink)(f)
```

3、stream 的顺序

在 Akka Stream 中几乎所有的计算 stage 都维护元素的输入顺序。也就是说，如果输入{IA1,IA2,...,IAn} 导致输出 {OA1,OA2,...,OAk}，输入{IB1,IB2,...,IBm} 导致输出 {OB1,OB2,...,OBl}，IAi 发送在所有的 IBi 之前，OAi 发送在所有的 OBi 之前。

这个特征即使是在一些异步操作中也是支持的，例如 mapAsync。但是 mapAsyncUnordered 则不会维护元素的顺序。

然而，在处理多个输入流（例如合并）的结点的情况下，一般来说，对于到达不同输入端口的元素，输出的顺序是没被定义的。

如果你发现你需要在 fan-in 情况下很好的控制元素的顺序，可以考虑使用 MergePreferred、MergePrioritized 或者 GraphStage，它们让你完全控制合并是怎样执行的。

4、Actor Materializer 的生命周期

TODO

Graph 的使用 *

在 Akka Stream 中，graph 计算不像线性计算那样使用流式 DSL，而是使用一种类似 graph 的 DSL，使用这种 DSL 的目的是为翻译 graph 的图示，并且让代码更加简洁。

Graph 的使用场景：

Graphs are needed whenever you want to perform any kind of **fan-in (“multiple inputs”)** or **fan-out (“multiple outputs”)** operations.

1、构造 Graph

Graphs are built from simple **Flows** which serve as the **linear connections** within the graphs as well as **junctions** which serve as **fan-in and fan-out points** for Flows.

一旦 Graph DSL 实例创建了，这个实例将不再可变，意味着它是线程安全的，可以安全的多线程共享。---- 这个规则适也用于其它 Akka Stream 的组件：Source、Sink、Flow。

基于 Junctions(交汇点)的行为使得 Junctions 有很多有意义的类型，这也让 junction 变得很简单易用。

Akka Stream 提供了下面这些 junctions：

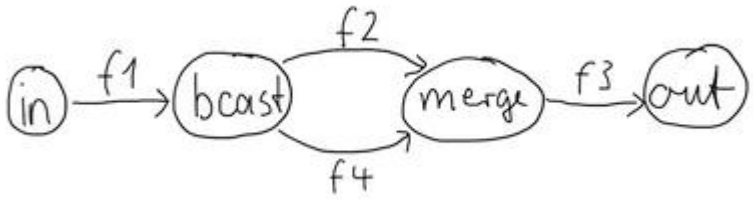
Fan-out

- Broadcast[T] -- (一个输入，多个输出)，将一个输入发射到每一个输出 (就是广播，所有的输出都无差别的获得输入)
- Balance[T] -- (一个输入，多个输出)，将一个输出发射到任意的一个输出 (就是类似负载均衡，或者说路由选择吧)
- UnzipWith[In,A,B,...] -- (一个输入，多个输出)，使用一个函数将一个输入的元素拆分成多个下游(s)。
- UnZip[A,B] -- (一个输入，两个输出)，把有着(A, B)元素的 stream 拆分成两个 stream，一个类型是 A 一个类型是 B。

Fan-in

- Merge[In] -- (多个输入，一个输出)，合并多个 sources，如果所有的 source 都有了准备好的元素，则从任意输入中选择元素，一个接一个地推送至输出。
- MergePreferred[In] -- 合并多个 sources，如果所有的 source 都有了准备好的元素，但是要发射的元素是从首选端口采集，否则再从其他任意端口采集 (拥有一个优先级高于端口的输入，优先采集该端口的)
- MergePrioritized[In] -- 合并多个 sources，如果所有的 source 都有了准备好的元素，优先依据优先级选择可以发射元素的 sources。
- zipWith[A,B,...,Out] -- (多个输入，一个输出)，通过一个函数将多个 sources 的元素进行合并，然后把函数的返回值传递到下游。
- Zip[A,B] -- (两个输入，一个输出)，将输入的 A 类型的 stream 和 B 类型的 stream 合并成(A, B)元组类型的输出，是特殊的 zipWith。
- Concat[A] -- (两个输入，一个输出)，连接两个流 (先消费其中一个，再消费另一个)。

让我们看看如何使用 Graph DSL 来描述下面的流程图：



上面这张图示很容易翻译成 Graph DSL 的。因为：

每一个线性元素对应为一个 Flow； (例如下面代码中的 f1, f2, f3, f4)

如果一个圆圈是一个 Flow 的开始或者结束，则圆圈对应为一个 Junction 或者 Source 或者 Sink；

Junction 在创建时一定是带有定义好的类型参数的，否则将会被推断为 Nothing 类型。

代码：

```
val g = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder: GraphDSL.Builder[NotUsed] =>
  import GraphDSL.Implicits._
  val in = Source(1 to 10)
  val out = Sink.ignore
```

```
    val bcast = builder.add(Broadcast[Int](2))
    val merge = builder.add(Merge[Int](2))

    val f1, f2, f3, f4 = Flow[Int].map(_ + 10)

    in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
    bcast ~> f4 ~> merge
    ClosedShape
})
```

下面展示一个新的例子：**创建一个包含两个并行 streams 的 Graph**，我们重复使用同一个 Flow 实例，这将会物化成两个 connections。

```
val topHeadSink = Sink.head[Int]
val bottomHeadSink = Sink.head[Int]
val sharedDoubler = Flow[Int].map(_ * 2)
RunnableGraph.fromGraph(GraphDSL.create(topHeadSink, bottomHeadSink)((_, _)) { implicit builder =>
    (topHS, bottomHS) =>
    import GraphDSL.Implicits._

    val broadcast = builder.add(Broadcast[Int](2))
    Source.single(1) ~> broadcast.in

    broadcast.out(0) ~> sharedDoubler ~> topHS.in
    broadcast.out(1) ~> sharedDoubler ~> bottomHS.in
    ClosedShape
})
```

2、构造并组合部分 Graph

Sometimes it is not possible (or needed) to construct the entire computation graph in one place, but instead construct all of its different phases in different places and in the end connect them all into a complete graph and run it.
This can be achieved by returning a different Shape than ClosedShape, for example FlowShape(in, out), from the function given to GraphDSL.create.

Making a Graph a **RunnableGraph** requires **all ports to be connected**, and if they are not it will throw an exception at construction time, which helps to avoid simple wiring errors while working with graphs.
A **partial graph** however allows you to return the set of yet to be connected ports from the code block that performs the internal wiring.

TODO

3、从部分 Graph 中构造 Source、Sink、Flow

有时候暴露一个复杂 graph 作为一个更简单的结构是很有用的，例如 Source、Sink、Flow。而不是把一个部分 Graph 当做一个可能尚未连接的 flows 和 junctions 的集合。

事实上，作为一个部分的的已连接的 graph 的特殊案例，这些概念是很容易表示的：

- Source is a partial graph with exactly one output, that is it returns a SourceShape.
- Sink is a partial graph with exactly one input, that is it returns a SinkShape.
- Flow is a partial graph with exactly one input and exactly one output, that is it returns a FlowShape.

为了能**从一个 Graph 来创建一个 Source**，可以使用 Source.fromGraph 方法。为了使用这个方法，我们必须有一个 Graph[SourceShape, T] 类型，而这个类型可以通过使用 GraphDSL.create 方法来构造。
单独的出口必须提供给 SourceShape.of 方法，这也符合“Sink 必须在 Source 运行前被连接上”。

下面看一个从 Graph 来创建 Source 的例子：We create a Source that zips together two numbers

```
val pairs = Source.fromGraph(GraphDSL.create() { implicit b ⇒
  import GraphDSL.Implicits._

  // prepare graph elements
  val zip = b.add(Zip[Int, Int]())
  def ints = Source.fromIterator(() ⇒ Iterator.from(1))

  // connect the graph
  ints.filter(_ % 2 != 0) ~> zip.in0
  ints.filter(_ % 2 == 0) ~> zip.in1

  // expose port
  SourceShape(zip.out)
})
// 使用从 Graph 创建的 Source
val firstPair: Future[(Int, Int)] = pairs.runWith(Sink.head)
```

下面展示**从一个 Graph 创建 Flow** 的例子：

```
val pairUpWithToString = Flow.fromGraph(GraphDSL.create() { implicit b ⇒
    import GraphDSL.Implicits._

    // prepare graph elements
    val broadcast = b.add(Broadcast[Int](2))
    val zip = b.add(Zip[Int, String]())

    // connect the graph
    broadcast.out(0).map(identity) ~> zip.in0
    broadcast.out(1).map(_.toString) ~> zip.in1

    // expose ports
```

```
    FlowShape(broadcast.in, zip.out)
  })
// 使用从 Graph 创建的 Flow
pairUpWithToString.runWith(Source(List(1)), Sink.head)
```

4、使用简化的 API 组合 Source、Sink

有一些简化的 API 用来组合 Source 或者 Sink，可以达到使用 junction(例如：Broadcast[T], Balance[T], Merge[In] and Concat[A])的效果，但是无需使用 Graph DSL。

下面展示**合并两个 Source 成一个 Source，达到 fan-in 操作中 Merge[In]**的效果：

```
val sourceOne = Source(List(1))
val sourceTwo = Source(List(2))
val merged = Source.combine(sourceOne, sourceTwo)(Merge(_))
val mergedResult: Future[Int] = merged.runWith(Sink.fold(0)(_ + _))   // 使用合并后的 Source
```

下面展示如何**组合两个 Sink，来达到 fan-out 操作中 Broadcast[T]**的效果：

```
val sendRmotely = Sink.actorRef(actorRef, "Done")
val localProcessing = Sink.foreach[Int](_ ⇒ /* do something usefull */ ())
val sink = Sink.combine(sendRmotely, localProcessing)(Broadcast[Int](_))
Source(List(0, 1, 2)).runWith(sink)
```

5、Predefined shapes

Akka Stream 已经提供了一些预定义好的 shapes：

- SourceShape、SinkShape 、FlowShape 适用于简单图形。
- UniformFanInShape 和 UniformFanOutShape 适用于有着相同类型的多个输入(或者输出)端口的交汇点。
- FanInShape1、FanInShape2...、FanOutShape1、FanOutShape2...适用于有着不同类型的多个输入(或者输出)端口的交汇点。

6、双向的 Flow

BidiFlow 是一个 graph，它有两个开放的输入和两个开放的输出。
BidiFlow 对应的 shape 叫做 BidiShape。

BidiShape 使用下面的类进行定义：

```
/**
 * A bidirectional(双向) flow of elements that consequently has two inputs and two outputs, arranged like this:
 *
 * {{{
 *               +------+
 *    In1 ~>|          |~> Out1
 *               | bidi |
 * Out2 <~|          |<~ In2
 *               +------+
 * }}}
 */
final case class BidiShape[-In1, +Out1, -In2, +Out2](
   in1:    Inlet[In1 @uncheckedVariance],
   out1: Outlet[Out1 @uncheckedVariance],
   in2:    Inlet[In2 @uncheckedVariance],
   out2: Outlet[Out2 @uncheckedVariance]) extends Shape {
   override val inlets: immutable.Seq[Inlet[_]] = in1 :: in2 :: Nil
   override val outlets: immutable.Seq[Outlet[_]] = out1 :: out2 :: Nil

   /**
    * Java API for creating from a pair of unidirectional flows.
    */
   def this(top: FlowShape[In1, Out1], bottom: FlowShape[In2, Out2]) = this(top.in, top.out, bottom.in, bottom.out)

   override def deepCopy(): BidiShape[In1, Out1, In2, Out2] =
      BidiShape(in1.carbonCopy(), out1.carbonCopy(), in2.carbonCopy(), out2.carbonCopy())
}
```

下面的示例代码中演示了如何从一个 Graph 来创建一个 BidiFlow：

```
trait Message
case class Ping(id: Int) extends Message
case class Pong(id: Int) extends Message

def toBytes(msg: Message): ByteString = {
   implicit val order = ByteOrder.LITTLE_ENDIAN
   msg match {
      case Ping(id) ⇒ ByteString.newBuilder.putByte(1).putInt(id).result()
      case Pong(id) ⇒ ByteString.newBuilder.putByte(2).putInt(id).result()
   }
}

def fromBytes(bytes: ByteString): Message = {
```

```
    implicit val order = ByteOrder.LITTLE_ENDIAN
    val it = bytes.iterator
    it.getByte match {
      case 1     ⇒ Ping(it.getInt)
      case 2     ⇒ Pong(it.getInt)
      case other ⇒ throw new RuntimeException(s"parse error: expected 1|2 got $other")
    }
}

val codecVerbose = BidiFlow.fromGraph(GraphDSL.create() { b ⇒
    // construct and add the top flow, going outbound
    val outbound = b.add(Flow[Message].map(toBytes))
    // construct and add the bottom flow, going inbound
    val inbound = b.add(Flow[ByteString].map(fromBytes))
    // fuse them together into a BidiShape
    BidiShape.fromFlows(outbound, inbound)
})

// 上面的代码演示了如何从从一个 Graph 来创建一个 BidiFlow，但是在这种 1:1 转换的情况下，下面这行代码展示了一种更加具体方便的方法：
// this is the same as the above
val codec = BidiFlow.fromFunctions(toBytes _, fromBytes _)
```

7、访问 Graph 中的物化值

在某些特殊场景中，获取 graph(部分图，完全图，或者支持 Source,Sink,Flow,BidiFlow)的物化值是非常有必要的。

通过使用 builder.materializedValue，他返回一个 Outlet 类型，被用在 graph 中作为普通的 source 或者 outlet，并最终发射物化值。
如果物化值在不止一个地方需要，无论调用多少次的 materializedValue 都能取得必要的输出。

示例：

```
import GraphDSL.Implicits._
val foldFlow: Flow[Int, Int, Future[Int]] = Flow.fromGraph(GraphDSL.create(Sink.fold[Int, Int](0)(_ + _)) { implicit builder ⇒ fold ⇒
    FlowShape(fold.in, builder.materializedValue.mapAsync(4)(identity).outlet)
})
```

8、Graph 循环、活跃性、死锁

在有限流拓扑中的循环需要特别注意避免潜在的死锁以及其他活跃度的问题。

举一个会产生死锁的例子：

```
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val merge = b.add(Merge[Int](2))
  val bcast = b.add(Broadcast[Int](2))

  source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
            merge                          <~                          bcast
  ClosedShape
})
```

运行上面的代码我们发现，在少量的元素被打印之后，将不再有元素被打印到控制台，所有的操作在某个时间之后被终止。

因为 Akka Stream 保证有界的处理，这意味着在任何时间范围内只有有限数量的元素被缓冲。
由于上面代码中的循环获得的元素越来越多，最终使它内部的缓冲区被填满，导致永远的 backpressure source。

通过使用 buffer()方法，并给一个丢弃策略 OverflowStrategy.dropHead，就可以保证上面的循环一直保持活跃，不会死锁。

```
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val merge = b.add(Merge[Int](2))
  val bcast = b.add(Broadcast[Int](2))

  source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
      merge <~ Flow[Int].buffer(10, OverflowStrategy.dropHead) <~ bcast
  ClosedShape
})
```

模块化、组合、层级 *

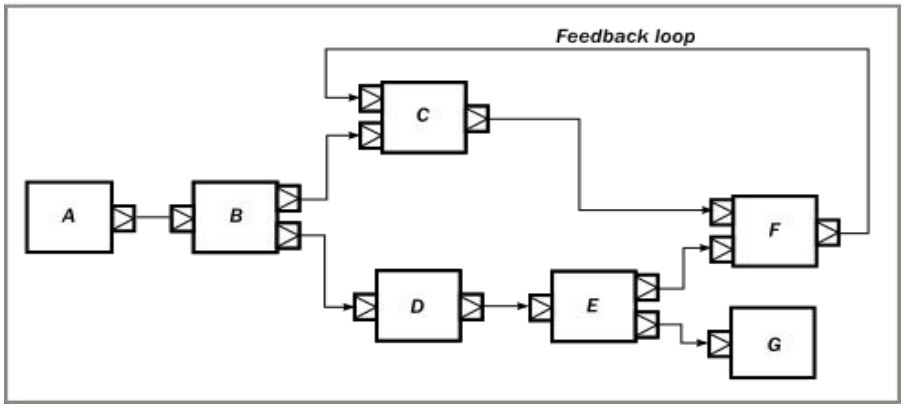
1、组合和模块化的基础

TODO

2、组成复杂的系统

首先来看一个复杂的布局：

RunnableGraph



上面的图展示了一个 RunnableGraph，它包含 fan-in，fan-out，直接和非直接的循环。

上面的 RunnableGraph 可以用下面的代码描述：

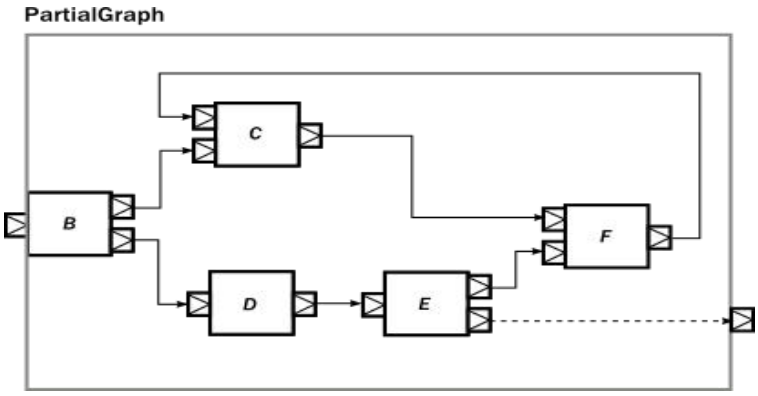
```
import GraphDSL.Implicits._
RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  val A: Outlet[Int]                  = builder.add(Source.single(0)).out
  val B: UniformFanOutShape[Int, Int] = builder.add(Broadcast[Int](2))
  val C: UniformFanInShape[Int, Int]  = builder.add(Merge[Int](2))
  val D: FlowShape[Int, Int]          = builder.add(Flow[Int].map(_ + 1))
  val E: UniformFanOutShape[Int, Int] = builder.add(Balance[Int](2))
  val F: UniformFanInShape[Int, Int]  = builder.add(Merge[Int](2))
  val G: Inlet[Any]                   = builder.add(Sink.foreach(println)).in

                C  <~  F
  A  ~>  B  ~>  C  ~>  F
         B  ~>  D  ~>  E  ~>  F
                       E  ~>  G

  ClosedShape
})
```

然而，到目前为止我们还没有考虑“**模块化**”。在上面我们创建了一个复杂的 RunnableGraph，但是他是太臃肿了，不是“**模块化**”的。

接下来，我们将会使用 graph DSL 创建一个可以重用的组件，这可以通过 GraphDSL 单例对象的 create()方法实现。
如果我们删除上图中的 source 和 sink，将会得到一个 partial graph：



使用下面的代码描述上面的 partial graph：

```
import GraphDSL.Implicits._
val partial = GraphDSL.create() { implicit builder =>
  val B = builder.add(Broadcast[Int](2))
  val C = builder.add(Merge[Int](2))
  val E = builder.add(Balance[Int](2))
  val F = builder.add(Merge[Int](2))
                                      C  <~  F
  B  ~>                               C  ~>  F
  B  ~>  Flow[Int].map(_ + 1)  ~>  E  ~>  F

  // 每一个 graph 都有一个 shape，由于我们这个例子中 partial graph 有一个输入和一个输出，所以可以使用 FlowShape
  FlowShape(B.in, E.out(1))
}.named("partial")
```

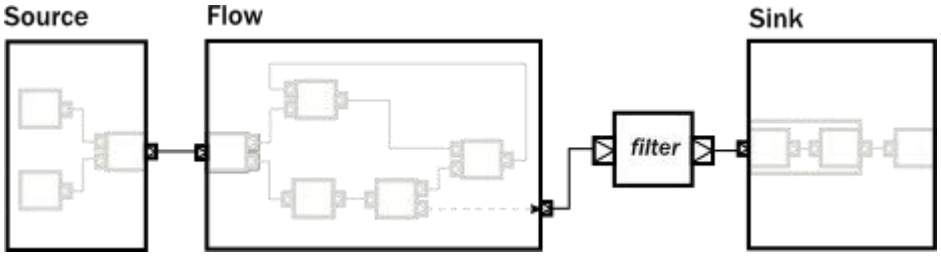
上面已经定义好了 partial graph，接下来就可以使用它了。(由前面知道这个 partial graph 缺少一个 source 和一个 sink)

```
Source.single(0).via(partial).to(Sink.ignore)
```

注意：
所有的 graph 生成器部分会检查：生成的 **partial graph** 除了暴露的端口外，所有其他端口是否被连接。如果违反这，将抛出异常。

上面虽然定义了由一个输入和一个输出的 partial graph，但是确不能将它作为 Flow 使用(例如我们不能调用 Flow 的 filter()方法)。但是 Flow 有一个 fromGraph()方法 (当然，Source、Sink、BidiFlow 也有类似的方法)，接收一个 graph DSL 返回一个 FlowShape。此处就体现了“**模块化**”，一旦定义了一个 partial graph，就可以重复使用。

下面是一个使用上面定义好的 partial graph 的例子：



代码描述：

```
// Convert the partial graph of FlowShape to a Flow to get
// access to the fluid DSL (for example to be able to call .filter())
val flow = Flow.fromGraph(partial)

// Simple way to create a graph backed Source
val source = Source.fromGraph( GraphDSL.create() { implicit builder =>
   val merge = builder.add(Merge[Int](2))
   Source.single(0)          ~> merge
   Source(List(2, 3, 4))     ~> merge

   // Exposing exactly one output port
   SourceShape(merge.out)
})

// Building a Sink with a nested Flow, using the fluid DSL
val sink = {
   val nestedFlow = Flow[Int].map(_ * 2).drop(10).named("nestedFlow")
   nestedFlow.to(Sink.head)
}

// Putting all together
val closed = source.via(flow.filter(_ > 1)).to(sink)
```

其实，**RunnableGraph** 还是这样的一个组件：**可以嵌入到其它的 graph 中**。

例如：

```
val closed1 = Source.single(0).to(Sink.foreach(println))
val closed2 = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder ⇒
  val embeddedClosed: ClosedShape = builder.add(closed1)
  // …
  embeddedClosed
})
```

3、物化值

在意识到 RunnableGraph 就是一个还未使用端口的模块之后，就会很清晰的知道在物化之后，唯一能和运行流处理逻辑沟通的方式是通过 side-channel。这个 side-channel 被表示为一个物化值。

每一次物化都会创建一个新的运行网络，与 RunnableGraph 提供的蓝图一致。
为了能和这个运行网络交互，每一次物化需要返回一个不同的对象，这个对象提供了必要的交互的能力。
换句话说，RunnableGraph 能被看做一个工厂，它创建了：
1. 一个运行时处理实体的网络，这个网络从外部不可见。
2. 一个物化值，可选地提供了与运行网络交互的控制能力。

由于每个 processing stage 都会提供一个物化值，所以当组合多个 processing stages 时，也需要合并这些 processing stages 产生的物化值。具体的细节看前面的“stream 的物化#组合物化值”一节。

4、Attributes 和 Attributes 继承

在这之前我们已经看到，在一个流式 DSL 使用 named()方法引入一个嵌套的层次。
named()除了有这个作用外，实际上它是 withAttributes(Attributes.name("someName"))方法的简写形式。

Attributes 提供了一种方式，可以针对物化的指定方面进行微调。例如，缓冲区大小可以通过属性控制。

下面的代码是对以前的例子的一种修改，在某个指定的模块上设置了 inputBuffer 属性：

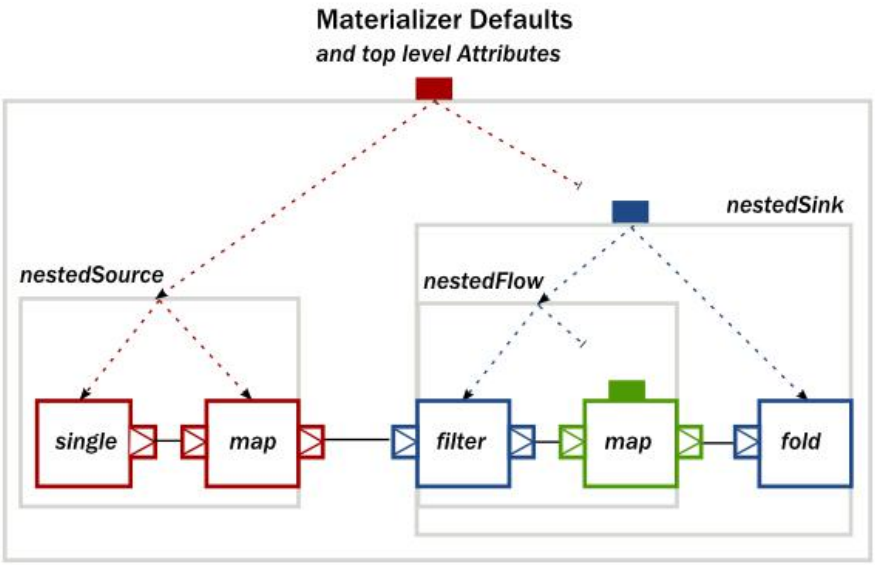
```
import Attributes._
val nestedSource = Source.single(0)
    .map(_ + 1)
    .named("nestedSource")        // Wrap, no inputBuffer set

val nestedFlow = Flow[Int].filter(_ != 0)
    .via(Flow[Int].map(_ - 2).withAttributes(inputBuffer(4, 4))) // override
    .named("nestedFlow")          // Wrap, no inputBuffer set

val nestedSink = nestedFlow.to(Sink.fold(0)(_ + _))        // wire an atomic sink to the nestedFlow
    .withAttributes(name("nestedSink") and inputBuffer(3, 3)) // override
```

- nestedSource 从 maretializer 获取默认的属性；
- nestedSink 显示地设置了属性，并且会适用到所有内部的模块；
- nestedFlow 因为是 nestedSink 的内部模块，所以会从 nestedSink 继承到 inputBuffer 属性，但是 map stage 有它自己显示设置的属性(覆盖了继承的)。

可以用一张图描述上面的继承过程：



Buffer 和处理速率

下面的部分将介绍：如何在 Akka Stream 中使用 buffer。

1、Buffers for asynchronous stages

在“stream 的物化#操作符融合”一节，介绍了使用 async 方法手动插入异步边界，可以让 processing stage 在不同的 actor 中异步执行。

异步执行意味着：一个 processing stage 在将一个元素派发到下游的消费者之后，就可以立即处理下一个元素，而不用等上一个元素完全的通过所有的 processing stages。

异步执行的示例：

```
Source(1 to 3)
  .map { i ⇒ println(s"A: $i"); i }.async
  .map { i ⇒ println(s"B: $i"); i }.async
  .map { i ⇒ println(s"C: $i"); i }.async
  .runWith(Sink.ignore)
```

运行上面的例子，会输出：

```
A: 1
A: 2
B: 1
A: 3
B: 2
C: 1
B: 3
C: 2
C: 3
```

而不是这个顺序：A:1, B:1, C:1, A:2, B:2, C:2。

如果去掉上面示例中的 async 方法，那么由于 Akka Stream 默认会融合 stream 的操作符，即 stream 的所有 processing stages 在同一个 actor 中同步地执行。

这种情况下输出的结果就是：A:1, B:1, C:1, A:2, B:2, C:2 这个顺序。

在实践中，传递一个元素穿过异步边界的成本是很显著的，为了分摊这个开销，Akka Stream 内部使用了一个窗口化、批量的 backpressure 策略。

This batching strategy reduces the communication cost of propagating the backpressure signal through the asynchronous boundary.

由于性能的原因，Akka Stream 为每一个 asynchronous process stage 提供了一个 buffer。

提供 buffer 目的纯粹是为了优化。

如果没有提升吞吐量的需求，那么将 buffer 的尺寸设为 1 是最自然的选择。因此建议把 buffer 的尺寸保持在小容量，并且仅仅增加到与应用吞吐量需求相适配的水平。

默认，buffer 的大小可以通过下面的配置项进行配置：

```
akka.stream.materializer.max-input-buffer-size = 16
```

当然，也可以为 materializer 传递一个 ActorMaterializerSettings 进行设置：

```
val materializer = ActorMaterializer(
  ActorMaterializerSettings(system)
    .withInputBuffer(
      initialSize = 64,
```

maxSize = 64))

如果只想为 stream 的其中某一个 processing stage 设置 buffer，可以用下面的方式进行配置：

```
val section = Flow[Int].map(_ * 2).async
  .addAttributes(Attributes.inputBuffer(initial = 1, max = 1))    // the buffer size of this map is 1
val flow = section.via(Flow[Int].map(_ / 2)).async                // the buffer size of this map is the default
```

2、Akka Stream 中的各种 buffer 策略

下面的例子来自外部系统的 1000(但不超过)各个作业被入队，并存储在本地缓存中。

```
// Getting a stream of jobs from an imaginary external system as a Source
val jobs: Source[Job, NotUsed] = inboundJobsConnector()
jobs.buffer(1000, OverflowStrategy.backpressure)
```

下面的 buffer 也会保存 1000 个作业到本地，但是如果有更多的作业想入队，那么将会通过删除 buffer 中最末尾(即最年轻)的元素来创造空间接受新的元素。

```
jobs.buffer(1000, OverflowStrategy.dropTail)
```

如果不想删除 buffer 中最末尾的元素，可以直接将想入队的新元素删除，根本不让它入队。

```
jobs.buffer(1000, OverflowStrategy.dropNew)
```

如果不想删除 buffer 中最末尾的元素，而是想删除开头(即最老)的元素，可以这样：

```
jobs.buffer(1000, OverflowStrategy.dropHead)
```

也有一种更粗暴的策略，就是一旦当 buffer 满了，就删除 buffer 中所有的元素。

```
jobs.buffer(1000, OverflowStrategy.dropBuffer)
```

下面的 buffer 会在 buffer 满了之后产生一个 error，并 fail the stream。这在强迫元素提供方不能有超过 1000 个元素的场景中很有用。

```
jobs.buffer(1000, OverflowStrategy.fail)
```

3、速率转换 --- 背压感知的 stages *

(1) 合并 conflate

当不能使用 backpressure 或者其他信号通知速率过快的生产者减缓速率时，conflate(合并)可能是有用的去合并来自生产者的元素，直到消费者有能力处理时。

只有有 backpressure 产生了，conflate 允许**速率过慢的下游**传递收到的元素和一个汇总信息到一个聚合函数中。

下面的例子片段展示了汇总快速流生产的元素到一个非标准偏差，统计已经到达的元素数量及其平均值。

```
val statsFlow = Flow[Double]
  .conflateWithSeed(Seq(_))(_ :+ _)
  .map { s ⇒
    val μ = s.sum / s.size
    val se = s.map(x ⇒ pow(x - μ, 2))
    val σ = sqrt(se.sum / se.size)
    (σ, μ, s.size)
```

```
  }
```

conflate 的另一个示例是当生产者产生元素太快时，不考虑全部的元素。

下面的示例说明在消费者(的消费能力)跟不上生产者(的生产能力)时如何使用 conflate 来随机丢弃元素：

```
val p = 0.01
val sampleFlow = Flow[Double]
  .conflateWithSeed(Seq(_)) {
    case (acc, elem) if Random.nextDouble < p ⇒ acc :+ elem
    case (acc, _)                             ⇒ acc
  }
  .mapConcat(identity)
```

(2) 展开 expand

expand 有助于处理那些无法跟上来自消费者需求的**缓慢生产者**。

expand 允许**速率过快的下游** expanding 最后收到的元素到一个 Iterator 中。

例如下面 expand 示例展示：当生产者没有新元素要发送时，则发送相同的元素到消费者

```
val lastFlow = Flow[Double]
  .expand(Iterator.continually(_))
```

动态地 stream 处理

1、用 KillSwitch 控制 graph 的完成

(1) KillSwitch 介绍

作用：使用 KillSwitch 允许从**外部**来**完成 FlowShape** graph。
这有点类似 RxJava 里面的 onComplete 和 onError。

KillSwitch 特质允许：

- 通过 shutdown()方法完成 graph(s)
- 通过 abort(Throwable error) 方法 fail graph(s)

```
trait KillSwitch {
  // After calling [[KillSwitch#shutdown()]] the linked [[Graph]]s of [[FlowShape]] are completed normally.
  def shutdown(): Unit
  // After calling [[KillSwitch#abort()]] the linked [[Graph]]s of [[FlowShape]] are failed.
  def abort(ex: Throwable): Unit
}
```

一旦调用了 shutdown()或者 abort()方法，如果后面还会调用这两个方法，那么将会被忽略。

通过下面两个步骤来完成 graph：

- completing its downstream
- cancelling (in case of shutdown) or failing (in case of abort) its upstream.

一个 KillSwitch 能控制一个或者多个 streams 的完成，因此导致两种不同的风格。
下面就介绍这两种不同的风格。

(2) UniqueKillSwitch

UniqueKillSwitch 允许控制“**一个 FlowShape graph**”的完成。
参考下面的示例：

shutdown()方法的示例：

```
val countingSrc = Source(Stream.from(1)).delay(1.second, DelayOverflowStrategy.backpressure)
val lastSnk = Sink.last[Int]
val (killSwitch, last) = countingSrc
  .viaMat(KillSwitches.single)(Keep.right)
  .toMat(lastSnk)(Keep.both)
  .run()
doSomethingElse()
killSwitch.shutdown()
Await.result(last, 1.second) shouldBe 2
```

abort()方法的示例：

```
val countingSrc = Source(Stream.from(1)).delay(1.second, DelayOverflowStrategy.backpressure)
val lastSnk = Sink.last[Int]
val (killSwitch, last) = countingSrc
  .viaMat(KillSwitches.single)(Keep.right)
```

```
    .toMat(lastSnk)(Keep.both).run()
val error = new RuntimeException("boom!")
killSwitch.abort(error)
Await.result(last.failed, 1.second) shouldBe error
```

(3) SharedKillSwitch

SharedKillSwitch 允许控制“**任意数量的 FlowShape graph**”的完成。

可以通过 SharedKillSwitch 的 flow 方法物化它多次，并且所有和 SharedKillSwitch 关联的 materialized graphs 都被这个 SharedKillSwitch 控制着。

参考下面的示例：

shutdown()方法的示例：

```
val countingSrc = Source(Stream.from(1)).delay(1.second, DelayOverflowStrategy.backpressure)
val lastSnk = Sink.last[Int]
val sharedKillSwitch = KillSwitches.shared("my-kill-switch")

val last = countingSrc
    .via(sharedKillSwitch.flow)
    .runWith(lastSnk)

val delayedLast = countingSrc
    .delay(1.second, DelayOverflowStrategy.backpressure)
    .via(sharedKillSwitch.flow)
    .runWith(lastSnk)

doSomethingElse()

sharedKillSwitch.shutdown()

Await.result(last, 1.second) shouldBe 2
Await.result(delayedLast, 1.second) shouldBe 1
```

abort()方法的示例：

```
val countingSrc = Source(Stream.from(1)).delay(1.second)
val lastSnk = Sink.last[Int]
val sharedKillSwitch = KillSwitches.shared("my-kill-switch")

val last1 = countingSrc.via(sharedKillSwitch.flow).runWith(lastSnk)
val last2 = countingSrc.via(sharedKillSwitch.flow).runWith(lastSnk)

val error = new RuntimeException("boom!")
sharedKillSwitch.abort(error)

Await.result(last1.failed, 1.second) shouldBe error
Await.result(last2.failed, 1.second) shouldBe error
```

2、Dynamic fan-in and fan-out with MergeHub, BroadcastHub and PartitionHub

有些场景中，consumer 和 producer 预先并不知道。
而要创建一个 RunnableGraph 必须要求：all connections of the graph must be known in advance and must be connected upfront。

为了允许动态的 fan-in 和 fan-out，就应该使用 Hubs。

(1) MergeHub 的使用

MergeHub 允许在一个 graph 中实现一个动态的 fan-in junction。
The hub itself comes as a Source to which the single consumer can be attached.

MergeHub 的使用示例：

```
// A simple consumer that will print to the console for now
val consumer = Sink.foreach(println)

// Attach a MergeHub Source to the consumer. This will materialize to a corresponding Sink.
val runnableGraph: RunnableGraph[Sink[String, NotUsed]] =
   MergeHub.source[String](perProducerBufferSize = 16).to(consumer)

// By running/materializing the consumer we get back a Sink,
// and hence now have access to feed elements into it.
// This Sink can be materialized any number of times, and every element that enters the Sink will be consumed by our consumer.
val toConsumer: Sink[String, NotUsed] = runnableGraph.run()

// Feeding two independent sources into the hub.
Source.single("Hello!").runWith(toConsumer)
Source.single("Hub!").runWith(toConsumer)
```

(2) BroadcastHub 的使用

BroadcastHub 允许在一个 graph 中实现一个动态的 fan-out junction。
producer 的发送速率将会自动地适配最慢的 consumer。

BroadcastHub 的使用示例：

```
// A simple producer that publishes a new "message" every second
val producer = Source.tick(1.second, 1.second, "New message")

// Attach a BroadcastHub Sink to the producer. This will materialize to a corresponding Source.
// (We need to use toMat and Keep.right since by default the materialized value to the left is used)
val runnableGraph: RunnableGraph[Source[String, NotUsed]] =
   producer.toMat(BroadcastHub.sink(bufferSize = 256))(Keep.right)

// By running/materializing the producer, we get back a Source, which gives us access to the elements published by the producer.
val fromProducer: Source[String, NotUsed] = runnableGraph.run()
```

```
// Print out messages from the producer in two independent consumers
fromProducer.runForeach(msg ⇒ println("consumer1: " + msg))
fromProducer.runForeach(msg ⇒ println("consumer2: " + msg))
```

自定义 stream 处理 *

虽然 Akka Stream 提供了丰富的处理词汇，但是由于已经存在的操作缺少某些功能，或者由于性能考虑，我们需要定义新的 transformation stages。
下面的部分就会介绍如何自定义 processing stage 和 graph junction。

一、Custom processing with GraphStage

GraphStage 能被用来创建带有任意数量输入和输出的 graph processing stages。
GraphStage 和 GraphDSL.create()方法很相似，可以通过组合其他的 stream processing stages 来创建新的 stream processing stage。
然而 GraphStage 不同的是，它创建的 stage 不能被分割成更小的，并且允许状态以一种安全的方式保持。

1、构造一个自定义的 Source

下面这个例子只是简单地发射数字，从 1 直到被取消。
首先我们需要定义 stage 的"接口"，这在 Akka Stream 术语中叫做 shape。

```
class NumbersSource extends GraphStage[SourceShape[Int]] {
  // 定义这个 stage 唯一的输出端口
  val out: Outlet[Int] = Outlet("NumbersSource")
  // 定义这个 stage 的 shape：SourceShape，将上面定义的 out 传给 SourceShape
  override val shape: SourceShape[Int] = SourceShape(out)
  // 这是实际的逻辑将激活的地方
  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = ???
}
```

在上面这个例子中，实际的运行逻辑被建模为一个 GraphStageLogic 类型的实例，这个实例会在物化时由 materializer 调用 createLogic()方法来创建。
换句话说，所有我们需要做的就是：创建一个合适的逻辑来发送我们想要的数据。

为了在 back-pressured stream 中从 Source 发射数据，你首先需要有来自下游的 demand。
为了接受必要的事件，你需要为输出端口(Outlet)注册一个 OutHandler 的子类。这个 handler 会接收与端口的生命周期相关的事件。

下面的代码中，我们重写了 onPull()方法，表明我们可以自由地发射一个单独的元素。
也有另一个方法 onDownstreamFinish()，这在下游发生取消操作时被调用。

```
class NumbersSource extends GraphStage[SourceShape[Int]] {
  val out: Outlet[Int] = Outlet("NumbersSource")
  override val shape: SourceShape[Int] = SourceShape(out)
  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      // All state MUST be inside the GraphStageLogic, never inside the enclosing GraphStage.
      // This state is safe to access and modify from all the callbacks that are provided by GraphStageLogic and the registered handlers.
      private var counter = 1
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          push(out, counter)
          counter += 1
        }
      })
```

```
    }
}
```

上面 GraphStage 的实例是 Graph[SourceShape[Int],NotUsed]的子类，这意味着这个实例已经可以用在大部分场景中。

为了将这个 graph 转换成 Source，我们需要使用 Source.fromGraph(看”模块化、组合、层级”一节了解更多细节)将 graph 包装起来。

接下来就看看如何使用上面我们定义的 NumbersSource：

```
// A GraphStage is a proper Graph, just like what GraphDSL.create would return
val sourceGraph: Graph[SourceShape[Int], NotUsed] = new NumbersSource
// Create a Source from the Graph to access the DSL
val mySource: Source[Int, NotUsed] = Source.fromGraph(sourceGraph)

// Returns 55
val result1: Future[Int] = mySource.take(10).runFold(0)(_ + _)

// The source is reusable. This returns 5050
val result2: Future[Int] = mySource.take(100).runFold(0)(_ + _)
```

2、构造一个自定义的 Sink

上面介绍了如何构造一个自定义的 Source，下面介绍下如何构造一个自定义的 Sink。

类似自定义 Source 时用到了 Outlet 和 OutHandler，我们也需要为 Inlet 注册一个 InHandler 的子类。

onPush()方法被用来通知 handler 一个新的元素已经被 push 到 stage，这个元素可以被获取和使用了。onPush()方法也可以被重写提供自定义的行为。

请注意，大多数的 Sink 一旦它们创建好了，就需要向它的上游请求数据，这可以在 preStart()方法里调用 pull(inlet)来实现。

构造一个自定义 Sink 的示例：

```
class StdoutSink extends GraphStage[SinkShape[Int]] {
  val in: Inlet[Int] = Inlet("StdoutSink")
  override val shape: SinkShape[Int] = SinkShape(in)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      // This requests one element at the Sink startup.
      override def preStart(): Unit = pull(in)

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          println(grab(in))
          pull(in)
        }
      })
    }
}
```

3、端口状态，InHandler 和 OutHandler *

为了能和 stage 的输入端口(Inlet)或者输出端口(Outlet)进行交互，我们需要能够接收事件并且生成属于这个端口的新的事件。

下面的来自 GraphStageLogic 的操作可以**应用在“输出端口”上**：

- **push(out,elem)** 把一个元素推到输出端口。只有是端口由下游拉取 pulled 之后发生。
- complete(out) 正常关闭输出端口。
- fail(out,exception) 用一个故障信号关闭端口。

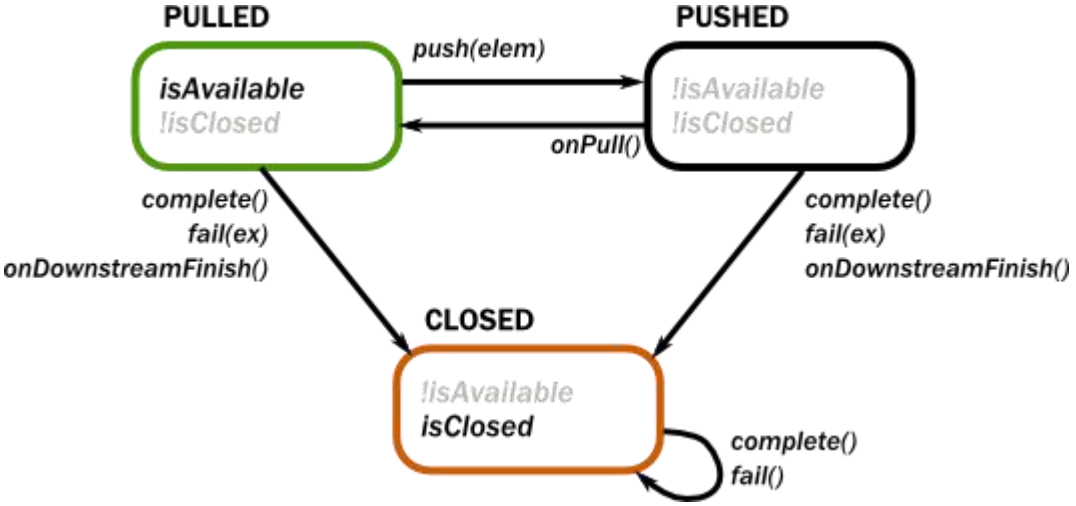
OutHandler 有两个回调方法：

- **onPull()**：当输出端口准备发射下一个元素时被调用，push(out, elem)现在就可以被调用了。
- onDownstreamFinish()：一旦下游发生了取消操作，并且不再允许上游向下游推数据，这个方法就会被调用。
  默认情况下，如果不重写这个方法的话，将会终止(stop)stage。

还有其他的方法可以应用在输出端口上：

- isAvailable(out)：检查输出端口是否能够推送数据。
- isClosed(out)：如果输出端口是关闭的，返回 true。如果端口是关闭的，此时就不能再推送数据。

上面各个事件可以汇总成一个状态机：
绿色部分表示：初始状态
橘黄色部分表示：结束的状态



下面的操作可以**应用在“输入端口”上**：

- **pull(in)**：向输入端口请求下一个元素。只有在端口被上游推送 pushed 后发生。
- grab(in)：获取一个 onPush()接收到的元素。它不能再次调用直到输入端口重新被上游推送了数据。
- cancel(in)：关闭输入端口。

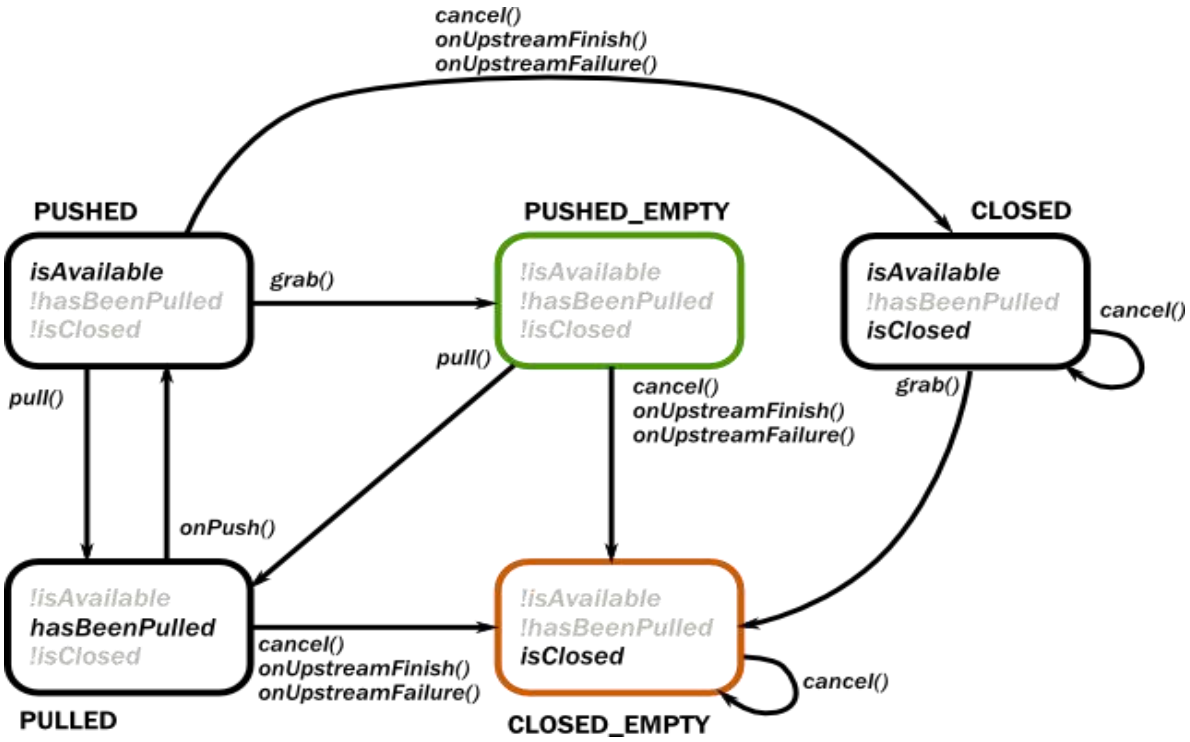
IntHandler 有三个回调方法：

- **onPush()**：当输入端口有新元素到来的时候会被调用。这时候就可以在 onPush()方法里面调用 grab(in)方法获取新到达的数据，并调用 pull(in)方法请求下一个元素。
  注意：如果调用了 pull(in)，但是没有调用 grab(in)，那么新到达的元素将会被丢弃。
- onUpstreamFinish()：当上游已经把所有数据推送完时，这个方法会被调用。
  默认情况下，如果不重写这个方法的话，将会终止(stop)stage。
- onUpstreamFailure()：如果上游因为 exception 发生了失败，并且不会再向下游推送新数据，这时候这个方法就会被调用。
  默认情况下，如果不重写这个方法的话，将会 fail stage。

还有其他的方法可以应用在输入端口上：

- isAvailable(in)：如果输入端口还能接收上游的元素，则返回 true。
- hasBeenPulled(in)：returns true if the port has been already pulled. Calling pull(in) in this state is illegal.
- isClosed(in)：如果输入端口是关闭的，则返回 true。At this point the port can not be pulled and will not be pushed anymore.

上面各个事件可以汇总成一个状态机：
绿色部分表示：初始状态
橘黄色部分表示：结束的状态



最后，还有两个方法可以很方便的 complete stage and all of its ports。

- completeStage() is equivalent to closing all output ports and cancelling all input ports.
- failStage(exception) is equivalent to failing all output ports and cancelling all input ports.

然而，在某些场景中，使用上面的 API 是很不方便的，并且很容易出错。
为了简化操作并降低额外分配的开销，提供了另外一种 API：

- emit(out, elem) 和 emitMultiple(out, Iterable(elem1, elem2))：用来代替 **OutHandler**，当下游有需要的时候，可以发射一个或者多个元素。
- read(in)(andThen) 和 readN(in, n)(andThen)：代替 **IntHandler**，可以读取一个或多个上游推送的元素。
- abortEmitting() 和 abortReading()：取消一个正在进行中的推送或读取操作。

由于上面的方法是通过临时的替换 stage 的 handler 来实现的，因此不要在调用 emit()方法或者 read()方法时调用 setHandler 方法。
下面的方法在 emit 和 read 方法调用之后调用时安全的：
complete(out), completeStage(), emit, emitMultiple, abortEmitting() and abortReading()

4、Completion

默认情况下，一旦所有的端口都关闭了，那么 stages 将自动地终止。
如果不想这样，可以通过调用 setKeepGoing(true)方法。
一旦调用了上面的方法，那么要想关闭 stage，就必须显示的调用 completeStage() 和 failStage(exception)方法。如果没有调用这两个方法，就可能会造成内存泄漏，所以需要小心的使用 setKeepGoing(true)方法。

5、在 GraphStage 内部使用 log

在 GraphStage 内部使用 log 进行 debug 或者使用 log 记录一些重要的信息，这种行为有时候是很有用的。

只要你使用的 Materializer 能为你提供一个 logger，就可以使用 akka.stream.stage.StageLogging 特质可以让你在 GraphStage 内部非常容易的获得一个 LoggingAdapter。

使用示例：

```
final class RandomLettersSource extends GraphStage[SourceShape[String]] {
  val out = Outlet[String]("RandomLettersSource.out")
  override val shape: SourceShape[String] = SourceShape(out)
  override def createLogic(inheritedAttributes: Attributes) =
    new GraphStageLogic(shape) with StageLogging {
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          val c = nextChar() // ASCII lower case letters

          // `log` is obtained from materializer automatically (via StageLogging)
          log.debug("Randomly generated: [{}]", c)

          push(out, c.toString)
        }
      })
    }
}
```

6、使用定时器（顺便展示自定义 Flow）

通过使用 **TimerGraphStageLogic** 可以在 GraphStage 内部使用定时器。

- 通过调用 scheduleOnce(key,delay)、schedulePeriodically(key,period)、schedulePeriodicallyWithInitialDelay(key,delay,period)方法来执行定时器。传递一个对象作为定时器的 key(可以是任何对象，例如一个 String)。
- onTimer(key)方法需要被重写，并且一旦定时器的 key 被触发 onTimer(key)方法将会被调用。
- 使用 cancelTimer(key)来取消定时器。
- 使用 isTimerActive(key)检查定时器的状态。
- 当 stage 完成时，与这个 stage 有关的定时器也会自动的被清除。

下面定义的 stage 是这样的：

- 这个 stage 开始时是关闭的；
- 一旦有元素被推送到下游，然后 open 变量将会变成打开状态并持续一段时间，在这段时间内会消费并丢弃上游的元素；
- 在过了一段时间之后，定时器会把 open 变量置位关闭状态，这时就可以向下游推送数据了。

```
// each time an event is pushed through it will trigger a period of silence
class TimedGate[A](silencePeriod: FiniteDuration) extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("TimedGate.in")
  val out = Outlet[A]("TimedGate.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new TimerGraphStageLogic(shape) {
      var open = false
      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          if (open) pull(in)
          else {
            push(out, elem)
            open = true
            scheduleOnce(None, silencePeriod)
          }
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = { pull(in) }
      })
      override protected def onTimer(timerKey: Any): Unit = {
        open = false
      }
    }
}
```

7、处理异步的事件

In order to receive asynchronous events that are not arriving as stream elements (for example a completion of a future or a callback from a 3rd party API)。

我们必须通过调用 getAsyncCallback()方法来获得一个 AsyncCallback。

getAsyncCallback()方法接收一个回调作为参数，一旦异步的事件触发了，这个回调参数将会被执行。

外部的 API 必须调用 getAsyncCallback()方法返回的 AsyncCallback 实例的 **invoke(event)**方法。

执行引擎会使用线程安全的方式调用 getAsyncCallback()方法的回调参数，并且这个回调可以安全地方法 GraphStageLogic 内部的状态数据。

将 AsyncCallback 放在 GraphStageLogic 的构造函数进行共享会有多线程竞争的问题，因此建议将 AsyncCallback 实例方法生命周期方法 preStart()方法里面。

下面的例子展示：**当一个 future 完成时，closing all output ports and cancelling all input ports：**

```
class KillSwitch[A](switch: Future[Unit]) extends GraphStage[FlowShape[A, A]] {
  val in = Inlet[A]("KillSwitch.in")
  val out = Outlet[A]("KillSwitch.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      override def preStart(): Unit = {
        val callback = getAsyncCallback[Unit] { (_) ⇒
          completeStage()
        }
        switch.foreach(callback.invoke)
      }

      setHandler(in, new InHandler {
        override def onPush(): Unit = { push(out, grab(in)) }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = { pull(in) }
      })
    }
}
```

8、自定义物化值

通过继承 **GraphStageWithMaterializedValue**，而不是之前使用的 GraphStage，可以让自定义的 stage 返回非 NotUsed 类型的物化值。GraphStageWithMaterializedValue 的第一个类型参数表示想要创建的 graph stage 是哪种 shape；第二个类型参数表示返回的物化值的类型。

使用 GraphStageWithMaterializedValue 与使用 GraphStage 的不同是：
我们必须重写 createLogicAndMaterializedValue(inheritedAttributes)方法，并且也必须提供物化值。

下面的例子展示了：**返回的物化值是一个 future，这个 future 包含了流过 stream 的第一个元素**。

```
class FirstValue[A] extends GraphStageWithMaterializedValue[FlowShape[A, A], Future[A]] {
  val in = Inlet[A]("FirstValue.in")
  val out = Outlet[A]("FirstValue.out")

  val shape = FlowShape.of(in, out)

  override def createLogicAndMaterializedValue(inheritedAttributes: Attributes): (GraphStageLogic, Future[A]) = {
    val promise = Promise[A]()

    val logic = new GraphStageLogic(shape) {
      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          promise.success(elem)
          push(out, elem)

          // 因为返回的 future 只包含第一个元素，所以当使用第一个元素 complete 了 promise，就可以使用新的 InHandler 代替原来
          // 的 InHandler 了，保证不会使用后面接收到的元素来 complete promise。
          setHandler(in, new InHandler {
            override def onPush(): Unit = {
              push(out, grab(in))
            }
          })
        }
      })

      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })
    }

    (logic, promise.future)
  }
}
```

9、stage 的生命周期和资源管理

在 stage 的生命周期中可以管理着一个资源。
当 stage 被 shutdown 时，被这个 stage 管理着的资源也应该相应的 shutdown。

清理资源的操作应该在 GraphStageLogic.postStop 方法里执行，而不应该在 InHandler 或者 OutHandler 的回调中执行。

集成

1、和 Actor 集成

你可以在 mapAsync 中使用 ask，或者使用 Sink.actorRefWithAck，把 Stream 中的元素作为消息传送给原生的 Actor。

可以使用 Source.queue，或者使用 Source.actorRef 物化的 ActorRef，把 Actor 的消息传送给 Stream。

mapAsync+ask 的使用方式：

Stream 的 **back-pressure** 由 ask 返回的 Future 维护，并且填充到 Actor 邮箱的消息的数量不会超过在 mapAsync 阶段设置的 parallelism 属性值。

因为使用 ask 模式将 Stream 的元素发送给 Actor，所以 Actor 必须 sender 进行响应，响应的内容将会填充到 Future，并且响应的内容还可以传递到 Stream 的下游。

- 当 Actor 使用 akka.actor.Status.Failure 来对 sender 进行响应时，则 Stream 会因这个 failure 而 completed。
- 如果因为 timeout 导致 ask 操作失败了，则 Stream 会因为 TimeoutException 而 completed。
- 如果你不关心 Actor 的回应内容，可以在 mapAsync 后面使用 Sink.ignore。

注意：

Actor 收到的消息的顺序和 stream 中元素的顺序是一样的，parallelism 属性不会改变消息的顺序。

把 parallelism 属性的值设置为>1，会有一个性能的提升。

示例：

```
import akka.pattern.ask
implicit val askTimeout = Timeout(5.seconds)
val words: Source[String, NotUsed] = Source(List("hello", "hi"))
words
  .mapAsync(parallelism = 5)(elem ⇒ (ref ? elem).mapTo[String])
  // 继续处理 actor 返回的 future
  .map(_.toLowerCase)
  .runWith(Sink.ignore)
```

Sink.actorRefWithAck 的使用方式：

在“Basics and working with Flows”一节，我们介绍了如何定义 Sink 的各种方式，而现在介绍的 Sink.actorRefWithAck 是另外一种定义 Sink 的方式。

Sink.actorRefWithAck 将会发送 Stream 的元素到指定的 ActorRef，而 ActorRef 将会回馈 back-pressure 信号。

Sink.actorRefWithAck 首先会把第一个元素发送到指定的 ActorRef，然后等待 ActorRef 回应的确认，一旦 Stream 收到了 ActorRef 回应的确认就表明 ActorRef 准备好了处理下一个元素。

- 如果目的 Actor 终止了，则 Stream 将会被 cancelled；
- 如果 Stream is completed successfully，则指定的 onCompleteMessage 将会发送到目的 Actor；
- 如果 Stream is completed with failure，则一个 akka.actor.Status.Failure 消息将会发送到目的 Actor。

对上面两种方式进行一个总结：

注意：如果你使用 Sink.actorRef，或者在 map、foreach 阶段使用 tell 消息发送模式，则不会有 back-pressure 机制。

如果 Actor 不能以足够快的速度消费消息，则 Actor 的邮箱会不断增长。

所以最好使用 Sink.actorRefWithAck，或者在 mapAsync 阶段使用 ask 消息发送模式。

Source.queue 的使用方式：

在“Basics and working with Flows”一节，我们介绍了如何定义 Source 的各种方式，而现在介绍的 Source.queue 是另外一种定义 Source 的方式。

我们可以使用 Source.queue 让 Stream 发射来自 Actor(or from anything running outside the stream)的元素。
这些元素将会被缓冲起来，直到 Stream 能处理它们。
你可以 offer 元素到 queue 中，当 Stream 收到了来自下游的请求时，就会发射这些元素。

为了避免当 buffer 满时元素被丢弃，你可以使用 akka.stream.OverflowStrategy.backpressure 溢出策略。

SourceQueue.offer 返回一个 Future[QueueOfferResult]类型的结果：

- 当元素被添加到 bufer 中，或者元素被传送到 Stream 的下游，则使用 QueueOfferResult.Enqueued 来 complete 上面的 Future；
- 如果元素被丢弃了，则使用 QueueOfferResult.Dropped 来 complete 上面的 Future；
- 如果 Stream failed，则使用 QueueOfferResult.Failure 来 complete 上面的 Future；
- 当下游 completed，则使用 QueueOfferResult.QueueClosed 来 complete 上面的 Future；

Source.actorRef 的使用方式：

Messages sent to the actor that is materialized by Source.actorRef will be emitted to the stream if there is demand from downstream, otherwise they will be buffered until request for demand is received.

依赖于 OverflowStrategy，你可以在 buffer 满了时丢弃元素。
但是使用 Source.actorRef 这种方式，并不支持 OverflowStrategy.backpressure，如果你想要有这种策略，请使用上面介绍的 Source.queue 方式。

- 可以通过发送 akka.actor.PoisonPill 或者 akka.actor.Status.Success 到 actor reference，来 successfully completed Stream。
- 可以通过发送 akka.actor.Status.Failure 到 actor reference，来 completed Stream with failure。

当 Stream completed、failed，或者 Stream 的下游 cancelled，则 Source.actorRef 返回的 actor 将会被终止。
你可以监控这种情况，并在这种情况发生时收到通知。

Stream 中的错误处理

记住：当 Stream 中某一个 stage 失败了，将会导致整个 Stream 崩溃。

失败的 stage 的所有下游都会收到这个 failure 的通知，所有的上游都会看到 cancellation。

有很多方式可以避免以失败的方式完成 Stream：

- recover：当上游发生了 failure 的时候，发射最后一个元素，然后正常地完成 Stream。
- recoverWithRetries：当上游发生了 failure 的时候，创建一个新的上游代替发生 failure 的上游，并且开始从这个新的上游消费元素。
- 在 backoff 之后重启 Stream 的一部分。
- 使用一个监管策略。

1、Recover

使用 recover 允许你：**当上游发生了 failure 的时候，发射最后一个元素，然后正常地完成 Stream**。

使用一个偏函数来决定遇到哪些 exception 可以让 Stream 恢复。如果一个 exception 不符合任何一个模式匹配，则 Stream 会 fail。

示例：当上游发射元素 6 时，抛出了 exception，但是使用了 recover 捕获了这种失败，避免 Stream 崩溃。

```
Source(0 to 6).map(n ⇒
  if (n < 5) n.toString
  else throw new RuntimeException("Boom!")
).recover {
  case _: RuntimeException ⇒ "stream truncated"
}.runForeach(println)
```

输出：

```
0
1
2
3
4
stream truncated
```

2、recoverWithRetries

recoverWithRetries 允许你：**当上游发生了 failure 的时候，创建一个新的上游代替发生 failure 的上游，并且开始从这个新的上游消费元素**。

使用一个偏函数来决定遇到哪些 exception 可以让 Stream 恢复。如果一个 exception 不符合任何一个模式匹配，则 Stream 会 fail。

示例：

```
val planB = Source(List("five", "six", "seven", "eight"))
Source(0 to 10).map(n ⇒
  if (n < 5) n.toString
  else throw new RuntimeException("Boom!")
).recoverWithRetries(attempts = 1, {
```

```
  case _: RuntimeException ⇒ planB    // 这个 planB 就是新的上游
}).runForeach(println)
```

输出：

```
0
1
2
3
4
five
six
seven
eight
```

3、Delayed restarts with a backoff stage

当因为某些外部资源不可用导致 Stream fail 或者 complete 时，使用这种模式是很有用的。你需要提供一些时间让 Stream 再次重启。

Akka Stream 提供了 RestartSource, RestartSink 和 RestartFlow 来实现这种模式。**当一个 stage fail 或者 complete 时，再次启动它。每次重启的延迟时间是不断增长的**。

使用 randomFactor 来让每次重启时的延迟时间增加一点，可以避免多个 Stream 在完全相同的时间同时重启。

示例：

下面使用 akka.stream.scaladsl.RestartSource 创建一个 backoff supervisor，它将会监控给定的 Source。

如果这个 Stream 在任何时候 fail 或者 complete 了，下面的 HttpRequest 都会被重新创建，每次重新启动的时间间隔为 3, 6, 12, 24，最终为 30 seconds。

```
val restartSource = RestartSource.withBackoff(
  minBackoff = 3.seconds,
  maxBackoff = 30.seconds,
  randomFactor = 0.2       // adds 20% "noise" to vary the intervals slightly
) { () ⇒
  // Create a source from a future of a source
  Source.fromFutureSource {
    // Make a single request with akka-http
    Http().singleRequest(HttpRequest(
      uri = "http://example.com/eventstream"
    ))
      // Unmarshall it as a source of server sent events
      .flatMap(Unmarshal(_).to[Source[ServerSentEvent, NotUsed]])
  }
}
```

需要注意，上面的 RestartSource 永远不会终止，除非下游的 Sink 被取消了。

如果你想在需要的时候能控制上面 Stream 的完成，可以使用之前介绍的 KillSwitch 来实现。

```
val killSwitch = restartSource
  .viaMat(KillSwitches.single)(Keep.right)
  .toMat(Sink.foreach(event ⇒ println(s"Got event: $event")))(Keep.left)
  .run()
```

doSomethingElse()
killSwitch.shutdown()

4、监管策略

有三种方式可以处理 exception：

- Stop 终止：Stream 将会以 failure 来 completed。
- Resume 恢复：有问题的元素将会被丢弃，Stream 继续正常运行。
- Restart 重启：有问题的元素将会被丢弃，Stream 会在 stage 重启之后继续运行。
  重启一个 stage 意味着之前积累的状态会丢失，重启一个 stage 是通过创建一个新的 stage 实现的。

默认情况下，Stop 终止策略会应用到所有的 exceptions。

示例：

```
implicit val materializer = ActorMaterializer()
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ + _))
// 由于在 map 中使用了 0 做除数，导致 stream fail。并且 result will be a Future completed with Failure(ArithmeticException)
```

可以在 materializer 的 settings 配置中设置自定义的监管策略。

```
val decider: Supervision.Decider = {
  case _: ArithmeticException ⇒ Supervision.Resume
  case _                      ⇒ Supervision.Stop
}
implicit val materializer = ActorMaterializer(
  ActorMaterializerSettings(system).withSupervisionStrategy(decider))
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ + _))
// 导致除 0 错误的元素将会被丢弃，并且 result will be a Future completed with Success(228)
```

也可以为单个 flow 的所有操作定义监控策略：

```
implicit val materializer = ActorMaterializer()
val decider: Supervision.Decider = {
  case _: ArithmeticException ⇒ Supervision.Resume
  case _                      ⇒ Supervision.Stop
}
val flow = Flow[Int]
  .filter(100 / _ < 50).map(elem ⇒ 100 / (5 - elem))
  .withAttributes(ActorAttributes.supervisionStrategy(decider))
val source = Source(0 to 5).via(flow)

val result = source.runWith(Sink.fold(0)(_ + _))
// 导致除 0 错误的元素将会被丢弃，并且 result will be a Future completed with Success(150)
```

Restart 策略和 Resume 策略的工作方式类似，只是它会清除 Stream 之前积累的状态。

```
implicit val materializer = ActorMaterializer()
val decider: Supervision.Decider = {
  case _: IllegalArgumentException => Supervision.Restart
  case _ => Supervision.Stop
}

val flow = Flow[Int]
  .scan(0) { (acc, elem) =>
    if (elem < 0) throw new IllegalArgumentException("negative not allowed")
    else acc + elem
  }
  .withAttributes(ActorAttributes.supervisionStrategy(decider))

val source = Source(List(1, 3, -1, 5, 7)).via(flow)
val result = source.grouped(1000).runWith(Sink.head)
// 负数元素会导致`scan`这步骤重启,
// 即 重新从 0 开始
// 流将以`Success(Vector(0, 1, 4, 0, 5, 12)`为值的`Future`作为完成结果
```

Stream 的监管策略也可以作用于 mapAsync 和 mapAsyncUnordered 返回的 future 上，即使 failure 发生在 future 内部而不是 stage 自身内部。

```
val authors: Source[Author, NotUsed] =
  tweets
    .filter(_.hashtags.contains(akkaTag))
    .map(_.author)

// 如果 email 没有找到，则使用 failure 填充返回的 Future
def lookupEmail(handle: String): Future[String] =

// 我们使用 Supervision.resumingDecider 策略来丢弃未知的 email 地址，并保证 stream 可以继续运行
val emailAddresses: Source[String, NotUsed] =
  authors.via(
    Flow[Author].mapAsync(4)(author ⇒ addressSystem.lookupEmail(author.handle))
      .withAttributes(supervisionStrategy(resumingDecider)))
```

流水线和并行性

在“stream 的物化#操作符融合”一节，已经介绍了：默认情况下，Akka Stream 将会融合 stream 的操作符。这意味着一个 flow 或者 graph 的 processing steps 将在一个相同的 actor 中执行。并且 Akka Stream 的 procesing stages 时串行执行的。

在大部分情况下，并行地执行 the stages of a flow 时非常有用的。这可以通过使用 async 方法显示地将它们标记为异步来实现。

1、流水线

举一个流水线方式摊煎饼的例子：

Roland 以不同的方式使用两个煎锅. 第一个煎锅只用来烤煎饼的一面, 然后半面完成的煎饼被倒置到第二个煎锅中来煎烤另一面来完成整个过程. 一旦第一个煎锅变得可用, 它就得到一勺面糊. 这种方式的效果, 大部分的时间两个煎锅同时工作, 一个饼在烤一面同时第二个饼烤另一面然后完成.

以下是这个过程用流来实现大概的样子：

```
// 获取一勺面糊接着创造出一个半面煎饼
val fryingPan1: Flow[ScoopOfBatter, HalfCookedPancake, NotUsed] = Flow[ScoopOfBatter].map { batter => HalfCookedPancake() }
// 完成一个半面煎饼
val fryingPan2: Flow[HalfCookedPancake, Pancake, NotUsed] = Flow[HalfCookedPancake].map { halfCooked => Pancake() }
// 通过两个半面煎饼我们就完成了烘烤煎饼
val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] = Flow[ScoopOfBatter].via(fryingPan1.async).via(fryingPan2.async)
```

使用流水线 Pipelining 的好处是：流水线这种操作模式可以应用到任何无法并行化的串行 processing steps (例如，其中一个 processing stip 依赖上游 processing step 的数据)。

使用流水线 Pipelining 的缺点是：如果 processing stages 的处理时间各不相同，可能某些 processing stages 不能发挥全部的处理，因为它们可能在等待上游的结果。

2、并行处理　　#

举一个并行摊煎饼的例子：

Patrik 以相同的方式来使用两个煎锅. 他使用两个锅都完全烤完煎饼的两个面, 然后把结果放置到一个共享的盘子上. 每当一个煎锅空了, 他从一碗共享的面糊中拿出下一勺. 本质上他在多个煎锅上并行了同样的过程.

以下是这个过程用流来实现大概的样子：

```
// 将一勺面糊烤成一个完整的煎饼
val fryingPan: Flow[ScoopOfBatter, Pancake, NotUsed] = Flow[ScoopOfBatter].map { batter ⇒ Pancake() }

val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] = Flow.fromGraph(GraphDSL.create() { implicit builder ⇒
  val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
  val mergePancakes = builder.add(Merge[Pancake](2))

  // 并行地使用两个煎锅，这两个煎锅都会把面糊完全地烤成一个煎饼.
  // 我们总是把下一勺面糊放置到第一个变得空闲的煎锅上.
  dispatchBatter.out(0) ~> fryingPan.async ~> mergePancakes.in(0)
  // 注意到：我们并没有通过 builder.add()的方式来把 ftyingPan 引入进来.
  // 使用这种方式的 Flow 会被自动导入进来。在当前这个例子中，这两个 ftyingPan 实际上是不同的 stages.
  dispatchBatter.out(1) ~> fryingPan.async ~> mergePancakes.in(1)
```

```
  FlowShape(dispatchBatter.in, mergePancakes.out)
})
```

使用并行处理的优点是：可以很容易进行扩展。
使用并行处理的缺点是：没有考虑到数据的顺序。

3、合并流水线和并行处理

Akka Stream 提供了一个优雅同意的语言来表达和组合流水线和并行处理。

首先，我们看看如何**将 pipelined processing stages 进行并行化**。
举个并行化 pipelined processing stages 的例子：
我们雇佣了两个厨师，每一个都使用流水线的方式进行摊煎饼，并行地使用的两个厨师，就达到了并行化 pipelined processing stages。

```
val pancakeChef: Flow[ScoopOfBatter, Pancake, NotUsed] =
  Flow.fromGraph(GraphDSL.create() { implicit builder ⇒

    val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
    val mergePancakes = builder.add(Merge[Pancake](2))

    // 使用两条流水线，每天流水线使用两个煎锅，共使用四个煎锅
    dispatchBatter.out(0) ~> fryingPan1.async ~> fryingPan2.async ~> mergePancakes.in(0)
    dispatchBatter.out(1) ~> fryingPan1.async ~> fryingPan2.async ~> mergePancakes.in(1)

    FlowShape(dispatchBatter.in, mergePancakes.out)
  })
```

接下来看看，如何**将 parallelized processing stages 组织成流水线形式**。
还是以摊煎饼为例，这意味着将要雇佣 4 位厨师：
前两名厨师需要并行的准备从面糊到烤完半面的煎饼，然后把它们放置到一个够大的平面上.
后两名厨师拿走半面煎饼并在各自的煎锅中烘烤另一边，然后他们把煎饼放置到一个共享的盘子中.
使用 Stream API 来实现这个场景：

```
val pancakeChefs1: Flow[ScoopOfBatter, HalfCookedPancake, NotUsed] =
  Flow.fromGraph(GraphDSL.create() { implicit builder ⇒
    val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
    val mergeHalfPancakes = builder.add(Merge[HalfCookedPancake](2))

    // Two chefs work with one frying pan for each, half-frying the pancakes then putting
    // them into a common pool
    dispatchBatter.out(0) ~> fryingPan1.async ~> mergeHalfPancakes.in(0)
    dispatchBatter.out(1) ~> fryingPan1.async ~> mergeHalfPancakes.in(1)

    FlowShape(dispatchBatter.in, mergeHalfPancakes.out)
  })

val pancakeChefs2: Flow[HalfCookedPancake, Pancake, NotUsed] =
  Flow.fromGraph(GraphDSL.create() { implicit builder ⇒
```

```
    val dispatchHalfPancakes = builder.add(Balance[HalfCookedPancake](2))
    val mergePancakes = builder.add(Merge[Pancake](2))

    // Two chefs work with one frying pan for each, finishing the pancakes then putting
    // them into a common pool
    dispatchHalfPancakes.out(0) ~> fryingPan2.async ~> mergePancakes.in(0)
    dispatchHalfPancakes.out(1) ~> fryingPan2.async ~> mergePancakes.in(1)

    FlowShape(dispatchHalfPancakes.in, mergePancakes.out)
  })

val kitchen: Flow[ScoopOfBatter, Pancake, NotUsed] = pancakeChefs1.via(pancakeChefs2)
```

测试 Stream

有多种代码模式和类库可以验证 Akka Stream 的 source、flow、sink 的行为。

通过分隔你的 source、flow、sink 让你的数据处理保持流水线形式，这是非常重要的。可以让测试变得很容易。

1、内建的 source、sink、组合器

测试一个自定义的 sink 是非常简单的，只需要附着一个从预定义集合发射元素的 source，运行一个构造好的 flow，然后在 sink 产生的结果上进行断言就 OK 了。

下面是一个测试 sink 的例子：

```
val sinkUnderTest = Flow[Int].map(_ * 2).toMat(Sink.fold(0)(_ + _))(Keep.right)   // 待测试的 sink
val future = Source(1 to 4).runWith(sinkUnderTest)
val result = Await.result(future, 3.seconds)
assert(result == 20)
```

相同的测试套路也可以应用到测试 source。

下面的例子中，有一个会产生无数元素的 source，这样的 source 可以通过断言最开始满足某些条件的任意个数的元素来进行测试。

在这里 take 方法和 Sink.seq 是很有用的。

```
val sourceUnderTest = Source.repeat(1).map(_ * 2)          // 待测试的 source
val future = sourceUnderTest.take(10).runWith(Sink.seq)
val result = Await.result(future, 3.seconds)
assert(result == Seq.fill(10)(2))
```

当**测试 flow** 时，需要为 flow 附着一个 source 和 sink。

我们可以使用多种 source 来测试 flow 的各种边界场景，使用 sink 的结果进行断言。

```
val flowUnderTest = Flow[Int].takeWhile(_ < 5)          // 待测试的 flow
val future = Source(1 to 10).via(flowUnderTest).runWith(Sink.fold(Seq.empty[Int])(_ :+ _))
val result = Await.result(future, 3.seconds)
assert(result == (1 to 4))
```

2、Stream 测试套件

Akka Stream 提供了一个单独的模块 akka-stream-testkit，此模块提供很多特殊的工具可以帮助我们写 stream 测试。
这个模块有两个主要的组件 TestSource 和 TestSink，它们提供了把 Source 和 Sink 具象化到 Probes 的流式 API.

通过 **TestSink.probe** 返回的 sink，可以对到达下游的元素进行断言，并允许我们手动地按需控制要断言的元素的数量。
示例：

```
val sourceUnderTest = Source(1 to 4).filter(_ % 2 == 0).map(_ * 2)     // 待测试的 source
sourceUnderTest
  .runWith(TestSink.probe[Int])
  .request(2)
  .expectNext(4, 8)
  .expectComplete()
```

通过 **TestSource.probe** 返回的 source，可以被用来按需断言，或者控制 stream 何时正常完成，或者以一个 error 结束。
示例 1：

```
val sinkUnderTest = Sink.cancelled   // 待测试的 sink
TestSource.probe[Int]
  .toMat(sinkUnderTest)(Keep.left)
  .run()
  .expectCancellation()    // 期待 stream 被 cancel
```

示例 2：可以手动注入 exception，并且测试 sink 在错误条件下的行为

```
val sinkUnderTest = Sink.head[Int]          // 待测试的 sink
val (probe, future) = TestSource.probe[Int]
  .toMat(sinkUnderTest)(Keep.both)
  .run()
probe.sendError(new Exception("boom"))   // 手动注入 exception
Await.ready(future, 3.seconds)
val Failure(exception) = future.value.get
assert(exception.getMessage == "boom")   // 检查 sink 的行为是否以 exception 结束
```

使用上面两个组件，我们可以测试 flow：

```
// 待测试的 flow
val flowUnderTest = Flow[Int].mapAsyncUnordered(2) { sleep ⇒
  pattern.after(10.millis * sleep, using = system.scheduler)(Future.successful(sleep))
}

val (pub, sub) = TestSource.probe[Int]
  .via(flowUnderTest)
  .toMat(TestSink.probe[Int])(Keep.both)
  .run()

sub.request(n = 3)
pub.sendNext(3)
pub.sendNext(2)
pub.sendNext(1)
```

```
sub.expectNextUnordered(1, 2, 3)

pub.sendError(new Exception("Power surge in the linear subroutine C-47!"))
val ex = sub.expectError()
assert(ex.getMessage.contains("C-47"))
```

3、Fuzzing Mode

对于测试，我们可以开启一个特殊的 stream 执行模式，在这个模式下会并发的执行(减少性能的消耗)。通过并发执行可以帮助我们暴露出多线程竞争情况(race conditions)。
使用下面的配置可以开启这种模式：

```
akka.stream.materializer.debug.fuzzing-mode = on
```

警告：
不要在生产环境或者在 benchmark 时使用这种模式。
这个测试工具可以对你的代码提供更多的测试覆盖率，但是这它减少了 stream 的吞吐量。
如果你开启了这个模式，将会有一个警告日志打印出来。